Assignment 1 – Gradient-based Learning

Yoav Goldberg

In this assignment we will implement a log-linear classifier and a multi-layer perceptron classifier. We will also write the training code for them. All the classifiers will use the hard cross-entropy loss.

We will then test the classifiers on the language identification task, and the XOR task.

1 Gradient Checks – 10 pts

When implementing the classifiers, you will write code for computing the gradients. This kind of code is very error prone. Luckily, there is a method for verifying your gradients are correct. This is done through the *gradient checking* technique. The assignment page includes a reference to an explanation of this technique.

As the first part of this assignment, you will implement gradient checking. Complete the missing parts in the file `grad_check.py` according to the instructions.

You can then run the file (`python grad_check.py`) for some basic sanity checks to ensure your code is not completely failing. These checks are not comprehensive, you can add additional checks of your own. If you do, add them such that they only run when running the file (like the current checks), not when importing it.

2 Log-linear classifier – 20 points

We will implement a log-linear classifier. This is a classifier of the form:

$$f(\mathbf{x}) = \text{softmax}(\mathbf{W}\mathbf{x} + \mathbf{b})$$

With the decision rule:

$$\hat{y} = \arg\max_i \text{softmax}(\mathbf{W}\mathbf{x} + \mathbf{b})_{[i]}$$

The file `loglinear.py` has skeleton code for a log-linear classifier. Complete the needed code in that file.

The file contains some sanity checks that can be run by running the file. It is advised that you understand what they are doing. These checks are not comprehensive, and you can add more.

Note: numpy arrays are **not** matrices, so multiplication does element-wise multiplication rather than matrix-multiplication. To get matrix multiplication behavior, you can use `np.dot(x,y)`.

3 Training a log-linear classifier with SGD – 20 points

We are now in place to train our classifier using stochastic gradient descent. The file `train_loglin.py` has skeleton code, which you should complete.

We will be training a classifier on the language identification task. We will be using bigram features, as we saw in class.

The `data.tgz` file contains train, dev (validation) and test datasets for the language identification task. The file format is one training example per line, in the format:

```
language<TAB>text
```

The file `utils.py` contains some helper code for reading these files, extracting letter bigrams, and some more. You may use this code if you find it helpful, but are not required to.

The skeleton code in `train_loglin.py` is setup to print the train-set loss, as well as the train-set and dev-set accuracies in each iteration through the data. A correct implementation should easily achieve a dev-set accuracy of above 80%.

4 Predicting on Test Data – 10 points

The supplied data contains also a blind test set. It is in the same format as the train and dev sets, but the labels are hidden (replaced with '?'). Write code to compute predictions on the test data, and write them to a file named `test.pred`. The file should be in the following format: a file with the exact same number of lines as in the test file (300). Each line should contain a single 2-letters language ID. For example, if the test file had 5 lines, the following is a valid prediction file:

```
en
it
en
fr
fr
```

You will be submitting this prediction file. You will be evaluated on the quality of the prediction, so use your best model (it can also be a model from a later section).

5 Adding a hidden layer – 20 points

The log-linear model is nice, but cannot solve the XOR problem.

We now move on to implementing a non-linear classifier: multi-layer perceptron with a single hidden layer:

$$f(\mathbf{x}) = \text{softmax}(\mathbf{U}\,tanh(\mathbf{W}\mathbf{x} + \mathbf{b}) + \mathbf{b}')$$

Based on the implementations in `loglinear.py` and `train_loglin.py`, implement `mlp1.py` (skeleton provided) and `train_mlp1.py` (not provided, but follow the same interface as in `train_loglin.py`). Note that the only change in the function signatures is the `create_classifier` function in `mlp1.py`, which now accepts also the dimension of the hidden-layer.

You are advised but not required to use the gradient checker to verify your gradient calculations.

1. Can you get better accuracies on the language identification task using the multi-layer perceptron?
2. Switch the feature set of the language identification from letter-bigrams to letter-unigrams (single letters). What's the best you can do with the log-linear model with these features? What's the best you can do with the MLP?
3. Verify that your MLP can learn the XOR function (you can see a training-set for XOR in the file `xor_data.py`). How many iterations does it take to correctly solve xor?

Write your answers to the above questions in a file named `answers.txt`. It should be a plain-text (ascii only) file, written in English.

6 Arbitrary MLPs – 20 points

Implement an arbitrary-depth MLP. Your code should reside the file `mlpn.py` and follow the signatures within it. In particular, the `create_classifier` function now accepts a single list, specifying the dimensions of all layers. That is, `create_classifier([20,30,40,10])` should return the parameters for a network accepting a 20-dim input, passing it to a 30-dim layer, then to a 40-dim layer, and finally to a 10-dim output.

Your implementation of classifier output and loss and gradient calculation should likewise be able to handle an arbitrary number of layers.

You are advised but not required to use the gradient checker to verify your gradient calculations.