modelnotebook

April 18, 2024

1 DEDS_Portfolio_Week-10

Van Pjotr en Sennen

De opdracht van deze week is:

1.1 Opdracht:

Bouw een neuraal netwerk met generatieve AI

1.2 Doel:

Het doel van deze opdracht is om een basisbegrip van neurale netwerken te ontwikkelen door een eenvoudig neuraal netwerk te implementeren, zonder gebruik te maken van backpropagation en gradient descent. Er dient voor het trainen van het model een simpel algoritme gebruikt te worden. Je moet de code zelf goed kan uitleggen.

1.2.1 Requirements:

1. NN heeft 4 input nodes, 1 hidden layer met een door de student gekozen aantal nodes en 1 output node.
2. Gebruik 1 tot 5 input datapunten met bijbehorende output (antwoorden)
3. Maak gebruik van arrays
4. Het mag geen backpropagation gebruiken en ook niet Gradient Descent algoritme.

1.2.2 Stappen:

De volgende algemene stappen zou je terug moeten kunnen vinden of herkennen in de gegenereerde code van je NN. Dit is een hulpmiddel voor je om de code te begrijpen. Het is niet erg als jou gegenereerde code hier iets van afwijkt. 1. Definieer de structuur van het neurale netwerk, inclusief het aantal input nodes, het aantal nodes in de hidden layer en het aantal output nodes. 2. Initialiseer de gewichten van het netwerk willekeurig. 3. Implementeer de feedforward-methode om de input door het netwerk te sturen en de output te berekenen. 4. Bereken de error of fout tussen de voorspelde output en de werkelijke output. 5. Pas de gewichten niet aan met behulp van backpropagation en gradient descent. In plaats daarvan kunnen de studenten ervoor kiezen om de gewichten met een eenvoudige regel aan te passen. 6. Train het netwerk met behulp van de gegeven training samples en evalueer de prestaties ervan.

1.2.3 Training:

Het kan zijn dat de LLM alsnog, direct of indirect, de backpropagation en gradient descent trainingsalgoritme gebruikt. Andere termen die hier direct te maken mee hebben zijn de 'afgeleide' (in het engels de derivative). Dat zie je als je bijvoorbeeld het volgende ziet in de code die de LLM genereerd: - inputToHiddenWeights[i, j] += error * input[i] * hiddenOutput[j] * (1 - hiddenOutput[j]); Metname als het "… (1 - …)" gedeelte. Je wil het liefst code zien dat er als volgt uit ziet: - inputToHiddenWeights[i, j] += error * input[i] * hiddenOutput[j];

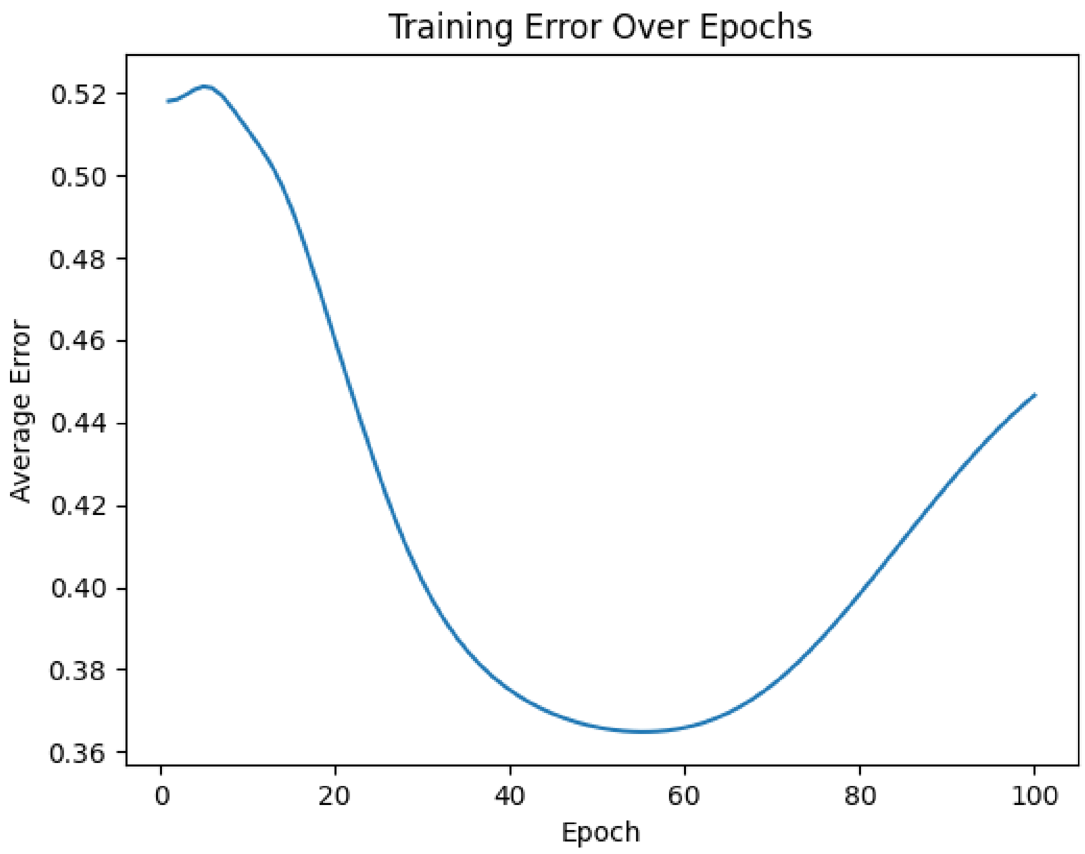
1.3 Maak het model from scratch:

Dit is een model die we hebben gemaakt met behulp van ChatGPT. Het model is een simpel neuraal netwerk met 4 input nodes, 1 hidden layer met 3 nodes en 1 output node. Het model maakt gebruik van 1 input datapunt met bijbehorende output. Het model maakt gebruik van arrays en maakt geen gebruik van backpropagation en gradient descent algoritme.

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

     class NeuralNetwork:
         def __init__(self, input_size, hidden_size, output_size):
             self.input_size = input_size
             self.hidden_size = hidden_size
             self.output_size = output_size

             # Initialize weights randomly
             self.input_to_hidden_weights = np.random.randn(input_size, hidden_size)
             self.hidden_to_output_weights = np.random.randn(hidden_size,␣
      ↪output_size)

         def sigmoid(self, x):
             return 1 / (1 + np.exp(-x))

         def sigmoid_derivative(self, x):
             return x * (1 - x)

         def feedforward(self, inputs):
             # Input to hidden layer
             hidden_sum = np.dot(inputs, self.input_to_hidden_weights)
             hidden_output = self.sigmoid(hidden_sum)

             # Hidden to output layer
             output_sum = np.dot(hidden_output, self.hidden_to_output_weights)
             output = self.sigmoid(output_sum)

             return output

         def train(self, inputs, targets, epochs):
```

```
        errors = []
        for epoch in range(epochs):
            epoch_error = 0
            for i in range(len(inputs)):
                # Feedforward
                input_data = inputs[i]
                target = targets[i]

                hidden_sum = np.dot(input_data, self.input_to_hidden_weights)
                hidden_output = self.sigmoid(hidden_sum)

                output_sum = np.dot(hidden_output, self.
 ↪hidden_to_output_weights)
                output = self.sigmoid(output_sum)

                # Calculate error
                error = target - output
                epoch_error += np.mean(np.abs(error))

                # Update weights (no backpropagation)
                self.input_to_hidden_weights += np.outer(input_data,␣
 ↪hidden_output) * error
                self.hidden_to_output_weights += np.outer(hidden_output,␣
 ↪output) * error

            # Append average error for this epoch
            errors.append(epoch_error / len(inputs))

            # Print average error for this epoch
            print(f'Epoch {epoch + 1}, Average Error: {errors[-1]}')

        # Plot the training error over epochs
        plt.plot(range(1, epochs + 1), errors)
        plt.xlabel('Epoch')
        plt.ylabel('Average Error')
        plt.title('Training Error Over Epochs')
        plt.show()

# Example usage
inputs = np.array([[0, 0, 1, 1],
                   [0, 1, 1, 0],
                   [1, 0, 1, 1],
                   [1, 1, 1, 0]])
targets = np.array([[0], [1], [1], [0]])

nn = NeuralNetwork(input_size=4, hidden_size=3, output_size=1)
nn.train(inputs, targets, epochs=100)
```

```
Epoch 1, Average Error: 0.5179027522388071
Epoch 2, Average Error: 0.5183124178591839
Epoch 3, Average Error: 0.5194785174012861
Epoch 4, Average Error: 0.5207404699417699
Epoch 5, Average Error: 0.521447017421613
Epoch 6, Average Error: 0.5210393773441602
Epoch 7, Average Error: 0.519391618095874
Epoch 8, Average Error: 0.5168712847990039
Epoch 9, Average Error: 0.5139844797629337
Epoch 10, Average Error: 0.5110463849368972
Epoch 11, Average Error: 0.5080970809516165
Epoch 12, Average Error: 0.5049669747542785
Epoch 13, Average Error: 0.501390125445528
Epoch 14, Average Error: 0.497132531230478
Epoch 15, Average Error: 0.49209955440534636
Epoch 16, Average Error: 0.4863657213786727
Epoch 17, Average Error: 0.48011497528572583
Epoch 18, Average Error: 0.47354985782732495
Epoch 19, Average Error: 0.4668316459384649
Epoch 20, Average Error: 0.46006626155525576
Epoch 21, Average Error: 0.4533185453276015
Epoch 22, Average Error: 0.44663425888431724
Epoch 23, Average Error: 0.4400575183714592
Epoch 24, Average Error: 0.4336393389524458
Epoch 25, Average Error: 0.42743781797833136
Epoch 26, Average Error: 0.42151276843825414
Epoch 27, Average Error: 0.41591822952066937
Epoch 28, Average Error: 0.4106958191402471
Epoch 29, Average Error: 0.40587078615818095
Epoch 30, Average Error: 0.40145131134733936
Epoch 31, Average Error: 0.39743052562631875
Epoch 32, Average Error: 0.3937901348612472
Epoch 33, Average Error: 0.39050448949265565
Epoch 34, Average Error: 0.38754424142174343
Epoch 35, Average Error: 0.38487915180312876
Epoch 36, Average Error: 0.3824799699386793
Epoch 37, Average Error: 0.3803195180157281
Epoch 38, Average Error: 0.3783731965977943
Epoch 39, Average Error: 0.37661911838526707
Epoch 40, Average Error: 0.3750380313211569
Epoch 41, Average Error: 0.37361313945039726
Epoch 42, Average Error: 0.3723298866909142
Epoch 43, Average Error: 0.37117573874857385
Epoch 44, Average Error: 0.37013998002096105
Epoch 45, Average Error: 0.3692135320674024
Epoch 46, Average Error: 0.3683887949717334
Epoch 47, Average Error: 0.3676595104857429
Epoch 48, Average Error: 0.36702064485251373
Epoch 49, Average Error: 0.36646828890341765
Epoch 50, Average Error: 0.36599957299640823
Epoch 51, Average Error: 0.36561259441240906
Epoch 52, Average Error: 0.36530635484989304
Epoch 53, Average Error: 0.365080705605888
Epoch 54, Average Error: 0.36493629788310267
Epoch 55, Average Error: 0.36487453541468173
Epoch 56, Average Error: 0.36489752626443117
Epoch 57, Average Error: 0.36500803027504947
Epoch 58, Average Error: 0.36520939825785914
Epoch 59, Average Error: 0.3655054987307896
Epoch 60, Average Error: 0.3659006279325872
Epoch 61, Average Error: 0.36639939911138475
Epoch 62, Average Error: 0.367006607857316
Epoch 63, Average Error: 0.36772707165970997
Epoch 64, Average Error: 0.3685654440034066
Epoch 65, Average Error: 0.3695260061571903
Epoch 66, Average Error: 0.37061244318111425
Epoch 67, Average Error: 0.3718276142392405
Epoch 68, Average Error: 0.3731733305284453
Epoch 69, Average Error: 0.3746501563908466
Epoch 70, Average Error: 0.37625724984458625
Epoch 71, Average Error: 0.3779922573887016
Epoch 72, Average Error: 0.3798512743760309
Epoch 73, Average Error: 0.38182887678455907
Epoch 74, Average Error: 0.38391822354784333
Epoch 75, Average Error: 0.3861112217200616
Epoch 76, Average Error: 0.38839874072691416
Epoch 77, Average Error: 0.39077085771079056
Epoch 78, Average Error: 0.3932171140840132
Epoch 79, Average Error: 0.3957267639711487
Epoch 80, Average Error: 0.3982889979316956
Epoch 81, Average Error: 0.40089312957860146
Epoch 82, Average Error: 0.40352873767994807
Epoch 83, Average Error: 0.40618576131890116
Epoch 84, Average Error: 0.40885455011824284
Epoch 85, Average Error: 0.4115258750568701
Epoch 86, Average Error: 0.41419090787873086
Epoch 87, Average Error: 0.4168411785505026
Epoch 88, Average Error: 0.41946852079513475
Epoch 89, Average Error: 0.4220650155821207
Epoch 90, Average Error: 0.4246229417454898
Epoch 91, Average Error: 0.42713474173809307
Epoch 92, Average Error: 0.4295930089794838
Epoch 93, Average Error: 0.43199050134521455
Epoch 94, Average Error: 0.43432018310310005
Epoch 95, Average Error: 0.43657529508062204
Epoch 96, Average Error: 0.438749450159338
Epoch 97, Average Error: 0.4408367485249721
Epoch 98, Average Error: 0.44283190471588707
Epoch 99, Average Error: 0.44473037671028337
Epoch 100, Average Error: 0.44652848636495174
```



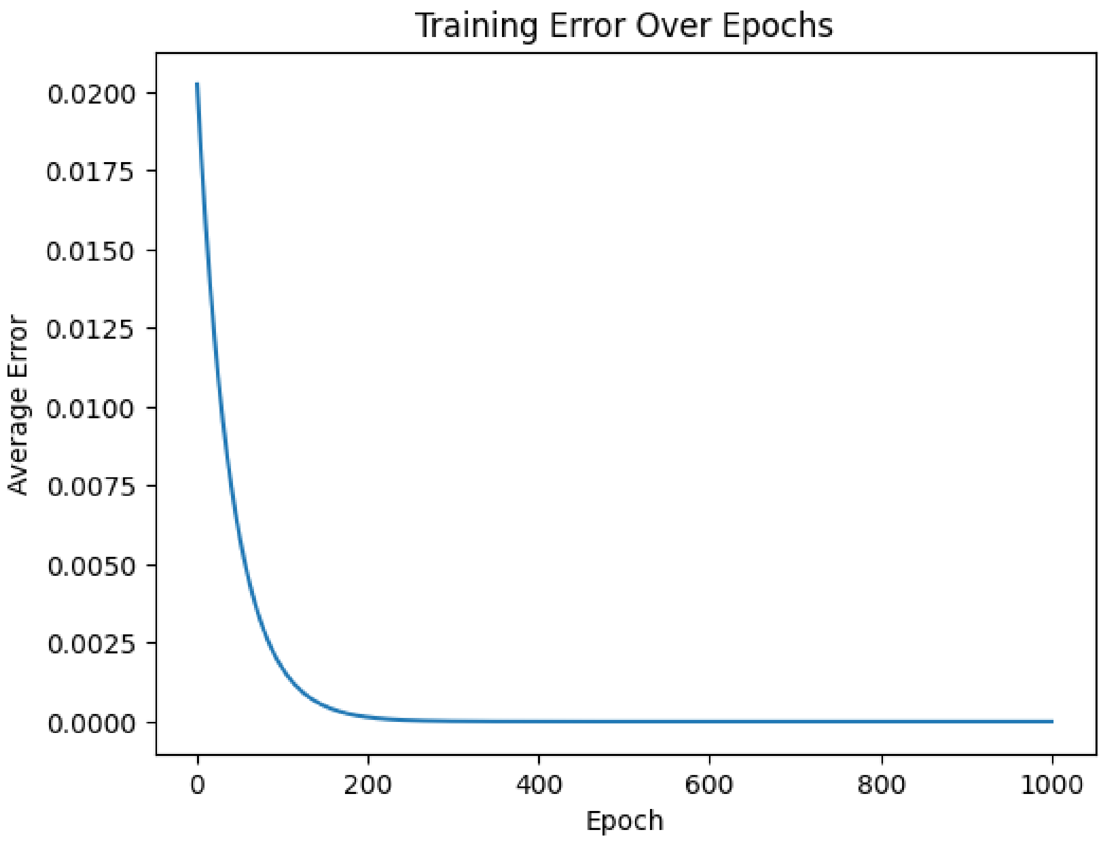
1.4 Gebaseerd op het C# Script:

In het begin van deze opdracht moesten we ook een c# script maken. Waar we ook ons eigen neural network moesten bouwen. Hieronder is een python script gebasseerd op het c# script.

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

     class NeuralNetwork:
         def __init__(self, input_size, hidden_layer_size, learning_rate):
             self.input_size = input_size
             self.hidden_layer_size = hidden_layer_size
             self.learning_rate = learning_rate
```

```
        self.weights_input_to_hidden = np.random.uniform(-1, 1, (input_size,␣
↪hidden_layer_size))
        self.weights_hidden_to_output = np.random.uniform(-1, 1,␣
↪hidden_layer_size)

    def sigmoid(self, x):
        return 1 / (1 + np.exp(-x))

    def feed_forward(self, inputs):
        hidden_layer_output = self.sigmoid(np.dot(inputs, self.
↪weights_input_to_hidden))
        output = self.sigmoid(np.dot(hidden_layer_output, self.
↪weights_hidden_to_output))
        return output

    def train(self, inputs, target, epochs):
        errors = []
        for _ in range(epochs):
            hidden_layer_output = self.sigmoid(np.dot(inputs, self.
↪weights_input_to_hidden))
            output = self.sigmoid(np.dot(hidden_layer_output, self.
↪weights_hidden_to_output))

            error = target - output

            self.weights_hidden_to_output += error * self.learning_rate *␣
↪hidden_layer_output
            self.weights_input_to_hidden += np.outer(inputs, error * self.
↪learning_rate * self.weights_hidden_to_output * hidden_layer_output * (1 -␣
↪hidden_layer_output))

            errors.append(np.mean(np.abs(error)))

        # Plot the training error over epochs
        plt.plot(range(1, epochs + 1), errors)
        plt.xlabel('Epoch')
        plt.ylabel('Average Error')
        plt.title('Training Error Over Epochs')
        plt.show()

# Example usage
np.random.seed(0)  # for reproducibility

neural_network = NeuralNetwork(4, 3, 0.1)

inputs = np.array([0.1, 0.2, 0.3, 0.4])
```

```
target = 0.5

neural_network.train(inputs, target, epochs=1000)

test_inputs = np.array([0.5, 0.6, 0.7, 0.8])
output = neural_network.feed_forward(test_inputs)

print("Output:", output)
```



```
Output: 0.4971570953933695
```

1.5 AO die nummers kan voorspellen.

We wilden een Neural Netwerk maken die makkelijk geschreven nummers kan begrijpen en voorspellen welke nummer opgeschreven is. We maken hierbij een model gemaakt die gebruikt maakt van scipy minimize functie, die ervoor zorgt dat

[ ]:
```
import numpy as np
from scipy.optimize import minimize
from scipy.io import loadmat
from sklearn.metrics import accuracy_score, precision_score
```

```
def predict(Model, Dummies, X):
        m = X.shape[0]
        one_matrix = np.ones((m, 1))
        X = np.append(one_matrix, X, axis=1) # Adding bias unit to first layer
        z2 = np.dot(X, Model.transpose())
        a2 = 1 / (1 + np.exp(-z2)) # Activation for second layer
        one_matrix = np.ones((m, 1))
        a2 = np.append(one_matrix, a2, axis=1) # Adding bias unit to hidden␣
 ↪layer
        z3 = np.dot(a2, Dummies.transpose())
        a3 = 1 / (1 + np.exp(-z3)) # Activation for third layer
        p = (np.argmax(a3, axis=1)) # Predicting the class on the basis of max␣
 ↪value of hypothesis
        return p


def initialise(a, b):
    epsilon = 0.15
    c = np.random.rand(a, b + 1) * (
      # Randomly initialises values of data between [-epsilon, +epsilon]
      2 * epsilon) - epsilon
    return c


def neural_network(nn_params, input_layer_size, hidden_layer_size, num_labels,␣
 ↪X, y, lamb):
    # Weights are split back to Model, Dummies
    Model = np.reshape(nn_params[:hidden_layer_size * (input_layer_size + 1)],
                        (hidden_layer_size, input_layer_size + 1))
    Dummies = np.reshape(nn_params[hidden_layer_size * (input_layer_size + 1):],
                        (num_labels, hidden_layer_size + 1))

    # Forward propagation
    m = X.shape[0]
    one_matrix = np.ones((m, 1))
    X = np.append(one_matrix, X, axis=1)  # Adding bias unit to first layer
    a1 = X
    z2 = np.dot(X, Model.transpose())
    a2 = 1 / (1 + np.exp(-z2))  # Activation for second layer
    one_matrix = np.ones((m, 1))
    a2 = np.append(one_matrix, a2, axis=1)  # Adding bias unit to hidden layer
    z3 = np.dot(a2, Dummies.transpose())
    a3 = 1 / (1 + np.exp(-z3))  # Activation for third layer

    # Changing the y labels into vectors of boolean values.
    # For each label between 0 and 9, there will be a vector of length 10
    # where the ith element will be 1 if the label equals i
    y_vect = np.zeros((m, 10))
    for i in range(m):
        y_vect[i, int(y[i])] = 1

    # Calculating cost function
    J = (1 / m) * (np.sum(np.sum(-y_vect * np.log(a3) - (1 - y_vect) * np.log(1␣
 ↪- a3)))) + (lamb / (2 * m)) * (
                sum(sum(pow(Model[:, 1:], 2))) + sum(sum(pow(Dummies[:, 1:],␣
 ↪2))))

    # backprop
    Delta3 = a3 - y_vect
    Delta2 = np.dot(Delta3, Dummies) * a2 * (1 - a2)
    Delta2 = Delta2[:, 1:]

    # gradient
    Model[:, 0] = 0
    Model_grad = (1 / m) * np.dot(Delta2.transpose(), a1) + (lamb / m) * Model
    Dummies[:, 0] = 0
    Dummies_grad = (1 / m) * np.dot(Delta3.transpose(), a2) + (lamb / m) *␣
 ↪Dummies
    grad = np.concatenate((Model_grad.flatten(), Dummies_grad.flatten()))

    return J, grad


# Loading mat file
data = loadmat('mnist-original.mat')

# Extracting features from mat file
X = data['data']
X = X.transpose()

# Normalizing the data
X = X / 255

# Extracting labels from mat file
y = data['label']
y = y.flatten()

# Splitting data into training set with 60,000 examples
X_train = X[:60000, :]
y_train = y[:60000]

# Splitting data into testing set with 10,000 examples
```

```
X_test = X[60000:, :]
y_test = y[60000:]

m = X.shape[0]
input_layer_size = 784  # Images are of (28 X 28) px so there will be 784␣
 ↪features
hidden_layer_size = 100
num_labels = 10  # There are 10 classes [0, 9]

# Randomly initialising The Model itself and the Dummy variables
initial_Model = initialise(hidden_layer_size, input_layer_size)
initial_Dummies = initialise(num_labels, hidden_layer_size)

# Unrolling parameters into a single column vector
initial_nn_params = np.concatenate((initial_Model.flatten(), initial_Dummies.
 ↪flatten()))
maxiter = 100
lambda_reg = 0.1  # To avoid overfitting
myargs = (input_layer_size, hidden_layer_size, num_labels, X_train, y_train,␣
 ↪lambda_reg)

# Calling minimize function to minimize cost function and to train weights
results = minimize(neural_network, x0=initial_nn_params, args=myargs,
          options={'disp': True, 'maxiter': maxiter}, method="L-BFGS-B",␣
 ↪jac=True)

nn_params = results["x"]  # Trained Data is extracted

# Weights are split back to Model, Dummies
Model = np.reshape(nn_params[:hidden_layer_size * (input_layer_size + 1)], (
                              hidden_layer_size, input_layer_size + 1))  #␣
 ↪shape = (100, 785)
Dummies = np.reshape(nn_params[hidden_layer_size * (input_layer_size + 1):],
                      (num_labels, hidden_layer_size + 1))  # shape = (10, 101)

# Checking test set accuracy of our model
pred = predict(Model, Dummies, X_test)
print('Test Set Accuracy: {:f}'.format((np.mean(pred == y_test) * 100)))

# Checking train set accuracy of our model
pred = predict(Model, Dummies, X_train)
print('Training Set Accuracy: {:f}'.format((np.mean(pred == y_train) * 100)))

# Evaluating precision of our model
true_positive = 0
for i in range(len(pred)):
    if pred[i] == y_train[i]:
        true_positive += 1
false_positive = len(y_train) - true_positive
print('Precision =', true_positive/(true_positive + false_positive))

# Saving the data in .txt file
np.savetxt('Model.txt', Model, delimiter=' ')
np.savetxt('Dummies.txt', Dummies, delimiter=' ')
```

```
Test Set Accuracy: 97.530000
Training Set Accuracy: 99.503333
Precision = 0.9950333333333333
```

[ ]:
```
from tkinter import *
import numpy as np
from PIL import ImageGrab

window = Tk()
window.title("Handwritten digit recognition")
l1 = Label()

def predict(Model, Dummies, X):
        m = X.shape[0]
        one_matrix = np.ones((m, 1))
        X = np.append(one_matrix, X, axis=1) # Adding bias unit to first layer
        z2 = np.dot(X, Model.transpose())
        a2 = 1 / (1 + np.exp(-z2)) # Activation for second layer
        one_matrix = np.ones((m, 1))
        a2 = np.append(one_matrix, a2, axis=1) # Adding bias unit to hidden␣
 ↪layer
        z3 = np.dot(a2, Dummies.transpose())
        a3 = 1 / (1 + np.exp(-z3)) # Activation for third layer
        p = (np.argmax(a3, axis=1)) # Predicting the class on the basis of max␣
 ↪value of hypothesis
        return p

def Prediction():
    global l1

    widget = cv
    # Setting co-ordinates of canvas
    x = window.winfo_rootx() + widget.winfo_x()
    y = window.winfo_rooty() + widget.winfo_y()
    x1 = x + widget.winfo_width()
    y1 = y + widget.winfo_height()

    # Image is captured from canvas and is resized to (28 X 28) px
    img = ImageGrab.grab().crop((x, y, x1, y1)).resize((28, 28))
    # Converting rgb to grayscale image
    img = img.convert('L')

    # Extracting pixel matrix of image and converting it to a vector of (1, 784)
    x = np.asarray(img)
    vec = np.zeros((1, 784))
    k = 0
    for i in range(28):
        for j in range(28):
            vec[0][k] = x[i][j]
            k += 1

    # Loading the Text.
    Model = np.loadtxt('Model.txt')
    Dummies = np.loadtxt('Dummies.txt')

    # Calling function for prediction
    pred = predict(Model, Dummies, vec / 255)

    # Displaying the result
    l1 = Label(window, text="Digit = " + str(pred[0]), font=('Calibri', 20))
    l1.place(x=260, y=420)


lastx, lasty = None, None


# Clears the canvas
def clear_widget():
    global cv, l1
    cv.delete("all")
    l1.destroy()


# Activate canvas
def event_activation(event):
    global lastx, lasty
    cv.bind('<B1-Motion>', draw_lines)
    lastx, lasty = event.x, event.y


# To draw on canvas
def draw_lines(event):
    global lastx, lasty
    x, y = event.x, event.y
    cv.create_line((lastx, lasty, x, y), width=20, fill='white',␣
 ↪capstyle=ROUND, smooth=TRUE, splinesteps=12)
    lastx, lasty = x, y


# Label
L1 = Label(window, text="Handwritten Digit Recoginition", font=('Calibri', 25),␣
 ↪fg="blue")
L1.place(x=100, y=10)

# Button to clear canvas
b1 = Button(window, text="1. Clear Canvas", font=('Calibri', 15), bg="orange",␣
 ↪fg="black", command=clear_widget)
b1.place(x=120, y=370)

# Button to predict digit drawn on canvas
b2 = Button(window, text="2. Prediction", font=('Calibri', 15), bg="white",␣
 ↪fg="red", command=Prediction)
b2.place(x=355, y=370)

# Setting properties of canvas
cv = Canvas(window, width=350, height=290, bg='black')
cv.place(x=120, y=70)

cv.bind('<Button-1>', event_activation)
window.geometry("600x500")
window.mainloop()
```