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.

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:

```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:

```python
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:

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

2.3 Dependencies

Random, matplotlib, gzip, json

2.4 Visualisation

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 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:

```python
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.