ECE 285 – MLIP – Assignment 2
CNNs and PyTorch

Compiled by Sneha Gupta, Shobhit Trehan and Charles Deledalle from PyTorch tutorials.
Last updated: October 1, 2019.

As for Assignment 1, we focus on classification on the MNIST Dataset but using deep Convolutional Neural Networks (CNNs) and PyTorch (instead of shallow ANNs and manual implementation).

1 Convolutional Neural Networks (CNNs)?

CNNs are specific artificial neural networks composed of *convolutional* layers, *maxpooling* operations, and *fully connected layers*.

- Convolutional layers are like typical layers except the weight matrix has a specific structure that is relevant for signals and images. They take as input N images and produce as output C images (called *feature maps* or *channels*). They are parameterized by a collection of coefficients that defines a *filter bank*. Each filter performs a weighted average of its inputs within local sliding windows of size $K \times K$ (pixels) where K is a hyperparameter (a small odd number: 3, 5, 7, 9). As for classical layers, each feature map is next processed by an activation function such as `ReLU`.
- Maxpooling operations reduce the dimensions of each feature map by picking the maximum value within local but non-overlapping sliding windows of size $L \times L$ (pixels) where L is another hyper-parameter (usually 2). Maxpooling does not introduce new parameters to be learned.
- Fully connected layers are standard layers where the weight matrix does not have a specific structure: each of the N output units is connected to each of the M input units.

This was a very short description but CNNs will be studied more in depth during the class (Chapter 4).

2 PyTorch?

PyTorch is a Python based scientific computing package targeted at two sets of audiences:

- a replacement for NumPy to use the power of GPUs,
- a deep learning research platform that provides maximum flexibility and speed.

For more reading please visit http://pytorch.org/tutorials/beginner/deep_learning_60min_blitz.html

3 Getting started

First of all, connect to DSMLP (`dsmlp-login.ucsd.edu` server) and start a *pod* with enabled GPU/CUDA capabilities

```
$ launch-scipy-ml-gpu.sh
```

Next connect to your Jupyter Notebook from your web browser (refer to Assignment 0 for more details). Create a new notebook `assignment2.ipynb` and import

```
import numpy as np
import torch
```

For the following questions, please write your code and answers directly in your notebook. Organize your notebook with headings, markdown and code cells (following the numbering of the questions).

4 Tensors

Torch tensors are similar to NumPy ndarrays, with the addition being that they can also be used on a GPU to accelerate computing. The documentation for tensors is available here http://pytorch.org/docs/stable/tensors.html

1. Construct a 5×3 tensor and print it with the following commands

   ```
   x = torch.Tensor(5, 3)
   print(x)
   ```

   How was `x` initialized? What is the type of `x`?

2. Construct a randomly initialized matrix:

   ```
   y = torch.rand(5, 3)
   print(y)
   ```

   How are the random values distributed? What is the type of `y`? What if we use instead `y = torch.randn(5, 3)`?

3. Run the following

   ```
   x = x.double()
   y = y.double()
   print(x)
   print(y)
   ```

   What is the type displayed when you print `x` and `y`?

4. We can also directly initialize tensors with prescribed values. Create the following two tensors:

   ```
   x = torch.Tensor([[-0.1859,  1.3970,  0.5236],
                     [ 2.3854,  0.0707,  2.1970],
                     [-0.3587,  1.2359,  1.8951],
                     [-0.1189, -0.1376,  0.4647],
                     [-1.8968,  2.0164,  0.1092]])
   y = torch.Tensor([[ 0.4838,  0.5822,  0.2755],
                     [ 1.0982,  0.4932, -0.6680],
                     [ 0.7915,  0.6580, -0.5819],
                     [ 0.3825, -1.1822,  1.5217],
                     [ 0.6042, -0.2280,  1.3210]])
   ```

   What are their shapes?

5. You can stack the two 2d tensors in a 3d tensor as

```
z = torch.stack((x, y))
```

What is the shape of `z`?
How does it compare to `torch.cat((x, y), 0)` and `torch.cat((x, y), 1)`?

6. Report the value of the element at the 5th row and 3rd column in the 2d tensor `y`, and try accessing the same element in the 3d tensor `z`.

   Hint: You can use standard NumPy-like indexing on all tensors.

7. Similarly print all the elements corresponding to the 5th row and 3rd column in `z`. How many elements are there?

8. Adding two tensors is fairly straightforward. There are multiple syntaxes for this operation. Sum `x` and `y` (from question 4) as follows

```
print(x + y)
print(torch.add(x, y))
print(x.add(y))
torch.add(x, y, out=x)
print(x)
```

Check whether the above instructions are printing the same output? Are they equivalent?

9. To reshape a tensor, you can use `torch.view`:

```
x = torch.randn(4, 4)
y = x.view(16)
z = x.view(-1, 8)
print(x.size(), y.size(), z.size())
```

Interpret the effect of each of these instructions. What does the `-1` mean?

10. Generate 2 random tensors `x` and `y` of dimensions 10×10 and 2×100, respectively, resize them such that the instruction `torch.mm(x, y)` performs a row vector by matrix multiplication resulting in a row vector of dimensions 1×2.

Go through the documentation and get familiar with other operations as well.

5 NumPy and PyTorch

Converting a Torch tensor to a NumPy array and vice versa is a breeze.

11. Run the below code snippet and report your observation

```
a = torch.ones(5)
print(a)
b = a.numpy()
print(b)
```

What are the types of `a` and `b`?

12. Let us add one to the first element of the tensor `a` from the previous question and report the new values of `a` and `b`.

```
a[0] += 1
print(a)
print(b)
```

Do they match? Do they share their underlying memory locations?

13. Compare the effect of each of these three instructions separately:

```
a.add_(1)
a[:] += 1
a = a.add(1)
```

For each of them, report the new values of `a` and `b`. What are the similarities and differences?

14. Let us study the reverse operation, converting NumPy arrays to a Torch tensor. Run the following code and examine the output.

```
a = np.ones(5)
b = torch.from_numpy(a)
np.add(a, 1, out=a)
print(a)
print(b)
```

All the Torch tensors on the CPU (except a `CharTensor`) support converting to NumPy and back.

15. Torch tensors can be moved onto GPU device's memory using `.to('cuda')` or `.cuda()` and back to CPU device with `.to('cpu')` or `.cpu()`. Alternatively, a tensor can be directly allocated into the GPU using the `device` optional argument. Run the following script

```
device = 'cuda' if torch.cuda.is_available() else 'cpu'
print(device)
x = torch.randn(5, 3).to(device)
y = torch.randn(5, 3, device=device)
z = x + y
```

Interpret each of these instructions. Compare the two allocation instructions for `x` and `y`, which one is the most efficient?

16. Run and interpret the result of the following two instructions

```
print(z.cpu().numpy())
print(z.numpy())
```

Note that moving data from CPU to GPU is slow, so try to do all the computations in GPU, and transfer the data back to CPU once all computations have been performed. Make sure your program produces the same result whether it is run on an architecture with GPU capabilities or not (you can start a pod without GPU capabilities by running `launch-scipy-ml.sh` instead). Note also that newer versions of PyTorch come with more advanced tools for manipulating GPU/CPU devices.

6 Autograd: automatic differentiation

A central feature in PyTorch is the `autograd` package. It provides tools performing automatic differentiation for all operations based on Torch tensors. This means that the gradients of such operations will not require to be explicitly programmed. It uses a *define-by-run framework* that computes these gradients dynamically during runtime. This is really useful for backprop algorithm, since autograd will deduce automatically from the succession of forward operations what are the corresponding backward updates.

A Torch tensor stores a flag `requires_grad`. By default, this attribute is set to `False`. If you set it as `True`, autograd starts to track all operations where this Tensor is involved. Any other tensor resulting from operations involving this tensor will have its attribute `requires_grad` set to `True` automatically. When you finish your computation you can call `.backward()` and have the gradients computed automatically for all these tensors. The gradient for these tensors will be accumulated into the `.grad` attribute.

17. Run the following script

```
x = torch.ones(2, 2, requires_grad=True)
print(x)
y = x + 2
print(y)
```

    What is the `requires_grad` attribute of `y`? What are the `grad` attributes of `x` and `y`?

18. Run the following script

```
z = y * y * 3
f = z.mean()
print(z, f)
```

    Inspect the results and write the equation linking `f` with the four entries x_1, x_2, x_3, x_4 of `x`:

$$f = f(x_1, x_2, x_3, x_4)$$

19. Because the variable `f` contains a single scalar, we can now back-propagate the gradient of `f` with respect to `x` into the variable `x` as

```
f.backward() # That's it!
```

    Report what the gradient $\nabla_{\boldsymbol{x}} f(\boldsymbol{x}) = \left(\frac{\partial f(\boldsymbol{x})}{\partial \boldsymbol{x}}\right)^T$ is.

    Hint: read the introduction to this section to recall where the gradient for tensor $\boldsymbol{x}$ *is stored.*

20. Based on question 18, try to figure out mathematically if autograd produces the correct answer by deriving, yourself, the partial derivatives for each x_i:

$$\left(\nabla_{\boldsymbol{x}} f(\boldsymbol{x})\right)_i = \frac{\partial f(x_1, x_2, x_3, x_4)}{\partial x_i}$$

7 MNIST Data preparation

21. Reuse the code from Assignment 1 to load and normalize MNIST testing and training data.

22. Torch expects that the input of a convolutional layer is stored in the following format

    `Batch size` × `Number of input channels` × `Image height` × `Image width`

    The number of input channels in our case is 1 because MNIST is composed of grayscale images. It would have been 3 if the images were in RGB color. In deeper layers, the number of input channels will be the number of input feature maps. Reorganise the tensors `xtrain` and `xtest` accordingly.

    Hint: Reshape them first with shape `(28, 28, 1, 60000)` *and* `(28, 28, 1, 10000)` *respectively and then use* `np.moveaxis`*.*

23. Check that your data are well reorganized by making sure that

    ```
    MNISTtools.show(xtrain[42, 0, :, :])
    ```

    display a digit that indeed corresponds to `ltrain[42]`.

24. Finally wrap all the data into torch Tensor

    ```
    xtrain = torch.from_numpy(xtrain)
    ltrain = torch.from_numpy(ltrain)
    xtest = torch.from_numpy(xtest)
    ltest = torch.from_numpy(ltest)
    ```

8 Convolutional Neural Network (CNN) for MNIST classification

Neural networks can be constructed using the `torch.nn` package, which relies on `autograd` differentiation tools. This package provides an implementation of CNNs (introduced in Section 1) as follows:

- Convolutional layers can be created as `nn.Conv2d(N, C, K)`. For input images of size $W \times H$, the output feature maps have size $[W - K + 1] \times [H - K + 1]$.
- In PyTorch, maxpooling is implemented like any other non-linear function (such as `ReLU` or `softmax`). For input images of size $W \times H$, the output feature maps have size $\lceil W/L \rceil \times \lceil H/L \rceil$.
- A fully connected layer can be created as `nn.Linear(M, N)`.

25. Our LeNet network will be composed successively of

    (a) a convolutional layer (i) connecting the input image to 6 feature maps with 5×5 convolutions ($K = 5$) and followed by ReLU and maxpooling (ii) ($L = 2$),

    (b) a convolutional layer (iii) connecting the 6 input channels to 16 output channels with 5×5 convolutions and followed by ReLU and maxpooling (iv) ($L = 2$),

    (c) a fully-connected layer (v) connecting 16 feature maps to 120 output units and followed by ReLU,

    (d) a fully-connected layer connecting 120 inputs to 84 output units and followed by ReLU,

    (e) a final linear layer connecting 84 inputs to 10 linear outputs (one for each of our digits).

Determine the size of the feature maps after each convolution and maxpooling operation i.e. at points (i)-(iv) processing steps. How many input units does the third layer have ie. input at (v)?

26. Interpret and complete the following code initializing our LeNet network

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

# This is our neural networks class that inherits from nn.Module
class LeNet(nn.Module):

    # Here we define our network structure
    def __init__(self):
        super(LeNet, self).__init__()
        self.conv1 = nn.Conv2d(1, 6, 5)
        self.conv2 = COMPLETE
        self.fc1   = COMPLETE
        self.fc2   = nn.Linear(120, 84)
        self.fc3   = nn.Linear(84, 10)

    # Here we define one forward pass through the network
    def forward(self, x):
        x = F.max_pool2d(F.relu(self.conv1(x)), (2, 2))
        x = COMPLETE
        x = x.view(-1, self.num_flat_features(x))
        x = COMPLETE
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x

    # Determine the number of features in a batch of tensors
    def num_flat_features(self, x ):
        size = x.size()[1:]
        return np.prod(size)

net = LeNet()
print(net)
```

Note that you just have to define the forward function, and the backward function (where gradients are computed) will be automatically defined for you using `autograd`. You can use any of the Torch tensor operations in the forward function. For more details, please refer to https://pytorch.org/docs/stable/nn.html.

27. The learnable parameters of a model are returned by `net.parameters()`. Run the following and interpret the results:

```
for name, param in net.named_parameters():
    print(name, param.size(), param.requires_grad)
```

What are the learnable parameters? Are gradients going to be tracked by `autograd` for all parameters?

28. To run a forward pass of your initial network over your testing set, simply run:

```
with torch.no_grad():
    yinit = net(xtest)
```

which is equivalent to `yinit = net.forward(xtest)`. Note that `with torch.no_grad()` is used to avoid tracking for gradient during testing and then save some computation time (refer to https://pytorch.org/docs/stable/autograd.html#torch.autograd.no_grad). Run the following and interpret the result

```
_, lpred = yinit.max(1)
print(100 * (ltest == lpred).float().mean())
```

29. We will use (Mini-Batch) Stochastic Gradient Descent (SGD) with cross-entropy and momentum. Start implementing the following function:

```
def backprop_deep(xtrain, ltrain, net, T, B=100, gamma=.001, rho=.9):
    N = xtrain.size()[0]     # Training set size
    NB = COMPLETE            # Number of minibatches
    criterion = nn.CrossEntropyLoss()
    optimizer = torch.optim.SGD(net.parameters(), lr=COMPLETE, momentum=rho)
```

where `T` will be the number of epochs, `B` the minibatch size, `gamma` the step size, and `rho` the momentum. For more details, refer to https://pytorch.org/docs/stable/nn.html and https://pytorch.org/docs/stable/optim.html. Note that PyTorch's CrossEntropyLoss is the composition of a softmax activation with the standard cross-entropy loss.

30. We have everything ready now to train our model with SGD and backprop.
Finish implementing `backprop_deep` by completing each part in the piece of code below

```
def backprop_deep(xtrain, ltrain, net, T, B=100, gamma=.001, rho=.9):
    ...
    for epoch in range(T):
        running_loss = 0.0
        shuffled_indices = COMPLETE
        for k in range(NB):
            # Extract k-th minibatch from xtrain and ltrain
            minibatch_indices = COMPLETE
            inputs = xtrain[COMPLETE]
            labels = ltrain[COMPLETE]

            # Initialize the gradients to zero
            optimizer.zero_grad()

            # Forward propagation
            outputs = net(inputs)

            # Error evaluation
            loss = criterion(outputs, labels)
```

```
            # Back propagation
            COMPLETE

            # Parameter update
            optimizer.step()

            # Print averaged loss per minibatch every 100 mini-batches
            # Compute and print statistics
            with torch.no_grad():
                running_loss += loss.item()
            if k % 100 == 99:
                print('[%d, %5d] loss: %.3f' %
                      (epoch + 1, k + 1, running_loss / 100))
                running_loss = 0.0

net = LeNet()
backprop_deep(xtrain, ltrain, net, T=3)
```

Again, refer to https://pytorch.org/docs/stable/nn.html and https://pytorch.org/docs/stable/optim.html for more details. Run the function for 3 epochs, it may take several minutes. The loss per minibatch should decay as $2.30, 2.29, 2.27, 2.24, 2.15, 1.82, 1.07, 0.65, \ldots$ until 0.22 (these are approximative values that slightly vary each time you reinitiate the network). If it does not, you may have a bug, then you can interrupt it by pressing `Esc+i+i`. You may also interrupt it and move to the next question if you are impatient. Note that the function updates the network `net` given in argument, though you interrupt it.

31. Re-evaluate the predictions of your trained network on the testing dataset. By how much did the accuracy improve compared to the initialization?

32. Move the training data to GPU. Reinitialize a new network and transfer it to GPU simply as

```
net = LeNet().to(device)
```

Rerun `backprop_deep` for 10 epochs. The learning should run in less than a minute!

33. Re-evaluate the predictions of your trained network on the testing dataset. By how much did the accuracy improve compared to 3 epochs?