Build_train_NN_MNIST_part2

October 15, 2022

[1]:
```
# We will add more layers.
```

[2]:
```
# Step 1. Import Tensorflow and other helper libraries

# make sure tensorflow is installed; uncomment the line before if you need to
# pip install tensorflow

# TensorFlow and tf.keras
import tensorflow as tf

# Helper libraries
import numpy as np
import matplotlib.pyplot as plt
```

The MNIST database (Modified National Institute of Standards and Technology database) is a large database of handwritten digits that is commonly used for training various image processing systems.

[3]:
```
# Step 2: load the MNIST data and convert pixel intensities to doubles
# Explore the shape of the data
mnist = tf.keras.datasets.mnist

(x_train, y_train), (x_test, y_test) = mnist.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0
print(x_train.shape)
print(x_test.shape)
```

```
(60000, 28, 28)
(10000, 28, 28)
```

[4]:
```
# A Sequential model is appropriate for a plain stack of
# layers where each layer has exactly one input tensor and one output tensor.
# A Sequential model is not appropriate when:
#  - Your model has multiple inputs or multiple outputs
#  - Any of your layers has multiple inputs or multiple outputs
#  - You need to do layer sharing
#  - You want non-linear topology (e.g. a residual connection, a multi-branch␣
↪model)
```

[6]:
```
# Step 3: Build the tf.keras.Sequential model by stacking the following three
 ↪layers:

# A. The first layer in the neural network takes input signals(values) and
 ↪passes
# them on to the next layer. It doesn't apply any operations on the input
 ↪signals(values)
# and has no weights and biases values associated. In our network the input
 ↪signals
# are of size 28 by 28
# The first layer is of type "Flatten" and you can use an optional input shape
# (the input images are 28 by 28)
# Flattening is converting the data into a 1-dimensional array for input
# into to the next layer.

# B. The LAST layer is Dense (fully connected layer), the output shape is 1 x 10
# The size of the output is 10 because we have 10 possible characters: 0,1,2,..
 ↪,9

# C. Add one fully connected layer, before the last layer.
# IMPORTANT:
# 1) In artificial neural networks, hidden layers are required if and only if
 ↪the data must be
# separated non-linearly (which you would usually know before you begin
 ↪thinking of a NN).
# 2) Empirically derived rules: the number of neurons in that layer is the mean
 ↪of the neurons
# in the input and output layers.
# The number of neurons in the input layer = number of samples

# Task: modify the sequential model you built in Part 1 by adding a fully
 ↪connected (Dense)
# layer, before the last layer. It should have 392 neurons.


# Insert your code below:

model = tf.keras.models.Sequential([
    tf.keras.layers.Flatten(input_shape=(28,28)),
    tf.keras.layers.Dense(units=392),
    tf.keras.layers.Dense(10),
])
```

[7]:
```
# Once a model is "built", you can call its summary() method to display its
 ↪contents:
```

```
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
flatten (Flatten)            (None, 784)               0
_________________________________________________________________
dense (Dense)                (None, 392)               307720
_________________________________________________________________
dense_1 (Dense)              (None, 10)                3930
=================================================================
Total params: 311,650
Trainable params: 311,650
Non-trainable params: 0
_________________________________________________________________
```

[8]:
```
# For each example the model returns a vector of "logits" or "log-odds" scores,␣
↪one for each class.
# pass 1 training data image to the model and convert the predictions into a␣
↪numpy array
predictions = model(x_train[:1]).numpy()
predictions
```

```
WARNING:tensorflow:Layer flatten is casting an input tensor from dtype float64
to the layer's dtype of float32, which is new behavior in TensorFlow 2.  The
layer has dtype float32 because it's dtype defaults to floatx.

If you intended to run this layer in float32, you can safely ignore this
warning. If in doubt, this warning is likely only an issue if you are porting a
TensorFlow 1.X model to TensorFlow 2.

To change all layers to have dtype float64 by default, call
`tf.keras.backend.set_floatx('float64')`. To change just this layer, pass
dtype='float64' to the layer constructor. If you are the author of this layer,
you can disable autocasting by passing autocast=False to the base Layer
constructor.
```

[8]:
```
array([[ 0.34318733, -0.02728777,  0.9576351 ,  0.19785333,  0.7500511 ,
        -0.4475672 , -0.3138823 ,  0.6010897 , -0.11918278,  0.01538008]],
      dtype=float32)
```

[9]:
```
# Use the tf.nn.softmax function to convert these logits into "probabilities"␣
↪for each class:
tf.nn.softmax(predictions).numpy()
```

```
[9]: array([[0.1050172 , 0.07250454, 0.19413853, 0.09081189, 0.15774629,
         0.04762556, 0.05443757, 0.13591458, 0.0661387 , 0.07566509]],
       dtype=float32)
```

```
[10]: # Choose an optimizer and loss function for training

# Deep learning neural networks are trained using the stochastic gradient␣
 ↪descent optimization
# algorithm. As part of the optimization algorithm, the error for the current␣
 ↪state of the
# model must be estimated repeatedly. This requires the choice of an error␣
 ↪function,
# conventionally called a loss function, that can be used to estimate the loss␣
 ↪of the model so
# that the weights can be updated to reduce the loss on the next evaluation.

# The losses.SparseCategoricalCrossentropy loss takes a vector of logits and a␣
 ↪True index and
# returns a scalar loss for each example.

loss_fn = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True)

# This loss is equal to the negative log probability of the true class: It is␣
 ↪zero if the model
# is sure of the correct class. This untrained model gives probabilities close␣
 ↪to random
# (1/10 for each class), so the initial loss should be close to -tf.math.log(1/
 ↪10) ~= 2.3.

loss_fn(y_train[:1], predictions).numpy()
```

```
[10]: 3.0443854
```

```
[11]: # Step 4: Ready to compile. optimizer parameter = 'adam'. Other optimizer␣
 ↪options here:
# https://www.tensorflow.org/api_docs/python/tf/keras/optimizers
# loss = the name of the loss function
# Typically you will use metrics=['accuracy']
model.compile(optimizer='adam',
              loss=loss_fn,
              metrics=['accuracy'])
# The Model.fit method adjusts the model parameters to minimize the loss

# Task: Call the model.fit method to train the model for 10 iterations

# Insert your code below:
```

```
model.fit(x_train, y_train, epochs=10)
```

```
Train on 60000 samples
Epoch 1/10
60000/60000 [==============================] - 30s 496us/sample - loss: 0.3626 -
accuracy: 0.8975
Epoch 2/10
60000/60000 [==============================] - 28s 463us/sample - loss: 0.3140 -
accuracy: 0.9112
Epoch 3/10
60000/60000 [==============================] - 28s 463us/sample - loss: 0.3020 -
accuracy: 0.9157
Epoch 4/10
60000/60000 [==============================] - 27s 453us/sample - loss: 0.2948 -
accuracy: 0.9172- loss: 0.2952 - accuracy:
Epoch 5/10
60000/60000 [==============================] - 27s 453us/sample - loss: 0.2889 -
accuracy: 0.9189
Epoch 6/10
60000/60000 [==============================] - 27s 452us/sample - loss: 0.2841 -
accuracy: 0.9202
Epoch 7/10
60000/60000 [==============================] - 27s 455us/sample - loss: 0.2796 -
accuracy: 0.9210
Epoch 8/10
60000/60000 [==============================] - 27s 457us/sample - loss: 0.2774 -
accuracy: 0.9222
Epoch 9/10
60000/60000 [==============================] - 27s 453us/sample - loss: 0.2738 -
accuracy: 0.9237
Epoch 10/10
60000/60000 [==============================] - 27s 452us/sample - loss: 0.2725 -
accuracy: 0.9242
```

[11]:
```
<tensorflow.python.keras.callbacks.History at 0x7fbf19222350>
```

[12]:
```
# Step 5: Evaluate the model: compare how the model performs on the test dataset

# Task: Use the Model.evaluate method to check the model's performanceon the
 ↪test
# set (x_test, y_test). It would be useful to print the model's testing
 ↪accuracy as well.

# Insert your code below:
```

```
test_loss, test_acc = model.evaluate(x_test, y_test, verbose = 2)
print('\n Test Accuracy: ', test_acc)
```

```
10000/10000 - 1s - loss: 0.3034 - accuracy: 0.9177

 Test Accuracy:  0.9177
```

[ ]:

```

```