Gradient Descent Optimization Algorithms and PyTorch

Giacomo Domeniconi, Ulrich Finkler, Wei Zhang

CSCI-GA.3033-022 HPML

ML Performance Factors

Algorithms Performance

- **PyTorch Optimizer (training): Momentum, Nesterov, Adagrad, AdaDelta**

Hyperparameters Performance

- **Learning rate, Momentum, Batch size, Others**

Implementation Performance

- **Pytorch Multiprocessing, PyTorch DataLoader, PyTorch CUDA**

Framework Performance

- ML Frameworks: PyTorch, TensorFlow, Caffe, MXNET

Libraries Performance

- Math libraries (cuDNN), Communication Libraries (MPI, GLOO)

Hardware Performance

- CPU, DRAM, GPU, HBM, Tensor Units, Disk/Filesystem, Network

Summary

- PyTorch Optimizer
- PyTorch Multiprocessing
- PyTorch Dataloader
- PyTorch CUDA

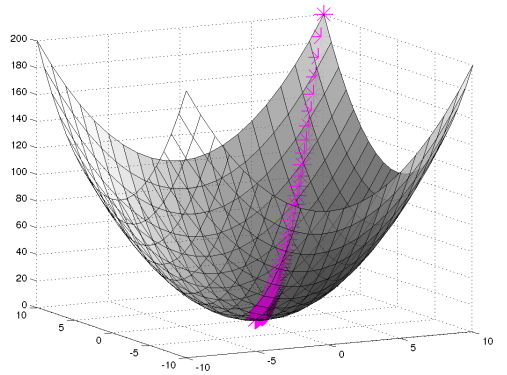
PyTorch Optimizer

Gradient Descent - Recap

- Simplest and very popular
- Main Idea: take a step proportional to the negative of the gradient (i.e. minimize the loss function):

$$\theta = \theta - \alpha \nabla J(\theta)$$

  - Where θ is the parameters vector, α is the learning rate, and $\nabla J(\theta)$ is the gradient of the cost
- Easy to implement
- Each iteration is relatively cheap
- **Can be slow to converge**
- Gradient descent variants:
  - **Batch gradient descent**: Update after computing all the training samples
  - **Stochastic gradient descent**: Update for each training sample
  - **Mini-batch gradient descent**: Update after a subset (mini-batch) of training samples



Learning Rate Challenges

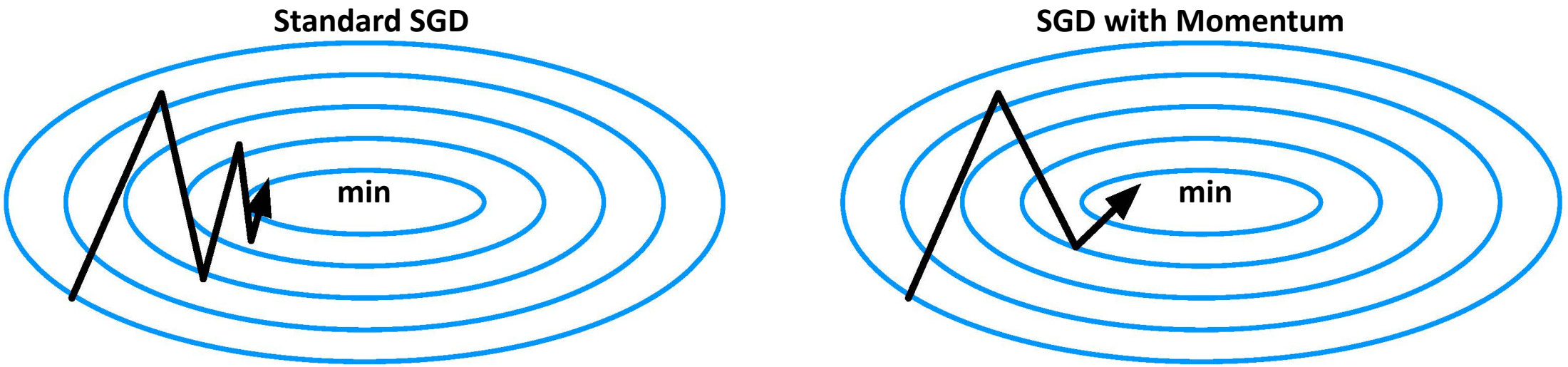
- The Learning rate has a large impact on convergence
  - Too small → too slow
  - Too large → oscillatory and may even diverge
- Choosing a proper (initial) learning rate
- Should learning rate be fixed or adaptive?
  - **Decaying learning rate:** drop by 10 after N iterations and then again after other M iterations, and so on…
  - How to do define an **adaptive learning rate**?
- Avoid to get trapped in local minima
- Changing learning rate for each parameter

Learning Rate Challenges (II)

- Traversing efficiently through error differentiable functions' surfaces is an important research area today
- Some of the recently popular techniques **take the gradient history into account** such that each "move" on the error function surface relies on previous ones a bit
- Many algorithms already implemented in PyTorch
  - However, these algorithms often used as black-box tools: need to understand their strength and weakness

Optimizer Algorithms – Momentum

- SGD with Momentum:
  - Descent with momentum keeps going in the same direction longer
  - Descent is faster because it takes less steps (*W* updates)



From: http://www.del2z.com/2016/06/param-optimiz-3/

Optimizer Algorithms - Momentum

- Momentum is a simple method that helps accelerate SGD by adding a fraction γ of the update vector of the past time step to the current update vector
- In simple words momentum adds a **velocity** component to the parameter update routine

$$v_t = \gamma v_{t-1} + \alpha \nabla J(\theta)$$

$$\theta = \theta - v_t$$

- In PyTorch, momentum is implemented in the default SGD method

```
optimizer = torch.optim.SGD(model.parameters(), lr=0.1, momentum=0.9)
optimizer.zero_grad()
loss_fn(model(input), target).backward()
optimizer.step()
```

Optimizer Algorithms- Nesterov Momentum

- Instead of compute the gradient of the current position, it computes the gradient at the approximated new position
- Use the **next approximated position's gradient** with the hope that it will give us better information when we're taking the next step:

$$v_t = \gamma v_{t-1} + \alpha \nabla J(\theta - \gamma v_{t-1})$$

$$\theta = \theta - v_t$$

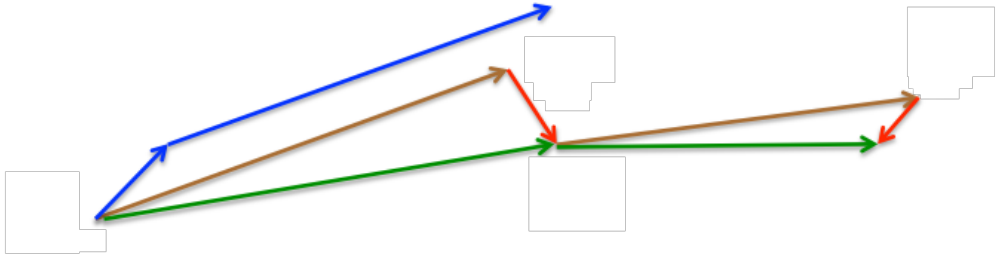
- Momentum is usually set to 0.9
- In PyTorch, also the Nesterov momentum is implemented in the default SGD method

```
optimizer = torch.optim.SGD(model.parameters(), lr=0.1, momentum=0.9, nesterov=True)
```

Classical vs Nesterov Momentum

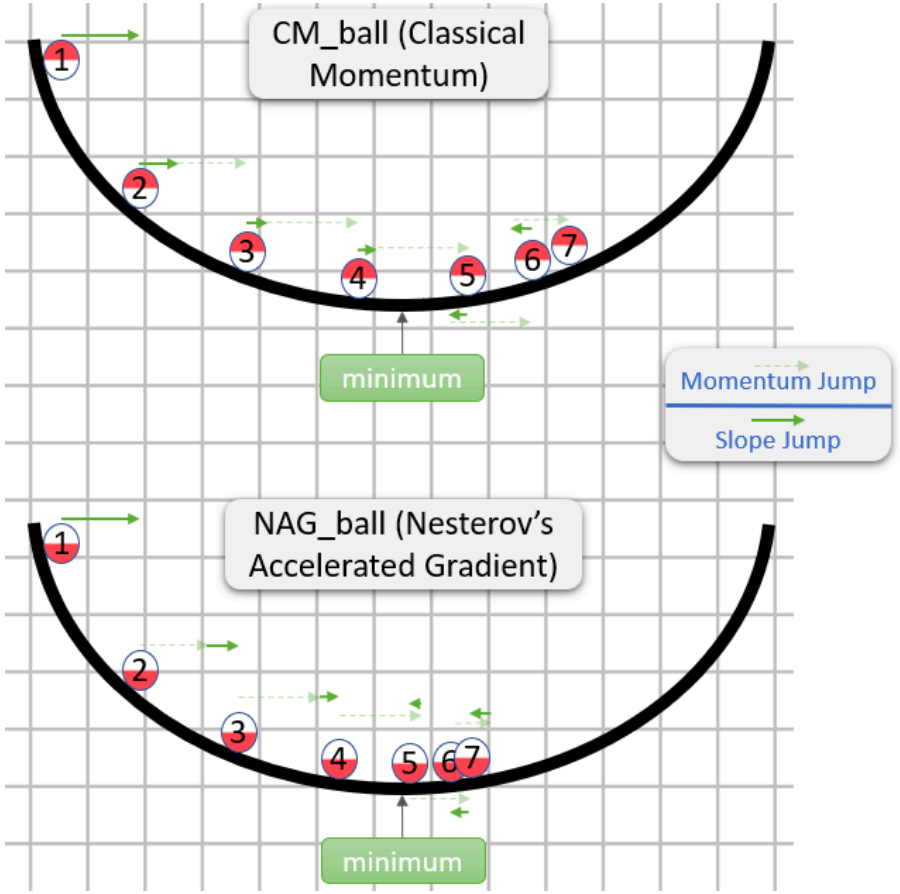
Nesterov Momentum steps:

- First make a big jump in the direction of the previous accumulated gradient.
- Then measure the gradient where you end up and make a correction.



brown vector = jump, red vector = correction, green vector = accumulated gradient

blue vectors = standard momentum



Optimizer Algorithms - Adagrad

- Adapts learning rate to parameters: larger updates for more frequent parameters
- AdaGrad update rule is given by the following formula:

$$\theta_{t+1} = \theta_t - \frac{\alpha}{\sqrt{diag(G_t)+\epsilon}} \odot \nabla J(\theta_t)$$

- θ_t vector of parameters at step t (size *d*)
- $\nabla J(\theta_t)$ vector of gradients at step t
- $\odot$ is the element-wise product
- G_t is the historical gradient information until step t:
  - diagonal matrix *d x d* (d = #parameters) with sum of squares of gradients until time t
- *diag()*: Transform G_t diagonal into a vector
- ϵ is the smoothing term to avoid division by 0 (usually 1e-8)
- In PyTorch:

```
optimizer = torch.optim.Adagrad(params, lr=0.01)
```

Optimizer Algorithms - Adagrad II

- Pros:
  - It is well-suited for dealing with sparse data
  - It greatly improved the robustness of SGD
  - It eliminates the need to manually tune the learning rate
- Cons:
  - Main weakness is its accumulation of the squared gradients in the denominator
  - This causes the learning rate to shrink and become infinitesimally small
  - The algorithm can no longer acquire additional knowledge
- In PyTorch:

```
optimizer = torch.optim.Adagrad(params, lr=0.01)
```

Optimizer Algorithms - Adadelta

- One of the inspiration for AdaDelta was to improve AdaGrad weakness of learning rate converging to zero with increase of time
- Adadelta mixes two ideas:
  1. to scale learning rate based on historical gradient while taking into account only recent time window – not the whole history, like AdaGrad
  2. to use component that serves an acceleration term, that accumulates historical updates (similar to momentum)
- Adadelta step is composed of the following phases:
  1. Compute gradient g_t at current step t
  2. Accumulate gradients (AdaGrad-like step)
  3. Compute update
  4. Accumulate updates (momentum-like step)
  5. Apply the update
- In PyTorch:

```
optimizer = torch.optim.Adadelta(params, lr=1.0, rho=0.9, eps=1e-06)
```

Optimizer Algorithms - Adam

- Adam might be seen as a generalization of AdaGrad
  - AdaGrad is Adam with certain parameters choice
- The idea is to mix benefits of:
  - **AdaGrad** that maintains a per-parameter learning rate that improves performance on problems with sparse gradients (e.g. NLP and computer vision problems).
  - **RMSProp** that also maintains per-parameter learning rates that are adapted based on the average of recent magnitudes of the gradients for the weight (e.g. how quickly it is changing).
    - This means the algorithm does well on online and non-stationary problems (e.g. noisy)
- Specifically, the algorithm calculates an exponential moving average of the gradient and the squared gradient, and the parameters β_1 and β_2 control the decay rates of these moving averages

1. $m_k=\beta_1 m_{k-1}+(1-\beta_1)g_k$ The first moment
2. $v_k=\beta_2 v_{k-1}+(1-\beta_2)g_k^2$ The second moment
3. $\hat{m}_k = \frac{m_k}{1-\beta_1^{\,k}}$ Compute bias-corrected first moment estimate
4. $\hat{\hat{v}}_k = \frac{v_k}{1-\beta 2^{k}}$ Compute bias-corrected second raw moment estimate
5. $\hat{\hat{\theta}}_k = \theta_{k-1}-\alpha \frac{m_k}{\mathrm{sqrt}(\hat{\hat{v}}_k)-\epsilon}$ Update parameters

- In PyTorch:

```
torch.optim.Adam(params, lr=0.001, betas=(0.9, 0.999), eps=1e-08)
```

And many others…

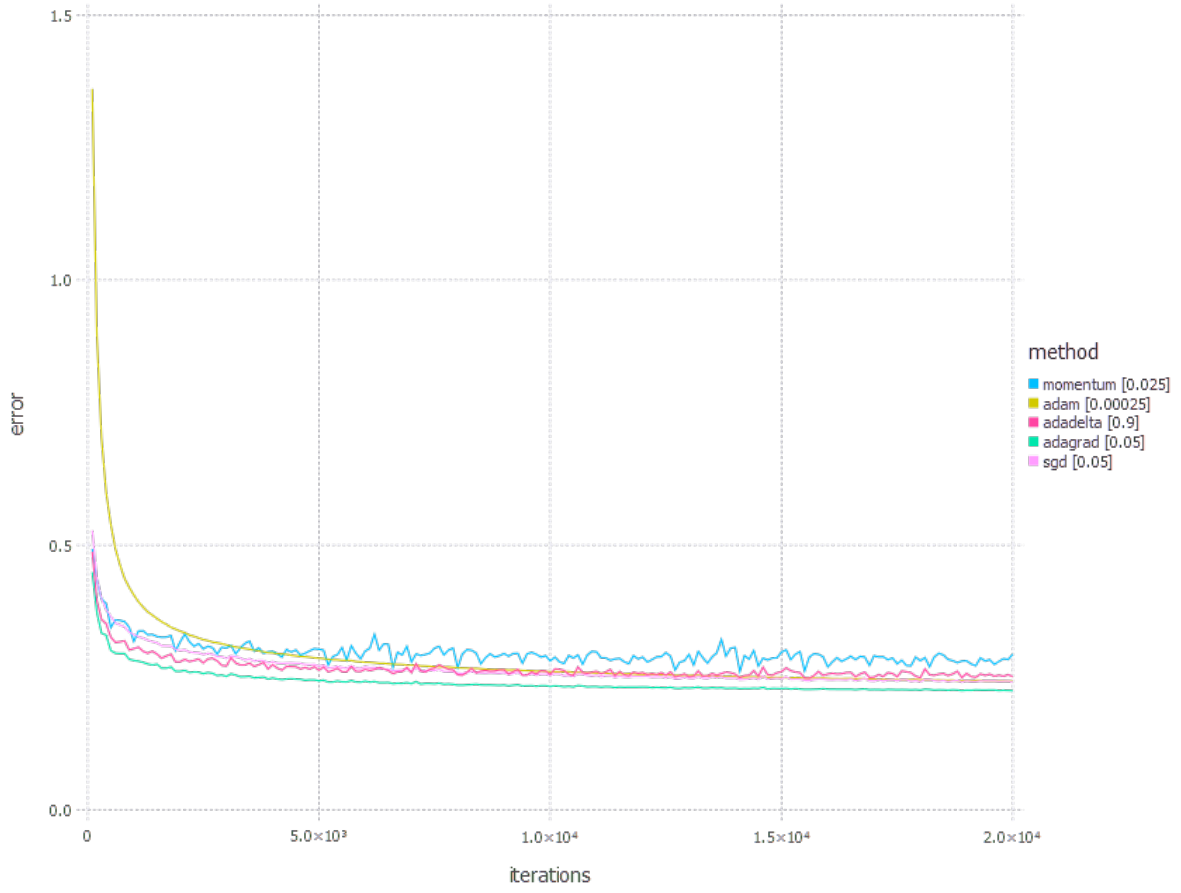
- AdaMax: variant of Adam
- Nadam
  - Adam with Nesterov momentum instead of classical momentum
- RMSprop
  - Divide the gradient by a running average of its recent magnitude
- Yellowfin
  - an automatic tuner for momentum and learning rate in SGD
  - the newer and with results that seem to overcome all the other
- ….
- Here some interesting readings:
  - https://arxiv.org/abs/1609.04747
  - http://ruder.io/optimizing-gradient-descent/

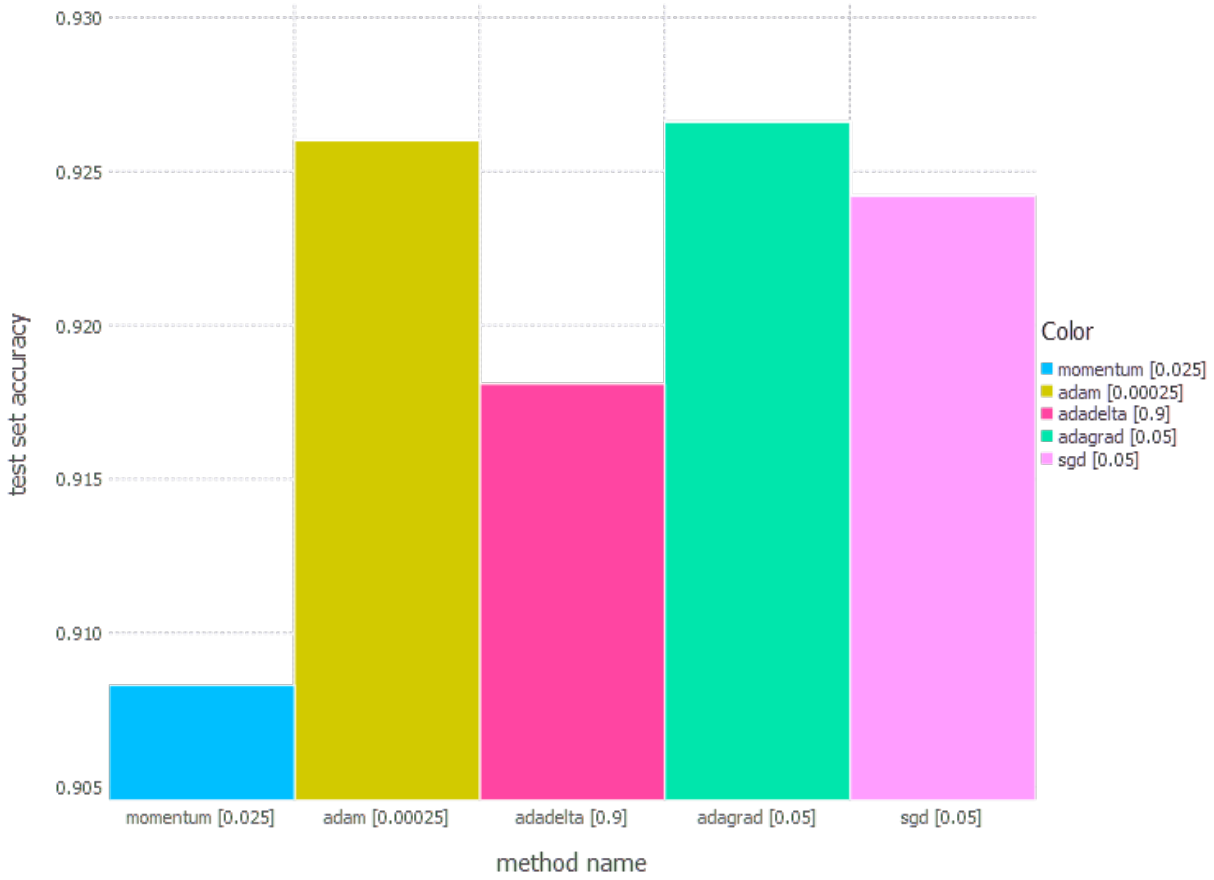
Optimization Techniques Comparison (I)

- Toy example on MNIST Classification
- Three different network architecture tested:
  1. network with linear layer and softmax output (softmax classification)
  2. network with sigmoid layer (100 neurons), linear layer and softmax output
  3. network with sigmoid layer (300 neurons), ReLU layer (100 neurons), sigmoid layer (50 neurons) again, linear layer and softmax output
- Mini-batch size of 128
- Run the algorithm for approx. 42 epochs (20000 iterations)
- http://int8.io/comparison-of-optimization-techniques-stochastic-gradient-descent-momentum-adagrad-and-adadelta/

Results on net 1

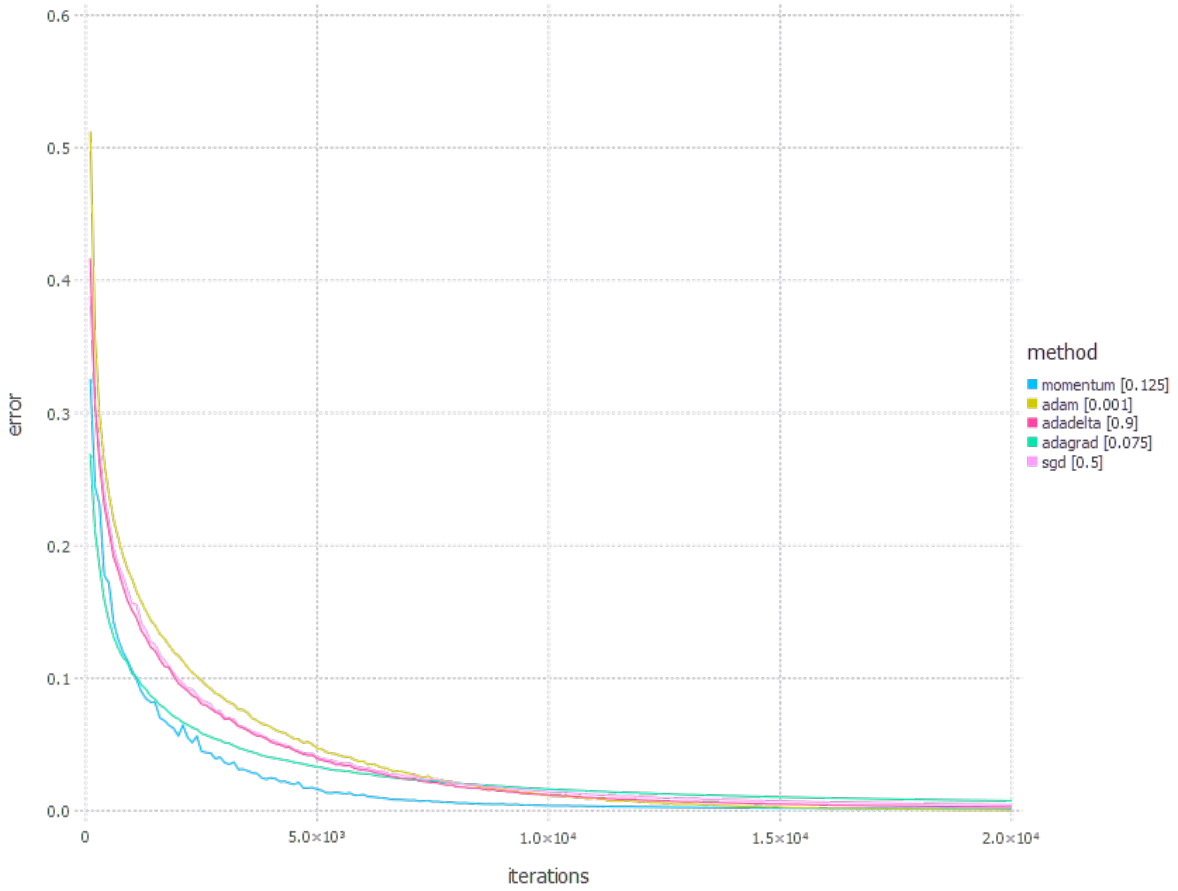
network with linear layer and softmax output (softmax classification)

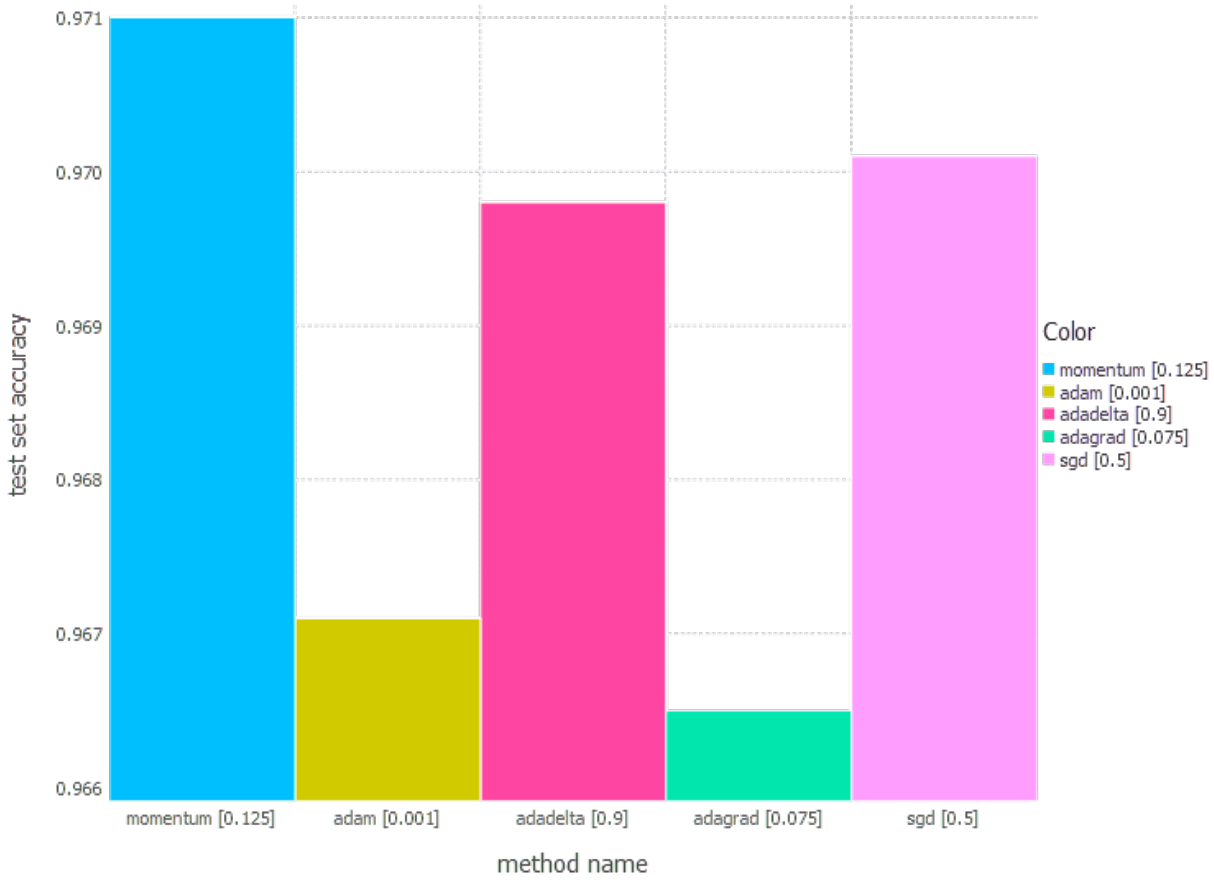




Results on net 2

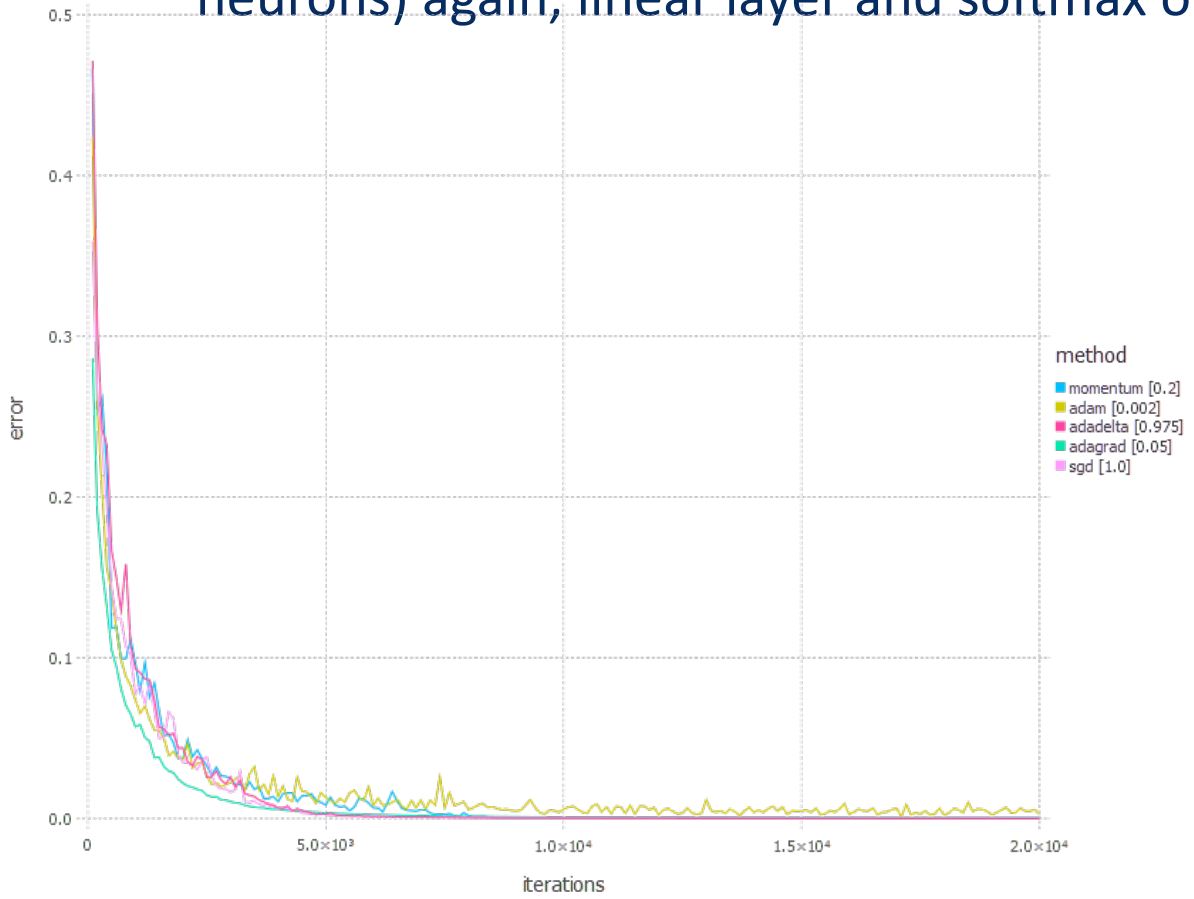
network with sigmoid layer (100 neurons), linear layer and softmax output

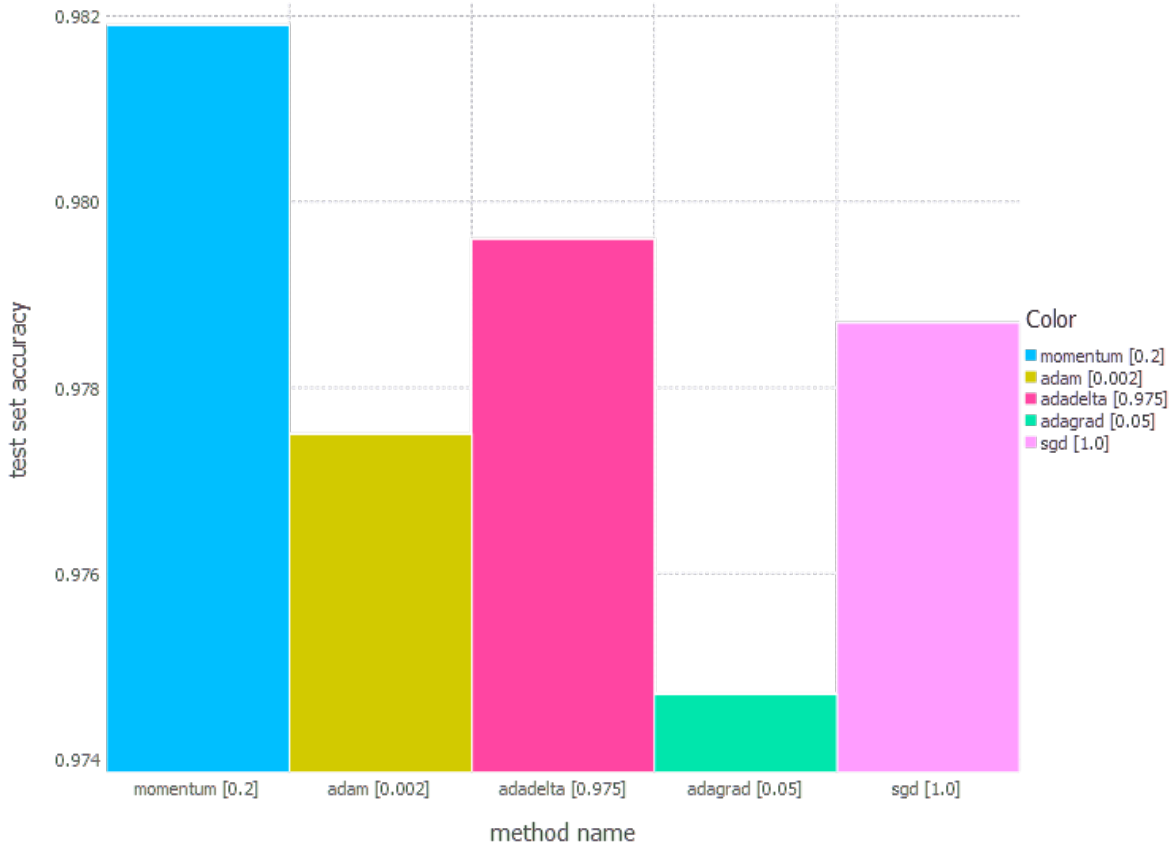




Results on net 3

network with sigmoid layer (300 neurons), ReLU layer (100 neurons), sigmoid layer (50 neurons) again, linear layer and softmax output





So….Which optimizer to use?

- Unfortunately there isn't a clear answer
- As in the toy example before, different networks have completely different behaviors
- If your input data is **sparse**…?
  - You likely achieve the best results using one of the **adaptive learning-rate** methods
  - An additional benefit is that you will not need to tune the learning rate
- Which one is the best?
  - Adagrad, Adadelta, and Adam are very similar algorithms that do well in similar circumstances
  - Insofar, **Adam** might be the best overall choice as trade off between time and accuracy
- Interestingly, many recent papers use SGD without momentum and a simple learning rate annealing schedule
  - SGD usually achieves to find a minimum but it takes much longer time than others, is much more reliant on a robust initialization and annealing schedule

  → If you care about **fast convergence** and train a deep or complex NN, you should choose one of the **adaptive learning** rate methods

Comments on optimizer

- All the optimizers takes in the parameters and gradients of dimension d and manipulate over them.
- None of the optimizers are theoretically proven to have better convergence rate than SGD, except Nesterov momentum.
- Even Nesterov momentum is only proved to work on strongly convex problems.

Can we create a ‘Super-optimizer’ ?

- Evolutionary Stochastic Gradient Descent for Optimization of Deep Neural Networks (Cui et al., NeurIPS’2018)
  - https://papers.nips.cc/paper/7844-evolutionary-stochastic-gradient-descent-for-optimization-of-deep-neural-networks.pdf

Algorithm 1: Evolutionary Stochastic Gradient Descent (ESGD)

Input: generations K, SGD steps K_s, evolution steps K_v, parent population size μ, offspring population size λ and elitist level m.

Initialize population $\Psi_\mu^{(0)} \leftarrow \{\theta_1^{(0)}, \cdots, \theta_\mu^{(0)}\}$;

// K generations

for $k = 1 : K$ **do**

  Update population $\Psi_\mu^{(k)} \leftarrow \Psi_\mu^{(k-1)}$;

  // in parallel

  for $j = 1 : \mu$ **do**

    Pick an optimizer $\pi_j^{(k)}$ for individual $\theta_j^{(k)}$;

    Select hyper-parameters of $\pi_j^{(k)}$ and set a learning schedule;

    // K_s SGD steps

    for $s = 1 : K_s$ **do**

      SGD update of individual $\theta_j^{(k)}$ using $\pi_j^{(k)}$;

      If the fitness degrades, the individual backs off to the previous step $s-1$.

    end

  end

  // K_v evolution steps

  for $v = 1 : K_v$ **do**

    Generate offspring population $\Psi_\lambda^{(k)} \leftarrow \{\theta_1^{(k)}, \cdots, \theta_\lambda^{(k)}\}$;

    Sort the fitness of the parent and offspring population $\Psi_{\mu+\lambda}^{(k)} \leftarrow \Psi_\mu^{(k)} \bigcup \Psi_\lambda^{(k)}$;

    Select the top m ($m \leq \mu$) individuals with the best fitness (m-elitist);

    Update population $\Psi_\mu^{(k)}$ by combining m-elitist and randomly selected $\mu - m$ non-m-elitist candidates;

  end

end

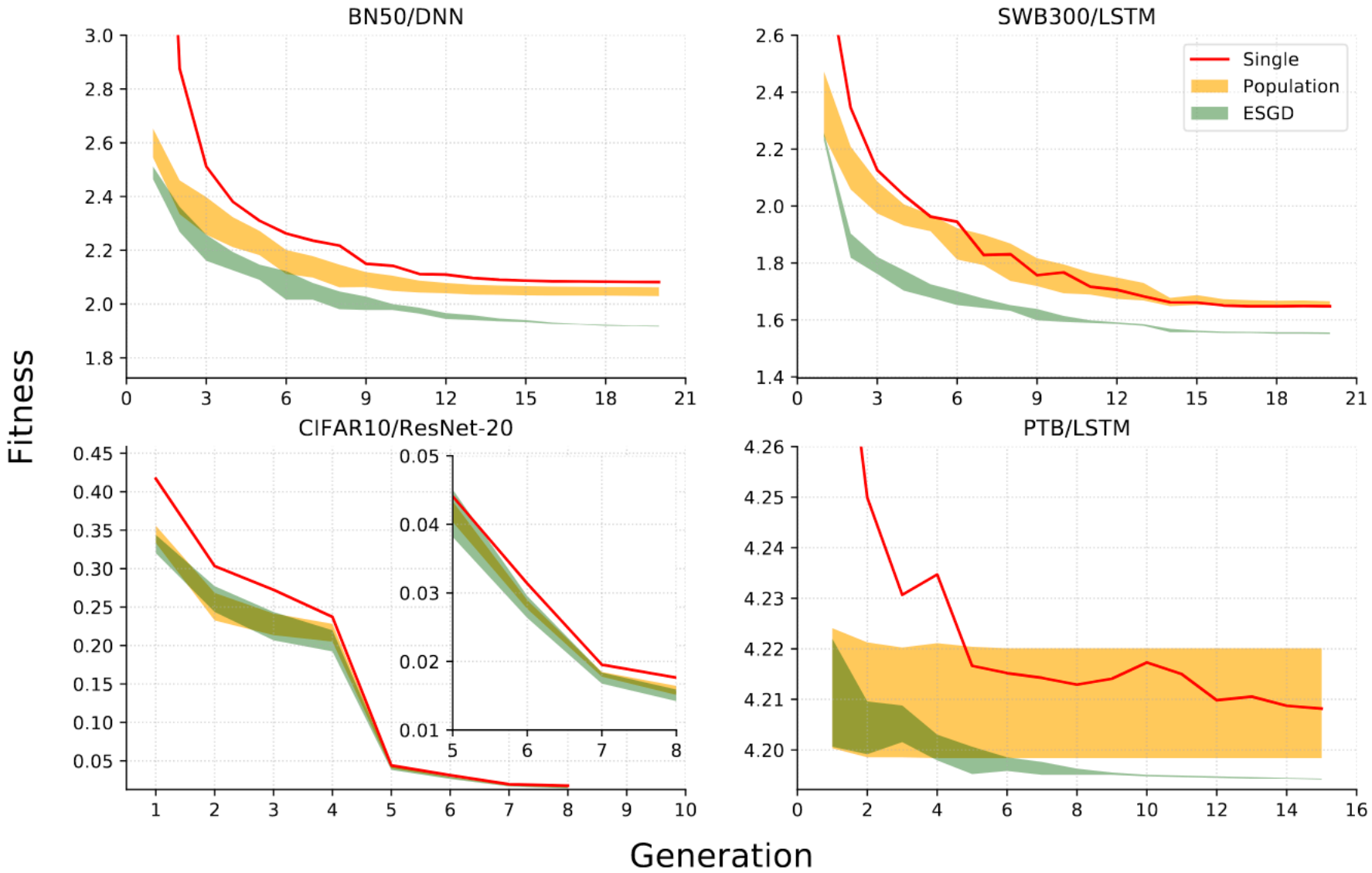

BN50/DNN
SWB300/LSTM
CIFAR10/ResNet-20
PTB/LSTM
Single
Population
ESGD
Fitness
Generation

Table 1: Performance of single baseline, population baseline and ESGD on BN50, SWB300, CIFAR10 and PTB. For ESGD, the tables show the losses and classification error rates of the best individual as well as the top 15 individuals in the population for the first three tasks. In PTB, the perplexities (ppl), which is the exponent of loss, of the validation set and test set are presented. The tables also present the results of the ablation experiments where the evolution step is removed from ESGD.

BN50	loss		WER	
	$\theta_{1:\mu}$	$\theta_{1:\mu} \leftrightarrow \theta_{15:\mu}$	$\theta_{1:\mu}$	$\theta_{1:\mu} \leftrightarrow \theta_{15:\mu}$
single baseline	2.082	–	17.4	–
population baseline	2.029	[2.029, 2.062]	17.1	[16.9, 17.6]
ESGD w/o evolution	2.036	[2.036, 2.075]	17.4	[17.1, 17.7]
ESGD	1.916	[1.916, 1.920]	16.4	[16.2, 16.4]

SWB300	loss		SWB WER		CH WER	
	$\theta_{1:\mu}$	$\theta_{1:\mu} \leftrightarrow \theta_{15:\mu}$	$\theta_{1:\mu}$	$\theta_{1:\mu} \leftrightarrow \theta_{15:\mu}$	$\theta_{1:\mu}$	$\theta_{1:\mu} \leftrightarrow \theta_{15:\mu}$
single baseline	1.648	–	10.4	–	18.5	–
population baseline	1.645	[1.645, 1.666]	10.4	[10.3, 10.7]	18.2	[18.2, 18.8]
ESGD w/o evolution	1.626	[1.626, 1.641]	10.3	[10.3, 10.7]	18.3	[18.0, 18.6]
ESGD	1.551	[1.551, 1.557]	10.0	[10.0, 10.1]	18.2	[18.0, 18.3]

CIFAR10	loss		error rate	
	$\theta_{1:\mu}$	$\theta_{1:\mu} \leftrightarrow \theta_{15:\mu}$	$\theta_{1:\mu}$	$\theta_{1:\mu} \leftrightarrow \theta_{15:\mu}$
single baseline	0.0176	–	8.34	–
population baseline	0.0151	[0.0151, 0.0164]	8.24	[7.90, 8.69]
ESGD w/o evolution	0.0147	[0.0147, 0.0166]	8.49	[7.86, 8.53]
ESGD	0.0142	[0.0142, 0.0159]	7.52	[7.43, 8.10]

PTB	validation ppl		test ppl	
	$\theta_{1:\mu}$	$\theta_{1:\mu} \leftrightarrow \theta_{15:\mu}$	$\theta_{1:\mu}$	$\theta_{1:\mu} \leftrightarrow \theta_{15:\mu}$
single baseline	67.27	–	64.58	–
population baseline	66.58	[66.58, 68.04]	63.96	[63.96, 64.58]
ESGD w/o evolution	67.27	[67.27, 79.25]	64.58	[64.58, 76.64]
ESGD	66.29	[66.29, 66.30]	63.73	[63.72, 63.74]

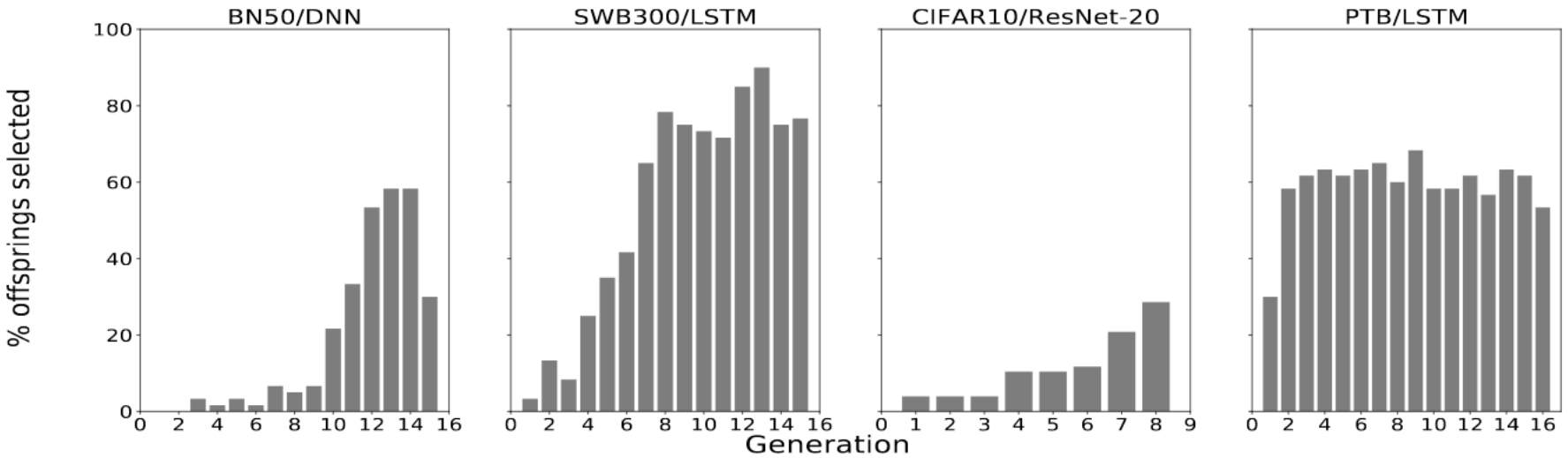


Figure 2: Percentage of offsprings selected in the 60% m-elitist over generations of ESGD in BN50, SWB300, CIFAR10 and PTB.

Table 6: The optimizers selected by the best candidate in the population over generations in ESGD in BN50 DNNs and SWB300 LSTMs.

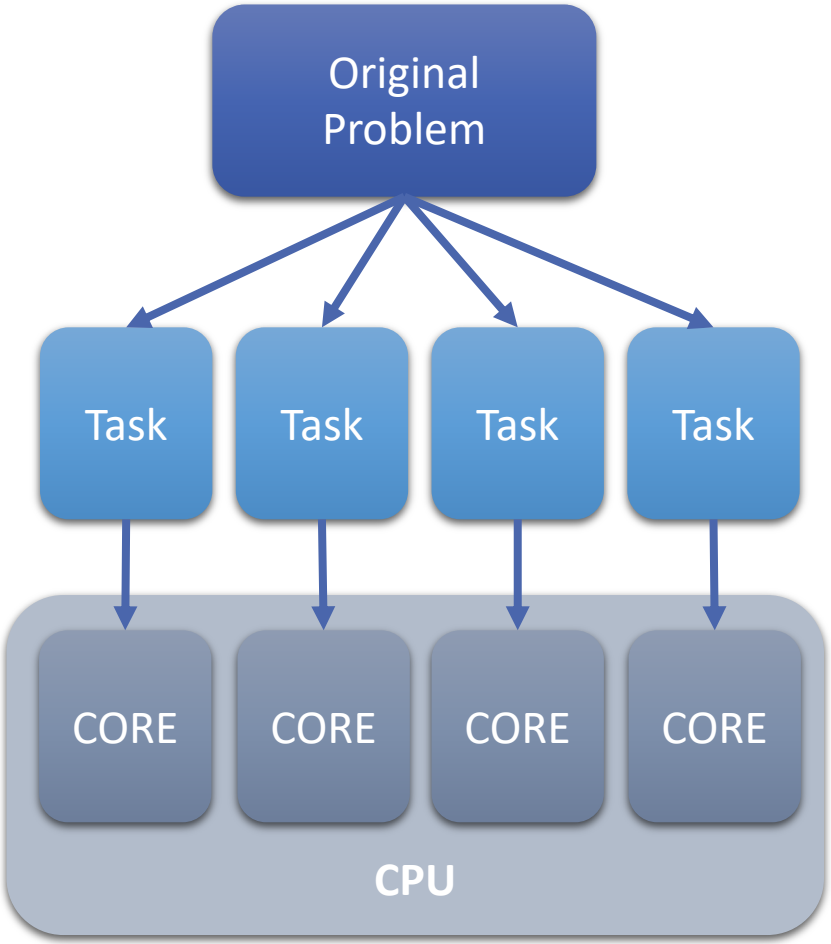
generation	optimizer	
	BN50 DNN	SWB300 LSTM
1	adam, lr=2.68e-4	adam, lr=5.39e-4
2	adam, lr=1.10e-4	adam, lr=4.61e-5
3	adam, lr=1.87e-4	adam, lr=9.65e-5
4	sgd, nesterov=F, lr=3.61e-4, momentum=0	sgd, nesterov=F, lr=9.92e-3, momentum=0.21
5	adam, lr=9.07e-5	sgd, nesterov=T, lr=6.60e-3, momentum=0.35
6	adam, lr=1.04e-4	adam, lr=4.48e-5
7	sgd, nesterov=F, lr=5.20e-4, momentum=0	sgd, nesterov=T, lr=5.60e-3, momentum=0.11
8	sgd, nesterov=T, lr=1.00e-4, momentum=0.44	sgd, nesterov=T, lr=5.15e-3, momentum=0.49
9	adam, lr=6.45e-5	adam, lr=3.20e-5
10	adam, lr=7.51e-5	adam, lr=2.05e-5
11	adam, lr=4.51e-5	adam, lr=1.97e-5
12	sgd, nesterov=F, lr=4.19e-5, momentum=0	adam, lr=2.98e-5
13	sgd, nesterov=F, lr=6.17e-5, momentum=0.25	adam, lr=1.94e-5
14	sgd, nesterov=F, lr=6.73e-5, momentum=0.45	adam, lr=1.33e-5
15	sgd, nesterov=F, lr=1.00e-4, momentum=0	adam, lr=1.45e-5

Complementary optimizers In each generation of ESGD, an individual selects an optimizer from a pool of optimizers with certain hyper-parameters. In most of the experiments, the pool of optimizers consists of SGD variants and ADAM. It is often observed that ADAM tends to be aggressive in the early stage but plateaus quickly. SGD, however, starts slow but can get to better local optima. ESGD can automatically choose optimizers and their appropriate hyper-parameters based on the fitness value during the evolution process so that the merits of both SGD and ADAM can be combined to seek a better local optimal solution to the problem of interest. In the supplementary material examples are given where we show the optimizers with their training hyper-parameters selected by the best individuals in ESGD in each generation. It indicates that over generations different optimizers are automatically chosen by ESGD giving rise to a better fitness value.

PyTorch Multiprocessing

Why Multiprocessing

- **CPU** has many cores and hardware threads
- **When a problem can be parallelized (i.e. divided in parallel tasks) use parallel tasks to complete it in less time**
- Examples of parallelizable ML tasks:
  - Load multiple files from disk
  - Applying transformations to each dataset sample
  - Send/Receive data from Multiple GPUs



Python Threading

```
class threading.Thread(group=None,target=None,name=None,args=(),kwargs={},*,daemon=None)
```

- Allows creation and handling of *Thread* objects: independent execution context
- Threads share data unless is thread local:
  - *mydata = threading.local()*
- Threads in Python:
  - **CONCURRENCY** but **NOT PARALLELISM** (global interpreter lock)
  - Memory is shared unless is specifically thread local
- Daemon threads:
  - They do not die with the parent (wait for interpreter shutdown)

Not useful for Parallelism Performance!

See https://docs.python.org/3.6/library/threading.html#module-threading

Python Multiprocessing

- *multiprocessing.Process* class: create another python interpreter process
  - 1) SPAWN method: start new fresh process with minimal resource inherited
    - Slower, Default on Windows
  - 2) FORK method: fork() a new process and inherit all resources (files, pipes, etc.)
    - Faster, Default on Unix
    - Can be unstable or incompatible (need to try)
  - 3) FORK-SERVER method: a server-process is created that forks the new process
    - keeps the original process safer
    - Minimal set of resources inherited
  - In general SPAWN and FORK-SERVER methods are safer but slower

  - **Creating a process is much slower and heavy than creating a thread**
  - https://docs.python.org/3.6/library/multiprocessing.html#multiprocessing-programming

Python Multiprocessing and Pool examples

- Example: creation of a *process*
- Example: creation of a *pool*

```
from multiprocessing import Process, Pool

def f(name):
    print('hello', name)

def f(x):
    return x*x

if __name__ == '__main__':
    p = Process(target=f, args=('bob',))
    p.start()
    p.join()

    with Pool(5) as p:
        print(p.map(f, [1, 2, 3]))
```

PyTorch *torch.multiprocessing*

- Replaces standard Python *multiprocessing*
  - Provides ability to send tensor*s efficiently*
- Why is it difficult or not efficient to send *tensors* among Python processes?
  - It is inefficient to send tensors because they need to be serialized before being sent to another process (pickle class), since the address space is different (cpython pointers cannot be passed)
  - Serializing and sending Tensors also implies a memory copy
    - This is called deep copy
- High Performance Solution: *multiprocessing.Queue*
  - Copy and Serialize tensors in a *shared memory area* and only pass handles
  - Additional performance improvement: *multiprocessing.Queue* multiple threads to serialize and send objects (lots of work!)
- High Performance tip: **try to reuse buffers in *multiprocessing.Queue !***
  - See https://pytorch.org/docs/master/notes/multiprocessing.html#reuse-buffers-passed-through-a-queue

Shared Memory

- **Shared memory** is a memory area that the OS (eg Linux) maps on the address space of the processes, allowing in this way to be simultaneously accessed by multiple programs with an intent to provide communication among them or avoid redundant copies.
  - It is an efficient means of passing data between programs
- **PyTorch HP tip**: Remember that each time you put a Tensor into a multiprocessing.Queue, it has to be moved into shared memory. If it's already shared, it is a no-op, otherwise it will incur an additional memory copy that can slow down the whole process. Even if you have a pool of processes sending data to a single one, make it send the buffers back - this is nearly free and will let you avoid a copy when sending next batch.
  - from: https://pytorch.org/docs/stable/notes/multiprocessing.html
- Do not confuse this shared memory used as communication between OS processes with the concept of the shared memory in the GPU that is a HW component to reduce GPU memory access latency
  - see also: https://devblogs.nvidia.com/using-shared-memory-cuda-cc/

Multi-processing in real-life

- **GaDei: On Scale-up Training As A Service For Deep Learning (Zhang et al. ICDM'2017)**
  - **https://arxiv.org/pdf/1611.06213.pdf**

Deep Learning Training-as-a-Service (TaaS)

Deep learning Training-as-a-Service (TaaS) has unique requirements

- Satisfy a wide range of workload
- Serve customers with no experience/resource to tune hyper-parameters

IBM Watson Natural Language Classification (NLC) TaaS service

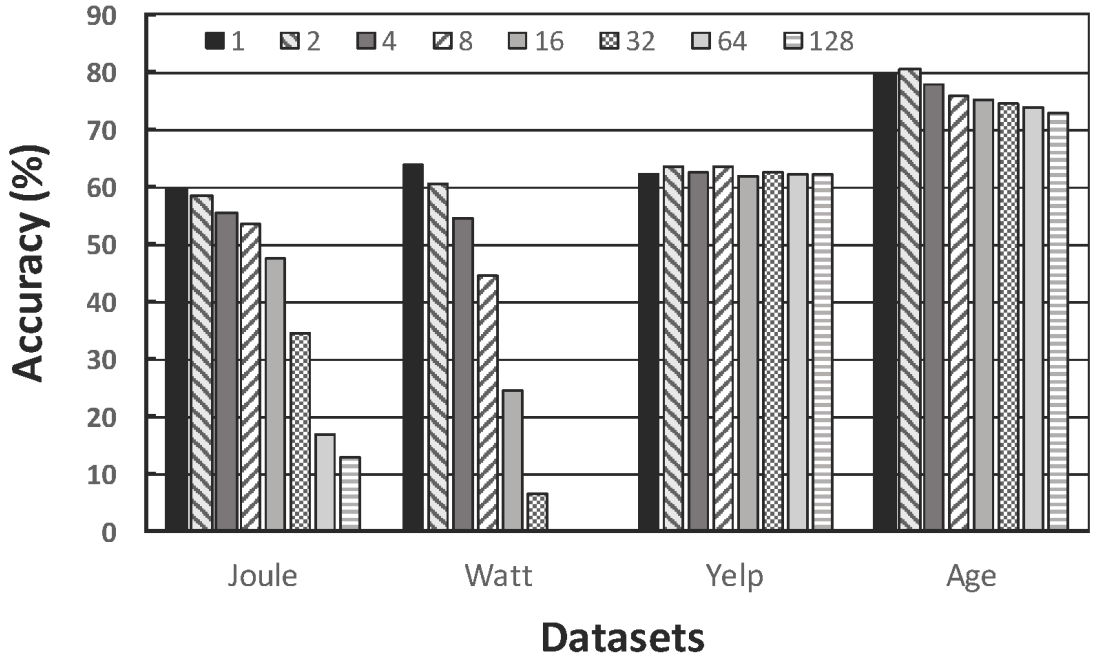
- Classify input sentences into categories with predefined labels
  - Applications: sentiment analysis, topic classification, question classification, author profiling, etc.
- Thousands of enterprise-level customers globally
  - Customers usually use the service as it is w/o parameter tunings
- No.1 most used IBM Watson service

TaaS Hyper-parameters are usually fixed and should be applicable to all users

NLC Observation: Batch Size vs. Accuracy

- **Large batch size introduces significant accuracy loss**

How do different batch sizes affect the accuracy in IBM Watson NLC tasks?



- DL method is on average 3%-6% more accurate than less computation intensive methods (e.g. SVM)
- Accuracy loss larger than 3%-6% will invalidate using DL method for NLC tasks

Tip!

LESSONS LEARNED

small batch size

NLC Observation: Communication Frequency vs. Accuracy

- **Low communication frequency decreases model accuracy**

How different communication frequency affect the accuracy in IBM Watson NLC tasks

workload	Accuracy		GPU Utilization	
	communication interval = 1	communication interval = 8	communication interval = 1	communication interval = 8
Joule	57.70%	39.80%	7.90%	35.80%
Watt	57.80%	55.10%	4.90%	29.40%
Age	74.10%	63.60%	4.00%	24.90%
MR	77.20%	50.00%	5.50%	33.70%
Yelp	56.90%	20.00%	8.80%	42.90%

LESSONS LEARNED

high comm. freq.

- **Existing research mainly focuses one specific dataset and model**
  - Usually heavy parameter tuning is required

Dataset	CIFAR	ImageNet
Model	VGG	AlexNet
Mini-batch Size	128	256
Number of epochs	100	40
Learning Rate	[Epoch 001 - 025] 1 [Epoch 026 - 050] 0.5 [Epoch 051 - 075] 0.25 [Epoch 075 - 100] 0.125	[Epoch 01 - 18] 0.01 [Epoch 19 - 30] 0.005 [Epoch 31 - 40] 0.001

Dataset Size	small	medium	large
Model	NLC		
Mini-batch Size	2	4	32
Number of epochs	200		
Learning Rate	0.01		

- **TaaS h[illegible] wide range of customers**
  - TaaS users usually don't have expertise or resource for parameter tuning
  - Minimizing the training time has become the top priority for IBM Watson NLC Service
  - Small batch size, high communication frequency and small learning rate are crucial to satisfy different costumers

Research questions

- **What is the right hyper-parameter setup in TaaS to counter staleness and guarantee convergence?**

- **What is the unique system design challenges for TaaS thereof and how to address them ?**

Minimum Bandwidth Requirement for Any Speedup

- Extremely high bandwidth requirement

Mini-batch size	Joule		Watt		Age		Yelp		CIFAR		ImageNet	
	2.46K, 7.69MB		7K, 20.72MB		68.48K, 72.86MB		500K, 98.60MB		50K, 59.97MB		1280K, 244.48MB	
	TPE (sec)	RB (GB/s)	TPE (sec)	RB (GB/s)	TPE (sec)	RB (GB/s)	TPE (sec)	RB (GB/s)	TPE (sec)	RB (GB/s)	TPE (sec)	RB (GB/s)
1	5.61	7.00	25.22	11.50	574.80	17.36	4376.12	22.53	n/a	n/a	n/a	n/a
2	3.22	6.10	14.11	10.28	309.51	16.12	2367.04	20.83	792.45	3.78	26957.54	11.61
4	1.77	5.54	7.46	9.72	169.99	14.68	1299.42	18.97	502.63	2.98	15596.41	10.03
8	1.00	4.89	4.06	8.93	104.39	11.95	802.04	15.37	356.46	2.10	9499.38	8.24
16	0.75	3.27	2.79	6.49	77.79	8.02	571.10	10.79	290.95	1.29	7294.30	5.36
32	0.72	1.69	2.38	3.82	69.52	4.49	451.61	6.82	261.23	0.72	5769.76	3.39
64	0.80	0.76	2.37	1.91	77.14	2.02	403.69	3.82	245.68	0.38	5014.85	1.95
128	1.00	0.31	2.77	0.82	100.22	0.78	402.97	1.91	234.38	0.20	4784.22	1.02

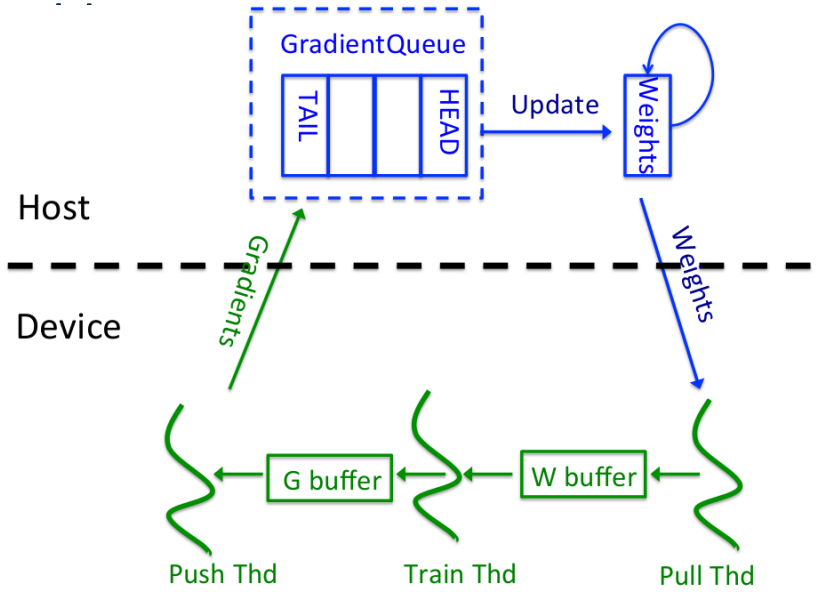
For a TaaS system requiring mini-batch size

- A scale-out solution is unlikely to work due to the network bandwidth limits
- Existing scale-up solutions are insufficient as large mini-batch is NOT allowed

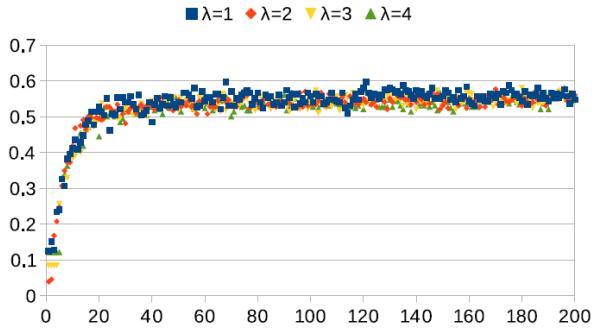
Our solution

- **GaDei: a scale-up solution designed for TaaS with mini-batch size requirement**
  - Communications via minimum memory copies
  - Hogwild! style weights update rule
  - On-device double buffering with GPU multi-streaming for better performance

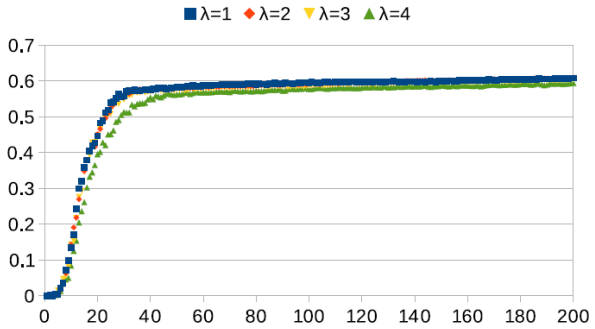
- **Characteristics**
  - Independent from the underlying gradient-calculation b blocks
  - Model convergence guarantee
  - Deadlock-free
  - Fault tolerance



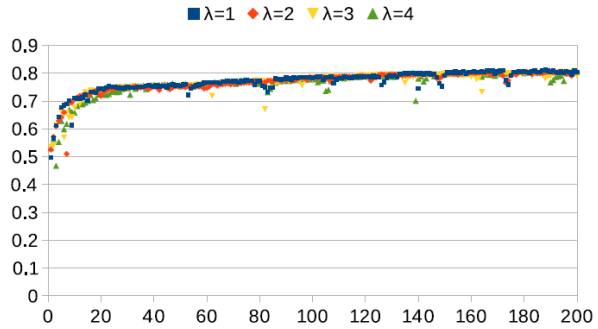
Evaluation: Convergence Result



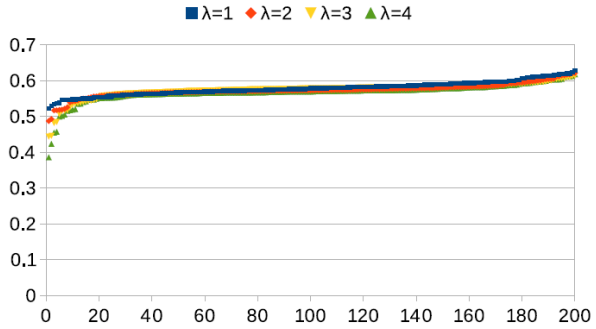
(a) Joule accuracy



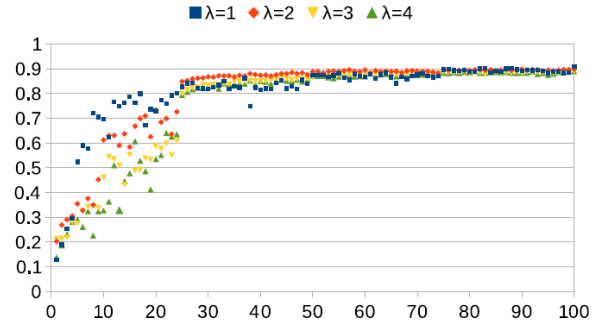
(b) Watt accuracy



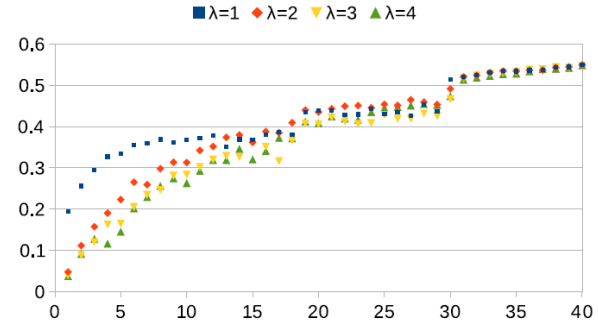
(c) Age accuracy



(d) Yelp accuracy

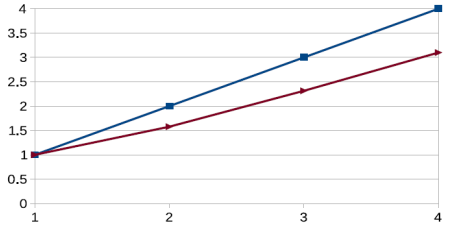


(e) CIFAR accuracy

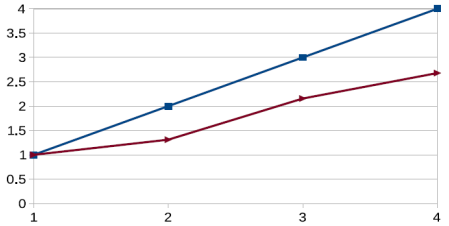


(f) ImageNet accuracy

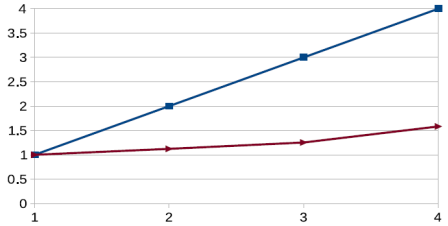
Evaluation: Speedup and Memory Analysis



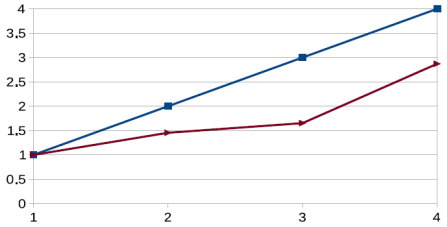
(a) Joule speedup



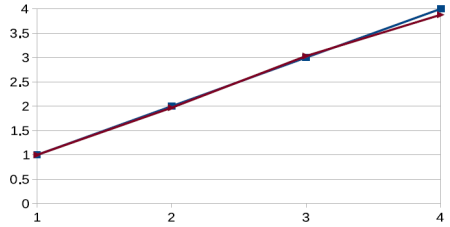
(b) Watt speedup



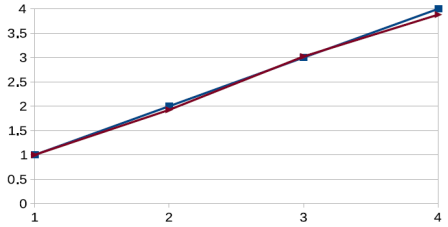
(c) Age speedup



(d) Yelp speedup



(e) CIFAR speedup

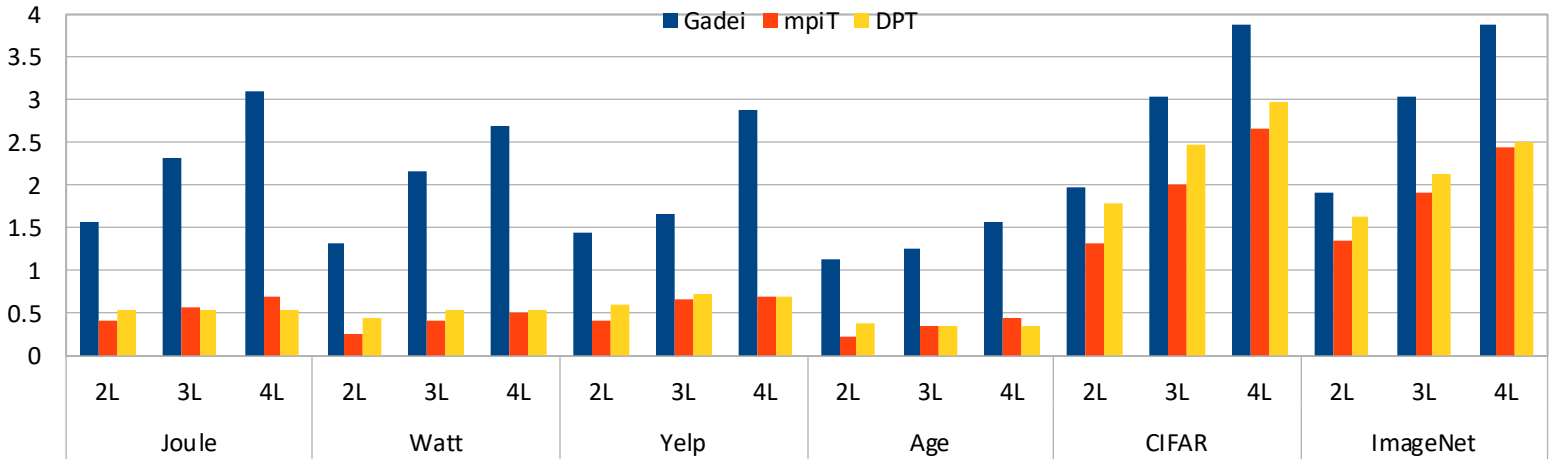


(f) ImageNet speedup

workload	Memory Bandwidth Utilized Rate (GB/s)		
	2 learners	3 learners	4 learners
Joule	18.56	27.25	36.48
Watt	26.93	44.33	55.07
Age	32.94	36.76	46.42
Yelp	19.93	22.64	39.34
CIFAR	0.78	1.21	1.45
ImageNet	3.98	6.25	8.02

- **NLC workload**: 1.5x -- 3x speedup, 36-55GB/s bandwidth (close to hardware limit)
- **Non-TaaS workload**: near linear speedup

Evaluation: Comparison with DPT and mpiT



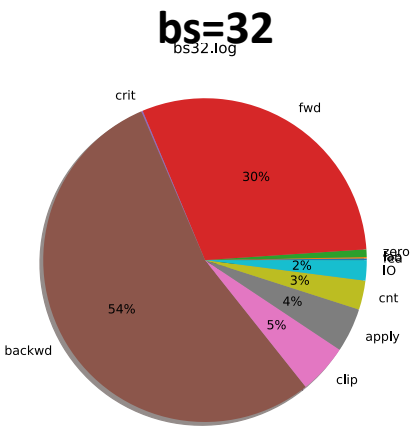
Minimum Mini-batch Size While Still Achieving Speedup			
workload	**GaDei**	**mpiT**	**DPT**
Joule	1	16	32
Watt	1	16	64
Age	1	32	32
Yelp	16	64	128
CIFAR	2	2	8
ImageNet	2	8	8

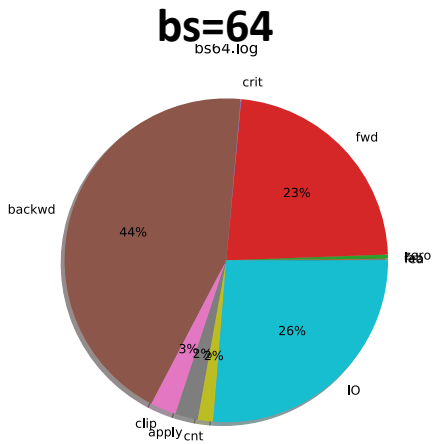
DPT– synchronous approach
mpiT – asynchronous approach (via MPI)

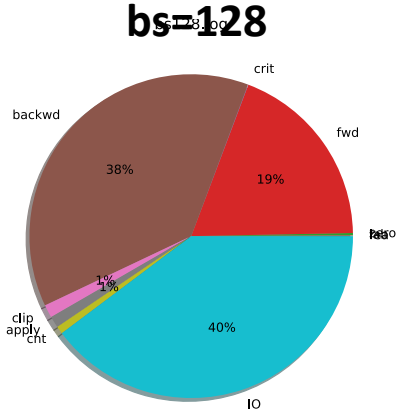
PyTorch Data loading and Preparation

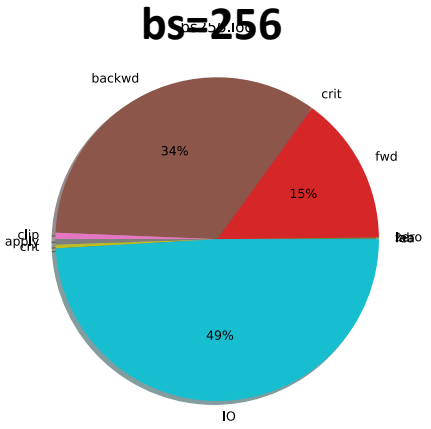
I/O could be a big problem

- As GPU gets faster, I/O becomes a bottleneck









I/O is as expensive as computation!

Loading data - *torch.utils.data.Dataset*

- *torch.utils.data.Dataset* represents a dataset (abstract class)

- Create your own class and override the following:
  - *__len__* so that len(dataset) returns the size of the dataset.
  - *__getitem__* to support the indexing such that dataset[i] can be used to get i-th sample

  - *__getitem__* will be called to load 1 item at a time

Custom Dataset Class Example

- Define a set of transformation primitives for each sample
- Pass the transformation primitives through the *transform* argument

```
class FaceLandmarksDataset(Dataset):
    """Face Landmarks dataset."""

    def __init__(self, csv_file, root_dir, transform=None):
        """
        Args:
            csv_file (string):
                Path to the csv file with annotations.
            root_dir (string):
                Directory with all the images.
            transform (callable, optional):
                Optional transform to be applied on a sample.
        """
        self.landmarks_frame = pd.read_csv(csv_file)
        self.root_dir = root_dir
        self.transform = transform
```

from: http://pytorch.org/tutorials/beginner/data_loading_tutorial.html#dataset-class

Custom Dataset Class Example

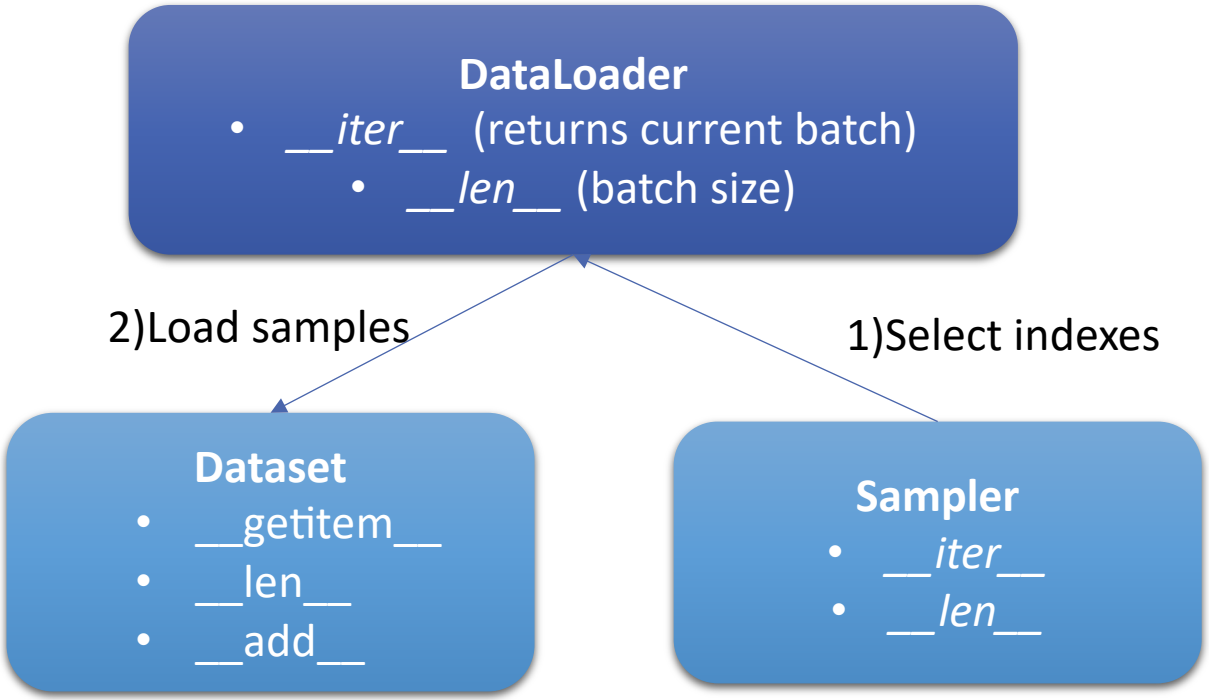
- Transform primitives are applied in __getitem__ to each sample after loading it with *io.imread()*

```
def __len__(self):
        return len(self.landmarks_frame)

def __getitem__(self, idx):
    img_name = os.path.join(self.root_dir,
                            self.landmarks_frame.iloc[idx, 0])
    image = io.imread(img_name)
    landmarks = self.landmarks_frame.iloc[idx, 1:].as_matrix()
    landmarks = landmarks.astype('float').reshape(-1, 2)
    sample = {'image': image, 'landmarks': landmarks}
    if self.transform:
          sample = self.transform(sample)

    return sample
```

from: http://pytorch.org/tutorials/beginner/data_loading_tutorial.html#dataset-class

Loading data - *torch.utils.data.Dataloader*

- Selects and Loads data from the Dataset
- Composed of a **Dataset** and a **Sampler**

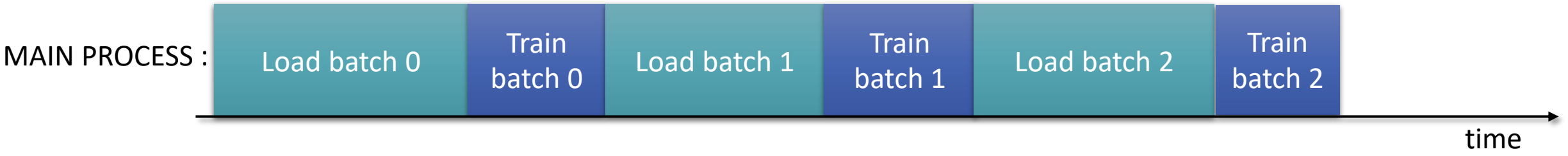


Loading data - *torch.utils.data.Dataloader*

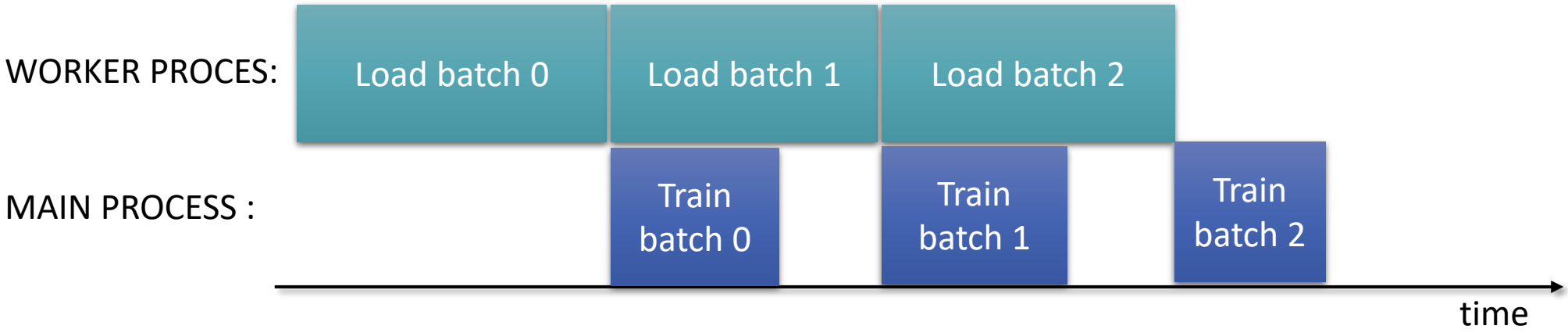
- Useful Parameters:
  - *batch_size:* batch size
  - *sampler:* define which sampler to use
  - *pin_memory:* copy into CUDA pinned memory before returning the data
  - *shuffle:* reshuffle data at each epoch
- Available *samplers:*
  - *SequentialSampler*
  - *RandomSampler*
  - *SubsetRandomSampler*
  - *WeighetRandomSampler*
  - Create your *CustomSampler*

Data Prefetching

- Normal pipeline:



- Pipeline with Load Prefetching:



Load time still limiting factor. Can we do better?

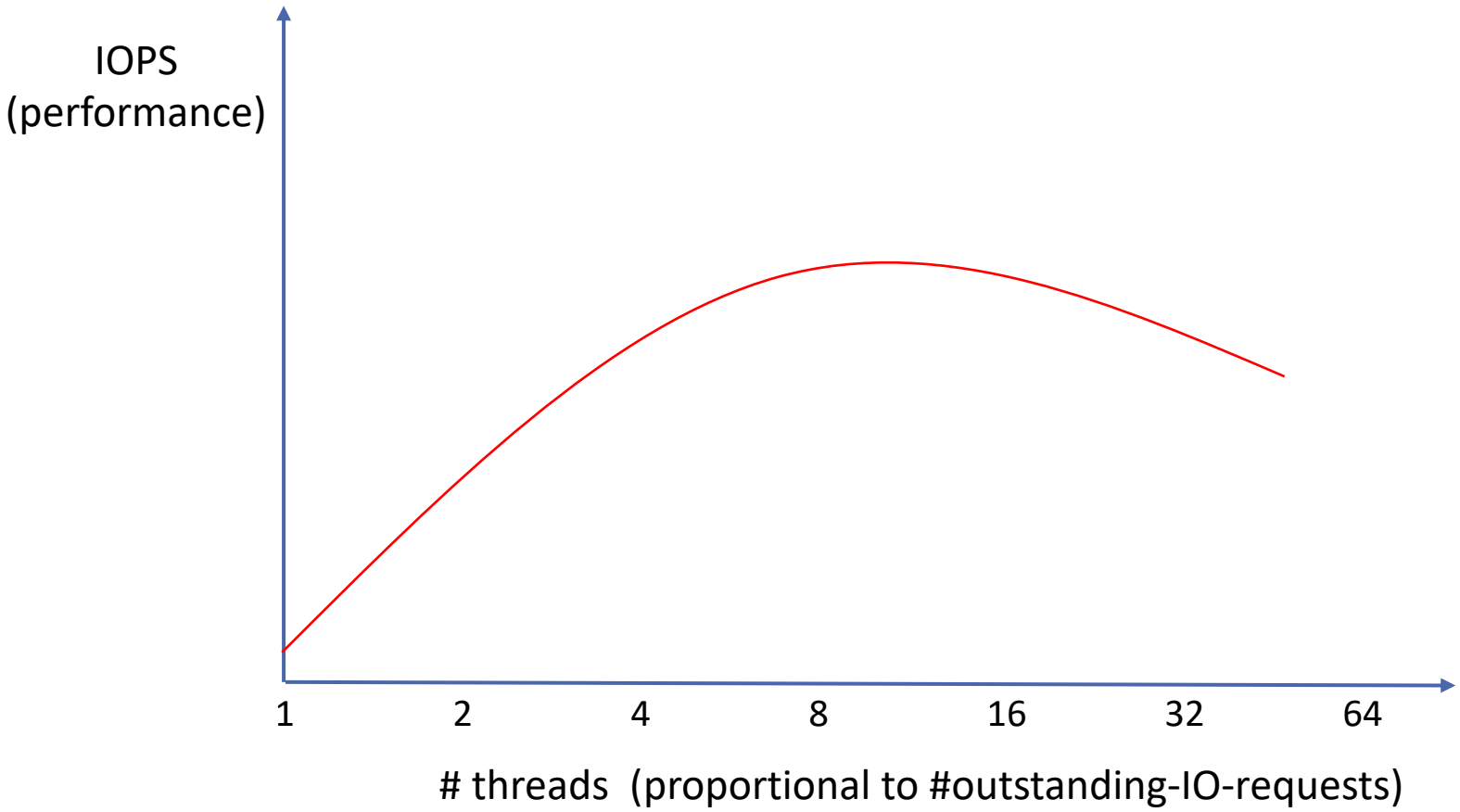
Disk Performance Characteristics

- IOPS: IO-ops/sec
  - read/write
  - random/sequential
- Bandwidth = IOPS * BlockSize [bytes/sec]
  - Block size: 4KB+

- Queue Depth: maximum number of Outstanding IO requests in a device
- **Important fact: IOPS can be higher with multiple I/O outstanding requests**

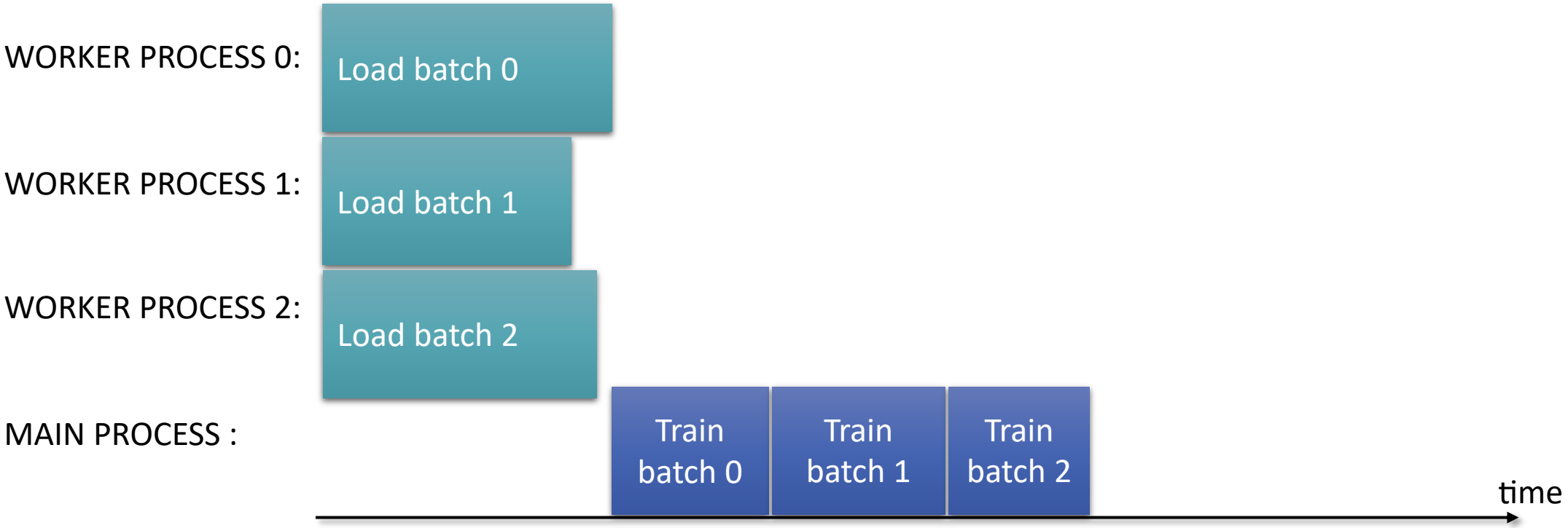
https://en.wikipedia.org/wiki/IOPS

Disk Performance Characteristics



Multi-threaded Data Prefetching

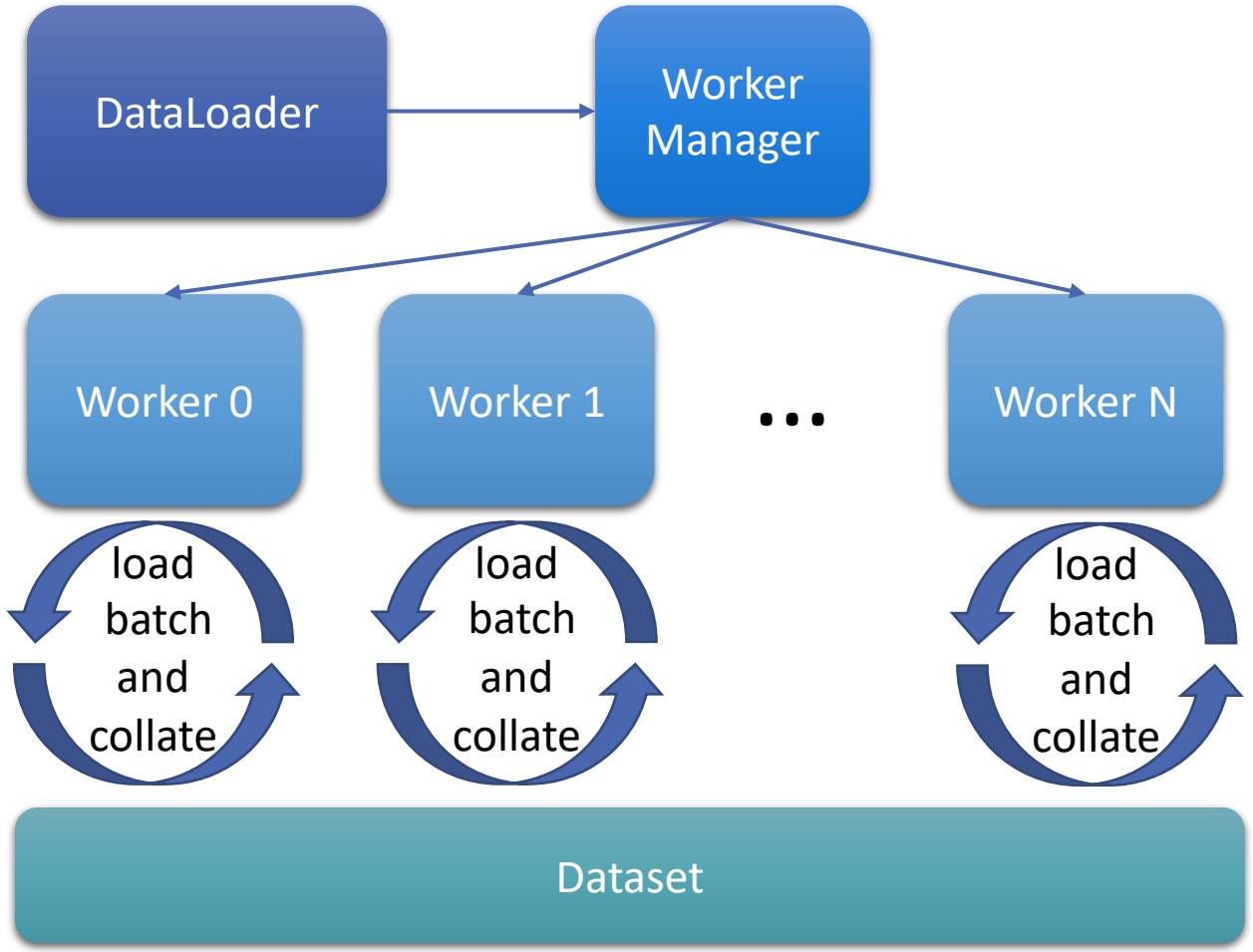
- Issue many outstanding I/O requests using multiple threads:



- Prefetching limitations?

torch.utils.data.Dataloader Architecture

- DataLoader (main process):
  - Requests a set of batches based on *sampler*
- WorkerManager (thread):
  - Dispatch batches to workers
- Workers (processes):
  - Load all samples in each batch
  - Collate batches:
    - Build a batch Tensor with samples
- Synchronization:
  - Producer/Consumer using shared queues and Signals/Exceptions



Example: Images loading and CNN training

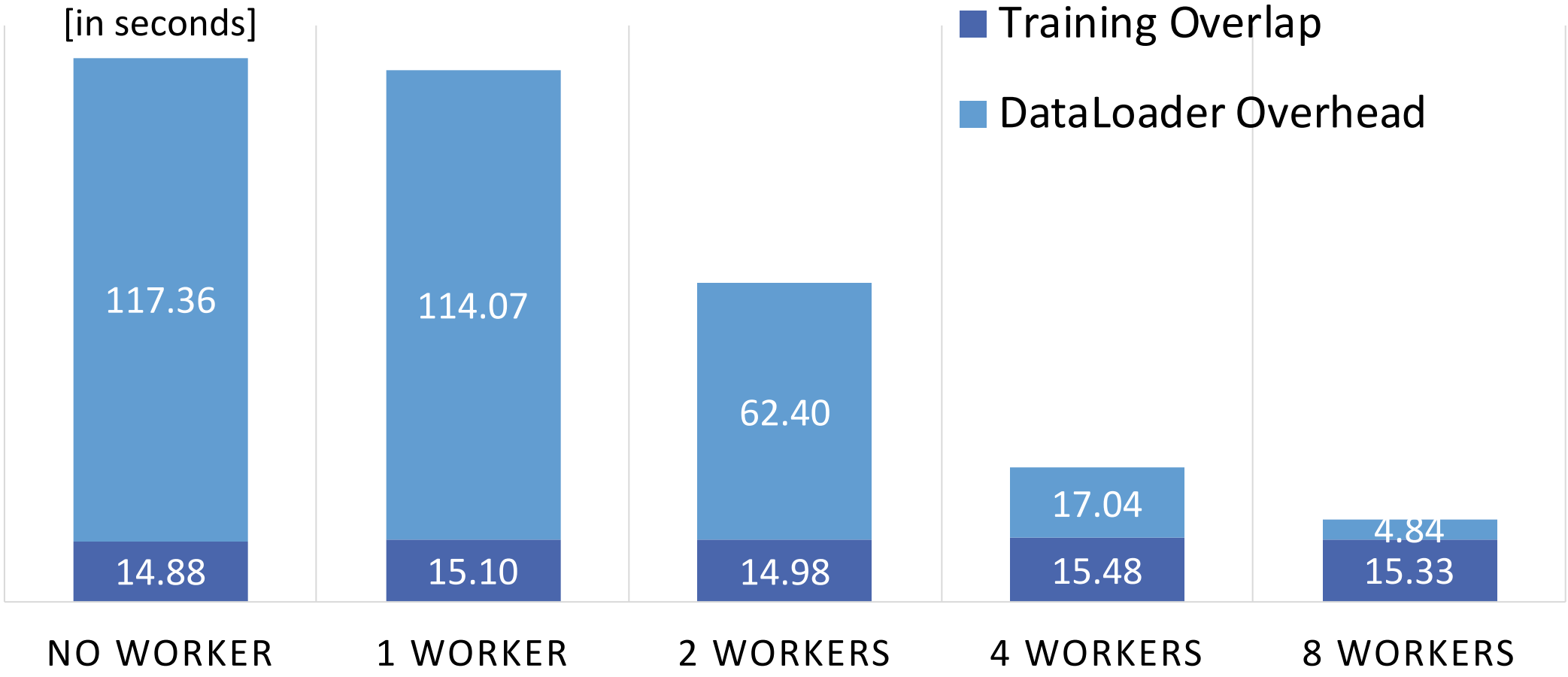
- dataset = ucf101 (video dataset)
- Load images with DataLoader + CNN Network training
- batch_size = 32, num_workers = 0 and 4
- I/O: NFS over ethernet 100 Gbit
  - Several seconds per minibatch
  - Most time spent in __getitem__ of data loader iterator
    - ReadSegmentFlow: 18 ~ 600ms
    - video_transform: 13 ~ 27 ms
- How many workers is optimal?

- Nvprof - num_workers = 0

- Nvprof num_workers =4

Fewer gaps, not completely eliminated

Example: Number of workers effect



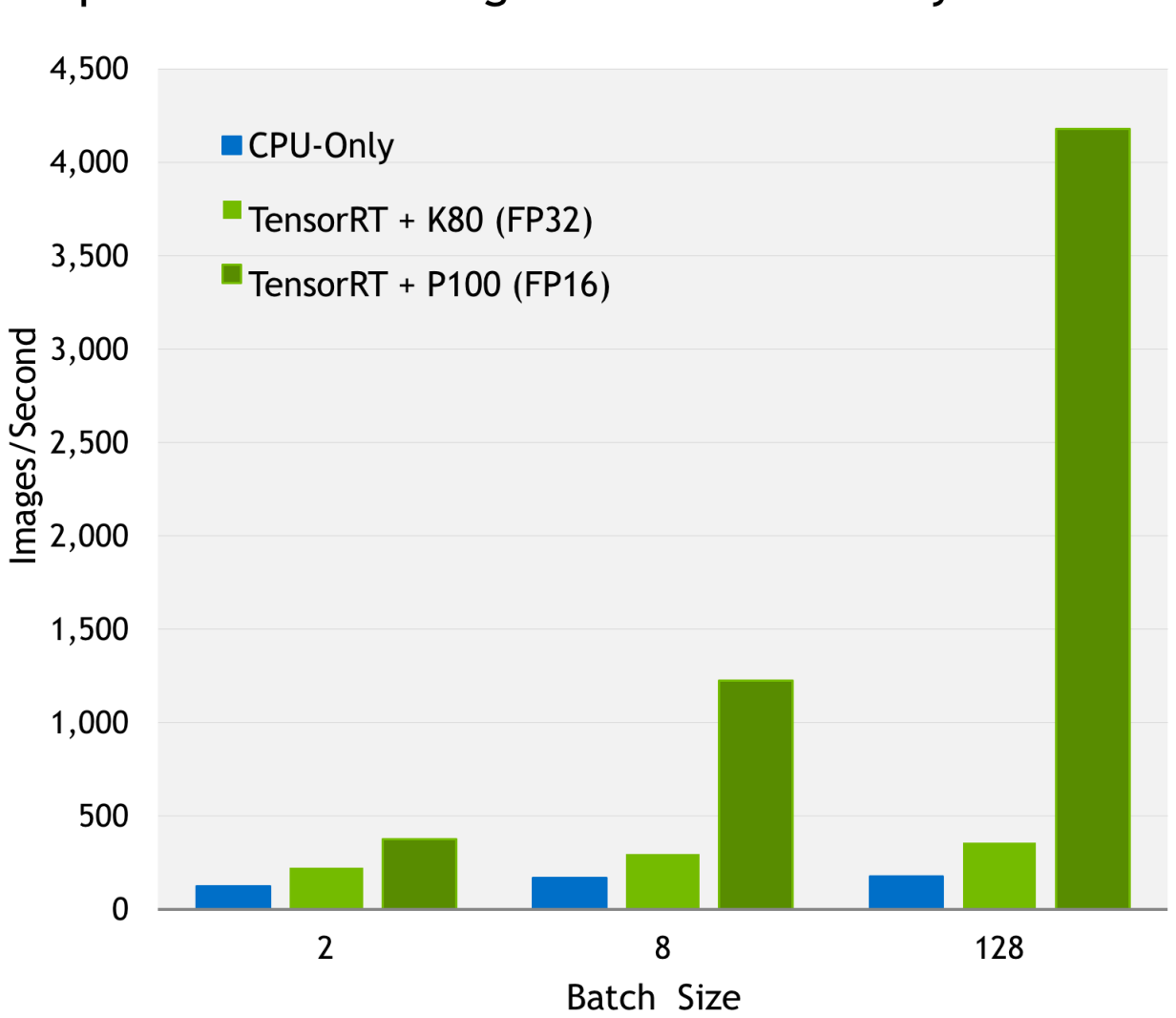
Ideal speed-up vs actual speed-up:

- The 8x speedup should be (114.07 + 15.10) / 8 = 16.15.
- The ideal overhead with 8 workers would be 16.15 - 15.33 = 0.82, but the actual measured is 4.84

PyTorch CUDA

Why CUDA - Inference

- Intel Xeon E5-2690v4 CPU
- Tesla P100 GPU



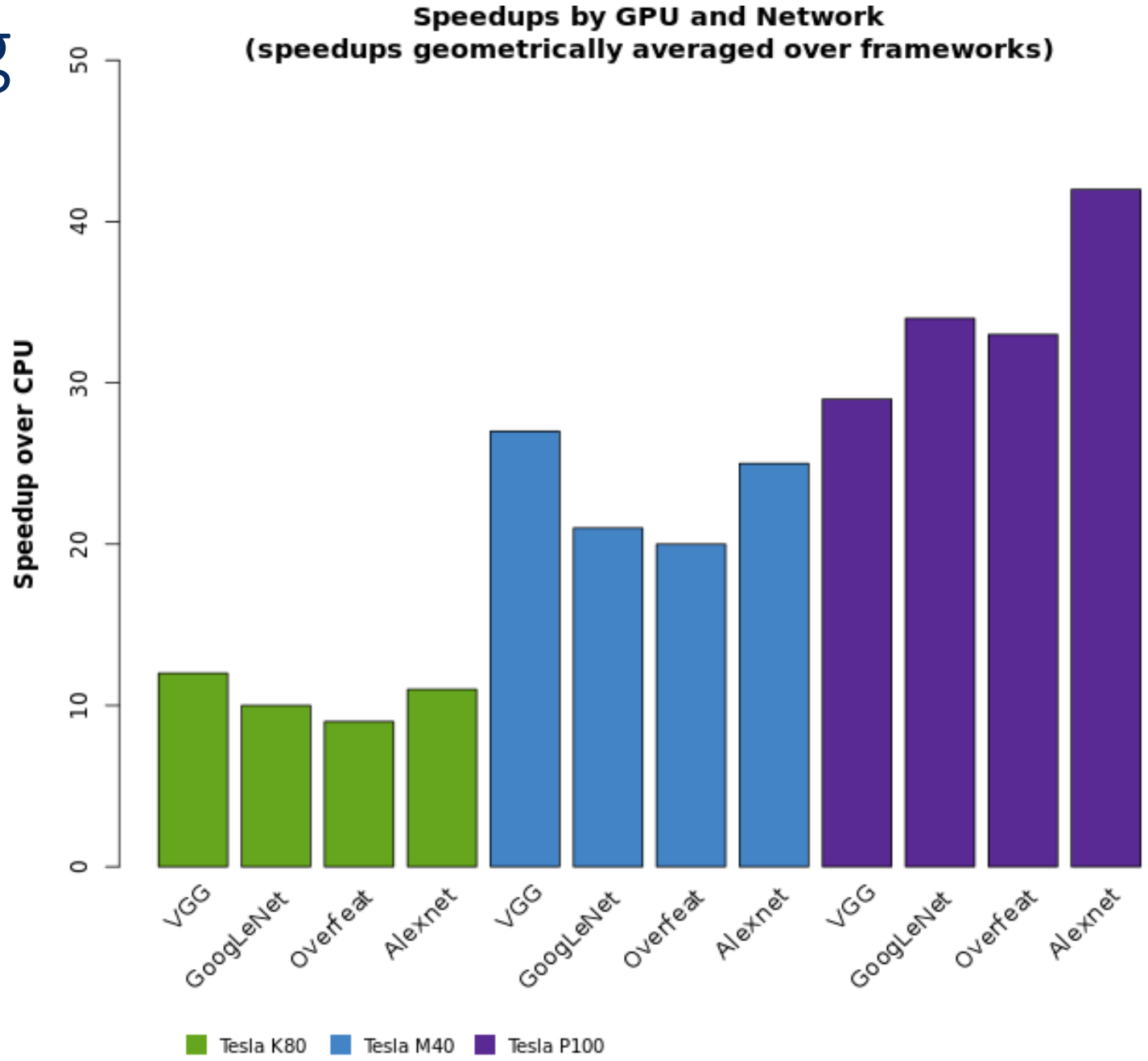
GoogLeNet, Tesla P100 + TensorRT (FP16), Tesla K80 + TensorRT (FP32), CPU-Only + Caffe (FP32)
CPU: 1 Socket Broadwell E5-2690 v4@2.6GHz with HT off

from: NVIDIA

From: https://devblogs.nvidia.com/deploying-deep-learning-nvidia-tensorrt/

Why CUDA - Training

- Intel Xeon E5-2690v4 CPU
  - 256GB of DRAM
- Tesla K80 GPU
- Tesla M40 GPU
- Tesla P100 GPU



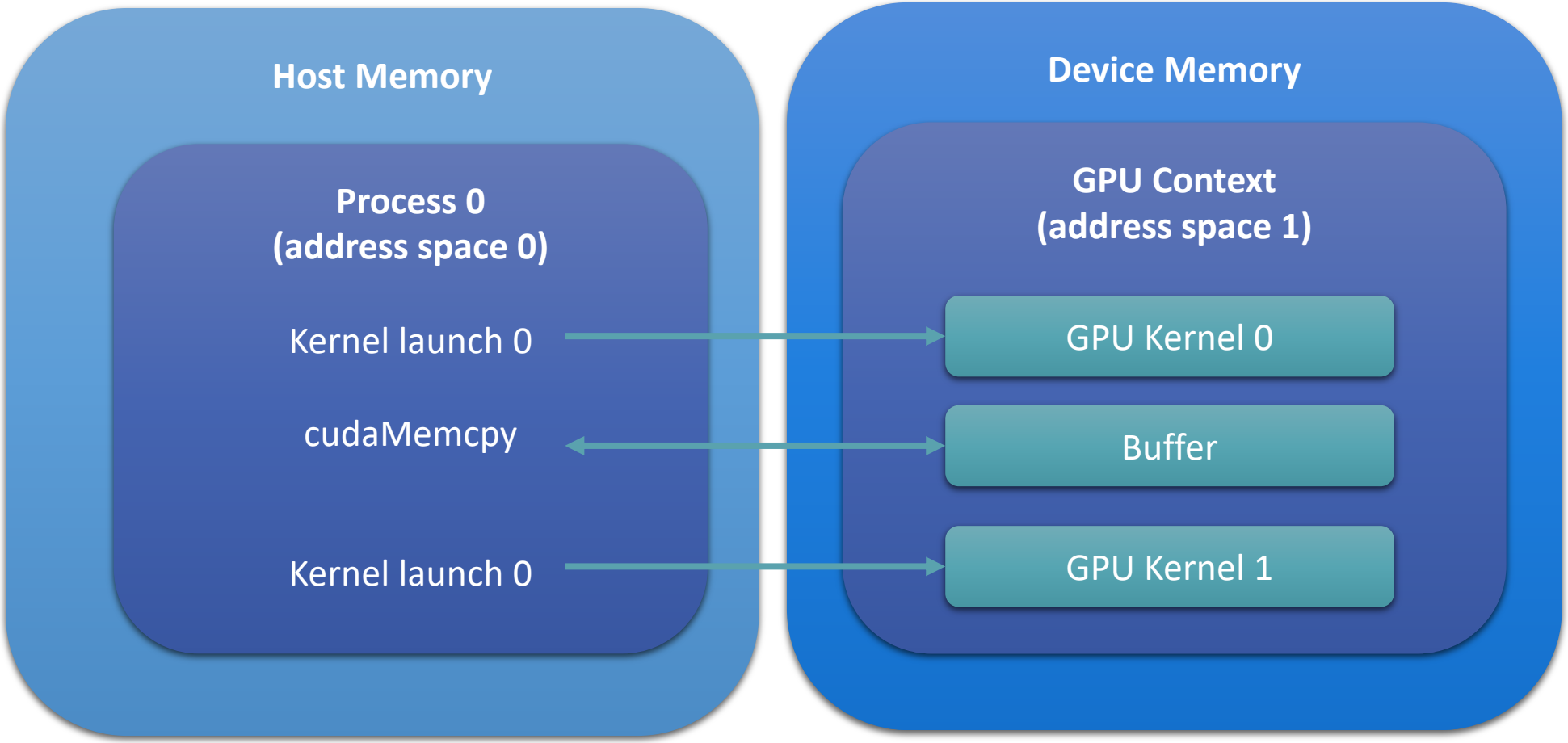
From: https://www.microway.com/hpc-tech-tips/deep-learning-benchmarks-nvidia-tesla-p100-16gb-pcie-tesla-k80-tesla-m40-gpus/

Don’t give up on CPUs too quickly!

- **Caffe con Troll: Shallow Ideas to Speed Up Deep Learning**
  - **https://arxiv.org/abs/1504.04343**
  - **Take-away: training speed is the same as long as FLOPs are the same, independent from hardware**

- **Scaling deep learning on GPU and knights landing clusters**
  - **https://dl.acm.org/citation.cfm?id=3126912**
  - **Take-away: when batch size is sufficiently large, a large computer cluster is much more economical than a gpu cluster because GPUs have high-margins.**

Host/Device Memory and Address Spaces



torch.cuda devices

- Keeps track of existing GPUS devices
- ***torch.cuda*** is used to set up and run **CUDA** operations
- The selected device can be changed with a *torch.cuda.device* contex manager
- once a tensor is allocated, you can do operations on it irrespective of the selected device, and the results will be always placed in on the same device as the tensor.
- Many objects can be moved with .to() or .cuda() (ex. Models)
- https://pytorch.org/docs/stable/notes/cuda.html?highlight=cuda

```
cuda = torch.device('cuda')     # Default CUDA device
cuda0 = torch.device('cuda:0')
cuda2 = torch.device('cuda:2')  # GPU 2 (these are 0-indexed)
x = torch.tensor([1., 2.], device=cuda0)
# x.device is device(type='cuda', index=0)
y = torch.tensor([1., 2.]).cuda()
# y.device is device(type='cuda', index=0)
with torch.cuda.device(1):
    # allocates a tensor on GPU 1
    a = torch.tensor([1., 2.], device=cuda)
    # transfers a tensor from CPU to GPU 1
    b = torch.tensor([1., 2.]).cuda()
    # a.device and b.device are device(type='cuda', index=1)
    # You can also use ``Tensor.to`` to transfer a tensor:
    b2 = torch.tensor([1., 2.]).to(device=cuda)
    # b.device and b2.device are device(type='cuda', index=1)
    c = a + b
    # c.device is device(type='cuda', index=1)
    z = x + y
    # z.device is device(type='cuda', index=0)
    # even within a context, you can specify the device
    # (or give a GPU index to the .cuda call)
    d = torch.randn(2, device=cuda2)
    e = torch.randn(2).to(cuda2)
    f = torch.randn(2).cuda(cuda2)
    # d.device, e.device, and f.device are all device(type='cuda', index=2)
```

Good practice: Device-agnostic code

- Better to write code-agnostic code
- Use a variable to define which device you are going to use
- In a system with multiple GPUs, the environment flag *CUDA_VISIBLE_DEVICES* can be used to manage which GPUs are available

```
import argparse
import torch

parser = argparse.ArgumentParser(description='PyTorch Example')
parser.add_argument('--disable-cuda', action='store_true',
                    help='Disable CUDA')
args = parser.parse_args()
args.device = None
if not args.disable_cuda and torch.cuda.is_available():
   args.device = torch.device('cuda')
else:
   args.device = torch.device('cpu’)
#  create a Tensor on the desired device
x = torch.empty((8, 42), device=args.device)
net = Network().to(device=args.device)

cuda0 = torch.device('cuda:0')  # CUDA GPU 0
for i, x in enumerate(train_loader):
   x = x.to(cuda0)

print("Outside device is 0")  # On device 0 (default in most scenarios)
with torch.cuda.device(1):
   print("Inside device is 1")  # On device 1
print("Outside device is still 0")  # On device 0
```

Pinned and Persistent Memory

- **Pinned memory: Host memory** that cannot be “swapped” to disk
  - **Only data allocated in pinned memory allows for asynchronous copies**
  - Passing an additional *non_blocking=True* argument to a .cuda() call
  - It can be used to overlap data transfers with computation
  - Use *pin_memory=True* to make the DataLoader return batches placed in pinned memory
- **Persistent memory: Device memory** that will still be in the context across multiple kernel launches

Lesson Key Points

- PyTorch Optimizer Algorithms
- PyTorch DataLoader and Disk performance
- PyTorch Multiprocessing
- PyTorch CUDA