UNIVERSITY OF OXFORD

Machine Learning
Hilary Term 2016
Week 4

Classifying digits and tuning optimizers

In this practical, we address the problem of classification.

Given the MNIST dataset which consists of 1024-dimensional inputs corresponding to pixels of a 32×32 image showing a handwritten digit from 0 to 9, and corresponding labels stating what digit the image shows (in the code the class labels range from $1 - 10$ since Lua is 1-indexed), the goal is to learn a classifier that predicts the class (label) of several test-set inputs.

This practical's files can be found here: https://github.com/oxford-cs-ml-2016/practical3

Read `README.md` for setup instructions for the lab machine. Clone the entire repository or download all 3 `.lua` files.

Task 1: Introduction and Review

Here, we'll look at optimizing a simple function of one variable, just to get a feel of how the optimizer works.

In the models we will be and have been looking at, our strategy always takes this form:

- get a dataset, consisting of m pairs $(\mathbf{x}_i, y_i)$
- define a model with parameters $\mathbf{w}$, called $f_{\mathbf{w}}(\mathbf{x})$, that tries to **learn** a mapping from our inputs $\mathbf{x}$ to our targets (aka labels) y
- define a loss function that tells us how good our model fits our data, i.e.

$$L(\mathbf{w}) = \sum_{i=1}^{n} \ell(f_{\mathbf{w}}(\mathbf{x}), y_i)$$

  where $\ell(\text{prediction}, \text{target})$ is our loss (error penalty) function, e.g. the squared error $\ell(\text{prediction}, \text{target}) = (\text{prediction} - \text{target})^2$ which we used for regression last time.
- find a model that makes our error on the training set, $L(\mathbf{w})$, small. To do this we **minimize** it over $\mathbf{w}$. Each value of $\mathbf{w}$ gives a different model $f_{\mathbf{w}}$, and we're searching an infinite set of models for a good one.

To understand how the last step translates into code, we'll minimize a **useless** toy example function:

$$f(x) = \frac{1}{2}x^2 + x\sin(x)$$

which has the following gradient: (1-dimensional, since f is univariate)

$$\nabla_x f(x) = [x + x\cos x + \sin x].$$

Take a look at `simple_example.lua` closely, making sure you understand what this `feval` function does each time it is called by the `optim` library.

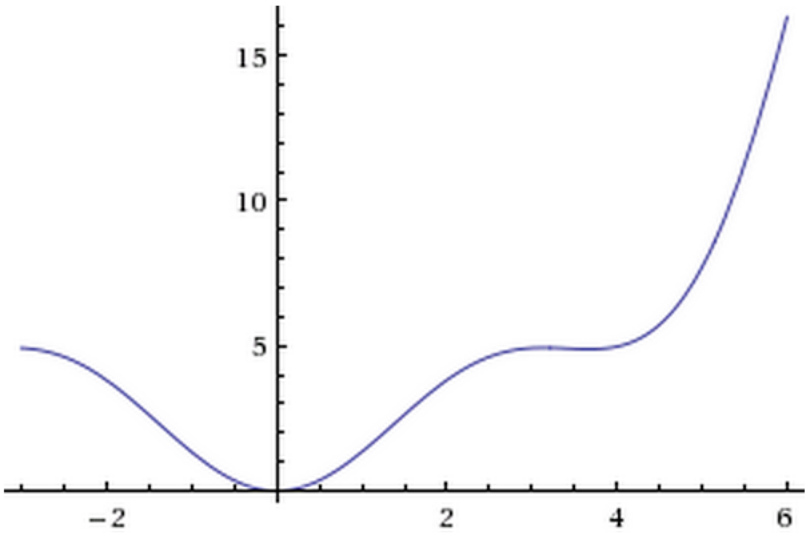

Figure 1: plot of the function

The plot was generated in WolframAlpha: http://www.wolframalpha.com/input/?i=plot+0.5*x%5E2+%2B+x+sin%28x%29+from+-3+to+6.

Exercise (not for handin): change the initialization point of the function to make the optimization converge to a different local minimum. Compare the local minimum you find to the one seen in the plot.

When training our models, the only difference is inside the `feval` function and the setup code that we need before it, to load our dataset. Last week, the way `feval` was written, instead of computing the *real* gradient of the function $L(\mathbf{w})$ that we have above which involves handling all the data, we computed the gradient of just one term in the sum so we only use one data point. If we randomly sample this point, on average we will get the same value as $L(\mathbf{w})$. The difference is that this estimate has high variance (it is noisy, i.e., it varies a lot depending on which data point we sampled) but it is far **cheaper** to compute, especially for a large dataset. SGD turns out to converge fine even though the gradient is noisy.

The problem is that our estimate of the gradient will be very noisy. We could make a tradeoff by taking a small subset of the dataset, and computing the gradient over this instead, i.e., using **mini-batches**. This will give us gradients that are both less noisy and still cheap to compute.

This is what we will be doing this week, albeit with a different loss function and with different data and a different task: classification of handwritten digits using logistic regression.

Read over the code in `practical3.lua`, focusing on the sections after the data gets loaded and particularly the `feval` function.

Optional: Brief overview of `backward` and `forward`

We'll see how these functions are implemented later, but here is a brief overview if you are curious. For sake of explanation, let's say we only feed in one data point, and we'll use the same notation as earlier.

- `out = model:forward(x_i)` computes $f_{\mathbf{w}}(\mathbf{x}_i)$ where $f_{\mathbf{w}}$ is our model with its current parameters $\mathbf{w}$, and stores the result in `out`.
- `loss = criterion:forward(out, y_i)` computes the loss $\ell(f_{\mathbf{w}}(x_i), y_i)$ with respect to the true value y_i
- `dl_dout = criterion:backward(out, y_i)` computes $\frac{\partial \ell(\cdots)}{\partial f_{\mathbf{w}}(\mathbf{x}_i)}$
- `model:backward(x_i, dl_dout)` computes $\nabla_{\mathbf{w}} \ell$ and stores this gradient in a place we have a reference to, usually called `gradParameters` in our code. (This is why you see no variable assignments on this line.)

Task 2: Tuning the optimizer

Our main task is to find a configuration (learning rates, mini-batch size, line search, momentum, etc.) and corresponding meta-parameter values for an optimizer algorithm (SGD, adagrad, or L-BFGS) so as to minimize the number of errors on the test set.

To find out all the configuration options for the optimization algorithms you will have to read the documentation here: https://github.com/torch/optim.

To help you, we have provided some starter configuration options for SGD. However, more options are possible and the values provided are not necessarily optimal. With a better choice of parameters, you will converge much **faster**. Additionally, you can tweak other options such as the minibatch size.

- Modify the code so that it evaluates its performance on the test set after every **minibatch** (or epoch if you prefer), then plot both the **test loss** and **training loss** on the same plot, rather than just the training set loss as the code does now.
  Handin: the plot, and a brief explanation of the code required to do this.

- Find a configuration that works well (for the optimizer and perhaps mini-batch size and others). The classification error is the percentage of instances that are misclassified, in either the training or test set.
  Handin: Find a configuration that predicts well, and report your **training set** and **test set classification error** as we just defined. Explain your findings about which optimizers were easier to configure than others. Very briefly explain why the final model you pick solution is "good". Show your code for computing the classification error, which should be short.

Handin

See the bolded "**Handin:**" parts above.

In the source file, you can search for sections marked `TODO` for where you can enter your code. At this point, however, you will have **completed lectures** and since you have 2 weeks for this practical, you should have enough time to read and understand all the relevant parts of the code.

The advanced task and the introduction example are not for handin.

Advanced: For enthusiastic students (optional)

1. Implement a gradient checker that uses a finite-difference approximation to compute the gradient $\frac{\partial \text{ criterion output}}{\partial w}$ (criterion output is Torch's name for the loss function output), then compare this to the value produced in the `feval` function.
2. Implement a Jacobian checker to verify the `nn.Linear` class computes derivatives correctly. This Jacobian computation task will become easier once we understand the purpose of backpropagation (next lecture).

This is the standard way people unit-test numerical code involving derivatives, both when prototyping and when writing large software systems.

Hint: See http://en.wikipedia.org/wiki/Finite_difference#Relation_with_derivatives for details on finite-difference approximations. Think about how this extends to first-order partial derivatives. The bottom of the Wikipedia page has some examples for partial derivatives.