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 is a 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:

```
1 def predict(network: NetW, image: img) -> Matrix:
2   x = image_to_vector(image)
3   A = Matrix(network[0])
4   b = Matrix(network[1])
5   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:

```
1 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 and assertion 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:

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

2.3 Dependencies

Random, matplotlib, gzip, json

2.4 Visualisation

2.5 Ideas for optimization

3 Reflection upon implementation

3.1 Challenges during development