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.

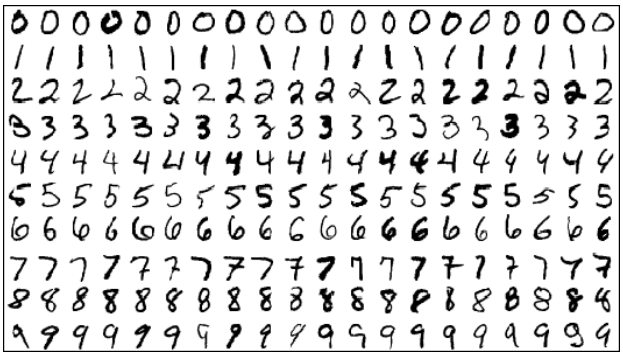

Figure 1: Examples from the MNIST dataset

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 $b^{(2)}$:

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

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

30 points total

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`).

The most common source of doubts for this question was about ordering of the data. The memory space pointed to by `output` holds `batchSize x nOutputs` numbers. We use the convention that it is a matrix with `nOutputs` columns and `batchSize` rows. Moreover, we decide that we use row-major ordering. This means that the column-id is the fast index and we move from one row to the next by moving a pointer by `nOutputs`.

```
for(int b=0; b<batchSize; b++)
    for(int n=0; n<nOutputs; n++)
        output[n + b*nOutputs] = bias[n];
```

BLAS `gemm` functions ask for a leading dimension (LD) for each array. In row-major ordering, the LD tells BLAS how to move a pointer from one row of the matrix to the next. In our case this is always the number of columns of the matrix. In general, we might want to operate on sub-matrices, or we might have allocated each row to be aligned with vector registers. In those case, the LD will be different from the number of columns.

```
gemm(CblasRowMajor, CblasNoTrans, CblasNoTrans,
             batchSize,  nOutputs,   nInputs,
             (Real)1.0,  inputs,     nInputs,
             weight,     nOutputs,
             (Real)1.0,  output,     nOutputs);
```

- **5 maximum points for correctly adding the bias**
- **8 maximum points for correctly calling BLAS**
- **-1 point for each incorrect argument of gemm, to a minimum of 0 points**

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 $G_{\mathbf{b}^{(k)}}$.

Resetting the memory field is optional. Both ordering of the for loops is accepted here.

```
Real* const grad_B = grad[ID]->biases; // size nOutputs
std::fill(grad_B, grad_B + nOutputs, 0);
for(int b=0; b<batchSize; b++)
    for(int n=0; n<nOutputs; n++)
        grad_B[n] += deltas[n + b*nOutputs];
```

The only twist compared to the previous question is that the LD does not change when you ask BLAS to operate on transposed matrices. The LD tells BLAS how to iterate through your matrices as they currently are represented in memory.

```
Real* const grad_W = grad[ID]->weights; // size nInputs * nOutputs
gemm(CblasRowMajor, CblasTrans, CblasNoTrans,
    nInputs,     nOutputs,  batchSize,
    (Real)1.0,   inputs,    nInputs,
    deltas,      nOutputs,
    (Real)0.0,   grad_W,    nOutputs);
```

```
Real* const errinp = act[ID-1]->dError_dOutput; // batchSize * nInputs
gemm(CblasRowMajor, CblasNoTrans, CblasTrans,
    batchSize,  nInputs,    nOutputs,
    (Real)1.0,  deltas,     nOutputs,
```

```
5       weight,      nOutputs,
6       (Real)0.0,   errinp,      nInputs);
```

- **5 maximum points for correctly accumulating the bias gradient**
- **6 maximum points for correctly computing grad W**
- **6 maximum points for correctly computing errinp**
- **-1 point for each incorrect argument of gemm, to a minimum of 0 points**

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

10 points total (+4 bonus)

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`.

The update equations are easily converted to the following loop:

```
1  for (int i = 0; i < size; i++)
2  {
3      mom1st[i] = beta * mom1st[i] - eta * normalization * grad[i];
4      param[i] = param[i] + mom1st[i];
5  }
```

- **6 points**

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:

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

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.

The gradient of the modified loss only adds one term compared to the previous case:

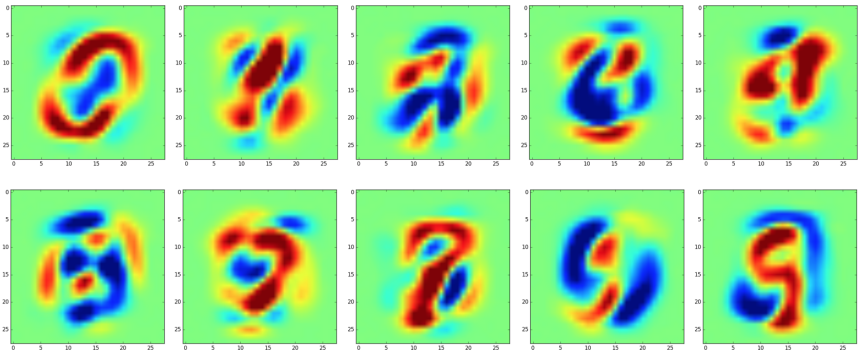

Figure 2: Possible output for the MNIST modes found by the auto-associative net.

```
for (int i = 0; i < size; i++) {
    const Real G = normalization * grad[i] + lambda * param[i];
    mom1st[i] = beta * mom1st[i] - eta * G;
    param[i] = param[i] + mom1st[i];
}
```

- **4 points (bonus)**

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.

The actual outcomes depend on the initialization. However, you should see modes that look somewhat like the ones in figure 2.

- **4 points if the code at this point produces the PCA modes**

Question 3: Non-linearity

15 points total

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`.

The forward step can be performed as:

```
for (int i=0; i<batchSize * size; i++) output[i] = eval(inputs[i]);
```

Where the eval function is:

```
static inline Real eval(const Real in) {
    const Real e2x = std::exp(-2*in);
    return (1-e2x)/(1+e2x);
}
```

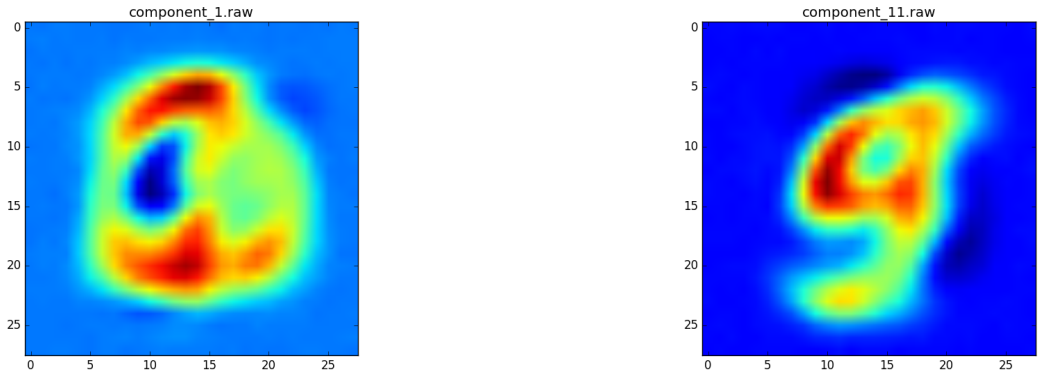

Figure 3: Examples of outputs that could result from one unit in the compression layer being activated with value 1 (left) and -1 (right).

The back propagation can be done either as:

```
for(int i=0; i<batchSize*size; i++) errinp[i] = deltas[i]*(1-output[i]*output[i]);
```

Or as:

```
for (int i=0; i<batchSize * size; i++) errinp[i] = deltas[i] * evalDiff(inputs[i]);
```

Where evalDiff:

```
static inline Real evalDiff(const Real in) {
    const Real e2x = std::exp(-2*in);
    return 4*e2x/((1+e2x)*(1+e2x));
}
```

- **5 points for the forward step**
- **6 points for the back-propagation**

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?

Figure 3 is a simple result which highlights the power of non-linear function approximators. With the same information allowed to cross the compression layer, we can now represent a greater number of modes. Here we show this only by activating compression-layer units with 1 and -1, but in principle even intermediate values may be associated with their own modes.

- **4 Points for including in the submission an explanation mentioning non-linearity.**

Question 4: Parallelization

15 points total

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

Unless otherwise stated, the solution here amounts to writing:

```
1     #pragma omp parallel for
```

before a for loop. Optionally the schedule may be specified (ie. `schedule(static)`).

a) In `LinearLayer::forward`, copying/adding the bias vector `param[ID]->biases` onto the layer's output `act[ID]->output`.

- **1 point.**

b) In `LinearLayer::bckward`, the computation of the bias gradient `grad[ID]->biases` (Careful!).

No points if the for loop causes a race condition. For example the following is **wrong**:

```
1 #pragma omp parallel for
2  for(int b=0; b<batchSize; b++)
3      for(int n=0; n<nOutputs; n++)
4          grad_B[n] += deltas[n + b*nOutputs];
```

- **2 points.**

c) In `TanhLayer::forward`, computation of the non-linearity.

- **1 point.**

d) In `TanhLayer::bckward`, computation of the gradient of the loss w.r.t. the previous layer `act[ID-1]->dError_dOutput`.

- **1 point.**

e) 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).

- **2 points.**

f) In `Network::bckward`, copying the error gradients onto the gradient of the output layer (line 110 of the skeleton).

- **1 point.**

g) The parameter update in `Optimizer::update` and MomentumSGD::step.

This loop is the hardest to parallelize because of the possibly asymmetric work loads between weights and biases of different layers. Any implementation that does not lead to race conditions will not be corrected. Here we propose an approach that dynamically distributes the threads while avoiding false sharing.

```
#pragma omp parallel
for (size_t j = 0; j < parms.size(); j++) {
    if (parms[j] == nullptr) continue; //layer does not have parameters
    if (parms[j]->nWeights > 0) {
        algo.step(parms[j]->nWeights, parms[j]->weights, grads[j]->weights,
            momentum_1st[j]->weights, momentum_2nd[j]->weights);
    }
    if (parms[j]->nBiases > 0) {
        algo.step(parms[j]->nBiases, parms[j]->biases, grads[j]->biases,
            momentum_1st[j]->biases, momentum_2nd[j]->biases);
    }
}
// Clearing the gradients moved out of the parallel region to avoid race conditions
for (size_t j = 0; j < parms.size(); j++)
    if ( parms[j] not_eq nullptr ) {
        grads[j]->clearBias();
        grads[j]->clearWeight();
    }
```

And for the step function:

```
#pragma omp for schedule (dynamic, 64/sizeof(Real)) nowait
for (int i = 0; i < size; i++)
{
    mom1st[i] = beta * mom1st[i] - eta *  normalization * grad[i];
    param[i] = param[i] + mom1st[i];
}
```

- **2 point.**

h) In `main_linear.cpp` processing the MNIST dataset to be placed in the input vector-of-vectors, line 88. (Careful!)

Erasing from the back of the sample indices vector should be done after the parallel loop.

```
#pragma omp parallel for
for (int i = 0; i < batchsize; i++)
{
    const int sample = sample_ids[sample_ids.size() - 1 - i];
    prepare_input(dataset.training_images[sample], INP[i]);
}
sample_ids.erase(sample_ids.end()-batchsize, sample_ids.end());
```

- **2 points.**

i) In `main_linear.cpp` compute the training error and gradient of the NN output, line 97. (Careful!)

Here we only need to issue a reduction.

```
#pragma omp parallel for reduction(+ : epoch_mse)
```

- **2 points.**

j) The same steps as the previous two points for the test set, lines 120 and 128.
Same as the prior point.

- **1 point.**

Maximum number of points for this exercise: 74.
Number of points required for grade 6.0: 60

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.