# PyTorch Introduction
# Training a network

David Rotermund & Udo Ernst

# Open Book: Dive into Deep Learning

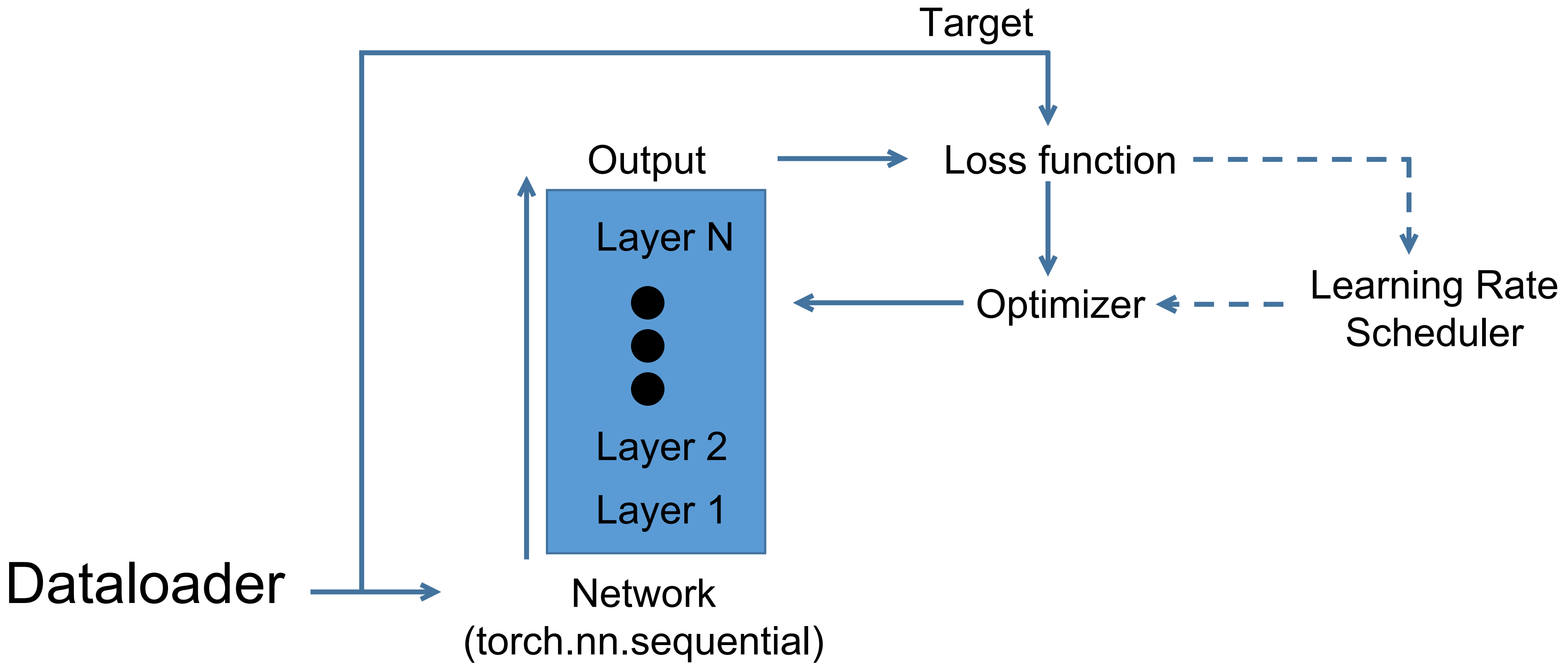
- ASTON ZHANG
- ZACHARY C. LIPTON
- MU LI
- ALEXANDER J. SMOLA

https://d2l.ai/d2l-en.pdf

Fig. 1.3.3 A donkey, a dog, a cat, and a rooster.

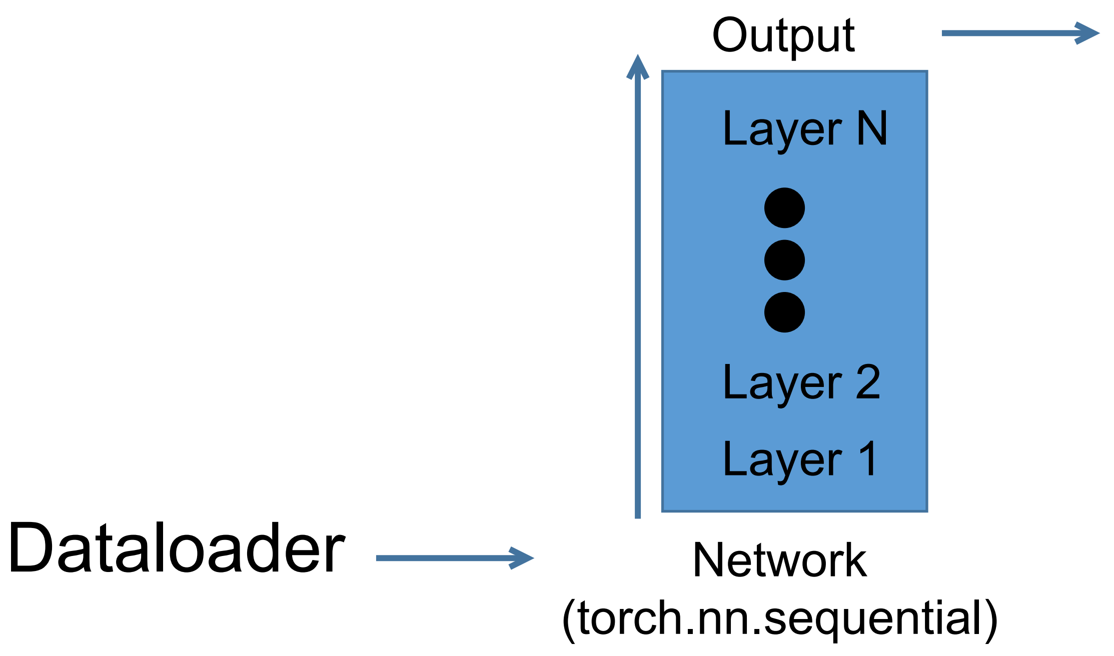
# Anatomy of a PyTorch Network + Support

**Training of the weights**



# Anatomy of a PyTorch Network + Support

**Inference**



# torch.tensor is the numpy.ndarray of PyTorch

== torch.tensor

== stored data

# Example network

I contains everything but it is not very well optimized.

Allowing **you** to improve on it.

(Or depending on your computer, switch it to the Fashion MNIST benchmark first...)

```python
import torch
import torchvision  # type: ignore
from torchvision.transforms import v2  # type: ignore
import time
import os

number_of_epoch: int = 500
lr_parameter_max: float = 1e-9

ModelsPath: str = "Models"
os.makedirs(ModelsPath, exist_ok=True)

# Tensorboard
os.environ["TF_CPP_MIN_LOG_LEVEL"] = "3"
from torch.utils.tensorboard
import SummaryWriter

tb = SummaryWriter(log_dir="run")
```

**The imports**

**Setting up tensorboard**

```python
# GPU ?
device: torch.device = (
    torch.device("cuda:0") if torch.cuda.is_available() else torch.device("cpu")
)
torch.set_default_dtype(torch.float32)

# Data augmentation
test_processing_chain = v2.Compose(
    transforms=[
        v2.ToImage(),
        v2.ToDtype(torch.float32, scale=True),
        v2.CenterCrop((28, 28)),
    ],
)

train_processing_chain = v2.Compose(
    transforms=[
        v2.ToImage(),
        v2.ToDtype(torch.float32, scale=True),
        v2.RandomCrop((28, 28)),
        v2.AutoAugment(),
    ],
)
```

**GPU: Yes / No?**

**Data augmentation for test data**

**Data augmentation for training data**

```python
# Data provider
tv_dataset_train = torchvision.datasets.CIFAR10(
    root="data",
    train=True,
    download=True,
    transform=train_processing_chain,
)
tv_dataset_test = torchvision.datasets.CIFAR10(
    root="data",
    train=False,
    download=True,
    transform=test_processing_chain,
)
```

**CIFAR10 training dataset (images, labels)**

**CIFAR10 test dataset (images, labels)**

```
# Data loader
train_data_load = torch.utils.data.DataLoader(
    tv_dataset_train, batch_size=100, shuffle=True)

test_data_load = torch.utils.data.DataLoader(
    tv_dataset_test, batch_size=100, shuffle=False)

# Network
network = torch.nn.Sequential(
    torch.nn.Conv2d(
        in_channels=3,
        out_channels=32,
        kernel_size=5,
        stride=1,
        padding=0,
    ),
    torch.nn.ReLU(),
    torch.nn.BatchNorm2d(32),
    torch.nn.MaxPool2d(kernel_size=2, stride=2, padding=0),
```

**data loader training data**

**data loader test data**

**network (layer in sequential container)**

**2D convolutional Layer (3 channel in, 32 out, 5x5 kernel, 1x1 stride, no padding)**

**ReLu Layer**

**BatchNorm2D Layer**

**max pooling (2x2 kernel, 2x2 stride, no padding)**

```
torch.nn.Conv2d(
    in_channels=32,
    out_channels=64,
    kernel_size=5,
    stride=1,
    padding=0,
),
torch.nn.ReLU(),
torch.nn.BatchNorm2d(64),
torch.nn.MaxPool2d(kernel_size=2, stride=2, padding=0),
torch.nn.Flatten(
    start_dim=1,
),
torch.nn.Linear(
    in_features=1024,
    out_features=1024,
    bias=True,
),
```

**2D convolutional Layer (32 channel in, 64 out, 5x5 kernel, 1x1 stride, no padding)**

**ReLu Layer**

**BatchNorm2D Layer**

**max pooling (2x2 kernel, 2x2 stride, no padding)**

**Batch x 4 x 4 x 64 -> Batch x 1024**

**Fully connected layer (1024 x 1024, with bias)**

```
    torch.nn.ReLU(),
    torch.nn.Linear(
        in_features=1024,
        out_features=10,
        bias=True,
    ),
).to(device)

# Optimizer
optimizer = torch.optim.Adam(network.parameters(), lr=0.001)

# LR Scheduler
lr_scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(optimizer)

# Loss function
loss_function = torch.nn.CrossEntropyLoss()
```

**ReLu layer**

**Fully connected layer (1024 x 10, with bias)**

**No ReLU / Softmax since CrossEntropyLoss contains Softmax**

**Network to GPU if there is a GPU**

**Adam optimizer**

**Learning rate scheduler**

**Loss function**

```python
# Main loop
for epoch_id in range(0, number_of_epoch):
    print(f"Epoch: {epoch_id}")
    t_start: float = time.perf_counter()

    train_loss: float = 0.0
    train_correct: int = 0
    train_number: int = 0
    test_correct: int = 0
    test_number: int = 0

    # Switch the network into training mode
    network.train()
```

**Main loop over up to 500 epochs**

**Getting a timestamp**

**Setting the network into training mode**

```python
# This runs in total for one epoch split up into mini-batches
for image, target in train_data_load:
    # Clean the gradient
    optimizer.zero_grad()

    # Run data through network
    output = network(image.to(device))

    # Measure the loss
    loss = loss_function(output, target.to(device))
    train_loss += loss.item()

    # Classifiy
    train_correct += (
        (output.argmax(dim=1) == target.to(device)).sum().detach().cpu().numpy()
    )
    train_number += target.shape[0]

    # Calculate backprop
    loss.backward()

    # Update the parameter
    optimizer.step()
```

**Training mini batches**

**Clean the accumulated gradients from the parameter**

**Pushing the mini batch through the network**

**Getting the loss for the mini batch**

**Comparing the argmax(output) with the target**

**Performing error back-propagation**

**Updating the parameter based on this mini batch**

```python
# Update the learning rate
lr_scheduler.step(train_loss)

t_training: float = time.perf_counter()

# Switch the network into evalution mode
network.eval()
with torch.no_grad():
    for image, target in test_data_load:

        # Run data through network
        output = network(image.to(device))

        # Classifiy
        test_correct += (
            (output.argmax(dim=1) == target.to(device)).sum().detach().cpu().numpy()
        )

        test_number += target.shape[0]
```

**Learning rate scheduler checks if learning rate needs adjustment**

**Setting the network into evaluation mode**

**We don’t need gradients**

**Pushing the mini batch through the network**

**Comparing the argmax(output) with the target**

```python
t_testing = time.perf_counter()

perfomance_test_correct: float = 100.0 * test_correct / test_number
perfomance_train_correct: float = 100.0 * train_correct / train_number

tb.add_scalar("Train Loss", train_loss, epoch_id)
tb.add_scalar("Train Number Correct", train_correct, epoch_id)
tb.add_scalar("Test Number Correct", test_correct, epoch_id)
tb.add_scalar("Error Test", 100.0 - perfomance_test_correct, epoch_id)
tb.add_scalar("Error Train", 100.0 - perfomance_train_correct, epoch_id)
tb.add_scalar("Learning Rate", optimizer.param_groups[-1]["lr"], epoch_id)
tb.flush()
```

**Store the data in Tensorboard**

```
print(
  f"Training: Loss={train_loss:.5f} Correct={perfomance_train_correct:.2f}% LR:{optimizer.param_groups[-1]["lr"]}"
)
print(f"Testing: Correct={perfomance_test_correct:.2f}%")
print(
  f"Time: Training={(t_training - t_start):.1f}sec, Testing={(t_testing - t_training):.1f}sec"
)
```

**Save the network**

```
torch.save(network, os.path.join(ModelsPath, f"Model_MNIST_A_{epoch_id}.pt"))

print()
```

**Did we reach the final learning rate?**

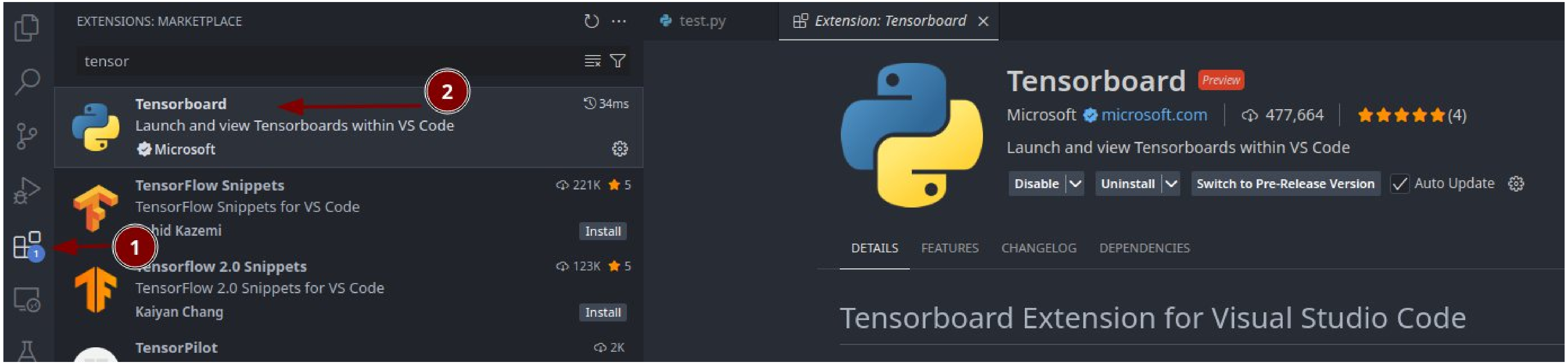
```
if optimizer.param_groups[-1]["lr"] < lr_parameter_max:
    tb.close()
    print("Done (lr_limit)")
    exit()

tb.close()
```

**Close the Tensorboard connection**

# VS Code Tensorboard extension

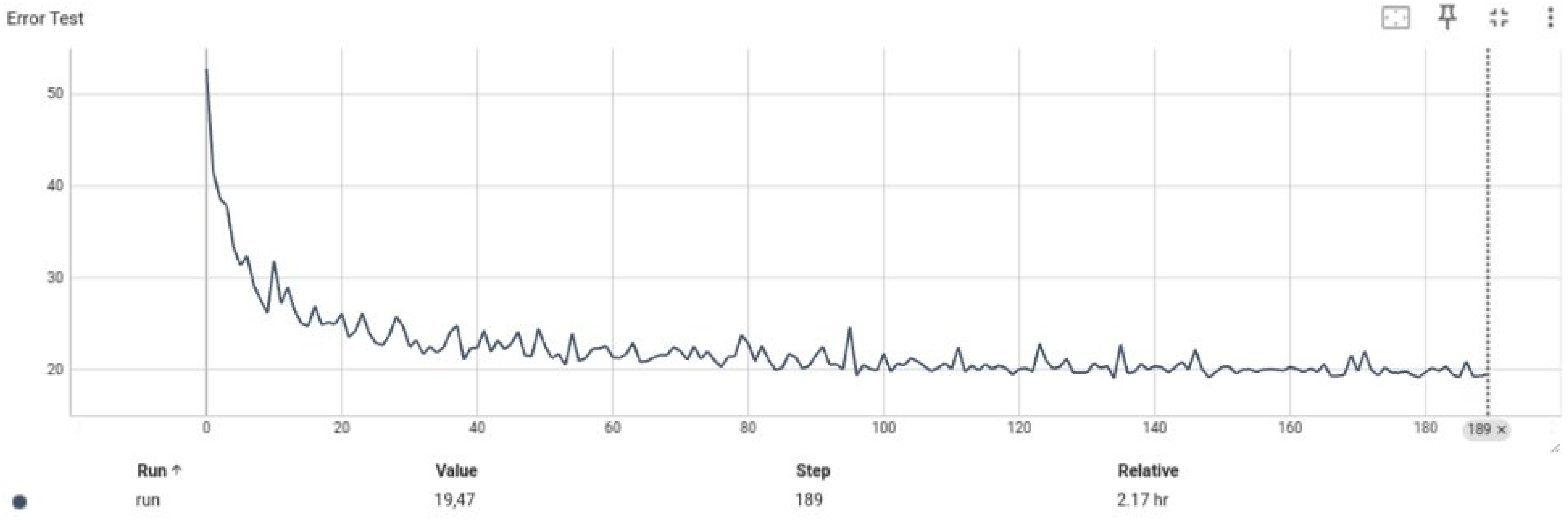
Observing the learning process...



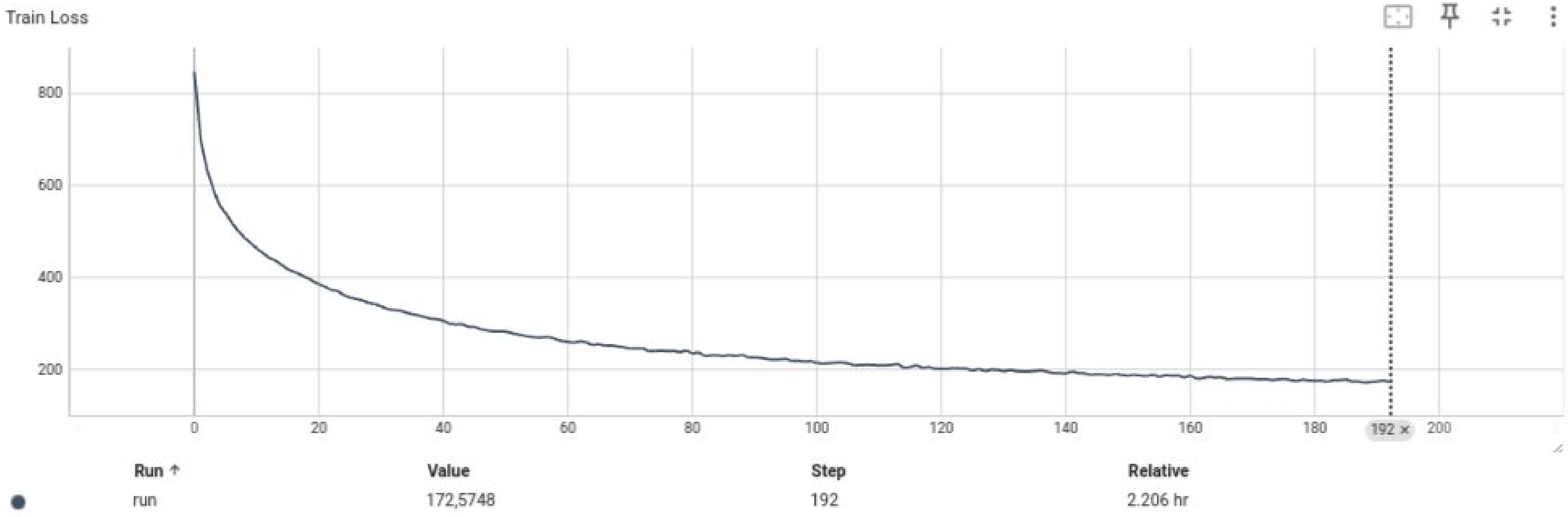

```python
# Tensorboard
os.environ["TF_CPP_MIN_LOG_LEVEL"] = "3"
▶ Launch TensorBoard Session
from torch.utils.tensorboard import SummaryWriter

tb = SummaryWriter(log_dir="run")
```

# A complete network... but not a good one



# A complete network... but not a good one



# A complete network... but not a good one

e.g. bad LR Scheduler settings