Deep Learning Workshop - Torch

July 10, 2016

Exercise 1 - visualizing data

We provide a dataset named "fourclass" downloaded from the libsvm web site [1] which consists in 862 2-dimensional datapoints.

- Execute `visualize_data.lua` to load and visualize the data using gnuplot.
- Modify the code to plot the data separately for each class.

Exercise 2 - implementing a new layer

We will now consider a simple linear model that given an input $\boldsymbol{x} \in \mathbf{R}^d$ (where $d = 2$ in this example) consists in computing the decision function

$$f(\boldsymbol{x}) = \sigma(W\boldsymbol{x} + b), \tag{1}$$

where W is a matrix of weights and $\sigma(.)$ is a softmax function that squashes real values to the range (0, 1).

We defined a function called `create_model.lua` that creates and returns the model and the criterion objects. The model that computes the function defined in Eq. 1 consists in the following layers:

$$\text{input} \rightarrow \text{linear} \rightarrow \text{softmax} \rightarrow \text{cross-entropy loss}$$

Train the model First train the model by running `main.lua`. The script will periodically output the loss in the terminal.

The file we provided uses stochastic gradient descent to optimize the weights of the network. If you are curious, you can modify the code to use a different optimizer, e.g. Adam.

Implement a new layer We will now replace the sigmoid layer of the previous model by a softsign function:

$$f(\boldsymbol{x}) = \frac{\boldsymbol{x}}{1 + |\boldsymbol{x}|},$$

whose derivative is

$$f'(\boldsymbol{x}) = \frac{1}{(1 + |\boldsymbol{x}|)^2}.$$

In order to implement a new module, you have to override the following methods in `softsign.lua`:

- *updateOutput* to implement the forward pass, to compute z from x
- *updateGradInput* to implement part of the backward pass, to compute the derivative of the loss wrt your layers inputs ($\frac{\partial L}{\partial \boldsymbol{x}}$, in terms of the derivative of the loss wrt your layers outputs ($\frac{\partial L}{\partial \boldsymbol{z}}$).

$$\frac{\partial L}{\partial \boldsymbol{x}} = \frac{\partial L}{\partial \boldsymbol{z}} \cdot \frac{\partial \boldsymbol{z}}{\partial \boldsymbol{x}}$$

This is also illustrated in Figure 1.

Construct a more complex network We will be constructing the following network:

$$\text{input} \rightarrow \text{linear} \rightarrow \text{sigmoid} \rightarrow \text{linear} \rightarrow \text{sigmoid} \rightarrow \text{softmax} \rightarrow \text{cross-entropy loss}$$

[1] `https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/`

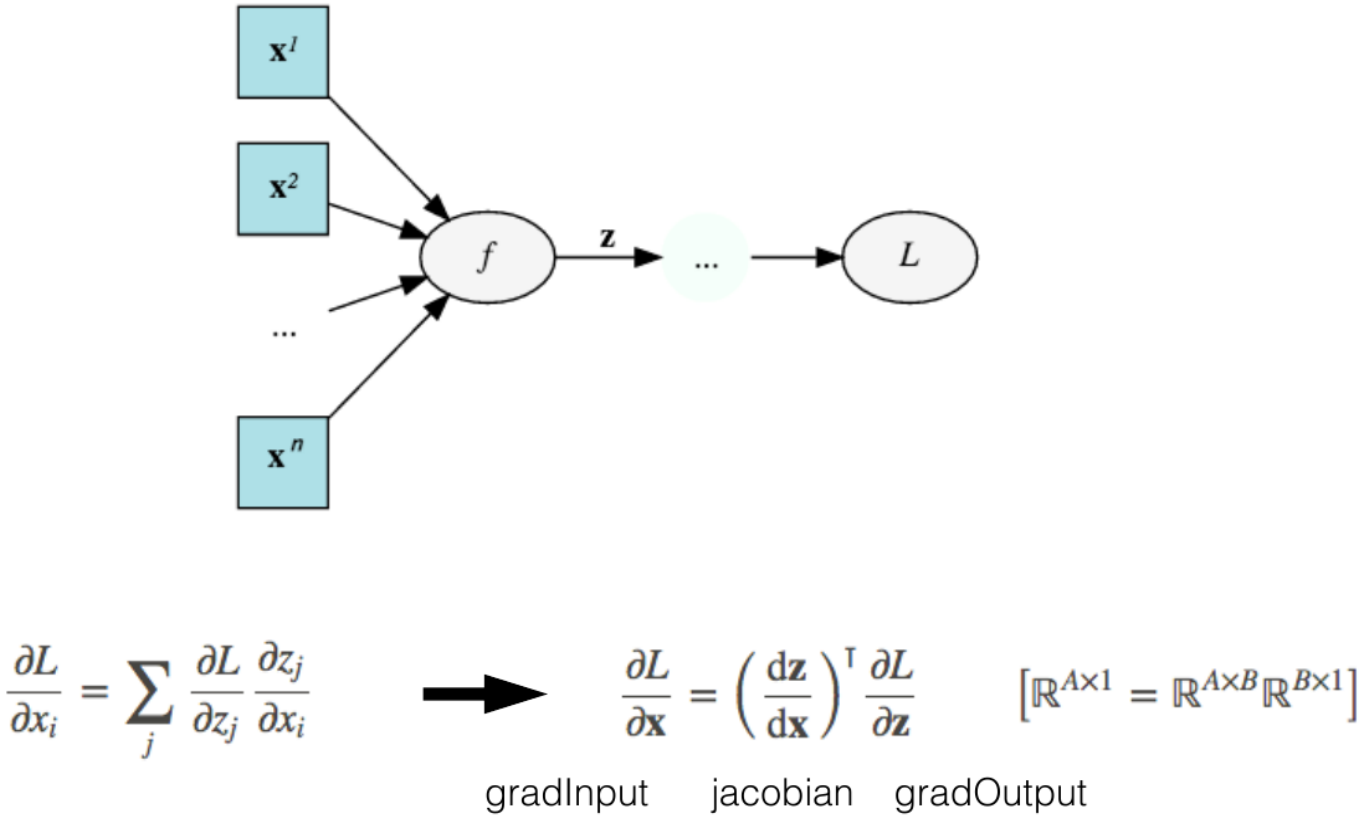

Figure 1: Illustration from `http://deepdish.io/2015/11/21/building-blocks-of-deep-learning/`

Exercise 3 - visualize the layers on MNIST

Dataset We now turn to the classification of handwritten digits from the MNIST dataset that contains 60,000 training images and 10,000 testing images. We will only be using a subset of the data in this exercise although you can use the full set by passing the argument "–full".

CNN We provide a function `train-on-mnist.lua` that implements a neural network with two convolutional layers, each followed by a pooling and a non-linearity layer. The code will download the data from the Internet unless the data is already present in a directory "mnist.t7". Familiarize yourself with the code and run it with the default parameters. This will run the code for 1 epoch and outputs the training scores in a directory named "logs".

Your task

- Visualize the weights of the first layer of the neural network after training. You can e.g. retrieve the weights of the second filter in the first layer using the following instruction:

  ```
  model:get(1).weight[2]
  ```

- The training function we provided samples one datapoint at each iteration. Instead we would like to take a mini-batch. This is a common practice that consists in sampling a small set of datapoints which reduces the variance of the estimated gradient. This operation can usually be parallelized and thus made relatively cheap. Modify the train function to compute a mini-batch.

- Add an ℓ_2 penalty on the norm of the weight vectors by modifying the feval function at the marked place in the code. Recall that you also have to modify the gradient accordingly.

- Optional: When you implement a new layer, one potential source of error is the computation of the gradients. One common practice is to check that the gradients are being correctly computed using finite difference, i.e. using the formula

$$\frac{\partial f}{\partial x_i} = \frac{f(x_1, \ldots, x_i + \epsilon, \ldots, x_n) - f(x_1, \ldots, x_i, \ldots, x_n)}{\epsilon} \tag{2}$$

  Use the softsign function you implemented in the previous exercise and make sure that the gradient is correct using finite difference.

Exercise 4 - sentiment classification

We will now implement a model similar to Kim Yoon's Convolutional Neural Networks for Sentence Classification [**?**].

Model The model is shown in Figure 2. The input matrix shown on the left contains n row vectors each representing a k-dimension embedding of a word. The first layer embeds words into low-dimensional vectors. The next layer performs convolutions over the embedded word vectors using multiple filter sizes. For example, sliding over 3, 4 or 5 words at a time. Next, we max-pool the result of the convolutional layer into a long feature vector, add dropout regularization, and classify the result using a softmax layer.

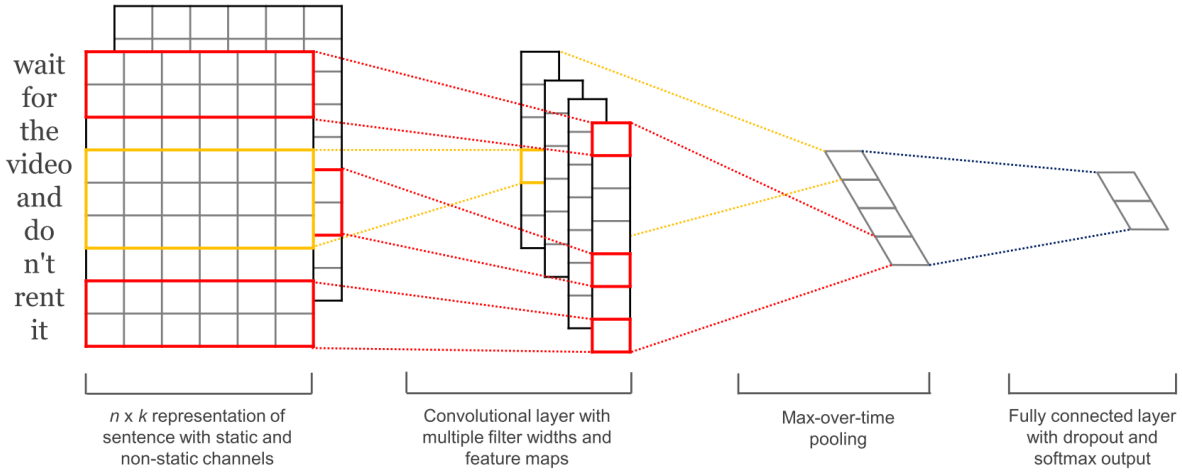

Figure 2: CNN model for sentiment classification

Initialization It is common practice to initialize the input matrix with precomputed word vectors, either word2vec or Glove [2]. We will be using the later in this exercise.

Dataset The dataset is accessible from

`https://inclass.kaggle.com/c/cls-text-classification/data`.

It consists in a set of annotated tweets `train_pos.txt`, `train_neg.txt` and `test_data.txt`. The format of each file is a simple text file with one tweet per line. The file `test_data.txt` contains 10'000 tweets without any labels, each line numbered by the tweet-id.

All the tweets used to contain a smiley, :) for positive and :(for negative which was used to assign a label. Note that all the smileys (labels) have been removed. All tweets have already been pre-processed so that all words (tokens) are separated by a single whitespace.

The Glove vectors for all words contained in this dataset are provided on the web site. We also provide an example of a submission file to be submitted to Kaggle `sampleSubmission.csv`.

Your task Predict if a tweet message used to contain a positive :) or negative :(smiley, by considering only the remaining text. More specifically you are asked to

1. **Load the word embeddings:** We provide a reader to load the Glove vectors downloaded from the Kaggle web site, as well as a simple example that shows how to use the reader (see `test_reader.lua`).
2. **Load the tweets:** Load the training tweets given in `train_pos_full.txt` and `train_neg_full.txt` (or a suitable subset depending on RAM requirements), and construct a vocabulary list of words appearing at least 5 times.
3. **Implement the model shown in Figure 2:** Using the word embeddings as input, implement a CNN with a convolutional and pooling layer, followed by a fully connected layer and a softmax layer.

[2] `http://nlp.stanford.edu/projects/glove/`

4. **Train your CNN**

5. **Prediction:** Predict labels for all tweets in the test set.

6. **Submission / Evaluation:** Submit your predictions to Kaggle to get a misclassification error score. Recall that your submission file for the 10'000 tweets must be of the form `<tweet-id>, <prediction>`, see `sampleSubmission.csv`. Try to tune your system for best evaluation score. You can also use a local separate validation set to get faster feedback on the accuracy of your system.