Introduction to High-Performance Machine Learning

Lecture 4 02/15/24

Dr. Parijat Dube

Dr. Kaoutar El Maghraoui

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

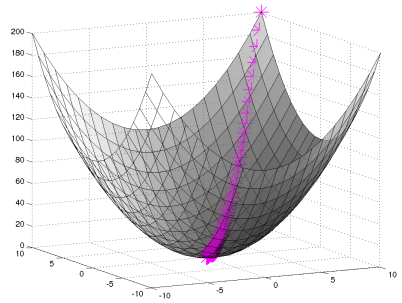
PyTorch Optimizer

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

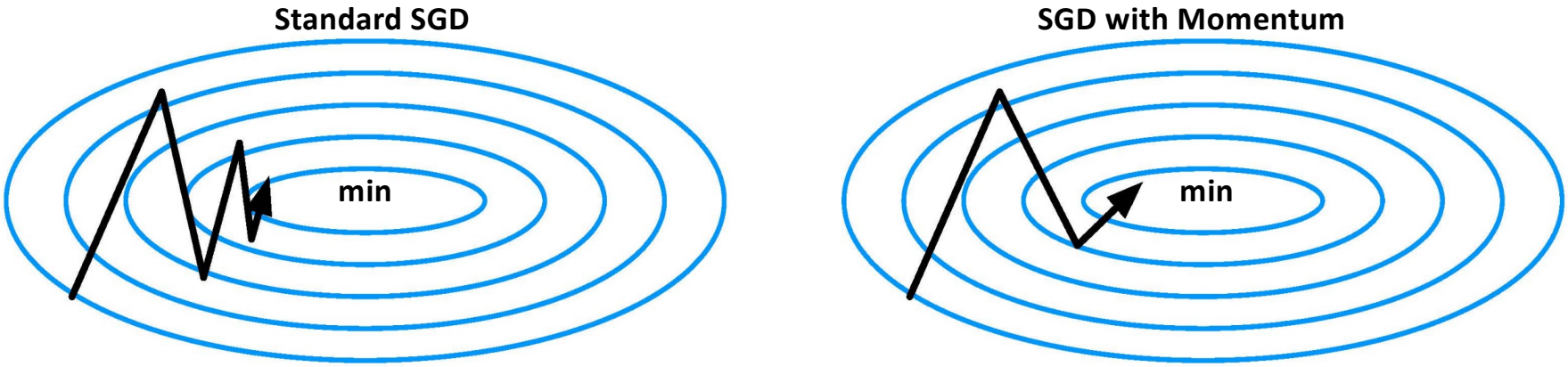
- The Learning rate has a large impact on convergence
  - Too small → too slow
  - Too large → oscillatory and may even diverge
- Choosing a proper (initial) learning rate
- Should learning rate be fixed or adaptive?
  - **Decaying learning rate:** drop by 10 after N iterations and then again after other 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

- 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 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

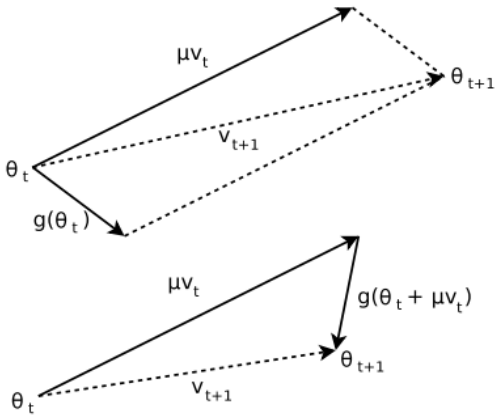
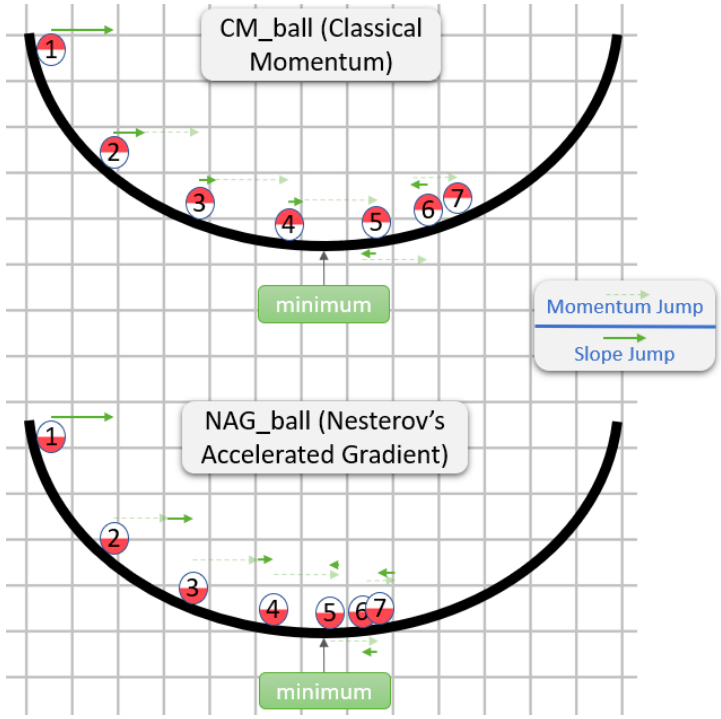


Figure 1. **(Top)** Classical Momentum **(Bottom)** Nesterov Accelerated Gradient



Optimizer Algorithms - Adagrad

- 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:

```
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. $\hat{\hat{\theta}}_k = \theta_{k-1}-\alpha \frac{\hat{m}_k}{sqrt(\hat{v}_k)-\epsilon}$ Update parameters

- In PyTorch:

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

And many others…

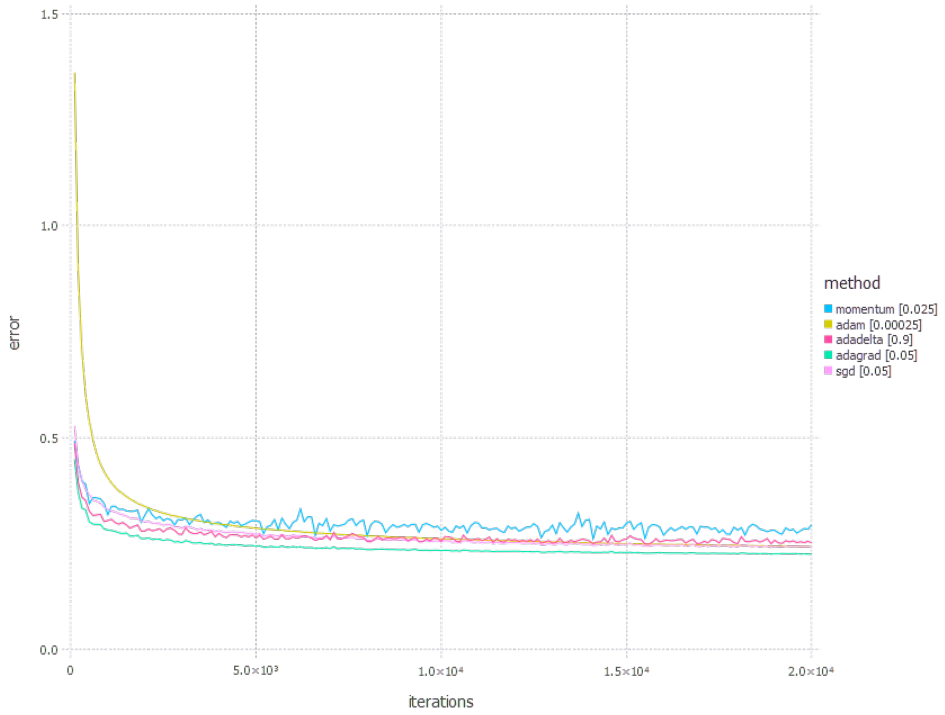
- AdaMax: variant of Adam
- Nadam
  - Adam with Nesterov momentum instead of classical momentum
- RMSprop
  - Divide the gradient by a running average of its recent magnitude
- Yellowfin
  - an automatic tuner for momentum and learning rate in SGD
  - the newer and with results that seem to overcome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

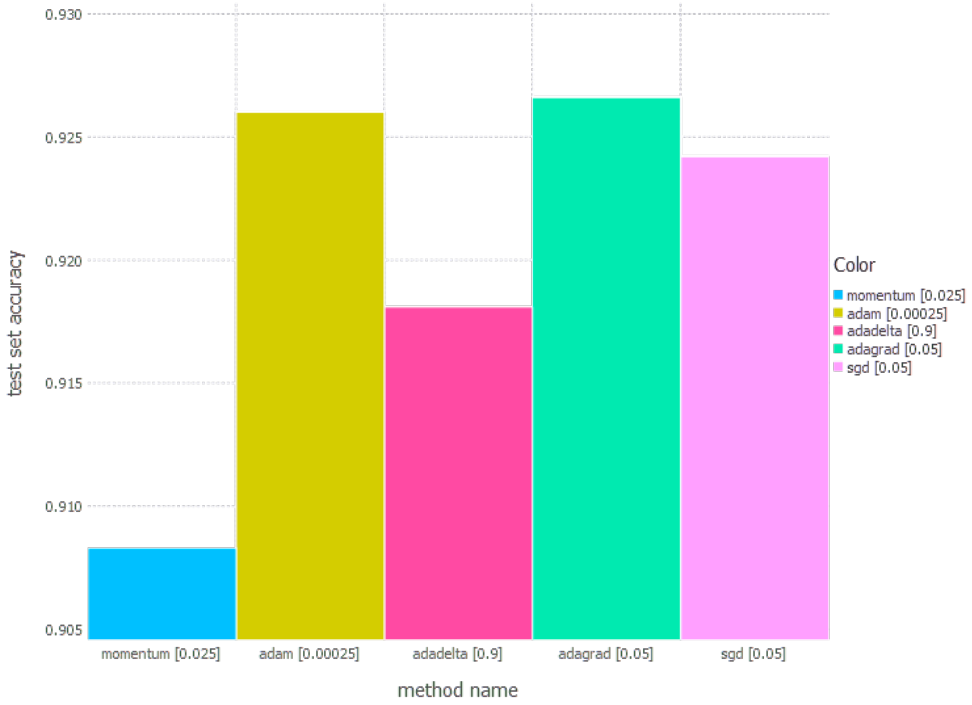
Optimization Techniques Comparison (I)

- 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

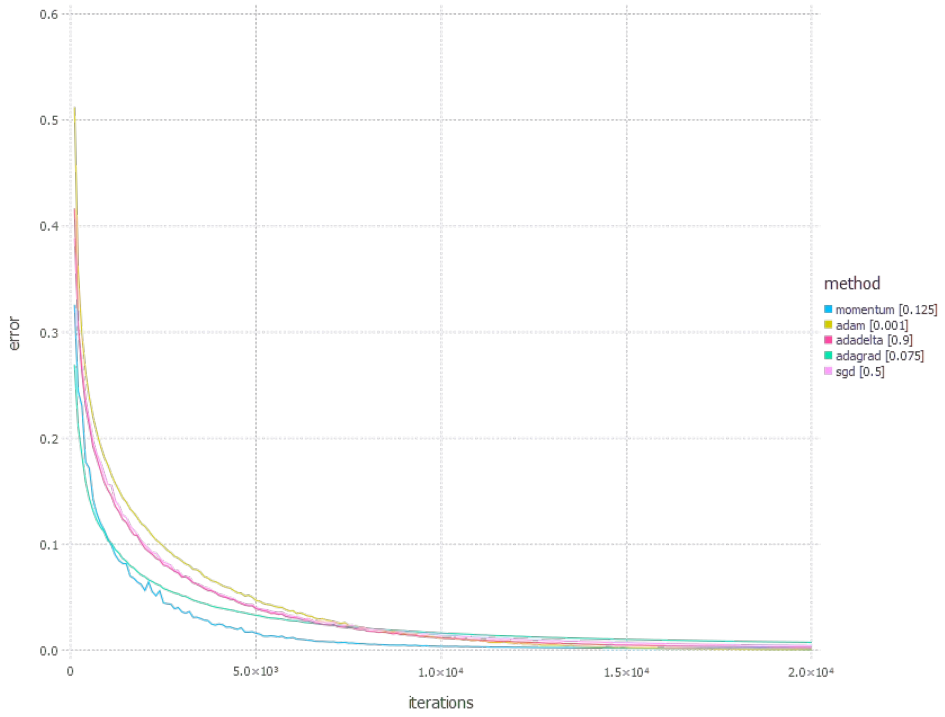
network with linear layer and softmax output (softmax classification)

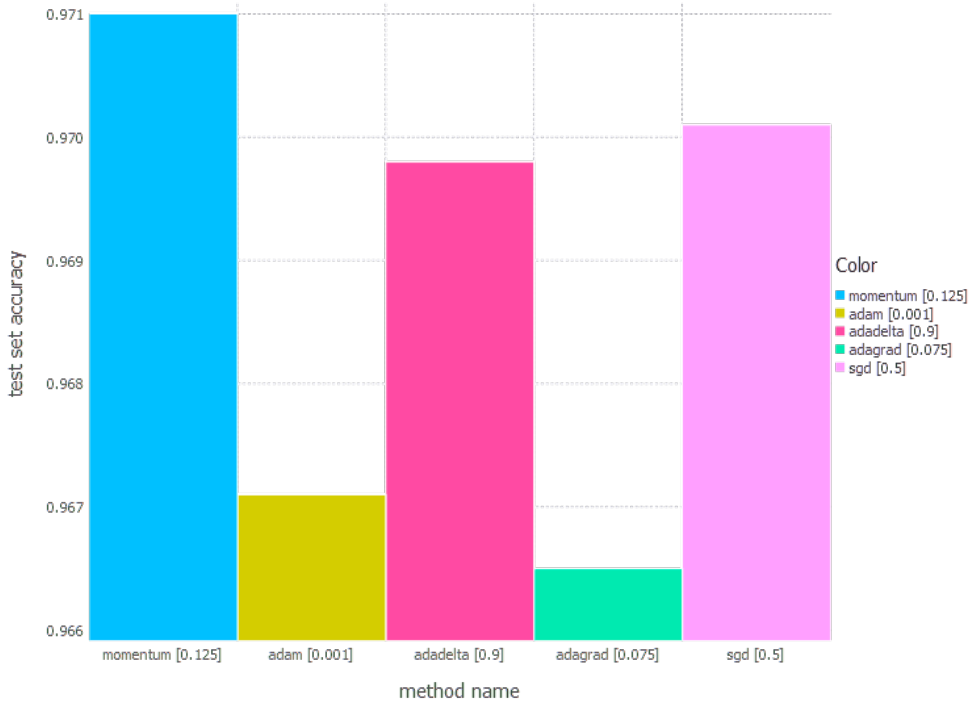




Results on net 2

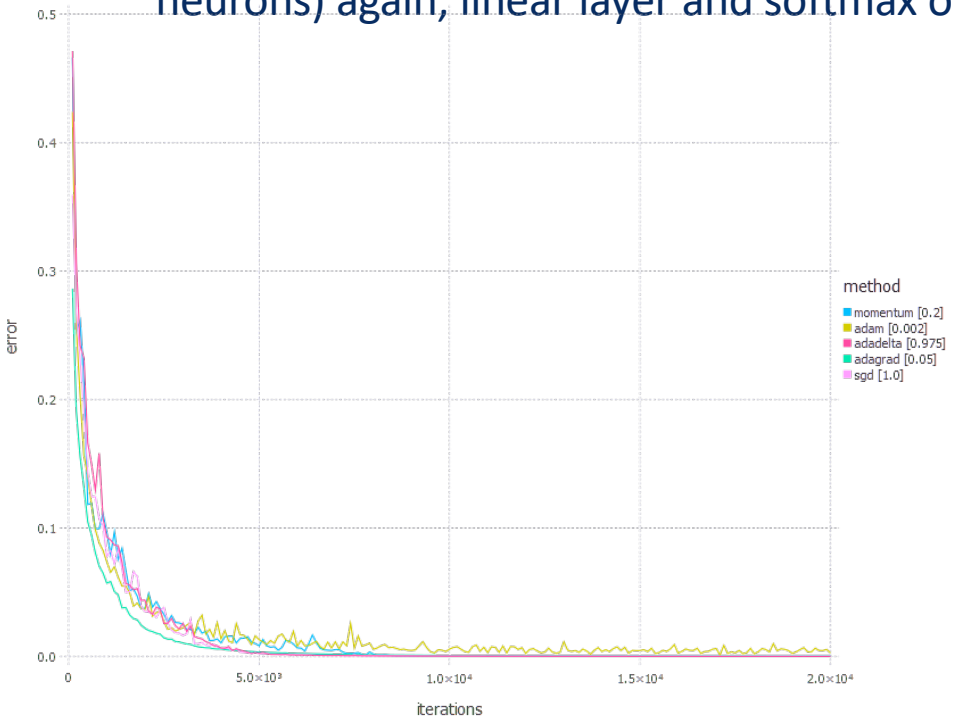
network with sigmoid layer (100 neurons), linear layer and softmax output

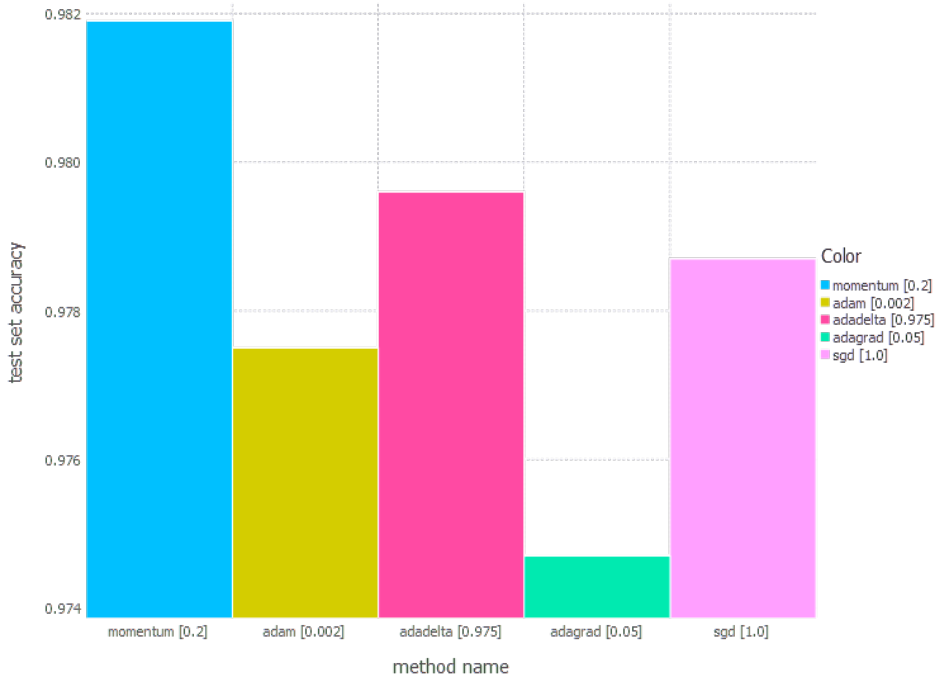




Results on net 3

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 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 (poly, cosine) often produce better final accuracy with some tuning than automatic update algorithms, in particular for fp32.

Can we create a ‘Super-optimizer’ ?

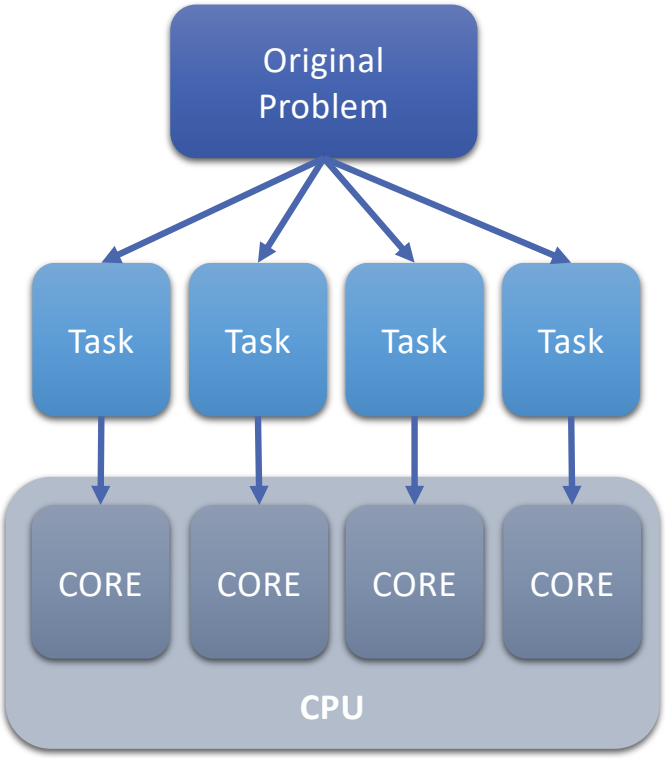
- Evolutionary Stochastic Gradient Descent for Optimization of Deep Neural Networks (Cui et al., NeurIPS’2018)
  - https://papers.nips.cc/paper/7844-evolutionary-stochastic-gradient-descent-for-optimization-of-deep-neural-networks.pdf

PyTorch Multiprocessing

Why Multiprocessing

- **CPU** has many cores and hardware threads
  - 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



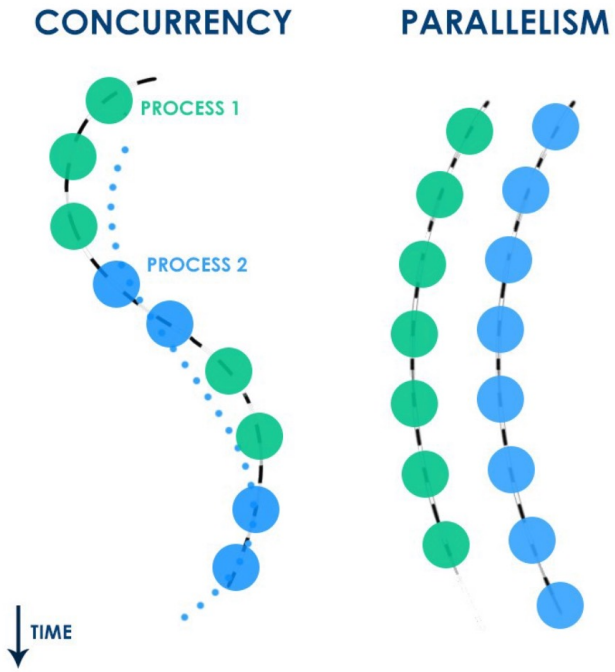
Python Threading

```
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()*
- 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


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 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

- **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

Managing I/O efficiently is a key step in optimising ML performance

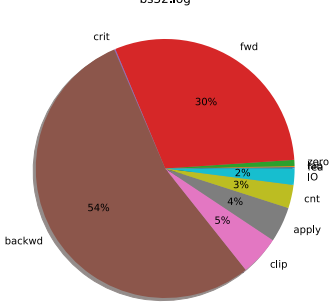
I/O could be a big problem

Data -
Disk -> CPU -> GPU

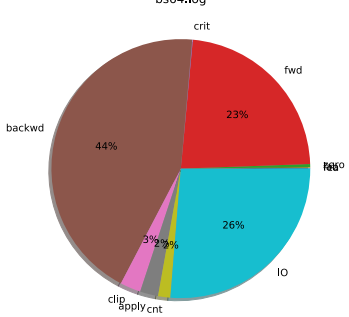
- As GPU gets faster, I/O becomes a bottleneck

bs - batch size

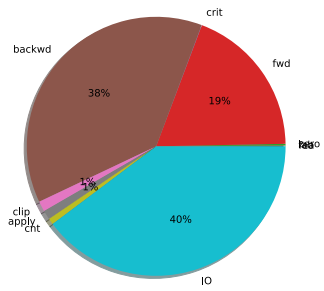
bs=32



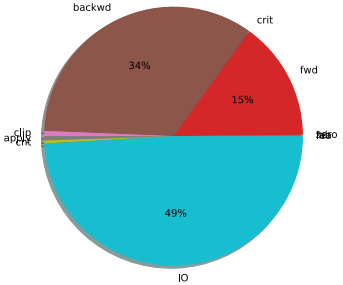
bs=64



bs=128

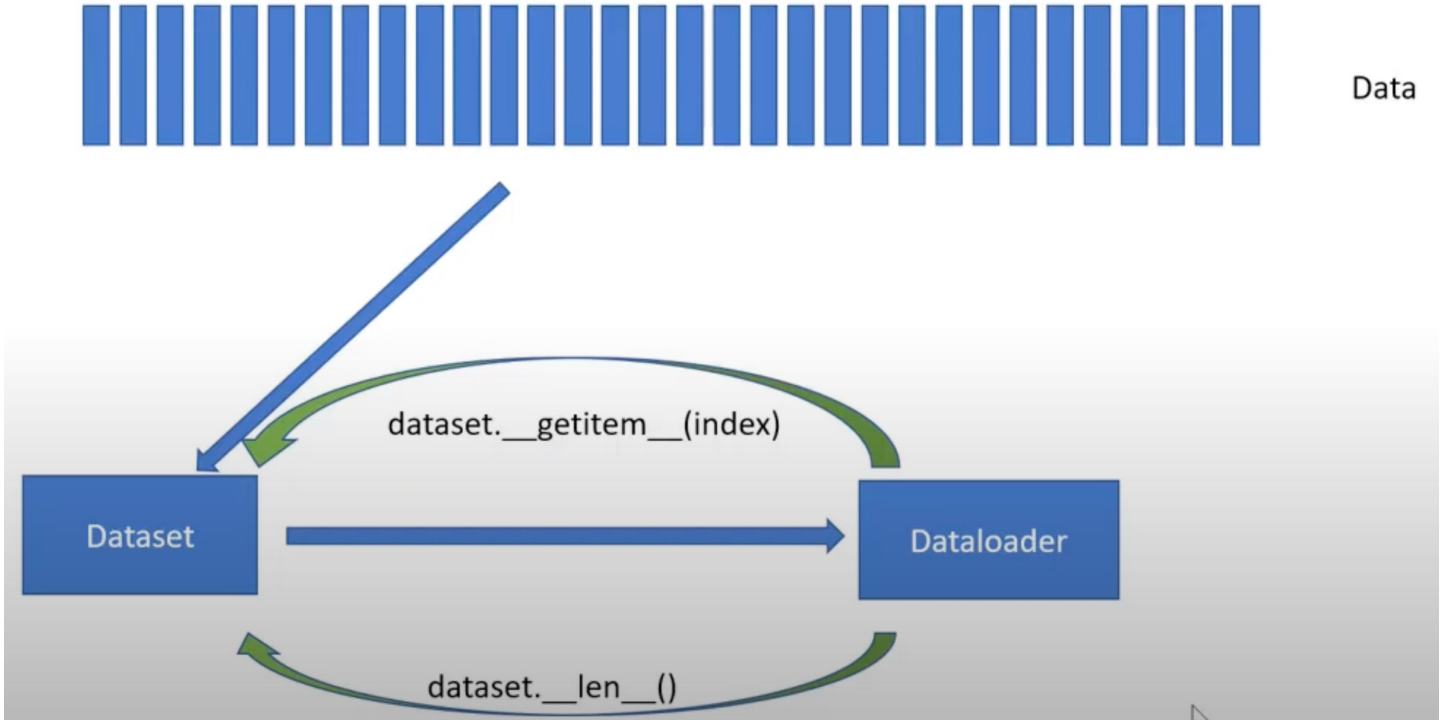


bs=256



I/O is as expensive as computation!

Datasets and Dataloaders



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 → so that our dataset can be used as a dictionary

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

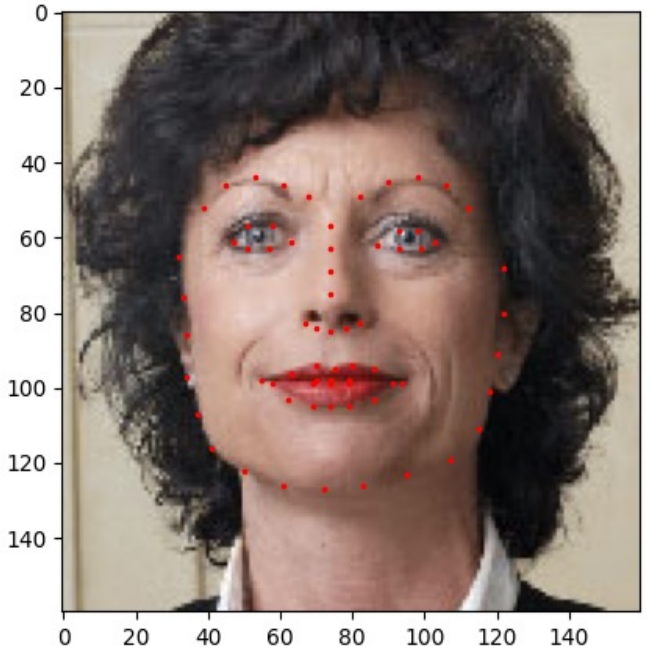
Landmark dataset

Dataset comes with a csv file with annotations which looks this:

image_name,part_0_x,part_0_y,part_1_x,part_1_y,part_2 ... ,part_67_x,part_67_y 0805

personali01.jpg,27,83,27,98, **...** 84,134

1084239450**_e76e00b7e7.jpg,**70,236,71,257, **... ,**128,312



68 different landmark points are annotated for each face.

Custom Dataset Class Example

- torch.utils.data.Dataset is an abstract class representing a dataset. Your custom dataset should inherit Dataset and override the following methods:
- __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.

- 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()*

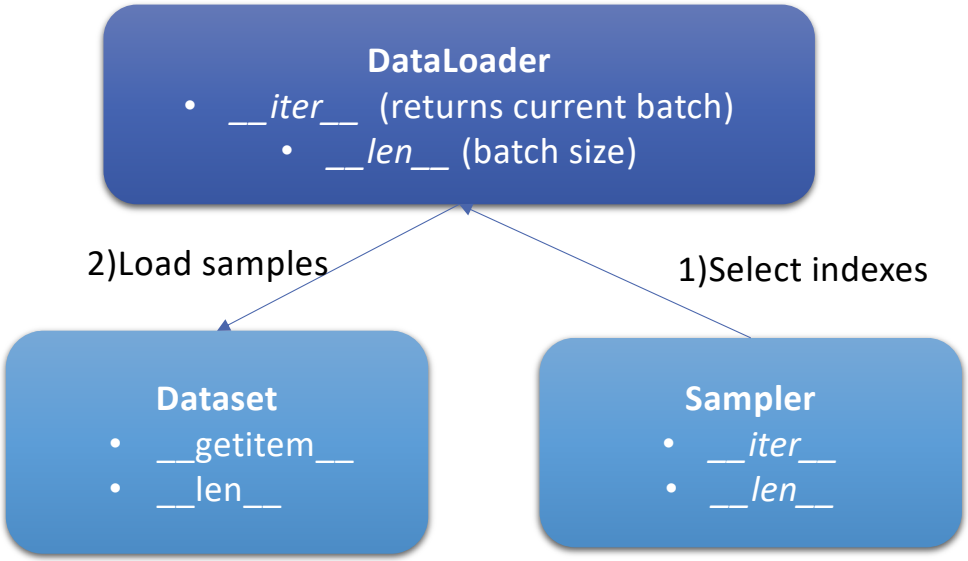
```
def __len__(self):
        return len(self.landmarks_frame)

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:]
    landmarks = np.array([landmarks])
    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**

tells us how to sample data from the dataset



how do you sample batch from your data

PyTorch: DataLoaders

A **DataLoader** wraps a **Dataset** and provides mini-batching, shuffling, multithreading, for you

When you need to load custom data, just write your own Dataset class

```
import torch
from torch.utils import data

class Dataset(data.Dataset):
  'Characterizes a dataset for PyTorch'
  def __init__(self, list_IDs, labels):
        'Initialization'
        self.labels = labels
        self.list_IDs = list_IDs

  def __len__(self):
        'Denotes the total number of samples'
        return len(self.list_IDs)

  def __getitem__(self, index):
        'Generates one sample of data'
        # Select sample
        ID = self.list_IDs[index]

        # Load data and get label
        X = torch.load('data/' + ID + '.pt')
        y = self.labels[ID]

        return X, y
```

num_workers (int, optional) – how many subprocesses to use for data loading. 0 means that the data will be loaded in the main process. (default: 0)

PyTorch: DataLoaders

```
# Parameters
params = {'batch_size': 64,
          'shuffle': True,
          'num_workers': 6}
max_epochs = 100

# Datasets
partition = # IDs
labels = # Labels

# Generators
training_set = Dataset(partition['train'], labels)
training_generator = data.DataLoader(training_set, **params)

validation_set = Dataset(partition['validation'], labels)
validation_generator = data.DataLoader(validation_set, **params)

# Loop over epochs
for epoch in range(max_epochs):
    # Training
    for local_batch, local_labels in training_generator:
        # Transfer to GPU
        local_batch, local_labels = local_batch.to(device), local_labels.to(device)

        # Model computations
        [...]
```

we need to explicitly transfer data to GPU memory

PyTorch DataLoader Capabilities

1. **Define a dataset to work with**: identifying where the data is coming from and how it should be accessed.

2. **Batch the data**: define how many training or testing samples to use in a single iteration. Because data are often split across training and testing sets of large sizes, being able to work with batches of data can allow your training and testing processes to be more manageable.

3. **Shuffle the data**: PyTorch can handle shuffling data for you as it loads data into batches. This can increase representativeness in your dataset and prevent accidental skewness.

4. **Support multi-processing**: PyTorch is optimized to run multiple processes at once to make better use of modern CPUs and GPUs and to save time in training and testing your data.
   The DataLoader class lets you define how many workers should go at once.

5. **Merge datasets**: optionally, PyTorch also allows you to merge multiple datasets together. While this may not be a common task, having it available is a great feature.

6. **Load data directly on CUDA tensors**: because PyTorch can run on the GPU, you can load the data directly onto the CUDA before they're returned.

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

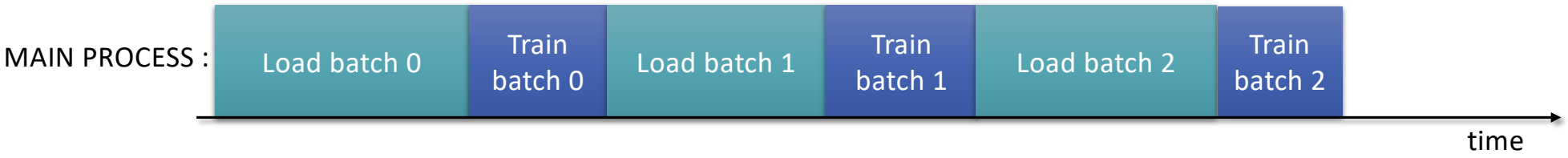
- Useful Parameters:
  - *batch_size:* batch size
  - *sampler:* define which sampler to use
  - *pin_memory:* copy into CUDA pinned memory before returning the data
  - *shuffle:* reshuffle data at each epoch
- Available *samplers:*
  - *SequentialSampler*
  - *RandomSampler*
  - *SubsetRandomSampler*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

For data loading, passing pin_memory=True to a DataLoader will automatically put the fetched data Tensors in pinned memory, and thus enables faster data transfer to CUDA-enabled GPUs.

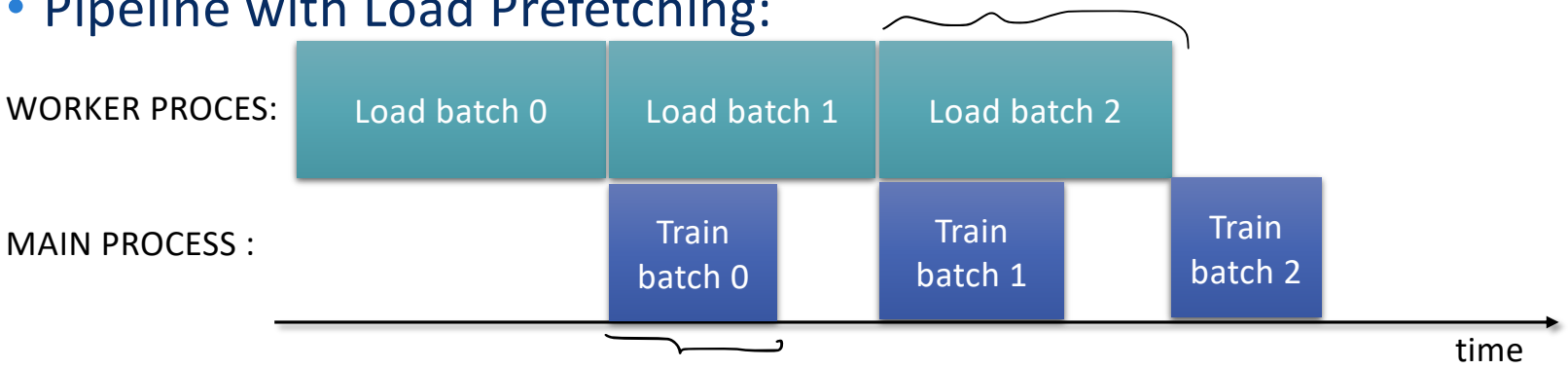
to make sure that this memory chunk is not paged out.

Data Prefetching

- Normal pipeline:



- Pipeline with Load Prefetching:



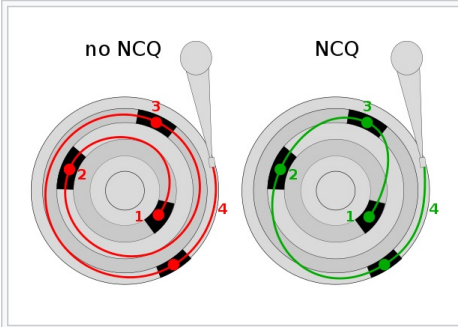
Load time still limiting factor. Can we do better?

Disk Performance Characteristics

- IOPS: IO-ops/sec
  - read/write
  - random/sequential
- Throughput = 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
  - Tagged Command Queueing (TCQ) and Native Command Queueing (NCQ) techniques

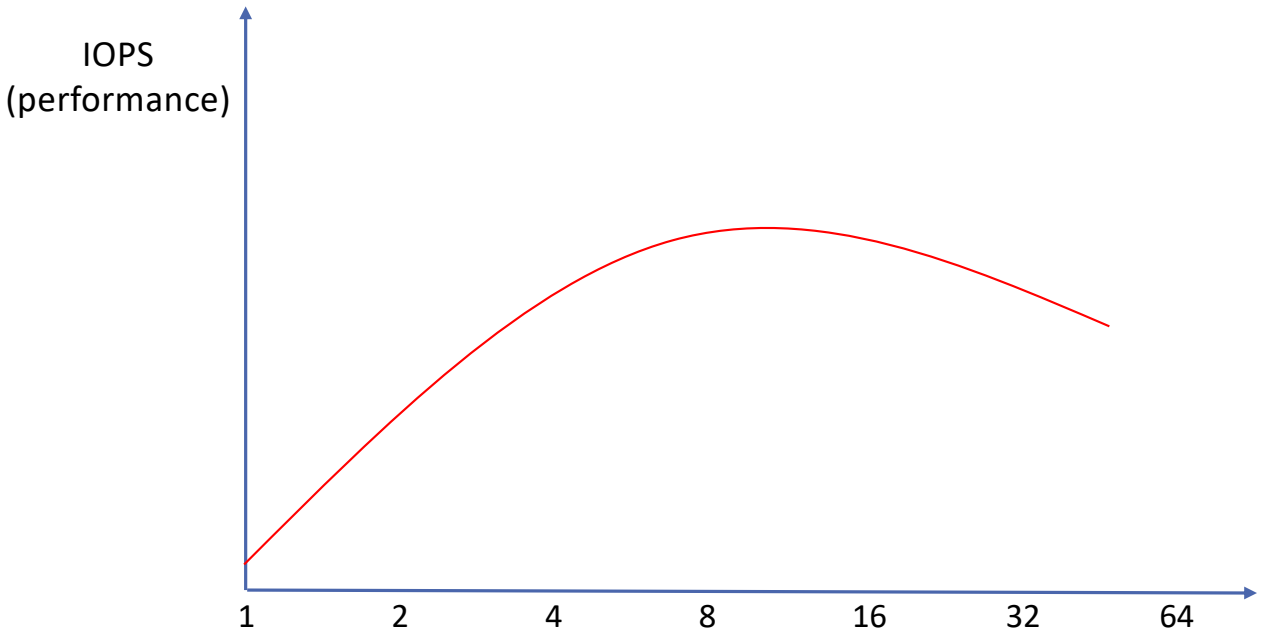
as our disk scheduler can optimize i/o tasks by looking at all the requests in the queue and processing them in a way such that overhead is minimum

In a file system, a *block* is the largest contiguous amount of disk space that can be allocated to a file and the largest amount of data that can be transferred in a single I/O operation. The block size determines the maximum size of a read request or write request that a file system sends to the I/O device driver.



disk optimization technique

Disk Performance Characteristics



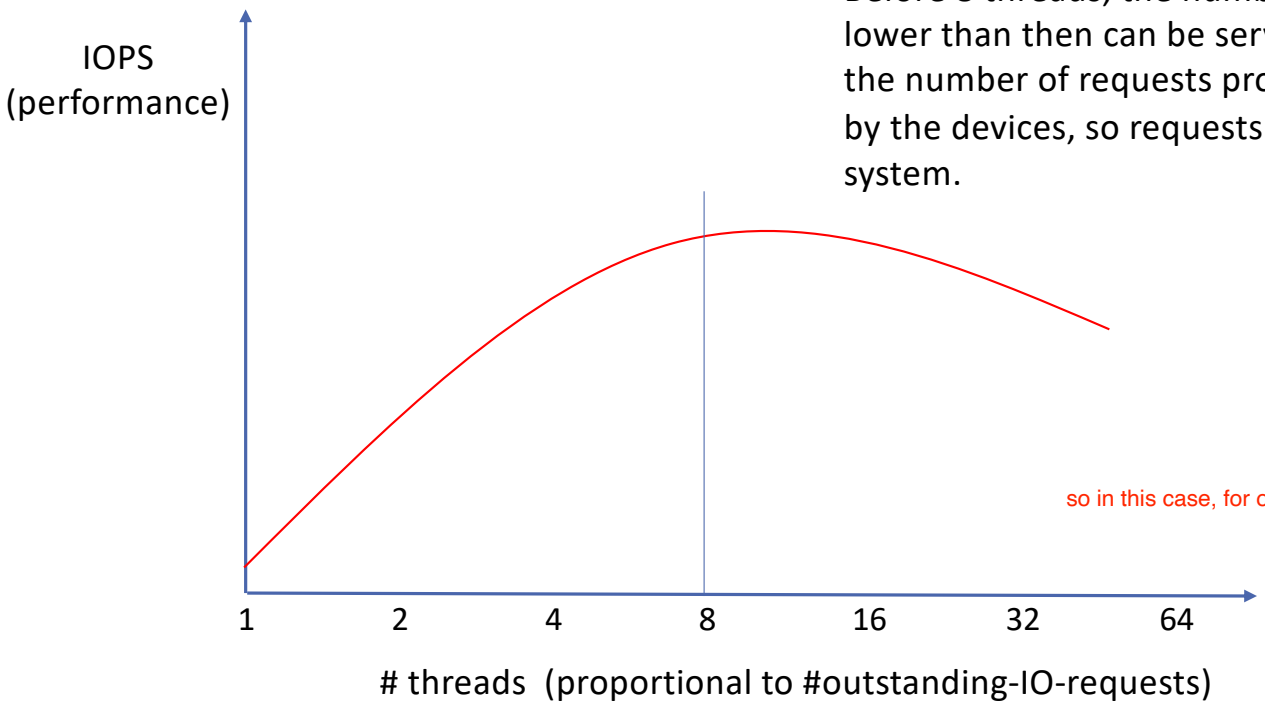
Number of requests in the queue

each thread handling one IO request

this means, just incerasing workers in data loader will not lead to increase in throughput always

every worker is fetching data independently

Disk Performance Characteristics



Before 8 threads, the number of IO requests per second is lower than then can be served by the device. After 8 threads the number of requests produced is higher than can be served by the devices, so requests queue up and slow down the system.

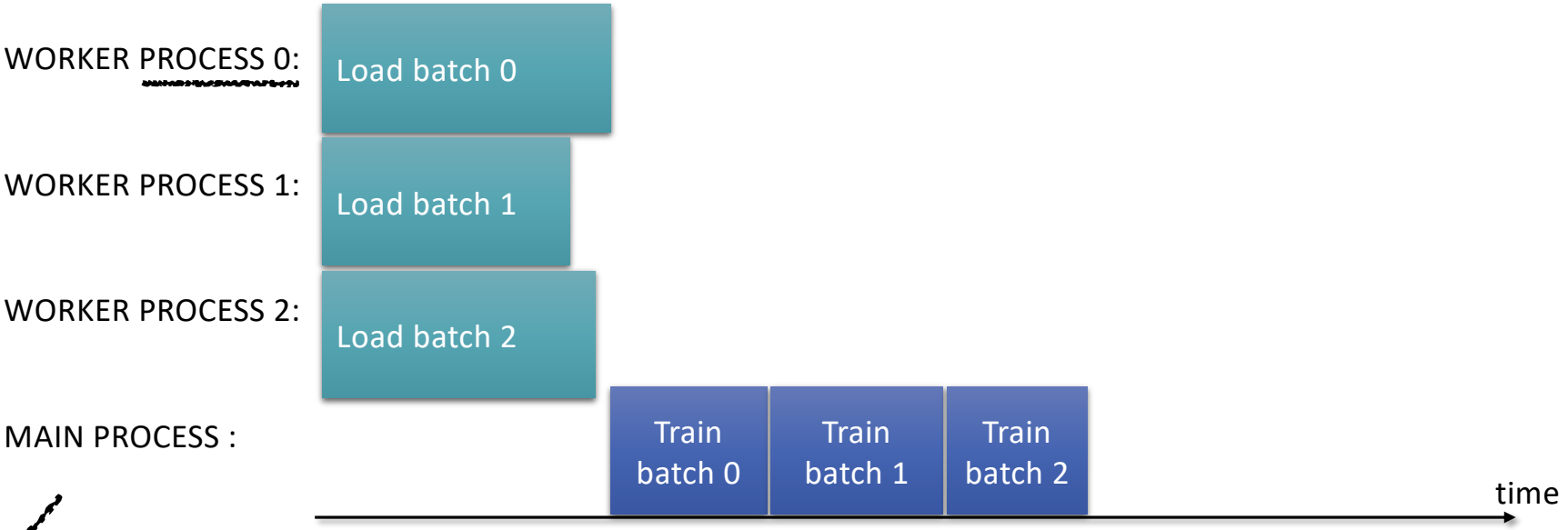
this has to be measured emperically

so in this case, for optimal use, we can have 8 workers in our dataloader

we have to emperically decide optimal number of workers for our use case.

Multi-threaded Data Prefetching

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



- Prefetching limitations: buffer memory, outstanding I/O requests limit, predicting next needed address

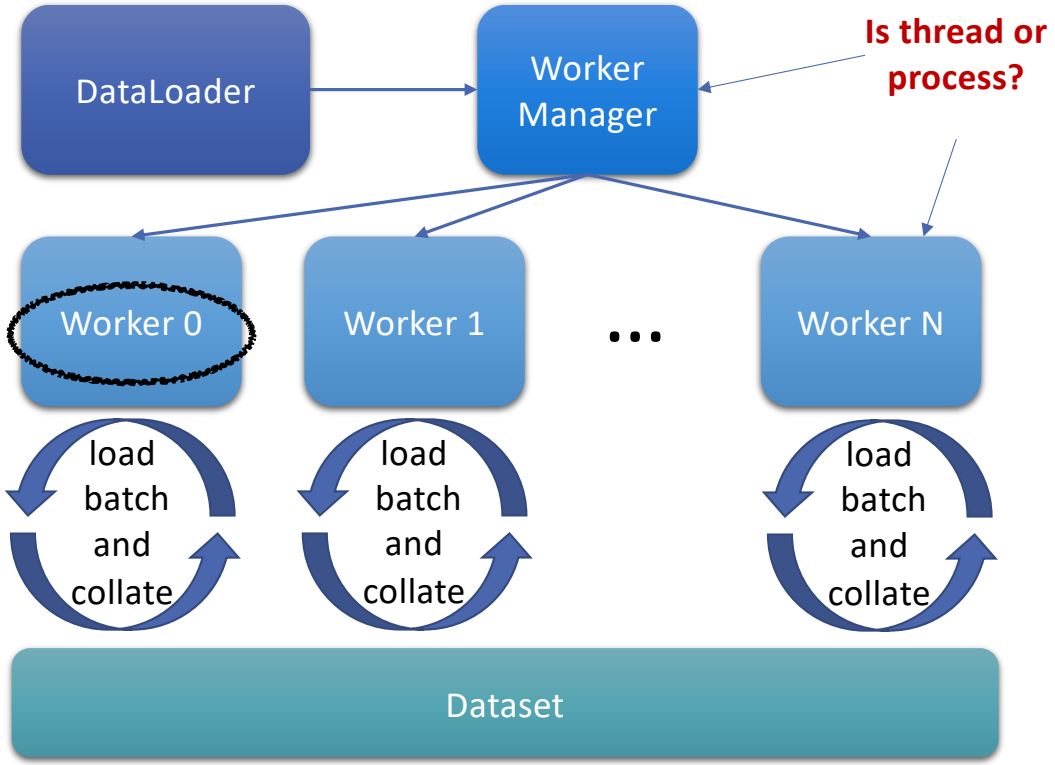
thread share memory but process dont

since thread share memory, there's a need of synchronization (to access share memory) - this is an overhhead

And we dont want this overhead with our workers, as we want them to work quickly and independently

torch.utils.data.Dataloader Architecture

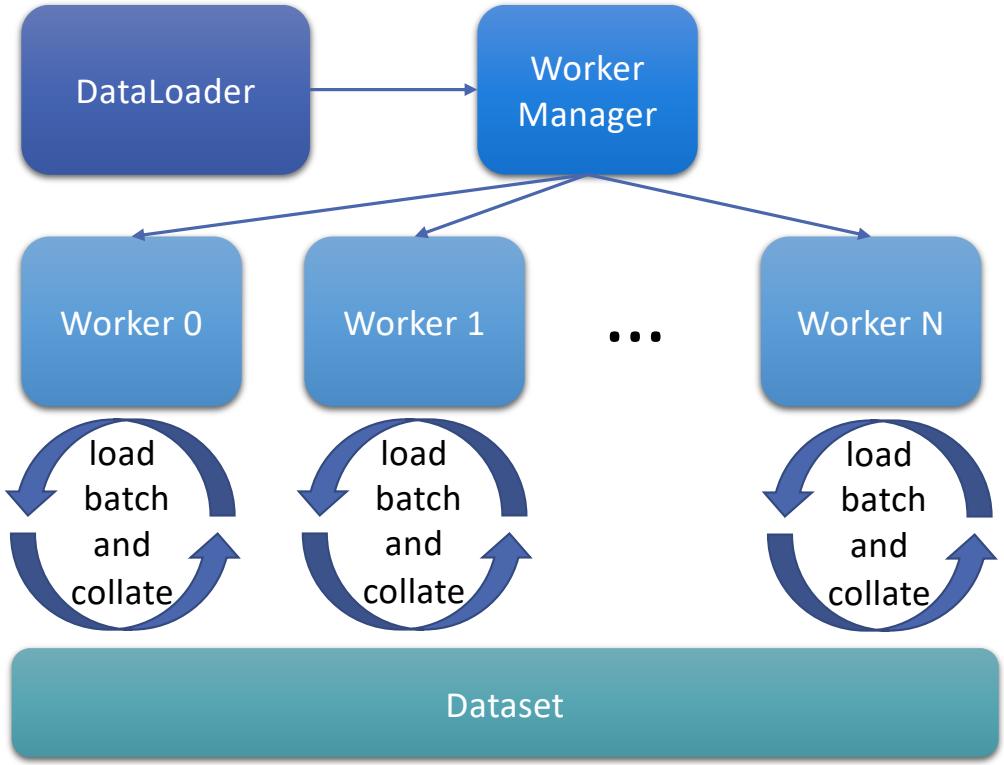
- DataLoader (main process):
  - Requests a set of batches based on *sampler*
- WorkerManager:
  - Dispatch batches to workers
- Workers :
  - Load all samples in each batch
  - Collate batches:
    - Build a batch Tensor with samples
- Synchronization:
  - Producer/Consumer using shared queues and Signals/Exceptions



Thats why - worker's are processes. Worker manager is a thread.

torch.utils.data.Dataloader Architecture

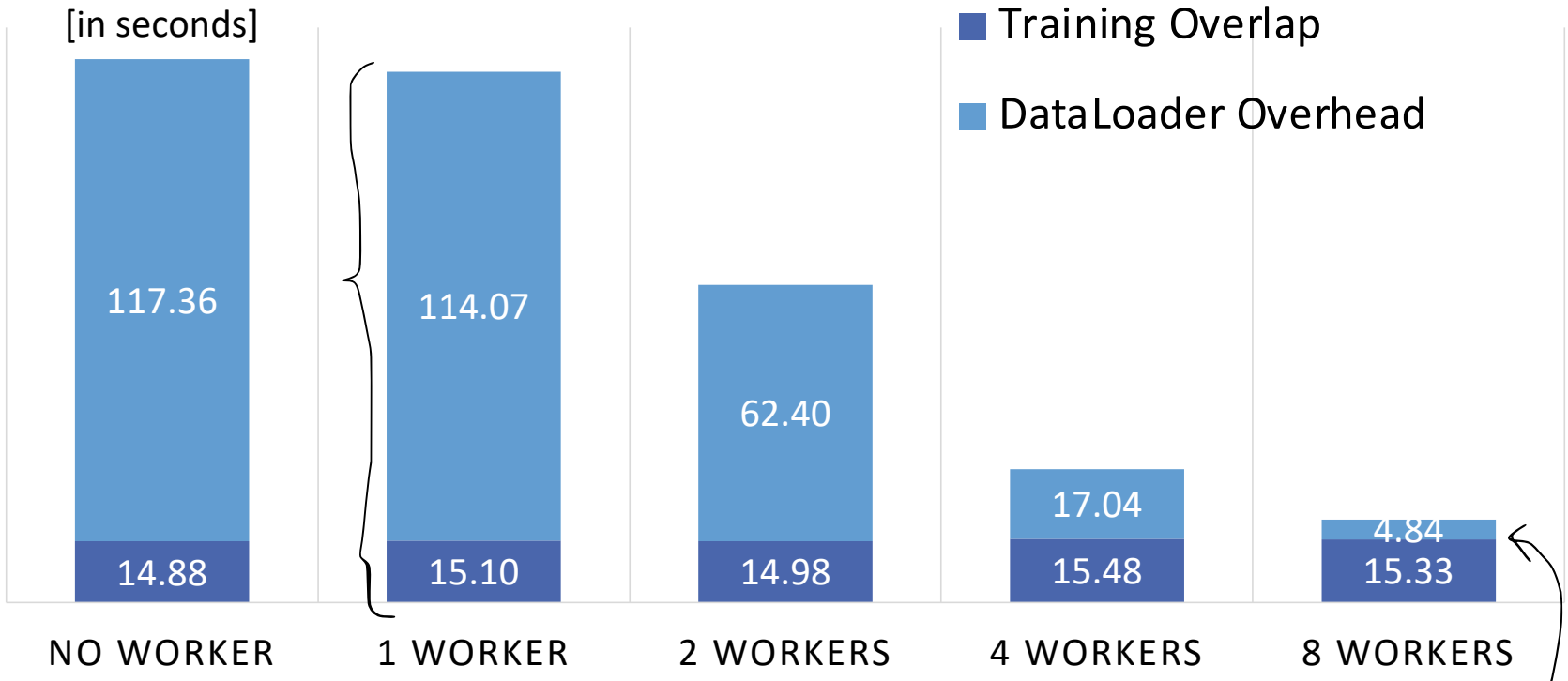
- DataLoader (main process):
  - Requests a set of batches based on *sampler*
- WorkerManager (thread):
  - Dispatch batches to workers
- Workers **(processes):**
  - Load all samples in each batch
  - Collate batches:
    - Build a batch Tensor with samples
- Synchronization:
  - Producer/Consumer using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

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

for dataloading

Lesson Key Points

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

Acknowledgements

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