Evaluate Functional Programming Solutions

Yiqiao Yin
Ph.D. Student

Abstract

This report develops a comparison in matrix form of two functional programming languages. The report summarizes some technical papers in regarding to the particular programming process of the functional programming languages and their workflow required to be put in production. It concludes different scenarios with technical explanation and sample code in regards to the programming structure. Moreover, this report evaluates the programming and scripting languages such as PowerShell across different applications.

1 Introduction

This report explores the internal relationship of different functional programming languages as software development solutions. The report aims to develop a comparison matrix of selected functional programming languages. In addition, the report discusses different scenarios from different technical perspectives such as repetitive processes, computation and calculations, and procedural steps taken to execute the program.

The functional programming languages is a type of computer programming language that is designed to take human thought process and execute in different stages. Each stage can be sent in another and the procedure acts like a mathematical function Goldberg (1996). It also is more difficult to learn than its many peers due to its internal relationship with mathematics Khanfor and Yang (2017). Many researchers and scholars have argued that functional programmers are able to execute mathematical ideas at a much higher magnitude Goldberg (1996); Hudak (1989); Hughes (1989); Khanfor and Yang (2017); Wadler (1992).

The previous report or assignment discussed detailed information of what constitutes procedural languages such as FORTRAN or C. The variables are required to be defined before any assignment or modification. In basic mathematical form, a function is a map from one set to another and we can write

$$f : x \to y \tag{1}$$

where x is the input of the function and y is the output. In set theory, x is also known as the domain where y is known as the range. While this functional form is clearly defined mathematically, it yet lacks the mobility to be transferred into a program. This is the motivation for functional programming languages. Hence, a pseudo code of a functional programming language may look like the following

```
let x = 2
define f(x):
    return x*2

print(f(2)) # this would produce 4, because 2*2=4
```

This is very similar to the lambda-function in python, which we present a similar example below.

```
y = lambda x: x*2
print(y(2)) # this will return 4, because 2*2 is 4
```

Due to the internal structure of the coding design, software programs written using functional programming languages are compiled to send into execution. Take the above simple operation, $2 \times 2 = 4$, as an example. The functional programming language writes a function that lives abstractly (it is not ran yet) but also physically (in an actual directory). The program cannot be broken apart. In other words, we cannot run "define f(x):" and then run "return x*2" separately. The program is by itself a whole and it executes as a whole body. Hence, this is why functional programming languages are often times accompanied with scripting language such as PowerShell.

2 Motivations

This section introduces the motivation of our investigation. Software engineers, data scientists, and IT professionals are the major support system of today's technology improvement especially in large corporations. Their workflow can consist of applying their knowledge and coding experience to design creative software platforms are smart and intelligent to serve certain business functions or directly serve the consumers. The above section as well as the previous assignment we have discussed the internal coding structure of procedural programming languages (from the last assignment) and functional programming languages (in this assignment). However, the motivation has not yet been clearly discussed.

One important theme of software development is the repetitive processes that need to be omitted from the software pipeline. This could refer to as easy as redefining a variable or as complicated as rewriting an entire script that has thousands of lines of code. It is the responsibility of the software engineers and data scientists to ensure the efficiency of the production chain.

2.1 Repetitive Process and Calculations

In a data science project, the workflow starts with a motivational research question that usually addresses certain business needs. The research leads to an optimal machine learning model or algorithm of which can be used every time the client faces the same task. Upon the approval of this model, the repetitive process occurs whenever the model is deployed. This calls for the need of packaging code and ship to production environment. The model or the algorithm does certain tasks that can involve certain level of mathematical computation. In this case, calculations are also involved every time the function is called.

Consider the following supervised example. There is a task designed to learn from features X and to produce an educated guess of Y. For simplicity purpose, a simple model can be built using weights $\vec{w}$ such that the linear transformation $\vec{w}X$ is fed into a non-linear transformation called sigmoid function that takes the following form

$$\text{output} = \sigma(\text{input}) = \frac{1}{1 + \exp(-\text{input})} \tag{2}$$

where the input is the linear transformation $\vec{w}X$. In this case, the calculation is purely mathematical and the final output can be formally written as

$$\text{output} = \sigma(\vec{w}X) = \frac{1}{1 + \exp(-\vec{w}X)} \tag{3}$$

To search for the most optimal sets of weights $\vec{w}$, it is efficient to use an optimization algorithm called gradient descent. This algorithm requires a for loop that compares the losses of the output with the training data output. The loss function can be formally written as the following

$$\mathcal{L}(\text{output}, \hat{\text{output}}) = \sum_{i=1} \left(\text{output} - \hat{\text{output}}\right)^2 \tag{4}$$

where i is the running index tracking the index of each data point in the training sample.

The loss function serves as a guide to indicate the amount of mistakes created using this estimated output from the model. The gradient descent helps the program to minimize the loss. Formally, gradient descent takes partial derivative of the loss function with respect to the weight

$$\nabla\mathcal{L}(\text{output}, \hat{\text{output}}) = \frac{\partial}{\partial \vec{w}}\mathcal{L}(\text{output}, \hat{\text{output}}) \tag{5}$$

and hence this requires a for loop to iteratively update the weights using the gradients calculated above. The for loop updates the weights using the following equation

$$\vec{w}_s := \vec{w}_{s-1} - \eta\nabla\mathcal{L}(\text{output}, \hat{\text{output}}) \tag{6}$$

where the running index s tracks the steps in the gradient descent algorithm. In other words, as s increases, the loss of eq. 4 is expected to produce smaller values.

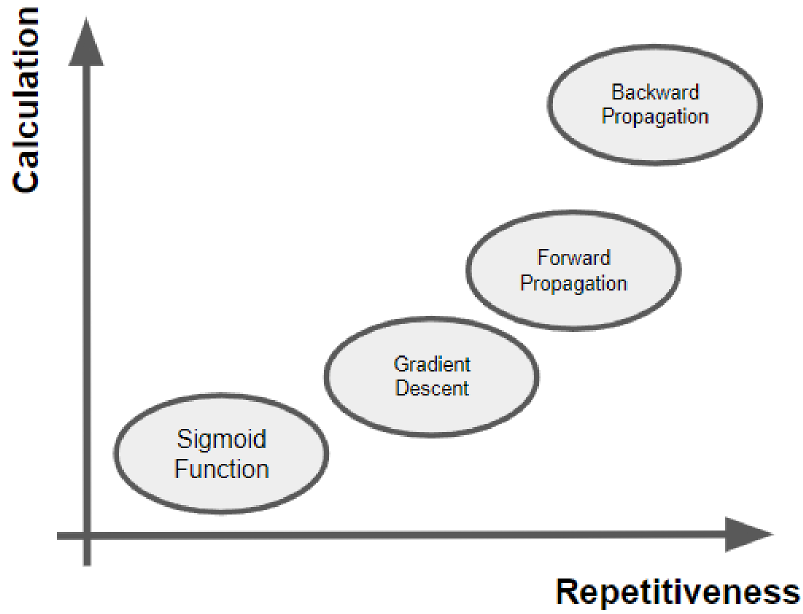
2.2 Matrix Comparison

This subsection we focus on developing a comparison matrix summarizing the attributes from the previous subsection. From the previous subsection, we discussed the mathematical setup of a neural network. The setup provides the technical detail of one unique neuron. In practice, as shown in the C-base implementation, it is possible and sometimes also desirable for the researcher to develop deeper neural network models. The development of neural network models consist of a forward propagation and a backward propagation. The forward propagation consists of repetitively feed in neurons with a linear transformation and a non-linear transformation. The backward propagation consists of using gradient descent to update the weights in the neural network layer by layer in a backward manner.

The core mathematical concept of sigmoid function itself is a very simple task. It requires very simple calculation and there is no repetitiveness. Hence, an activation such as a sigmoid function would sit on the left bottom of the Figure 1.

The gradient descent acts as a building block of backward propagation. The gradient descent must have a for loop to iteratively update the weights using gradient which is partial derivative of the loss function. Hence, it requires higher calculations and much

Figure 1: **Comparison Matrix**. The matrix presents the scenarios discussed in previous section with respect to repetitiveness and level of calculations.



higher repetitiveness than a simple sigmoid function. Hence, gradient descent would sit on the top right corner of the sigmoid function. Forward propagation can increase some calculations, however, the repetitiveness is the key component that is killing the memory. Hence, forward propagation would be at the top right corner of the gradient descent algorithm.

The backward propagation is essentially many gradient descent algorithms propagating backward from the output layer to the first layer, because the algorithm is updating the weights. Hence, the backward propagation would sit on the very top right corner of the diagram in Figure 1.

These four components (sigmoid function, gradient descent, forward propagation, and backward propagation) are all listed in Figure 1.

3 Solution

3.1 Address Technical Explanation

Now that the algorithm and model is clearly written above mathematically it is up to computer scientist to execute these mathematical expression into computer program. Due to the length of the C-based implementation, we refer our readers to this source[1], which gives us the C implementation of a simple neural network that follows the same mathematical workflow above.

The technical solution can be proposed using functional programming. First, the entire script in source mentioned above is in a script. In other words, the script can be called upon desire and the program executes with a line of code in PowerShell. PowerShell is a

[1]Source: To see the link, press here.

very powerful command language used by the Windows system. It is designed to automate the task and enable simplified configuration when execute a scripted programming software. In this case of the report, the C-based implementation can be executed using PowerShell.

The next step is to write the idea from the above mathematical expressions into the code. Upon define the variables, we need

```
double sigmoid(double x) { return 1 / (1 + exp(-x)); }
double dSigmoid(double x) { return x * (1 - x); }
double init_weight() { return ((double)rand())/((double)RAND_MAX); }
```

and these variables are defined globally so that they can be used directly downstream anywhere in the script. Then we shuffle the data and this code is omitted because it is trivial for the purpose of this report. Then there are the following for loop to executes the main body of the neural network. The first for loop defines the hidden weights. The second for loop defines the bias. The third for loop defines the output layer.

```
for (int i=0; i<numInputs; i++) {
    for (int j=0; j<numHiddenNodes; j++) {
        hiddenWeights[i][j] = init_weight();
    }
}
for (int i=0; i<numHiddenNodes; i++) {
    hiddenLayerBias[i] = init_weight();
    for (int j=0; j<numOutputs; j++) {
        outputWeights[i][j] = init_weight();
    }
}
for (int i=0; i<numOutputs; i++) {
    outputLayerBias[i] = init_weight();
}
```

The above wraps up the design of the neural network which corresponds to eq. 2 and eq. 1.

The body of the neural network is the main part and component that requires the most amount of iteration, repetitiveness processes, and calculations. There is a big for loop covers up the forward propagation and the backward propagation. The pseudo code is presented as the following

```
for (int n=0; n < 10000; n++) {
    \\ shuffle
    for (int x=0; x<numTrainingSets; x++) {
        // Forward pass
        for (int j=0; j<numHiddenNodes; j++) {
            ... \\ omit the body here
        }

       // Backprop
        double deltaOutput[numOutputs];
        for (int j=0; j<numOutputs; j++) {
            ... \\ omit the body here
        }
    }
```

```
    \\backward pass
}
```

Though more cumbersome than its python version, the script does get the job done and it executes faster than python in its most basic operations. To execute this in a PowerShell, we can run the following code

```
cd "C:\path\to\" # whatever path desired
.\"a_neuralnetwork_model.exe"
```

As a summary, this section addresses some of the scenarios where the operation can consist of repetitive processes and calculations that may not be efficient to be programmed line by line. Hence, the technical solution proposed is to write them using functional programming. In this section, we provide a small neural network example with C-based implementation to showcase that workflow can be much more efficient when functional programming is used.

3.2 Functional Programming as Solution

The matrix discussed in the first section lays out the major challenges of the repetitiveness process and the level of complexities in calculations of programming environment. The beginning subsections of this section lays out the potential solutions using programming functions. In addition, the report sourced a C-based implementation of the neural network model to demonstrate the level of efficiency that can be done using functional programming.

As a conclusion, the report exhibits evidence to showcase the major benefits of functional programming language with a real world modeling example of neural networks.

References

Goldberg, B. (1996). Functional programming languages. *ACM Computing Surveys (CSUR)*, 28(1):249–251.

Hudak, P. (1989). Conception, evolution, and application of functional programming languages. *ACM Computing Surveys (CSUR)*, 21(3):359–411.

Hughes, J. (1989). Why functional programming matters. *The computer journal*, 32(2):98–107.

Khanfor, A. and Yang, Y. (2017). An overview of practical impacts of functional programming. In *2017 24th Asia-Pacific Software Engineering Conference Workshops (APSECW)*, pages 50–54. IEEE.

Wadler, P. (1992). The essence of functional programming. In *Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–14.