Deep Learning

Exercise 6: MNIST and LeNet in PyTorch

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

Friday, April 16, 2020

Outline

Outline

Installation

Cuda Installation

- Install Cuda driver (if you have a Cuda-enabled GPU)
  - → List of supported GPUs: https://developer.nvidia.com/cuda-legacy-gpus
  - → Install driver for your OS https://www.nvidia.com/Download

PyTorch Installation

- Install from `pytorch` channel via Conda:

  ```
  conda install pytorch torchvision cudatoolkit -c pytorch
  ```
- Extensive documentation and tutorials:
  - → Tutorials: https://pytorch.org/tutorials
  - → Reference documentation: https://pytorch.org/docs/stable

Building a Network

Types of 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` $= U \vee (U, V)$
  $\rightarrow$ `padding`, `stride`, `bias`

Related Functions

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

Building a Network

Defining Network Topology

- Derive class from `torch.nn.Module`
- Instantiating layers and functions in constructor `__init__(self)`
  - → Separate instances for each layer
  - → One instance for each function

Executing Network

- Implement `forward(self, x)`
  - → Call your layers sequentially
  - → Return last output
- `backward` not required
  - → Details in next lecture

Example: Non-Linear Regression

```
class Regression (torch.nn.Module):
  def __init__(self, K):
    # call base class constructor
    super(Regression,self).__init__()

    # initialize components
    self.W1 = torch.nn.Linear(1, K, bias=True)
    self.activation = torch.nn.Sigmoid()
    self.w2 = torch.nn.Linear(K, 1)

  def forward(self, x):
    a = self.W1(x)
    h = self.activation(a)
    z = self.w2(h)
    return z

  def forward(self, x):
    return self.w2(self.activation(self.W1(x)))
```

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

- `torch.utils.data.DataLoader`
  - → `dataset`: see above
  - → `batch_size` $= B$
  - → `shuffle` after each epoch
  - → `num_workers`: parallel execution on the CPU

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

Loss Functions

- `torch.nn.MSELoss`: $\mathcal{J}^{L_2}$
- `torch.nn.BCEWithLogitsLoss`: binary $\mathcal{J}^{\mathrm{CE}}$ and σ activation
- `torch.nn.CrossEntropyLoss`: categorical $\mathcal{J}^{\mathrm{CE}}$ and SoftMax

Learning Strategy

- `torch.optim.SGD`
  - $\rightarrow$ `params`: All optimizable weights
  - $\rightarrow$ `lr`: Learning rate η
  - $\rightarrow$ `momentum`: μ

Example Training Loop

```
# instantiate everything
network = Network()
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:
    # DO NOT FORGET:
    optimizer.zero_grad()
    z = network(x)
    J = loss(z, t)
    J.backward()
    optimizer.step()
  # compute test accuracy
```

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()
  # compute validation accuracy
```

Outline

2 MNIST Training with PyTorch

MNIST Training with PyTorch

Task 1

1. Implement a network in PyTorch including:
   - One fully-connected layer with 784 inputs and K hidden neurons
   - A sigmoid activation function
   - One fully-connected layer with K inputs and $O = 10$ logits
2. Create training and test set for the MNIST dataset
   $\rightarrow$ Transform 28×28 image into vector of size 784
3. Choose optimizer SGD with nicely-working parameters
4. Train the network using categorical cross-entropy and SoftMax
5. Compute and print test set accuracy after each epoch
6. Train the network for 100 epochs and get highest test accuracy

MNIST Training with PyTorch

Task 2

1. Improve fully-connected network with convolutions
   1. Add one or more convolutional layers in the beginning
      → Select kernel parameters U, V, stride, padding
   2. Add a pooling layer after each convolution layer
      → Select type of pooling and its parameters
   3. Have one fully-connected layer with $K = ???$ inputs and $O = 10$ logits
2. Create a training and test set for the MNIST dataset
   → What kinds of transforms do you need here?
3. Choose optimizer SGD with nicely-working parameters
4. Train the network using categorical cross-entropy and SoftMax
5. Compare best test set accuracy to fully-connected model