Brunton and Kutz Problem 4.3: Polynomial Regression with Regularization

Source Filename: /main.py

Rico A. R. Picone

Begin by loading the necessary libraries:

```
import numpy as np
import matplotlib.pyplot as plt
import cvxpy as cp
```

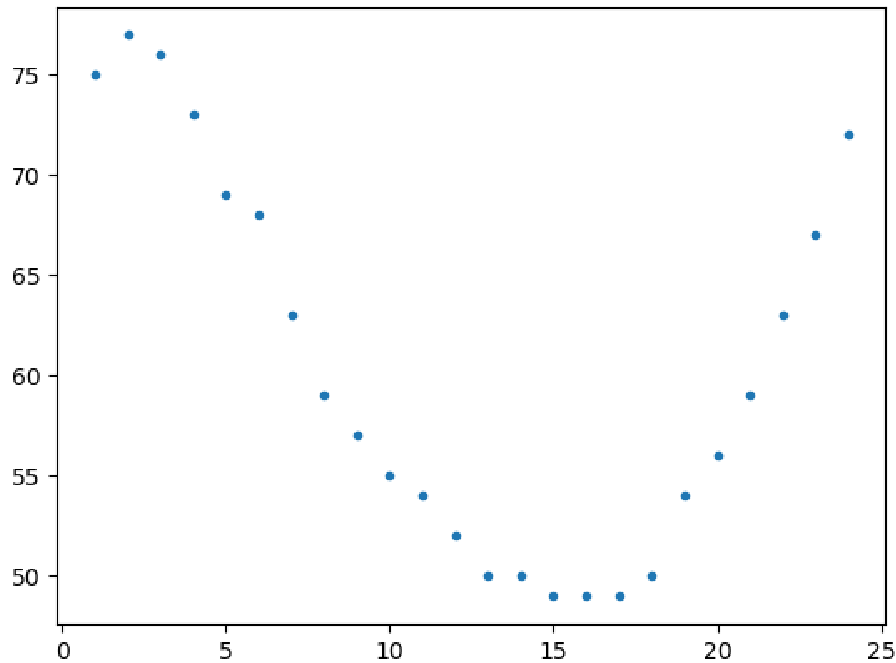
Define the data. We define two vectors: time and temperature.

```
temperature = np.array([
    75, 77, 76, 73, 69, 68, 63, 59, 57, 55, 54, 52, 50, 50, 49, 49, 49, 50, \
    54, 56, 59, 63, 67, 72
])
n_data = len(temperature)
time = np.arange(1, n_data + 1)
```

Visualize the data. The temperature data is plotted against time:

```
fig, ax = plt.subplots()
ax.plot(time, temperature, '.')
```

```
[<matplotlib.lines.Line2D at 0x13f869090>]
```



Create `n_samples` random corrupted data samples from corrupting by 90 percent a single original data point. The first sample is the original temperature data.

```
n_samples = 30
np.random.seed(4)  # Set random seed for reproducibility
y = np.zeros((n_samples, len(temperature)))
y[0] = temperature
for i in range(1, n_samples):
    y[i] = temperature
    jrand = np.random.randint(n_data)
    y[i, jrand] = y[i, jrand] * (1 + .9*np.array([-1,1])[np.random.randint(2)])
```

Define the objective functions. The L1 regularization term encourages sparsity in the coefficients, while the L2 regularization term encourages small coefficients. The elastic net loss is the sum of the squared residuals and the L1 and L2 regularization terms. The elastic net loss function trades off between L1 and L2 regularization. For a Lasso objective, set lambda2 = 0. For a Ridge penalty, set lambda1 = 0. For a standard least squares objective, set lambda1 = lambda2 = 0.

```
def least_squares_objective(y, x, beta):
    return cp.sum_squares(y - x @ beta)/y.shape[0]

def lasso_objective(y, x, beta, lambda1):
```

```python
    return cp.sum_squares(y - x @ beta)/y.shape[0] + \
        lambda1 * cp.norm1(beta)

def ridge_objective(y, x, beta, lambda2):
    return cp.sum_squares(y - x @ beta)/y.shape[0] + \
        lambda2 * cp.sum_squares(beta)

def elastic_objective(y, x, beta, lambda1, lambda2):
    return cp.sum_squares(y - x @ beta)/y.shape[0] + \
        lambda1 * cp.norm1(beta) + lambda2 * cp.sum_squares(beta)
```

Solve the elastic net optimization problem for each corrupted data sample with all four objective functions: least squares, Lasso, Ridge, and elastic net. Fit to a 10th order polynomial with coefficients alpha.

```python
def x_matrix(n_powers, time):
    return np.vander(time, n_powers + 1, increasing=True)
```

```python
n_powers = 10  # Polynomial order
n_alpha = n_powers + 1
n_models = 4
alpha_values = np.zeros((n_samples, n_models, n_alpha))
lambda_amp = 0.1
lambda_pairs = lambda_amp * np.array([[0, 0], [1, 0], [0, 0.01], [0.99, 0.01]])
model_names = ['Least Squares', 'Lasso', 'Ridge', 'Elastic Net']
x = x_matrix(n_powers, time)  # Polynomial power matrix
for i in range(n_samples):
    for j in range(n_models):
        beta = cp.Variable(n_alpha)
        lambda1, lambda2 = lambda_pairs[j]
        if model_names[j] == 'Least Squares':
            objective = cp.Minimize(least_squares_objective(y[i], x, beta))
        elif model_names[j] == 'Lasso':
            objective = cp.Minimize(lasso_objective(y[i], x, beta, lambda1))
        elif model_names[j] == 'Ridge':
            objective = cp.Minimize(ridge_objective(y[i], x, beta, lambda2))
        elif model_names[j] == 'Elastic Net':
            objective = cp.Minimize(elastic_objective(y[i], x, beta, lambda1, lambda2))
        problem = cp.Problem(objective)
        problem.solve(
            solver=cp.SCS,
            verbose=False,
            # eps_rel=1e-10,
            # eps_abs=1e-9,
        )
        alpha_values[i, j] = beta.value
```

```
/Users/ricopicone/anaconda3/envs/595/lib/python3.11/site-packages/cvxpy/problems/problem.py
```

```
  warnings.warn(
```

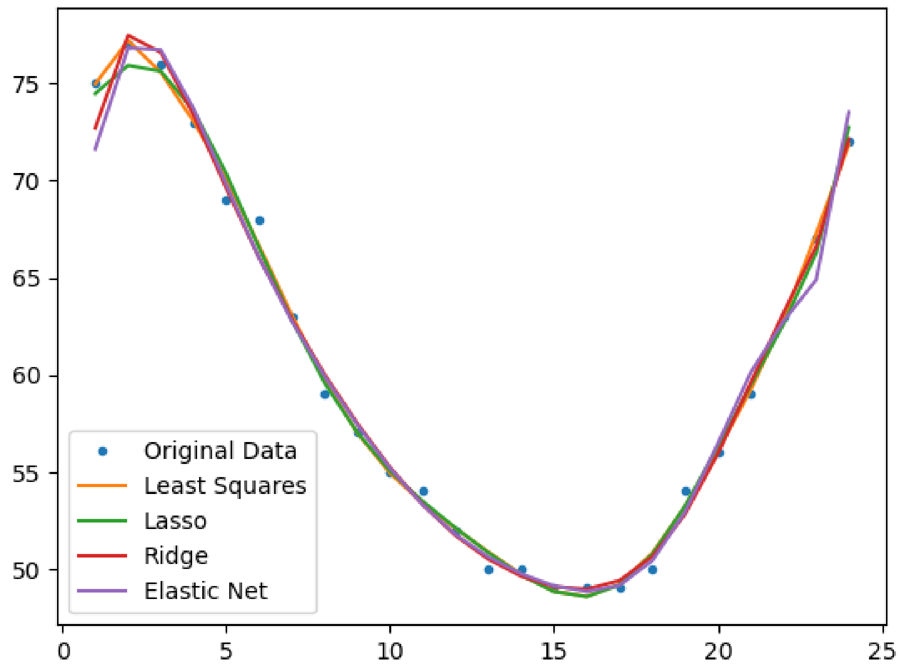
```
print("Alpha values for the original data fit:")
print(f"Least Squares: {alpha_values[0, 0]}")
print(f"Lasso: {alpha_values[0, 1]}")
print(f"Ridge: {alpha_values[0, 2]}")
print(f"Elastic Net: {alpha_values[0, 3]}")
```

```
Alpha values for the original data fit:
Least Squares: [ 5.79073669e+01  3.19066710e+01 -2.05166991e+01  6.93431576e+00
 -1.44045979e+00  1.88958390e-01 -1.58515704e-02  8.46271739e-04
 -2.77723523e-05  5.10312685e-07 -4.01721740e-09]
Lasso: [ 7.18786216e+01  2.80678608e+00  4.41512624e-11 -2.20532232e-01
 -3.46393123e-10  8.48372336e-03 -1.30910243e-03  9.16229784e-05
 -3.39211116e-06  6.41786500e-08 -4.85515256e-10]
Ridge: [ 5.71544201e+01  2.30174866e+01 -8.61312020e+00  1.15879066e+00
 -3.35503650e-03 -1.97376692e-02  2.85698202e-03 -2.03302157e-04
  8.12135632e-06 -1.73930247e-07  1.55626569e-09]
Elastic Net: [ 6.06875001e+01  1.24636900e+01  1.63707706e-11 -2.14466603e+00
  7.17565944e-01 -1.17285140e-01  1.13360987e-02 -6.77236507e-04
  2.45696627e-05 -4.96057234e-07  4.27305246e-09]
```

Plot the fits for the original data:

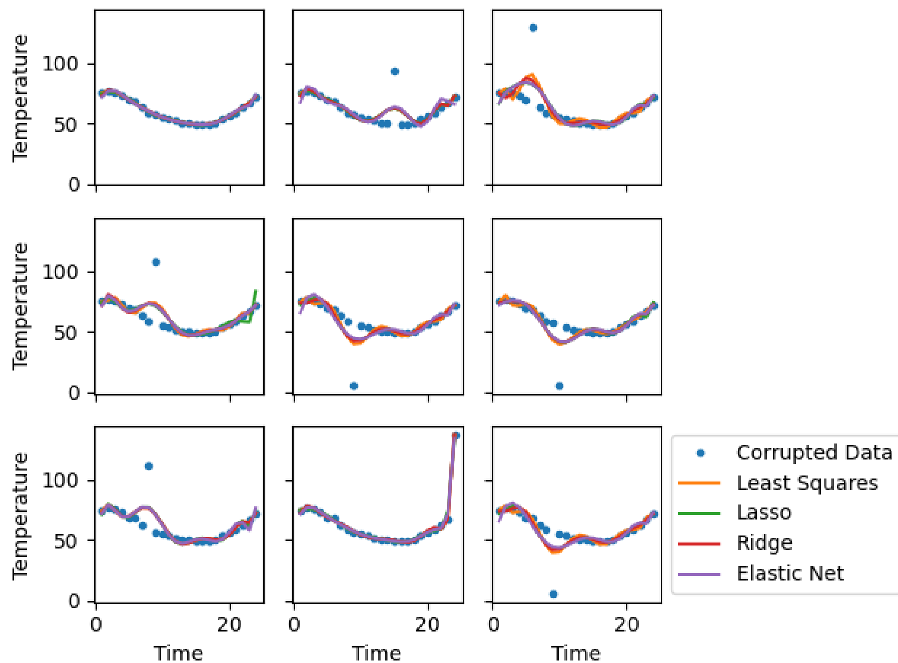
```
fig, ax = plt.subplots()
ax.plot(time, temperature, '.', label='Original Data')
for j in range(n_models):
    ax.plot(time, x @ alpha_values[0, j], label=['Least Squares', 'Lasso', 'Ridge', 'Elastic
ax.legend()
```

```
<matplotlib.legend.Legend at 0x13f8c3c10>
```



All the fits are very similar for the original data. Now we will plot the fits for the original data with the corrupted data. Plot a 3x3 grid of samples and fits for the corrupted data:

```
fig, ax = plt.subplots(3, 3, sharex=True, sharey=True)
for i in range(3):
    for j in range(3):
        ax[i, j].plot(time, y[i * 3 + j], '.', label='Corrupted Data')
        for k in range(n_models):
            ax[i, j].plot(time, x @ alpha_values[i * 3 + j, k], label=['Least Squares', 'Las
            if i == 2:
                ax[i, j].set_xlabel('Time')
            if j == 0:
                ax[i, j].set_ylabel('Temperature')
ax[i, j].legend(loc='center left', bbox_to_anchor=(1, 0.5))
fig.tight_layout()
```

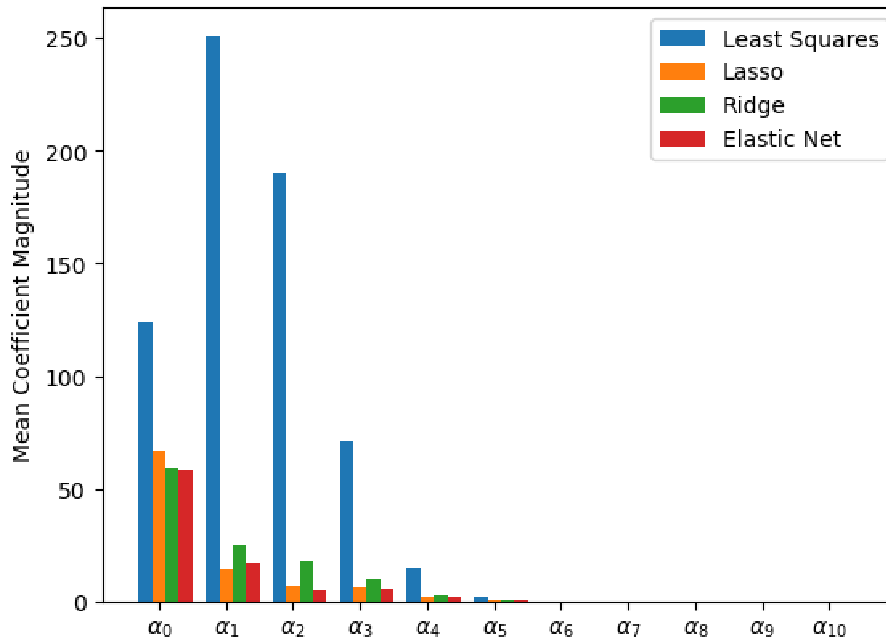


The fits are different. From this selection of samples, it's hard to evaluate the relative performance of the models.

To get a better sense of the magnitude of the coefficients, we can plot the mean of the absolute values of the coefficients for each model. Multi bar chart the mean magnitudes of the coefficients for each model:

```
fig, ax = plt.subplots()
bar_width = 0.2
bar_positions = np.arange(n_alpha)
for j in range(n_models):
    ax.bar(bar_positions + j * bar_width, np.mean(np.abs(alpha_values[:, j]), axis=0), bar_w
ax.set_xticks(bar_positions + bar_width)
ax.set_xticklabels([f'$\\alpha_{{{i}}}$' for i in range(n_alpha)])
ax.set_ylabel('Mean Coefficient Magnitude')
ax.legend()
```

```
<matplotlib.legend.Legend at 0x13ffab310>
```

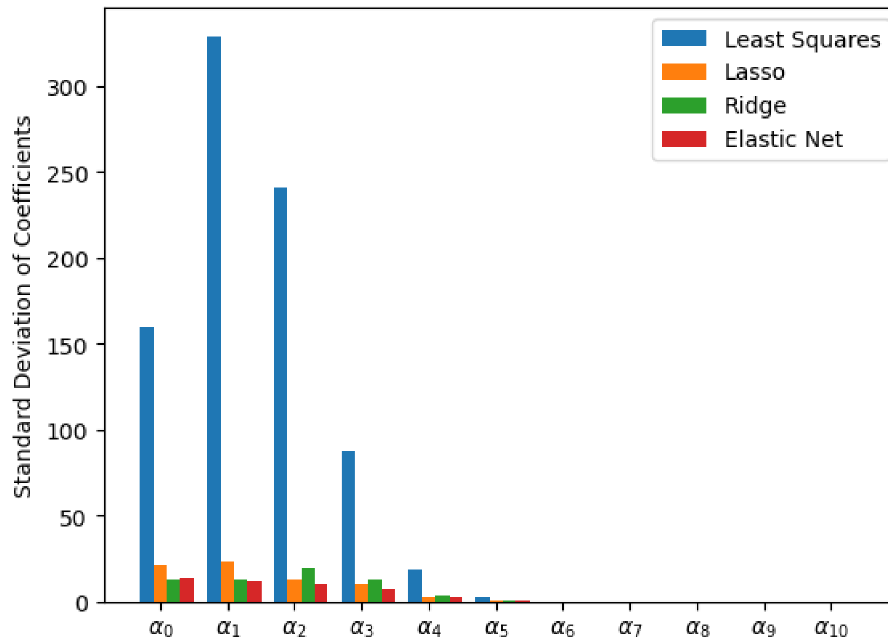


The least squares model has the largest coefficients, as expected. The lasso model is not as sparse as we might want, but weighting the L1 norm more heavily would increase sparsity. I didn't do so because it made the fits quite a bit worse.

To get a better sense of the variation in the coefficients, we can plot the standard deviation of the coefficients for each model. Multi bar chart the standard deviation of the coefficients for each model:

```
fig, ax = plt.subplots()
bar_width = 0.2
bar_positions = np.arange(n_alpha)
for j in range(n_models):
    ax.bar(
        bar_positions + j * bar_width, np.std(alpha_values[:, j], axis=0),
        bar_width,
        label=['Least Squares', 'Lasso', 'Ridge', 'Elastic Net'][j]
    )
ax.set_xticks(bar_positions + bar_width)
ax.set_xticklabels([f'$\\alpha_{{{i}}}$' for i in range(n_alpha)])
ax.set_ylabel('Standard Deviation of Coefficients')
ax.legend()
```

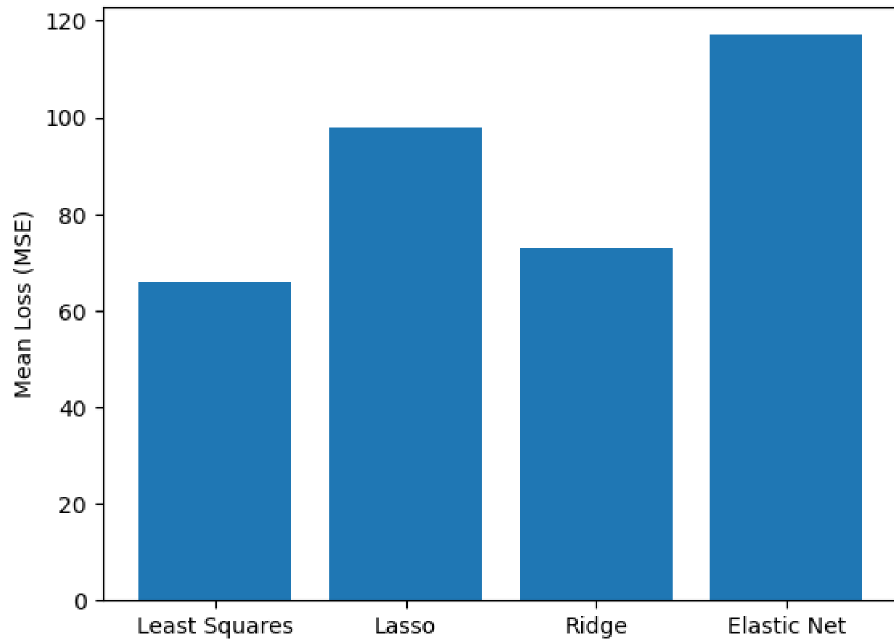
```
<matplotlib.legend.Legend at 0x14811b310>
```



For most coefficients, the standard deviation is largest for the least squares model and smallest for the elastic net model. Having a small standard deviation is a guard against overfitting.

Bar chart the loss for each model:

```
def loss(y, x, alpha):
    # Use the squared error loss (i.e., least squares loss)
    return least_squares_objective(y, x, alpha)
loss_values = np.zeros((n_samples, n_models))
for i in range(n_samples):
    for j in range(n_models):
        loss_values[i, j] = loss(y[i], x, alpha_values[i, j]).value
fig, ax = plt.subplots()
bar_positions = np.arange(n_models)
ax.bar(bar_positions, np.mean(np.abs(loss_values), axis=0))
ax.set_xticks(bar_positions)
ax.set_xticklabels(['Least Squares', 'Lasso', 'Ridge', 'Elastic Net'])
ax.set_ylabel('Mean Loss (MSE)')
plt.show()
```



As we expect, the least squares model has the highest loss because it has prioritized fit over sparsity or small coefficients, which can lead to overfitting.

Brunton and Kutz Problem 4.4: MNIST Classification with Multilinear Regression

Source Filename: /main.py

Rico A. R. Picone

Begin by importing the necessary libraries:

```
import numpy as np
import matplotlib.pyplot as plt
import cvxpy as cp
from mnist import MNIST
```

Load MNIST data as follows:

```
mndata = MNIST('.')
images, labels = mndata.load_training()
images_test, labels_test = mndata.load_testing()
images = np.array(images)  # Images are 28x28, already flattened to 784
images_test = np.array(images_test)
labels = np.array(labels)  # Convert to numpy array
labels_test = np.array(labels_test)
```

Print the shapes of the data:

```
print(f"images.shape: {images.shape}")
print(f"labels.shape: {labels.shape}")
print(f"images_test.shape: {images_test.shape}")
print(f"labels_test.shape: {labels_test.shape}")
```

```
images.shape: (60000, 784)
labels.shape: (60000,)
images_test.shape: (10000, 784)
labels_test.shape: (10000,)
```

Due to limited computing resources available to me, select a random subset of data as follows:

```
n_samples = 5000
np.random.seed(4)  # Set random seed for reproducibility
random_indices = np.random.choice(images.shape[0], n_samples, replace=False)
images = images[random_indices]
```

```
labels = labels[random_indices]
print(f"Subset images.shape: {images.shape}")
print(f"Subset labels.shape: {labels.shape}")
```

```
Subset images.shape: (5000, 784)
Subset labels.shape: (5000,)
```

One-hot encode the labels as follows:

```
labels = np.eye(10)[labels]
labels_test = np.eye(10)[labels_test]
print(f"labels.shape: {labels.shape}")
print(f"Head of labels (check 1-hot encoding): {labels[:5]}")
```

```
labels.shape: (5000, 10)
Head of labels (check 1-hot encoding): [[0. 0. 1. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 1. 0. 0.]
 [0. 0. 0. 0. 0. 0. 1. 0. 0. 0.]
 [0. 1. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 1. 0.]]
```

The problem doesn't specify the specific objective function to use, so we choose lasso. Define loss and lasso objective function as follows:

```
def loss(y, x, beta):
    return cp.sum_squares(y - x @ beta)/y.shape[0]

def l1_regularization(beta, lambda1):
    return lambda1 * cp.norm1(beta)

def lasso_objective(y, x, beta, lambda1):
    return loss(y, x, beta) + l1_regularization(beta, lambda1)
```

Create and solve the optimization problem as follows:

```
n = images.shape[0]
d = images.shape[1]
beta = cp.Variable((d, 10))
lambda1 = 3.0
y = labels  # One-hot encoded labels (labeled output data)
x = images  # Images (input data)
objective = lasso_objective(y, x, beta, lambda1)
problem = cp.Problem(cp.Minimize(objective))
problem.solve(solver=cp.CLARABEL)  # Use (default) CLARABEL solver
print(f"Optimal beta: {beta.value}")
```

```
Optimal beta: [[1.61446357e-28 1.61446357e-28 1.61446357e-28 ... 1.61446357e-28
  1.61446357e-28 1.61446357e-28]
 [1.61446357e-28 1.61446357e-28 1.61446357e-28 ... 1.61446357e-28
  1.61446357e-28 1.61446357e-28]
```

```
 [1.61446357e-28 1.61446357e-28 1.61446357e-28 ... 1.61446357e-28
  1.61446357e-28 1.61446357e-28]
 ...
 [1.61446357e-28 1.61446357e-28 1.61446357e-28 ... 1.61446357e-28
  1.61446357e-28 1.61446357e-28]
 [1.61446357e-28 1.61446357e-28 1.61446357e-28 ... 1.61446357e-28
  1.61446357e-28 1.61446357e-28]
 [1.61446357e-28 1.61446357e-28 1.61446357e-28 ... 1.61446357e-28
  1.61446357e-28 1.61446357e-28]]
```

Calculate accuracy on test set. First, define a function to decode one-hot encoded labels:

```
def decode_one_hot(one_hot, axis=-1):
    return np.argmax(one_hot, axis=axis)
```

Calculate accuracy on test set as follows:

```
y_test = labels_test
x_test = images_test
predictions = x_test @ beta.value
print(f"predictions.shape: {predictions.shape}")
accuracy = np.mean(decode_one_hot(predictions) == decode_one_hot(y_test))
print(f"Test accuracy: {100*accuracy:.3g} percent")
```

```
predictions.shape: (10000, 10)
Test accuracy: 80.4 percent
```

Print the first 10 predictions and true labels:

```
print("First 10 predictions:")
print(decode_one_hot(predictions[:10], axis=1))
print("True labels:")
print(decode_one_hot(y_test[:10], axis=1))
```

```
First 10 predictions:
[7 2 1 0 4 1 4 9 5 7]
True labels:
[7 2 1 0 4 1 4 9 5 9]
```

Decode beta values to images and plot the matrices. First, define a function to decode beta values:

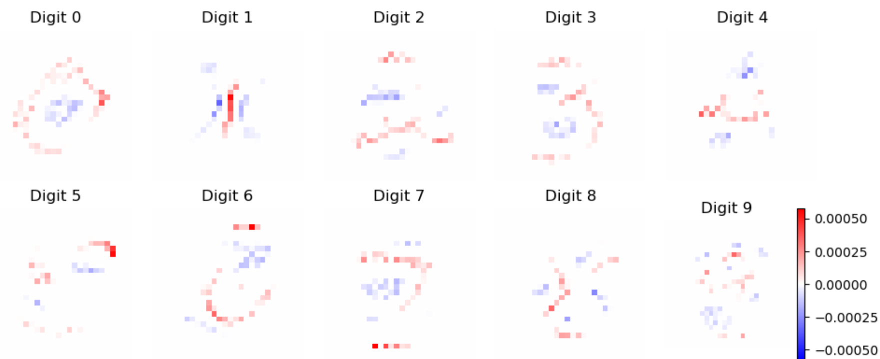
```
def decode_beta(beta):
    return beta.reshape(28, 28, 10)
```

Decode beta values:

```
beta_images = decode_beta(beta.value)
```

Plot the beta images:

```
fig, ax = plt.subplots(2, 5, figsize=(10, 4))
max_val = np.max(np.abs(beta_images))
min_val = -max_val
for i in range(10):
    if i != 9:
        ax[i//5, i%5].imshow(
            beta_images[:, :, i], vmin=min_val, vmax=max_val, cmap='bwr'
        )
    else:
        plt.colorbar(
            ax[i//5, i%5].imshow(
                beta_images[:, :, i], vmin=min_val, vmax=max_val,
                cmap='bwr'
            ),
            ax=ax[i//5, i%5]
        )
    ax[i//5, i%5].axis('off')
    ax[i//5, i%5].set_title(f"Digit {i}")
plt.tight_layout()
plt.show()
```



The above images show the learned weights for each digit (0-9) in the MNIST dataset. Interestingly, the weights appear to be somewhat interpretable, with the weights for each digit resembling the digit itself.

Special Problem: MNIST Classification with a Deep Feedforward Neural Network

Source Filename: /main.py

Rico A. R. Picone

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:

```
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]:

```
(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(10, activation="relu")(x)  # Fifth hidden layer
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, 10)                        170

 dense_5 (Dense)                 (None, 10)                        110

 Total params: 27,256 (106.47 KB)

 Trainable params: 27,256 (106.47 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"],
)
```

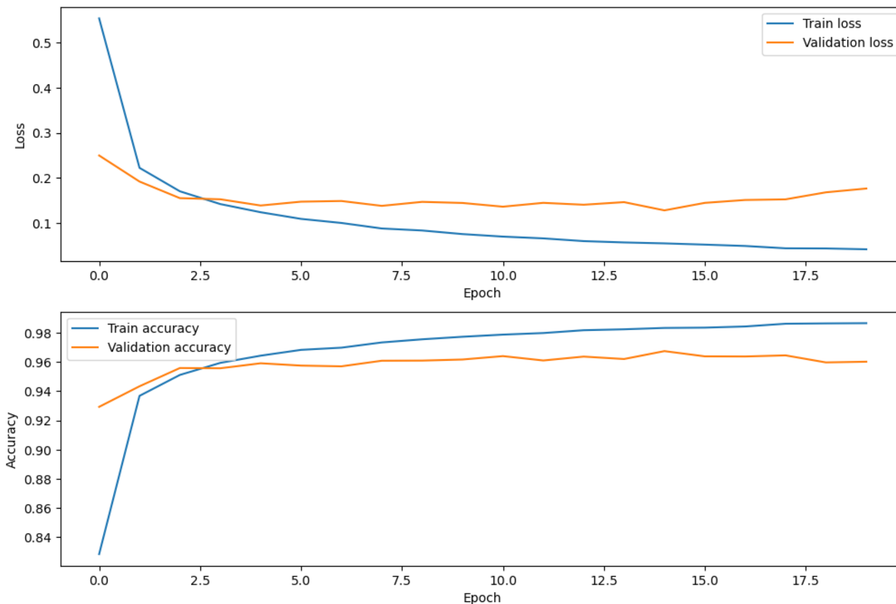
Train the model. We train the model for 20 epochs with a batch size of 32. We use 20 percent of the training data for validation. Validation data is used to evaluate the model after each epoch and to tune the hyperparameters. The training data is shuffled before each epoch. Cross-validation using the 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 0x331aa6a90>
```



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 - 1ms/step - accuracy: 0.9592 - loss: 0.1783
```

```
Test loss: 0.1782819777727127
Test accuracy: 0.9592000246047974
```

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()
```