NOVEMBER 11, 2023

CSE-311

COURSE PROJECT

MOTURI VENKATA SATYA KARTHIK
2021BCS0137
Batch 1

HAND WRITTEN DIGIT RECOGNITION

Overview:
This project aims to create a user-friendly interface for handwritten digit recognition using Pygame and a pre-trained Convolutional Neural Network (CNN). The interface allows users to draw single digits, and the AI model predicts the digit in real-time, offering an interactive experience.

Objective:
Develop an intuitive interface enabling users to draw digits, with the system recognizing and displaying the predicted digit. This serves as an interactive tool for users to experience real-time handwritten digit recognition.

Components:

- Drawing Canvas:
  - Users utilize a blank canvas to draw single digits using the mouse.
  - A clear button resets the canvas for new drawings.

- Handwritten Digit Recognition:
  - The interface employs a pre-trained CNN model, specifically trained on the MNIST dataset for digits 0 to 9.
  - Real-time processing predicts the drawn digit as users draw on the canvas.

- Prediction Display:
  - The predicted digit is displayed on the interface, updating dynamically as users modify or draw new digits.
  - Displayed near the drawing area for immediate feedback.

- Interactive Features:
  - Real-time recognition allows users to see predictions as they draw.
  - Users receive immediate feedback on the recognized digit.

Implementation:

- Pygame is used for the interface, handling graphical elements and user input.
- A pre-trained CNN model, developed with Keras, recognizes handwritten digits.
- Real-time recognition is achieved by processing drawn images through the CNN model.
- Visual feedback is provided by displaying the predicted digit on the interface.

Novelty:

- Interactive Handwritten Digit Recognition Game:
    - Fusion of deep learning and game development, enabling real-time digit drawing with instant AI predictions.

- Pygame and Neural Network Integration:
    - Unique integration of Pygame for the graphical interface and user input with a pre-trained neural network for real-time recognition.

- Dynamic Canvas Handling:
    - Real-time processing of drawn digits, allowing users to draw multiple digits with immediate prediction feedback.

- Scalability and Extension:
    - Design permits easy extension, such as recognizing multiple digits sequentially or adding a scoring system.

- Cross-Disciplinary Fusion:
    - Integration of computer vision, deep learning, and game development concepts, showcasing an interdisciplinary approach.

Complexity:

- Model Complexity:
    - Implementation of a CNN using Keras with convolutional layers, max-pooling layers, dropout for regularization, and a densely connected layer.

- Image Processing:
    - Normalizing pixel values, reshaping images, and converting class labels to one-hot vectors.

- Pygame Interaction:
    - Utilization of the Pygame library for creating the graphical user interface, handling events, and real-time rendering.

- Callbacks and Checkpoints:
    - Use of Keras callbacks (EarlyStopping and ModelCheckpoint) to monitor and control the training process.

- Real-time Prediction:
    - Dynamically processing drawn images through the trained model, displaying predictions, and updating the screen.

CODE :

```
import numpy as np
import matplotlib.pyplot as plt

import keras
from keras.datasets import mnist

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


from keras.callbacks import ModelCheckpoint , EarlyStopping

(X_train , y_train) , (X_test , y_test) = mnist.load_data()

X_train.shape , y_train.shape , X_test.shape , y_test.shape

def plot_input_img(i):
    plt.imshow(X_train[i] , cmap = 'binary')
    plt.title(y_train[i])
    plt.axis('off')
    plt.show()

for i in range(10):
    plot_input_img(i)
```

- Importing Libraries:
  - numpy for numerical operations.
  - matplotlib.pyplot for plotting.
  - keras for building and training neural networks.
  - Specific Keras modules for working with the MNIST dataset and building a neural network.
- Loading MNIST Dataset:
  - The MNIST dataset is loaded into four variables: X_train and y_train for training images and labels, and X_test and y_test for testing images and labels.
- Printing Dataset Shapes:
  - Prints the shapes of the training and testing datasets to understand their dimensions.
- Defining a Function to Plot Images:
  - Defines a function plot_input_img(i) that takes an index i and plots the corresponding image from the training set along with its label.
- Loop for Plotting Images:
  - Loops through the first 10 images in the training set and uses the plot_input_img function to display each image with its label.

```
#pre Process the images

#normalizing the image to [0,1] range
X_train = X_train.astype('float32')/255
X_test = X_test.astype('float32')/255
```

The above code is normalizing the pixel values of these images to the range [0, 1]. This is commonly done to ensure that pixel values are within a consistent range, which can make it easier for neural networks to learn from the data. Typically, image pixel values are represented as integers in the range [0, 255], so dividing by 255 scales them to the [0, 1] range.

```
# REshape / expand the dimension of the image to (28,28,1)
X_train = np.expand_dims(X_train , -1)
X_test = np.expand_dims(X_test , -1)
```

The above code is reshaping the images to have an additional dimension with size 1. it expands the dimensions of the images from a 2D shape (e.g., 28x28 pixels) to a 3D shape (28x28x1). The additional dimension (1) is typically used to represent the number of color channels (e.g., grayscale images have one channel, while RGB images have three channels).

```
# convert  classes to one hot vectors
y_train = keras.utils.to_categorical(y_train)
y_test = keras.utils.to_categorical(y_test)
```

Here y_train and y_test represent the class labels for the images. To prepare these labels for use in a machine learning model, they are converted into one-hot vectors. One-hot encoding represents each class as a binary vector where one element is set to 1 to indicate the class, and all other elements are set to 0. This encoding is often used for classification tasks, and it helps the model understand the categorical nature of the labels.

```
model = Sequential()
model.add(Conv2D(32 ,  (3,3) ,input_shape = (28,28,1) , activation = 'relu' ))
model.add(MaxPooling2D(pool_size = (2,2)))
model.add(Conv2D(64 ,  (3,3) , activation = 'relu' ))
model.add(MaxPooling2D(pool_size = (2,2)))
model.add(Flatten())
model.add(Dropout(0.25))
model.add(Dense(10 , activation = 'softmax'))
```

The provided code defines a sequential Keras model, a linear stack of layers used for constructing neural networks. The architecture can be described as follows:

- **Convolutional Layer (32 filters, 3x3 kernel, ReLU activation):**
  - The model begins with a convolutional layer employing 32 filters, each utilizing a 3x3 kernel.
  - Rectified Linear Unit (ReLU) is the chosen activation function, facilitating the extraction of image features through convolution operations.
- **MaxPooling Layer (2x2 window):**
  - Following each convolutional layer, a max-pooling layer is incorporated with a 2x2 window. This operation reduces the spatial dimensions of the data, retaining essential information.
- **Convolutional Layer (64 filters, 3x3 kernel, ReLU activation):**
  - Subsequently, another convolutional layer is introduced, featuring 64 filters and a 3x3 kernel. Like before, a ReLU activation function follows to further capture intricate features.
- **MaxPooling Layer (2x2 window):**
  - Similar to the previous convolutional layer, a max-pooling layer is applied post the second convolution, downscaling the data.
- **Flatten Layer:**
  - A Flatten layer is employed to transform the 2D feature maps generated by the convolutional layers into a 1D vector. This format is necessary for subsequent fully connected layers.
- **Dropout Layer (Dropout rate: 25%):**
  - To prevent overfitting, a Dropout layer with a 25% dropout rate is introduced. This layer randomly deactivates a portion of neurons during training.
- **Fully Connected Layer (10 units, Softmax activation):**
  - The final layer is a dense (fully connected) layer hosting 10 units, corresponding to the number of output classes.
  - Softmax activation is utilized, assigning probabilities to each class and rendering it suitable for classification tasks.

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

The provided code compiles a Keras model, specifying its training configuration. Here's a breakdown of the compilation settings:

- **Loss Function (Categorical Crossentropy):**
  - The chosen loss function is 'categorical_crossentropy.' This particular loss function is widely employed in multi-class classification scenarios. It quantifies the disparity between the predicted class probabilities and the actual class labels.
- **Optimizer (Adam):**
  - The optimizer selected is 'adam.' Adam is a prevalent optimization algorithm used to adjust the model's weights during training. Its adaptive learning rate and momentum contribute to efficient convergence.
- **Evaluation Metrics (Accuracy):**
  - The model is set to monitor and report accuracy during training. Accuracy is a standard evaluation metric for classification tasks, representing the ratio of correctly predicted instances to the total number of instances.

the compilation step finalizes the model's configuration for training, specifying how it should learn from the data, minimize errors, and evaluate its performance.

```
#early stopping
es = EarlyStopping(monitor = 'val_accuracy' ,min_delta =0.01, patience = 4 , verbose = 1 )
```

EarlyStopping is a callback that monitors the validation accuracy of the model during training and stops the training process when certain conditions are met.

- **monitor='val_accuracy'** specifies that it should monitor the validation accuracy of the model.
- **min_delta=0.01** sets a minimum change in validation accuracy that must be observed to consider it an improvement. If the validation accuracy doesn't improve by at least 0.01, training may be stopped.
- **patience=4** indicates how many consecutive epochs with no improvement are allowed before stopping the training. In this case, if there is no improvement for four consecutive epochs, training will stop.
- **verbose=1** specifies that the callback will print messages when it stops the training due to lack of improvement.

```
#model checkpoint
mc = ModelCheckpoint('./best_model.h5' , monitor = 'val_accuracy' , verbose = 1 ,
save_best_only = True)
```

ModelCheckpoint is a callback that saves the model's weights to a file during training. It can save the best model based on a monitored metric.

- **./best_model.h5** is the file path where the best model's weights will be saved.
- **monitor='val_accuracy'** specifies that it should monitor the validation accuracy to decide when to save the model.
- **verbose=1** indicates that it will print messages when it saves the model.
- **save_best_only=True** ensures that only the best model (based on validation accuracy) will be saved. If a new model with higher validation accuracy is found during training, it will overwrite the previously saved model.

```
cb= [es , mc]
```

This line creates a list **cb** that contains the configured callback objects **es** (EarlyStopping) and **mc** (ModelCheckpoint). These callbacks can be passed to the fit method of a Keras model to enable them during training.

```
his = model.fit(X_train , y_train , epochs = 5 , validation_split = 0.3 , callbacks = cb)
```

This line trains the Keras model using the fit method.

- X_train and y_train are the training data and labels.
- epochs=5 specifies that the training process will run for 5 epochs.
- validation_split=0.3 means that 30% of the training data will be used for validation during training.
- callbacks=cb applies the previously defined callback functions, es (EarlyStopping) and mc (ModelCheckpoint), during training to monitor the model's performance and take actions such as stopping training when necessary.

The training history and metrics are stored in the his variable, allowing you to analyze the model's performance after training.

```
model_S = keras.models.load_model('./best_model.h5')
```

This code loads a previously trained Keras model from a file named *'best_model.h5'.*

- **keras.models.load_model()** is a function that loads a saved Keras model from a file.
- **'./best_model.h5'** specifies the file path where the saved model is located. The model is loaded into the variable model_S, allowing you to use it for predictions or further analysis.

```
score = model_S.evaluate(X_test , y_test )
```

This line evaluates the Keras model model_S on the test dataset X_test with corresponding labels y_test. The **evaluate()** method computes the model's performance on the test data.

```
print(f"THE MODEL ACCURACY IS {score[1]*100}")
```

This line prints the model's accuracy as a percentage. The accuracy is obtained from the score variable, which is a list containing different evaluation metrics. In this case, score[1] represents the accuracy, and it's multiplied by 100 to express it as a percentage.

```
import pygame , sys
from pygame.locals import *
import numpy as np
from keras.models import load_model
import cv2

WINDOWSIZEX = 640
WINDOWSIZEY = 480

WHITE = (255, 255, 255)
BLACK = (0, 0, 0)
RED = (255, 0 , 0)

IMAGESAVE = False

BOUNDRY = 5

# load model
MODEL = load_model("best_model.h5")

LABELS ={0: "ZERO", 1: "ONE", 2: "TWO", 3: "THREE", 4: "FOUR", 5: "FIVE" , 6: "SIX", 7:
"SEVEN", 8: "EIGHT", 9: "NINE"}

# initialize pygame
pygame.init()

FONT = pygame.font.SysFont("Arial", 20)
DISPLAYSURF = pygame.display.set_mode((WINDOWSIZEX, WINDOWSIZEY))
```

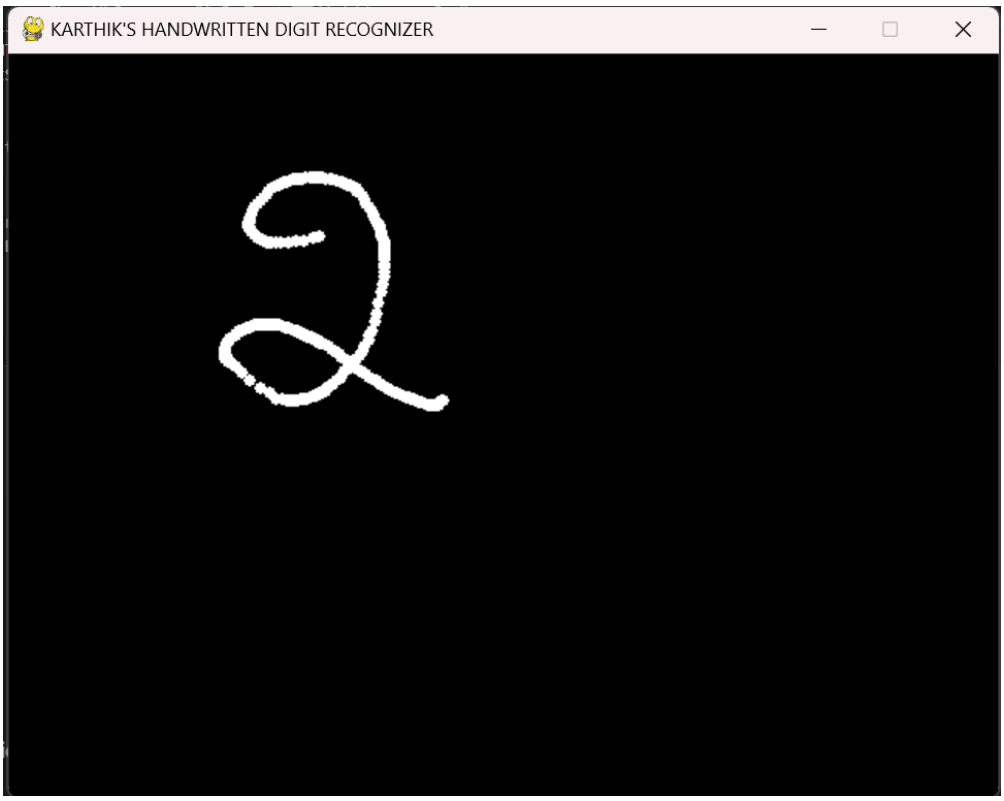
```
pygame.display.set_caption("KARTHIK'S HANDWRITTEN DIGIT RECOGNIZER ")

iswriting = False

number_xcord = []
number_ycord = []
image_cnt = 1

PREDICT = True

while True:
    for event in pygame.event.get():
        if event.type == QUIT:
            pygame.quit()
            sys.exit()

        if event.type == MOUSEMOTION and iswriting:
            xcord, ycord = event.pos
            pygame.draw.circle(DISPLAYSURF, WHITE, (xcord, ycord), 4, 0)

            number_xcord.append(xcord)
            number_ycord.append(ycord)

        if event.type == MOUSEBUTTONDOWN:
            iswriting = True

        if event.type == MOUSEBUTTONUP:
            iswriting = False
            number_xcord = sorted(number_xcord)
            number_ycord = sorted(number_ycord)

            rect_min_x , rect_max_x = max(number_xcord[0] - BOUNDRY, 0),
min(number_xcord[-1] + BOUNDRY, WINDOWSIZEX)
            rect_min_y , rect_max_y = max(number_ycord[0] - BOUNDRY, 0),
min(number_ycord[-1] + BOUNDRY, WINDOWSIZEY)
            number_xcord = []
            number_ycord = []

            img_arr = np.array(pygame.PixelArray(DISPLAYSURF))[rect_min_x:rect_max_x,
rect_min_y:rect_max_y].T.astype(np.float32)

            if IMAGESAVE:
                cv2.imwrite(f"image_{image_cnt}.png", img_arr)  # Corrected line
                image_cnt += 1  # Corrected line

            if PREDICT:
                image = cv2.resize(img_arr, (28, 28))
                image = np.pad(image, (10,10), "constant", constant_values=0)
                image = cv2.resize(image, (28, 28))/255

                label = str(LABELS[np.argmax(MODEL.predict(image.reshape(1, 28, 28, 1)))])
                textSurface = FONT.render(label, True, RED, WHITE)
```

```
            textRecObj = textSurface.get_rect()
            textRecObj.left , textRecObj.bottom = rect_min_x, rect_min_y

            DISPLAYSURF.blit(textSurface, textRecObj)

    if event.type == KEYDOWN:
        if event.unicode == "n":
            DISPLAYSURF.fill(BLACK)

pygame.display.update()
```

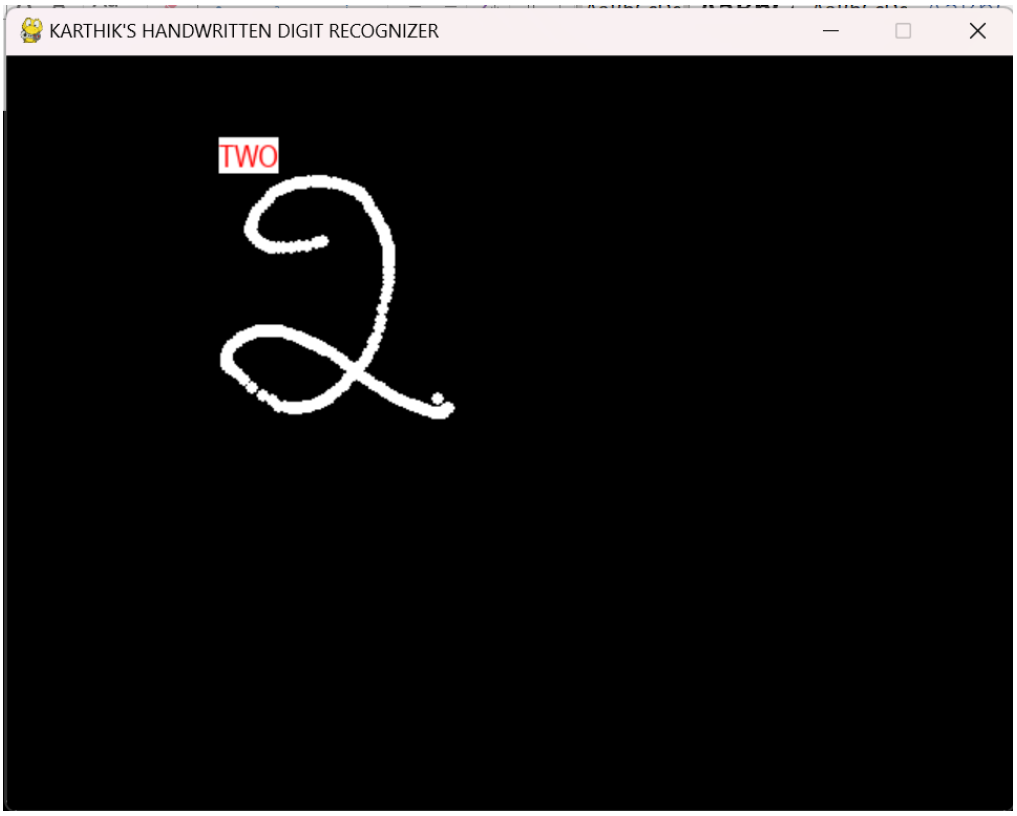
Handwritten Digit Recognition Application:

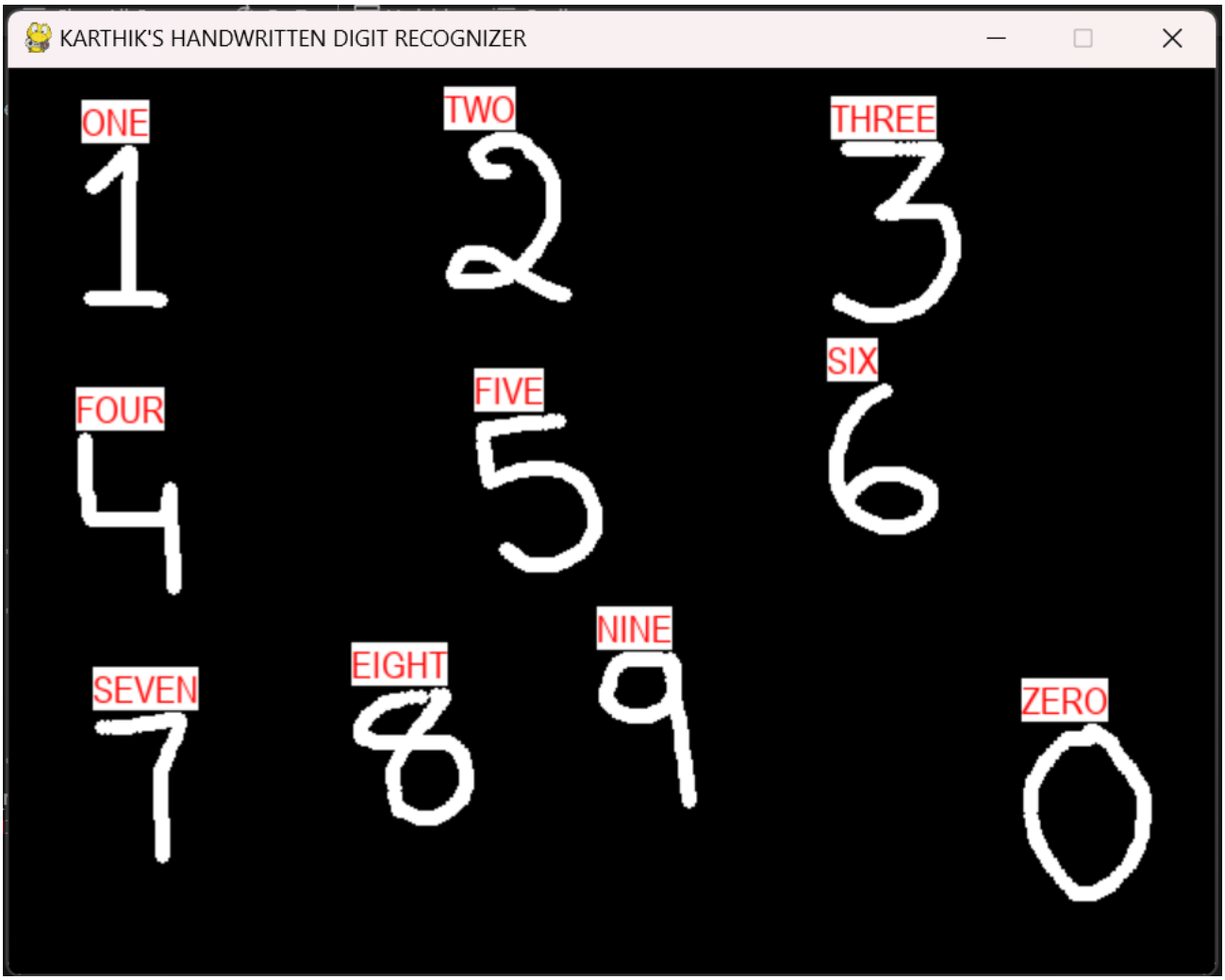
- This Python script uses the Pygame library to create an interactive application for handwritten digit recognition.
- The user can draw a digit on the screen, and the script uses a pre-trained Keras model to predict and display the recognized digit.
- It initializes Pygame, loads a pre-trained model, and sets up a graphical window.
- Users can draw digits using the mouse, and the script records the drawn path and displays it on the screen.
- When the drawing is complete, the script extracts the digit, processes it for recognition, and displays the predicted digit label.
- Users can also clear the drawing area by pressing the "n" key.
- The script continuously updates the Pygame window, allowing real-time interaction with the recognition model.

OUTPUT :



After writing on the canvas and releasing the mouse, it takes a few milliseconds before displaying the predicted value on top of it as shown below





We can write as many numbers on the canvas as we want.
We can clear the screen by typing 'n'.