Harvard John A. Paulson
School of Engineering
and Applied Sciences

F. Wermelinger
Office: Pierce 211

Pair-Programming 5

Python classes, Linear autoencoder

Issued:	September 30, 2022
Due:	October 14, 2022 11:59pm

In this pair-programming session you will continue working on the simple neural network introduced last week using Python classes instead.

You should work on the exercises in groups of 3 to 4 students via a `tmate` session. Your team members can submit the same file. Please indicate your names in a header in the files. See the tutorials on the class website for an example pair-programming workflow.[1] Do not forget to commit and push your work when you are done. Ensure that you are on your *default branch* for this and not, possibly, on your homework branch.

Exercise 1: Python Class for Neural Network Layer

Deliverables:

1. `exercise_1.py`

The goal in this exercise is to build a hidden layer object using Python classes for simple, fully connected neural networks. Use your code from last week as a starting point. The API of your layer class should adhere to the following specification:

- Initialization of an instance should accept two arguments:

  `shape`: a list-like object of two numbers, where the first number corresponds to the number of inputs to the layer and the second number corresponds to the number of neural units in the layer.

  `activation`: an activation function object.

- When a new layer instance is created, its weights and biases should be initialized to random values. The weights and biases should be *attributes* of your class.
- The activation function should be an attribute of your class as well.
- The class should implement the `__call__` special method which computes the output of the layer. Calling this method should require one argument called `inputs`.

[1] https://harvard-iacs.github.io/2022-CS107/pages/tutorials.html#tutorial-pp

- The call method of a layer instance should further assert that the passed arguments have the correct size and dimension. Raise an exception if there is a dimension mismatch for the passed argument.
- The class should further implement the `__str__` and `__repr__` special methods. Recall the difference of the two.

An example pseudo-code for the application of this class is

```
l1 = Layer(shape1, activation)
l2 = Layer(shape2, activation)
inp = np.random.uniform(0.0, 1.0, 100).reshape(1, -1)
out = l2(l1(inp))
```

Write a similar application code in your `exercise_1.py` file including some test code to catch exceptions for invalid input dimensions.

Please see `solution/exercise_1.py` for the solution code.

Recall that `__str__` is used for a pretty string representation of the class that possibly includes information about its state. On the other hand, `__repr__` is used for low-level debugging use. Ideally the returned string representation can be used to create another instance of the object possibly with the same state.

Exercise 2: Linear Autoencoder

Deliverables:

1. `exercise_2.py`

In this task you are going to implement a simple autoencoder[2] network with one *linear* layer for the encoding of the input. This hidden layer is called the *code* and describes the *latent space* of the input (a *compressed* representation of the original input data). Learning the weights and biases for a code with n units will recover the n principal components[3] of the input data. Compressing the input is called *encoding* and decompression of the code is called *decoding*. Since the original data is encoded with less information (compressing the data), information will be lost and when the latent space is decoded again it is not possible to recover the exact original data. Principal component analysis will minimize this reconstruction error. Autoencoders can be used for *classification* tasks and we will use the MNIST database[4] to classify handwritten digits in this exercise. Note that in reality you would not train such a simple linear autoencoder network since it can be done more efficiently using singular value decomposition (SVD).[5]

Import your layer class from the previous example and implement a new *linear* layer class that *inherits* from your general layer class of the previous exercise. Note that the activation

[2] https://en.wikipedia.org/wiki/Autoencoder
[3] https://en.wikipedia.org/wiki/Principal_component_analysis
[4] https://en.wikipedia.org/wiki/MNIST_database
[5] https://en.wikipedia.org/wiki/Singular_value_decomposition

function for a linear layer simply is the identity. The linear layer class should implement the following specification in addition to what is inherited:

- Initialization of an instance should accept one argument:

  `shape`: a list-like object of two numbers, where the first number corresponds to the number of inputs to the layer and the second number corresponds to the number of neural units in the layer.

  The initialization should properly initialize the inherited base class as well.

- The class should have a method called `update` that updates the weights and biases in the layer. The method takes the following two arguments:

  `weights`: a NumPy array with weight values that are used to replace the existing values in the layer.

  `biases`: a NumPy array with bias values that are used to replace the existing values in the layer.

  The method must raise an exception if there is a dimension mismatch in the provided arguments.

  Hint: *Try to set your weights to zero and choose a constant bias to test your implementation.*

In addition, implement an autoencoder class with the following specification:

- Initialization of an instance should accept one argument called `shape` that is a list-like object of two numbers, where the first number corresponds to the number of inputs and the second number corresponds to the number of elements in the code (the size of the compressed representation).
- The class uses two linear layers one for the encoding and the other for the decoding. The output of the second layer is the output of the decoder and has the same size as the input to the encoder (this is called an autoassociative mapping).
- A method called `encode` that takes one argument called `inputs` and returns the code vector (compressed representation of the input). The method must raise an exception if there is a dimension mismatch in the provided argument.
- A method called `decode` that takes one argument called `code` and returns the reconstructed representation of the input (the data will not be exactly the same). The method must raise an exception if there is a dimension mismatch in the provided argument.
- A method called `from_pretrained` which is a `@classmethod` that returns an instance of your autoencoder class with weights and biases initialized from pre-trained weights and biases. The pre-trained arguments are *flat* NumPy arrays[6] called `encoder_weight`, `encoder_bias`, `decoder_weight` and `decoder_bias`.

Finally, you can find these pre-trained weights in `lab/pp5/data/autoencoder.npz` which is a compressed NumPy file.[7] You can load this data and a few test digits with

[6] https://numpy.org/doc/stable/reference/generated/numpy.ndarray.flatten.html

[7] https://numpy.org/doc/stable/reference/generated/numpy.savez_compressed.html

```
data = np.load('data/autoencoder.npz')
test_digit, enc_w, enc_b, dec_w, dec_b = [data[x] for x in data.files]
ae = Autoencoder.from_pretrained(enc_w, enc_b, dec_w, dec_b)
```

Note that `enc_w` are the encoder weights and have dimension shape. You can use matplotlib to visualize an original MNIST digit and its reconstruction based on the provided weights with code similar to:

```
import matplotlib.pyplot as plt
recon = ae.decode(ae.encode(test_digit[0]))  # encode/decode pass
fig, ax = plt.subplots(1, 2, sharex=True, sharey=True)
ax[0].imshow(test_digit[0].reshape(28, 28), cmap='gray_r')
ax[1].imshow(recon.reshape(28, 28), cmap='gray_r')
fig.savefig('test_digit.png', bbox_inches='tight')
```

Recall that the digits in the MNIST database are images with a 28×28 pixel dimension. If your linear autoencoder class works correctly, you should be able to reconstruct the image shown in figure 1. The code vector in this example contains 32 elements compared to the 784 elements in the original image, a compression ratio of 24.5. To assess the quality between the original and reconstructed images the peak signal-to-noise ratio (PSNR) is often used.[8] You can also try your own handwriting as shown in the bottom row in figure 1.

Please see `solution/exercise_2.py` for the solution code.

[8] https://en.wikipedia.org/wiki/Peak_signal-to-noise_ratio

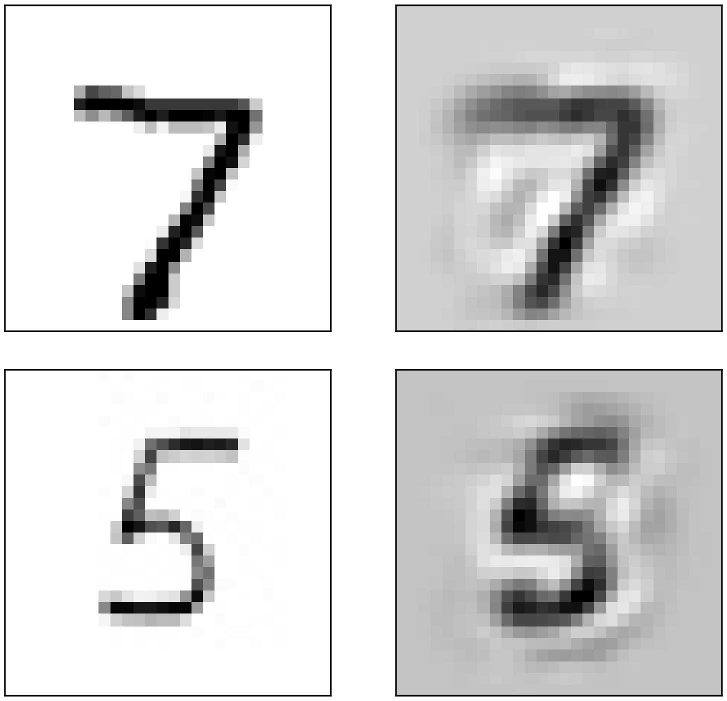

FIGURE 1: Image reconstruction with linear autoencoder. The left column shows the original image and the right column corresponds to the reconstruction. Top row is an image from the MNIST database and bottom row is self made using a black ball pen and a cell phone camera.