2

Build Your First Neural Network with Google Colab

We work through a complete deep learning example with Python’s TensorFlow 2.x library in the Google Colab cloud service. Notebooks for this chapter are located at the following URL: <https://github.com/paperd/tensorflow>.

Building a competent neural network with TensorFlow 2.x is relatively easy. Data science professionals follow a few steps:

1. get raw data
2. explore and preprocess raw data
3. split raw data into train-test sets
4. create a tf.data.Dataset
5. prepare the input pipeline
6. create and train a neural network model

Data scientists want raw data because it is untouched. They want to clean, munge, and sculpt raw data to chisel out meaning for their specific problem at hand. If data is already processed, much of its meaning may already be lost. It is always a good idea to explore data to get a sense of what it looks like before attempting to preprocess. Once data is cleansed and wrangled, it is split into train-test sets.

We create a tf.data.Dataset for TensorFlow consumption. Once in the proper form, we prepare the input pipeline. During preparation, further data cleansing and wrangling may be necessary. We then create and train a neural network model with input pipeline data.

Although reality requires all of these steps, we focus on data modeling with TensorFlow 2.x as it possesses a steep learning curve for the novice. So, we begin with a simple, preprocessed dataset to free learners from the burden of data preprocessing and other associated tasks.

# The load\_digits Dataset

The load\_digits dataset is part of the Scikit-Learn library, which is a free software machine learning library for the Python programming language. The library features various classification, regression, and clustering algorithms that are easy to use and manipulate.

The load\_digits dataset is heavily preprocessed, so we don’t have to worry about cleaning or wrangling. It consists of 1,797 8 x 8 pixel images. Each image is a 64-pixel matrix that represents a hand-written digit from 0-9. We get 64 pixels by multiplying 8 by 8. A **pixel** is an integer with a value between 0 and 255 used to represent image data. The load\_digits dataset is commonly used for training machine learning systems to recognize images of hand-written digits through algorithmic classification.

The images container in load\_digits holds image data of hand-written digits. The data container holds flattened feature vectors. The flattening step is needed to prepare image data for input into fully connected neural network layers. Feature vectors are length of 64 because each 8 x 8 image is flattened by multiplying 8 rows and 8 columns of pixels. The target container holds target values and the target\_names container holds target names. The DESCR container holds descriptions of the dataset.

# Explore the Dataset

Load the dataset:

from sklearn.datasets import load\_digits

# get data

digits = load\_digits()

# get available containers (or keys) from dataset

br = '\n' # create a newline variable

print (digits.keys())

Get the data and display its keys.

Create variables to hold data, images, targets and target names:

# create variables

data = digits.data

images = digits.images

targets = digits.target

target\_names = digits.target\_names

Listing 2-1 displays basic information about the dataset:

# display tensor information

print ('data container:')

print (str(data.ndim) + 'D tensor')

print (data.shape)

print (data.dtype, br)

print ('image container:')

print (str(images.ndim) + 'D tensor')

print (images.shape)

print (images.dtype, br)

print ('targets container:')

print (str(targets.ndim) + 'D tensor')

print (targets.shape)

print (targets.dtype, br)

print ('target names container:')

print (target\_names)

Listing 2-1. Information about the dataset

The data container is a 2D tensor with 1,797 flattened vectors of 64 pixel elements. The image container is a 3D tensor with 1,797 8 x 8 matrices. The image container holds the original un-flattened images. The targets container is a 1D tensor with 1,797 target values between 0 and 9. The target\_names container holds classification labels 0 through 9.

Let’s visualize the 1st data element:

# first image begins at index 0

image = images[0]

import matplotlib.pyplot as plt

plt.imshow(image, cmap='binary')

plt.show()

We see that the image is zero, but computers can’t see as we do. However, we can train them to understand that the digit is zero.

Display the target value (or label) and the feature image 8 x 8 matrix of the first image:

# first digit begins at index 0

target = targets[0]

print ('digit is:', target, br)

# image matrix of first image

print ('image matrix:', br)

print (image)

The computer recognizes the first label as 0. And, it associates the first image matrix with that label. With images and their associated labels, we can train a computer to distinguish between digit images.

# Image Matrix

A closer inspection of the image matrix is key to training:

1. numbers in the matrix represent grayscale intensity
2. every 0 represents white space
3. higher numbers represent darker shades of gray to black

The lower the number, the lighter the gray (with zero being white). The higher the number, the darker shade of gray (with high numbers approaching black). So, a computer is able to understand the shape of a digit with grayscale intensity matrix mapping.

# Split Data into Train-Test Sets

We split a dataset to find a good fitting model with training data and generalize the model to new data with test data. With the load\_digits dataset, we can use images data directly or reshape the flattened data. Let’s reshape flattened data. This is a good exercise because you may have to reshape a dataset in the future.

Create variables to hold input dimensions:

# Input image dimensions

img\_rows, img\_cols = 8, 8

Reshape feature data:

# Reshape

X = data.reshape(data.shape[0], img\_rows, img\_cols)

print ('X reshaped:', X.shape)

print ('number of dimensions:', X.ndim)

The new feature dataset is (1797, 8, 8), which is what we want.

Establish the target dataset:

y = targets

y.shape

The target dataset is (1797,).

Now that we have the feature dataset and associated targets, we are ready to split:

from sklearn.model\_selection import train\_test\_split

X\_train, X\_test, y\_train, y\_test = train\_test\_split(

    X, y, test\_size=0.33, random\_state=0)

Import the train\_test\_split method. Split data into train and test sets. Two-thirds of the dataset is for training and one-third is for testing. Random state is set for reproducibility of results. That is, results are consistent.

Alternatively, we can use images directly as shown in Listing 2-2:

X\_alt = images

y\_alt = targets

print ('X:', X\_alt.shape)

print ('number of dimensions:', X\_alt.ndim, br)

X\_tra, X\_tes, y\_tra, y\_tes = train\_test\_split(

    X\_alt, y\_alt, test\_size=0.33, random\_state=0)

print ('X\_train:', X\_tra.shape)

print ('number of dimensions:', X\_tra.ndim)

train\_percent = X\_tra.shape[0] / X\_alt.shape[0]

print ('train data percent of X data:', train\_percent, br)

print ('X\_test:', X\_tes.shape)

print ('number of dimensions:', X\_tes.ndim)

test\_percent = X\_tes.shape[0] / X\_alt.shape[0]

print ('test data percent of X data:', test\_percent, br)

num\_images = X\_tra.shape[0] + X\_tes.shape[0]

print ('total number of images:', num\_images)

Listing 2-2. Split image data

We know that all is well because X\_train contains two-thirds and X\_test contains one-third of the data.

# GPU Hardware Accelerator

To vastly speed up processing, we can use the Google Colab GPU. However, we must set the GPU in each notebook:

1. click Runtime in the top left menu
2. click Change runtime type from the drop-down menu
3. choose GPU from the Hardware accelerator drop-down menu
4. click SAVE

Test if GPU is active:

import tensorflow as tf

# display tf version and test if GPU is active

tf.\_\_version\_\_, tf.test.gpu\_device\_name()

If '/device:GPU:0' is displayed, the GPU is active. If '..' is displayed, the regular CPU is active.

# Build the Input Pipeline

A TensorFlow input pipeline expects feature data as float32 or float64, and label data as int32 or int64:

X\_train.dtype, y\_train.dtype

Feature data is float64 and label data is int64.

Feature data with a large spread of values may incite error in a neural network, which makes the learning process unstable. Scaling mitigates this problem. Scaling may also speed up calculations in an algorithm. Feature scaling is a technique to scale the range of features data.

Scale train and test feature data:

# scale by dividing by the number of pixels in an image

s\_train = X\_train / 255.0

s\_test = X\_test / 255.0

We scale feature images by dividing them by 255. Image pixels are stored as integer numbers in the range 0 to 255, which is the range that a single 8-bit byte can hold. The division ensures that input pixels are scaled between 0.0 and 1.0.

Create data objects for TensorFlow consumption:

train\_dataset = tf.data.Dataset.from\_tensor\_slices((s\_train,

                                                    y\_train))

test\_dataset = tf.data.Dataset.from\_tensor\_slices((s\_test,

                                                   y\_test))

Let’s see what our tensors look like:

print ('train:', train\_dataset)

print ('test: ', test\_dataset)

We see that both train and test tensors consist of 8 x 8 float64 images and int64 scalar target values.

# Explore TensorFlow Data

Let’s explore what is actually inside the TensorFlow dataset we just created.

Display a slice from the first feature image and its target from the train set:

for feature, label in train\_dataset.take(1):

  print (feature[0], br)

  print (label)

We display the first slice from the first image for brevity. We also display the first target. The take() method grabs samples from a TensorFlow dataset. In this case, we grabbed only the first sample, but we can take n elements.

Let’s grab the first two labels from the test set:

for \_, label in test\_dataset.take(2):

  print (label)

Now, we build the input pipeline.

# Shuffle Data

Before we discuss data shuffling, we need to understand a few concepts. An **epoch** is one cycle through the full training set. Training a neural network typically requires more than a few epochs. A **batch** of data is a group of training samples. Deep learning models don’t process an entire dataset at once. They break data into small batches.

Shuffling data after each epoch ensures that we won’t be stuck with too many bad batches. How does this work? Shuffling reduces model variance, which produces more generalized results and reduces overfitting. **Overfitting** is when a model trains data too well. Overfitting happens when a model learns detail and noise in the training data so well that it negatively impacts performance of the model with new data.

Shuffling ensures that each data point creates an independent change on the model without being biased by preceding data points. That is, it ensures that training data fed to the model contains all flavors of the data.

A great metaphor is shuffling a deck of cards. We shuffle a deck of cards before playing a card game because we want to make sure that each player has the same chance of getting a specific card as another player. Just like shuffling a deck of cards removes bias so does shuffling data before each epoch when training a neural network.

# Continue Pipeline Construction

We need to shuffle and batch our TensorFlow consumable train data. We only batch test data. It doesn’t need to be shuffled because it is new data. Once we set a batch and buffer size, we are ready to shuffle and batch.

Set batch and buffer size:

BATCH\_SIZE = 64

SHUFFLE\_BUFFER\_SIZE = 100

Tweaking batch and buffer size can increase performance. We arbitrarily set batch size to 64 and shuffle buffer size to 100. You can try different values to see how results are impacted. We suggest making buffer size relatively large or shuffling won’t be very effective.

It is a good idea to give the new datasets their own names:

train\_ds = train\_dataset\

.shuffle(SHUFFLE\_BUFFER\_SIZE)\

.batch(BATCH\_SIZE)

test\_ds = test\_dataset.batch(BATCH\_SIZE)

Notice that we only shuffle training data. The reason is because test data is supposed to represent data that our model has not seen. That is, test data is supposed to represent new data. Once we run the shuffle() method on training data, the model automatically shuffles the data before each epoch!

Let’s explore our new datasets:

train\_ds, test\_ds

Shapes are (None, 8, 8). We get None as an extra dimension. What happened? This extra dimension is added because TensorFlow models can accommodate any batch size!

# Feedforward Neural Networks

A feedforward model is the simplest type of network. A **feedforward** neural network is one where information only travels forward in the network. Data moves from the input nodes, through the hidden nodes (if any), and to the output nodes. There are no cycles or loops in the network.

The layers are fully connected, which means that each layer is fully connected to the next one. Fully connected layers connect every node (or neuron) in the previous layer with every node in the successive layer. That is, each neuron in a layer receives an input from all neurons present in the previous layer. Fully connected layers are typically referred to as densely connected.

# Number of Layers

Typically, more layers increase performance. But, more layers require more computing resources. When we don’t have much training data, a simple network with few layers tends to perform as good as or better than a complex network with many layers.

It is a good idea to start with a simple network when exploring a new dataset. We also suggest drawing a small random sample on the dataset and training it with the simple network to get an idea of its potential performance. Following this suggestion saves time and computing resources upfront. And, we get acquainted with the dataset. If the simple model doesn’t train well on the small sample, we can get more date and/or try to find out why we got poor performance before moving to a more complex model.

# Our Model

The first model we build contains an input layer, hidden layer, and an output layer. The first layer in a neural network is always an input layer to inform the model of the shape of incoming data.

The first layer of our model is a Flatten layer. **Flattening** is the process of converting data into 1D arrays. When training a fully connected network, input layer data should always be either 1D vectors or 2D matrices.

The Flatten layer reshapes each tensor into a shape that is equal to the number of elements contained in the tensor not including the batch dimension. This layer has no parameters. Since this is the input layer, we specify input\_shape as (8, 8) to inform the model that each feature image is represented by an 8 x 8 matrix.

The second layer is a densely connected hidden layer. The Dense layer adds the fully connected layer to the neural network. It contains 256 neurons (or nodes) and uses the relu activation function.

The final layer is the output layer. It is also a densely connected layer that contains 10 neurons and uses the softmax activation function. The output layer must always contain the same amount of neurons as it has classes. Since we are classifying digits 0 through 9, we have 10 classes.

An **activation function** defines the output of a node given an input or set of inputs. It is an algorithm that activates each node in a neural net.

**Relu** (or rectified linear activation) is a piecewise linear function that outputs input values directly if positive or zero if negative. It is the default activation function for many networks because it facilitates easier training and often achieves better performance than other activation functions.

**Softmax** output is large if the score is large or small if the score is small. It is frequently used in classification problems where classes are mutually exclusive.

Let’s begin by defining the input shape:

for item in train\_ds.take(1):

  s = item[0].shape

in\_shape = s[1:]

in\_shape

Take the first sample from the train dataset and retrieve its shape. Since the Flatten layer needs the shape of each image, we slice this portion.

Import libraries to build layers:

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Dense, Flatten

Build the model:

model = Sequential([

    Flatten(input\_shape=in\_shape),

    Dense(256, activation='relu'),

    Dense(10, activation='softmax')

])

# Model Summary

The **summary()** method displays the characteristics of a model. It shows layers, output shapes, and parameters.

Let’s try the method:

model.summary()

The layers, output shapes, and parameters are displayed. The model begins with a Flatten layer with output shape (None, 64) and 0 parameters. Batch size can be any number, so None is shown. Each input tensor is a 64 element vector. There are 0 parameters because this layer doesn’t act on the data. Remember that each layer receives output from the previous layer.

The Dense hidden layer receives 64 element vectors. The output shape is (None, 256) because it has 256 neurons. This layer has 16,640 parameters because it inputs 64 neurons from the previous layer that are passed to 256 neurons. Multiply 64 by 256 to get 16, 384. But, we have 256 neurons in this layer. So, add 16,384 to 256 to get 16, 640 parameters!

The output layer has 10 neurons as indicated by shape (None, 10). The hidden layer passes 256 neurons to this layer. Multiply 10 by 256 to get 2,560 parameters. But, we have 10 neurons at this layer. So, add 2,560 to 10 to get 2,570 parameters.

# Compile the Model

The **compile()** method configures the model for training. We set the optimizer, loss function, and metrics. The **loss function** (or objective function) is the quantity minimized during training. It represents a measure of success. The **optimizer** finds parameters that minimize the given loss function.

We use the Adam optimizer. Adam is an adaptive learning rate method that computes learning rates for different parameters. Adam is great because it automatically adapts learning rate for optimum training performance.

We use the sparse\_categorical\_crossentropy loss function because our targets are integers that are mutually exclusive. That is, a digit can only be one of the ten digits.

Compile the model:

model.compile(optimizer='adam',

              loss='sparse\_categorical\_crossentropy',

              metrics=['accuracy'])

We use the adam optimizer because it performs well. We tried other optimizer options, but this one was the best for this case.

Peruse the following URL to see the available optimizers:

<https://www.tensorflow.org/api_docs/python/tf/keras/optimizers>

# Train the Model

Since train\_ds and test\_ds are composed of both images and labels, we include them as parameters to the fit() method for training. We run the model for 60 epochs, which means that we pass the data to the model 60 times. We train with train\_ds and validate with test\_ds.

Train the model:

history = model.fit(train\_ds, epochs=60,

                    validation\_data=(test\_ds))

Our training accuracy is close to 97%. And, test accuracy is close to 95%. Our model is overfitting a bit because test accuracy is lower than train accuracy. Due to randomization effects, your results may differ slightly.

It is always a good idea to evaluate the model for generalization purposes:

model.evaluate(test\_ds)

So, our model generalizes at about 95% with new data.

# Model History

The fit() method automatically keeps a record of the loss and metric values during training. This is why we assigned training to variable history. The history.history object is a dictionary that contains the training record.

Assign the training record to a variable:

history\_dict = history.history

Display a list of the keys in the dictionary:

keys = history\_dict.keys()

print ('keys:', keys, br)

We use the loss, accuracy, val\_loss, and val\_accuracy keys to refer to the training metrics.

Get the length of the dictionary so we can reference the final metric values:

length = len(history\_dict['loss']) - 1

Subtract 1 from the length because Python list indexing starts at 0.

Listing 2-3 shows the final metric values:

final\_loss = history\_dict['loss'][length]

final\_loss\_val = history\_dict['val\_loss'][length]

final\_acc = history\_dict['accuracy'][length]

final\_acc\_val = history\_dict['val\_accuracy'][length]

print ('final loss (train/test):')

print (final\_loss, final\_loss\_val, br)

print ('final accuracy (train/test):')

print (final\_acc, final\_acc\_val)

Listing 2-3. Display the final training metrics

Since we have the training metrics, we can plot the training and validation loss as well as the training and validation accuracy as shown in Listing 2-4. Validation loss and accuracy are based on test data because it is new to the model.

import matplotlib.pyplot as plt

acc = history\_dict['accuracy']

val\_acc = history\_dict['val\_accuracy']

loss = history\_dict['loss']

val\_loss = history\_dict['val\_loss']

epochs = range(1, len(acc) + 1)

plt.figure(figsize=(12,9))

plt.plot(epochs, loss, 'bo', label='Training loss')

plt.plot(epochs, val\_loss, 'b', label='Validation loss')

plt.title('Training and validation loss')

plt.xlabel('Epochs')

plt.ylabel('Loss')

plt.legend()

plt.show()

# clear previous figure

plt.clf()

plt.figure(figsize=(12,9))

plt.plot(epochs, acc, 'bo', label='Training acc')

plt.plot(epochs, val\_acc, 'b', label='Validation acc')

plt.title('Training and validation accuracy')

plt.xlabel('Epochs')

plt.ylabel('Accuracy')

plt.legend(loc='lower right')

plt.ylim((0.5,1))

plt.show()

Listing 2-4. Visualize train and test loss and accuracy

The code imports matplotlib.pyplot to enable plotting. Accuracy and loss metrics are saved in variables. The remainder of the code uses plotting methods to display the results. From the visualization, we see the training process. We recommend plotting training loss and accuracy with every neural network training exercise.

The visualization verifies that our model is overfitting because training accuracy is higher than validation (or test) accuracy. Of course, the overfitting is not drastic. The visualization also shows where training and validation metrics converge or diverge. To generalize so that a model works with new data, training and test accuracy should be as closely aligned as possible.

# Predictions

Deep learning leverages algorithms to automatically model and find patterns in data with the goal of predicting target outputs or responses. If we can predict from new data, we can gain insights to help decision making.

One way to make predictions is with the predict() method on the test data:

predictions = model.predict(test\_ds)

The predictions variable holds all of the predictions based on test\_ds. Each prediction is represented by an array of values that provide a set of probabilities. The number of values in the array is based on the number of target classes. So, each array holds ten values. The value in an array with the highest probability is the predicted digit.

Let’s look at the first prediction (at index 0):

predictions[0]

It’s difficult to identify the highest probability from float numbers, so let’s make it easier to see:

predictions[0].round(2)

We know that the prediction is the digit 2 because the highest probability is located at the third position in the array. Each array position represents a digit between 0 and 9. So, position one is for digit 0, position two is for digit 1, and so on.

We use the following algorithm to elicit the confidence in the first prediction:

import numpy as np

confidence = 100\*np.max(predictions[0])

print (str(np.round(confidence, 2)) + '%')

So, we are very confident that our prediction was correct.

With this algorithm, we predict the digit based on the first image in the test dataset:

first\_pred = np.argmax(predictions[0])

print ('predicted:', first\_pred)

We predict digit 2 based on the first image in the test dataset. Was the prediction correct?

Display the first label from the test dataset:

print ('actual:', y\_test[0])

Our prediction was correct!

Let’s make predictions based on the first five images in the test dataset.

Listing 2-5 displays the first five predictions from test data and then displays the confidence we have in each prediction:

# first five predictions based on test data:

print ('first five preditions:', end=' ')

p5 = []

for i in range(5):

  p = predictions[i]

  v = np.max(p)

  p5.append(p.tolist().index(v))

print (p5)

# confidence in first five predictions:

print ()

print ('Confidence in our predictions:', br)

c = []

for i in range(5):

  conf = str(round(100\*np.max(predictions[i]), 2))

  c.append(conf)

  print (conf + '% for prediction:', p5[i])

Listing 2-5. Five predictions and their confidences

We can also compare our first five predictions compare with the actual target values.

Listing 2-6 displays the predicted digit, actual digit, confidence in the prediction, and actual image for the first five data elements from the test dataset:

# first five predictions from test data

prediction\_5 = [np.argmax(predictions[i])\

                for i, row in enumerate(p5)]

# display predicted digits, actual digits, confidences and images

for i, row in enumerate(prediction\_5):

  print ('predicted:', target\_names[row])

  print ('actual:', target\_names[y\_test[i]])

  print (str(c[i]) + '%')

  fig, ax = plt.subplots()

  image = ax.imshow(X\_test[i], cmap='bone')

  plt.title(target\_names[y\_test[i]])

  plt.show()

  print (br)

Listing 2-6. First five predictions, actual labels, confidences, and images

# Mount Google Drive to Display an Image

We know that Colab notebooks are saved to Google Drive. But, we can also load images and other data from Google Drive into a Colab notebook. If you don’t have a Google email account, create one. Just follow three simple steps:

1. install the Pillow module
2. mount Google Drive
3. point to the image and show it

Install the Pillow library:

!pip install Pillow

The ! symbol enables us to invoke shell commands like installing a Python module in the notebook.

Run the code snippet to begin the mounting process:

from google.colab import drive

drive.mount('/content/gdrive')

To continue mounting, click the URL link, choose the Google account you wish to use, click Allow, copy the authorization code, paste it in the Enter your authorization code: text box, and click the Enter key on your keyboard. It sounds like a lot of work, but it is actually really easy. The drive is mounted to gdrive/My Drive/Colab Notebooks.

Now, be sure you have the image on Google Drive. The image is included on our GitHub site for the book. You just have to save it to your computer and drag it into a Google Drive directory. Of course, you can use any image that you wish.

We saved the image into the Colab Notebooks directory. You can save it anywhere you wish, but be sure to point to it correctly.

Listing 2-7 creates the path to the image and displays the image:

# Be sure to copy the image to this directory on Google Drive

img\_path = 'gdrive/My Drive/Colab Notebooks/simple\_nn\_6.png'

from PIL import Image

import matplotlib.pyplot as plt

img  = Image.open(img\_path)

plt.imshow(img)

Listing 2-7. Display an image from Google Drive

The image name is simple\_nn\_6.png. We import Image from the PIL library. Python expect PIL when accessing the Pillow module. We also import matplotlib.pyplot, with is a module in the Python plotting library matplotlib. We open the image and display it in the Colab notebook.