Deep Learning
Mini-projects

François Fleuret

https://fleuret.org/dlc/

November 1, 2022

1 Project 1 – Classification, weight sharing, auxiliary losses

The objective of this project is to test different architectures to compare two digits visible in a two-channel image. It aims at showing in particular the impact of weight sharing, and of the use of an auxiliary loss to help the training of the main objective.

It should be implemented with PyTorch only code, in particular without using other external libraries such as scikit-learn or numpy.

1.1 Data

The goal of this project is to implement a deep network such that, given as input a series of $2 \times 14 \times 14$ tensor, corresponding to pairs of 14×14 grayscale images, it predicts for each pair if the first digit is lesser or equal to the second.

The training and test set should be $1,000$ pairs each, and the size of the images allows to run experiments rapidly, even in the VM with a single core and no GPU.

You can generate the data sets to use with the function `generate_pair_sets(N)` defined in the file `dlc_practical_prologue.py`. This function returns six tensors:

Name	**Tensor dimension**	**Type**	Content
`train_input`	$N \times 2 \times 14 \times 14$	`float32`	Images
`train_target`	N	`int64`	Class to predict $\in \{0, 1\}$
`train_classes`	$N \times 2$	`int64`	Classes of the two digits $\in \{0, \dots, 9\}$
`test_input`	$N \times 2 \times 14 \times 14$	`float32`	Images
`test_target`	N	`int64`	Class to predict $\in \{0, 1\}$
`test_classes`	$N \times 2$	`int64`	Classes of the two digits $\in \{0, \dots, 9\}$

1.2 Objective

The goal of the project is to compare different architectures, and assess the performance improvement that can be achieved through weight sharing, or using auxiliary losses. For the latter, the training can in particular take advantage of the availability of the classes of the two digits in each pair, beside the Boolean value truly of interest.

All the experiments should be done with $1,000$ pairs for training and test. A convnet with $\sim 70,000$ parameters can be trained with 25 epochs in the VM in less than 2s and should achieve $\sim 15\%$ error

rate.

Performance estimates provided in your report should be estimated through 10+ rounds for each architecture, where both data and weight initialization are randomized, and you should provide estimates of standard deviations.

2 Project 2 – Mini deep-learning framework

The objective of this project is to design a mini "deep learning framework" using only pytorch's tensor operations and the standard math library, hence in particular **without using autograd or the neural-network modules.**

2.1 Objective

Your framework should import only `torch.empty`, and use no pre-existing neural-network python toolbox. Your code should work with autograd globally off, which can be achieved with

```
torch.set_grad_enabled(False)
```

Your framework must provide the necessary tools to:

- build networks combining fully connected layers, Tanh, and ReLU,
- run the forward and backward passes,
- optimize parameters with SGD for MSE.

You must implement a test executable named `test.py` that imports your framework and

- Generates a training and a test set of $1,000$ points sampled uniformly in $[0,1]^2$, each with a label 0 if outside the disk centered at $(0.5, 0.5)$ of radius $1/\sqrt{2\pi}$, and 1 inside,
- builds a network with two input units, one output unit, three hidden layers of 25 units,
- trains it with MSE, logging the loss,
- computes and prints the final train and the test errors.

2.2 Suggested structure

You are free to come with any new ideas you want, and grading will reward originality. The suggested simple structure is to define a class

```
class Module(object):

    def forward(self, *input):
        raise NotImplementedError

    def backward(self, *gradwrtoutput):
        raise NotImplementedError

    def param(self):
        return []
```

and to implement several modules and losses that inherit from it.

Each such module may have tensor parameters, in which case it should also have for each a similarly sized tensor gradient to accumulate the gradient during the back-pass, and

- `forward` should get for input, and returns, a tensor or a tuple of tensors.
- `backward` should get as input a tensor or a tuple of tensors containing the gradient of the loss with respect to the module's output, accumulate the gradient w.r.t. the parameters, and return a tensor or a tuple of tensors containing the gradient of the loss w.r.t. the module's input.
- `param` should return a list of pairs, each composed of a parameter tensor, and a gradient tensor of same size. This list should be empty for parameterless modules (e.g. ReLU).

Some modules may requires additional methods, and some modules may keep track of information from the forward pass to be used in the backward.

You should implement at least the modules `Linear` (fully connected layer), `ReLU`, `Tanh`, `Sequential` to combine several modules in basic sequential structure, and `LossMSE` to compute the MSE loss.