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?

Building a DNN with five hidden layers of 100 neurons each, He initialization, and the ELU activation function:

import tensorflow as tf

from tensorflow import keras

# Define the model architecture

model = keras.models.Sequential([

keras.layers.Dense(100, activation="elu", kernel\_initializer="he\_normal", input\_shape=(28\*28,)),

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

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

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

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

keras.layers.Dense(5, activation="softmax")

])  
b. Training the model using Adam optimization and early stopping on MNIST for digits 0 to 4:

# Load the MNIST dataset

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

# Create a subset of the dataset for digits 0 to 4

train\_indices = (y\_train\_full <= 4)

test\_indices = (y\_test <= 4)

X\_train = X\_train\_full[train\_indices]

y\_train = y\_train\_full[train\_indices]

X\_test = X\_test[test\_indices]

y\_test = y\_test[test\_indices]

# Preprocess the data

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

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

# Define the checkpoint and early stopping callbacks

checkpoint\_cb = keras.callbacks.ModelCheckpoint("mnist\_model.h5", save\_best\_only=True)

early\_stopping\_cb = keras.callbacks.EarlyStopping(patience=10, restore\_best\_weights=True)

# Compile and train the model

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

history = model.fit(X\_train, y\_train, epochs=100, validation\_data=(X\_test, y\_test), callbacks=[checkpoint\_cb, early\_stopping\_cb])

Tuning hyperparameters using cross-validation:

# Use K-fold cross-validation to evaluate the model

k = 5

num\_val\_samples = len(X\_train) // k

val\_scores = []

for fold in range(k):

val\_start = fold \* num\_val\_samples

val\_end = (fold + 1) \* num\_val\_samples

val\_X = X\_train[val\_start:val\_end]

val\_y = y\_train[val\_start:val\_end]

partial\_X\_train = np.concatenate([X\_train[:val\_start], X\_train[val\_end:]], axis=0)

partial\_y\_train = np.concatenate([y\_train[:val\_start], y\_train[val\_end:]], axis=0)

model = keras.models.Sequential([...]) # Rebuild the model

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

history = model.fit(partial\_X\_train, partial\_y\_train, epochs=100, validation\_data=(val\_X, val\_y), callbacks=[checkpoint\_cb, early\_stopping\_cb])

val\_scores.append(model.evaluate(val\_X, val\_y))

mean\_val\_score = np.mean(val\_scores)

Adding Batch Normalization and comparing the learning curves:

model = keras.models.Sequential([

keras.layers.Dense(100, activation="elu", kernel\_initializer="he\_normal", input\_shape=(28\*28,)),

keras.layers.BatchNormalization(),

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

keras.layers.BatchNormalization(),

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

keras.layers.BatchNormalization(),

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

keras.layers.BatchNormalization(),

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

keras.layers.BatchNormalization(),

keras.layers.Dense(5, activation="softmax")

])

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

history = model.fit(X\_train, y\_train, epochs=100, validation\_data=(X\_test, y\_test), callbacks=[checkpoint\_cb, early\_stopping\_cb])

Adding dropout to every layer:

model = keras.models.Sequential([

keras.layers.Dense(100, activation="elu", kernel\_initializer="he\_normal", input\_shape=(28\*28,)),

keras.layers.BatchNormalization(),

keras.layers.Dropout(0.5),

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

keras.layers.BatchNormalization(),

keras.layers.Dropout(0.5),

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

keras.layers.BatchNormalization(),

keras.layers.Dropout(0.5),

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

keras.layers.BatchNormalization(),

keras.layers.Dropout(0.5),

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

keras.layers.BatchNormalization(),

keras.layers.Dropout(0.5),

keras.layers.Dense(5, activation="softmax")

])

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

history = model.fit(X\_train, y\_train, epochs=100, validation\_data=(X\_test, y\_test), callbacks=[checkpoint\_cb, early\_stopping\_cb])

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.
   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?

Creating a new DNN that reuses the pretrained hidden layers:

import tensorflow as tf

from tensorflow import keras

# Load the pretrained model

pretrained\_model = keras.models.load\_model("mnist\_model.h5")

# Freeze the pretrained hidden layers

for layer in pretrained\_model.layers:

layer.trainable = False

# Replace the softmax output layer with a new one

output\_layer = keras.layers.Dense(5, activation="softmax")

model = keras.models.Sequential(pretrained\_model.layers[:-1] + [output\_layer])

b. Training the new DNN on digits 5 to 9 with limited data:

# Load the MNIST dataset

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

# Create a subset of the dataset for digits 5 to 9

train\_indices = (y\_train\_full >= 5)

test\_indices = (y\_test >= 5)

X\_train = X\_train\_full[train\_indices][:500]

y\_train = y\_train\_full[train\_indices][:500]

X\_test = X\_test[test\_indices]

y\_test = y\_test[test\_indices]

# Preprocess the data

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

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

# Compile and train the model

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

history = model.fit(X\_train, y\_train, epochs=10, validation\_data=(X\_test, y\_test))

c. Caching the frozen layers and training the model again:

# Cache the frozen layers to speed up training

for layer in model.layers[:-1]:

layer.trainable = False

# Compile and train the model

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

history = model.fit(X\_train, y\_train, epochs=10, validation\_data=(X\_test, y\_test))

d. Reusing four hidden layers instead of five:

# Load the pretrained model

pretrained\_model = keras.models.load\_model("mnist\_model.h5")

# Freeze the top four hidden layers

for layer in pretrained\_model.layers[:-4]:

layer.trainable = False

# Replace the softmax output layer with a new one

output\_layer = keras.layers.Dense(5, activation="softmax")

model = keras.models.Sequential(pretrained\_model.layers[:-4] + [output\_layer])

# Compile and train the model

model.com # Unfreeze the top two hidden layers

for layer in model.layers[-2:]:

layer.trainable = True

# Compile and train the model

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

history = model.fit(X\_train, y\_train, epochs=10, validation\_data=(X\_test, y\_test))pile(loss="sparse\_categorical\_crossentropy", optimizer="adam", metrics=["accuracy"])

history = model.fit(X\_train, y\_train, epochs=10, validation\_data=(X\_test, y\_test))

e. Unfreezing the top two hidden layers and continuing training:

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.
   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.

Building DNN A and DNN B:

import tensorflow as tf

from tensorflow import keras

# DNN A

dnn\_a = keras.models.Sequential([

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

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

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

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

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

])

# DNN B

dnn\_b = keras.models.Sequential([

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

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

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

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

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

])

# Concatenating the outputs of DNN A and DNN B

concatenated = keras.layers.Concatenate()([dnn\_a.output, dnn\_b.output])

# Hidden layer on top of the concatenated outputs

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

# Output layer with logistic activation

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

# Create the model

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

b. Generating training batches:

import numpy as np

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

half\_batch = batch\_size // 2

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

X\_batch = []

y\_batch = []

for i in range(half\_batch):

# Pair of images from the same class

class\_idx = np.random.randint(0, 10)

class\_indices = np.where(y == class\_idx)[0]

idx1, idx2 = np.random.choice(class\_indices, size=2, replace=False)

X\_batch.append([X[idx1], X[idx2]])

y\_batch.append(0)

# Pair of images from different classes

class\_indices = np.where(y != class\_idx)[0]

idx1 = np.random.choice(class\_indices)

class\_indices = np.where(y == class\_idx)[0]

idx2 = np.random.choice(class\_indices)

X\_batch.append([X[idx1], X[idx2]])

y\_batch.append(1)

return np.array(X\_batch), np.array(y\_batch)

# Split the MNIST training set

(X\_train\_full, y\_train\_full), \_ = keras.datasets.mnist.load\_data()

X\_train1 = X\_train\_full[:55000]

y\_train1 = y\_train\_full[:55000]

X\_train2 = X\_train\_full[55000:]

y\_train2 = y\_train\_full[55000:]

# Generate training batches

X\_batch, y\_batch = generate\_batch(X\_train1, y\_train1, batch\_size=32)

c. Training the DNN on the training set:

# Preprocess the data

X\_batch = X\_batch.reshape(-1, 784) / 255.0

# Compile and train the model

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

model.fit([X\_batch[:, 0], X\_batch[:, 1]], y\_batch, epochs=10)

d. Reusing the hidden layers of DNN A:

# Freeze the hidden layers of DNN A

for layer in dnn\_a.layers:

layer.trainable = False

# Output layer with softmax activation

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

# Create the model for split #2

model2 = keras.models.Model(inputs=dnn\_a.input, outputs=output\_layer)

# Compile and train the model on split #2

model2.compile(loss="sparse\_categorical\_crossentropy", optimizer="adam", metrics=["accuracy"])

model2.fit(X\_train2.reshape(-1, 784) / 255.0, y\_train2, epochs=10)