ML4NS

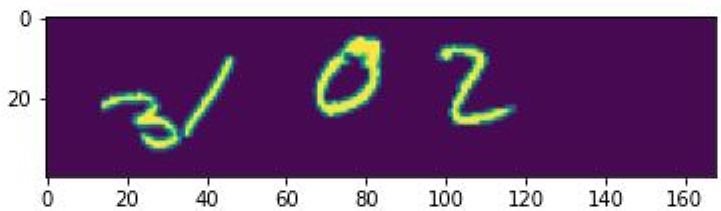
# Mini Project

Kushagra Agarwal
2018113012

## Problem Statement:

Given an image, calculate the sum of digits appearing in the image.

Example:




Correct Label: 6

**Constraints:** **Any weights that are submitted should be reproducible by code. This is why I did not use any pre-trained networks (like VGG or Resnet with pre-trained weights on the Imagenet dataset).**

**Methodology:**

I used 4 different types of networks and then applied **a smart majority voting strategy** to select which model output should be chosen among them. I used 3 end to end neural networks and 1 digit extractor + neural net model.

So basically 3 end to end Models described under the **End to End models section** and the Digit extractor approach described in **Digit Extractor Module** and the **Neural Network trained on the MNIST dataset section**.

# End to End models

https://colab.research.google.com/drive/1HEFDaIn8sQwSL-NDQQIwMgTFjErkoS9_?usp=sharing

I am not describing all the 3 models in detail as all were relatively similar and the same analogies described below more or less extend to them as well.

Train : Validation split used was 80:20

The image was normalized. Predictions were converted to 37 categorical outputs, representing sums from 0 to 36.

```
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_6 (Conv2D)            (None, 38, 166, 32)       320
_________________________________________________________________
batch_normalization_6 (Batch (None, 38, 166, 32)       128
_________________________________________________________________
conv2d_7 (Conv2D)            (None, 36, 164, 32)       9248
_________________________________________________________________
batch_normalization_7 (Batch (None, 36, 164, 32)       128
_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 18, 82, 32)        0
_________________________________________________________________
conv2d_8 (Conv2D)            (None, 16, 80, 64)        18496
_________________________________________________________________
batch_normalization_8 (Batch (None, 16, 80, 64)        256
_________________________________________________________________
conv2d_9 (Conv2D)            (None, 14, 78, 64)        36928
_________________________________________________________________
batch_normalization_9 (Batch (None, 14, 78, 64)        256
_________________________________________________________________
max_pooling2d_4 (MaxPooling2 (None, 7, 39, 64)         0
_________________________________________________________________
conv2d_10 (Conv2D)           (None, 5, 37, 128)        73856
_________________________________________________________________
batch_normalization_10 (Batc (None, 5, 37, 128)        512
_________________________________________________________________
conv2d_11 (Conv2D)           (None, 3, 35, 128)        147584
_________________________________________________________________
batch_normalization_11 (Batc (None, 3, 35, 128)        512
_________________________________________________________________
max_pooling2d_5 (MaxPooling2 (None, 1, 17, 128)        0
_________________________________________________________________
flatten_1 (Flatten)          (None, 2176)              0
_________________________________________________________________
dense_3 (Dense)              (None, 100)               217700
_________________________________________________________________
dense_4 (Dense)              (None, 74)                7474
_________________________________________________________________
dropout_1 (Dropout)          (None, 74)                0
_________________________________________________________________
dense_5 (Dense)              (None, 37)                2775
=================================================================
Total params: 516,173
Trainable params: 515,277
Non-trainable params: 896
```

I also tried various other model architectures:

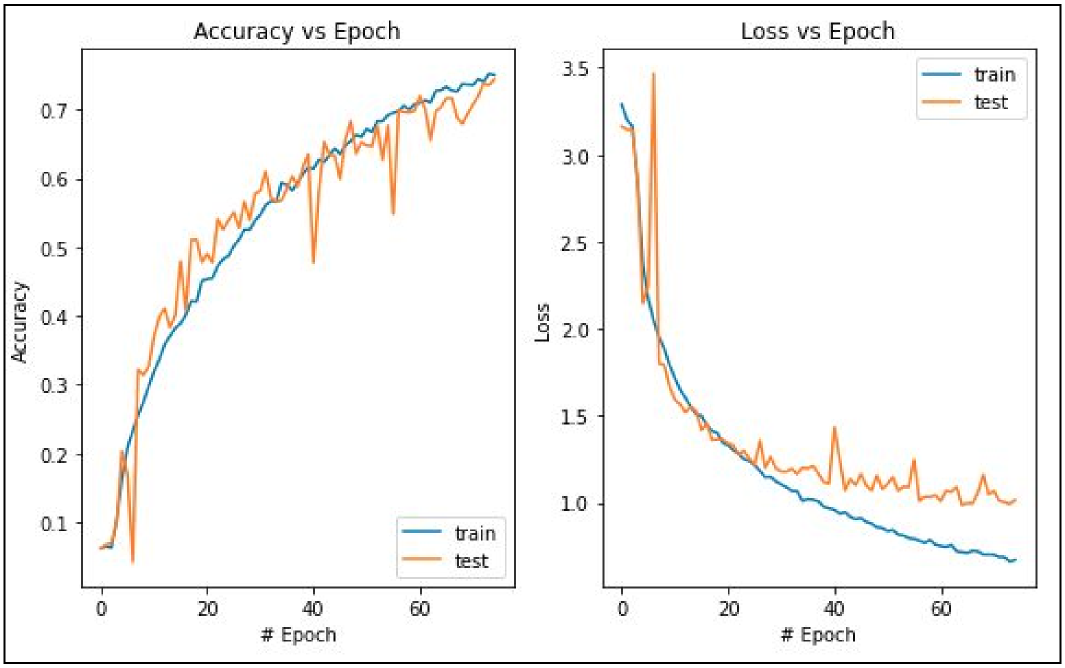
**Simple CNN with FCs**

```
######################## SIMPLE CNN WITH FCs ###############################
# model = Sequential()
# model.add(Conv2D(8, kernel_size=(3, 3), padding = "same", activation='relu', input_shape=input_shape))
# model.add(Conv2D(8, (3, 3), activation='relu', padding = "same"))
# model.add(MaxPool2D(pool_size=(2, 2), strides = 2))
# model.add(Conv2D(16, kernel_size=(3, 3), padding = "same", activation='relu'))
# model.add(Conv2D(16, (3, 3), activation='relu', padding = "same"))
# model.add(MaxPool2D(pool_size=(2, 2), strides = 2))
# model.add(Conv2D(32, kernel_size=(3, 3), padding = "same", activation='relu'))
# model.add(Conv2D(64, (3, 3), activation='relu', padding = "same"))
# model.add(Conv2D(128, (3, 3), activation='relu', padding = "same"))
# model.add(MaxPool2D(pool_size=(2, 2), strides = 2))
# model.add(Flatten())
# model.add(Dense(500, activation='relu'))
# model.add(Dense(100, activation='relu'))
# model.add(Dense(num_category*2, activation='relu'))
# model.add(Dense(num_category, activation='softmax'))
# model.compile(loss=keras.losses.categorical_crossentropy, optimizer='adam', metrics=['accuracy'])
```

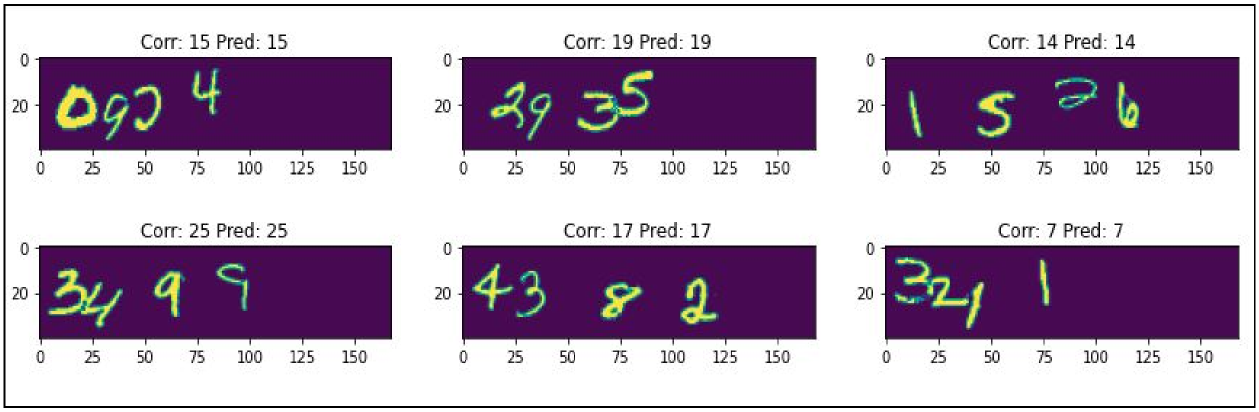
**YANN LECUN's LENET-5 (Modified version)**

```
####################### YANN LECUN's LENET-5 (Modified) ###########################
# model = Sequential()
# model.add(Conv2D(filters=6, kernel_size=(5,5), padding='valid', input_shape=input_shape, activation='relu'))
# model.add(MaxPool2D(pool_size=(2,2)))
# model.add(Conv2D(filters=16, kernel_size=(5,5), padding='valid', activation='relu'))
# model.add(MaxPool2D(pool_size=(2,2)))
# model.add(Flatten())
# model.add(Dense(120, activation='relu'))
# model.add(Dense(84, activation='relu'))
# model.add(Dense(num_category, activation='softmax'))
```

**Accuracy and Traning Loss vs Epochs graphs:**




**Predictions on some images from the Validation Set:**



**Report on Different Metrics:**

############## Classification Report ##############

| | precision | recall | f1-score | support |
|---|---|---|---|---|
| 0 | 0.00 | 0.00 | 0.00 | 1 |
| 1 | 0.00 | 0.00 | 0.00 | 1 |
| 2 | 0.33 | 0.12 | 0.18 | 8 |
| 3 | 0.00 | 0.00 | 0.00 | 15 |
| 4 | 0.15 | 0.14 | 0.15 | 28 |
| 5 | 0.38 | 0.96 | 0.55 | 24 |
| 6 | 0.80 | 0.81 | 0.81 | 59 |
| 7 | 0.88 | 0.79 | 0.83 | 81 |
| 8 | 0.74 | 0.86 | 0.80 | 110 |
| 9 | 0.82 | 0.72 | 0.77 | 132 |
| 10 | 0.78 | 0.79 | 0.78 | 174 |
| 11 | 0.81 | 0.78 | 0.80 | 203 |
| 12 | 0.83 | 0.86 | 0.85 | 272 |
| 13 | 0.86 | 0.80 | 0.83 | 308 |
| 14 | 0.84 | 0.85 | 0.84 | 326 |
| 15 | 0.87 | 0.81 | 0.84 | 381 |
| 16 | 0.78 | 0.85 | 0.81 | 389 |
| 17 | 0.87 | 0.84 | 0.85 | 420 |
| 18 | 0.79 | 0.86 | 0.83 | 355 |
| 19 | 0.79 | 0.85 | 0.82 | 361 |
| 20 | 0.91 | 0.78 | 0.84 | 398 |
| 21 | 0.89 | 0.78 | 0.83 | 374 |
| 22 | 0.79 | 0.87 | 0.83 | 330 |
| 23 | 0.79 | 0.90 | 0.84 | 273 |
| 24 | 0.88 | 0.83 | 0.85 | 246 |
| 25 | 0.86 | 0.83 | 0.84 | 190 |
| 26 | 0.75 | 0.89 | 0.82 | 159 |
| 27 | 0.79 | 0.75 | 0.77 | 127 |
| 28 | 0.81 | 0.68 | 0.74 | 84 |
| 29 | 0.70 | 0.82 | 0.76 | 62 |
| 30 | 0.57 | 0.65 | 0.61 | 48 |
| 31 | 0.38 | 0.29 | 0.33 | 31 |
| 32 | 1.00 | 0.18 | 0.30 | 17 |
| 33 | 0.55 | 0.75 | 0.63 | 8 |
| 34 | 0.00 | 0.00 | 0.00 | 3 |
| 35 | 0.00 | 0.00 | 0.00 | 1 |
| 36 | 0.00 | 0.00 | 0.00 | 1 |
| accuracy | | | 0.81 | 6000 |
| macro avg | 0.62 | 0.61 | 0.60 | 6000 |
| weighted avg | 0.82 | 0.81 | 0.81 | 6000 |

All three models had nearly the same accuracies. **93% on the training set and 80% on the validation set.**

# Digit extractor module:

https://colab.research.google.com/drive/1hHpZSUM7QikMYqbveqsDhHlZgOR1FKj5?usp=sharing

For this, I used contour finding methods to first identify connected components of pixels and then demarcate this using a rectangular boundary.

First I padded the image with zero paddings to make sure no digits touch the border. Then I did the contour finding approach to get the boundaries. Now there were some problems with this approach. For example, there were some digits that touched each other, in this scenario, the connected component analysis was returning 2 digits together. So I used a set of well-defined if and else statements to find out if the aspect ratio of the extracted digit matched with the expected one. If not, then it would divide the boundary with a centerline, with the presumption that it had originally concatenated 2 digits together. A similar method was used to divide digits if 3 of them were concatenated.

After this, I used the bilateral filter on the extracted image and resized it using a custom-defined function resize_to_fit to reshape it to a 28x28 image, which is the image size that our MNIST trained Neural net had learned.

```
for (i,image) in enumerate(train_data[start:end]):

    if(i>0 and i%100 == 0):
      print(i, correct, semicorrect, incorrect, int(correct/(correct+incorrect+semicorrect)*10000)/100,
            int(semicorrect/(correct+incorrect+semicorrect)*10000)/100,
            int(incorrect/(correct+incorrect+semicorrect)*10000)/100, "Done")

    padded_image = cv2.copyMakeBorder(image, 8, 8, 8, 8, cv2.BORDER_REPLICATE)

    if(show == 1):
      plt.imshow(padded_image)
      plt.show()

    # Finding the contours in the image
    contours = cv2.findContours(padded_image.copy(), cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
    contours = contours[1] if imutils.is_cv3() else contours[0]

    letter_image_regions = []

    for contour in contours:
        (x, y, w, h) = cv2.boundingRect(contour)

        if (w > 40):
            # print("Splitting into 3")
            # This contour is too wide to be a single letter, hence split
            onethird_width = int(w / 3)
            letter_image_regions.append((x, y, onethird_width, h))
            letter_image_regions.append((x + onethird_width, y, onethird_width, h))
            letter_image_regions.append((x + 2*onethird_width, y, onethird_width, h))

        elif (w > 20):
            # print("Splitting into 2")
            # This contour is too wide to be a single letter, hence split
            half_width = int(w / 2)
            letter_image_regions.append((x, y, half_width, h))
            letter_image_regions.append((x + half_width, y, half_width, h))

        elif (w< 3 or h<9):
            # print("Random pixel block")
            # Some error in contouring, ignore!
            continue

        else:
            letter_image_regions.append((x, y, w, h))

    # If less/more than 4 detected digits take note of the error
    if len(letter_image_regions) != 4:
        bad_image_count+=1
        if (str(len(letter_image_regions)) not in bad_image_dict):
            bad_image_dict[str(len(letter_image_regions))] = [i]
        else:
            bad_image_dict[str(len(letter_image_regions))].append(i)

    # Sort the detected letter images based on the x coordinate
    letter_image_regions = sorted(letter_image_regions, key=lambda x: x[0])

    sum = 0

    for letter_bounding_box in letter_image_regions:
        x, y, w, h = letter_bounding_box
        letter_image = padded_image[y-2:y + h+2, x-2:x + w+2]
        modified_image = resize_to_fit(letter_image, 28, 28)
        modified_image = modified_image.astype('float32')
```

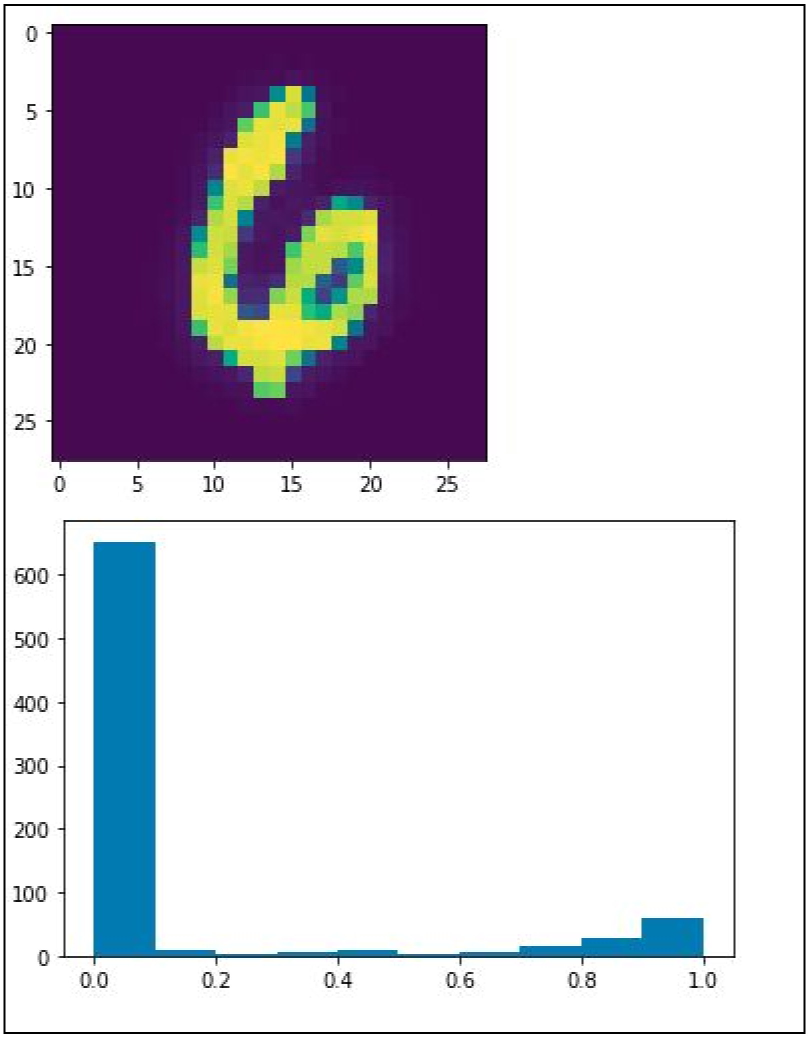
# Neural Net trained on MNIST dataset:

https://colab.research.google.com/drive/1hHpZSUM7QikMYqbveqsDhHlZgOR1FKj5?usp=sharing

```
Model: "sequential_9"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_27 (Conv2D)           (None, 26, 26, 32)        320
_________________________________________________________________
max_pooling2d_18 (MaxPooling (None, 13, 13, 32)        0
_________________________________________________________________
conv2d_28 (Conv2D)           (None, 11, 11, 64)        18496
_________________________________________________________________
conv2d_29 (Conv2D)           (None, 9, 9, 64)          36928
_________________________________________________________________
max_pooling2d_19 (MaxPooling (None, 4, 4, 64)          0
_________________________________________________________________
flatten_9 (Flatten)          (None, 1024)              0
_________________________________________________________________
dense_21 (Dense)             (None, 100)               102500
_________________________________________________________________
dense_22 (Dense)             (None, 10)                1010
=================================================================
Total params: 159,254
Trainable params: 159,254
Non-trainable params: 0
_________________________________________________________________
None
```

The model trained on the MNIST dataset had **99.93% accuracy on training and 99.33% on validation.**

Now I examined the pixel distribution intensity values for the MNIST dataset and the extracted images by the digit extractor module and observed some differences.




In the extracted images, this strong rise in frequency wasn't observed near 1.0. To make it similar to the MNIST dataset, I tried adding different types of noises.

**Poisson Noise:**

```
# poisson noise
def poission_noise(dataset):

  vals = len(np.unique(dataset))
  vals = 2 ** np.ceil(np.log2(vals))
  dataset = np.random.poisson(dataset * vals) / float(vals)
  return dataset
```

**Salt and Pepper Noise:**

```
# salt and pepper noise
def salt_pepper_noise(dataset, prob = 0.005):

    thres = 1 - prob
    for i in range(dataset.shape[0]):
      for j in range(dataset.shape[1]):
        for x in range(dataset.shape[2]):
          rdn = random.random()
          if rdn < prob:
              dataset[i][j][x] = 0
          elif rdn > thres:
              dataset[i][j][x] = 255
          else:
              dataset[i][j][x] = dataset[i][j][x]
    return dataset
```

**Gaussian Noise:**

```
# gaussian noise
def gaussian_noise(dataset, mult=1):

  gauss = np.random.normal(0,1, dataset.size)
  gauss = gauss.reshape(dataset.shape[0],dataset.shape[1],dataset.shape[2]).astype('float32')
  dataset += mult*gauss

  return dataset
```

**Bilateral Noise:**

```
# bilateral filter
def bilateral_filter(dataset, num_pix = 15):

  for i in range(dataset.shape[0]):
    image = dataset[i, :, :]
    mean_image = cv2.bilateralFilter(image, num_pix, 75, 75)
    dataset[i, :, :] = mean_image

  return dataset
```

On trying various combinations, the **bilateral filter added to the extracted images served the best purpose, with 7 and 75*75 neighborhood as parameters.**

After applying the noise, I then normalized the image and predicted its label using my trained Neural Net (which was trained on the MNIST dataset). Then I simply add the predicted digits for all 4 extracted images and report the sum as the label for the image.

On the whole pipeline, **I achieved an 83% accuracy on the validation set**. This was so as the model needed to first extract the four digits correctly and then predict all the 4 labels correctly to make sure that the predicted label matches with the correct label.

# Smart Voting Strategy

https://colab.research.google.com/drive/1_xVTzYKlg45hee_1WAfQ3j2rNCoBQJ9S?usp=sharing

So basically, 3 of my 4 models used an end-to-end approach to predict, and 1 used a 2 step digit extraction and then prediction.

Detailed Description of the 4 models:

| Index | Model Type | Train Accuracy | Validation Accuracy |
|---|---|---|---|
| 1 | End-to-End | 93% | 80% |
| 2 | Digit Extractor | 90+% | 83% |
| 3 | End-to-End | 96% | 75% |
| 4 | End-to-End | 92% | 81% |

So I compared the predictions from all 4 models and the following results were observed.

```
All agree: 5010
1 disagrees: 575
2 disagrees: 1164
3 disagrees: 943
4 disagrees: 471
1 and 2 agree: 49
1 and 3 agree: 283
1 and 4 agree: 370
2 and 3 agree: 44
2 and 4 agree: 71
3 and 4 agree: 283
2 on both sides: 604
None agree: 133
```

Out of 10000 test set images, 5010 were predicted to have the same output label by all 4 models. Other subcategories are in a similar way self-explanatory.

So for all these categories, I chose different models output, which is described below:

| Category | Count | Chosen model |
|---|---|---|
| All agree | 5010 | 1 or 2 or 3 or 4 |
| 1 disagrees | 575 | 2 or 3 or 4 |
| 2 disagrees | 1164 | 1 or 3 or 4 |
| 3 disagrees | 943 | 1 or 2 or 4 |
| 4 disagrees | 471 | 1 or 2 or 3 |
| 1 and 2 agree | 49 | 1 or 2 |
| 1 and 3 agree | 283 | 1 or 3 |
| 1 and 4 agree | 370 | 1 or 4 |
| 2 and 3 agree | 44 | 2 or 3 |
| 2 and 4 agree | 71 | 2 or 4 |
| 3 and 4 agree | 283 | 3 or 4 |
| 2 agree on both sides | 604 | 2 |
| None Agree | 133 | 2 |
| **Total** | **10000** | |

All the other chosen models are self-explanatory except for 2 agree on both sides and None Agree. For 2 agree on both sides, I chose Model 2, as it is the only independent model uncorrelated with the other 3 and if its prediction (which is indeed a sum of 4 individual predictions) matches with the output of any other model, then it must be onto something. A similar analogy for None agree exists, as if none of the models are giving a correct answer, we should instead go for either the highest accuracy model or the Independent model, and here as all the models have nearly the same accuracies, I chose to go with the model 2 which is truly uncorrelated with other 3.