Deep Learning	November 9, 2015
Exercise 1	
Lecturer:Lior Wolf	*TA: Adam Polyak*

Introduction

In this exercise you will:

1. Train a simple network with the sigmoid activation function (the nonlinearity between linear layers) on a small dataset.
2. Implement a module for a new activation function, and use these to replace the sigmoid
3. Check that the module's backward function is correct using an approximation to the Jacobian.

Use the same environment you used in the previous exercise. Additionally, you are provided with template code to help you complete the exercise, with parts to fill-in marked as "TODO".

Question 1: Xor network

In this question we will train a simple network to execute binary XOR function. Use the following links for documentation and reference:

- nn package
- example for neural network training
- optim package

Follow these steps:

1. First, create a tensor that will contain a truth table for binary XOR. This will be our training data:

Table 1: My caption

input	1	1	0	0
input	1	0	1	0
output	0	1	1	0

2. Define a neural network with the following structure:

   - Input layer: tesnor of size 1×2 (two "bits")
   - Hidden layer: Fully connected with 3 hidden units
   - Activation layer: Tanh
   - Output layer: Fully connected with single output unit

3. Define a criterion for the neural network - use mean squared error.

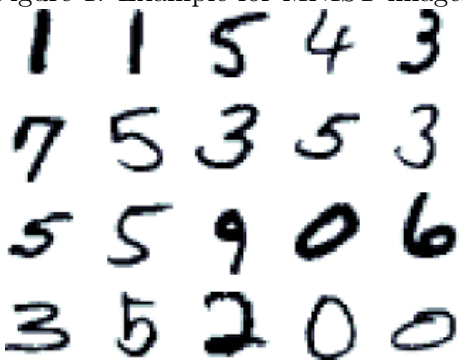
4. Train the network using optim package. As a template for training you can use the file "learn_xor.lua" in the git repository which contains "TODO" comments for you to implement. Train the network and test it for each line in the dataset you created. Write the results in .txt file.

Submit your code, test results and save your model on nova.

Question 2: MNIST

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.

Figure 1: Example for MNIST images



You are provided with training code "q2/doall.lua" and data loading code "q2/dataset-mnist.lua". The code trains a simple logistic regression model to solve the task at hand. Your 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. As before, the code contains "TODO" comments in the relevant locations in the code.

1. Currently, the code evaluates the model on the training data each mini-batch. Modify the code to report the loss per epoch.

2. Modify the code so that it evaluates its performance on the test set after every epoch, then plot both the test loss and training loss, rather than just the training set loss as the code does now. In addition, make sure you use all available training data.

3. The classification error is the percentage of instances that are misclassified, in either the training or test set. Find a configuration that predicts well, and report your training set and test set classification error as just defined.

4. Find an optimization configuration that works well (learning rates, mini-batch size, line search, momentum, etc.). You are provided with 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. The new configuration should result **faster** (in terms of #epochs) convergence. **Optional:** You are not limited to SGD and are encouraged to try other optimization methods, see Torch7 optim package.

Submit your modified code together with plots of the original configuration and the new one. The plots should contain test/train loss and classification error for test/train.

NOTE: Evaluate your new configuration vs the original for at least 100 epochs.

Question 3: ReQU activation

Network

In this question we will train a model on the iris flower dataset. You are provided with code for training and setup, in "q3/train.lua", "q3/iris_loader.lua" for loading the dataset, and "q3/doall.lua" that actually runs the training process. The dataset is read from iris.data.csv. The code is similar to the previous question, except now the model is deeper and we are using a simpler dataset. Additionally, we are now doing everything in full batches, computing the loss and gradient on all of the data in each iteration. Some nice figures showing what the dataset looks like:

The model the code implements is:

input (4 dim) => linear => non-linearity => linear => log softmax => cross-entropy loss

where the non-linearity is a sigmoid or "ReQU", the latter of which is not implemented yet.

Figure 2: scatterplot of the 4 input features, with colour-coded classes (source: Wikipedia)

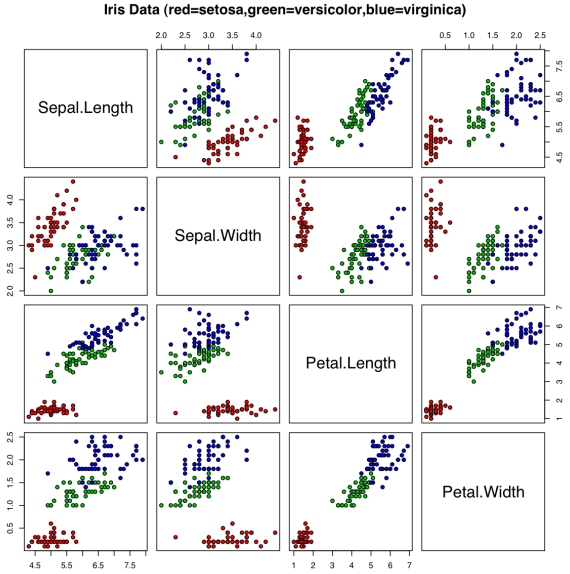


Figure 3: one of the types of iris (source: Wikipedia)

Implementing a new layer

Read the Torch tutorial on this topic: http://code.madbits.com/wiki/doku.php?id=tutorial_morestuff, it has a useful code example.

Summary: When we implement a model, keep in mind:

- `forward` and `backward` methods (in the parent nn.Module class) already call the other methods below, so don't override them directly.
- override the `updateOutput` method to implement the forward pass, to the layer activation, z from input x.
- override the `updateGradInput` method to implement part of the backward pass, to compute the derivative of the loss wrt your layer's inputs ($\frac{\partial loss}{\partial x}$), in terms of the derivative of the loss wrt your layer's outputs ($\frac{\partial loss}{\partial z}$):

$$\underbrace{\frac{\partial loss}{\partial x}}_{\text{gradInput}} = \underbrace{\frac{\partial loss}{\partial z}}_{\text{gradOutput}} \cdot \overbrace{\frac{\partial z}{\partial x}}^{\text{deriv. of output wrt input}}$$

where the dot is a matrix multiplication; the right-hand side is the Jacobian matrix of our layer's function f that we never explicitly create. See lecture for details. Make sure you understand this, as this is the recursion we do in backprop.

- override the `accGradParameters` method for the other part of the backward pass if your layer has parameters, to compute the gradient of the loss wrt your layer's parameters.

ReQU

Here, we'll implement a made-up activation function that we'll call the Rectified Quadratic Unit(ReQU). Like the sigmoid and ReLU and several others, it is applied element-wise to all its inputs:

$$z_i = \mathbb{I}[x_i > 0] x_i^2 = \begin{cases} x_i^2, & \text{if } x_i > 0 \\ 0, & \text{otherwise} \end{cases}$$

Or in matrix operations, where $\odot$ is the element-wise (aka component-wise) product, and the parenthesized expression is an element-wise truth test giving a vector of 0s (falses) and 1s (trues):

$$z = (x > 0) \odot x \odot x$$

Since the problem is easy, both models easily overfit to the training data. We don't have test data and we didn't split the training data into parts since it is small. A viable way to evaluate a model on such small data would be k-fold cross validation but we will not do this.

Remarks/tips:

- Your layer must be able to handle minibatches. Doing so should not be difficult, though. You should not need to write a special case for 1 and 2 dimensions, since you're just doing an element-wise operation.
- `resizeAs` will rarely reallocate memory because the minibatch size rarely changes.
- You will be able to check your answer in the next section, so don't worry if you're not completely sure if your gradient is correct.
- These may be helpful:
  - https://github.com/torch/torch7/blob/master/doc/tensor.md#querying-elements

- https://github.com/torch/torch7/blob/master/doc/maths.md#logical-operations-on-tensors

1. Run the code. It will output several plots. The code's comments explain what they are, but you only need to worry about the loss curve at first. The heatmaps show the decision boundaries between two variables of your choice.
2. Compute the derivatives for "ReQU" layer. That is, write a formula for `gradInput` ($\frac{\partial loss}{\partial x}$) in terms of `gradOutput` ($\frac{\partial loss}{\partial z}$). It will help to write them in matrix notation, even if you compute it element-wise first. Again, follow the "TODO" comments in the code, the "ReQU" module should be implemented in the relevant file.
3. Implement this layer as shown in the tutorial linked to above. For speed reasons, do not use a loop in your `updateOutput` or `updateGradInput`, and do not use the apply function. Try to minimize memory usage, as in the Torch tutorial's example.

Submit your code (i.e. modified original code) and the convergence plots for the model using sigmoid and ReQU activation.

Question 4: Jacobian testing

Our next task is to modify a Jacobian checker. Recall the Jacobian is a $m \times n$ matrix of derivatives for a multivariate function $f : \mathbb{R}^n \to \mathbb{R}^m$:

$$\frac{d\mathbf{f}}{d\mathbf{x}} = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \cdots & \frac{\partial f_m}{\partial x_n} \end{bmatrix}$$

, each ith row being a gradient of an element, f_i, of the output vector, f. Note that we're doing it to compute the derivative of the output wrt the input because that's what `updateGradInput` does, and this is the function we want to test.

This matrix is implicitly what we're computing in the backward pass. Put another way, all mn of these derivatives are used to compute the backward pass, so numerically verifying these, using finite differences, allows us to check that our backward pass is correct in isolation. Note that this is the standard way people unit-test numerical code involving derivatives, both when prototyping and when writing large software systems.

The computation goes as follows:

Using finite difference approximations, we can compute $\frac{\partial f_i}{\partial x_j}$ for all i and j and compare this to the values produced using backprop. Instead of perturbing one input and looking at a scalar function value, we can get *one whole column* of the Jacobian at once:

$$\frac{\partial \mathbf{f}}{\partial x_i} \approx \frac{\mathbf{f}(x_1, ..., x_i + \varepsilon, ..., x_n) - \mathbf{f}(x_1, ..., x_i - \varepsilon, ..., x_n)}{2\varepsilon}$$

One part of backprop computes:

$$\underbrace{\frac{\partial loss}{\partial x}}_{\text{gradInput}} = \frac{\partial loss}{\partial z} \cdot \frac{\partial z}{\partial x} = \underbrace{\frac{\partial loss}{\partial z}}_{\text{gradOutput}} \cdot \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \cdots & \frac{\partial f_m}{\partial x_n} \end{bmatrix}$$

so selecting `gradOutput` to be a vector with only one 1 and the rest of the elements 0 lets you select out one whole row, by giving this to backward or `updateGradInput`.

We can repeat this to compute an entire approximate Jacobian and supposedly-true Jacobian, then compare.

If our layer had parameters, we could do the same to check those derivatives. Remember, a layer can have two vector-valued inputs, as in $\mathbf{f}(\mathbf{x}; \mathbf{w})$ where $\mathbf{w}$ are the parameters, so we could actually compute the approximate Jacobian wrt either one of these, as we do in `backprop`. The only difference would be that we're perturbing $\mathbf{w}$ instead of $\mathbf{x}$, and when we call backward or `accGradParameters` to get the true Jacobian, we look at `getParameters` as returned by the module instead of `gradInput`.

To simplify your task, you are provided with code for a simplistic method of estimating the Jacobian. The method in the code computes the single-sided finite difference:

$$\frac{\partial \mathbf{f}}{\partial x_i} \approx \frac{\mathbf{f}(x_1, ..., x_i + \varepsilon, ..., x_n) - \mathbf{f}(x_1, ..., x_i, ..., x_n)}{\varepsilon}$$

but this estimate is less accurate than the two-sided version above. For such a simple function, we should not notice much difference.

Note that the result for our ReQU layer is a diagonal matrix: since it is an element-wise operation, z_i depends on x_j if and only if $i = j$ (on the diagonal).

Modify the provided code to use the two-sided version. Test the ReQU activation written in the previous question using the modified code. Submit the code and report the results of your testing - i.e. the diff between the Jacobians (L_2 distance for example).

NOTE: You should be able to do all these steps without allocating any more memory than the existing code, so the code will be fast. Hint: make clever use of the tensor that we write the estimated Jacobian to. You can write to it twice in an iteration.