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?**

**A.** Dropout is indeed a powerful tool in combating overfitting in neural networks. It's like giving each neuron multiple backup dancers during training so that they don't rely too much on any single one. Then, during inference, the network performs without this extra randomness, allowing for faster and more predictable predictions. It's a clever way to balance learning and generalization.

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?**
2. To create a new DNN that utilizes transfer learning:
3. **import tensorflow as tf**
4. **from tensorflow.keras import layers, models**
5. **from tensorflow.keras.datasets import mnist**
6. **# Load MNIST data**
7. **(train\_images, train\_labels), (test\_images, test\_labels) = mnist.load\_data()**
8. **# Filter out digits 5 to 9**
9. **train\_mask = (train\_labels >= 5)**
10. **test\_mask = (test\_labels >= 5)**
11. **train\_images = train\_images[train\_mask]**
12. **train\_labels = train\_labels[train\_mask] - 5**
13. **test\_images = test\_images[test\_mask]**
14. **test\_labels = test\_labels[test\_mask] - 5**
15. **# Reshape and normalize data**
16. **train\_images = train\_images.reshape((-1, 28, 28, 1)).astype('float32') / 255.0**
17. **test\_images = test\_images.reshape((-1, 28, 28, 1)).astype('float32') / 255.0**
18. **# Load pre-trained model**
19. **pretrained\_model = tf.keras.applications.VGG16(weights='imagenet', include\_top=False, input\_shape=(28, 28, 3))**
20. **# Freeze pretrained layers**
21. **for layer in pretrained\_model.layers:**
22. **layer.trainable = False**
23. **# Add new output layer**
24. **x = layers.Flatten()(pretrained\_model.output)**
25. **output = layers.Dense(5, activation='softmax')(x)**
26. **# Create new model**
27. **model = models.Model(inputs=pretrained\_model.input, outputs=output)**
28. **# Compile the model**
29. **model.compile(optimizer='adam',**
30. **loss='sparse\_categorical\_crossentropy',**
31. **metrics=['accuracy'])**
32. **# Print model summary**
33. **model.summary()**

**import tensorflow as tf**

**from tensorflow.keras import layers, models**

**from tensorflow.keras.datasets import mnist**

**# Load MNIST data**

**(train\_images, train\_labels), (test\_images, test\_labels) = mnist.load\_data()**

**# Filter out digits 5 to 9**

**train\_mask = (train\_labels >= 5)**

**test\_mask = (test\_labels >= 5)**

**train\_images = train\_images[train\_mask]**

**train\_labels = train\_labels[train\_mask] - 5**

**test\_images = test\_images[test\_mask]**

**test\_labels = test\_labels[test\_mask] - 5**

**# Reshape and normalize data**

**train\_images = train\_images.reshape((-1, 28, 28, 1)).astype('float32') / 255.0**

**test\_images = test\_images.reshape((-1, 28, 28, 1)).astype('float32') / 255.0**

**# Load pre-trained model**

**pretrained\_model = tf.keras.applications.VGG16(weights='imagenet', include\_top=False, input\_shape=(28, 28, 3))**

**# Freeze pretrained layers**

**for layer in pretrained\_model.layers:**

**layer.trainable = False**

**# Add new output layer**

**x = layers.Flatten()(pretrained\_model.output)**

**output = layers.Dense(5, activation='softmax')(x)**

**# Create new model**

**model = models.Model(inputs=pretrained\_model.input, outputs=output)**

**# Compile the model**

**model.compile(optimizer='adam',**

**loss='sparse\_categorical\_crossentropy',**

**metrics=['accuracy'])**

**# Print model summary**

**model.summary()**

**import time**

**# Define number of examples per digit**

**num\_examples\_per\_digit = 100**

**# Sample the data**

**train\_samples = []**

**for digit in range(5, 10):**

**digit\_indices = np.where(train\_labels == digit)[0][:num\_examples\_per\_digit]**

**train\_samples.extend(digit\_indices)**

**# Shuffle the samples**

**np.random.shuffle(train\_samples)**

**# Train the model**

**start\_time = time.time()**

**model.fit(train\_images[train\_samples], train\_labels[train\_samples], epochs=10, batch\_size=32, validation\_data=(test\_images, test\_labels))**

**end\_time = time.time()**

**training\_time = end\_time - start\_time**

**print("Training time:", training\_time)**

Achieving high precision with such a small number of examples might be challenging, but it's possible to obtain decent performance.

c. Caching the frozen layers and training the model again can significantly speed up the training process. Here's how you can do it:

**# Cache the frozen layers**

**pretrained\_features = pretrained\_model.predict(train\_images)**

**# Train the model again**

**start\_time = time.time()**

**model.fit(pretrained\_features[train\_samples], train\_labels[train\_samples], epochs=10, batch\_size=32, validation\_data=(test\_images, test\_labels))**

**end\_time = time.time()**

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

**print("Training time with cached features:", training\_time\_cached)**

Reusing just four hidden layers instead of five might reduce the model's capacity, potentially affecting its performance. However, it's worth experimenting with:

**# Load pre-trained model with four frozen layers**

**pretrained\_model = tf.keras.applications.VGG16(weights='imagenet', include\_top=False, input\_shape=(28, 28, 3))**

**# Freeze only four layers**

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

**layer.trainable = False**

**# Add new output layer**

**x = layers.Flatten()(pretrained\_model.output)**

**output = layers.Dense(5, activation='softmax')(x)**

**# Create new model**

**model = models.Model(inputs=pretrained\_model.input, outputs=output)**

**# Compile the model**

**model.compile(optimizer='adam',**

**loss='sparse\_categorical\_crossentropy',**

**metrics=['accuracy'])**

**# Train the model**

**start\_time = time.time()**

**model.fit(train\_images[train\_samples], train\_labels[train\_samples], epochs=10, batch\_size=32, validation\_data=(test\_images, test\_labels))**

**end\_time = time.time()**

**training\_time\_four\_layers = end\_time - start\_time**

**print("Training time with four frozen layers:", training\_time\_four\_layers)** **# Load pre-trained model with four frozen layers**

**pretrained\_model = tf.keras.applications.VGG16(weights='imagenet', include\_top=False, input\_shape=(28, 28, 3))**

**# Freeze only four layers**

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

**layer.trainable = False**

**# Add new output layer**

**x = layers.Flatten()(pretrained\_model.output)**

**output = layers.Dense(5, activation='softmax')(x)**

**# Create new model**

**model = models.Model(inputs=pretrained\_model.input, outputs=output)**

**# Compile the model**

**model.compile(optimizer='adam',**

**loss='sparse\_categorical\_crossentropy',**

**metrics=['accuracy'])**

**# Train the model**

**start\_time = time.time()**

**model.fit(train\_images[train\_samples], train\_labels[train\_samples], epochs=10, batch\_size=32, validation\_data=(test\_images, test\_labels))**

**end\_time = time.time()**

**training\_time\_four\_layers = end\_time - start\_time**

**print("Training time with four frozen layers:", training\_time\_four\_layers)** Unfreezing the top two hidden layers and continuing training might allow the model to learn more specific features related to the new task:

**# Unfreeze the top two layers**

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

**layer.trainable = True**

**# Compile the model (after unfreezing layers, it's good to recompile the model)**

**model.compile(optimizer='adam',**

**loss='sparse\_categorical\_crossentropy',**

**metrics=['accuracy'])**

**# Train the model**

**start\_time = time.time()**

**model.fit(train\_images[train\_samples], train\_labels[train\_samples], epochs=10, batch\_size=32, validation\_data=(test\_images, test\_labels))**

**end\_time = time.time()**

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

**print("Training time with unfrozen layers:", training\_time\_unfrozen\_layers)**

**a. To create a new DNN that utilizes transfer learning:**

**```python**

**import tensorflow as tf**

**from tensorflow.keras import layers, models**

**from tensorflow.keras.datasets import mnist**

**# Load MNIST data**

**(train\_images, train\_labels), (test\_images, test\_labels) = mnist.load\_data()**

**# Filter out digits 5 to 9**

**train\_mask = (train\_labels >= 5)**

**test\_mask = (test\_labels >= 5)**

**train\_images = train\_images[train\_mask]**

**train\_labels = train\_labels[train\_mask] - 5**

**test\_images = test\_images[test\_mask]**

**test\_labels = test\_labels[test\_mask] - 5**

**# Reshape and normalize data**

**train\_images = train\_images.reshape((-1, 28, 28, 1)).astype('float32') / 255.0**

**test\_images = test\_images.reshape((-1, 28, 28, 1)).astype('float32') / 255.0**

**# Load pre-trained model**

**pretrained\_model = tf.keras.applications.VGG16(weights='imagenet', include\_top=False, input\_shape=(28, 28, 3))**

**# Freeze pretrained layers**

**for layer in pretrained\_model.layers:**

**layer.trainable = False**

**# Add new output layer**

**x = layers.Flatten()(pretrained\_model.output)**

**output = layers.Dense(5, activation='softmax')(x)**

**# Create new model**

**model = models.Model(inputs=pretrained\_model.input, outputs=output)**

**# Compile the model**

**model.compile(optimizer='adam',**

**loss='sparse\_categorical\_crossentropy',**

**metrics=['accuracy'])**

**# Print model summary**

**model.summary()**

**```**

**b. Training the new DNN with only 100 images per digit:**

**```python**

**import time**

**# Define number of examples per digit**

**num\_examples\_per\_digit = 100**

**# Sample the data**

**train\_samples = []**

**for digit in range(5, 10):**

**digit\_indices = np.where(train\_labels == digit)[0][:num\_examples\_per\_digit]**

**train\_samples.extend(digit\_indices)**

**# Shuffle the samples**

**np.random.shuffle(train\_samples)**

**# Train the model**

**start\_time = time.time()**

**model.fit(train\_images[train\_samples], train\_labels[train\_samples], epochs=10, batch\_size=32, validation\_data=(test\_images, test\_labels))**

**end\_time = time.time()**

**training\_time = end\_time - start\_time**

**print("Training time:", training\_time)**

**```**

**Achieving high precision with such a small number of examples might be challenging, but it's possible to obtain decent performance.**

**c. Caching the frozen layers and training the model again can significantly speed up the training process. Here's how you can do it:**

**```python**

**# Cache the frozen layers**

**pretrained\_features = pretrained\_model.predict(train\_images)**

**# Train the model again**

**start\_time = time.time()**

**model.fit(pretrained\_features[train\_samples], train\_labels[train\_samples], epochs=10, batch\_size=32, validation\_data=(test\_images, test\_labels))**

**end\_time = time.time()**

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

**print("Training time with cached features:", training\_time\_cached)**

**```**

**d. Reusing just four hidden layers instead of five might reduce the model's capacity, potentially affecting its performance. However, it's worth experimenting with:**

**```python**

**# Load pre-trained model with four frozen layers**

**pretrained\_model = tf.keras.applications.VGG16(weights='imagenet', include\_top=False, input\_shape=(28, 28, 3))**

**# Freeze only four layers**

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

**layer.trainable = False**

**# Add new output layer**

**x = layers.Flatten()(pretrained\_model.output)**

**output = layers.Dense(5, activation='softmax')(x)**

**# Create new model**

**model = models.Model(inputs=pretrained\_model.input, outputs=output)**

**# Compile the model**

**model.compile(optimizer='adam',**

**loss='sparse\_categorical\_crossentropy',**

**metrics=['accuracy'])**

**# Train the model**

**start\_time = time.time()**

**model.fit(train\_images[train\_samples], train\_labels[train\_samples], epochs=10, batch\_size=32, validation\_data=(test\_images, test\_labels))**

**end\_time = time.time()**

**training\_time\_four\_layers = end\_time - start\_time**

**print("Training time with four frozen layers:", training\_time\_four\_layers)**

**```**

**e. Unfreezing the top two hidden layers and continuing training might allow the model to learn more specific features related to the new task:**

**```python**

**# Unfreeze the top two layers**

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

**layer.trainable = True**

**# Compile the model (after unfreezing layers, it's good to recompile the model)**

**model.compile(optimizer='adam',**

**loss='sparse\_categorical\_crossentropy',**

**metrics=['accuracy'])**

**# Train the model**

**start\_time = time.time()**

**model.fit(train\_images[train\_samples], train\_labels[train\_samples], epochs=10, batch\_size=32, validation\_data=(test\_images, test\_labels))**

**end\_time = time.time()**

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

**print("Training time with unfrozen layers:", training\_time\_unfrozen\_layers)**

**```**

**By unfreezing layers, the model might perform even better by adapting to the specific characteristics of the new 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.**
   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.**

### A. Build DNNs A and B

**import tensorflow as tf**

**# Define DNN architecture**

**n\_hidden = 100**

**n\_layers = 5**

**n\_outputs = 1**

**# Define ELU activation**

**activation = tf.nn.elu**

**# Define input placeholders**

**X\_A = tf.placeholder(tf.float32, shape=(None, 28, 28))**

**X\_B = tf.placeholder(tf.float32, shape=(None, 28, 28))**

**# Define DNN A**

**with tf.name\_scope("DNN\_A"):**

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

**X\_A = tf.layers.dense(X\_A, n\_hidden, activation=activation, kernel\_initializer=tf.keras.initializers.he\_normal())**

**output\_A = tf.layers.dense(X\_A, n\_outputs, activation=None)**

**# Define DNN B**

**with tf.name\_scope("DNN\_B"):**

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

**X\_B = tf.layers.dense(X\_B, n\_hidden, activation=activation, kernel\_initializer=tf.keras.initializers.he\_normal())**

**output\_B = tf.layers.dense(X\_B, n\_outputs, activation=None)**

**# Concatenate outputs of DNN A and DNN B**

**concatenated\_outputs = tf.concat([output\_A, output\_B], axis=1)**

**# Add additional hidden layer**

**hidden\_layer = tf.layers.dense(concatenated\_outputs, 10, activation=activation, kernel\_initializer=tf.keras.initializers.he\_normal())**

**# Output layer**

**output\_layer = tf.layers.dense(hidden\_layer, n\_outputs, activation=tf.nn.sigmoid)**

### Step 2: Generate Training Batch

**import numpy as np**

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

**batch\_X1, batch\_X2, batch\_y = [], [], []**

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

**# Select two random indices**

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

**# Append images to batch\_X1 and batch\_X2**

**batch\_X1.append(X[idx1])**

**batch\_X2.append(X[idx2])**

**# Append label based on whether images belong to same class or not**

**batch\_y.append(0 if y[idx1] == y[idx2] else 1)**

**return np.array(batch\_X1), np.array(batch\_X2), np.array(batch\_y)**

**# Example usage:**

**# batch\_X1, batch\_X2, batch\_y = generate\_batch(X\_train\_split1, y\_train\_split1, batch\_size)**

### Train the DNN

**# Define loss and optimizer**

**loss = tf.reduce\_mean(tf.nn.sigmoid\_cross\_entropy\_with\_logits(labels=tf.cast(output\_y, tf.float32), logits=logits))**

**optimizer = tf.train.AdamOptimizer(learning\_rate)**

**training\_op = optimizer.minimize(loss)**

**# Train the model**

**n\_epochs = ...**

**batch\_size = ...**

**with tf.Session() as sess:**

**sess.run(tf.global\_variables\_initializer())**

**for epoch in range(n\_epochs):**

**for iteration in range(len(X\_train\_split1) // batch\_size):**

**batch\_X1, batch\_X2, batch\_y = generate\_batch(X\_train\_split1, y\_train\_split1, batch\_size)**

**sess.run(training\_op, feed\_dict={X\_A: batch\_X1, X\_B: batch\_X2, y: batch\_y})**

**acc\_train = accuracy.eval(feed\_dict={X\_A: X\_train\_split1, X\_B: X\_train\_split1, y: y\_train\_split1})**

**print(epoch, "Train accuracy:", acc\_train)**

### Create New DNN and Train on Split #2

**# Freeze lower layers of DNN A**

**train\_vars = tf.get\_collection(tf.GraphKeys.TRAINABLE\_VARIABLES, scope="DNN\_A")**

**training\_op = optimizer.minimize(loss, var\_list=train\_vars)**

**# Define softmax output layer with 10 neurons**

**n\_outputs = 10**

**softmax\_output\_layer = tf.layers.dense(hidden\_layer, n\_outputs, activation=tf.nn.softmax)**

**# Define loss and optimizer**

**loss = tf.reduce\_mean(tf.nn.sparse\_softmax\_cross\_entropy\_with\_logits(labels=y, logits=logits))**

**optimizer = tf.train.AdamOptimizer(learning\_rate)**

**training\_op = optimizer.minimize(loss)**

**# Train the model on split #2**

**with tf.Session() as sess:**

**sess.run(tf.global\_variables\_initializer())**

**for epoch in range(n\_epochs):**

**for iteration in range(len(X\_train\_split2) // batch\_size):**

**batch\_X, batch\_y = next\_batch(X\_train\_split2, y\_train\_split2, batch\_size)**

**sess.run(training\_op, feed\_dict={X: batch\_X, y: batch\_y})**

**acc\_train = accuracy.eval(feed\_dict={X: X\_train\_split2, y: y\_train\_split2})**

**print(epoch, "Train accuracy:", acc\_train)** **Here's a step-by-step guide to accomplish the tasks described:**

**### Step 1: Build DNNs A and B**

**```python**

**import tensorflow as tf**

**# Define DNN architecture**

**n\_hidden = 100**

**n\_layers = 5**

**n\_outputs = 1**

**# Define ELU activation**

**activation = tf.nn.elu**

**# Define input placeholders**

**X\_A = tf.placeholder(tf.float32, shape=(None, 28, 28))**

**X\_B = tf.placeholder(tf.float32, shape=(None, 28, 28))**

**# Define DNN A**

**with tf.name\_scope("DNN\_A"):**

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

**X\_A = tf.layers.dense(X\_A, n\_hidden, activation=activation, kernel\_initializer=tf.keras.initializers.he\_normal())**

**output\_A = tf.layers.dense(X\_A, n\_outputs, activation=None)**

**# Define DNN B**

**with tf.name\_scope("DNN\_B"):**

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

**X\_B = tf.layers.dense(X\_B, n\_hidden, activation=activation, kernel\_initializer=tf.keras.initializers.he\_normal())**

**output\_B = tf.layers.dense(X\_B, n\_outputs, activation=None)**

**# Concatenate outputs of DNN A and DNN B**

**concatenated\_outputs = tf.concat([output\_A, output\_B], axis=1)**

**# Add additional hidden layer**

**hidden\_layer = tf.layers.dense(concatenated\_outputs, 10, activation=activation, kernel\_initializer=tf.keras.initializers.he\_normal())**

**# Output layer**

**output\_layer = tf.layers.dense(hidden\_layer, n\_outputs, activation=tf.nn.sigmoid)**

**```**

**### Step 2: Generate Training Batch**

**```python**

**import numpy as np**

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

**batch\_X1, batch\_X2, batch\_y = [], [], []**

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

**# Select two random indices**

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

**# Append images to batch\_X1 and batch\_X2**

**batch\_X1.append(X[idx1])**

**batch\_X2.append(X[idx2])**

**# Append label based on whether images belong to same class or not**

**batch\_y.append(0 if y[idx1] == y[idx2] else 1)**

**return np.array(batch\_X1), np.array(batch\_X2), np.array(batch\_y)**

**# Example usage:**

**# batch\_X1, batch\_X2, batch\_y = generate\_batch(X\_train\_split1, y\_train\_split1, batch\_size)**

**```**

**### Step 3: Train the DNN**

**```python**

**# Define loss and optimizer**

**loss = tf.reduce\_mean(tf.nn.sigmoid\_cross\_entropy\_with\_logits(labels=tf.cast(output\_y, tf.float32), logits=logits))**

**optimizer = tf.train.AdamOptimizer(learning\_rate)**

**training\_op = optimizer.minimize(loss)**

**# Train the model**

**n\_epochs = ...**

**batch\_size = ...**

**with tf.Session() as sess:**

**sess.run(tf.global\_variables\_initializer())**

**for epoch in range(n\_epochs):**

**for iteration in range(len(X\_train\_split1) // batch\_size):**

**batch\_X1, batch\_X2, batch\_y = generate\_batch(X\_train\_split1, y\_train\_split1, batch\_size)**

**sess.run(training\_op, feed\_dict={X\_A: batch\_X1, X\_B: batch\_X2, y: batch\_y})**

**acc\_train = accuracy.eval(feed\_dict={X\_A: X\_train\_split1, X\_B: X\_train\_split1, y: y\_train\_split1})**

**print(epoch, "Train accuracy:", acc\_train)**

**```**

**### Step 4: Create New DNN and Train on Split #2**

**```python**

**# Freeze lower layers of DNN A**

**train\_vars = tf.get\_collection(tf.GraphKeys.TRAINABLE\_VARIABLES, scope="DNN\_A")**

**training\_op = optimizer.minimize(loss, var\_list=train\_vars)**

**# Define softmax output layer with 10 neurons**

**n\_outputs = 10**

**softmax\_output\_layer = tf.layers.dense(hidden\_layer, n\_outputs, activation=tf.nn.softmax)**

**# Define loss and optimizer**

**loss = tf.reduce\_mean(tf.nn.sparse\_softmax\_cross\_entropy\_with\_logits(labels=y, logits=logits))**

**optimizer = tf.train.AdamOptimizer(learning\_rate)**

**training\_op = optimizer.minimize(loss)**

**# Train the model on split #2**

**with tf.Session() as sess:**

**sess.run(tf.global\_variables\_initializer())**

**for epoch in range(n\_epochs):**

**for iteration in range(len(X\_train\_split2) // batch\_size):**

**batch\_X, batch\_y = next\_batch(X\_train\_split2, y\_train\_split2, batch\_size)**

**sess.run(training\_op, feed\_dict={X: batch\_X, y: batch\_y})**

**acc\_train = accuracy.eval(feed\_dict={X: X\_train\_split2, y: y\_train\_split2})**

**print(epoch, "Train accuracy:", acc\_train)**

**```**

**This should guide you through the implementation process. Let me know if you need further clarification or assistance!**