Statistical Learning Methods

Practical session — January 22, 2024

During this practical session, our final goal is to code and train a neural network able to recognize which digit is present in an image. We are going to use the Pytorch framework introduced in the lectures. It may be easier to use Google colab (`https://colab.research.google.com/`), but if your machine is set up (Pytorch installed, GPU available) you can use it. In that event you may skip the first part.

Setting up colab with GPUs

1. Create a new notebook, click on "Execute", then "modify execution type" and choose GPU if available (T4 should be).
2. Click on "connect" if the notebook is not reconnecting automatically.
3. Check that Pytorch is indeed recognizing the GPU by running the commands `import torch` and `torch.cuda.is_available()`, which should then return `True`.

Exercise I: getting familiar with tensors

1. The basic data structure in Pytorch is a *tensor*. It can be initialized directly from data using `torch.tensor` or from a `numpy` array using `torch.from_numpy()`. Initialize a tensor containing the digits $1, 2, 3, 4$ and shape 2×2 with the two methods.
2. A tensor can also be filled with ones, zeros, or random values. Find the appropriate commands and try them. Which distribution is used to initialize the random tensor?
3. Apart from the shape attribute, a tensor also has a `dtype` and a `device` attribute. What do they correspond to? On which device live the tensors created in 1.?
4. One can move tensors from devices to others with the `.to()` method. Move a tensor to the GPU.
5. Similar to `numpy` arrays, you can access to slices of the tensor, multiply them together, compute the exponential, etc. Try it out!
6. Are matrix operations really faster on a GPU? Let us try and find out. Create two random tensors of size $10^4 \times 10^4$. How much time does it take to multiply them together when they live on the CPU? On the GPU?

Exercise II: A very short introduction to autodiff

1. Tensors have a `requires_grad` attribute. What is its default value? Gradients are stored when equal to `True`.
2. Create a tensor `test` of any shape, keeping track of the gradient. Compute `out`, the sum of the squares of its elements **using only pytorch operations**. The command `out.backward()` computes the gradient with respect to `test`. Check this manually by looking at `test.grad` and computing it by yourself (pen and paper!).
3. Run `out.backward()` a second time. What happens?
4. Transform `out` one more time. Is something happening to `test.grad`? What if we try and run `out.backward()`?

Exercise III: preparing the data

1. A number of datasets are readily available within the Pytorch framework. We are going to use the MNIST dataset.[1] Take a moment to make yourself familiar with the data: what do the initials stand for? What is the shape of the images?

2. Import the `torchvision` module. Download the **train data** in your space using the `torchvision.datasets.MNIST` command.

3. How many examples are there in the train? What is the type of each example? Visualize a few of them. How many classes are they? Check that they match on a few examples.

4. We are not happy with PIL images. Define a transformation as a composition of `ToTensor()` and a flatten (use `Lambda` to do this). Use this transformation in the import using the `transform` option.

5. Split the data into a train and a validation set using `torch.utils.data.random_split` ($\approx 80\%/20\%$ for train / val is a good rule of thumb).

6. Define a batch size and use `torch.utils.data.DataLoader` to define train and val dataloader. There is no absolute rule for batch size except that it should fit in memory!

7. Check that everything went well by fetching a new batch, running

   `train_features, train_labels = next(iter(train_dataloader))`

Exercise IV: creating a neural network

1. In Pytorch, a neural network is coded as a class. The parent class is `torch.nn.Module`. Therefore, our neural network's definition starts as

   `class MyNetwork(nn.Module):`

   We have yet to define the `__init__` method. Use the `super` method to inherit this from the `nn.Module` class.

2. Inside the `__init__` method, we are going to define the structure of the network. Let us start simple with a fully-connected ReLU network with one-hidden layer, **with softmax**. What are the sizes here? Define the layers inside the `__init__` method using `nn.Linear`.

3. Now all that is left to do is to implement the `forward` method. Call sequentially the layers and do not forget the ReLU activation (`nn.ReLU()`). Remember, we have 10 classes, so we want ten outputs.

4. Instantiate the network by calling `mynet = MyNetwork()`. We can get a quick summary using `print`. Check that everything runs smoothly by doing a forward pass on a random input of the correct size.

5. The real strength of a framework such as Pytorch is automatic differentiation. Try it out by calling the `backward` method on the output of a forward pass. As seen in Exercise II, it is important to set the gradient to zero first, by calling `mynet.zero_grad()`.

Exercise V: training the network

1. To train the network, we first need to define a loss function. Initialize the cross entropy loss.

2. We also need to specify which gradient descent algorithm we are going to use. This is called an optimizer in Pytorch. Let us use stochastic gradient descent (`torch.optim.SGD`) as a first try. What is the learning rate? Can you change it?

3. Now we move to the main training loop, iterating over our dataloader:

   `for batch, (X, y) in enumerate(dataloader)`

   Inside this loop, **in this order**, we must (i) compute the model's prediction on X, (ii) compute the loss with respect to y, (iii) zero out the gradients of our optimizer, (iv) do a backward pass on the loss computation, and (v) take a gradient step.

[1] `https://en.wikipedia.org/wiki/MNIST_database`

4. One run of the previous loop is called an epoch. It is good practice to track the loss values when running the training loop, to know whether it is steadily decreasing or not. Add this functionality.
5. Even better: we should also track the loss (or, even better, the accuracy) on the validation set. Add this functionality.
6. Explore different architectures (number of layers, neurons per layers, activation function, loss function), and different optimizers (learning rates, batch size, algorithm). What is the best accuracy that you can obtain on the validation set?
7. For the best network, compute the accuracy on the test. How do you compare to the state-of-the-art (99.87% as of January 21, 2024)?