Introduction to Programming with Scientific Applications (Spring 2024)

Final project

Study ID	Name	% contributed
202204939	Emil Beck Aagaard Korneliussen	60
202208528	Mathias Kristoffer Nejsum	40

max 3 students

Briefly state the contributions of each of the group members to the project

Since Mathias had some handins due, before he was able to contribute to the project, Emil started doing the first part. Therefore Emils contribution is a bit higher than Mathias.
Most of the work we did besides each other but, some code was written purely by Emil or Mathias.

Emil has written all the code which works with loading and saving of files, as well as functions such as `predict`, `catagorical` and `learn`.

Mathias has written all the code for the plotting the network, as well as the functions `update` and `plot_images` and also some matrix algebra.

Note on plagiarism

Since the evaluation of the project report and code will be part of the final grade in the course, **plagiarism in your project handin will be considered cheating at the exam**. Whenever adopting code or text from elsewhere you must state this and give a reference/link to your source. It is perfectly fine to search information and adopt partial solutions from the internet – actually, this is encouraged – but always state your source in your handin. Also discussing your problems with your project with other students is perfectly fine, but remember each group should handin their own solution. If you are in doubt if you solution will be very similar to another group because you discussed the details, please put a remark that you have discussed your solution with other groups.

For more Aarhus University information on plagiarism, please visit
http://library.au.dk/en/students/plagiarism/

1 Introduction

For our final project in Introduction to Programming with Scientific Applications (IPSA), we have decided to do project IV on MNIST Image Classification. In this project we will create a linear classifier that identifies handwritten digits. We have written code for all mandatory questions in all three parts 1-3.

2 Discussion of code

This chapter serves as a introduction to the general codebase that we have written. We will discuss both the design choices, dependencies and general structure of the implementation, we will also discuss our main ideas for optimization.

2.1 Structure of code

The MNIST project questions consists of three parts:

1. Loading and saving of MNIST database files, and visualisation.
2. Testing and evaluation of a set of weights for a linear classifier.
3. Updating and learning a set of weights for a linear classifier.

This provides a natural test based development approach to the project, since code written in parts 1. and later 2. is used extensively to test any new code written for the later parts. Naturally this progession is also used in the structure of our codebase, reading from the top we first have imports such that any dependencies are not hidden in the code base, then we have type hint definitions which are used as abbreviations for specific types. These type hints are used to make the code more readable, while providing a clear understanding of both function argument types and return types.

After these definitions the actual code begins, the functions appear in the same order as they are described on the project page. Thus, as already mentioned any function that can be used to test another function will be stated above that function. As a specific example all the loading and saving of files is stated before any function that utilizes the content of said files.

2.2 Design choices

A major design choice of our codebase is that we have extracted all the linear algebra functions into their own class contained in a separate file `linalg.py`. This is a common practice during development of larger codebases known as subprocess extraction, and it allows us to make the code more readable and maintainable. The main goal was that we would define operations such as matrix addition, scalar multiplication and matrix multiplication without the need for appending a matrix object with `Matrix.add(Matrix)`. To do this we have implemented a lot of dunder[1] methods. This allows us to write clear and concise functions, for instance, have a look at the prediction function, in which a network consisting of a weight matrix A, and a basis vector b is used to generate a guess vector:

```python
def predict(network: NetW, image: img) -> Matrix:
    x = image_to_vector(image)
    A = Matrix(network[0])
    b = Matrix(network[1])
    return x*A+b
```

By defining methods `__mul__` and `__add__` we can effectively *hide* list comprehensions in the well known operators * and +. Thus, using this extraction principal, it becomes strikingly clear what the prediction function does, which helps with debugging.

One important design choice that we want to highlight in this linear algebra module, is that we actually dont make a distinction between (row)vectors i.e. 1-dimensional lists and matrices, 2-dimensional lists. When we first started our development, we actually did make that distinction, and therefore we initially

[1] abbreviation for double underscore

made two subclasses one for vectors and one for matrices. But when we started actually using the module we discovered that the difference between the two classes was miniscule. Honestly the fact that we had made a clear distinction between the two types, lead to ugly code. A good example of this problem would be when, we wanted to convert a row vector into a column vector, then we would have to write:

```
Matrix([Vector.elements]).transpose()
```

To solve this problem we wrote a new `Matrix.__init__()` constructor to handle inputs of both 1- and 2-dimensional lists. One problem we then had to fix was that the codebase has some code that can only be used on row vectors, to accommodate any potential errors we decided to implement a boolean property that all matrices have, appropriately named `Matrix.row_vector`. Then any function that is only defined as a row vector can just use an assert statement to check that the provided `Matrix` input is correct. This, new implementation did also fix the before mentioned problem of converting row vectors to column vectors:

```
row_vec = Matrix([x,...,z]) # Create a row vector using a 1D list
col_vec = row_vec.transpose()
```

2.3 Dependencies

In this project we were told that we could not use libraries such as NumPy, Keras or others except if it was stated otherwise. This means that we have generally avoided using dependencies. However we have deemed it fit to use a few modules anyway, for certain purposes. The modules are limited to:

- `random` (for generating random numbers)
- `matplotlib` (for visualisation)
- `gzip` (for unpacking MNIST `.gz` files)
- `json` (for reading/writing network weights from/to files)

We decided to use these libraries since, they are very convenient for their
certain purpose and the role they played in the project did not seem to be the main learning objective. Whereas if we had used something like `numpy` some of the questions in the project would have become redundant, since `numpy` had already implemented it.

2.4 Visualisation and Performance of our neural network

Throughout this project we have tested the network in different capacities, this section elaborates both on the visual testing we have done, and the performance we have measured.

Part 2

It started in part 2 where we had to create a function, `evaluate`, based on a given (already trained) network could evaluate the prediction and tell how often the network comprehended the image satisfactory and returned the right number. From this we have a accuracy of 92%.

From this we also had to plot the first few images from the set of images we did this both as just the image where the label the asssed wether or not we guessed right. This plot also holds a plot of the number 0 through 9 and how their linear classifier weights are distributed. which gives a bit of an understanding of how it works.

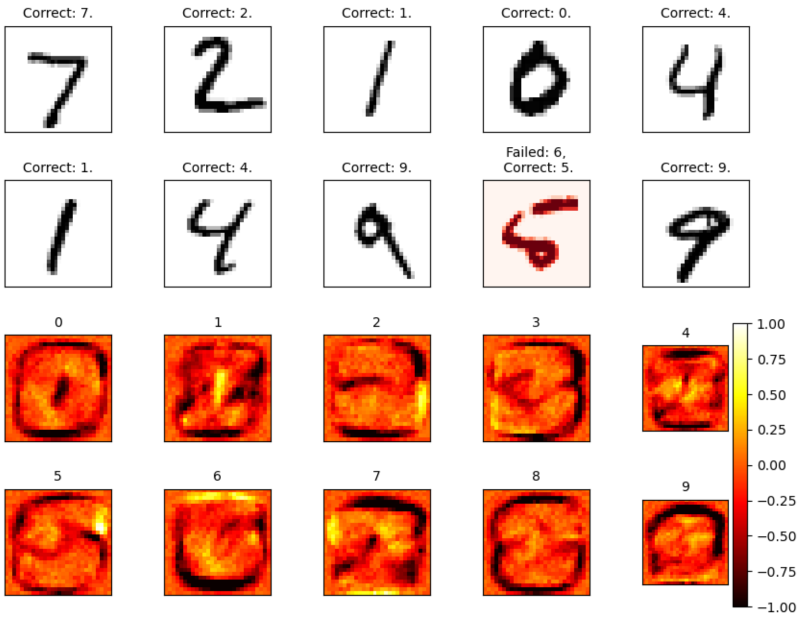

Figure 2.1: The first few images of image classification from the trained network

Part 3

In Part three we have to train the network. We then found it interesting to asses how it performs throughout the iterations. Thus we have constructed this plot of the development of accuracy and cost over evaluated after ever update so we can get a closer look at the rate of learning for the network.

As we see the accuracy tops at around 86% and a cost of around 0.04. We have deemed this to be ok results, but we could probably have better results if we decreased the step size leading to a more precise estimation of the minimum of the cost function. Thus we see that our accuracy is slightly lower than the trained network that we were given.

2.5 Challenges during development

One problem we faced during development was that in the third part of the project, which as mentioned in section 2.1, consists of updating a linear classifier given some evaluation was quite hard. This was due to two different aspects. Firstly the functions `update()` and `learn()` depend on almost the full codebase. This meant that locating any bugs or bad code during development of these functions was way harder, since the bug could have come from other places in the codebase that were misbehaving. To solve this we made sure to properly test all functions in both parts 1. and 2. such that any error were less likely to come from these parts of the codebase. The other reason as to this part posing more difficulties during development is that the maths simply got harder. This meant that we had to spend some time at a blackboard in order to figure out the expected results from the matrix operations. The fact that this part would be the challenging part stood clear to us after the first read of the project description. Thus, in order to ensure we had enough time to meet the project deadline, we started development of parts 1. and 2. before we finished our handins. This meant that we had time to finish these parts and focus on the third more challenging part of the project

2.6 Ideas for optimization

The first problem that comes to mind, when reflecting upon the performance of our code is that both evaluation and training a new set of weights is quite slow. There are many reasons for this, but we suspect

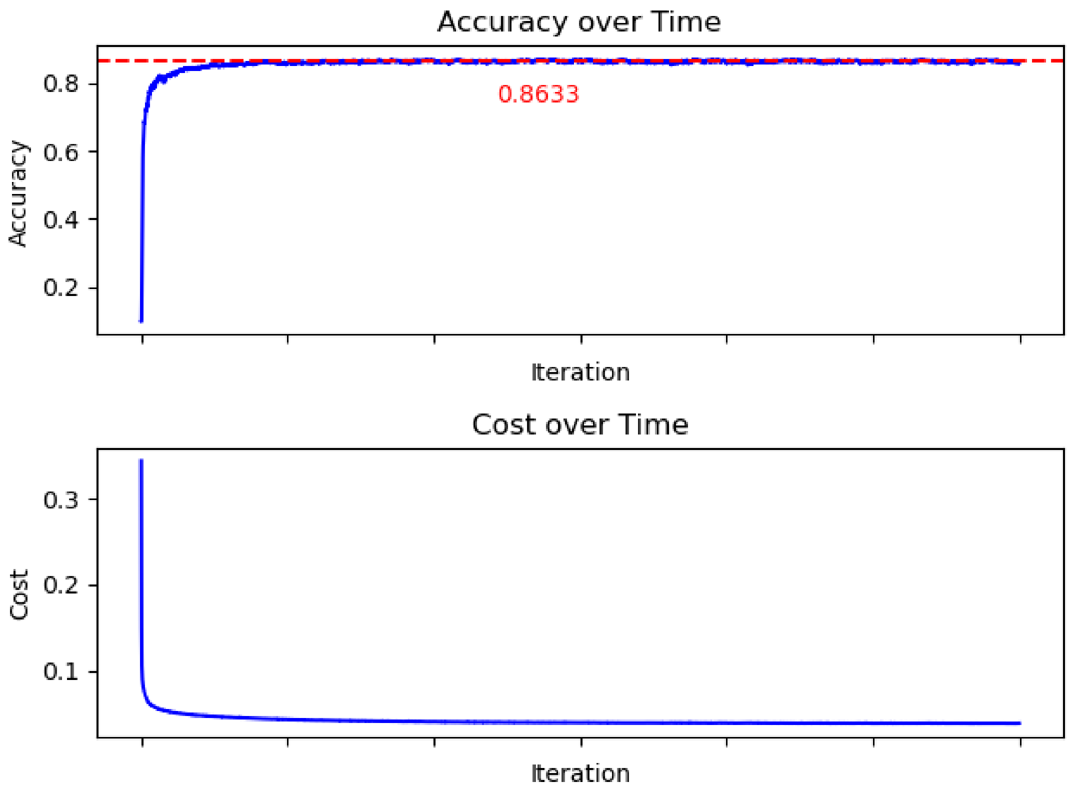

Figure 2.2: The accuracy and cost through each batch we have trained the data on

that the main reason is they way that we compute our numbers. Currently we have written our own linear algebra module, but even though we think of our selves as principalled programmers. There is no way that our module can even come close to the computation time that a module like `numpy` would be able to. Thus, we suspect that a major optimization would be to just implement their module, and let `numpy.array` handle the computations. This would of course add another dependency, but since `numpy` is a very well maintained codebase, it would not pose a major concern.

The design choice of extracting all the maths into its own module as described in section 2.2. Probably also poses a small drawback in terms of runtime, this is because every time we return a computation such as an addition, we return a new instance of the class, as such:

```
def add(self, matrix: Mat) -> Mat:
  # we have removed assertion statements for clarity
  return Matrix([[x+y for (x, y) in zip(row_self, row_other)] for
     row_self, row_other in zip(self.elements, matrix.elements)])
```

In general creating a new instance of a class not only means calling `Matrix.__init__()` again but the storage in bytes is also bigger. This is because there is a lot of overhead in the storage of a class, and we suspect that this poses a drawback for the runtime complexity of our codebase, since we are doing a lot of operations per epoch. Thus, even though the class syntax is much more elegant, our design choice might be slowing the main functionality a bit down. In order to combat this we could as mentioned either not create a new instance of the class, or simply remove the class and write functions to do the computation instead. As this is a project with a deadline we unfortunately did not have time to make a comparison between our current implementation and a functional implementation. Such a comparison would have been interesting since, if it were the case that the code would run a lot faster, then such an implementation would not add another dependency to the codebase, as apposed to the aforementioned `numpy` implementation.

3 Appendix

3.1 Codebase

final-project.py

```python
import gzip
import random
import matplotlib.pyplot as plt
import json
import csv
from linalg import Matrix

img = list[list[int]]  # 2d object of integer values
NetW = list[list[int | float], list[int, float]]

#part 1
def read_labels(filename: str) -> list[int]:
    """
    Read the labels from a gzip file following the byteroder described
        in
    http://yann.lecun.com/exdb/mnist/
    Magic number should be 2049

    Args:
    1. filename (str): The filename of the .gz file

    Returns:
    * list[int]: A list of the labels in the file.
    """
    with gzip.open(filename, 'rb') as f:
        magic_num = int.from_bytes(f.read(4), byteorder="big")
        assert magic_num == 2049, "The magic number of the read file is
            not 2049"
        num_labels = int.from_bytes(f.read(4), byteorder="big")
        return [byte for byte in f.read(num_labels)]


def read_images(filename: str) -> list[img]:
    """
    Read the images from a gzip file following the byteroder described
        in
    http://yann.lecun.com/exdb/mnist/
    Magic number should be 2051

    Args:
    1. filename (str): The filename of the .gz file

    Returns:
    * list[img]: A list of the images in the file.
    """
    with gzip.open(filename, "rb") as f:
        magic_num = int.from_bytes(f.read(4), byteorder="big")
        assert magic_num == 2051, "The magic number of the read file is
            not 2051"
        num_img = int.from_bytes(f.read(4), byteorder="big")
```

```
        num_row = int.from_bytes(f.read(4), byteorder="big")
        num_col = int.from_bytes(f.read(4), byteorder="big")

        return [[[byte for byte in f.read(num_row)] for _col in range(
            num_col)] for _img in range(num_img)]


def plot_images(images: list[img], labels: list[int],  Weight_matrix:
    Matrix, prediction: list[int] = []) -> None:
    """
    Plot the first images in a list of images, along with the
        corresponding labels.

    Args:
    1. images (list[img]): A list of the images.
    2. labels (list[int]): A list of the image labels.
    3. rows [optional] (int): The amount of image rows to plot.
    4. cols [optional] (int): The amount of image cols to plot.
    5. prediction[optional] (list[int]): A list of predicted labels for
        the images.

    Returns:
    * Opens a matplotlib plot of the first rows x cols images.
    """

    fig, axes = plt.subplots(nrows= 4, ncols=5, figsize=(10, 8),
        gridspec_kw={'wspace': 0.5})
    axes1 = axes[0:2]
    axes2 = axes[2:]
    A_T = Weight_matrix.transpose()
    weight_images = [Matrix(row).reshape(28) for row in A_T]

    for idx, ax in enumerate(axes1.flat):
        ax.tick_params(left=False, right=False, labelleft=False,
                       labelbottom=False, bottom=False)
        # if there is a prediction for image
        color = "gray_r"
        try:
            prediction[idx]
        except IndexError and TypeError:
            label = str(labels[idx])
        else:
            if prediction[idx] == labels[idx]:
                label = f"Correct: {labels[idx]}."
            else:
                label = f"Failed: {prediction[idx]},\n Correct: {labels[
                    idx]}."
                color = "Reds"
        ax.imshow(images[idx], cmap=color, vmin=0, vmax=255)
        ax.set_title(label, fontsize=10)

    for idx, ax in enumerate(axes2.flat):
        ax.tick_params(left=False, right=False, labelleft=False,
                       labelbottom=False, bottom=False)
        im = ax.imshow(weight_images[idx].elements, cmap="hot", vmin=-1,
            vmax=1)
        ax.set_title(idx, fontsize=10)
```

```
    fig.colorbar(im, ax=axes2[:,-1])
    plt.show()
    return None

#part 2
def linear_load(filename: str) -> NetW:
    """
    Load a json file of filename in as a NetW
    Args:
    1. filename (str): The filename of the .weights file

    Returns:
    * NetW: A network consisting of a list of A and b.

    ## Example use
    >>> import tempfile
    >>> with tempfile.NamedTemporaryFile('w', delete=False) as tmp:
    ...     filename = tmp.name
    ...     json.dump([[1, 2], [3, 4]], tmp)
    >>> linear_load(filename)
    [[1, 2], [3, 4]]
    """
    with open(filename) as f:
        weights = json.load(f)
    return weights


def linear_save(filename: str, network: NetW) -> None:
    """
    inspiration from: https://www.geeksforgeeks.org/create-a-file-if-not
       -exists-in-python/
    Save a .weights file

    Args:
    1. filename (str): The filename of the .weights file.

    Returns:
    * None: It only saves the .weights file.

    ## Example use

    >>> import tempfile
    >>> network = [[1, 2], [3, 4]]
    >>> with tempfile.NamedTemporaryFile(delete=False) as tmp:
    ...     filename = tmp.name
    >>> linear_save(filename, network)
    >>> linear_load(filename)
    [[1, 2], [3, 4]]
    """
    try:
        with open(filename, 'x') as f:
            f.write(str(network))
    except FileExistsError:
        with open(filename, "w") as f:
            f.write(str(network))
    return None

```

```python
def image_to_vector(image: img) -> Matrix:
    """
    Takes a image an makes it to a vector and normalize each entry.

    Args:
    1. image (img): an image that satisfies the criteria for the MNIST
       images.

    Returns:
    * Matrix: a row vector with entries in the range [0,1]

    ## Example use
    >>> image = [[0, 255], [127, 255]]
    >>> v1 = image_to_vector(image)
    >>> print(v1)
    |                 0.0                  1.0 0.4980392156862745
                       1.0  |
    <BLANKLINE>
    """
    return Matrix([x/255 for row in image for x in row])


def mean_square_error(v1: Matrix, v2: Matrix) -> float:
    """
    Define the mean squared error between two vectors

    Args:
    1. v1 (Matrix): The first vector
    2. v2 (Matrix): The second vector

    Returns:
    * float: The mean squared error

    ## Example use
    >>> v1 = Matrix([1, 2, 3])
    >>> v2 = Matrix([1, 2, 4])
    >>> mean_square_error(v1, v2)
    0.3333333333333333
    """
    assert v1.row_vector and v2.row_vector, "mean squared error is only
       defined between row vetors"
    return sum(((v1 - v2)**2)[0])/v1.col_space()


def argmax(v1: Matrix) -> int:
    """
    Define argmax for a vector.

    Args:
    1. v1 (Matrix): is a row vector

    Returns:
    * int: the index of the largest element of a vector

    ## Example use
    >>> v1 = Matrix([1, 2, 3])
    >>> argmax(v1)
```

```python
    2
    """
    assert v1.row_vector, "argmax is only defined for vectors"
    return v1.elements[0].index(max(v1.elements[0]))


def catagorical(label: int, classes: int = 10) -> Matrix:
    """
    Define catagorical, which is a list where all indeces are 0 besides
        the number that is given which is 1

    Args:
    1. label (int): a single label
    2. classes (int): the amount of different outcomes

    Returns:
    * Matrix: a row vector (Matrix) of the length classes

    # Example use
    >>> print(catagorical(2, 10))
    | 0 0 1 0 0 0 0 0 0 0 |
    <BLANKLINE>
    """
    assert label <= classes, "labels cannot be longer than classes."
    return Matrix([1 if i == label else 0 for i in range(classes)])


def predict(network: NetW, image: img) -> Matrix:
    """
    Returns x * A + b

    Args:
    1. Network (NetW): A network that contain both A and b
    2. Image (img): a single image is given

    Returns:
    * x * A + b

    ## Example use
    >>> network = [[[0.1, 0.2], [0.3, 0.4], [0.5, 0.6], [0.7, 0.8]],
        [0.1, 0.2]]
    >>> image = [[0, 255], [127, 255]]
    >>> prediction = predict(network, image)
    >>> print(prediction)
    | 1.3490196078431373 1.6988235294117648 |
    <BLANKLINE>
    """
    x = image_to_vector(image)
    A = Matrix(network[0])
    b = Matrix(network[1])
    return x * A + b


def evaluate(network: NetW, images: list[img], labels: list[int]) ->
    tuple:
    """
```

```
    Evaluates predictions of the numbers, and returns the predictions,
        accracy of the predictions and the cost.

    Args:
    1. Network (NetW): A network that contain both A and b
    2. images (list[img]): A list of the images.
    3. labels (list[int]): A list of the image labels.

    Returns:
    * Predictions (list): is a list of the predictions for the given
        image
    * cost (float): the value of cost, which is the average MSE
    * Accuracy (float): is the fraction of times we predicted correctly
    """
    guesses = [predict(network, img) for img in images]
    predictions = [argmax(guess) for guess in guesses]

    cost = sum([mean_square_error(guesses[i], catagorical(labels[i]))
                for i in range(len(images))])/len(images)

    accuracy = sum([1 if predictions[i] == labels[i]
                    else 0 for i in range(len(images))])/len(images)

    return (predictions, cost, accuracy)


# part 3
def create_batches(values: list[int | float], batch_size: int) -> list[
    list[int | float]]:
    """
    Creates permuted batches e.g.

    Args:
    Values: this is the list that should be made into batches

    Returns:
    * A list of the batches
    """
    random.shuffle(values)

    # https://www.geeksforgeeks.org/break-list-chunks-size-n-python/
    return [values[i:i + batch_size] for i in range(0, len(values),
        batch_size)]


def update(network: NetW, images: list[img], labels: list[int],
    step_size: float=0.1) -> tuple:
    """
    Updates the network using gradient descent

    Args:
    1. Network (NetW): A network that contain both A and b
    2. images (list[img]): A list of the images.
    3. labels (list[int]): A list of the image labels.
    4. Stepsize (float): a stepsize for the gradient decent

    Returns
```

```python
    * Tuple containing the elements of A and b that have been updated
    """
    A, b = network

    A = Matrix(A)
    b = Matrix(b)
    n = len(images)

    for img, lab in zip(images, labels):
        x = image_to_vector(img)
        a = x * A + b
        y = catagorical(lab)
        error = 1 / 5 * (a - y)
        b -= step_size/n * error
        A -= step_size/n * (x.transpose() * error)
    return A.elements, b.elements


def learn(images: list[img], labels: list[int], epochs: int, batch_size: int, step_size: float=0.1, test_image_file: str="t10k-images-idx3-ubyte.gz", test_labels_file: str="t10k-labels-idx1-ubyte.gz") -> tuple:
    """
    This function does some training on the data, such that we better can predict the numbers

    Args:
    1. images (list[img]): The list of images
    2. labels (list[int]): The list of labels
    3. epochs (int): The number of iterations
    4. batch_size (int): The size of the batches
    5. step_size (float): The step size for the gradient descent
    6. test_image_file (str): The filename for the test images
    7. test_labels_file (str): The filename for the labels that fit with the images

    Returns:
    * Predictions (list): is a list of the predictions for the given image
    * cost (float): the value of cost, which is the average MSE
    * Accuracy (float): is the fraction of times we predicted correctly
    * it also creates a plot of the development of the cost and accurace through the iterations
    """
    test_img = read_images(test_image_file)
    test_labs = read_labels(test_labels_file)

    A_random = [[random.uniform(0, 1/784) for j in range(10)]
                for i in range(784)]
    b_random = [random.random() for i in range(10)]

    print("Random weights generated. Testing")

    linear_save("trained.weights", [A_random, b_random])

    evaluation = evaluate([A_random, b_random], test_img, test_labs)
    cost_list = [evaluation[1]] #track cost
```

```python
    accuracy_list = [evaluation[2]] #track accuracy
    print(f"Test done, cost {evaluation[1]}, accuracy {evaluation[2]}")

    for epoch in range(epochs):
        batch_mask = create_batches(
            [i for i in range(len(images))], batch_size)

        print(f"Itreration --- {epoch} --- ")

        NW = linear_load("trained.weights")

        for idx, batch in enumerate(batch_mask):
            print(f"Batch: {idx} ")
            image_batch = [img for i, img in enumerate(images) if i in batch]
            label_batch = [lab for j, lab in enumerate(labels) if j in batch]

            NW = update(NW, image_batch, label_batch, step_size)


        evaluation = evaluate(NW, test_img, test_labs)
        cost_list.append(evaluation[1])
        accuracy_list.append(evaluation[2])

        linear_save("trained.weights", list(NW))


        print(f"Training done, cost: {evaluation[1]}, accuracy {evaluation[2]}")


    return evaluation, cost_list, accuracy_list

def plot_ca(cost_list:list, accuracy_list: list) -> None:
    #plot the cost and accuracy
    fig, (ax1, ax2) = plt.subplots(nrows=2, ncols=1)
    ax1.plot(accuracy_list, color='blue', marker='', linestyle='-')
    ax2.plot(cost_list, color='blue', marker='', linestyle='-')
    ax1.set_xlabel('Iteration')
    ax1.set_ylabel('Accuracy')
    ax1.set_title('Accuracy over Time')
    ax1.axhline(y=accuracy_list[-1], color='r', linestyle='--', label = str(accuracy_list[-1]))
    ax1.text(len(accuracy_list) // 2, accuracy_list[-1] - 0.1, str(accuracy_list[-1]), color='r', va='center', ha='right', backgroundcolor='white')
    ax1.set_xticklabels([])

    ax2.set_xlabel('Iteration')
    ax2.set_ylabel('Cost')
    ax2.set_title('Cost over Time')
    ax2.set_xticklabels([])

    plt.tight_layout()
    plt.show()

```

```python
    return None

if __name__ == "__main__":
    #nw = linear_load("mnist_linear.weights")
    #imgs = read_images("train-images-idx3-ubyte.gz")
    #labs = read_labels("train-labels-idx1-ubyte.gz")

    # test_img = read_images("t10k-images-idx3-ubyte.gz")
    # test_labs = read_labels("t10k-labels-idx1-ubyte.gz")

    # Code to learn a new network of random weights.
    #learned = learn(imgs, labs, 5, 100)
    #cost_list = learned[1]
    #accuracy_list = learned[2]

    with open('accuracy_list.csv', 'r', newline='') as infile:
        for row in csv.reader(infile):
            acc = row
    with open('cost_list.csv', 'r', newline='') as infile:
        for row in csv.reader(infile):
            cos = row
    cos = [float(cosel) for cosel in cos]
    acc = [float(accel) for accel in acc]
    plot_ca(cos, acc)
    #plot_ca(cost_list, accuracy_list)
    # Code to test trained weight
    # print(evaluate(linear_load("trained.weights"), test_img, test_labs
        ))

    # Code to test random weights
    # print(evaluate(linear_load("random.weights"), read_images(
    #    "train-images-idx3-ubyte.gz"), read_labels("train-labels-idx1-
        ubyte.gz")))

    #labels = read_labels("t10k-labels-idx1-ubyte.gz")
    #images = read_images("t10k-images-idx3-ubyte.gz")
    #filename = "mnist_linear.weights"
    #nw = linear_load(filename)
    #predicions = evaluate(nw, images, labels)
    #print(f"cost: {predicions[1]} and accuracy: {predicions[2]}")
    #plot_images(images, labels, Matrix(nw[0]), predicions[0])
```

LinAlg.py

```python
"""
Homemade linear algebra module to use for MNIST.

This module provides Classes and methods for basic linear algebra

## Classes:
    LinAlg: This is a base class for linear algebra
    Matrix: provides various matrix operations

## Example use
    >>> from linalg import Matrix
```

```

    # Create a matrix
    >>> A = Matrix([[1, 2], [3, 4]])

    # Print the matrix
    >>> print(A)
    | 1 2 |
    | 3 4 |
    <BLANKLINE>

    # Matrix addition
    >>> B = Matrix([[5, 6], [7, 8]])
    >>> C = A + B
    >>> print(C)
    |  6  8 |
    | 10 12 |
    <BLANKLINE>
"""

from typing import Type, Union, List

from scipy import linalg

Mat = Type["Matrix"]
matrix_input = Union[List[List[Union[int, float]]], List[Union[int,
    float]]]


class LinAlg:
    """
    Base class for linear algebra operations
    """

    def col_space(self):
        """
        Method to return the column space of a LinAlg class

        ## Example use
        >>> m = LinAlg()
        >>> m.elements = [[1, 2], [3, 4]]
        >>> m.col_space()
        2
        """
        return len(self.elements[0])

    def row_space(self):
        """
        Method to return the row space of a LinAlg class

        ## Example use
        >>> m = LinAlg()
        >>> m.elements = [[1, 2], [3, 4]]
        >>> m.row_space()
        2
        """
        return len(self.elements)

```

```python
    def __iter__(self):
        """
        Method to run when a iterator is called on a LinAlg class
        """
        self.idx = 0
        return self

    def __next__(self):
        """
        Method to return the next element in a LinAlg class
        """
        if self.idx < len(self.elements):
            x = self.elements[self.idx]
            self.idx += 1
            return x
        else:
            raise StopIteration

    def __getitem__(self, i: int):
        """
        Method to return a element at index of vector
        """
        return self.elements[i]


class Matrix(LinAlg):
    """
    Represents a matrix and provides various matrix operations.
    """

    def __init__(self, elements: matrix_input) -> None:
        """
        initiate a 2d-matrix class

        Args:
        1. Elements of type Union[List[List[Union[int, float]]], List[
           Union[int, float]]]

        Returns:
        * None

        ## Example use
        >>> m = Matrix([[1, 2], [3, 4]])
        >>> m.elements
        [[1, 2], [3, 4]]
        >>> v = Matrix([1, 2, 3])
        >>> v.elements
        [[1, 2, 3]]
        """
        assert isinstance(elements, list), "elements must be a list"

        # Vector input
        if all(isinstance(item, (int, float)) for item in elements):
            self.elements = [elements]

        else:  # 2D matrix
```

```python
            assert all(isinstance(sublist, list) for sublist in elements) and all(len(sublist) == len(
                elements[0]) for sublist in elements), "elements must be a list of lists with same length"
            assert all(isinstance(item, (int, float))
                       for sublist in elements for item in sublist), "sublist must contain only integers or floats"
            self.elements = elements

        self.row_vector = self.row_space() == 1

        return None

    def add(self, matrix: Mat) -> Mat:
        """
        Addition of two matrices of same dimensions

        ## Example use
        >>> A = Matrix([[1, 2], [3, 4]])
        >>> B = Matrix([[5, 6], [7, 8]])
        >>> C = A.add(B)
        >>> print(C)
        |  6   8 |
        | 10  12 |
        <BLANKLINE>
        """
        assert isinstance(
            matrix, Matrix), "Addition is only defined between two matricies."
        assert self.row_space() == matrix.row_space() and self.col_space() == matrix.col_space(
        ), "addition is only defined between matricies with the same row and column dimension."

        return Matrix([[x+y for (x, y) in zip(row_self, row_other)] for row_self,
                       row_other in zip(self.elements, matrix.elements)])

    def __add__(self, matrix: Mat) -> Mat:
        """
        Method for addition of matrices of same dimensions

        ## Example use
        >>> A = Matrix([[1, 2], [3, 4]])
        >>> B = Matrix([[5, 6], [7, 8]])
        >>> C = A + B
        >>> print(C)
        |  6   8 |
        | 10  12 |
        <BLANKLINE>
        """
        return self.add(matrix)

    def __str__(self) -> str:
        """
        Method to print a matrix
```

```

        ## Example use
        >>> A = Matrix([[1, 2], [3, 4]])
        >>> print(A)
        | 1 2 |
        | 3 4 |
        <BLANKLINE>

        """
        max_width = max(max(len(str(x)) for x in row) for row in self.elements)
        matrix_str = ""
        for row in self.elements:
            matrix_str += "| " + \
                " ".join(f"{x:>{max_width}}" for x in row) + " |\n"
        return matrix_str

    def sub(self, matrix: Mat) -> Mat:
        """
        Define subtraction between matrices as the elementwise inverse addition

        ## Example use
        >>> A = Matrix([[5, 6], [7, 8]])
        >>> B = Matrix([[1, 2], [3, 4]])
        >>> C = A.sub(B)
        >>> print(C)
        | 4 4 |
        | 4 4 |
        <BLANKLINE>
        """
        return self.add(-1 * matrix)

    def __sub__(self, matrix: Mat) -> Mat:
        """
        Subtract Matrices of same dimensions

        ## Example use
        >>> A = Matrix([[5, 6], [7, 8]])
        >>> B = Matrix([[1, 2], [3, 4]])
        >>> C = A - B
        >>> print(C)
        | 4 4 |
        | 4 4 |
        <BLANKLINE>
        """
        return self.sub(matrix)

    def fact_mult(self, factor: int | float) -> Mat:
        """
        Factor multiplication for a matrix and a number

        ## Example use
        >>> A = Matrix([[1, 2], [3, 4]])
        >>> B = A.fact_mult(2)
        >>> print(B)
        | 2 4 |
```

```
        | 6 8 |
        <BLANKLINE>
        """
        assert isinstance(
            factor, (int, float)), "factor multiplication of matricies is only defined with integers or floats."
        return Matrix([[factor*x for x in row] for row in self.elements])

    def transpose(self) -> Mat:
        """
        Transpose a matrix

        ## Example use
        >>> A = Matrix([[1, 2], [3, 4]])
        >>> B = A.transpose()
        >>> print(B)
        | 1 3 |
        | 2 4 |
        <BLANKLINE>
        """
        return Matrix([[row[i] for row in self.elements] for i in range(len(self.elements[0]))])

    def mat_mult(self, matrix: Mat) -> Mat:
        """
        Define matrix multiplication for matrices of compatible dimensions

        ## Example use
        >>> A = Matrix([[1, 2], [3, 4]])
        >>> B = Matrix([[2, 0], [1, 2]])
        >>> C = A.mat_mult(B)
        >>> print(C)
        |  4  4 |
        | 10  8 |
        <BLANKLINE>
        """
        assert isinstance(
            matrix, Matrix), "matrix multiplication is only defined between matricies"
        assert self.col_space() == matrix.row_space(
        ), "columnspace and rowspace of the matricies do not match."
        return Matrix([[sum(a * b for a, b in zip(row, col)) for col in zip(*matrix.elements)] for row in self.elements])

    def __mul__(self, other: Mat | int | float) -> Mat | int | float:
        """
        Define multiplication operator to use matrix-product for matricies and scalar multiplication for factors.

        ## Example use
        #matrix multiplication
        >>> A = Matrix([[1, 2], [3, 4]])
        >>> B = Matrix([[2, 0], [1, 2]])
        >>> C = A * B
        >>> print(C)
```

```
        |   4   4  |
        |  10   8  |
        <BLANKLINE>

        #factor multiplication
        >>> D = A * 2
        >>> print(D)
        | 2 4 |
        | 6 8 |
        <BLANKLINE>
        """
        assert isinstance(
            other, (Matrix, int, float)), "Matrix multiplication is only
                defined with scalars and other matricies"
        if isinstance(other, Matrix):
            return self.mat_mult(other)
        return self.fact_mult(other)

    def __rmul__(self, factor: int | float) -> Mat:
        """
        Define scalarmultiplication for rhs

        ## Example use
        >>> A = Matrix([[1, 2], [3, 4]])
        >>> B = 2 * A
        >>> print(B)
        | 2 4 |
        | 6 8 |
        <BLANKLINE>
        """
        assert isinstance(
            factor, (int, float)), "Matrix multiplication is only
                defined with scalars and other matricies"
        return self.fact_mult(factor)

    def pow(self, n: int, elementwise: bool = True) -> str:
        """
        Define powers of vectors as the elementwise power.


        ## Example use
        #Elementwise
        >>> A = Matrix([[1, 2], [3, 4]])
        >>> B = A.pow(2)
        >>> print(B)
        |   1   4  |
        |   9  16  |
        <BLANKLINE>

        #Matrix power
        >>> C = A.pow(2, elementwise = False)
        >>> print(C)
        |   7  10  |
        |  15  22  |
        <BLANKLINE>
        """
        assert isinstance(
```

```python
            n, int), "elementwise power of matricies is only defined for factors"
        if elementwise:
            return Matrix([[x**n for x in row] for row in self.elements])
        else:
            mat = Matrix(self.elements)

            for _ in range(n-1):
                mat *= self
            return mat

    def __pow__(self, n: int):
        """
        Will only work for elementwise power

        ## Example use
        >>> A = Matrix([[1, 2], [3, 4]])
        >>> B = A ** 2
        >>> print(B)
        |  1   4 |
        |  9  16 |
        <BLANKLINE>
        """
        return self.pow(n)

    def flatten(self) -> Mat:
        """
        Flattens the matrix into a vector

        ## Example use
        >>> A = Matrix([[1, 2], [3, 4]])
        >>> B = A.flatten()
        >>> print(B)
        | 1 2 3 4 |
        <BLANKLINE>
        """
        return Matrix([[val for row in self.elements for val in row]])

    def reshape(self, cols: int) -> Mat:
        """
        Reshapes the matrix to a square matrix of size cols:

        >>> A = Matrix([1, 2, 3, 4])
        >>> B = A.reshape(2)
        >>> print(B)
        | 1 2 |
        | 3 4 |
        <BLANKLINE>
        """
        values = self.flatten()[0]
        assert (len(values) ** 0.5).is_integer(
        ) or cols == 1, "size of matrix must satsify len((sqrt(matrix))) is an integer"
        return Matrix([values[i:i+cols] for i in range(0, len(values), cols)])

```

```


if __name__ == "__main__":
    # m1 = Matrix([[1, 2, 3], [1, 2, 3], [1, 2, 3]])
    # m3 = Matrix([[1, 2, 3], [1, 2, 3], [1, 2, 3]])
    # m2 = Matrix([[1], [3], [4], [5]])
    A = Matrix([[5, 6], [7, 8]])
    B = Matrix([[1, 2], [3, 4]])
    C = A.sub(B)
    print(C)
    # print(m1*m2, m1.transpose(), m2.transpose())
    # print(m1+m3, m1*m3, m2.flatten(), m3.pow(2, elementwise=False), m3
        .transpose(), sep="\n")
```