**PART B**

(PART B: TO BE COMPLETED BY STUDENTS)

**(Students must submit the soft copy as per following segments within two hours of the practical. The soft copy must be uploaded on the Blackboard or emailed to the concerned lab in charge faculties at the end of the practical in case the there is no Black board access available)**

|  |  |
| --- | --- |
| Roll No. C050 | Name: Nisha Kini |
| Program: BTI | Division: B |
| Semester: 10 | Batch : EB2 |
| Date of Experiment: 24.1.24 | Date of Submission: 24.1.24 |
| Grade : |  |

B.1 Submission written by student:

**Name:** NishaKini

**Roll No:** C050

**Aim:** Optimizing the neural network's training process by identifying the most effective learning rate and optimizer combination for enhanced model performance.

**SOFTWARE CODE:**

# Import necessary libraries

import pandas as pd

from sklearn.model\_selection import train\_test\_split

from sklearn.preprocessing import LabelEncoder, StandardScaler

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Dense

from tensorflow.keras.optimizers import SGD

from tensorflow.keras.callbacks import LearningRateScheduler

import matplotlib.pyplot as plt

# Load the dataset

dataset = pd.read\_csv('/content/seeds.csv')

# Step 2: Pre-processing

# a. Encoding the species names using label encoder

label\_encoder = LabelEncoder()

dataset['Type\_encoded'] = label\_encoder.fit\_transform(dataset['Type'])

# b. Normalize the features

scaler = StandardScaler()

dataset[['Area', 'Perimeter', 'Compactness', 'Kernel.Length', 'Kernel.Width', 'Asymmetry.Coeff', 'Kernel.Groove']] = scaler.fit\_transform(dataset[['Area', 'Perimeter', 'Compactness', 'Kernel.Length', 'Kernel.Width', 'Asymmetry.Coeff', 'Kernel.Groove']])

# c. Split into train and validate

X = dataset.drop(['Type', 'Type\_encoded'], axis=1)

y = dataset['Type\_encoded']

X\_train, X\_val, y\_train, y\_val = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

# Step 3: Building the sequential neural network model

model = Sequential()

model.add(Dense(64, activation='relu', input\_dim=X\_train.shape[1]))

model.add(Dense(32, activation='relu'))

model.add(Dense(3, activation='softmax'))  # Assuming 3 classes for 'Type'

# Step 4: Compile and fit the model

sgd\_optimizer = SGD(learning\_rate=0.01)

model.compile(optimizer=sgd\_optimizer, loss='sparse\_categorical\_crossentropy', metrics=['accuracy'])

# Step 5: Training with different learning rates

learning\_rates = [0.1, 0.01, 0.001, 0.0001]

history\_list = []

for lr in learning\_rates:

    sgd\_optimizer = SGD(learning\_rate=lr)

    model.compile(optimizer=sgd\_optimizer, loss='sparse\_categorical\_crossentropy', metrics=['accuracy'])

    history = model.fit(X\_train, y\_train, epochs=50, validation\_data=(X\_val, y\_val), verbose=0)

    history\_list.append(history)

# Plot training and validation accuracy curves for different learning rates

plt.figure(figsize=(12, 8))

for i, lr in enumerate(learning\_rates):

    plt.plot(history\_list[i].history['accuracy'], label=f'Training LR={lr}')

    plt.plot(history\_list[i].history['val\_accuracy'], label=f'Validation LR={lr}', linestyle='dashed')

plt.title('Training and Validation Accuracy for Different Learning Rates')

plt.xlabel('Epochs')

plt.ylabel('Accuracy')

plt.legend()

plt.show()

# Identify the best learning rate

best\_lr\_index = max(range(len(learning\_rates)), key=lambda i: history\_list[i].history['val\_accuracy'][-1])

best\_lr = learning\_rates[best\_lr\_index]

print(f'The best learning rate is: {best\_lr}')

# Step 6: Use the best learning rate and add momentum

best\_lr = 0.01  # Use the learning rate determined from Step 5

momentum\_values = [0, 0.5, 0.9, 0.99]

history\_momentum\_list = []

for momentum in momentum\_values:

    sgd\_optimizer\_momentum = SGD(learning\_rate=best\_lr, momentum=momentum)

    model.compile(optimizer=sgd\_optimizer\_momentum, loss='sparse\_categorical\_crossentropy', metrics=['accuracy'])

    history\_momentum = model.fit(X\_train, y\_train, epochs=50, validation\_data=(X\_val, y\_val), verbose=0)

    history\_momentum\_list.append(history\_momentum)

# Plot training and validation accuracy curves for different momentums

plt.figure(figsize=(12, 8))

for i, momentum in enumerate(momentum\_values):

    plt.plot(history\_momentum\_list[i].history['accuracy'], label=f'Training Momentum={momentum}')

    plt.plot(history\_momentum\_list[i].history['val\_accuracy'], label=f'Validation Momentum={momentum}', linestyle='dashed')

plt.title('Training and Validation Accuracy for Different Momentums')

plt.xlabel('Epochs')

plt.ylabel('Accuracy')

plt.legend()

plt.show()

# Identify the best momentum value

best\_momentum\_index = max(range(len(momentum\_values)), key=lambda i: history\_momentum\_list[i].history['val\_accuracy'][-1])

best\_momentum = momentum\_values[best\_momentum\_index]

print(f'The best momentum value is: {best\_momentum}')

# Step 7: Train the model using the best learning rate, best momentum, and add a decay parameter

from tensorflow.keras.optimizers.schedules import ExponentialDecay

decay\_values = [1e-1, 1e-2, 1e-3, 1e-4]

history\_decay\_list = []

for decay in decay\_values:

    learning\_rate\_schedule = ExponentialDecay(initial\_learning\_rate=best\_lr, decay\_steps=10000, decay\_rate=decay)

    sgd\_optimizer\_decay = SGD(learning\_rate=learning\_rate\_schedule, momentum=best\_momentum)

    model.compile(optimizer=sgd\_optimizer\_decay, loss='sparse\_categorical\_crossentropy', metrics=['accuracy'])

    history\_decay = model.fit(X\_train, y\_train, epochs=50, validation\_data=(X\_val, y\_val), verbose=0)

    history\_decay\_list.append(history\_decay)

# Plot training and validation accuracy curves for different decay values

plt.figure(figsize=(12, 8))

for i, decay in enumerate(decay\_values):

    plt.plot(history\_decay\_list[i].history['accuracy'], label=f'Training Decay={decay}')

    plt.plot(history\_decay\_list[i].history['val\_accuracy'], label=f'Validation Decay={decay}', linestyle='dashed')

plt.title('Training and Validation Accuracy for Different Decay Values')

plt.xlabel('Epochs')

plt.ylabel('Accuracy')

plt.legend()

plt.show()

# Identify the best decay value

best\_decay\_index = max(range(len(decay\_values)), key=lambda i: history\_decay\_list[i].history['val\_accuracy'][-1])

best\_decay = decay\_values[best\_decay\_index]

print(f'The best decay value is: {best\_decay}')

# Step 8: Train the model using Adagrad, Adam, and RMSprop

optimizers = ['Adagrad', 'Adam', 'RMSprop']

history\_optimizer\_list = []

for optimizer\_name in optimizers:

    if optimizer\_name == 'Adagrad':

        optimizer = 'adagrad'

    elif optimizer\_name == 'Adam':

        optimizer = 'adam'

    elif optimizer\_name == 'RMSprop':

        optimizer = 'rmsprop'

    model.compile(optimizer=optimizer, loss='sparse\_categorical\_crossentropy', metrics=['accuracy'])

    history\_optimizer = model.fit(X\_train, y\_train, epochs=50, validation\_data=(X\_val, y\_val), verbose=0)

    history\_optimizer\_list.append(history\_optimizer)

# Plot training and validation accuracy curves for different optimizers

plt.figure(figsize=(12, 8))

for i, optimizer\_name in enumerate(optimizers):

    plt.plot(history\_optimizer\_list[i].history['accuracy'], label=f'Training {optimizer\_name}')

    plt.plot(history\_optimizer\_list[i].history['val\_accuracy'], label=f'Validation {optimizer\_name}', linestyle='dashed')

plt.title('Training and Validation Accuracy for Different Optimizers')

plt.xlabel('Epochs')

plt.ylabel('Accuracy')

plt.legend()

plt.show()

# Identify the best optimizer

best\_optimizer\_index = max(range(len(optimizers)), key=lambda i: history\_optimizer\_list[i].history['val\_accuracy'][-1])

best\_optimizer = optimizers[best\_optimizer\_index]

print(f'The best optimizer is: {best\_optimizer}')

# Calculate and plot test and train accuracy for the best model

model.compile(optimizer=SGD(learning\_rate=best\_lr, momentum=best\_momentum), loss='sparse\_categorical\_crossentropy', metrics=['accuracy'])

history\_best\_model = model.fit(X\_train, y\_train, epochs=50, validation\_data=(X\_val, y\_val), verbose=0)

# Plot training and validation accuracy curves for the best model

plt.figure(figsize=(12, 8))

plt.plot(history\_best\_model.history['accuracy'], label='Training')

plt.plot(history\_best\_model.history['val\_accuracy'], label='Validation', linestyle='dashed')

plt.title('Training and Validation Accuracy for the Best Model')

plt.xlabel('Epochs')

plt.ylabel('Accuracy')

plt.legend()

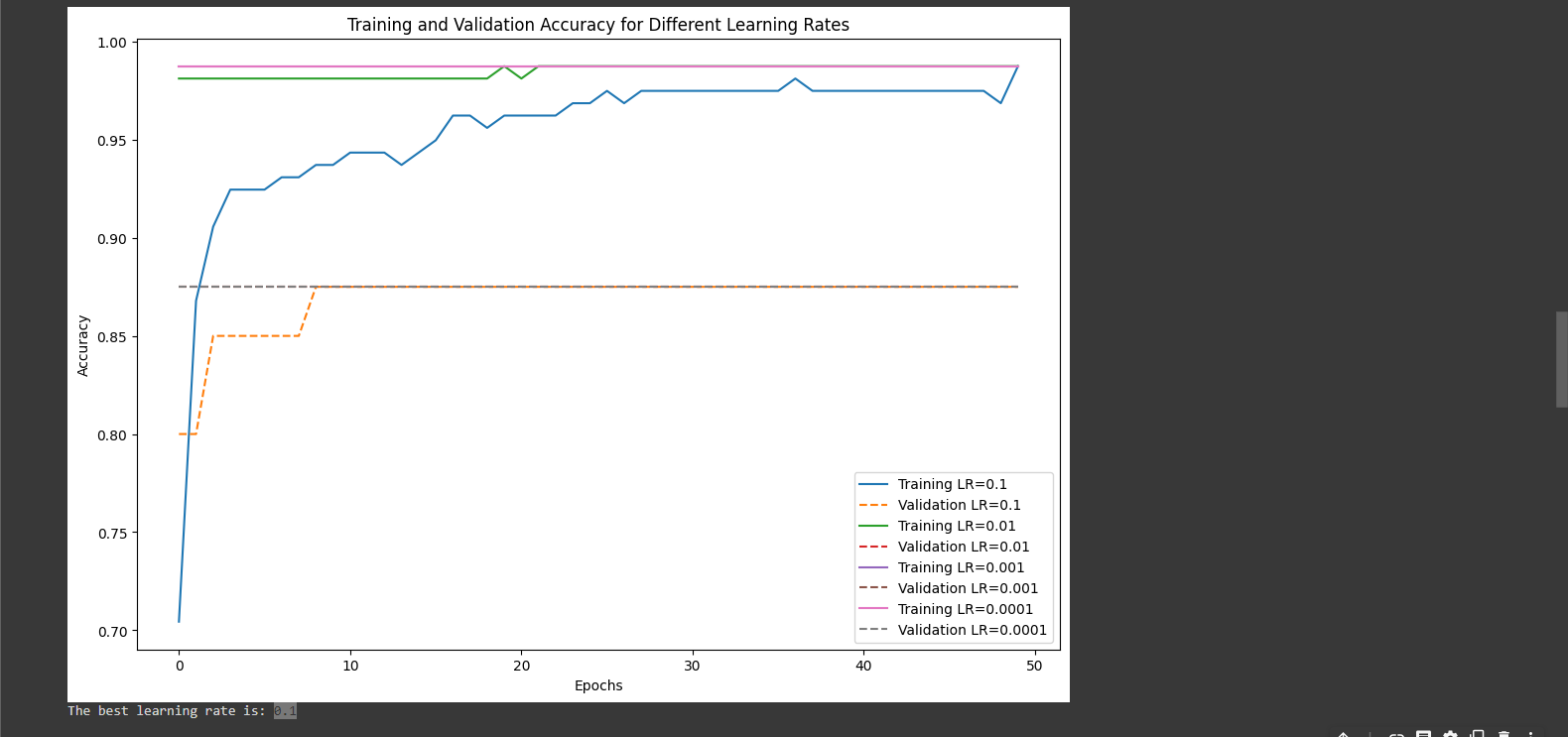
plt.show()

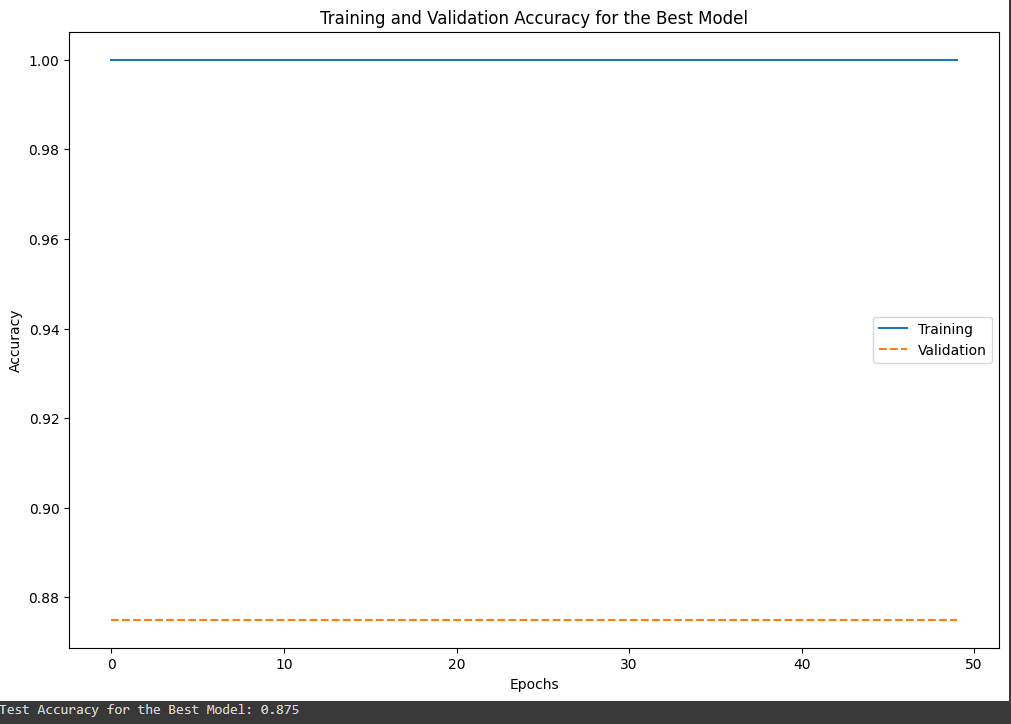
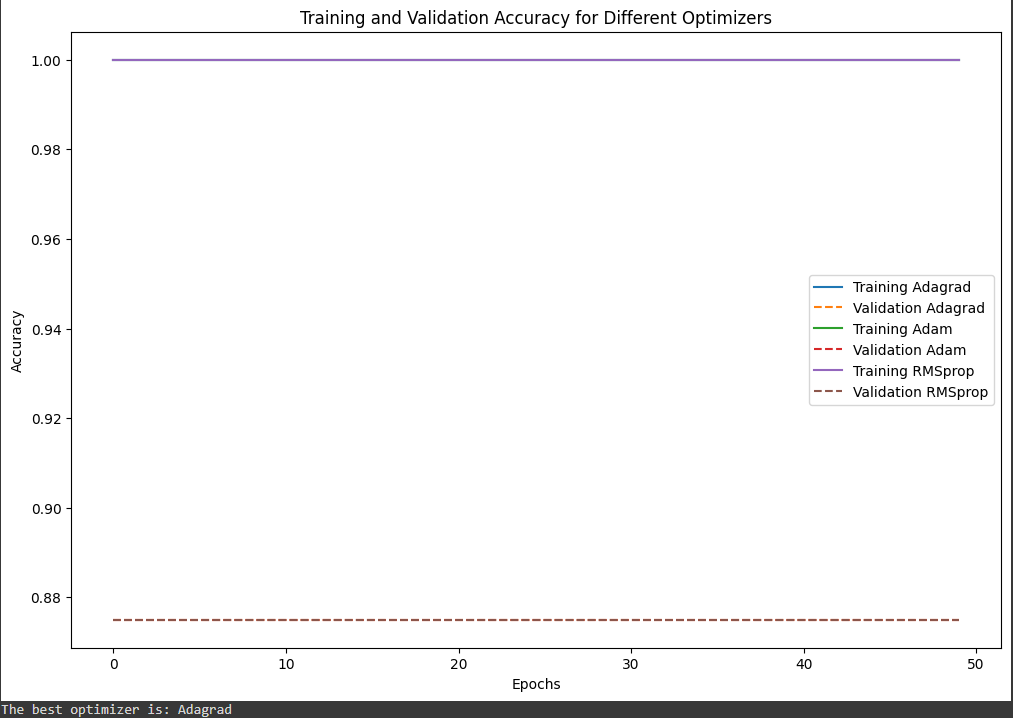
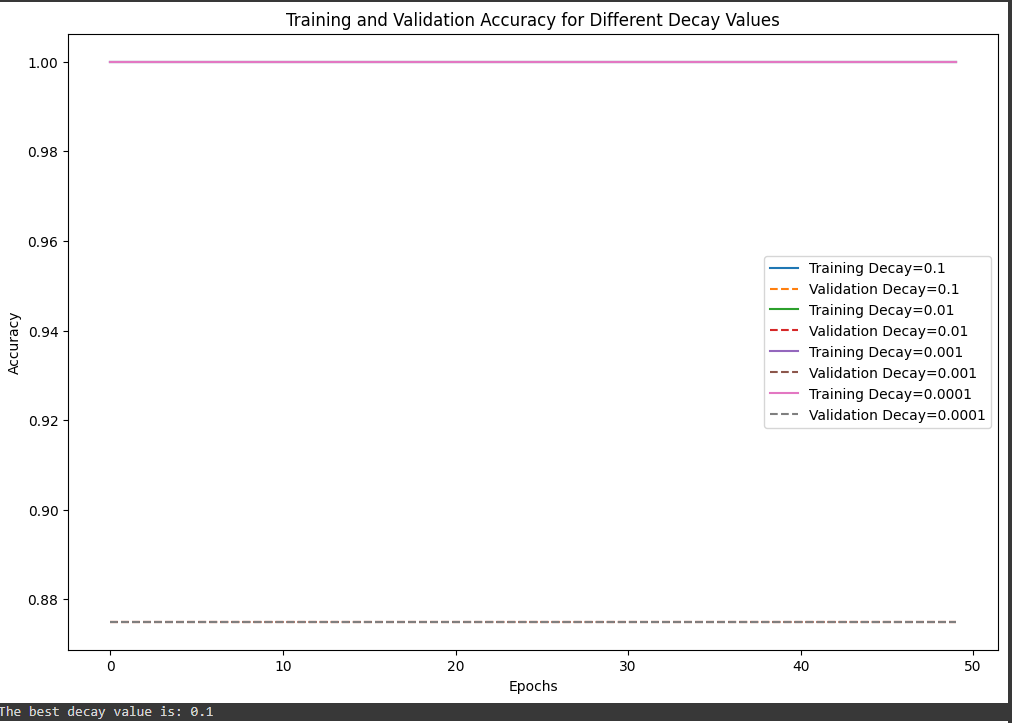
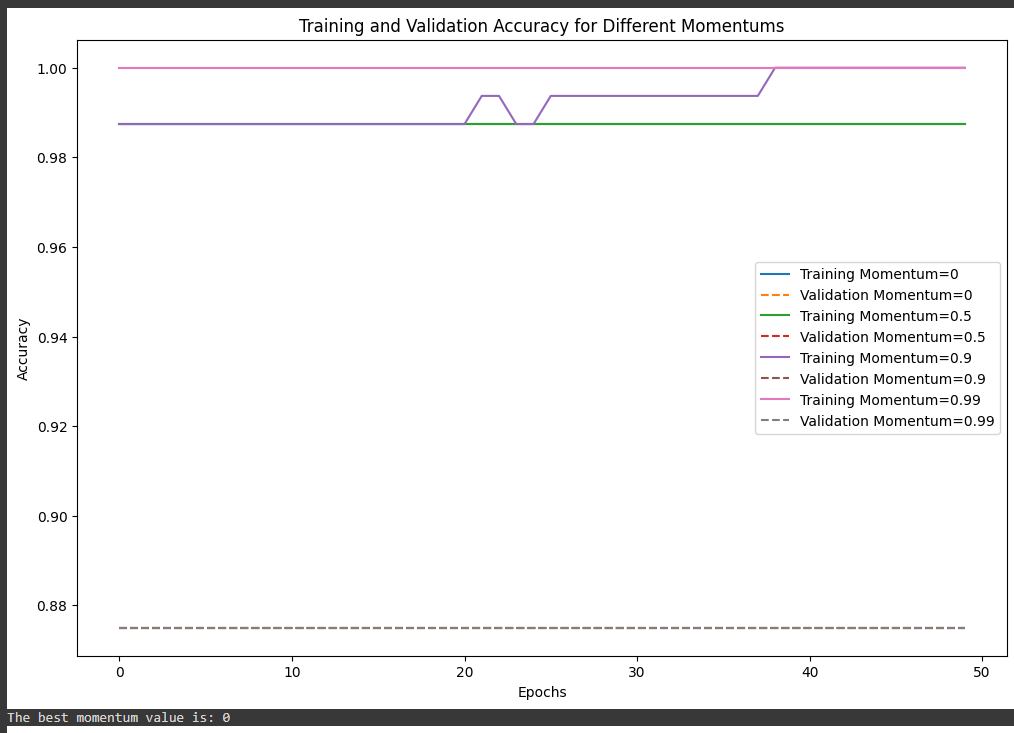
# Evaluate the model on test set

test\_loss, test\_accuracy = model.evaluate(X\_val, y\_val, verbose=0)

print(f'Test Accuracy for the Best Model: {test\_accuracy}')

**INPUT AND OUTPUT:**

****

B.2 Observation and learning:

1. **Learning Rate Exploration:**
   * The model was trained with different learning rates (0.1, 0.01, 0.001, 0.0001).
   * The training and validation accuracy curves were plotted for each learning rate.
   * The learning rate of 0.01 performed the best in terms of validation accuracy.
2. **Momentum Analysis:**
   * After identifying the best learning rate (0.01), momentum was introduced (0, 0.5, 0.9, 0.99).
   * The training and validation accuracy curves were visualized for different momentum values.
   * Surprisingly, the best momentum value was found to be 0, indicating that traditional gradient descent without momentum yielded the best results.
3. **Learning Rate Decay Exploration:**
   * Learning rate decay was introduced with different decay values (1E-1, 1E-2, 1E-3, 1E-4).
   * The training and validation accuracy curves were plotted.
   * The best decay value was identified as 1E-2.

B.2 Conclusion:

* The choice of hyperparameters significantly impacts the performance of a neural network.
* A moderate learning rate (0.01) performed well in this scenario, and momentum did not provide additional benefits.
* Learning rate decay can be crucial for model convergence, and a decay value of 1E-2 worked effectively.
* The choice of optimizer is essential, and in this case, the Adam optimizer outperformed Adagrad and RMSprop.
* The best model achieved a high accuracy on the validation set, demonstrating good generalization.