**1. Deep Learning.**

**a. Build a DNN with five hidden layers of 100 neurons each, He initialization, and the**

**ELU activation function.**

**b. 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.**

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

**achieve.**

**d. Now try adding Batch Normalization and compare the learning curves: is it**

**converging faster than before? Does it produce a better model?**

**e. Is the model overfitting the training set? Try adding dropout to every layer and try**

**again. Does it help?**

Absolutely! Due to the complexity of this task, I'll provide a structured guide, code snippets, and explanations. Let's tackle this step-by-step.

**Caveats**

* **Full Implementation:** A complete implementation will be quite lengthy. I'll aim to provide crucial code blocks and conceptual guidance.
* **Environment Setup:** I'm assuming you have a suitable environment (TensorFlow/Keras, Python, etc.) set up.

**a. Building the Baseline DNN**

Python

import tensorflow as tf

from tensorflow import keras

model = keras.models.Sequential([

keras.layers.Flatten(input\_shape=[28, 28]), # Assuming MNIST image format

\*[keras.layers.Dense(100, activation='elu', kernel\_initializer='he\_normal') for \_ in range(5)],

keras.layers.Dense(5, activation='softmax') # 5 classes (0-4)

])

**b. Training with Adam and Early Stopping**

Python

optimizer = keras.optimizers.Adam()

model.compile(loss='sparse\_categorical\_crossentropy',

optimizer=optimizer,

metrics=['accuracy'])

early\_stopping = keras.callbacks.EarlyStopping(patience=10)

model\_checkpoint = keras.callbacks.ModelCheckpoint("my\_mnist\_model.h5", save\_best\_only=True)

# Load and prepare MNIST data (filtering for digits 0-4) ...

history = model.fit(X\_train, y\_train, epochs=50,

validation\_data=(X\_val, y\_val),

callbacks=[early\_stopping, model\_checkpoint])

**c. Hyperparameter Tuning (Cross-Validation)**

* **Techniques:** Grid search or randomized search with libraries like Scikit-learn.
* **Parameters:** Learning rate, batch size, dropout rate (we'll add this later), etc.
* **Evaluation:** Track precision on your validation set.

**d. Adding Batch Normalization**

Python

# ... (Previous layers)

for \_ in range(5):

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

model.add(keras.layers.BatchNormalization())

# ... (Output layer)

* **Retraining:** Repeat the training process, observing learning curves for training and validation error/accuracy.

**e. Overfitting and Dropout**

Python

# ... (Previous layers)

for \_ in range(5):

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

model.add(keras.layers.BatchNormalization())

model.add(keras.layers.Dropout(0.2)) # Start with a dropout rate of 0.2

# ... (Output layer)

* **Retrain and Monitor:** Evaluate overfitting via training vs. validation performance. Experiment with different dropout rates.

**2. Transfer learning.**

**a. 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.**

**b. 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?**

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

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

**precision?**

**e. Now unfreeze the top two hidden layers and continue training: can you get the**

**model to perform even better?**

Let's break down the transfer learning steps. I'll provide code guidance and explanations – remember to have your previous model saved for this part!

**a. Creating the New DNN**

Python

# 1. Load the previously trained model

base\_model = keras.models.load\_model("my\_mnist\_model.h5")

# 2. Freeze hidden layers

for layer in base\_model.layers[:-1]:

layer.trainable = False

# 3. New output layer

new\_output\_layer = keras.layers.Dense(5, activation='softmax') # 5 classes (5-9)

# 4. Construct the new model

new\_model = keras.Sequential([

\*base\_model.layers[:-1],

new\_output\_layer

])

**b. Training on Digits 5-9**

Python

optimizer = keras.optimizers.Adam() # You may fine-tune the optimizer as needed

new\_model.compile(loss='sparse\_categorical\_crossentropy',

optimizer=optimizer,

metrics=['accuracy'])

# Prepare your data with only digits 5-9 ...

start\_time = time.time()

history = new\_model.fit(X\_train\_small, y\_train\_small, epochs=20, # Adjust epochs

validation\_data=(X\_val\_small, y\_val\_small))

end\_time = time.time()

print("Training time:", end\_time - start\_time)

* **Precision on Limited Data:** You might be surprised at achieving decent precision even with fewer images thanks to the power of transfer learning!

**c. Caching Frozen Layers**

(This depends on your framework, Keras offers ways to cache layer outputs)

**d. Reusing Four Layers**

* Modify the model creation to only include the first four layers of the base model. Retrain and compare precision.

**e. Unfreezing Top Layers**

Python

for layer in new\_model.layers[-2:]:

layer.trainable = True

* **Fine-Tuning:** Retrain with a potentially *lower* learning rate. Experiment and observe if performance improves further.

**Key Points**

* **Transfer Learning Power:** Leveraging pre-trained knowledge significantly reduces training time and data requirements.
* **Experimentation is Key:** Explore the optimal number of layers to freeze/unfreeze for your specific task.

**3. Pretraining on an auxiliary task.**

**a. 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.**

**b. 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.**

**c. 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.**

**d. 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.**

Let's implement this auxiliary task pretraining strategy! I'll provide the code structure, explanations, and guidance.

**a. Building DNN A and DNN B**

Python

import tensorflow as tf

from tensorflow import keras

# Model definition for DNN A and DNN B

def create\_base\_model():

model = keras.models.Sequential()

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

for \_ in range(5):

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

return model

dnn\_a = create\_base\_model()

dnn\_b = create\_base\_model()

# Concatenation and Output layers

input\_a = keras.layers.Input(shape=[28, 28])

input\_b = keras.layers.Input(shape=[28, 28])

hidden\_a = dnn\_a(input\_a)

hidden\_b = dnn\_b(input\_b)

concatenated = keras.layers.concatenate([hidden\_a, hidden\_b], axis=1)

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

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

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

**b. Training Batch Generator**

Python

import numpy as np

def generate\_image\_pairs(dataset, batch\_size=32):

while True:

X1, X2, y = [], [], []

for \_ in range(batch\_size // 2):

# Same Class Pair

idx = np.random.randint(0, len(dataset))

img1, img2 = dataset[idx][0], dataset[idx][1] # Assuming the dataset structure

X1.append(img1); X2.append(img2); y.append(0)

# Different Class Pair

idx1, idx2 = np.random.randint(0, len(dataset), size=2)

while dataset[idx1][1] == dataset[idx2][1]: # Ensure different labels

idx1, idx2 = np.random.randint(0, len(dataset), size=2)

X1.append(dataset[idx1][0]); X2.append(dataset[idx2][0]); y.append(1)

yield [np.array(X1), np.array(X2)], np.array(y)

**c. Training the Siamese-like DNN**

Python

# Prepare your MNIST splits

(X\_train1, y\_train1), (\_, \_) = tf.keras.datasets.mnist.load\_data() # Adapt loading as needed

optimizer = keras.optimizers.Adam()

model.compile(loss="binary\_crossentropy", optimizer=optimizer, metrics=["accuracy"])

history = model.fit(generate\_image\_pairs(X\_train1), epochs=10, ...)

**d. Transfer Learning to MNIST Classifier**

Python

# Reuse DNN A

classifier = keras.models.Sequential()

classifier.add(dnn\_a)

for layer in classifier.layers: # Freeze the weights

layer.trainable = False

classifier.add(keras.layers.Dense(10, activation='softmax'))

# Train on split #2

classifier.compile(...)

classifier.fit(X\_train2, y\_train2, ...)

**Key Ideas**

* **Siamese-like Architecture:** We compare pairs of images for similarity detection.
* **Data as Curriculum:** The auxiliary task guides the lower layers to extract meaningful features before doing the main classification task.