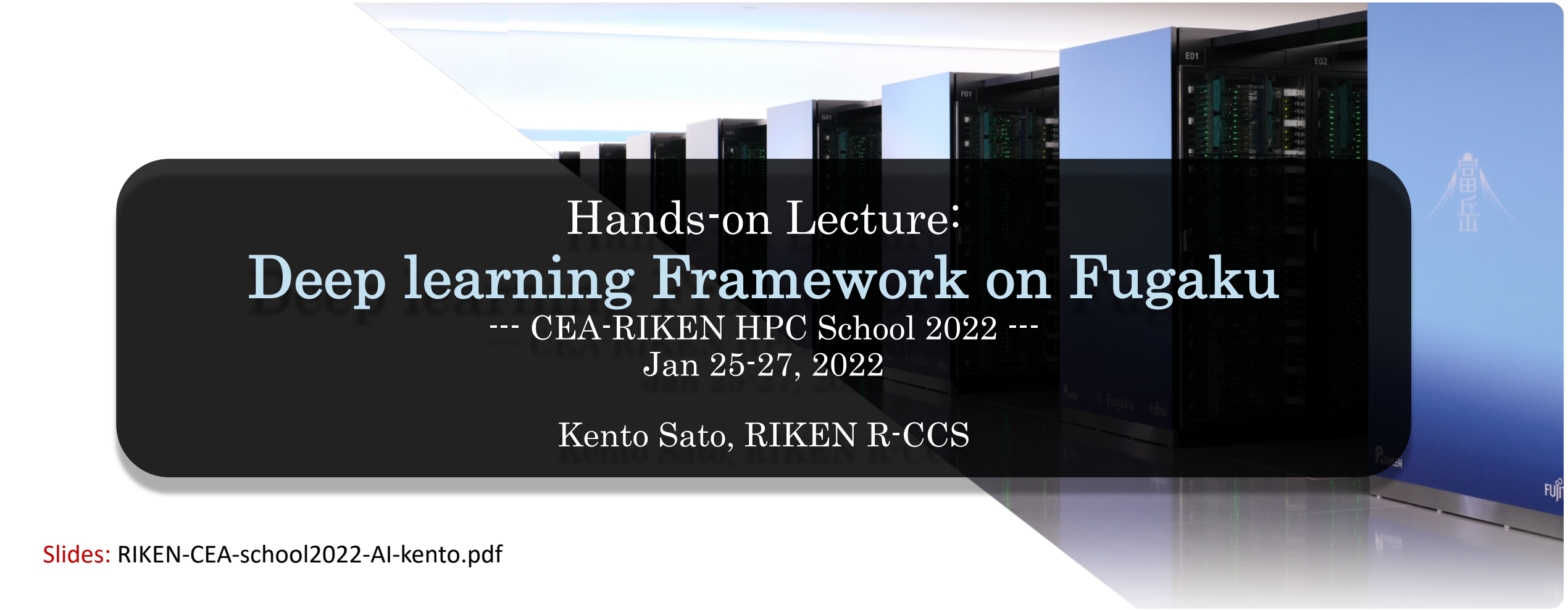


Hands-on Lecture:
Deep learning Framework on Fugaku

--- CEA-RIKEN HPC School 2022 ---
Jan 25-27, 2022

Kento Sato, RIKEN R-CCS

Slides: RIKEN-CEA-school2022-AI-kento.pdf

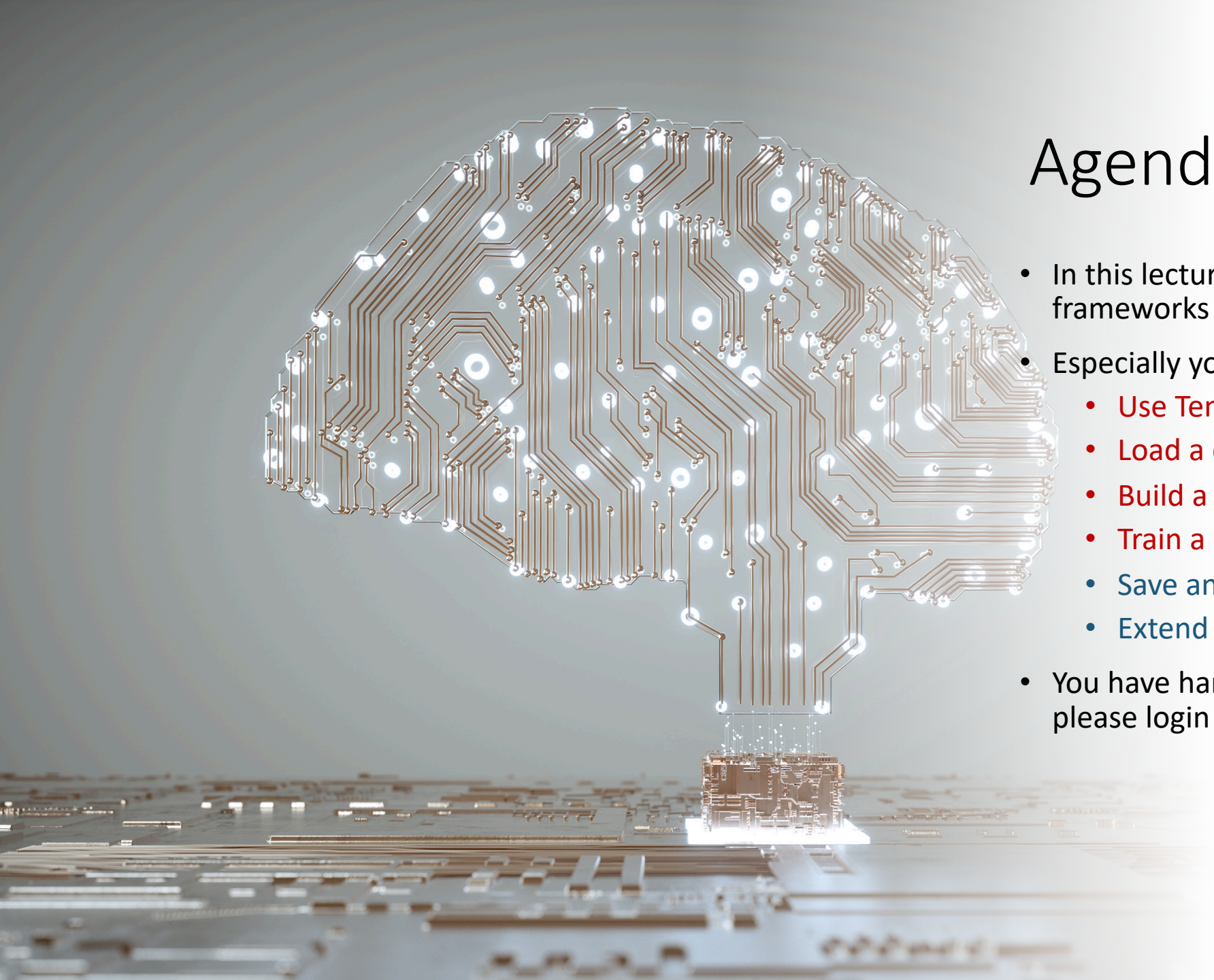
Please run this command to reserve an intaractive node during the break:

```
$ pjsub --interact -L "node=1" -L "rscgrp=int" -L "elapse=2:00:00" --sparam "wait-time=6000" --mpi "proc=8"
-x "PJM_LLIO_GFSCACHE=/vol0004"
```

Self-introduction

- Kento Sato (佐藤賢斗)
- Team leader of High Performance Big Data Research team at RIKEN R-CCS

https://www.hpbd.r-ccs.riken.jp



Agenda

- In this lecture, you learn how to use one of AI frameworks (PyTorch) on Fugaku
- Especially you learn basics of how to
  - Use Tensor operations on PyTroch/Python
  - Load a dataset with dataloaders
  - Build a neural network
  - Train a neural netowork
  - Save and load a trained neural network
  - Extend your code for distrubuted training
- You have hands-on exercise during this lecture, please login to Fugaku !

Preparation for hands-on

- Login to Fugaku

```
$ ssh <user name>@fugaku.r-ccs.riken.jp
```

- Download lecture materials

```
$ mkdir cea-riken-school-2022
$ cd cea-riken-school-2022
$ git clone https://github.com/kento/hpc_tools_examples.git
$ cd ./hpc_tools_examples/deep_learning/pytorch_fugaku/00_basics
```

- Reserve an inteactive node

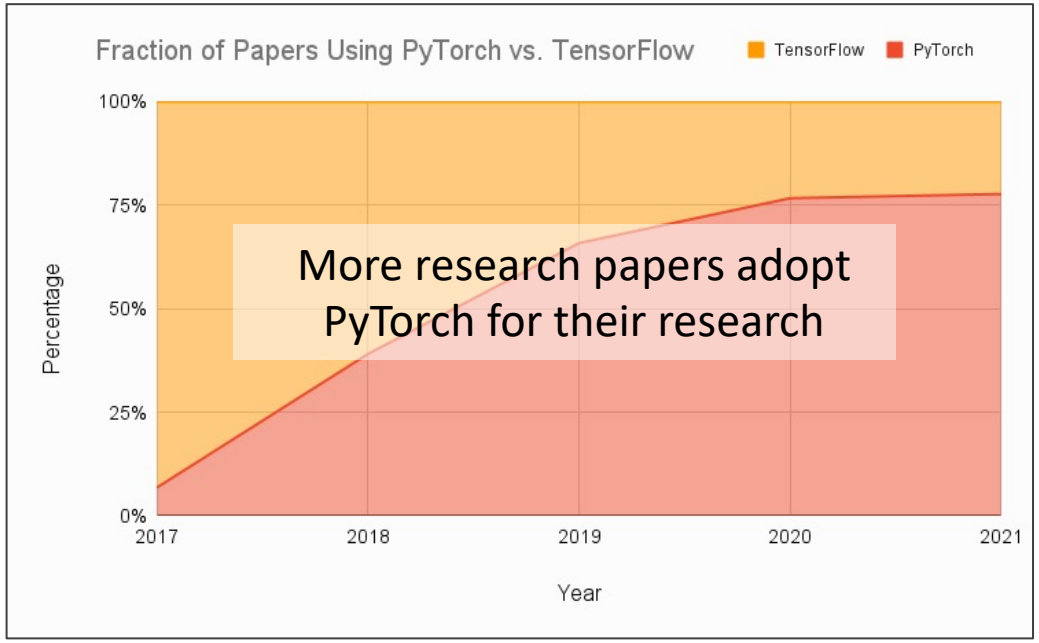
```
$ pjsub --interact -L "node=1" -L "rscgrp=int" -L "elapse=3:00:00" --sparam "wait-time=600" --mpi "proc=8”
-x "PJM_LLIO_GFSCACHE=/vol0004"
```

  - Today, we use an interactive node, please reserve your interactive node !
  - An iteractive node is a compute node which you can keep using as if it is your own compute node without submiting a job to the batch scheduler

PyTorch vs. TensorFlow

- Today, we use PyTorch
- Both PyTorch and TensorFlow are very mature frameworks, and their core Deep Learning features overlap significantly
- PyTorch has become the *de facto "research framework"* after its explosive adoption by the research community

PyTorch



Source: https://www.assemblyai.com/blog/pytorch-vs-tensorflow-in-2022/

Set environemnt valiables

- Set environment valiables
  - By running a script “fugaku_setup.sh”

```
$ . ../fugaku_setup.sh
```

  - By seting one by one

```
$ export PYTORCH_PATH=/home/apps/oss/PyTorch-1.7.0
$ export LD_LIBRARY_PATH=${PYTORCH_PATH}/lib:${LD_LIBRARY_PATH}
$ export PATH=${PYTORCH_PATH}/bin:${PATH}
```

Tensors

- Tensors are a specialized data structure that are very similar to arrays and matrices
- In PyTorch, you use tensors to encode the inputs and outputs of a model, as well as the model's parameters
- Tensors are like NumPy's ndarrays (e.g., Matrix operations between arrays)
  - Except that tensors can run on GPUs or other hardware accelerators with the same interface

```
# Directly from data
import torch
data = [[1, 2],[3, 4]]
x_data = torch.tensor(data)
print (data)
print (x_data)
```

```
# From a NumPy array
import numpy as np
np_array = np.array(data)
x_np = torch.from_numpy(np_array)
print (data)
print (x_np)
```

```
print (x_np + x_np)
print (x_np * x_np)
```

00_basics/00_tensors.py

Exercise

- Compute following tensor operations with Tensor data type

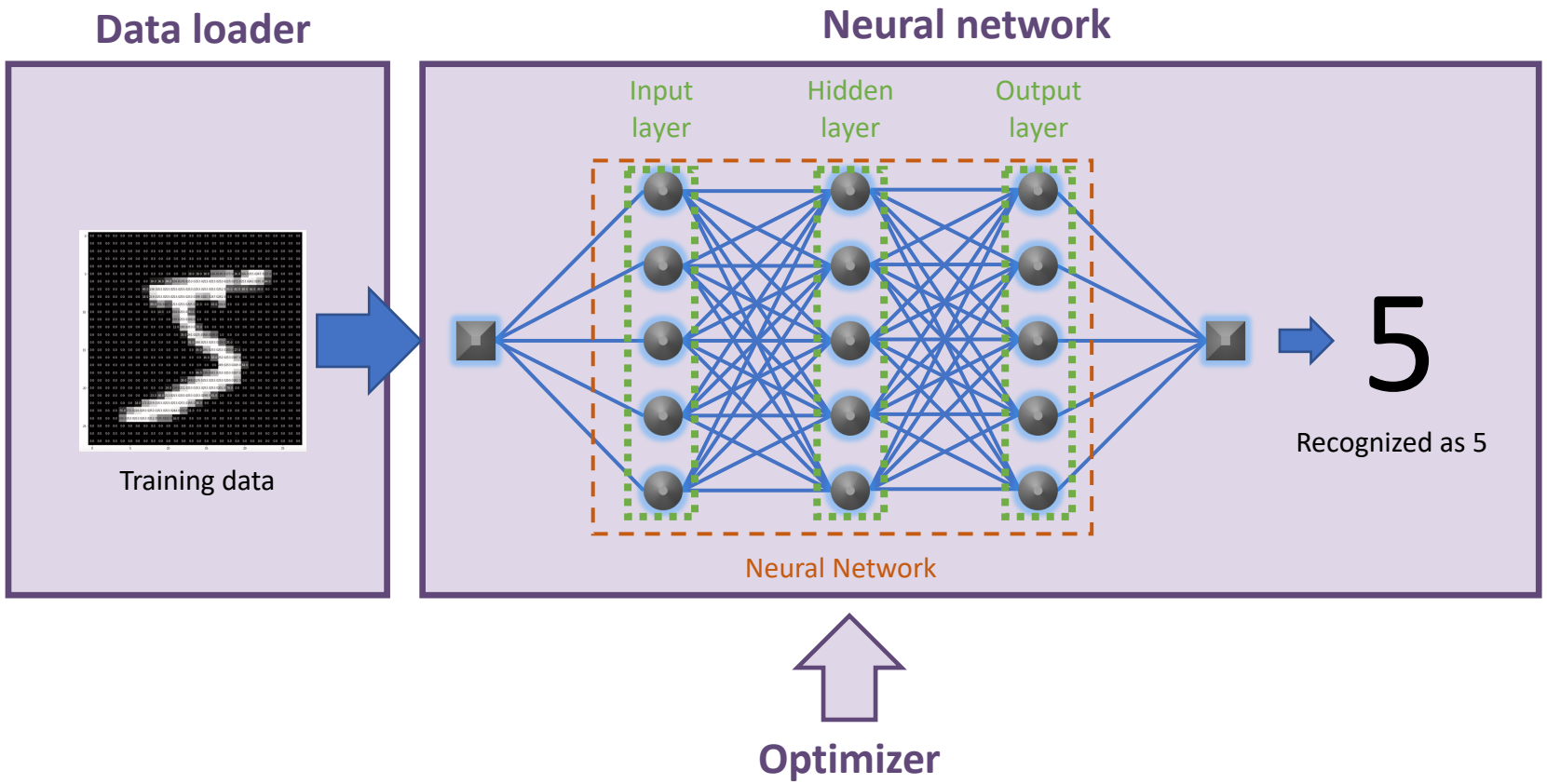
1. $\begin{pmatrix} 1 & 2 & 3 \\ 2 & 3 & 4 \\ 3 & 4 & 5 \end{pmatrix} + \begin{pmatrix} 5 & 4 & 3 \\ 4 & 3 & 2 \\ 3 & 2 & 1 \end{pmatrix} = ?$

2. $\begin{pmatrix} 1 & 2 & 3 \\ 2 & 3 & 4 \\ 3 & 4 & 5 \end{pmatrix} * \begin{pmatrix} 5 & 4 & 3 \\ 4 & 3 & 2 \\ 3 & 2 & 1 \end{pmatrix} = ?$

Answer: 00_basics/00_tensors_ex.py

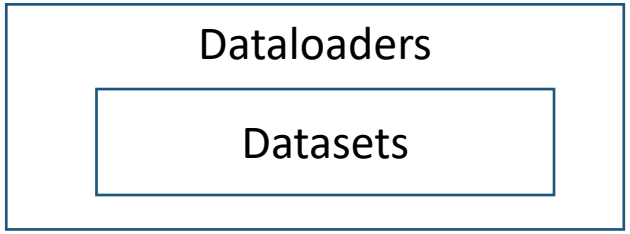
Basic concept of PyTorch

- Major three components
  - **Neural network**: Define a class and implement **neural networks** that you want to train
  - **Data loader**: Interface to **datasets** for training and test
  - **Optimizer**: Optimize paramenters of a **neural network** trained with **datasets**



Datasets and Dataloaders

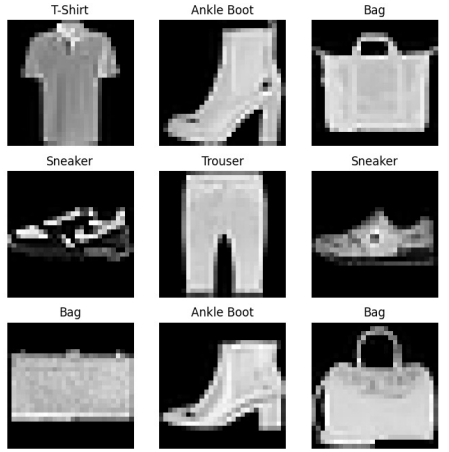
- Decoupling data processing code from model training code for better readability and modularity
- PyTorch provides two data primitives that allow you to use pre-loaded datasets as well as your own data
  - Dataset: torch.utils.data.Dataset
    - Stores the data and its corresponding labels
    - Provides a number of pre-loaded datasets (such as MNIST) for prototyping and benchmarking your model
      - Image Datasets: https://pytorch.org/vision/stable/datasets.html
      - Text Datasets: https://pytorch.org/text/stable/datasets.html
      - Audio Datasets: https://pytorch.org/audio/stable/datasets.html
  - Dataloader: torch.utils.data.Dataloader
    - Wrapper wraps an iterable around the Dataset to enable easy access to the samples.



Loading a Dataset:

```
import torch
from torch.utils.data import Dataset
from torchvision import datasets
from torchvision.transforms import ToTensor

training_data = datasets.FashionMNIST(root="../data", train=True, download=True,
transform=ToTensor())

test_data = datasets.FashionMNIST(root="../data", train=False, download=True,
transform=ToTensor())
```

- **root** (*string*) – Root directory of dataset where you want to cache
- **train** (*bool, optional*) – If True, creates dataset from training.pt, otherwise from test.pt.
- **download** (*bool, optional*) – If true, downloads the dataset from the internet and puts it in root directory. If dataset is already downloaded, it is not downloaded again.
- **transform** (*callable, optional*) – A function/transform that takes in an PIL image and returns a transformed version. E.g, transforms.RandomCrop
  - Please specify ToTensor() → It transforms an image into tensor datatype

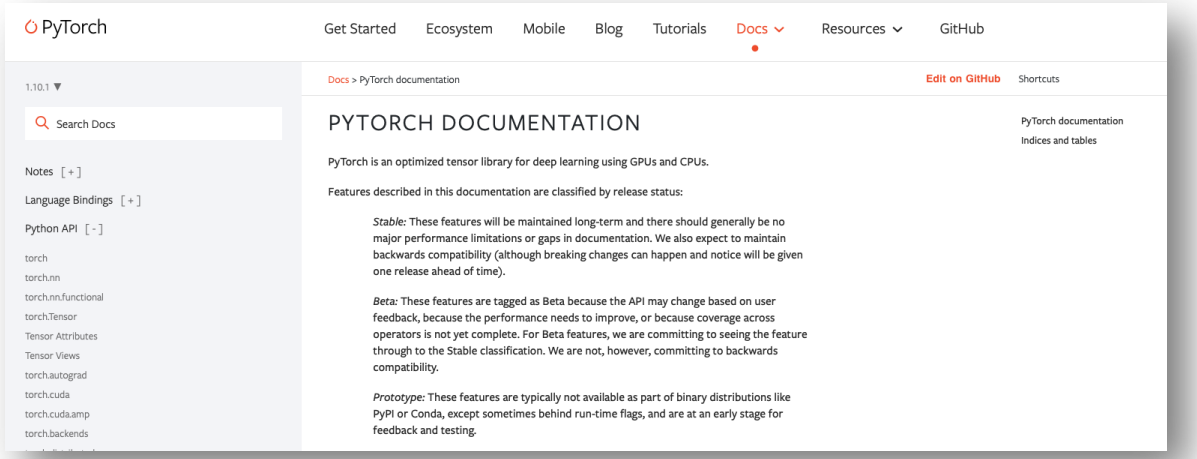


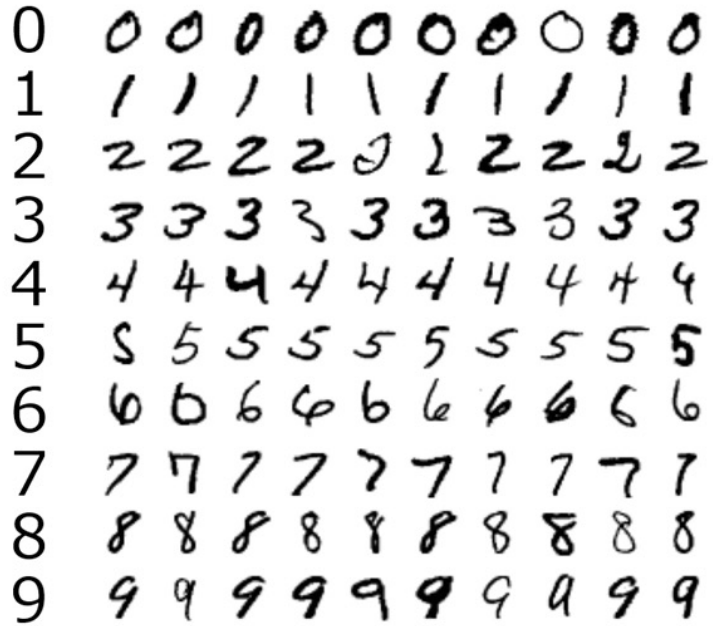
Fashion-MNIST

Exercise

- Modify 01_datasets.py to download MNIST instead of FashionMNIST by refering to PyTorch API documents
- MNIST is a dataset of handwritten digits that is commonly used for training various image processing systems

- PyTroch API documens
  - https://pytorch.org/vision/stable/datasets.html



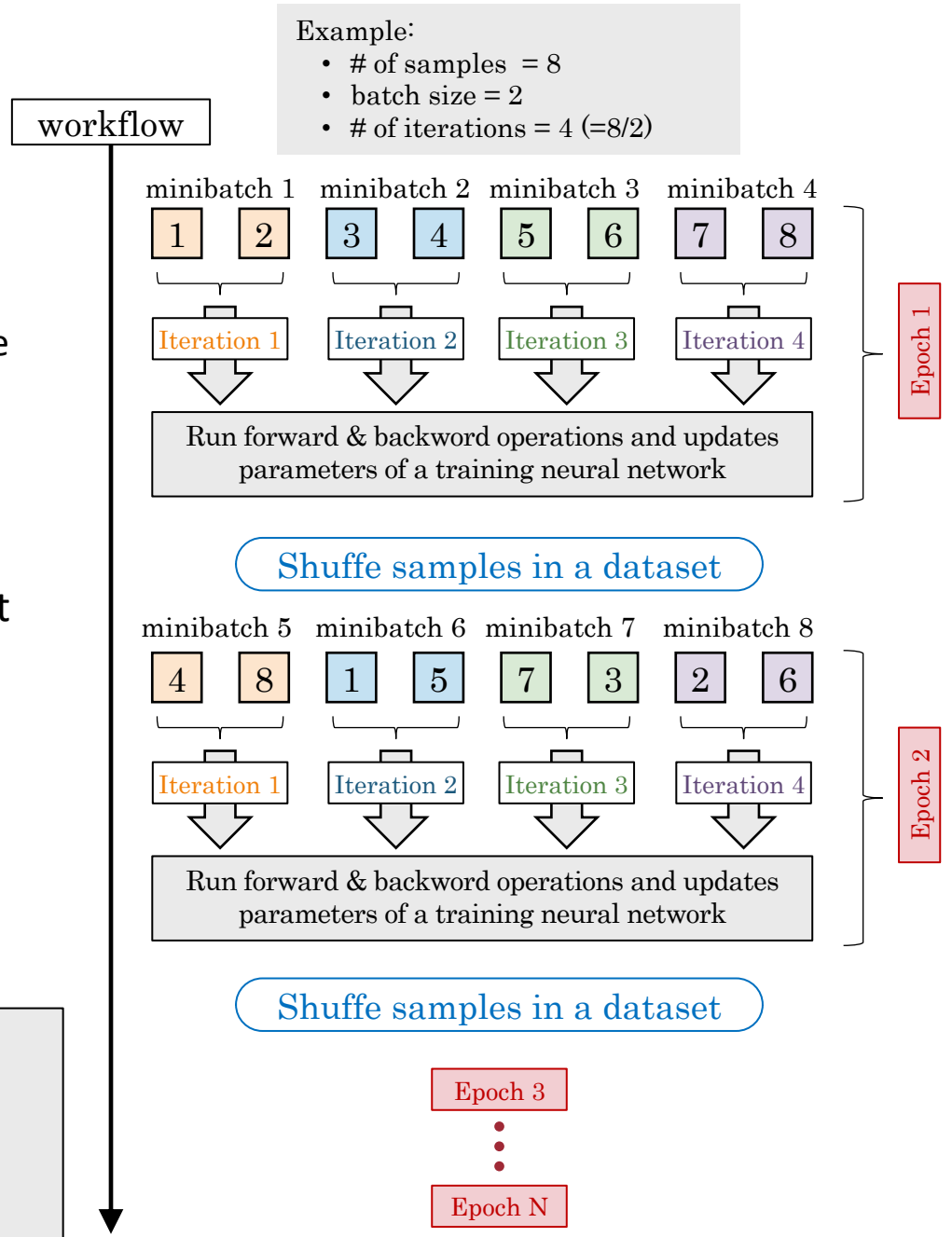


MNIST

Workflow for training NN

- Workflow
  - **Step 1:** It retrives multiple samples (“mini-batch” or “batch”) from a dataset
    - A minibatch refers to equally sized subsets of the dataset over which the gradient is calculated, and weights are updated
    - The number of simpales in a mini-batch is called “batch size”
  - **Step 2:** Run forward and backword operations, then updatees parameters of a training neural network
  - **Step 3:** Once all the samples are fed to the training neural network, it shuffuls samples in a dataset and repeat from step 1
- Epoch
  - One Epoch is when an ENTIRE dataset is passed forward and backward through the neural network only ONCE
- Iterations
  - Iterations is the number of batches needed to complete one epoch

- Epoch 1
  - Iteration 1
  - …
  - Iteeration N
- Epoch 2
  - Iteration 1
  - …
  - Iteration N
- Epoch 3
- …

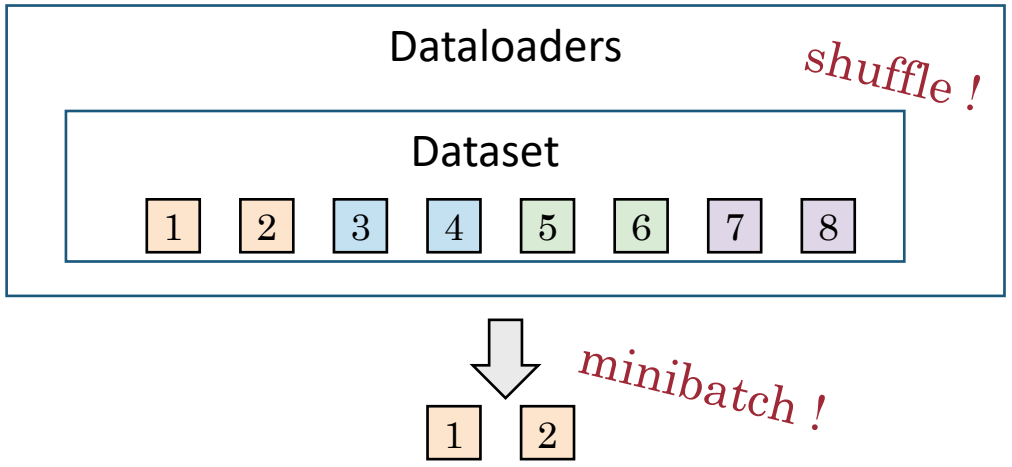


DataLoaders

- As explained in the previous slide, while training a model, we need to pass samples in a form of “mini-batches”, reshuffle the data at every epoch to reduce model overfitting
- The DataLoader abstracts this complexity for us in an easy API

Snippet (dataloaders)

```
from torch.utils.data import DataLoader

train_dataloader = DataLoader(dataset=training_data, batch_size=64, shuffle=True)
test_dataloader = DataLoader(dataset=test_data, batch_size=64, shuffle=True)
```



- **dataset** (*Dataset*) – dataset from which to load the data.
- **batch_size** (*int, optional*) – how many samples per batch to load (default: 1).
- **shuffle** (*bool, optional*) – set to True to have the data reshuffled at every epoch (default: False).

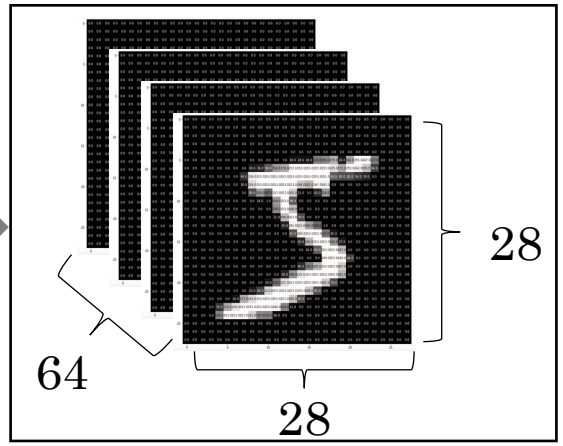
Preparing data for training with DataLoaders

- We have loaded that dataset into the DataLoader and can iterate through the dataset as needed
- Each iteration below returns a batch of ”train_features” and ”train_labels”
  - batch_size=64 features and labels respectively
- Because we specified shuffle=True, the data is shuffled after we iterate over all batches,

Snippet (dataloaders print)

```
# Display image and label
train_features, train_labels = next(iter(train_dataloader))
print(f"Feature batch shape: {train_features.size()}")
print(f"Labels batch shape: {train_labels.size()}")
img = train_features[0] # get the first image from mini-batch
label = train_labels[0] # get the first label
print(f"Label class: {label}")
```

train_features



test_features

[5, 8, 1, …, 3]

```
Feature batch shape: torch.Size([64, 1, 28, 28])
Labels batch shape: torch.Size([64])
Label: 8
```

Exercise

- Get the first MNIST minibatch from dataloaders and print out shape of the minibatch and the labels by extending “01_datasets_ex.py”

- Hint: Adding the code snippets to 01_datasets_ex.py

00_basics/01_datasets_ex.py

+

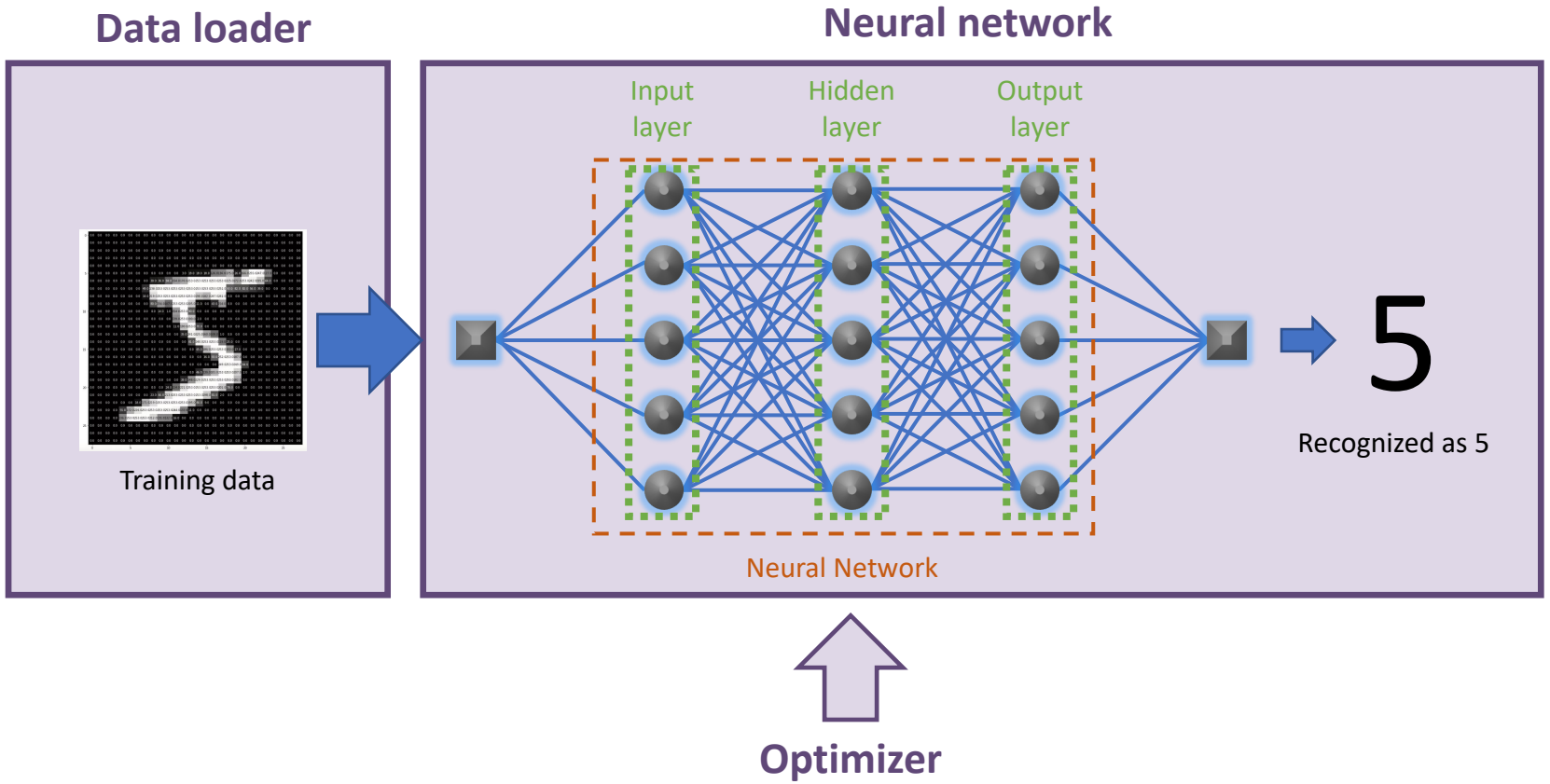
Snippet (dataloaders)

+

Snippet (dataloaders print)

Answer: 00_basics/02_dataloaders.py

Basic concept of PyTorch



Building a neural network

- Neural networks comprise of layers/modules that perform operations on data.
- The torch.nn namespace provides all the building blocks that you need to build your own neural network
- Every module in PyTorch subclasses the nn.Module
- A neural network is a module itself that consists of other modules (such as layers)
- This nested structure allows for building and managing complex architectures easily

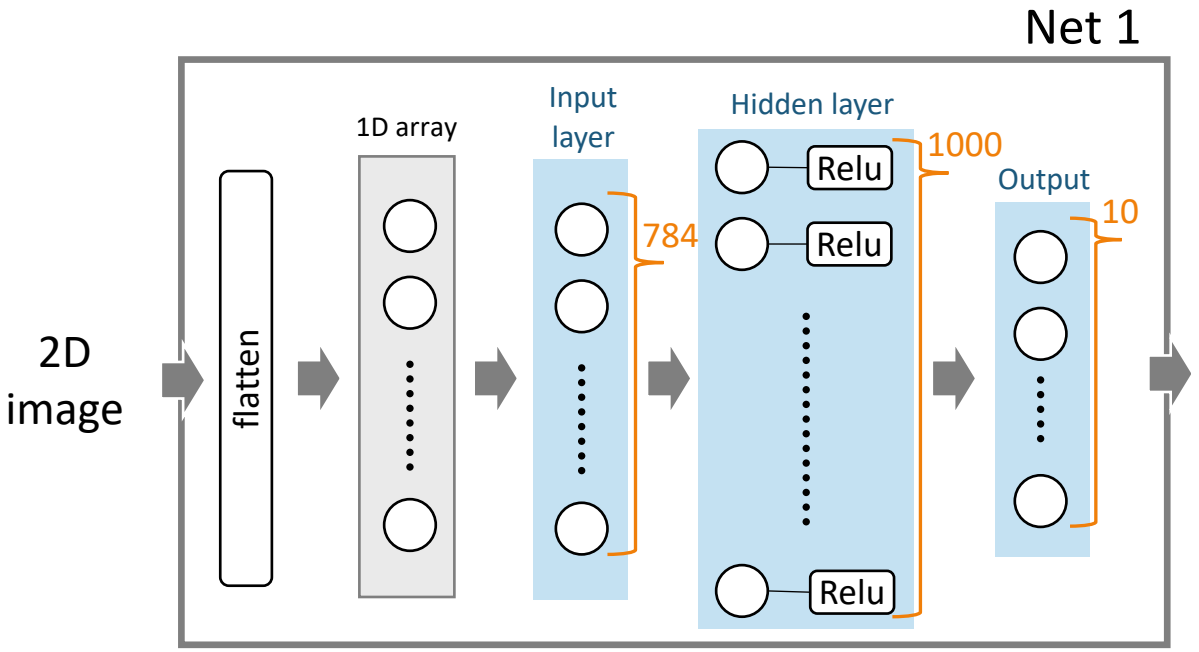
- Necessary modules to import:

Snippet (import nn & functioonal)

```
from torch import nn
from torch.nn import functional as F
```

Building a 3-layer neural network

- Now, we define our neural network (**Net1**) by subclassing nn.Module and initialize the neural network layers in (1) __init__
  - In __init__, we initialize layers
- Every nn.Module subclass implements the operations on input data in the (2) forward method
  - In forward, we codify how to connect the initialized layers



Snippet (Building Net1)

```
class Net1(nn.Module):
  def __init__(self):
    super(Net1, self).__init__()
    self.flatten = nn.Flatten()
    self.fc1 = nn.Linear(28 * 28, 1000)
    self.fc2 = nn.Linear(1000, 10)

  def forward(self, x):
    x = self.flatten(x)
    x = self.fc1(x)
    x = F.relu(x)
    x = self.fc2(x)
    return x
```

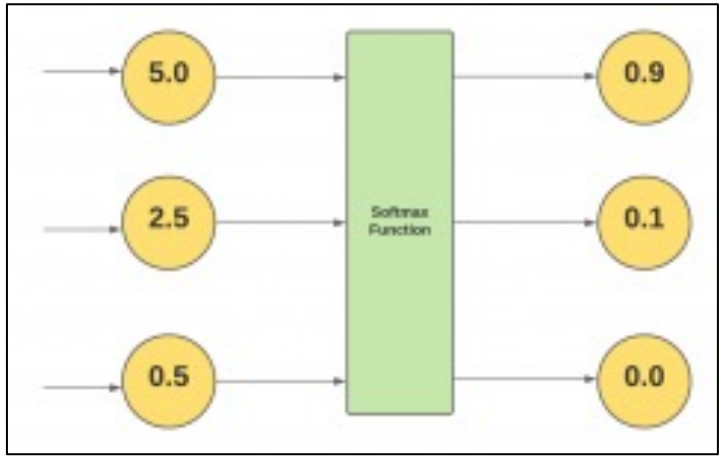
1. __init__: init layers

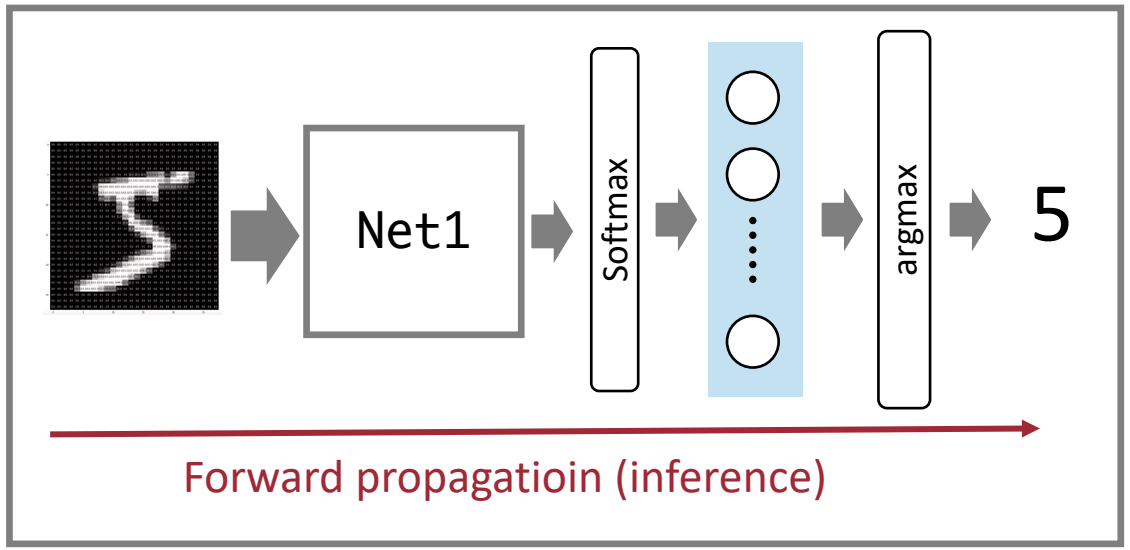
2. forward: Connect layers

Running inference of Net 1

- To run inference with the built model, we pass the input data to the model
  - This id done by executing the model's forward
- Calling the model on the input returns a 10-dimensional tensor with raw predicted values for each class
- We get the prediction probabilities by passing it through an instance of the "nn.Softmax" module.
- To get the inference result, we get the index whose prediction probability is highest by calling "argmax"

Softmax example





Snippet (Running inference)

```
model = Net1()
print(model)
logits = model(img)
pred_probab = nn.Softmax(dim=1)(logits)
img_pred = pred_probab.argmax(1)
print(f"Predicted class: {img_pred}")
```

$$\text{Softmax}(x_i) = \frac{\exp(x_i)}{\sum_j \exp(x_j)}$$

Exercise

- Build Net1 and run inference of the untrained Net1 on the first image of MNIST by extending “02_dataloaders.py”

- Hint: Adding the code snipets to 02_dataloaders.py

00_basics/02_dataloaders.py

+

Snippet (import nn & functioonal)

+

Snippet (Building Net1)

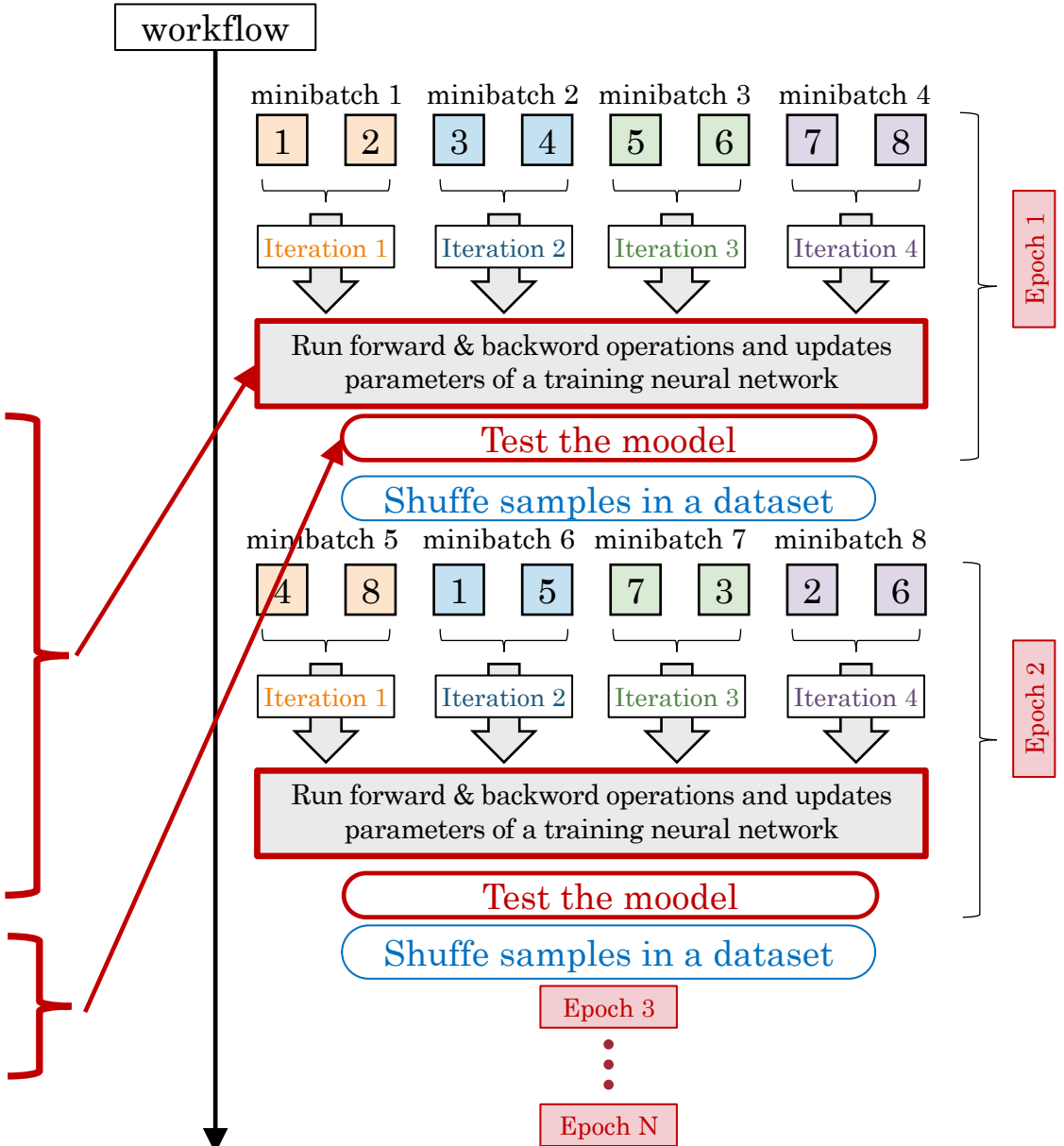
+

Snippet (Running inference)

Answer: 00_basics/03_build_NN.py

Training: Optimizaing Model Parameters

- Now, we have a model and data
- It's time to train and test our model by optimizing its parameters on our data
- Training a model is an iterative process
  - In each iteration with "training data", the model
    - **makes** a guess about the output (forward)
    - **calculates** the error in its guess (*loss*)
    - **collects** the derivatives of the error with respect to its parameters
    - **optimizes** these parameters using gradient descent
  - After each epoch, test the model accuracy with "test data"



Hyperparameters

- Hyperparameters are adjustable parameters that let you control the model optimization process
- Different hyperparameter values can impact model training and convergence rates
  - c.f) Hyperparameter tuning https://pytorch.org/tutorials/beginner/hyperparameter_tuning_tutorial.html
- We define the following hyperparameters for training:
  - **Number of Epochs:** the number times to iterate over the dataset
  - **Batch Size:** the number of data samples propagated through the network before the parameters are updated
    - Specify how many samples (e.g., image frames) to be trained at one time
  - **Learning Rate:** how much to update model parameters at each batch
    - Smaller values yield slow learning speed, while large values may result in unpredictable behavior during training

Snippet (Hyperparameters)

```
learning_rate = 1e-3
batch_size = 64
epochs = 1
```

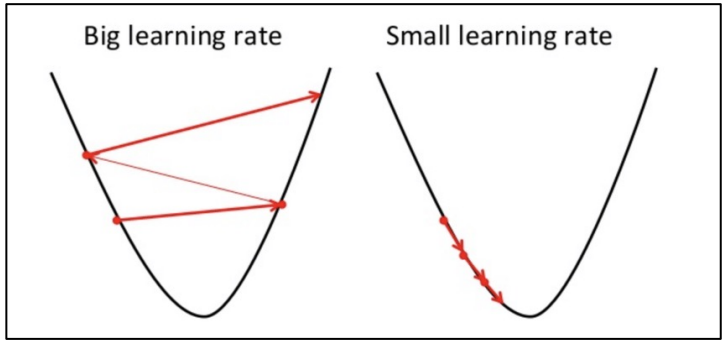
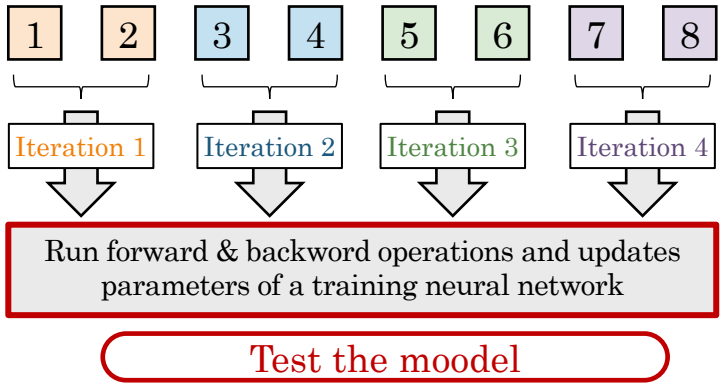


Image source: Nagy, Mohamed & Mohamed, Moataz & El-Sersy, Mohamed & Aloufi, Khalid. (2019). A Modified Method for Detecting DDoS Attacks Based on Artificial Neural Networks.

Optimization Loop (i.e., *Epoch loop*) : Training + Testing

- Once we set our hyperparameters, we can then train and optimize our model with an optimization loop
- Each iteration of the optimization loop is called an **epoch**.
- Each epoch consists of two main parts
  - **The Train Loop:** iterate over the training dataset and try to converge to optimal parameters.
  - **The Test Loop:** iterate over the test dataset to check if model accuracy is improving without overfitting to training data

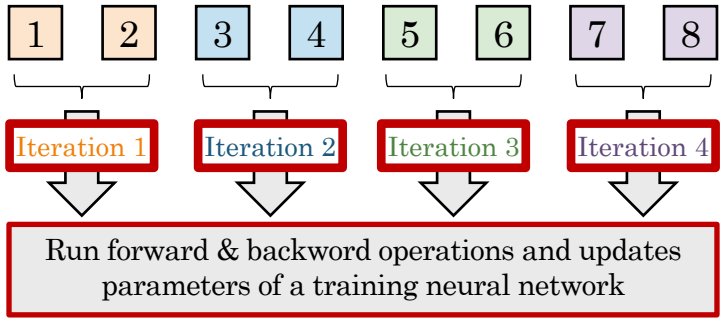


Snippet (Optimization loop)

```
for t in range(epochs):
    print(f"Epoch {t+1}\n-------------------------------")
    train_loop(train_dataloader, model, loss_fn, optimizer)
    test_loop(test_dataloader, model, loss_fn)
```

Train loop (i.e., *Iterations*)

- Inside the training loop, optimization happens in six steps
  - (1) Get a minibatch from dataloader in each loop
  - (2) Call model(X) to Compute forward propagation
  - (3) Call a loss function to compute loss
  - (4) Call optimizer.zero_grad() to reset the gradients of model parameters
    - Gradients by default add up; to prevent double-counting, we explicitly zero them at each iteration
  - (5) Backpropagate the prediction loss with a call to loss.backward()
    - PyTorch deposits the gradients of the loss w.r.t. each parameter
  - (6) Once we have our gradients, we call optimizer.step() to adjust the parameters by the gradients collected in the backward pass



Snippet (Training loop)

```
def train_loop(dataloader, model, loss_fn, optimizer):
  for batch, (X, y) in enumerate(dataloader): # (1)
    # Compute prediction and loss
    pred = model(X) # (2)
    loss = loss_fn(pred, y) # (3)
    #Backpropagation
    optimizer.zero_grad() # (4)
    loss.backward() # (5)
    optimizer.step() # (6)
```

Train loop (i.e., *Iterations*) with progress print

- Periodic prints will help you to know the progress of training

Snippet (Training loop with progress print)

```
def train_loop(dataloader, model, loss_fn, optimizer):
  size = len(dataloader.dataset)
  for batch, (X, y) in enumerate(dataloader):
    # Compute prediction and loss
    pred = model(X) # forward propagation
    loss = loss_fn(pred, y) # compute loss
    #Backpropagation
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
    if batch % 100 == 0:
      loss, current = loss.item(), batch * len(X)
      print(f"loss: {loss:>7f}
            [{current:>5d}/{size:>5d}]")
```

Loss Function

- When presented with some training data, our untrained network (Net1) is likely not to give the correct answer
- Loss function measures the degree of dissimilarity of obtained result to the target value, and it is the loss function that we want to minimize during training
- To calculate the loss we make a prediction using the inputs of our given data sample and compare it against the true data label value
- Common loss functions include:
  - nn.MSELoss (Mean Square Error) for regression tasks
  - nn.NLLLoss (Negative Log Likelihood) for classification
  - nn.CrossEntropyLoss combines nn.LogSoftmax and nn.NLLLoss
- We pass our model's output logits to nn.CrossEntropyLoss, which will normalize the logits and compute the prediction error

Snippet (Loss function)

```
loss_fn = nn.CrossEntropyLoss()
```

Cross Entropy Loss

$$H(p,q) = -\sum_{i} p(x) \log q(x)$$

Optimizer

- Optimization is the process of adjusting model parameters to reduce model error in each training step (i.e., minimizing a result of the loss function)
- **Optimization algorithms** define how this process is performed
- All optimization logic is encapsulated in the optimizer object
- Here, we use the SGD (Stochastic Gradient Descent) optimizer
  - Additionally, there are many different optimizers available in PyTorch such as ADAM and RMSProp, that work better for different kinds of models and data.
- We initialize the optimizer by registering the model's parameters that need to be trained, and passing in the learning rate hyperparameter.

Snippet (Optimizer)

```
optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)
```

Test loop

- **The Test Loop:** iterate over the test dataset to check if model accuracy is improving without overfitting to training data
- When testing, disabling gradient calculation is useful for inference by calling "torch.no_grad()", when you are sure that you will not run backpropagation
- It will reduce memory consumption for computations that would otherwise have requires_grad=True.

Snippet (Test loop)

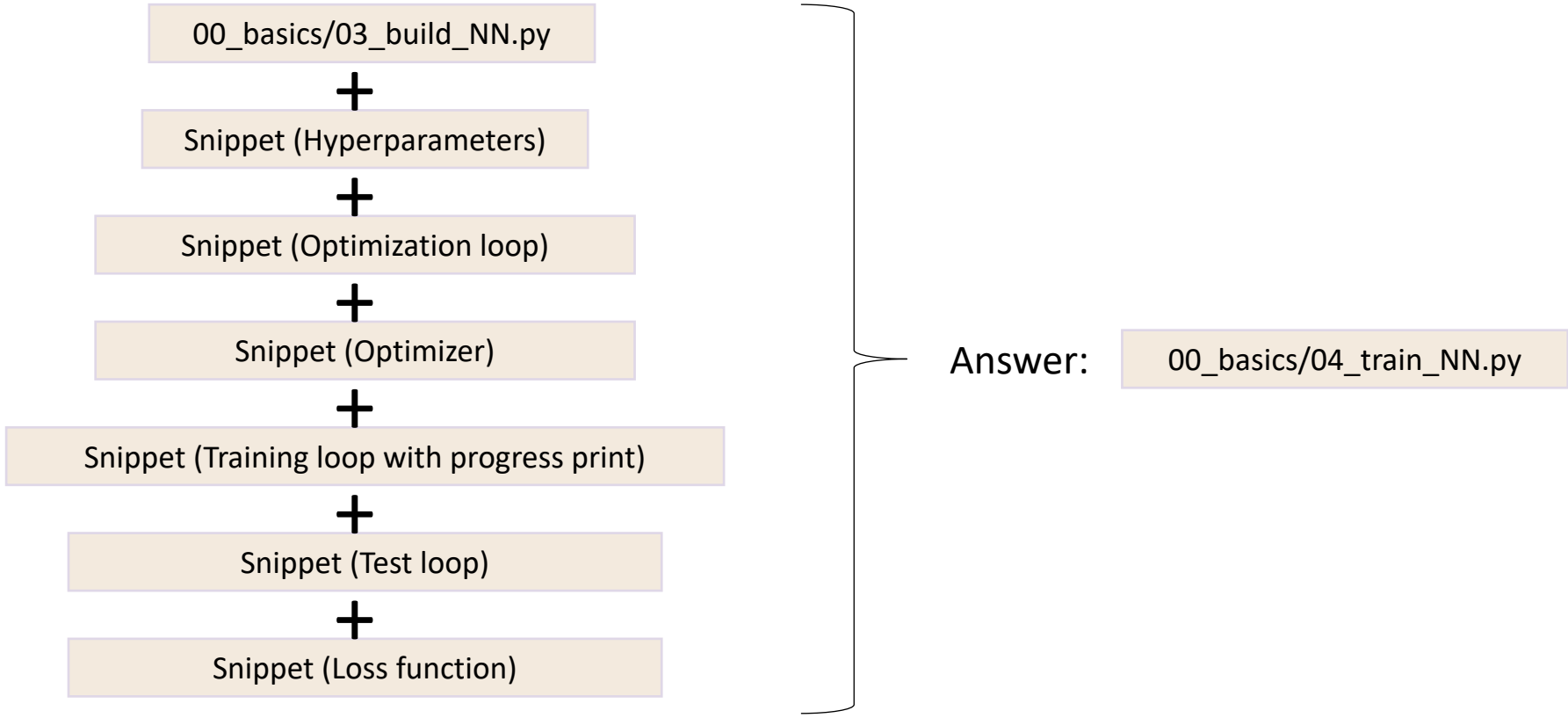
```
def test_loop(dataloader, model, loss_fn):
    size = len(dataloader.dataset)
    num_batches = len(dataloader)
    test_loss, correct = 0, 0
    with torch.no_grad():
        for X, y in dataloader:
            pred = model(X)
            test_loss += loss_fn(pred, y).item()
            correct += (pred.argmax(1) == y).type(torch.float).sum().item()
    test_loss /= num_batches
    correct /= size
    print(f"Test Error: \n Accuracy: {(100*correct):>0.1f}%, Avg loss: {test_loss:>8f} \n")
```

```
# item() returns a value of tensor when one
element
>>> x = torch.tensor([1.0])
>>> x.item()
1.0
```

Exercise

- Write a code to train Net1 with MNIST by extending 03_build_NN.py

- Hint: Add all the code snippets for training



Save and Load the model

- Once you fininshed training the model, you need to persist its model stat with saving, loading and running model predictioin
  - **Save**: PyTorch models store the learned parameters in an internal state dictionary, called ”state_dict”. These can be persisted via the torch.save method:
  - **Load**: To load model weights, you need to create an instance of the same model first, and then load the parameters using load_state_dict() method.

Snippet (save model)

```
# Saving model
torch.save(model.state_dict(), "Net1_weights.pth")
print("Saving Done!")
```

Snippet (load model)

```
# Load Model
model.load_state_dict(torch.load('Net1_weights.pth'))
model.eval()
print("Loading model Done !")
```

Exercise: Save the model

- Save the trained model by extending “04_train_NN.py”
- Hint: Add the code snipet to 04_train_NN.py

00_basics/04_train_NN.py

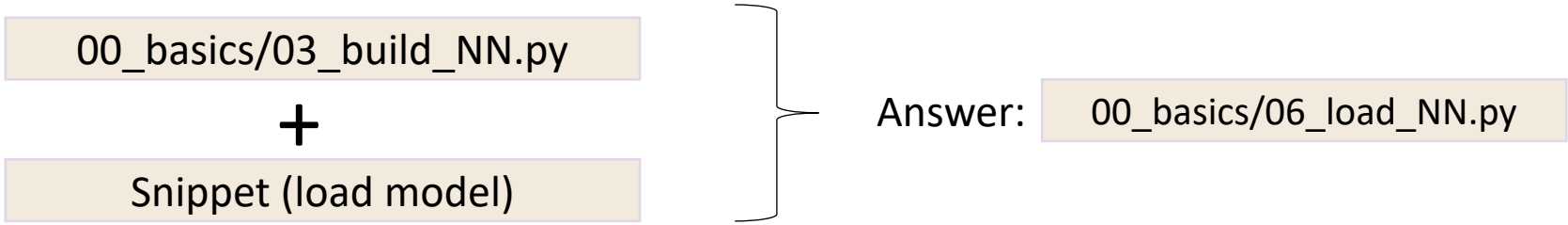
+

Snippet (save model)

Answer: 00_basics/05_save_NN.py

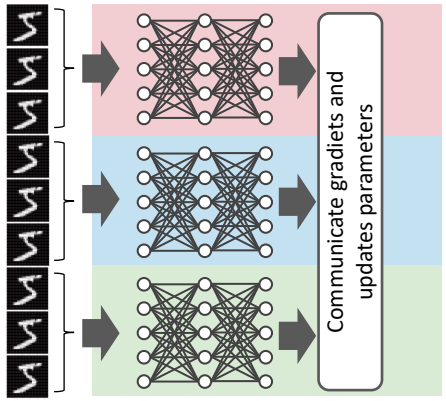
Exercise: Load the model

- Load the trained model by extending “03_build_NN.py”
  - Please recall 03_build_NN.py creates Net1 but does not train so it gave many wrong answers
  - By loading trained model before inference 03_build_NN.py will give more correct ansers
- Hint: Add the code snipet to 03_train_NN.py

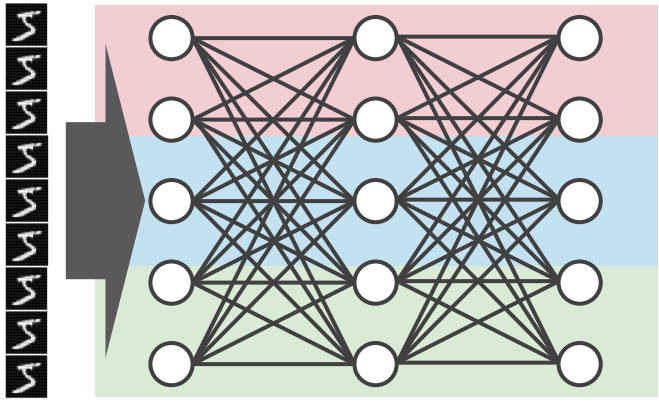


Distributed training

- Overview
  - Distrubuted training distribute training workloads across multiple PEs (e.g., cores/CPUs/GPUs) to accelerate process by training samples in parallel
- Data parallelism v.s. Model parallelism
  - Data parallelism
    - Data parallelism devides training data and distribute across multiple PEs
    - Each PEs trains a copy of the model on a different batch of training data and communicate its results (i.e., gradients) after computation to keep the model parameters sychronized across all PEs
  - Model parallelism
    - Model parallelism devides the model itself into parts
    - The parts are trained simultaineously across different PEs



Data parallelism

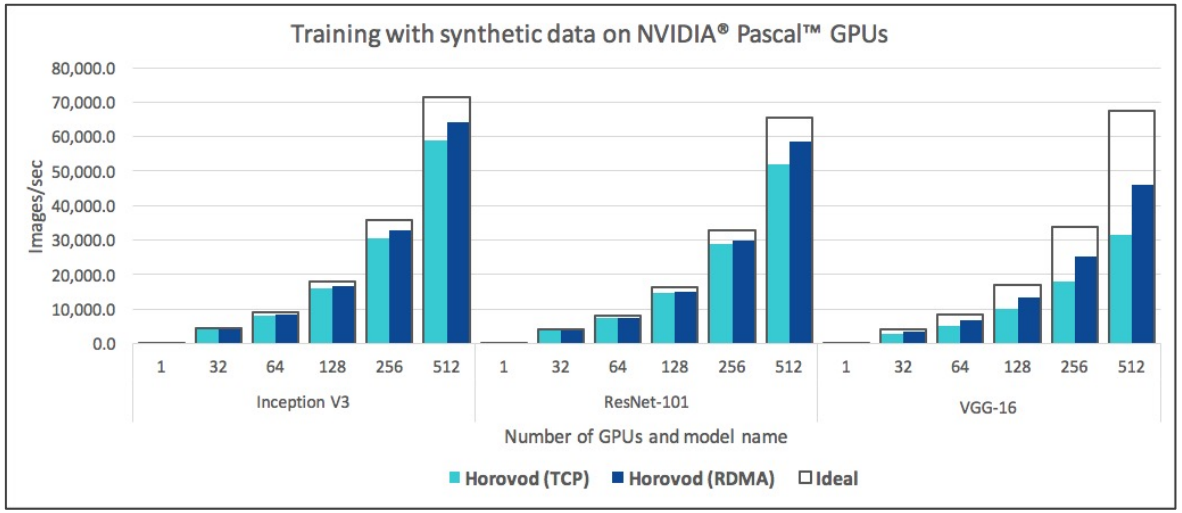


Model parallelism

→ Data parallelism is more common so we focus on data parallelism

Horovod

- High coverage across different DL frameworks
  - Horovod is an MPI-based distributed deep learning training framework for TensorFlow, Keras, PyTorch, and Apache MXNet
- Easy-to-use
  - The goal of Horovod is to make distributed deep learning fast and easy to use
  - Once a training script has been written for scale with Horovod, it can run on a single-CPU/GPU, multiple-CPUs/GPUs, or even multiple hosts without any further code changes
- Performance
  - Horovod achieves 90% scaling efficiency for both Inception V3 and ResNet-101, and 68% scaling efficiency for VGG-16



128 servers with 4 Pascal GPUs each connected by RoCE-capable 25 Gbit/s network:

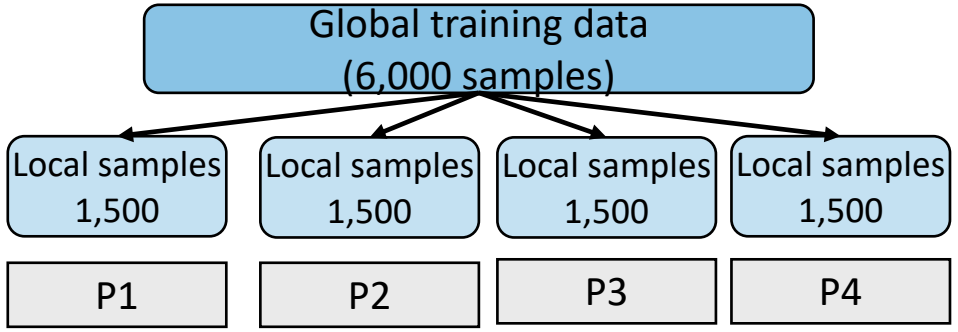
Modification to a single-PE script with 5 Steps

- 1. import Horovod module
- 2. Add Initialization
  - Call **hvd.init()**

Snippet (import horovod)

```
import horovod.torch as hvd
hvd.init()
```

- 3. Modify the Dataloader part
  - Partition dataset among workers by adding **DistributedSampler**
  - **Dataset → DistributedSampler → DataLoader**
  - shuffle is handled by DistributetdSampler so remove “shuffle=True” in DataLoader




```
train_dataloader = DataLoader(dataset=training_data, batch_size=batch_size, shuffle=True)
test_dataloader = DataLoader(dataset=test_data, batch_size=batch_size, shuffle=True)
```

Snippet (Distributed Sampler)

```
train_sampler = torch.utils.data.distributed.DistributedSampler(training_data, num_replicas=hvd.size(), rank=hvd.rank())
test_sampler  = torch.utils.data.distributed.DistributedSampler(test_data, num_replicas=hvd.size(), rank=hvd.rank())

train_dataloader = DataLoader(dataset=training_data, batch_size=batch_size, sampler=train_sampler)
test_dataloader = DataLoader(dataset=test_data, batch_size=batch_size, sampler=test_sampler)
```

Modification to a single-PE script with 5 Steps

- 4. Modify Optimizer part
  - Wrap the optimizer in **hvd.DistributedOptimizer**
  - **Optimizer → DistributedOptimizer**
- 5. Add Broadcast for synchronizing the initial variable states from rank 0 to all other ranks
  - Call **hvd.broadcast_parameter**

Snippet (DistributedOptitmizer and Broadcast)

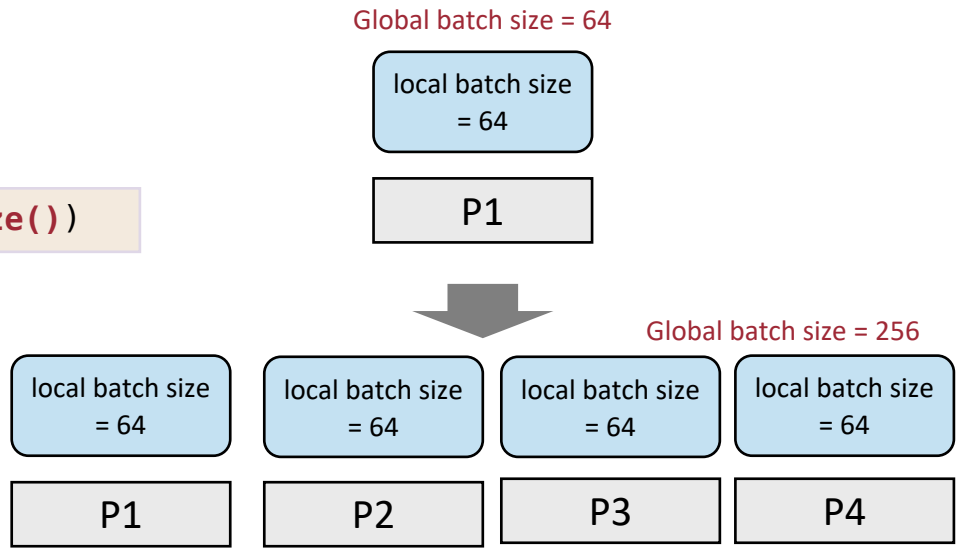
```
optimizer = hvd.DistributedOptimizer(optimizer, named_parameters=model.named_parameters())

hvd.broadcast_parameters(model.state_dict(), root_rank=0)
```

Other modifications (1/2)

- Learning rate and batch size [1]
  - In general, if you increase batch size too high, it degrease its accuracy
  - Roughly speaking, it is known that we can keep up accuracy by increasing both learning rate and batch size as long as the number of training samples is high enough
  - We increase learning rate depending on the number of processes

```
optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)
```

```
optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate * hvd.size())
```



[1] Samuel L. Smith, Pieter-Jan Kindermans, Chris Ying, Quoc V. Le. *Don't Decay the Learning Rate, Increase the Batch Size*. 2017. ICLR 2018
https://arxiv.org/abs/1711.00489

Other modifications (2/2)

- Get # of local training testing samples

```
def train_loop(dataloader, model, loss_fn):
    size = len(dataloader.dataset)
```

```
def train_loop(dataloader, model, loss_fn):
    size = len(dataloader.sampler)
```

```
def test_loop(dataloader, model, loss_fn):
    size = len(dataloader.dataset)
```

```
def test_loop(dataloader, model, loss_fn):
    size = len(dataloader.sampler)
```

- Printing by only rank 0 anf flush
  - mpirun buffers the stdout and dump it later
  - “flush=True” enables immediate print out on stdout

```
print("...")
```

```
if hvd.rank() == 0: print("...", flush=True)
```

Running distrubuted training and CPU binding

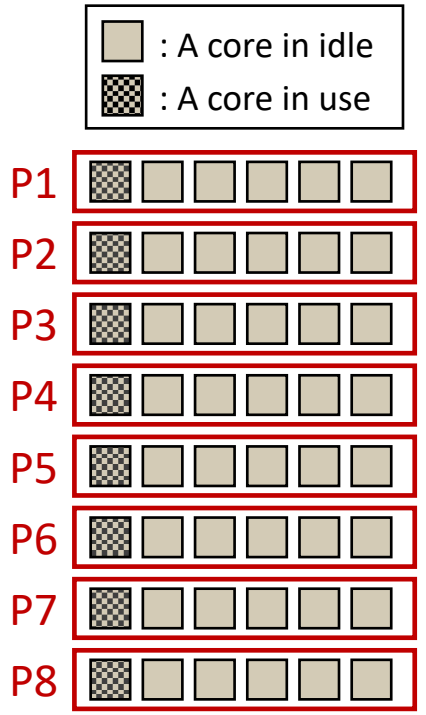
- Reserving an interective node

```
$ pjsub --interact -L "node=1" -L "rscgrp=int" -L "elapse=3:00:00" --sparam "wait-time=600" --mpi "proc=8"
-x "PJM_LLIO_GFSCACHE=/vol0004"
```

- Since Horovode is based on MPI, we use mpirun for launching processes

```
$ mpirun –n 8 ./07_dist_train_NN.py
```

- Process topology on A64FFX
  - Fugaku/A64FX has 48 computation cores
  - --mpi “procs=8” lets to use 8 processes at maxmum
    - Also, 8 processe are bined to each isolated set of 6 cores

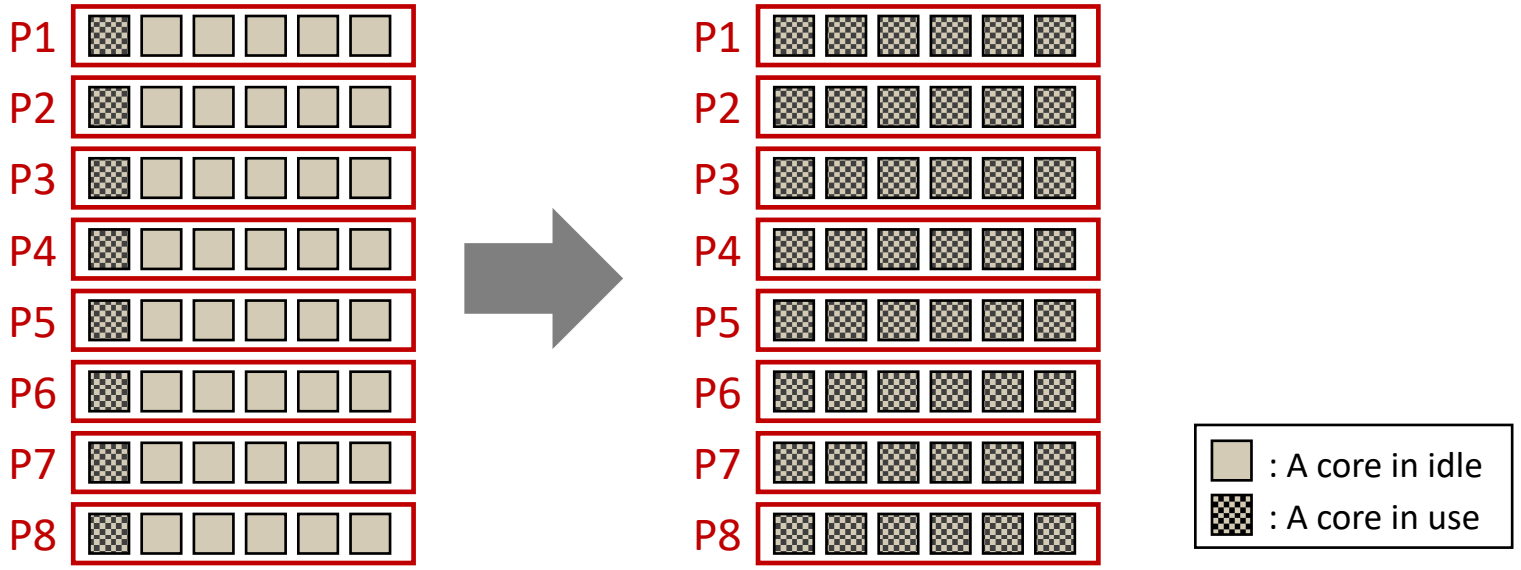


Tuning for Fugaku: MPI+OpenMP

- To make use of idle cores, we need to enable muti-threading
  - However, use of all the cores can rather degreate the performance due to contentionn with other threads (e.g., communication)

```
torch.set_num_threads(3)
```

- Another way is to use flat MPI with 48 processes, but in my experience MPI+OpenMP runs faster on Fugaku in PyTorch+Horovod



Exercise: Distributed training with data parallelism

- Write a code for distributed training with data parallelism by extending “05_save_NN_cleaned.py”
  - We have been accumulating unneccssary code used in eariler exercises, I removed these unnecessary code in “05_tran_NN_cleaned.py”
- Hint: Add and Modify the code based on the snippets

00_basics/05_save_NN_cleaned.py

+

Snippet (import horovod)

+

Modify based on Snippet (Distributed Sampler)

+

Snippet (DistributedOptitmizer and Broadcast)

+

1. Other modifications
2. Tuning for Fugaku: MPI+OpenMP

Answer: 00_basics/07_dist_train_NN.py

Hint: Template code

```
import ...
import horovod.torch as hvd

# Initialize Horovod
hvd.init()

# Define dataset...
training_dataset = ...

# Partition dataset among workers using DistributedSampler
train_sampler = torch.utils.data.distributed.DistributedSampler(training_data, num_replicas=hvd.size(), rank=hvd.rank())
test_sampler = ...
train_loader  = torch.utils.data.DataLoader(training_data, batch_size=..., sampler=train_sampler)
test_loader = ...

# Build model...
model = ...

# Add Horovod Distributed Optimizer
optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate * hvd.size())
optimizer = hvd.DistributedOptimizer(optimizer, named_parameters=model.named_parameters())

# Broadcast parameters from rank 0 to all other processes.
hvd.broadcast_parameters(model.state_dict(), root_rank=0)

# Start training
for t in range(epochs):
    ...
```

Exercise: Performance measurement

- Measure training times with 1, 2, 4, 8 proccesses

References

- PyTorch
  - https://pytorch.org/tutorials/index.html
- HOROVOD
  - https://horovod.readthedocs.io/en/stable/pytorch.html
- Machine Learning Testing: A Step to Perfection
  - https://serokell.io/blog/machine-learning-testing