Notes for ISL Chapter 10: Deep Learning

Justin Burruss

2026-03-22

Background

The Python code and notes below are for the *Introduction to Statistical Learning* (ISL) study group. This is to try out some of the concepts from Chapter 10. The document was created in RMarkdown with the Python code running via the `reticulate` library plus a little LaTeX.

Our ground rule: when implementing concepts from the chapter, just use basic Python + Pandas + NumPy. It's OK to use more when visualizing or evaluating results.

Single Layer Neural Networks

To try out the concepts from chapter 10, we'll use the same MINST handwritten digits data that the book used in the section on Multilayer Neural Networks. First though, let's make sure we can get a single layer working as expected, starting with the ReLU activation function defined on page 401.

ReLU

ReLU (**Re**ctified **L**inear **U**nit) is given as

$$g(z) = (z)_+ = \begin{cases} 0 & \text{if } z < 0 \\ z & \text{otherwise} \end{cases}$$

In basic Python + Pandas + NumPy we might implement this with the elementwise maximum function `np.maximum()`.

Softmax

The text introduces softmax on page 145 and and gives us a formula for our specific 10-category digits problem on page 405. If Z_m is our response for digit m, then our softmax is:

$$\Pr(Y = m|X) = \frac{e^{Z_m}}{\sum_{\ell=0}^{9} e^{Z_\ell}}$$

NumPy does not include a softmax function, but it's an easy 1-liner with NumPy.

```
import numpy as np
import pandas as pd
import tensorflow as tf
from sklearn.datasets import fetch_openml

def softmax(x):
    """softmax exp(x) / sum(exp(x))"""
    return np.exp(x) / np.sum(np.exp(x))
```

Digits

The first step in building a single-layer version of the multilayer neural network from p. 403–405 is to load the MNIST digits data. We'll want to put it in a format usable by TensorFlow.

```
# fetch the 784-pixels MNIST digits
mnist = fetch_openml('mnist_784')

# use TensorFlow-compatible data types for our target and independent vars
y = pd.get_dummies(mnist.target, dtype='float32') # categorical 0...9
X = mnist.data.astype('float32') / 255.0 # 784 pixels (28x28), scale to 0.0-1.0
```

Let's do an 80/20 train/test split.

```
# apply an 80/20 train/test split
X_train = X.sample(frac=0.8, random_state=2026)
train_idx = X_train.index
X_test = X.drop(train_idx)
y_train = y.loc[train_idx]
y_test  = y.drop(train_idx)
```

We'll use TensorFlow as our reference implementation. The network on p. 403–405 has 128 nodes for layer 2—let's go with that for our 1-layer.

```
# demonstrate 1 layer with ReLU and categorical cross-entropy
nn = tf.keras.models.Sequential([
    tf.keras.layers.Dense(128, activation='relu', input_shape=(784,)),
    tf.keras.layers.Dense(10, activation='softmax')
])
nn.compile(optimizer='adam',
              loss='categorical_crossentropy',
              metrics=['accuracy'])
```

Now to fit the model. Five epochs with a batch size of 256 gives us a quick but reasonable result. We'll leave verbose output on for this first run.

```
# fit over 5 epochs
hist = nn.fit(X_train.values,
              y_train.values,
              epochs=5,
              batch_size=256)
```

```
## Epoch 1/5
##
##   1/219 [..............................] - ETA: 1:18 - loss: 2.3690 - accuracy: 0.0977
##  53/219 [======>.......................] - ETA: 0s - loss: 0.8988 - accuracy: 0.7643
## 101/219 [============>.................] - ETA: 0s - loss: 0.6433 - accuracy: 0.8277
## 147/219 [====================>.........] - ETA: 0s - loss: 0.5407 - accuracy: 0.8543
## 193/219 [=========================>....] - ETA: 0s - loss: 0.4807 - accuracy: 0.8699
## 219/219 [==============================] - 1s 1ms/step - loss: 0.4541 - accuracy: 0.8767
## Epoch 2/5
##
##   1/219 [..............................] - ETA: 0s - loss: 0.2210 - accuracy: 0.9375
##  49/219 [=====>........................] - ETA: 0s - loss: 0.2250 - accuracy: 0.9384
## 101/219 [============>.................] - ETA: 0s - loss: 0.2235 - accuracy: 0.9390
## 155/219 [====================>.........] - ETA: 0s - loss: 0.2191 - accuracy: 0.9395
## 200/219 [==========================>...] - ETA: 0s - loss: 0.2120 - accuracy: 0.9411
## 219/219 [==============================] - 0s 1ms/step - loss: 0.2086 - accuracy: 0.9420
## Epoch 3/5
##
##   1/219 [..............................] - ETA: 0s - loss: 0.1618 - accuracy: 0.9336
##  43/219 [====>.........................] - ETA: 0s - loss: 0.1710 - accuracy: 0.9520
##  89/219 [===========>..................] - ETA: 0s - loss: 0.1616 - accuracy: 0.9547
## 135/219 [=================>............] - ETA: 0s - loss: 0.1568 - accuracy: 0.9562
## 180/219 [========================>.....] - ETA: 0s - loss: 0.1561 - accuracy: 0.9562
## 219/219 [==============================] - 0s 1ms/step - loss: 0.1534 - accuracy: 0.9566
## Epoch 4/5
##
##   1/219 [..............................] - ETA: 0s - loss: 0.1071 - accuracy: 0.9688
##  41/219 [====>.........................] - ETA: 0s - loss: 0.1208 - accuracy: 0.9652
##  89/219 [===========>..................] - ETA: 0s - loss: 0.1213 - accuracy: 0.9668
## 147/219 [====================>.........] - ETA: 0s - loss: 0.1221 - accuracy: 0.9660
## 198/219 [==========================>...] - ETA: 0s - loss: 0.1229 - accuracy: 0.9657
## 219/219 [==============================] - 0s 1ms/step - loss: 0.1219 - accuracy: 0.9657
## Epoch 5/5
##
##   1/219 [..............................] - ETA: 0s - loss: 0.1515 - accuracy: 0.9531
##  62/219 [=======>......................] - ETA: 0s - loss: 0.1056 - accuracy: 0.9705
## 111/219 [==============>...............] - ETA: 0s - loss: 0.1032 - accuracy: 0.9709
## 161/219 [=====================>........] - ETA: 0s - loss: 0.1014 - accuracy: 0.9716
## 212/219 [============================>.] - ETA: 0s - loss: 0.1002 - accuracy: 0.9716
## 219/219 [==============================] - 0s 1ms/step - loss: 0.0999 - accuracy: 0.9716
```

Let's just check that we got the expected number of parameters. We're expecting 101,770 as follows.

$$\text{parameters} = 101,770 \begin{cases} \text{Layer 1} \ = 100,480 \begin{cases} 784 \text{ inputs} \times 128 \text{ nodes} \ \rightarrow 100,352 \text{ weights} \\ 128 \text{ nodes} \rightarrow 128 \text{ biases} \end{cases} \\ \text{Layer 2} \ = 1,290 \begin{cases} 128 \text{ inputs} \times 10 \text{ nodes} \ \rightarrow 1,280 \text{ weights} \\ 10 \text{ nodes} \rightarrow 10 \text{ biases} \end{cases} \end{cases}$$

The `.summary()` method shows us the structure, including parameter counts.

```
# inspect resulting model--did we get 101,770 params?
nn.summary()
```

```
## Model: "sequential"
```

```
## _________________________________________________________________
##  Layer (type)                 Output Shape              Param #
## =================================================================
##  dense (Dense)                (None, 128)               100480
##
##  dense_1 (Dense)              (None, 10)                1290
##
## =================================================================
## Total params: 101,770
## Trainable params: 101,770
## Non-trainable params: 0
## _________________________________________________________________
```

Parameters are as expected.

Now to build the hand-code neural network. First we use `.get_weights()` to get the weights and biases from the reference implementation.

```
wb = nn.get_weights()
w1 = wb[0]
b1 = wb[1]
w2 = wb[2]
b2 = wb[3]
```

Our hand-coded neural network is simply matrix multiplication plus the bias and the ReLU for layer 1, matrix multiplication plus the bias and the softmax for the output. It's really just one line per layer.

```
g1 = np.maximum(X_train.values @ w1 + b1, 0) # elementwise max for ReLU
g2 = softmax(g1 @ w2 + b2)
```

Let's validate against the reference implementation. With TensorFlow we can use the `.predict_on_batch()` method to apply the final trained TensorFlow neural network on the training data. In both cases we use `np.argmax()` to select the most likely digit. Will all predictions from the hand-coded network match the reference implementation over the training data?

```
y_hat = g2.argmax(axis=1)
tf_y_hat = nn.predict_on_batch(X_train.values).argmax(axis=1)
print(f"Matches  (TF vs. HC) = {(y_hat == tf_y_hat).sum()} of {y_hat.size}")
```

```
## Matches  (TF vs. HC) = 56000 of 56000
```

Everything matched.

Let's evaluate accuracy.

```
# try on the test data
h1 = np.maximum(X_test.values @ w1 + b1, 0)
h2 = softmax(h1 @ w2 + b2)
y_hat_te = h2.argmax(axis=1)
print(f"Train Accuracy = {(y_hat == y_train.values.argmax(axis=1)).mean():.3f}")
```

```
## Train Accuracy = 0.976
```

```
print(f"Test Accuracy  = {(y_hat_te == y_test.values.argmax(axis=1)).mean():.3f}")
```

```
## Test Accuracy  = 0.967
```

We'll see if we improve upon that in the multilayer case.

Multilayer Neural Networks

Now let's attempt to recreate the complete neural network from p. 403–405. The text gives us 256 nodes for the first layer and 128 for the second. First we'll construct a reference implementation.

```
# now follow with multilayer from p. 403-405
dnn = tf.keras.models.Sequential([
    tf.keras.layers.Dense(256, activation='relu', input_shape=(784,)),
    tf.keras.layers.Dense(128, activation='relu'),
    tf.keras.layers.Dense(10, activation='softmax')
])
dnn.compile(optimizer='adam',
              loss='categorical_crossentropy',
              metrics=['accuracy'])

# fit over 5 epochs
hist = dnn.fit(X_train.values,
               y_train.values,
               epochs=5,
               batch_size=256,
               verbose=0)
```

Do we end up with 235,146 parameters as indicated in the book?

```
# inspect resulting model--did we get 235,146 params as per the book?
dnn.summary()
```

```
## Model: "sequential_1"
## _________________________________________________________________
##  Layer (type)                Output Shape              Param #
## =================================================================
##  dense_2 (Dense)             (None, 256)               200960
##
##  dense_3 (Dense)             (None, 128)               32896
##
##  dense_4 (Dense)             (None, 10)                1290
##
## =================================================================
## Total params: 235,146
## Trainable params: 235,146
## Non-trainable params: 0
## _________________________________________________________________
```

So far so good. Now let's construct the multilayer neural network. Just as before, it's one line per layer to run the network.

```
# hand-code the DNN and compare with TensorFlow
wb = dnn.get_weights()
w1 = wb[0]
b1 = wb[1]
w2 = wb[2]
b2 = wb[3]
w3 = wb[4]
b3 = wb[5]

# two hidden layers and the output layer
g1 = np.maximum(X_train.values @ w1 + b1, 0)
g2 = np.maximum(g1 @ w2 + b2, 0)
g3 = softmax(g2 @ w3 + b3)
y_hat = g3.argmax(axis=1)
```

Let's see if the two networks tie.

```
# TensorFlow and the hand-code should tie in every case
tf_y_hat = dnn.predict_on_batch(X_train.values).argmax(axis=1)
print(f"Matches (TF vs. HC) = {(y_hat == tf_y_hat).sum()} of {y_hat.size}")
```

```
## Matches (TF vs. HC) = 56000 of 56000
```

Everything ties.

Is the accuracy similar to what we saw with the 1-layer, or perhaps a little higher?

```
# run on test data to check accuracy--any better then 1-layer?
# two hidden layers and the output layer
h1 = np.maximum(X_test.values @ w1 + b1, 0)
h2 = np.maximum(h1 @ w2 + b2, 0)
h3 = softmax(h2 @ w3 + b3)
y_hat_te = h3.argmax(axis=1)
print(f"Train Accuracy = {(y_hat == y_train.values.argmax(axis=1)).mean():.3f}")
```

```
## Train Accuracy = 0.989
```

```
print(f"Test Accuracy  = {(y_hat_te == y_test.values.argmax(axis=1)).mean():.3f}")
```

```
## Test Accuracy  = 0.975
```