Project Report

Handwritten Digit Recognition with Neural Network

Course Code: CSE366

Course Title: Artificial Intelligence

Section No: 01

Semester: Summer 22

Submitted By

Israfil Arman (ID: 2020-1-60-086)

Md. Habibuzzaman (2020-1-60-212)

Tanjilul Haq (ID: 2020-1-60-214)

Submitted To

Jesan Ahammed Ovi

Senior Lecturer

Department of CSE

East West University

Date of Submission: 13-09-2022

Problem Statement

Handwritten digit recognition is the ability of a computer to recognize the human handwritten digits from different sources like images, papers, touch screens, etc., and classify them into 10 predefined classes (0-9). It is a hard task for the machine because handwritten digits are not perfect and can be made with many different shapes and sizes. In the project we built a Neural Network model and trained and tested the model with the MNIST database of handwritten digits. Moreover, we have also tested the model with images outside the database and found satisfactory results.

System Requirements

1. Processor: Intel(R) Core (TM) i3-1005G1 CPU @ 1.20GHz   1.20 GHz
2. RAM: 4GB
3. Operating System: Windows
4. IDE: Google Colaboratory

System Design

In the project we used **Adam** (*adaptive moment estimation)* optimization algorithm.

The Adam optimization algorithm is an extension to stochastic gradient descent that has recently seen broader adoption for deep learning applications in computer vision and natural language processing. Adam was presented by **DiederikKingma** from **OpenAI** and **Jimmy Ba** from the University of Toronto in their 2015 ICLR paper (poster) titled "Adam: A Method for Stochastic Optimization". It can be used instead of the classical stochastic gradient descent procedure to update network weights iteratively based on training data.

Adam is different to classical stochastic gradient descent. Stochastic gradient descent maintains a single learning rate (termed alpha) for all weight updates and the learning rate does not change during training. But Adam's learning rate is maintained for each network weight (parameter) and separately adapted as learning unfolds.

The authors describe Adam as combining the advantages of two other extensions of stochastic gradient descent. Specifically:

- **Adaptive Gradient Algorithm** (AdaGrad) that maintains a per-parameter learning rate that improves performance on problems with sparse gradients (e.g. natural language and computer vision problems).
- **Root Mean Square Propagation** (RMSProp) that also maintains per-parameter learning rates that are adapted based on the average of recent magnitudes of the gradients for the weight (e.g. how quickly it is changing). This means the algorithm does well on online and non-stationary problems (e.g. noisy).

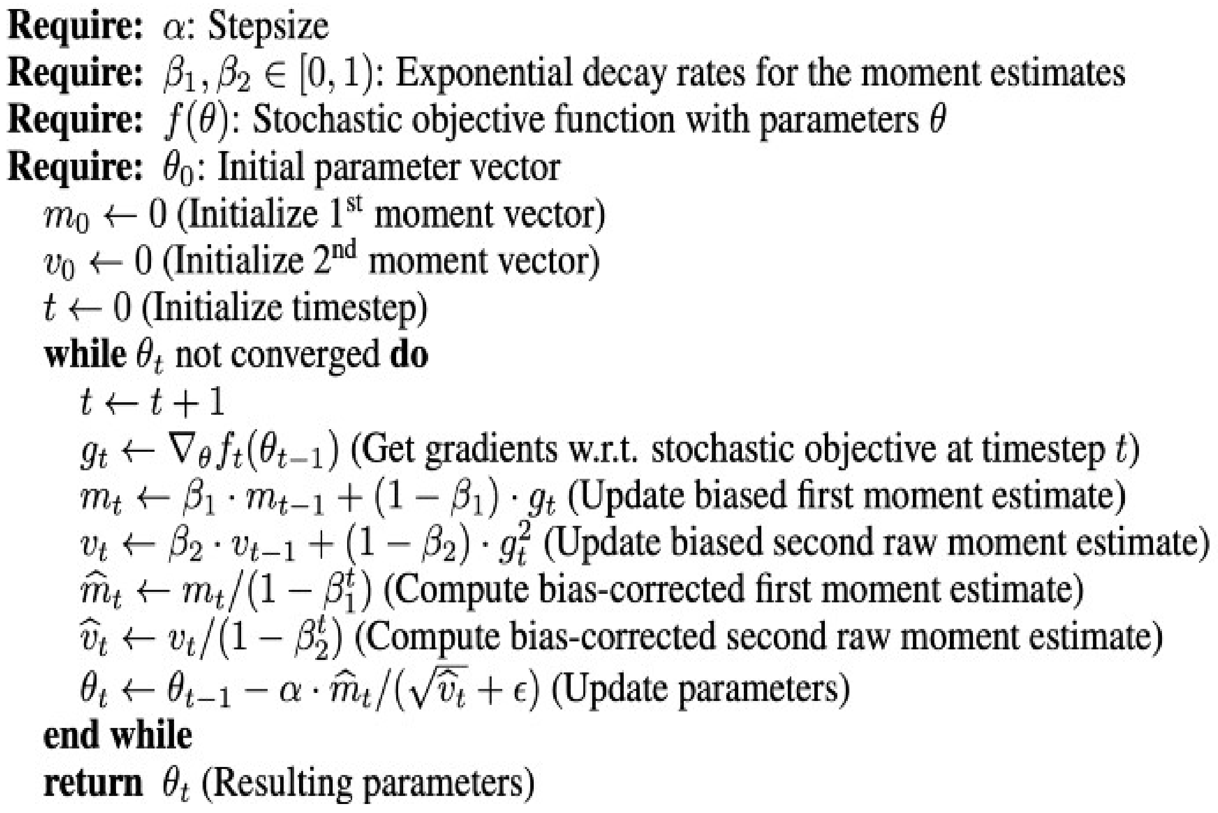
Adam realizes the benefits of both AdaGrad and RMSProp. Instead of adapting the parameter learning rates based on the average first moment (the mean) as in RMSProp, Adam also makes use of the average of the second moments of the gradients (the uncentered variance). Specifically, the algorithm calculates an exponential moving average of the gradient and the squared gradient, and the parameters beta1 and beta2 control the decay rates of these moving averages. The initial value of the moving averages and beta1 and beta2 values close to 1.0 (recommended) result in a bias of moment estimates towards zero. This bias is overcome by first calculating the biased estimates before then calculating bias-corrected estimates.

Adam configuration parameters:

- **alpha**. Also referred to as the learning rate or step size. The proportion that weights are updated (e.g. 0.001). Larger values (e.g. 0.3) results in faster initial learning before the rate is updated. Smaller values (e.g. 1.0E-5) slow learning right down during training
- **beta1**. The exponential decay rate for the first moment estimates (e.g. 0.9).

- **beta2**. The exponential decay rate for the second-moment estimates (e.g. 0.999). This value should be set close to 1.0 on problems with a sparse gradient (e.g. NLP and computer vision problems).
- **epsilon**. Is a very small number to prevent any division by zero in the implementation (e.g. 10E-7).

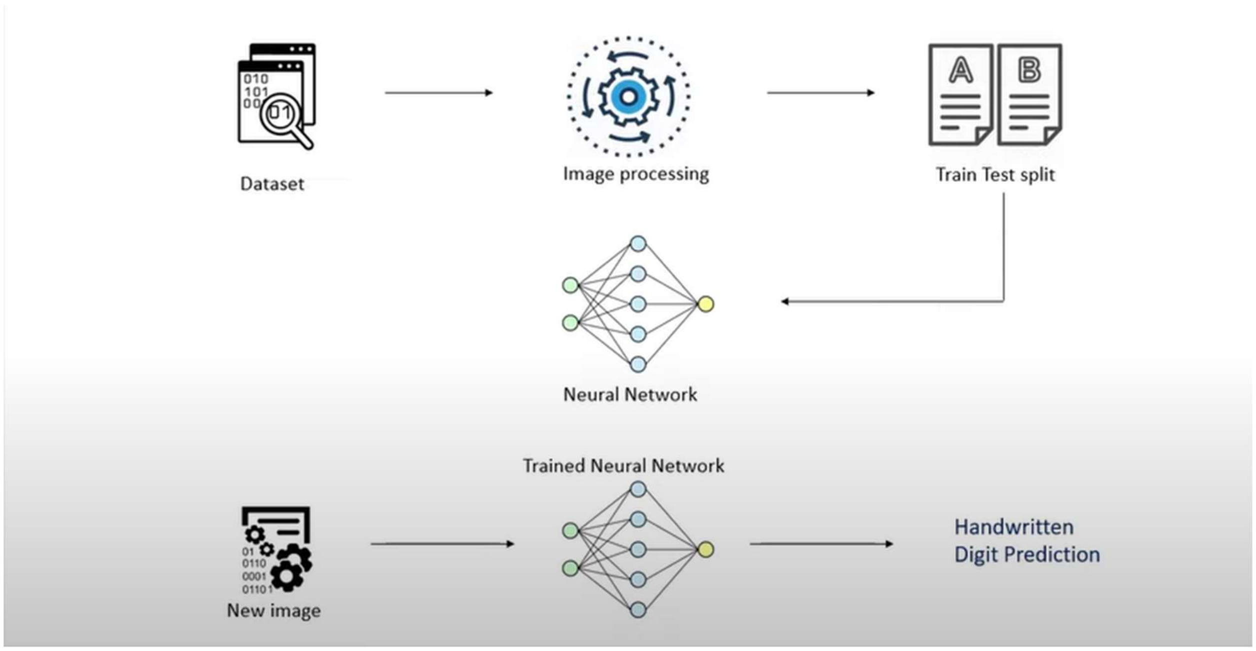
Algorithm:



```
Require: $\alpha$: Stepsize
Require: $\beta_1, \beta_2 \in [0, 1)$: Exponential decay rates for the moment estimates
Require: $f(\theta)$: Stochastic objective function with parameters $\theta$
Require: $\theta_0$: Initial parameter vector
  $m_0 \leftarrow 0$ (Initialize 1st moment vector)
  $v_0 \leftarrow 0$ (Initialize 2nd moment vector)
  $t \leftarrow 0$ (Initialize timestep)
  while $\theta_t$ not converged do
    $t \leftarrow t + 1$
    $g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$ (Get gradients w.r.t. stochastic objective at timestep $t$)
    $m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$ (Update biased first moment estimate)
    $v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$ (Update biased second raw moment estimate)
    $\widehat{m}_t \leftarrow m_t / (1 - \beta_1^t)$ (Compute bias-corrected first moment estimate)
    $\widehat{v}_t \leftarrow v_t / (1 - \beta_2^t)$ (Compute bias-corrected second raw moment estimate)
    $\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \widehat{m}_t / (\sqrt{\widehat{v}_t} + \epsilon)$ (Update parameters)
  end while
  return $\theta_t$ (Resulting parameters)
```

Architecture of the project:

First, we imported a dataset from Keras MNIST dataset for training our algorithm. Using some library, we processed the images from the dataset. Then we trained and tested the neural network. Finally, we inserted new handwritten digit image in a trained neural network and got the predicted handwritten digit as output.



Implementation

Importing the necessary libraries

```
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import cv2
from google.colab.patches import cv2_imshow
from PIL import Image
import tensorflow as tf
```

```
tf.random.set_seed(3)
from tensorflow import keras
from keras.datasets import mnist
from tensorflow.math import confusion_matrix
```

Loading the MNIST data from keras.datasets

```
(X_train, Y_train), (X_test, Y_test) =  mnist.load_data()
```

Building the Neural Network

```
#setting up the layers of the Neural Network

model = keras.Sequential([
               keras.layers.Flatten(input_shape=(28,28)),#input layer
               keras.layers.Dense(80, activation='relu'),#hidden layer1
               keras.layers.Dense(80, activation='relu'), #hidden layer2
               keras.layers.Dense(10, activation='sigmoid')#output layer
                        ])
```

Here we have used ReLU (Rectified Linear Unit) activation function for our inner/hidden layers. The main advantage of using the ReLU function over other activation functions is that it does not activate all the neurons at the same time.

This means that the neurons will only be deactivated if the output of the linear transformation is less than 0.

Function: f(x) = max (0, x)

Moreover, the Sigmoid activation function for the output layer. It transforms the values between the range 0 and 1 as we have reshaped the trained and tested data between 0 and 1.

Function: f(x) = 1/(1+e^-x)

Compiling the Neural Network

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

Training the Neural Network

```
model.fit(X_train, Y_train, epochs=250)
```

Checking Accuracy on Test data

```
loss, accuracy = model.evaluate(X_test, Y_test)
print(accuracy)
```

Creating a confusion matrix to evaluate the test data

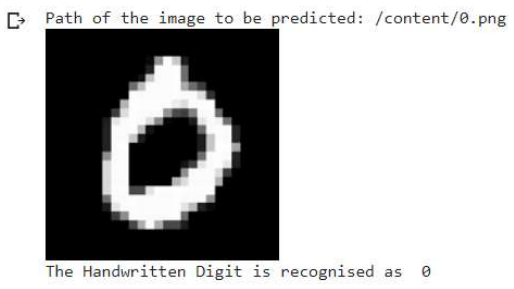
```
conf_mat = confusion_matrix(Y_test, Y_pred_labels)
plt.figure(figsize=(15,7))
sns.heatmap(conf_mat, annot=True, fmt='d', cmap='Reds')
plt.ylabel('True Labels')
plt.xlabel('Predicted Labels')
```

Predictive System

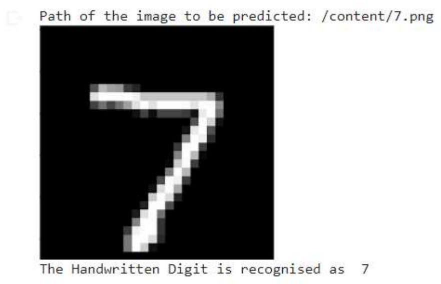
```
input_image_path = input ('Path of the image to be predicted: ')

input_image = cv2.imread(input_image_path)

cv2_imshow(input_image)

input_image = cv2.cvtColor(input_image, cv2.COLOR_RGB2GRAY)

input_image_resize = cv2.resize(input_image, (28, 28))

input_image_resize = input_image_resize/255

image_reshaped = np.reshape(input_image_resize, [1,28,28])

input_prediction = model.predict(image_reshaped)

input_pred_label = np.argmax(input_prediction)

print ("The Handwritten Digit is recognised as ", input_pred_label)
```

Here, after taking the image path to be predicted, cv2 library is used to process for visualizing the image. Using cv2.cvtColor we converted the image color from RGB to grayscale. After resizing and reshaping, the image is sent as an input to the neural network for predicting. After prediction it takes the maximum prediction of that digit is taken and printed as the result.

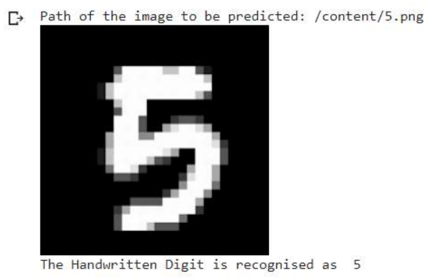
Testing Results:



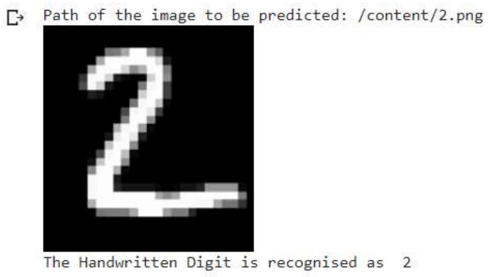
```
Path of the image to be predicted: /content/0.png
The Handwritten Digit is recognised as  0
```



```
Path of the image to be predicted: /content/7.png
The Handwritten Digit is recognised as  7
```



```
Path of the image to be predicted: /content/5.png
The Handwritten Digit is recognised as  5
```



```
Path of the image to be predicted: /content/2.png
The Handwritten Digit is recognised as  2
```

We have uploaded the image file in the project directory and copied and given the path into the program. After execution the trained neural network successfully recognized the image.

Future Scope:

Limitations:

- We must use the same image background for prediction as the training dataset.
- This model cannot recognize RGB images.

Future Scope:

- We will try to detect handwritten words in the future.