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

Ans : We will need similar DNNs in the next exercises, so let's create a function to build this DNN:

he\_init = tf.variance\_scaling\_initializer()

def dnn(inputs, n\_hidden\_layers=5, n\_neurons=100, name=None,

activation=tf.nn.elu, initializer=he\_init):

with tf.variable\_scope(name, "dnn"):

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

inputs = tf.layers.dense(inputs, n\_neurons, activation=activation,

kernel\_initializer=initializer,

name="hidden%d" % (layer + 1))

return inputs

n\_inputs = 28 \* 28 # MNIST

n\_outputs = 5

reset\_graph()

X = tf.placeholder(tf.float32, shape=(None, n\_inputs), name="X")

y = tf.placeholder(tf.int32, shape=(None), name="y")

dnn\_outputs = dnn(X)

logits = tf.layers.dense(dnn\_outputs, n\_outputs, kernel\_initializer=he\_init, name="logits")

Y\_proba = tf.nn.softmax(logits, name="Y\_proba")

* 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.

Ans : Let's complete the graph with the cost function, the training op, and all the other usual components:

learning\_rate = 0.01

xentropy = tf.nn.sparse\_softmax\_cross\_entropy\_with\_logits(labels=y, logits=logits)

loss = tf.reduce\_mean(xentropy, name="loss")

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

training\_op = optimizer.minimize(loss, name="training\_op")

correct = tf.nn.in\_top\_k(logits, y, 1)

accuracy = tf.reduce\_mean(tf.cast(correct, tf.float32), name="accuracy")

init = tf.global\_variables\_initializer()

saver = tf.train.Saver()

Now let's create the training set, validation and test set (we need the validation set to implement early stopping):

X\_train1 = X\_train[y\_train < 5]

y\_train1 = y\_train[y\_train < 5]

X\_valid1 = X\_valid[y\_valid < 5]

y\_valid1 = y\_valid[y\_valid < 5]

X\_test1 = X\_test[y\_test < 5]

y\_test1 = y\_test[y\_test < 5]

n\_epochs = 1000

batch\_size = 20

max\_checks\_without\_progress = 20

checks\_without\_progress = 0

best\_loss = np.infty

with tf.Session() as sess:

init.run()

for epoch in range(n\_epochs):

rnd\_idx = np.random.permutation(len(X\_train1))

for rnd\_indices in np.array\_split(rnd\_idx, len(X\_train1) // batch\_size):

X\_batch, y\_batch = X\_train1[rnd\_indices], y\_train1[rnd\_indices]

sess.run(training\_op, feed\_dict={X: X\_batch, y: y\_batch})

loss\_val, acc\_val = sess.run([loss, accuracy], feed\_dict={X: X\_valid1, y: y\_valid1})

if loss\_val < best\_loss:

save\_path = saver.save(sess, "./my\_mnist\_model\_0\_to\_4.ckpt")

best\_loss = loss\_val

checks\_without\_progress = 0

else:

checks\_without\_progress += 1

if checks\_without\_progress > max\_checks\_without\_progress:

print("Early stopping!")

break

print("{}\tValidation loss: {:.6f}\tBest loss: {:.6f}\tAccuracy: {:.2f}%".format(

epoch, loss\_val, best\_loss, acc\_val \* 100))

with tf.Session() as sess:

saver.restore(sess, "./my\_mnist\_model\_0\_to\_4.ckpt")

acc\_test = accuracy.eval(feed\_dict={X: X\_test1, y: y\_test1})

print("Final test accuracy: {:.2f}%".format(acc\_test \* 100))

0 Validation loss: 0.116407 Best loss: 0.116407 Accuracy: 97.58%

1 Validation loss: 0.180534 Best loss: 0.116407 Accuracy: 97.11%

2 Validation loss: 0.227535 Best loss: 0.116407 Accuracy: 93.86%

3 Validation loss: 0.107346 Best loss: 0.107346 Accuracy: 97.54%

4 Validation loss: 0.302668 Best loss: 0.107346 Accuracy: 95.35%

5 Validation loss: 1.631054 Best loss: 0.107346 Accuracy: 22.01%

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

Ans : Let's create a DNNClassifier class, compatible with Scikit-Learn's RandomizedSearchCV class, to perform hyperparameter tuning. Here are the key points of this implementation:

the \_\_init\_\_() method (constructor) does nothing more than create instance variables for each of the hyperparameters.

the fit() method creates the graph, starts a session and trains the model:

it calls the \_build\_graph() method to build the graph (much lile the graph we defined earlier). Once this method is done creating the graph, it saves all the important operations as instance variables for easy access by other methods.

the \_dnn() method builds the hidden layers, just like the dnn() function above, but also with support for batch normalization and dropout (for the next exercises).

if the fit() method is given a validation set (X\_valid and y\_valid), then it implements early stopping. This implementation does not save the best model to disk, but rather to memory: it uses the \_get\_model\_params() method to get all the graph's variables and their values, and the \_restore\_model\_params() method to restore the variable values (of the best model found). This trick helps speed up training.

After the fit() method has finished training the model, it keeps the session open so that predictions can be made quickly, without having to save a model to disk and restore it for every prediction. You can close the session by calling the close\_session() method.

the predict\_proba() method uses the trained model to predict the class probabilities.

the predict() method calls predict\_proba() and returns the class with the highest probability, for each instance.

from sklearn.base import BaseEstimator, ClassifierMixin

from sklearn.exceptions import NotFittedError

class DNNClassifier(BaseEstimator, ClassifierMixin):

def \_\_init\_\_(self, n\_hidden\_layers=5, n\_neurons=100, optimizer\_class=tf.train.AdamOptimizer,

learning\_rate=0.01, batch\_size=20, activation=tf.nn.elu, initializer=he\_init,

batch\_norm\_momentum=None, dropout\_rate=None, random\_state=None):

"""Initialize the DNNClassifier by simply storing all the hyperparameters."""

self.n\_hidden\_layers = n\_hidden\_layers

self.n\_neurons = n\_neurons

self.optimizer\_class = optimizer\_class

self.learning\_rate = learning\_rate

self.batch\_size = batch\_size

self.activation = activation

self.initializer = initializer

self.batch\_norm\_momentum = batch\_norm\_momentum

self.dropout\_rate = dropout\_rate

self.random\_state = random\_state

self.\_session = None

def \_dnn(self, inputs):

"""Build the hidden layers, with support for batch normalization and dropout."""

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

if self.dropout\_rate:

inputs = tf.layers.dropout(inputs, self.dropout\_rate, training=self.\_training)

inputs = tf.layers.dense(inputs, self.n\_neurons,

kernel\_initializer=self.initializer,

name="hidden%d" % (layer + 1))

if self.batch\_norm\_momentum:

inputs = tf.layers.batch\_normalization(inputs, momentum=self.batch\_norm\_momentum,

training=self.\_training)

inputs = self.activation(inputs, name="hidden%d\_out" % (layer + 1))

return inputs

def \_build\_graph(self, n\_inputs, n\_outputs):

"""Build the same model as earlier"""

if self.random\_state is not None:

tf.set\_random\_seed(self.random\_state)

np.random.seed(self.random\_state)

X = tf.placeholder(tf.float32, shape=(None, n\_inputs), name="X")

y = tf.placeholder(tf.int32, shape=(None), name="y")

if self.batch\_norm\_momentum or self.dropout\_rate:

self.\_training = tf.placeholder\_with\_default(False, shape=(), name='training')

else:

self.\_training = None

dnn\_outputs = self.\_dnn(X)

logits = tf.layers.dense(dnn\_outputs, n\_outputs, kernel\_initializer=he\_init, name="logits")

Y\_proba = tf.nn.softmax(logits, name="Y\_proba")

xentropy = tf.nn.sparse\_softmax\_cross\_entropy\_with\_logits(labels=y,

logits=logits)

loss = tf.reduce\_mean(xentropy, name="loss")

optimizer = self.optimizer\_class(learning\_rate=self.learning\_rate)

training\_op = optimizer.minimize(loss)

correct = tf.nn.in\_top\_k(logits, y, 1)

accuracy = tf.reduce\_mean(tf.cast(correct, tf.float32), name="accuracy")

init = tf.global\_variables\_initializer()

saver = tf.train.Saver()

# Make the important operations available easily through instance variables

self.\_X, self.\_y = X, y

self.\_Y\_proba, self.\_loss = Y\_proba, loss

self.\_training\_op, self.\_accuracy = training\_op, accuracy

self.\_init, self.\_saver = init, saver

def close\_session(self):

if self.\_session:

self.\_session.close()

def \_get\_model\_params(self):

"""Get all variable values (used for early stopping, faster than saving to disk)"""

with self.\_graph.as\_default():

gvars = tf.get\_collection(tf.GraphKeys.GLOBAL\_VARIABLES)

return {gvar.op.name: value for gvar, value in zip(gvars, self.\_session.run(gvars))}

def \_restore\_model\_params(self, model\_params):

"""Set all variables to the given values (for early stopping, faster than loading from disk)"""

gvar\_names = list(model\_params.keys())

assign\_ops = {gvar\_name: self.\_graph.get\_operation\_by\_name(gvar\_name + "/Assign")

for gvar\_name in gvar\_names}

init\_values = {gvar\_name: assign\_op.inputs[1] for gvar\_name, assign\_op in assign\_ops.items()}

feed\_dict = {init\_values[gvar\_name]: model\_params[gvar\_name] for gvar\_name in gvar\_names}

self.\_session.run(assign\_ops, feed\_dict=feed\_dict)

def fit(self, X, y, n\_epochs=100, X\_valid=None, y\_valid=None):

"""Fit the model to the training set. If X\_valid and y\_valid are provided, use early stopping."""

self.close\_session()

# infer n\_inputs and n\_outputs from the training set.

n\_inputs = X.shape[1]

self.classes\_ = np.unique(y)

n\_outputs = len(self.classes\_)

# Translate the labels vector to a vector of sorted class indices, containing

# integers from 0 to n\_outputs - 1.

# For example, if y is equal to [8, 8, 9, 5, 7, 6, 6, 6], then the sorted class

# labels (self.classes\_) will be equal to [5, 6, 7, 8, 9], and the labels vector

# will be translated to [3, 3, 4, 0, 2, 1, 1, 1]

self.class\_to\_index\_ = {label: index

for index, label in enumerate(self.classes\_)}

y = np.array([self.class\_to\_index\_[label]

for label in y], dtype=np.int32)

self.\_graph = tf.Graph()

with self.\_graph.as\_default():

self.\_build\_graph(n\_inputs, n\_outputs)

# extra ops for batch normalization

extra\_update\_ops = tf.get\_collection(tf.GraphKeys.UPDATE\_OPS)

# needed in case of early stopping

max\_checks\_without\_progress = 20

checks\_without\_progress = 0

best\_loss = np.infty

best\_params = None

# Now train the model!

self.\_session = tf.Session(graph=self.\_graph)

with self.\_session.as\_default() as sess:

self.\_init.run()

for epoch in range(n\_epochs):

rnd\_idx = np.random.permutation(len(X))

for rnd\_indices in np.array\_split(rnd\_idx, len(X) // self.batch\_size):

X\_batch, y\_batch = X[rnd\_indices], y[rnd\_indices]

feed\_dict = {self.\_X: X\_batch, self.\_y: y\_batch}

if self.\_training is not None:

feed\_dict[self.\_training] = True

sess.run(self.\_training\_op, feed\_dict=feed\_dict)

if extra\_update\_ops:

sess.run(extra\_update\_ops, feed\_dict=feed\_dict)

if X\_valid is not None and y\_valid is not None:

loss\_val, acc\_val = sess.run([self.\_loss, self.\_accuracy],

feed\_dict={self.\_X: X\_valid,

self.\_y: y\_valid})

if loss\_val < best\_loss:

best\_params = self.\_get\_model\_params()

best\_loss = loss\_val

checks\_without\_progress = 0

else:

checks\_without\_progress += 1

print("{}\tValidation loss: {:.6f}\tBest loss: {:.6f}\tAccuracy: {:.2f}%".format(

epoch, loss\_val, best\_loss, acc\_val \* 100))

if checks\_without\_progress > max\_checks\_without\_progress:

print("Early stopping!")

break

else:

loss\_train, acc\_train = sess.run([self.\_loss, self.\_accuracy],

feed\_dict={self.\_X: X\_batch,

self.\_y: y\_batch})

print("{}\tLast training batch loss: {:.6f}\tAccuracy: {:.2f}%".format(

epoch, loss\_train, acc\_train \* 100))

# If we used early stopping then rollback to the best model found

if best\_params:

self.\_restore\_model\_params(best\_params)

return self

def predict\_proba(self, X):

if not self.\_session:

raise NotFittedError("This %s instance is not fitted yet" % self.\_\_class\_\_.\_\_name\_\_)

with self.\_session.as\_default() as sess:

return self.\_Y\_proba.eval(feed\_dict={self.\_X: X})

def predict(self, X):

class\_indices = np.argmax(self.predict\_proba(X), axis=1)

return np.array([[self.classes\_[class\_index]]

for class\_index in class\_indices], np.int32)

def save(self, path):

self.\_saver.save(self.\_session, path)

Let's see if we get the exact same accuracy as earlier using this class (without dropout or batch norm):

dnn\_clf = DNNClassifier(random\_state=42)

dnn\_clf.fit(X\_train1, y\_train1, n\_epochs=1000, X\_valid=X\_valid1, y\_valid=y\_valid1)

0 Validation loss: 0.116407 Best loss: 0.116407 Accuracy: 97.58%

1 Validation loss: 0.180534 Best loss: 0.116407 Accuracy: 97.11%

2 Validation loss: 0.227535 Best loss: 0.116407 Accuracy: 93.86%

3 Validation loss: 0.107346 Best loss: 0.107346 Accuracy: 97.54%

4 Validation loss: 0.302668 Best loss: 0.107346 Accuracy: 95.35%

5 Validation loss: 1.631054 Best loss: 0.107346 Accuracy: 22.01%

6 Validation loss: 1.635262 Best loss: 0.107346 Accuracy: 18.73%

7 Validation loss: 1.671200 Best loss: 0.107346 Accuracy: 22.01%

8 Validation loss: 1.695277 Best loss: 0.107346 Accuracy: 19.27%

9 Validation loss: 1.744607 Best loss: 0.107346 Accuracy: 20.91%

10 Validation loss: 1.629857 Best loss: 0.107346 Accuracy: 22.01%

11 Validation loss: 1.810803 Best loss: 0.107346 Accuracy: 22.01%

12 Validation loss: 1.675703 Best loss: 0.107346 Accuracy: 18.73%

13 Validation loss: 1.633233 Best loss: 0.107346 Accuracy: 20.91%

14 Validation loss: 1.652905 Best loss: 0.107346 Accuracy: 20.91%

15 Validation loss: 1.635937 Best loss: 0.107346 Accuracy: 20.91%

16 Validation loss: 1.718919 Best loss: 0.107346 Accuracy: 19.08%

17 Validation loss: 1.682458 Best loss: 0.107346 Accuracy: 19.27%

DNNClassifier(activation=,

batch\_norm\_momentum=None, batch\_size=20, dropout\_rate=None,

initializer=,

learning\_rate=0.01, n\_hidden\_layers=5, n\_neurons=100,

optimizer\_class=,

random\_state=42)

The model is trained, let's see if it gets the same accuracy as earlier:

from sklearn.metrics import accuracy\_score

y\_pred = dnn\_clf.predict(X\_test1)

accuracy\_score(y\_test1, y\_pred)

0.9725627553998832

Yep! Working fine. Now we can use Scikit-Learn's RandomizedSearchCV class to search for better hyperparameters (this may take over an hour, depending on your system):

from sklearn.model\_selection import RandomizedSearchCV

def leaky\_relu(alpha=0.01):

def parametrized\_leaky\_relu(z, name=None):

return tf.maximum(alpha \* z, z, name=name)

return parametrized\_leaky\_relu

param\_distribs = {

"n\_neurons": [10, 30, 50, 70, 90, 100, 120, 140, 160],

"batch\_size": [10, 50, 100, 500],

"learning\_rate": [0.01, 0.02, 0.05, 0.1],

"activation": [tf.nn.relu, tf.nn.elu, leaky\_relu(alpha=0.01), leaky\_relu(alpha=0.1)],

# you could also try exploring different numbers of hidden layers, different optimizers, etc.

#"n\_hidden\_layers": [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10],

#"optimizer\_class": [tf.train.AdamOptimizer, partial(tf.train.MomentumOptimizer, momentum=0.95)],

}

rnd\_search = RandomizedSearchCV(DNNClassifier(random\_state=42), param\_distribs, n\_iter=50,

cv=3, random\_state=42, verbose=2)

rnd\_search.fit(X\_train1, y\_train1, X\_valid=X\_valid1, y\_valid=y\_valid1, n\_epochs=1000)

# If you have Scikit-Learn 0.18 or earlier, you should upgrade, or use the fit\_params argument:

# fit\_params = dict(X\_valid=X\_valid1, y\_valid=y\_valid1, n\_epochs=1000)

# rnd\_search = RandomizedSearchCV(DNNClassifier(random\_state=42), param\_distribs, n\_iter=50,

# fit\_params=fit\_params, random\_state=42, verbose=2)

# rnd\_search.fit(X\_train1, y\_train1)

Fitting 3 folds for each of 50 candidates, totalling 150 fits

[CV] n\_neurons=10, learning\_rate=0.05, batch\_size=100, activation=

[Parallel(n\_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.

0 Validation loss: 0.143224 Best loss: 0.143224 Accuracy: 95.82%

1 Validation loss: 0.143304 Best loss: 0.143224 Accuracy: 96.60%

2 Validation loss: 0.106488 Best loss: 0.106488 Accuracy: 96.95%

3 Validation loss: 0.307107 Best loss: 0.106488 Accuracy: 92.34%

4 Validation loss: 0.157948 Best loss: 0.106488 Accuracy: 95.50%

5 Validation loss: 0.131002 Best loss: 0.106488 Accuracy: 96.40%

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

Ans : Let's train the best model found, once again, to see how fast it converges (alternatively, you could tweak the code above to make it write summaries for TensorBoard, so you can visualize the learning curve):

dnn\_clf = DNNClassifier(activation=leaky\_relu(alpha=0.1), batch\_size=500, learning\_rate=0.01,

n\_neurons=140, random\_state=42)

dnn\_clf.fit(X\_train1, y\_train1, n\_epochs=1000, X\_valid=X\_valid1, y\_valid=y\_valid1)

0 Validation loss: 0.083541 Best loss: 0.083541 Accuracy: 97.54%

1 Validation loss: 0.052198 Best loss: 0.052198 Accuracy: 98.40%

2 Validation loss: 0.044553 Best loss: 0.044553 Accuracy: 98.71%

3 Validation loss: 0.051113 Best loss: 0.044553 Accuracy: 98.48%

4 Validation loss: 0.046304 Best loss: 0.044553 Accuracy: 98.75%

5 Validation loss: 0.037796 Best loss: 0.037796 Accuracy: 98.91%

6 Validation loss: 0.048525 Best loss: 0.037796 Accuracy: 98.67%

7 Validation loss: 0.039877 Best loss: 0.037796 Accuracy: 98.75%

8 Validation loss: 0.038729 Best loss: 0.037796 Accuracy: 98.98%

9 Validation loss: 0.064167 Best loss: 0.037796 Accuracy: 98.24%

10 Validation loss: 0.057274 Best loss: 0.037796 Accuracy: 98.79%

DNNClassifier(activation=.parametrized\_leaky\_relu at 0x13e25ea60>,

batch\_norm\_momentum=None, batch\_size=500, dropout\_rate=None,

initializer=,

learning\_rate=0.01, n\_hidden\_layers=5, n\_neurons=140,

optimizer\_class=,

random\_state=42)

The best loss is reached at epoch 5.

Let's check that we do indeed get 98.9% accuracy on the test set:

y\_pred = dnn\_clf.predict(X\_test1)

accuracy\_score(y\_test1, y\_pred)

0.9898812998637867

Good, now let's use the exact same model, but this time with batch normalization:

dnn\_clf\_bn = DNNClassifier(activation=leaky\_relu(alpha=0.1), batch\_size=500, learning\_rate=0.01,

n\_neurons=90, random\_state=42,

batch\_norm\_momentum=0.95)

dnn\_clf\_bn.fit(X\_train1, y\_train1, n\_epochs=1000, X\_valid=X\_valid1, y\_valid=y\_valid1)

0 Validation loss: 0.046685 Best loss: 0.046685 Accuracy: 98.63%

1 Validation loss: 0.040820 Best loss: 0.040820 Accuracy: 98.79%

2 Validation loss: 0.046557 Best loss: 0.040820 Accuracy: 98.67%

3 Validation loss: 0.032236 Best loss: 0.032236 Accuracy: 98.94%

4 Validation loss: 0.056148 Best loss: 0.032236 Accuracy: 98.44%

5 Validation loss: 0.035988 Best loss: 0.032236 Accuracy: 98.98%

6 Validation loss: 0.037958 Best loss: 0.032236 Accuracy: 98.94%

7 Validation loss: 0.034588 Best loss: 0.032236 Accuracy: 99.02%

8 Validation loss: 0.031261 Best loss: 0.031261 Accuracy: 99.34%

9 Validation loss: 0.050791 Best loss: 0.031261 Accuracy: 98.79%

10 Validation loss: 0.035324 Best loss: 0.031261 Accuracy: 99.02%

The best params are reached during epoch 20, that's actually a slower convergence than earlier. Let's check the accuracy:

y\_pred = dnn\_clf\_bn.predict(X\_test1)

accuracy\_score(y\_test1, y\_pred)

0.9941622883829538

Great, batch normalization improved accuracy! Let's see if we can find a good set of hyperparameters that will work even better with batch normalization:

from sklearn.model\_selection import RandomizedSearchCV

param\_distribs = {

"n\_neurons": [10, 30, 50, 70, 90, 100, 120, 140, 160],

"batch\_size": [10, 50, 100, 500],

"learning\_rate": [0.01, 0.02, 0.05, 0.1],

"activation": [tf.nn.relu, tf.nn.elu, leaky\_relu(alpha=0.01), leaky\_relu(alpha=0.1)],

# you could also try exploring different numbers of hidden layers, different optimizers, etc.

#"n\_hidden\_layers": [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10],

#"optimizer\_class": [tf.train.AdamOptimizer, partial(tf.train.MomentumOptimizer, momentum=0.95)],

"batch\_norm\_momentum": [0.9, 0.95, 0.98, 0.99, 0.999],

}

rnd\_search\_bn = RandomizedSearchCV(DNNClassifier(random\_state=42), param\_distribs, n\_iter=50, cv=3,

random\_state=42, verbose=2)

rnd\_search\_bn.fit(X\_train1, y\_train1, X\_valid=X\_valid1, y\_valid=y\_valid1, n\_epochs=1000)

# If you have Scikit-Learn 0.18 or earlier, you should upgrade, or use the fit\_params argument:

# fit\_params = dict(X\_valid=X\_valid1, y\_valid=y\_valid1, n\_epochs=1000)

# rnd\_search\_bn = RandomizedSearchCV(DNNClassifier(random\_state=42), param\_distribs, n\_iter=50,

# fit\_params=fit\_params, random\_state=42, verbose=2)

# rnd\_search\_bn.fit(X\_train1, y\_train1)

Fitting 3 folds for each of 50 candidates, totalling 150 fits

[CV] n\_neurons=70, learning\_rate=0.01, batch\_size=50, batch\_norm\_momentum=0.99, activation=

[Parallel(n\_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.

0 Validation loss: 0.098522 Best loss: 0.098522 Accuracy: 97.81%

1 Validation loss: 0.080233 Best loss: 0.080233 Accuracy: 98.08%

2 Validation loss: 0.068767 Best loss: 0.068767 Accuracy: 98.01%

3 Validation loss: 0.057095 Best loss: 0.057095 Accuracy: 98.28%

4 Validation loss: 0.067008 Best loss: 0.057095 Accuracy: 98.12%

5 Validation loss: 0.058910 Best loss: 0.057095 Accuracy: 98.55%

6 Validation loss: 0.038421 Best loss: 0.038421 Accuracy: 98.91%

7 Validation loss: 0.071075 Best loss: 0.038421 Accuracy: 98.36%

8 Validation loss: 0.063073 Best loss: 0.038421 Accuracy: 98.28%

RandomizedSearchCV(cv='warn', error\_score='raise-deprecating',

estimator=DNNClassifier(activation=,

batch\_norm\_momentum=None, batch\_size=20, dropout\_rate=None,

initializer=,

learning\_rate=0.01, n\_hidden\_layers=5, n\_neurons=100,

optimizer\_class=,

random\_state=42),

fit\_params={'X\_valid': array([[0., 0., ..., 0., 0.],

[0., 0., ..., 0., 0.],

...,

[0., 0., ..., 0., 0.],

[0., 0., ..., 0., 0.]], dtype=float32), 'y\_valid': array([0, 4, ..., 1, 2], dtype=int32), 'n\_epochs': 1000},

iid='warn', n\_iter=50, n\_jobs=None,

param\_distributions={'n\_neurons': [10, 30, 50, 70, 90, 100, 120, 140, 160], 'batch\_size': [10, 50, 100, 500], 'learning\_rate': [0.01, 0.02, 0.05, 0.1], 'activation': [, , .parametrized\_leaky\_relu at 0x1500bd2f0>, .parametrized\_leaky\_relu at 0x1500bd378>], 'batch\_norm\_momentum': [0.9, 0.95, 0.98, 0.99, 0.999]},

pre\_dispatch='2\*n\_jobs', random\_state=42, refit=True,

return\_train\_score='warn', scoring=None, verbose=2)

rnd\_search\_bn.best\_params\_

{'n\_neurons': 160,

'learning\_rate': 0.01,

'batch\_size': 10,

'batch\_norm\_momentum': 0.98,

'activation': }

y\_pred = rnd\_search\_bn.predict(X\_test1)

accuracy\_score(y\_test1, y\_pred)

0.9949406499318934

Slightly better than earlier: 99.49% vs 99.42%. Let's see if dropout can do better.

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

Ans : Let's go back to the model we trained earlier and see how it performs on the training set:

y\_pred = dnn\_clf.predict(X\_train1)

accuracy\_score(y\_train1, y\_pred)

0.9950781082816178

The model performs significantly better on the training set than on the test set (99.51% vs 99.00%), which means it is overfitting the training set. A bit of regularization may help. Let's try adding dropout with a 50% dropout rate:

dnn\_clf\_dropout = DNNClassifier(activation=leaky\_relu(alpha=0.1), batch\_size=500, learning\_rate=0.01,

n\_neurons=90, random\_state=42,

dropout\_rate=0.5)

dnn\_clf\_dropout.fit(X\_train1, y\_train1, n\_epochs=1000, X\_valid=X\_valid1, y\_valid=y\_valid1)

0 Validation loss: 0.131152 Best loss: 0.131152 Accuracy: 96.91%

1 Validation loss: 0.105306 Best loss: 0.105306 Accuracy: 97.46%

2 Validation loss: 0.091219 Best loss: 0.091219 Accuracy: 97.73%

3 Validation loss: 0.089638 Best loss: 0.089638 Accuracy: 97.85%

4 Validation loss: 0.091288 Best loss: 0.089638 Accuracy: 97.69%

5 Validation loss: 0.081112 Best loss: 0.081112 Accuracy: 98.05%

6 Validation loss: 0.075575 Best loss: 0.075575 Accuracy: 98.24%

7 Validation loss: 0.084841 Best loss: 0.075575 Accuracy: 97.77%

8 Validation loss: 0.075269 Best loss: 0.075269 Accuracy: 97.65%

9 Validation loss: 0.076625 Best loss: 0.075269 Accuracy: 98.12%

10 Validation loss: 0.072509 Best loss: 0.072509 Accuracy: 97.97%

11 Validation loss: 0.071006 Best loss: 0.071006 Accuracy: 98.44%

12 Validation loss: 0.073272 Best loss: 0.071006 Accuracy: 98.08%

13 Validation loss: 0.076293 Best loss: 0.071006 Accuracy: 98.16%

14 Validation loss: 0.074955 Best loss: 0.071006 Accuracy: 98.05%

15 Validation loss: 0.066207 Best loss: 0.066207 Accuracy: 98.20%

16 Validation loss: 0.067388 Best loss: 0.066207 Accuracy: 98.08%

17 Validation loss: 0.061916 Best loss: 0.061916 Accuracy: 98.40%

18 Validation loss: 0.064908 Best loss: 0.061916 Accuracy: 98.40%

19 Validation loss: 0.064921 Best loss: 0.061916 Accuracy: 98.40%

20 Validation loss: 0.069939 Best loss: 0.061916 Accuracy: 98.40

batch\_norm\_momentum=None, batch\_size=500, dropout\_rate=0.5,

initializer=,

learning\_rate=0.01, n\_hidden\_layers=5, n\_neurons=90,

optimizer\_class=,

random\_state=42)

The best params are reached during epoch 17. Dropout somewhat slowed down convergence.

Let's check the accuracy:

y\_pred = dnn\_clf\_dropout.predict(X\_test1)

accuracy\_score(y\_test1, y\_pred)

0.9861840825063242

We are out of luck, dropout does not seem to help. Let's try tuning the hyperparameters, perhaps we can squeeze a bit more performance out of this model:

from sklearn.model\_selection import RandomizedSearchCV

param\_distribs = {

"n\_neurons": [10, 30, 50, 70, 90, 100, 120, 140, 160],

"batch\_size": [10, 50, 100, 500],

"learning\_rate": [0.01, 0.02, 0.05, 0.1],

"activation": [tf.nn.relu, tf.nn.elu, leaky\_relu(alpha=0.01), leaky\_relu(alpha=0.1)],

# you could also try exploring different numbers of hidden layers, different optimizers, etc.

#"n\_hidden\_layers": [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10],

#"optimizer\_class": [tf.train.AdamOptimizer, partial(tf.train.MomentumOptimizer, momentum=0.95)],

"dropout\_rate": [0.2, 0.3, 0.4, 0.5, 0.6],

}

rnd\_search\_dropout = RandomizedSearchCV(DNNClassifier(random\_state=42), param\_distribs, n\_iter=50,

cv=3, random\_state=42, verbose=2)

rnd\_search\_dropout.fit(X\_train1, y\_train1, X\_valid=X\_valid1, y\_valid=y\_valid1, n\_epochs=1000)

# If you have Scikit-Learn 0.18 or earlier, you should upgrade, or use the fit\_params argument:

# fit\_params = dict(X\_valid=X\_valid1, y\_valid=y\_valid1, n\_epochs=1000)

# rnd\_search\_dropout = RandomizedSearchCV(DNNClassifier(random\_state=42), param\_distribs, n\_iter=50,

# fit\_params=fit\_params, random\_state=42, verbose=2)

# rnd\_search\_dropout.fit(X\_train1, y\_train1)

[Parallel(n\_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.

Fitting 3 folds for each of 50 candidates, totalling 150 fits

[CV] n\_neurons=70, learning\_rate=0.01, dropout\_rate=0.5, batch\_size=100, activation=

0 Validation loss: 0.218595 Best loss: 0.218595 Accuracy: 93.63%

1 Validation loss: 0.210470 Best loss: 0.210470 Accuracy: 94.61%

2 Validation loss: 0.224635 Best loss: 0.210470 Accuracy: 95.50%

3 Validation loss: 0.200494 Best loss: 0.200494 Accuracy: 94.84%

4 Validation loss: 0.184056 Best loss: 0.184056 Accuracy: 95.58%

5 Validation loss: 0.187698 Best loss: 0.184056 Accuracy: 96.33%

6 Validation loss: 0.151692 Best loss: 0.151692 Accuracy: 96.17%

7 Validation loss: 0.176633 Best loss: 0.151692 Accuracy: 96.21%

8 Validation loss: 0.187090 Best loss: 0.151692 Accuracy: 96.01%

9 Validation loss: 0.204406 Best loss: 0.151692 Accuracy: 96.40%

10 Validation loss: 0.193938 Best loss: 0.151692 Accuracy: 95.74%

11 Validation loss: 0.190056 Best loss: 0.151692 Accuracy: 96.21%

12 Validation loss: 0.183601 Best loss: 0.151692 Accuracy: 96.05%

13 Validation loss: 0.179737 Best loss: 0.151692 Accuracy: 96.25%

RandomizedSearchCV(cv='warn', error\_score='raise-deprecating',

estimator=DNNClassifier(activation=,

batch\_norm\_momentum=None, batch\_size=20, dropout\_rate=None,

initializer=,

learning\_rate=0.01, n\_hidden\_layers=5, n\_neurons=100,

optimizer\_class=,

random\_state=42),

fit\_params={'X\_valid': array([[0., 0., ..., 0., 0.],

[0., 0., ..., 0., 0.],

...,

[0., 0., ..., 0., 0.],

[0., 0., ..., 0., 0.]], dtype=float32), 'y\_valid': array([0, 4, ..., 1, 2], dtype=int32), 'n\_epochs': 1000},

iid='warn', n\_iter=50, n\_jobs=None,

param\_distributions={'n\_neurons': [10, 30, 50, 70, 90, 100, 120, 140, 160], 'batch\_size': [10, 50, 100, 500], 'learning\_rate': [0.01, 0.02, 0.05, 0.1], 'activation': [, , .parametrized\_leaky\_relu at 0x14e850620>, .parametrized\_leaky\_relu at 0x14e850d08>], 'dropout\_rate': [0.2, 0.3, 0.4, 0.5, 0.6]},

pre\_dispatch='2\*n\_jobs', random\_state=42, refit=True,

return\_train\_score='warn', scoring=None, verbose=2)

rnd\_search\_dropout.best\_params\_

{'n\_neurons': 160,

'learning\_rate': 0.01,

'dropout\_rate': 0.2,

'batch\_size': 100,

'activation': }

y\_pred = rnd\_search\_dropout.predict(X\_test1)

accuracy\_score(y\_test1, y\_pred)

0.9889083479276124

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.

Ans : Let's load the best model's graph and get a handle on all the important operations we will need. Note that instead of creating a new softmax output layer, we will just reuse the existing one (since it has the same number of outputs as the existing one). We will reinitialize its parameters before training.

reset\_graph()

restore\_saver = tf.train.import\_meta\_graph("./my\_best\_mnist\_model\_0\_to\_4.meta")

X = tf.get\_default\_graph().get\_tensor\_by\_name("X:0")

y = tf.get\_default\_graph().get\_tensor\_by\_name("y:0")

loss = tf.get\_default\_graph().get\_tensor\_by\_name("loss:0")

Y\_proba = tf.get\_default\_graph().get\_tensor\_by\_name("Y\_proba:0")

logits = Y\_proba.op.inputs[0]

accuracy = tf.get\_default\_graph

To freeze the lower layers, we will exclude their variables from the optimizer's list of trainable variables, keeping only the output layer's trainable variables:

learning\_rate = 0.01

output\_layer\_vars = tf.get\_collection(tf.GraphKeys.TRAINABLE\_VARIABLES, scope="logits")

optimizer = tf.train.AdamOptimizer(learning\_rate, name="Adam2")

training\_op = optimizer.minimize(loss, var\_list=

correct = tf.nn.in\_top\_k(logits, y, 1)

accuracy = tf.reduce\_mean(tf.cast(correct, tf.float32), name="accuracy")

init = tf.global\_variables\_initializer()

five\_frozen\_saver = tf.train.Saver()

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?

Ans : Let's create the training, validation and test sets. We need to subtract 5 from the labels because TensorFlow expects integers from 0 to n\_classes-1.

X\_train2\_full = X\_train[y\_train >= 5]

y\_train2\_full = y\_train[y\_train >= 5] - 5

X\_valid2\_full = X\_valid[y\_valid >= 5]

y\_valid2\_full = y\_valid[y\_valid >= 5] - 5

X\_test2 = X\_test[y\_test >= 5]

y\_test2 = y\_test[y\_test >= 5] - 5

Also, for the purpose of this exercise, we want to keep only 100 instances per class in the training set (and let's keep only 30 instances per class in the validation set). Let's create a small function to do that:

def sample\_n\_instances\_per\_class(X, y, n=100):

Xs, ys = [], []

for label in np.unique(y):

idx = (y == label)

Xc = X[idx][:n]

yc = y[idx][:n]

Xs.append(Xc)

ys.append(yc)

return np.concatenate(Xs), np.concatenate(ys)

X\_train2, y\_train2 = sample\_n\_instances\_per\_class(X\_train2\_full, y\_train2\_full, n=100)

X\_valid2, y\_valid2 = sample\_n\_instances\_per\_class(X\_valid2\_full, y\_valid2\_full, n=30

Now let's train the model. This is the same training code as earlier, using early stopping, except for the initialization: we first initialize all the variables, then we restore the best model trained earlier (on digits 0 to 4), and finally we reinitialize the output layer variables.

import time

n\_epochs = 1000

batch\_size = 20

max\_checks\_without\_progress = 20

checks\_without\_progress = 0

best\_loss = np.infty

with tf.Session() as sess:

init.run()

restore\_saver.restore(sess, "./my\_best\_mnist\_model\_0\_to\_4")

t0 = time.time()

for epoch in range(n\_epochs):

rnd\_idx = np.random.permutation(len(X\_train2))

for rnd\_indices in np.array\_split(rnd\_idx, len(X\_train2) // batch\_size):

X\_batch, y\_batch = X\_train2[rnd\_indices], y\_train2[rnd\_indices]

sess.run(training\_op, feed\_dict={X: X\_batch, y: y\_batch})

loss\_val, acc\_val = sess.run([loss, accuracy], feed\_dict={X: X\_valid2, y: y\_valid2})

if loss\_val < best\_loss:

save\_path = five\_frozen\_saver.save(sess, "./my\_mnist\_model\_5\_to\_9\_five\_frozen")

best\_loss = loss\_val

checks\_without\_progress = 0

else:

checks\_without\_progress += 1

if checks\_without\_progress > max\_checks\_without\_progress:

print("Early stopping!")

break

print("{}\tValidation loss: {:.6f}\tBest loss: {:.6f}\tAccuracy: {:.2f}%".format(

epoch, loss\_val, best\_loss, acc\_val \* 100))

t1 = time.time()

print("Total training time: {:.1f}s".format(t1 - t0))

with tf.Session() as sess:

five\_frozen\_saver.restore(sess, "./my\_mnist\_model\_5\_to\_9\_five\_frozen")

acc\_test = accuracy.eval(feed\_dict={X: X\_test2, y: y\_test2})

print("Final test accuracy: {:.2f}%".format(acc\_test \* 100))

INFO:tensorflow:Restoring parameters from ./my\_best\_mnist\_model\_0\_to\_4

0 Validation loss: 1.361167 Best loss: 1.361167 Accuracy: 43.33%

1 Validation loss: 1.154602 Best loss: 1.154602 Accuracy: 57.33%

2 Validation loss: 1.054218 Best loss: 1.054218 Accuracy: 53.33%

3 Validation loss: 0.981128 Best loss: 0.981128 Accuracy: 62.67%

4 Validation loss: 0.995353 Best loss: 0.981128 Accuracy: 59.33%

5 Validation loss: 0.967000 Best loss: 0.967000 Accuracy: 65.33%

6 Validation loss: 0.955700 Best loss: 0.955700 Accuracy: 61.33%

7 Validation loss: 1.015331 Best loss: 0.955700 Accuracy: 58.67%

8 Validation loss: 0.978280 Best loss: 0.955700 Accuracy: 62.00%

9 Validation loss: 0.923389 Best loss: 0.923389 Accuracy: 69.33%

10 Validation loss: 0.996236 Best loss: 0.923389 Accuracy: 63.33%

11 Validation loss: 0.976757 Best loss: 0.923389 Accuracy: 62.67%

12 Validation loss: 0.969096 Best loss: 0.923389 Accuracy: 63.33%

13 Validation loss: 1.023069 Best loss: 0.923389 Accuracy: 63.33%

14 Validation loss: 1.104664 Best loss: 0.923389 Accuracy: 55.33%

15 Validation loss: 0.950175 Best loss: 0.923389 Accuracy: 65.33%

16 Validation loss: 1.002944 Best loss: 0.923389 Accuracy: 63.33%

17 Validation loss: 0.895543 Best loss: 0.895543 Accuracy: 70.67%

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

Ans : Let's start by getting a handle on the output of the last frozen layer:

hidden5\_out = tf.get\_default\_graph().get\_tensor\_by\_name("hidden5\_out:0")

Now let's train the model using roughly the same code as earlier. The difference is that we compute the output of the top frozen layer at the beginning (both for the training set and the validation set), and we cache it. This makes training roughly 1.5 to 3 times faster in this example (this may vary greatly, depending on your system):

import time

n\_epochs = 1000

batch\_size = 20

max\_checks\_without\_progress = 20

checks\_without\_progress = 0

best\_loss = np.infty

with tf.Session() as sess:

init.run()

restore\_saver.restore(sess, "./my\_best\_mnist\_model\_0\_to\_4")

t0 = time.time()

hidden5\_train = hidden5\_out.eval(feed\_dict={X: X\_train2, y: y\_train2})

hidden5\_valid = hidden5\_out.eval(feed\_dict={X: X\_valid2, y: y\_valid2})

for epoch in range(n\_epochs):

rnd\_idx = np.random.permutation(len(X\_train2))

for rnd\_indices in np.array\_split(rnd\_idx, len(X\_train2) // batch\_size):

h5\_batch, y\_batch = hidden5\_train[rnd\_indices], y\_train2[rnd\_indices]

sess.run(training\_op, feed\_dict={hidden5\_out: h5\_batch, y: y\_batch})

loss\_val, acc\_val = sess.run([loss, accuracy], feed\_dict={hidden5\_out: hidden5\_valid, y: y\_valid2})

if loss\_val < best\_loss:

save\_path = five\_frozen\_saver.save(sess, "./my\_mnist\_model\_5\_to\_9\_five\_frozen")

best\_loss = loss\_val

checks\_without\_progress = 0

else:

checks\_without\_progress += 1

if checks\_without\_progress > max\_checks\_without\_progress:

print("Early stopping!")

break

print("{}\tValidation loss: {:.6f}\tBest loss: {:.6f}\tAccuracy: {:.2f}%".format(

epoch, loss\_val, best\_loss, acc\_val \* 100))

t1 = time.time()

print("Total training time: {:.1f}s".format(t1 - t0))

with tf.Session() as sess:

five\_frozen\_saver.restore(sess, "./my\_mnist\_model\_5\_to\_9\_five\_frozen")

acc\_test = accuracy.eval(feed\_dict={X: X\_test2, y: y\_test2})

print("Final test accuracy: {:.2f}%".format(acc\_test \* 100))

0 Validation loss: 1.416103 Best loss: 1.416103 Accuracy: 44.00%

1 Validation loss: 1.099216 Best loss: 1.099216 Accuracy: 53.33%

2 Validation loss: 1.024954 Best loss: 1.024954 Accuracy: 59.33%

3 Validation loss: 0.969193 Best loss: 0.969193 Accuracy: 60.00%

4 Validation loss: 0.973461 Best loss: 0.969193 Accuracy: 64.67%

5 Validation loss: 0.949333 Best loss: 0.949333 Accuracy: 64.67%

6 Validation loss: 0.922953 Best loss: 0.922953 Accuracy: 66.67%

7 Validation loss: 0.957186 Best loss: 0.922953 Accuracy: 62.67%

8 Validation loss: 0.950264 Best loss: 0.922953 Accuracy: 68.00%

9 Validation loss: 1.053465 Best loss: 0.922953 Accuracy: 59.33%

10 Validation loss: 1.069949 Best loss: 0.922953 Accuracy: 54.00%

11 Validation loss: 0.965197 Best loss: 0.922953 Accuracy: 62.67%

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

Ans : Let's load the best model again, but this time we will create a new softmax output layer on top of the 4th hidden layer:

reset\_graph()

n\_outputs = 5

restore\_saver = tf.train.import\_meta\_graph("./my\_best\_mnist\_model\_0\_to\_4.meta")

X = tf.get\_default\_graph().get\_tensor\_by\_name("X:0")

y = tf.get\_default\_graph().get\_tensor\_by\_name("y:0")

hidden4\_out = tf.get\_default\_graph().get\_tensor\_by\_name("hidden4\_out:0")

logits = tf.layers.dense(hidden4\_out, n\_outputs, kernel\_initializer=he\_init, name="new\_logits")

Y\_proba = tf.nn.softmax(logits)

xentropy = tf.nn.sparse\_softmax\_cross\_entropy\_with\_logits(labels=y, logits=logits)

loss = tf.reduce\_mean(xentropy)

correct = tf.nn.in\_top\_k(logits, y, 1)

accuracy = tf.reduce\_mean(tf.cast(correct, tf.float32), name="accuracy"

And now let's create the training operation. We want to freeze all the layers except for the new output layer:

learning\_rate = 0.01

output\_layer\_vars = tf.get\_collection(tf.GraphKeys.TRAINABLE\_VARIABLES, scope="new\_logits")

optimizer = tf.train.AdamOptimizer(learning\_rate, name="Adam2")

training\_op = optimizer.minimize(loss, var\_list=output\_layer\_vars)

init = tf.global\_variables\_initializer()

four\_frozen\_saver = tf.train.Saver()

And once again we train the model with the same code as earlier. Note: we could of course write a function once and use it multiple times, rather than copying almost the same training code over and over again, but as we keep tweaking the code slightly, the function would need multiple arguments and if statements, and it would have to be at the beginning of the notebook, where it would not make much sense to readers. In short it would be very confusing, so we're better off with copy & paste.

n\_epochs = 1000

batch\_size = 20

max\_checks\_without\_progress = 20

checks\_without\_progress = 0

best\_loss = np.infty

with tf.Session() as sess:

init.run()

restore\_saver.restore(sess, "./my\_best\_mnist\_model\_0\_to\_4")

for epoch in range(n\_epochs):

rnd\_idx = np.random.permutation(len(X\_train2))

for rnd\_indices in np.array\_split(rnd\_idx, len(X\_train2) // batch\_size):

X\_batch, y\_batch = X\_train2[rnd\_indices], y\_train2[rnd\_indices]

sess.run(training\_op, feed\_dict={X: X\_batch, y: y\_batch})

loss\_val, acc\_val = sess.run([loss, accuracy], feed\_dict={X: X\_valid2, y: y\_valid2})

if loss\_val < best\_loss:

save\_path = four\_frozen\_saver.save(sess, "./my\_mnist\_model\_5\_to\_9\_four\_frozen")

best\_loss = loss\_val

checks\_without\_progress = 0

else:

checks\_without\_progress += 1

if checks\_without\_progress > max\_checks\_without\_progress:

print("Early stopping!")

break

print("{}\tValidation loss: {:.6f}\tBest loss: {:.6f}\tAccuracy: {:.2f}%".format(

epoch, loss\_val, best\_loss, acc\_val \* 100))

with tf.Session() as sess:

four\_frozen\_saver.restore(sess, "./my\_mnist\_model\_5\_to\_9\_four\_frozen")

acc\_test = accuracy.eval(feed\_dict={X: X\_test2, y: y\_test2})

print("Final test accuracy: {:.2f}%".format(acc\_test \* 100))

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

Ans : learning\_rate = 0.01

unfrozen\_vars = tf.get\_collection(tf.GraphKeys.TRAINABLE\_VARIABLES, scope="hidden[34]|new\_logits")

optimizer = tf.train.AdamOptimizer(learning\_rate, name="Adam3")

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

init = tf.global\_variables\_initializer()

two\_frozen\_saver = tf.train.Saver()

n\_epochs = 1000

batch\_size = 20

max\_checks\_without\_progress = 20

checks\_without\_progress = 0

best\_loss = np.infty

with tf.Session() as sess:

init.run()

four\_frozen\_saver.restore(sess, "./my\_mnist\_model\_5\_to\_9\_four\_frozen")

for epoch in range(n\_epochs):

rnd\_idx = np.random.permutation(len(X\_train2))

for rnd\_indices in np.array\_split(rnd\_idx, len(X\_train2) // batch\_size):

X\_batch, y\_batch = X\_train2[rnd\_indices], y\_train2[rnd\_indices]

sess.run(training\_op, feed\_dict={X: X\_batch, y: y\_batch})

loss\_val, acc\_val = sess.run([loss, accuracy], feed\_dict={X: X\_valid2, y: y\_valid2})

if loss\_val < best\_loss:

save\_path = two\_frozen\_saver.save(sess, "./my\_mnist\_model\_5\_to\_9\_two\_frozen")

best\_loss = loss\_val

checks\_without\_progress = 0

else:

checks\_without\_progress += 1

if checks\_without\_progress > max\_checks\_without\_progress:

print("Early stopping!")

break

print("{}\tValidation loss: {:.6f}\tBest loss: {:.6f}\tAccuracy: {:.2f}%".format(

epoch, loss\_val, best\_loss, acc\_val \* 100))

with tf.Session() as sess:

two\_frozen\_saver.restore(sess, "./my\_mnist\_model\_5\_to\_9\_two\_frozen")

acc\_test = accuracy.eval(feed\_dict={X: X\_test2, y: y\_test2})

print("Final test accuracy: {:.2f}%".format(acc\_test \* 100))

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.

Ans : You could have two input placeholders, X1 and X2, one for the images that should be fed to the first DNN, and the other for the images that should be fed to the second DNN. It would work fine. However, another option is to have a single input placeholder to hold both sets of images (each row will hold a pair of images), and use tf.unstack() to split this tensor into two separate tensors, like this:

n\_inputs = 28 \* 28 # MNIST

reset\_graph()

X = tf.placeholder(tf.float32, shape=(None, 2, n\_inputs), name="X")

X1, X2 = tf.unstack(X, axis=1)

We also need the labels placeholder. Each label will be 0 if the images represent different digits, or 1 if they represent the same digit:

y = tf.placeholder(tf.int32, shape=[None, 1])

Now let's feed these inputs through two separate DNNs:

dnn1 = dnn(X1, name="DNN\_A")

dnn2 = dnn(X2, name="DNN\_B")

And let's concatenate their outputs:

dnn\_outputs = tf.concat([dnn1, dnn2], axis=1)

Each DNN outputs 100 activations (per instance), so the shape is [None, 100]:

dnn1.shape

TensorShape([Dimension(None), Dimension(100)])

dnn2.shape

TensorShape([Dimension(None), Dimension(100)])

And of course the concatenated outputs have a shape of [None, 200]:

dnn\_outputs.shape

TensorShape([Dimension(None), Dimension(200)])

Now lets add an extra hidden layer with just 10 neurons, and the output layer, with a single neuron:

hidden = tf.layers.dense(dnn\_outputs, units=10, activation=tf.nn.elu, kernel\_initializer=he\_init)

logits = tf.layers.dense(hidden, units=1, kernel\_initializer=he\_init)

y\_proba = tf.nn.sigmoid(logits)

The whole network predicts 1 if y\_proba >= 0.5 (i.e. the network predicts that the images represent the same digit), or 0 otherwise. We compute instead logits >= 0, which is equivalent but faster to compute:

y\_pred = tf.cast(tf.greater\_equal(logits, 0), tf.int32)

Now let's add the cost function:

y\_as\_float = tf.cast(y, tf.float32)

xentropy = tf.nn.sigmoid\_cross\_entropy\_with\_logits(labels=y\_as\_float, logits=logits)

loss = tf.reduce\_mean(xentropy)

And we can now create the training operation using an optimizer:

learning\_rate = 0.01

momentum = 0.95

optimizer = tf.train.MomentumOptimizer(learning\_rate, momentum, use\_nesterov=True)

training\_op = optimizer.minimize(loss)

We will want to measure our classifier's accuracy.

y\_pred\_correct = tf.equal(y\_pred, y)

accuracy = tf.reduce\_mean(tf.cast(y\_pred\_correct, tf.float32))

And the usual init and saver:

init = tf.global\_variables\_initializer()

saver = tf.train.Saver()

* 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.

Ans : The MNIST dataset returned by TensorFlow's input\_data() function is already split into 3 parts: a training set (55,000 instances), a validation set (5,000 instances) and a test set (10,000 instances). Let's use the first set to generate the training set composed image pairs, and we will use the second set for the second phase of the exercise (to train a regular MNIST classifier). We will use the third set as the test set for both phases.

X\_train1 = X\_train

y\_train1 = y\_train

X\_train2 = X\_valid

y\_train2 = y\_valid

X\_test = X\_test

y\_test = y\_test

Let's write a function that generates pairs of images: 50% representing the same digit, and 50% representing different digits. There are many ways to implement this. In this implementation, we first decide how many "same" pairs (i.e. pairs of images representing the same digit) we will generate, and how many "different" pairs (i.e. pairs of images representing different digits). We could just use batch\_size // 2 but we want to handle the case where it is odd (granted, that might be overkill!). Then we generate random pairs and we pick the right number of "same" pairs, then we generate the right number of "different" pairs. Finally we shuffle the batch and return it:

def generate\_batch(images, labels, batch\_size):

size1 = batch\_size // 2

size2 = batch\_size - size1

if size1 != size2 and np.random.rand() > 0.5:

size1, size2 = size2, size1

X = []

y = []

while len(X) < size1:

rnd\_idx1, rnd\_idx2 = np.random.randint(0, len(images), 2)

if rnd\_idx1 != rnd\_idx2 and labels[rnd\_idx1] == labels[rnd\_idx2]:

X.append(np.array([images[rnd\_idx1], images[rnd\_idx2]]))

y.append([1])

while len(X) < batch\_size:

rnd\_idx1, rnd\_idx2 = np.random.randint(0, len(images), 2)

if labels[rnd\_idx1] != labels[rnd\_idx2]:

X.append(np.array([images[rnd\_idx1], images[rnd\_idx2]]))

y.append([0])

rnd\_indices = np.random.permutation(batch\_size)

return np.array(X)[rnd\_indices], np.array(y)[rnd\_indices]

Let's test it to generate a small batch of 5 image pairs:

batch\_size = 5

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

Each row in X\_batch contains a pair of images:

X\_batch.shape, X\_batch.dtype

((5, 2, 784), dtype('float32'))

Let's look at these pairs:

plt.figure(figsize=(3, 3 \* batch\_size))

plt.subplot(121)

plt.imshow(X\_batch[:,0].reshape(28 \* batch\_size, 28), cmap="binary", interpolation="nearest")

plt.axis('off')

plt.subplot(122)

plt.imshow(X\_batch[:,1].reshape(28 \* batch\_size, 28), cmap="binary", interpolation="nearest")

plt.axis('off')

plt.show()

And let's look at the labels (0 means "different", 1 means "same"):

y\_batch

array([[1],

[0],

[0],

[1],

[0]])

Perfect!

* 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.

Ans : Let's generate a test set composed of many pairs of images pulled from the MNIST test set:

X\_test1, y\_test1 = generate\_batch(X\_test, y\_test, batch\_size=len(X\_test))

And now, let's train the model. There's really nothing special about this step, except for the fact that we need a fairly large batch\_size, otherwise the model fails to learn anything and ends up with an accuracy of 50%:

n\_epochs = 100

batch\_size = 500

with tf.Session() as sess:

init.run()

for epoch in range(n\_epochs):

for iteration in range(len(X\_train1) // batch\_size):

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

loss\_val, \_ = sess.run([loss, training\_op], feed\_dict={X: X\_batch, y: y\_batch})

print(epoch, "Train loss:", loss\_val)

if epoch % 5 == 0:

acc\_test = accuracy.eval(feed\_dict={X: X\_test1, y: y\_test1})

print(epoch, "Test accuracy:", acc\_test)

save\_path = saver.save(sess, "./my\_digit\_comparison\_model.ckpt")

0 Train loss: 0.69103277

0 Test accuracy: 0.542

1 Train loss: 0.6035354

2 Train loss: 0.54946035

3 Train loss: 0.47047246

4 Train loss: 0.4060757

5 Train loss: 0.38308156

5 Test accuracy: 0.824

6 Train loss: 0.39047274

7 Train loss: 0.3390794

8 Train loss: 0.3210671

9 Train loss: 0.31792685

10 Train loss: 0.24494292

10 Test accuracy: 0.8881

11 Train loss: 0.2929235

12 Train loss: 0.23225449

13 Train loss: 0.23180929

14 Train loss: 0.19877923

15 Train loss: 0.20065464

15 Test accuracy: 0.9203

16 Train loss: 0.19700499

* 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.

Ans : Let's create the model, it is pretty straightforward. There are many ways to freeze the lower layers, as explained in the book. In this example, we chose to use the tf.stop\_gradient() function. Note that we need one Saver to restore the pretrained DNN A, and another Saver to save the final model:

reset\_graph()

n\_inputs = 28 \* 28 # MNIST

n\_outputs = 10

X = tf.placeholder(tf.float32, shape=(None, n\_inputs), name="X")

y = tf.placeholder(tf.int32, shape=(None), name="y")

dnn\_outputs = dnn(X, name="DNN\_A")

frozen\_outputs = tf.stop\_gradient(dnn\_outputs)

logits = tf.layers.dense(frozen\_outputs, n\_outputs, kernel\_initializer=he\_init)

Y\_proba = tf.nn.softmax(logits)

xentropy = tf.nn.sparse\_softmax\_cross\_entropy\_with\_logits(labels=y, logits=logits)

loss = tf.reduce\_mean(xentropy, name="loss")

optimizer = tf.train.MomentumOptimizer(learning\_rate, momentum, use\_nesterov=True)

training\_op = optimizer.minimize(loss)

correct = tf.nn.in\_top\_k(logits, y, 1)

accuracy = tf.reduce\_mean(tf.cast(correct, tf.float32))

init = tf.global\_variables\_initializer()

dnn\_A\_vars = tf.get\_collection(tf.GraphKeys.GLOBAL\_VARIABLES, scope="DNN\_A")

restore\_saver = tf.train.Saver(var\_list={var.op.name: var for var in dnn\_A\_vars})

saver = tf.train.Saver()

Now on to training! We first initialize all variables (including the variables in the new output layer), then we restore the pretrained DNN A. Next, we just train the model on the small MNIST dataset (containing just 5,000 images):

n\_epochs = 100

batch\_size = 50

with tf.Session() as sess:

init.run()

restore\_saver.restore(sess, "./my\_digit\_comparison\_model.ckpt")

for epoch in range(n\_epochs):

rnd\_idx = np.random.permutation(len(X\_train2))

for rnd\_indices in np.array\_split(rnd\_idx, len(X\_train2) // batch\_size):

X\_batch, y\_batch = X\_train2[rnd\_indices], y\_train2[rnd\_indices]

sess.run(training\_op, feed\_dict={X: X\_batch, y: y\_batch})

if epoch % 10 == 0:

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

print(epoch, "Test accuracy:", acc\_test)

save\_path = saver.save(sess, "./my\_mnist\_model\_final.ckpt")

INFO:tensorflow:Restoring parameters from ./my\_digit\_comparison\_model.ckpt

0 Test accuracy: 0.9455

10 Test accuracy: 0.9634

20 Test accuracy: 0.9659

30 Test accuracy: 0.9656

40 Test accuracy: 0.9655

50 Test accuracy: 0.9656

60 Test accuracy: 0.9655

70 Test accuracy: 0.9656

80 Test accuracy: 0.9654

90 Test accuracy: 0.9654

Well, 96.5% accuracy, that's not the best MNIST model we have trained so far, but recall that we are only using a small training set (just 500 images per digit). Let's compare this result with the same DNN trained from scratch, without using transfer learning:

reset\_graph()

n\_inputs = 28 \* 28 # MNIST

n\_outputs = 10

X = tf.placeholder(tf.float32, shape=(None, n\_inputs), name="X")

y = tf.placeholder(tf.int32, shape=(None), name="y")

dnn\_outputs = dnn(X, name="DNN\_A")

logits = tf.layers.dense(dnn\_outputs, n\_outputs, kernel\_initializer=he\_init)

Y\_proba = tf.nn.softmax(logits)

xentropy = tf.nn.sparse\_softmax\_cross\_entropy\_with\_logits(labels=y, logits=logits)

loss = tf.reduce\_mean(xentropy, name="loss")

optimizer = tf.train.MomentumOptimizer(learning\_rate, momentum, use\_nesterov=True)

training\_op = optimizer.minimize(loss)

correct = tf.nn.in\_top\_k(logits, y, 1)

accuracy = tf.reduce\_mean(tf.cast(correct, tf.float32))

init = tf.global\_variables\_initializer()

dnn\_A\_vars = tf.get\_collection(tf.GraphKeys.GLOBAL\_VARIABLES, scope="DNN\_A")

restore\_saver = tf.train.Saver(var\_list={var.op.name: var for var in dnn\_A\_vars})

saver = tf.train.Saver()

n\_epochs = 150

batch\_size = 50

with tf.Session() as sess:

init.run()

for epoch in range(n\_epochs):

rnd\_idx = np.random.permutation(len(X\_train2))

for rnd\_indices in np.array\_split(rnd\_idx, len(X\_train2) // batch\_size):

X\_batch, y\_batch = X\_train2[rnd\_indices], y\_train2[rnd\_indices]

sess.run(training\_op, feed\_dict={X: X\_batch, y: y\_batch})

if epoch % 10 == 0:

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

print(epoch, "Test accuracy:", acc\_test)

save\_path = saver.save(sess, "./my\_mnist\_model\_final.ckpt")

0 Test accuracy: 0.8694

10 Test accuracy: 0.9276

20 Test accuracy: 0.9299

30 Test accuracy: 0.935

40 Test accuracy: 0.942