Machine Learning

February 25, 2015

Implementing your own layer

Today we'll implement our own neural net module/layer. In lecture, we saw several, such as linear layer which computes $\mathbf{z} = \mathbf{W}\mathbf{x} + \mathbf{b}$ and the sigmoid module which computes $z_i = \frac{1}{1+e^{-x_i}}$ (i.e. element-wise).

This practical's files can be found here: https://github.com/oxford-cs-ml-2015/practical4

Read `README.md` for setup instructions for the lab machine. Clone the repository to get all the files:

```
git clone https://github.com/oxford-cs-ml-2015/practical4
cd practical4
```

Outline

We will:

1. (code provided) Train a simple network with the sigmoid activation function (the non-linearity between linear layers) on the Iris dataset.
2. Implement a module for a new activation function, and use these to replace the sigmoid.
3. Check that our gradient is correct by writing a test to see if the whole model's derivatives are correct.
4. Check that the module's backward function is correct using an approximation to the Jacobian.

In this document, let $\mathbf{z}$ represent the output of a module, and $\mathbf{x}$ its input. That is, $\mathbf{z}$ is a function of $\mathbf{x}$. In our custom layer we won't have parameters.

Train the basic simple network

We provide code for training and setup, in `train.lua`, `iris_loader.lua` for loading the dataset, and `main.lua` that actually runs the training process. The dataset is read from `iris.data.csv`. The code is similar to last time, 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.

For some nice figures showing what the dataset looks like, see http://en.wikipedia.org/wiki/Iris_flower_data_set.

We defined a function called `create_model` that creates and returns the model and the criterion objects, so we can separate this step from the training, as we will make use of the model and criterion later in the gradient checker. The model it implements is:

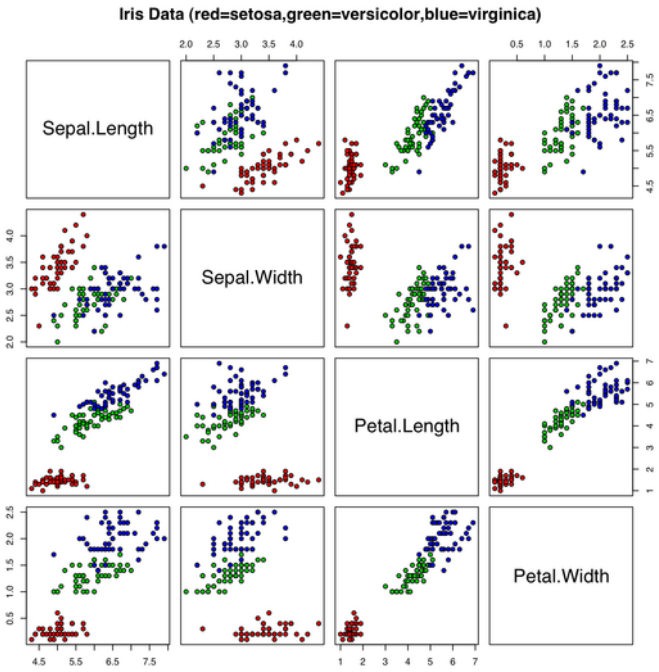

Figure 1: scatterplot of the 4 input features, with colour-coded classes (source: Wikipedia)

Figure 2: one of the types of iris (source: Wikipedia)

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.

Try running the code if you like.

Implementing a new layer/module

Read the Torch tutorial on this topic: http://code.cogbits.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 compute $\mathbf{z}$ from $\mathbf{x}$
- override the `updateGradInput` method to implement part of the backward pass, to compute the derivative of the loss wrt your layer's inputs ($\frac{\partial loss}{\partial \mathbf{x}}$), in terms of the derivative of the loss wrt your layer's outputs ($\frac{\partial loss}{\partial \mathbf{z}}$):

$$\underbrace{\frac{\partial loss}{\partial \mathbf{x}}}_{\texttt{gradInput}} = \underbrace{\frac{\partial loss}{\partial \mathbf{z}}}_{\texttt{gradOutput}} \cdot \frac{\partial \mathbf{z}}{\partial \mathbf{x}}$$

  where these are matrix operations. *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

The "ReQU" unit

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):

$$\mathbf{z} = (\mathbf{x} > 0) \odot \mathbf{x} \odot \mathbf{x}.$$

Handin: Compute the derivatives for this layer. That is, write a formula for `gradInput` ($\frac{\partial loss}{\partial \mathbf{x}}$) in terms of `gradOutput` ($\frac{\partial loss}{\partial \mathbf{z}}$). It will help to write them in matrix notation, even if you compute it element-wise first.

Handin: 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.

You can now rerun the code. 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 if you haven't already seen them in a previous practical:
  - 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

Testing the module in the full network, via gradient checking

Background/hint: Say we have a univariate function of n variables, $E(w_1, \ldots, w_n)$. Its gradient is

$$\frac{dE}{d\mathbf{w}} = \begin{bmatrix} \frac{\partial E}{\partial w_1} \\ \vdots \\ \frac{\partial E}{\partial w_n} \end{bmatrix}.$$

So to (approximately) compute a gradient, we need to (approximately) compute all the partial derivatives. This gives us our finite difference approximation, based on the definition of the derivative:

$$\frac{\partial E}{\partial w_i} \approx \frac{f(w_1, \ldots, w_i + \epsilon, \ldots, w_n) - E(w_1, \ldots, w_i - \epsilon, \ldots, w_n)}{2\epsilon}$$

for small ϵ. A reasonable ϵ would be around 10^{-4} to 10^{-7}.

We can use this idea for neural nets, if we treat the whole net (*including* the loss function) as a univariate function $E(\mathbf{x}; \mathbf{w})$. If we pick a random $\mathbf{x}$ (i.e. generate a fake minibatch), and random

parameters $\mathbf{w}$, we can approximate the gradient of f wrt $\mathbf{w}$, evaluated at this specific value of $\mathbf{w}$ (and $\mathbf{x}$). Then we can do one forward pass followed by a backprop to get our supposedly correct gradient.

If it is "close", our derivative computation for ReQU is probably correct. If not, Torch's code is unlikely to have an error, so it's probably your ReQU module. We will be able to check this more conclusively in the next section. To measure if these two vectors are close, we can compute the symmetric relative error:

$$2 \cdot \text{relative error} = \frac{\|\mathbf{g}_1 - \mathbf{g}_2\|}{\|\mathbf{g}_1 + \mathbf{g}_2\|},$$

where $\mathbf{g}_i$ are the two ways of computing the gradient. This ratio should have similar magnitude to the ϵ we picked above. If it is not, we've either found a problem in the gradient checking code, or in the model's gradient computation.

Of course, we need to evaluate the gradient at a specific value of the parameters $\mathbf{w}$, since the gradient is a function. In the code, we're evaluating the derivative at parameters sampled from something similar to $w_i \sim N(0, 0.01^2)$ or another value close to 0 using the default initialization provided by torch's `nn.Linear`. This is to help avoid the vanishing gradient problem, where many gradients being multiplied together (by chain rule) in the backprop make the gradient tend to 0. In short, if we veer too far from zero, we may land in the "flat" part of the sigmoid, where its slope is very flat. (See the bonus question for a similar conceptual question.) This is a problem during optimization, but here it causes us to lose a lot of precision due to rounding error.

Note: it is essential to include the loss function in this computation and not just `model`, so that the function that we are approximating the derivative of is scalar-valued. Otherwise, we'd need to compute a Jacobian instead of a gradient, which is the next part.

Handin: use this finite-difference gradient approximation to compute the gradient of the full model and check its correctness. The `gradcheck.lua` file has blanks that you may fill in.

Remarks/tips:

- If there are n parameters in the model, we need to do $2n$ forward passes (evaluations of f) to use the above formula.
- Remember how to get a vector **reference** to all the parameters in the model and the gradient of the loss wrt these same parameters: `model:getParameters()` returns these two things.
- We should be able to do all these steps without performing any copies at all, besides allocating space to store the approximate gradient, so that your code is fast.

Testing the module in isolation (unit tests), via Jacobian

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

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

each ith row being a gradient of an element, f_i, of the output vector, $\mathbf{f}$. 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 like in the previous part) 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 (similar to the one from before):

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:

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

One part of backprop computes

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

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 them similarly to before.

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, we have provided code for a simplistic method of estimating the Jacobian. The method in the code computes the single-sided finite difference:

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

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 output is a diagonal matrix: since this is an element-wise operation, z_i depends on x_j if and only if $i = j$ (on the diagonal).

Handin: modify the provided code to use the two-sided version. Very briefly give an overview of your changes.

Handin

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

Advanced: conceptual question (optional)

1. Explain why we initialize the bias to random numbers larger than 0. What happens if we initialize it to a value below zero? Does this affect our ability to train?
2. Suppose we have a simple network of the shape (linear => sigmoid => linear => sigmoid => ... => linear). Write out the chain rule for computing the derivative of the final outputs of this network with respect to the parameters of the first linear layer. What can you say about the vanishing gradient problem using this expression?