MASTERING THE BASICS OF PYTORCH: AN INTRODUCTORY GUIDE TO DEEP LEARNING

Youva ADDAD

Normandie Univ, UNICAEN, ENSICAEN, CNRS, GREYC, Caen, FRANCE

youva.addad@unicaen.fr

GREYC Electronics and Computer Science Laboratory

Normandie Université

UNICAEN

ENSI CAEN ÉCOLE PUBLIQUE D'INGÉNIEURS CENTRE DE RECHERCHE

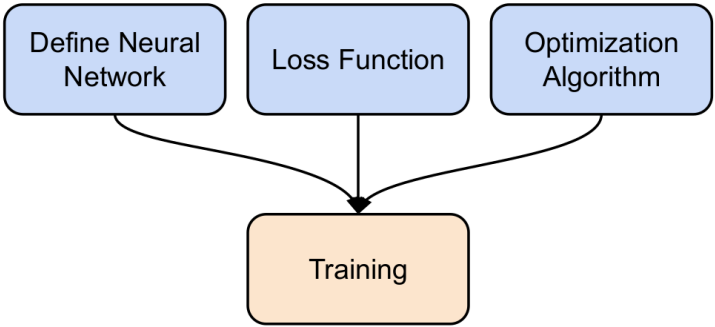
cnrs

RÉGION NORMANDIE

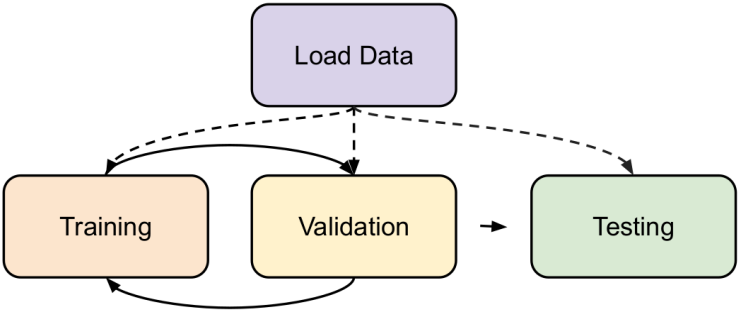
Outline

Deep Learning Workflow

Training Neural Networks

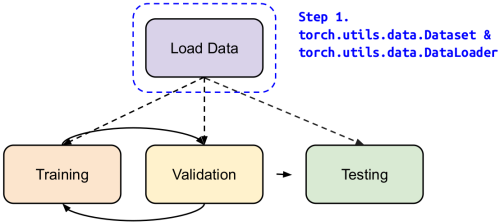


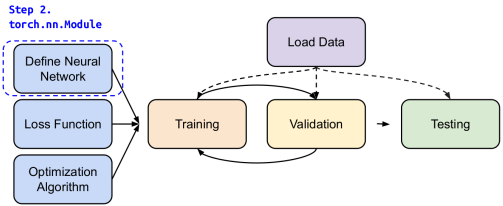
Training & Testing Neural Networks

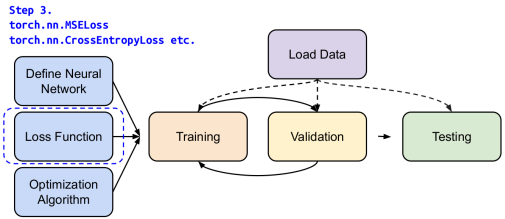


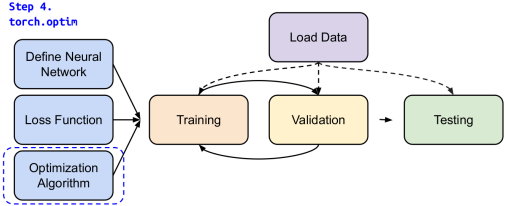
Training Step

Training Step

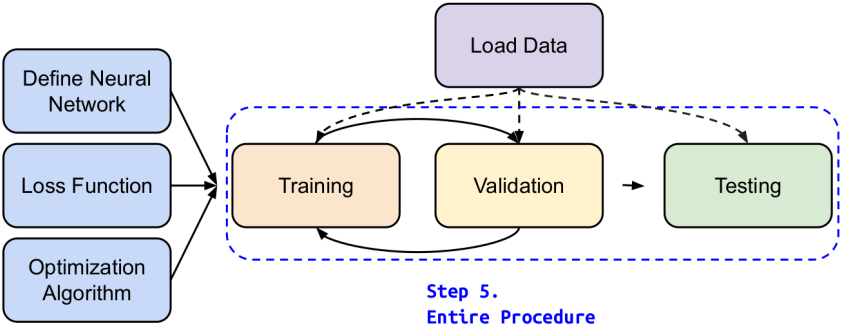








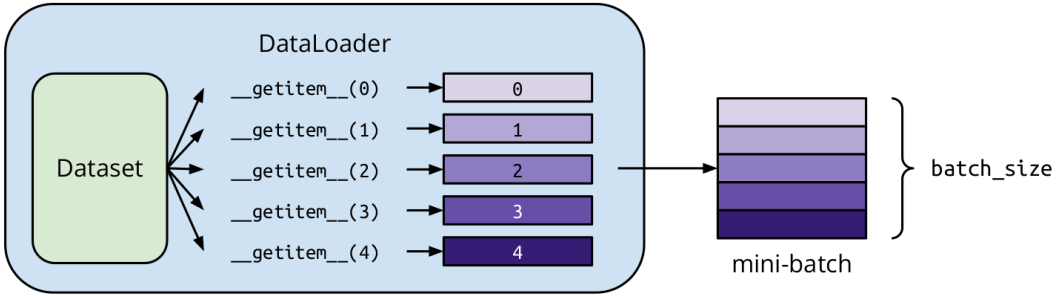
Training Step



Recape

torch.utils.data.Dataset & torch.utils.data.Dataloader

```
dataset = MyDataset(file)

dataloader = DataLoader(dataset, batch_size=5, shuffle=False)
```



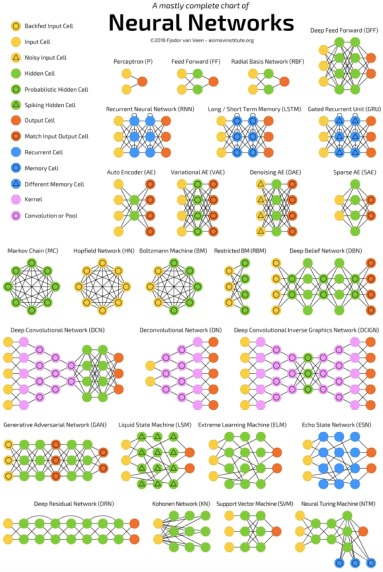
torch.utils.data.Dataset & torch.utils.data.Dataloader

```
from torch.utils.data import DataLoader

tr_loader = DataLoader(tr_data, batch_size=64, shuffle=True)
ts_loader = DataLoader(ts_data, batch_size=64, shuffle=True)
```

```
torch.utils.data.DataLoader(dataset, batch_size=1, shuffle=None
    , sampler=None, batch_sampler=None, num_workers=0,
    collate_fn=None, pin_memory=False, drop_last=False, timeout
    =0, worker_init_fn=None, multiprocessing_context=None,
    generator=None, *, prefetch_factor=None, persistent_workers=
    False, pin_memory_device='')
```

torch.nn

- torch.nn contains the basic components to define your neural networks, loss functions, regularization techniques and optimizers.



torch.nn.Loss

`nn.L1Loss`	Creates a criterion that measures the mean absolute error (MAE) between each element in the input x and target y.
`nn.MSELoss`	Creates a criterion that measures the mean squared error (squared L2 norm) between each element in the input x and target y.
`nn.CrossEntropyLoss`	This criterion computes the cross entropy loss between input logits and target.
`nn.CTCLoss`	The Connectionist Temporal Classification loss.
`nn.NLLLoss`	The negative log likelihood loss.
`nn.PoissonNLLLoss`	Negative log likelihood loss with Poisson distribution of target.
`nn.GaussianNLLLoss`	Gaussian negative log likelihood loss.
`nn.KLDivLoss`	The Kullback-Leibler divergence loss.

`nn.BCELoss`	Creates a criterion that measures the Binary Cross Entropy between the target and the input probabilities:
`nn.BCEWithLogitsLoss`	This loss combines a *Sigmoid* layer and the *BCELoss* in one single class.
`nn.MarginRankingLoss`	Creates a criterion that measures the loss given inputs $x1$, $x2$, two 1D mini-batch or 0D *Tensors*, and a label 1D mini-batch or 0D *Tensor* y (containing 1 or -1).
`nn.HingeEmbeddingLoss`	Measures the loss given an input tensor x and a labels tensor y (containing 1 or -1).
`nn.MultiLabelMarginLoss`	Creates a criterion that optimizes a multi-class multi-classification hinge loss (margin-based loss) between input x (a 2D mini-batch *Tensor*) and output y (which is a 2D *Tensor* of target class indices).
`nn.HuberLoss`	Creates a criterion that uses a squared term if the absolute element-wise error falls below delta and a delta-scaled L1 term otherwise.

torch.optim

```
opt = optim.SGD(model.parameters(), lr=0.01, momentum=0.9)
opt = optim.Adam([var1, var2], lr=0.0001)
```

```
optim.SGD([
                {'params': model.base.parameters(), 'lr': 1e
                    -2},
                {'params': model.classifier.parameters()}
            ], lr=1e-3, momentum=0.9)
```

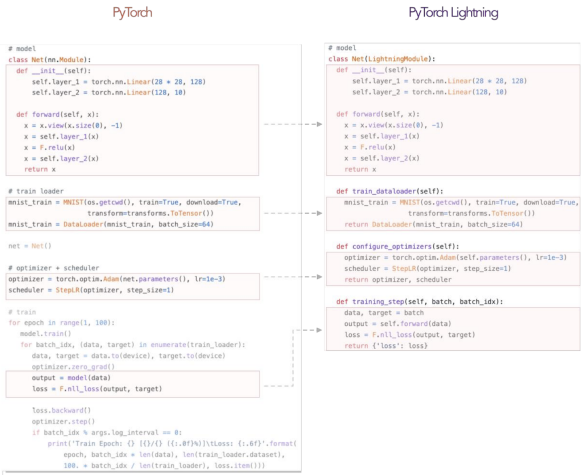
```
bias_params = [p for name, p in self.named_parameters() if '
    bias' in name]
others = [p for name, p in self.named_parameters() if 'bias'
    not in name]

optim.SGD([{'params': others},
           {'params': bias_params, 'weight_decay': 0}
            ], weight_decay=1e-2, lr=1e-2)
```

torch.optim

`Adadelta`	Implements Adadelta algorithm.
`Adafactor`	Implements Adafactor algorithm.
`Adagrad`	Implements Adagrad algorithm.
`Adam`	Implements Adam algorithm.
`AdamW`	Implements AdamW algorithm.
`SparseAdam`	SparseAdam implements a masked version of the Adam algorithm suitable for sparse gradients.
`Adamax`	Implements Adamax algorithm (a variant of Adam based on infinity norm).
`ASGD`	Implements Averaged Stochastic Gradient Descent.
`LBFGS`	Implements L-BFGS algorithm.
`NAdam`	Implements NAdam algorithm.
`RAdam`	Implements RAdam algorithm.
`RMSprop`	Implements RMSprop algorithm.
`Rprop`	Implements the resilient backpropagation algorithm.
`SGD`	Implements stochastic gradient descent (optionally with momentum).

Pytorch lightning

- High-level keras-like library that implements most of the boilerplatecode for you.
- Uses a callback system that allows the user to customize the training loop and easily add functionality.



Cuda Memory

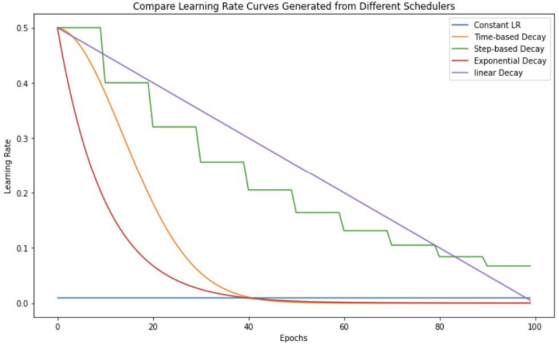
https://huggingface.co/blog/train_memory

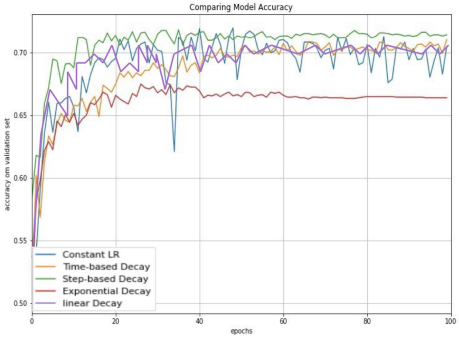
Outline

Learning Rate Scheduler

- Learning Rate scheduler.
- Cyclic scheduler.
- One Cycle scheduler.
- ScheduleFree.
- LR Finder.

Learning rate decay

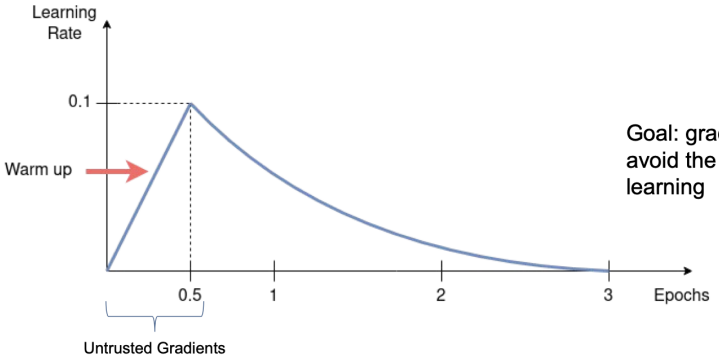




Learning rate decay

WARMUP for *large batches*

Problems: The first iterations have too much effect on the model (significant losses, high gradients, bias, etc.), a high learning rate can cause strong instability or divergence

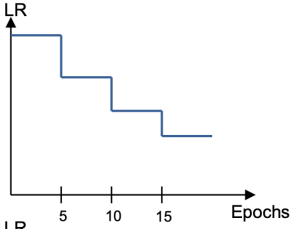


Goal: gradually increase the learning rate to avoid the risk of divergence at the start of learning

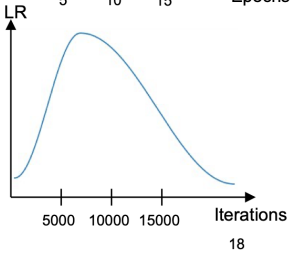
Learning rate decay

Each **scheduler** has *its own settings*

```
import torch.optim as opt

scheduler = opt.lr_scheduler.StepLR(optimizer, step_size=5, gamma=0.1)

for epoch in range(100):
        train(...)
        validate(...)
        scheduler.step()
```




```
import torch.optim as opt

scheduler = opt.lr_scheduler.CyclicLR(optimizer, base_lr=0.01, max_lr=0.1)


for epoch in range(10):
        for batch in data_loader:
                train_batch(...)
                scheduler.step()
```



18

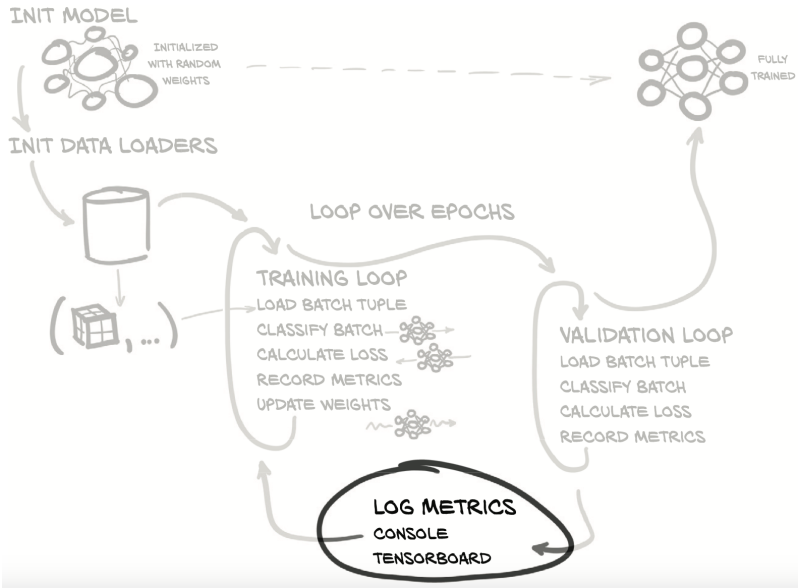
Learning rate decay

```
optimizer = optim.SGD(model.parameters(), lr=0.01, momentum
    =0.9)
scheduler = ExponentialLR(optimizer, gamma=0.9)

for epoch in range(20):
    for input, target in dataset:
        optimizer.zero_grad()
        output = model(input)
        loss = loss_fn(output, target)
        loss.backward()
        optimizer.step()
    scheduler.step()
```

Learning rate decay

```
optimizer = optim.SGD(model.parameters(), lr=0.01, momentum
    =0.9)
scheduler1 = ExponentialLR(optimizer, gamma=0.9)
scheduler2 = MultiStepLR(optimizer, milestones=[30,80], gamma
    =0.1)

for epoch in range(20):
    for input, target in dataset:
        optimizer.zero_grad()
        output = model(input)
        loss = loss_fn(output, target)
        loss.backward()
        optimizer.step()
    scheduler1.step()
    scheduler2.step()
```

Outline


INIT MODEL
INITIALIZED WITH RANDOM WEIGHTS
FULLY TRAINED
INIT DATA LOADERS
LOOP OVER EPOCHS
TRAINING LOOP
LOAD BATCH TUPLE
CLASSIFY BATCH
CALCULATE LOSS
RECORD METRICS
UPDATE WEIGHTS
VALIDATION LOOP
LOAD BATCH TUPLE
CLASSIFY BATCH
CALCULATE LOSS
RECORD METRICS
LOG METRICS
CONSOLE
TENSORBOARD

Why use a visualization tool ?

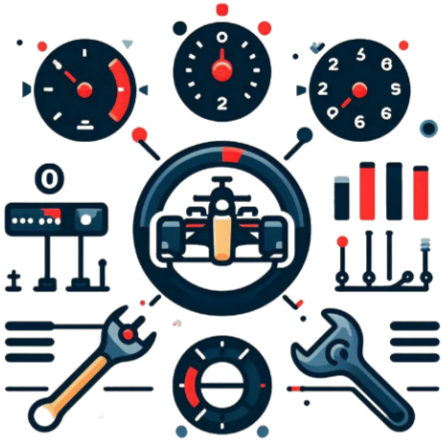
- Can’t fix what you can’t see.
- Training $\Rightarrow$ metrics.
- Experiment $\Rightarrow$ comparaison.

Can’t fix what you can’t see

Training : metrics

Specific to learning:

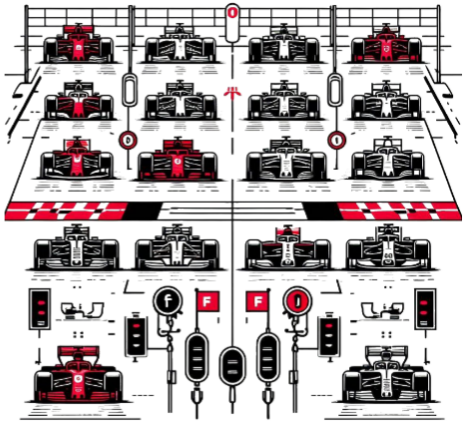
- train/val loss/acc.
- prototyping.
- profiling & debugging.



Experiment : comparaison

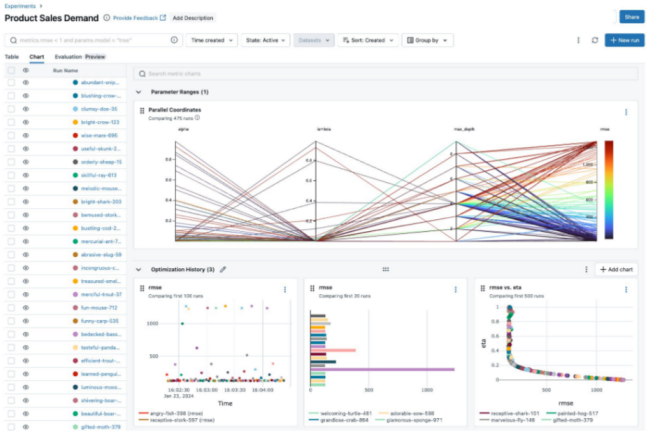
Not only training specific :

- hyperparameters.
- dataset & source code.
- hardware tracking.
- multi-user / collaboration.



Outline

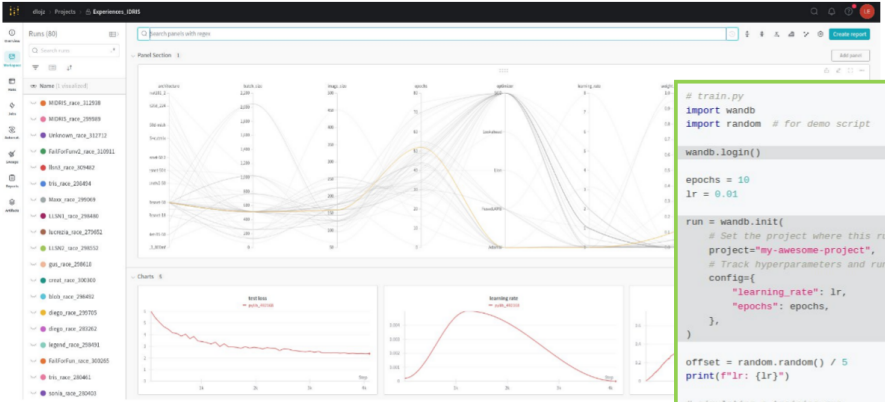
MLFlow




```python
import mlflow

# Start MLflow run
mlflow.start_run()

# Your code to log metrics, parameters, and artifacts
# For example:
mlflow.log_param("param1", 0.001)
mlflow.log_metric("metric1", 0.987)

# Log an artifact (file)
with open("example.txt", "w") as f:
    f.write("This is an example artifact.")
mlflow.log_artifact("example.txt")

# End MLflow run
mlflow.end_run()
```

Outline

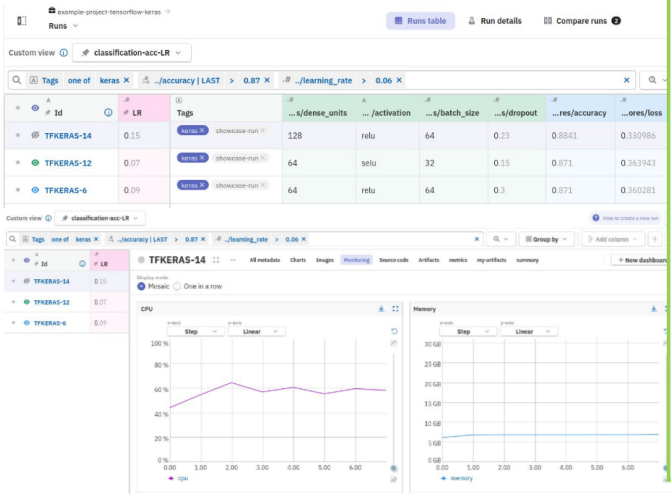
W&B




```python
# train.py
import wandb
import random  # for demo script

wandb.login()

epochs = 10
lr = 0.01

run = wandb.init(
    # Set the project where this run will be logged
    project="my-awesome-project",
    # Track hyperparameters and run metadata
    config={
        "learning_rate": lr,
        "epochs": epochs,
    },
)

offset = random.random() / 5
print(f"lr: {lr}")

# simulating a training run
for epoch in range(2, epochs):
    acc = 1 - 2**-epoch - random.random() / epoch - offset
    loss = 2**-epoch + random.random() / epoch + offset
    print(f"epoch={epoch}, accuracy={acc}, loss={loss}")
    wandb.log({"accuracy": acc, "loss": loss})

# run.log_code()
```

Outline

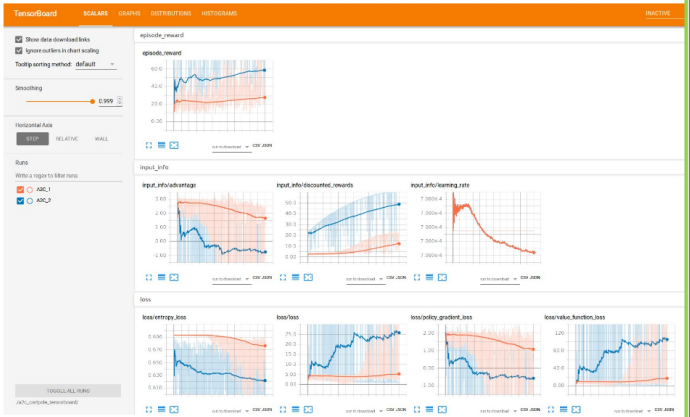
Neptune




```python
import neptune

# Create a Neptune run object
run = neptune.init_run(
    project="your-workspace-name/your-project-name",
    api_token="YourNeptuneApiToken",
    name="lotus-alligator",
    tags=["quickstart", "script"],  # optional
)

# Log a single value
# Specify a field name ("seed") inside the run and assign a value to it
run["seed"] = 0.42

# Log a series of values
from random import random

epochs = 10
offset = random() / 5

for epoch in range(epochs):
    acc = 1 - 2**-epoch - random() / (epoch + 1) - offset
    loss = 2**-epoch + random() / (epoch + 1) + offset

    run["accuracy"].append(acc)
    run["loss"].append(loss)

# Upload an image
run["single_image"].upload("Lenna_test_image.png")

# Download the MNIST dataset
import mnist

train_images = mnist.train_images()
train_labels = mnist.train_labels()

# Upload a series of images
from neptune.types import File

for i in range(10):
    run["image_series"].append(
        File.as_image(
            train_images[i] / 255
        ),  # You can upload arrays as images using the File.as_image() method
        name=f"{train_labels[i]}",
    )

# Stop the connection and synchronize the data with the Neptune servers
run.stop()
```

Outline

TensorBoard




```python
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.tensorboard import SummaryWriter

# Generate some dummy data
x = torch.randn(100, 1)
y = 2 * x + 1

# Define a simple linear model
class LinearModel(nn.Module):
    def __init__(self):
        super(LinearModel, self).__init__()
        self.linear = nn.Linear(1, 1)

    def forward(self, x):
        return self.linear(x)

model = LinearModel()

# Define the loss function
criterion = nn.MSELoss()

# Define the optimizer
optimizer = optim.SGD(model.parameters(), lr=0.01)

# Create a summary writer for TensorBoard
summary_writer = SummaryWriter('./logs')

# Training loop
for epoch in range(1000):
    optimizer.zero_grad()

    # Forward pass
    outputs = model(x)
    loss = criterion(outputs, y)

    # Backward pass and optimization
    loss.backward()
    optimizer.step()

    # Write summary to TensorBoard
    summary_writer.add_scalar('loss', loss.item(), epoch)

# Close the summary writer
```

TensorBoard setup:

```
from torch.utils.tensorboard import SummaryWriter

# default `log_dir` is "runs" - we'll be more specific here
writer = SummaryWriter('runs/fashion_mnist_experiment_1')
```

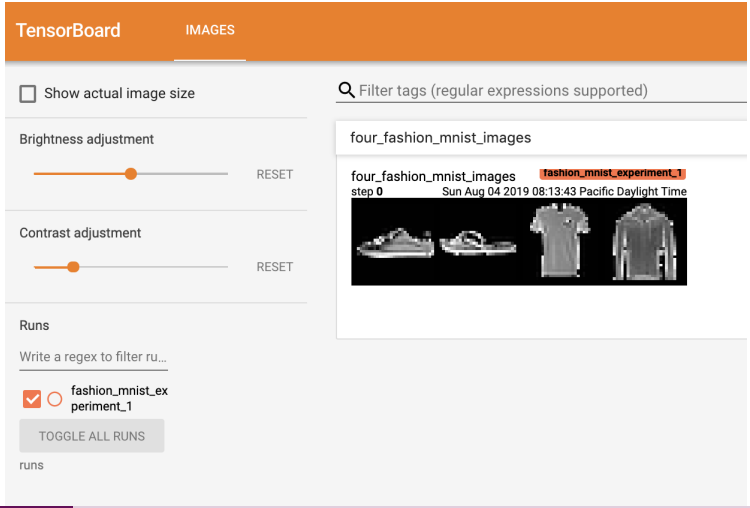
Writing to TensorBoard:

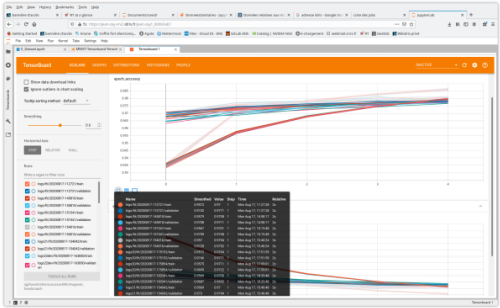
```
# get some random training images
dataiter = iter(trainloader)
images, labels = next(dataiter)

# create grid of images
img_grid = torchvision.utils.make_grid(images)

# show images
matplotlib_imshow(img_grid, one_channel=True)

# write to tensorboard
writer.add_image('four_fashion_mnist_images', img_grid)
```

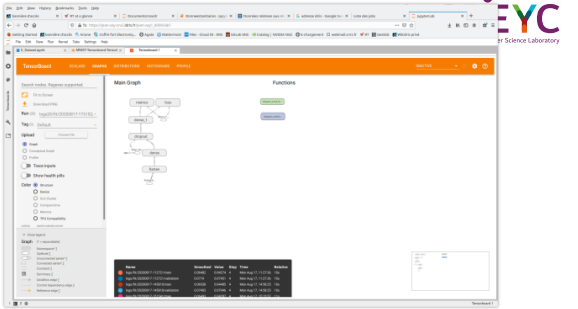
Running tensorboard:

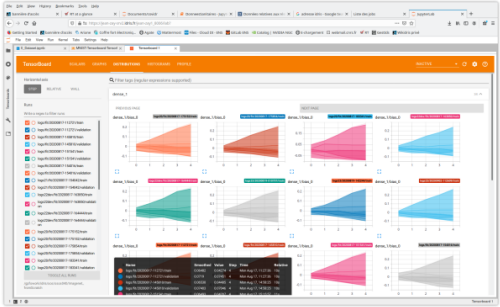
```
tensorboard --logdir=runs
```

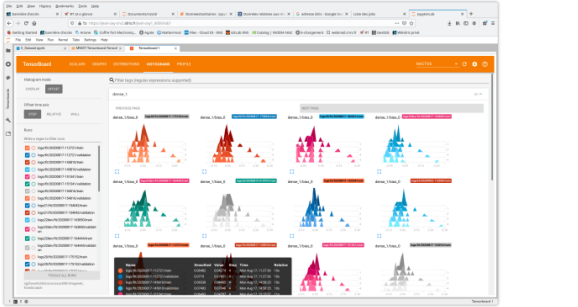
Should show the following:











```
from torch.utils.tensorboard import SummaryWriter

# default `log_dir` is "runs" - we'll be more specific here
writer = SummaryWriter('runs/fashion_mnist_experiment_1')
```

```
# TensorBoard callback for Scalars visualisation
# here we report running loss value at each iteration i
writer.add_scalar('training loss',
                  running_loss / 1000,
                  epoch * len(trainloader) + i)
```

```
# TensorBoard callback for ImageData visualisation
writer.add_image('four_fashion_mnist_images', img_grid)
```

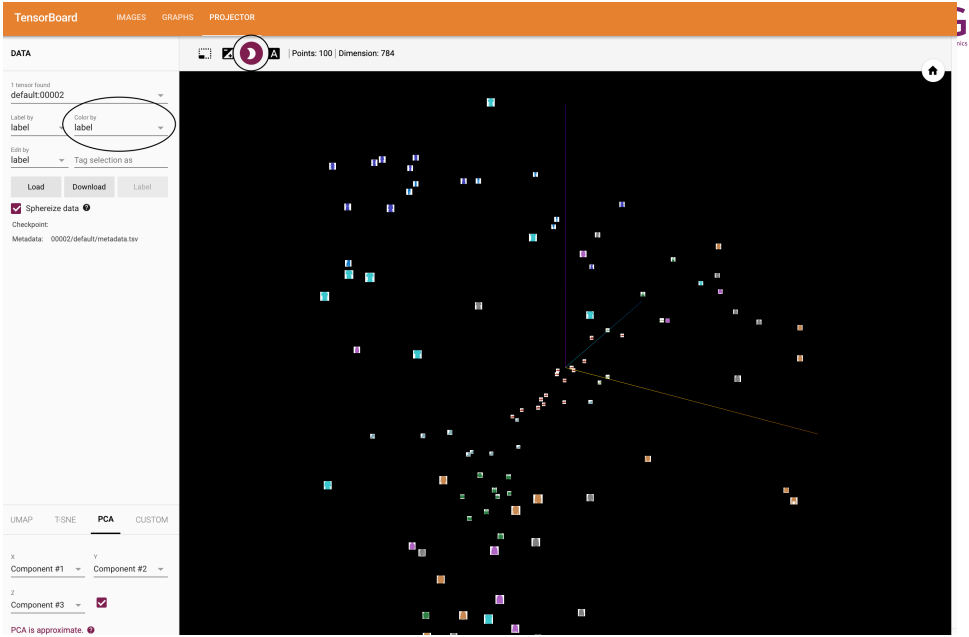
```
# TensorBoard callback for Graphs visualisation
writer.add_graph(net, images)
```

```
# TensorBoard callback for Embedding Projector visualisation
writer.add_embedding(features,
                     metadata=class_labels,
                     label_img=images.unsqueeze(1))
```

```
# TensorBoard callback for Weights/Bias histogramms
writer.add_histogram('distribution_weight', np.concatenate([j[1].
    detach().cpu().numpy().flatten()
        for j in model.named_parameters()
        if 'bn' not in j[0] and 'weight' in j[0]]), epoch + 1)
writer.add_histogram('distribution_bias', np.concatenate([j[1].detach
    ().cpu().numpy().flatten()
        for j in model.named_parameters()
        if 'bn' not in j[0] and 'bias' in j[0]]), epoch + 1)
```

Experiment : comparaison

```
# helper function
def select_n_random(data, labels, n=100):
    assert len(data) == len(labels)

    perm = torch.randperm(len(data))
    return data[perm][:n], labels[perm][:n]

# select random images and their target indices
images, labels = select_n_random(trainset.data, trainset.targets)

# get the class labels for each image
class_labels = [classes[lab] for lab in labels]

# log embeddings
features = images.view(-1, 28 * 28)
writer.add_embedding(features, metadata=class_labels,
                     label_img=images.unsqueeze(1))
writer.close()
```


TensorBoard
IMAGES
GRAPHS
PROJECTOR
DATA
Points: 100 | Dimension: 784
1 tensor found
default:00002
Label by
label
Color by
label
Edit by
label
Tag selection as
Load
Download
Label
Sphereize data
Checkpoint:
Metadata: 00002/default/metadata.tsv
UMAP
T-SNE
PCA
CUSTOM
X
Component #1
Y
Component #2
Z
Component #3
PCA is approximate.

https://pytorch.org/tutorials/intermediate/tensorboard_tutorial.html

Outline

What is distributed training?

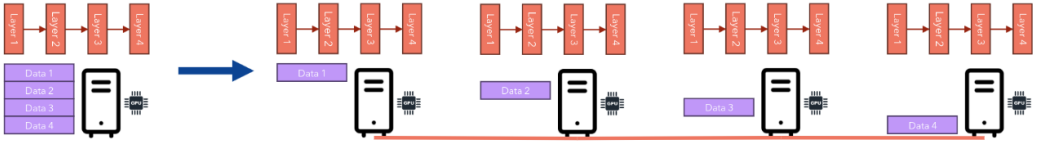
Imagine you want to train a Language Model on a very big dataset, for example the entire content of Wikipedia. The dataset is quite big, because it is made up of millions of articles, each of them with thousands of tokens. To train this model on a single GPU may be possible, but it poses some challenges:

- The model may not fit on a single GPU: this happens when the model has many parameters.
- You are forced to use a small batch size because a bigger batch size leads to an Out Of Memory error on CUDA.
- The model may take years to train because the dataset is huge.

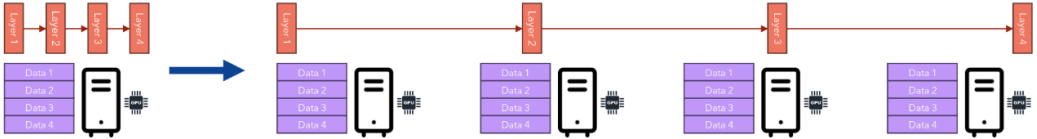
If any of the above applies to you, then you need to scale your training setup. Scaling can be done vertically, or horizontally. Let's compare these two options.

Data Parallelism vs Model Parallelism

If the model **can** fit within a single GPU, then we can distribute the training on multiple servers (each containing one or multiple GPUs), with each GPU processing a subset of the entire dataset in parallel and synchronizing the gradients during backpropagation. This option is known as **Data Parallelism**.

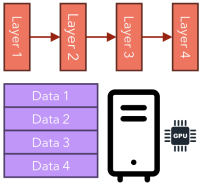


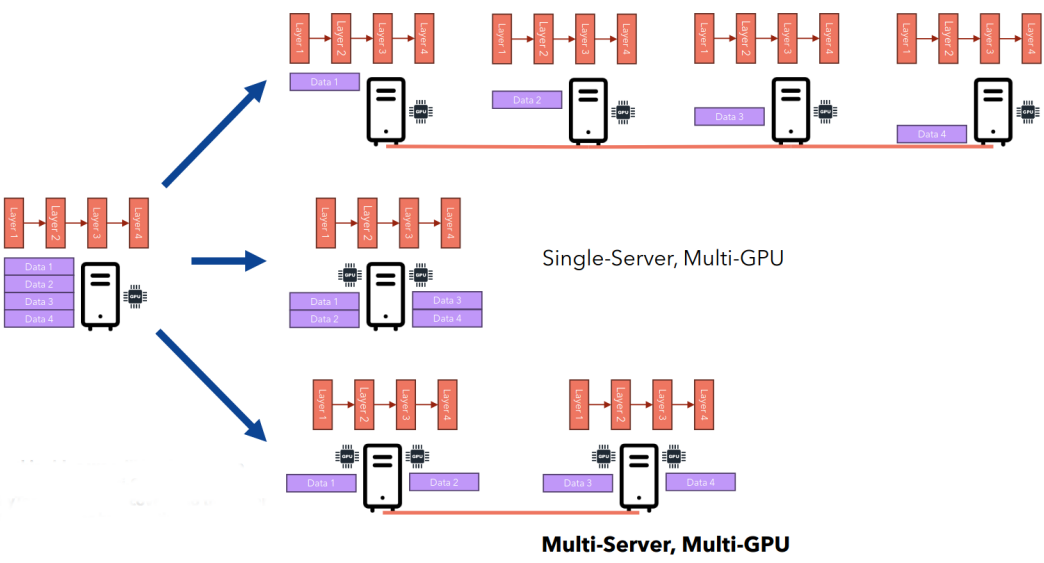
If the model **cannot** fit within a single GPU, then we need to "break" the model into smaller layers and let each GPU process a part of the forward/backward step during gradient descent. This option is known as **Model Parallelism**.



Distributed Data Parallel in detail

Imagine you have a training script that is running on a single computer/GPU, but it's very slow, because: the dataset is big and you can't use a big batch size as it will result in an Out Of Memory error on CUDA. Distributed Data Parallel is the solution in this case. It works in the following scenarios:



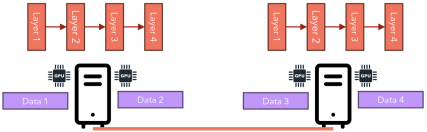

Multi-Server, Single-GPU
Layer 1
Layer 2
Layer 3
Layer 4
Data 1
Data 2
Data 3
Data 4
GPU
Single-Server, Multi-GPU
Multi-Server, Multi-GPU

Distributed Data Parallel in detail

If a cluster is made up of 2 computers, each having 2 GPUs, then we have 4 nodes in total.

Distributed Data Parallel works in the following way:

1. At the beginning of the training, the model's weights are initialized on one node and sent to all the other nodes (**Broadcast**).
2. Each node trains the same model (with the same initial weights) on a subset of the dataset.
3. Every few batches, the gradients of each node are accumulated on one node (summed up), and then sent back to all the other nodes (**All-Reduce**).
4. Each node updates the parameters of its local model with the gradients received using its own optimizer.
5. Go back to step 2.



```
import os
import sys
import tempfile
import torch
import torch.distributed as dist
import torch.nn as nn
import torch.optim as optim
import torch.multiprocessing as mp

from torch.nn.parallel import DistributedDataParallel as DDP
```

```python
def example(rank, world_size):
    # create default process group
    dist.init_process_group("gloo", rank=rank, world_size=world_size)
    # create local model
    model = nn.Linear(10, 10).to(rank)
    # construct DDP model
    ddp_model = DDP(model, device_ids=[rank])
    # define loss function and optimizer
    loss_fn = nn.MSELoss()
    optimizer = optim.SGD(ddp_model.parameters(), lr=0.001)

    # forward pass
    outputs = ddp_model(torch.randn(20, 10).to(rank))
    labels = torch.randn(20, 10).to(rank)
    # backward pass
    loss_fn(outputs, labels).backward()
    # update parameters
    optimizer.step()
```

```
def main():
    world_size = 2
    mp.spawn(example,
        args=(world_size,),
        nprocs=world_size,
        join=True)

if __name__=="__main__":
    # Environment variables which need to be
    # set when using c10d's default "env"
    # initialization mode.
    os.environ["MASTER_ADDR"] = "localhost"
    os.environ["MASTER_PORT"] = "29500"
    main()
```

Distributed Sampler

```python
# Initialize the process group
def init_process(rank, world_size, backend='nccl'):
    dist.init_process_group(
        backend=backend,
        init_method='env://',  # Uses environment variables for
            communication
        rank=rank,
        world_size=world_size
    )
```

```python
# Example model (simple fully connected network)
class SimpleModel(nn.Module):
    def __init__(self):
        super(SimpleModel, self).__init__()
        self.fc1 = nn.Linear(784, 256)
        self.fc2 = nn.Linear(256, 10)

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

```python
# Dummy dataset (for example purposes)
class SimpleDataset(Dataset):
    def __init__(self, size=1000):
        self.size = size

    def __len__(self):
        return self.size

    def __getitem__(self, idx):
        return torch.randn(784), torch.randint(0, 10, (1,))
```

```python
# Function to train the model
def train(rank, world_size):
    torch.cuda.set_device(rank) # Set the device to GPU
    init_process(rank, world_size) # Initialize the process group

    # Create the model and move it to the correct GPU
    model = SimpleModel().cuda(rank)

    # Wrap the model with DistributedDataParallel
    model = DDP(model, device_ids=[rank])
```

```
# Create a dummy dataset and DataLoader
dataset = SimpleDataset()
sampler = DistributedSampler(dataset, num_replicas=world_size, rank=
    rank, shuffle=True)
dataloader = DataLoader(dataset, batch_size=32, sampler=sampler)
```

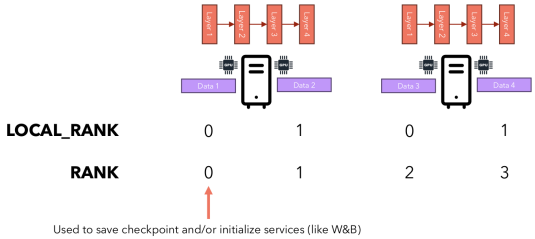
```
# Define the loss function and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)

# Training loop
model.train()
```

```
for epoch in range(5):  # Train for 5 epochs
    sampler.set_epoch(epoch)  # Important to shuffle data differently
        every epoch

    for data, target in dataloader:
        data, target = data.cuda(rank), target.cuda(rank)

        optimizer.zero_grad()
        output = model(data)
        loss = criterion(output, target)
        loss.backward()
        optimizer.step()

        if rank == 0:  # Only print from the main process
            print(f"Epoch [{epoch+1}/5], Loss: {loss.item():.4f}")

# Clean up distributed training
dist.destroy_process_group()
```

```python
import torch.multiprocessing as mp

def main():
    world_size = 4  # Total number of GPUs (or processes)
    mp.spawn(train, args=(world_size,), nprocs=world_size, join=True)

if __name__ == "__main__":
    main()
```

LOCAL_RANK vs RANK

The environment variable **LOCAL_RANK** indicates the ID of the GPU on the local computer, while the **RANK** variable indicates the a globally unique ID among all the nodes in the cluster.



Merci
Pour votre attention

Des Questions ?

youva.addad@unicaen.fr