Deep Learning

Exercise 6: Convolutional Networks

Room: **BIN-1-B.01**

Instructor: Manuel Günther
Email: guenther@ifi.uzh.ch
Office: AND 2.54

Friday, April 1, 2022

Outline

1. PyTorch

2. MNIST Training with PyTorch

Outline

1. PyTorch
   - Types of Layers
   - Datasets and Batches
   - The Training Loop
   - Running on the GPU

Types of Layers

Learnable Layers

- Fully connected layer `torch.nn.Linear`
  - $\rightarrow$ `in_features` $= D$
  - $\rightarrow$ `out_features` $= K$
  - $\rightarrow$ `bias = True`
- Convolutional layer `torch.nn.Conv2d`
  - $\rightarrow$ `in_channels` $= C$
  - $\rightarrow$ `out_channels` $= Q$
  - $\rightarrow$ `kernel_size` $= U$ or (U, V)
  - $\rightarrow$ `padding`, `stride`, `bias`

Non-Learnable Layers

- Activation functions: `torch.nn.Sigmoid`, `torch.nn.Tanh`, `torch.nn.Softmax` ...
- Pooling: `torch.nn.MaxPool2d`, `torch.nn.AvgPool2d`
  - $\rightarrow$ `kernel_size`, `stride`
- Input flattening: `torch.nn.Flatten`

Datasets and Batches

Datasets

- Many available in `torchvision.datasets`
  - → torchvision.datasets.MNIST
  - → torchvision.datasets.ImageNet
- Common interface:
  - → `root`: Directory of raw data
  - → `train`: set to `False` for test set
  - → `download`: downloads data if required
  - → `transform`: preprocessing
- Return `PIL` images

Transforms

- Prepares the input
  - → Depends on original data
- Implemented in `torchvision.transforms`
  - → `Resize((D, E))`: height, width
  - → `Normalize(mean, std)`
  - → `Lambda(callable)`: generic
  - → `ToTensor`: `PIL` → `tensor`
- Combining several transforms:
  - → `Compose((trans1,trans2))`
- Additionally: target transforms

Datasets and Batches

Data Loader

- Prepares batches of data
  - → For both input and target
- `torch.utils.data.DataLoader`
  - → `dataset`: see above
  - → `batch_size` $= B$
  - → `shuffle` after each epoch
  - → `num_workers`: parallel execution on the CPU (might slow down processing)

Example MNIST Dataset

```
# obtain datasets
transform = torchvision.transforms.ToTensor()
train_set = torchvision.datasets.MNIST(
              root="/temp/MNIST",
              train=True, download=True,
              transform=transform
            )
test_set = torchvision.datasets.MNIST(
              root="/temp/MNIST",
              train=False, download=True,
              transform=transform
            )
# loaders
train_loader = torch.utils.data.DataLoader(
    train_set, shuffle=True, batch_size=64
)
test_loader = torch.utils.data.DataLoader(
    test_set, shuffle=False, batch_size=100
)
```

The Training Loop

Example Training Loop

```
# training loop
for epoch in range(epochs):
  for x,t in train_loader:
    # DO NOT FORGET:
    optimizer.zero_grad()
    z = network(x)
    J = loss(z, t)
    J.backward()
    optimizer.step()

    # compute train accuracy
    ...
```

Example Test Loop

```
# testing loop
  with torch.no_grad():
    correct = 0
    for x,t in test_loader:
      z = network(x)
      # optional: compute test loss
      J = loss(z, t)

      # compute test accuracy
      correct += torch.sum(
        torch.argmax(z, dim=1) == t
      ).item()
    acc = correct/ len(test_set)
```

Running on the GPU

Preparation

- Test CUDA availability:
  `torch.cuda.is_available()`
- Select device `"cpu"` or `"cuda"`:
  `device = torch.device("cuda")`
- Move everything to the device:
  `network.to(device)`
  `x.to(device)`
  `t.to(device)`

Speed Warning

Can be slow on CPU (1 min per epoch)

Example Training Loop

```
# instantiate everything
device = torch.device("cuda")
network = Network().to(device)
loss = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(
  params=network.parameters(),
  lr=0.01, momentum=0.9
)

# training epoch
for epoch in range(epochs):
  for x,t in train_loader:
    optimizer.zero_grad()
    z = network(x.to(device)
    J = loss(z, t.to(device))
    J.backward()
    optimizer.step()
```

Running on the GPU

Enabling GPU Processing on Google Colaboratory

- Google Colaboratory allows usage of Google GPU servers
- Open a notebook on Google Colaboratory
- Select Runtime → Change Runtime Type
- Select Acceleration GPU (does not work for TPU)
- Check that GPU is enabled:

  → `torch.cuda.is_available()` should return `True`

Outline

2 MNIST Training with PyTorch
- Dataset and Data Loader
- Networks
- Network Training

MNIST Training with PyTorch

Goal of Exercise

- Compare fully-connected and convolutional networks
- Get to know PyTorch `Dataset` and `DataLoader` classes
- Familiarize with PyTorch training procedure with batches

Task 1: Dataset

- Implement a function to return two datasets with a given `transform`
- Instantiate MNIST dataset from `torchvision.datasets`
  → Set dataset to automatically download
- Provide both training and test set

Datasets and Data Loaders

Test 1: Data Types

- Instantiate datasets with `transform=None`
- Check that the dataset returns `PIL.Image.Image`'s

Task 2: Data Loaders

- Load datasets with `torchvision.transforms.ToTensor` transform
- Create two data loaders, one for each dataset
  - → Choose training batch size to be $B = 64$ (B for test on your choice)
  - → Decide which of the two data loaders require shuffling

Test 2: Batches

- Check data types and ranges of batches: input and target
- Check that batch size equals B for all except the last batch

Networks

Compared Networks

- Fully-connected and convolutional network
  $\rightarrow$ Same number of layers with weights

Fully-connected Network

1. Flatten layer to convert 28×28 input into $28 * 28$ vector
2. $K \times D$ fully-connected layer
3. $\tanh$ activation
4. $K \times K$ fully-connected layer
5. $\tanh$ activation
6. $K \times O$ fully-connected layer

Convolutional Network

1. $Q_1 \times 1 \times 5 \times 5$ convolutional layer, stride 1, padding 2
2. Maximum pooling, 2×2 kernel, stride 2
3. $\tanh$ activation
4. $Q_2 \times Q_1 \times 5 \times 5$ convolutional layer, stride 1, padding 2
5. Maximum pooling, 2×2 kernel, stride 2
6. $\tanh$ activation
7. Flatten layer
8. $O \times ?$ fully-connected layer

Networks

Task 3: Fully-Connected Network

- Implement a function to return the fully-connected network
  → `torch.nn.Sequential` can still be used

Task 4: Convolutions Output (theoretical question)

- Analytically compute input size of fully-connected layer in convolutional network
  → Consider kernel sizes, strides, paddings and poolings

Task 5: Convolutional Network

- Implement a function to return the convolutional network
  → Consider result of task 4
  → `torch.nn.Sequential` can be used, too

Network Training

Task 6: Training and Validation Loop

- `training` function taking `network`, `epochs`, `eta`
- Instantiate categorical cross-entropy `loss`
- Instantiate stochastic gradient decent `optimizer`
- For `epochs` iterate:
  1. Train on all batches of the training set
  2. Compute loss and accuracy on test set

     → Store both in separate lists
- Return both lists of losses and accuracies

Network Training

Task 7: Fully-connected Training

- Instantiate fully-connected network with $K = 100$ and $O = 10$
- Train fully-connected network for 10 epochs with $\eta = 0.01$
  $\rightarrow$ Store lists of losses and accuracies

Task 8: Convolutional Training

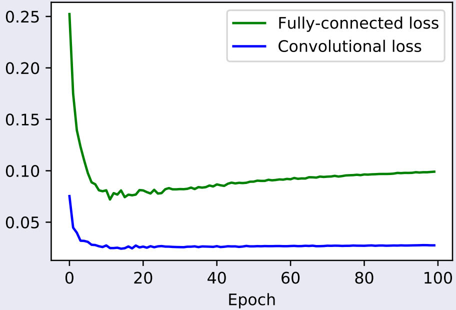
- Create convolutional network with $Q_1 = 32$, $Q_2 = 64$ and $O = 10$
- Train convolutional network for 10 epochs with $\eta = 0.01$
  $\rightarrow$ Store lists of losses and accuracies

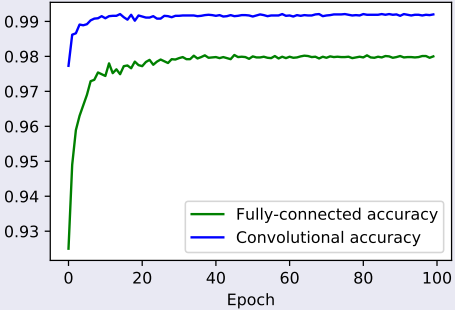
Task 9: Plotting

- Plot losses and accuracies in two separate plots

Network Training

Exemplary Losses and Accuracies (for 100 Epochs)





Task 10: Learnable Parameters

- Compute number of learnable parameters for both networks
  - $\rightarrow$ Both analytically and via PyTorch