COP290 (Design Practices)

Assignment Report : Task 1

Kshitij Alwadhi (2019EE10577)[1] and Vaibhav Verma (2019EE10543)[2]

[1]Department of Electrical Engineering, IIT Delhi
[2]Department of Electrical Engineering, IIT Delhi

5 March 2022

Contents

1 Introduction

In the assignment given to us in Task 1, we make a library for classifying a given audio clip into various classes. This library has been broken down into various functions which we finally stitch together at the end. We go over these functions in the first section. We follow this up with the performance comparison while implementing the fully connected layer. At the end, we talk about the API.

2 Functions

Various functions are implemented in this library which are then stitched together to make the API. Here's a brief description of the most important functions:

1. **Fully connected layer**: This function performs the forward propagation step of a fully connected layer. It takes as input the input matrix, the weights matrix and the bias vector. It returns the output matrix.

2. **Activation layer**: This function performs the activation function on the input matrix. It takes as input the input matrix and the activation function. It returns the output matrix.

3. **Pooling**: This function performs the pooling operation on the input matrix. This is used for subsampling. It takes as input the input matrix, the pooling type and the stride length. It returns the output matrix.
4. **Probability**: This function is used for implementing the activation function of the final layer of the neural network (the probability functions for binary/multi-class classification). It takes as input the input matrix and the activation function. It returns the output matrix.

3 Performance Comparison

One of the components used in this library is the Fully Connected Layer. This component comprises of a **matrix multiplication** step (multiplication of the weight matrix with the input matrix). Now there are various ways in which we can perform this matrix multiplication and we explore their performance in this section.

3.1 Matrix Multiplication Approaches

3.1.1 Naive Method

The naive method for matrix multiplication is the $O(n^3)$ implementation which we use as the baseline for making our comparisons. Little to none optimization has been done in this implementation so as to get the complete picture of how much improvement we can get from using other libraries.

3.1.2 Threading (using pthreads)

To make the naive method of matrix multiplication faster, we bring about parallelism in the other for loop of the code. To implement this, we make use of **pthreads** in C++. We restrict the number of threads to 4 for our system in which we perform these experiments.

3.1.3 Linear Algebra Libraries

We also try out various libraries which are highly optimized for linear algebra tasks (matrix multiplication in our case). These libraries include **OpenBlas** and **Intel MKL**. We use these libraries as a black box but under the hood they have various levels of optimizations such as utilising multi-processing and better caching.

3.2 Experimental Setup

To perform our experiments, we perform matrix multiplication of two matrices each of size $n \times n$. These matrices are randomly generated by the code. We vary the parameter n across a range of values and at every iteration, we multiply the previous value of n by 2. Moreover, to get stable results, at each iteration, we perform this calculation 30 times and take the mean/standard deviation.

3.3 Performance Comparison

3.3.1 Performance vs Number of Threads

In this set of experiments, we observe the performance of implementation as we vary the number of threads. We vary the number of threads from 1 to 4.

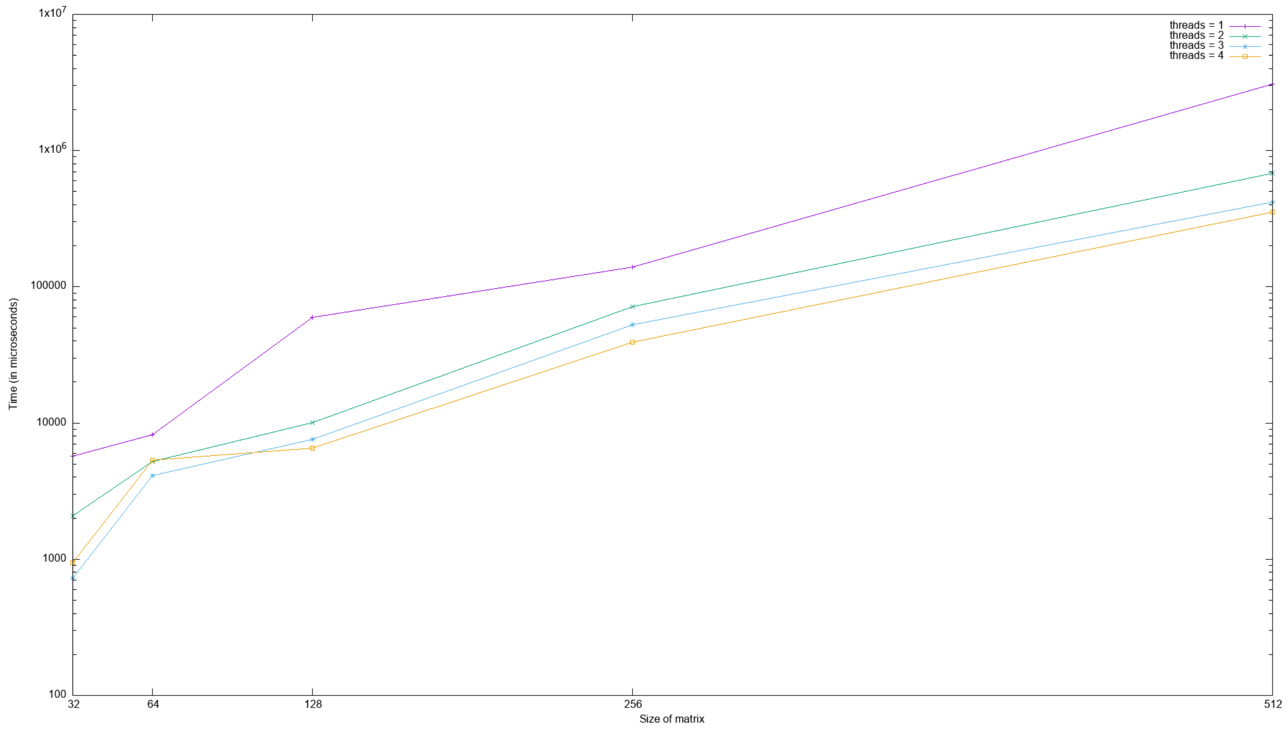


Figure 1: Time vs. Size of Matrix for Fixed Number of Threads

As seen on the X-axis, we have chosen the range [32, 64, 128, 256, 512] for the size of the matrices. We also tried running our implementation for values beyond this range, however, the matrices were too large to fit into the memory. Hence the upper limit of n has been chosen as 512.

n	**num_threads**	**time(μS)**
32	1	5710.13
32	2	2073.8
32	3	730.433
32	4	943.9
64	1	8247.97
64	2	5193.43
64	3	4097.07
64	4	5336.47
128	1	59544.5
128	2	10092.8
128	3	7571.97
128	4	6572.23
256	1	138898
256	2	71755.1
256	3	52945.7
256	4	39400.5
512	1	3074930
512	2	683593
512	3	418104
512	4	352216

Table 1: Performance Comparison across different number of threads

3.3.2 Performance across various implementations

Here, we bring about the comparison across all the implementations that we have in our code. We present the observed data through boxplots for $n = [32, 64, 128, 256, 512]$. Note that the number of threads in the pthread implementation has been set at 4 in these experiments and the time reported is in μS.

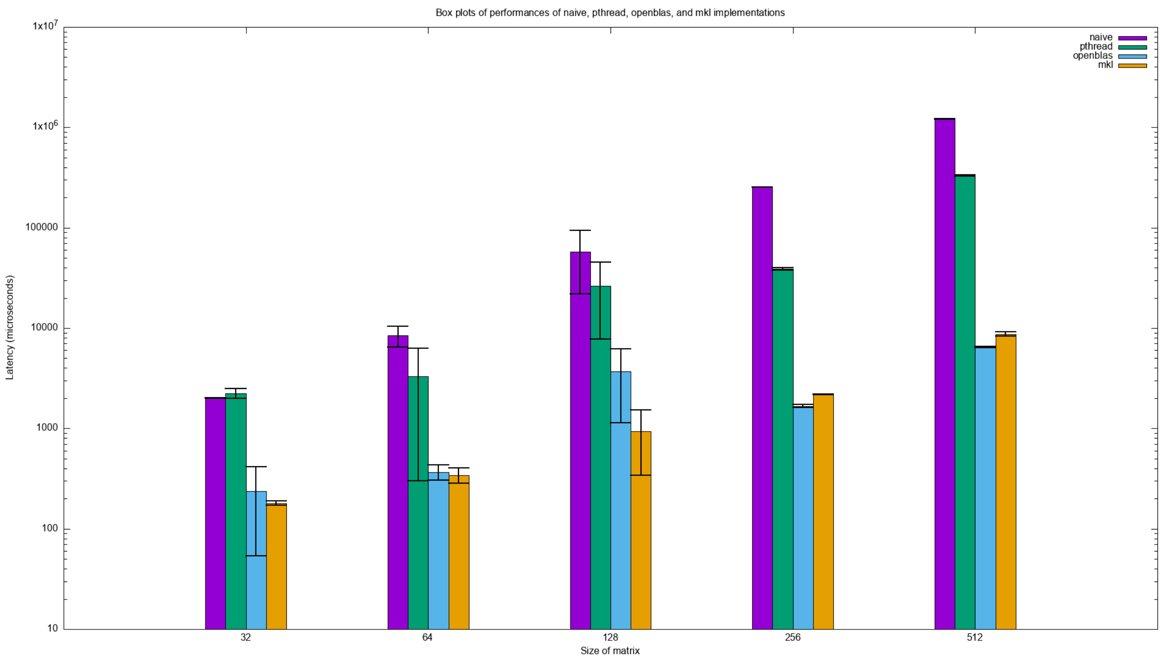


Figure 2: Performance across different implementations

n	Naive	pthread	openBLAS	MKL
32	2009.23	2239.87	235.833	180.333
64	8449.83	3309.23	368.9	343.1
128	57857	26450.8	3674.37	939.7
256	255061	38711.1	1672.27	2191.37
512	1219320	330361	6461.6	8742.47

Table 2: Performance Comparison across different implementations

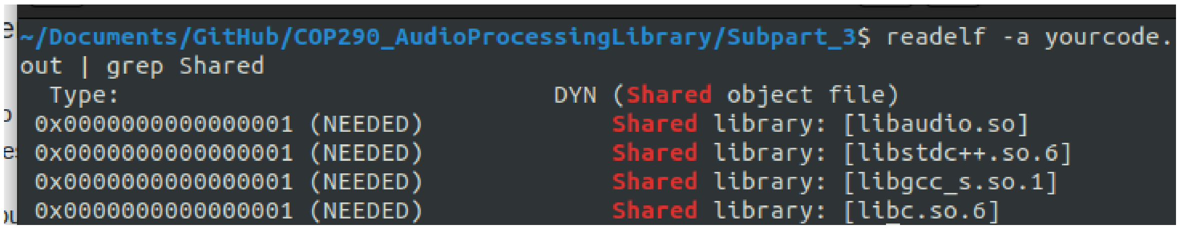
3.4 Conclusion

From figure 1, we conclude that having more number of threads leads to better performance, however, the results don't necessarily scale linearly with the increase in number of threads. From figure 2, we observe that we are able to get a significant increase in performance when we move from our implementations to the linear algebra libraries. The performance of both the libraries (openBLAS and MKL) is similar with openBLAS performing slightly better for bigger inputs. Moreover, we do see an improvement in performance when we switch to pthreads from the naive approach. However the performance of the libraries is superior as they are much more optimized. However, when the input size is small (32), the pthreads approach performs poorer as more time is spent in initializing the threads as compared to time spent in the actual computation.

4 API/Library Implementation

We stitch together the various functions implemented in the subpart 1 and 2 to give ourselves an API which can be used for classifying the audio sample. We first create a shared library for this purpose which is then linked to our 'yourcode.out' file.

Here is the output(4) of the command used to check the libraries used by the executable file:

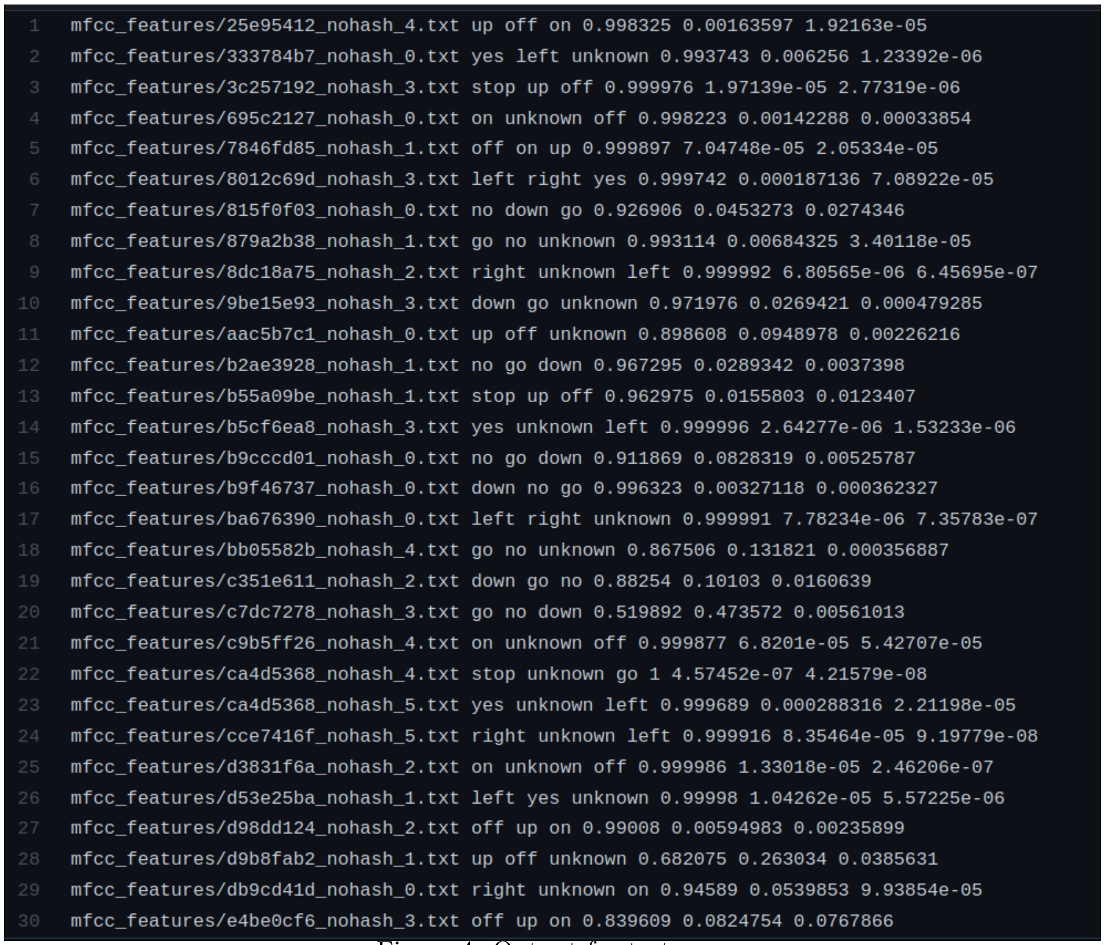


```
~/Documents/GitHub/COP290_AudioProcessingLibrary/Subpart_3$ readelf -a yourcode.
out | grep Shared
  Type:                              DYN (Shared object file)
 0x0000000000000001 (NEEDED)             Shared library: [libaudio.so]
 0x0000000000000001 (NEEDED)             Shared library: [libstdc++.so.6]
 0x0000000000000001 (NEEDED)             Shared library: [libgcc_s.so.1]
 0x0000000000000001 (NEEDED)             Shared library: [libc.so.6]
```

Figure 3: Libraries used by the executable

Here we can see how the executable file depends on the 'libaudio.so' file that we have created using the make command. This is our shared library.

Here is the output that we get by using our library for the given test cases in the format of audio file, top 3 labels, and their associated probabilities.



```
mfcc_features/25e95412_nohash_4.txt up off on 0.998325 0.00163597 1.92163e-05
mfcc_features/333784b7_nohash_0.txt yes left unknown 0.993743 0.006256 1.23392e-06
mfcc_features/3c257192_nohash_3.txt stop up off 0.999976 1.97139e-05 2.77319e-06
mfcc_features/695c2127_nohash_0.txt on unknown off 0.998223 0.00142288 0.00033854
mfcc_features/7846fd85_nohash_1.txt off on up 0.999897 7.04748e-05 2.05334e-05
mfcc_features/8012c69d_nohash_3.txt left right yes 0.999742 0.000187136 7.08922e-05
mfcc_features/815f0f03_nohash_0.txt no down go 0.926906 0.0453273 0.0274346
mfcc_features/879a2b38_nohash_1.txt go no unknown 0.993114 0.00684325 3.40118e-05
mfcc_features/8dc18a75_nohash_2.txt right unknown left 0.999992 6.80565e-06 6.45695e-07
mfcc_features/9be15e93_nohash_3.txt down go unknown 0.971976 0.0269421 0.000479285
mfcc_features/aac5b7c1_nohash_0.txt up off unknown 0.898608 0.0948978 0.00226216
mfcc_features/b2ae3928_nohash_1.txt no go down 0.967295 0.0289342 0.0037398
mfcc_features/b55a09be_nohash_1.txt stop up off 0.962975 0.0155803 0.0123407
mfcc_features/b5cf6ea8_nohash_3.txt yes unknown left 0.999996 2.64277e-06 1.53233e-06
mfcc_features/b9cccd01_nohash_0.txt no go down 0.911869 0.0828319 0.00525787
mfcc_features/b9f46737_nohash_0.txt down no go 0.996323 0.00327118 0.000362327
mfcc_features/ba676390_nohash_0.txt left right unknown 0.999991 7.78234e-06 7.35783e-07
mfcc_features/bb05582b_nohash_4.txt go no unknown 0.867506 0.131821 0.000356887
mfcc_features/c351e611_nohash_2.txt down go no 0.88254 0.10103 0.0160639
mfcc_features/c7dc7278_nohash_3.txt go no down 0.519892 0.473572 0.00561013
mfcc_features/c9b5ff26_nohash_4.txt on unknown off 0.999877 6.8201e-05 5.42707e-05
mfcc_features/ca4d5368_nohash_4.txt stop unknown go 1 4.57452e-07 4.21579e-08
mfcc_features/ca4d5368_nohash_5.txt yes unknown left 0.999689 0.000288316 2.21198e-05
mfcc_features/cce7416f_nohash_5.txt right unknown left 0.999916 8.35464e-05 9.19779e-08
mfcc_features/d3831f6a_nohash_2.txt on unknown off 0.999986 1.33018e-05 2.46206e-07
mfcc_features/d53e25ba_nohash_1.txt left yes unknown 0.99998 1.04262e-05 5.57225e-06
mfcc_features/d98dd124_nohash_2.txt off up on 0.99008 0.00594983 0.00235899
mfcc_features/d9b8fab2_nohash_1.txt up off unknown 0.682075 0.263034 0.0385631
mfcc_features/db9cd41d_nohash_0.txt right unknown on 0.94589 0.0539853 9.93854e-05
mfcc_features/e4be0cf6_nohash_3.txt off up on 0.839609 0.0824754 0.0767866
```

Figure 4: Output for test cases