



by **Gregor Koehler** · Feb 17 2020

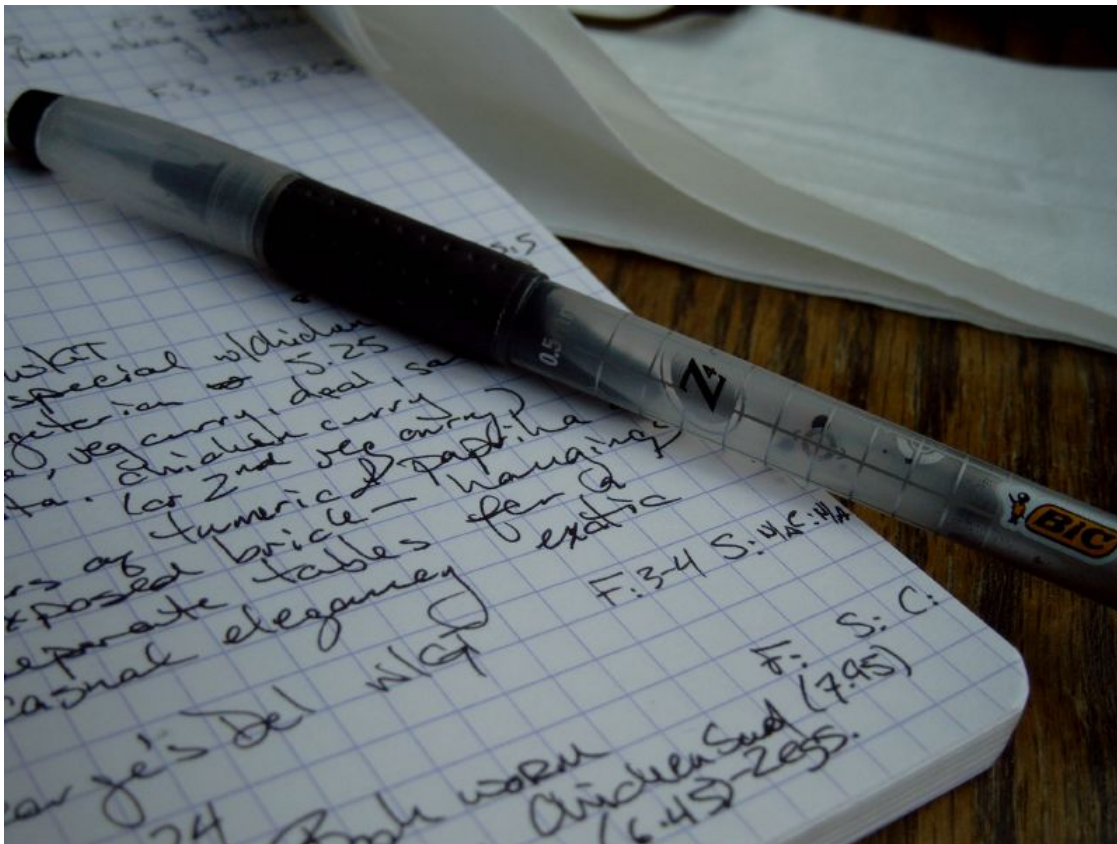
PhD Student in Computer Science. Articles about Machine Learning.

with **Andrea Amantini** and **Philipp Markovics**

Remix of **PyTorch Environment** by **Martin Kavalár**

# MNIST Handwritten Digit Recognition in PyTorch

In this article we'll build a simple convolutional neural network in PyTorch and train it to recognize handwritten digits using the MNIST dataset. Training a *classifier* on the MNIST dataset can be regarded as the *hello world* of image recognition.



cheapeats, Notes (<https://www.flickr.com/photos/cheapeats/55154820/in/photolist-5SFBb-dJHpHx-bpXtXc-bpPx1g-9nfgiq-a1p73h-EZSGK-gHE3bY-nkU9D-eza7r-cDUaqf-6cgrN3-8Y36fN-9H1M8e-8XZ3n6-6TWva-4vDaVf-6E6LFF-2Gf4z-9pikoR-bpPx1D-a1m9eB-4JAH3z-8XZ3gD-76oZKx-a1meki-6pt9WN-47Uyn-79nqaN-6pt8M9-zwmab-8vusH4-RdAKH-5viFJ9-6poZi6-6pte5J-21id5gS-wx5ep-DLpVMj-uhWoxT-LtCN2g-ahJJ49-DjTbfN-66a28U-8Eut2s-jEFPOJ-buWv9D-8tJtdS-bh6zXt-ahJKdC>),

2005, photograph

MNIST contains 70,000 images of handwritten digits: 60,000 for training and 10,000 for testing. The images are grayscale, 28x28 pixels, and centered to reduce preprocessing and get started quicker.

## Setting up the Environment

We will be using **PyTorch** (<https://pytorch.org/>) to train a convolutional neural network to recognize MNIST's handwritten digits in this article. PyTorch is a very popular framework for deep learning like **Tensorflow** ([tensorflow.org](https://tensorflow.org)), **CNTK** (<https://docs.microsoft.com/en-us/cognitive-toolkit/>) and **Caffe2** (<https://caffe2.ai/>). But

unlike these other frameworks PyTorch has dynamic execution graphs, meaning the computation graph is created on the fly.

Since there's already a PyTorch environment from another article, we can just transclude it and use it here.

```
import torch
import torchvision
```

✓ 1.6s

Imports | PyTorch & TorchVision (Python) [PyTorch Install](#)

## Preparing the Dataset

With the imports in place we can go ahead and prepare the data we'll be using. But before that we'll define the hyperparameters we'll be using for the experiment. Here the number of epochs defines how many times we'll loop over the complete training dataset, while `learning_rate` and `momentum` are hyperparameters for the optimizer we'll be using later on.

```
n_epochs = 3
batch_size_train = 64
batch_size_test = 1000
learning_rate = 0.01
momentum = 0.5
log_interval = 10

random_seed = 1
torch.backends.cudnn.enabled = False
torch.manual_seed(random_seed)
```

✓ 0.3s

PyTorch & TorchVision (Python) [PyTorch Install](#)

For repeatable experiments we have to set random seeds for anything using random number generation - this means `numpy` and `random` as well! It's also worth mentioning that cuDNN uses nondeterministic algorithms which can be disabled setting `torch.backends.cudnn.enabled = False`.

Now we'll also need DataLoaders for the dataset. This is where TorchVision comes into play. It let's use load the MNIST dataset in a handy way. We'll use a `batch_size` of 64 for training and size 1000 for testing on this dataset. The values `0.1307` and `0.3081` used for the `Normalize()` transformation below are the global mean and standard deviation of the MNIST dataset, we'll take them as a given here.

TorchVision offers a lot of handy transformations, such as cropping or normalization.

```
train_loader = torch.utils.data.DataLoader(
    torchvision.datasets.MNIST('/files/', train=True, download=True,
                               transform=torchvision.transforms.Compose([
                                   torchvision.transforms.ToTensor(),
                                   torchvision.transforms.Normalize(
                                       (0.1307,), (0.3081,))
                               ])),
    batch_size=batch_size_train, shuffle=True)
```

```
test_loader = torch.utils.data.DataLoader(
    torchvision.datasets.MNIST('/files/', train=False, download=True,
                              transform=torchvision.transforms.Compose([
                                  torchvision.transforms.ToTensor(),
                                  torchvision.transforms.Normalize(
                                      (0.1307,), (0.3081,))
                              ])),
    batch_size=batch_size_test, shuffle=True)
```

✓ 3.3s

Creating DataLoader | PyTorch & TorchVision (Python) [PyTorch Install](#)

```
Downloading http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.gz
Processing...
Done!
```

PyTorch's `DataLoader` contain a few interesting options other than the dataset and batch size. For example we could use `num_workers > 1` to use subprocesses to asynchronously load data or using pinned RAM (via `pin_memory`) to speed up RAM to GPU transfers. But since these mostly matter when we're using a GPU we can omit them here.

Now let's take a look at some examples. We'll use the `test_loader` for this.

```
examples = enumerate(test_loader)
batch_idx, (example_data, example_targets) = next(examples)
```

✓ 0.5s

PyTorch & TorchVision (Python) [PyTorch Install](#)

Ok let's see what one test data batch consists of.

```
example_data.shape
```

✓ 0.2s

PyTorch & TorchVision (Python) [PyTorch Install](#)

So one test data batch is a tensor of shape: `torch.Size([1000, 1, 28, 28])`. This means we have 1000 examples of 28x28 pixels in grayscale (i.e. no rgb channels, hence the one). We can plot some of them using `matplotlib`.

```
<torch._C.Generator object at 0x7f174b129470>
```

```
import matplotlib.pyplot as plt

fig = plt.figure()
for i in range(6):
    plt.subplot(2,3,i+1)
    plt.tight_layout()
    plt.imshow(example_data[i][0], cmap='gray', interpolation='none')
    plt.title("Ground Truth: {}".format(example_targets[i]))
    plt.xticks([])
    plt.yticks([])
fig
```

✓ 1.2s

PyTorch & TorchVision (Python) [PyTorch Install](#)

Alright, those shouldn't be too hard to recognize after some training.

Interested in a new type of notebook?

**Try Nextjournal. The notebook  
for reproducible research.**

**SIGN UP**

Automatically version-controlled all the time  
Supports Python, R, Julia, Clojure and more  
Invite co-workers, collaborate in real-time  
Import your existing Jupyter notebooks

[Learn more about Nextjournal](#)

```
torch.Size([1000, 1, 28, 28])
```

## Building the Network

Now let's go ahead and build our network. We'll use two 2-D convolutional layers followed by two fully-connected (or *linear*) layers. As activation function we'll choose rectified linear units ([https://en.wikipedia.org/wiki/Rectifier\\_\(neural\\_networks\)](https://en.wikipedia.org/wiki/Rectifier_(neural_networks))) (ReLU in short) and as a means of regularization we'll use two dropout layers. In PyTorch a nice way to build a network is by creating a new class for the network we wish to build. Let's import a few submodules here for more readable code.

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

✓ 0.2s

PyTorch & TorchVision (Python) [PyTorch Install](#)

```
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(1, 10, kernel_size=5)
        self.conv2 = nn.Conv2d(10, 20, kernel_size=5)
        self.conv2_drop = nn.Dropout2d()
        self.fc1 = nn.Linear(320, 50)
        self.fc2 = nn.Linear(50, 10)

    def forward(self, x):
        x = F.relu(F.max_pool2d(self.conv1(x), 2))
        x = F.relu(F.max_pool2d(self.conv2_drop(self.conv2(x)), 2))
        x = x.view(-1, 320)
        x = F.relu(self.fc1(x))
        x = F.dropout(x, training=self.training)
        x = self.fc2(x)
        return F.log_softmax(x)
```

✓ 0.2s

Creating the Network

PyTorch & TorchVision (Python) [PyTorch Install](#)

Broadly speaking we can think of the `torch.nn` layers as which contain trainable parameters while `torch.nn.functional` are purely functional. The `forward()` pass defines the way we compute our output using the given layers and functions. It would be perfectly fine to print out tensors somewhere in the forward pass for easier debugging. This comes in handy when experimenting with more complex models. Note that the forward pass could make use of e.g. a member variable or even the data itself to determine the execution path - and it can also make use of multiple arguments!

Now let's initialize the network and the optimizer.

```
network = Net()
optimizer = optim.SGD(network.parameters(), lr=learning_rate,
```

momentum=momentum)

✓ 0.2s

Network & Optimizer Setup | PyTorch & TorchVision (Python) [PyTorch Install](#)

Note: If we were using a GPU for training, we should have also sent the network parameters to the GPU using e.g. `network.cuda()`. It is important to transfer the network's parameters to the appropriate device before passing them to the optimizer, otherwise the optimizer will not be able to keep track of them in the right way.

## Training the Model

Time to build our training loop. First we want to make sure our network is in training mode. Then we iterate over all training data once per epoch. Loading the individual batches is handled by the `DataLoader`. First we need to manually set the gradients to zero using `optimizer.zero_grad()` since PyTorch by default accumulates gradients. We then produce the output of our network (forward pass) and compute a negative log-likelihood loss between the output and the ground truth label. The `backward()` call we now collect a new set of gradients which we propagate back into each of the network's parameters using `optimizer.step()`. For more detailed information about the inner workings of PyTorch's automatic gradient system, see [the official docs for autograd](https://pytorch.org/docs/stable/notes/autograd.html#) (<https://pytorch.org/docs/stable/notes/autograd.html#>) (highly recommended).

We'll also keep track of the progress with some printouts. In order to create a nice training curve later on we also create two lists for saving training and testing losses. On the x-axis we want to display the number of training examples the network has seen during training.

```
train_losses = []
train_counter = []
test_losses = []
test_counter = [i*len(train_loader.dataset) for i in range(n_epochs + 1)]
```

✓ 0.2s

PyTorch & TorchVision (Python) [PyTorch Install](#)

We'll run our test loop once before even starting the training to see what accuracy/loss we achieve just with randomly initialized network parameters. Can you guess what our accuracy might look like for this case?

```
def train(epoch):
    network.train()
    for batch_idx, (data, target) in enumerate(train_loader):
        optimizer.zero_grad()
        output = network(data)
        loss = F.nll_loss(output, target)
        loss.backward()
        optimizer.step()
        if batch_idx % log_interval == 0:
            print('Train Epoch: {} [{}/{} ({:.0f}%)]\tLoss: {:.6f}'.format(
                epoch, batch_idx * len(data), len(train_loader.dataset),
                100. * batch_idx / len(train_loader), loss.item()))
            train_losses.append(loss.item())
            train_counter.append(
                (batch_idx*64) + ((epoch-1)*len(train_loader.dataset)))
            torch.save(network.state_dict(), '/results/model.pth')
            torch.save(optimizer.state_dict(), '/results/optimizer.pth')
```

✓ 0.2s

Train Loop | PyTorch & TorchVision (Python) [PyTorch Install](#)

Neural network modules as well as optimizers have the ability to save and load their internal state using `.state_dict()`. With this we can continue training from previously saved state dicts if needed - we'd just need to call `.load_state_dict(state_dict)`.

Now for our test loop. Here we sum up the test loss and keep track of correctly classified digits to compute the accuracy of the network.

```
def test():
    network.eval()
    test_loss = 0
    correct = 0
    with torch.no_grad():
        for data, target in test_loader:
            output = network(data)
            test_loss += F.nll_loss(output, target, size_average=False).item()
            pred = output.data.max(1, keepdim=True)[1]
            correct += pred.eq(target.data.view_as(pred)).sum()
    test_loss /= len(test_loader.dataset)
    test_losses.append(test_loss)
    print('\nTest set: Avg. loss: {:.4f}, Accuracy: {}/{} ({:.0f}%)\n'.format(
        test_loss, correct, len(test_loader.dataset),
        100. * correct / len(test_loader.dataset)))
```

✓ 0.2s

Test Loop | PyTorch & TorchVision (Python) [PyTorch Install](#)

Using the context manager `no_grad()` we can avoid storing the computations done producing the output of our network in the computation graph.

Time to run the training! We'll manually add a `test()` call before we loop over `n_epochs` to evaluate our model with randomly initialized parameters.

```
test()
for epoch in range(1, n_epochs + 1):
    train(epoch)
    test()
```

✓ 105.2s

Training Loop | PyTorch & TorchVision (Python) [PyTorch Install](#)

```
Train Epoch: 3 [51200/60000 (85%)] Loss: 0.285094
Train Epoch: 3 [51840/60000 (86%)] Loss: 0.223256
Train Epoch: 3 [52480/60000 (87%)] Loss: 0.220552
Train Epoch: 3 [53120/60000 (88%)] Loss: 0.235116
Train Epoch: 3 [53760/60000 (90%)] Loss: 0.416173
Train Epoch: 3 [54400/60000 (91%)] Loss: 0.427750
Train Epoch: 3 [55040/60000 (92%)] Loss: 0.286725
Train Epoch: 3 [55680/60000 (93%)] Loss: 0.256722
Train Epoch: 3 [56320/60000 (94%)] Loss: 0.313162
Train Epoch: 3 [56960/60000 (95%)] Loss: 0.325091
Train Epoch: 3 [57600/60000 (96%)] Loss: 0.184333
Train Epoch: 3 [58240/60000 (97%)] Loss: 0.098307
```



## Evaluating the Model's Performance

And that's it. With just 3 epochs of training we already managed to achieve 97% accuracy on the test set! We started out with randomly initialized parameters and as expected only got about 10% accuracy on the test set before starting the training.

Let's plot our training curve.

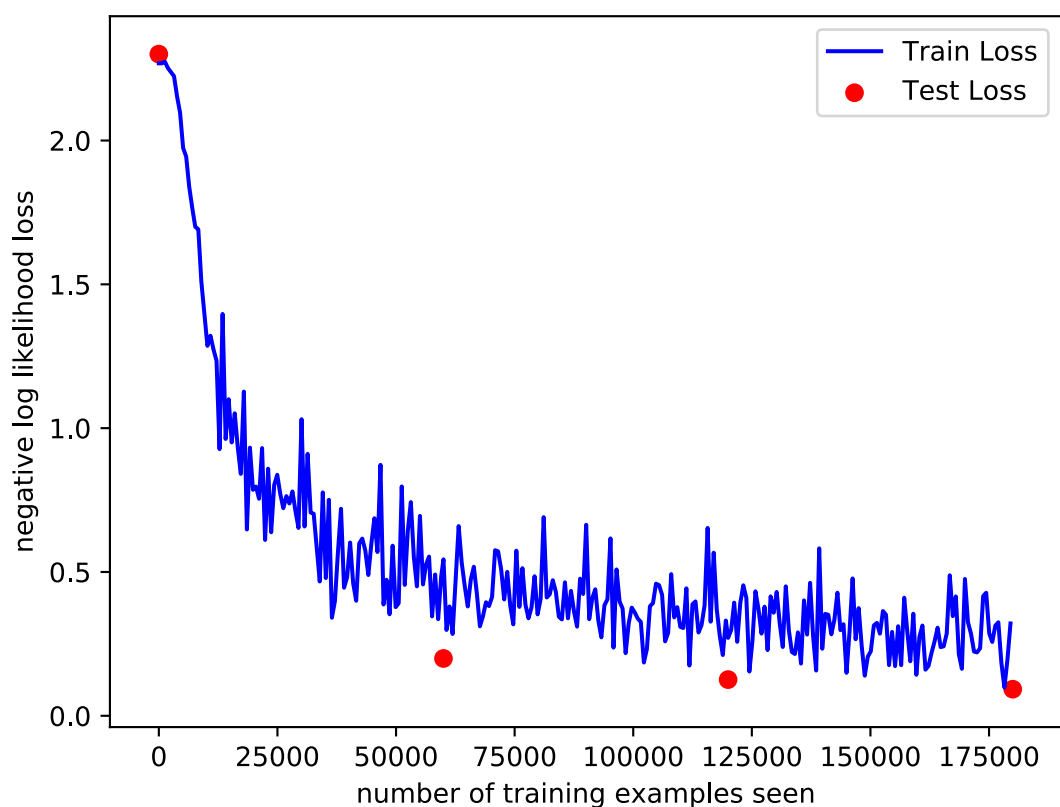
```
fig = plt.figure()
plt.plot(train_counter, train_losses, color='blue')
plt.scatter(test_counter, test_losses, color='red')
plt.legend(['Train Loss', 'Test Loss'], loc='upper right')
plt.xlabel('number of training examples seen')
plt.ylabel('negative log likelihood loss')
fig
```

✓ 0.8s

PyTorch & TorchVision (Python) [PyTorch Install](#)



Judging from the *training curve* (



like we could even continue training for a few more epochs!

) it looks

But before that let's again look at a few examples as we did earlier and compare the model's output.

```
with torch.no_grad():
    output = network(example_data)
```

✓ 0.4s

PyTorch & TorchVision (Python) [PyTorch Install](#)

```
fig = plt.figure()
for i in range(6):
    plt.subplot(2,3,i+1)
    plt.tight_layout()
    plt.imshow(example_data[i][0], cmap='gray', interpolation='none')
    plt.title("Prediction: {}".format(
        output.data.max(1, keepdim=True)[1][i].item()))
    plt.xticks([])
    plt.yticks([])
fig
```

✓ 1.0s

PyTorch & TorchVision (Python) [PyTorch Install](#)

 optimizer.pth
88.68 KB 
 model.pth
88.64 KB 

Our model's predictions seem to be on point for those examples!

## Continued Training from Checkpoints

Now let's continue training the network, or rather see how we can continue training from the state\_dicts we saved during our first training run. We'll initialize a new set of network and optimizers.

```
continued_network = Net()
continued_optimizer = optim.SGD(network.parameters(), lr=learning_rate,
                                momentum=momentum)
```

✓ 0.2s

PyTorch & TorchVision (Python) [PyTorch Install](#)

Using `.load_state_dict()` we can now load the internal state of the network and optimizer when we last saved them.

```
network_state_dict = torch.load(model.pth)
continued_network.load_state_dict(network_state_dict)

optimizer_state_dict = torch.load(optimizer.pth)
continued_optimizer.load_state_dict(optimizer_state_dict)
```

✓ 0.3s

PyTorch & TorchVision (Python) [PyTorch Install](#)

Again running a training loop should immediately pick up the training where we left it. To check on that let's simply use the same `10` as before to keep track of the loss. Due to the way we constructed the test counter for the number of training examples seen we manually have to update it here.

```
for i in range(4,9):
```

```
test_counter.append(i*len(train_loader.dataset))
train(i)
test()
```

✓ 163.9s

PyTorch & torchvision (Python) [PyTorch Install](#)

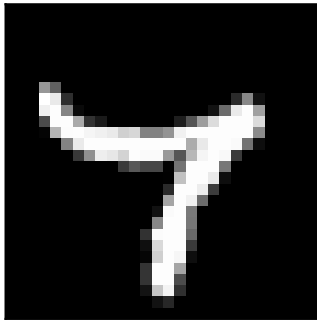
```
Train Epoch: 8 [47720/60000 (80%)] Loss: 0.188427
Train Epoch: 8 [50560/60000 (84%)] Loss: 0.238012
Train Epoch: 8 [51200/60000 (85%)] Loss: 0.363770
Train Epoch: 8 [51840/60000 (86%)] Loss: 0.099876
Train Epoch: 8 [52480/60000 (87%)] Loss: 0.161177
Train Epoch: 8 [53120/60000 (88%)] Loss: 0.085485
Train Epoch: 8 [53760/60000 (90%)] Loss: 0.282653
Train Epoch: 8 [54400/60000 (91%)] Loss: 0.202441
Train Epoch: 8 [55040/60000 (92%)] Loss: 0.120490
Train Epoch: 8 [55680/60000 (93%)] Loss: 0.206396
Train Epoch: 8 [56320/60000 (94%)] Loss: 0.193615
Train Epoch: 8 [56960/60000 (95%)] Loss: 0.331969
Train Epoch: 8 [57600/60000 (96%)] Loss: 0.283257
Train Epoch: 8 [58240/60000 (97%)] Loss: 0.195820
```

Great! We again see a (much slower) increase in test set accuracy from epoch to epoch. Let's visualize this to further inspect the training progress.

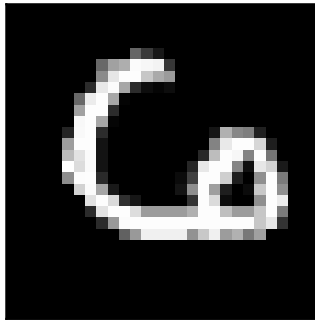
```
fig = plt.figure()
plt.plot(train_counter, train_losses, color='blue')
plt.scatter(test_counter, test_losses, color='red')
plt.legend(['Train Loss', 'Test Loss'], loc='upper right')
plt.xlabel('number of training examples seen')
plt.ylabel('negative log likelihood loss')
fig
```

✓ 0.8s

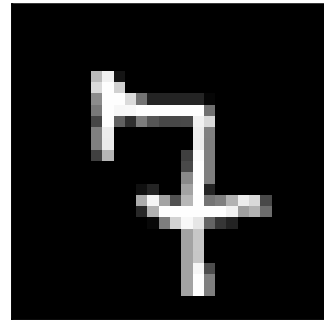
PyTorch & torchvision (Python) [PyTorch Install](#)



Prediction: 5



Prediction: 6



Prediction: 7

That still looks like a rather smooth learning curve just as if we initially would've trained for 8 epochs! Remember that we just appended values to the same lists starting from the 5th red dot onward.

From this we can conclude two things:

1. Continuing from the checkpointed internal state worked as intended.
2. We still don't seem to run into overfitting issues! It looks like our dropout layers did a good job regularizing the model.

In summary we built a new environment with PyTorch and TorchVision, used it to classify handwritten digits from the MNIST dataset and hopefully developed a good intuition using PyTorch. For further information the official [PyTorch documentation \(https://pytorch.org/docs/stable/index.html\)](https://pytorch.org/docs/stable/index.html) is really nicely written and the [forums \(https://discuss.pytorch.org/\)](https://discuss.pytorch.org/) are also quite active!

Interested in a new type of notebook?

**Try Nextjournal. The notebook  
for reproducible research.**


**SIGN UP**

Automatically version-controlled all the time  
Supports Python, R, Julia, Clojure and more  
Invite co-workers, collaborate in real-time  
Import your existing Jupyter notebooks

[➤ Learn more about Nextjournal](#)

 optimizer.pth

88.68 KB 

 model.pth

88.64 KB 