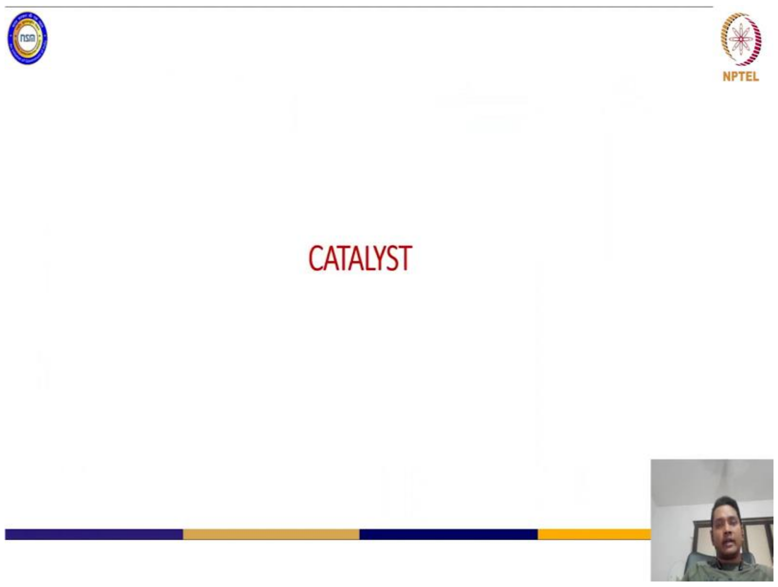
Applied Accelerated Artificial Intelligence

Dr. Satyajit Das
Department of Computer Science and Engineering
Indian Institute of Technology, Palakkad

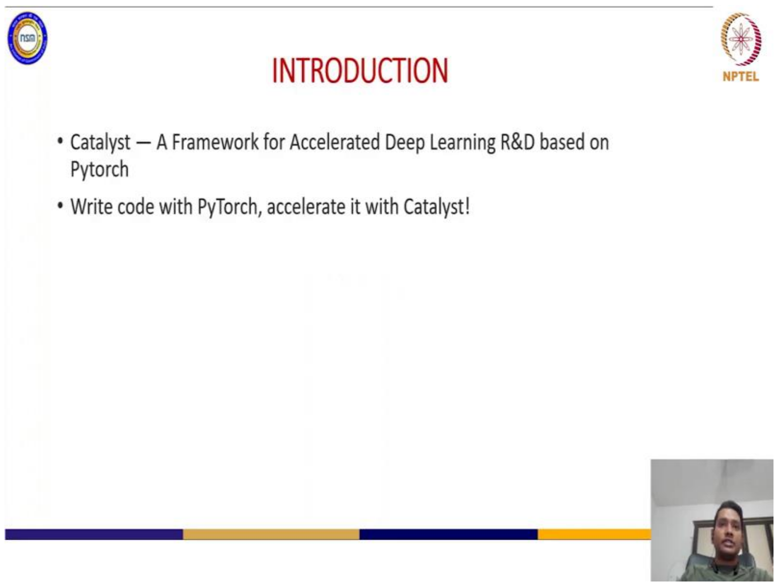
Introduction to Deep Learning
Lecture - 23
Profiling with DLProf PyTorch Catalyst Part - 2

(Refer Slide Time: 00:17)



CATALYST

(Refer Slide Time: 00:44)



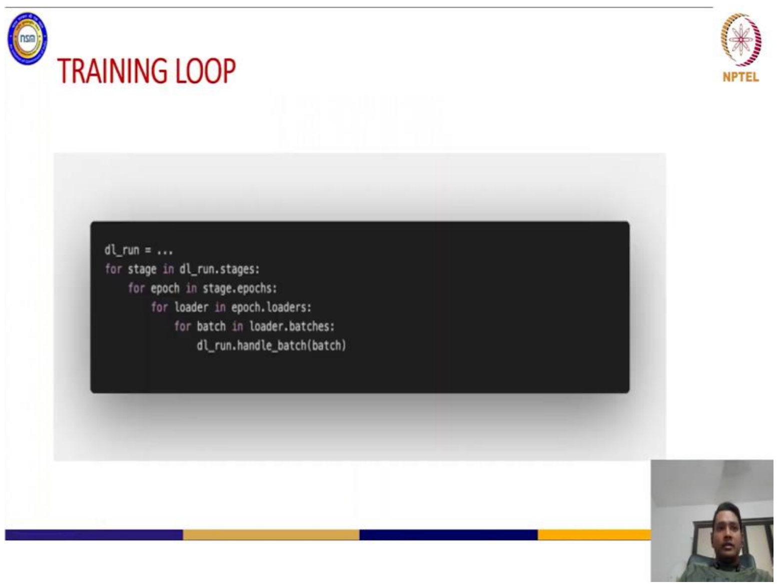
INTRODUCTION

- Catalyst — A Framework for Accelerated Deep Learning R&D based on Pytorch
- Write code with PyTorch, accelerate it with Catalyst!

Now, let us start with the Catalyst right. So, as I was mentioning that catalyst as name suggests, catalyst is basically the accelerated PyTorch framework right. And, this was the development was started in 2016, but it has been now it is very mature and lot of research and development pipeline using in catalyst. So, it is necessary nowadays to understand this framework.

So, this is a framework for accelerated deep learning and R&D based on PyTorch ok. And what it helps is that, it helps to write your code in very very compact way. So, that maintaining the huge pipeline of trainings will be seamless and we can reproduce the trainings, the metrics, the parameters again and again from several rounds.

(Refer Slide Time: 01:31)



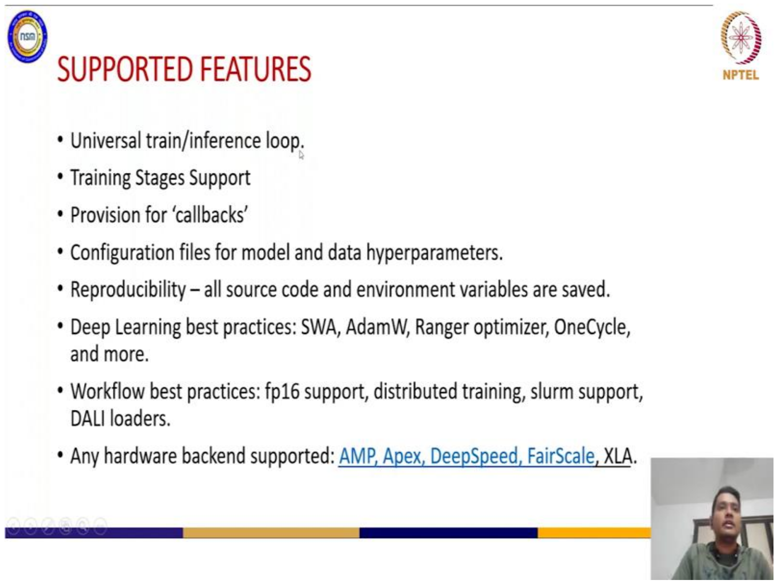
TRAINING LOOP

```
dl_run = ...
for stage in dl_run.stages:
    for epoch in stage.epochs:
        for loader in epoch.loaders:
            for batch in loader.batches:
                dl_run.handle_batch(batch)
```

So, now we will see the training group. So, if you are considering to explore your training module from different aspects. Let say I do not want only epochs to run right. I want several batches to run several loaders, maybe from different data sources I want to load several loaders, for PyTorch several stages I want to have multi stage pipeline in our training. So, now we are getting into some at most stage, but you understand that the training module is not just only one equation of epoch.

So, there can be several loads that can engage the training pipeline for different exploration. So, in the end of this exploration let us say I want to save the best three models out of them right. So, that is; so, all these pipelines management will be abstracted into catalysts in very simple custom problem. So, that is the flexibility of using this catalyst that we will see.

(Refer Slide Time: 02:43)



SUPPORTED FEATURES

- Universal train/inference loop.
- Training Stages Support
- Provision for 'callbacks'
- Configuration files for model and data hyperparameters.
- Reproducibility – all source code and environment variables are saved.
- Deep Learning best practices: SWA, AdamW, Ranger optimizer, OneCycle, and more.
- Workflow best practices: fp16 support, distributed training, slurm support, DALI loaders.
- Any hardware backend supported: AMP, Apex, DeepSpeed, FairScale, XLA.

So, supported features as I was mentioning you have universal train and inference loop. Universal train and inference loop that is from the perspective of the targets. So, whether you are doing supervised learning, whether you are doing reinforcement learning, whether you are doing maybe regression, whether you are doing some classification like classic classification which is also supervised; I mean subset of supervised learning.

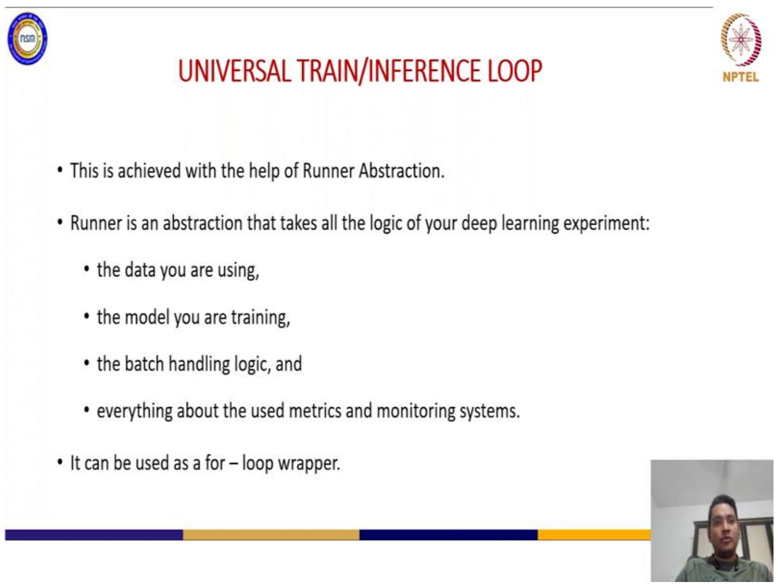
But whatever you do, it will have one universal structure that you can follow and also you will have the provisions for callbacks. So, once you wrap the entire thing with this universal customer which will we will define in the next slide, there will be provisions for call backs source. So, you can also induce some customized things inside the universal training or inference loop. Configuration files for models and that may help to have wide reproducibility.

So, if you maintain config file for your hyperparameters, you do not need to change anything inside your training module. You just change the config file for your training and you just go I mean you just have completely another configuration for your training without much of the changes. And, best practice for deep learning is using these kind of optimizers that you are seeing like AdamW or SWA optimizer.

So, different choices that if you want to have flexibility in the choices of optimizer or you want to have flexibility of choices inside the what are the matrix you want to see from the training, all these things you can actually abstract and define with the callbacks. Also in addition, you will have the support for mixed precision that I was mentioning that fp16 support, the distributed training support, slurm support and DALI loaders. So, DALI loader that you have seen that Nvidia DALI loaders is same that you use DALI loaders. But, that support also is available inside this catalyst framework ok.

And, this is also supported by several backend libraries like AMP, Apex. So, these two libraries like we talked about just now and DeepSpeed, FairScale or DeepSpeed from Microsoft, FairScale from Facebook, XLA so for your tensor cores. So, all these hardware back ends will be supported from this frameworks ok. So, this is this way it makes catalyst wide acceptable framework for accelerating your PyTorch training.

(Refer Slide Time: 05:50)



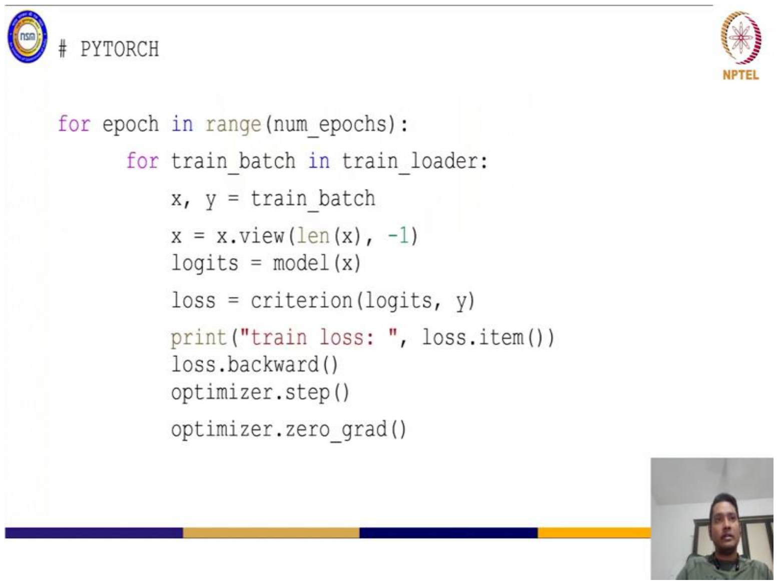
UNIVERSAL TRAIN/INFERENCE LOOP

- This is achieved with the help of Runner Abstraction.
- Runner is an abstraction that takes all the logic of your deep learning experiment:
  - the data you are using,
  - the model you are training,
  - the batch handling logic, and
  - everything about the used metrics and monitoring systems.
- It can be used as a for – loop wrapper.

Why we are talking about universal training loop? So, inference loop essentially will help you to construct inside your like custom runner, that I was talking about right. Now, runner is an abstraction that takes all the logic. So, whether you are how many metrics you want to handle, how many data, what kind of model you want to train, even some models if you want to you want to import from the libraries, you can do that seamlessly.

And, everything every metrics that that to be known let us say accuracy, loss, precision, recall F1 support. So, any metrics you can think of from universal training point of view, you can actually have complete abstraction into one wrapper called runner. And of course, there are a lot of more from wrapper, but we will see that eventually.

(Refer Slide Time: 06:53)

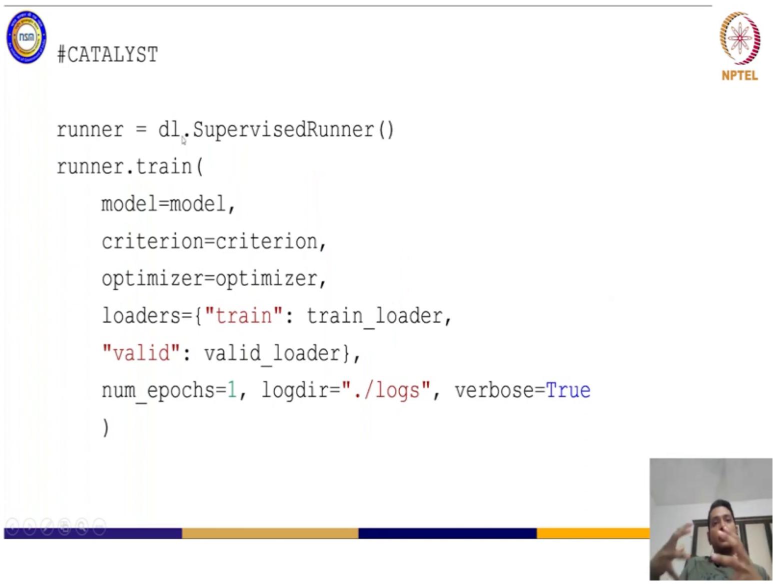


```
# PYTORCH

for epoch in range(num_epochs):
      for train_batch in train_loader:
          x, y = train_batch
          x = x.view(len(x), -1)
          logits = model(x)
          loss = criterion(logits, y)
          print("train loss: ", loss.item())
          loss.backward()
          optimizer.step()
          optimizer.zero_grad()
```

So, for PyTorch training, this is simple one batch training for several number of epochs you can see here. And, in the end we are printing the training loss. So, as I was mentioning that few statistics you need to print or maybe you need to keep output into your loggers or profilers to be profiled in the end. But, to print it every time whatever you need to track.

So, if you have let us say multistage pipeline for a big model and let us say several other models are also there to be taken into account in to several stages, you can see how big this pipeline can be right.

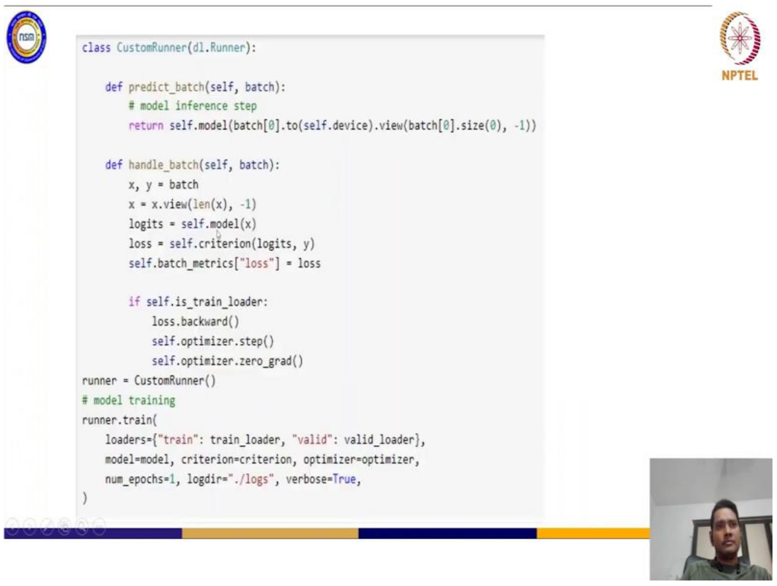
(Refer Slide Time: 07:40)



```
#CATALYST

runner = dl.SupervisedRunner()
runner.train(
    model=model,
    criterion=criterion,
    optimizer=optimizer,
    loaders={"train": train_loader,
    "valid": valid_loader},
    num_epochs=1, logdir="./logs", verbose=True
    )
```

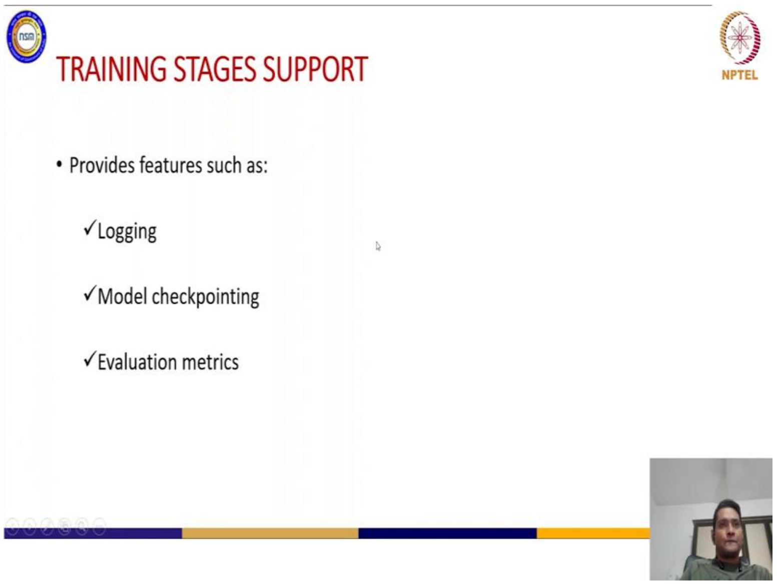
And, everything you can just customize into this custom runner which is either it can be custom runner or superset of SupervisedRunner and other runners also. So, we just; so, this is the universal training wrapper. So, basically you need define the model which model; criterion, optimizer, part of the train loader, valid loader ok; how many number of epochs, what will be the log directory. If you want verbose or not, you just define this. And, this will take care of indirect loops of or for loop of your frame.

(Refer Slide Time: 08:18)



```
class CustomRunner(dl.Runner):

    def predict_batch(self, batch):
        # model inference step
        return self.model(batch[0].to(self.device).view(batch[0].size(0), -1))

    def handle_batch(self, batch):
        x, y = batch
        x = x.view(len(x), -1)
        logits = self.model(x)
        loss = self.criterion(logits, y)
        self.batch_metrics["loss"] = loss

        if self.is_train_loader:
            loss.backward()
            self.optimizer.step()
            self.optimizer.zero_grad()
runner = CustomRunner()
# model training
runner.train(
    loaders={"train": train_loader, "valid": valid_loader},
    model=model, criterion=criterion, optimizer=optimizer,
    num_epochs=1, logdir="./logs", verbose=True,
)
```

(Refer Slide Time: 08:27)



TRAINING STAGES SUPPORT

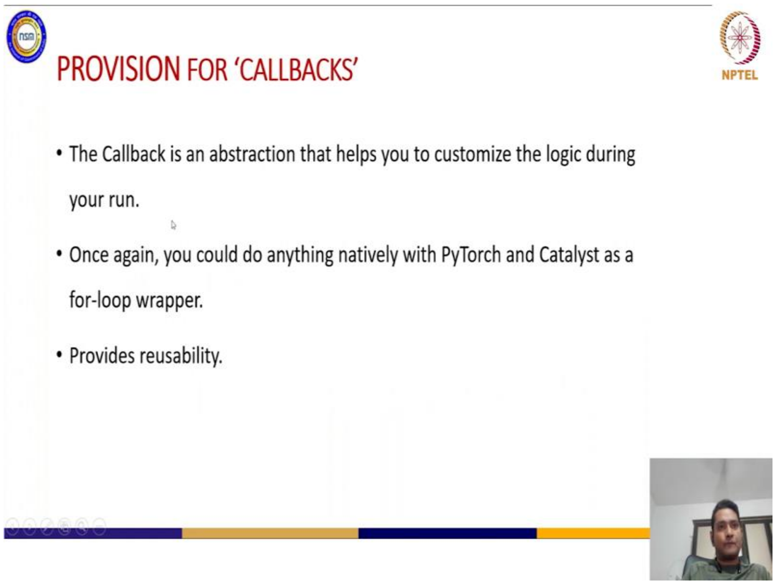
- Provides features such as:
  - ✓Logging
  - ✓Model checkpointing
  - ✓Evaluation metrics

So, this is just a simplified view of custom runner. But, we will come into this in the demo session ok. Also, in a training stage you can have logging model check pointing and evaluation metrics. Now, if you are a data scientist or you only not only just model training is necessary. Once you write the model for the training, you need to log your different metrics and for different metrics what kind of performance you are getting.

As I was showing from the TensorBoard right, last time what that size or how the hyper parameter set or combination of hyper parameters. So, those were very small number of parameters that we are tracking in the TensorBoard. Now, here you we can evaluate without writing any one piece mode of code. So, all these evaluation metrics that you can think of from any training pipeline point of view, it is already defined.

You just need to use that and you need to use that inside the callback and also this model check pointing will help you to evaluate this one. Because, you can have all these metrics stored in several checkpoints and evaluate based on those checkpoints. And, also you can select what will be the best model outcome and those model parameters you want to save it, we just say that inside this frame and it will do all the things automatically.

(Refer Slide Time: 10:13)



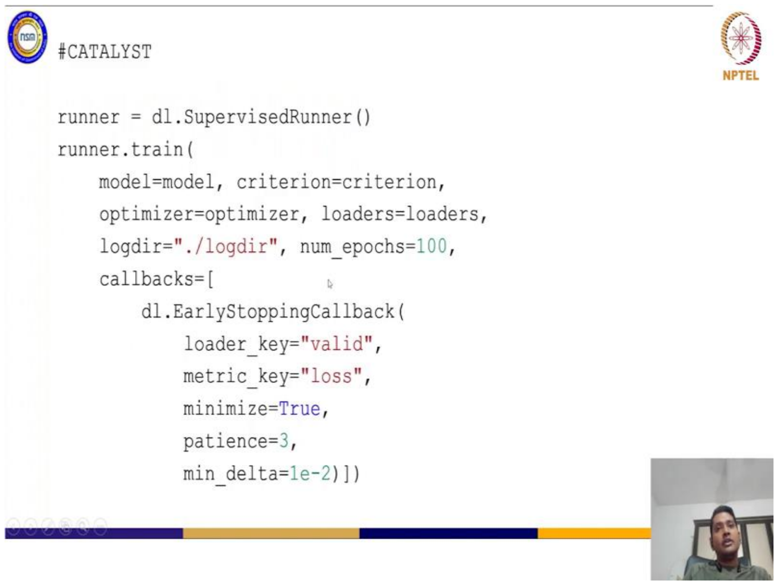
PROVISION FOR 'CALLBACKS'

- The Callback is an abstraction that helps you to customize the logic during your run.
- Once again, you could do anything natively with PyTorch and Catalyst as a for-loop wrapper.
- Provides reusability.

So, let us see one such. So, all this runner definition we will see in this demonstration. But, as I was mentioning the provision for callbacks will help you to customize your logic during. So, universal training will give you the flexibility to define the universal loops. Now, you want more right. You want to customize your metrics evaluation, you want to you want to log some of the performance metrics based on which you want to evaluate your model in the end.

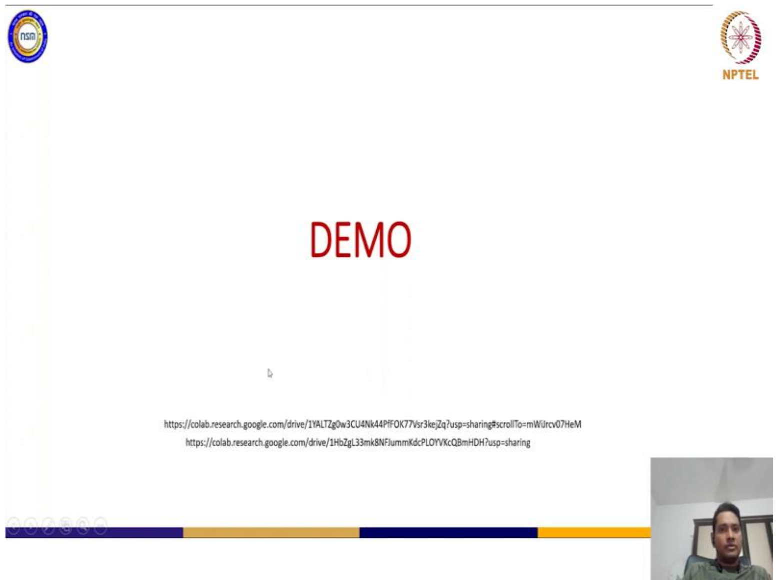
So, all this customization you want to do and that we can do using this callback and once again everything is PyTorch here. So, once you are familiarized with the PyTorch syntaxes, usage and maybe definitions. These are very easy to follow. So, the same things will be followed right and it provides usability, it will provide you to define your config file as well. So, all these things are there.

(Refer Slide Time: 11:18)



```
#CATALYST

runner = dl.SupervisedRunner()
runner.train(
    model=model, criterion=criterion,
    optimizer=optimizer, loaders=loaders,
    logdir="./logdir", num_epochs=100,
    callbacks=[
        dl.EarlyStoppingCallback(
            loader_key="valid",
            metric_key="loss",
            minimize=True,
            patience=3,
            min_delta=1e-2)])
```

(Refer Slide Time: 11:24)



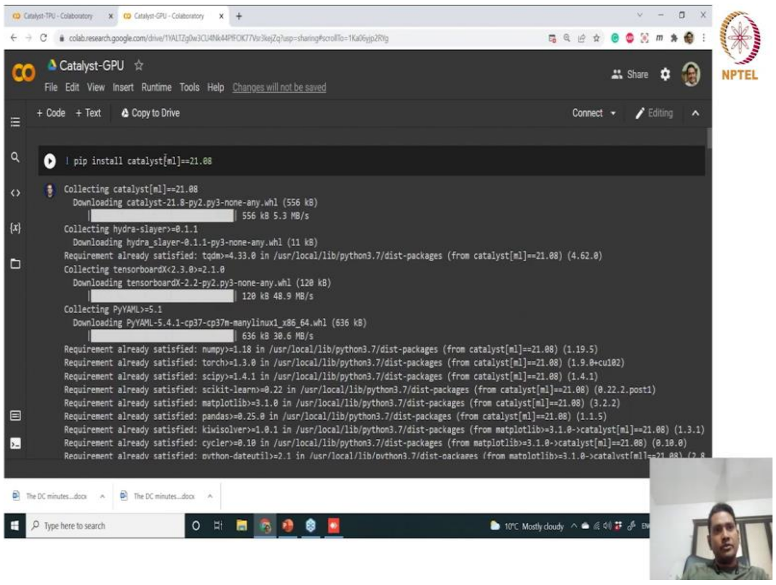
DEMO

https://colab.research.google.com/drive/1YALTZg0w3CU4Nk44PfFOK77Vsr3kejZq?usp=sharing#scrollTo=mWiJrcv07HeM

https://colab.research.google.com/drive/1HbZgL33mk8NFJummKdcPLOYVKcQBmHDH?usp=sharing

So, we will just go into the demonstration to see what kind of flexibility we are going to get.

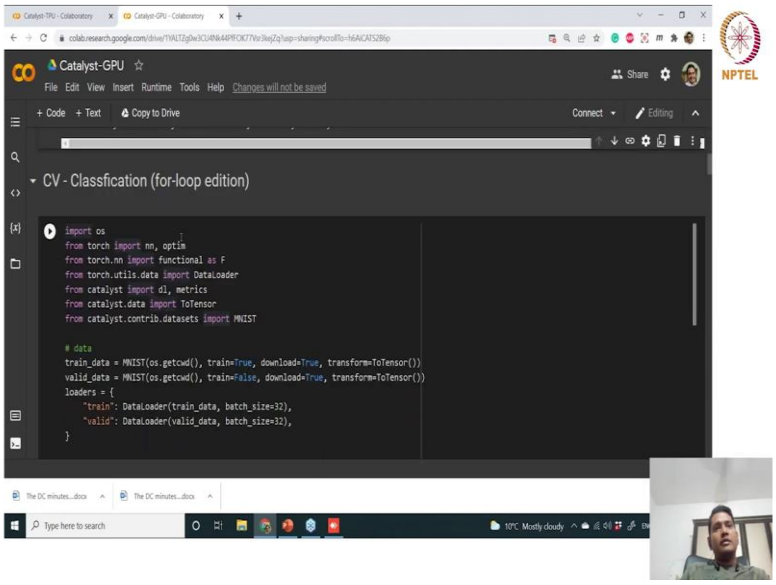
(Refer Slide Time: 11:33)


Catalyst-GPU
File Edit View Insert Runtime Tools Help Changes will not be saved
+ Code + Text Copy to Drive
Connect
Editing
Share
NPTEL

```
! pip install catalyst[ml]==21.08

Collecting catalyst[ml]==21.08
  Downloading catalyst-21.8-py2.py3-none-any.whl (556 kB)
     |████████████████████████████████| 556 kB 5.3 MB/s
Collecting hydra-slayer>=0.1.1
  Downloading hydra_slayer-0.1.1-py3-none-any.whl (11 kB)
Requirement already satisfied: tqdm>=4.33.0 in /usr/local/lib/python3.7/dist-packages (from catalyst[ml]==21.08) (4.62.0)
Collecting tensorboardX<2.3.0>=2.1.0
  Downloading tensorboardX-2.2-py2.py3-none-any.whl (120 kB)
     |████████████████████████████████| 120 kB 48.9 MB/s
Collecting PyYAML>=5.1
  Downloading PyYAML-5.4.1-cp37-cp37m-manylinux1_x86_64.whl (636 kB)
     |████████████████████████████████| 636 kB 30.6 MB/s
Requirement already satisfied: numpy>=1.18 in /usr/local/lib/python3.7/dist-packages (from catalyst[ml]==21.08) (1.19.5)
Requirement already satisfied: torch>=1.3.0 in /usr/local/lib/python3.7/dist-packages (from catalyst[ml]==21.08) (1.9.0+cu102)
Requirement already satisfied: scipy>=1.4.1 in /usr/local/lib/python3.7/dist-packages (from catalyst[ml]==21.08) (1.4.1)
Requirement already satisfied: scikit-learn>=0.22 in /usr/local/lib/python3.7/dist-packages (from catalyst[ml]==21.08) (0.22.2.post1)
Requirement already satisfied: matplotlib>=3.1.0 in /usr/local/lib/python3.7/dist-packages (from catalyst[ml]==21.08) (3.2.2)
Requirement already satisfied: pandas>=0.25.0 in /usr/local/lib/python3.7/dist-packages (from catalyst[ml]==21.08) (1.1.5)
Requirement already satisfied: kiwisolver>=1.0.1 in /usr/local/lib/python3.7/dist-packages (from matplotlib>=3.1.0->catalyst[ml]==21.08) (1.3.1)
Requirement already satisfied: cycler>=0.10 in /usr/local/lib/python3.7/dist-packages (from matplotlib>=3.1.0->catalyst[ml]==21.08) (0.10.0)
Requirement already satisfied: python-dateutil>=2.1 in /usr/local/lib/python3.7/dist-packages (from matplotlib>=3.1.0->catalyst[ml]==21.08) (2.8
```

So, this is the video we are provided by Sergey which who is the actually initiator for this project and now this is global community, completely open source. So, you can follow. So, all the things will be given to you and you can follow that you could as you are seen here. Now, you have to install a catalyst for ml with all the machine learning components from catalyst. So, it will be in downloading all the necessary packages to install it.

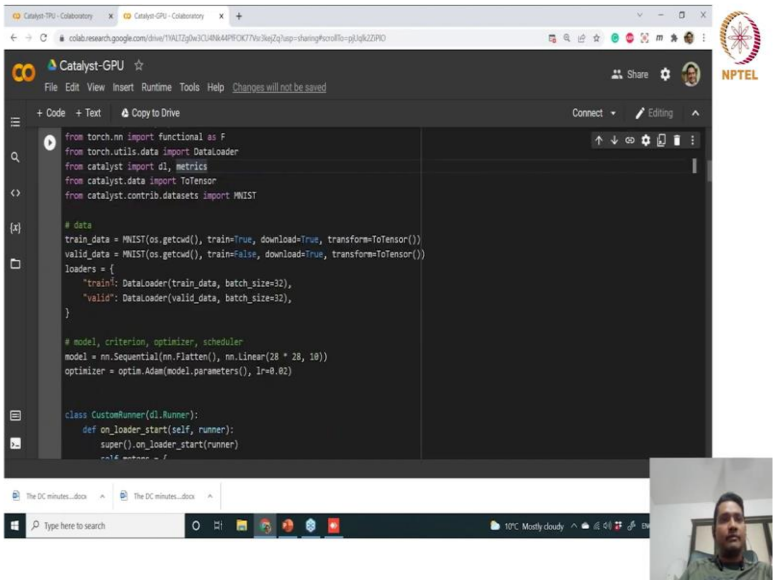
(Refer Slide Time: 12:14)


Catalyst-GPU
File Edit View Insert Runtime Tools Help Changes will not be saved
+ Code + Text Copy to Drive
Connect
Editing
Share
NPTEL

CV - Classfication (for-loop edition)

```python
import os
from torch import nn, optim
from torch.nn import functional as F
from torch.utils.data import DataLoader
from catalyst import dl, metrics
from catalyst.data import ToTensor
from catalyst.contrib.datasets import MNIST

# data
train_data = MNIST(os.getcwd(), train=True, download=True, transform=ToTensor())
valid_data = MNIST(os.getcwd(), train=False, download=True, transform=ToTensor())
loaders = {
    "train": DataLoader(train_data, batch_size=32),
    "valid": DataLoader(valid_data, batch_size=32),
}
```

(Refer Slide Time: 12:48)

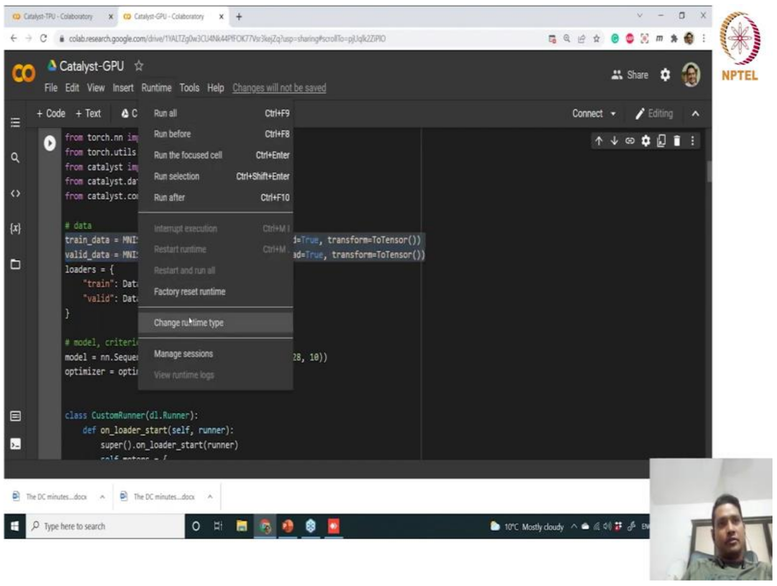

Catalyst-GPU
File Edit View Insert Runtime Tools Help Changes will not be saved
+ Code + Text Copy to Drive
Share
Connect
Editing
NPTEL

```
from torch.nn import functional as F
from torch.utils.data import DataLoader
from catalyst import dl, metrics
from catalyst.data import ToTensor
from catalyst.contrib.datasets import MNIST

# data
train_data = MNIST(os.getcwd(), train=True, download=True, transform=ToTensor())
valid_data = MNIST(os.getcwd(), train=False, download=True, transform=ToTensor())
loaders = {
    "train": DataLoader(train_data, batch_size=32),
    "valid": DataLoader(valid_data, batch_size=32),
}

# model, criterion, optimizer, scheduler
model = nn.Sequential(nn.Flatten(), nn.Linear(28 * 28, 10))
optimizer = optim.Adam(model.parameters(), lr=0.02)


class CustomRunner(dl.Runner):
    def on_loader_start(self, runner):
        super().on_loader_start(runner)
```

And, the let us see first classic computer vision classification ok. So, the same things that we have described so far, like one classification training model. So, basically we are having the same torch and libraries that we have imported so far and some catalyst libraries we need. And, also catalyst database has several database support for let us say all the popular data sets are also available inside this catalyst database, you can import them.

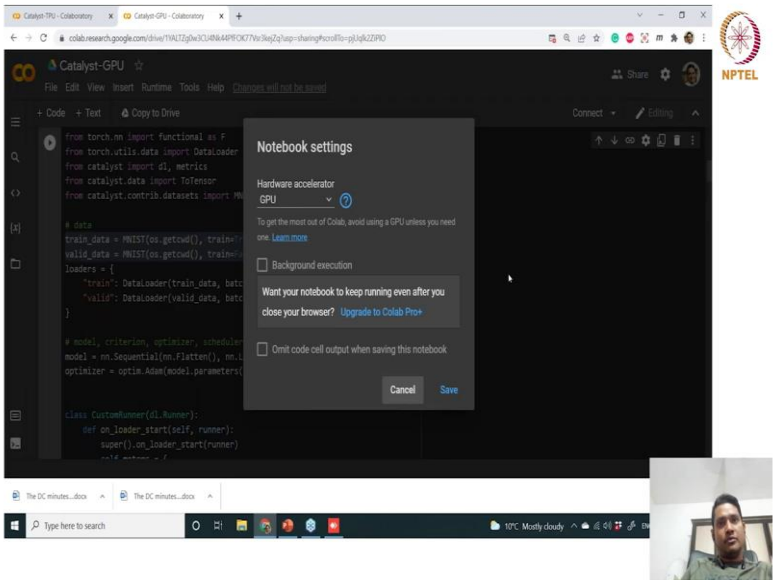
So, here we are not using torch vision for downloading the data, but we can use torch vision. So, since catalyst is being used, you can use catalyst. So, data loaded; so, basically with all the transformers you have we have defined.

(Refer Slide Time: 13:13)



```
from torch.nn im
from torch.utils
from catalyst im
from catalyst.da
from catalyst.co

# data
train_data = MNI                    d=True, transform=ToTensor())
valid_data = MNI                    ad=True, transform=ToTensor())
loaders = {
    "train": Dat
    "valid": Dat
}

# model, criteri
model = nn.Seque                    28, 10))
optimizer = opti


class CustomRunner(dl.Runner):
    def on_loader_start(self, runner):
        super().on_loader_start(runner)
```

(Refer Slide Time: 13:14)


Catalyst-GPU
File Edit View Insert Runtime Tools Help Changes will not be saved
+ Code + Text Copy to Drive
Cancel
Save

```
from torch.nn import functional as F
from torch.utils.data import DataLoader
from catalyst import dl, metrics
from catalyst.data import ToTensor
from catalyst.contrib.datasets import MN

# data
train_data = MNIST(os.getcwd(), train=Tr
valid_data = MNIST(os.getcwd(), train=Fa
loaders = {
    "train": DataLoader(train_data, batc
    "valid": DataLoader(valid_data, batc
}

# model, criterion, optimizer, schedule
model = nn.Sequential(nn.Flatten(), nn.L
optimizer = optim.Adam(model.parameters(


class CustomRunner(dl.Runner):
    def on_loader_start(self, runner):
        super().on_loader_start(runner)
```

Notebook settings

Hardware accelerator: GPU

To get the most out of Colab, avoid using a GPU unless you need one. Learn more

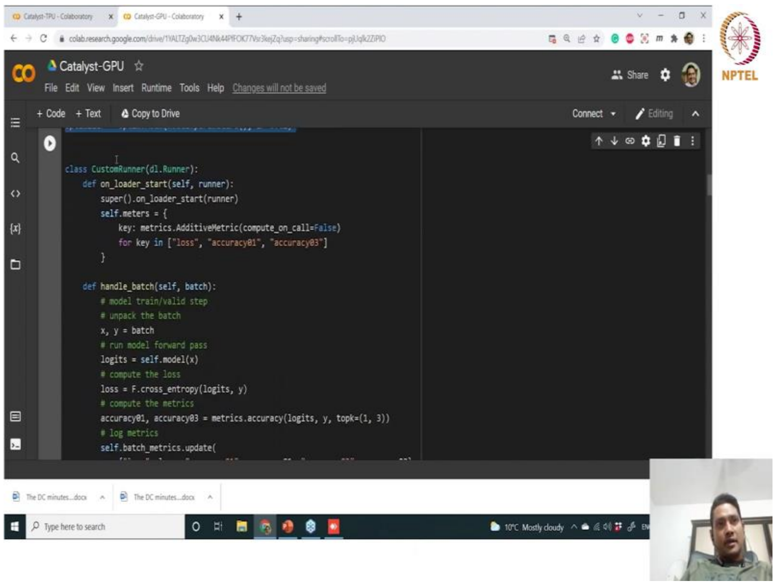
☐ Background execution

Want your notebook to keep running even after you close your browser? Upgrade to Colab Pro+

☐ Omit code cell output when saving this notebook

So, basically once you have to define the model and optimize for your device, just check that whether your run time is GPU or not.

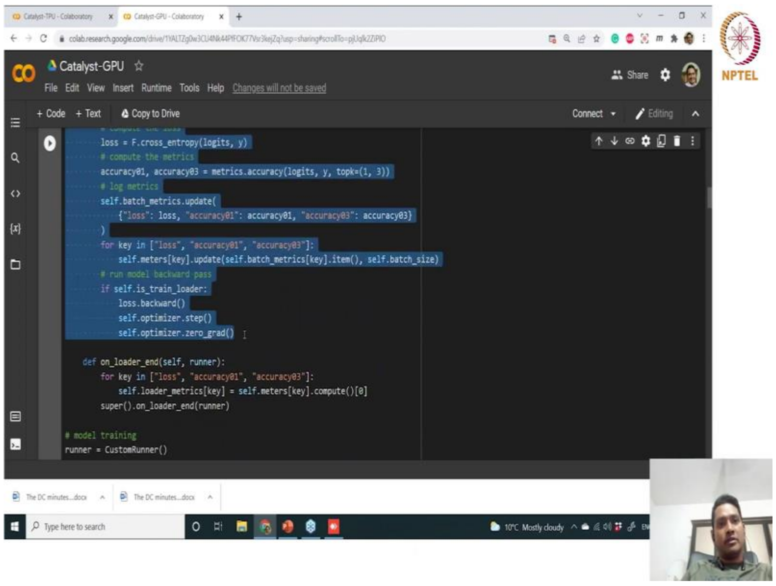
(Refer Slide Time: 13:28)



```
class CustomRunner(dl.Runner):
    def on_loader_start(self, runner):
        super().on_loader_start(runner)
        self.meters = {
            key: metrics.AdditiveMetric(compute_on_call=False)
            for key in ["loss", "accuracy01", "accuracy03"]
        }

    def handle_batch(self, batch):
        # model train/valid step
        # unpack the batch
        x, y = batch
        # run model forward pass
        logits = self.model(x)
        # compute the loss
        loss = F.cross_entropy(logits, y)
        # compute the metrics
        accuracy01, accuracy03 = metrics.accuracy(logits, y, topk=(1, 3))
        # log metrics
        self.batch_metrics.update(
```

So, defining the loaders. So, still now we are following the PyTorch, though this is simple PyTorch, model optimizer, simple PyTorch definition. Then, the CustomRunner definition.

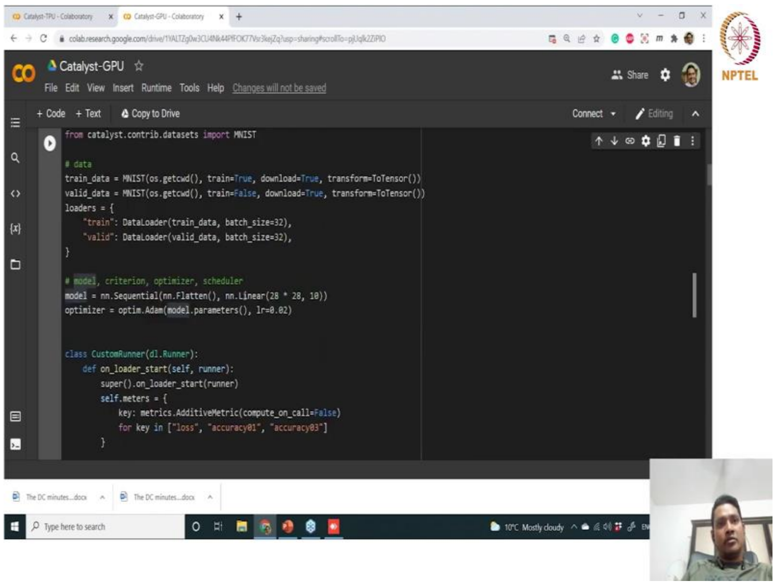
(Refer Slide Time: 13:39)



```
        loss = F.cross_entropy(logits, y)
        # compute the metrics
        accuracy01, accuracy03 = metrics.accuracy(logits, y, topk=(1, 3))
        # log metrics
        self.batch_metrics.update(
            {"loss": loss, "accuracy01": accuracy01, "accuracy03": accuracy03}
        )
        for key in ["loss", "accuracy01", "accuracy03"]:
            self.meters[key].update(self.batch_metrics[key].item(), self.batch_size)
        # run model backward pass
        if self.is_train_loader:
            loss.backward()
            self.optimizer.step()
            self.optimizer.zero_grad()

    def on_loader_end(self, runner):
        for key in ["loss", "accuracy01", "accuracy03"]:
            self.loader_metrics[key] = self.meters[key].compute()[0]
        super().on_loader_end(runner)

# model training
runner = CustomRunner()
```

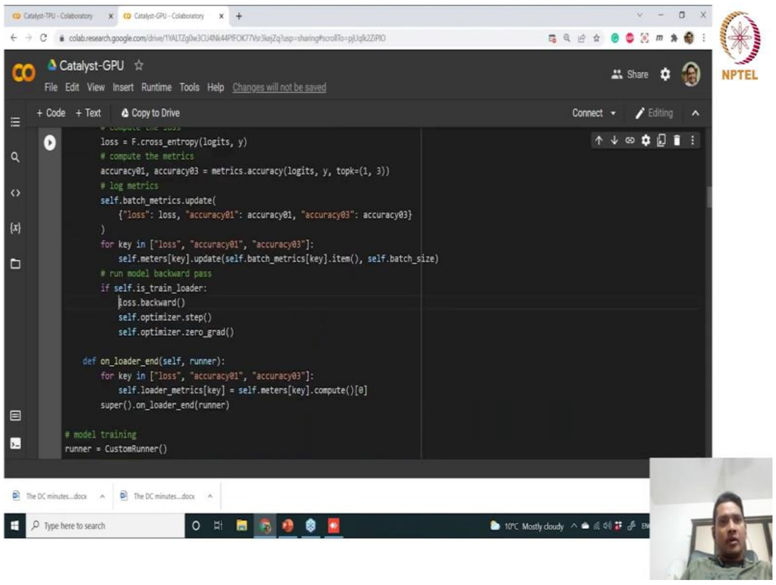
So, this is the CustomRunner class where the handle batch method is the most important batch method which will define your entire training pattern ok. So, handle batch has the unpacking of batch, running the model forward pass.

(Refer Slide Time: 13:55)



```
from catalyst.contrib.datasets import MNIST

# data
train_data = MNIST(os.getcwd(), train=True, download=True, transform=ToTensor())
valid_data = MNIST(os.getcwd(), train=False, download=True, transform=ToTensor())
loaders = {
    "train": DataLoader(train_data, batch_size=32),
    "valid": DataLoader(valid_data, batch_size=32),
}

# model, criterion, optimizer, scheduler
model = nn.Sequential(nn.Flatten(), nn.Linear(28 * 28, 10))
optimizer = optim.Adam(model.parameters(), lr=0.02)


class CustomRunner(dl.Runner):
    def on_loader_start(self, runner):
        super().on_loader_start(runner)
        self.meters = {
            key: metrics.AdditiveMetric(compute_on_call=False)
            for key in ["loss", "accuracy01", "accuracy03"]
        }
```

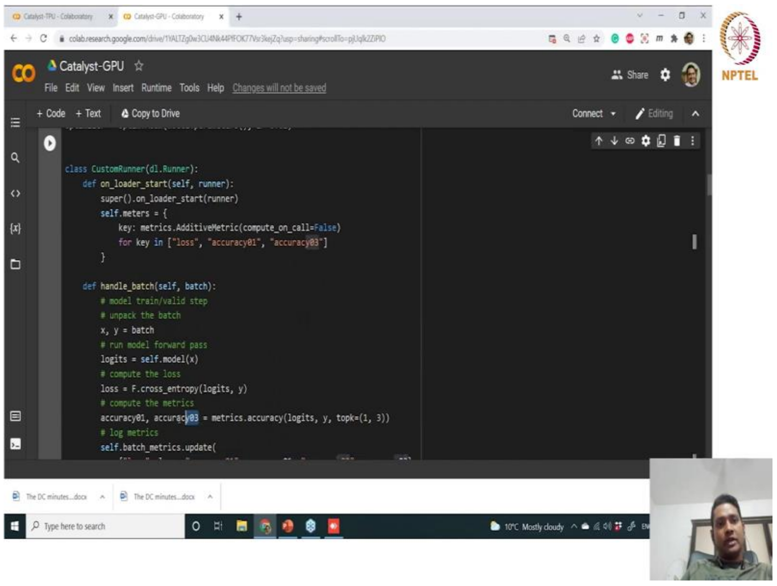
(Refer Slide Time: 14:00)



```
        loss = F.cross_entropy(logits, y)
        # compute the metrics
        accuracy01, accuracy03 = metrics.accuracy(logits, y, topk=(1, 3))
        # log metrics
        self.batch_metrics.update(
            {"loss": loss, "accuracy01": accuracy01, "accuracy03": accuracy03}
        )
        for key in ["loss", "accuracy01", "accuracy03"]:
            self.meters[key].update(self.batch_metrics[key].item(), self.batch_size)
        # run model backward pass
        if self.is_train_loader:
            loss.backward()
            self.optimizer.step()
            self.optimizer.zero_grad()

    def on_loader_end(self, runner):
        for key in ["loss", "accuracy01", "accuracy03"]:
            self.loader_metrics[key] = self.meters[key].compute()[0]
        super().on_loader_end(runner)

# model training
runner = CustomRunner()
```

So, this model is essentially one simple model that we have defined with the with one layer here right. Now, now we have then the train loader loaded and we are doing the backward pass. Then, updating the optimizer with the gradients and then zero grad initialization.

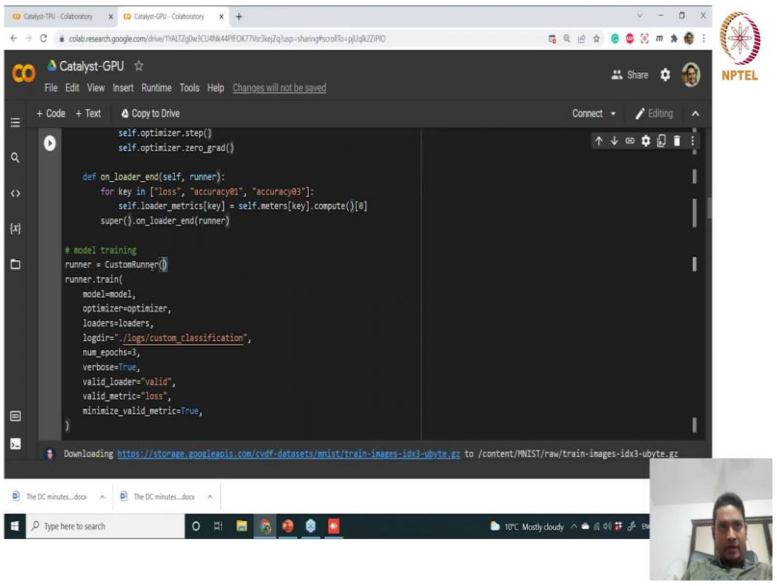
(Refer Slide Time: 14:31)



```
class CustomRunner(dl.Runner):
    def on_loader_start(self, runner):
        super().on_loader_start(runner)
        self.meters = {
            key: metrics.AdditiveMetric(compute_on_call=False)
            for key in ["loss", "accuracy01", "accuracy03"]
        }

    def handle_batch(self, batch):
        # model train/valid step
        # unpack the batch
        x, y = batch
        # run model forward pass
        logits = self.model(x)
        # compute the loss
        loss = F.cross_entropy(logits, y)
        # compute the metrics
        accuracy01, accuracy03 = metrics.accuracy(logits, y, topk=(1, 3))
        # log metrics
        self.batch_metrics.update(
```

So, this is the simple training pipeline. Now, we want to keep track of loss accuracy and so, which are defined by this batch metrics update. So, here we are seeing that only the metrics that we want to define ok and also these metrics would be actually computed and unpacked here ok.

(Refer Slide Time: 14:44)



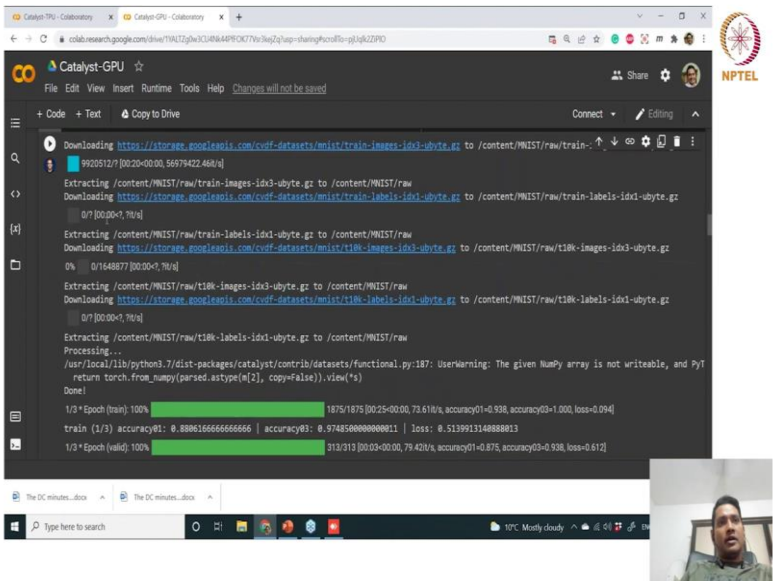
```
            self.optimizer.step()
            self.optimizer.zero_grad()

    def on_loader_end(self, runner):
        for key in ["loss", "accuracy01", "accuracy03"]:
            self.loader_metrics[key] = self.meters[key].compute()[0]
        super().on_loader_end(runner)

# model training
runner = CustomRunner()
runner.train(
    model=model,
    optimizer=optimizer,
    loaders=loaders,
    logdir="./logs/custom_classification",
    num_epochs=3,
    verbose=True,
    valid_loader="valid",
    valid_metric="loss",
    minimize_valid_metric=True,
)
```

```
Downloading https://storage.googleapis.com/cvdf-datasets/mnist/train-images-idx3-ubyte.gz to /content/MNIST/raw/train-images-idx3-ubyte.gz
```

So, this batch size for different batch sizes it will run and in the end we are creating the abstraction. So, this is the abstraction creation and take the runner. So, and this is the runner.train with all these models, optimizer, loader which will be the log directory for your metrics that you have defined so far, number of epochs train. So, it will take care of the loop automatically what goes through. So, we are seeing all these outputs and valid loader.

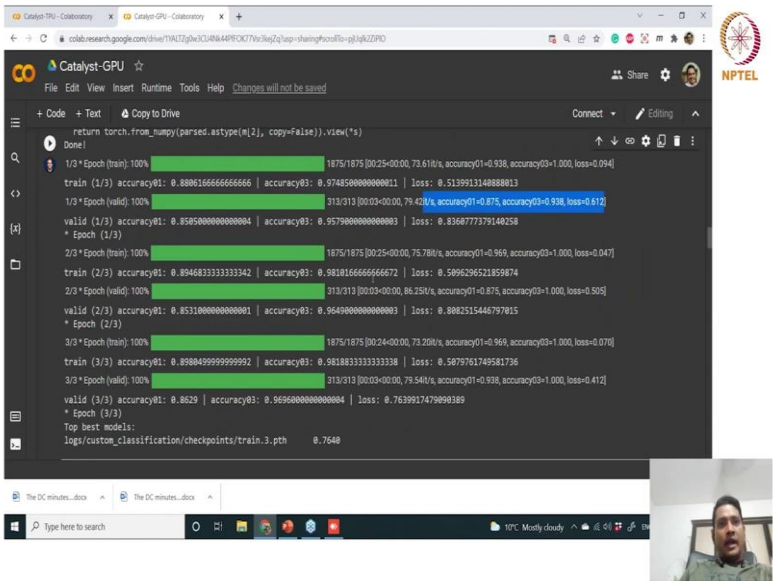
So, you want to validate. So, with this valid metrics loss and we want to minimize the valid metrics, we want to minimize the loss. So, this is the true. So, you see the inter pipelines for training and validation is actually defined in this simple training method here with this runner abstraction ok.

(Refer Slide Time: 15:44)



```
Downloading https://storage.googleapis.com/cvdf-datasets/mnist/train-images-idx3-ubyte.gz to /content/MNIST/raw/train-
9920512/? [00:20<00:00, 56979422.46it/s]
Extracting /content/MNIST/raw/train-images-idx3-ubyte.gz to /content/MNIST/raw
Downloading https://storage.googleapis.com/cvdf-datasets/mnist/train-labels-idx1-ubyte.gz to /content/MNIST/raw/train-labels-idx1-ubyte.gz
0/? [00:00<?, ?it/s]
Extracting /content/MNIST/raw/train-labels-idx1-ubyte.gz to /content/MNIST/raw
Downloading https://storage.googleapis.com/cvdf-datasets/mnist/t10k-images-idx3-ubyte.gz to /content/MNIST/raw/t10k-images-idx3-ubyte.gz
0%   0/1648877 [00:00<?, ?it/s]
Extracting /content/MNIST/raw/t10k-images-idx3-ubyte.gz to /content/MNIST/raw
Downloading https://storage.googleapis.com/cvdf-datasets/mnist/t10k-labels-idx1-ubyte.gz to /content/MNIST/raw/t10k-labels-idx1-ubyte.gz
0/? [00:00<?, ?it/s]
Extracting /content/MNIST/raw/t10k-labels-idx1-ubyte.gz to /content/MNIST/raw
Processing...
/usr/local/lib/python3.7/dist-packages/catalyst/contrib/datasets/functional.py:187: UserWarning: The given NumPy array is not writeable, and PyT
  return torch.from_numpy(parsed.astype(m[2], copy=False)).view(*s)
Done!
1/3 * Epoch (train): 100% 1875/1875 [00:25<00:00, 73.61it/s, accuracy01=0.938, accuracy03=1.000, loss=0.094]
train (1/3) accuracy01: 0.8806166666666666 | accuracy03: 0.9748500000000011 | loss: 0.5139913140888013
1/3 * Epoch (valid): 100% 313/313 [00:03<00:00, 79.42it/s, accuracy01=0.875, accuracy03=0.938, loss=0.612]
```

(Refer Slide Time: 15:47)

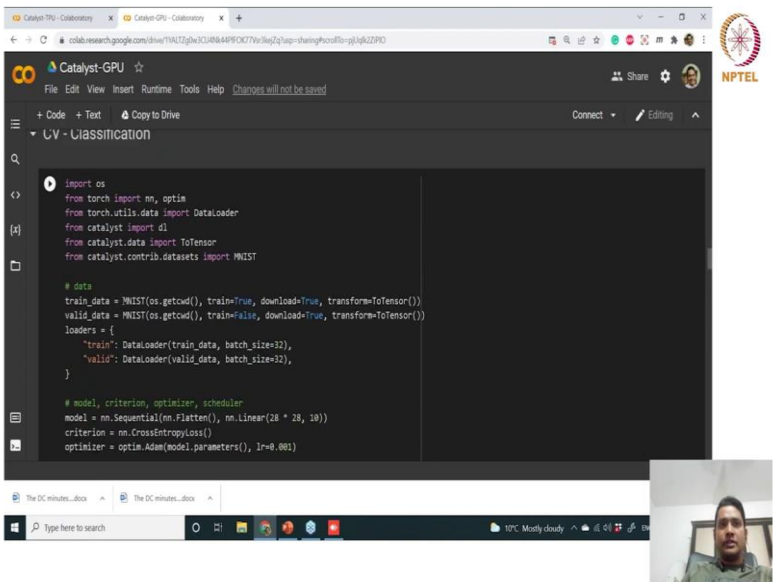


```
  return torch.from_numpy(parsed.astype(m[2], copy=False)).view(*s)
Done!
1/3 * Epoch (train): 100% 1875/1875 [00:25<00:00, 73.61it/s, accuracy01=0.938, accuracy03=1.000, loss=0.094]
train (1/3) accuracy01: 0.8806166666666666 | accuracy03: 0.9748500000000011 | loss: 0.5139913140888013
1/3 * Epoch (valid): 100% 313/313 [00:03<00:00, 79.42it/s, accuracy01=0.875, accuracy03=0.938, loss=0.612]
valid (1/3) accuracy01: 0.8505000000000004 | accuracy03: 0.9579000000000003 | loss: 0.8360777379140258
* Epoch (1/3)
2/3 * Epoch (train): 100% 1875/1875 [00:25<00:00, 75.78it/s, accuracy01=0.969, accuracy03=1.000, loss=0.047]
train (2/3) accuracy01: 0.8946833333333342 | accuracy03: 0.9810166666666672 | loss: 0.5096296521859874
2/3 * Epoch (valid): 100% 313/313 [00:03<00:00, 86.25it/s, accuracy01=0.875, accuracy03=1.000, loss=0.505]
valid (2/3) accuracy01: 0.8531000000000001 | accuracy03: 0.9649000000000003 | loss: 0.8082515446797015
* Epoch (2/3)
3/3 * Epoch (train): 100% 1875/1875 [00:24<00:00, 73.20it/s, accuracy01=0.969, accuracy03=1.000, loss=0.070]
train (3/3) accuracy01: 0.8980499999999992 | accuracy03: 0.9818833333333338 | loss: 0.5079761749581736
3/3 * Epoch (valid): 100% 313/313 [00:03<00:00, 79.54it/s, accuracy01=0.938, accuracy03=1.000, loss=0.412]
valid (3/3) accuracy01: 0.8629 | accuracy03: 0.9696000000000004 | loss: 0.7639917479090389
* Epoch (3/3)
Top best models:
logs/custom_classification/checkpoints/train.3.pth      0.7640
```

So, you can see once we hit run you are downloading the dataset here for number of epochs. So, 3 epochs we have defined. You can see all these metrics that is computed automatically and it is being logged inside your logs this part ok. And, best models is being saved here. So, best models is 3 out of 3.

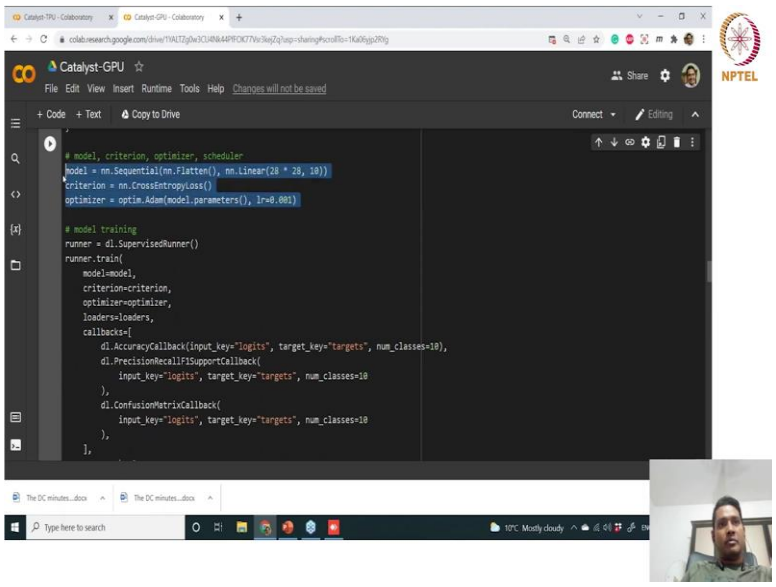
So, basically the third model is the best model and accuracy is you know for your loss is less. And so, this is just the 3 epochs that epochs showing. So, this was a simple pipeline for definition and more compaction you can do.

(Refer Slide Time: 16:28)


Catalyst-GPU
File Edit View Insert Runtime Tools Help Changes will not be saved
+ Code + Text Copy to Drive
Connect Editing
CV - Classification

```
import os
from torch import nn, optim
from torch.utils.data import DataLoader
from catalyst import dl
from catalyst.data import ToTensor
from catalyst.contrib.datasets import MNIST

# data
train_data = MNIST(os.getcwd(), train=True, download=True, transform=ToTensor())
valid_data = MNIST(os.getcwd(), train=False, download=True, transform=ToTensor())
loaders = {
    "train": DataLoader(train_data, batch_size=32),
    "valid": DataLoader(valid_data, batch_size=32),
}

# model, criterion, optimizer, scheduler
model = nn.Sequential(nn.Flatten(), nn.Linear(28 * 28, 10))
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)
```

(Refer Slide Time: 16:40)

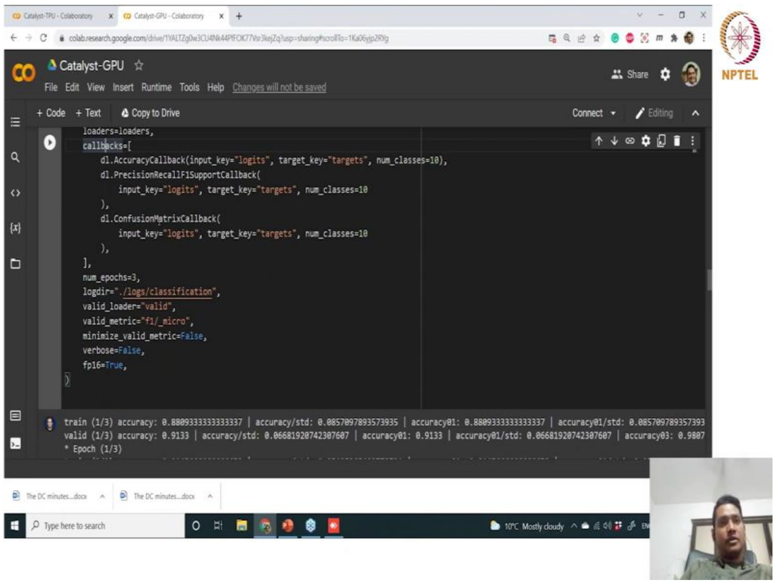

Catalyst-GPU
File Edit View Insert Runtime Tools Help Changes will not be saved
+ Code + Text Copy to Drive
Connect Editing

```
# model, criterion, optimizer, scheduler
model = nn.Sequential(nn.Flatten(), nn.Linear(28 * 28, 10))
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)

# model training
runner = dl.SupervisedRunner()
runner.train(
    model=model,
    criterion=criterion,
    optimizer=optimizer,
    loaders=loaders,
    callbacks=[
        dl.AccuracyCallback(input_key="logits", target_key="targets", num_classes=10),
        dl.PrecisionRecallF1SupportCallback(
            input_key="logits", target_key="targets", num_classes=10
        ),
        dl.ConfusionMatrixCallback(
            input_key="logits", target_key="targets", num_classes=10
        ),
    ],
```

So, if you see this classification. So, this is the same pipeline we are doing, but here the same training data, trainloader, DataLoader and valid loader. We have defined model, criterion and optimizer we have defined with this parameters. And, then we are using the SupervisedRunner as the global or universal training model ok. Now, inside this training model we are now defining some callbacks, because we want to have the callback for accuracy ok.

We want to callback we want to have callback for PrecisionRecallF1Support. So, all these will be taken care of instantly and without any writing of single piece of code. So, all these statistics will be actually stored. So, this is the callbacks support that I was talking about. So, instead of only running the model which is the vanilla in a model running for this runner ok, you can add the callbacks as you want to ok.

(Refer Slide Time: 17:36)



```
    loaders=loaders,
    callbacks=[
        dl.AccuracyCallback(input_key="logits", target_key="targets", num_classes=10),
        dl.PrecisionRecallF1SupportCallback(
            input_key="logits", target_key="targets", num_classes=10
        ),
        dl.ConfusionMatrixCallback(
            input_key="logits", target_key="targets", num_classes=10
        ),
    ],
    num_epochs=3,
    logdir="./logs/classification",
    valid_loader="valid",
    valid_metric="f1/_micro",
    minimize_valid_metric=False,
    verbose=False,
    fp16=True,
)
```

```
train (1/3) accuracy: 0.8809333333333337 | accuracy/std: 0.0857097893573935 | accuracy01: 0.8809333333333337 | accuracy01/std: 0.085709789357393
valid (1/3) accuracy: 0.9133 | accuracy/std: 0.06681920742307607 | accuracy01: 0.9133 | accuracy01/std: 0.06681920742307607 | accuracy03: 0.9807
* Epoch (1/3)
```

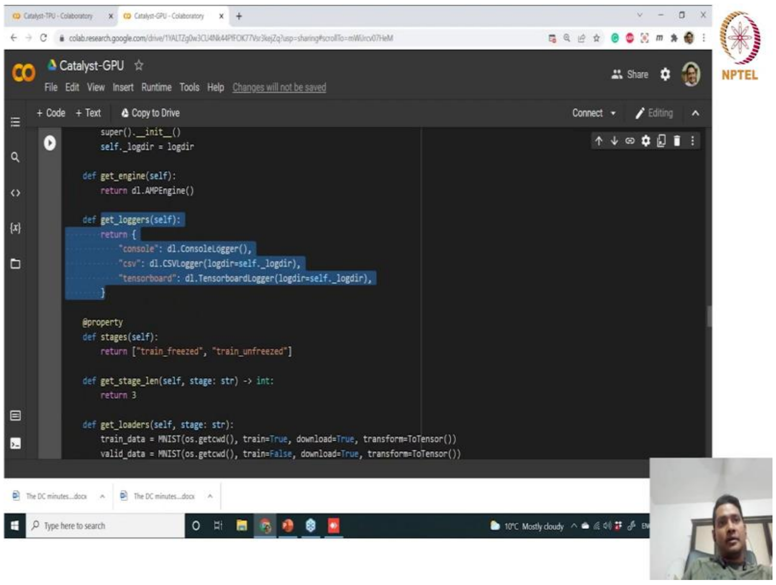
So, you want to have the ConfusionMatrixCallback. So, we added 3 callbacks here, number of epochs for directory and whether this we are doing the validity. Now, you can also support different validation metrics also ok. So, micro averaging, verbo averaging and also the weighted averaging, it supports f1 by this metrics. So, this metrics is also supported and you want to do let us say verbose False. And, also if you want to do the next precision training fp16 equal to True, you can enable it.

And, since we have enabled it you can see here that we are actually using Adam optimizer here but. So, once we have actually imported the dl; so, everything is imported here. You do not need to import the AMP explicitly. But, since everything is imported so, you can use mixed precision support automatically ok. And, this will actually transfer the mixed precision support to the native PyTorch support ok.

So, this will not use the Apex or NVIDIA Apex library, but if you use the native PyTorch mixed precision. So, one more thing here. So, as you are seeing that everything we are logging into this directory here for previous model as well right. So, in the logs directly with custom classification previously and this is the classification, simple classification with this callbacks. And, in the next we want to do the MultiStage Classification.

So, this is a bit advanced and this will actually help you to understand the flexibility of using catalyst in large scale training ok. So, class CustomRunner we have now we are now defining it. So, basically we are defining the engine here. So, AMPEngine we are using. So, again this is just to use AMP library and AMPEngine will enable the tensor board execution.

(Refer Slide Time: 20:11)

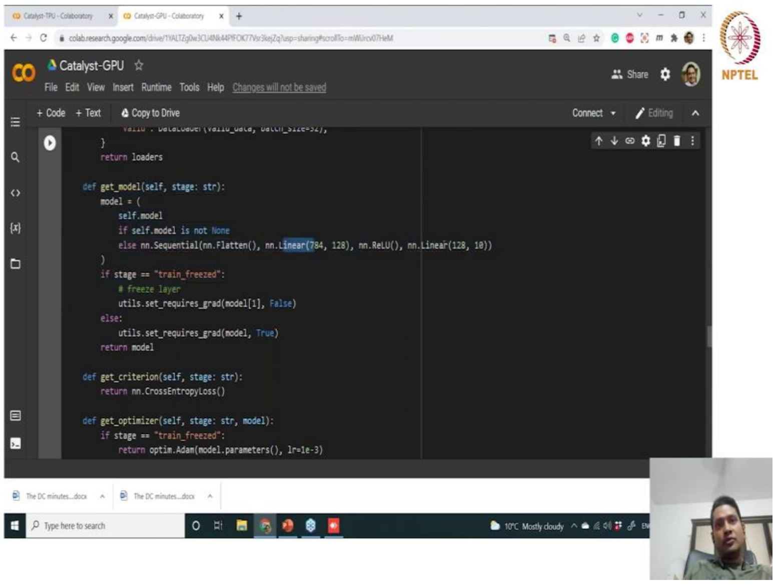

Catalyst-GPU
File Edit View Insert Runtime Tools Help Changes will not be saved
+ Code + Text Copy to Drive
NPTEL

```
        super().__init__()
        self._logdir = logdir

    def get_engine(self):
        return dl.AMPEngine()

    def get_loggers(self):
        return {
            "console": dl.ConsoleLogger(),
            "csv": dl.CSVLogger(logdir=self._logdir),
            "tensorboard": dl.TensorboardLogger(logdir=self._logdir),
        }

    @property
    def stages(self):
        return ["train_freezed", "train_unfreezed"]

    def get_stage_len(self, stage: str) -> int:
        return 3

    def get_loaders(self, stage: str):
        train_data = MNIST(os.getcwd(), train=True, download=True, transform=ToTensor())
        valid_data = MNIST(os.getcwd(), train=False, download=True, transform=ToTensor())
```

So, get logger will we want to have console, csv and tensorboard available to these directories and then we want to have multi stage pipeline for the training. So, now we want to have let us say two stages of training or two stages means let say first stage we want to do the training and the freeze the gradients.

And, then in the second stage we will use another optimizer and we will tune those parameters and do that. And, this is a very like normal way or typical way of training your big neural networks with at least two stages ok, to tune for your targeted data set. Because, whatever model you are using that is not particularly that will not be tuned for your particular data target data size.

(Refer Slide Time: 21:37)

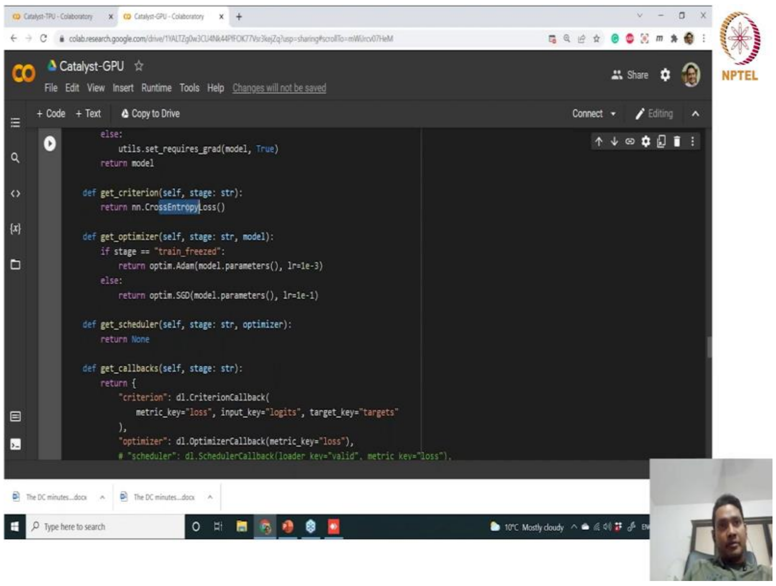


```
            "valid": DataLoader(valid_data, batch_size=32),
        }
        return loaders

    def get_model(self, stage: str):
        model = (
            self.model
            if self.model is not None
            else nn.Sequential(nn.Flatten(), nn.Linear(784, 128), nn.ReLU(), nn.Linear(128, 10))
        )
        if stage == "train_freezed":
            # freeze layer
            utils.set_requires_grad(model[1], False)
        else:
            utils.set_requires_grad(model, True)
        return model

    def get_criterion(self, stage: str):
        return nn.CrossEntropyLoss()

    def get_optimizer(self, stage: str, model):
        if stage == "train_freezed":
            return optim.Adam(model.parameters(), lr=1e-3)
```

So, this is a simple demonstration, how you can use this multi stage pipeline or manage multi stage pipeline inside your my catalyst custom runner right. And, we are saying that each stage will have 3 blocks and get the loaders, get the loaders, get the model. And so, get model will define the model. And so, basically here the model has two layers, two linear layers. And, then in the training phase stage we are freezing the gradients where we are not updating the gradients and here we are using the gradients to tune in the second stage.

So, basically these are the two stage. And, you can see how seamlessly easy to define multi stage training. Just two lines will help you to enable multi stage training without; so, 3 epochs we are using. So, you can imagine like how this will be once you try to code it from vanilla PyTorch training right.

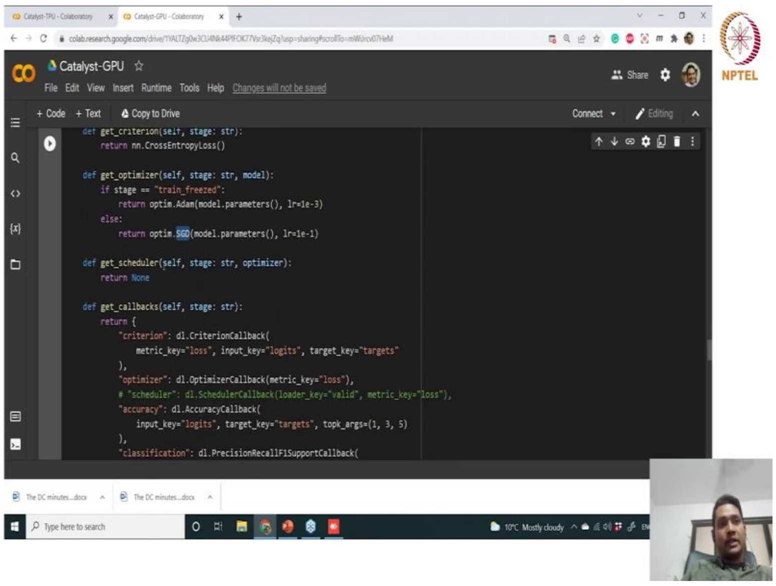
(Refer Slide Time: 22:25)



```
        else:
            utils.set_requires_grad(model, True)
        return model

    def get_criterion(self, stage: str):
        return nn.CrossEntropyLoss()

    def get_optimizer(self, stage: str, model):
        if stage == "train_freezed":
            return optim.Adam(model.parameters(), lr=1e-3)
        else:
            return optim.SGD(model.parameters(), lr=1e-1)

    def get_scheduler(self, stage: str, optimizer):
        return None

    def get_callbacks(self, stage: str):
        return {
            "criterion": dl.CriterionCallback(
                metric_key="loss", input_key="logits", target_key="targets"
            ),
            "optimizer": dl.OptimizerCallback(metric_key="loss"),
            # "scheduler": dl.SchedulerCallback(loader_key="valid", metric_key="loss"),
```

Now, get criterion you just do this. So, basically we are defining CrossEntropyLoss and optimizer for freezing stage, we are using Adam optimizer and for non-freezing stage, we are using SGD optimizer. So, this is also typical practice of using several optimizers into several stages to see what is the performance gain in the end.

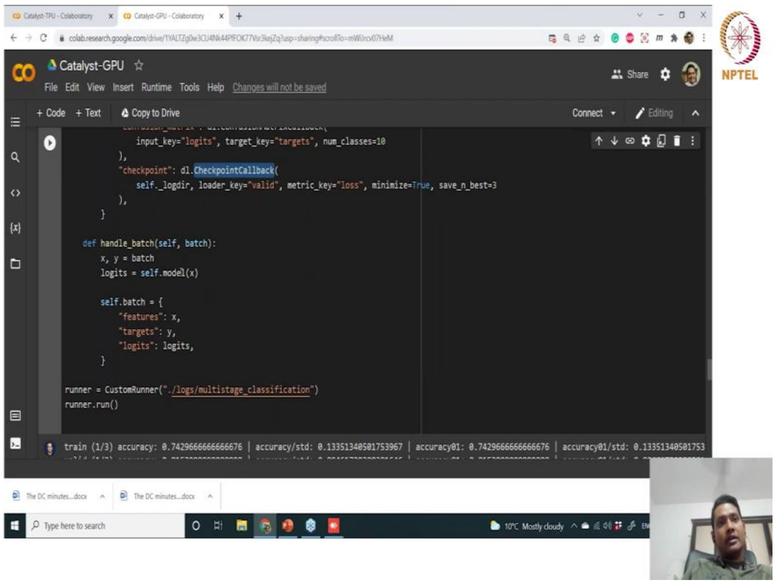
(Refer Slide Time: 22:50)



```
    def get_criterion(self, stage: str):
        return nn.CrossEntropyLoss()

    def get_optimizer(self, stage: str, model):
        if stage == "train_freezed":
            return optim.Adam(model.parameters(), lr=1e-3)
        else:
            return optim.SGD(model.parameters(), lr=1e-1)

    def get_scheduler(self, stage: str, optimizer):
        return None

    def get_callbacks(self, stage: str):
        return {
            "criterion": dl.CriterionCallback(
                metric_key="loss", input_key="logits", target_key="targets"
            ),
            "optimizer": dl.OptimizerCallback(metric_key="loss"),
            # "scheduler": dl.SchedulerCallback(loader_key="valid", metric_key="loss"),
            "accuracy": dl.AccuracyCallback(
                input_key="logits", target_key="targets", topk_args=(1, 3, 5)
            ),
            "classification": dl.PrecisionRecallF1SupportCallback(
```

And, for sure you will get better performance in it get scheduler. Now, this is to define whether how this optimizer will get scheduled inside your training pipeline. And, now in get callbacks we are defining all the callbacks for let us say Criterion, Accuracy, Optimizer, PrecisionRecallF1Support, ConfusionMatrix, CheckpointCallbacks.

(Refer Slide Time: 23:11)

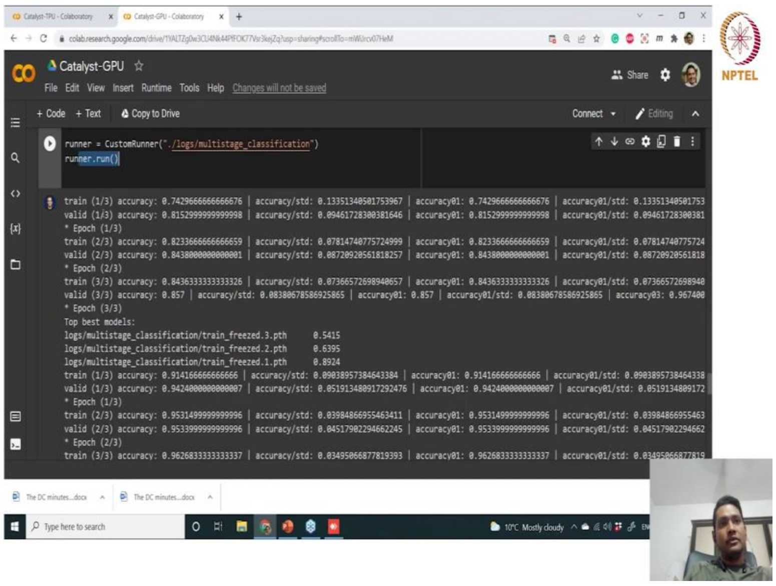


```
                input_key="logits", target_key="targets", num_classes=10
            ),
            "checkpoint": dl.CheckpointCallback(
                self._logdir, loader_key="valid", metric_key="loss", minimize=True, save_n_best=3
            ),
        }

    def handle_batch(self, batch):
        x, y = batch
        logits = self.model(x)

        self.batch = {
            "features": x,
            "targets": y,
            "logits": logits,
        }

runner = CustomRunner("./logs/multistage_classification")
runner.run()
```

```
train (1/3) accuracy: 0.7429666666666676 | accuracy/std: 0.13351340501753967 | accuracy01: 0.7429666666666676 | accuracy01/std: 0.13351340501753
```

So, CheckpointCallback will help you to in the end minimize the loss and save the best 3 choices. Let us say I want to say the best 4 choices, 5 choices, 3 choices or maybe top model with this metrics right. So, all these you can do just defining this context that is it, as simple as that. And, handle batch we actually do the training and defining the abstraction and hit the run and that will do the job.

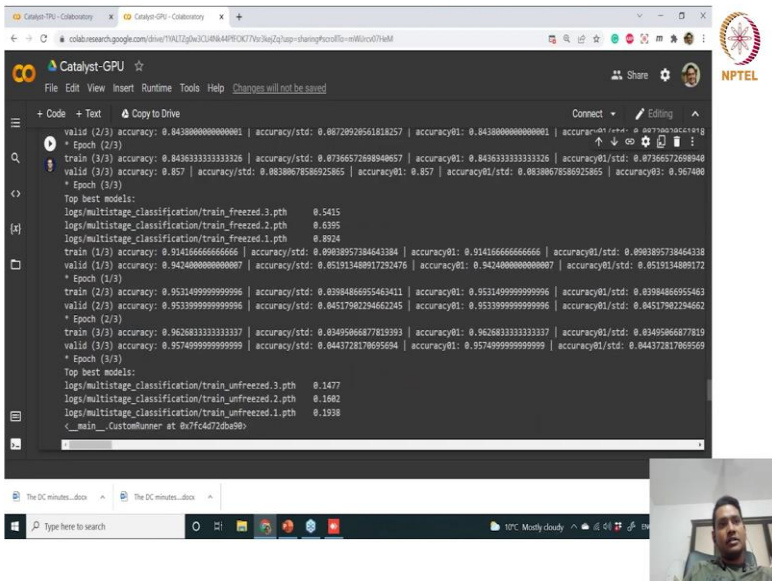
(Refer Slide Time: 23:45)



```
runner = CustomRunner("./logs/multistage_classification")
runner.run()
```

```
train (1/3) accuracy: 0.7429666666666676 | accuracy/std: 0.13351340501753967 | accuracy01: 0.7429666666666676 | accuracy01/std: 0.13351340501753
valid (1/3) accuracy: 0.8152999999999998 | accuracy/std: 0.09461728300381646 | accuracy01: 0.8152999999999998 | accuracy01/std: 0.09461728300381
* Epoch (1/3)
train (2/3) accuracy: 0.8233666666666659 | accuracy/std: 0.07814740775724999 | accuracy01: 0.8233666666666659 | accuracy01/std: 0.07814740775724
valid (2/3) accuracy: 0.8438000000000001 | accuracy/std: 0.08720920561818257 | accuracy01: 0.8438000000000001 | accuracy01/std: 0.08720920561818
* Epoch (2/3)
train (3/3) accuracy: 0.8436333333333326 | accuracy/std: 0.07366572698940657 | accuracy01: 0.8436333333333326 | accuracy01/std: 0.07366572698940
valid (3/3) accuracy: 0.857 | accuracy/std: 0.08380678586925865 | accuracy01: 0.857 | accuracy01/std: 0.08380678586925865 | accuracy03: 0.967400
* Epoch (3/3)
Top best models:
logs/multistage_classification/train_freezed.3.pth     0.5415
logs/multistage_classification/train_freezed.2.pth     0.6395
logs/multistage_classification/train_freezed.1.pth     0.8924
train (1/3) accuracy: 0.914166666666666 | accuracy/std: 0.09038957384643384 | accuracy01: 0.914166666666666 | accuracy01/std: 0.0903895738464338
valid (1/3) accuracy: 0.9424000000000007 | accuracy/std: 0.051913480917292476 | accuracy01: 0.9424000000000007 | accuracy01/std: 0.0519134809172
* Epoch (1/3)
train (2/3) accuracy: 0.9531499999999996 | accuracy/std: 0.03984866955463411 | accuracy01: 0.9531499999999996 | accuracy01/std: 0.03984866955463
valid (2/3) accuracy: 0.9533999999999996 | accuracy/std: 0.04517902294662245 | accuracy01: 0.9533999999999996 | accuracy01/std: 0.04517902294662
* Epoch (2/3)
train (3/3) accuracy: 0.9626833333333337 | accuracy/std: 0.03495066877819393 | accuracy01: 0.9626833333333337 | accuracy01/std: 0.03495066877819
```

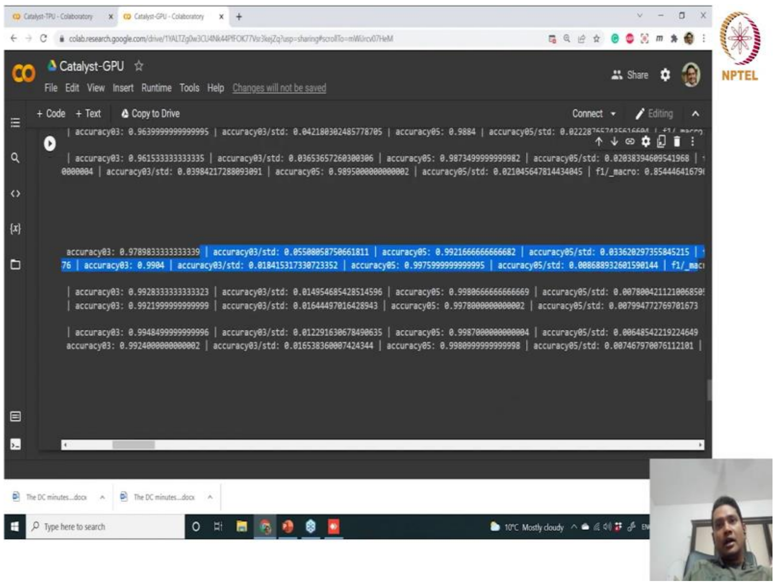
(Refer Slide Time: 23:50)



```
valid (2/3) accuracy: 0.8438000000000001 | accuracy/std: 0.08720920561818257 | accuracy01: 0.8438000000000001 | accuracy01/std: 0.08720920561818
* Epoch (2/3)
train (3/3) accuracy: 0.8436333333333326 | accuracy/std: 0.07366572698940657 | accuracy01: 0.8436333333333326 | accuracy01/std: 0.07366572698940
valid (3/3) accuracy: 0.857 | accuracy/std: 0.08380678586925865 | accuracy01: 0.857 | accuracy01/std: 0.08380678586925865 | accuracy03: 0.967400
* Epoch (3/3)
Top best models:
logs/multistage_classification/train_freezed.3.pth      0.5415
logs/multistage_classification/train_freezed.2.pth      0.6395
logs/multistage_classification/train_freezed.1.pth      0.8924
train (1/3) accuracy: 0.914166666666666 | accuracy/std: 0.09038957384643384 | accuracy01: 0.914166666666666 | accuracy01/std: 0.0903895738464338
valid (1/3) accuracy: 0.9424000000000007 | accuracy/std: 0.051913480917292476 | accuracy01: 0.9424000000000007 | accuracy01/std: 0.0519134809172
* Epoch (1/3)
train (2/3) accuracy: 0.9531499999999996 | accuracy/std: 0.03984866955463411 | accuracy01: 0.9531499999999996 | accuracy01/std: 0.03984866955463
valid (2/3) accuracy: 0.9533999999999996 | accuracy/std: 0.04517902294662245 | accuracy01: 0.9533999999999996 | accuracy01/std: 0.04517902294662
* Epoch (2/3)
train (3/3) accuracy: 0.9626833333333337 | accuracy/std: 0.03495066877819393 | accuracy01: 0.9626833333333337 | accuracy01/std: 0.03495066877819
valid (3/3) accuracy: 0.9574999999999999 | accuracy/std: 0.0443728170695694 | accuracy01: 0.9574999999999999 | accuracy01/std: 0.044372817069569
* Epoch (3/3)
Top best models:
logs/multistage_classification/train_unfreezed.3.pth    0.1477
logs/multistage_classification/train_unfreezed.2.pth    0.1602
logs/multistage_classification/train_unfreezed.1.pth    0.1938
<__main__.CustomRunner at 0x7fc4d72dba90>
```

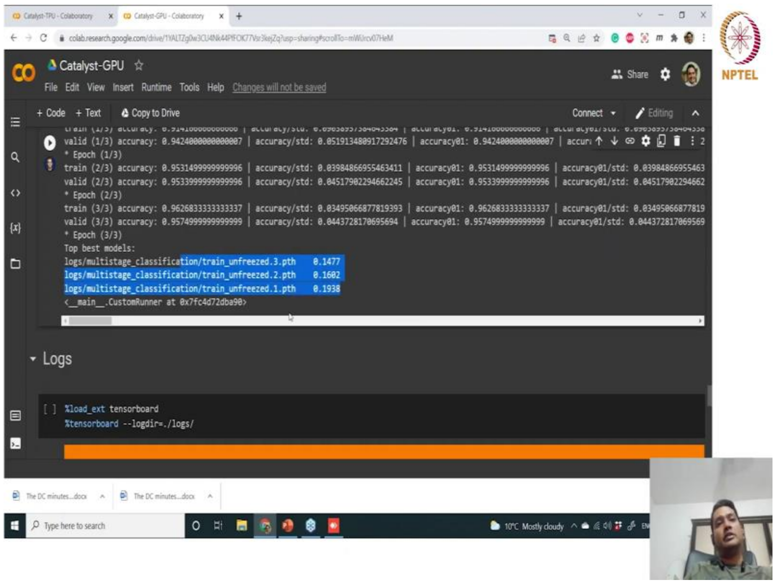
So, 1 2 3 or the first stage, multi stage, we switch to the second stage. Training on to 3 epochs and you will see the metrics. All the metrics are computed automatically, the accuracy, precision recall, f1 support.

(Refer Slide Time: 24:02)



```
 | accuracy03: 0.9639999999999995 | accuracy03/std: 0.042180302485778705 | accuracy05: 0.9884 | accuracy05/std: 0.02228765574356166 | f1/_macro

 | accuracy03: 0.9615333333333335 | accuracy03/std: 0.03653657260300306 | accuracy05: 0.9873499999999982 | accuracy05/std: 0.020383946095419 68 |
0000004 | accuracy03/std: 0.03984217288093091 | accuracy05: 0.9895000000000002 | accuracy05/std: 0.021045647814434045 | f1/_macro: 0.85444641679



 accuracy03: 0.9789833333333339 | accuracy03/std: 0.05500058750661811 | accuracy05: 0.9921666666666682 | accuracy05/std: 0.03362029735584 5215 |
76 | accuracy03: 0.9904 | accuracy03/std: 0.01841531733072 3352 | accuracy05: 0.9975999999999995 | accuracy05/std: 0.008688932601590144 | f1/_mac

 | accuracy03: 0.9928333333333323 | accuracy03/std: 0.014954685428514596 | accuracy05: 0.9980666666666669 | accuracy05/std: 0.00780042112100685
 | accuracy03: 0.9921999999999999 | accuracy03/std: 0.01644497016428943 | accuracy05: 0.9978000000000002 | accuracy05/std: 0.00799477276970 1673

 | accuracy03: 0.9948499999999996 | accuracy03/std: 0.012291630678490635 | accuracy05: 0.9987000000000004 | accuracy05/std: 0.00648542219224649
 accuracy03: 0.9924000000000002 | accuracy03/std: 0.01653836000 7424344 | accuracy05: 0.9980999999999998 | accuracy05/std: 0.007467970076112101 |
```

(Refer Slide Time: 24:10)



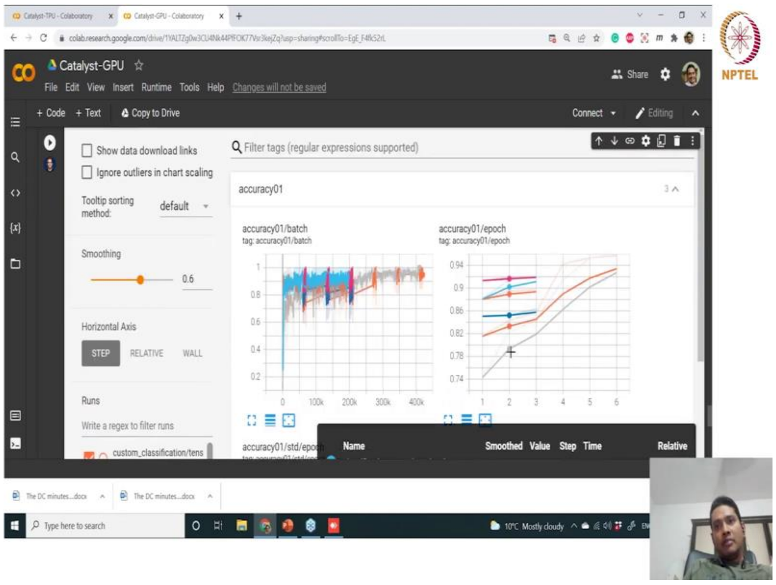
```
valid (1/3) accuracy: 0.9424000000000007 | accuracy/std: 0.051913488917292476 | accuracy01: 0.9424000000000007 | accur
* Epoch (1/3)
train (2/3) accuracy: 0.9531499999999996 | accuracy/std: 0.03984866955463411 | accuracy01: 0.9531499999999996 | accuracy01/std: 0.03984866955463
valid (2/3) accuracy: 0.9533999999999996 | accuracy/std: 0.04517902294662245 | accuracy01: 0.9533999999999996 | accuracy01/std: 0.04517902294662
* Epoch (2/3)
train (3/3) accuracy: 0.9626833333333337 | accuracy/std: 0.03495066877819393 | accuracy01: 0.9626833333333337 | accuracy01/std: 0.03495066877819
valid (3/3) accuracy: 0.9574999999999999 | accuracy/std: 0.0443728170695694 | accuracy01: 0.9574999999999999 | accuracy01/std: 0.044372817069569
* Epoch (3/3)
Top best models:
logs/multistage_classification/train_unfreezed.3.pth    0.1477
logs/multistage_classification/train_unfreezed.2.pth    0.1602
logs/multistage_classification/train_unfreezed.1.pth    0.1938
<__main__.CustomRunner at 0x7fc4d72dba90>
```

Logs

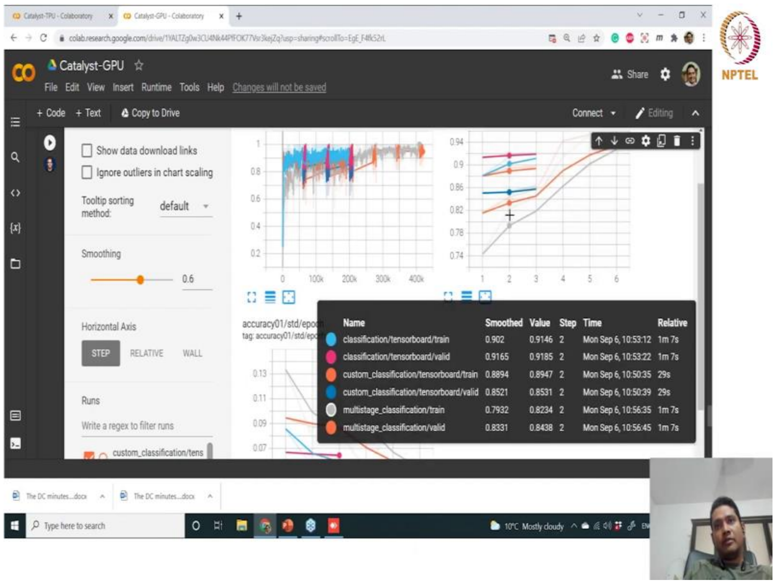
```
%load_ext tensorboard
%tensorboard --logdir=./logs/
```

So, everything is computed automatically and they have been logged also depending on the path that we have defined. And, the first three best performance model also is set in this path ok. So, that is that much of ease you will get using this framework to define the complex pipeline of MultiStage Classification or even multi stage training of your networks right.

(Refer Slide Time: 24:36)



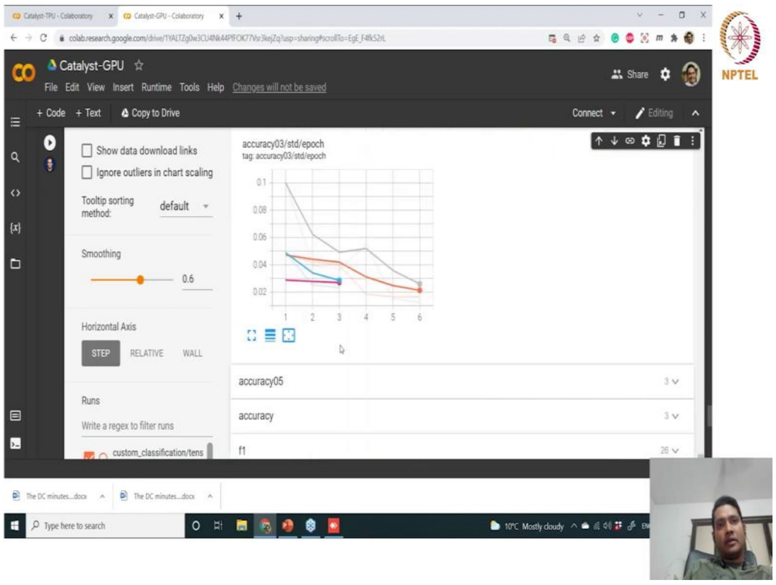
(Refer Slide Time: 24:53)


Catalyst-GPU
File Edit View Insert Runtime Tools Help Changes will not be saved
Show data download links
Ignore outliers in chart scaling
Tooltip sorting method: default
Smoothing 0.6
Horizontal Axis STEP RELATIVE WALL
Runs
Write a regex to filter runs
custom_classification/tens
accuracy01/std/epoch
tag: accuracy01/std/epoch
NPTEL

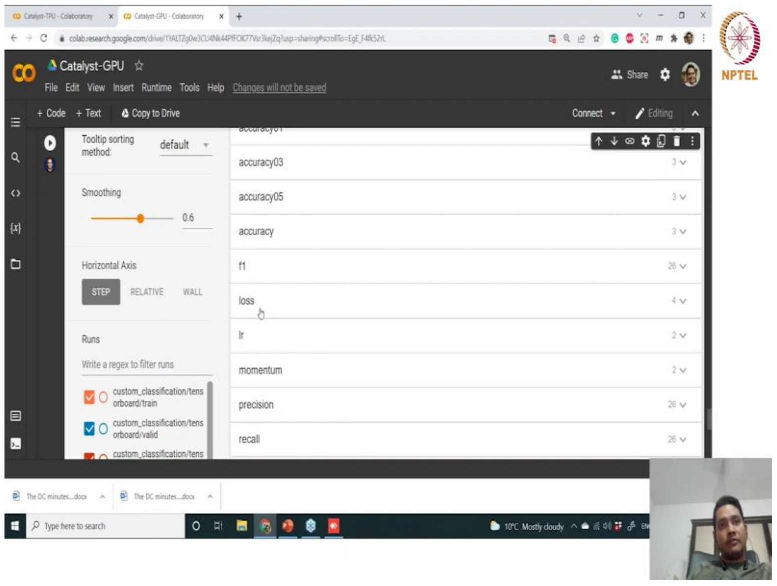
	Name	Smoothed	Value	Step	Time	Relative
●	classification/tensorboard/train	0.902	0.9146	2	Mon Sep 6, 10:53:12	1m 7s
●	classification/tensorboard/valid	0.9165	0.9185	2	Mon Sep 6, 10:53:22	1m 7s
●	custom_classification/tensorboard/train	0.8894	0.8947	2	Mon Sep 6, 10:50:35	29s
●	custom_classification/tensorboard/valid	0.8521	0.8531	2	Mon Sep 6, 10:50:39	29s
●	multistage_classification/train	0.7932	0.8234	2	Mon Sep 6, 10:56:35	1m 7s
●	multistage_classification/valid	0.8331	0.8438	2	Mon Sep 6, 10:56:45	1m 7s

Now, to see the logs. Again you can use tensorboard here to get the profiles. So, as you can see here. So, for the multi stage pipeline. So, basically this is the multi stage classification that we have done for right, got it. And so, you can see after first stage, the accuracy has hopped up ok.

(Refer Slide Time: 25:11)

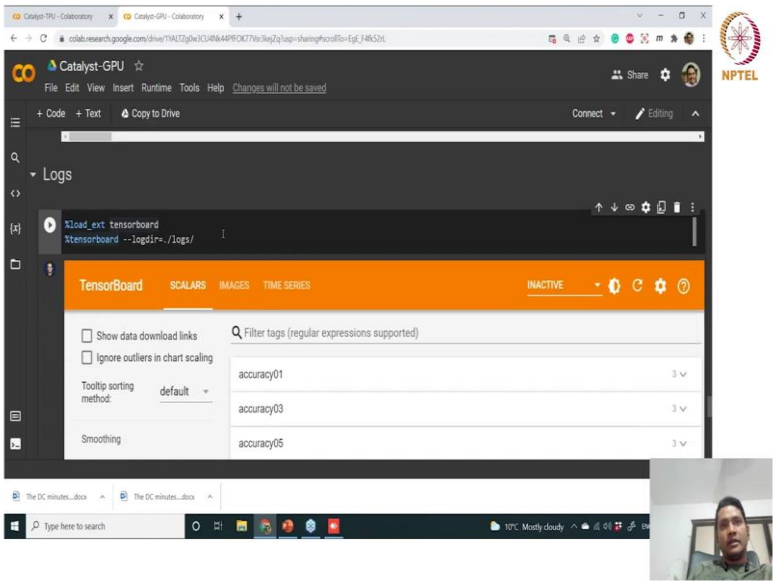

Catalyst-GPU
File Edit View Insert Runtime Tools Help Changes will not be saved
Show data download links
Ignore outliers in chart scaling
Tooltip sorting method: default
Smoothing 0.6
Horizontal Axis STEP RELATIVE WALL
Runs
Write a regex to filter runs
custom_classification/tens
accuracy03/std/epoch
tag: accuracy03/std/epoch
accuracy05
accuracy
f1
NPTEL

(Refer Slide Time: 25:22)



And so, this is the typical way of training your model and you can see for all the basically for different parameters, all the parameters have been checkpointed here. You can see accuracy, F1 score, loss, your learning rate, momentum, precision, recall, support etc and also yeah.

(Refer Slide Time: 25:33)

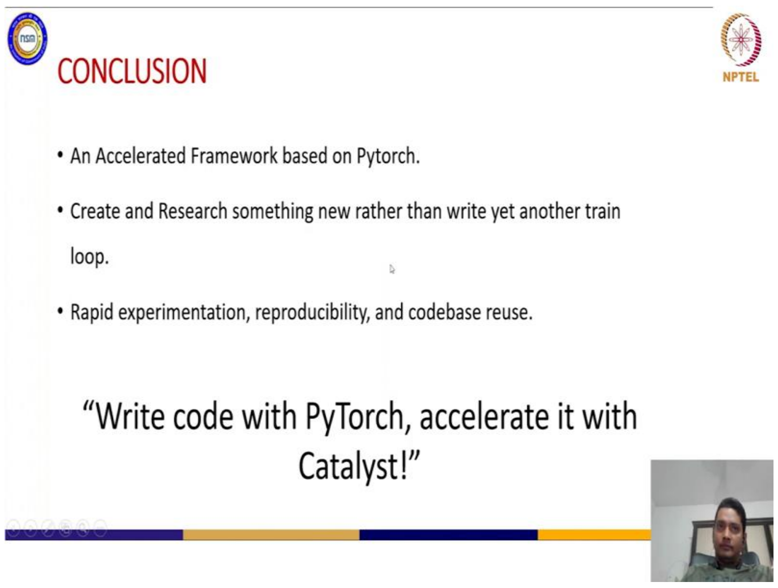


```
%load_ext tensorboard
%tensorboard --logdir=./logs/
```

So, this is you just you can just dig on it and see what kind of analysis you can do. So, this is the catalyst that we were talking. Also in the slide if you see ok, let us go quickly to the demo. So, in this section we have also given one name for you to refer the catalyst run in. So, here we are seeing catalyst run in GPU. You can also refer another name for catalyst run in TPU ok.

So, this is just one or two lines of additional code that you need to do for your TPU and you can change the run runtime for TPU. And, in the next week we will see more of TPU exhibition. But, we will not discuss TPU exhibition right now. But, you can refer to the link for how we can modify your training model for your TPU time ok.

(Refer Slide Time: 26:36)



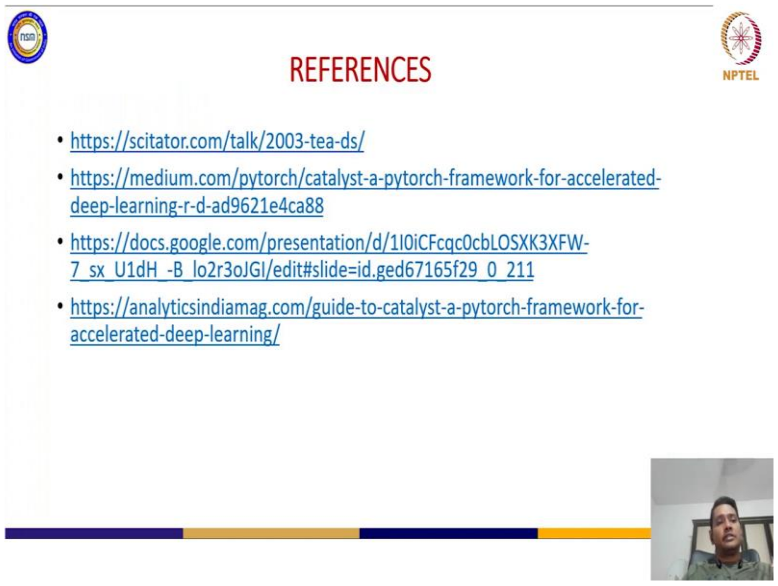
CONCLUSION

- An Accelerated Framework based on Pytorch.
- Create and Research something new rather than write yet another train loop.
- Rapid experimentation, reproducibility, and codebase reuse.

"Write code with PyTorch, accelerate it with Catalyst!"

So, to conclude these catalyst, this accelerated framework for based on PyTorch and it takes you to write your frame looks very very easily. And, you can do some rapid experimentation with reproducibility, codebase to use also. This is very important because once you have generated the config file, you can just reuse it for different trainings. So, as what he has written here, I do not know. This is the quote from the initiator; "Write code with PyTorch, accelerate it with Catalyst!" ok.

(Refer Slide Time: 27:23)



REFERENCES

- https://scitator.com/talk/2003-tea-ds/
- https://medium.com/pytorch/catalyst-a-pytorch-framework-for-accelerated-deep-learning-r-d-ad9621e4ca88
- https://docs.google.com/presentation/d/1I0iCFcqc0cbLOSXK3XFW-7_sx_U1dH_-B_lo2r3oJGI/edit#slide=id.ged67165f29_0_211
- https://analyticsindiamag.com/guide-to-catalyst-a-pytorch-framework-for-accelerated-deep-learning/

Thank you, with this references and we will go to the question answer session.