Special Problem: MNIST Classification with a Deep Feedforward Neural Network

Source Filename: /main.py

Rico A. R. Picone

Special Problem: MNIST Classification with a Deep Feedforward Neural Network

Create and train a feedforward NN for the MNIST data used in Brunton 4.4. Report plots of the training and testing loss for each training epoch. Report the testing accuracy of the trained model. Hand-optimize hyperparameters such as the learning rate and metaparameters such as the number of layers, number of units per layer, etc. Use Keras with TensorFlow.

Solution

Begin by importing the necessary packages:

```python
import numpy as np
import matplotlib.pyplot as plt
import keras  # Keras 3
from keras import layers
```

Load the MNIST dataset from keras's built-in datasets. The dataset consists of 60,000 training images and 10,000 testing images of handwritten digits. Each image is 28x28 pixels, which we reshape to a 1D array of 784 elements. We also normalize the pixel values to the range [0, 1]:

```python
(x_train, y_train), (x_test, y_test) = keras.datasets.mnist.load_data()
x_train = x_train.reshape(60000, 784).astype("float32") / 255
x_test = x_test.reshape(10000, 784).astype("float32") / 255
print(f"x_train.shape: {x_train.shape}")
print(f"x_test.shape: {x_test.shape}")
print(f"y_train.shape: {y_train.shape}")
print(f"y_test.shape: {y_test.shape}")
```

```
x_train.shape: (60000, 784)
x_test.shape: (10000, 784)
```

```
y_train.shape: (60000,)
y_test.shape: (10000,)
```

Define the model using the keras functional API. We create a feedforward neural network with 5 hidden layers. The input layer has 784 units (one for each pixel in the image), and the output layer has 10 units (one for each digit from 0 to 9). We use ReLU activation functions for the hidden layers and a softmax activation function for the output layer. The model is compiled with the sparse categorical crossentropy loss function, the Adam optimizer, and the accuracy metric.

```
inputs = keras.Input(shape=(784,))  # Input layer
x = layers.Dense(32, activation="relu")(inputs)  # First hidden layer
x = layers.Dense(32, activation="relu")(x)  # Second hidden layer
x = layers.Dense(16, activation="relu")(x)  # Third hidden layer
x = layers.Dense(16, activation="relu")(x)  # Fourth hidden layer
x = layers.Dense(8, activation="relu")(x)  # Fifth hidden laye
outputs = layers.Dense(10, activation="softmax")(x)  # Output layer
model = keras.Model(inputs=inputs, outputs=outputs)  # Create the model
model.summary()  # Display the model summary
```

```
Model: "functional"

 Layer (type)                     Output Shape                  Param #

 input_layer (InputLayer)         (None, 784)                         0

 dense (Dense)                    (None, 32)                     25,120

 dense_1 (Dense)                  (None, 32)                      1,056

 dense_2 (Dense)                  (None, 16)                        528

 dense_3 (Dense)                  (None, 16)                        272

 dense_4 (Dense)                  (None, 8)                         136

 dense_5 (Dense)                  (None, 10)                         90

 Total params: 27,202 (106.26 KB)

 Trainable params: 27,202 (106.26 KB)

 Non-trainable params: 0 (0.00 B)
```

Compile the model. We use the sparse categorical crossentropy loss function, the Adam optimizer with a learning rate of 1e-3, and the accuracy metric. Compiling the model configures the training process.

```
model.compile(
    loss=keras.losses.SparseCategoricalCrossentropy(from_logits=True),
    optimizer=keras.optimizers.Adam(learning_rate=1e-3),
    metrics=["accuracy"],
)
```

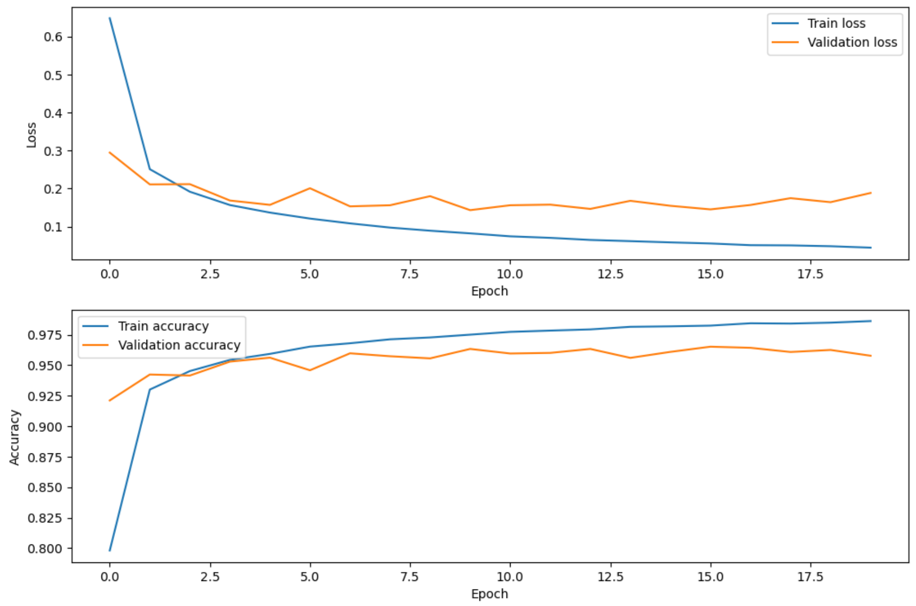
Train the model. We train the model for 20 epochs with a batch size of 32. We use 20 percent of the training data for validation. Validation data is used to evaluate the model after each epoch and to tune the hyperparameters. The training data is shuffled before each epoch. Cross-validation using the validation data helps prevent overfitting. The training history is stored in the `history` variable.

```
history = model.fit(
    x_train, y_train,  # Training data
    batch_size=32, epochs=20, validation_split=0.2  # Hyperparameters
)
```

Plot the training history. We plot the training and validation loss as well as the training and validation accuracy for each epoch.

```
fig, ax = plt.subplots(2, 1, figsize=(12, 8))
ax[0].plot(history.history["loss"], label="Train loss")
ax[0].plot(history.history["val_loss"], label="Validation loss")
ax[0].set_xlabel("Epoch")
ax[0].set_ylabel("Loss")
ax[0].legend()
ax[1].plot(history.history["accuracy"], label="Train accuracy")
ax[1].plot(
    history.history["val_accuracy"], label="Validation accuracy"
)
ax[1].set_xlabel("Epoch")
ax[1].set_ylabel("Accuracy")
ax[1].legend()
```

```
<matplotlib.legend.Legend at 0x3269d65d0>
```



Evaluate the model. We evaluate the model on the test data and print the test loss and accuracy. The test data is used to evaluate the model's performance on withheld data that the model has not seen during training.

```
test_scores = model.evaluate(x_test, y_test, verbose=2)
print("Test loss:", test_scores[0])
print("Test accuracy:", test_scores[1])
```

```
313/313 - 0s - 547us/step - accuracy: 0.9562 - loss: 0.1953
```

```
Test loss: 0.1952916830778122
Test accuracy: 0.9562000036239624
```

The model achieves high test accuracy after 20 epochs of training. This is a good result for a simple feedforward neural network on the MNIST dataset. The model can be further improved by tuning the hyperparameters, such as the learning rate, number of layers, number of units per layer, etc. Alternative models, such as convolutional neural networks (CNNs), can also be used to achieve higher accuracy on the MNIST dataset.

Finally, we display the plots of the training and validation loss and accuracy:

```
plt.show()
```