CSL7670 : Fundamentals of Machine Learning

Lab Report

Name:	**Ganesh Kumar Nagal**
Roll Number:	**M23MEA004**
Program:	**M.Tech Adavance Manufacturing And Design**

Chapter 1

Lab-4

1.1 Objective

Objective of this assignment is to gain familiarity with neural networks (multi-layer perception).

1.2 Problem-1

(Simple MLP) Please go through this blog on developing your first neural network: [Blog] and understand the code.

1. (A) Draw the model architecture by showing each perceptron,input/output/hidden layers.
2. (B) Change the code to use only 60%, 70%, and 80% data as training, and report test-set performance for all these three training data set sizes.
3. (C) Compare BCE loss with MSE loss.
4. (D) Change the number of hidden layers from 2 to 4 and compare the performance.

Solution 1:

```
# -*- coding: utf-8 -*-
"""Untitled5.ipynb

Automatically generated by Colaboratory.

Original file is located at
    https://colab.research.google.com/drive/1
    ↪ RPbrLSJgkg9r1bPcAGKDAHRcbmkg8kU0
"""

from google.colab import files
files = files.upload()

#sir code
import numpy as np
import torch
import torch.nn as nn
import torch.optim as optim

# load the dataset, split into input (X) and output (y) variables
dataset = np.loadtxt('pima-indians-diabetes.csv', delimiter=',')
```

```python
X = dataset[:,0:8]
y = dataset[:,8]

X = torch.tensor(X, dtype=torch.float32)
y = torch.tensor(y, dtype=torch.float32).reshape(-1, 1)

# define the model
class PimaClassifier(nn.Module):
    def __init__(self):
        super().__init__()
        self.hidden1 = nn.Linear(8, 12)
        self.act1 = nn.ReLU()
        self.hidden2 = nn.Linear(12, 8)
        self.act2 = nn.ReLU()
        self.output = nn.Linear(8, 1)
        self.act_output = nn.Sigmoid()

    def forward(self, x):
        x = self.act1(self.hidden1(x))
        x = self.act2(self.hidden2(x))
        x = self.act_output(self.output(x))
        return x

model = PimaClassifier()
print(model)

# train the model
loss_fn   = nn.BCELoss()  # binary cross entropy
optimizer = optim.Adam(model.parameters(), lr=0.001)

n_epochs = 100
batch_size = 10

for epoch in range(n_epochs):
    for i in range(0, len(X), batch_size):
        Xbatch = X[i:i+batch_size]
        y_pred = model(Xbatch)
        ybatch = y[i:i+batch_size]
        loss = loss_fn(y_pred, ybatch)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

# compute accuracy
y_pred = model(X)
accuracy = (y_pred.round() == y).float().mean()
print(f"Accuracy {accuracy}")

# make class predictions with the model
predictions = (model(X) > 0.5).int()
for i in range(5):
    print('%s => %d (expected %d)' % (X[i].tolist(), predictions[i], y[i]))
```

```
#q1 b
import numpy as np
import torch
import torch.nn as nn
import torch.optim as optim

# Load the dataset, split into input (X) and output (y) variables
dataset = np.loadtxt('pima-indians-diabetes.csv', delimiter=',')
X = dataset[:, 0:8]
y = dataset[:, 8]

X = torch.tensor(X, dtype=torch.float32)
y = torch.tensor(y, dtype=torch.float32).reshape(-1, 1)

# Iterate over different percentages (60%, 70%, and 80%)
for num in [0.6, 0.7, 0.8]:
    full_data = len(X)
    train_size = int(num* full_data)

    X_train = X[:train_size]
    y_train = y[:train_size]
    X_test = X[train_size:]
    y_test = y[train_size:]

    # Define the model
    class PimaClassifier(nn.Module):
        def __init__(self):
          super().__init__()
          self.hidden1 = nn.Linear(8, 12)
          self.act1 = nn.ReLU()
          self.hidden2 = nn.Linear(12, 8)
          self.act2 = nn.ReLU()
          self.output = nn.Linear(8, 1)
          self.act_output = nn.Sigmoid()

        def forward(self, x):
          x = self.act1(self.hidden1(x))
          x = self.act2(self.hidden2(x))
          x = self.act_output(self.output(x))
          return x

    model = PimaClassifier()
    #print(model)

    # Train the model
    loss_fn = nn.BCELoss()  # Binary Cross-Entropy
    optimizer = optim.Adam(model.parameters(), lr=0.001)

    n_epochs = 100
    batch_size = 10

    for epoch in range(n_epochs):
        for i in range(0, train_size, batch_size):
            Xbatch = X_train[i:i+batch_size]
```

```
            y_pred = model(Xbatch)
            ybatch = y_train[i:i+batch_size]
            loss = loss_fn(y_pred, ybatch)
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()

    # Compute accuracy on the test set
    y_pred_test = model(X_test)
    accuracy = (y_pred_test.round() == y_test).float().mean()
    print(f"Accuracy for  {int(num*100)}%  of the dataset as Test Data is: {accuracy}")

#q1 c
from google.colab import files
files = files.upload()
import numpy as np
import torch
import torch.nn as nn
import torch.optim as optim

dataset = np.loadtxt('pima-indians-diabetes.csv', delimiter=',')
X = dataset[:, 0:8]
y = dataset[:, 8]

X = torch.tensor(X, dtype=torch.float32)
y = torch.tensor(y, dtype=torch.float32).reshape(-1, 1)

class PimaClassifier(nn.Module):
    def __init__(self):
        super().__init__()
        self.hidden1 = nn.Linear(8, 12)
        self.act1 = nn.ReLU()
        self.hidden2 = nn.Linear(12, 8)
        self.act2 = nn.ReLU()
        self.output = nn.Linear(8, 1)
        self.act_output = nn.Sigmoid()

    def forward(self, x):
        x = self.act1(self.hidden1(x))
        x = self.act2(self.hidden2(x))
        x = self.act_output(self.output(x))
        return x

model = PimaClassifier()
print(model)

# Step 1: Train the model with BCE loss with learning rate 0.001
bce_loss_function = nn.BCELoss()
bce_optimizer = optim.Adam(model.parameters(), lr=0.001)

#Step 2: Train the model with MSE loss with learning rate 0.001
mse_loss_function = nn.MSELoss()
mse_optimizer = optim.Adam(model.parameters(), lr=0.001)
```

```

n_epochs = 100
batch_size = 10

for epoch in range(n_epochs):
    for i in range(0, len(X), batch_size):
        Xbatch = X[i:i+batch_size]
        ybatch = y[i:i+batch_size]

        # Train with BCE loss
        bce_optimizer.zero_grad()
        y_pred_bce = model(Xbatch)
        bce_loss = bce_loss_function(y_pred_bce, ybatch)
        bce_loss.backward()
        bce_optimizer.step()

        # Train with MSE loss
        mse_optimizer.zero_grad()
        y_pred_mse = model(Xbatch)
        mse_loss = mse_loss_function(y_pred_mse, ybatch)
        mse_loss.backward()
        mse_optimizer.step()

# Compute accuracy with BCE loss
y_pred_bce = model(X)
accuracy_bce = ((y_pred_bce.round() == y).float().mean()).item()
print(f"Accuracy with BCE Loss: {accuracy_bce}")

# Compute accuracy with MSE loss
accuracy_mse = ((y_pred_mse.round() == ybatch).float().mean()).item()
print(f"Accuracy with MSE Loss: {accuracy_mse}")

# Print final BCE and MSE losses
print(f"Final BCE Loss: {bce_loss.item()}")
print(f"Final MSE Loss: {mse_loss.item()}")
```

1.3 Problem-2

Develop a neural network that works for MNIST handwritten digit classi- fication.

1. (A) Draw the model architecture.
2. (B) Compare its performance with KNN classification.

Solution 2:

```
# -*- coding: utf-8 -*-
"""Assignment4.ipynb

Automatically generated by Colaboratory.

Original file is located at
    https://colab.research.google.com/drive/1
      ↪ aCwkbEDKi_fA0JzMVyK9X27sOTtsRH-3
```

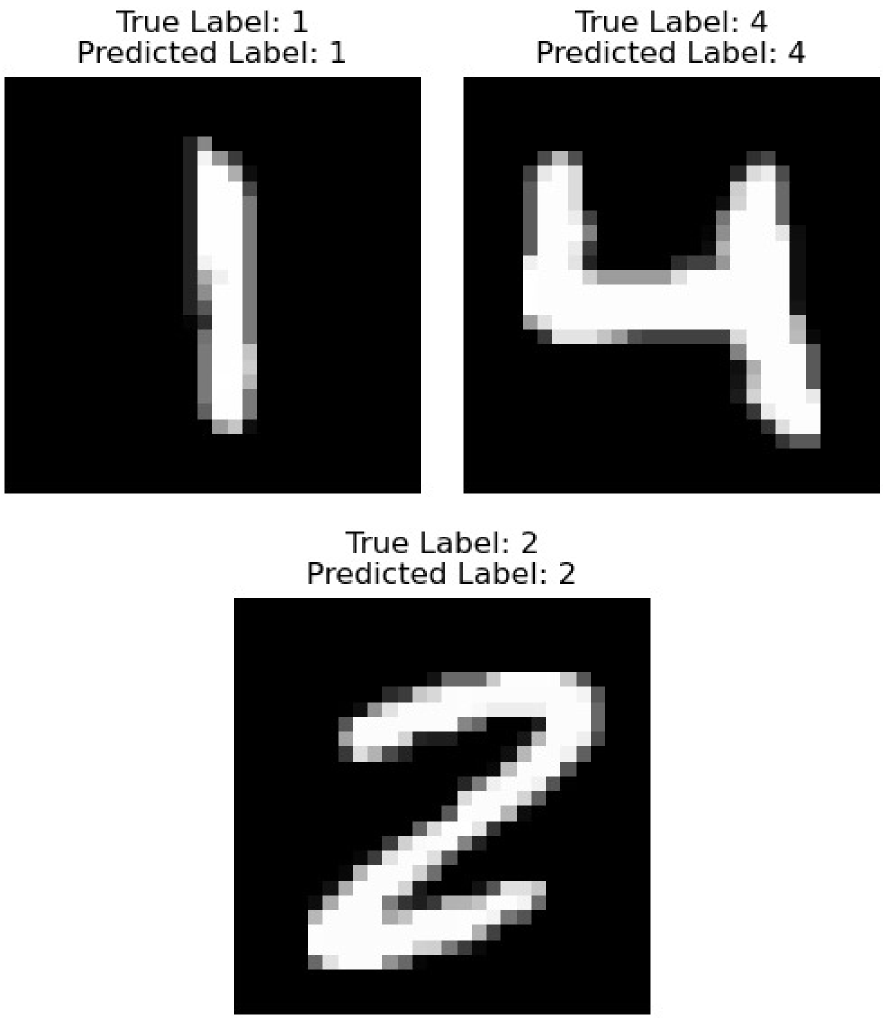
```python
"""

import tensorflow as tf
from tensorflow.keras import layers, models
from tensorflow.keras.datasets import mnist

# Load and preprocess the MNIST dataset
(train_images, train_labels), (test_images, test_labels) = mnist.load_data()

# Normalize pixel values to be in the range [0, 1]
train_images, test_images = train_images / 255.0, test_images / 255.0

# Define the neural network model
model = models.Sequential()

# Input layer: Flatten the 28x28 images into a vector
model.add(layers.Flatten(input_shape=(28, 28)))

# Hidden layers
model.add(layers.Dense(128, activation='relu'))  # 128 neurons, ReLU activation
model.add(layers.Dropout(0.2))  # Dropout for regularization
model.add(layers.Dense(64, activation='relu'))   # 64 neurons, ReLU activation

# Output layer: 10 neurons for 10 classes (digits 0-9) with softmax activation
model.add(layers.Dense(10, activation='softmax'))

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

# Train the model
model.fit(train_images, train_labels, epochs=5, batch_size=64, validation_split=0.2)

# Evaluate the model on the test data
test_loss, test_acc = model.evaluate(test_images, test_labels)
print(f"Test accuracy: {test_acc * 100:.2f}%")

import numpy as np
import matplotlib.pyplot as plt

# Select three random test images
num_samples = test_images.shape[0]
random_indices = np.random.choice(num_samples, 3, replace=False)
selected_images = test_images[random_indices]
selected_labels = test_labels[random_indices]

# Make predictions on the selected test images
predictions = model.predict(selected_images)
```

```

# Display the images and model predictions
for i in range(3):
    plt.figure(figsize=(3, 3))
    plt.imshow(selected_images[i], cmap='gray')
    plt.title(f"True Label: {selected_labels[i]}\nPredicted Label: {np.
        ↪ argmax(predictions[i])}")
    plt.axis('off')
    plt.show()



"""Diagram of the model architecture for the neural network

Input (28x28)
   |
Flatten
   |
   |--- Dense (128, ReLU)
   |    |
   |    |--- Dropout (0.2)
   |    |
   |--- Dense (64, ReLU)
   |    |
   |--- Dense (10, Softmax)


In this diagram:

Input (28x28)" represents the input layer with 28x28 pixels for each image
   ↪ .
Flatten" is used to flatten the 2D image into a 1D vector.
Dense (128, ReLU)" represents the first hidden layer with 128 neurons and
   ↪ ReLU activation.
Dropout (0.2)" is a dropout layer with a dropout rate of 0.2 for
   ↪ regularization.
Dense (64, ReLU)" is the second hidden layer with 64 neurons and ReLU
   ↪ activation.
Dense (10, Softmax)" is the output layer with 10 neurons for classifying
   ↪ digits 0-9 using softmax activation.
"""



"""B) Comparing Neural Network Performance with K-Nearest Neighbors (KNN):

To compare the performance of the neural network with K-Nearest Neighbors
   ↪ (KNN) classification on the MNIST handwritten digit classification
   ↪ task, we need to consider various parameters and settings for both
   ↪ approaches.

Neural Network (NN) Parameters and Settings:

Architecture:The neural network architecture consists of an input layer
```

```
(784 neurons), two hidden layers (128 neurons with ReLU activation and 64 neurons with ReLU activation), and an output layer (10 neurons with softmax activation).

Training: The model was trained with the following settings:

Loss Function: Sparse Categorical Cross-Entropy
Optimizer: Adam
Number of Epochs: 5
Batch Size: 64
Validation Split: 20% of the training data
Regularization: Dropout with a rate of 0.2 was applied to the second hidden layer for regularization.

K-Nearest Neighbors (KNN) Parameters and Settings:

Number of Neighbors (k): This is a critical hyperparameter for KNN. We would need to experiment with different values of k to find the optimal one.

Distance Metric: The choice of distance metric, such as Euclidean distance or Manhattan distance, can affect KNN's performance.

Data Preprocessing: It's essential to preprocess the data similarly to the neural network approach. We normalized the pixel values to be in the range [0, 1], which is a common preprocessing step for both methods.

Inference Time: KNN doesn't require training, so it's faster at inference. However, the inference time can vary depending on the size of the dataset and the chosen k value.

Memory Consumption: KNN stores the entire training dataset, which can be memory-intensive for large datasets.

Accuracy: The accuracy of KNN can vary significantly based on the choice of hyperparameters and data preprocessing.

Scalability: KNN's scalability can be an issue for large datasets, as it requires computing distances to all training examples.

Comparing the performance of NN and KNN would involve training the KNN model with various values of k and comparing their accuracy on the same MNIST test dataset. Typically, a well-tuned neural network would achieve higher accuracy on MNIST than KNN, but KNN can serve as a baseline or be suitable for simpler classification tasks.
"""
```

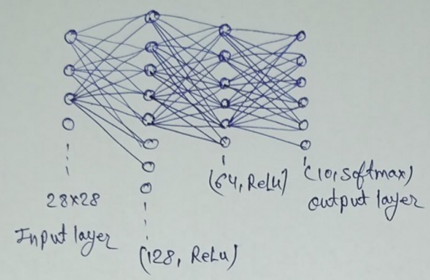
```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz
11490434/11490434 [==============================] - 11s 1us/step
Epoch 1/5
750/750 [==============================] - 2s 2ms/step - loss: 0.3715 - accuracy: 0.8899 - val_loss: 0.1776 - val_accuracy: 0.9490
Epoch 2/5
```

```
750/750 [==============================] - 1s 1ms/step - loss: 0.1637 - accuracy: 0.9512 - val_loss: 0.1243 - val_accuracy: 0.9619
Epoch 3/5
750/750 [==============================] - 1s 1ms/step - loss: 0.1202 - accuracy: 0.9630 - val_loss: 0.1042 - val_accuracy: 0.9689
Epoch 4/5
750/750 [==============================] - 1s 2ms/step - loss: 0.0988 - accuracy: 0.9694 - val_loss: 0.0905 - val_accuracy: 0.9723
Epoch 5/5
750/750 [==============================] - 1s 2ms/step - loss: 0.0839 - accuracy: 0.9735 - val_loss: 0.0888 - val_accuracy: 0.9726
313/313 [==============================] - 0s 990us/step - loss: 0.0857 - accuracy: 0.9722
Test accuracy: 97.22%
```



Name - Ganesh Kumar Nayal
Roll no - M23MEA004

Ans⇒ (2) (a) Architecture of model.



Line diagram of Architecture of model

<u>Input</u> (28x28)

```
Input (28x28)
  |
 Flatten
  |-- Dense (128, ReLU)
  |      |
  |      |-- Dropout (0.2)
  |      |
  |---- Dense (64, ReLU)
  |       |
  |-- --  Dense (10, softmax)
  |       output layer
```