ETH zürich

High Performance Computing for Science and Engineering I

P. Koumoutsakos
ETH Zentrum, CLT E 13
CH-8092 Zürich

Fall semester 2018

Set 6 - PCA with Neural Networks

Issued: November 2, 2018
Hand in (optional): November 9, 2017 23:59

In this exercise we will develop the basic components of a "deep-learning" library. We will define linear and non-linear layers. how to compute their gradients, and how to update their parameters. As a test-case we will begin by considering a linear auto-associative neural network (NN) used to encode the MNIST dataset. The MNIST dataset (figure 1) is a widely used benchmark dataset of 28 by 28 grayscale pixel images of handwritten digits.

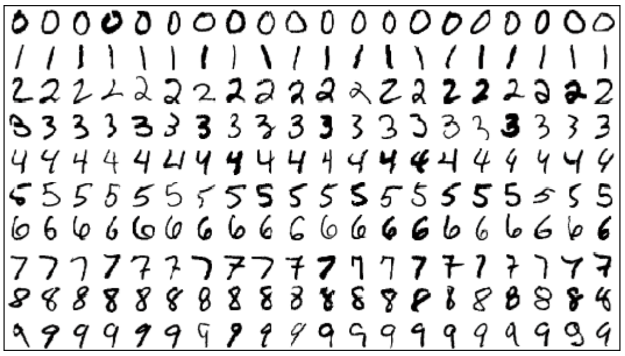
Figure 1: Examples from the MNIST dataset

The auto-associative NN is defined as two linear layers. The first layer maps from the space of the input $\mathbf{y} \in \mathbb{R}^{28 \cdot 28}$ to a latent (compressed) space $\mathbf{z} \in \mathbb{R}^Z$ (here we consider $Z = 10$):

$$\mathbf{z} = \mathbf{x}W^{(1)} + \mathbf{b}^{(1)} \tag{1}$$

here, $W^{(1)} \in \mathbb{R}^{28 \cdot 28} \times \mathbb{R}^Z$ are the weights of the first layer and $\mathbf{b}^{(1)} \in \mathbb{R}^Z$ is its bias. The second maps the compressed representation to an output $\mathbf{y} \in \mathbb{R}^{28 \cdot 28}$ in the same space as the input:

$$\mathbf{y} = \mathbf{z}W^{(2)} + \mathbf{b}^{(2)} \tag{2}$$

here, $W^{(2)} \in \mathbb{R}^Z \times \mathbb{R}^{28 \cdot 28}$ are the weights of the second layer and $\mathbf{b}^{(2)} \in \mathbb{R}^{28 \cdot 28}$ is the bias. The objective of the auto-associative NN is to provide an output that minimizes the squared distance from the input:

$$\mathcal{L} = \mathbb{E}_{\mathbf{x} \sim D}\left[\frac{1}{2}(\mathbf{x} - \mathbf{y})^T(\mathbf{x} - \mathbf{y})\right] \tag{3}$$

Here, with the expectation we denote that we want to minimize the expected loss over the samples contained in the dataset D. By minimizing this loss function, the auto-associative NN finds a compact encoding from which it is possible to reconstruct the input from just the information that flows through $\mathbf{z}$. It can be proven that the weight matrices $W^{(1)}$ and $W^{(2)}$ of the auto-associative NN are linear combinations of the first Z principal components [1].

We wish to update the weights of the NN by stochastic gradient descent (SGD), here W stands for each $W^{(1)}$, $W^{(2)}$, $b^{(1)}$ and $b^{(2)}$:

$$W \leftarrow W - \eta \frac{1}{B} \sum_{i=1}^{B} \mathsf{d}\mathcal{L}(\mathbf{x}^i, W)/\mathsf{d}W \tag{4}$$

Here, η is the learning rate and we use a minibatch of B samples to approximate the expectation over the dataset. The gradient can be computed through chain differentiation of the loss with respect to each network parameter (back-propagation).

With this exercise we provide a skeleton code with the main building blocks of a deep learning library. It contains the main function `main_linear.cpp` which: 1) initializes the linear NN; 2) pepares mini-batches of MNIST samples; 3) Computes the loss function and its gradient with respect to the output of the network for each input $\mathbf{x}$ in the mini-batch:

$$\boldsymbol{\delta}^{(K)} = \left.\frac{d\mathcal{L}}{d\mathbf{y}}\right|_{\mathbf{x}} = (\mathbf{y} - \mathbf{x}) \tag{5}$$

Here, by $\boldsymbol{\delta}^{(K)}$ we denote the gradient of $\mathcal{L}$ w.r.t. the last output of the layer. This gradient is back-propagated to all the parameters of the NN, starting from the last layer ($\boldsymbol{\delta}^{(K)}$ is an input for the back-propagation) to the first. 4) Calls the optimizer to update the parameters by SGD. 5) Once training terminates, computes the NN output by setting to 1 only one unit in the compression layer at the time. The network outputs are then saved to file and can be visualized with `visualize_components.py`, once the skeleton code is filled these should match the PCA components. See the tutorial slides for more information.

Question 1: Linear layer

Let's consider a general linear layer of index k that receives as input an array $H^{(k-1)} \in \mathbb{R}^B \times \mathbb{R}^{N_{\text{inputs}}}$, where N_{inputs} is the size of the output of the previous layer. Notice that we concatenated B row-vector layers $\mathbf{h}^{(k-1)}$ in order to compute mini-batches of outputs through matrix-matrix multiplication. The layer has weight matrix $W^{(k)} \in \mathbb{R}^{N_{\text{inputs}}} \times \mathbb{R}^{N_{\text{outputs}}}$ and bias vector $\mathbf{b}^{(k)} \in \mathbb{R}^1 \times \mathbb{R}^{N_{\text{outputs}}}$. The output of the linear layer is:

$$H^{(k)} = H^{(k-1)} W^{(k)} + J_{B,1}\mathbf{b}^{(k)} \tag{6}$$

Here, $J_{B,1} \in \mathbb{R}^B \times \mathbb{R}^1$ is the matrix with all entries equal to one.

Given the gradient of the loss w.r.t to the output of the linear layer $D^{(k+1)}$ we can compute the gradient w.r.t. $W^{(k)}$: $G_{W^{(k)}}$, $\mathbf{b}^{(k)}$: $G_{\mathbf{b}^{(k)}}$, and $H^{(k-1)}$; $D^{(k-1)}$.

$$G_{W^{(k)}} = (H^{(k-1)})^T D^{(k)} \tag{7}$$

$$G_{\mathbf{b}^{(k)}} = \sum_{b=1}^{B} \boldsymbol{\delta}_b^{(k)} \tag{8}$$

$$D^{(k-1)} = D^{(k)} (W^{(k)})^T \tag{9}$$

Here, with $G_{W^{(k)}}$ (and $G_{\mathbf{b}^{(k)}}$) we denote the unnormalized (not divided by B) gradient of the loss function w.r.t. $W^{(k)}$ (and $\mathbf{b}^{(k)}$) summed over the mini-batch.

a) Implement the forward operation of the linear layer by modifying `LinearLayer::forward` in `Layers.h` of the skeleton code. First, copy the bias vector onto each element of the mini-batch with a `for` loop. Second, use gemm to compute $H^{(k)}$ (note that in the skeleton code the function gemm() with the correct arguments calls the cblas function with the floating point precision of the type `Real`, see `network/Utils.h`).

b) Implement back-propagation for linear layer by modifying `LinearLayer::bckward` of the skeleton code. Use gemm to compute $G_{W^{(k)}}$ and $D^{(k-1)}$, and a `for` loop for $G_{\mathbf{b}^{(k)}}$.

You can test that your code is working correctly with `./exec_testGrad linear`, which checks that the gradient of a parameter computed through finite differences is equal to the same gradient computed by back-propagation:

$$\frac{dh^{(K)}}{dW} = \frac{h^{(K)}\big|_{\mathbf{x},W+\epsilon} - h^{(K)}\big|_{\mathbf{x},W-\epsilon}}{2\epsilon} \tag{10}$$

Question 2: Optimization algorithm

Once the code correctly computes all the gradients, the parameters can be updated by SGD. Instead of the vanilla SGD, we will use an algorithm with momentum to stabilize training:

$$M_W \leftarrow \beta M_W - \eta \frac{1}{B} G_W \tag{11}$$

$$W \leftarrow W + M_W \tag{12}$$

Here, β is the momentum coefficient, which takes into account prior steps to smoothen the noisy updates.

a) Implement the momentum-SGD step in `MomentumSGD::step` in `Optimizer.h`.

b) (OPTIONAL) The optimization algorithm contains a parameter `lambda`. This parameter should govern the L2 penalization which allows the user to modify the cost function:

$$\tilde{\mathcal{L}} = \mathcal{L} + \frac{1}{2}\sum W_i^2 \tag{13}$$

This penalization term keeps the parameters close to 0, which may stabilize training and may reduce overfitting. Extend the momentum-SGD step by adding the L2 penalization gradient to the update.

c) Once the code is behaving correctly and converging (the reported loss function should plateau to about 13). Run `exec_linear` and report the 10 principal components of the MNIST dataset.

Question 3: Non-linearity

The `TanhLayer` follows a linear layer of size `nOutputs` and for each input computes the hyperbolic tangent

$$h^{(k+1)} = \tanh(h^{(k)}) = \frac{\exp\left(2h^{(k)}\right) - 1}{\exp\left(2h^{(k)}\right) + 1} \tag{14}$$

a) Implement the `forward` and `bckward` steps of the `TanhLayer` layer in `Layers.h` of the skeleton code. Make sure it is correct by running `./exec_testGrad tanh`.

b) Once the code runs correctly, run the `exec_nonlinear`. This code is similar to the linear version, but adds multiple layers, uses the non-linearity, and prints out more principal components. In addition to printing each output obtained by setting one entry in the compression layer to 1, `exec_nonlinear` prints the outputs obtained by setting each unit to -1. Note, the file naming is a bit rudimentary and `component_$i.raw` is the component by setting only unit i to 1, and `component_1$i.raw` is the output after setting it to -1. Describe what you see. How is the outcome different from following the same procedure with the previous network?

Question 4: Parallelization

Parallelize with OpenMP threads and motivate your implementation for the following loops:

- In `LinearLayer::forward`, copying/adding the bias vector `param[ID]->biases` onto the layer's output `act[ID]->output`.
- In `LinearLayer::bckward`, the computation of the bias gradient `grad[ID]->biases` (Careful!).
- In `TanhLayer::forward`, computation of the non-linearity.
- In `TanhLayer::bckward`, computation of the gradient of the loss w.r.t. the previous layer `act[ID-1]->dError_dOutput`.
- In `Network::forward`, copying the input mini-batch onto the input layer (line 56 of the skeleton) and copying the output layer onto the return vectors (line 75 of the skeleton).
- In `Network::bckward`, copying the error gradients onto the gradient of the output layer (line 110 of the skeleton).
- The parameter update in `Optimizer::update` and MomentumSGD::step.
- In `main_linear.cpp` processing the MNIST dataset to be placed in the input vector-of-vectors, line 88. (Careful!)
- In `main_linear.cpp` compute the training error and gradient of the NN output, line 97. (Careful!)
- The same steps as the previous two points for the test set, lines 120 and 128.

1 Notes

The MNIST dataset can be downloaded with the script `setup_mnist.sh`. These files should be in the same folder as the executables. Test your work on `euler.ethz.ch` and by loading the environment:

```
$ module load new gcc/6.3.0 openblas/0.2.13_seq
```

and requesting an interactive node:

```
$ bsub -Is -n 1 -W 04:00 bash
```

Request additional processors to test your parallelization with `openblas/0.2.13_par` and:

```
$ bsub -Is -R fullnode -n 24 -W 04:00 bash
```

Once you have acquired a node, test the scaling by varying both `OMP_NUM_THREADS` and `OPENBLAS_NUM_THREADS`.

References

[1] Baldi, Pierre and Hornik, Kurt, Neural networks and principal component analysis: Learning from examples without local minima, Neural networks, 1989.