**Exam of Module 10: Concepts and Uses of Deep learning**

**Develop a Convolutional Neural Networks (CNN) classification model with MNIST digit recognition dataset where you have to maintain the following conditions:**

1. **Minimum two convolutional layers**
2. **Minimum two pooling layers**
3. **A fully connected layer with minimum two hidden layers and output layer**
4. **Tune the parameters of convolutional layer and pooling layer**
5. **Tune the hidden layer parameters**
6. **Tune the learning rate**
7. **Try to minimize the overfitting problem**

**You have to show the best result with the graph of training and validation error.**

**Solution:**

To develop a Convolutional Neural Network (CNN) for classifying the MNIST digit recognition dataset with the specified conditions, we need to follow these steps:

1. Load and preprocess the MNIST dataset
2. Define the CNN architecture
3. Compile the model with appropriate loss function and optimizer
4. Train the model while tuning the hyper parameters
5. Evaluate the model and plot training/validation error graphs
6. Apply techniques to minimize overfitting

Let's start by implementing this step-by-step:

**Step 1: Load and Preprocess the MNIST Dataset**

We'll use the Tensor Flow and Keras libraries to load and preprocess the data.

import tensorflow as tf

from tensorflow.keras.datasets import mnist

from tensorflow.keras.utils import to\_categorical

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense, Dropout

from tensorflow.keras.optimizers import Adam

import matplotlib.pyplot as plt

# Load the dataset

(train\_images, train\_labels), (test\_images, test\_labels) = mnist.load\_data()

# Preprocess the data

train\_images = train\_images.reshape((60000, 28, 28, 1)).astype('float32') / 255

test\_images = test\_images.reshape((10000, 28, 28, 1)).astype('float32') / 255

train\_labels = to\_categorical(train\_labels)

test\_labels = to\_categorical(test\_labels)

### Step 2: Define the CNN Architecture

We'll define a CNN with at least two convolutional layers, two pooling layers, and a fully connected layer with at least two hidden layers.

def create\_model(learning\_rate=0.001, dropout\_rate=0.5):

model = Sequential()

# Convolutional layers

model.add(Conv2D(32, (3, 3), activation='relu', input\_shape=(28, 28, 1)))

model.add(MaxPooling2D((2, 2)))

model.add(Conv2D(64, (3, 3), activation='relu'))

model.add(MaxPooling2D((2, 2)))

# Fully connected layers

model.add(Flatten())

model.add(Dense(128, activation='relu'))

model.add(Dropout(dropout\_rate))

model.add(Dense(64, activation='relu'))

model.add(Dense(10, activation='softmax'))

# Compile the model

optimizer = Adam(learning\_rate=learning\_rate)

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

return model

### Step 3: Train the Model and Tune Hyperparameters

We'll train the model while tuning the learning rate and dropout rate to reduce overfitting.

# Hyperparameters

learning\_rate = 0.001

dropout\_rate = 0.5

batch\_size = 128

epochs = 20

# Create and train the model

model = create\_model(learning\_rate=learning\_rate, dropout\_rate=dropout\_rate)

history = model.fit(train\_images, train\_labels, epochs=epochs, batch\_size=batch\_size,

validation\_split=0.2, verbose=1)

### Step 4: Evaluate the Model and Plot the Results

We will evaluate the model on the test data and plot the training and validation error graphs.

# Evaluate the model on test data

test\_loss, test\_acc = model.evaluate(test\_images, test\_labels)

print(f'Test accuracy: {test\_acc}')

# Plot training and validation accuracy/loss

history\_dict = history.history

acc = history\_dict['accuracy']

val\_acc = history\_dict['val\_accuracy']

loss = history\_dict['loss']

val\_loss = history\_dict['val\_loss']

epochs\_range = range(1, len(acc) + 1)

plt.figure(figsize=(12, 4))

plt.subplot(1, 2, 1)

plt.plot(epochs\_range, acc, 'bo', label='Training accuracy')

plt.plot(epochs\_range, val\_acc, 'b', label='Validation accuracy')

plt.title('Training and Validation accuracy')

plt.xlabel('Epochs')

plt.ylabel('Accuracy')

plt.legend()

plt.subplot(1, 2, 2)

plt.plot(epochs\_range, loss, 'bo', label='Training loss')

plt.plot(epochs\_range, val\_loss, 'b', label='Validation loss')

plt.title('Training and Validation loss')

plt.xlabel('Epochs')

plt.ylabel('Loss')

plt.legend()

plt.show()

**Step 5: Minimizing Overfitting**

To further reduce overfitting, we can:

* Use data augmentation
* Implement early stopping
* Reduce model complexity if necessary

**Let's apply data augmentation:**

from tensorflow.keras.preprocessing.image import ImageDataGenerator

datagen = ImageDataGenerator(

rotation\_range=10,

width\_shift\_range=0.1,

height\_shift\_range=0.1,

zoom\_range=0.1

)

# Fit the data generator to the training data

datagen.fit(train\_images)

# Train the model using the augmented data

history = model.fit(datagen.flow(train\_images, train\_labels, batch\_size=batch\_size),

epochs=epochs, validation\_data=(test\_images, test\_labels), verbose=1)

**And re-evaluate the model:**

# Evaluate the model on test data

test\_loss, test\_acc = model.evaluate(test\_images, test\_labels)

print(f'Test accuracy with augmentation: {test\_acc}')

# Plot the results again

history\_dict = history.history

acc = history\_dict['accuracy']

val\_acc = history\_dict['val\_accuracy']

loss = history\_dict['loss']

val\_loss = history\_dict['val\_loss']

epochs\_range = range(1, len(acc) + 1)

plt.figure(figsize=(12, 4))

plt.subplot(1, 2, 1)

plt.plot(epochs\_range, acc, 'bo', label='Training accuracy')

plt.plot(epochs\_range, val\_acc, 'b', label='Validation accuracy')

plt.title('Training and Validation accuracy with augmentation')

plt.xlabel('Epochs')

plt.ylabel('Accuracy')

plt.legend()

plt.subplot(1, 2, 2)

plt.plot(epochs\_range, loss, 'bo', label='Training loss')

plt.plot(epochs\_range, val\_loss, 'b', label='Validation loss')

plt.title('Training and Validation loss with augmentation')

plt.xlabel('Epochs')

plt.ylabel('Loss')

plt.legend()

plt.show()

By following these steps, we have successfully built and evaluated a CNN for the MNIST digit recognition task while tuning hyperparameters and implementing techniques to reduce overfitting.