ECE-GY 9143
Introduction to High Performance Machine Learning

Lecture 4 02/15/24

Parijat Dube

Gradient Descent Optimization Algorithms and PyTorch

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

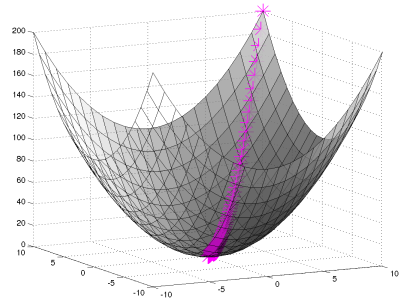
PyTorch Optimizer

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)

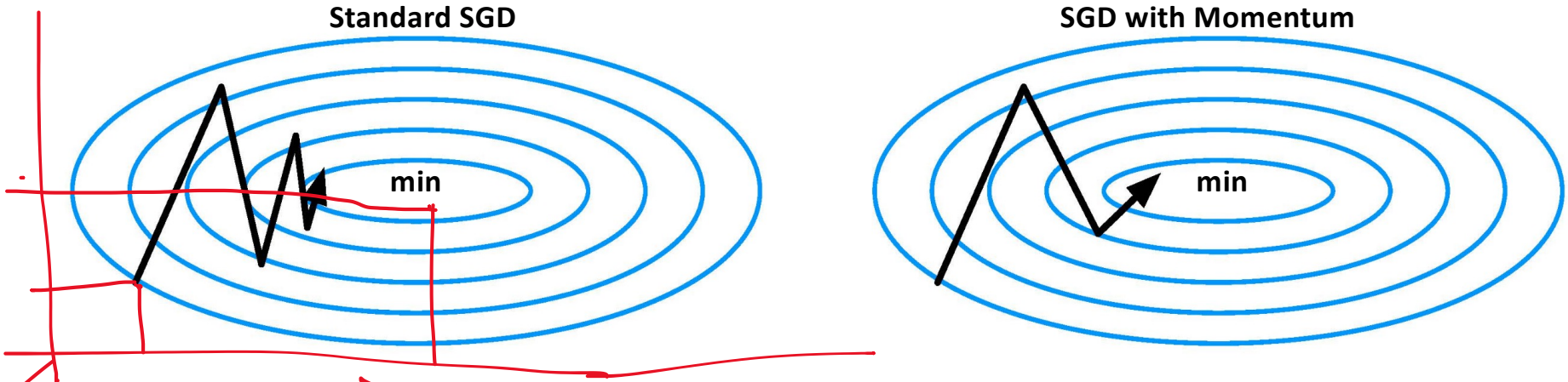
- Traversing efficiently through error differentiable functions' surfaces is an important research area today
- Some of the recently popular techniques **take the gradient history into account** such that each "move" on the error function surface relies on previous ones a bit
- Many algorithms already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

Optimizer Algorithms – Momentum

ex - J = 100x^2 + y^2

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

gradient along x is very large as compared to along y



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

to avoid osscilations

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()
```

drawback - due to momentum, we might overshoot the minima

to overcome this drawback -

Optimizer Algorithms - Nesterov Momentum

- Instead of computing 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, 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

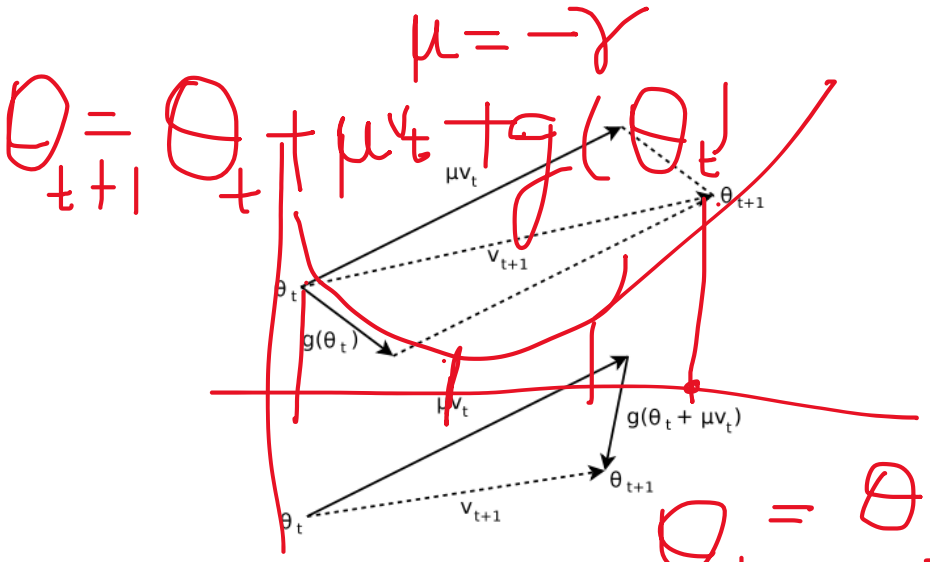
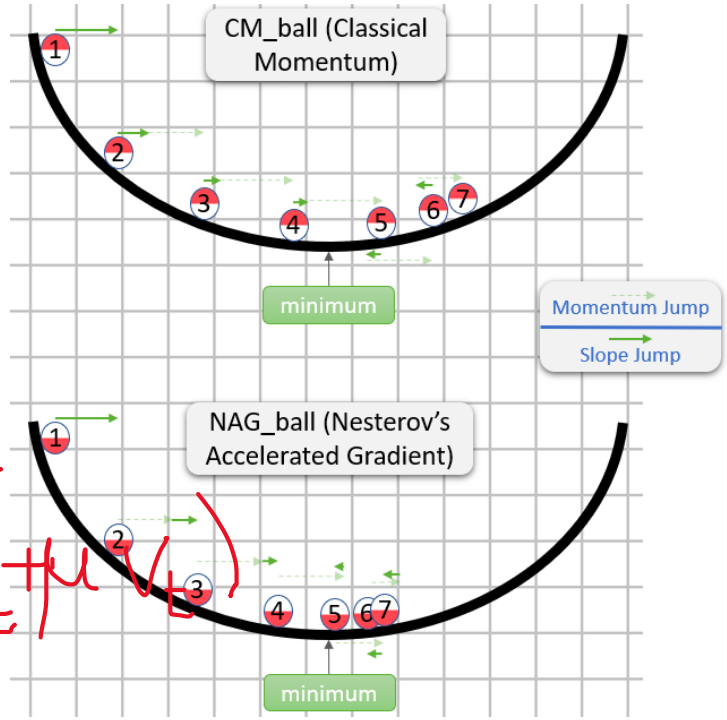


Figure 1. **(Top)** Classical Momentum **(Bottom)** Nesterov Accelerated Gradient



Adaptive gradient

Optimizer Algorithms - Adagrad

learning rate is depends on gradient (parameter)

- Adapts learning rate to 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:

G_t ~ variance of the past gradients

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

can be useful while dealing with sparse features

This algorithm performs best for sparse data because it decreases the learning rate faster for frequent parameters, and slower for parameters infrequent parameter.

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:

as the learning rate will not be same for all the features. It will change more for more frequent features

RMSProp - to overcome this drawback

```
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 phas
  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:

Algorithm 1 Computing ADADELTA update at time t

Require: Decay rate ρ, Constant ϵ
Require: Initial parameter x_1

Initialize accumulation variables $E[g^2]_0 = 0$, $E[\Delta x^2]_0 = 0$
for $t = 1 : T$ **do** %% Loop over # of updates
  Compute Gradient: g_t
  Accumulate Gradient: $E[g^2]_t = \rho E[g^2]_{t-1} + (1-\rho) g_t^2$
  Compute Update: $\Delta x_t = -\frac{\text{RMS}[\Delta x]_{t-1}}{\text{RMS}[g]_t} g_t$
  Accumulate Updates: $E[\Delta x^2]_t = \rho E[\Delta x^2]_{t-1} + (1-\rho)\Delta x_t^2$
  Apply Update: $x_{t+1} = x_t + \Delta x_t$
end for

```
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 similar to momentum technique.. but with exponential moving average
2. $v_k=\beta_2 v_{k-1}+(1-\beta_2)g_k^2$ The second moment similar to Adagrad.. but with exponential moving averga
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. $\hat{\hat{\theta}}_k = \theta_{k-1}-\alpha \frac{\hat{m}_k}{sqrt(\hat{\hat{v}}_k)-\epsilon}$ Update parameters

- In PyTorch:

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

And many others…

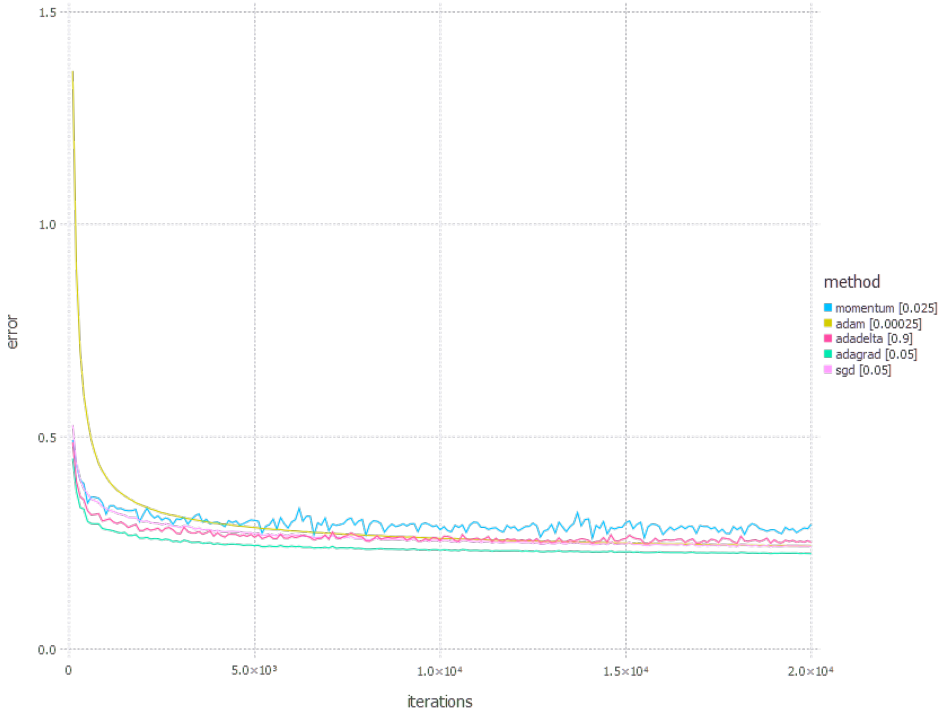
- AdaMax: variant of Adam
- Nadam
  - Adam with Nesterov momentum instead of classical momentum
- RMSprop
  - Divide the gradient by a running average of its recent magnitude
- ….
- Here some interesting readings:
  - https://arxiv.org/abs/1609.04747
  - http://ruder.io/optimizing-gradient-descent/

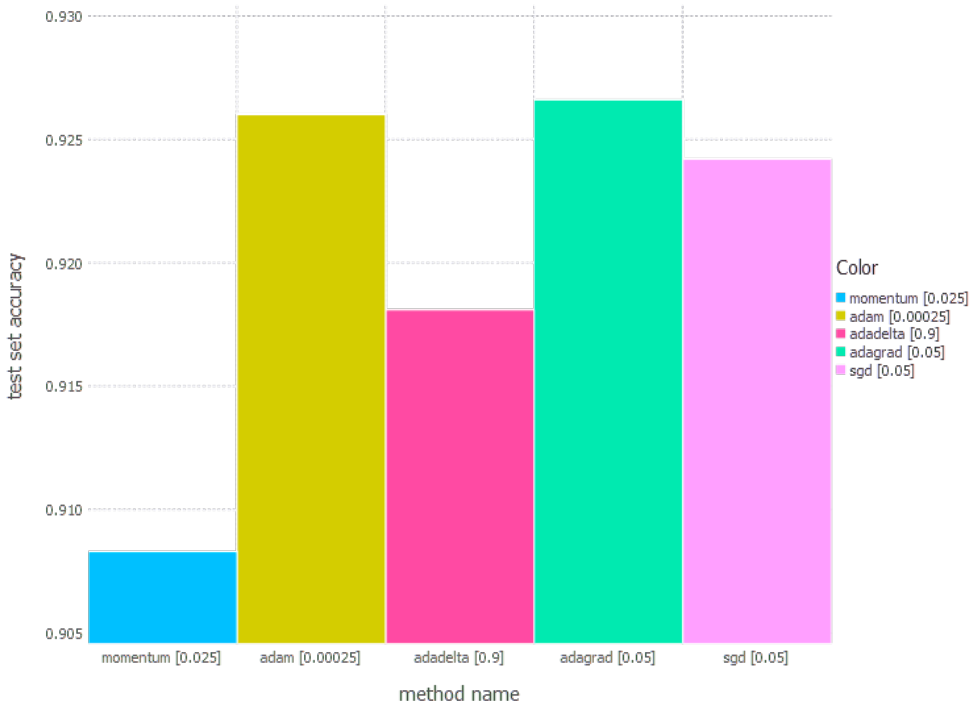
Optimization Techniques Comparison (I)

- 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

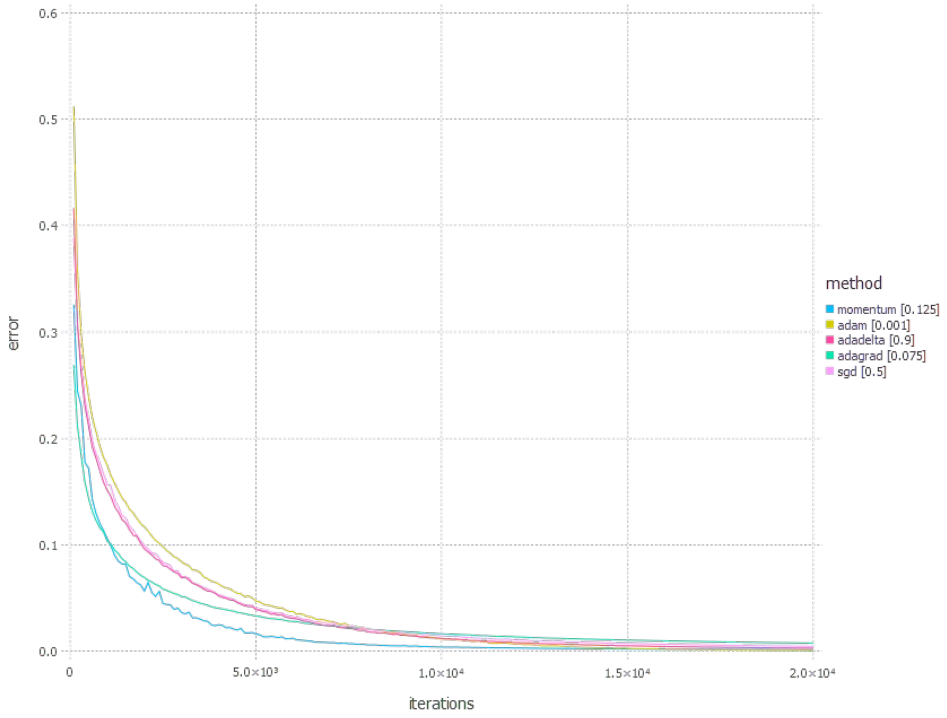
network with linear layer and softmax output (softmax classification)

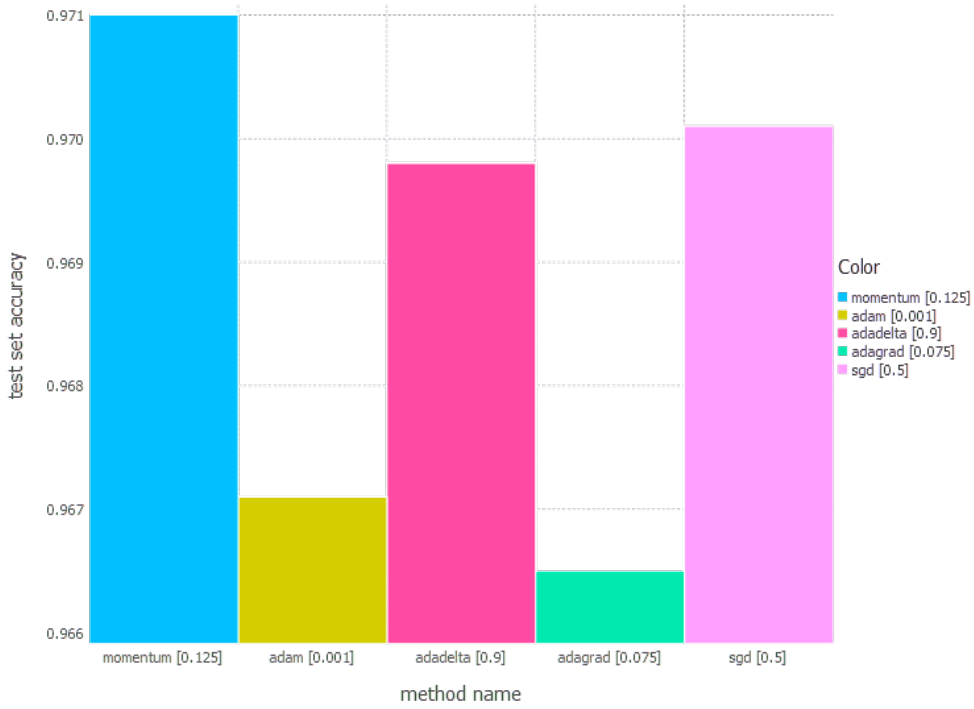




Results on net 2

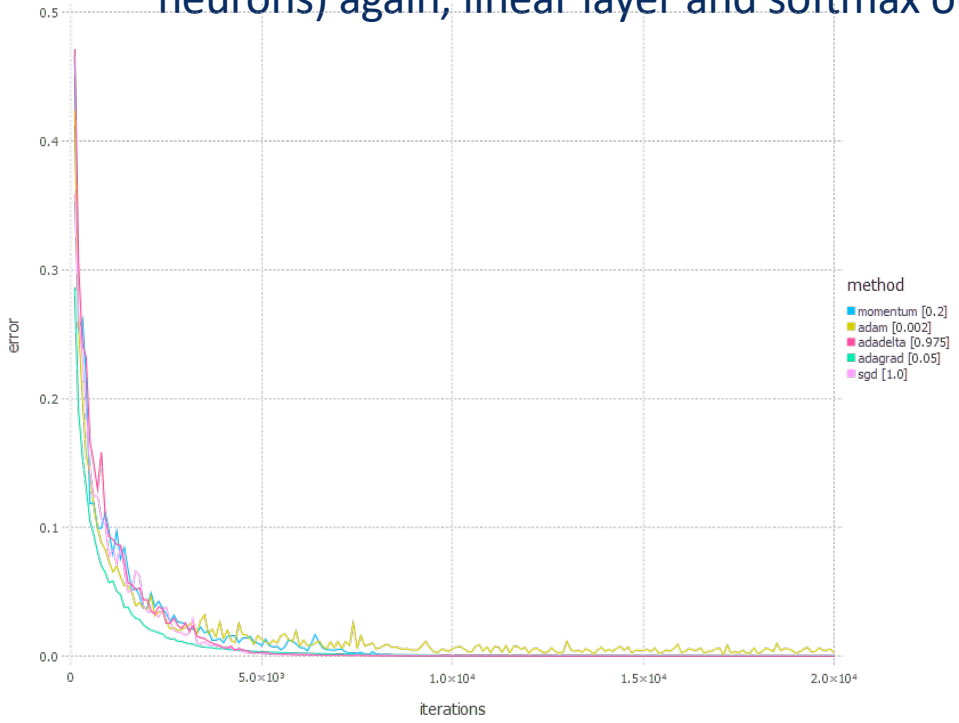
network with sigmoid layer (100 neurons), linear layer and softmax output

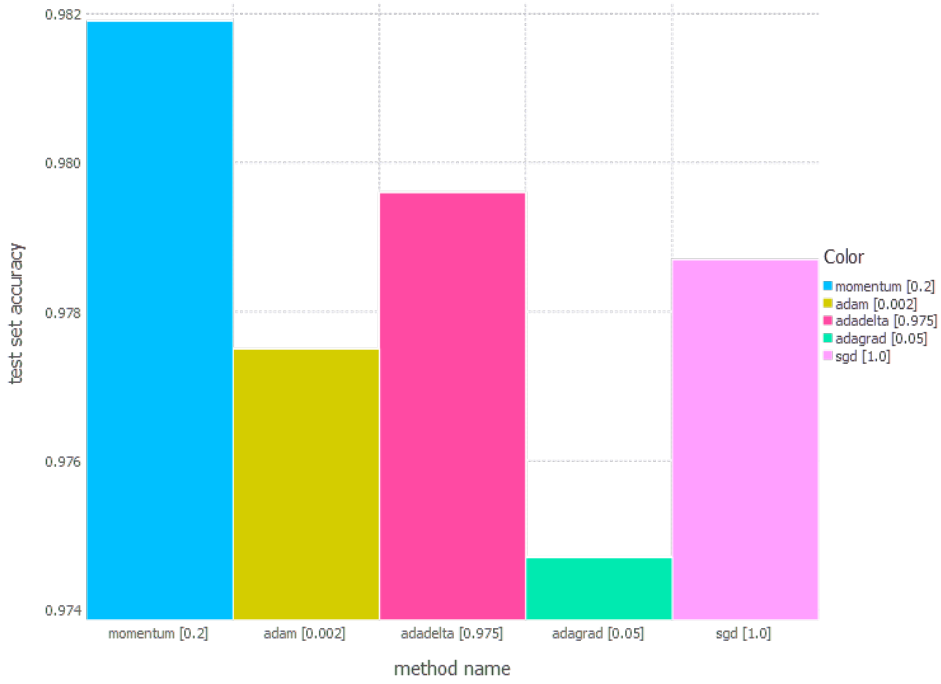




Results on net 3

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

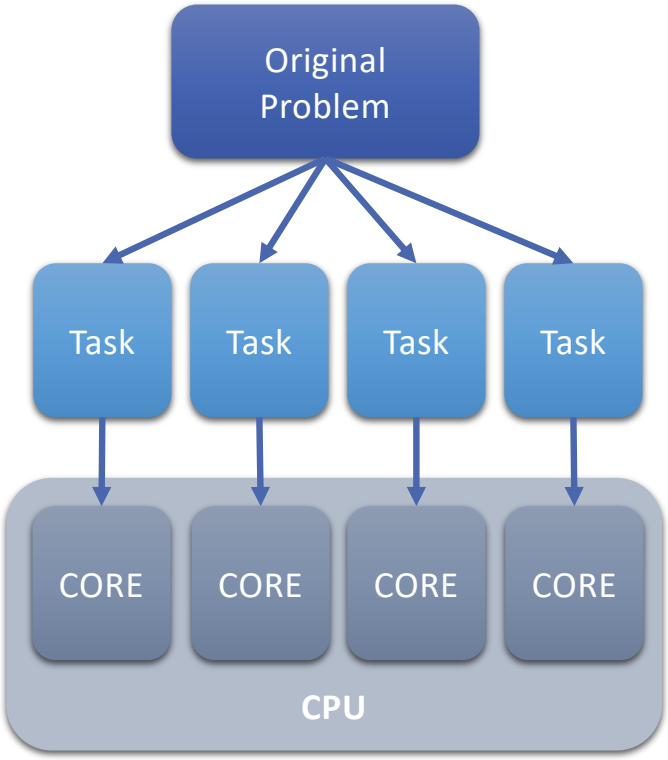
Comments on optimizer

- All the optimizers take in the parameters and gradients 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.
- In practice, SGD with momentum and a per-iteration learning rate schedule often produce better final accuracy with some tuning than automatic update algorithms.

PyTorch Multiprocessing

Why Multiprocessing

- **CPU** has many cores and hardware threads
  - Multi-core processors, hardware threads (hyperthreading)
  - Superscalar CPU cores with hardware multi-threading
- **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 its thread local:

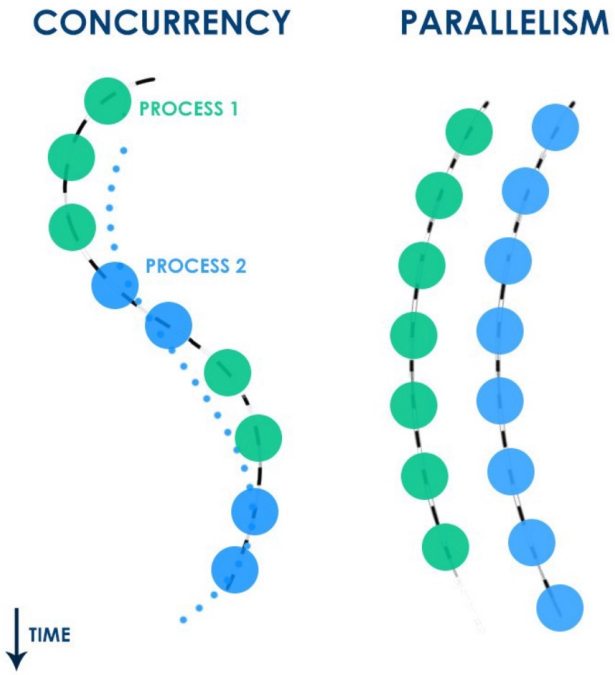
```
mydata = threading.local()
mydata.x = 1
```

- Threads in Python:
  - **CONCURRENCY** but **NOT PARALLELISM** (global interpreter lock--GIL)
  - Within a single process only one thread may be processing Python byte-code at any one time.
  - Memory is shared unless is specifically thread local
  - Cannot take advantage of multiprocessing
- Daemon threads:
  - They do not die with the parent (wait for interpreter shutdown) or needs to be explicitly killed

Not useful for Parallelism Performance!

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

https://superfastpython.com/thread-local-data/


CONCURRENCY
PARALLELISM
PROCESS 1
PROCESS 2
TIME

Python Multiprocessing

- *multiprocessing.Process* class: create another python interpreter process
- Forking and spawning are two different start methods for new processes.
  - 1) SPAWN method: start new fresh process with minimal resource inherited
    - Slower, Default on Windows and MacOS
  - 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); If the parent process has threads owning locks that may cause problems in the child process
  - 3) FORK-SERVER method: a new server-process is created that forks new child processes
    - 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*

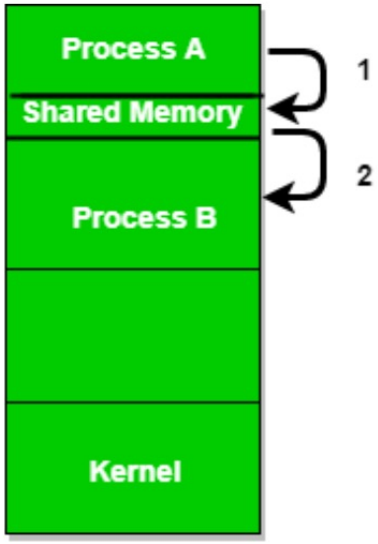
```
from multiprocessing import Process, Pool

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
- High Performance tips for torch.multiprocessing
- See https://pytorch.org/docs/master/notes/multiprocessing.html#reuse-buffers-passed-through-a-queue

Shared Memory for Inter Process Communication

Producer-Consumer pattern



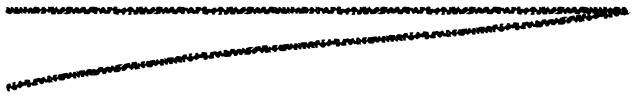
https://www.geeksforgeeks.org/inter-process-communication-ipc/

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
- 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

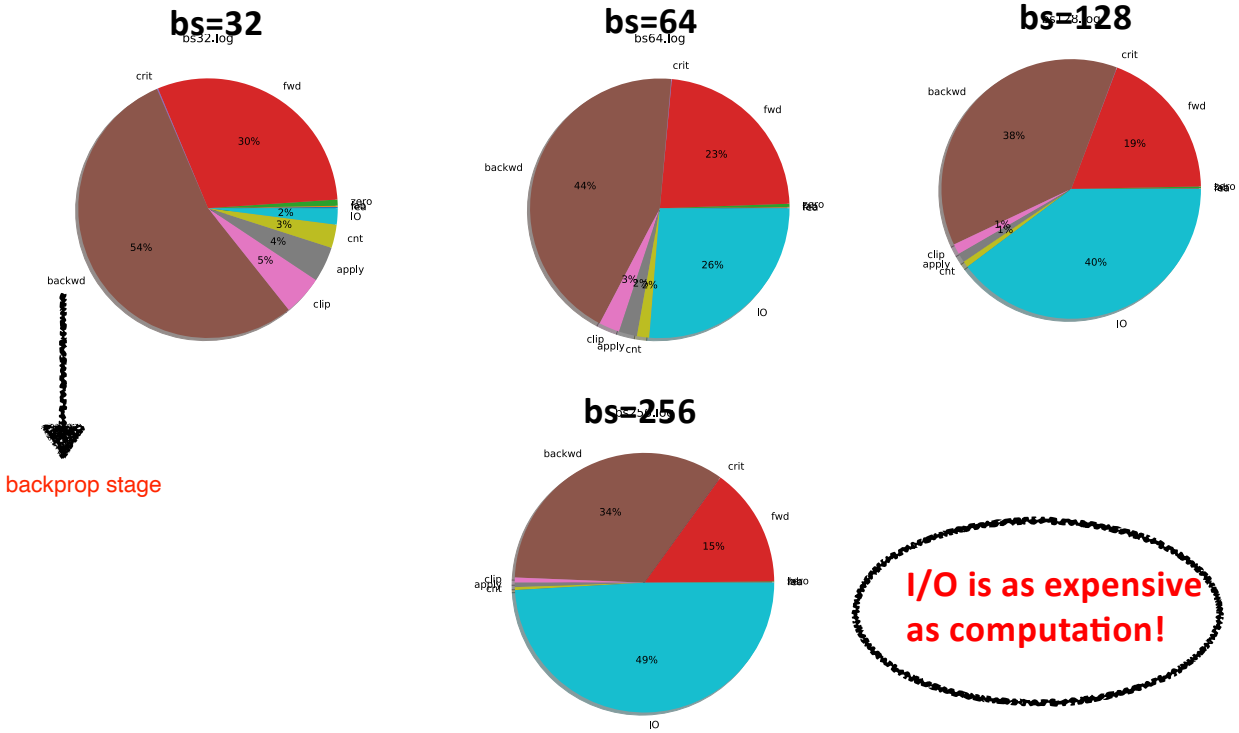
- **GaDei: On Scale-up Training As A Service For Deep Learning (Zhang et al. ICDM'2017)**
  - **https://arxiv.org/pdf/1611.06213.pdf**



PyTorch Data loading and Preparation

I/O could be a big problem

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



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

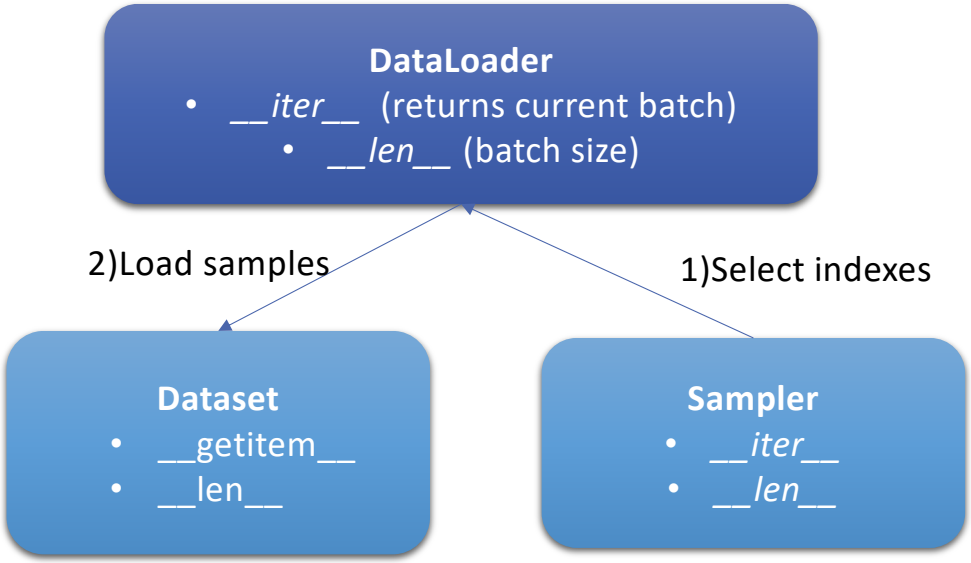
- Transform primitives are applied in __getitem__ to each sample after loading it with *io.imread()*

```
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**

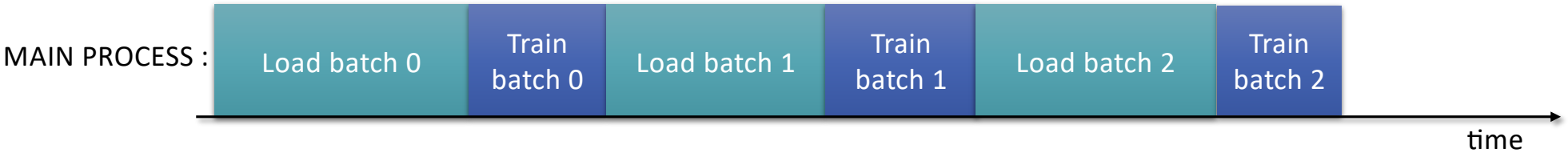


Loading data- *torch.utils.data.Dataloader*

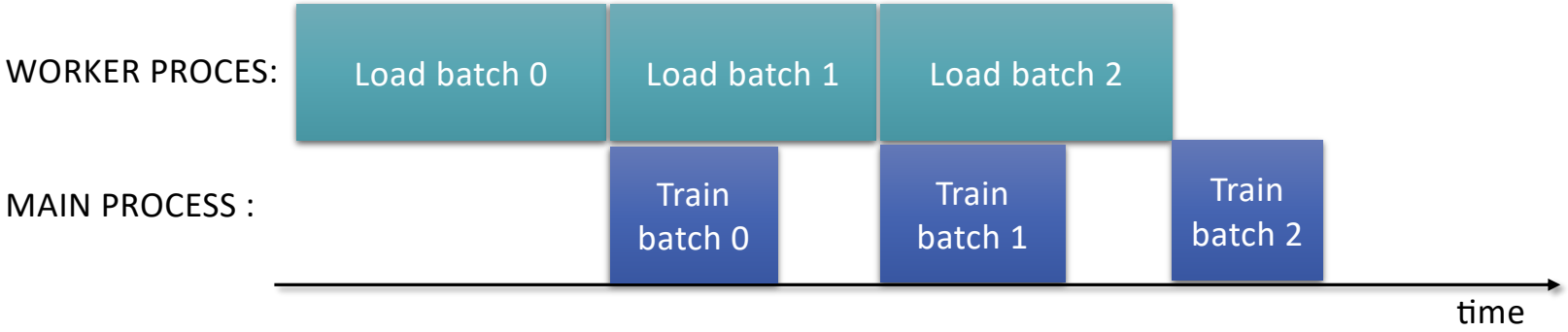
- Useful Parameters:
  - *batch_size:* batch size
  - *sampler:* define which sampler to use
  - *pin_memory:* copy into CUDA pinned memory before returning the data
  - *shuffle:* reshuffle data at each epoch
- Available *samplers:*
  - *SequentialSampler*
  - *RandomSampler*
  - *SubsetRandomSampler*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

Data Prefetching

- Normal pipeline:



- Pipeline with Load Prefetching:



Load time still limiting factor. Can we do better?

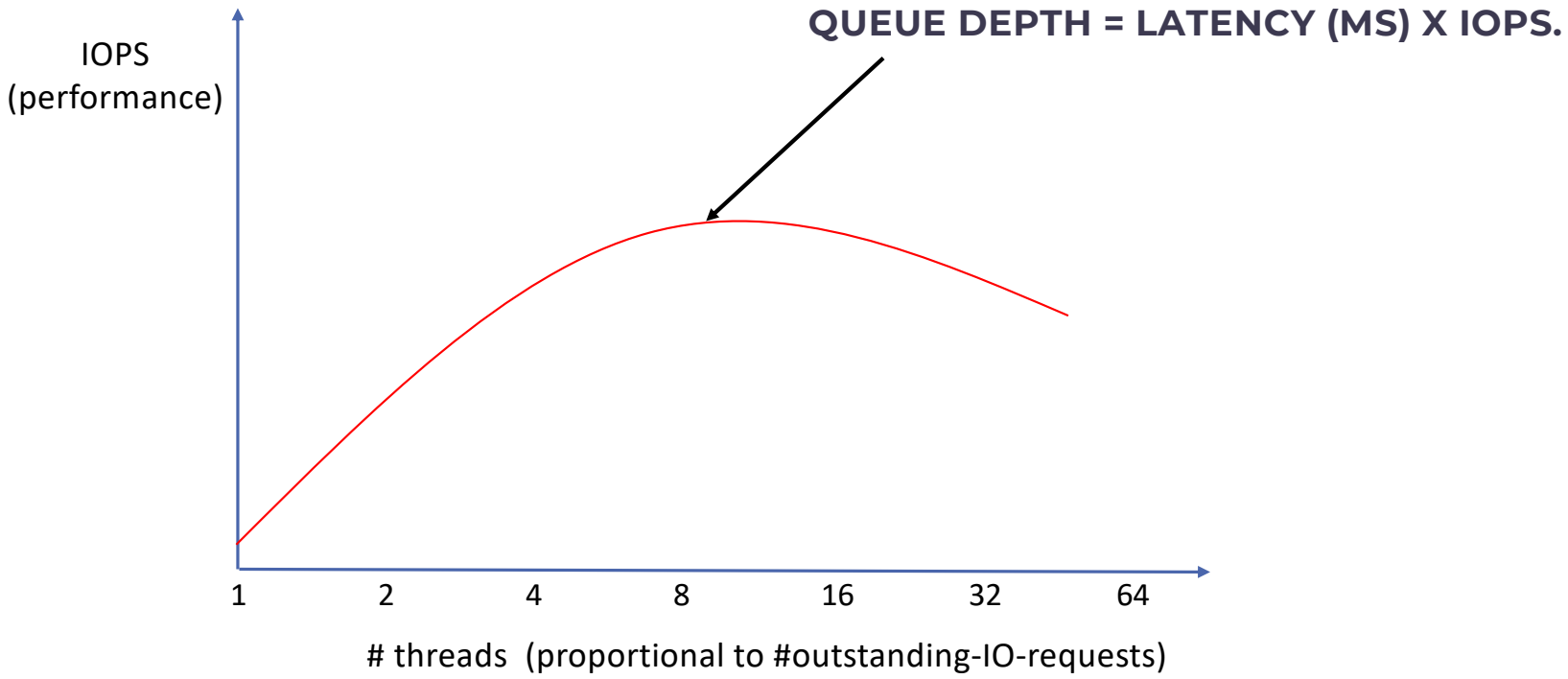
Disk Performance Characteristics

- 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

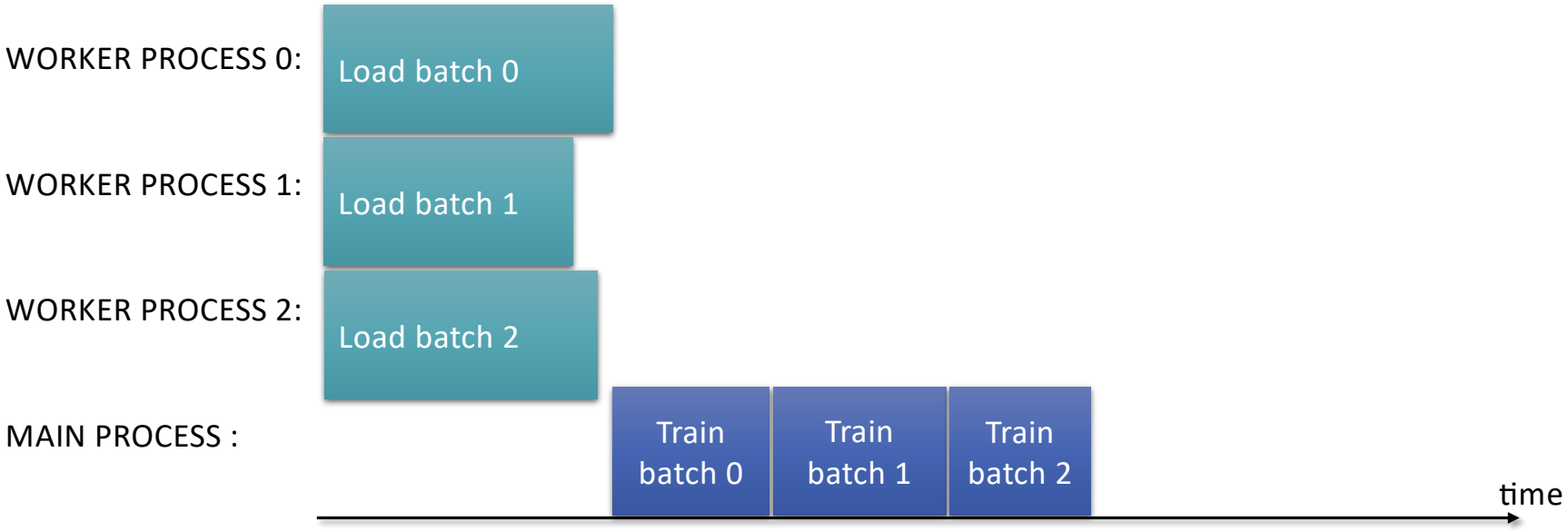
Disk Performance Characteristics



https://blog.docbert.org/queue-depth-iops-and-latency/

Multi-threaded Data Prefetching

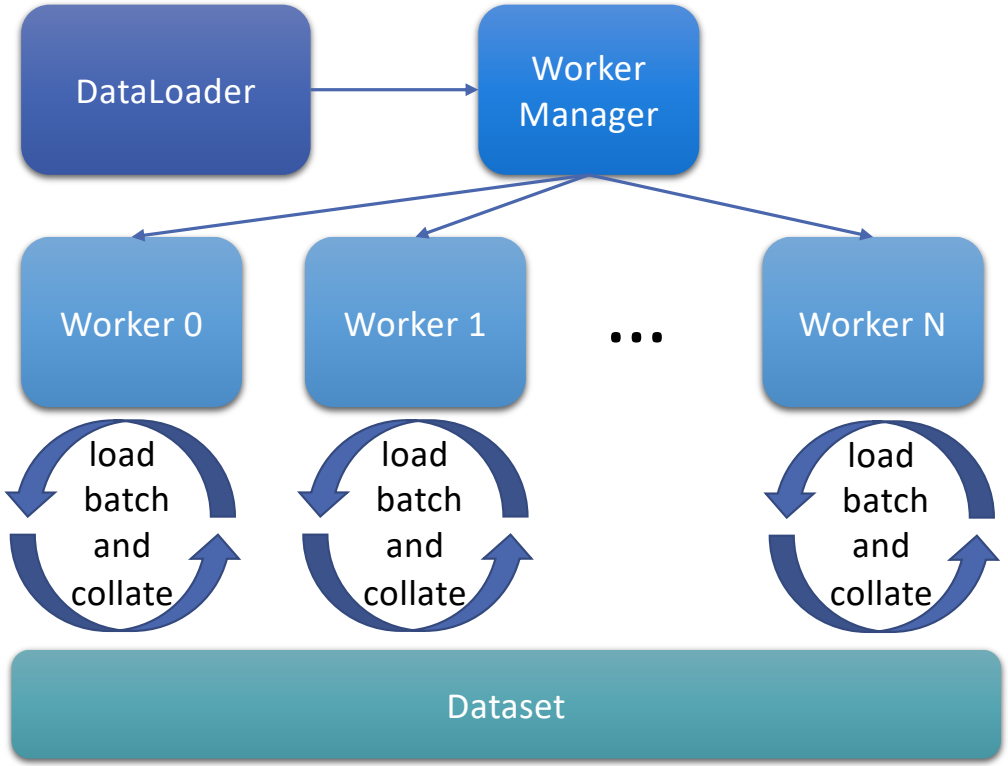
- Issue many outstanding I/O requests using multiple threads:



- Prefetching limitations?

torch.utils.data.Dataloader Architecture

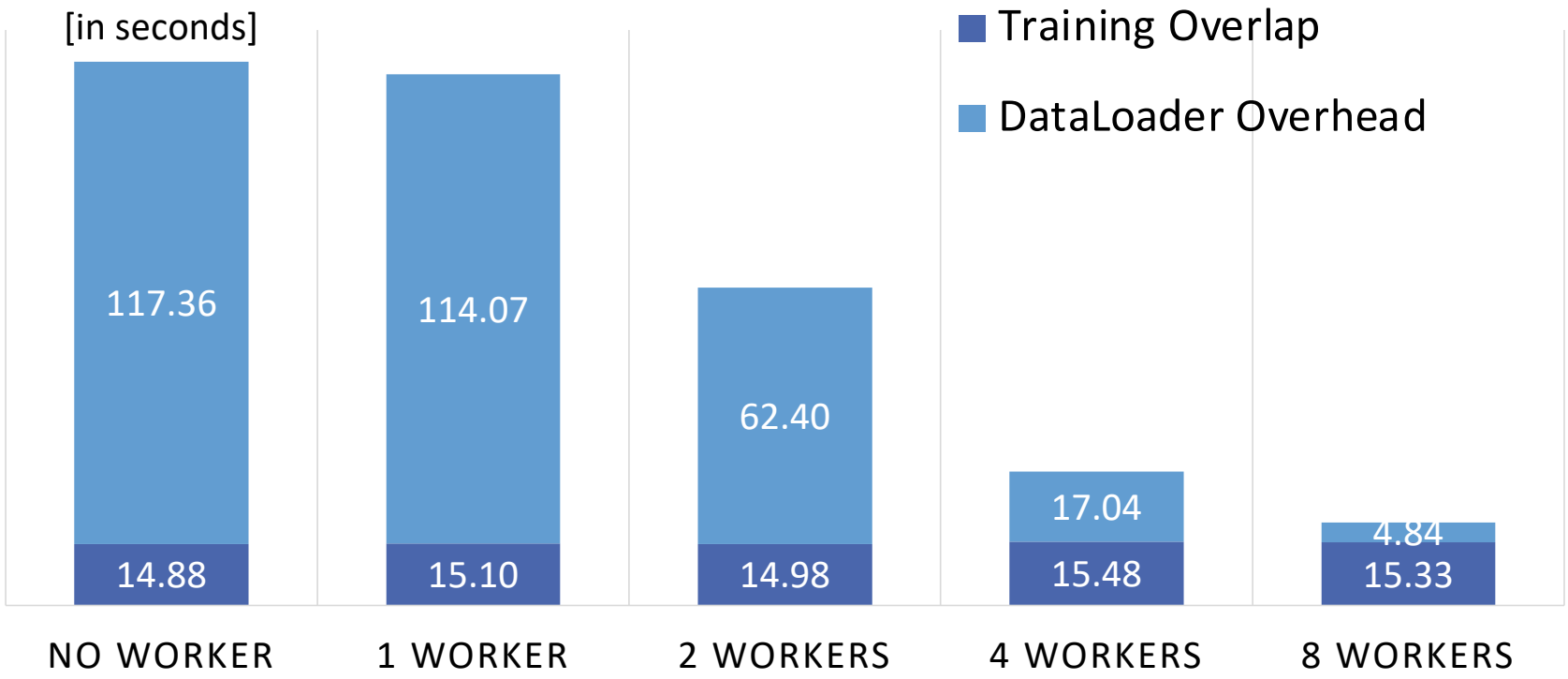
- DataLoader (main process):
  - Requests a set of batches based on *sampler*
- WorkerManager (thread):
  - Dispatch batches to workers
- Workers (processes):
  - Load all samples in each batch
  - Collate batches:
    - Build a batch Tensor with samples



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?

Example: Number of workers effect



Ideal speed-up vs actual speed-up:

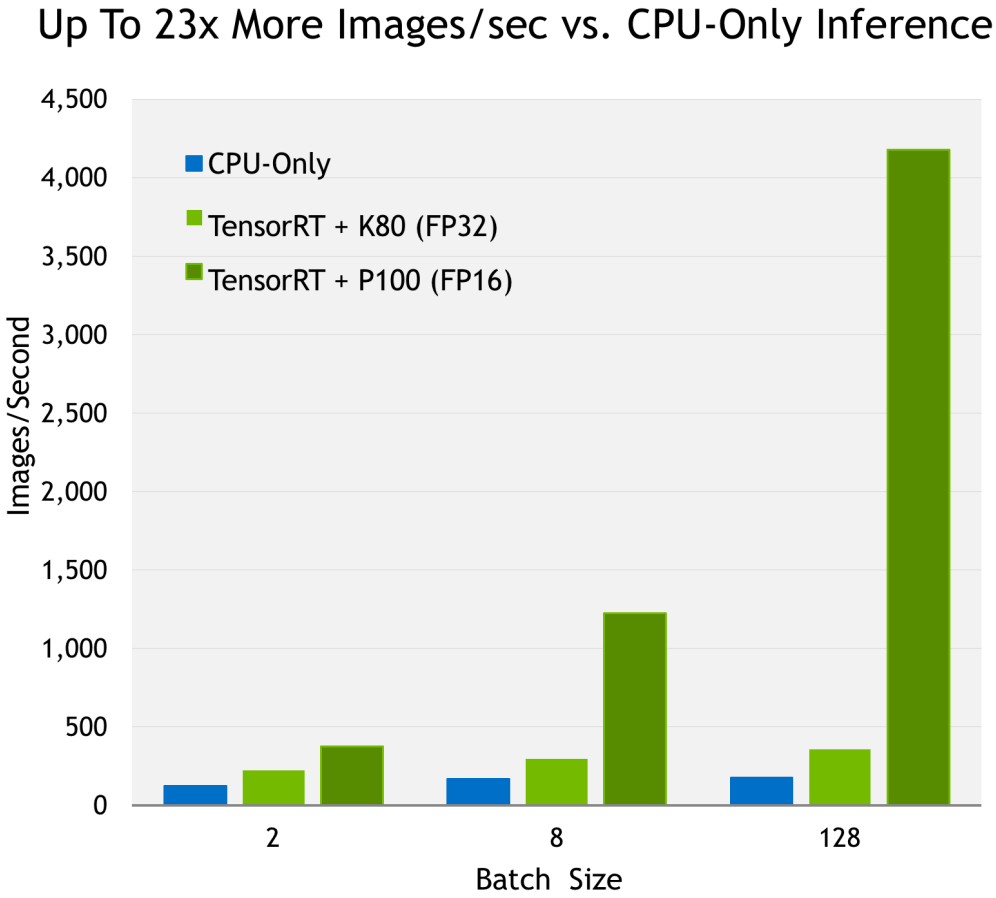
- The 8x speedup should be (114.07 + 15.10) / 8 = 16.15.
- The ideal overhead with 8 workers would be 16.15 - 15.33 = 0.82, but the actual measured is 4.84

PyTorch CUDA

CUDA is a parallel computing platform and programming model that makes using a GPU for general purpose computing simple and elegant.

Why CUDA- Inference

- Intel Xeon E5-2690v4 CPU
- Tesla P100 GPU
- TensorRT is a library developed by NVIDIA for faster inference on NVIDIA graphics processing units



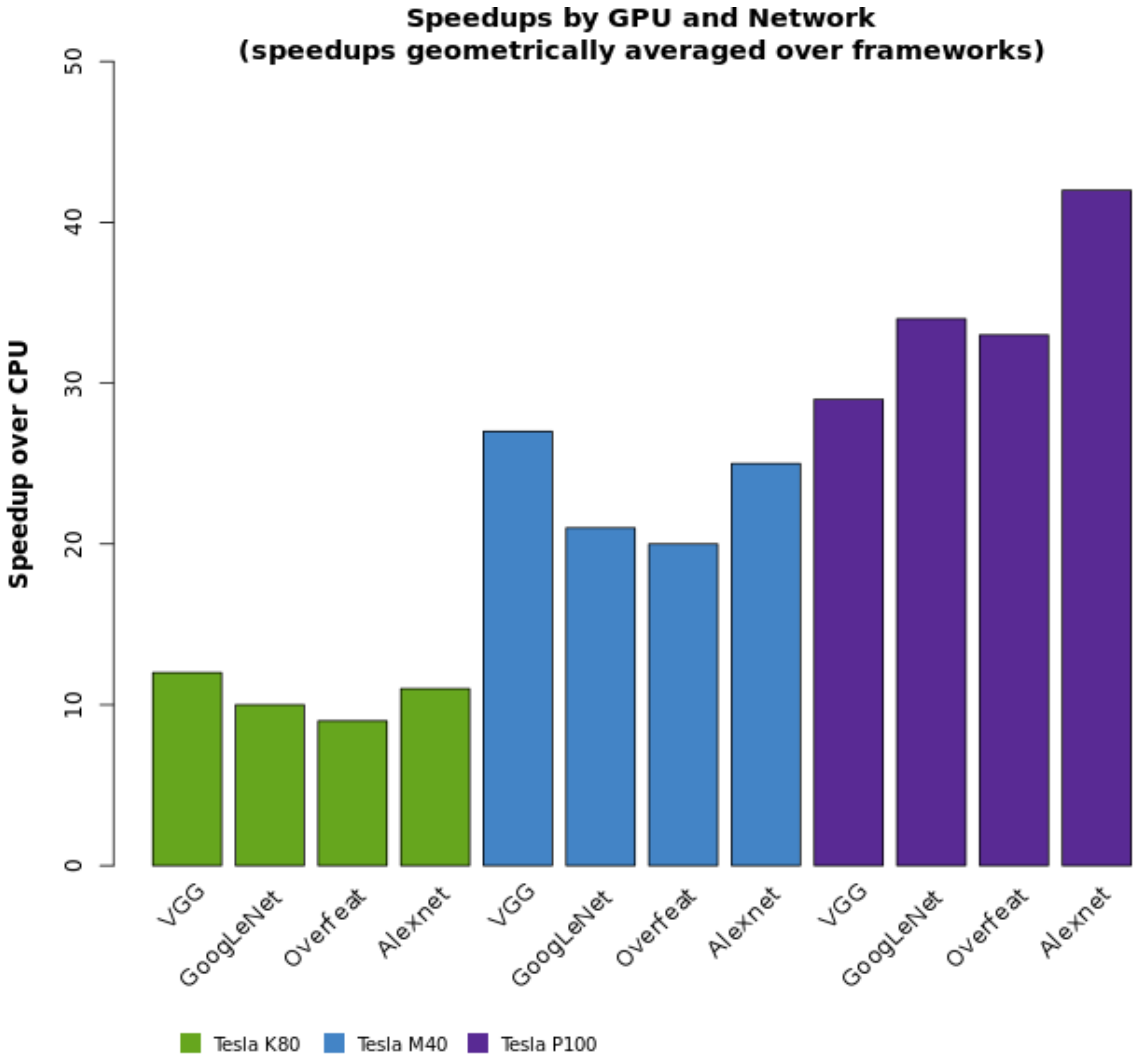
GoogLeNet, Tesla P100 + TensorRT (FP16), Tesla K80 + TensorRT (FP32), CPU-Only + Caffe (FP32)
CPU: 1 Socket Broadwell E5-2690 v4@2.6GHz with HT off

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



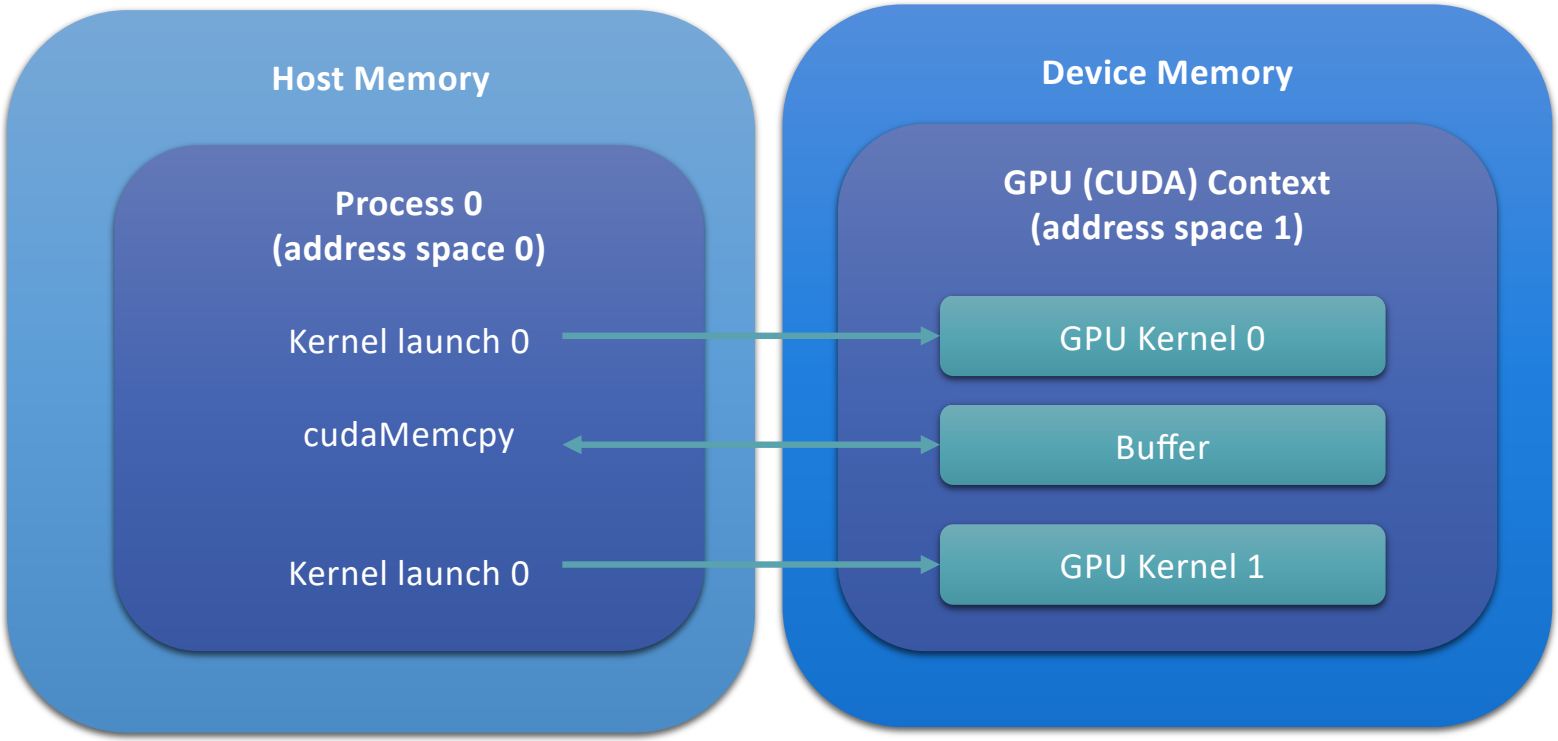
From: https://www.microway.com/hpc-tech-tips/deep-learning-benchmarks-nvidia-tesla-p100-16gb-pcie-tesla-k80-tesla-m40-gpus/

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



GPU/CUDA Context

- A CUDA context is associated with a single host program that is using the GPU.
- A context holds in memory all CUDA kernels and allocated memory that is making use of and is blind to the kernels and memory of other currently existing contexts.
- When a context is destroyed (at the end of a GPU based program, for example), the GPU performs a cleanup of all code and allocated memory within the context, freeing resources up for other current and future contexts.
- A single CUDA host program can generate and use multiple CUDA contexts on the GPU.
- No exact one-to-one relation between host processes and CUDA contexts

https://subscription.packtpub.com/book/application-development/9781788993913/5/ch05lvl1sec34/contexts

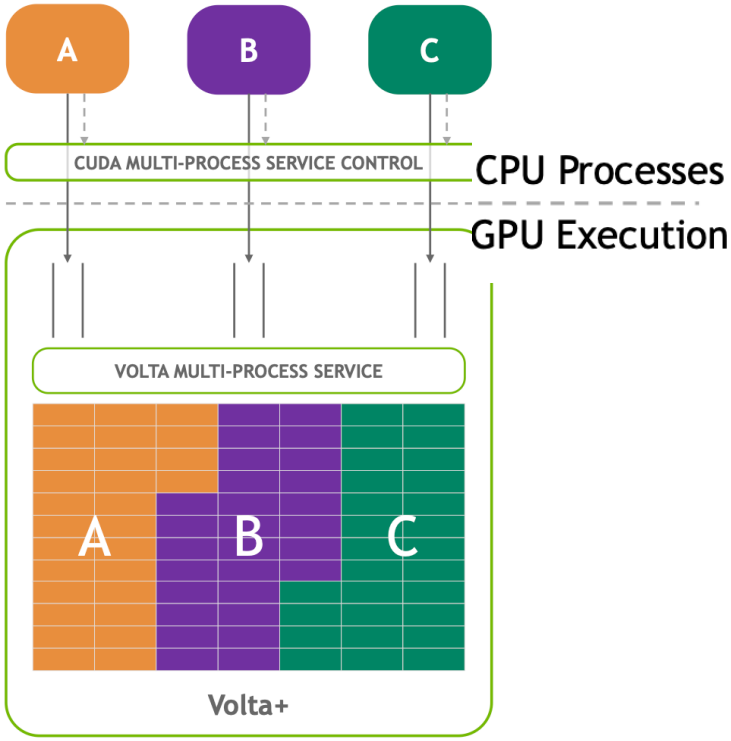
CUDA Context

Every process creates its own *CUDA context*

- The context is a stateful object required to run CUDA
- Automatically created for you when using the CUDA runtime API
- On V100, the size is ~300 MB + your GPU code size
- This limits the number of ranks we can fit on the GPU regardless of application data

Sharing GPU across multiple host processes

- NVIDIA Multi Process Service (MPS) allows multiple processes to share GPU compute resources
- Concurrently maps multiple GPU ranks to the same device
- Useful when each rank is too small to fill the GPU on its own



https://www.olcf.ornl.gov/wp-content/uploads/2021/06/MPS_ORNL_20210817.pdf

torch.cuda devices

- Keeps track of existing GPU devices
- ***torch.cuda*** is used to set up and run **CUDA** operations
- The selected device can be changed with a *torch.cuda.device* context 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)
- Can we add tensors from two different devices?
- 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 device-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

Overlapping kernel execution and data transfer

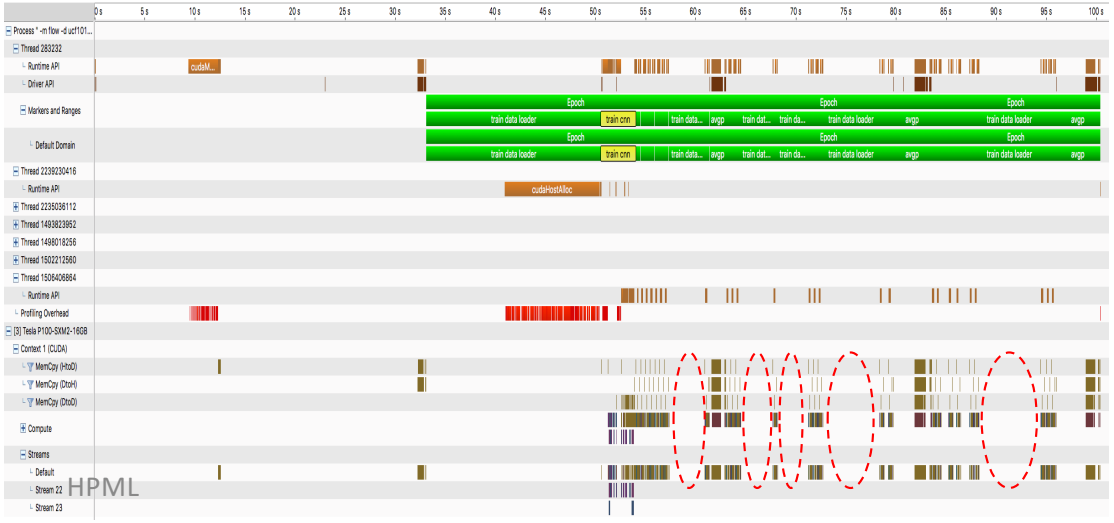
- The kernel execution and the data transfer to be overlapped must both occur in *different*, *non-default* streams.
- The host memory involved in the data transfer must be pinned memory.

Lesson Key Points

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

- Nvprof - num_workers = 0

- Nvprof num_workers =4



Fewer gaps, not completely eliminated

Acknowledgements

- The lecture material is prepared by Giacomo Domeniconi, Parijat Dube, Ulrich Finkler, and Alessandro Morari from IBM Research, USA.