1. Deep Learning.
   1. Build a DNN with five hidden layers of 100 neurons each, He initialization, and the ELU activation function.
   2. 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.
   3. Tune the hyperparameters using cross-validation and see what precision you can achieve.
   4. Now try adding Batch Normalization and compare the learning curves: is it converging faster than before? Does it produce a better model?
   5. Is the model overfitting the training set? Try adding dropout to every layer and try again. Does it help?
2. 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.
   2. 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?
   3. Try caching the frozen layers, and train the model again: how much faster is it now?
   4. Try again reusing just four hidden layers instead of five. Can you achieve a higher precision?
   5. Now unfreeze the top two hidden layers and continue training: can you get the model to perform even better?
3. 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.
   2. 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.
   3. 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.
   4. 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:

1.

a.

import tensorflow as tf

n\_inputs = 28 \* 28 # MNIST image size is 28x28

n\_hidden\_layers = 5

n\_neurons = 100

n\_outputs = 5 # we will only classify digits 0 to 4

# define the DNN architecture

model = tf.keras.models.Sequential()

# input layer

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

# hidden layers

for layer in range(n\_hidden\_layers):

model.add(tf.keras.layers.Dense(n\_neurons, activation="elu", kernel\_initializer="he\_normal"))

# output layer

model.add(tf.keras.layers.Dense(n\_outputs, activation="softmax"))

# print the model summary

model.summary()

b.

# load the MNIST dataset

mnist = tf.keras.datasets.mnist

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

# create training and validation sets

X\_valid, X\_train = X\_train\_full[:5000] / 255.0, X\_train\_full[5000:] / 255.0

y\_valid, y\_train = y\_train\_full[:5000], y\_train\_full[5000:]

# select only digits 0 to 4

idx\_train = (y\_train <= 4)

idx\_valid = (y\_valid <= 4)

idx\_test = (y\_test <= 4)

X\_train = X\_train[idx\_train]

X\_valid = X\_valid[idx\_valid]

X\_test = X\_test[idx\_test]

y\_train = y\_train[idx\_train]

y\_valid = y\_valid[idx\_valid]

y\_test = y\_test[idx\_test]

# compile the model

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

optimizer="adam",

metrics=["accuracy"])

# define early stopping callback

early\_stopping\_cb = tf.keras.callbacks.EarlyStopping(patience=10,

restore\_best\_weights=True)

# train the model

history = model.fit(X\_train, y\_train, epochs=100,

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

callbacks=[early\_stopping\_cb])

# evaluate the model on the test set

model.evaluate(X\_test, y\_test)

c.

from sklearn.model\_selection import GridSearchCV

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

# define a function to create the model

def create\_model(n\_hidden=5, n\_neurons=100, learning\_rate=1e-3):

model = tf.keras.models.Sequential()

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

for layer in range(n\_hidden):

model.add(tf.keras.layers.Dense(n\_neurons, activation="elu", kernel\_initializer="he\_normal"))

model.add(tf.keras.layers.Dense(n\_outputs, activation="softmax"))

optimizer = tf.keras.optimizers.Adam(lr=learning\_rate)

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

return model

# wrap the Keras model for use with scikit-learn

keras\_clf = KerasClassifier(create\_model)

# define the hyperparameters to tune and their possible values

param\_grid = {

"n\_hidden": [4, 5, 6],

"n\_neurons": [80, 100, 120],

"learning\_rate": [1e-3, 5e-4, 1e-4]

}

# perform grid

2.

For transfer learning, we can use a pre-trained model such as VGG, Inception, or ResNet that has been trained on a large dataset like ImageNet. You can then freeze the pre-trained layers and replace the final layer with a new output layer for your specific task.

Here are the general steps for transfer learning:

1. Load a pre-trained model and freeze the layers.
2. Replace the final layer with a new output layer.
3. Compile the model with a suitable loss function, optimizer, and evaluation metric.
4. Train the model on your specific dataset.
5. Fine-tune the model by unfreezing some of the pre-trained layers and continue training.

To achieve high precision with a small number of examples, you can use data augmentation techniques such as rotating, shifting, and flipping the images to generate more training data.

Caching the frozen layers can speed up training by reducing the computation required to propagate the input through the pre-trained layers.

Reducing the number of frozen layers can also improve performance by allowing the model to learn more from the new data.

Unfreezing some of the pre-trained layers and continuing training can also improve performance by allowing the model to adapt to the new task. However, this should be done carefully to avoid overfitting the model.

3.

a.

import tensorflow as tf

# Build DNN A

dnn\_A = tf.keras.models.Sequential([

tf.keras.layers.Dense(100, activation="elu", kernel\_initializer="he\_normal", input\_shape=(784,)),

tf.keras.layers.Dense(100, activation="elu", kernel\_initializer="he\_normal"),

tf.keras.layers.Dense(100, activation="elu", kernel\_initializer="he\_normal"),

tf.keras.layers.Dense(100, activation="elu", kernel\_initializer="he\_normal"),

tf.keras.layers.Dense(100, activation="elu", kernel\_initializer="he\_normal"),

])

# Build DNN B

dnn\_B = tf.keras.models.Sequential([

tf.keras.layers.Dense(100, activation="elu", kernel\_initializer="he\_normal", input\_shape=(784,)),

tf.keras.layers.Dense(100, activation="elu", kernel\_initializer="he\_normal"),

tf.keras.layers.Dense(100, activation="elu", kernel\_initializer="he\_normal"),

tf.keras.layers.Dense(100, activation="elu", kernel\_initializer="he\_normal"),

tf.keras.layers.Dense(100, activation="elu", kernel\_initializer="he\_normal"),

])

# Concatenate the outputs of DNN A and B

concat = tf.keras.layers.Concatenate(axis=1)([dnn\_A.output, dnn\_B.output])

# Add a hidden layer with 10 units on top of the concatenated output

hidden = tf.keras.layers.Dense(10, activation="elu", kernel\_initializer="he\_normal")(concat)

# Add an output layer with a single neuron and logistic activation function

output = tf.keras.layers.Dense(1, activation="sigmoid")(hidden)

# Build the final model

model = tf.keras.Model(inputs=[dnn\_A.input, dnn\_B.input], outputs=output)

b.

import numpy as np

def generate\_batch(X, y, batch\_size):

half\_batch = batch\_size // 2

indices = np.random.permutation(len(X))

X1 = X[indices[:half\_batch]]

X2 = X[indices[half\_batch:batch\_size]]

y1 = y[indices[:half\_batch]]

y2 = y[indices[half\_batch:batch\_size]]

labels = (y1 == y2).astype(np.float32)

labels[half\_batch:] = 1 - labels[half\_batch:]

return [X1, X2], labels

batch\_size = 32

X\_train1 = X\_train[:55000]

y\_train1 = y\_train[:55000]

n\_batches = len(X\_train1) // batch\_size

for epoch in range(10):

for batch in range(n\_batches):

X\_batch, y\_batch = generate\_batch(X\_train1, y\_train1, batch\_size)

model.train\_on\_batch(X\_batch, y\_batch)

c.

import tensorflow as tf

import numpy as np

# Define the network architecture for DNN A and B

n\_hidden = 100

n\_outputs = 1

# DNN A

dnn\_a = tf.keras.models.Sequential([

tf.keras.layers.Dense(n\_hidden, activation="elu", kernel\_initializer="he\_normal", input\_shape=(784,)),

tf.keras.layers.Dense(n\_hidden, activation="elu", kernel\_initializer="he\_normal"),

tf.keras.layers.Dense(n\_hidden, activation="elu", kernel\_initializer="he\_normal"),

tf.keras.layers.Dense(n\_hidden, activation="elu", kernel\_initializer="he\_normal"),

tf.keras.layers.Dense(n\_hidden, activation="elu", kernel\_initializer="he\_normal"),

])

# DNN B

dnn\_b = tf.keras.models.Sequential([

tf.keras.layers.Dense(n\_hidden, activation="elu", kernel\_initializer="he\_normal", input\_shape=(784,)),

tf.keras.layers.Dense(n\_hidden, activation="elu", kernel\_initializer="he\_normal"),

tf.keras.layers.Dense(n\_hidden, activation="elu", kernel\_initializer="he\_normal"),

tf.keras.layers.Dense(n\_hidden, activation="elu", kernel\_initializer="he\_normal"),

tf.keras.layers.Dense(n\_hidden, activation="elu", kernel\_initializer="he\_normal"),

])

# Concatenate the outputs of both DNNs

concat = tf.keras.layers.Concatenate(axis=1)([dnn\_a.output, dnn\_b.output])

# Add a hidden layer with 10 units

hidden = tf.keras.layers.Dense(10, activation="elu", kernel\_initializer="he\_normal")(concat)

# Add the output layer with a single neuron

output = tf.keras.layers.Dense(n\_outputs, activation="sigmoid")(hidden)

# Create the final model

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

# Compile the model

model.compile(loss="binary\_crossentropy", optimizer=tf.keras.optimizers.Adam(lr=0.01), metrics=["accuracy"])

# Load the MNIST data

mnist = tf.keras.datasets.mnist

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

# Create split #1 and split #2

X\_train\_split1, y\_train\_split1 = X\_train\_full[:55000], y\_train\_full[:55000]

X\_train\_split2, y\_train\_split2 = X\_train\_full[55000:], y\_train\_full[55000:]

# Define the batch generation function

def generate\_batch(X, y, batch\_size):

half\_batch = batch\_size // 2

while True:

indices = np.random.randint(0, len(X), half\_batch)

X1, X2 = X[indices], X[indices]

y\_batch = np.zeros((half\_batch, 1))

for i in range(half\_batch):

if np.random.rand() < 0.5:

X2[i] = np.roll(X2[i], 1, axis=1)

y\_batch[i] = 1

yield [X1, X2], y\_batch

# Train the model on split #1

batch\_size = 32

history = model.fit(generate\_batch(X\_train\_split1, y\_train\_split1, batch\_size), steps\_per\_epoch=len(X\_train\_split1)//batch\_size, epochs=10)

# Save the model

model.save("pretrained\_mnist\_model.h5")