BDAPro
@DIMA

Benchmarking MXNet and TensorFlow

Presented by:

- **Matteo Paltenghi**
- **Kaoutar Chennaf**

Mentor: **Behrouz Derakhshan**

Agenda

BDAPro
@DIMA

1. Recap of Previous Presentation
2. Hardware Setting
3. Benchmark A – Fundamental Operations (CPU vs **GPU**)
4. Benchmark B – End-to-end Training LeNet (CPU vs **GPU**)
5. **Optimizations Available**
6. **Benchmark C - Test of TensorFlow optimization on CIFAR10 dataset**
7. **Benchmark D – Optimizations on LeNet (CPU vs GPU)**
8. Summary
9. Future Work

1. Recap

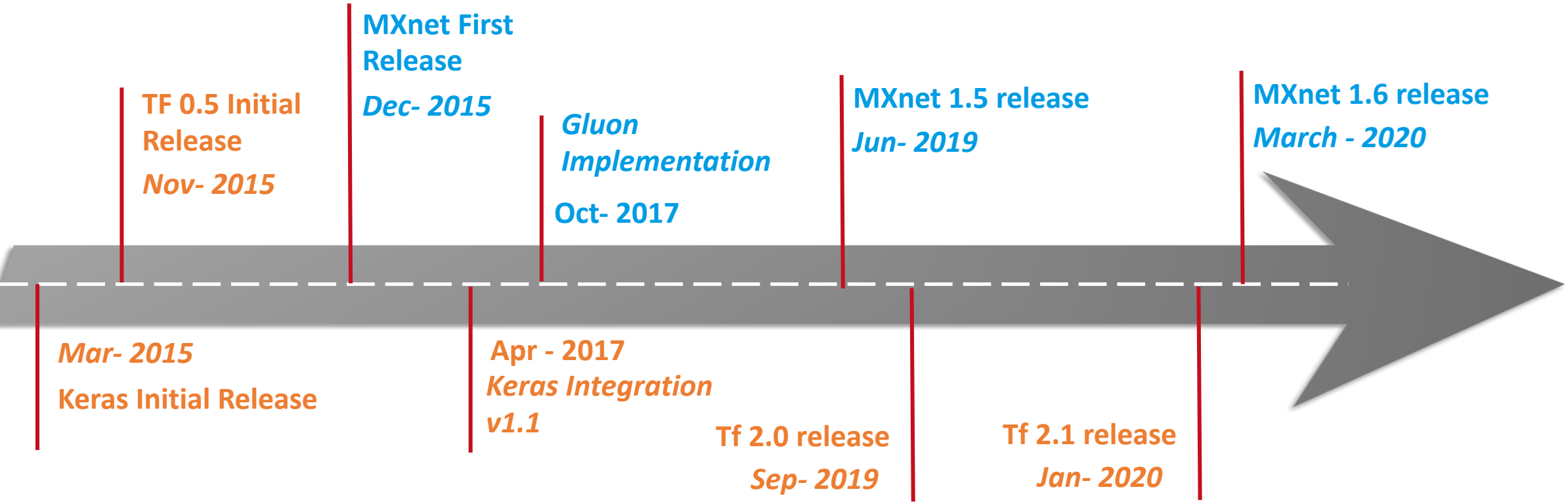
Performance Comparison between Deep Learning Libraries

BDAPro
@DIMA

System	Core Lang	Bindings Langs	Devices (beyond CPU)
TensorFlow	C++	Python	GPU/**TPU**/Mobile
MXNet	C++	Python/R/Julia/Go	GPU/Mobile

MXNet and TensorFlow Timeline

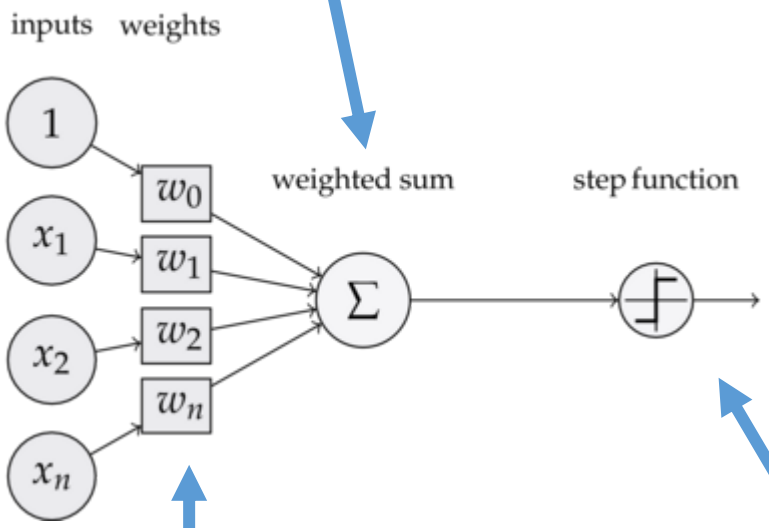
BDAPro
@DIMA



Benchmark A - Fundamental Operations

BDAPro
@DIMA

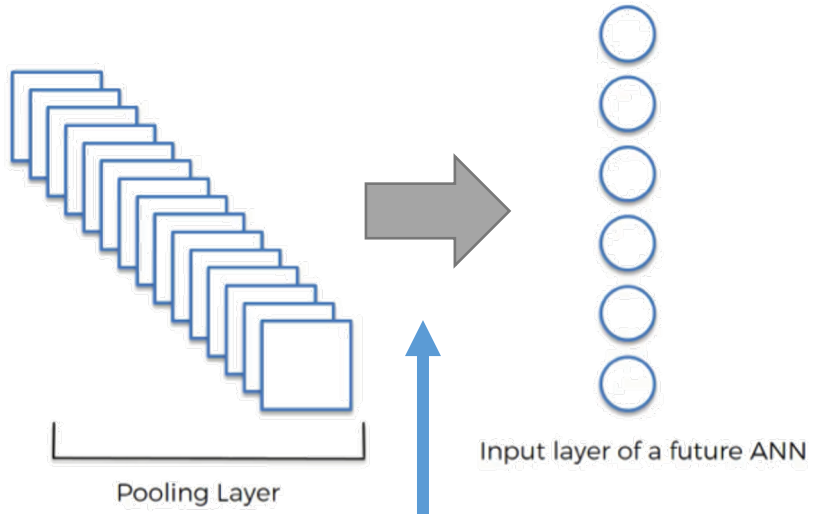
2. Dot product | 3. Matrix multiplication



1. Vector creation

4. Activation function

6. Vector Normalization



5. Flattening

Image source: https://blog.dbrgn.ch/2013/3/26/perceptrons-in-python/

Image source: https://www.superdatascience.com/blogs/convolutional-neural-networks-cnn-step-3-flattening

Benchmark B - LeNet Architecture

BDAPro
@DIMA

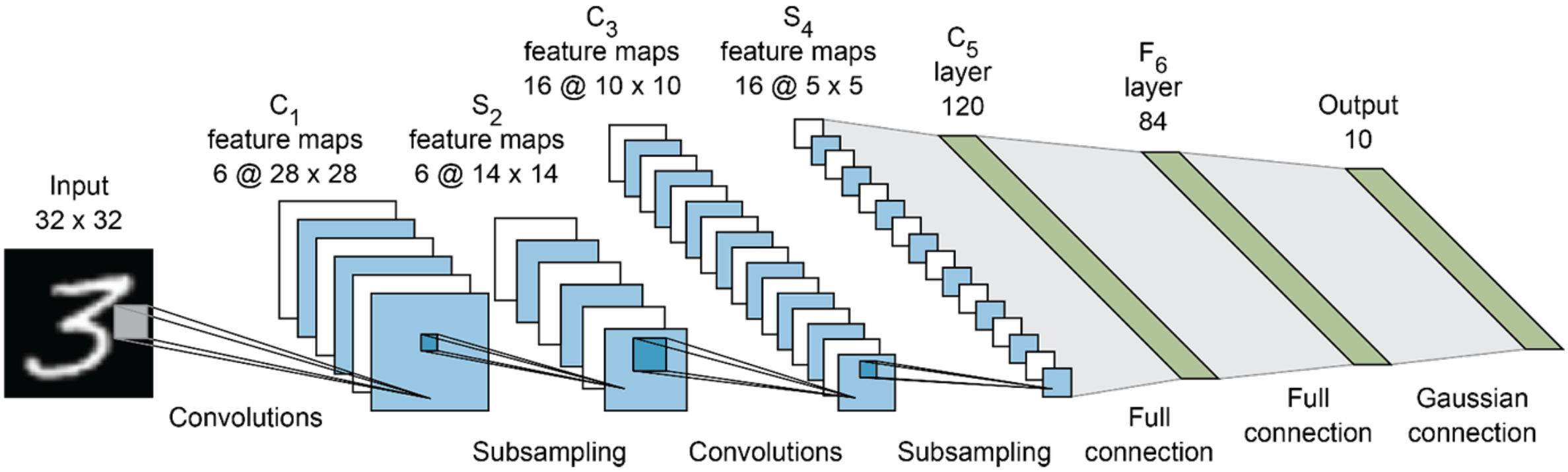


Image source: https://link.springer.com/article/10.1007/s13244-018-0639-9

Imperative

```
# Imperative
import numpy as np

a = np.ones(10)
b = np.ones(10) * 2
c = b * a
d = c + 1
```

(+) Tend to be More Flexible

(+) Easy to try New Ideas (for Researches)

(+) Easy to specify Arbitrary Control Flow

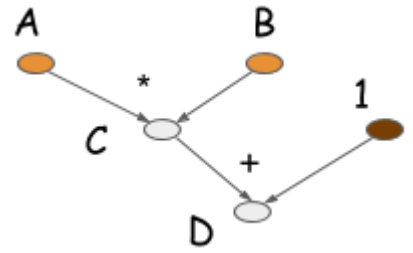
(-) Debugging happens during execution

(-) It can be More Difficult to Reuse

(-) Difficult to inspect, copy, or clone

Symbolic

```
# Symbolic
A = Variable('A')
B = Variable('B')
C = B * A
D = C + Constant(1)
# compiles the function
f = compile(D)
d = f(A=np.ones(10), B=np.ones(10)*2)
```



(+) Tend to be More Efficient

(+) Easy to inspect

(+) Run Extensive checks before Execution

(+) Easier to copy or clone

(+) Fit Most use cases

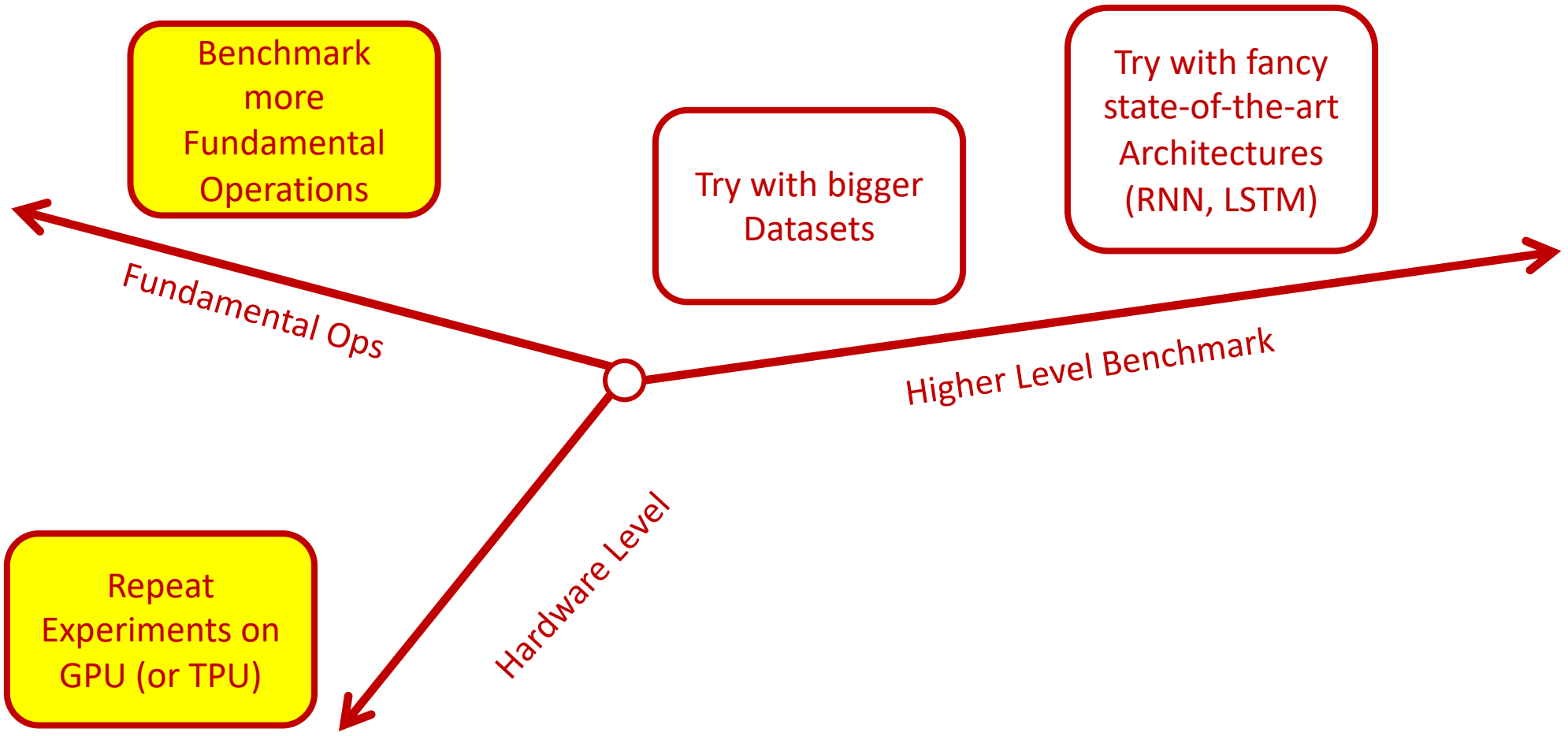
(-) Few Special use cases do not fit

Previous Presentation: Summary

BDAPro
@DIMA

- Two architectures are very similar in terms of high level features
- Same **mixed paradigm approach**: declarative (more efficient) and imperative (more intuitive)
- Very different in terms of performance speed (on **CPU**) both for fundamental operations and complete training
- Two huge documentations: TF bigger but not well organized, MXNet smaller but in constant expansion with high quality resources.
- **TensorFlow is considerably slower on CPU**

Future Work – Possible Directions

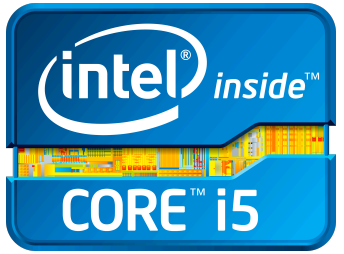


2. Hardware Setting

Previously - Hardware Setting on CPU

BDAPro
@DIMA

12 GB
RAM
TOTAL



New Hardware Setting on GPU

BDAPro
@DIMA

AMD

62 GB RAM

NVIDIA GTX 980

(Late 2014)

Latest version of **CUDA (Compute Unified Device Architecture)** is a parallel computing platform and application programming interface (API) model created by Nvidia.

```
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 440.33.01    Driver Version: 440.33.01    CUDA Version: 10.2     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|===============================+======================+======================|
|   0  GeForce GTX 980     Off  | 00000000:03:00.0 Off |                  N/A |
| 36%   33C    P2    41W / 180W |    578MiB /  4043MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+
```

```
mattepalte@gpu-3:~$ free -h
              total        used        free      shared  buff/cache   available
Mem:            62G        877M         42G        147M         19G         61G
Swap:          3,7G          0B        3,7G
```

How to enable GPU usage

BDAPro
@DIMA

1. Instal Nvidia Grapich Card & Drivers
2. Install compatible CUDA drivers
3. Install compatible cuDNN drivers
4. Check your newly installed drivers with “nvidia-smi” command

```
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 440.33.01    Driver Version: 440.33.01    CUDA Version: 10.2     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|===============================+======================+======================|
|   0  GeForce GTX 980     Off  | 00000000:03:00.0 Off |                  N/A |
| 36%   33C    P2    41W / 180W |    578MiB /  4043MiB |      0%      Default |
+-------------------------------+----------------------+----------------------+
```

TensorFlow

- Automatically used
- Just have to check that your TensorFlow can detect the GPUs

```
tf.config.experimental.list_physical_devices('GPU')
```

MXNet

- Set context

```
# MXNET
gpus = mx.test_utils.list_gpus()
ctx =  [mx.gpu()] if gpus else [mx.cpu(0), mx.cpu(1)]
print(ctx)
```

3. Benchmark – Fundamental Operations (CPU vs GPU)

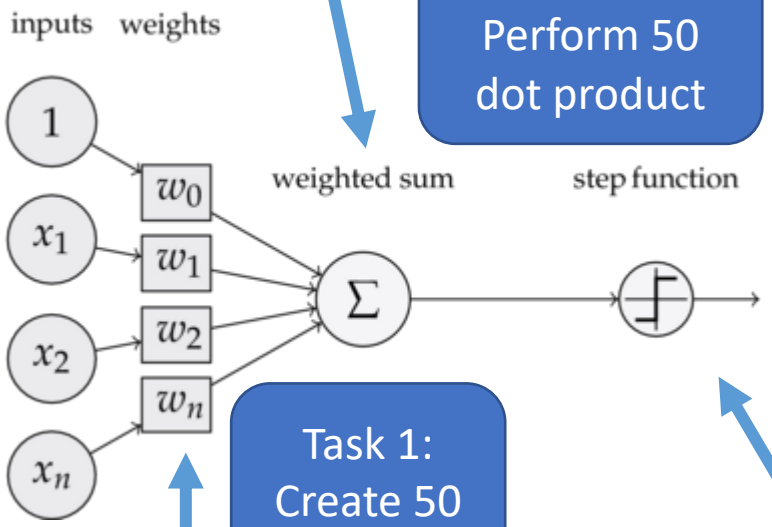
Fundamental Operations

BDAPro @DIMA

2. Dot product | 3. Matrix multiplication

Task 2: Perform 50 dot product

Task 3: Perform 10 matrix multiplication



Task 1: Create 50 vectors

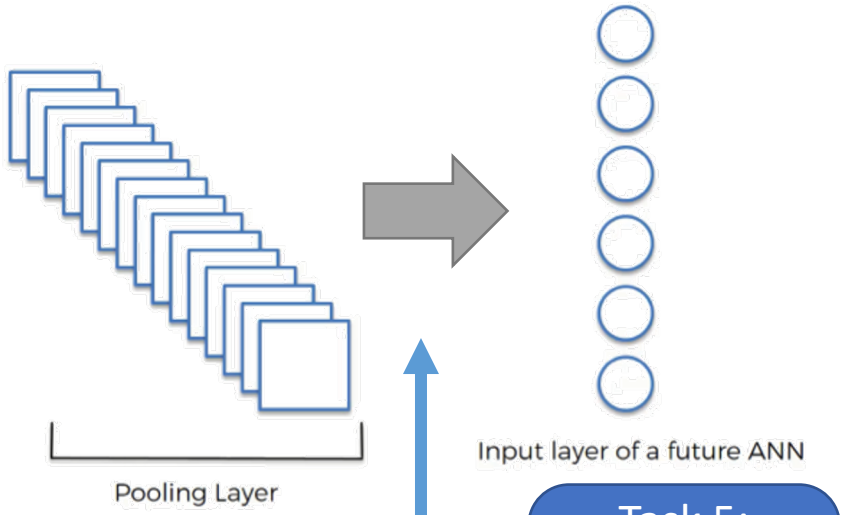
Task 4: Execute the activation function on 50 vectors

1. Vector creation

4. Activation function

Task 6: Normalize 50 vectors

6. Vector Normalization



Task 5: Flatten 50 matrices to vectors

5. Flattening

Image source: https://blog.dbrgn.ch/2013/3/26/perceptrons-in-python/

Image source: https://www.superdatascience.com/blogs/convolutional-neural-networks-cnn-step-3-flattening

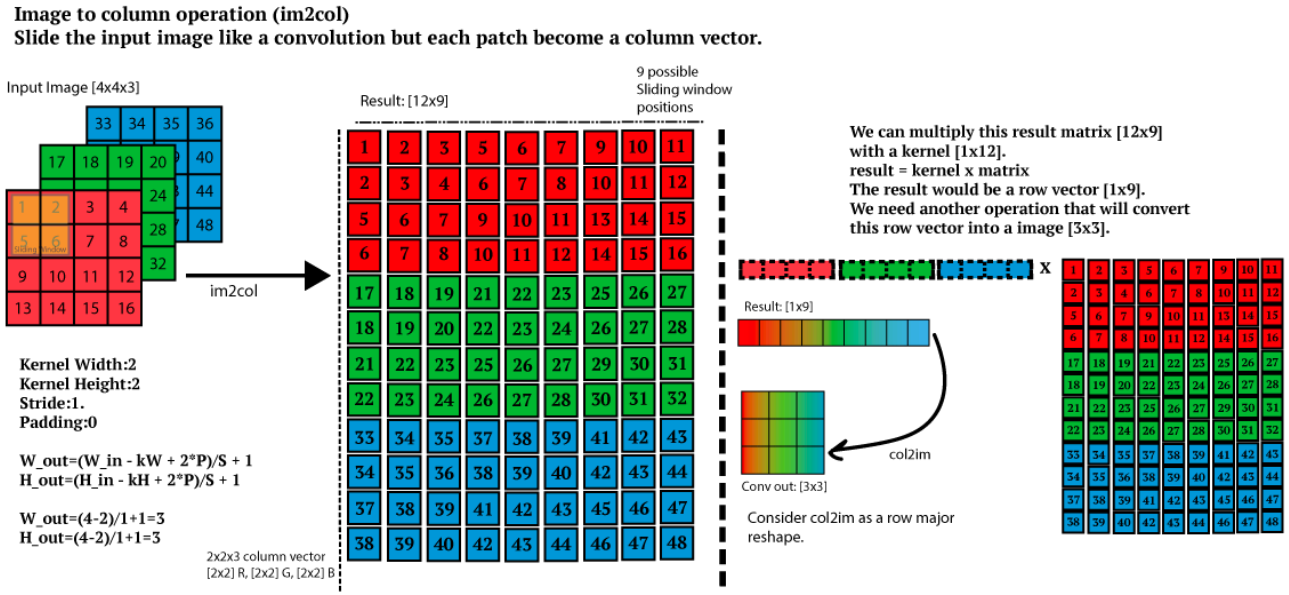
Fundamental Operations

BDAPro
@DIMA

Operation	MXNet	TensorFlow
1. Vector creation	mx.ndarray.from_numpy(np_vector)	tf.convert_to_tensor(np_vector)
2. Dot Product	mx.ndarray.dot(first, second)	tf.tensordot(first, tf.transpose(second), axes=1)
3. Matrix Multiplication	mx.nd.linalg.gemm2(first, second)	tf.tensordot(first, second, axes = [[1], [0]])
4. Activation Function	mx.ndarray.Activation(input_vector, act_type = ”sigmoid”)	tf.keras.activations.sigmoid(input_vector)
5. Flattening	matrix.reshape(shape=(-1,))	tf.reshape(matrix, [-1])
6. Normalization	matrix/mx.nd.norm(matrix, ord=2, axis=None)	tf.linalg.normalize(matrix, ord=2, axis=None)[0]

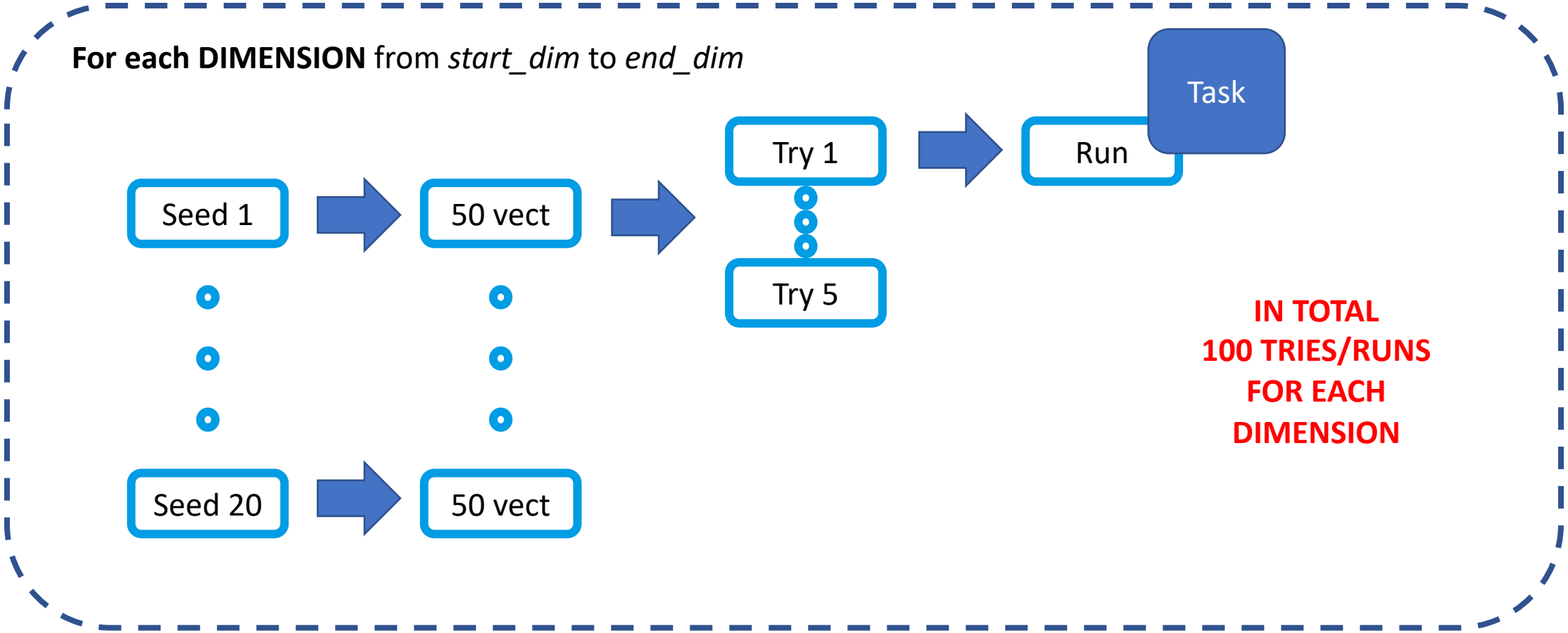
Benchmark Convolution Operation?

- A primitive operation was not present in none of the two frameworks.
- When using GPUs, cuDNN library is reconducting it to a GEMM (General Matrix Multiplication) with the algorithm im2col.
- Source: https://leonardoaraujosantos.gitbooks.io/artificial-inteligence/content/making_faster.html

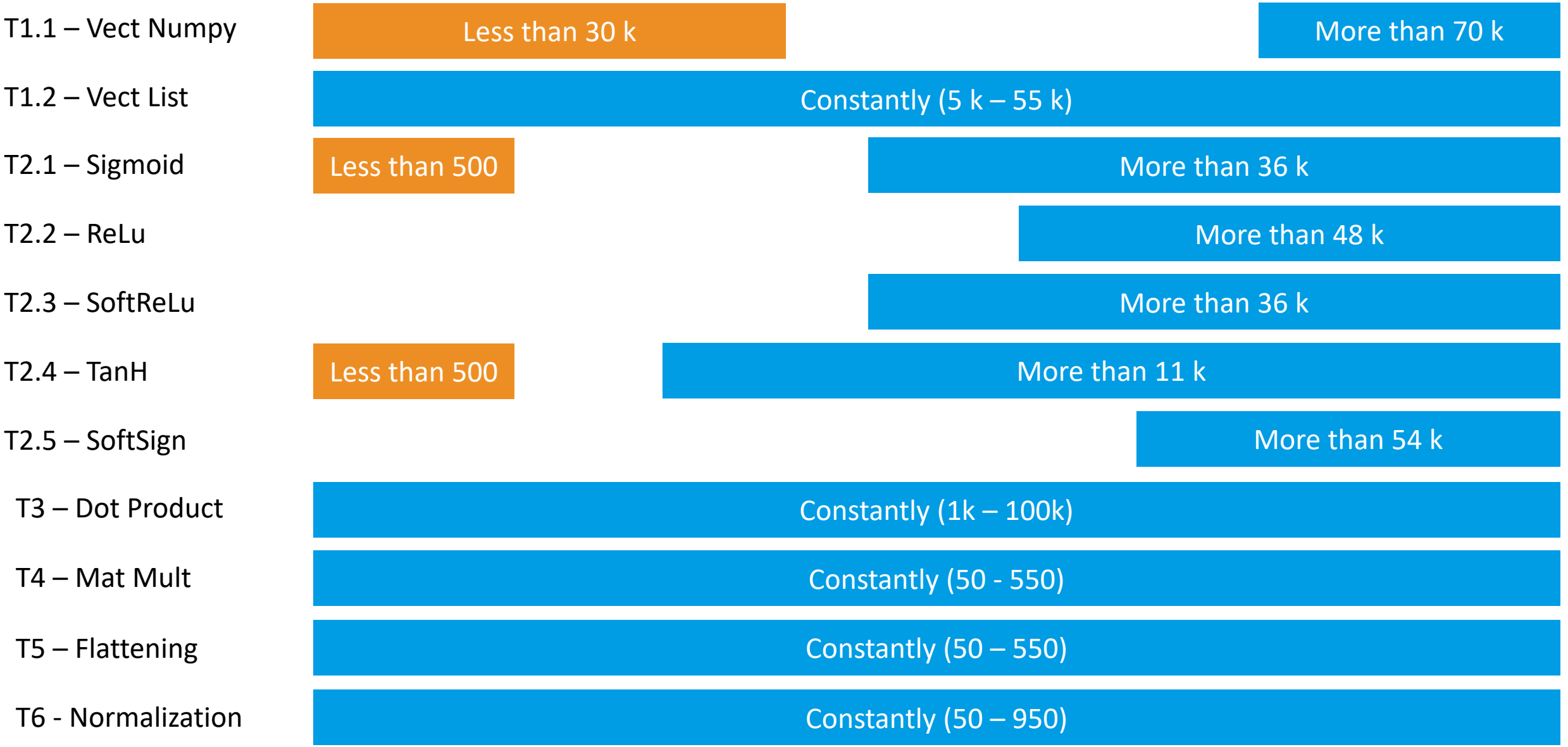


Experiment Description

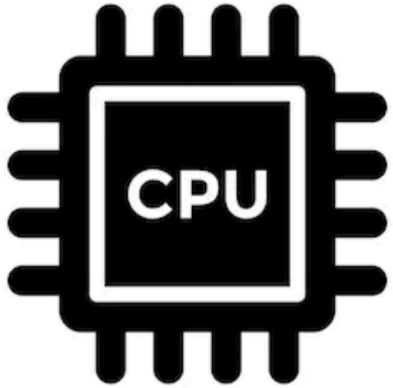
BDAPro
@DIMA

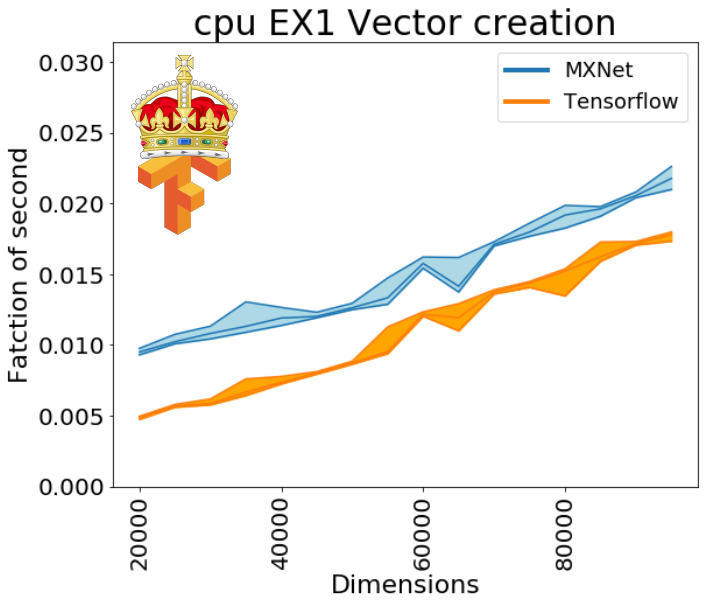


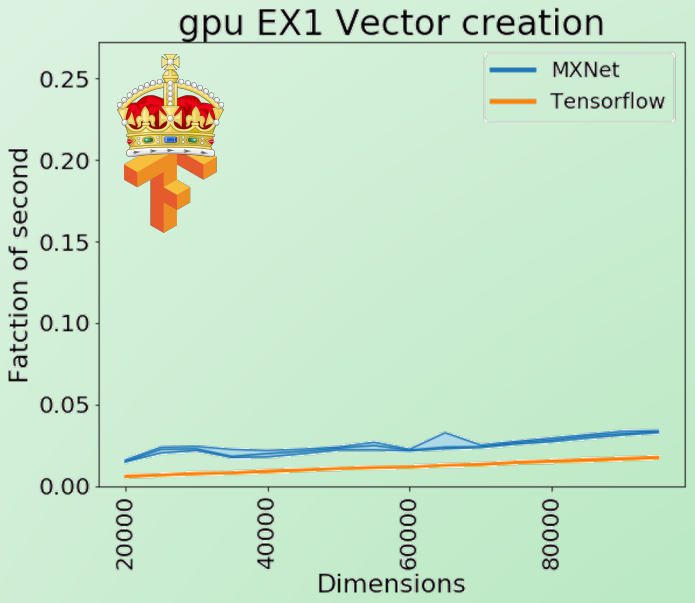
Benchmark Summary – Fundamental Operations on CPU – 12 GB – i5

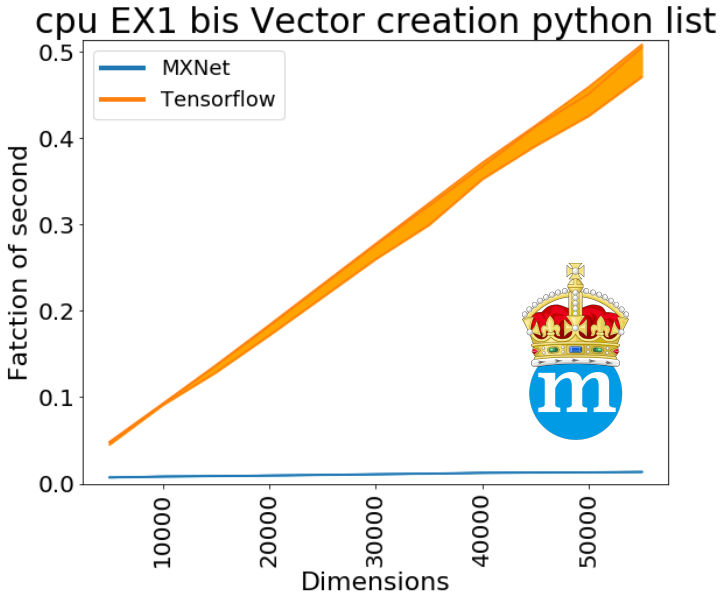


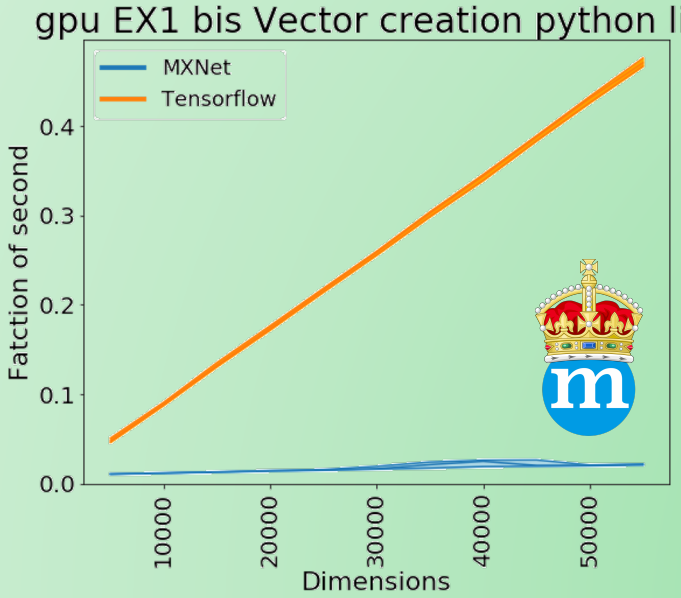
Vector creation











Tasks 2 – Activation Functions

BDAPro
@DIMA

Task 4: Execute the activation function on 50 vectors

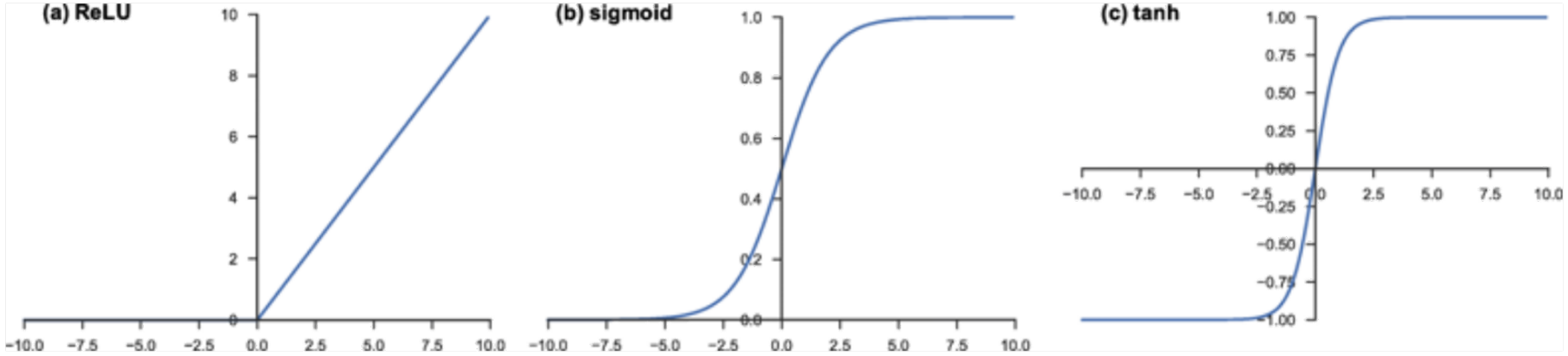
Task 4.1: Sigmoid

Task 4.2: ReLu

Task 4.3: SoftRelu

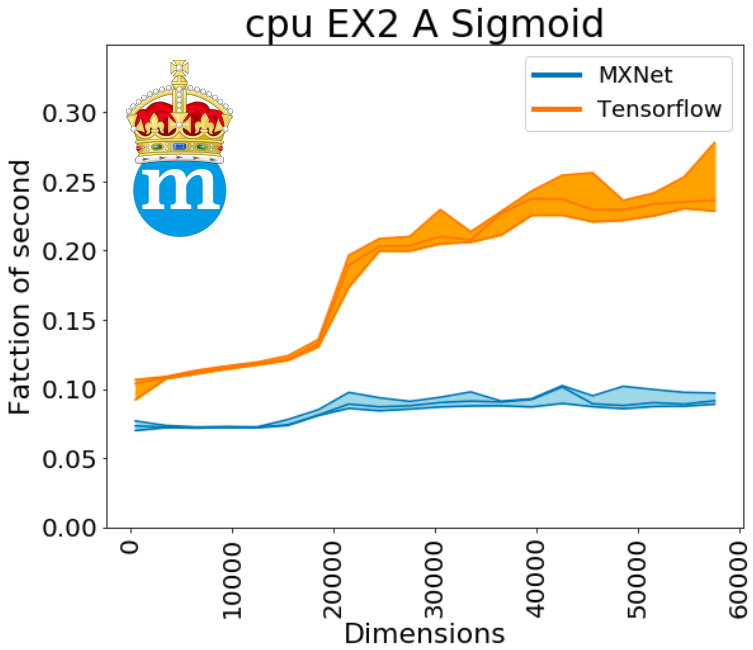
Task 4.4: TanH

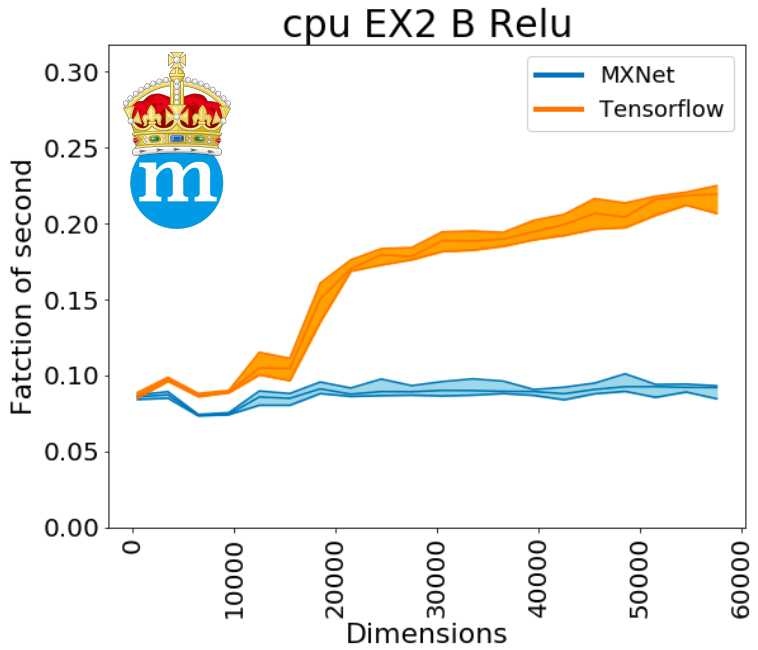
Task 4.5: softsign

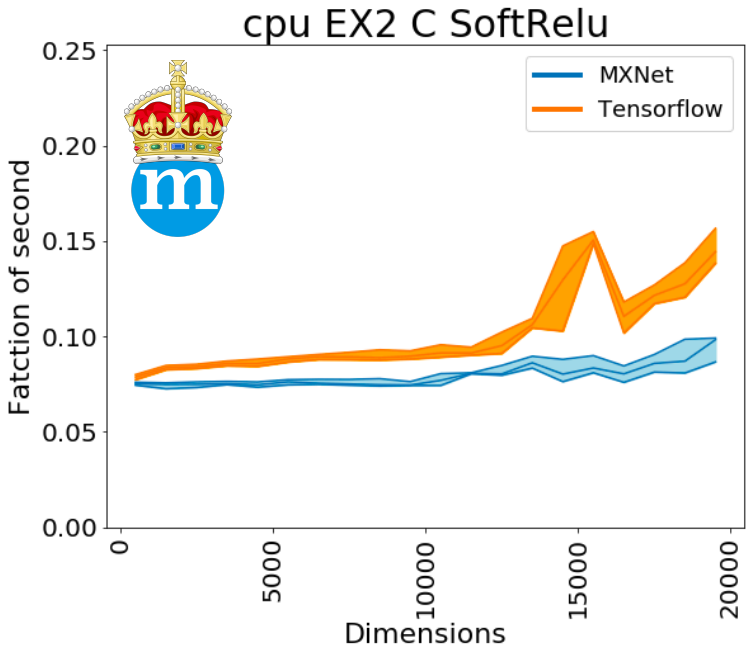


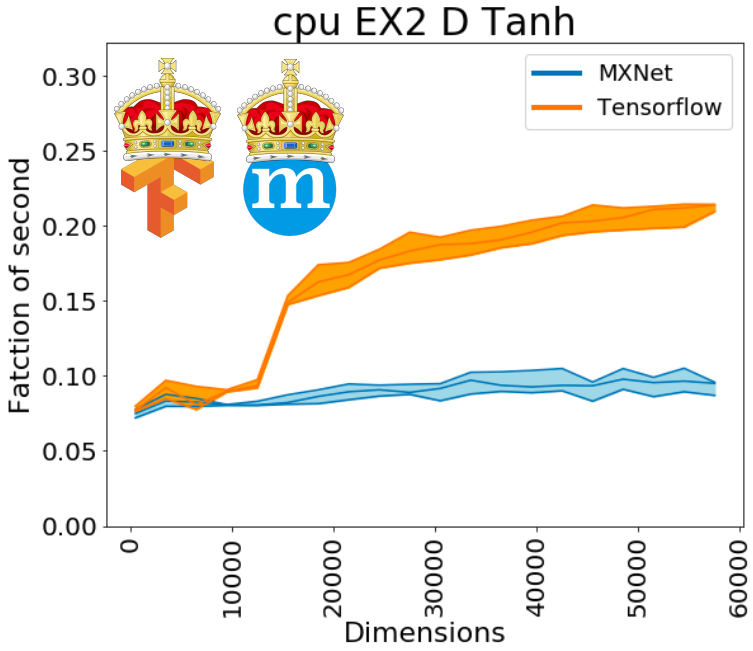
Activation functions commonly applied to neural networks: **a** rectified linear unit (ReLU), **b** sigmoid, and **c** hyperbolic tangent (tanh)

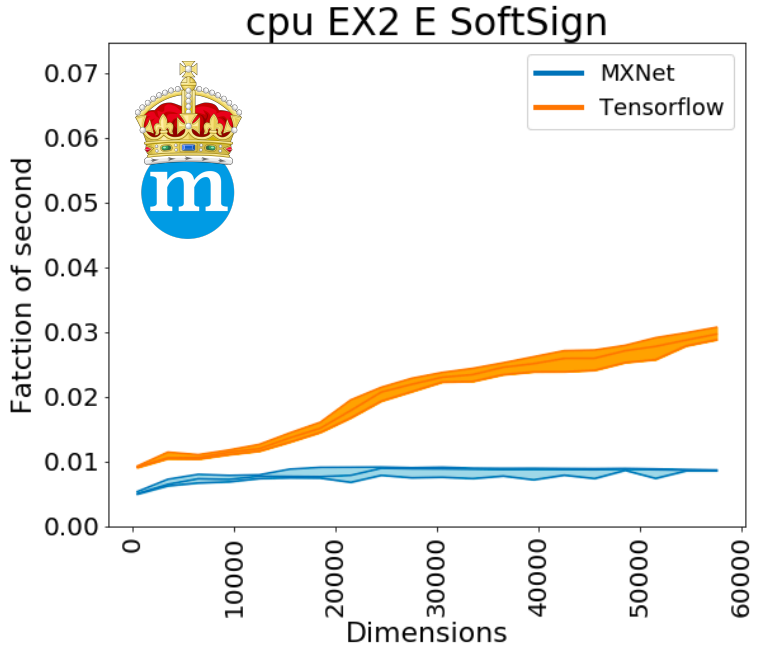
Activation Functions - CPU

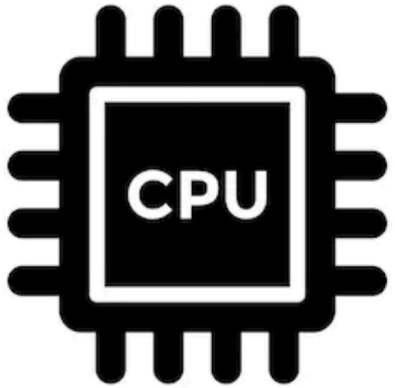




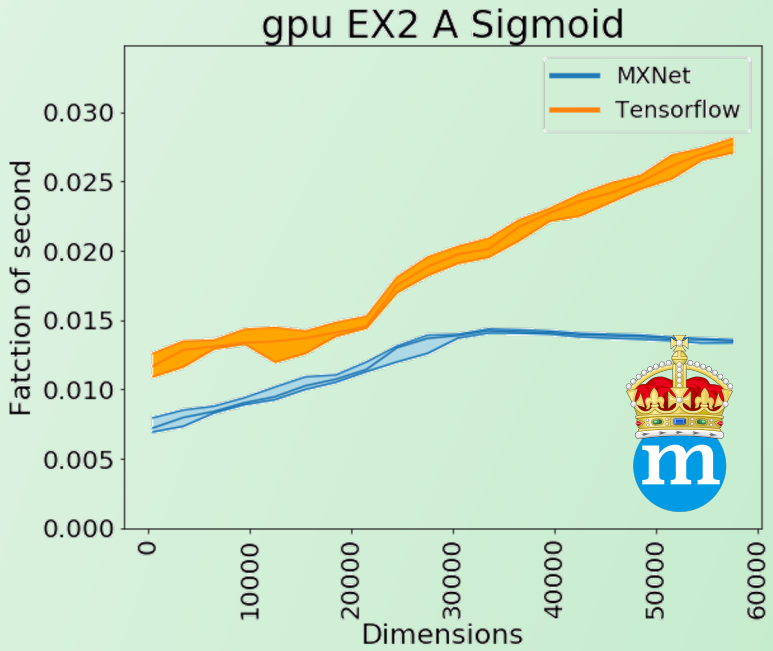


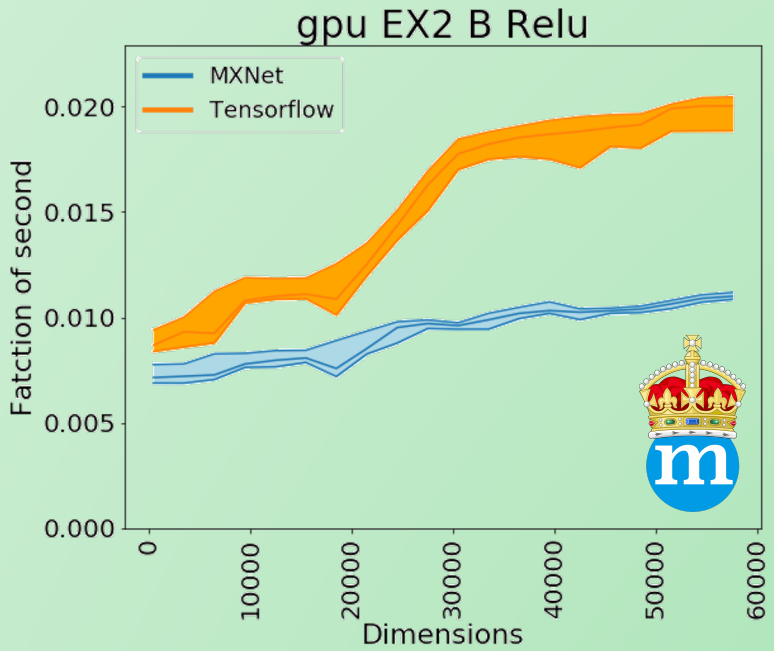


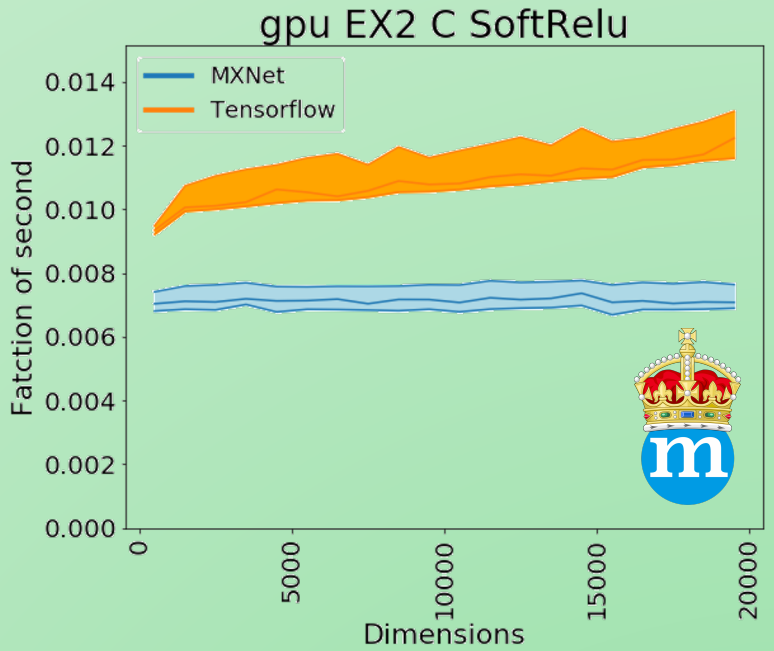


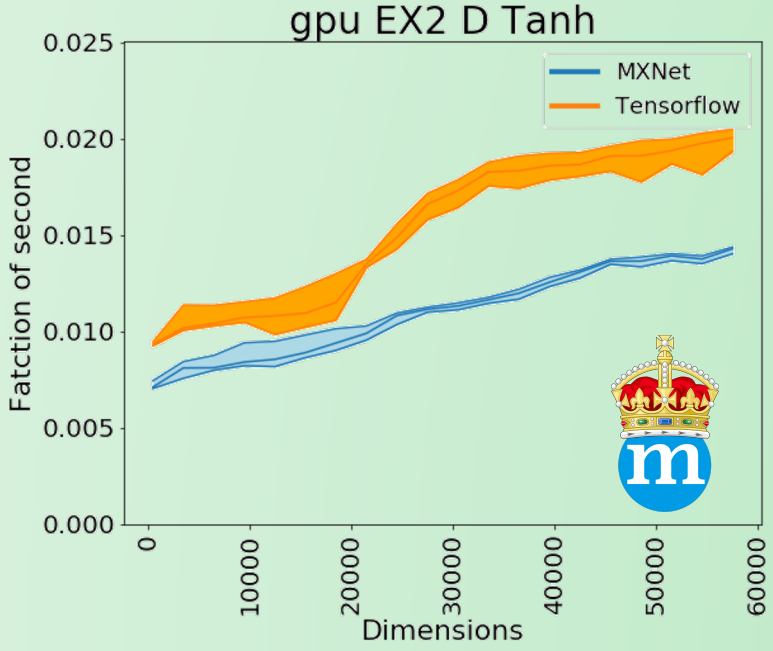


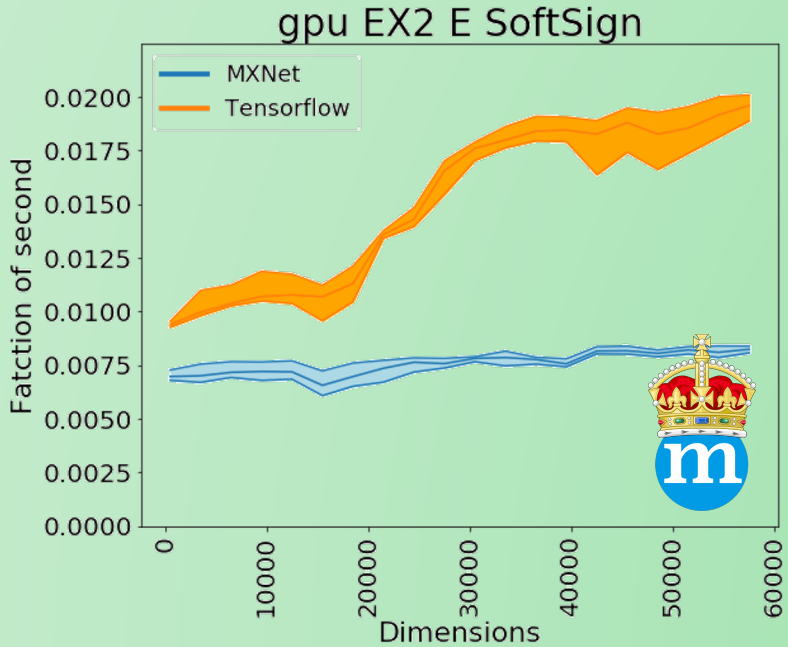
Activation Functions - GPU



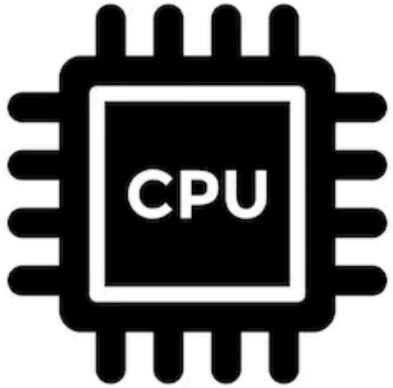


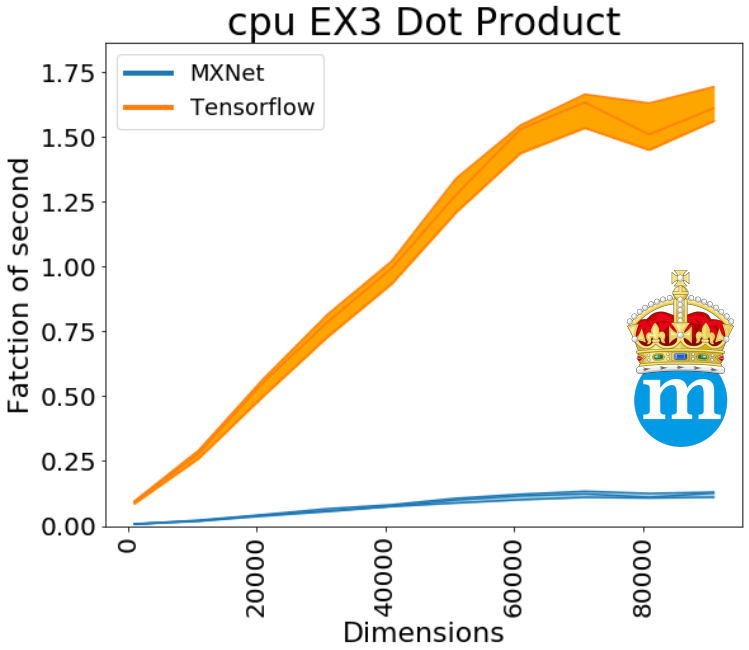


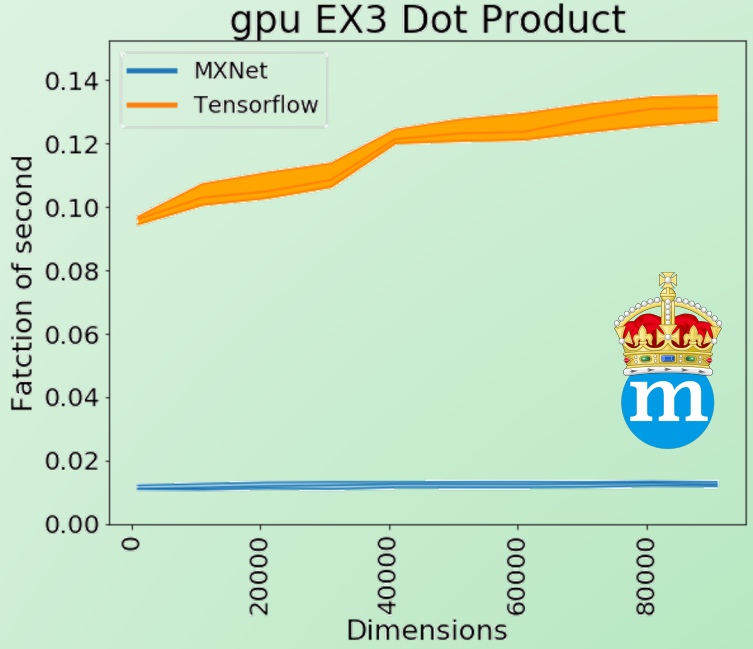




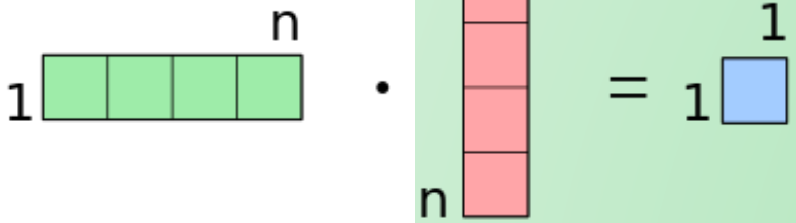
Dot Product


CPU

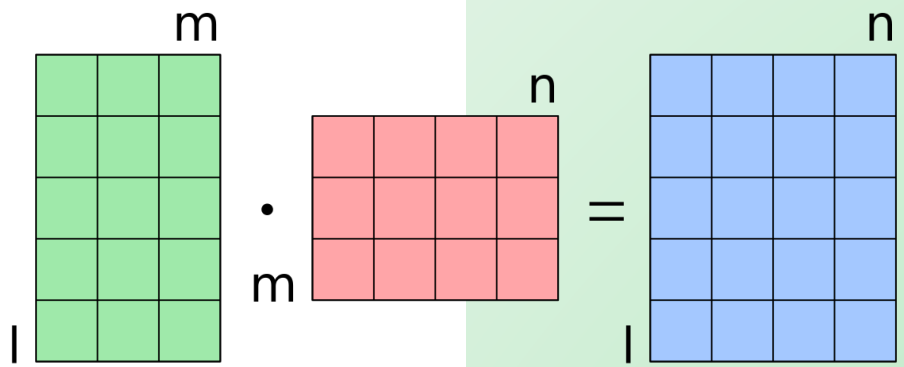

cpu EX3 Dot Product
MXNet
Tensorflow
Fatction of second
1.75
1.50
1.25
1.00
0.75
0.50
0.25
0.00
0
20000
40000
60000
80000
Dimensions


gpu EX3 Dot Product
MXNet
Tensorflow
Fatction of second
0.14
0.12
0.10
0.08
0.06
0.04
0.02
0.00
0
20000
40000
60000
80000
Dimensions

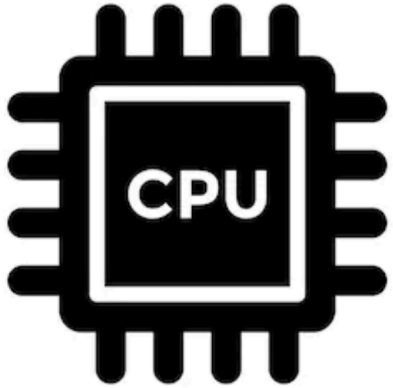

NVIDIA

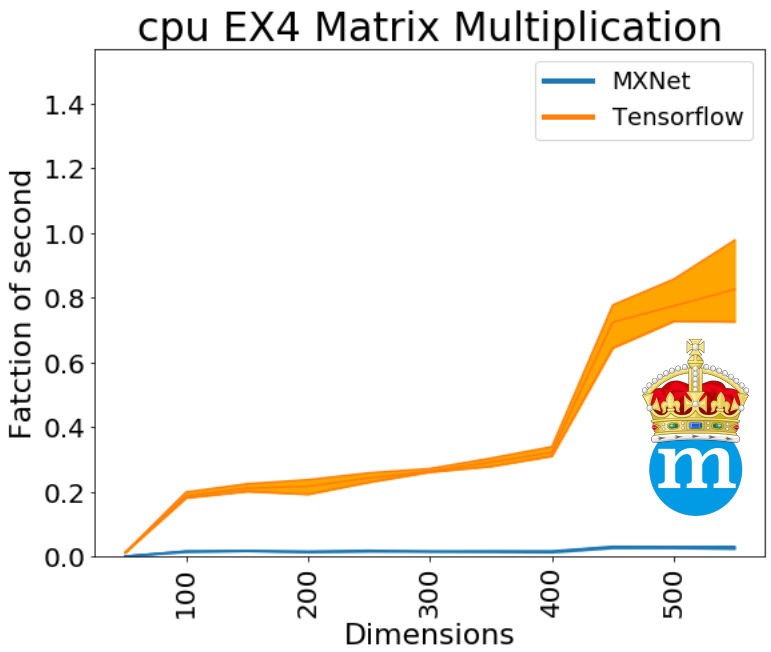

n
1
•
1
n
=
1
1

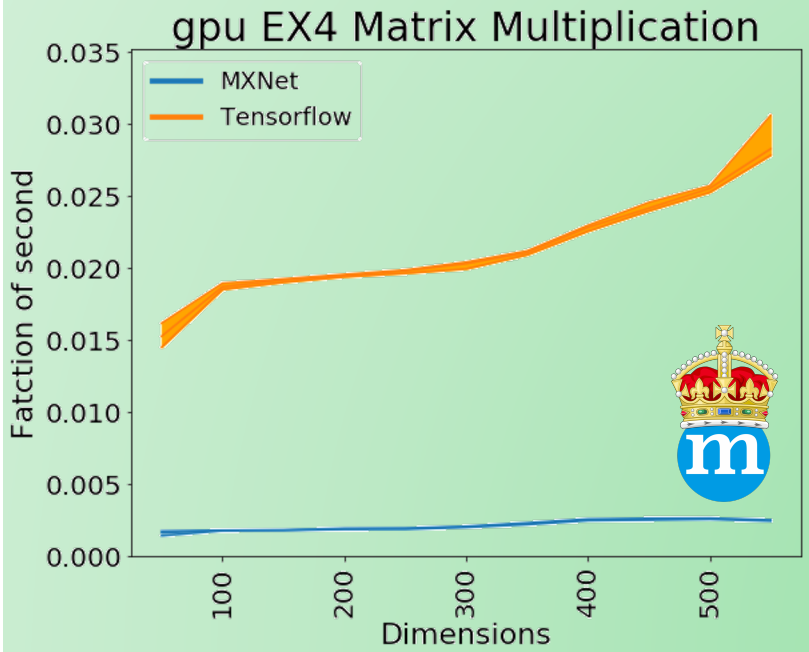
Matrix Multiplication



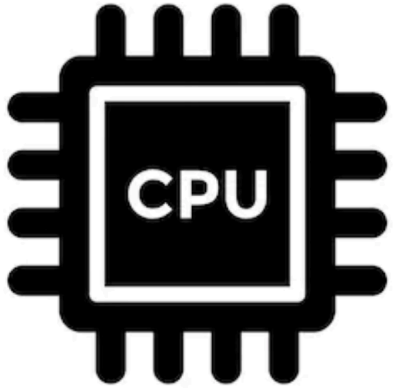
A · B = C

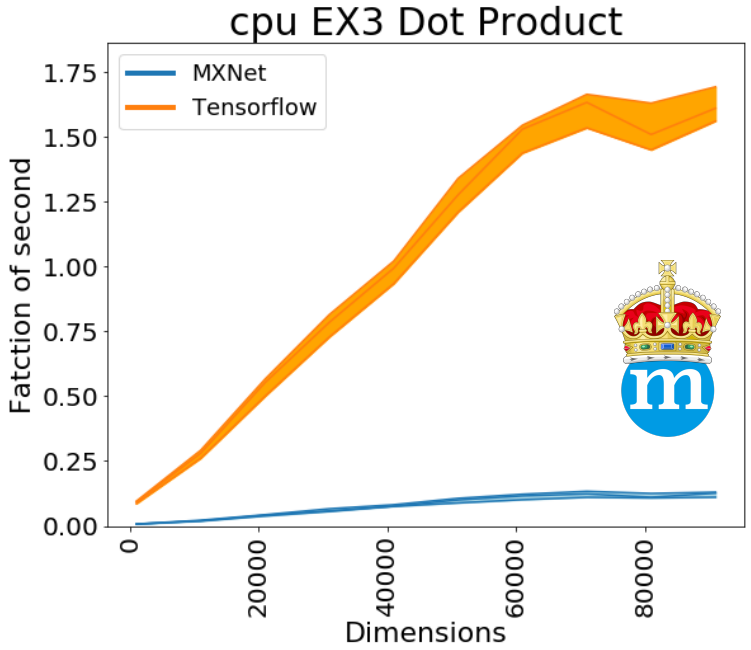


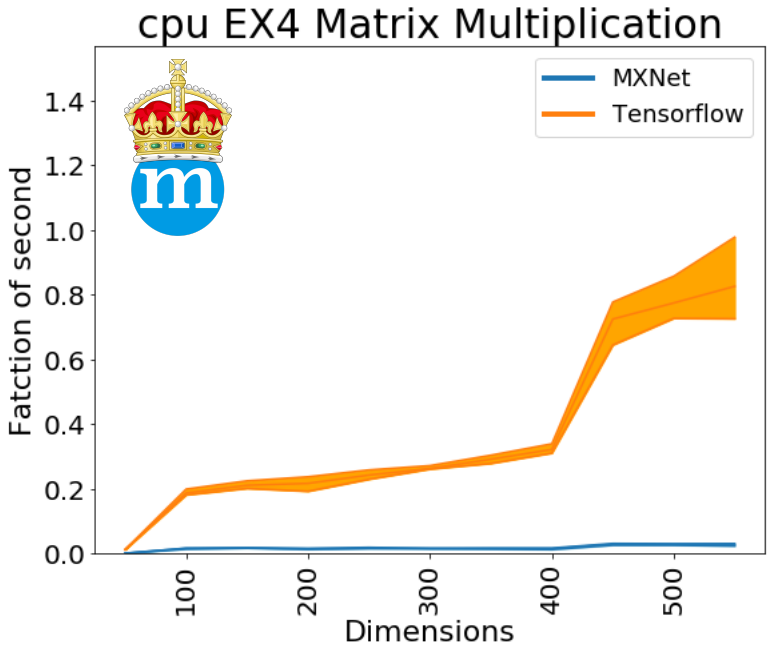


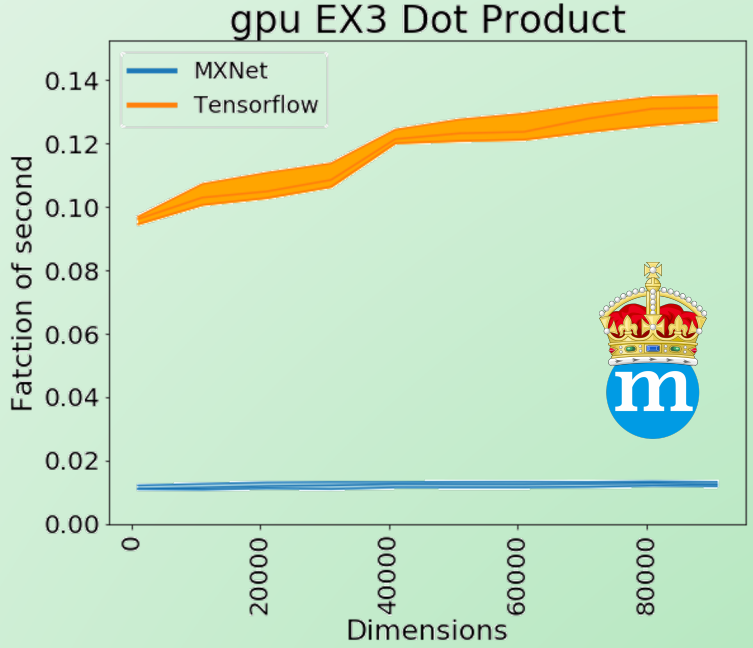


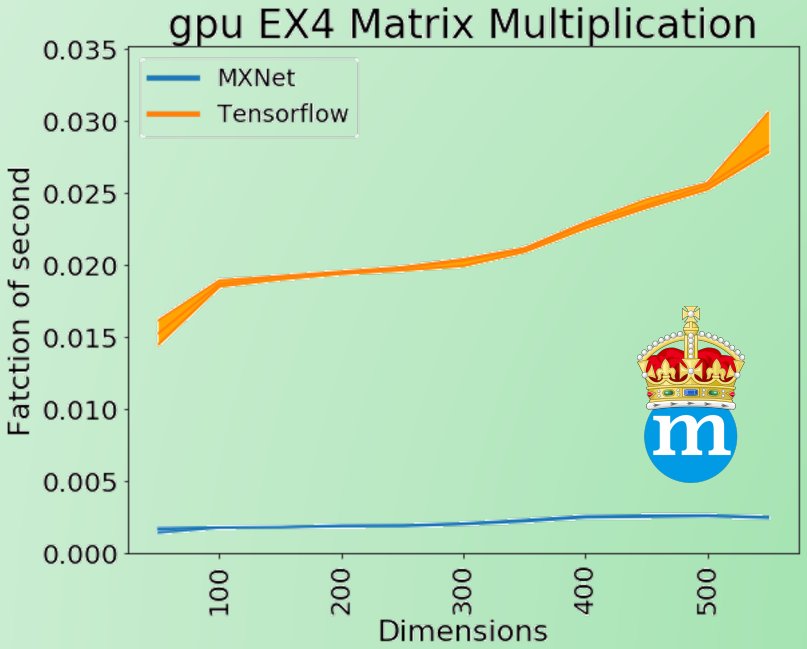
Dot Product & Matrix Multiplication



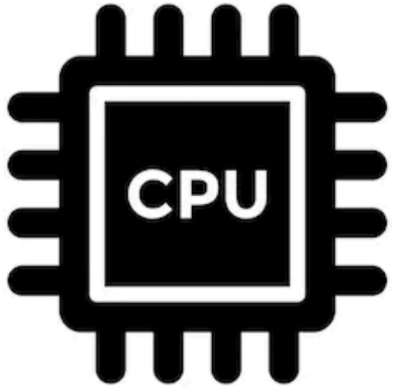


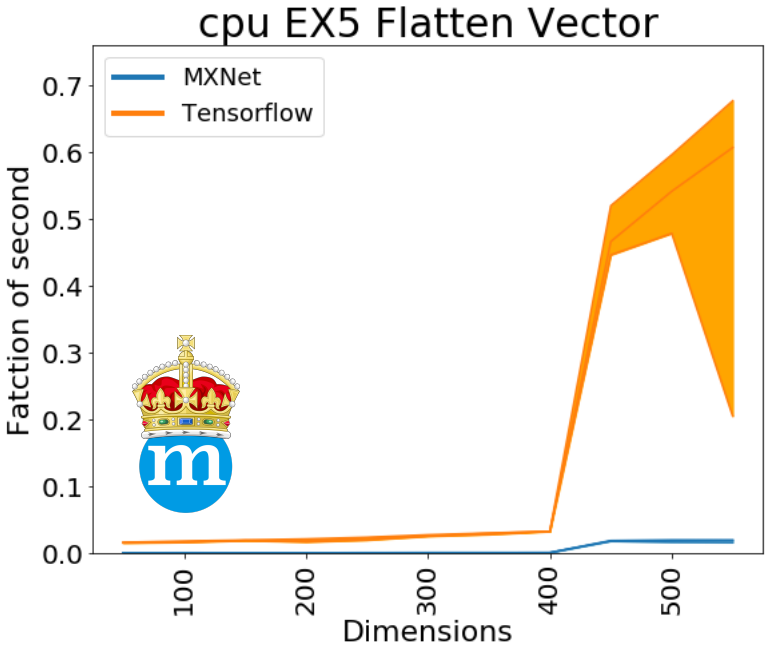


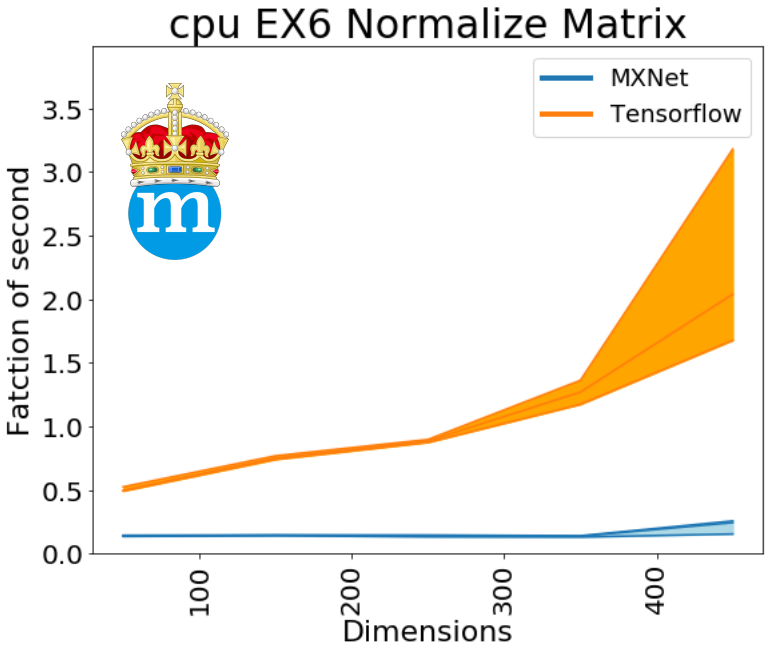


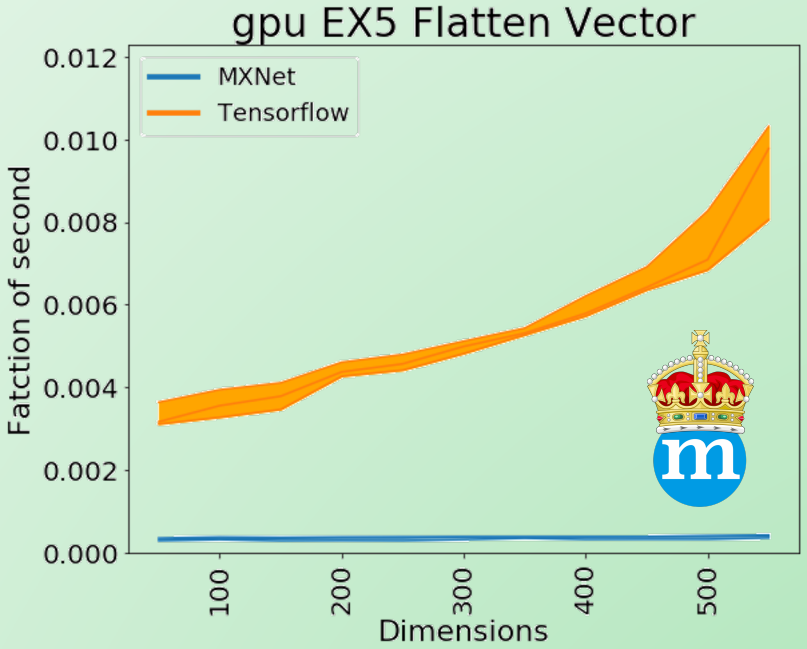


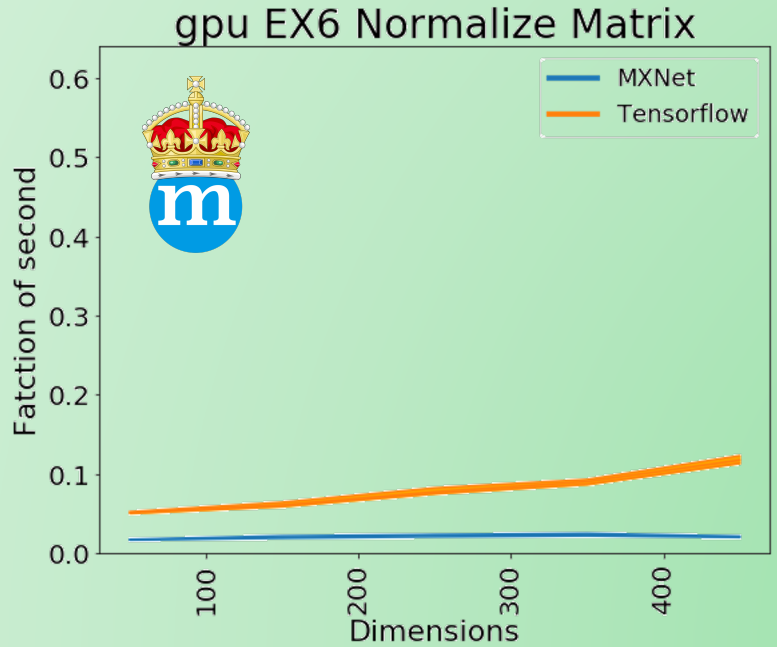
Flatten a vector & Matrix normalization











Benchmark Summary – Fundamental Operations on GPU

4. Benchmark B - LeNet on GPU

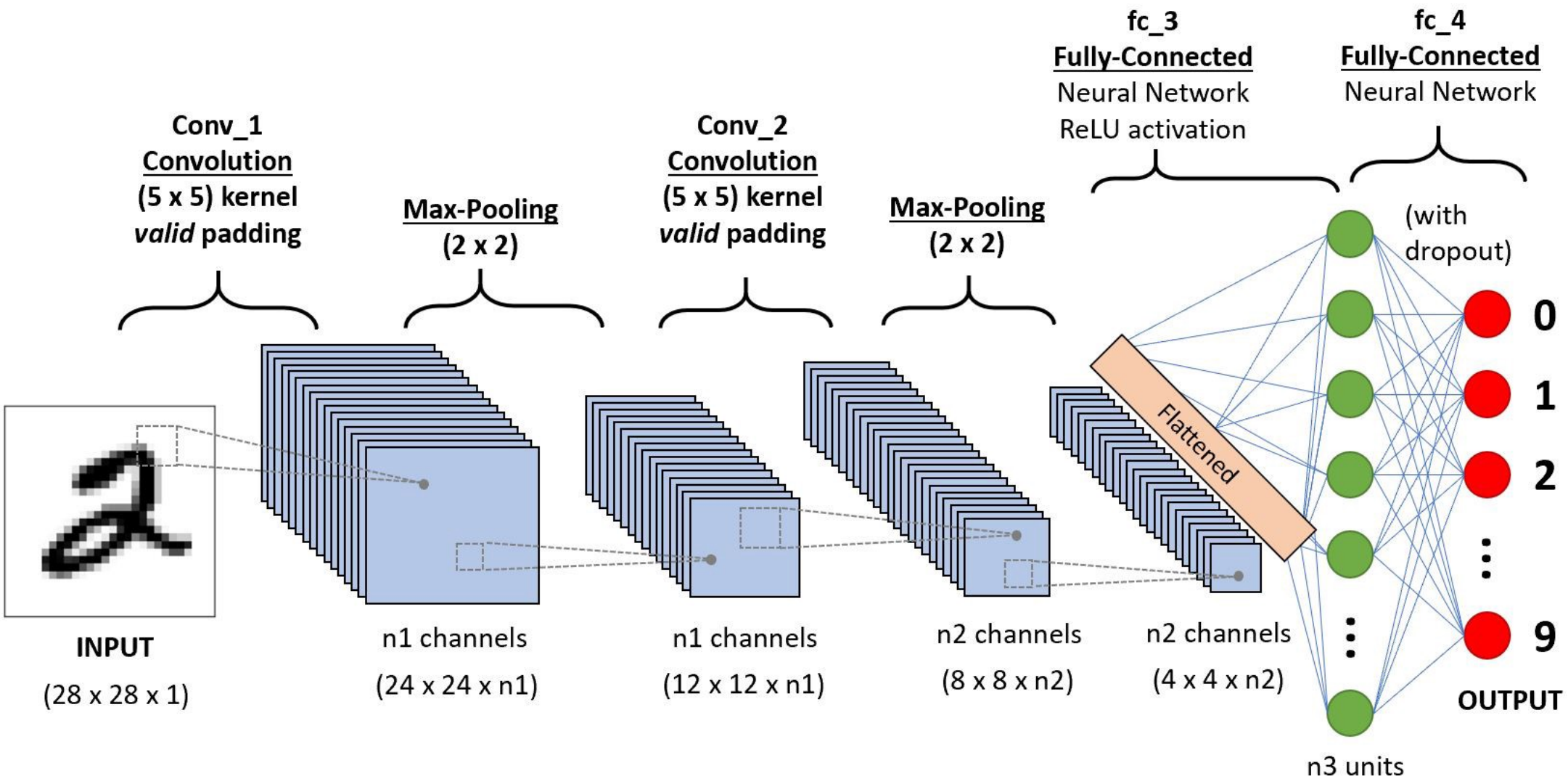


Image source: https://www.researchgate.net/figure/Toy-example-illustrating-the-drawbacks-of-max-pooling-and-average-pooling_fig2_300020038

LeNet Architecture

BDAPro
@DIMA

Layer		Feature Map	Input Size	Kernel Size	Stride	Activation
Input	Image	1	28*28	-	-	-
1	Convolution	6	28*28	5*5	1	Relu
2	Average Pooling	6	24*24	2	2	-
3	Convolution	16	12*12	3*3	1	Relu
4	Average Pooling	16	10*10	2	2	-
Flatten						
5	FC	-	120	-	-	Relu
6	FC	-	84	-	-	Relu
Output	FC	-	10	-	-	Softmax

LeNet construction

GLUON

```
handwritten_net = nn.Sequential()
handwritten_net.add(nn.Conv2D(channels=6, kernel_size=5, activation='relu'),
        nn.MaxPool2D(pool_size=2, strides=2),
        nn.Conv2D(channels=16, kernel_size=3, activation='relu'),
        nn.MaxPool2D(pool_size=2, strides=2),
        nn.Flatten(),
        nn.Dense(120, activation="relu"),
        nn.Dense(84, activation="relu"),
        nn.Dense(10))
```

K Keras

```
model = keras.Sequential()
model.add(layers.Conv2D(filters=6, kernel_size=(5, 5), activation='relu', input_shape=(28,28,1)))
model.add(layers.AveragePooling2D())
model.add(layers.Conv2D(filters=16, kernel_size=(3, 3), activation='relu'))
model.add(layers.AveragePooling2D())
model.add(layers.Flatten())
model.add(layers.Dense(units=120, activation='relu'))
model.add(layers.Dense(units=84, activation='relu'))
model.add(layers.Dense(units=10, activation = 'softmax'))
```

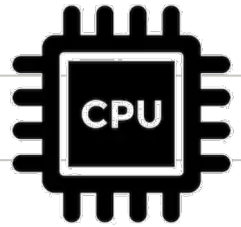
Model Summary

```
Model: "sequential_2"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_4 (Conv2D)            (None, 24, 24, 6)         156
_________________________________________________________________
average_pooling2d_4 (Average (None, 12, 12, 6)         0
_________________________________________________________________
conv2d_5 (Conv2D)            (None, 10, 10, 16)        880
_________________________________________________________________
average_pooling2d_5 (Average (None, 5, 5, 16)          0
_________________________________________________________________
flatten_2 (Flatten)          (None, 400)               0
_________________________________________________________________
dense_6 (Dense)              (None, 120)               48120
_________________________________________________________________
dense_7 (Dense)              (None, 84)                10164
_________________________________________________________________
dense_8 (Dense)              (None, 10)                850
=================================================================
Total params: 60,170
Trainable params: 60,170
Non-trainable params: 0
```

LeNet Architecture

BDAPro
@DIMA

Hyperparameter	Value
Optimizer	Adam
Learning rate	0.001
Loss function	SOFTMAX-CROSSENTROPY
mini-batch size	200
Epochs	5-15



Framework	Hardware	#Epoch	Start Acc	Val Acc	Total Time	Avg time/epoch
Tensorflow	CPU	5	0.8639	0.9825	256	49.2
Tensorflow	CPU	10	0.8594	0.9855	538	53.8
Tensorflow	CPU	15	0.8639	0.9861	693	46.2
MXNet	CPU	5	0.4874	0.9198	187	37.4
MXNet	CPU	10	0.3278	0.9243	342	34.2
MXNet	CPU	15	0.5893	0.9467	503	33.53
Tensorflow	GPU	5	0.8639	0.9835	32	6.4
Tensorflow	GPU	10	0.8743	0.9795	59	5.9
Tensorflow	GPU	15	0.8779	0.989	94	6.26
MXNet	GPU	5	0.4537	0.9294	30	5
MXNet	GPU	10	0.5736	0.944	53	5.3
MXNet	GPU	15	0.5987	0.9666	85	5.6

Results

BDAPro
@DIMA

- **Faster convergence to high accuracy** for TensorFlow
- **Faster** overall **training** execution for **MXNet**
- Smaller difference in speed between the two frameworks when compared to CPU

5. Optimization Available

Optimization Options

BDAPro
@DIMA

TensorFlow

- XLA (Accelerated Linear Algebra)
- Eager Execution Turned Off
- TFRecord: better reading efficiency (when data loading is bottleneck)
- ImageDataGen

MXNet

- Hybridize
- RecordIO: better reading efficiency (recommended for images)

MXNet - Hybridize

BDAPro
@DIMA

- It allows you to **convert your imperative code to a static symbolic graph**, which is much more efficient to execute.
- There are two main benefits of hybridizing your model:
  - **better performance** and
  - easier **serialization** for deployment.
- The best part is that it's as **simple** as just calling net.hybridize() before training.
- In MXNet, a symbolic program refers to a program that makes use of the **Symbol type**.

```
x = sym.var('data')
net(x)
```

How to - Hybridize your network

BDAPro
@DIMA

1. Use HybridBlocks (e.g. HybridSequential)
2. Call net.hybridize()
3. Done

Modification in the code

```
model_mx = nn.HybridSequential()
model_mx.add(nn.Conv2D(channels=6, kernel_size=5, activation='relu'),
        nn.AvgPool2D(pool_size=2, strides=2),
        nn.Conv2D(channels=16, kernel_size=3, activation='relu'),
        nn.AvgPool2D(pool_size=2, strides=2),
        nn.Flatten(),
        nn.Dense(120, activation="relu"),
        nn.Dense(84, activation="relu"),
        nn.Dense(10))

model_mx.hybridize()
```

What is not allowed in HybridBlocks

BDAPro
@DIMA

Access to specific elements in tensor

```
def hybrid_forward(self, F, x):
    return x[0,0]
```

No conditional logic on type

```
def hybrid_forward(self, F, x):
    if x.dtype =='float16':
        return x
    return x*2
```

No direct assignment

```
def hybrid_forward(self, F, x):
    x[0] = 2
    return x
```

No access to context of symbols

```
def hybrid_forward(self, F, x):
    if x.context == mx.cpu():
        return x
    return x*2
```

No access to shape of tensor

```
def hybrid_forward(self, F, x):
    return x*x.shape[0]
```

Work in progress to allow this operation in the Hybrid programming soon

Source: https://github.com/apache/incubator-mxnet/blob/b3bbbbe082ec96105c93a5ab3857fcff0c032505/docs/python_docs/python/tutorials/packages/gluon/blocks/hybridize.md

TensorFlow – XLA

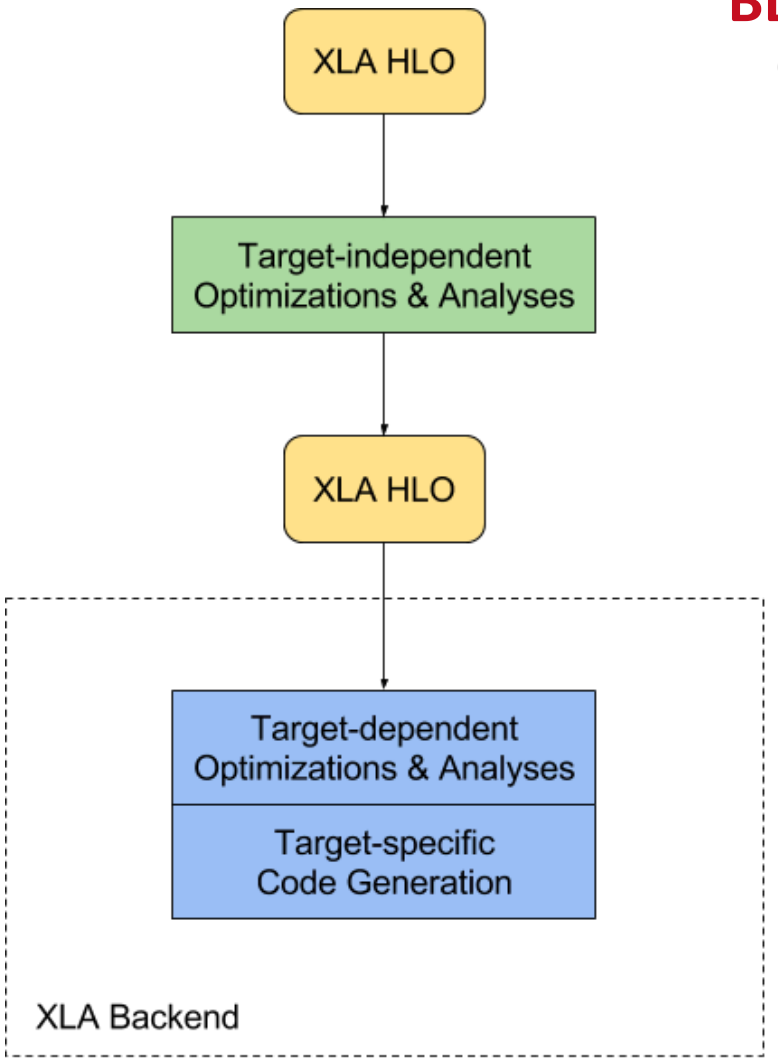
(Accelerated Linear Algebra)

BDAPro
@DIMA

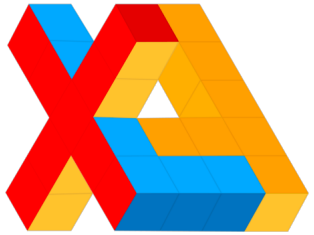
- No code change required

1. The input language to XLA is called "HLO IR", or just HLO (**High Level Optimizer Intermediate Representation**).
2. XLA takes graphs ("computations") defined in HLO and compiles them into machine instructions for various architectures.

- Modular: optimization can be platform dependent



TensorFlow - XLA



BDAPro
@DIMA

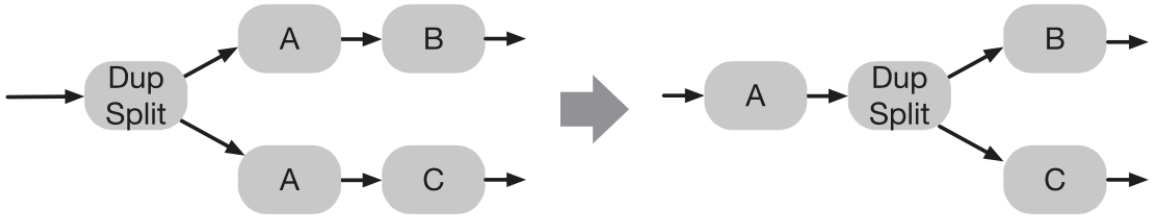
- XLA comes with several optimizations and analysis passes that are target-independent, such as:

CSE Common Subexpression **Elimination**

Target-independent Operation **Fusion**

Buffer analysis for allocating runtime memory for the computation.

Eliminate redundant computations.



Avoid the overhead of data serialization and transport.

Image source: **A Catalog of Stream Processing Optimizations,** Hirzel et al.

How to – activate XLA

BDAPro
@DIMA

1. Insert one line in your code to switch on the XLA
2. If you are running on GPU:
   - it is enabled
3. If on CPU:
   - Run your script with the following flags via command line

Modification in the code

```
import tensorflow as tf
tf.config.optimizer.set_jit(True) # XLA enabled.
```

```
$ TF_XLA_FLAGS="--tf_xla_auto_jit=2 --tf_xla_cpu_global_jit" python3 path/to/your/python/script.py
```

6. Benchmark C – XLA on CIFAR10 & Eagerexecution in TF

Classifying CIFAR-10 with XLA

BDAPro
@DIMA

- Adapted from: https://github.com/tensorflow/tensorflow/blob/master/tensorflow/compiler/xla/g3doc/tutorials/autoclustering_xla.ipynb

"On a machine with a Titan V GPU and an Intel Xeon E5-2690 CPU the speed up is ~1.17x."

Network construction

We define the model, adapted from the Keras CIFAR-10 example:

```
def generate_model():
  return tf.keras.models.Sequential([
    tf.keras.layers.Conv2D(32, (3, 3), padding='same', input_shape=x_train.shape[1:]),
    tf.keras.layers.Activation('relu'),
    tf.keras.layers.Conv2D(32, (3, 3)),
    tf.keras.layers.Activation('relu'),
    tf.keras.layers.MaxPooling2D(pool_size=(2, 2)),
    tf.keras.layers.Dropout(0.25),

    tf.keras.layers.Conv2D(64, (3, 3), padding='same'),
    tf.keras.layers.Activation('relu'),
    tf.keras.layers.Conv2D(64, (3, 3)),
    tf.keras.layers.Activation('relu'),
    tf.keras.layers.MaxPooling2D(pool_size=(2, 2)),
    tf.keras.layers.Dropout(0.25),

    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(512),
    tf.keras.layers.Activation('relu'),
    tf.keras.layers.Dropout(0.5),
    tf.keras.layers.Dense(10),
    tf.keras.layers.Activation('softmax')
  ])

model = generate_model()
```

BDAPro
@DIMA

XLA on/off	Hardware	#Epoch	Start Acc	Val Acc	Total Time	Avg time/epoch	Speed Gain
off	CPU	5	0.2536	0.4795	35min16	423.3s	
on	CPU	5	0.2411	0.4745	32min 28s	389.6s	1.08
off	GPU	5	0.2125	0.4761	26s	5.2s	
on	CPU	5	0.2481	0.4771	30s	6s	-0.86
off	GPU	10	0.2141	0.5505	52s	5.2s	
on	GPU	10	0.2314	0.5326	55s	5.5s	-0.94
off	GPU	20	0.2589	0.6455	118s	5.9s	
on	GPU	20	0.245	0.6342	105s	5.25s	1.12
off	GPU	25	0.2441	0.6452	131s	5s	26.4
on	GPU	25	0.2476	0.65	114s	4.56s	1.15
off	GPU	30	0.2139	0.6561	154s	5.13s	
on	GPU	30	0.212	0.6617	155s	5.16s	-0.99

Eager Execution

BDAPro
@DIMA

- An imperative programming environment that evaluates operations immediately, without building graphs.
- Operations return concrete values instead of constructing a computational graph to run later.

--> On by default in TF2.0

--> Easier to debug models

Check Status

```
>>> import tensorflow as tf
>>> print(tf.__version__)
2.0.0-beta1
>>> print(tf.executing_eagerly())
True
```

```
import tensorflow as tf

tf.compat.v1.disable_eager_execution()
```

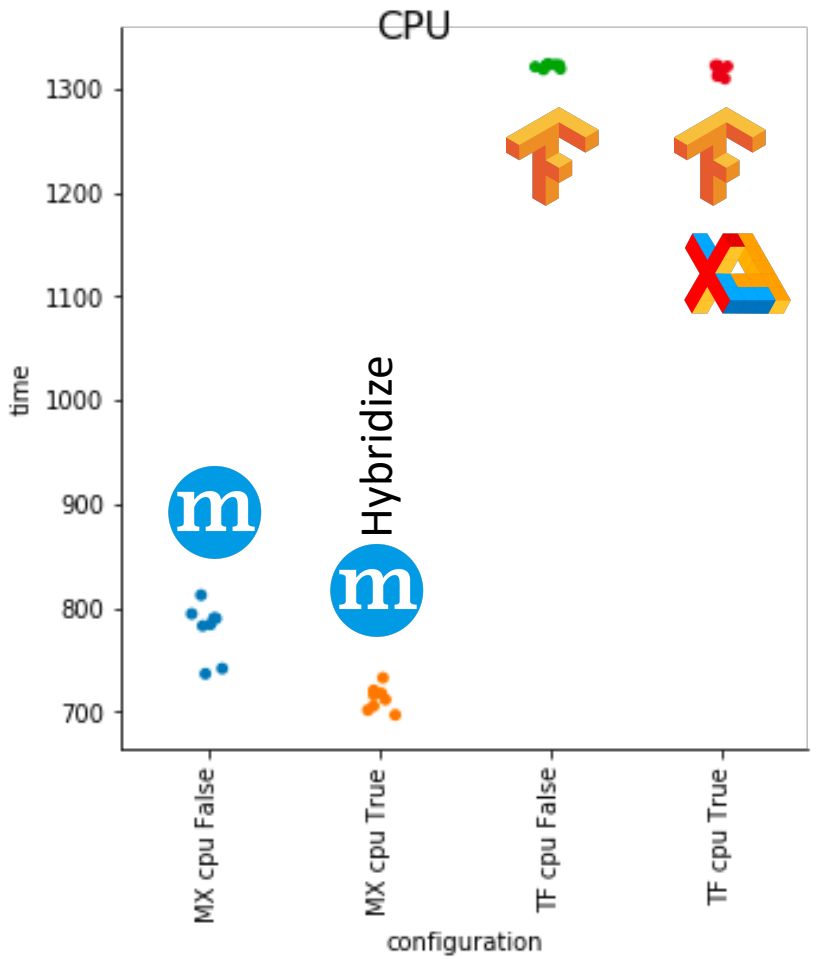
Results

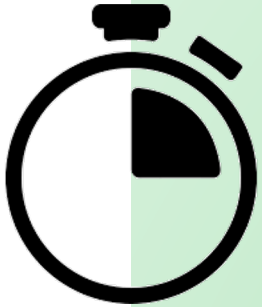
Eagerexecution	Epochs	Batch		Accuracy	Total time
On	5	200	CPU	0.9906	186
Off	5	200	CPU	0.9873	260

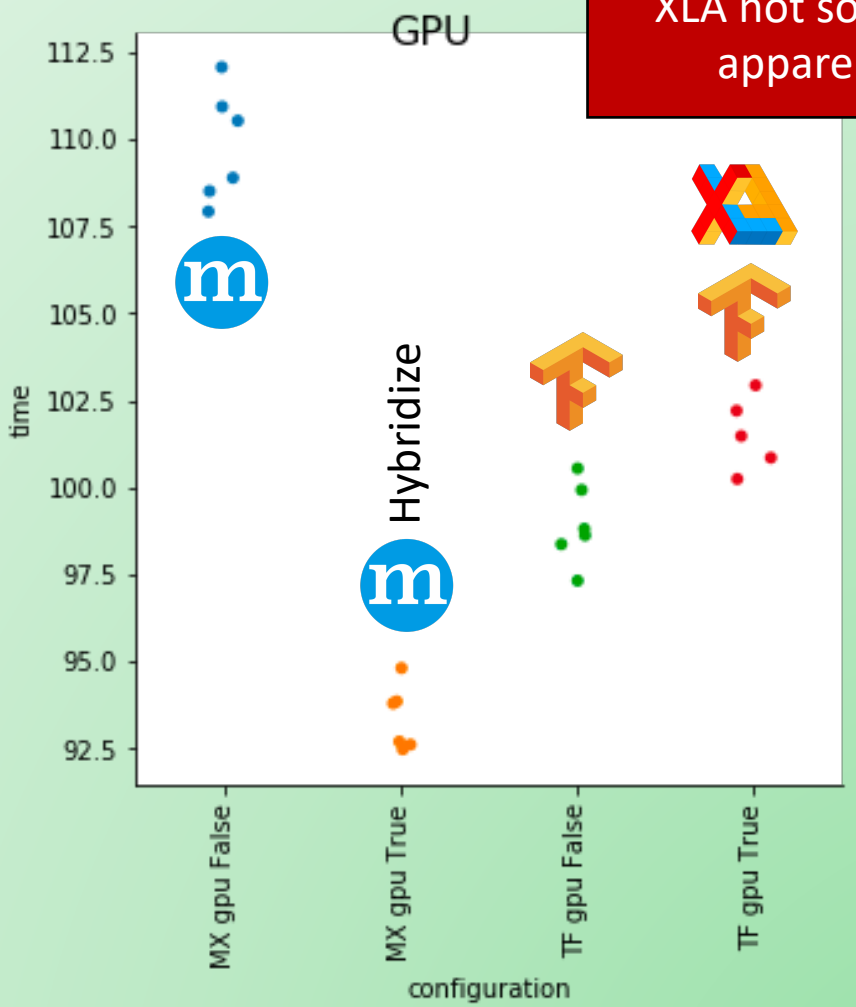
7. Benchmark D - Optimizations on LeNet (CPU vs GPU)

Comprehensive Comparison – Hybridize (MXNet) vs XLA (Tensorflow)

Optimization Comparisons – LeNet - 15 epochs - Time

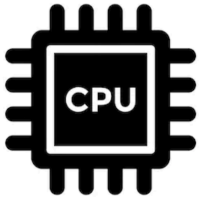






XLA not so useful apparently

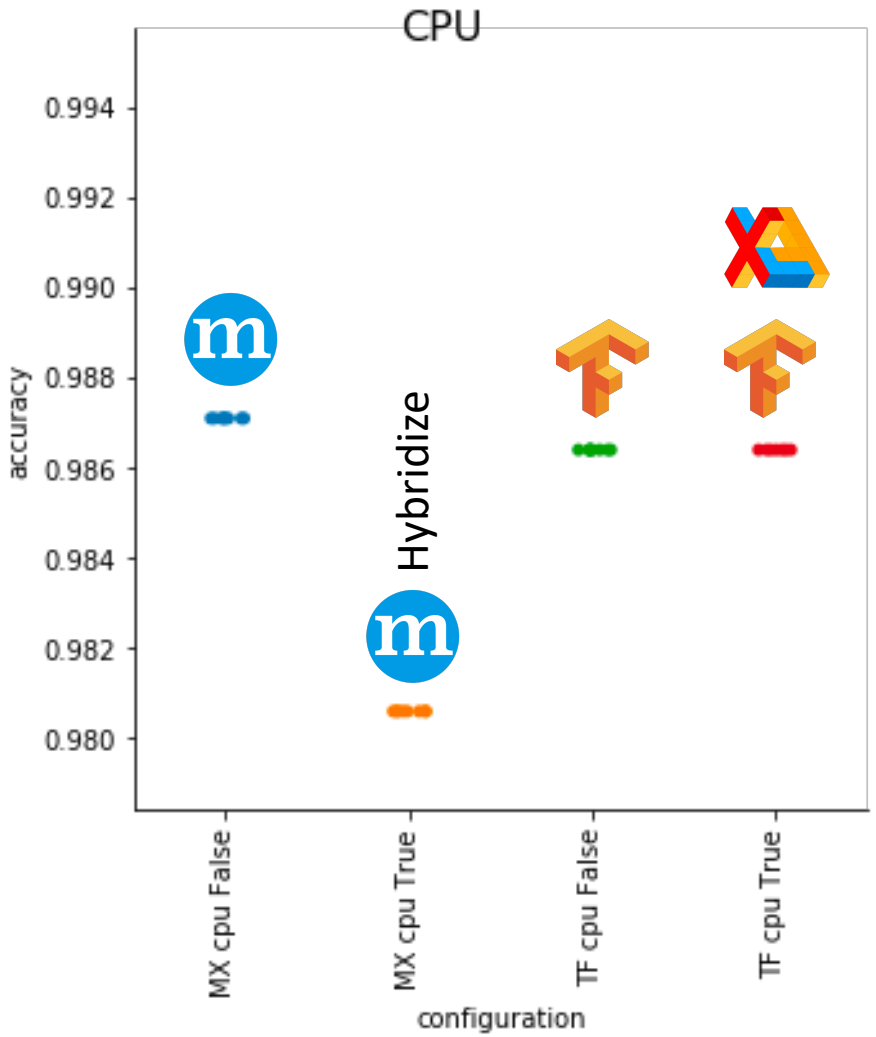
MNNet is ~1.73x faster than TF on CPU

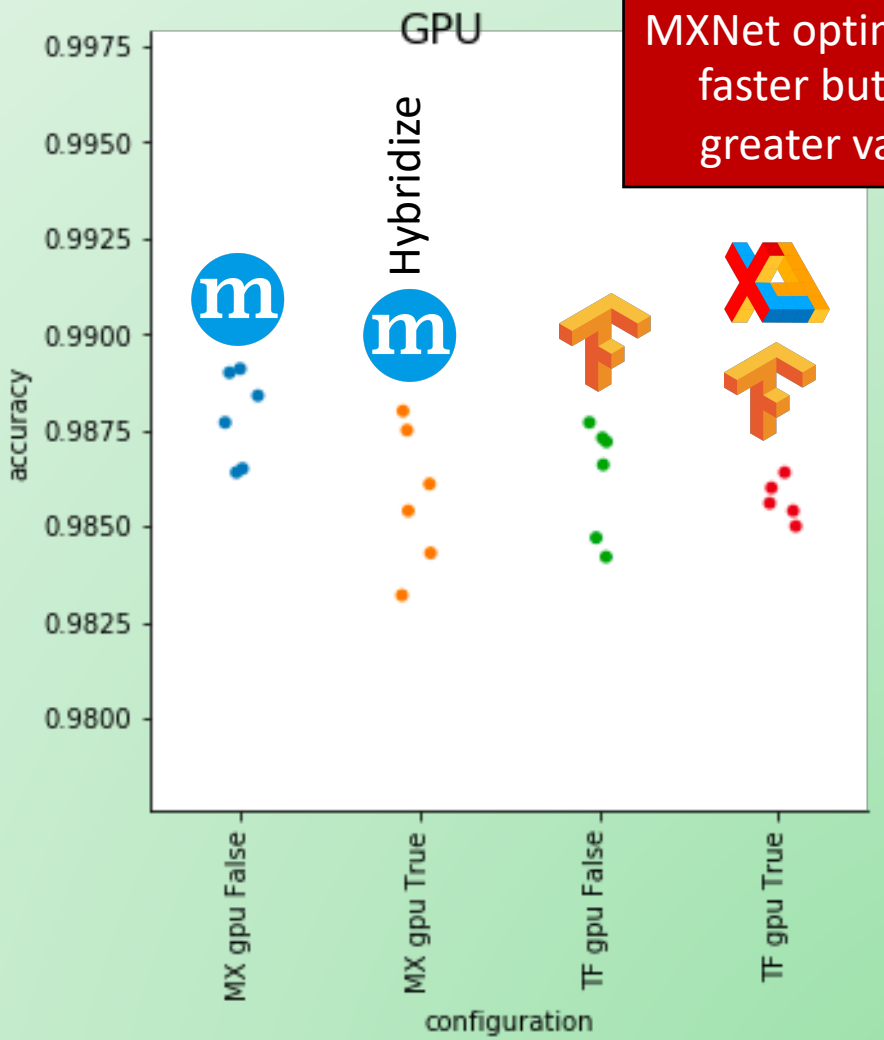


MNNet is ~1.06x faster than TF on GPU

NVIDIA

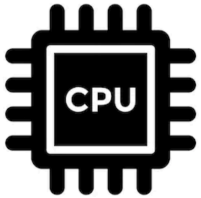
Optimization Comparisons – LeNet - 15 epochs - Accuracy





MXNet optimized was faster but with a greater variance

On CPU reproduciblity is mantained



Not reproducible on GPU

Results Interpretation

BDAPro
@DIMA

```
import random
np.random.seed(42)
random.seed(42)
for computing_unit in ctx:
    mx.random.seed(42, ctx = computing_unit)
tf.random.set_seed(42)
```

1. All the results with **cpu are reproducible** (same exact accuracy), while with gpu they are not
2. If you have **only CPU**, MXNet is almost double as fast (1.73x) as TensorFlow
3. On GPU, MXNet can be either the fastest (optimization on) or the more accurate (optimization off).
4. MXNet optimized is the fastest configuration, but at the price of a high variance of accuracy.
5. In this configuration, TensorFlow with XLA is not improving performance, but adding overhead probably.

Q: Why is XLA not improving the speed?

BDAPro
@DIMA

A: Probably the architecture of LeNet does not have any operator that can be optimized. Therefore the counter intuitive increase of time with XLA switched on may be due to the overhead of XLA that tried to analysed the graph looking for possible optimizations, but none was found.

Similar conclusions were found in this discussion on Github

Source: https://github.com/tensorflow/tensorflow/issues/30791

8. Summary – Takeaways

Summary

BDAPro
@DIMA

- Even on GPU, MXNet optimized outperforms TensorFlow
- MXNet optimization is giving a clear boost in performance
- XLA can be useless in some cases (like LeNet)
- MXNet still has slow convergence even on GPU

Takeaways

BDAPro
@DIMA

- Usability:
  - Almost equivalent and intuitive network definition
  - TensorFlow has an easier fit API
- Shape of tensors is the cause of most of the problems in the network
- Tensorflow is better for accuracy while MXnet is better for speed (Benchmark B)
- CPU-based distributed environments are not efficient in terms of speed for most frameworks.[1]
- MXnet remains the best performing for convolution [5 , 2]

9. Future Work

Future Directions

BDAPro
@DIMA

1. Benchmark tf.data.Dataset to read data
   - Benchmark of ImageDataGenerator [Kaoutar - Work in Progress]
2. Try on TPUs on Google Cloud
3. Investigate the effect of XLA on more complex networks
4. Extend to other Deep Learning frameworks (e.g. Pytorch)
5. Extend full comparison (MXNet vs TF) to other network architectures
   - Benchmark of CIFAR10 on MXNet [Work in Progress]

Things Already Addressed

BDAPro
@DIMA

- **Rysia framework** adaptation to TensorFlow 2.1
- Tried LeNet on **TPU in Colab** environment -> Worst than GPUs
- **cProfile** to explore which are the fundamental operations called during training -> TensorFlow core obfuscate the details

Questions?

Suggestions?

Thank you!

x. Porting Rysia Framework to TensorFlow 2.0

Paper: End-to-End Benchmarking of Deep Learning Platforms

Rysia:
Declarative Benchmarking Framework

BDAPro
@DIMA

Blueprint

1. Define the network (representative workload)

Rysia

TensorFlow

mxnet

PYTORCH

2. Translate into fundamental operations

Operator
Matrix-Multiplication
Convolution (2D)
Bias Addition
Max Pooling (2D)

amazon web services™

3. Execute and get metrics

Porting

Major Release

1.12.2 -> **2.0.0**

TensorFlow

1.4.1 -> **1**.5.1

mxnet

1.12.2

```
W = tf.Variable(
  tf.glorot_uniform_initializer()(
    (10, 10)))
b = tf.Variable(tf.zeros(10))
c = tf.Variable(0)

x = tf.placeholder(tf.float32)
ctr = c.assign_add(1)
with tf.control_dependencies([ctr]):
  y = tf.matmul(x, W) + b
init =
  tf.global_variables_initializer()

with tf.Session() as sess:
  sess.run(init)
  print(sess.run(y,
  feed_dict={x: make_input_value()}))
  assert int(sess.run(c)) == 1
```

2.0.0

```
W = tf.Variable(
  tf.glorot_uniform_initializer()(
    (10, 10)))
b = tf.Variable(tf.zeros(10))
c = tf.Variable(0)

@tf.function
def f(x):
  c.assign_add(1)
  return tf.matmul(x, W) + b

print(f(make_input_value())
assert int(c) == 1
```