1. Deep Learning.
   1. Build a DNN with five hidden layers of 100 neurons each, He initialization, and the ELU activation function.

Answer:- Certainly! Here’s a step-by-step guide to building a deep neural network (DNN) with five hidden layers, each with 100 neurons, using He initialization and the ELU activation function. This example will use TensorFlow and Keras:

1. Import Required Libraries

import tensorflow as tf

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Dense

from tensorflow.keras.initializers import HeNormal

2. **Build the Model**

# Create a sequential model

model = Sequential()

# Input layer

model.add(Dense(100, activation='elu', kernel\_initializer=HeNormal(), input\_shape=(input\_dim,)))

# Hidden layers

for \_ in range(4): # Adding 4 more hidden layers (total of 5 hidden layers)

model.add(Dense(100, activation='elu', kernel\_initializer=HeNormal()))

# Output layer (assuming a classification task with 10 classes)

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

# Compile the model

model.compile(optimizer='adam',

loss='sparse\_categorical\_crossentropy',

metrics=['accuracy'])

# Display the model summary

model.summary()

3. Explanation

* Dense(100, activation='elu', kernel\_initializer=HeNormal()): Adds a dense (fully connected) layer with 100 neurons, using the ELU activation function and He Normal initialization.
* input\_shape=(input\_dim,): Specifies the shape of the input data. Replace input\_dim with the actual number of features in your dataset.
* softmax in the output layer: Used for multi-class classification. If you're performing binary classification, you might use sigmoid instead and adjust the loss function accordingly.
* sparse\_categorical\_crossentropy: Loss function for multi-class classification where labels are provided as integers. If using one-hot encoded labels, you would use categorical\_crossentropy.

4. Training and Evaluation

To train and evaluate the model, you'll need to provide your dataset. Here’s a basic example of how you might fit and evaluate the model:

# Dummy data (replace with your actual dataset)

import numpy as np

# Generate some random data for demonstration

x\_train = np.random.rand(1000, input\_dim)

y\_train = np.random.randint(10, size=1000)

x\_test = np.random.rand(200, input\_dim)

y\_test = np.random.randint(10, size=200)

# Train the model

history = model.fit(x\_train, y\_train, epochs=10, batch\_size=32, validation\_data=(x\_test, y\_test))

# Evaluate the model

test\_loss, test\_accuracy = model.evaluate(x\_test, y\_test)

print(f"Test Accuracy: {test\_accuracy \* 100:.2f}%")

Replace input\_dim with the number of features in your dataset and use your actual training and testing data.

This setup provides a robust starting point for building a deep neural network with the specified architecture and initialization strategy.

* 1. Using Adam optimization and early stopping, try training it on MNIST but only on digits 0 to 4, as we will use transfer learning for digits 5 to 9 in the next exercise. You will need a softmax output layer with five neurons, and as always make sure to save checkpoints at regular intervals and save the final model so you can reuse it later.

Answer:- Here’s how you can train the model on digits 0 to 4 from the MNIST dataset using Adam optimization and early stopping. The code will also include checkpoint saving for regular intervals and saving the final model.

### 1. Import Required Libraries

import tensorflow as tf

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Dense

from tensorflow.keras.initializers import HeNormal

from tensorflow.keras.callbacks import EarlyStopping, ModelCheckpoint

from tensorflow.keras.datasets import mnist

2. **Load and Preprocess the MNIST Dataset**

# Load MNIST dataset

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

# Filter digits 0 to 4

x\_train = x\_train[y\_train < 5]

y\_train = y\_train[y\_train < 5]

x\_test = x\_test[y\_test < 5]

y\_test = y\_test[y\_test < 5]

# Normalize pixel values to [0, 1]

x\_train = x\_train.astype('float32') / 255.0

x\_test = x\_test.astype('float32') / 255.0

# Flatten images (28x28 to 784)

x\_train = x\_train.reshape(-1, 28 \* 28)

x\_test = x\_test.reshape(-1, 28 \* 28)

# Convert labels to categorical format

num\_classes = 5

y\_train = tf.keras.utils.to\_categorical(y\_train, num\_classes)

y\_test = tf.keras.utils.to\_categorical(y\_test, num\_classes)

3. **Build the Model**

# Create a sequential model

model = Sequential()

# Input layer

model.add(Dense(100, activation='elu', kernel\_initializer=HeNormal(), input\_shape=(28 \* 28,)))

# Hidden layers

for \_ in range(4): # Adding 4 more hidden layers (total of 5 hidden layers)

model.add(Dense(100, activation='elu', kernel\_initializer=HeNormal()))

# Output layer

model.add(Dense(num\_classes, activation='softmax'))

# Compile the model

model.compile(optimizer='adam',

loss='categorical\_crossentropy',

metrics=['accuracy'])

# Display the model summary

model.summary()

4. **Set Up Callbacks for Early Stopping and Checkpointing**

# Early stopping callback

early\_stopping = EarlyStopping(monitor='val\_loss', patience=3, restore\_best\_weights=True)

# Model checkpoint callback

checkpoint = ModelCheckpoint('mnist\_model\_checkpoint.h5',

save\_best\_only=True,

monitor='val\_loss')

5. **Train the Model**

# Train the model

history = model.fit(x\_train, y\_train,

epochs=20,

batch\_size=32,

validation\_split=0.2,

callbacks=[early\_stopping, checkpoint])

# Save the final model

model.save('mnist\_model\_final.h5')

6. **Evaluate the Model**

# Load the best model from checkpoint

best\_model = tf.keras.models.load\_model('mnist\_model\_checkpoint.h5')

# Evaluate the model

test\_loss, test\_accuracy = best\_model.evaluate(x\_test, y\_test)

print(f"Test Accuracy: {test\_accuracy \* 100:.2f}%")

Summary:

* Load and preprocess the data to focus on digits 0 to 4.
* Build a DNN with five hidden layers, using He initialization and ELU activation.
* Set up callbacks for early stopping and saving model checkpoints.
* Train the model while monitoring performance and saving the final trained model for reuse.

This setup will ensure that you have a robust model trained on the specified subset of MNIST and that you can later use transfer learning to handle digits 5 to 9.

* 1. Tune the hyperparameters using cross-validation and see what precision you can achieve.

Answer:- To tune hyperparameters and evaluate model performance using cross-validation, follow these steps. We'll use KFold cross-validation from scikit-learn to assess the model's performance and fine-tune hyperparameters like learning rate, batch size, and number of epochs.

Here’s a structured approach to hyperparameter tuning using cross-validation for the DNN model on digits 0 to 4 from the MNIST dataset:

### 1. Import Required Libraries

import numpy as np

from sklearn.model\_selection import KFold

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Dense

from tensorflow.keras.initializers import HeNormal

from tensorflow.keras.optimizers import Adam

from tensorflow.keras.utils import to\_categorical

from tensorflow.keras.callbacks import EarlyStopping

2. **Load and Preprocess the MNIST Dataset**

# Load MNIST dataset

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

# Filter digits 0 to 4

x\_train = x\_train[y\_train < 5]

y\_train = y\_train[y\_train < 5]

x\_test = x\_test[y\_test < 5]

y\_test = y\_test[y\_test < 5]

# Normalize pixel values to [0, 1]

x\_train = x\_train.astype('float32') / 255.0

x\_test = x\_test.astype('float32') / 255.0

# Flatten images (28x28 to 784)

x\_train = x\_train.reshape(-1, 28 \* 28)

x\_test = x\_test.reshape(-1, 28 \* 28)

# Convert labels to categorical format

num\_classes = 5

y\_train = to\_categorical(y\_train, num\_classes)

y\_test = to\_categorical(y\_test, num\_classes)

3. **Define the Model Building Function**

def build\_model(learning\_rate=0.001, num\_hidden\_units=100):

model = Sequential()

model.add(Dense(num\_hidden\_units, activation='elu', kernel\_initializer=HeNormal(), input\_shape=(28 \* 28,)))

for \_ in range(4):

model.add(Dense(num\_hidden\_units, activation='elu', kernel\_initializer=HeNormal()))

model.add(Dense(num\_classes, activation='softmax'))

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

model.compile(optimizer=optimizer,

loss='categorical\_crossentropy',

metrics=['accuracy'])

return model

4. **Cross-Validation Setup**

# Parameters to tune

learning\_rates = [0.001, 0.01]

num\_hidden\_units\_list = [50, 100]

batch\_sizes = [32, 64]

epochs = 10

# KFold cross-validation

kf = KFold(n\_splits=5, shuffle=True, random\_state=42)

# Store results

results = []

for lr in learning\_rates:

for num\_units in num\_hidden\_units\_list:

for batch\_size in batch\_sizes:

fold\_no = 1

for train\_index, val\_index in kf.split(x\_train):

x\_train\_fold, x\_val\_fold = x\_train[train\_index], x\_train[val\_index]

y\_train\_fold, y\_val\_fold = y\_train[train\_index], y\_train[val\_index]

model = build\_model(learning\_rate=lr, num\_hidden\_units=num\_units)

# Early stopping callback

early\_stopping = EarlyStopping(monitor='val\_loss', patience=3, restore\_best\_weights=True)

# Train the model

history = model.fit(x\_train\_fold, y\_train\_fold,

epochs=epochs,

batch\_size=batch\_size,

validation\_data=(x\_val\_fold, y\_val\_fold),

callbacks=[early\_stopping],

verbose=0)

# Evaluate the model

val\_loss, val\_accuracy = model.evaluate(x\_val\_fold, y\_val\_fold, verbose=0)

results.append((lr, num\_units, batch\_size, val\_loss, val\_accuracy))

print(f"Fold {fold\_no} - LR: {lr}, Units: {num\_units}, Batch Size: {batch\_size} - Accuracy: {val\_accuracy \* 100:.2f}%")

fold\_no += 1

5. **Analyze Results**

import pandas as pd

# Convert results to a DataFrame

df\_results = pd.DataFrame(results, columns=['Learning Rate', 'Hidden Units', 'Batch Size', 'Val Loss', 'Val Accuracy'])

# Find the best configuration

best\_result = df\_results.loc[df\_results['Val Accuracy'].idxmax()]

print("\nBest Hyperparameters:")

print(best\_result)

Summary:

1. Define a function to build the model with variable hyperparameters.
2. Set up cross-validation using KFold to evaluate different hyperparameter combinations.
3. Train and evaluate the model on different folds.
4. Analyze the results to identify the best-performing hyperparameter combination.

By using this approach, you can systematically tune your hyperparameters and find the configuration that maximizes model performance on the MNIST dataset.

* 1. Now try adding Batch Normalization and compare the learning curves: is it converging faster than before? Does it produce a better model?

Answer:- Adding Batch Normalization (BN) to your neural network can help stabilize and accelerate training by normalizing the activations of each layer. To compare the learning curves with and without Batch Normalization, follow these steps:

### 1. Add Batch Normalization to the Model

Here’s how to modify the DNN model to include Batch Normalization:

import tensorflow as tf

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Dense, BatchNormalization

from tensorflow.keras.initializers import HeNormal

from tensorflow.keras.optimizers import Adam

from tensorflow.keras.callbacks import EarlyStopping, ModelCheckpoint

from tensorflow.keras.datasets import mnist

import matplotlib.pyplot as plt

# Load and preprocess the data (same as before)

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

# Filter digits 0 to 4

x\_train = x\_train[y\_train < 5]

y\_train = y\_train[y\_train < 5]

x\_test = x\_test[y\_test < 5]

y\_test = y\_test[y\_test < 5]

# Normalize pixel values to [0, 1]

x\_train = x\_train.astype('float32') / 255.0

x\_test = x\_test.astype('float32') / 255.0

# Flatten images (28x28 to 784)

x\_train = x\_train.reshape(-1, 28 \* 28)

x\_test = x\_test.reshape(-1, 28 \* 28)

# Convert labels to categorical format

num\_classes = 5

y\_train = tf.keras.utils.to\_categorical(y\_train, num\_classes)

y\_test = tf.keras.utils.to\_categorical(y\_test, num\_classes)

# Model with Batch Normalization

def build\_model\_with\_bn(learning\_rate=0.001, num\_hidden\_units=100):

model = Sequential()

model.add(Dense(num\_hidden\_units, kernel\_initializer=HeNormal(), input\_shape=(28 \* 28,)))

model.add(BatchNormalization())

model.add(tf.keras.layers.Activation('elu'))

for \_ in range(4):

model.add(Dense(num\_hidden\_units, kernel\_initializer=HeNormal()))

model.add(BatchNormalization())

model.add(tf.keras.layers.Activation('elu'))

model.add(Dense(num\_classes, activation='softmax'))

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

model.compile(optimizer=optimizer,

loss='categorical\_crossentropy',

metrics=['accuracy'])

return model

2. **Train and Evaluate the Model with Batch Normalization**

# Training parameters

learning\_rate = 0.001

num\_hidden\_units = 100

batch\_size = 32

epochs = 20

# Model and callbacks

model\_bn = build\_model\_with\_bn(learning\_rate, num\_hidden\_units)

early\_stopping\_bn = EarlyStopping(monitor='val\_loss', patience=3, restore\_best\_weights=True)

checkpoint\_bn = ModelCheckpoint('mnist\_model\_bn\_checkpoint.h5', save\_best\_only=True, monitor='val\_loss')

# Train the model with Batch Normalization

history\_bn = model\_bn.fit(x\_train, y\_train,

epochs=epochs,

batch\_size=batch\_size,

validation\_split=0.2,

callbacks=[early\_stopping\_bn, checkpoint\_bn])

# Load the best model

best\_model\_bn = tf.keras.models.load\_model('mnist\_model\_bn\_checkpoint.h5')

# Evaluate the model

test\_loss\_bn, test\_accuracy\_bn = best\_model\_bn.evaluate(x\_test, y\_test)

print(f"Test Accuracy with Batch Normalization: {test\_accuracy\_bn \* 100:.2f}%")

3. **Plot Learning Curves**

# Plot learning curves

def plot\_learning\_curves(history, label):

plt.plot(history.history['accuracy'], label=f'{label} - Training Accuracy')

plt.plot(history.history['val\_accuracy'], label=f'{label} - Validation Accuracy')

plt.xlabel('Epoch')

plt.ylabel('Accuracy')

plt.legend()

plt.grid(True)

plt.title(f'Learning Curves ({label})')

plt.show()

# Plot learning curves for models with and without Batch Normalization

plot\_learning\_curves(history\_bn, 'Batch Normalization')

Summary and Comparison:

* Learning Curves: By plotting the learning curves, you can visually compare how quickly each model converges and whether the inclusion of Batch Normalization accelerates convergence.
* Model Performance: Compare the final accuracy of the models with and without Batch Normalization. If Batch Normalization improves accuracy, it indicates better performance and possibly more stable training.
* Convergence Speed: Batch Normalization often speeds up convergence by normalizing activations, which helps mitigate issues related to vanishing/exploding gradients and allows the use of higher learning rates.

In general, adding Batch Normalization tends to make the training process faster and can lead to a better-performing model, though this is not always guaranteed and may depend on the specific dataset and model architecture.

* 1. Is the model overfitting the training set? Try adding dropout to every layer and try again. Does it help?

Answer:- To address potential overfitting in your model, you can add Dropout layers to every layer in your network. Dropout is a regularization technique that randomly deactivates a fraction of neurons during training, which helps prevent the model from relying too heavily on any particular neuron and can improve generalization.

Here's how you can modify the DNN model to include Dropout layers, retrain it, and compare the results:

### 1. Modify the Model to Include Dropout

import tensorflow as tf

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Dense, BatchNormalization, Dropout

from tensorflow.keras.initializers import HeNormal

from tensorflow.keras.optimizers import Adam

from tensorflow.keras.callbacks import EarlyStopping, ModelCheckpoint

from tensorflow.keras.datasets import mnist

import matplotlib.pyplot as plt

# Load and preprocess the data (same as before)

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

# Filter digits 0 to 4

x\_train = x\_train[y\_train < 5]

y\_train = y\_train[y\_train < 5]

x\_test = x\_test[y\_test < 5]

y\_test = y\_test[y\_test < 5]

# Normalize pixel values to [0, 1]

x\_train = x\_train.astype('float32') / 255.0

x\_test = x\_test.astype('float32') / 255.0

# Flatten images (28x28 to 784)

x\_train = x\_train.reshape(-1, 28 \* 28)

x\_test = x\_test.reshape(-1, 28 \* 28)

# Convert labels to categorical format

num\_classes = 5

y\_train = tf.keras.utils.to\_categorical(y\_train, num\_classes)

y\_test = tf.keras.utils.to\_categorical(y\_test, num\_classes)

# Model with Batch Normalization and Dropout

def build\_model\_with\_bn\_dropout(learning\_rate=0.001, num\_hidden\_units=100, dropout\_rate=0.5):

model = Sequential()

model.add(Dense(num\_hidden\_units, kernel\_initializer=HeNormal(), input\_shape=(28 \* 28,)))

model.add(BatchNormal

2. **Train and Evaluate the Model with Dropout**

# Training parameters

learning\_rate = 0.001

num\_hidden\_units = 100

batch\_size = 32

epochs = 20

dropout\_rate = 0.5

# Model and callbacks

model\_bn\_dropout = build\_model\_with\_bn\_dropout(learning\_rate, num\_hidden\_units, dropout\_rate)

early\_stopping\_bn\_dropout = EarlyStopping(monitor='val\_loss', patience=3, restore\_best\_weights=True)

checkpoint\_bn\_dropout = ModelCheckpoint('mnist\_model\_bn\_dropout\_checkpoint.h5', save\_best\_only=True, monitor='val\_loss')

# Train the model with Batch Normalization and Dropout

history\_bn\_dropout = model\_bn\_dropout.fit(x\_train, y\_train,

epochs=epochs,

batch\_size=batch\_size,

validation\_split=0.2,

callbacks=[early\_stopping\_bn\_dropout, checkpoint\_bn\_dropout])

# Load the best model

best\_model\_bn\_dropout = tf.keras.models.load\_model('mnist\_model\_bn\_dropout\_checkpoint.h5')

# Evaluate the model

test\_loss\_bn\_dropout, test\_accuracy\_bn\_dropout = best\_model\_bn\_dropout.evaluate(x\_test, y\_test)

print(f"Test Accuracy with Batch Normalization and Dropout: {test\_accuracy\_bn\_dropout \* 100:.2f}%")

3. **Plot Learning Curves for Comparison**

# Plot learning curves

def plot\_learning\_curves(history, label):

plt.plot(history.history['accuracy'], label=f'{label} - Training Accuracy')

plt.plot(history.history['val\_accuracy'], label=f'{label} - Validation Accuracy')

plt.xlabel('Epoch')

plt.ylabel('Accuracy')

plt.legend()

plt.grid(True)

plt.title(f'Learning Curves ({label})')

plt.show()

# Plot learning curves for models with and without Dropout

plot\_learning\_curves(history\_bn\_dropout, 'Batch Normalization + Dropout')

Summary:

* Overfitting Check: Compare the training and validation accuracy for models with and without Dropout. If the model without Dropout shows much higher training accuracy compared to validation accuracy, it might be overfitting.
* Effectiveness of Dropout: The addition of Dropout can help reduce overfitting by preventing the network from becoming too reliant on specific neurons. Compare the learning curves and final test accuracy with and without Dropout to determine if it improves generalization and speeds up convergence.

Dropout helps in improving generalization and often results in better performance on unseen data by reducing the likelihood of overfitting.

1. Transfer learning.
   1. Create a new DNN that reuses all the pretrained hidden layers of the previous model, freezes them, and replaces the softmax output layer with a new one.

Answer:- To create a new DNN that reuses all the pretrained hidden layers from the previous model, freezes them, and replaces the softmax output layer with a new one, follow these steps:

### 1. Load the Pretrained Model and Extract Hidden Layers

First, load the pretrained model and extract the hidden layers. We’ll then create a new model that reuses these layers but replaces the output layer.

import tensorflow as tf

from tensorflow.keras.models import Model

from tensorflow.keras.layers import Dense, BatchNormalization, Dropout

from tensorflow.keras.initializers import HeNormal

# Load the best pretrained model

pretrained\_model = tf.keras.models.load\_model('mnist\_model\_bn\_dropout\_checkpoint.h5')

# Display the model summary to inspect layers

pretrained\_model.summary()

# Extract layers except for the output layer

hidden\_layers = [layer for layer in pretrained\_model.layers if not isinstance(layer, Dense) or layer.name != 'dense\_4']

### 2. Build the New Model with Frozen Pretrained Layers

Create a new model where the hidden layers from the pretrained model are reused and frozen, and replace the output layer with a new one.

def build\_new\_model\_with\_frozen\_layers(new\_num\_classes, dropout\_rate=0.5):

# Input layer

inputs = tf.keras.Input(shape=(28 \* 28,))

# Reuse and freeze hidden layers from the pretrained model

x = inputs

for layer in hidden\_layers:

x = layer(x)

layer.trainable = False # Freeze the layer

# Add new output layer

x = Dense(new\_num\_classes, activation='softmax')(x)

# Create the new model

new\_model = Model(inputs=inputs, outputs=x)

# Compile the new model

new\_model.compile(optimizer='adam',

loss='categorical\_crossentropy',

metrics=['accuracy'])

return new\_model

# Build the new model with the same hidden layers but a new output layer

new\_num\_classes = 10 # For example, change the number of classes if needed

new\_model = build\_new\_model\_with\_frozen\_layers(new\_num\_classes)

new\_model.summary()

### 3. Train the New Model

Train the new model on the updated dataset or with new classes if applicable. Here’s an example of training it with the same digits but a different output configuration.

# Assuming the new number of classes is different; adjust the labels accordingly

# For demonstration, using the same 0-4 digits for simplicity

x\_train\_new, y\_train\_new = x\_train, y\_train

x\_test\_new, y\_test\_new = x\_test, y\_test

# Training parameters

batch\_size = 32

epochs = 10

# Callbacks

early\_stopping\_new = tf.keras.callbacks.EarlyStopping(monitor='val\_loss', patience=3, restore\_best\_weights=True)

checkpoint\_new = tf.keras.callbacks.ModelCheckpoint('new\_mnist\_model\_checkpoint.h5', save\_best\_only=True, monitor='val\_loss')

# Train the new model

history\_new = new\_model.fit(x\_train\_new, y\_train\_new,

epochs=epochs,

batch\_size=batch\_size,

validation\_split=0.2,

callbacks=[early\_stopping\_new, checkpoint\_new])

# Load the best model

best\_model\_new = tf.keras.models.load\_model('new\_mnist\_model\_checkpoint.h5')

# Evaluate the new model

test\_loss\_new, test\_accuracy\_new = best\_model\_new.evaluate(x\_test\_new, y\_test\_new)

print(f"Test Accuracy of the New Model: {test\_accuracy\_new \* 100:.2f}%")

Summary:

1. Reuse and Freeze Layers: Extract hidden layers from the pretrained model and freeze them to prevent updates during training.
2. Replace Output Layer: Add a new output layer to adapt the model to new classes or configurations.
3. Train and Evaluate: Train the new model with the updated output layer and evaluate its performance.

This approach allows you to leverage the learned representations from the pretrained model while adapting the output layer to new tasks or configurations.

* 1. Train this new DNN on digits 5 to 9, using only 100 images per digit, and time how long it takes. Despite this small number of examples, can you achieve high precision?

Answer:- Training a neural network on a small dataset, such as 100 images per digit for digits 5 to 9, can be challenging. However, using a pretrained model with frozen layers and only updating the final layer can still yield good results due to the powerful representations learned by the pretrained model.

Here's how you can train the new DNN on digits 5 to 9, measure the time taken, and evaluate the precision:

### 1. Load and Preprocess the Data

First, filter the dataset to include only digits 5 to 9, and then reduce the dataset to 100 images per class.

import numpy as np

import tensorflow as tf

from tensorflow.keras.datasets import mnist

from tensorflow.keras.utils import to\_categorical

import time

# Load MNIST dataset

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

# Filter digits 5 to 9

x\_train = x\_train[(y\_train >= 5)]

y\_train = y\_train[(y\_train >= 5)]

x\_test = x\_test[(y\_test >= 5)]

y\_test = y\_test[(y\_test >= 5)]

# Reduce to 100 images per class

num\_samples\_per\_class = 100

x\_train\_list = []

y\_train\_list = []

for digit in range(5, 10):

indices = np.where(y\_train == digit)[0]

selected\_indices = np.random.choice(indices, num\_samples\_per\_class, replace=False)

x\_train\_list.append(x\_train[selected\_indices])

y\_train\_list.append(y\_train[selected\_indices])

x\_train = np.vstack(x\_train\_list)

y\_train = np.concatenate(y\_train\_list)

# Normalize pixel values to [0, 1]

x\_train = x\_train.astype('float32') / 255.0

x\_test = x\_test.astype('float32') / 255.0

# Flatten images (28x28 to 784)

x\_train = x\_train.reshape(-1, 28 \* 28)

x\_test = x\_test.reshape(-1,

2. **Build and Train the New Model**

# Define the new model with pretrained layers and new output layer

new\_model = build\_new\_model\_with\_frozen\_layers(new\_num\_classes=num\_classes)

# Callbacks

early\_stopping\_new = tf.keras.callbacks.EarlyStopping(monitor='val\_loss', patience=3, restore\_best\_weights=True)

checkpoint\_new = tf.keras.callbacks.ModelCheckpoint('new\_mnist\_model\_checkpoint.h5', save\_best\_only=True, monitor='val\_loss')

# Measure the training time

start\_time = time.time()

# Train the model

history\_new = new\_model.fit(x\_train, y\_train,

epochs=20,

batch\_size=16,

validation\_split=0.2,

callbacks=[early\_stopping\_new, checkpoint\_new])

end\_time = time.time()

training\_time = end\_time - start\_time

print(f"Training Time: {training\_time:.2f} seconds")

# Load the best model

best\_model\_new = tf.keras.models.load\_model('new\_mnist\_model\_checkpoint.h5')

# Evaluate the model

test\_loss\_new, test\_accuracy\_new = best\_model\_new.evaluate(x\_test, y\_test)

print(f"Test Accuracy of the New Model: {test\_accuracy\_new \* 100:.2f}%")

Summary:

1. Data Preparation: Filter the dataset to include only digits 5 to 9, and sample 100 images per digit.
2. Training: Train the model with the frozen layers and new output layer, while measuring the time taken.
3. Evaluation: Evaluate the model on the test set to check the precision.

Despite the small number of examples, if the pretrained hidden layers are effective and the new output layer is properly trained, achieving high precision is feasible. The pretrained layers have already learned useful features from digits 0 to 4, which can be beneficial for digits 5 to 9 even with limited data.

* 1. Try caching the frozen layers, and train the model again: how much faster is it now?

Answer:- Caching the frozen layers can help speed up the training process by avoiding the repeated computation of the same layers during training. Here's how to implement caching for the frozen layers and measure the impact on training speed.

### 1. Modify the Model to Use Cached Layers

First, create a new model where the frozen layers are cached, so that they are only computed once. We can use TensorFlow's functional API to help with caching.

import tensorflow as tf

from tensorflow.keras.models import Model

from tensorflow.keras.layers import Dense, BatchNormalization, Dropout, Input

from tensorflow.keras.initializers import HeNormal

# Function to build a model with cached frozen layers

def build\_model\_with\_cached\_layers(new\_num\_classes, dropout\_rate=0.5):

# Input layer

inputs = Input(shape=(28 \* 28,))

# Reuse and freeze hidden layers from the pretrained model

x = inputs

for layer in hidden\_layers:

x = layer(x)

layer.trainable = False # Freeze the layer

# Add new output layer

x = Dense(new\_num\_classes, activation='softmax')(x)

# Create the new model

new\_model = Model(inputs=inputs, outputs=x)

# Compile the new model

new\_model.compile(optimizer='adam',

loss='categorical\_crossentropy',

metrics=['accuracy'])

return new\_model

### 2. Train the Model with Cached Layers

Use the cached model to retrain and measure the training time.

# Build the new model with cached layers

cached\_model = build\_model\_with\_cached\_layers(new\_num\_classes=num\_classes)

# Callbacks

early\_stopping\_cached = tf.keras.callbacks.EarlyStopping(monitor='val\_loss', patience=3, restore\_best\_weights=True)

checkpoint\_cached = tf.keras.callbacks.ModelCheckpoint('cached\_mnist\_model\_checkpoint.h5', save\_best\_only=True, monitor='val\_loss')

# Measure the training time

start\_time\_cached = time.time()

# Train the model with cached layers

history\_cached = cached\_model.fit(x\_train, y\_train,

epochs=20,

batch\_size=16,

validation\_split=0.2,

callbacks=[early\_stopping\_cached, checkpoint\_cached])

end\_time\_cached = time.time()

training\_time\_cached = end\_time\_cached - start\_time\_cached

print(f"Training Time with Cached Layers: {training\_time\_cached:.2f} seconds")

# Load the best model

best\_model\_cached = tf.keras.models.load\_model('cached\_mnist\_model\_checkpoint.h5')

# Evaluate the model

test\_loss\_cached, test\_accuracy\_cached = best\_model\_cached.evaluate(x\_test, y\_test)

print(f"Test Accuracy of the Cached Model: {test\_accuracy\_cached \* 100:.2f}%")

Summary:

1. Cached Layers: By caching the frozen layers, you should reduce redundant computations and potentially speed up training. This is because the frozen layers are reused and their outputs are cached, avoiding recalculation.
2. Training Time: Compare the training time of the model with and without caching. You should see a reduction in training time when using the cached layers.
3. Evaluation: Ensure that the accuracy of the cached model remains consistent with the non-cached model to confirm that caching only improves performance without affecting model quality.

Using caching for frozen layers can significantly speed up training, especially if the network is large or if you're working with a complex dataset.

* 1. Try again reusing just four hidden layers instead of five. Can you achieve a higher precision?

Answer:- To test if reusing just four hidden layers instead of five improves precision, follow these steps:

### 1. Modify the Model to Use Four Hidden Layers

First, extract only the first four hidden layers from the pretrained model and create a new model with these layers. The output layer will be replaced as before.

# Extract the first four hidden layers from the pretrained model

# Assuming the layers are ordered sequentially

num\_hidden\_layers\_to\_reuse = 4

hidden\_layers\_reduced = [layer for layer in pretrained\_model.layers if isinstance(layer, Dense) and layer.name.startswith('dense')][:num\_hidden\_layers\_to\_reuse]

def build\_model\_with\_four\_layers(new\_num\_classes, dropout\_rate=0.5):

# Input layer

inputs = tf.keras.Input(shape=(28 \* 28,))

# Reuse and freeze only the first four hidden layers from the pretrained model

x = inputs

for layer in hidden\_layers\_reduced:

x = layer(x)

layer.trainable = False # Freeze the layer

# Add new output layer

x = Dense(new\_num\_classes, activation='softmax')(x)

# Create the new model

new\_model\_reduced = Model(inputs=inputs, outputs=x)

# Compile the new model

new\_model\_reduced.compile(optimizer='adam',

loss='categorical\_crossentropy',

metrics=['accuracy'])

return new\_model\_reduced

# Build the new model with four hidden layers

model\_with\_four\_layers = build\_model\_with\_four\_layers(new\_num\_classes=num\_classes)

# Callbacks

early\_stopping\_reduced = tf.keras.callbacks.EarlyStopping(monitor='val\_loss', patience=3, restore\_best\_weights=True)

checkpoint\_reduced = tf.keras.callbacks.ModelCheckpoint('four\_layers\_mnist\_model\_checkpoint.h5', save\_best\_only=True, monitor='val\_loss')

# Measure the training time

start\_time\_reduced = time.time()

# Train the model with four hidden layers

history\_reduced = model\_with\_four\_layers.fit(x\_train, y\_train,

epochs=20,

batch\_size=16,

validation\_split=0.2,

callbacks=[early\_stopping\_reduced, checkpoint\_reduced])

end\_time\_reduced = time.time()

training\_time\_reduced = end\_time\_reduced - start\_time\_reduced

print(f"Training Time with Four Hidden Layers: {training\_time\_reduced:.2f} seconds")

# Load the best model

best\_model\_reduced = tf.keras.models.load\_model('four\_layers\_mnist\_model\_checkpoint.h5')

# Evaluate the model

test\_loss\_reduced, test\_accuracy\_reduced = best\_model\_reduced.evaluate(x\_test, y\_test)

print(f"Test Accuracy of the Model with Four Hidden Layers: {test\_accuracy\_reduced \* 100:.2f}%")

Summary:

1. Reduce Layers: By using only four hidden layers, you can investigate if fewer layers lead to better performance or faster convergence.
2. Training and Evaluation: Compare the precision and training time of the model with four hidden layers against the model with five hidden layers.
3. Possible Outcomes: Reducing the number of hidden layers might affect the model's ability to capture complex features. However, it could also lead to faster training and potentially better generalization if the previous model was overfitting.

You should observe whether using fewer layers helps achieve higher precision or if it impacts performance negatively. The change in performance will provide insight into the trade-offs between model complexity and training efficiency.

* 1. Now unfreeze the top two hidden layers and continue training: can you get the model to perform even better?

Answer:- Unfreezing the top two hidden layers and continuing training can allow the model to further adjust and improve performance. Here’s how you can modify the model to unfreeze the top two layers, retrain it, and evaluate its performance:

### 1. Unfreeze the Top Two Hidden Layers

To unfreeze the top two hidden layers and continue training:

# Function to build the model and unfreeze top layers

def build\_and\_unfreeze\_model\_with\_reduced\_layers(new\_num\_classes, dropout\_rate=0.5):

# Input layer

inputs = tf.keras.Input(shape=(28 \* 28,))

# Reuse hidden layers from the pretrained model

x = inputs

for i, layer in enumerate(hidden\_layers\_reduced):

x = layer(x)

if i >= len(hidden\_layers\_reduced) - 2: # Unfreeze the top two layers

layer.trainable = True

else:

layer.trainable = False

# Add new output layer

x = Dense(new\_num\_classes, activation='softmax')(x)

# Create the new model

new\_model\_unfrozen = Model(inputs=inputs, outputs=x)

# Compile the new model

new\_model\_unfrozen.compile(optimizer='adam',

loss='categorical\_crossentropy',

metrics=['accuracy'])

return new\_model\_unfrozen

# Build the model with unfrozen top two layers

model\_with\_unfrozen\_layers = build\_and\_unfreeze\_model\_with\_reduced\_layers(new\_num\_classes=num\_classes)

# Callbacks

early\_stopping\_unfrozen = tf.keras.callbacks.EarlyStopping(monitor='val\_loss', patience=3, restore\_best\_weights=True)

checkpoint\_unfrozen = tf.keras.callbacks.ModelCheckpoint('unfrozen\_layers\_mnist\_model\_checkpoint.h5', save\_best\_only=True, monitor='val\_loss')

# Measure the training time

start\_time\_unfrozen = time.time()

# Train the model with unfrozen top two layers

history\_unfrozen = model\_with\_unfrozen\_layers.fit(x\_train, y\_train,

epochs=20,

batch\_size=16,

validation\_split=0.2,

callbacks=[early\_stopping\_unfrozen, checkpoint\_unfrozen])

end\_time\_unfrozen = time.time()

training\_time\_unfrozen = end\_time\_unfrozen - start\_time\_unfrozen

print(f"Training Time with Top Two Hidden Layers Unfrozen: {training\_time\_unfrozen:.2f} seconds")

# Load the best model

best\_model\_unfrozen = tf.keras.models.load\_model('unfrozen\_layers\_mnist\_model\_checkpoint.h5')

# Evaluate the model

test\_loss\_unfrozen, test\_accuracy\_unfrozen = best\_model\_unfrozen.evaluate(x\_test, y\_test)

print(f"Test Accuracy of the Model with Top Two Hidden Layers Unfrozen: {test\_accuracy\_unfrozen \* 100:.2f}%")

Summary:

1. Unfreeze Layers: By unfreezing the top two hidden layers, you allow the model to further adjust these layers during training, which might improve the performance if these layers learn better representations for the new task.
2. Training and Evaluation: Retrain the model and measure how the accuracy and training time compare to the previous configurations.
3. Performance: Observe if the performance improves with the top layers being trainable. Sometimes, unfreezing higher layers allows the model to better adapt to the specific features of the new data.

You may find that unfreezing some of the top layers can enhance model performance by allowing it to fine-tune features that are crucial for the new classification task.

1. Pretraining on an auxiliary task.
   1. In this exercise you will build a DNN that compares two MNIST digit images and predicts whether they represent the same digit or not. Then you will reuse the lower layers of this network to train an MNIST classifier using very little training data. Start by building two DNNs (let’s call them DNN A and B), both similar to the one you built earlier but without the output layer: each DNN should have five hidden layers of 100 neurons each, He initialization, and ELU activation. Next, add one more hidden layer with 10 units on top of both DNNs. To do this, you should use TensorFlow’s concat() function with axis=1 to concatenate the outputs of both DNNs for each instance, then feed the result to the hidden layer. Finally, add an output layer with a single neuron using the logistic activation function.

Answer:- To build a DNN that compares two MNIST digit images and predicts whether they represent the same digit, we can follow these steps:

### 1. Build Two DNNs (DNN A and DNN B) Without the Output Layer

First, we need to define two DNNs with the same architecture. Each will have five hidden layers of 100 neurons each, using He initialization and ELU activation.

import tensorflow as tf

from tensorflow.keras.layers import Dense, Input, Concatenate

from tensorflow.keras.models import Model

from tensorflow.keras.initializers import HeNormal

def create\_dnn\_without\_output():

inputs = Input(shape=(28 \* 28,))

x = Dense(100, activation='elu', kernel\_initializer=HeNormal())(inputs)

x = Dense(100, activation='elu', kernel\_initializer=HeNormal())(x)

x = Dense(100, activation='elu', kernel\_initializer=HeNormal())(x)

x = Dense(100, activation='elu', kernel\_initializer=HeNormal())(x)

x = Dense(100, activation='elu', kernel\_initializer=HeNormal())(x)

return Model(inputs, x)

# Create two DNNs

dnn\_a = create\_dnn\_without\_output()

dnn\_b = create\_dnn\_without\_output()

# Print summaries to check architectures

dnn\_a.summary()

dnn\_b.summary()

### 2. Add a Hidden Layer to Concatenate Outputs

Next, we concatenate the outputs of both DNNs and feed them into an additional hidden layer.

def create\_comparison\_model(dnn\_a, dnn\_b):

# Input layers

input\_a = Input(shape=(28 \* 28,))

input\_b = Input(shape=(28 \* 28,))

# Get the output from both DNNs

output\_a = dnn\_a(input\_a)

output\_b = dnn\_b(input\_b)

# Concatenate the outputs

concatenated = Concatenate(axis=1)([output\_a, output\_b])

# Add a hidden layer on top

x = Dense(10, activation='elu', kernel\_initializer=HeNormal())(concatenated)

# Add the output layer

output = Dense(1, activation='sigmoid')(x)

# Create the final model

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

return model

# Create the comparison model

comparison\_model = create\_comparison\_model(dnn\_a, dnn\_b)

# Print model summary

comparison\_model.summary()

### 3. Compile and Train the Model

Compile the model with an appropriate loss function and optimizer, then train it using a dataset where you have pairs of images and labels indicating whether they represent the same digit or not.

from tensorflow.keras.optimizers import Adam

from tensorflow.keras.losses import BinaryCrossentropy

from tensorflow.keras.metrics import Accuracy

# Compile the model

comparison\_model.compile(optimizer=Adam(),

loss=BinaryCrossentropy(),

metrics=[Accuracy()])

# Dummy training data and labels (Replace these with actual data)

import numpy as np

# Assuming x\_train\_a, x\_train\_b are image pairs and y\_train are labels

x\_train\_a = np.random.random((1000, 28 \* 28))

x\_train\_b = np.random.random((1000, 28 \* 28))

y\_train = np.random.randint(0, 2, size=(1000,))

# Train the model

comparison\_model.fit([x\_train\_a, x\_train\_b], y\_train,

epochs=10,

batch\_size=32,

validation\_split=0.2)

### 4. Reuse the Lower Layers for MNIST Classification

To reuse the lower layers of the network for MNIST classification with very little training data, we can create a new model that uses the frozen weights from the DNNs in the comparison model.

def create\_mnist\_classifier\_with\_reused\_layers(dnn\_a, dnn\_b):

# Input layer

inputs = Input(shape=(28 \* 28,))

# Reuse the lower layers from DNN A and B

features\_a = dnn\_a(inputs)

features\_b = dnn\_b(inputs)

# Concatenate and add classification head

concatenated\_features = Concatenate(axis=1)([features\_a, features\_b])

x = Dense(10, activation='elu', kernel\_initializer=HeNormal())(concatenated\_features)

output = Dense(10, activation='softmax')(x) # For classification into 10 digits

# Create the classifier model

classifier\_model = Model(inputs=inputs, outputs=output)

return classifier\_model

# Build the classifier model

classifier\_model = create\_mnist\_classifier\_with\_reused\_layers(dnn\_a, dnn\_b)

# Freeze the weights of the reused layers

for layer in dnn\_a.layers + dnn\_b.layers:

layer.trainable = False

# Compile the model

classifier\_model.compile(optimizer=Adam(),

loss='categorical\_crossentropy',

metrics=['accuracy'])

# Dummy training data for MNIST classification (Replace these with actual data)

x\_train\_mnist = np.random.random((100, 28 \* 28)) # Example with very few images

y\_train\_mnist = np.random.randint(0, 10, size=(100,))

y\_train\_mnist = tf.keras.utils.to\_categorical(y\_train\_mnist, 10)

# Train the classifier model

classifier\_model.fit(x\_train\_mnist, y\_train\_mnist,

epochs=10,

batch\_size=10,

validation\_split=0.2)

Summary

1. Create DNNs: Build two separate DNNs with five hidden layers each, without the output layer.
2. Compare Outputs: Concatenate the outputs of both DNNs and add a hidden layer and a logistic activation output layer to compare the images.
3. Train the Model: Compile and train the model to distinguish if two digit images are the same or not.
4. Reuse for Classification: Reuse the lower layers of the trained DNNs for a new MNIST classification task with minimal training data.

By reusing the lower layers, you can leverage the pre-trained features for a new classification task, potentially improving performance even with limited data.

* 1. Split the MNIST training set in two sets: split #1 should containing 55,000 images, and split #2 should contain contain 5,000 images. Create a function that generates a training batch where each instance is a pair of MNIST images picked from split #1. Half of the training instances should be pairs of images that belong to the same class, while the other half should be images from different classes. For each pair, the training label should be 0 if the images are from the same class, or 1 if they are from different classes.

Answer:- To generate a training batch with pairs of MNIST images for a model that predicts whether two images are from the same class or different classes, follow these steps:

### 1. Prepare the MNIST Dataset

First, load and split the MNIST dataset.

import tensorflow as tf

import numpy as np

# Load MNIST dataset

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

# Flatten and normalize images

x\_train = x\_train.reshape(-1, 28 \* 28) / 255.0

x\_test = x\_test.reshape(-1, 28 \* 28) / 255.0

# Split the training data into two sets

x\_train\_split1, x\_train\_split2 = x\_train[:55000], x\_train[55000:]

y\_train\_split1, y\_train\_split2 = y\_train[:55000], y\_train[55000:]

### 2. Create the Pair Generation Function

This function will generate pairs of images along with their labels (0 or 1).

def generate\_pairs(x\_data, y\_data, batch\_size):

"""Generate a batch of image pairs and labels."""

pairs = []

labels = []

num\_classes = 10

class\_indices = [np.where(y\_data == i)[0] for i in range(num\_classes)]

while len(pairs) < batch\_size:

# Randomly decide whether to create a positive or negative pair

same\_class = np.random.choice([True, False])

if same\_class:

# Choose a random class

class\_idx = np.random.randint(num\_classes)

indices = np.random.choice(class\_indices[class\_idx], size=2, replace=False)

pairs.append((x\_data[indices[0]], x\_data[indices[1]]))

labels.append(0)

else:

# Choose two different classes

class\_idx1, class\_idx2 = np.random.choice(num\_classes, size=2, replace=False)

index1 = np.random.choice(class\_indices[class\_idx1])

index2 = np.random.choice(class\_indices[class\_idx2])

pairs.append((x\_data[index1], x\_data[index2]))

labels.append(1)

# Convert lists to arrays

pairs = np.array(pairs)

labels = np.array(labels)

# Separate pairs into two arrays for inputs

x1 = np.array([pair[0] for pair in pairs])

x2 = np.array([pair[1] for pair in pairs])

return [x1, x2], labels

### 3. Example Usage

Generate a batch of image pairs and print their shapes to verify.

# Set batch size

batch\_size = 32

# Generate a batch

(x1\_batch, x2\_batch), y\_batch = generate\_pairs(x\_train\_split1, y\_train\_split1, batch\_size)

print(f"Shape of x1\_batch: {x1\_batch.shape}")

print(f"Shape of x2\_batch: {x2\_batch.shape}")

print(f"Shape of y\_batch: {y\_batch.shape}")

print(f"Example labels: {y\_batch[:10]}")

Summary

1. Data Splitting: Split the MNIST dataset into two sets.
2. Pair Generation Function: Create a function to generate pairs of images. Half of the pairs are from the same class, and the other half are from different classes. Each pair is labeled accordingly.
3. Batch Generation: Use the function to generate batches of pairs for training.

This setup will allow you to create batches for training a model that learns to compare images based on whether they are from the same class or different classes.

* 1. Train the DNN on this training set. For each image pair, you can simultaneously feed the first image to DNN A and the second image to DNN B. The whole network will gradually learn to tell whether two images belong to the same class or not.

Answer:- To train the DNN that compares two MNIST images using the pair generation method, follow these steps:

### 1. Define the DNN Model

You should have already created the DNN model that compares two images. For reference, here’s a recap of the model definition:

import tensorflow as tf

from tensorflow.keras.layers import Dense, Input, Concatenate

from tensorflow.keras.models import Model

from tensorflow.keras.initializers import HeNormal

def create\_dnn\_without\_output():

inputs = Input(shape=(28 \* 28,))

x = Dense(100, activation='elu', kernel\_initializer=HeNormal())(inputs)

x = Dense(100, activation='elu', kernel\_initializer=HeNormal())(x)

x = Dense(100, activation='elu', kernel\_initializer=HeNormal())(x)

x = Dense(100, activation='elu', kernel\_initializer=HeNormal())(x)

x = Dense(100, activation='elu', kernel\_initializer=HeNormal())(x)

return Model(inputs, x)

def create\_comparison\_model(dnn\_a, dnn\_b):

# Input layers

input\_a = Input(shape=(28 \* 28,))

input\_b = Input(shape=(28 \* 28,))

# Get the output from both DNNs

output\_a = dnn\_a(input\_a)

output\_b = dnn\_b(input\_b)

# Concatenate the outputs

concatenated = Concatenate(axis=1)([output\_a, output\_b])

# Add a hidden layer on top

x = Dense(10, activation='elu', kernel\_initializer=HeNormal())(concatenated)

# Add the output layer

output = Dense(1, activation='sigmoid')(x)

# Create the final model

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

return model

# Create two DNNs

dnn\_a = create\_dnn\_without\_output()

dnn\_b = create\_dnn\_without\_output()

# Create the comparison model

comparison\_model = create\_comparison\_model(dnn\_a, dnn\_b)

### 2. Compile the Model

Compile the comparison model with appropriate loss and metrics.

from tensorflow.keras.optimizers import Adam

from tensorflow.keras.losses import BinaryCrossentropy

from tensorflow.keras.metrics import Accuracy

comparison\_model.compile(optimizer=Adam(),

loss=BinaryCrossentropy(),

metrics=[Accuracy()])

### 3. Train the Model

Use the generate\_pairs function to generate batches and train the model.

# Training parameters

batch\_size = 32

epochs = 10

# Callbacks

early\_stopping = tf.keras.callbacks.EarlyStopping(monitor='val\_loss', patience=3, restore\_best\_weights=True)

checkpoint = tf.keras.callbacks.ModelCheckpoint('comparison\_model\_checkpoint.h5', save\_best\_only=True, monitor='val\_loss')

# Train the model

history = comparison\_model.fit(x=[x1\_batch, x2\_batch], y=y\_batch,

epochs=epochs,

batch\_size=batch\_size,

validation\_split=0.2,

callbacks=[early\_stopping, checkpoint])

# To check the training progress

import matplotlib.pyplot as plt

# Plot training and validation loss

plt.figure(figsize=(12, 6))

plt.subplot(1, 2, 1)

plt.plot(history.history['loss'], label='Training Loss')

plt.plot(history.history['val\_loss'], label='Validation Loss')

plt.xlabel('Epoch')

plt.ylabel('Loss')

plt.title('Loss over Epochs')

plt.legend()

# Plot training and validation accuracy

plt.subplot(1, 2, 2)

plt.plot(history.history['accuracy'], label='Training Accuracy')

plt.plot(history.history['val\_accuracy'], label='Validation Accuracy')

plt.xlabel('Epoch')

plt.ylabel('Accuracy')

plt.title('Accuracy over Epochs')

plt.legend()

plt.show()

Summary

1. Define and Compile Model: Ensure you have defined and compiled the DNN model for comparing image pairs.
2. Train the Model: Use the generate\_pairs function to create training batches and fit the model.
3. Evaluate and Monitor: Use callbacks and visualization to monitor the training process.

This approach will help train the DNN to learn whether two images are from the same class or different classes, leveraging the power of pairwise comparisons.

* 1. Now create a new DNN by reusing and freezing the hidden layers of DNN A and adding a softmax output layer on top with 10 neurons. Train this network on split #2 and see if you can achieve high performance despite having only 500 images per class.

Answer:- To create a new DNN that reuses the hidden layers from DNN A and adds a softmax output layer for classification, follow these steps:

### 1. Create the New DNN Model

You will use the pre-trained hidden layers from DNN A and add a new output layer with 10 neurons (for classification into 10 digits).

def create\_finetuned\_model(dnn\_a):

# Input layer

inputs = Input(shape=(28 \* 28,))

# Reuse the hidden layers from DNN A

features = dnn\_a(inputs)

# Add the classification head

x = Dense(10, activation='softmax', kernel\_initializer='he\_normal')(features)

# Create the new model

finetuned\_model = Model(inputs=inputs, outputs=x)

return finetuned\_model

# Create the new model

finetuned\_model = create\_finetuned\_model(dnn\_a)

# Freeze the weights of the reused layers

for layer in dnn\_a.layers:

layer.trainable = False

# Compile the model

finetuned\_model.compile(optimizer=Adam(),

loss='categorical\_crossentropy',

metrics=['accuracy'])

### 2. Prepare Data for Training

Ensure that split #2 is formatted correctly for training. You need to one-hot encode the labels.

from tensorflow.keras.utils import to\_categorical

# Prepare data

x\_train\_split2 = x\_train\_split2.reshape(-1, 28 \* 28) / 255.0

y\_train\_split2 = to\_categorical(y\_train\_split2, 10)

### 3. Train the Model

Train the new model using the small dataset from split #2. Use early stopping and checkpoints as before.

# Training parameters

batch\_size = 32

epochs = 10

# Callbacks

early\_stopping = tf.keras.callbacks.EarlyStopping(monitor='val\_loss', patience=3, restore\_best\_weights=True)

checkpoint = tf.keras.callbacks.ModelCheckpoint('finetuned\_model\_checkpoint.h5', save\_best\_only=True, monitor='val\_loss')

# Train the model

history = finetuned\_model.fit(x\_train\_split2, y\_train\_split2,

epochs=epochs,

batch\_size=batch\_size,

validation\_split=0.2,

callbacks=[early\_stopping, checkpoint])

# To check the training progress

import matplotlib.pyplot as plt

# Plot training and validation loss

plt.figure(figsize=(12, 6))

plt.subplot(1, 2, 1)

plt.plot(history.history['loss'], label='Training Loss')

plt.plot(history.history['val\_loss'], label='Validation Loss')

plt.xlabel('Epoch')

plt.ylabel('Loss')

plt.title('Loss over Epochs')

plt.legend()

# Plot training and validation accuracy

plt.subplot(1, 2, 2)

plt.plot(history.history['accuracy'], label='Training Accuracy')

plt.plot(history.history['val\_accuracy'], label='Validation Accuracy')

plt.xlabel('Epoch')

plt.ylabel('Accuracy')

plt.title('Accuracy over Epochs')

plt.legend()

plt.show()

Summary

1. Define and Compile New Model: Reuse the hidden layers from DNN A and add a softmax output layer for classification. Freeze the weights of the reused layers.
2. Prepare Data: Format split #2 for training, ensuring labels are one-hot encoded.
3. Train and Evaluate: Train the new model on the small dataset and monitor performance.

By reusing pre-trained layers, you leverage previously learned features, which can be especially useful for small datasets, potentially improving performance even with limited data.