Lab: PyTorch intro

Deep Learning (SS2020)
3. computer lab (10p)

Announcement

Plan for the next 4 labs:

3 pytorch Intro, train and test MNIST / FashionMNIST

4 Take a pretrained "big" model, finetune it for recognizing a couple of objects

5 Variant a) Deploy Lab4 to your mobile phone
Variant b) Work with data augmentation / dropout / normalization techniques

6 Adversarial patterns and activation visualization on the network form Lab4

Introduction

In this lab we setup a training/testing pipeline in PyTorch for a simple handwritten digit classification problem, which is lightweight and easy to work with. The outline is as follows:

- Get acquainted with PyTorch classes and infrastructure
- Define the training / validation / test split
- Define a neural network model
- Train it
- Evaluate training and test metrics
- Learn how to use tensorboard (somewhat interactive visualization)
- See calibration of the model and performance in recognition with rejection
- Experiment with architecture / other datasets (FashionMNIST / notMNIST)

Getting started

What is PyTorch? Python front end, C++ libraries (A10), Target devices libraries (cuDNN). These will be useful resources for this lab:

- Installing PyTorch: `https://pytorch.org/get-started/locally/`
  Develop and debug locally with CPU/GPU on any system. Choose a CUDA version, important to be able to write generic code.
- PyTorch docs: `https://pytorch.org/docs/stable/index.html`
- Can run large training remotely on servers with GPUs

- Two GPU servers for students at the department:
  `https://cyber.felk.cvut.cz/study/gpu-servers/`.
  In order to use pytorch load the respective module:

```
ml PyTorch/1.4.0-fosscuda-2019b-Python-3.7.4
```

- Google Research Colaboratory (also meant for education): `https://research.google.com/colaboratory/faq.html`
  IPython notebook environment, GPUs available (something like 12h running time limits).
- Tensorboard tools for visualization
  `https://pytorch.org/tutorials/intermediate/tensorboard_tutorial.html`
- Pycharm Remote deployment and debugging
  `https://www.jetbrains.com/help/pycharm/remote-debugging-with-product.html`

Assignments

Assignment 1 (Initial Template)**. (1p)**
We suggest following these steps to learn pytorch:

1. Get acquainted with `torch.Tensor`. It is much alike `numpy.array`, with the possibility to track the computation graph we discussed
2. Prepare the training and testing data. We will use the MNIST dataset, for which a convenient access is available. The following code snippet downloads the training and testing parts of the dataset. Accessing the data as tensors is done with the `Dataset` class. Shuffling, and batching is done with the help of `torch.utils.data.DataLoader`. We will get two for the training and test part. Later on we will split the training data into train and validation subsets.

```
import matplotlib.pyplot as plt
import numpy as np

import torch
import torchvision
import torchvision.transforms as transforms

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

# transforms
transform = transforms.Compose([transforms.ToTensor(), transforms.Normalize((0.5,), (0.5,))])

# datasets
trainset = torchvision.datasets.MNIST('./data', download=True, train=True,
    transform=transform)
testset = torchvision.datasets.MNIST('./data', download=True, train=False,
    transform=transform)

# dataloaders
train_loader = torch.utils.data.DataLoader(trainset, batch_size=128, shuffle=True,
    num_workers=0)
test_loader = torch.utils.data.DataLoader(testset, batch_size=128, shuffle=False,
    num_workers=0)

# lets verify how the loader packs the data
(data, target) = next(iter(train_loader))
# probably get [batch_size x 1 x 28 x 28]
print('Input size:', data.size())
# probably get [batch_size]
print('Labels size:', target.size())
# see number of trainig data:
n_train_data = len(trainset)
print('Train data size:', n_train_data)
```

3. Define a neural network model using the higher level building blocks: `nn.Sequential`, `nn.Linear`, `nn.Conv2d`, `nn.ReLU`. Start with a simple model for ease of debugging. Chose an optimizer, for example `optim.SGD`. Create the training loop. The basic variant, together with a simple network loss and optimizer may look as follows

```
# network, expect input images 28* 28 and 10 classes
net = nn.Sequential(nn.Linear(28 * 28, 10))

# loss function
loss = nn.CrossEntropyLoss(reduction='none')

# optimizer
optimizer = optim.SGD(net.parameters(), lr=0.001, momentum=0.9)

for epoch in range(10):
    # will accumulate total loss over the dataset
    L = 0
    # loop fetching a mini-batch of data at each iteration
    for i, (data, target) in enumerate(train_loader):
        # flatten the data size to [batch_size x 784]
        data_vectors = data.flatten(start_dim=1)
        # apply the network
        y = net.forward(data_vectors)
        # calculate mini-batch losses
        l = loss(y, target)
        # accumulate the total loss as a regular float number (important to sop graph tracking)
        L += l.sum().item()
        # the gradient usually accumulates, need to clear explicitly
        optimizer.zero_grad()
        # compute the gradient from the mini-batch loss
        l.mean().backward()
        # make the optimization step
        optimizer.step()
    print(f'Epoch: {epoch} mean loss: {L / n_train_data}')
```

4. Inspect the code and look up the used classes in pytorch docs. Why the network is so simple, where is softmax?

Assignment 2 (Monitoring)**. (3p)**
Extend the blank above with the following:

1. After completing training with the next 10% of the data in each epoch:
   - Print the stochastic estimate of the training loss from the last batch
   - Print the exponentially weighted average of the batch training losses (lecture 5, slide 13)
   - Compute EWA of the training accuracy
2. After completing full epoch estimate and print the test loss and test accuracy using a similar loop with `test_loader`
3. Stream the metrics above to tensorboard (see tutorial link). Inspect in tensorboard the network graph, training and test metrics.
4. After each epoch also save you trained model using `torch.save(net.state_dict(), PATH)`

Assignment 3 (Refinements)**. (3p)**

1. Split the training set into training and validation sets by creating two loaders that use different disjoint portions of the initial training set. This can be done using `SubsetRandomSampler` and passing that to the loader constructor. Use 10% of the initial training set for validation.
2. (Optional) To estimate the noise level of the losses monitored with EWA, do the following. Compute the variance $\mathbb{V}[X_t]$ of the loss in batch t as a sample variance of the losses for this batch. From these variances compute the variance of the EWA loss $\mathbb{V}[\mu_t]$ (lecture 5 slide 14). Print in the loss estimate in the format (EWA loss)$\pm\sqrt{\mathbb{V}[\mu_t]}$.

3. Refine the net architecture, using convolutions, max-pooling and a fully connected layer in the end.

4. Train to some reasonable validation accuracy (>95%), by monitoring the training and validation metrics and adjusting hyper-parameters as necessary (architecture, batch size, learning rate, etc.). This dataset is really easy, shouldn't take you long.

5. Report plots of training and validation metrics progress (using tensorboard or matplotlib). Given that the batch size or the dataset size may change, it is always a good idea to chose the scale of the axis to represent epochs for the plot. More specifically, rescale the iterations by batch size over the training data size to get (fractional) epochs.

6. Modify the code to automatically run on GPU if it is available. Follow `https://pytorch.org/docs/stable/notes/cuda.html#best-practices`. In the provided template notice the lines:

```
dev = torch.device('cuda') if torch.cuda.is_available() else torch.device('cpu')
...
net.to(dev)
...
data = data.to(dev)
```

   Note the different synatx of moving to device a `Tensor` and a `Model`.

Assignment 4 (Analysis)**. (3p)**
For a well-trained model that you have saved, analyze its performance and predictive probabilities on the test set.

1. Compute the test classification error.

2. How well does the classifier rank?
   For calculations in this assignment use numpy. Pytorch tensors can be converted to numpy arrays using `x.cpu().numpy()`
   Consider that we are allowed to use "reject from recognition" option. If the classifier picks the class $\hat{y}_i = \arg\max_y p(y|x_i;\theta)$ on test point i, let us call $c_i = p(y|x;\theta)$ its confidence. We will want to reject from recognition when we are not confident, i.e. when $c_i \leq \alpha$, where α is a confidence threshold. We will not decide this threshold, but plot the performance for all possible thresholds.

   a) Plot the number of errors as a function of the threshold value α. Since we work with a finite sample of test data, the test error rate will only change when α crosses one of the c_i values we have. So instead of doing very small steps on the threshold and recomputing the error rate each time anew, here's a better way to do it. Sort all the confidences c_i in ascending order. Let $e_i = 1$ if $\hat{y}_i \neq y_i^*$, i.e. we make an error and $e_i = 0$ if $\hat{y}_i = y_i^*$. If $c_(i)$ is the sorted sequence of confidences with error indicators $e_{(i)}$ then we can compute the number of errors with $\alpha = c_i$as as the sum of values $e_{(1)}, \ldots e_{(i)}$. The later sum can be computed with `cumsum()`. Set the range of thresholds from minimum to maximum c_i.

   b) Plot the number of points rejected from recognition as a function of the threshold value α. For this we need to just plot values 1 to n versus the sorted array $c_{(i)}$.

   c) Plot the error rate of accepted points (number of errors versus number of points accepted for recognition). This just combines the data from a) and b). If the relative error declines, the classifier is ranking well (we are rejecting erroneous points and keeping correct ones).

3. Is our classifier not overconfident?
   To investigate this, we can plot the *probability calibration curve* (also known as *reliability diagram*)
   e.g. `https://confluence.ecmwf.int/display/FUG/12.B+Statistical+Concepts+-+Probabilistic+Data` for a simple intro.
   Since we are dealing with rather accurate classifiers, creating uniform bins will not work. In the Plot 1.a) zoom-in the range of confidences from 0.9 to 1. Normalize the number of errors by the total number of points. Add a diagonal line from $(0, 0.1)$ to $(1, 0)$. If the error rate curve is above this line, it means we make lots of errors with high confidence, i.e. the classifier is overconfident.