Neural Networks: Python vs. C# vs. C++

CMPS 203: Project Report

Noujan Pashanasangi
npashana@ucsc.edu

Juraj Juraska
jjuraska@ucsc.edu

March 17, 2017

1 Introduction

Our project was motivated by the increasing interest in machine learning research and application. One of the most powerful learning models currently available is neural network. We set as our goal to compare three widely used programming languages, namely Python, C# and C++, in implementing neural networks. One of the classic applications of neural networks is handwritten digit recognition. Therefore, we implemented such a neural network in these three languages and compared them in different aspects, such as performance and expressiveness.

2 Neural Networks

Neural network is a machine learning model that excels in tasks such as regression, classification and various types of data processing. Neural networks have only recently been employed to solve a number of tough problems more successfully than any of the previous methods, and they have significantly contributed to the rapid progress in computer vision and natural language processing.

A neural network is built of simple computational units called neurons. Each neuron typically has multiple inputs and a single output. It computes a weighted sum of inputs, and applies an activation function to the sum to compute the output. Neurons in the network are arranged in multiple layers. Although the first layer, or the input layer, is represented as neurons, it merely forwards the input data to the next layer. The output of the last layer, or the output layer, represents the result. All the layers in between, called hidden layers, enable the network to learn features of the data. The inputs of each neuron are the outputs of the neurons in the previous layer. There is a weight assigned to each pair of neurons in two consecutive layers. When computing the weighted sum of inputs of each neuron, the outputs of the corresponding neurons in the previous layer is multiplied by the respective weights. Each neuron also has a value called bias which it adds to the weighted input sum before applying the activation function. The weights and biases are what the neural network ultimately learns using an optimization algorithm, such as the gradient descent.

2.1 Data

In our project, the implemented neural network classifies images of handwritten digits into ten groups representing the digits from 0 to 9. The data we use to train and test our network is the MNIST database[1] containing $70,000$ samples. Each sample contains a scanned image of a digit and a label $(0-9)$ specifying the digit. The resolution of each image is 28×28 pixels. Each pixel is represented by its intensity with a number between 0 and 1.

This dataset is available in different formats to the general public so as to enable people to test their learning models on real-world data of a considerable size. It is split into a training set of $60,000$ samples and a test set of the remaining $10,000$ samples. We used $50,000$ samples of the training set for training, and we left the remaining $10,000$ for validation. The images are

[1] http://yann.lecun.com/exdb/mnist/

already preprocessed for the digits to be centered and their sizes to be normalized. This lets us focus on the machine learning model, i.e. the neural network.

2.2 Design

Our neural network only contains a single hidden layer with 30 neurons. The input layer consist of 784 neurons, corresponding to $784 = 28 \times 28$ pixels. There are 10 neurons in the output layer, each of which outputs a number between 0 and 1. This number corresponds to the probability of the class represented by the output neuron being the true label of the image. The corresponding class of the output neuron with the greatest output value is the network's prediction of the true label of the input image (see Figure 1).

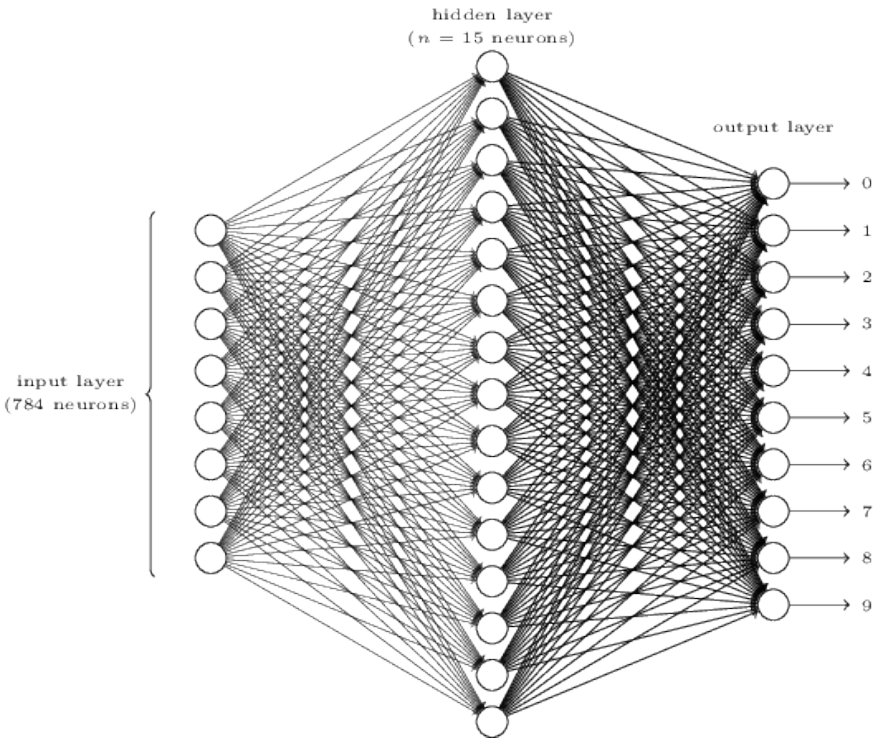

Figure 1: Architecture of our neural network

We initialize the weights randomly from the normal distribution and use the sigmoid function as the neurons' activation function. Stochastic gradient descent via backpropagation is the optimization algorithm we use to enable the network to learn the weights.

The sigmoid function is defined as follows:

$$\sigma(z) = \frac{1}{1 + \exp(-z)}$$

Hence, the output of a sigmoid neuron with inputs $x_1, x_2, \ldots$ and weights $w_1, w_2, \ldots$, is:

$$\frac{1}{1 + \exp\left(-\sum_j w_j x_j - b\right)}$$

where b is the bias of that sigmoid neuron.

Finally, we use the cross-entropy function as our cost function, which we minimize using the gradient descent. The cross-entropy cost function is defined as follows:

$$C = -\frac{1}{n} \sum_x [y \ln a_{out} + (1 - y) \ln(1 - a_{out})]$$

where a_{out} is the output of the neural network for sample x, and n is the number of the training samples. The cost function takes into account every sample in our training set.

As mentioned before, we use stochastic gradient descent to minimize the cost function and enable the network to learn the weights. The gradient descent can be done in two ways, iterative or vectorized. The vectorized version replaces a significant number of matrix-vector operations by only a few matrix-matrix operations. Since these matrix-matrix operations can be performed more efficiently than an equivalent set of matrix-vector operations, the vectorized version improves the performance substantially. In order to be able to utilize these operations in different languages, we need to use different libraries. This also gives us a good platform to compare our three programming languages with respect to the available libraries and their performance.

3 Languages

Training a neural network is computationally very expensive and, therefore, we expect it to be a good indicator of the programming languages' run-time performance. Furthermore, writing the code for the same neural network in three different languages, we will be able to compare their expressiveness, as well as the convenience of their features in the neural network development process.

We used the basic neural network developed by Nielsen [1] as the basis of our work. We implemented the neural network described in the previous section in Python, C# and C++.

3.1 Implementation Steps

The implementation consists of the following steps:

i Data loading

ii Neural network initialization

iii Training

iv Testing

Since the dataset is available in different formats, for each language we decided to choose the format that is the most convenient to work with. Although we test the running time of the data loading steps, the comparison does not provide relevant results since the data are loaded from different formats in each of the languages. The goal here was to examine the convenience of these languages to read the same data in the simplest way possible.

The neural network performs the bulk of the work in the training step. Hence, we chose to evaluate the running time of this step developed in the different languages to compare their run-time performance on the same neural network.

3.2 Python

The implementation was done usin Python 3.5. For the first step of the implementation, the data loading, we used the Python-native Pickle format, which appeared to be the most straightforward to work with in Python.

For the training step, particularly for the vectorized version, we used a linear algebra library. Python's popular library NumPy[2], which is implemented in C, is the best choice one has for this task. We use NumPy arrays, which represent vectors and matrices, and their arithmetic operations to achieve vectorization. The Python version of our neural network has about 250 lines of code.

```
# compute the output error
delta = self.cost.derivative(weighted_inputs[-1], activations[-1], y)
nabla_w[-1] = np.dot(delta, activations[-2].transpose())
nabla_b[-1] = delta.sum(1)[:, np.newaxis]

```

[2] http://www.numpy.org/

```
# backpropagate the error layer by layer (starting from the output layer)
for layer in range(2, self.num_layers):
    z = weighted_inputs[-layer]
    delta = np.dot(self.weights[-layer + 1].transpose(), delta) * sigmoid_prime(z
    )

    nabla_w[-layer] = np.dot(delta, activations[ layer - 1].transpose())
    nabla_b[-layer] = delta.sum(1)[:, np.newaxis]
```

Listing 1: Code snippet from backpropagation

One of the most convenient features of the Python language is the list comprehension, which serves for a quick and concise creation of a complex list instead of doing it in a loop (see Listing 2). Another Python-specific feature is the negative indexing, which makes accessing elements of a list in reversed order easier (see Listing 1). This proved especially useful in the backpropagation, where the weights are updated layer by layer starting at the output layer. Moreover, the NumPy library enables us to perform column-wise addition of a vector to a matrix by simply using the addition operation (see Listing 3), which is a distinctive feature of this library when compared to the linear algebra libraries we used in the other two languages.

```
nabla_w = [np.zeros(w.shape) for w in self.weights]
```

Listing 2: Code snippet showing list comprehension

```
z = np.dot(w, a) + b
```

Listing 3: Code snippet showing vector-matrix addition

It is important to mention that Python is an interpreted language and strongly typed. However, type errors are only prevented at runtime, as Python does not use static type checking due to the absence of a compiler that would check the type constraints.

3.3 C#

The code was written in C# 6.0. For the data loading, we used the JSON format, a common platform-independent data-exchange format, which we found the easiest to work with among the available formats.

For the vectorized version of the training step, we used the Math.NET[3] library, which is implemented in C# itself. We use the *matrix<double>* and *vector<double>* data types from this library, representing matrices and vectors, respectively, along with their available operations. Our program written in C# is approximately 350 lines long. Althought this is 40% more than the Python code, it is mostly due to the C# coding conventions. Those make the code significantly less compact compared to an equivalent Python code, but arguably more readable.

```
// compute the output error
Matrix<double> delta = CrossEntropyCost.Derivative(weightedInputs.Last(),
    activations.Last(), y);
nablaW[nablaW.Count - 1] = delta * activations[activations.Count - 2].Transpose()
    ;
nablaB[nablaB.Count - 1] = delta.RowSums();

// backpropagate the error layer by layer (starting from the output layer)
for (int layer = numLayers - 3; layer >= 0; layer--)
{
  Matrix<double> z = weightedInputs[layer];
  delta = (weights[layer + 1].Transpose() * delta).PointwiseMultiply(SigmoidPrime
    (z));
  nablaW[layer] = delta * activations[layer].Transpose();
  nablaB[layer] = delta.RowSums();
}
```

Listing 4: Code snippet from backpropagation

[3]https://www.mathdotnet.com/

A convenient feature in the C# language worth mentioning is the LINQ (Language Integrated Query) component. It enables us to perform queries on various data structures in a convenient way (see the *Select* method in Listing 5). Lambda expressions used as the parameters is what gives the LINQ methods power while keeping them concise.

```
var xBatch = Matrix<double>.Build.DenseOfColumns(miniBatch.Select(x => x.Item1.
    ToColumnMajorArray()));
```

Listing 5: Code snippet showing the use of LINQ and lambda expressions

In contrast with Python interpreter, C# compiler does static type checking at compile-time. As a result, type errors are caught before running the program, which is certainly very helpful in avoiding unexpected bugs in the program.

3.4 C++

We used C++11 to be specific. We used the data in CSV format for C++, since it was the only format that could be used with the chosen linear algebra library in a straightforward way.

The Armadillo library[4] is a powerful linear algebra library written in C++. We used the *mat* data structure and its available operations, which represents matrices and matrix operations, respectively. Our C++ version of the program is roughly 350 lines long.

```
//compute the output error
mat delta;
costDerivative(activations[num_layers_-1], y, delta);
nabla_w[num_layers_-2] = delta * activations[num_layers_-2].t();
nabla_b[num_layers_-2] = sum(delta, 1);
// backpropagate the error layer by layer (starting from the output layer)
for(int i = num_layers_-3; i >= 0; i--) {
  mat z = weighted_inputs[i];
  mat sigmoid_prime_mult_term = zeros(arma::size(a));
  sigmoidPrime(z, sigmoid_prime_mult_term);
  delta = (weights_[i+1].t() * delta) % sigmoid_prime_mult_term;
  nabla_w[i] = delta * activations[i].t();
  nabla_b[i] = sum(delta, 1);
}
```

Listing 6: Code snippet from backpropagation

One of the distinctive features of C++ is its header files, which can be used to specify the interface of a program. This feature is useful in making the program more encapsulated (see Listing 7.

```
#ifndef DATA_LOADER_H
#define DATA_LOADE_H

#include <iostream>
#include <armadillo>
using namespace std;
using namespace arma;

typedef pair <mat, mat> mpair;

constexpr int training_size = 50000;
constexpr int test_size = 10000;
constexpr int input_layer_size = 784;

void tester();
void dataLoader(vector <mpair> &, vector < pair <mat, int> > &);
mat vectorizedResults(int);

#endif
```

Listing 7: Code snippet showing the DATA_LOADER header file

[4] http://arma.sourceforge.net/

In contrast to Python and C#, in which arguments are passed by value or by reference depending on the argument's data type, in C++ it is possible to decide the passing method. We can see examples of both passing by reference and value in Listing 8.

```
void updateMiniBatch(vector <mpair>& mini_batch, double learning_rate, double
    reg_param, int n);
```

Listing 8: Code snippet showing call by reference

Similar to C#, in C++ type errors are caught at compile-time too. The probability of encountering an unexpected error during run-time is thus decreased.

4 Evaluation

The most valuable test with respect to this project's goals is the running time evaluation of the three implementations. We performed all the evaluations on a computer using Core i7-5600U running at 2.6 GHz. We can see the results in Table 1.

Table 1: Running time comparison

	Data loading [s]	Iterative [s]	Vectorized [s]
Python	0.82	87.23	14.61
C#	19.93	395.18	74.92
C++	13.33	57.67	10.23

As we mentioned before, the running time for data loading cannot be used to compare the performance of the languages, since we used different input data formats in each of them. The running time of the neural network's training, however, shows a clear performance difference. We should focus on the difference between the three iterative versions, and the difference between the three vectorized versions. Also, the different ratios of vectorized version's performance to the iterative version's is worth observing.

It is clear from the results that the C++ implementation was the fastest among the three, and that in both the iterative and the vectorized version. The Python implementation is 40-50 percent slower. As we can see, the C# implementation is far more behind, performing 7-8 times slower than the C++ implementation. We believe that the primary reason for the C# implementation being so slow is the linear algebra library used. Math.NET is likely slow because, unlike NumPy and Armadillo, it is not written in C/C++.

Optimization plays an important role in improving the running time. All the results in Table 1 are achieved using optimization. To see the difference between running with and without an optimization, we ran the C++ implementation both ways. We notice that the iterative version becomes about 4 times faster when using optimization, but the vectorized version is only about 2.5 times faster. We think the reason is that the optimization improves the performance of loops, which there are many of in the iterative version as opposed to the vectorized version. See Table 2 for the exact running times.

Table 2: C++ optimization effect on running time

	Data loading(s)	Iterative(s)	Vectorized
Without optimization	14.67	212.33	26.67
O2 optimization	13.33	57.67	10.23

5 Conclusion

The main goal of this project was to compare the three selected languages: Python, C# and C++. As we can see, C++ is the winner performance-wise, while we proclaim Python as the most convenient language to program in, at least in this project. Also, among the three different

libraries we used, Python's NumPy offers the most powerful features for implementing neural networks from scratch like we did.

As a result, we can see that there is trade-off between convenience and performance. For the purpose of learning new concepts or just seeing the results of implementing algorithms on specific data, it is easier in general to use Python. However, whenever achieving better efficiency is essential, one is better off choosing C++.

References

[1] Michael A. Nielsen, *Neural Networks and Deep Learning*. Determination Press, 2015.