Math 6397
Computational and Mathematical Methods in Data Science
Problem Set 1
Due on Friday, March 1, at 10 PM

1 Background

1.1 Image Classification

The MNIST database of handwritten digits 0 through 9 has a training set $\mathbf{Y}_{\text{train}}$ of 60 000 examples, and a test set $\mathbf{Y}_{\text{test}}$ of 10 000 examples. The digits have been size-normalized and centered in a fixed-size image. Each digit is digitized as a 28×28 matrix. More information can be found here: http://yann.lecun.com/exdb/mnist. You can see some exemplary data in Fig. 1.

The task is to develop a method to classify these handwritten digits. We will train our method on the examples (with known ground truth classifications) and test the resulting classifier

Problem Formulation

We seek to develop a simple, supervised method for training a classifier for the MNIST dataset. Our method will be based on a non-linear least-squares problem formulation. The unconstrained optimization problem is given by

$$\underset{\mathbf{X}\in\mathbb{R}^{n,m}}{\text{minimize}} \quad \frac{1}{2}\|\sigma(\mathbf{YX}) - \mathbf{C}\|_2^2 \tag{1}$$

where $\mathbf{Y} \in \mathbb{R}^{m,n}$ represents the training examples, where m is the number of examples ($m = 60\,000$) and n is the number of features ($n = 784$). The matrix $\mathbf{C} \in \mathbb{R}^{m,p}$, $c_{ij} \in \{0,1\}$, is a binary matrix that identifies the class each dataset belongs to (i.e., which digit is shown). For example, if the j-th feature vector (i.e., image) contains a 7, the associated jth row in $\mathbf{C}$ is given by $(0,0,0,0,0,0,0,1,0,0)$. The matrix $\mathbf{X} \in \mathbb{R}^{n,p}$ are the sought after weights for mapping the features to the classes. The function $\sigma : \mathbb{R}^{m,p} \to \mathbb{R}^{m,p}$ represents the activation function.

We consider the intensity values of each image as features; consequently, $n = 28(28) = 784$. The number of training examples is $m = 60\,000$. The classes are the digits from 0 to 9, i.e., $p = 10$.

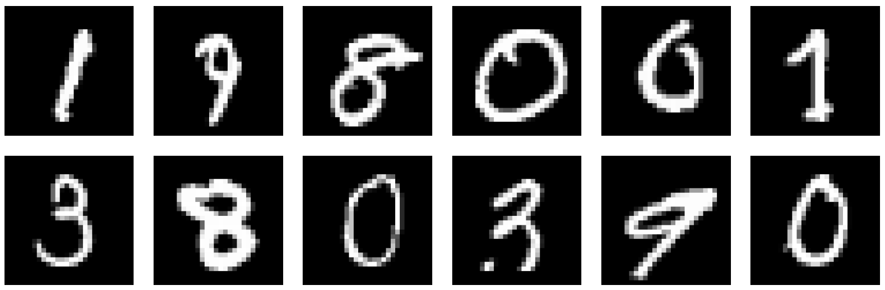

Figure 1: Representative digits from the MNIST dataset. Each figure is of size 28×28.

Activation Functions

For the activation function $\sigma : \mathbb{R}^{m,p} \to \mathbb{R}^{m,p}$, we can consider different non-linearities. Possible examples include $\sigma(\mathbf{X}) = \sin(\mathbf{X})$, the hyperbolic tangent function $\sigma(\mathbf{X}) = \tanh(\mathbf{X})$, the ReLu function $\sigma(\mathbf{X}) = \max(0, \mathbf{X})$, the sigmoid function $\sigma(\mathbf{X}) = \mathbf{E} \oslash (\mathbf{E} + \exp(-\mathbf{X}))$, where $\oslash : \mathbb{R}^{n,m} \to \mathbb{R}^{n,m}$ denotes the Hadamard division and E is a matrix of all ones. Notice that sin, tanh, max and exp are elementwise operations. For example, $\exp(\mathbf{X}) := (\exp(x_{ij}))_{i,j=1}^{n,n}$.

The Dataset

The repository contains a file (class) called `Data.py` located in the `data` subfolder (data/Data.py). This class implements a function called `_read_mnist` that allows you to read (download) the `MNIST` dataset. One of the input argument is a flag to identify if you want to read the training or test data. To read the training data set this flag to `"train"`. To read the test data set it to `"test"`. That is,

```
[Y_train, C_train, L_train] = data._read_mnist("train").
```

Here, $\mathbf{Y}_{\text{train}} \in \mathbb{R}^{60\,000,784}$ corresponds to the training examples and $\mathbf{C}_{\text{train}} \in \mathbb{R}^{60\,000,10}$ the associated binary classification. Similarly, $\mathbf{Y}_{\text{test}} \in \mathbb{R}^{10\,000,784}$ and $\mathbf{C}_{\text{test}} \in \mathbb{R}^{10\,000,10}$ corresponds to the test data, respectively.

An example of how to load and visualize the data can be found in the `xmpls` subfolder (xmpl/viz_mnist.py).

Performance Evaluation

To assess the performance of the classifier, we need to apply the weights we have "learned" to the training and test data and compare it to the given classification. Let $\mathbf{X} = \mathbf{X}_{\text{opt}}$ denote the "optimal" weights. To generate the predicted labels we apply the weights to the data and compose the resulting matrix with the activation function σ. That is, $\mathbf{C}_{\text{pred}} = \sigma(\mathbf{XY})$, where $\mathbf{Y}$ corresponds to the training data $\mathbf{Y}_{\text{train}}$ or the test data $\mathbf{Y}_{\text{test}}$, respectively. To map this prediction to a binary classifier, we assign the predicted class to the entries with the highest value.

1.2 Optimization

This project will expose you to first- and second-order optimization methods for unconstrained optimization problems.

Line Search Methods

We are going to consider the numerical solution of unconstrained optimization problems of the general form

$$\underset{\mathbf{x}\in\mathbb{R}^n}{\text{minimize}} \quad f(\mathbf{x}), \tag{2}$$

where $\mathbf{x} \in \mathbb{R}^n$ is the unknown and $f : \mathbb{R}^n \to \mathbb{R}$ is the objective function. The type of methods we are going to use fall into the category of so-called iterative line search methods. These methods are of the general form $\mathbf{x}^{(k+)1} = \mathbf{x}^{(k)} + t^{(k)}\mathbf{s}^{(k)}$, $k = 1, 2, 3\ldots,$ with line search parameter $t^{(k)} > 0$ and search direction $\mathbf{s}^{(k)} \in \mathbb{R}^n$. More specifically, we will consider approaches of the form

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} - t^{(k)}(\mathbf{B}^{(k)})^{-1}\nabla f(\mathbf{x}^{(k)}), \tag{3}$$

where $\nabla f \in \mathbb{R}^n$ denotes the gradient of the objective function f and $\mathbf{B}^{(k)} \in \mathbb{R}^{n,n}$ is a scaling matrix. (We are going to consider different choices for $\mathbf{B}^{(k)}$ below.) The key questions we are going to address are how to select the line search parameter $t^{(k)}$ to obtain a sufficient decrease in the objective function f and how to compute adequate search directions $\mathbf{s}^{(k)} = -\mathbf{B}^{(k)}\nabla f(\mathbf{x}^{(k)})$. We consider a search direction to be a descent direction if the directional derivative satisfies $(\mathbf{s}^{(k)})^\mathsf{T}\nabla f(\mathbf{x}^{(k)}) < 0$. An outline of the code can be found in Alg. 1.

Algorithm 1 Generic line search method for solving an unconstrained optimization problem of the form (2). This pseudo-code is implemented in LineSearchOpt.py.

```
1:  $\mathbf{x}^{(0)} \leftarrow \mathbf{0}$,   $k \leftarrow 0$,   tol $\leftarrow$ 1.00e−4,   maxit $\leftarrow$ 1000
2:  $[f^{(0)}, \mathbf{g}^{(0)}] \leftarrow \text{fctn}(\mathbf{x}^{(0)})$
3:  while not converged do
4:      $\mathbf{s}^{(k)} \leftarrow \text{computeSearchDirection}(\text{fun}, \mathbf{x}^{(k)}, \mathbf{g}^{(k)})$
5:      $t^{(k)} \leftarrow \text{doLineSearch}(\text{fctn}, \mathbf{x}^{(k)}, \mathbf{s}^{(k)})$
6:      if $t^{(k)} == 0$ then
7:          break
8:      end if
9:      $\mathbf{x}^{(k+1)} \leftarrow \mathbf{x}^{(k)} + t^{(k)}\mathbf{s}^{(k)}$,   $k \leftarrow k + 1$
10:     $[f^{(k)}, \mathbf{g}^{(k)}] \leftarrow \text{fun}(\mathbf{x}^{(k)})$
11:     converged $\leftarrow \text{checkConvergence}(\mathbf{g}^{(k)}, k, \text{tol}, \text{maxit})$
12: end while
```

Notice that iterative schemes like the one presented in (3) require an initial guess $\mathbf{x}^{(0)}$. If not noted otherwise, you can always use an initial guess of $\mathbf{x}^{(0)} = \mathbf{0}$. An implementation of this algorithm can be found in the file LineSearchOpt.py. The main function to execute this algorithm is called `_run`.

Derivative Check

One of the many traps in optimization is working with an erroneous derivative. The following test provides a simple way of checking the implementation of a derivative. To this end, let f be a multivariate function $f : \mathbb{R}^n \to \mathbb{R}$ and let $\mathbf{v} \in \mathbb{R}^n$ be an arbitrary vector in the Taylor expansion

$$f(\mathbf{x} + h\mathbf{v}) = f(\mathbf{x}) + h\nabla f(\mathbf{x})^\mathsf{T}\mathbf{v} + \mathcal{O}(h^2).$$

The vector $\mathbf{g} \in \mathbb{R}^n$ is the derivative of f if and only if the difference

$$|f(\mathbf{x} + h\mathbf{v}) - f(\mathbf{x}) - h\mathbf{g}^\mathsf{T}\mathbf{v}|$$

is essentially quadratic in h. Similarly, we can check the implementation of the Hessian based on the Taylor expansion

$$f(\mathbf{x} + h\mathbf{v}) = f(\mathbf{x}) + h\nabla f(\mathbf{x})^\mathsf{T}\mathbf{v} + \frac{1}{2}h^2\mathbf{v}^\mathsf{T}\nabla^2 f(\mathbf{x})\mathbf{v} + \mathcal{O}(h^3).$$

Accordingly, the matrix $\mathbf{H}$ is the second derivative of f if and only if the difference

$$|f(\mathbf{x} + h\mathbf{v}) - f(\mathbf{x}) - h\mathbf{g}^\mathsf{T}\mathbf{v} - 0.5h^2\mathbf{v}^\mathsf{T}H\mathbf{v}|$$

is essentially cubic in h. An implementation of this derivative check can be found in the file LineSearchOpt.py. The main function to execute this check is called `_deriv_check`. An example of how to use this derivative check can be found in the `xmpl` subfolder (xmpl/rlsq_derivcheck.py)

Notice that you should use random vectors to make sure that the derivative check generalizes. The "output" of the derivative check is the following: A plot of the trend of the error (first, second, and third order) versus the step size h. More importantly, the exact values will be printed in your command window. Inspect the trend of the exponent of the different errors. You will see the predicted behavior (if the derivatives are correct) up until approximately machine precision, i.e., $\mathcal{O}(1.00\text{e}{-16})$.

Check Convexity

A function $f : \mathbb{R}^n \to \mathbb{R}$ is convex if and only if $g : \mathbb{R} \to \mathbb{R}$, $g(t) := f(\mathbf{x} + t\mathbf{v})$ is convex in t. This statement not only allows you to proof convexity of a function in a simple way but also allows you to empirically (heuristically) check if a function f is convex or not (at least, to get an idea). To do this, you can simply select lower and upper bounds $l \in \mathbb{R}$ and $u \in \mathbb{R}$ and plot g as a function of t for random perturbations $\mathbf{v}$. This will allow you to inspect the trend of your objective function along random directions in one dimension. If you observe that your function is not convex along any of those directions, this is a "proof" that it is not convex (counter example).

This "convexity check" is implemented in LineSearchOpt.py. The main function to execute this check is called `_cvx_check`. An example of how to use this function can be found in the `xmpl` subfolder (xmpl/rlsq_cvx_check.py)

2 Assignments

Notice that we will develop first- and second-order optimization algorithms. In a first step, it is probably easier to write the first order optimization method. Once this works, develop the second-order (Newton) method.

1. We are going to consider the objective function of the quadratic problem

$$f(\mathbf{x}) = \frac{1}{2}\mathbf{x}^\mathsf{T}\mathbf{Q}\mathbf{x} + \mathbf{b}^\mathsf{T}\mathbf{x} + c,$$

   where $\mathbf{Q} \in \mathbb{R}^{n,n}$, $\mathbf{b} \in \mathbb{R}^n$, $c \in \mathbb{R}$, and $\mathbf{x} \in \mathbb{R}^n$.

   a) Derive the gradient ∇f and Hessian $\nabla^2 f$. Implement the objective function and check if the derivatives are correct. Create a script named `quadobj_deriv_check.py` and submit it. You can use xmpl/rlsq_deriv_check.py as a template.

   b) Check empirically if f is, in general, a convex function for a random $n \times n$ matrix $\mathbf{Q}$. Subsequently, replace $\mathbf{Q}$ with a positive semi-definite matrix (see `get_spd_mat` in data/Data.py). Create a script named `quadobj_cvx_check.py` and submit it. You can use xmpl/rlsq_cvx_check.py as a template for your implementation. Justify your observations as a comment in your submission.

2. As a prototype for the non-linear least squares problem for image classification we consider the non-linear regularized least squares problem

$$\underset{\mathbf{x}\in\mathbb{R}^n}{\text{minimize}} \quad \frac{1}{2}\|\sin(\mathbf{A}\mathbf{x})-\mathbf{b}\|_2^2+\frac{\beta}{2}\|\mathbf{L}\mathbf{x}\|_2^2,$$

with regularization parameter $\beta > 0$, regularization operator $\mathbf{L} \in \mathbb{R}^{n,n}$, $\mathbf{A} \in \mathbb{R}^{m,n}$, $\mathbf{b} \in \mathbb{R}^m$, and $\mathbf{x} \in \mathbb{R}^n$. For simplicity, we are going to select $\mathbf{L} = \mathbf{I}_n = \text{diag}(1,\dots,1) \in \mathbb{R}^{n,n}$. Provide code for the evaluation of the objective function, its gradient $\nabla f(\mathbf{x})$, as well as the Hessian matrix $\nabla^2 f(\mathbf{x})$. Implement the objective function and check if your implementations of the first and second-order derivative are correct. Create a script named `nlrlsq_deriv_check.py` and submit it. You can use xmpl/rlsq_deriv_check.py as a template.

3. Next, we are going to solve the optimization problem stated in (1), i.e.,

$$\underset{\mathbf{X}\in\mathbb{R}^{n,m}}{\text{minimize}} \quad \frac{1}{2}\|\sigma(\mathbf{Y}\mathbf{X})-\mathbf{C}\|_2^2.$$

We consider tanh as an activation function σ. An implementation of this activation function and its derivatives is provided in xmpl/actnfctn_tanh.py. To assess the performance of the classification after "training" (i.e., after you solved the optimization problem) you can use the function called `check_class` in data/Data.py.

   a) Derive the gradient $\nabla f(\mathbf{X})$ of the objective function. In a first step, use an identity matrix for the Hessian $\nabla^2 f(\mathbf{X})$. Create a script named `nllsqmat_deriv_check.py` and submit it. You can use xmpl/rlsq_deriv_check.py as a template.

   b) Use a (globalized) gradient descent algorithm to solve the optimization problem (1). Execute it for 1 iteration and report the classification accuracy for the training and testing datasets. Execute the solver for 100 iterations and compare it to the results you obtained after one iteration. Name this script `nllsqmat_classify_mnist.py` and submit it. You can use xmpl/rlsq.py as a template.

   c) Derive the Hessian. Consider the activation function tanh. For the Hessian, you will have to implement the action of the Hessian on a vector, i.e., as a function handle for a matrix-vector product. That is, if $\mathbf{H}$ is the Hessian matrix, we implement an "anonymous function" (a lambda function) to apply it to a vector, i.e., $\mathbf{y} = \mathbf{H}\mathbf{x}$. Here, $\mathbf{x} \in \mathbb{R}^{np,1}$ is a vectorized representation of $\mathbf{X} \in \mathbb{R}^{n,p}$. To evaluate the "matvec", the vector $\mathbf{x}$ has to be "tensorized" to a matrix $\mathbf{X} \in \mathbb{R}^{n,p}$ and the output needs to be vectorized again to a vector $\mathbf{y} \in \mathbb{R}^{np,1}$. This will allow you to use the code you used in the former exercises for solving the optimization problem and checking the gradient. Once you have derived these expressions, add the Hessian to your script `nllsqmat_classify_mnist.py` for the derivative check. You can use xmpl/rlsq_deriv_check.py as a template.

   d) Use a globalized Newton algorithm to solve the optimization problem (1). Execute it for 10 iterations and report the classification accuracy for the training and testing datasets. Since switching from Newton to gradient descent only involves a switch of one flag, you can add this to the script `nllsqmat_classify_mnist.py` you have completed in the former part for executing the gradient descent. Notice that you will use an iterative Krylov subspace method (conjugate gradient; CG) to invert the Hessian. This method only requires a matrix-vector product, i.e., you do not need to form or store the Hessian matrix.