CPE 4903 Final Project: Cat and Dog Classifier, MNIST Handwriting Recognition

VarunKrishnan Raghuraman, Sophomore: Electrical Enginering

Project Overview

- **Objective**
  - Develop a real-time image classification system using your own trained CNN models, along with appropriate harware to generate tangible results.
    i. Cat and Dog Classifier. - Picture of Dog, Cat and human.
    ii. MNIST Handwriting Recognition. - predict black numbers presented on white backround with high confidence.
- CNN models will be trained on a computer IDE and deployed on to a Rasberry Pi to perform inference on the images captures by the camera module.

Learning Outcome

- Train and Deploy CNN models that satisfy the performance requirements.
- Deal with real-world contraints and conditions:
  - sensors
  - image quality
  - real-time speed and performane
  - Hardware/Software compaibility with recent updates, etc.
- Present Hardware and Software integration.
- Set up development platform.
  - Python environment on hardware.
  - Practice trouble shooting skills.

Development System

Hardware:

- Raspberry Pi 4
- Raspberry Pi Camera Module V2-8
- Raspberry Pi Sense HAT
  - Integrated LED Display
- 64GB SD Card
  - Allocated for Raspberry Pi OS storage

- Micro-HDMI to HDMI Cable
  - Used for connecting Raspberry Pi to a monitor
- Type-C Cable
  - Employed as the power supply cable
- USB Drive
  - Facilitates data transfer between laptop and Raspberry Pi

Software:

On Raspberry Pi:

- Python Version: 3.7.12
- TensorFlow Version: 2.4.0
  - Description: Open-source machine learning framework, utilized for various artificial intelligence applications.
- NumPy Version: 1.19.5
  - Description: Fundamental package for scientific computing with Python, essential for array operations.
- OpenCV (CV-2) Version: 4.7
  - Description: Open Source Computer Vision Library, employed for computer vision and image processing tasks.
- Sense HAT
  - Description: An add-on board for Raspberry Pi equipped with sensors, LEDs, and a joystick, often used for environmental monitoring and interactive projects. In this setup, it is specifically employed to display classification outputs.
- Thonny
  - Description: Integrated development environment (IDE) for Python, providing a user-friendly platform for Python programming on Raspberry Pi.

Other:

- IDE: Jupyter Notebook
  - Description: Interactive development environment widely used for data analysis, machine learning, and scientific research.
- Remote Desktops:
  - Thin Client
    - Description: A lightweight client for accessing a remote desktop environment, enhancing accessibility and resource efficiency.
  - VNC Viewer
    - Description: Virtual Network Computing (VNC) viewer enabling remote access to graphical desktops, promoting seamless control and monitoring.

Project Delivery timeline:

	Objective	Date Delivered	Revised-Submission	Reason for resubmission
Prep	Setup Rasberry-pi, Sensehat	11/1/23		VNC issues, resolved with thinclient
Phase -1	Handwriting Digit Classifer	11/7/23	12/8/23	Improved accuracy to 99%>
Phase -1	Cat-Dog Classifier	11/22/23		
Phase -2	Python Environemnt Setup	11/22/23	11/27/23	Issues with camera resolved
Phase -3	Final System Integration	12/10/23		

Theoretical Framework: Understanding the Operations of Convolutional Neural Networks (CNNs)

- Convolutional Neural Networks (CNNs) are particularly well-suited for image classification tasks due to their ability to effectively capture spatial hierarchies and local patterns within images. CNNs are versatile and applicable to various computer vision tasks beyond image classification, such as object detection, segmentation, and even tasks outside traditional computer vision, like natural language processing.
- CNNs have become the standard architecture for image classification tasks, consistently achieving state-of-the-art results on various benchmark datasets as tested through our Homework-6.

Annotated snippet of CNN model from Cat-Dog classifier using Keras

- Here is a layer by layer explanation of the CNN model in my cat/dog classifier

- Importing tools Tensorflow

```
    - from tensorflow.keras.models import Sequential
    - from tensorflow.keras.layers import Conv2D, MaxPooling2D,
Flatten, Dense
```

- Step 1: Create Sequential Model

```
    - model = sequential()
```

- This initializes a sequential model which is a linear stack of the layer where you can add one layer at a time.

- Step 2: Layer 1:

```
    - model.add(Conv2D(32, (3, 3), activation='relu', input_shape=(64,
64, 3)))
    - model.add(MaxPooling2D((2, 2)))
```

- Conv2D: This is the first convolutional layer with 32 filters of size 3x3. The activation function is ReLU, introducing non-linearity. The input shape is specified as (64, 64, 3), assuming the input images are 64x64 pixels with 3 color channels (RGB).
- MaxPooling2D: This layer performs max pooling with a 2x2 pool size, reducing the spatial dimensions by half. Max pooling helps in downsampling and retaining essential features. A simpler but intuitive way to look at max pooling is that it gets rid of the useless pixles while only retaining the least amount of pivotal pixels to recreate the image
- Activation function: relu

- Step 3: Layer 2:

```
    - model.add(Conv2D(64, (3, 3), activation='relu'))
    - model.add(MaxPooling2D((2, 2)))
```

- Conv2D : This is the second convolutional layer with 64 filters of size 3x3. Again, ReLU is used as the activation function.
- MaxPooling2D: Another max pooling layer follows to further downsample the spatial dimensions.

- Step 4: layer 3:

```
    - model.add(Conv2D(128, (3, 3), activation='relu'))
    - model.add(MaxPooling2D((2, 2)))
```

- Conv2D : The third convolutional layer with 128 filters of size 3x3 and ReLU activation.
- MaxPooling2D: Another max pooling layer to downsample.

- Step 5: Flatten layer

```
    - model.add(Flatten())
```

- Flatten: This layer flattens the output of the previous layers into a 1D array, preparing the data for input into the fully connected layers.

- Step 6: Fully Connected (FC) Layers

```
    - model.add(Dense(128, activation='relu'))
    - model.add(Dense(1, activation='sigmoid'))
```

- Dense (Fully Connected Layer 1): A fully connected layer with 128 neurons and ReLU activation.
- Dense (Output Layer): The final layer with a single neuron and sigmoid activation, suitable for binary classification tasks (Cat/dog in our case).

Notable Differences between Cat/Dog Binary classifier and MNIST handwriting recognition model

- Output Layer Activation: - The output layer activation in the MNIST model uses softmax with 10 units, suitable for multi-class classification (10 digits) whereas the Cat/Dog classifier used Sigmoid which is better suited for Binary classfication.

```
    - model_mnist.add(Dense(units=100, activation='relu'))
    - model_mnist.add(Dense(units=10, activation='softmax'))
```

- Nature of classification:

- The MNIST model is designed for a multi-class classification task where each image can belong to one of 10 classes.
- The Cat/Dog model is designed for a binary classification task, where each image is classified as either a cat or a dog.

Design And Implementation:

Software: Cat and Dog Classifier

Installing Tensorflow and Keras

- Before proceeding, it is essential to install Tensorflow and Keras within the Jupyter Notebook environment. These open-source tools are prerequisites for running the upcoming code and are integral to our analytical workflows.
  - !pip install tensorflow[and-cuda]
  - !pip install keras

1) With the provided training zip file (train.zip) containing 25,000 labelled images of cats and dogs (12,500 each)

- Out of those 25,000 images, 6000 images were choses at in order to create sub-directories which can then be used to train and test the model demonstrated above.

train	Validation	Test
60%	15%	25%

2) We then create an Image Data Generator to create data generators for training, validation and testing sets. The objective is to ensure that the Neural Network is fed with properly formatted and preprocessed data during training process.

- The ImageDataGenerator is configured with a rescaling factor of 1.0/255.0. This normalization step ensures that pixel values in the input images are within the standardized range of [0, 1].

```
In [11]:  # create data generator
          datagen = ImageDataGenerator(rescale=1.0/255.0)
```

- The flow_from_directory method is employed to generate a data generator for the training set. The directory structure is assumed to follow a class-wise organization. Each subdirectory contains images belonging to a specific class. In this case, the class_mode is set to 'binary', indicating a binary classification task. The target size of images is set to (64, 64).

```
train_data = datagen.flow_from_directory(train_sub_path + '/train',
             class_mode='binary', batch_size=64, target_size=(64, 64))
```

- Similar configurations are applied to generate data generators for the validation and test sets. These generators are crucial for evaluating the model's performance on unseen data, ensuring that the trained model generalizes well to new examples.

```
val_data = datagen.flow_from_directory(train_sub_path + '/validation',
           class_mode='binary', batch_size=64, target_size=(64, 64))

test_data = datagen.flow_from_directory(train_sub_path + '/test',
            class_mode='binary', batch_size=64, target_size=(64, 64))
```

```
Found 7200 images belonging to 2 classes.
Found 2800 images belonging to 2 classes.
Found 5000 images belonging to 2 classes.
```

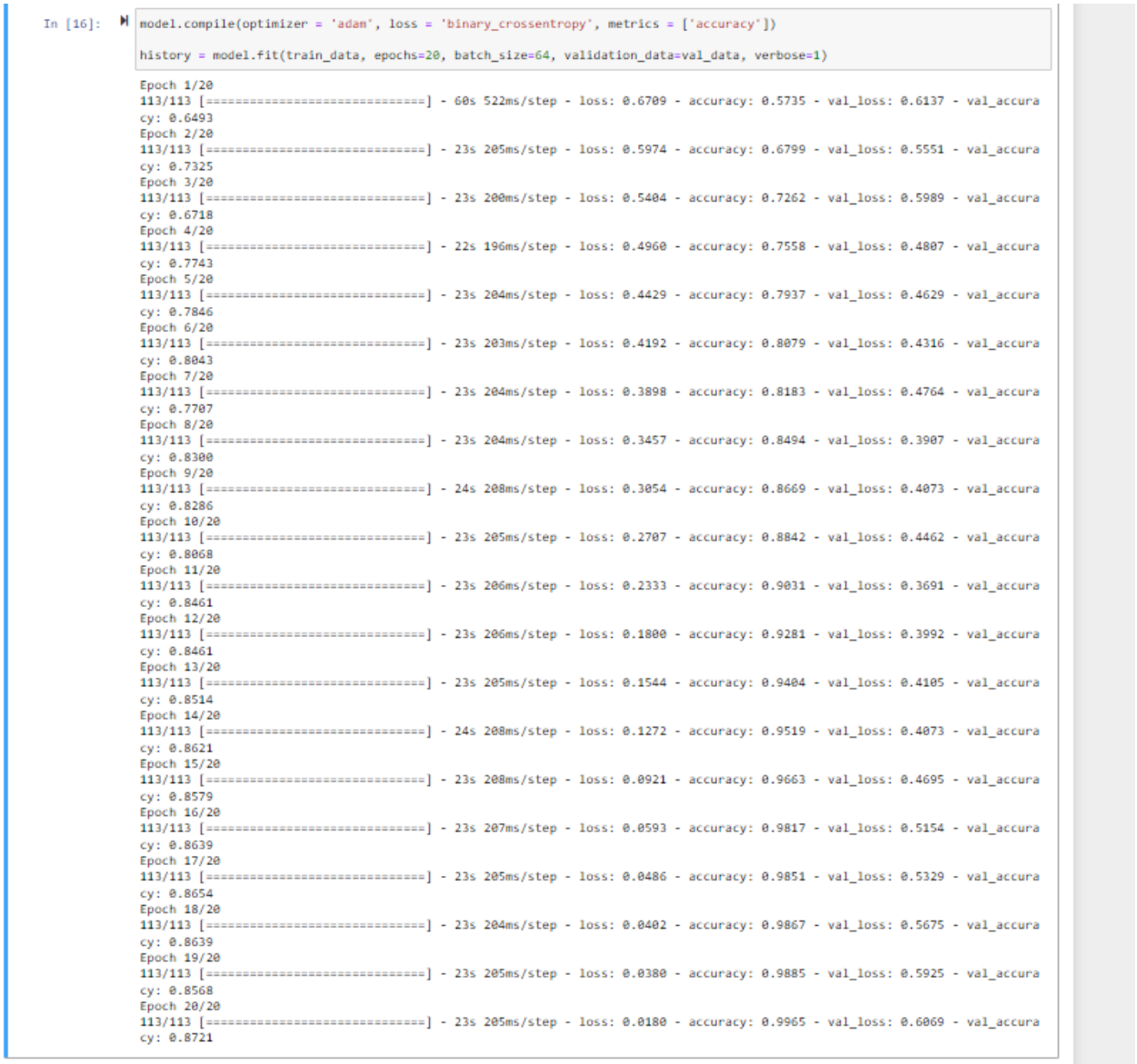
3) After employing the CNN model described above, it is now ready to be trained.

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

- Optimizer='adam': The choice of the Adam optimizer, a popular optimization algorithm for deep learning. It adapts the learning rates during training.
- Loss='binary_crossentropy': This specifies the loss function to be used during training. Binary crossentropy is commonly used for binary classification tasks.
- Metrics=['accuracy']: During training, the model's performance is monitored using accuracy as the evaluation metric.

```
history = model.fit(train_data, epochs=20, batch_size=64, validation_data=val_data, verbose=1)
```

- Train_data: The data generator for training images and labels.
- Epochs=20: The number of times the entire training dataset is passed through the neural network. In this case, it's set to 20 epochs.
- Batch_size=64: The number of samples processed in each training step. It is set to 64.
- Validation_data=val_data: The data generator for validation images and labels. The model's performance on this set is evaluated after each epoch.
- Verbose=1: This parameter controls the amount of information displayed during training. A value of 1 indicates that progress bars are shown for each epoch.
- Screenshot of Cat-Dog training from Jupyter notebook



```
In [16]: model.compile(optimizer = 'adam', loss = 'binary_crossentropy', metrics = ['accuracy'])

         history = model.fit(train_data, epochs=20, batch_size=64, validation_data=val_data, verbose=1)

Epoch 1/20
113/113 [==============================] - 60s 522ms/step - loss: 0.6709 - accuracy: 0.5735 - val_loss: 0.6137 - val_accura
cy: 0.6493
Epoch 2/20
113/113 [==============================] - 23s 205ms/step - loss: 0.5974 - accuracy: 0.6799 - val_loss: 0.5551 - val_accura
cy: 0.7325
Epoch 3/20
113/113 [==============================] - 23s 200ms/step - loss: 0.5404 - accuracy: 0.7262 - val_loss: 0.5989 - val_accura
cy: 0.6718
Epoch 4/20
113/113 [==============================] - 22s 196ms/step - loss: 0.4960 - accuracy: 0.7558 - val_loss: 0.4807 - val_accura
cy: 0.7743
Epoch 5/20
113/113 [==============================] - 23s 204ms/step - loss: 0.4429 - accuracy: 0.7937 - val_loss: 0.4629 - val_accura
cy: 0.7846
Epoch 6/20
113/113 [==============================] - 23s 203ms/step - loss: 0.4192 - accuracy: 0.8079 - val_loss: 0.4316 - val_accura
cy: 0.8043
Epoch 7/20
113/113 [==============================] - 23s 204ms/step - loss: 0.3898 - accuracy: 0.8183 - val_loss: 0.4764 - val_accura
cy: 0.7707
Epoch 8/20
113/113 [==============================] - 23s 204ms/step - loss: 0.3457 - accuracy: 0.8494 - val_loss: 0.3907 - val_accura
cy: 0.8300
Epoch 9/20
113/113 [==============================] - 24s 208ms/step - loss: 0.3054 - accuracy: 0.8669 - val_loss: 0.4073 - val_accura
cy: 0.8286
Epoch 10/20
113/113 [==============================] - 23s 205ms/step - loss: 0.2707 - accuracy: 0.8842 - val_loss: 0.4462 - val_accura
cy: 0.8068
Epoch 11/20
113/113 [==============================] - 23s 206ms/step - loss: 0.2333 - accuracy: 0.9031 - val_loss: 0.3691 - val_accura
cy: 0.8461
Epoch 12/20
113/113 [==============================] - 23s 206ms/step - loss: 0.1800 - accuracy: 0.9281 - val_loss: 0.3992 - val_accura
cy: 0.8461
Epoch 13/20
113/113 [==============================] - 23s 205ms/step - loss: 0.1544 - accuracy: 0.9404 - val_loss: 0.4105 - val_accura
cy: 0.8514
Epoch 14/20
113/113 [==============================] - 24s 208ms/step - loss: 0.1272 - accuracy: 0.9519 - val_loss: 0.4073 - val_accura
cy: 0.8621
Epoch 15/20
113/113 [==============================] - 23s 208ms/step - loss: 0.0921 - accuracy: 0.9663 - val_loss: 0.4695 - val_accura
cy: 0.8579
Epoch 16/20
113/113 [==============================] - 23s 207ms/step - loss: 0.0593 - accuracy: 0.9817 - val_loss: 0.5154 - val_accura
cy: 0.8639
Epoch 17/20
113/113 [==============================] - 23s 205ms/step - loss: 0.0486 - accuracy: 0.9851 - val_loss: 0.5329 - val_accura
cy: 0.8654
Epoch 18/20
113/113 [==============================] - 23s 204ms/step - loss: 0.0402 - accuracy: 0.9867 - val_loss: 0.5675 - val_accura
cy: 0.8639
Epoch 19/20
113/113 [==============================] - 23s 205ms/step - loss: 0.0380 - accuracy: 0.9885 - val_loss: 0.5925 - val_accura
cy: 0.8568
Epoch 20/20
113/113 [==============================] - 23s 205ms/step - loss: 0.0180 - accuracy: 0.9965 - val_loss: 0.6069 - val_accura
cy: 0.8721
```

- Total training time 12:48.

4) Once the Model has been trained, its now ready to be tested:

```
In [21]: loss, accuracy = model.evaluate(test_data, verbose=1)

         y_pred = model.predict(test_data)
```

```
79/79 [==============================] - 17s 218ms/step - loss: 0.5662 - ac
curacy: 0.8764
79/79 [==============================] - 9s 110ms/step
```

- Accuracy of 87% was reached which means we now have a valid model that can be deployed on our Rasberry Pi.

Software: MNIST Handwriting Digit Classifier

1) Import tools and MNIST database

```
In [277]: import numpy as np
          import matplotlib.pyplot as plt
```

```
In [278]: import tensorflow as tf
          from tensorflow import keras
          from keras.utils import np_utils #np utils error
          from tensorflow.keras.preprocessing.image import load_img, img_to_array
          from keras.models import load_model
```

Load CNN models

```
In [279]: from keras.models import Sequential
          from keras.layers import Dense
          from keras.layers import Conv2D
          from keras.layers import MaxPooling2D
          from keras.layers import Flatten
          from keras.layers import Dropout
```

Load the dataset

```
In [169]: from keras.datasets import mnist
```

Troubleshooting

- While using np_utils in order to convert the output y_train and y_test lables to categorical data, i kept getting an "np_utils not found" error.
- In order to get around that, an updated code was deployed as repalacement to the commented:

```
    - from tensorflow.keras.utils import to_categorical
```

- "In order to resolve potential conflicts related to duplicate symbols in the Intel Math Kernel Library (MKL) when using TensorFlow in a macOS environment, the code sets the `KMP_DUPLICATE_LIB_OK` environment variable to 'true'. This adjustment allows for the dynamic loading of libraries with duplicate symbols, addressing a specific issue related to MKL in macOS. This workaround aims to ensure smooth execution of code reliant on TensorFlow and Intel MKL in the Jupyter Notebook environment."

2) After the modules have been loaded, the data can now be Preprocessed

In [6]:
```
(X_train, y_train), (X_test, y_test) = mnist.load_data()
X_train = X_train.reshape(X_train.shape[0], 28, 28, 1)
X_test = X_test.reshape(X_test.shape[0], 28, 28, 1)
y_train = y_train.reshape(-1,1)
y_test = y_test.reshape(-1,1)

print("Training matrix shape-X", X_train.shape)
print("Testing matrix shape-X", X_test.shape)
print("Training matrix shape-Y", y_train.shape)
print("Testing matric shape-Y", y_test.shape)
```

```
Training matrix shape-X (60000, 28, 28, 1)
Testing matrix shape-X (10000, 28, 28, 1)
Training matrix shape-Y (60000, 1)
Testing matric shape-Y (10000, 1)
```

- These lines reshape the input images. The original shape of each image is 28x28 pixels. The reshaping is done to add a new dimension for the channel (since MNIST images are grayscale, there is only one channel). The resulting shape is (number of samples, height, width, channels), which is a common format for convolutional neural networks (CNNs).
- The target labels, "y_train" and "y_test" are also being reshaped. The -1 in the reshape function means that the size of that dimension is inferred based on the size of the array and the other dimensions. The purpose is to reshape the labels into a column vector, which is a common format for the target labels in machine learning tasks.

Normalized Pixel values in order to prevent certain features from dominating the learning process.

In [7]:
```
X_train = X_train.astype('float32')
X_test = X_test.astype('float32')
X_train /= 255
X_test /= 255
```

4) Convert output Labels (y_train and y_test) to categorical data.

In [8]:
```
nb_classes = 10 # number of classification outputs 0..9

Y_train = to_categorical(y_train, nb_classes)
Y_test = to_categorical(y_test, nb_classes)

print('Shape of Y_train =', Y_train.shape)
print('Shape of Y_test =', Y_test.shape)
```

```
Shape of Y_train = (60000, 10)
Shape of Y_test = (10000, 10)
```

- The code transforms original digit labels (0 to 9) into one-hot encoded vectors for a classification task with ten classes. The variable `nb_classes` is set to 10, representing the number of classes. The resulting matrices (`Y_train` and `Y_test`) are printed to confirm the successful one-hot encoding.This is where the troubleshoot was deployed where instead of "np_utils.to_categorical" we can just use "to_categroical".
- A one-hot vector is a way to represent categories in a binary format. Each category is assigned a unique binary code, where only one bit is set to 1 (indicating the presence of that category) and the rest are set to 0. It's commonly used in machine learning to efficiently represent and process categorical data.

3) Define the CNN model.

In [9]:
```
model_cnn = Sequential()

# Layer 1
model_cnn.add(Conv2D(32, (3, 3), input_shape=(28, 28, 1), activation='relu'))
model_cnn.add(Conv2D(32, (3, 3), activation='relu'))
model_cnn.add(MaxPooling2D(pool_size=(2, 2)))

# Layer 2
model_cnn.add(Conv2D(64, (3, 3), activation='relu'))
model_cnn.add(Conv2D(64, (3, 3), activation='relu'))
model_cnn.add(MaxPooling2D(pool_size=(2, 2)))

model_cnn.add(Flatten())

# FC Layers
model_cnn.add(Dense(units=100, activation='relu'))
model_cnn.add(Dense(units=10, activation='softmax'))
```

- As mentioned before, the output layer activation function uses softmax which is ideally preffered in our scenario where we have to differentiate between 10 different numbers.

4) Train the CNN model

```
In [11]: model_cnn.compile(optimizer = 'adam', loss = 'categorical_crossentropy', metr

         history = model_cnn.fit(X_train, Y_train, epochs=10, batch_size=64, validatic
```

```
Epoch 1/10
750/750 [==============================] - 42s 53ms/step - loss: 0.1613 - a
ccuracy: 0.9509 - val_loss: 0.0478 - val_accuracy: 0.9857
Epoch 2/10
750/750 [==============================] - 39s 52ms/step - loss: 0.0452 - a
ccuracy: 0.9860 - val_loss: 0.0485 - val_accuracy: 0.9852
Epoch 3/10
750/750 [==============================] - 50s 66ms/step - loss: 0.0324 - a
ccuracy: 0.9893 - val_loss: 0.0363 - val_accuracy: 0.9891
Epoch 4/10
750/750 [==============================] - 50s 67ms/step - loss: 0.0242 - a
ccuracy: 0.9924 - val_loss: 0.0364 - val_accuracy: 0.9877
Epoch 5/10
750/750 [==============================] - 49s 66ms/step - loss: 0.0167 - a
ccuracy: 0.9950 - val_loss: 0.0311 - val_accuracy: 0.9912
Epoch 6/10
750/750 [==============================] - 50s 66ms/step - loss: 0.0154 - a
ccuracy: 0.9949 - val_loss: 0.0414 - val_accuracy: 0.9886
Epoch 7/10
750/750 [==============================] - 49s 66ms/step - loss: 0.0119 - a
ccuracy: 0.9963 - val_loss: 0.0300 - val_accuracy: 0.9916
Epoch 8/10
750/750 [==============================] - 59s 78ms/step - loss: 0.0110 - a
ccuracy: 0.9964 - val_loss: 0.0487 - val_accuracy: 0.9889
Epoch 9/10
750/750 [==============================] - 56s 74ms/step - loss: 0.0101 - a
ccuracy: 0.9966 - val_loss: 0.0394 - val_accuracy: 0.9912
Epoch 10/10
750/750 [==============================] - 50s 66ms/step - loss: 0.0081 - a
ccuracy: 0.9974 - val_loss: 0.0369 - val_accuracy: 0.9917
```

- Total training time: 6:07.
- Training code is almost identical to cat/dog classifier expect for a few key details:
- Loss = categorical_crossentropy specifies the type of loss function we'll be using his time, specifically 'categroical_crossentropy' was chosen due to the multi-class classification nature of our program.
- Dataset only runs throught the CNN, 10 times (10 epochs) as we dont have as big of a dataset anymore.
- Validation_split is now 0.2 meaning that 20% of data will be randomly selected and set aside as validation set while the remaining 80% will be used to traint the model.

5) Test CNN model

```
In [15]: score = model_cnn.evaluate(X_test, Y_test, verbose=1)

         Y_pred = model_cnn.predict(X_test)

313/313 [==============================] - 4s 11ms/step - loss: 0.0278 - ac
curacy: 0.9923
313/313 [==============================] - 4s 11ms/step
```

Print the final loss and accuracy of the test data

```
In [16]: score[1]#accuracy

Out[16]: 0.9922999739646912
```

- upon testing out model, we can now see that it has achieved an accuracy of 99.22% (99.% >) and is now ready to be deployed onto the Rasberry pi

Hardware: Raspberry-Pi 4

At the end of both of our models, we can save our parameters by using the following commands as an h5 file which can now be deployed onto our Raspberry pi terimnal.

```
model.save('final.h5')
```

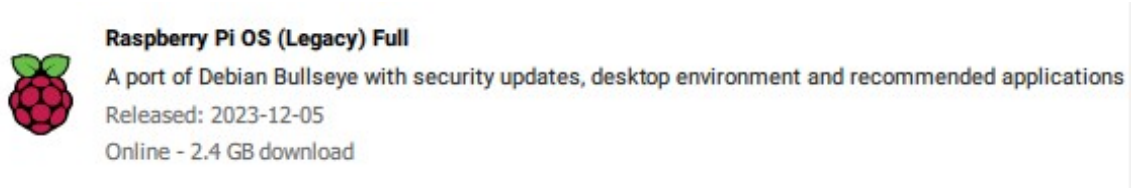
```
model_cnn.save('MNIST_model.h5')
```

Setting up Raspberry-Pi

In order to be able to deploy our code onto our Raspberry pi, we must first do the following:

```
- Configure and flash SD card with and operating system.
- Enable Camera access.
- Setup Python Environment on terminal.
- Download appropriate modules.
        - Tensorflow
        - CV2
        - Numpy
        - SenseHat
```

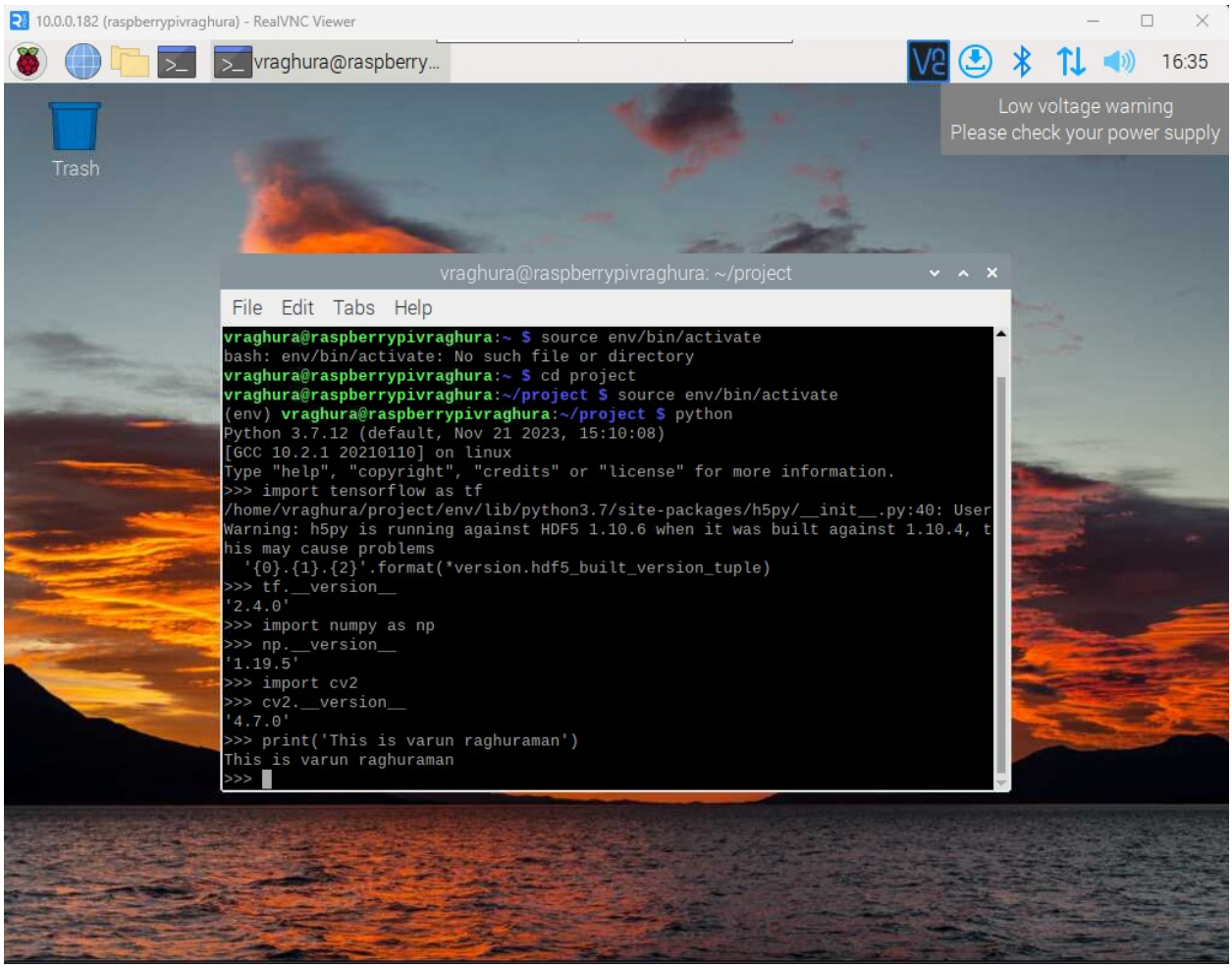
Current OS on Rasberry pi



- upon trial and error, this seemed to be the OS where all of our tools can work in harmorny.

Setting up python Environment

- Following the instructions provided by our TA, was able to successfully established the Python environment on the Raspberry Pi terminal. Despite encountering numerous errors during the setup process, it was determined that all issues stemmed from compatibility issues with the respective operating system.



Thonny - Python IDE on Rasberry-PI

- Thonny, a Python text editor and integrated development environment (IDE), serves as a versatile platform for coding classifier implementations on Raspberry Pi OS. This environment facilitates the creation of classifier scripts, which can then be

conveniently saved to files and executed by calling their source through the terminal.

- Moreover, Thonny played a pivotal role in testing various components of the Raspberry Pi, such as the camera and SenseHAT. Its functionality extends beyond coding, providing a seamless interface for experimentation and verification of hardware functionalities. This integration with both coding and testing aspects underscores Thonny's utility as a comprehensive tool within the Raspberry Pi development ecosystem.

Project Deployment:

Process of Deploying the program from Laptop to Raspberry Pi

1) Once you have succefully trained and tested your model, save it as an h.5 file.

2) Locate and move the h5 file from your local file path to a flashdrive.

3) Open flashdrive on Rasberry pi and move it into your project folder where you will also save your final deployment code.

4) Create code on thonny and save it as a "ImageClassy.py" and "DigitClassy.py".

5) Open terminal and activate environment by using the code: - Source env/bin/activate

6) Now activate python throught the environemnt in order to import nescessary tools and modules: - python - ">> import tensorflow as tf" - ">> import numpy as np" - ">> import cv2" - ">> from sense_hat import SenseHat" - ">> exit()"

7) without exiting the environemnt, use the following lines to deploy and run your respective program program: - python /home/vraghura/numberclassy/Final_number_classifier.py (for Handwriting recognition) - python /home/vraghura/Imageclassy/Final_Image_classifier.py (for cat and dog classifier)

8) With the respective image pulled up on another display, hold camera upto the image to take a clear picture.

9) Prediction with porediction confidence will now be displayed

Annotated TXT Version of my Cat and Dog Classifier code from Thonny.

```
import cv2
import numpy as np
from tensorflow.keras.models import load_model
from sense_hat import SenseHat
import subprocess

# Load the pre-trained model
model = load_model("/home/vraghura/Downloads/final.h5", compile=False)

# Define the expected input shape for the model
expected_input_shape = (64, 64, 3)
```

- all the modules have to be imported twice: - Once onto the thonny file and once onto the python environment before running the code since the enviornment can be considered a clean slate everytime its activated.

```
# Function to preprocess input image
def preprocess_image(image):
    # Convert image to RGB format
    image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
    # Resize the image to match the model's expected input shape
    image = cv2.resize(image, (expected_input_shape[0], expected_input_shape[1]))
    # Normalize pixel values to the range [0, 1]
    image = image / 255.0
    # Add an extra dimension to match the model's input shape
    image = np.expand_dims(image, axis=0)
    return image
```

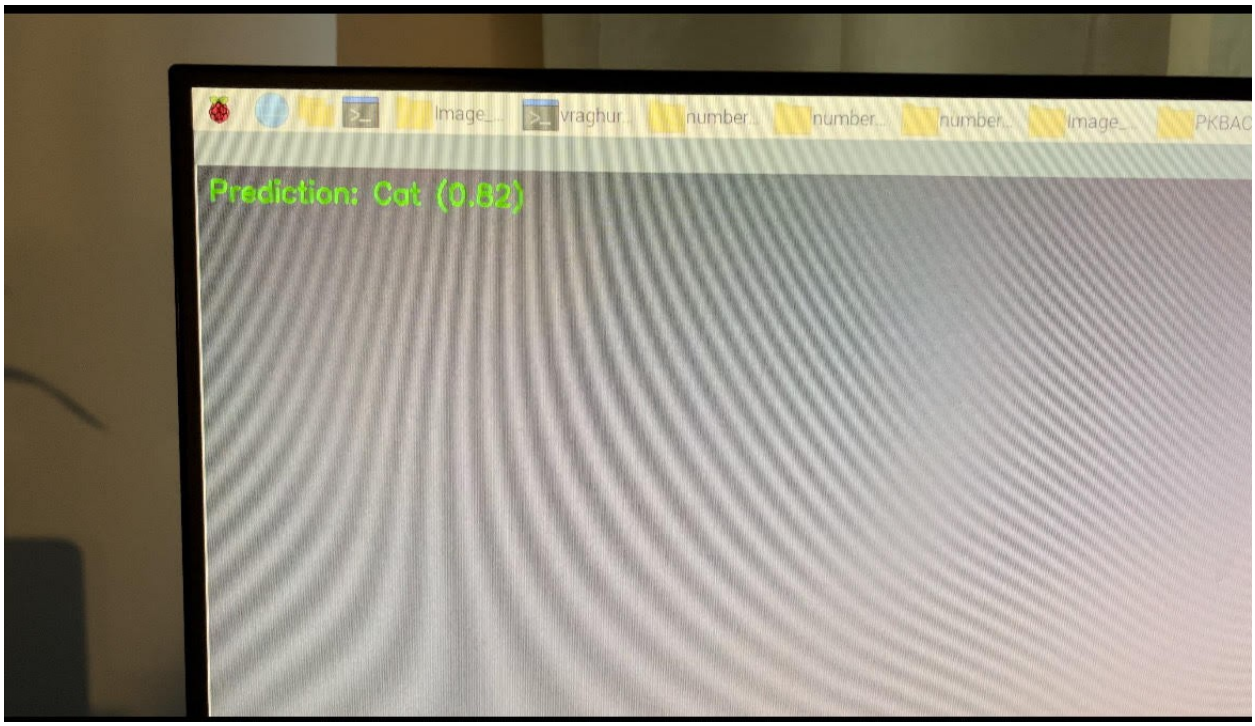
- Converts Image to RGB format and normalizes pixels.

```
# Function to decode predictions
def decode_predictions(predictions):
    # Determine the class label based on the model's output
    class_label = "Dog" if predictions[0, 0] > 0.5 else "Cat"
    # Calculate confidence based on the predicted probability
    confidence = predictions[0, 0] if class_label == "Dog" else 1 - predictions[0, 0]
    return class_label, confidence
```

- Before taking a picture, we create a function to decode prediciton and classify it into two classes, cat or dog.
- Class_label = "Dog" if predictions[0, 0] > 0.5 else "Cat
  - checks to see if the predicted probability output from the models activation function for its specific class and represents the models confidence in predicting that class.

```
# Capture an image using raspistill
subprocess.run(["raspistill", "-o", "/home/vraghura/Image_Classy.py/Captured_image.jpg"])

# Read the captured image
image = cv2.imread('/home/vraghura/Image_Classy.py/Captured_image.jpg')

# Preprocess the image for model input
preprocessed_image = preprocess_image(image)

# Perform inference using the pre-trained model
predictions = model.predict(preprocessed_image)

# Decode the model predictions
class_label, confidence = decode_predictions(predictions)

# Display the results in the terminal
print("Prediction:", class_label)
print("Confidence:", confidence)

# Initialize the Sense HAT
sense = SenseHat()

# Determine LED display color based on the predicted class
color = (0, 255, 0) if class_label == "Dog" else (255, 0, 0)

# Display the predicted class on the Sense HAT LED matrix
sense.show_message(class_label, text_colour=color, back_colour=(0, 0, 0), scroll_speed=0.05)

# Display the image with prediction results
cv2.putText(image, f"Prediction: {class_label} ({confidence:.2f})", (10, 30),
            cv2.FONT_HERSHEY_SIMPLEX, 0.8, (0, 255, 0), 2)
cv2.imshow("Captured Image with Prediction", image)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

- The image capture process is seamlessly integrated into the script without the need for manual button pressing. The use of `raspistill` captures an image, saving it to a specified file path. This approach eliminates the need for physical interaction to trigger image capture, streamlining the process and enhancing the automation of image prediction when the code is initiated in the terminal.
- The program then access the file path, predicts a class and promts up a new window with the image, a prediction and its prediction confidence



- Once the prediciton is made, the code now prints the predicted class name on our Rasberry Pi's SenseHat Display with its assigned colour and scroll speed.
- Visually better represented through the video delivarable

Annotated TXT Version of my Handwriting Digit Classifier code from Thonny.

```
import tensorflow as tf
import cv2
import numpy as np
import subprocess
from sense_hat import SenseHat
from tensorflow.keras.models import load_model
from tensorflow.keras.preprocessing.image import load_img, img_to_array

# Load the pre-trained MNIST model
model = load_model("/home/vraghura/Downloads/MNIST_model.h5", compile=False)

# Initialize the Sense HAT
sense = SenseHat()

# Function to capture an image using raspistill
def capture_image(file_path='/home/vraghura/Image_Classy.py/Captured_image.jpg'):
    # Capture an image using raspistill with specified parameters
    subprocess.run(['raspistill', '-o', file_path, '-w', '640', '-h', '480', '-t', '1000'])
```

- Loading modules and initialing camera using raspistill to take picture automatically, similar to cat/dog classifier.

```
# Function to preprocess and predict the digit
def predict_digit(file_path):
    # Load and prepare the image
    img = load_image(file_path)

    # Predict the class
    pred = model.predict(img)
    digit = np.argmax(pred)
    confidence = pred[0, digit]

    return digit, confidence
```

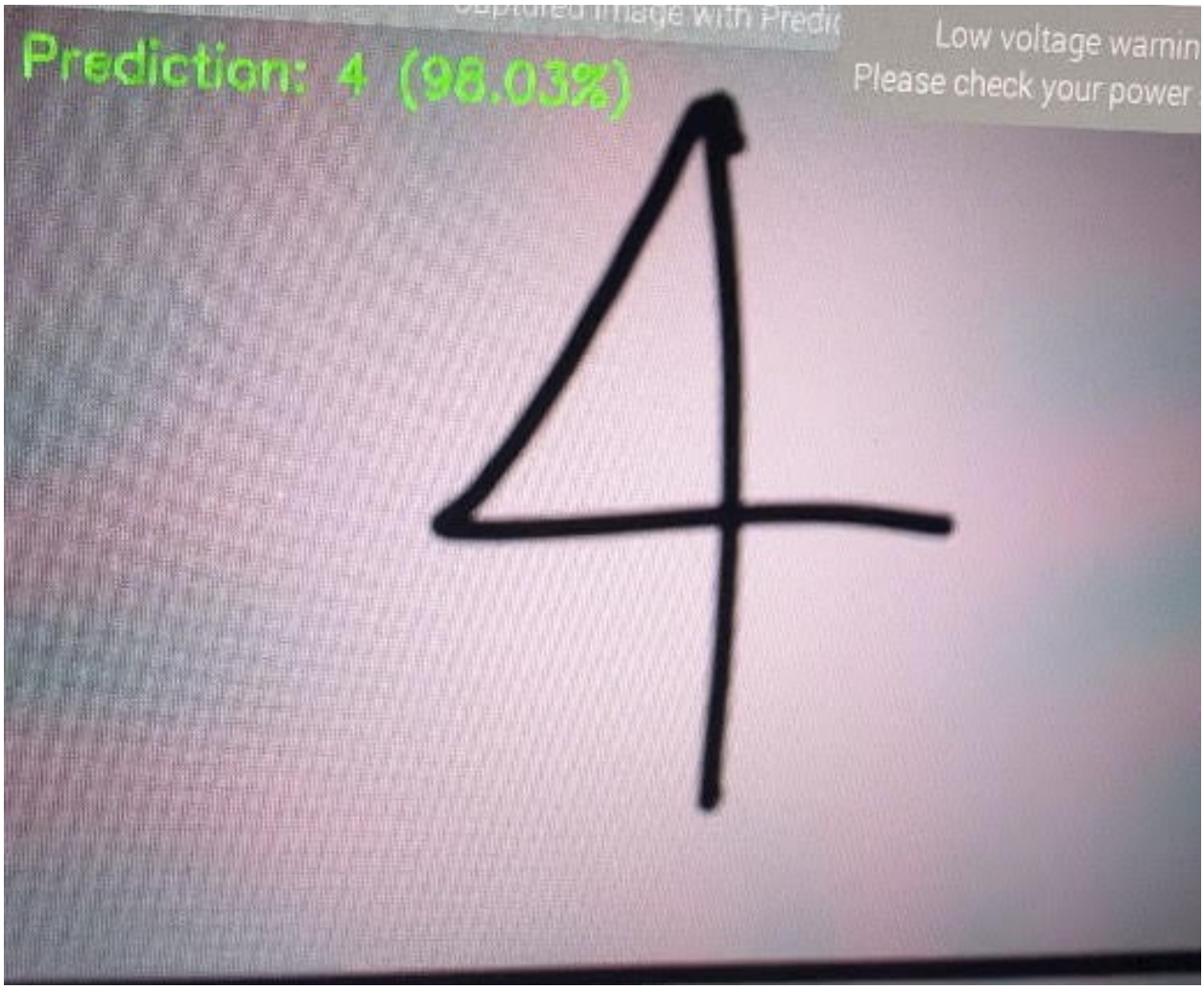
- Function to create multiclass prediction and create confidence.

```
# Function to load and preprocess the image
def load_image(file_path):
    # Load the image
    img = load_img(file_path, color_mode='grayscale', target_size=(28, 28))

    # Convert to array
    img = img_to_array(img)


    img = 255 - img
    #img = cv2.bitwise_not(img)

    # Reshape into a single sample with 1 channel
    img = img.reshape(1, 28, 28, 1)

    # Prepare pixel data
    img = img.astype('float32')
    img = img / 255.0

    return img
```

- Once the camera takes the picture, the image is now converted to array in order to invert the grayscale.
- Since the input images are black numbers on white backrounds when our model was trained with white numbers with black backrounds, we use "img = 255 - img" in order to invert the grayscale.
- img = cv2.bitwise_not(img) is commented off as it was a different method of inverting the grascale i was trying earlier which kept resulting in a "not a number error".

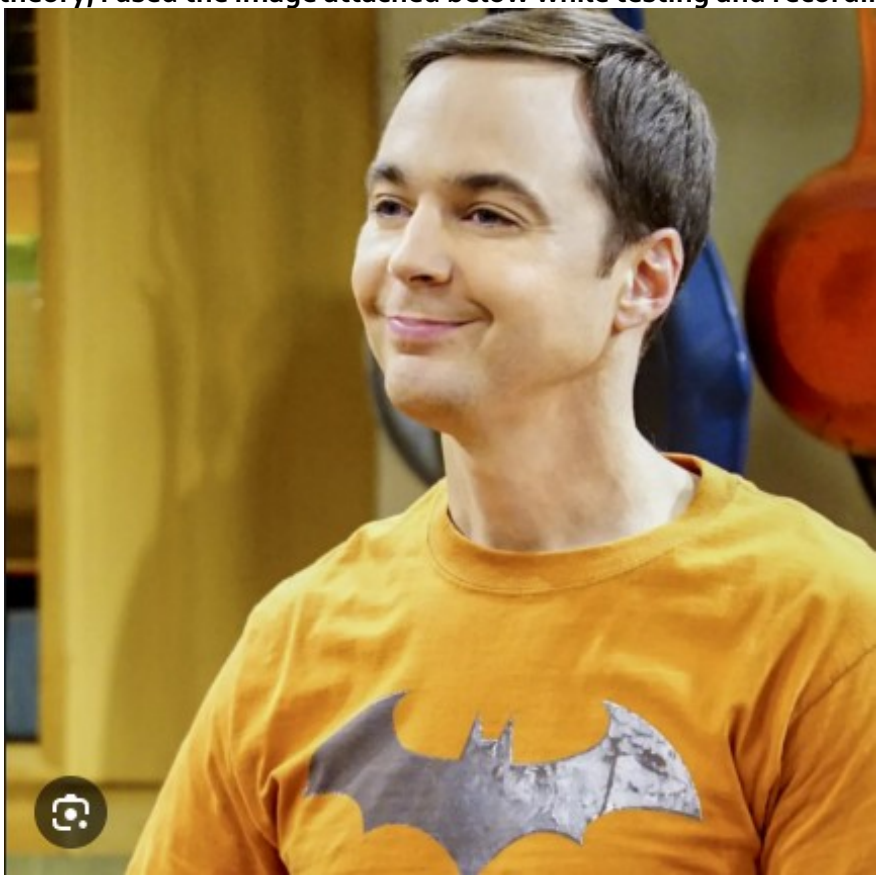
```
# Function to display the digit on the Sense HAT
def display_digit_on_sensehat(digit):
    # Display the predicted digit on the Sense HAT LED matrix
    sense.show_letter(str(digit))

# Capture an image
capture_image()

# Process the captured image and display the result
file_path = '/home/vraghura/Image_Classy.py/Captured_image.jpg'
predicted_digit, confidence = predict_digit(file_path)
print(f"Predicted Digit: {predicted_digit}, Confidence: {confidence:.4f}")

# Display the digit on the Sense HAT
display_digit_on_sensehat(predicted_digit)

# Display the image with prediction results
image = cv2.imread(file_path)
cv2.putText(image, f"Prediction: {predicted_digit} ({confidence:.2%})", (10, 30),
            cv2.FONT_HERSHEY_SIMPLEX, 0.8, (0, 255, 0), 2)
cv2.imshow("Captured Image with Prediction", image)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

- the last part of my Digit classifier code contains the parts where captures image is proccesed the and results including the predicted digit, confidence rating and original image is displayed on the monitor while the predicted number is being displayed on SenseHat.
- Better representation of SenseHat Display on Video Deliverable.


Prediction: 4 (98.03%)
Low voltage warnin
Please check your power

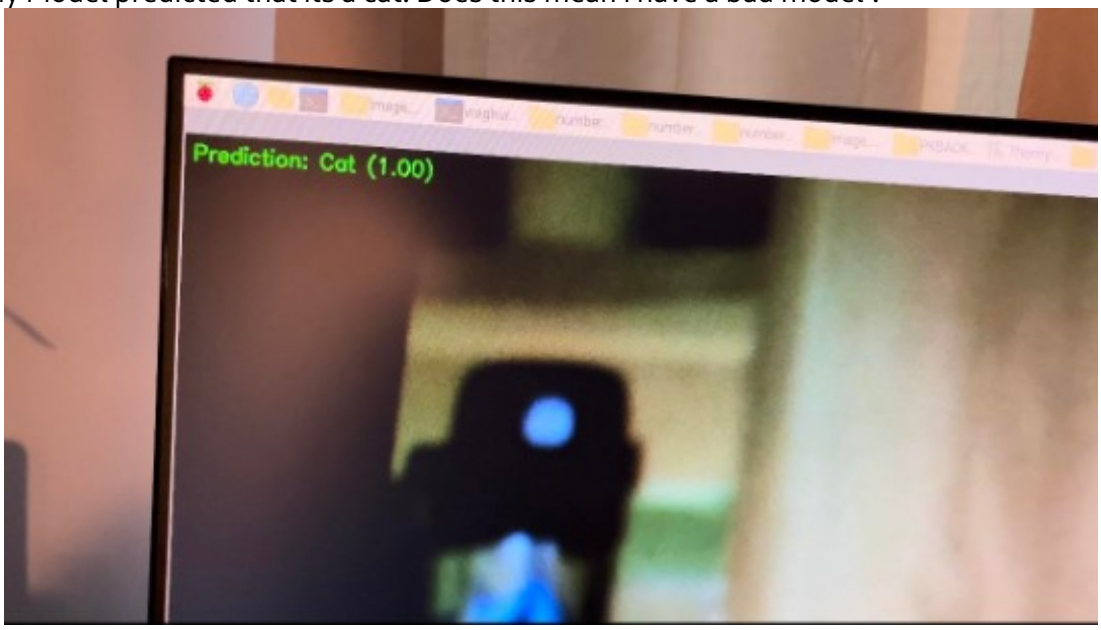
If you show the cat and dog classifier a picture of something else, what will happen?

In order to test this theory, i used the image attached below while testing and recording my deliverable video.



Results:

- My Model predicted that its a cat. Does this mean i have a bad model ?



- the answer depends on various factors, including the context of the image and the complexity of the classification task. If the model is presented with data outside its training distribution, such as a picture of something other than cats or dogs, it may provide inaccurate predictions. This limitation is not indicative of the model being 'bad' but rather highlights its challenge in generalizing to new classes.

- When recording my deliverable video, I mentioned the possibility of overfitting as a potential reason for mispredictions. Overfitting occurs when a model becomes too tailored to the training data, capturing noise and details that do not generalize well to new data. However, it's crucial to note that the prediction of a single image alone doesn't necessarily confirm or refute overfitting. Occasional misclassifications on out-of-distribution samples might occur, emphasizing the importance of evaluating the model's performance on a diverse set of data to ensure robust generalization. It's essential to consider the model's intended use and training objectives to better interpret and address mispredictions in specific scenarios."

- upon further testing across the next few days, i recieved similar prediction but with lower confidence rates as well. (closer to 50%, 60%)

Noteworthy troubleshooting challenges:

RTIMU error while trying to import SenseHat

- Found a Github fix where it was a version mismatch and led me to delete and re-install sensehat into a specific folder where i download RTIMU from online, into the environment. Link to RTIMU error fix

Unable to Load model from .h5 file

- Used this to trouble shoot the error when i was saving my model as weights instead of model which was causing me issues when i was using "load_module" on thonny. Link to Model loading error fix

Value Error: Unkown optimizer: Custom > Adam

- This stack overflow thread suggested a few different solutions to this problem. The first one was the try to import "Adam" through "from tensorflow.keras.optimizers import Adam" but it was in vain. Lastly, i added the parameter (Compile=False) at the end of the command where i import my model which ultimately ended up fixing it. Link to Unkown optimizer fix

Pycamera module not found

- Figuired out that pycamera was not compatible with my operating system which resulted in me using "raspistill" instead. Raspistill fix

Thinclinet remote desktop solution

- During phase: 1 where my remote desktop wasnt working due to incompatibility with its operating system, i used this video as reference to install the nescessary packages to install thinclient's remote dekstop instead which didnt limit me to a particlar operating system.

Thinclient solution

np_utils not found error

- A solution that was posted in our class's group-me suggested using "categorical_" instead of "np_utils_categorical" in MNIST model jupyter file in order to resolve that issue.

Conclusion

In conclusion, navigating the integration of new hardware and software has been a lengthy yet intriguing journey. As a relatively new college student, with three semesters under my belt, the experience of problem-solving and troubleshooting unique errors provided both challenge and enjoyment. This process allowed me to delve into the intricacies of CNN implementation, witnessing the theory in action. Through these endeavors, I've unquestionably attained a heightened understanding of the practical aspects of convolutional neural networks, marking a significant step in my academic journey. The journey, though challenging, has been a source of valuable lessons, shaping my problem-solving skills and deepening my comprehension into tangible applications in the realm of machine learning.