Report of Hand-Written Digits Classification Problem

Junzhou Fang
ECE, UIUC
Champaign, United States
junzhou5@illinois.edu

***Abstract*—This report is a summary of the hand-written digits classification problem. We included three methods: logistic regression with feature extraction, perceptron, CNN, and a simple algorithm, to solve the problem.**

***Index Terms*—Binary classification, logistic regression, CNN, MNIST**

I. Introduction

Professor Do's research team has collected a large amount of image hand-written data, but some of the image are blurred during the collection due to technological errors. The goal of this report is to classify the images into two categories: blurred (represent as 1) and non-blurred (represent as -1).

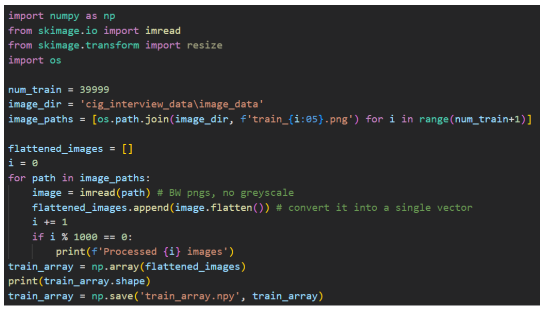
II. Data Preprocessing

A. Data Decription

There are three sets of images: train set, validation set and test set. The train set contains 40000 images, the validation set contains 10000 images and the test set contains 8000 images. Each image is in size of (28,28) and is in black and white. In addition to images, there are three text files containing labels of images in each set. We use 1 to represent unblurred images and -1 to represent blurred images.

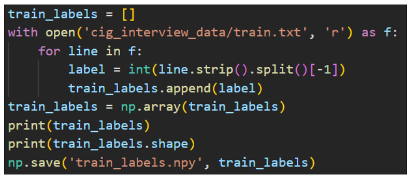
B. Data Loading

We use skimage to load all the images and convert them into a numpy array. We use flatten() to contruct image sets into a 2D array, with each image is a vector.
We also read labels into a numpy array. The index of the array is the index of corresponding image.



```python
import numpy as np
from skimage.io import imread
from skimage.transform import resize
import os

num_train = 39999
image_dir = 'cig_interview_data\image_data'
image_paths = [os.path.join(image_dir, f'train_{i:05}.png') for i in range(num_train+1)]

flattened_images = []
i = 0
for path in image_paths:
    image = imread(path) # BW pngs, no greyscale
    flattened_images.append(image.flatten()) # convert it into a single vector
    i += 1
    if i % 1000 == 0:
        print(f'Processed {i} images')
train_array = np.array(flattened_images)
print(train_array.shape)
train_array = np.save('train_array.npy', train_array)
```

Fig. 1. Loading train images



```python
train_labels = []
with open('cig_interview_data/train.txt', 'r') as f:
    for line in f:
        label = int(line.strip().split()[-1])
        train_labels.append(label)
train_labels = np.array(train_labels)
print(train_labels)
print(train_labels.shape)
np.save('train_labels.npy', train_labels)
```

Fig. 2. Loading train labels

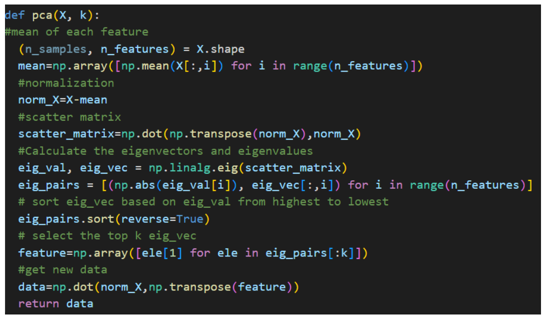
C. Data Normalization

Since BW images has color range from 0 to 255, we use normalization to scale the color range to 0 to 1. Initially, we perform an additional normalization using the mean and standard deviation. But later we find out that this step has little effect on the accuracy, so we remove it.

III. Logistic Regression

A. Feature Extraction

We perform PCA to extract original $28 * 28$ features into 50. First, we implement the PCA on our own. However, later we find out that sklearn's PCA function is much faster and more efficient. Our own PCA is incapable for handling this large amount of data, particularly on a personal laptop. Also, we use PCA several times in model tunning. Therefore, we use sklearn's PCA function to perform PCA.

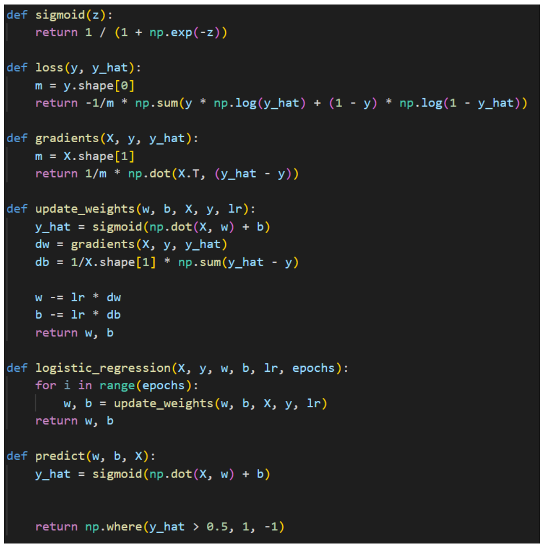


```python
def pca(X, k):
#mean of each feature
  (n_samples, n_features) = X.shape
  mean=np.array([np.mean(X[:,i]) for i in range(n_features)])
  #normalization
  norm_X=X-mean
  #scatter matrix
  scatter_matrix=np.dot(np.transpose(norm_X),norm_X)
  #Calculate the eigenvectors and eigenvalues
  eig_val, eig_vec = np.linalg.eig(scatter_matrix)
  eig_pairs = [(np.abs(eig_val[i]), eig_vec[:,i]) for i in range(n_features)]
  # sort eig_vec based on eig_val from highest to lowest
  eig_pairs.sort(reverse=True)
  # select the top k eig_vec
  feature=np.array([ele[1] for ele in eig_pairs[:k]])
  #get new data
  data=np.dot(norm_X,np.transpose(feature))
  return data
```

Fig. 3. Our PCA

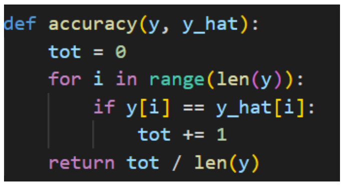
B. Code

We implement the logistic regression basing only on numpy. Also, we use the following accuracy() to evaluate the model. It's worth mentioning that to fit this model, we need to first



```python
def sigmoid(z):
    return 1 / (1 + np.exp(-z))

def loss(y, y_hat):
    m = y.shape[0]
    return -1/m * np.sum(y * np.log(y_hat) + (1 - y) * np.log(1 - y_hat))

def gradients(X, y, y_hat):
    m = X.shape[1]
    return 1/m * np.dot(X.T, (y_hat - y))

def update_weights(w, b, X, y, lr):
    y_hat = sigmoid(np.dot(X, w) + b)
    dw = gradients(X, y, y_hat)
    db = 1/X.shape[1] * np.sum(y_hat - y)

    w -= lr * dw
    b -= lr * db
    return w, b

def logistic_regression(X, y, w, b, lr, epochs):
    for i in range(epochs):
        w, b = update_weights(w, b, X, y, lr)
    return w, b

def predict(w, b, X):
    y_hat = sigmoid(np.dot(X, w) + b)


    return np.where(y_hat > 0.5, 1, -1)
```

Fig. 4. Logistic Regression



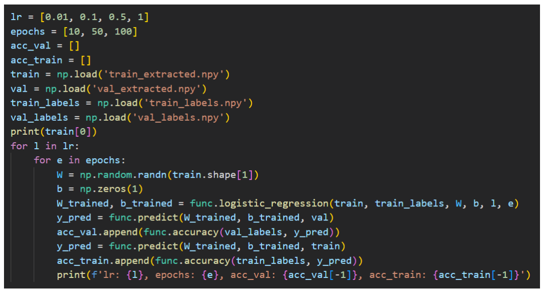
```python
def accuracy(y, y_hat):
    tot = 0
    for i in range(len(y)):
        if y[i] == y_hat[i]:
            tot += 1
    return tot / len(y)
```

Fig. 5. Accuracy

modify the labels as 0 and 1. The output of the model is also 0 and 1, so we further add some code to convert it back to -1 and 1.

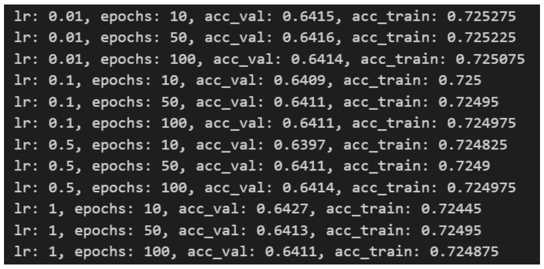
C. Output & Evaluation

We use following code to evaluate and show the result of the model. Here we combine four choices of learning rate (lr) and three choices of epochs. The output is shown as follows. For train accuracy, we are using the entire train set. As you can see, the best accuracy is given by (lr = 0.01, epochs = 100). However, even the best accuracy can only achieve 0.73 on train set and 0.64 on validation set. I think perhaps the reason is that the values for each pixel in the image is too extreme, i.e. they are either close to black(0) or close to white(1), as shown in fig8. This makes feature extraction and normalization



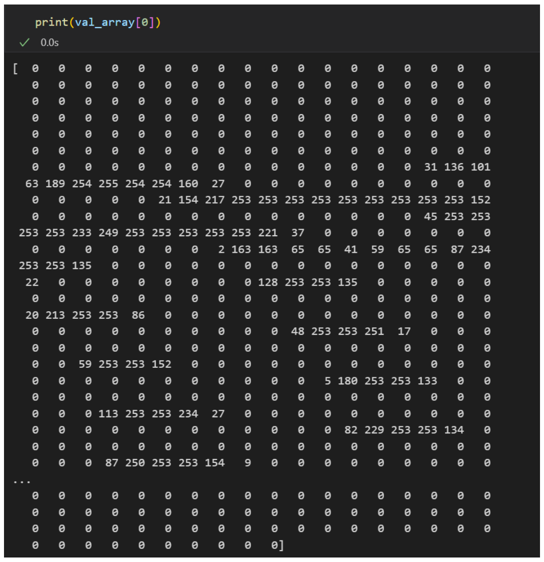
```python
lr = [0.01, 0.1, 0.5, 1]
epochs = [10, 50, 100]
acc_val = []
acc_train = []
train = np.load('train_extracted.npy')
val = np.load('val_extracted.npy')
train_labels = np.load('train_labels.npy')
val_labels = np.load('val_labels.npy')
print(train[0])
for l in lr:
    for e in epochs:
        W = np.random.randn(train.shape[1])
        b = np.zeros(1)
        W_trained, b_trained = func.logistic_regression(train, train_labels, W, b, l, e)
        y_pred = func.predict(W_trained, b_trained, val)
        acc_val.append(func.accuracy(val_labels, y_pred))
        y_pred = func.predict(W_trained, b_trained, train)
        acc_train.append(func.accuracy(train_labels, y_pred))
        print(f'lr: {l}, epochs: {e}, acc_val: {acc_val[-1]}, acc_train: {acc_train[-1]}')
```

Fig. 6. Code for output



```
lr: 0.01, epochs: 10, acc_val: 0.6415, acc_train: 0.725275
lr: 0.01, epochs: 50, acc_val: 0.6416, acc_train: 0.725225
lr: 0.01, epochs: 100, acc_val: 0.6414, acc_train: 0.725075
lr: 0.1, epochs: 10, acc_val: 0.6409, acc_train: 0.725
lr: 0.1, epochs: 50, acc_val: 0.6411, acc_train: 0.72495
lr: 0.1, epochs: 100, acc_val: 0.6411, acc_train: 0.724975
lr: 0.5, epochs: 10, acc_val: 0.6397, acc_train: 0.724825
lr: 0.5, epochs: 50, acc_val: 0.6411, acc_train: 0.7249
lr: 0.5, epochs: 100, acc_val: 0.6414, acc_train: 0.724975
lr: 1, epochs: 10, acc_val: 0.6427, acc_train: 0.72445
lr: 1, epochs: 50, acc_val: 0.6413, acc_train: 0.72495
lr: 1, epochs: 100, acc_val: 0.6411, acc_train: 0.724875
```

Fig. 7. Output



```
print(val_array[0])
✓ 0.0s
[   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
    0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
    0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
    0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
    0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
    0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
    0   0   0   0   0   0   0   0   0   0   0   0   0   0   0  31 136 101
   63 189 254 255 254 254 160  27   0   0   0   0   0   0   0   0   0   0
    0   0   0   0   0  21 154 217 253 253 253 253 253 253 253 253 253 152
    0   0   0   0   0   0   0   0   0   0   0   0   0   0   0  45 253 253
  253 253 233 249 253 253 253 253 253 221  37   0   0   0   0   0   0   0
    0   0   0   0   0   0   0   2 163 163  65  65  41  59  65  65  87 234
  253 253 135   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
   22   0   0   0   0   0   0   0   0 128 253 253 135   0   0   0   0   0
    0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
   20 213 253 253  86   0   0   0   0   0   0   0   0   0   0   0   0   0
    0   0   0   0   0   0   0   0   0   0  48 253 253 251  17   0   0   0
    0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
    0   0  59 253 253 152   0   0   0   0   0   0   0   0   0   0   0   0
    0   0   0   0   0   0   0   0   0   0   0   5 180 253 253 133   0   0
    0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
    0   0   0 113 253 253 234  27   0   0   0   0   0   0   0   0   0   0
    0   0   0   0   0   0   0   0   0   0   0   0  82 229 253 253 134   0
    0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
    0   0   0  87 250 253 253 154   9   0   0   0   0   0   0   0   0   0
...
    0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
    0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
    0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
    0   0   0   0   0   0   0   0   0   0]
```

Fig. 8. Vector of an image

less effective. We have also tried different numbers of features to extract to, but features number ranging from 50 to 256 gives similar result. It's hardly visialilzed bacause 256 features nearly blow up my computer. Therefore, we decide to use CNN to solve this problem.

IV. Perceptron

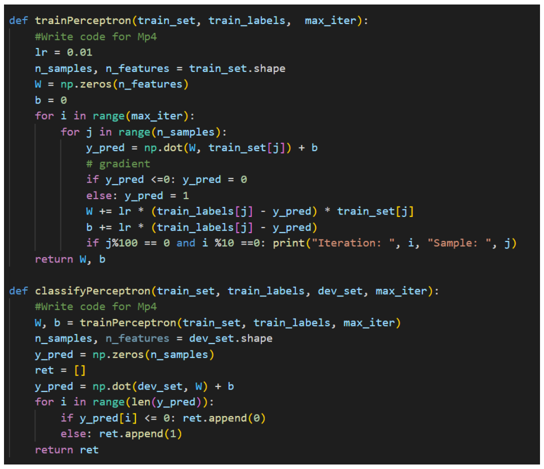
Since the logistic regression is not working well, we try to use perceptron to solve the problem.

A. Code

Perceptron is relatively simple for implementation.

B. Output & Evaluation

Initially, we use PCA to extract dataset to 50 features. But the result accuracy is close to 0.5, a complete failure. Therefore, we discard the feature extraction and directly perform perceptron. This time the accuracy reach around 0.89 on validation set. Also, running time is not a big consideration for this project, since whether using PCA or not only make 10 seconds difference. Therefore, we draw a conclusion that the feature extraction is not essential for this problem.
We also try to tune the learning rate and epochs. It turns out that if the epochs is 100, a learning rate ranging from 0.01 to 0.1 give similar performance. This can be explained by the



```python
def trainPerceptron(train_set, train_labels,  max_iter):
    #Write code for Mp4
    lr = 0.01
    n_samples, n_features = train_set.shape
    W = np.zeros(n_features)
    b = 0
    for i in range(max_iter):
        for j in range(n_samples):
            y_pred = np.dot(W, train_set[j]) + b
            # gradient
            if y_pred <=0: y_pred = 0
            else: y_pred = 1
            W += lr * (train_labels[j] - y_pred) * train_set[j]
            b += lr * (train_labels[j] - y_pred)
            if j%100 == 0 and i %10 ==0: print("Iteration: ", i, "Sample: ", j)
    return W, b

def classifyPerceptron(train_set, train_labels, dev_set, max_iter):
    #Write code for Mp4
    W, b = trainPerceptron(train_set, train_labels, max_iter)
    n_samples, n_features = dev_set.shape
    y_pred = np.zeros(n_samples)
    ret = []
    y_pred = np.dot(dev_set, W) + b
    for i in range(len(y_pred)):
        if y_pred[i] <= 0: ret.append(0)
        else: ret.append(1)
    return ret
```

Fig. 9. Perceptron

fact that blurred and unblurred images are quite different, and thus a local mininum of loss function is easily reached.
This also lead us to wonder if there is a even simpler way to solve the problem.

V. A Simple Approach

When I try to improve the logistic regression, it's really hard due to the discussion above. However, I find some interesting feature. If you compare a blurred and an unblurred image of the same digit, you will realize that the most significant difference is the number of grey pixels. Typically, a blurred image will have more pixels with value ranging between $(0.3 * 255, 0.7 * 255)$, about 2 to 6 times of the number of pixels in this range for an unblurred image. Therefore, we come up with a simple algorithm that counts the number of pixels in range $(0.3 * 255, 0.7 * 255)$, use different thresholds values to categorize whether a picure is blurred or not.

A. Code

First, process the data to get an array to count the number of pixels that lies in the range for each image. And make the decision compeletely on the thresholds, and evaluate the accuracy.

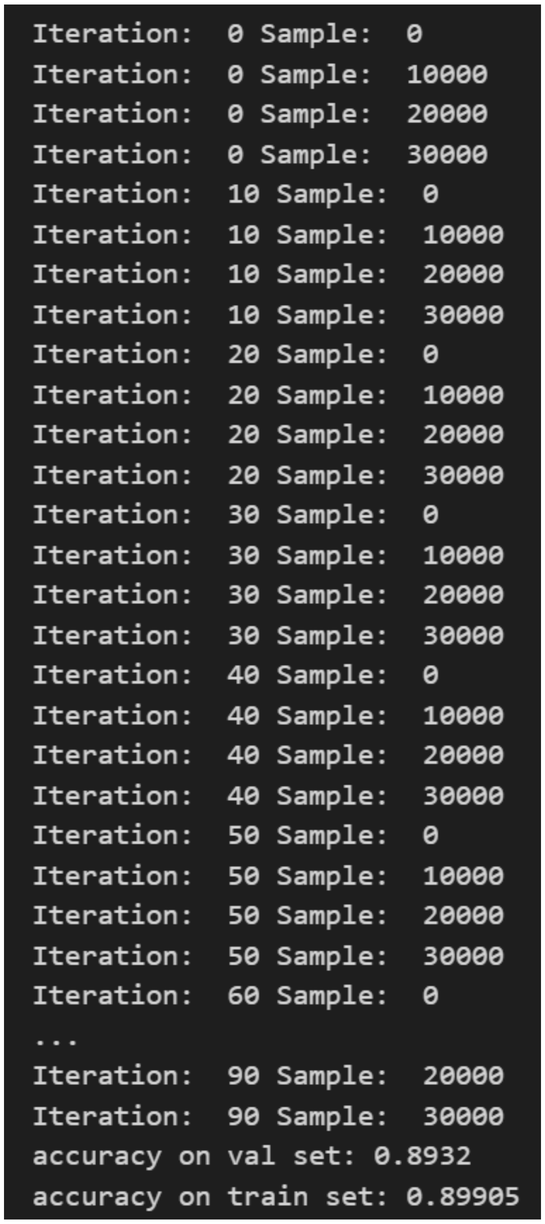
B. Threshold

We try several threshold, and find the best one on both train set and validation set is 100. The accuracy can achieve 0.89 on both train set and validation set.

VI. CNN

A. Data Preprocessing

The reason we adopt CNN to solve the problem is that CNN is powerful in handling image data, especially in this case the image is in good quality and the ouput is only binary. Its convolution layer will be so happy to get this problem done. Here we directly load all the train images into a (28*28, 40000) array. Given the dicovery that both blurred and unblurred images have same value in black area, one may



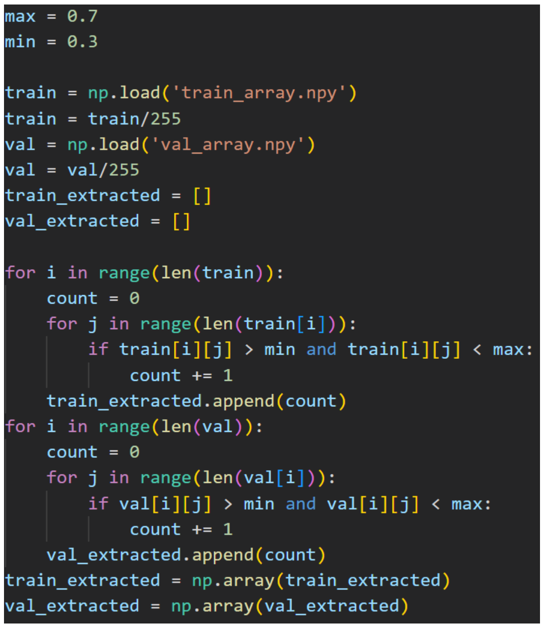
```
Iteration:  0 Sample:  0
Iteration:  0 Sample:  10000
Iteration:  0 Sample:  20000
Iteration:  0 Sample:  30000
Iteration:  10 Sample:  0
Iteration:  10 Sample:  10000
Iteration:  10 Sample:  20000
Iteration:  10 Sample:  30000
Iteration:  20 Sample:  0
Iteration:  20 Sample:  10000
Iteration:  20 Sample:  20000
Iteration:  20 Sample:  30000
Iteration:  30 Sample:  0
Iteration:  30 Sample:  10000
Iteration:  30 Sample:  20000
Iteration:  30 Sample:  30000
Iteration:  40 Sample:  0
Iteration:  40 Sample:  10000
Iteration:  40 Sample:  20000
Iteration:  40 Sample:  30000
Iteration:  50 Sample:  0
Iteration:  50 Sample:  10000
Iteration:  50 Sample:  20000
Iteration:  50 Sample:  30000
Iteration:  60 Sample:  0
...
Iteration:  90 Sample:  20000
Iteration:  90 Sample:  30000
accuracy on val set: 0.8932
accuracy on train set: 0.89905
```

Fig. 10. Accuracy of Perceptron

leave the outer black edges to reduce input size. However, later we find that CNN is so powerful, that the time to train the model in original size is even faster than a for loop to remove all the edge and then train. Therefore, we directly use original size to train the model.

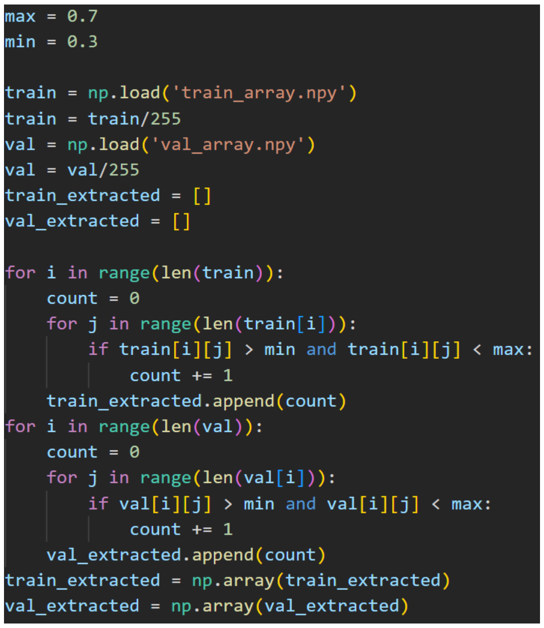
B. Model Construction

We use tensorflow and keras to contruct the model. More specifically, there are three convolutional layers with ReLU activation function, each layer is followed by a max pooling layer. And then we use a flatten layer to convert the 2D array into 1D array, applying a dense layer of 512 neurons to keep information. Finally, a dense layer of 1 neuron is used for binary output.



```
max = 0.7
min = 0.3

train = np.load('train_array.npy')
train = train/255
val = np.load('val_array.npy')
val = val/255
train_extracted = []
val_extracted = []

for i in range(len(train)):
    count = 0
    for j in range(len(train[i])):
        if train[i][j] > min and train[i][j] < max:
            count += 1
    train_extracted.append(count)
for i in range(len(val)):
    count = 0
    for j in range(len(val[i])):
        if val[i][j] > min and val[i][j] < max:
            count += 1
    val_extracted.append(count)
train_extracted = np.array(train_extracted)
val_extracted = np.array(val_extracted)
```

Fig. 11. Simple Algorithm Data



```
max = 0.7
min = 0.3

train = np.load('train_array.npy')
train = train/255
val = np.load('val_array.npy')
val = val/255
train_extracted = []
val_extracted = []

for i in range(len(train)):
    count = 0
    for j in range(len(train[i])):
        if train[i][j] > min and train[i][j] < max:
            count += 1
    train_extracted.append(count)
for i in range(len(val)):
    count = 0
    for j in range(len(val[i])):
        if val[i][j] > min and val[i][j] < max:
            count += 1
    val_extracted.append(count)
train_extracted = np.array(train_extracted)
val_extracted = np.array(val_extracted)
```

Fig. 12. Simple Algorithm

C. *Output Evaluation*

Initially, we set epochs to 5 and batch size to 50. This small epochs compeletely comes from my faith in CNN. It's still surprising to see this combiantion of epochs and batch size gives an accuracy of 1. To further concrete this is not a lie, I print the first 31 albels on train set to mannually check, and they are indeed the same.
Good work CNN!

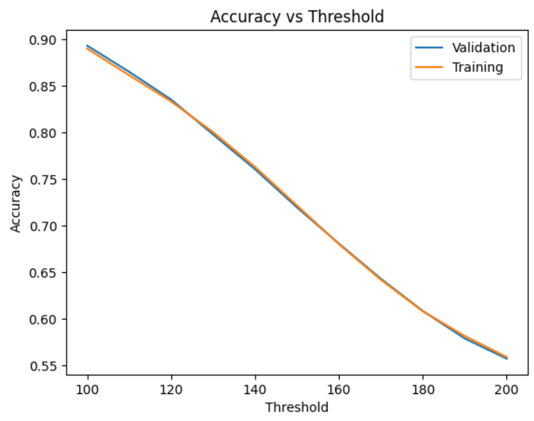
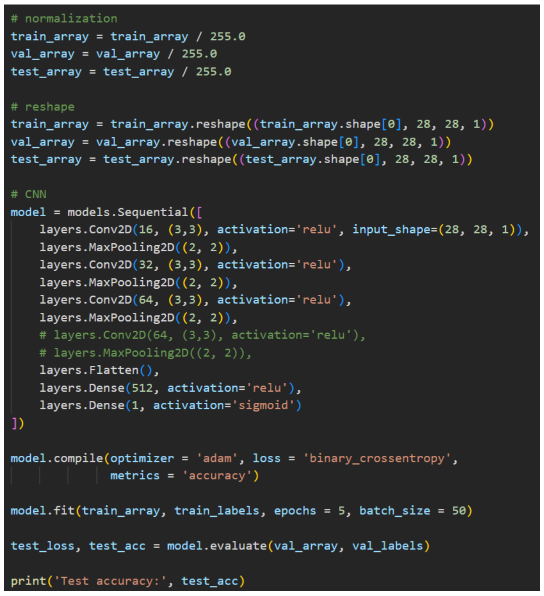


Fig. 13. Thresholds Evaluation



```
# normalization
train_array = train_array / 255.0
val_array = val_array / 255.0
test_array = test_array / 255.0

# reshape
train_array = train_array.reshape((train_array.shape[0], 28, 28, 1))
val_array = val_array.reshape((val_array.shape[0], 28, 28, 1))
test_array = test_array.reshape((test_array.shape[0], 28, 28, 1))

# CNN
model = models.Sequential([
    layers.Conv2D(16, (3,3), activation='relu', input_shape=(28, 28, 1)),
    layers.MaxPooling2D((2, 2)),
    layers.Conv2D(32, (3,3), activation='relu'),
    layers.MaxPooling2D((2, 2)),
    layers.Conv2D(64, (3,3), activation='relu'),
    layers.MaxPooling2D((2, 2)),
    # layers.Conv2D(64, (3,3), activation='relu'),
    # layers.MaxPooling2D((2, 2)),
    layers.Flatten(),
    layers.Dense(512, activation='relu'),
    layers.Dense(1, activation='sigmoid')
])

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

model.fit(train_array, train_labels, epochs = 5, batch_size = 50)

test_loss, test_acc = model.evaluate(val_array, val_labels)

print('Test accuracy:', test_acc)
```

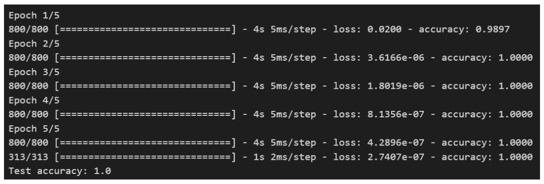
Fig. 14. CNN

VII. Test Set

We use both logistic regression and CNN to predict the test set and make several prediction files: test_PERCP.txt test_CNN.txt, test_LR.txt, and test_SIMPLE.txt, with the similar format as the train.txt and validation.txt.

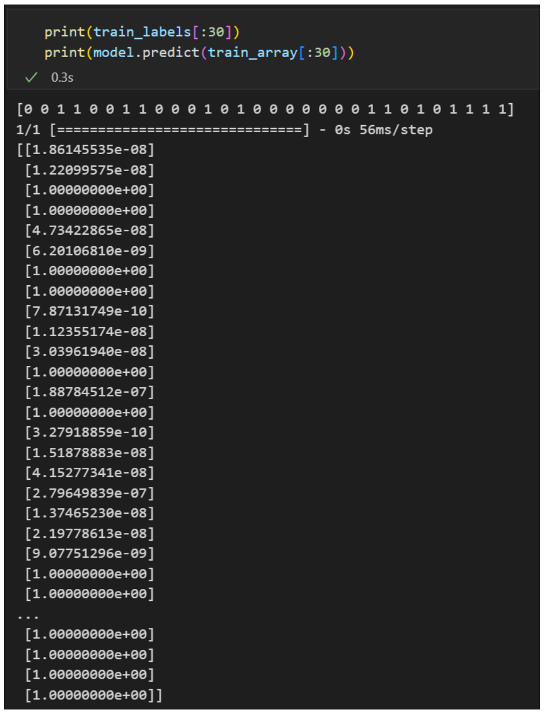
Acknowledgment

Thanks Professor Do and Qian Jiang for the chance to perform this task. To be honest, I'm not sure whether my logistic



```
Epoch 1/5
800/800 [==============================] - 4s 5ms/step - loss: 0.0200 - accuracy: 0.9897
Epoch 2/5
800/800 [==============================] - 4s 5ms/step - loss: 3.6166e-06 - accuracy: 1.0000
Epoch 3/5
800/800 [==============================] - 4s 5ms/step - loss: 1.8019e-06 - accuracy: 1.0000
Epoch 4/5
800/800 [==============================] - 4s 5ms/step - loss: 8.1356e-07 - accuracy: 1.0000
Epoch 5/5
800/800 [==============================] - 4s 5ms/step - loss: 4.2896e-07 - accuracy: 1.0000
313/313 [==============================] - 1s 2ms/step - loss: 2.7407e-07 - accuracy: 1.0000
Test accuracy: 1.0
```

Fig. 15. Output



```
print(train_labels[:30])
print(model.predict(train_array[:30]))
```

```
[0 0 1 1 0 0 1 1 0 0 0 1 0 1 0 0 0 0 0 0 0 1 1 0 1 0 1 1 1 1]
1/1 [==============================] - 0s 56ms/step
[[1.86145535e-08]
 [1.22099575e-08]
 [1.00000000e+00]
 [1.00000000e+00]
 [4.73422865e-08]
 [6.20106810e-09]
 [1.00000000e+00]
 [1.00000000e+00]
 [7.87131749e-10]
 [1.12355174e-08]
 [3.03961940e-08]
 [1.00000000e+00]
 [1.88784512e-07]
 [1.00000000e+00]
 [3.27918859e-10]
 [1.51878883e-08]
 [4.15277341e-08]
 [2.79649839e-07]
 [1.37465230e-08]
 [2.19778613e-08]
 [9.07751296e-09]
 [1.00000000e+00]
 [1.00000000e+00]
...
 [1.00000000e+00]
 [1.00000000e+00]
 [1.00000000e+00]
 [1.00000000e+00]]
```

Fig. 16. mannually check

regression is correctly implemented because the accuracy is so low. But this is a good chance to dig deeply into the concept, as previouly I will directly import libriaries to do the job. I have learned a lot from this implementation.

References

[1] Verma, M. (2022, May 8). Binary classification using Convolution Neural Network (CNN) model. Medium. https://medium.com/@mayankverma05032001/binary-classification-using-convolution-neural-network-cnn-model-6e35cdf5bdbb