**1. Deep Learning**

**a. Build a DNN with Five Hidden Layers of 100 Neurons Each, He Initialization, and ELU Activation Function**

Here’s how you can build a DNN in TensorFlow/Keras with five hidden layers of 100 neurons each, using He initialization and ELU activation:

import tensorflow as tf

from tensorflow.keras import layers, models

# Build the model

model = models.Sequential()

# Input layer (flatten MNIST images)

model.add(layers.Flatten(input\_shape=(28, 28)))

# Hidden layers with He initialization and ELU activation

for \_ in range(5):

model.add(layers.Dense(100, kernel\_initializer='he\_normal', activation='elu'))

# Output layer (softmax with 5 neurons for digits 0-4)

model.add(layers.Dense(5, activation='softmax'))

model.summary()

**b. Training on MNIST (Digits 0-4) with Adam Optimization and Early Stopping**

Here’s the code for training on digits 0-4, using Adam optimizer, early stopping, and saving checkpoints:

from tensorflow.keras.callbacks import EarlyStopping, ModelCheckpoint

from tensorflow.keras.optimizers import Adam

from tensorflow.keras.datasets import mnist

import numpy as np

# Load MNIST data

(x\_train, y\_train), (x\_test, y\_test) = mnist.load\_data()

# Select digits 0-4

train\_filter = np.isin(y\_train, [0, 1, 2, 3, 4])

x\_train, y\_train = x\_train[train\_filter], y\_train[train\_filter]

test\_filter = np.isin(y\_test, [0, 1, 2, 3, 4])

x\_test, y\_test = x\_test[test\_filter], y\_test[test\_filter]

# Normalize images

x\_train, x\_test = x\_train / 255.0, x\_test / 255.0

# Compile model

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

# Set callbacks for early stopping and model checkpoint

early\_stopping = EarlyStopping(monitor='val\_loss', patience=3)

checkpoint = ModelCheckpoint('model\_checkpoint.h5', save\_best\_only=True)

# Train the model

model.fit(x\_train, y\_train, epochs=20, batch\_size=32, validation\_data=(x\_test, y\_test),

callbacks=[early\_stopping, checkpoint])

**c. Hyperparameter Tuning with Cross-Validation**

For hyperparameter tuning, you can use **GridSearchCV** or **RandomizedSearchCV** from sklearn along with KerasClassifier to perform cross-validation. You can tune parameters such as learning rate, batch size, and number of neurons.

from sklearn.model\_selection import GridSearchCV

from tensorflow.keras.wrappers.scikit\_learn import KerasClassifier

def create\_model(learning\_rate=0.001, batch\_size=32):

model = models.Sequential()

model.add(layers.Flatten(input\_shape=(28, 28)))

for \_ in range(5):

model.add(layers.Dense(100, kernel\_initializer='he\_normal', activation='elu'))

model.add(layers.Dense(5, activation='softmax'))

model.compile(optimizer=Adam(learning\_rate=learning\_rate),

loss='sparse\_categorical\_crossentropy', metrics=['accuracy'])

return model

model = KerasClassifier(build\_fn=create\_model)

param\_grid = {

'learning\_rate': [0.001, 0.01],

'batch\_size': [32, 64],

'epochs': [10, 20]

}

grid = GridSearchCV(estimator=model, param\_grid=param\_grid, n\_jobs=-1, cv=3)

grid\_result = grid.fit(x\_train, y\_train)

print(f"Best Params: {grid\_result.best\_params\_}")

**d. Adding Batch Normalization and Comparing Learning Curves**

To add **Batch Normalization** after each hidden layer, update the model as follows:

for \_ in range(5):

model.add(layers.Dense(100, kernel\_initializer='he\_normal', activation=None))

model.add(layers.BatchNormalization())

model.add(layers.ELU())

You can then compare the learning curves by training models with and without batch normalization and plotting the training/validation loss or accuracy over time using **Matplotlib**.

**e. Overfitting and Adding Dropout**

To add **dropout**, update the model as follows:

for \_ in range(5):

model.add(layers.Dense(100, kernel\_initializer='he\_normal', activation=None))

model.add(layers.BatchNormalization())

model.add(layers.ELU())

model.add(layers.Dropout(0.5)) # Dropout after each hidden layer

You can experiment with different dropout rates (e.g., 0.3, 0.5) to see how it helps reduce overfitting.

**2. Transfer Learning**

**a. Reusing Pretrained Layers and Freezing Them**

To reuse the pretrained layers and freeze them, you can do this:

for layer in model.layers[:-1]: # Freeze all layers except the output layer

layer.trainable = False

# Create new output layer

output\_layer = layers.Dense(5, activation='softmax')(model.layers[-2].output)

new\_model = models.Model(inputs=model.input, outputs=output\_layer)

**b. Training on Digits 5-9 with 100 Images per Digit**

# Select digits 5-9

train\_filter = np.isin(y\_train, [5, 6, 7, 8, 9])

x\_train, y\_train = x\_train[train\_filter], y\_train[train\_filter]

# Train the model on the new task

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

new\_model.fit(x\_train, y\_train, epochs=10, batch\_size=32)

**c. Caching Frozen Layers for Faster Training**

When caching the frozen layers, the model will not update them during training, which can speed up the training process.

# Cache the frozen layers for faster training

new\_model.fit(x\_train, y\_train, epochs=10, batch\_size=32, use\_multiprocessing=True)

**d. Reusing Four Hidden Layers and Testing Precision**

# Reuse the first four hidden layers and fine-tune the last layer

new\_model.layers[-5].trainable = False # Freeze the last layer

new\_model.fit(x\_train, y\_train, epochs=10, batch\_size=32)

**e. Unfreezing the Top Two Layers for Fine-Tuning**

Unfreeze the top two layers and train the model again:

new\_model.layers[-6].trainable = True # Unfreeze the second last layer

new\_model.layers[-7].trainable = True # Unfreeze the last layer

new\_model.fit(x\_train, y\_train, epochs=10, batch\_size=32)

**3. Pretraining on an Auxiliary Task**

**a. Building DNNs A and B for Image Pair Comparison**

For DNNs A and B:

def build\_dnn():

model = models.Sequential()

model.add(layers.Flatten(input\_shape=(28, 28)))

for \_ in range(5):

model.add(layers.Dense(100, kernel\_initializer='he\_normal', activation='elu'))

return model

dnn\_a = build\_dnn()

dnn\_b = build\_dnn()

# Concatenate outputs of DNN A and B

concatenated = layers.concatenate([dnn\_a.output, dnn\_b.output], axis=1)

# Add a final hidden layer and output layer

output\_layer = layers.Dense(10, activation='elu')(concatenated)

output = layers.Dense(1, activation='sigmoid')(output\_layer)

# Define the full model

model = models.Model(inputs=[dnn\_a.input, dnn\_b.input], outputs=output)

**b. Creating Pairs of MNIST Images for Training**

To create pairs of images, create a function like:

def create\_pairs(x, y, batch\_size):

pairs = []

labels = []

for \_ in range(batch\_size):

# Create pairs of the same class and different classes

same\_class = np.random.choice(len(x)) # Same class pair

diff\_class = np.random.choice(len(x)) # Different class pair

pairs.append((x[same\_class], x[diff\_class]))

labels.append(0 if y[same\_class] == y[diff\_class] else 1)

return np.array(pairs), np.array(labels)

**c. Training on Pairs**

pairs, labels = create\_pairs(x\_train, y\_train, batch\_size=32)

model.fit([pairs[:, 0], pairs[:, 1]], labels, epochs=10, batch\_size=32)

**d. Fine-Tuning the Pretrained DNN on a Small Dataset**

# Reuse DNN A layers for the new classifier

dnn\_a.trainable = False # Freeze DNN A layers

output\_layer = layers.Dense(10, activation='softmax')(dnn\_a.output)

new\_model = models.Model(inputs=dnn\_a.input, outputs=output\_layer)

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

# Train on split #2

new\_model.fit(x\_train\_split\_2, y\_train\_split\_2, epochs=10, batch\_size=32)

ss