Programming Assignment 1: Loss Functions and Backprop

Deadline: Monday, Feb. 6, at 11:59pm

Submission: You must submit two files through MarkUs[1]: a PDF file containing your writeup, titled `a1-writeup.pdf`, and your code file `models.py`. Your writeup must be typeset using LaTeX.

The programming assignments are individual work. See the Course Information handout[2] for detailed policies.

You should attempt all questions for this assignment. Most of them can be answered at least partially even if you were unable to finish earlier questions. If you think your computational results are incorrect, please say so; that may help you get partial credit.

Introduction

This assignment is meant to get your feet wet with computing the gradients for a model using backprop, and then translating your mathematical expressions into vectorized Python code. It's also meant to give you practice reasoning about the behavior of different loss functions.

We will consider a multilayer perceptron model with one hidden layer, similar to the ones from class:

$$\mathbf{r} = \mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)} \tag{1}$$
$$\mathbf{h} = \phi^{(1)}(\mathbf{r}) \tag{2}$$
$$z = \mathbf{w}^{(2)\top}\mathbf{h} + b^{(2)} \tag{3}$$
$$y = \phi^{(2)}(z) \tag{4}$$

But we'll explore another activation function we haven't discussed yet, called *softsign*:

$$\mathrm{softsign}(z) = \frac{z}{1 + |z|}. \tag{5}$$

This is another example of a sigmoidal function, somewhat similar to the logistic and tanh functions; its range is $[-1, 1]$. We'll consider softsign as the activation function for both the hidden units and the output unit of the network, and see how it compares against alternatives.

In order to use softsign as our output activation function, we need to transform the output so that the predictions are in $[0, 1]$:

$$s = \mathrm{softsign}(z) \tag{6}$$
$$y = \frac{s + 1}{2}. \tag{7}$$

The task will be our running example from class — handwritten digit classification using the MNIST dataset. We'll use a subset of the data, and consider the binary classification task of distinguishing 4's and 9's. To make things more interesting, the labels are noisy: 10% of the training labels have been flipped.

[1] `https://markus.teach.cs.toronto.edu/csc321-2017-01`
[2] `http://www.cs.toronto.edu/~rgrosse/courses/csc321_2017/syllabus.pdf`

Starter Code

The starter code contains four files:

- `models.py`, the Python module which has full or partial implementations of all the models considered in this assignment. This is the only file you will need to modify.
- `util.py`, a file contining some helpful utilities (e.g. gradient checking). You don't need to use anything here directly, but you might read through it if you're interested.
- `data.pk`, a Pickle file containing the training, validation, and test data.
  - This contains the inputs and targets for the training set, validation set, and test set. (We only use the training and validation data in this assignment.) For the training targets, you have both the clean targets `data['t_train_clean']` and the noisy ones `data['t_train_noisy']` (i.e. where 10% of the labels were flipped).
- `trained-model.pk`, a partially trained model which we use to test your derivative calculations.

Before doing this assignment, you should read through `models.py`. In particular, observe a few things:

- This file includes two sets of classes: those implementing models, and those implementing cost functions. The cost functions know how to compute their values and derivatives given the predictions and targets, but are agnostic about how the predictions are made. Similarly, the model classes know how to compute their predictions and backpropagate derivatives given the loss derivatives. This provides modularity, since we can easily swap in different models and cost functions.
- In addition to the model described above, we've also given you implementations of two activation/loss pairs from lecture (logistic activation with squared error, and logistic activation with cross-entropy). We've also given you a linear model. You may like to refer to these classes as a guide when doing your implementation.
- Even though the activation function for the output unit is technically part of the model, for the implementation it is often more convenient to group it together with the loss function, since that makes the computations simpler and more numerically stable. This is why the model classes produce z rather than y, and the loss function classes take in z rather than y. Notice how short the `LogisticCrossEntropy` implementation is!
- We are generous enough to provide you with a finite differences gradient checker. The function `check_derivatives` checks your derivative calculations for both the model and the loss function. You might like to check out the implementations of the checking routines in `util.py`.
- The `print_derivatives` function is only there for marking purposes; more on this below.
- The `train_model` class trains the model using stochastic gradient descent.

Part 1: Implementation (5 points)

Your first task is to finish the implementation of the gradient computations for the multilayer perceptron and softsign-with-cross-entropy loss function. In particular, you will need to write three functions:

1. `softsign_prime`: this function should compute the derivative of the softsign function with respect to its argument. It takes an array `z` and operates elementwise.

2. `SoftsignCrossEntropy.derivatives`: This class represents the softsign activation function followed by cross-entropy loss. The method `derivatives` should compute the loss derivatives with respect to z; it takes an array and operates elementwise.

3. `MultilayerPerceptron.cost_derivatives`: This class implements the MLP model; you need to implement the backprop calculations for the derivatives. The method `cost_derivatives` takes in a batch of inputs $\mathbf{X}$, a `dict` of activations (see the method `compute_activations`), and a vector `dLdz` containing the loss derivative with respect to z for each training example.

In order for us to evaluate the correctness of your implementation, **please provide the output of the function** `models.print_derivatives()`, which prints out certain entries of the derivative matrices and vectors. **You must also submit your completed** `models.py`.

In order to make your life easier, we have provided you with the function `models.check_derivatives`, which checks your derivatives using finite differences. Since we are giving you a gradient checker, you have no excuse for not getting full marks on this part!

Part 2: Conceptual Questions (5 points)

1. [**2 points**] Recall our discussion from Lecture 4 about how the logistic activation function with squared error loss has a weak gradient signal when the model's predictions are very wrong, whereas the logistic activation function with cross-entropy loss has a strong gradient signal.

   Plot all three activation-function-loss functions as a function of z for the target $t = 1$, with z ranging from $[-5, 5]$. Based on this plot, which of the loss functions do you think would cause the training loss to decrease the fastest if you start from a very bad initialization? Which would cause it to decrease the slowest?

   Now train MLPs with each of the three loss functions, using the function `train_from_scratch`, but setting the initial standard deviation of the parameters to 10. Do the results support your prediction? You may want to repeat each run several times, since there is some variability between runs.

2. [**2 points**] Now let's consider training with the noisy labels; you can enable this by passing the flag `noisy=True` to `train_from_scratch`. Different loss functions have differing degrees of robustness to mislabeled examples. Based on your plot of the cost functions, which one do you think will get the lowest *training* error, and which one will get the highest training error? (I.e., which loss function will try hardest to nail every training example, including the mislabeled ones, and which one is most willing to give up?)

   Which one do you think will get the lowest validation error? (Note: there isn't a clear-cut answer to this question, so you might discuss some factors which would cause certain ones to get higher or lower validation error.)

Now train MLPs with each of the three cost functions and see what happens. Were your predictions correct? (Use the default standard deviation of 0.1 for the initialization, rather than 10.) You may want to repeat each run several times, since there is some variability between runs.

3. **[1 point]** There's a separate optimization-related issue: sometimes hidden units can *saturate* when their activations are very close to 0 or 1. To see why this happens, look at the backprop equations you derived for your implementation; they include the derivative of the activation function for the hidden units. When the inputs r_k to the hidden unit activation function are large in magnitude, you land in the flat region of the activation function, so the derivatives are small in magnitude.

   Let's say you have a choice between using the logistic or softsign nonlinearity for the hidden unit activations. Which one would be preferable from the perspective of getting saturated units unstuck? I.e., if the input r_k is large in magnitude, which activation function will result in a stronger gradient signal?

   You do not need to run any experiments for this part.

What you need to submit

- A PDF file `a1-writeup.pdf`, typeset using LaTeX, containing your answers to the conceptual questions as well as the output of `print_derivatives`.
- Your completed implementation of `models.py`.