50.039 – Theory and Practice of Deep Learning

Alex

Week 03: Basic NNs –due Week 4 Wednesday 10pm

[The following notes are compiled from various sources such as textbooks, lecture materials, Web resources and are shared for academic purposes only, intended for use by students registered for a specific course. In the interest of brevity, every source is not cited. The compiler of these notes gratefully acknowledges all such sources.]

1 In class task: Fully connected neural net for FashionMNIST

I am not a fan of MNIST because **MNIST misleads in a lot of aspects of deep learning!!!!!!** (to GPU-sized datasets) – see Ian Goodfellow's comments on MNIST such as `https://twitter.com/goodfellow_ian/status/852591106655043584?lang=en`. For now we will use it a similar small dataset it allows training without a GPU.

The main objective is to implement a neural network for FashionMNIST.
The dataset: `https://github.com/zalandoresearch/fashion-mnist`

This dataset has a train/test split. While it is wrong **(WHY IT IS WRONG?)** to decide which model to use on the test split, we leave it to the next lecture to correct this problem by introducing a train/val/test split and stay with the train/test split.
The computational flow is as follows (similar to logistic regression):

1. define dataset class and dataloader class
2. define model
3. define loss
4. define an optimizer
5. initialize model parameters, usually when model gets instantiated

6. loop over epochs. every epoch has a train and a validation phase
7. train phase: loop over minibatches of the training data
   (a) set model to train mode
   (b) set model parameters gradients to zero
   (c) fetch input and ground truth tensors, move them to a device
   (d) compute model prediction
   (e) compute loss
   (f) run `loss.backward()` to compute gradients of the loss function with respect to parameters
   (g) run optimizer (here SGD from `torch.optim`) to apply gradients to update model parameters
8. validation phase: loop over minibatches of the validation data INCORRECTLY, here use validation set equal to the test set
   (a) set model to evaluation mode
   (b) fetch input and ground truth tensors, move them to a device
   (c) in a `with torch.no_grad():` context, compute model prediction
   (d) in the same context compute loss in minibatch
   (e) compute loss averaged/accumulated over all minibatches
   (f) if averaged loss is lower than the best loss so far, save the state dictionary of the model containing the model parameters
9. return best model parameters
10. load best model parameters into your model `https://pytorch.org/tutorials/beginner/saving_loading_models.html`, and compute performance on the test set

How to program this ? What components do we need ? Note: you can google for examples to guide you programming this!

- you can follow and modify a tutorial (easiest) pytorch: `https://github.com/pytorch/examples/blob/master/mnist/main.py` - but in this tutorial you need to add the validation phase within each epoch where you run training – so you need to modify the code somewhat more (as we did with logistic regression).
- you should also go through the class definitions for `torch.nn.dataset` and `torch.nn.Module`, as you will need in week 4 and 5 to modify more in the code

Things that need particular attention:

1. a subclass of `torch.utils.data.Dataset` class.

   Its functionality:

   `thisclass[i]` returns a tuple of pytorch tensors belonging to the i-th datasample.

   FashionMNIST has a dedicated Dataset subclass in pytorch.

   - in pytorch derived from a `torch.utils.data.Dataset` class. Needs to implement two member functions:

     ```
     def __getitem__(self,idx):
     ```

     and

     ```
     def __len__(self):
     ```

     and also

     ```
     def __init__(self,someparams):
     ```

     for reading your dataset.

     ```
     def __len__(self):
     ```

     is called when one uses your dataset instance `yourdataset` with a call like `len(yourdataset)`. It returns the number of samples in your dataset

     ```
     def __getitem__(self,idx):
     ```

     is called when one uses your dataset instance `yourdataset` with a call like `yourdataset[idx]` – it returns the i-th element of the dataset.

   - Here you need a train/val/test split, so you need three dataset class instances - for train, val and test data.

2. a `torch.nnDataloader` class: takes the Dataset as input, and returns a python iterator which produces a minibatch of some batch size every time it is called. Common parameters are batchsize, whether the data should be shuffled (suggested during training), or a sampler class which allows to control how minibatches are created.

3. define a model which takes samples as input, and produces an input. In pytorch it is a class derived from `nn.Module`. A standard neural network module (no matter pytorch or mxnet) needs usually two functions:

   - `def __init__(self,parameter1,parameter2,...):`
     where you define all layers which have model parameters or you define your parameter variables

- `def forward(self,x):`
  where the input x is processed until the output of your network. This implements $x \mapsto f(x)$ where usually $f(x)$ is a sequence of neural network layers applied to x one after the other.

The neural network structure will be:

- l1: fully connected layer with 300 neurons, followed by a relu
- l2: fully connected layer with 100 neurons, followed by a relu
- l3: fully connected layer with 10 neurons

- find out, its easy, what is the name of the layer in `torch.nn` that does it
- Often the input and the intermediate layers have shape $(batchsize, dim_1, dim_2, dim_3, \ldots, dim_n)$. Printing the shape helps: in pytorch `sometensor.size()`.
- for fully connected layers we need to flatten the digit into a 1-dim vector, so we need a shape like $(batchsize, dim_1)$

4. a loss function, used at training phase to measure difference between prediction and ground truth. Possibly, also a loss function at validation phase to measure the quality of your prediction on your validation dataset
5. an optimizer to apply the gradients to model parameters. In this lecture we will use the pytorch SGD optimizer `torch.optim.SGD`,
6. we will try a convolutional network on this thing later on. For this homework in principle you have to do compared to the logistic regression code is to define in the class derived from `nn.Module` the fully connected layers, and use them in the `forward` function

- submit your code
- submit a short report, showing: a graph of training loss over epochs, a graph of validation loss over epochs, report the best accuracy obtained as an average over all classes.
- report for the best accuracy a ranking of classes by their class-wise accuracies. What is the hardest class to predict ?