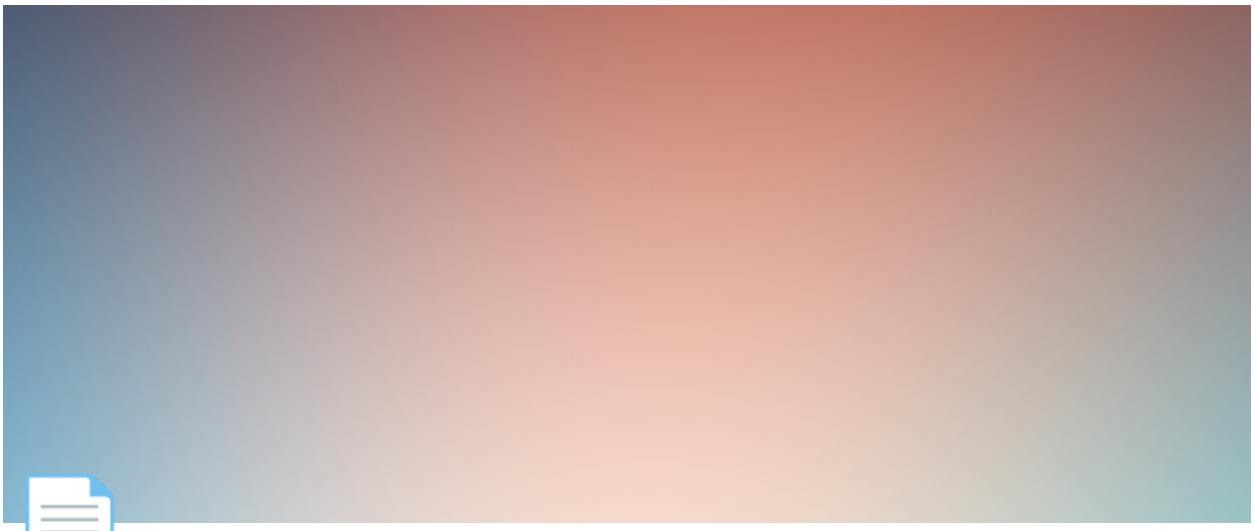


Topic 10: Libraries and Tools

Supervisor:	Posted:	Course:
Dr. Tsui-Wei Weng	Dec 7, 2022	DSC 210 FA'22 Numerical Linear Algebra

Team:

- Tanvi Joshi
  PID: A59019615
  HDSI

1. Introduction

Here, we are comparing and analyzing different tools and libraries based on these parameters:

- Hardware Support
- Time Complexity
- Memory Utilization

1.1 History/Background:

In the mid-term presentation, we proposed that we will be evaluating a few linear algebra operations across all the available and compare their working on each of them.Our primary goal is to identify and compare various modern tools and libraries that perform these NLA operations. As the operation demand large scale computation power, it is increasingly necessary to optimize our methods to reduce the cost involved.

We have compared the libraries on a few linear algebra operations using the in-build linear-algebra functions offered by them.

1.2 Applications:

Linear Algebra is widely used in:

- Deep Learning
- NLP
- Computer Vision
- Machine Learning
- Data Mining

And many more …..

1.3 State-of-the-art:

The libraries that we have compared are:

1. TensorFlow
2. Numpy
3. PyTorch

We tried to compare these libraries on a few parameters

a. Time complexity : Time taken to perform a given linear-algebra operation.

b. Hardware complexity: The various hardware options used by each library to improve the performance while doing complex operations with large data.

---

2. Problem Formulation

2.1 Relation to numerical linear algebra:

Linear algebra has a wide range of applications and as the amount of available data increases, so does the need for better computation. This is why these modern libraries that offer in-built packages for linear algebra operations play a vital role.

2.2 Approach description:

Traditionally these operations were performed in a simplistic approach. Some of the most efficient libraries include BLAS (Basic Linear Algebra Subprograms) and LAPACK (Linear Algebra Package). NLA operations are divided into smaller subprograms and implemented in routines also known as batched processes.

However, this traditional approach has some drawbacks:

- Routines to solve equations
- Block operations redundancy
- Performance load on complex operations

To overcome these hurdles, the modern approaches involves state of the art approach to enhance performance. We will perform certain NLA operations using these modern approaches like

- Numpy
- PyTorch
- TensorFlow

to simulate the real-time complex calculations.

2.3 SOTA Approach description:

Modern Approach offer many advantages like:

- Machine Learning Frameworks
- Hardware Support
- Operational Performance Enhancements

Advantages of libraries and tools:

Numpy:

- Open source library written in Python and C
- Huge collection of NLA operations for multidimensional array
- Fixed array size, sequential memory access
- Matrix operations are performed using BLAS method.
- BLAS implements basic linear algebra routines to solve vector and matrix operations.
- Package: **numpy.linalg** includes number of linear algebra operations
    - Scalar and vector product
    - Matrix operations
    - Solving equations - LU, QR
    - Eigenvalue decomposition
    - Norm

PyTorch:

- Written in: Python, Cuda and C++
- Stores and operates on homogeneous multidimensional rectangular arrays of numbers.
- Huge collection of mathematical operations

- Package: **torch.linalg** includes number of linear algebra operations
  - Scalar and vector product
  - Matrix operations
  - LU decomposition
  - Eigenvalue decomposition
  - Norm

TensorFlow:

- Written in: Python, Cuda and C++
- Graph represents a mathematical operation, and each connection or edge between nodes is a multidimensional data array.
- Huge collection of mathematical operations.
- Package: **tf.linalg** includes number of linear algebra operations
  - Scalar and vector product
  - Matrix operations
  - LU decomposition
  - Eigenvalue decomposition
  - Norm

2.4 Evaluation:

Once algorithms are implemented, we will analyze the process and the resource utilization to compare the efficiency of each algorithm. As of now, we have decided to compare the processes based on factors:

- Time
- FLOPS
- Memory utilization
- Hardware Acceleration

Based on the operation, we can conclude which library or tool is most suited and what are the advantages of the optimization.

---

3. Experiments

3.1 Setup and logistics:

- Understand and Compare various linear algebra operations in Numpy, Tensorflow and Pytorch
  - Vector addition, subtraction, multiplication
  - Matrix multiplication
  - Solving linear equations
  - Linear Decomposition
  - Eigen Decomposition
- **Comparison Factors**
  - Time
  - FLOPS
  - Memory usage
- **Tech stack used:** Google Colab for NumPy, TensorFlow and PyTorch
- **Documentation:**
  - Version Control on Git
  - Proposal and report on Notion

3.2 Dataset and preprocessing:

- Created random matrices of size in the range of 1 to 1000, implementing the NLA operations on all these matrices.
- Checking the time taken by each library as the matrix size increases.
- Changing the runtime to check differences in output with GPU and TPU Hardware accelerator.

3.3 Implementation:

- Randomly created matrix is processed with traditional approach using numpy

1. Eigenvalue decompostion

```
Time_list_numpy=[]
for i in range(1,100,1):
  start=time.time()
  temp= np.random.rand(i,i)
  Lambda, U = np.linalg.eig(np.array(temp))
  end=time.time()
  Time_list_numpy.append(end-start)
```

2. Least Square

```
Time_list_numpy=[]
for i in range(1,1000,1):
  A= np.random.rand(i,i)
  y= np.random.rand(i)
  start=time.time()
  w= np.linalg.lstsq(A.T,y,rcond=-1)
  end=time.time()
  Time_list_numpy.append(end-start)

len(Time_list_numpy)
```

3. Norm

```
Time_list_numpy=[]
for i in range(1,1000,1):
  start=time.time()
  temp= np.random.rand(i,i)
LA.norm(temp,ord=None, axis=None, keepdims=False)
end=time.time()
  Time_list_numpy.append(end-start)
```

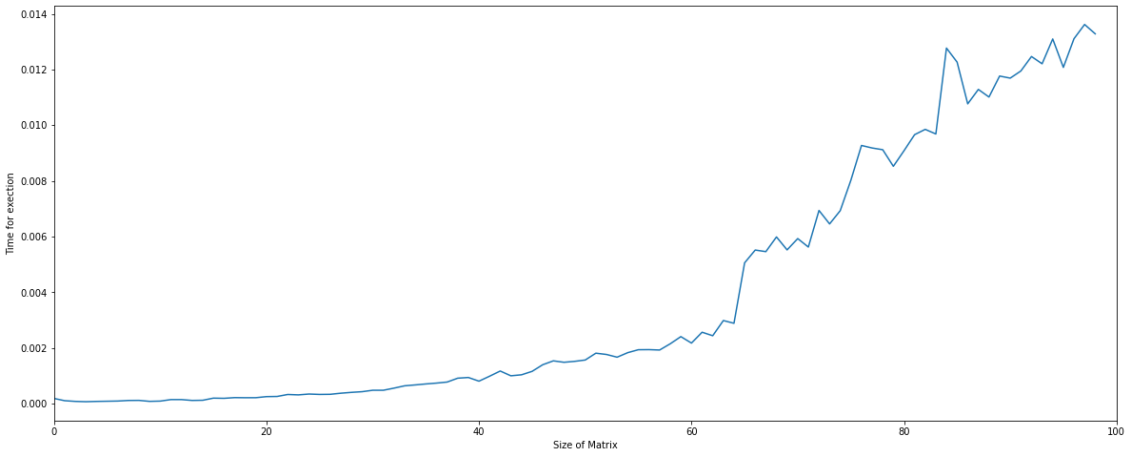
4. SVD Decomposition

```
Time_list_numpy=[]
for i in range(1,1000,1):
  start=time.time()
  temp= np.random.rand(i,i)
  #start=time.time()
  u, s, vh = np.linalg.svd(temp, full_matrices=True, compute_uv=True, hermitian=False)
  #print('Lambda: ',Lambda)
  #print('U: ', U)
  end=time.time()
  Time_list_numpy.append(end-start)
```

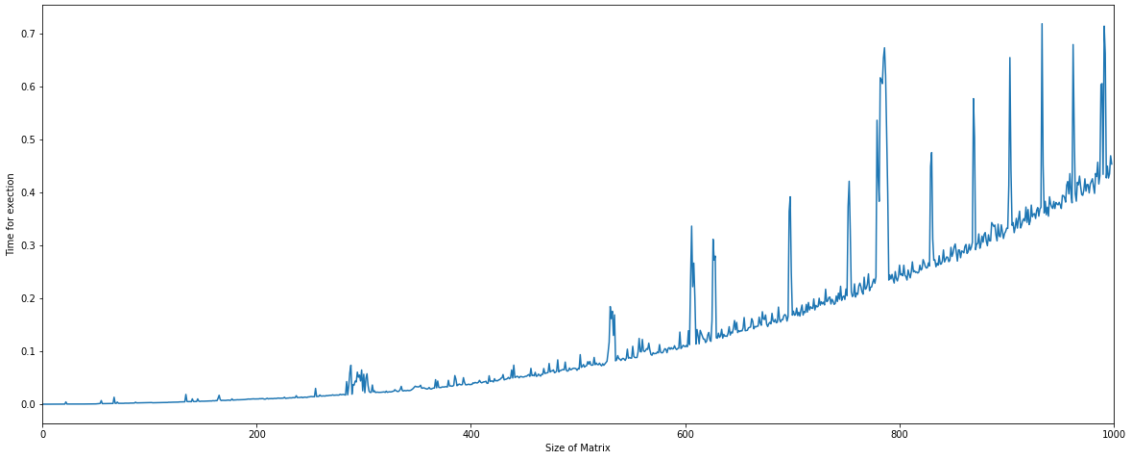
3.4 Results:

- After performing each operation using traditional approach we get the following result

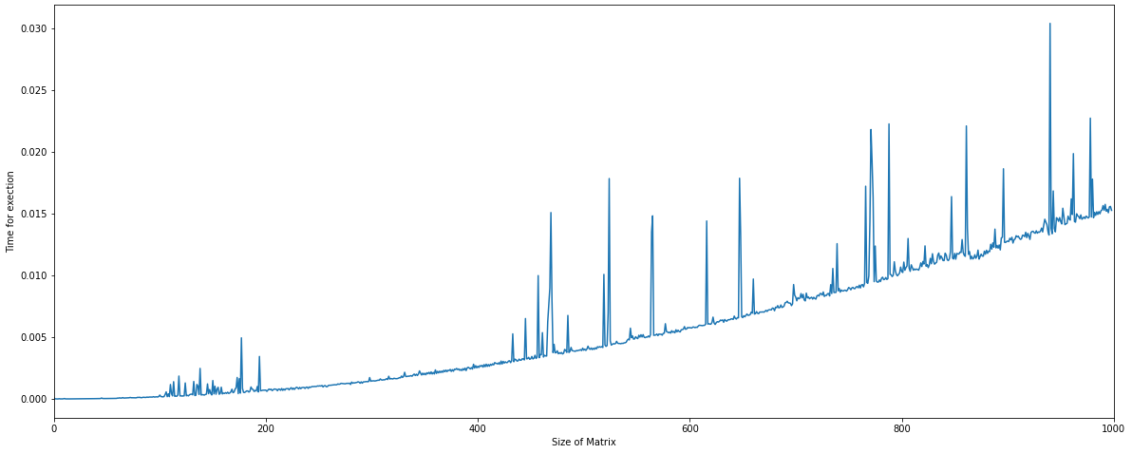
1. Eigenvalue



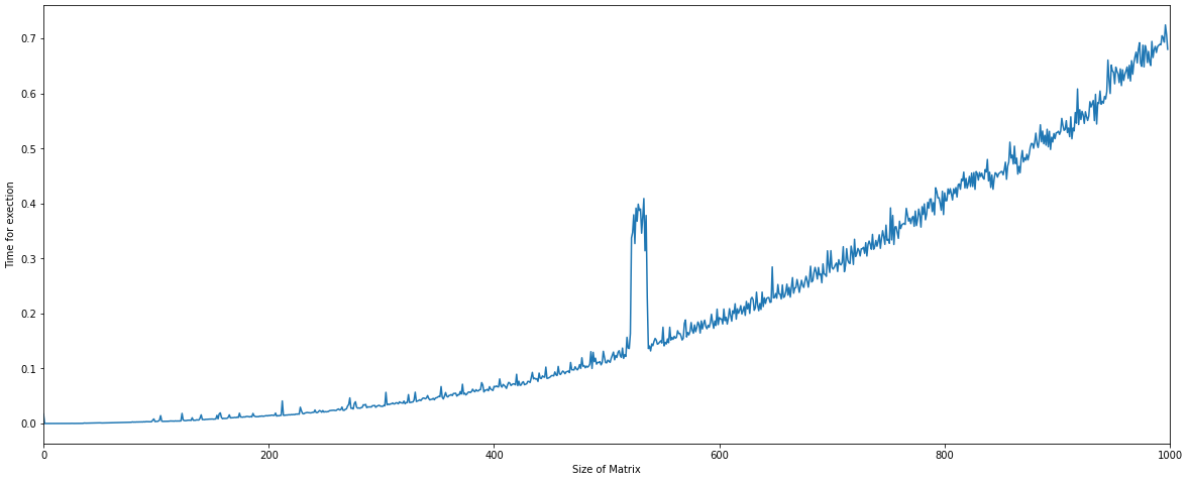
2. Least Square



3. Norm



4. SVD decomposition



- Numpy supports large scale matrix operations. However, cost of computing for numpy library increases exponentially as the matrix size increases. This reduces the operational scalability of the system.

3.5 SOTA Implementation:

PyTorch and TensorFlow are the SOTA approaches for this problem because:

- They have better operational speed then most of their competitors
- They perform even better when hardware acceleration is used (GPUs and TPUs)

1. **Least Square:**

   ▼ PyTorch:

```
p=[]
for i in range(1,1000,1):
  p.append(i)

Time_list_py=[]

for k in range(1,1000,1):
  b=torch.rand(k,k)
  start=time.time()
  y=torch.rand(k)
  sol,res,rank,val = torch.linalg.lstsq(b.T,y)
  end = time.time()
  Time_list_py.append(end-start)
```

   ▼ TensorFlow:

```
Time_list_tf=[]
for i in range(1,1000,1):
  A=torch.rand(i,i)
  y=torch.rand(i)
  start=time.time()
  sol,res,rank,val = torch.linalg.lstsq(A.T,y)
  end=time.time()
  Time_list_tf.append(end-start)
```

2. **SVD (Singular Value Decomposition):**

▼ PyTorch:

```
p=[]
for i in range(1,1000,1):
  p.append(i)

Time_list_py=[]

for k in range(1,1000,1):

  start=time.time()
  b=torch.rand(k,k)
  U, S, Vh = torch.linalg.svd(b, full_matrices=False)
  end = time.time()
  Time_list_py.append(end-start)
```

▼ TensorFlow:

```
for i in range(1,1000,1):

  g1 = tf.random.Generator.from_seed(1)
  b=g1.normal(shape=[i,i])
  start=time.time()
  #a,c=tf.linalg.eig(b)
  s, u, v=tf.linalg.svd(b, full_matrices=False, compute_uv=True, name=None)
   #print('Lambda: ',a)
   #print('U: ', c)
  end=time.time()
  Time_list_tf.append(end-start)
```

3. **LU Decomposition (lower–upper):**

▼ PyTorch:

```
p=[]
for i in range(1,100,1):
```

```
  p.append(i)

Time_list_1=[]
for k in range(len(p)):
  start=time.time()
  A=torch.randn(p[k],p[k],p[k])
  B=torch.randn(p[k],p[k],p[k])
  LU, pivots = torch.linalg.lu_factor(A)
  x = torch.lu_solve(B, LU, pivots)
  torch.dist(A @ x, B)
  end = time.time()
  Time_list_1.append(end-start)
```

▼ TensorFlow:

```
p=[]
Time_list_2=[]
for i in range(1,100,1):
  p.append(i)
for i in range(1,100,1):
  start=time.time()
  tensor_10000 = tf.convert_to_tensor(matrix_dummy_list_1[i-1])
  tf.linalg.lu(
  tensor_10000,
  output_idx_type=tf.dtypes.int32,
  name=None)
  end=time.time()
  Time_list_2.append(end-start)
```

4. **Eigenvalue Decomposition:**

▼ PyTorch:

```
for k in range(1,100,1):

  start=time.time()
  b=torch.rand(k,k)
  Lambda,U=torch.linalg.eig(b)
  end = time.time()
  Time_list_py.append(end-start)
```

▼ TensorFlow:

```
for i in range(1,100,1):

  g1 = tf.random.Generator.from_seed(1)
  b=g1.normal(shape=[i,i])
  start=time.time()
  a,c=tf.linalg.eig(b)
   #print('Lambda: ',a)
   #print('U: ', c)
  end=time.time()
  Time_list_tf.append(end-start)
```

5. Norm:

▼ PyTorch:

```
p=[]
for i in range(1,1000,1):
  p.append(i)

Time_list_py=[]

for k in range(1,1000,1):

  start=time.time()
  b=torch.rand(k,k)
  torch.norm(b)
  end = time.time()
  Time_list_py.append(end-start)
```

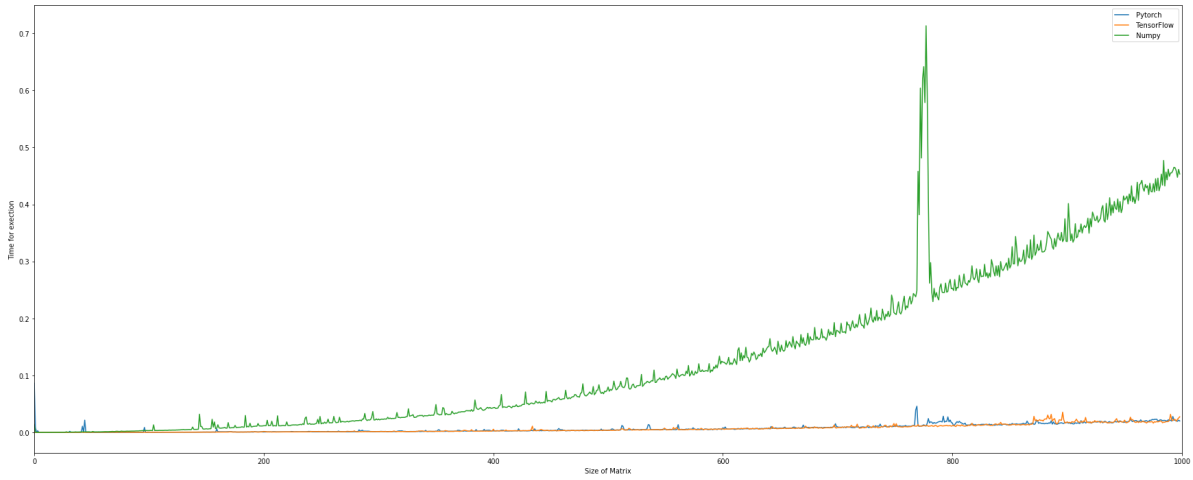
▼ TensorFlow:

```
Time_list_tf=[]

for i in range(1,1000,1):

  g1 = tf.random.Generator.from_seed(1)
  b=g1.normal(shape=[i,i])
  start=time.time()
  #a,c=tf.linalg.eig(b)
  tf.norm(b, ord='euclidean', axis=None, keepdims=None, name=None)
   #print('Lambda: ',a)
   #print('U: ', c)
  end=time.time()
  Time_list_tf.append(end-start)
```

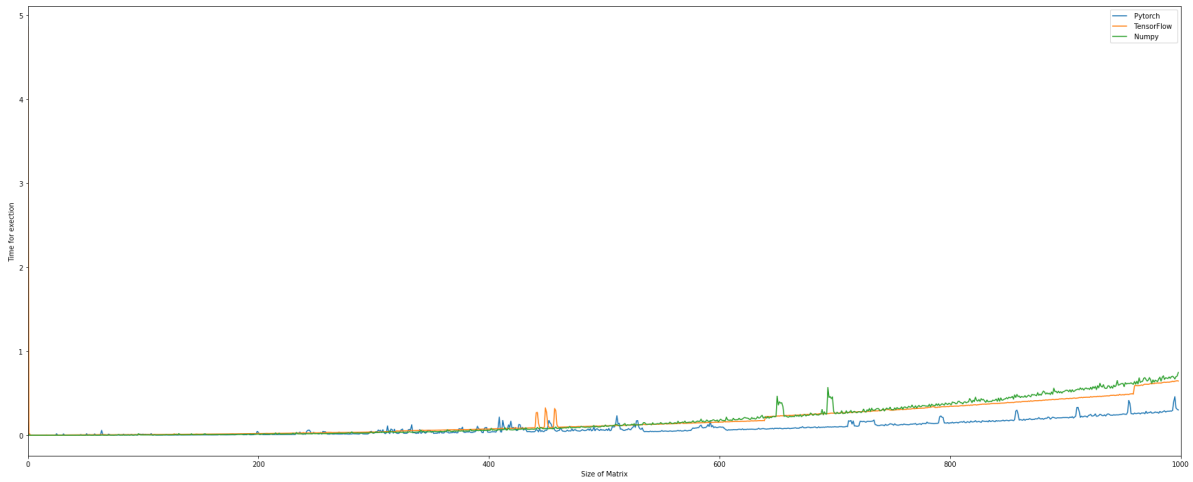
3.6 SOTA Results:

Comparing the three libraries: PyTorch vs. TensorFlow vs. NumPy on the following operations :

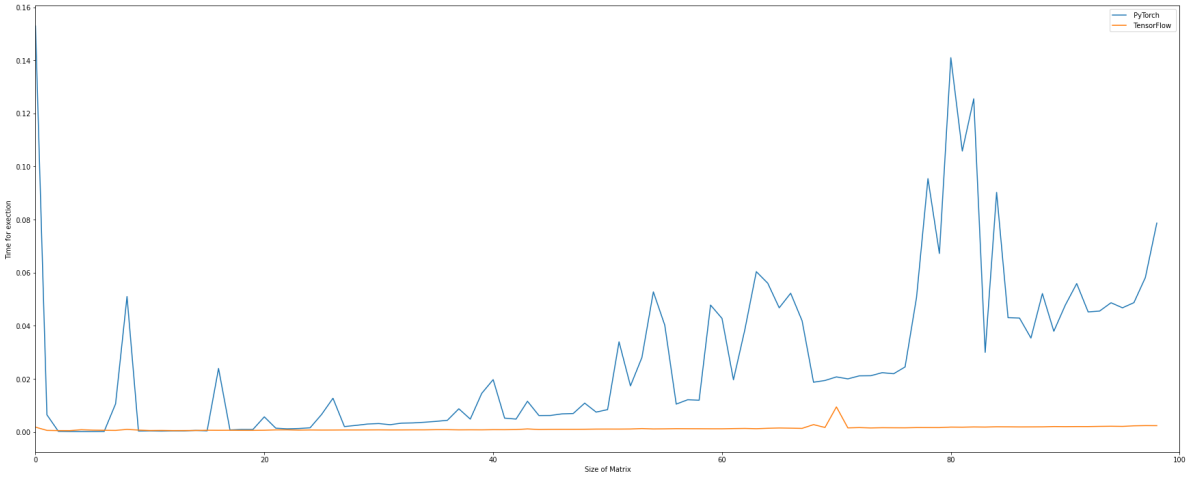
1. **Least Square:**



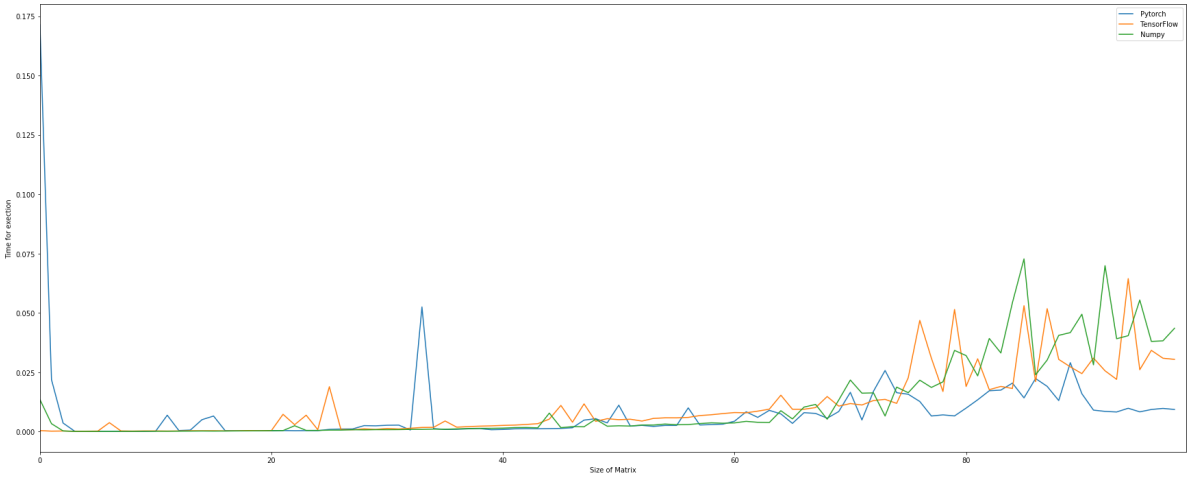
2. **SVD (Singular Value Decomposition):**



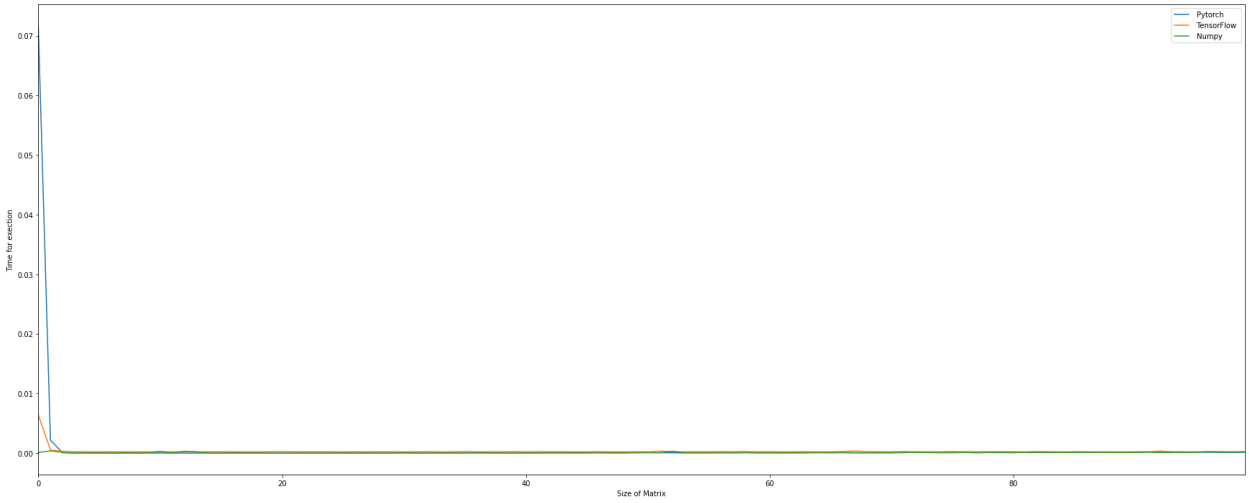
3. **LU Decomposition (lower–upper):**



4. **Eigenvalue Decomposition:**



5. **Norm:**



3.7 Compare and contrast:

Comparing the three libraries: PyTorch vs. TensorFlow vs. Numpy

Operation	PyTorch	TensorFlow	Numpy
Least Square	1 (similar to Tf)	1	3
SVD(Singular Value Decomposition)	1	2	3
LU Decomposition	3	1	-
Eigenvalue Decomposition	1	2	3
Norm	2	1	1

1: Fastest

3: Slowest

4. Conclusion

- This project involved exploring various Linear Algebra operations offered by several libraries.
- We compared various frameworks and libraries that offer linear algebra operations.
  - Numpy
  - PyTorch

    - Tensorflow
- We compared various Linear Algebra operations on varying data size and noted the computational speed offered by these libraries.
- We also explored how hardware acceleration has an effect on the working of each of these libraries.
- We ran our code on Google Colab and used the GPU,TPU Hardware acceleration.
- Finally, we visualized all the outputs that we received for better understanding of the results.

---

5. Acknowledgements

We are extremely thankful to Prof. Tsui-Wei (Lily) Weng, Halıcıoğlu Data Science Institute, San Diego for her continuous support, and encouragement to help us throughout this project. We would also like to thank our teaching assistants - Manoj and Ali, and our feedback rehearsal team- group 7 for their feedback and guidance toward the completion of our project.

---

6. References

- John T Foster(2019) Matlab-vs-python -A rebuttal. https://johnfoster.pge.utexas.edu/blog/posts/matlab-vs-python/
- Bryan Weber matlab-vs-python: Why and How to make the switch https://realpython.com/matlab-vs-python/
- Fei-Fei Li,Justin Johnson,Serena Yung (2019) Hardware and software for convolutional neural networks http://cs231n.stanford.edu/slides/2019/cs231n_2019_lecture06.pdf
- Numpy Developers (2018) Numpy for Linear Algebra networks https://numpy.org/doc/stable/reference/routines.linalg.html

- The PyTorch Foundation(2022) Linear Algebra Operations https://pytorch.org/docs/stable/linalg.html
- Google Developers Network(2019) TensorFlow Linear Algebra Operations https://www.tensorflow.org/api_docs/python/tf/linalg
- Numpy Linear Algebra Library Documentation- Numpy.lin.alg

  https://numpy.org/doc/stable/reference/routines.linalg.html
- TensorFlow LU Decomposition Function

  https://www.tensorflow.org/api_docs/python/tf/linalg/lu
- PyTorch Common Linear Algebra Operations

  https://pytorch.org/docs/stable/linalg.html