Deep Neural Network using Keras

Group No :151

Group Member Names:

Peyala Samarasimha Reddy - 2023AA05072 100% Contribution

Monisha G - 2023AA05536 100% Contribution

Akshay Mohan - 2023AA05315 100% Contribution

Sreelakshmi Ajith - 2023AA05316 100% Contribution

In [2]:
```
import tensorflow as tf

from tensorflow.keras import models
from tensorflow.keras import layers

import random
import numpy as np
```

In [3]:
```
random.seed(42)         # Initialize the random number generator.
np.random.seed(42)      # With the seed reset, the same set of
                        # numbers will appear every time.
tf.random.set_seed(42)  # sets the graph-level random seed
```

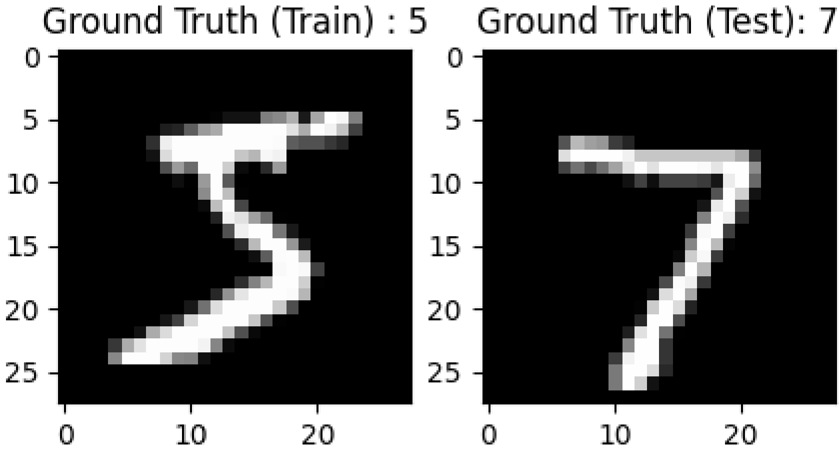
Dataset

In [3]:
```
# Use the MNIST dataset  of Keras.

mnist = tf.keras.datasets.mnist

(Xtrain, Ytrain) , (Xtest, Ytest) = mnist.load_data()
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnis
t.npz
11490434/11490434 ━━━━━━━━━━━━━━━━━━━━ 0s 0us/step
```

In [4]:
```
import matplotlib.pyplot as plt

plt.figure(figsize=[5,5])

# Display the first image in training data
plt.subplot(121)
plt.imshow(Xtrain[0,:,:], cmap='gray')
plt.title("Ground Truth (Train) : {}".format(Ytrain[0]))

# Display the first image in testing data
plt.subplot(122)
plt.imshow(Xtest[0,:,:], cmap='gray')
plt.title("Ground Truth (Test): {}".format(Ytest[0]))
```

Out[4]:
```
Text(0.5, 1.0, 'Ground Truth (Test): 7')
```



In [6]:
```
# size of the datsets

print(Xtrain.shape)
print(Xtest.shape)
print(Ytrain.shape)
print(Ytest.shape)
```

```
(60000, 28, 28)
(10000, 28, 28)
(60000,)
(10000,)
```

In [7]:
```
# print a sample data

print('Xtrain \n', Xtrain[10,10])
print('Xtest \n', Xtest[10,10])
print('Ytrain \n', Ytrain[10,])
print('Ytest \n', Ytest[10,])
```

```
Xtrain
 [  0   0   0   0   0   0   0   0   0   0   0   0   0  24 209 254 254 254
 171   0   0   0   0   0   0   0   0   0]
Xtest
 [  0   0   0   0   0   0   0   0 194 254 103   0   0   0   0   0   0   0
   0   0 150 254 213   0   0   0   0   0]
Ytrain
 3
Ytest
 0
```

In [8]:
```
# Normalize the data
# 60000 input images are in the train set.
# 10000 input images are in the test set.

Xtrain = Xtrain.reshape((60000, 28*28))    # reshape the input set to size 28*28.
Xtrain = Xtrain.astype('float32')/255      # normalize to grayscale; set datatype as

Xtest = Xtest.reshape((10000, 28*28))      # reshape the input set to size 28*28.
Xtest = Xtest.astype('float32')/255        # normalize to grayscale; set datatype as

Ytrain = tf.keras.utils.to_categorical(Ytrain)
Ytest = tf.keras.utils.to_categorical(Ytest)
```

In [9]:
```
# print a sample data

print('Xtrain \n', Xtrain[10,10])
print('Xtest \n', Xtest[10,10])
print('Ytrain \n', Ytrain[10,])
print('Ytest \n', Ytest[10,])
```

```
Xtrain
 0.0
Xtest
 0.0
Ytrain
 [0. 0. 0. 1. 0. 0. 0. 0. 0. 0.]
Ytest
 [1. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
```

DNN Model

Using Keras, create the DNN or Sequential Model

In [10]:
```
# Create a model object

dnnModel = models.Sequential()
```

Add dense layers, specifying the number of units in each layer and the activation function used in the layer.

In [11]:
```
# Layer 1 = input layer
# specify the input size in the first layer.

dnnModel.add(layers.Dense(50, activation='relu', input_shape= (28*28,)))

# Layer 2 = hidden layer
dnnModel.add(layers.Dense(60, activation='relu'))

# Layer 3 = hidden layer
dnnModel.add(layers.Dense(30, activation='relu'))


# Layer 4 = output layer
dnnModel.add(layers.Dense(10, activation='softmax'))

dnnModel.summary()
```

```
/usr/local/lib/python3.10/dist-packages/keras/src/layers/core/dense.py:87: UserWarnin
g: Do not pass an `input_shape`/`input_dim` argument to a layer. When using Sequential
models, prefer using an `Input(shape)` object as the first layer in the model instead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
```

Model: "sequential"

Layer (type)	Output Shape	
dense (Dense)	(None, 50)	
dense_1 (Dense)	(None, 60)	
dense_2 (Dense)	(None, 30)	
dense_3 (Dense)	(None, 10)	

Total params: 44,450 (173.63 KB)
Trainable params: 44,450 (173.63 KB)
Non-trainable params: 0 (0.00 B)

In [12]:
```
dnnModel_adam = models.Sequential()
dnnModel_adam.add(layers.Dense(50, activation='relu', input_shape= (28*28,)))
dnnModel_adam.add(layers.Dense(60, activation='relu'))
dnnModel_adam.add(layers.Dense(30, activation='relu'))
```

```
dnnModel_adam.add(layers.Dense(10, activation='softmax'))
dnnModel_adam.summary()
```

Model: "sequential_1"

Layer (type)	Output Shape	
dense_4 (Dense)	(None, 50)	
dense_5 (Dense)	(None, 60)	
dense_6 (Dense)	(None, 30)	
dense_7 (Dense)	(None, 10)	

Total params: 44,450 (173.63 KB)

Trainable params: 44,450 (173.63 KB)

Non-trainable params: 0 (0.00 B)

In [13]:
```
dnnModel_rmsprop = models.Sequential()
dnnModel_rmsprop.add(layers.Dense(50, activation='relu', input_shape= (28*28,)))
dnnModel_rmsprop.add(layers.Dense(60, activation='relu'))
dnnModel_rmsprop.add(layers.Dense(30, activation='relu'))
dnnModel_rmsprop.add(layers.Dense(10, activation='softmax'))
dnnModel_rmsprop.summary()
```

Model: "sequential_2"

Layer (type)	Output Shape	
dense_8 (Dense)	(None, 50)	
dense_9 (Dense)	(None, 60)	
dense_10 (Dense)	(None, 30)	
dense_11 (Dense)	(None, 10)	

Total params: 44,450 (173.63 KB)

Trainable params: 44,450 (173.63 KB)

Non-trainable params: 0 (0.00 B)

Regularization and Optimizations of DNN

In [14]:
```
# Configure  the model for training, by using appropriate optimizers and regularizati
# Available optimizer: adam, rmsprop, adagrad, sgd
# loss:  objective that the model will try to minimize.
# Available loss: categorical_crossentropy, binary_crossentropy, mean_squared_error
# metrics: List of metrics to be evaluated by the model during training and testing.

dnnModel.compile( optimizer = 'sgd', loss = 'categorical_crossentropy', metrics=['acc
dnnModel_adam.compile( optimizer = 'adam', loss = 'categorical_crossentropy', metrics
dnnModel_rmsprop.compile( optimizer = 'rmsprop', loss = 'categorical_crossentropy', m
```

Train the Model

In [15]:
```
# train the model

h_sgd  = dnnModel.fit( Xtrain, Ytrain, epochs=25, batch_size=64, validation_split=0.1
h_adam  = dnnModel_adam.fit( Xtrain, Ytrain, epochs=25, batch_size=64, validation_spl
h_rms  = dnnModel_rmsprop.fit( Xtrain, Ytrain, epochs=25, batch_size=64, validation_s
```

```
Epoch 1/25
844/844 ━━━━━━━━━━━━━━━━━━━━ 3s 3ms/step - accuracy: 0.3466 - loss: 1.9892 - val_accur
acy: 0.8655 - val_loss: 0.5466
Epoch 2/25
844/844 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - accuracy: 0.8531 - loss: 0.5383 - val_accur
acy: 0.9150 - val_loss: 0.3128
Epoch 3/25
844/844 ━━━━━━━━━━━━━━━━━━━━ 3s 4ms/step - accuracy: 0.8921 - loss: 0.3771 - val_accur
acy: 0.9290 - val_loss: 0.2628
Epoch 4/25
844/844 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - accuracy: 0.9059 - loss: 0.3222 - val_accur
acy: 0.9368 - val_loss: 0.2338
Epoch 5/25
844/844 ━━━━━━━━━━━━━━━━━━━━ 3s 3ms/step - accuracy: 0.9159 - loss: 0.2864 - val_accur
acy: 0.9435 - val_loss: 0.2121
Epoch 6/25
844/844 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - accuracy: 0.9231 - loss: 0.2590 - val_accur
acy: 0.9485 - val_loss: 0.1942
Epoch 7/25
844/844 ━━━━━━━━━━━━━━━━━━━━ 2s 3ms/step - accuracy: 0.9298 - loss: 0.2365 - val_accur
acy: 0.9500 - val_loss: 0.1800
Epoch 8/25
844/844 ━━━━━━━━━━━━━━━━━━━━ 3s 4ms/step - accuracy: 0.9352 - loss: 0.2177 - val_accur
acy: 0.9533 - val_loss: 0.1683
Epoch 9/25
844/844 ━━━━━━━━━━━━━━━━━━━━ 4s 3ms/step - accuracy: 0.9406 - loss: 0.2016 - val_accur
acy: 0.9558 - val_loss: 0.1584
Epoch 10/25
844/844 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - accuracy: 0.9458 - loss: 0.1876 - val_accur
acy: 0.9587 - val_loss: 0.1500
Epoch 11/25
844/844 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - accuracy: 0.9492 - loss: 0.1756 - val_accur
acy: 0.9597 - val_loss: 0.1428
Epoch 12/25
844/844 ━━━━━━━━━━━━━━━━━━━━ 3s 4ms/step - accuracy: 0.9526 - loss: 0.1650 - val_accur
acy: 0.9612 - val_loss: 0.1367
Epoch 13/25
844/844 ━━━━━━━━━━━━━━━━━━━━ 2s 3ms/step - accuracy: 0.9547 - loss: 0.1555 - val_accur
acy: 0.9632 - val_loss: 0.1314
Epoch 14/25
844/844 ━━━━━━━━━━━━━━━━━━━━ 2s 3ms/step - accuracy: 0.9572 - loss: 0.1470 - val_accur
acy: 0.9652 - val_loss: 0.1268
Epoch 15/25
844/844 ━━━━━━━━━━━━━━━━━━━━ 2s 3ms/step - accuracy: 0.9596 - loss: 0.1393 - val_accur
acy: 0.9658 - val_loss: 0.1226
Epoch 16/25
844/844 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - accuracy: 0.9617 - loss: 0.1324 - val_accur
acy: 0.9670 - val_loss: 0.1193
Epoch 17/25
844/844 ━━━━━━━━━━━━━━━━━━━━ 3s 4ms/step - accuracy: 0.9636 - loss: 0.1262 - val_accur
acy: 0.9677 - val_loss: 0.1162
Epoch 18/25
844/844 ━━━━━━━━━━━━━━━━━━━━ 2s 3ms/step - accuracy: 0.9648 - loss: 0.1206 - val_accur
acy: 0.9673 - val_loss: 0.1136
Epoch 19/25
844/844 ━━━━━━━━━━━━━━━━━━━━ 2s 3ms/step - accuracy: 0.9665 - loss: 0.1154 - val_accur
acy: 0.9677 - val_loss: 0.1114
Epoch 20/25
844/844 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - accuracy: 0.9678 - loss: 0.1105 - val_accur
acy: 0.9677 - val_loss: 0.1097
Epoch 21/25
844/844 ━━━━━━━━━━━━━━━━━━━━ 3s 3ms/step - accuracy: 0.9689 - loss: 0.1060 - val_accur
acy: 0.9680 - val_loss: 0.1079
Epoch 22/25
844/844 ━━━━━━━━━━━━━━━━━━━━ 6s 6ms/step - accuracy: 0.9700 - loss: 0.1017 - val_accur
acy: 0.9677 - val_loss: 0.1061
```

```
Epoch 23/25
844/844 ━━━━━━━━━━━━━━━━━━━━ 7s 2ms/step - accuracy: 0.9713 - loss: 0.0977 - val_accur
acy: 0.9680 - val_loss: 0.1044
Epoch 24/25
844/844 ━━━━━━━━━━━━━━━━━━━━ 3s 3ms/step - accuracy: 0.9723 - loss: 0.0940 - val_accur
acy: 0.9683 - val_loss: 0.1031
Epoch 25/25
844/844 ━━━━━━━━━━━━━━━━━━━━ 3s 4ms/step - accuracy: 0.9735 - loss: 0.0905 - val_accur
acy: 0.9695 - val_loss: 0.1017
Epoch 1/25
844/844 ━━━━━━━━━━━━━━━━━━━━ 4s 3ms/step - accuracy: 0.7863 - loss: 0.6804 - val_accur
acy: 0.9540 - val_loss: 0.1538
Epoch 2/25
844/844 ━━━━━━━━━━━━━━━━━━━━ 3s 3ms/step - accuracy: 0.9505 - loss: 0.1663 - val_accur
acy: 0.9645 - val_loss: 0.1154
Epoch 3/25
844/844 ━━━━━━━━━━━━━━━━━━━━ 3s 4ms/step - accuracy: 0.9636 - loss: 0.1227 - val_accur
acy: 0.9693 - val_loss: 0.1035
Epoch 4/25
844/844 ━━━━━━━━━━━━━━━━━━━━ 4s 3ms/step - accuracy: 0.9707 - loss: 0.0988 - val_accur
acy: 0.9707 - val_loss: 0.0984
Epoch 5/25
844/844 ━━━━━━━━━━━━━━━━━━━━ 2s 3ms/step - accuracy: 0.9760 - loss: 0.0814 - val_accur
acy: 0.9723 - val_loss: 0.0999
Epoch 6/25
844/844 ━━━━━━━━━━━━━━━━━━━━ 3s 3ms/step - accuracy: 0.9799 - loss: 0.0685 - val_accur
acy: 0.9722 - val_loss: 0.1050
Epoch 7/25
844/844 ━━━━━━━━━━━━━━━━━━━━ 3s 4ms/step - accuracy: 0.9838 - loss: 0.0570 - val_accur
acy: 0.9715 - val_loss: 0.1098
Epoch 8/25
844/844 ━━━━━━━━━━━━━━━━━━━━ 4s 3ms/step - accuracy: 0.9862 - loss: 0.0476 - val_accur
acy: 0.9745 - val_loss: 0.1062
Epoch 9/25
844/844 ━━━━━━━━━━━━━━━━━━━━ 2s 3ms/step - accuracy: 0.9876 - loss: 0.0411 - val_accur
acy: 0.9723 - val_loss: 0.1155
Epoch 10/25
844/844 ━━━━━━━━━━━━━━━━━━━━ 3s 3ms/step - accuracy: 0.9887 - loss: 0.0375 - val_accur
acy: 0.9740 - val_loss: 0.1115
Epoch 11/25
844/844 ━━━━━━━━━━━━━━━━━━━━ 4s 4ms/step - accuracy: 0.9892 - loss: 0.0340 - val_accur
acy: 0.9735 - val_loss: 0.1187
Epoch 12/25
844/844 ━━━━━━━━━━━━━━━━━━━━ 4s 3ms/step - accuracy: 0.9895 - loss: 0.0306 - val_accur
acy: 0.9753 - val_loss: 0.1167
Epoch 13/25
844/844 ━━━━━━━━━━━━━━━━━━━━ 3s 3ms/step - accuracy: 0.9908 - loss: 0.0275 - val_accur
acy: 0.9733 - val_loss: 0.1291
Epoch 14/25
844/844 ━━━━━━━━━━━━━━━━━━━━ 3s 3ms/step - accuracy: 0.9924 - loss: 0.0231 - val_accur
acy: 0.9718 - val_loss: 0.1361
Epoch 15/25
844/844 ━━━━━━━━━━━━━━━━━━━━ 6s 4ms/step - accuracy: 0.9908 - loss: 0.0267 - val_accur
acy: 0.9755 - val_loss: 0.1207
Epoch 16/25
844/844 ━━━━━━━━━━━━━━━━━━━━ 5s 3ms/step - accuracy: 0.9936 - loss: 0.0201 - val_accur
acy: 0.9715 - val_loss: 0.1417
Epoch 17/25
844/844 ━━━━━━━━━━━━━━━━━━━━ 6s 4ms/step - accuracy: 0.9937 - loss: 0.0184 - val_accur
acy: 0.9738 - val_loss: 0.1381
Epoch 18/25
844/844 ━━━━━━━━━━━━━━━━━━━━ 3s 4ms/step - accuracy: 0.9943 - loss: 0.0167 - val_accur
acy: 0.9753 - val_loss: 0.1318
Epoch 19/25
844/844 ━━━━━━━━━━━━━━━━━━━━ 5s 3ms/step - accuracy: 0.9940 - loss: 0.0166 - val_accur
acy: 0.9735 - val_loss: 0.1541
```

```
Epoch 20/25
844/844 ━━━━━━━━━━━━━━━━━━━━ 3s 3ms/step - accuracy: 0.9934 - loss: 0.0173 - val_accur
acy: 0.9727 - val_loss: 0.1558
Epoch 21/25
844/844 ━━━━━━━━━━━━━━━━━━━━ 4s 4ms/step - accuracy: 0.9950 - loss: 0.0143 - val_accur
acy: 0.9707 - val_loss: 0.1766
Epoch 22/25
844/844 ━━━━━━━━━━━━━━━━━━━━ 4s 3ms/step - accuracy: 0.9945 - loss: 0.0164 - val_accur
acy: 0.9720 - val_loss: 0.1650
Epoch 23/25
844/844 ━━━━━━━━━━━━━━━━━━━━ 5s 3ms/step - accuracy: 0.9951 - loss: 0.0140 - val_accur
acy: 0.9778 - val_loss: 0.1407
Epoch 24/25
844/844 ━━━━━━━━━━━━━━━━━━━━ 3s 4ms/step - accuracy: 0.9936 - loss: 0.0180 - val_accur
acy: 0.9727 - val_loss: 0.1656
Epoch 25/25
844/844 ━━━━━━━━━━━━━━━━━━━━ 4s 3ms/step - accuracy: 0.9949 - loss: 0.0156 - val_accur
acy: 0.9722 - val_loss: 0.1718
Epoch 1/25
844/844 ━━━━━━━━━━━━━━━━━━━━ 4s 3ms/step - accuracy: 0.8110 - loss: 0.6460 - val_accur
acy: 0.9548 - val_loss: 0.1552
Epoch 2/25
844/844 ━━━━━━━━━━━━━━━━━━━━ 6s 5ms/step - accuracy: 0.9456 - loss: 0.1839 - val_accur
acy: 0.9642 - val_loss: 0.1208
Epoch 3/25
844/844 ━━━━━━━━━━━━━━━━━━━━ 4s 3ms/step - accuracy: 0.9602 - loss: 0.1321 - val_accur
acy: 0.9698 - val_loss: 0.1060
Epoch 4/25
844/844 ━━━━━━━━━━━━━━━━━━━━ 2s 3ms/step - accuracy: 0.9686 - loss: 0.1054 - val_accur
acy: 0.9697 - val_loss: 0.1048
Epoch 5/25
844/844 ━━━━━━━━━━━━━━━━━━━━ 3s 3ms/step - accuracy: 0.9742 - loss: 0.0877 - val_accur
acy: 0.9705 - val_loss: 0.1010
Epoch 6/25
844/844 ━━━━━━━━━━━━━━━━━━━━ 4s 4ms/step - accuracy: 0.9784 - loss: 0.0738 - val_accur
acy: 0.9713 - val_loss: 0.1064
Epoch 7/25
844/844 ━━━━━━━━━━━━━━━━━━━━ 3s 3ms/step - accuracy: 0.9814 - loss: 0.0638 - val_accur
acy: 0.9710 - val_loss: 0.1109
Epoch 8/25
844/844 ━━━━━━━━━━━━━━━━━━━━ 5s 3ms/step - accuracy: 0.9842 - loss: 0.0556 - val_accur
acy: 0.9712 - val_loss: 0.1153
Epoch 9/25
844/844 ━━━━━━━━━━━━━━━━━━━━ 3s 3ms/step - accuracy: 0.9859 - loss: 0.0497 - val_accur
acy: 0.9703 - val_loss: 0.1248
Epoch 10/25
844/844 ━━━━━━━━━━━━━━━━━━━━ 4s 5ms/step - accuracy: 0.9878 - loss: 0.0444 - val_accur
acy: 0.9705 - val_loss: 0.1309
Epoch 11/25
844/844 ━━━━━━━━━━━━━━━━━━━━ 2s 3ms/step - accuracy: 0.9890 - loss: 0.0400 - val_accur
acy: 0.9708 - val_loss: 0.1253
Epoch 12/25
844/844 ━━━━━━━━━━━━━━━━━━━━ 3s 3ms/step - accuracy: 0.9898 - loss: 0.0356 - val_accur
acy: 0.9688 - val_loss: 0.1393
Epoch 13/25
844/844 ━━━━━━━━━━━━━━━━━━━━ 3s 3ms/step - accuracy: 0.9909 - loss: 0.0322 - val_accur
acy: 0.9713 - val_loss: 0.1477
Epoch 14/25
844/844 ━━━━━━━━━━━━━━━━━━━━ 6s 5ms/step - accuracy: 0.9919 - loss: 0.0301 - val_accur
acy: 0.9700 - val_loss: 0.1484
Epoch 15/25
844/844 ━━━━━━━━━━━━━━━━━━━━ 2s 3ms/step - accuracy: 0.9923 - loss: 0.0259 - val_accur
acy: 0.9717 - val_loss: 0.1535
Epoch 16/25
844/844 ━━━━━━━━━━━━━━━━━━━━ 3s 3ms/step - accuracy: 0.9928 - loss: 0.0240 - val_accur
acy: 0.9733 - val_loss: 0.1537
```

```
Epoch 17/25
844/844 ━━━━━━━━━━━━━━━━━━━━ 3s 3ms/step - accuracy: 0.9933 - loss: 0.0228 - val_accur
acy: 0.9730 - val_loss: 0.1714
Epoch 18/25
844/844 ━━━━━━━━━━━━━━━━━━━━ 6s 4ms/step - accuracy: 0.9938 - loss: 0.0207 - val_accur
acy: 0.9708 - val_loss: 0.1738
Epoch 19/25
844/844 ━━━━━━━━━━━━━━━━━━━━ 5s 3ms/step - accuracy: 0.9945 - loss: 0.0192 - val_accur
acy: 0.9722 - val_loss: 0.1696
Epoch 20/25
844/844 ━━━━━━━━━━━━━━━━━━━━ 5s 4ms/step - accuracy: 0.9947 - loss: 0.0176 - val_accur
acy: 0.9713 - val_loss: 0.1812
Epoch 21/25
844/844 ━━━━━━━━━━━━━━━━━━━━ 5s 3ms/step - accuracy: 0.9959 - loss: 0.0162 - val_accur
acy: 0.9742 - val_loss: 0.1803
Epoch 22/25
844/844 ━━━━━━━━━━━━━━━━━━━━ 3s 3ms/step - accuracy: 0.9954 - loss: 0.0154 - val_accur
acy: 0.9730 - val_loss: 0.2123
Epoch 23/25
844/844 ━━━━━━━━━━━━━━━━━━━━ 3s 3ms/step - accuracy: 0.9946 - loss: 0.0159 - val_accur
acy: 0.9720 - val_loss: 0.2140
Epoch 24/25
844/844 ━━━━━━━━━━━━━━━━━━━━ 6s 4ms/step - accuracy: 0.9959 - loss: 0.0144 - val_accur
acy: 0.9687 - val_loss: 0.2237
Epoch 25/25
844/844 ━━━━━━━━━━━━━━━━━━━━ 3s 3ms/step - accuracy: 0.9960 - loss: 0.0122 - val_accur
acy: 0.9722 - val_loss: 0.2206
```

In [17]:

```
#print('Final training loss \t', h.history['loss'][-1])
print('SGD Final training accuracy ', h_sgd.history['accuracy'][-1])
print('Adam Final training accuracy ', h_adam.history['accuracy'][-1])
print('RMSProp Final training accuracy ', h_rms.history['accuracy'][-1])
```

```
SGD Final training accuracy  0.9737407565116882
Adam Final training accuracy  0.9953148365020752
RMSProp Final training accuracy  0.995888888835907
```

Testing the Model

In [18]:

```
# testing the model

testLoss_sgd, testAccuracy_sgd = dnnModel.evaluate( Xtest, Ytest)
testLoss_adam, testAccuracy_adam = dnnModel_adam.evaluate( Xtest, Ytest)
testLoss_rms, testAccuracy_rms = dnnModel_rmsprop.evaluate( Xtest, Ytest)
```

```
313/313 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.9592 - loss: 0.1311
313/313 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.9629 - loss: 0.1929
313/313 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.9638 - loss: 0.2246
```

In [21]:

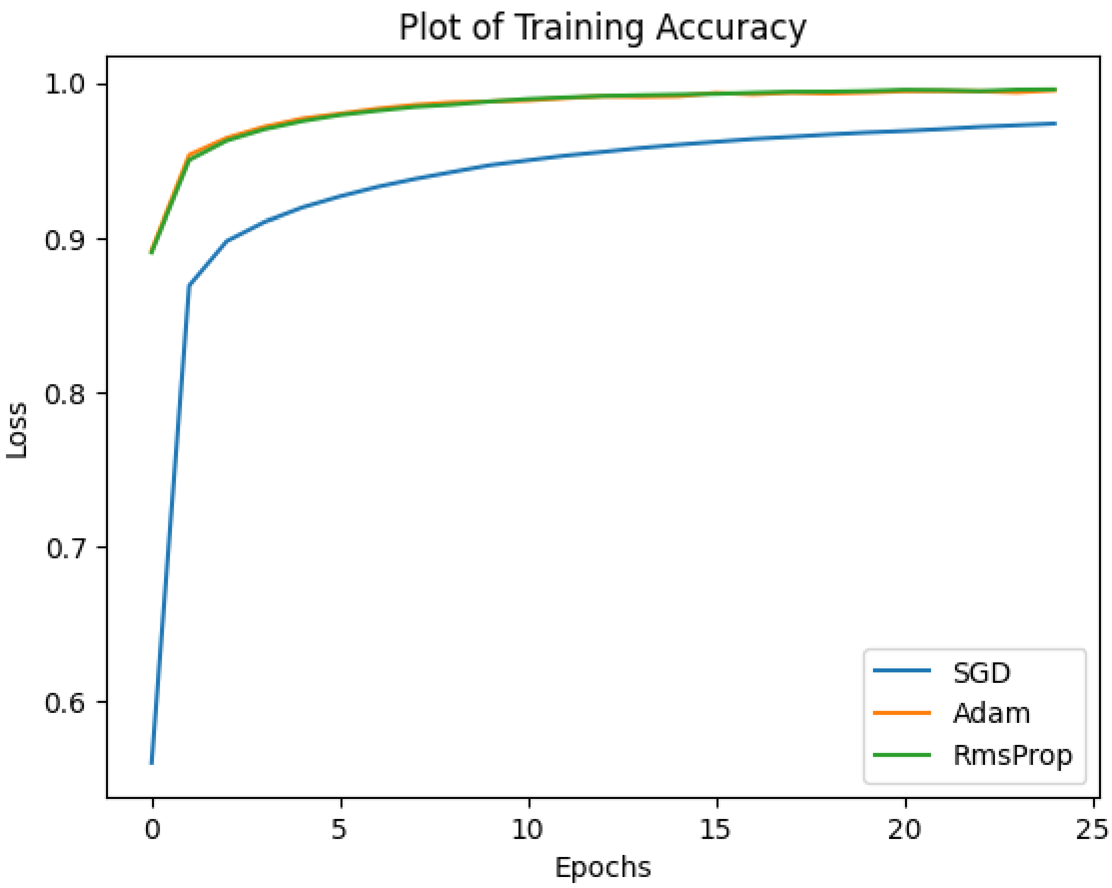
```
print('Testing loss SGD\t', testLoss_sgd) # Added '_sgd' to 'testLoss'
print('Testing loss Adam\t', testLoss_adam) # Added '_adam' to 'testLoss'
print('Testing loss RMS\t', testLoss_rms) # Added '_rms' to 'testLoss'

print('Testing accuracy SGD', testAccuracy_sgd) # Added '_sgd' to 'testAccuracy'
print('Testing accuracy Adam', testAccuracy_adam) # Added '_adam' to 'testAccuracy'
print('Testing accuracy RMS', testAccuracy_rms) # Added '_rms' to 'testAccuracy'
```

```
Testing loss SGD	 0.11305821686983109
Testing loss Adam	 0.16522839665412903
Testing loss RMS	 0.19557134807109833
Testing accuracy SGD 0.9650999903678894
Testing accuracy Adam 0.9675999879837036
Testing accuracy RMS 0.9699000120162964
```

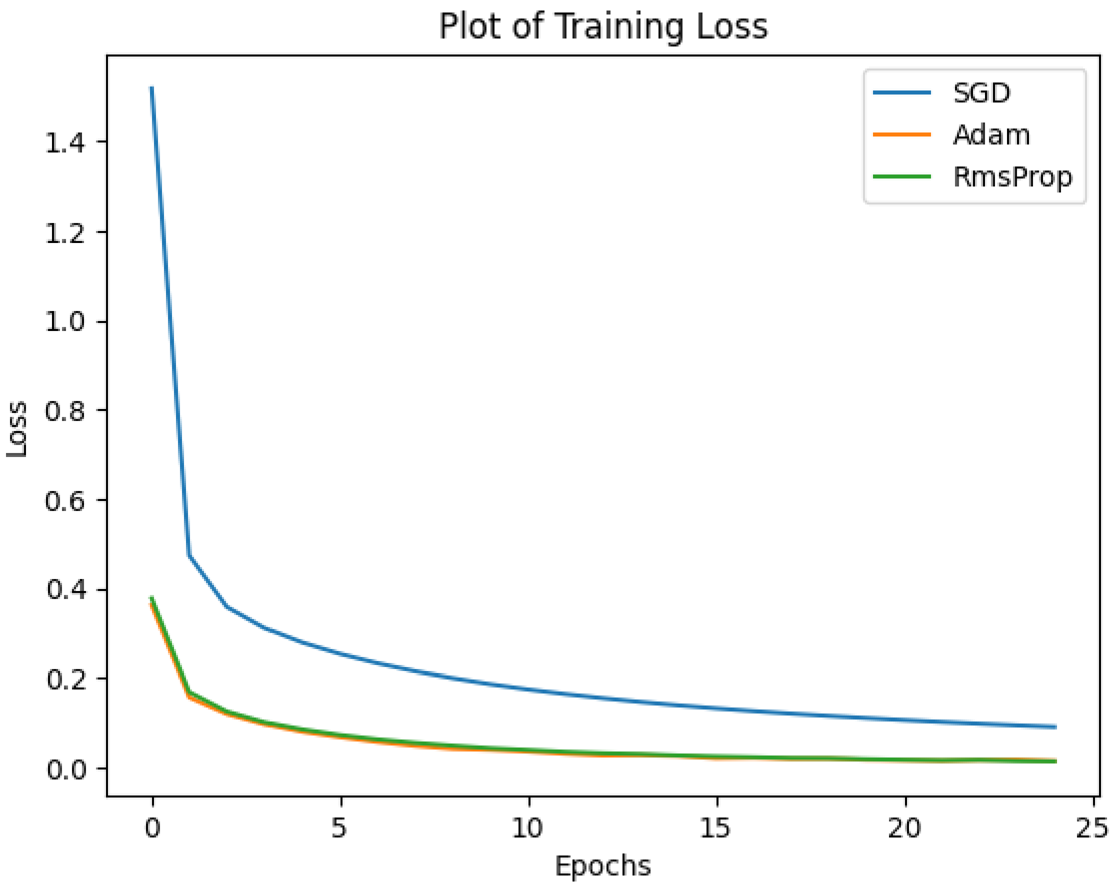
In [23]:

```
# plot the training accuracy
```

```python
plt.plot(h_sgd.history['accuracy'], label='SGD')
plt.plot(h_adam.history['accuracy'], label='Adam')
plt.plot(h_rms.history['accuracy'], label='RmsProp')
#plt.plot(h.history['val_acc'], label='Val Acc')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.title('Plot of Training Accuracy')
plt.legend()
plt.show()
```



In [24]:
```python
# plot the training loss

plt.plot(h_sgd.history['loss'], label='SGD')
plt.plot(h_adam.history['loss'], label='Adam')
plt.plot(h_rms.history['loss'], label='RmsProp')
#plt.plot(hes.history['val_loss'], label='Val loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.title('Plot of Training Loss')
plt.legend()
plt.show()
```



Exercise

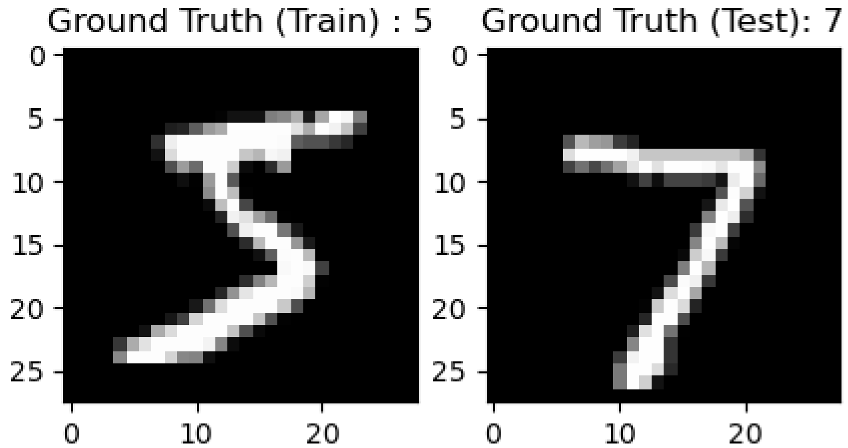
Modify the code to get a better testing accuracy.

- Change the number of hidden units
- Increase the number of hidden layers
- Use a different optimizer

In [1]:
```
import tensorflow as tf

from tensorflow.keras import models
from tensorflow.keras import layers

import random
import numpy as np
random.seed(42)         # Initialize the random number generator.
np.random.seed(42)      # With the seed reset, the same set of
                        # numbers will appear every time.
tf.random.set_seed(42) # sets the graph-level random seed

mnist = tf.keras.datasets.mnist

(Xtrain, Ytrain) , (Xtest, Ytest) = mnist.load_data()
import matplotlib.pyplot as plt

plt.figure(figsize=[5,5])

# Display the first image in training data
plt.subplot(121)
plt.imshow(Xtrain[0,:,:], cmap='gray')
plt.title("Ground Truth (Train) : {}".format(Ytrain[0]))

# Display the first image in testing data
plt.subplot(122)
```

```
plt.imshow(Xtest[0,:,:], cmap='gray')
plt.title("Ground Truth (Test): {}".format(Ytest[0]))
# size of the datsets

print(Xtrain.shape)
print(Xtest.shape)
print(Ytrain.shape)
print(Ytest.shape)

print('Xtrain \n', Xtrain[10,10])
print('Xtest \n', Xtest[10,10])
print('Ytrain \n', Ytrain[10,])
print('Ytest \n', Ytest[10,])
```

```
2024-09-22 22:15:30.211245: I tensorflow/core/util/port.cc:153] oneDNN custom operatio
ns are on. You may see slightly different numerical results due to floating-point roun
d-off errors from different computation orders. To turn them off, set the environment
variable `TF_ENABLE_ONEDNN_OPTS=0`.
2024-09-22 22:15:30.219078: E external/local_xla/xla/stream_executor/cuda/cuda_fft.cc:
485] Unable to register cuFFT factory: Attempting to register factory for plugin cuFFT
when one has already been registered
2024-09-22 22:15:30.228197: E external/local_xla/xla/stream_executor/cuda/cuda_dnn.cc:
8454] Unable to register cuDNN factory: Attempting to register factory for plugin cuDN
N when one has already been registered
2024-09-22 22:15:30.230857: E external/local_xla/xla/stream_executor/cuda/cuda_blas.c
c:1452] Unable to register cuBLAS factory: Attempting to register factory for plugin c
uBLAS when one has already been registered
2024-09-22 22:15:30.237967: I tensorflow/core/platform/cpu_feature_guard.cc:210] This
TensorFlow binary is optimized to use available CPU instructions in performance-critic
al operations.
To enable the following instructions: AVX2 AVX_VNNI FMA, in other operations, rebuild
TensorFlow with the appropriate compiler flags.
2024-09-22 22:15:30.760275: W tensorflow/compiler/tf2tensorrt/utils/py_utils.cc:38] TF
-TRT Warning: Could not find TensorRT
(60000, 28, 28)
(10000, 28, 28)
(60000,)
(10000,)
Xtrain
 [  0   0   0   0   0   0   0   0   0   0   0   0   0  24 209 254 254 254
 171   0   0   0   0   0   0   0   0   0]
Xtest
 [  0   0   0   0   0   0   0   0 194 254 103   0   0   0   0   0   0   0
   0   0 150 254 213   0   0   0   0   0]
Ytrain
 3
Ytest
 0
```



In [2]:
```
# Normalize the data
# 60000 input images are in the train set.
```

```
# 10000 input images are in the test set.

Xtrain = Xtrain.reshape((60000, 28*28))    # reshape the input set to size 28*28.
Xtrain = Xtrain.astype('float32')/255      # normalize to grayscale; set datatype as

Xtest = Xtest.reshape((10000, 28*28))      # reshape the input set to size 28*28.
Xtest = Xtest.astype('float32')/255        # normalize to grayscale; set datatype as

Ytrain = tf.keras.utils.to_categorical(Ytrain)
Ytest = tf.keras.utils.to_categorical(Ytest)
# print a sample data

print('Xtrain \n', Xtrain[10,10])
print('Xtest \n', Xtest[10,10])
print('Ytrain \n', Ytrain[10,])
print('Ytest \n', Ytest[10,])
```

```
Xtrain
 0.0
Xtest
 0.0
Ytrain
 [0. 0. 0. 1. 0. 0. 0. 0. 0. 0.]
Ytest
 [1. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
```

In [3]:
```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.optimizers import Nadam

# Build a new deeper model
model = Sequential()
# Adding more layers and increasing hidden units
model.add(Dense(256, input_dim=30, activation='relu'))
model.add(Dense(128, activation='relu'))
model.add(Dense(128, activation='relu'))
model.add(Dense(64, activation='relu'))
model.add(Dense(32, activation='relu'))
model.add(Dense(1, activation='sigmoid'))  # Output layer

# Compile model with Nadam optimizer
model.compile(loss='binary_crossentropy', optimizer=Nadam(), metrics=['accuracy'])
# Model summary
model.summary()
```

```
/home/samara/anaconda3/lib/python3.11/site-packages/keras/src/layers/core/dense.py:87:
UserWarning: Do not pass an `input_shape`/`input_dim` argument to a layer. When using
Sequential models, prefer using an `Input(shape)` object as the first layer in the mod
el instead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
2024-09-22 22:16:00.894938: E external/local_xla/xla/stream_executor/cuda/cuda_driver.
cc:266] failed call to cuInit: CUDA_ERROR_UNKNOWN: unknown error
2024-09-22 22:16:00.894955: I external/local_xla/xla/stream_executor/cuda/cuda_diagnos
tics.cc:135] retrieving CUDA diagnostic information for host: hitloop
2024-09-22 22:16:00.894958: I external/local_xla/xla/stream_executor/cuda/cuda_diagnos
tics.cc:142] hostname: hitloop
2024-09-22 22:16:00.895018: I external/local_xla/xla/stream_executor/cuda/cuda_diagnos
tics.cc:166] libcuda reported version is: 560.35.3
2024-09-22 22:16:00.895027: I external/local_xla/xla/stream_executor/cuda/cuda_diagnos
tics.cc:170] kernel reported version is: 560.35.3
2024-09-22 22:16:00.895029: I external/local_xla/xla/stream_executor/cuda/cuda_diagnos
tics.cc:249] kernel version seems to match DSO: 560.35.3
```

Model: "sequential"

Layer (type)	Output Shape	Param #
dense (Dense)	(None, 256)	7,936
dense_1 (Dense)	(None, 128)	32,896
dense_2 (Dense)	(None, 128)	16,512
dense_3 (Dense)	(None, 64)	8,256
dense_4 (Dense)	(None, 32)	2,080
dense_5 (Dense)	(None, 1)	33

Total params: 67,713 (264.50 KB)

Trainable params: 67,713 (264.50 KB)

Non-trainable params: 0 (0.00 B)

In [4]:
```
dnnModel = models.Sequential()
# Layer 1 = input layer
# specify the input size in the first layer.

dnnModel.add(layers.Dense(50, activation='relu', input_shape= (28*28,)))

# Layer 2 = hidden layer
dnnModel.add(layers.Dense(60, activation='relu'))

# Layer 3 = hidden layer
dnnModel.add(layers.Dense(30, activation='relu'))


# Layer 4 = output layer
dnnModel.add(layers.Dense(10, activation='softmax'))

dnnModel.summary()
```

Model: "sequential_1"

Layer (type)	Output Shape	Param #
dense_6 (Dense)	(None, 50)	39,250
dense_7 (Dense)	(None, 60)	3,060
dense_8 (Dense)	(None, 30)	1,830
dense_9 (Dense)	(None, 10)	310

Total params: 44,450 (173.63 KB)

Trainable params: 44,450 (173.63 KB)

Non-trainable params: 0 (0.00 B)

In [5]:
```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.optimizers import Nadam

# Build a new deeper model
model = Sequential()
# Adding more layers and increasing hidden units
model.add(Dense(256, input_dim=30, activation='relu'))
model.add(Dense(128, activation='relu'))
model.add(Dense(128, activation='relu'))
model.add(Dense(64, activation='relu'))
```

```
model.add(Dense(32, activation='relu'))
model.add(Dense(1, activation='sigmoid'))  # Output layer

# Compile model with Nadam optimizer
model.compile(loss='binary_crossentropy', optimizer=Nadam(), metrics=['accuracy'])
# Model summary
model.summary()
```

Model: "sequential_2"

Layer (type)	Output Shape	Param #
dense_10 (Dense)	(None, 256)	7,936
dense_11 (Dense)	(None, 128)	32,896
dense_12 (Dense)	(None, 128)	16,512
dense_13 (Dense)	(None, 64)	8,256
dense_14 (Dense)	(None, 32)	2,080
dense_15 (Dense)	(None, 1)	33

Total params: 67,713 (264.50 KB)

Trainable params: 67,713 (264.50 KB)

Non-trainable params: 0 (0.00 B)

In [6]:
```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.optimizers import Nadam

# Build a new deeper model
model = Sequential()
# Adding more layers and increasing hidden units
model.add(Dense(256, input_dim=30, activation='relu'))
model.add(Dense(128, activation='relu'))
model.add(Dense(128, activation='relu'))
model.add(Dense(64, activation='relu'))
model.add(Dense(32, activation='relu'))
model.add(Dense(1, activation='sigmoid'))  # Output layer

# Compile model with Nadam optimizer
model.compile(loss='binary_crossentropy', optimizer=Nadam(), metrics=['accuracy'])
# Model summary
model.summary()
```

Model: "sequential_3"

Layer (type)	Output Shape	Param #
dense_16 (Dense)	(None, 256)	7,936
dense_17 (Dense)	(None, 128)	32,896
dense_18 (Dense)	(None, 128)	16,512
dense_19 (Dense)	(None, 64)	8,256
dense_20 (Dense)	(None, 32)	2,080
dense_21 (Dense)	(None, 1)	33

Total params: 67,713 (264.50 KB)

Trainable params: 67,713 (264.50 KB)

Non-trainable params: 0 (0.00 B)

In [13]:
```
# Configure  the model for training, by using appropriate optimizers and regularizati
# Available optimizer: adam, rmsprop, adagrad, sgd
# loss:  objective that the model will try to minimize.
# Available loss: categorical_crossentropy, binary_crossentropy, mean_squared_error
# metrics: List of metrics to be evaluated by the model during training and testing.

dnnModel.compile( optimizer = 'sgd', loss = 'categorical_crossentropy', metrics=['acc
dnnModel_adam = models.Sequential()
dnnModel_adam.add(layers.Dense(50, activation='relu', input_shape= (28*28,)))
dnnModel_adam.add(layers.Dense(60, activation='relu'))
dnnModel_adam.add(layers.Dense(30, activation='relu'))
dnnModel_adam.add(layers.Dense(10, activation='softmax'))
dnnModel_adam.summary()
dnnModel_adam.compile( optimizer = 'adam', loss = 'categorical_crossentropy', metrics
dnnModel_rmsprop = models.Sequential()
dnnModel_rmsprop.add(layers.Dense(50, activation='relu', input_shape= (28*28,)))
dnnModel_rmsprop.add(layers.Dense(60, activation='relu'))
dnnModel_rmsprop.add(layers.Dense(30, activation='relu'))
dnnModel_rmsprop.add(layers.Dense(10, activation='softmax'))
dnnModel_rmsprop.summary()
dnnModel_rmsprop.compile( optimizer = 'rmsprop', loss = 'categorical_crossentropy', m
# train the model

h_sgd  = dnnModel.fit( Xtrain, Ytrain, epochs=25, batch_size=64, validation_split=0.1
h_adam  = dnnModel_adam.fit( Xtrain, Ytrain, epochs=25, batch_size=64, validation_spl
h_rms  = dnnModel_rmsprop.fit( Xtrain, Ytrain, epochs=25, batch_size=64, validation_s
#print('Final training loss \t', h.history['loss'][-1])
```

Model: "sequential_23"

Layer (type)	Output Shape	
dense_122 (Dense)	(None, 50)	
dense_123 (Dense)	(None, 60)	
dense_124 (Dense)	(None, 30)	
dense_125 (Dense)	(None, 10)	

Total params: 44,450 (173.63 KB)

Trainable params: 44,450 (173.63 KB)

Non-trainable params: 0 (0.00 B)

Model: "sequential_24"

Layer (type)	Output Shape	
dense_126 (Dense)	(None, 50)	
dense_127 (Dense)	(None, 60)	
dense_128 (Dense)	(None, 30)	
dense_129 (Dense)	(None, 10)	

Total params: 44,450 (173.63 KB)

Trainable params: 44,450 (173.63 KB)

Non-trainable params: 0 (0.00 B)

```
Epoch 1/25
844/844 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - accuracy: 0.4235 - loss: 1.7634 - val_accur
acy: 0.8787 - val_loss: 0.4564
Epoch 2/25
844/844 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - accuracy: 0.8631 - loss: 0.4741 - val_accur
acy: 0.9170 - val_loss: 0.3043
Epoch 3/25
844/844 ━━━━━━━━━━━━━━━━━━━━ 3s 3ms/step - accuracy: 0.8943 - loss: 0.3578 - val_accur
acy: 0.9278 - val_loss: 0.2597
Epoch 4/25
844/844 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - accuracy: 0.9089 - loss: 0.3087 - val_accur
acy: 0.9378 - val_loss: 0.2321
Epoch 5/25
844/844 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - accuracy: 0.9196 - loss: 0.2741 - val_accur
acy: 0.9413 - val_loss: 0.2110
Epoch 6/25
844/844 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - accuracy: 0.9280 - loss: 0.2467 - val_accur
acy: 0.9473 - val_loss: 0.1945
Epoch 7/25
844/844 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - accuracy: 0.9345 - loss: 0.2244 - val_accur
acy: 0.9525 - val_loss: 0.1812
Epoch 8/25
844/844 ━━━━━━━━━━━━━━━━━━━━ 3s 3ms/step - accuracy: 0.9389 - loss: 0.2060 - val_accur
acy: 0.9545 - val_loss: 0.1699
Epoch 9/25
844/844 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - accuracy: 0.9434 - loss: 0.1907 - val_accur
acy: 0.9580 - val_loss: 0.1608
Epoch 10/25
844/844 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - accuracy: 0.9469 - loss: 0.1778 - val_accur
acy: 0.9607 - val_loss: 0.1532
Epoch 11/25
844/844 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - accuracy: 0.9501 - loss: 0.1667 - val_accur
acy: 0.9618 - val_loss: 0.1469
Epoch 12/25
844/844 ━━━━━━━━━━━━━━━━━━━━ 3s 3ms/step - accuracy: 0.9527 - loss: 0.1569 - val_accur
acy: 0.9627 - val_loss: 0.1413
Epoch 13/25
844/844 ━━━━━━━━━━━━━━━━━━━━ 2s 3ms/step - accuracy: 0.9554 - loss: 0.1481 - val_accur
acy: 0.9637 - val_loss: 0.1365
Epoch 14/25
844/844 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - accuracy: 0.9582 - loss: 0.1401 - val_accur
acy: 0.9647 - val_loss: 0.1322
Epoch 15/25
844/844 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - accuracy: 0.9603 - loss: 0.1329 - val_accur
acy: 0.9653 - val_loss: 0.1283
Epoch 16/25
844/844 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - accuracy: 0.9620 - loss: 0.1262 - val_accur
acy: 0.9662 - val_loss: 0.1248
Epoch 17/25
844/844 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - accuracy: 0.9642 - loss: 0.1202 - val_accur
acy: 0.9673 - val_loss: 0.1217
Epoch 18/25
844/844 ━━━━━━━━━━━━━━━━━━━━ 4s 3ms/step - accuracy: 0.9664 - loss: 0.1146 - val_accur
acy: 0.9677 - val_loss: 0.1191
Epoch 19/25
844/844 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - accuracy: 0.9678 - loss: 0.1094 - val_accur
acy: 0.9683 - val_loss: 0.1164
Epoch 20/25
844/844 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - accuracy: 0.9687 - loss: 0.1047 - val_accur
acy: 0.9690 - val_loss: 0.1143
Epoch 21/25
844/844 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - accuracy: 0.9701 - loss: 0.1003 - val_accur
acy: 0.9693 - val_loss: 0.1122
Epoch 22/25
844/844 ━━━━━━━━━━━━━━━━━━━━ 3s 3ms/step - accuracy: 0.9714 - loss: 0.0962 - val_accur
acy: 0.9697 - val_loss: 0.1103
```

```
Epoch 23/25
844/844 ━━━━━━━━━━━━━━━━━━━━ 2s 3ms/step - accuracy: 0.9729 - loss: 0.0924 - val_accur
acy: 0.9697 - val_loss: 0.1088
Epoch 24/25
844/844 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - accuracy: 0.9739 - loss: 0.0889 - val_accur
acy: 0.9707 - val_loss: 0.1075
Epoch 25/25
844/844 ━━━━━━━━━━━━━━━━━━━━ 3s 3ms/step - accuracy: 0.9749 - loss: 0.0855 - val_accur
acy: 0.9712 - val_loss: 0.1059
Epoch 1/25
844/844 ━━━━━━━━━━━━━━━━━━━━ 4s 3ms/step - accuracy: 0.7964 - loss: 0.6852 - val_accur
acy: 0.9537 - val_loss: 0.1560
Epoch 2/25
844/844 ━━━━━━━━━━━━━━━━━━━━ 4s 4ms/step - accuracy: 0.9488 - loss: 0.1701 - val_accur
acy: 0.9655 - val_loss: 0.1186
Epoch 3/25
844/844 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - accuracy: 0.9626 - loss: 0.1246 - val_accur
acy: 0.9712 - val_loss: 0.1047
Epoch 4/25
844/844 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - accuracy: 0.9695 - loss: 0.0996 - val_accur
acy: 0.9698 - val_loss: 0.1027
Epoch 5/25
844/844 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - accuracy: 0.9747 - loss: 0.0823 - val_accur
acy: 0.9692 - val_loss: 0.1035
Epoch 6/25
844/844 ━━━━━━━━━━━━━━━━━━━━ 3s 3ms/step - accuracy: 0.9802 - loss: 0.0684 - val_accur
acy: 0.9682 - val_loss: 0.1046
Epoch 7/25
844/844 ━━━━━━━━━━━━━━━━━━━━ 3s 3ms/step - accuracy: 0.9828 - loss: 0.0586 - val_accur
acy: 0.9685 - val_loss: 0.1122
Epoch 8/25
844/844 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - accuracy: 0.9854 - loss: 0.0507 - val_accur
acy: 0.9700 - val_loss: 0.1121
Epoch 9/25
844/844 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - accuracy: 0.9870 - loss: 0.0446 - val_accur
acy: 0.9700 - val_loss: 0.1130
Epoch 10/25
844/844 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - accuracy: 0.9883 - loss: 0.0386 - val_accur
acy: 0.9708 - val_loss: 0.1180
Epoch 11/25
844/844 ━━━━━━━━━━━━━━━━━━━━ 3s 4ms/step - accuracy: 0.9892 - loss: 0.0353 - val_accur
acy: 0.9713 - val_loss: 0.1146
Epoch 12/25
844/844 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - accuracy: 0.9896 - loss: 0.0308 - val_accur
acy: 0.9720 - val_loss: 0.1228
Epoch 13/25
844/844 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - accuracy: 0.9902 - loss: 0.0313 - val_accur
acy: 0.9730 - val_loss: 0.1172
Epoch 14/25
844/844 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - accuracy: 0.9914 - loss: 0.0259 - val_accur
acy: 0.9737 - val_loss: 0.1219
Epoch 15/25
844/844 ━━━━━━━━━━━━━━━━━━━━ 3s 3ms/step - accuracy: 0.9919 - loss: 0.0241 - val_accur
acy: 0.9713 - val_loss: 0.1368
Epoch 16/25
844/844 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - accuracy: 0.9922 - loss: 0.0250 - val_accur
acy: 0.9718 - val_loss: 0.1402
Epoch 17/25
844/844 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - accuracy: 0.9919 - loss: 0.0238 - val_accur
acy: 0.9703 - val_loss: 0.1430
Epoch 18/25
844/844 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - accuracy: 0.9944 - loss: 0.0184 - val_accur
acy: 0.9717 - val_loss: 0.1507
Epoch 19/25
844/844 ━━━━━━━━━━━━━━━━━━━━ 3s 3ms/step - accuracy: 0.9940 - loss: 0.0169 - val_accur
acy: 0.9693 - val_loss: 0.1686
```

```
Epoch 20/25
844/844 ━━━━━━━━━━━━━━━━━━━━ 3s 3ms/step - accuracy: 0.9917 - loss: 0.0225 - val_accur
acy: 0.9735 - val_loss: 0.1474
Epoch 21/25
844/844 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - accuracy: 0.9947 - loss: 0.0169 - val_accur
acy: 0.9703 - val_loss: 0.1724
Epoch 22/25
844/844 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - accuracy: 0.9937 - loss: 0.0181 - val_accur
acy: 0.9730 - val_loss: 0.1459
Epoch 23/25
844/844 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - accuracy: 0.9933 - loss: 0.0192 - val_accur
acy: 0.9732 - val_loss: 0.1608
Epoch 24/25
844/844 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - accuracy: 0.9947 - loss: 0.0155 - val_accur
acy: 0.9733 - val_loss: 0.1667
Epoch 25/25
844/844 ━━━━━━━━━━━━━━━━━━━━ 3s 3ms/step - accuracy: 0.9956 - loss: 0.0133 - val_accur
acy: 0.9700 - val_loss: 0.1830
Epoch 1/25
844/844 ━━━━━━━━━━━━━━━━━━━━ 3s 3ms/step - accuracy: 0.8122 - loss: 0.6427 - val_accur
acy: 0.9567 - val_loss: 0.1580
Epoch 2/25
844/844 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - accuracy: 0.9438 - loss: 0.1903 - val_accur
acy: 0.9687 - val_loss: 0.1110
Epoch 3/25
844/844 ━━━━━━━━━━━━━━━━━━━━ 3s 3ms/step - accuracy: 0.9593 - loss: 0.1373 - val_accur
acy: 0.9723 - val_loss: 0.1011
Epoch 4/25
844/844 ━━━━━━━━━━━━━━━━━━━━ 3s 3ms/step - accuracy: 0.9676 - loss: 0.1091 - val_accur
acy: 0.9735 - val_loss: 0.0951
Epoch 5/25
844/844 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - accuracy: 0.9743 - loss: 0.0900 - val_accur
acy: 0.9732 - val_loss: 0.0946
Epoch 6/25
844/844 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - accuracy: 0.9779 - loss: 0.0769 - val_accur
acy: 0.9742 - val_loss: 0.0929
Epoch 7/25
844/844 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - accuracy: 0.9813 - loss: 0.0663 - val_accur
acy: 0.9728 - val_loss: 0.0983
Epoch 8/25
844/844 ━━━━━━━━━━━━━━━━━━━━ 3s 3ms/step - accuracy: 0.9845 - loss: 0.0581 - val_accur
acy: 0.9723 - val_loss: 0.1057
Epoch 9/25
844/844 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - accuracy: 0.9861 - loss: 0.0518 - val_accur
acy: 0.9707 - val_loss: 0.1110
Epoch 10/25
844/844 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - accuracy: 0.9876 - loss: 0.0460 - val_accur
acy: 0.9720 - val_loss: 0.1143
Epoch 11/25
844/844 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - accuracy: 0.9881 - loss: 0.0429 - val_accur
acy: 0.9725 - val_loss: 0.1118
Epoch 12/25
844/844 ━━━━━━━━━━━━━━━━━━━━ 2s 3ms/step - accuracy: 0.9899 - loss: 0.0382 - val_accur
acy: 0.9723 - val_loss: 0.1192
Epoch 13/25
844/844 ━━━━━━━━━━━━━━━━━━━━ 3s 3ms/step - accuracy: 0.9909 - loss: 0.0353 - val_accur
acy: 0.9720 - val_loss: 0.1264
Epoch 14/25
844/844 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - accuracy: 0.9908 - loss: 0.0328 - val_accur
acy: 0.9725 - val_loss: 0.1344
Epoch 15/25
844/844 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - accuracy: 0.9921 - loss: 0.0285 - val_accur
acy: 0.9732 - val_loss: 0.1332
Epoch 16/25
844/844 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - accuracy: 0.9924 - loss: 0.0270 - val_accur
acy: 0.9723 - val_loss: 0.1439
```

```
Epoch 17/25
844/844 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - accuracy: 0.9922 - loss: 0.0266 - val_accur
acy: 0.9733 - val_loss: 0.1502
Epoch 18/25
844/844 ━━━━━━━━━━━━━━━━━━━━ 2s 3ms/step - accuracy: 0.9933 - loss: 0.0238 - val_accur
acy: 0.9748 - val_loss: 0.1526
Epoch 19/25
844/844 ━━━━━━━━━━━━━━━━━━━━ 3s 3ms/step - accuracy: 0.9939 - loss: 0.0207 - val_accur
acy: 0.9753 - val_loss: 0.1550
Epoch 20/25
844/844 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - accuracy: 0.9932 - loss: 0.0217 - val_accur
acy: 0.9740 - val_loss: 0.1652
Epoch 21/25
844/844 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - accuracy: 0.9944 - loss: 0.0185 - val_accur
acy: 0.9733 - val_loss: 0.1741
Epoch 22/25
844/844 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - accuracy: 0.9951 - loss: 0.0172 - val_accur
acy: 0.9730 - val_loss: 0.1712
Epoch 23/25
844/844 ━━━━━━━━━━━━━━━━━━━━ 3s 3ms/step - accuracy: 0.9948 - loss: 0.0163 - val_accur
acy: 0.9743 - val_loss: 0.1670
Epoch 24/25
844/844 ━━━━━━━━━━━━━━━━━━━━ 2s 3ms/step - accuracy: 0.9954 - loss: 0.0156 - val_accur
acy: 0.9747 - val_loss: 0.1841
Epoch 25/25
844/844 ━━━━━━━━━━━━━━━━━━━━ 2s 2ms/step - accuracy: 0.9956 - loss: 0.0141 - val_accur
acy: 0.9745 - val_loss: 0.1833
```

In [14]:

```
print('SGD Final training accuracy ', h_sgd.history['accuracy'][-1])
print('Adam Final training accuracy ', h_adam.history['accuracy'][-1])
print('RMSProp Final training accuracy ', h_rms.history['accuracy'][-1])
# testing the model

testLoss_sgd, testAccuracy_sgd = dnnModel.evaluate( Xtest, Ytest)
testLoss_adam, testAccuracy_adam = dnnModel_adam.evaluate( Xtest, Ytest)
testLoss_rms, testAccuracy_rms = dnnModel_rmsprop.evaluate( Xtest, Ytest)


print('Testing loss SGD\t', testLoss_sgd) # Added '_sgd' to 'testLoss'
print('Testing loss Adam\t', testLoss_adam) # Added '_adam' to 'testLoss'
print('Testing loss RMS\t', testLoss_rms) # Added '_rms' to 'testLoss'

print('Testing accuracy SGD', testAccuracy_sgd) # Added '_sgd' to 'testAccuracy'
print('Testing accuracy Adam', testAccuracy_adam) # Added '_adam' to 'testAccuracy'
print('Testing accuracy RMS', testAccuracy_rms) # Added '_rms' to 'testAccuracy'
```

```
SGD Final training accuracy  0.975074052810669
Adam Final training accuracy  0.9957777857780457
RMSProp Final training accuracy  0.9956111311912537
313/313 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.9628 - loss: 0.1274
313/313 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.9654 - loss: 0.1831
313/313 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.9653 - loss: 0.2271
Testing loss SGD	 0.11439009010791779
Testing loss Adam	 0.17058593034744263
Testing loss RMS	 0.2012793868780136
Testing accuracy SGD 0.967199981212616
Testing accuracy Adam 0.9688000082969666
Testing accuracy RMS 0.9693999886512756
```

In [15]:

```
plt.plot(h_sgd.history['accuracy'], label='SGD')
plt.plot(h_adam.history['accuracy'], label='Adam')
plt.plot(h_rms.history['accuracy'], label='RmsProp')
#plt.plot(h.history['val_acc'], label='Val Acc')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.title('Plot of Training Accuracy')
```

```
plt.legend()
plt.show()
plt.plot(h_sgd.history['loss'], label='SGD')
plt.plot(h_adam.history['loss'], label='Adam')
plt.plot(h_rms.history['loss'], label='RmsProp')
#plt.plot(hes.history['val_loss'], label='Val loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.title('Plot of Training Loss')
plt.legend()
plt.show()
```