COMPGI13/COMPM050 (Advanced Topics in ML): Assignment #1

Due on Tuesday, January 31, 2017

Thore Graepel, Koray Kavukcuoglu

Setup and Logistics

Submissions

- **Due date**: **07 March 2017, 11:55** (Start date: 10th Jan 2017)
- Submit .zip via Moodle containing: **report, code, readme** (with instructions how to run the code), **trained weights** (final weights only).
- (Optional, but encouraged) If you want to use **github** to manage your code, please send the **TA (d.borsa@cs.ucl.ac.uk) your github account** and we will provide an invitation to a private repo.
- Individual assignment.

Install TensorFlow (recommended: $\geq$ r0.08)

Follow the instructions at www.tensorflow.org/get_started/os_setup. Test installation to avoid problems. For Windows machines, we recommend using a virtual environment if you encounter problems with the installation or linking.

MNIST Dataset

In this assignment we will be using the MNIST digit dataset (http://yann.lecun.com/exdb/mnist/). The dataset contains images of hand-written digits $(0-9)$, 28×28 pixels and the corresponding labels. You can use the tensorflow build-in functionality to download and import the dataset into python.

```
from tensorflow.examples.tutorials.mnist import input_data

# import dataset with one-hot class encoding
mnist = input_data.read_data_sets(data_dir, one_hot=True)
```

Whether you choose the above, or downloading the datasets from the website, please make sure you have the following: **training set**: 55K samples, **testing set**: 10k samples.

As a reference here is a list of error rates obtained on MNIST.

P1: MNIST with TensorFlow (50pts)

In this part, you will have to implement several neural network models to classify MNIST digits, using TensorFlow:

(a) 1 linear layer, followed by a softmax [20 pts]
(input → linear layer → softmax → class probabilities)

(b) 1 hidden layer (128 units) with a ReLU non-linearity, followed by a softmax [10 pts]
(input → non-linear layer → linear layer → softmax → class probabilities)

(c) 2 hidden layers (256 units) each, with ReLU non-linearity, follow by a softmax [10 pts]
(input → non-linear layer → non-linear layer → linear layer → softmax → class probabilities)

(d) 3 layer convolutional model (2 convolutional layers followed by max pooling) + 1 non-linear layer (256 units), followed by softmax. [10 pts]
(input(28x28) → conv(3x3x16) + maxpool(2x2) → conv(3x3x16) + maxpool(2x2) → flatten → non-linear → linear layer → softmax → class probabilities)

Use *padding = 'SAME'* for both the convolution and the max pooling layers. Employ plain convolution (no stride) and for max pooling operations use 2x2 sliding windows, with no overlapping pixels (note: this operation will down-sample the input image by 2x2).

All models, should be trained using a **cross-entropy loss** function:

$$loss = -\sum_{i=1}^{N} \log p(y_i|x_i,\theta) = -\sum_{i=1}^{N} \log \underbrace{\left(\frac{\exp(z_i[y_i])}{\sum_{c=1}^{10}\exp(z_i[c])}\right)}_{\text{softmax output}} = \sum_{i=1}^{N}\left(-z_i[y_i] + \log\left(\sum_{c=1}^{10}\exp(z_i[c])\right)\right)$$

where $z \in \mathbb{R}^{10}$ is the input to the softmax layer and $z[c]$ denotes the c-th entry of vector z. And i is a index for the dataset $\{(x_i, y_i)\}_{i=1}^{N}$.
Use **stochastic gradient descent** (SGD) for optimizing this objective.
Note: Performance will vary considerably depending on the learning rate used and on the model, so please allow time to run multiple experiments and properly tune this hyperparameter.

Hint: If you need some extra help, familiarizing yourselves with the dataset and the task of building models in TensorFlow, you can check the TF tutorial for MNIST. This tutorial will walk you through the MNIST classification task step-by-step, building and optimizing a model in TensorFlow. (Please do not copy the provided code, though. Walk through the tutorial, but write your own implementation).

[Extra (0pts)] Feel free to experiment with other optimization procedures and adapting learning rates. Here you can find the documentation for several optimizers already implemented in TensorFlow, as well as the original papers proposing these methods.

P2: MNIST without TensorFlow (50pts)

In this part, you will need to come up with your own implementation of the models in P1: (a)-(d). You are **not allowed to use TensorFlow** for anything else, but importing the data. (By default, modules you can use: `numpy, random, sys` + anything for visualization/serializing. **Ask** should you require anything else.) Keep in mind, the purpose of this exercise is to implement and optimize your own neural networks architectures without the toolbox/library tailored for doing this.

P2: MNIST without TensorFlow (50pts) continued on next page...

This also means, in order to train and evaluate your models, you will need to implement your **own optimization procedure**. You are to use the same **cross-entropy loss** as above for optimization and your own implementation of SGD or any other gradient based method you wish.

Hint: Remind yourselves of the chain rule and read through the lecture notes on back-propagation (computing the gradients by recursively applying the chain rule). This is a general procedure that applies to all model architectures you will have to code in the following steps. Thus, you are strongly encourage to implement an optimization procedure that generalizes and can be re-used to train all your models. Recall the only things that you will need for each layer are: (i) the ***gradients of layer with respect to its input****, and (ii) the* ***gradients with respect to its parameters****, if any.*

(a) Compute the following derivatives:

(i) Derivative of the loss function wrt. the scores z: $\frac{\partial loss}{\partial z}$ [2 pts]

(ii) Given the model in (P1:a), compute the derivative of the loss wrt input x, derivative of the loss with to the layer's parameters W, b. [3 pts]

(iii) Compute the derivative of a convolution layer wrt. to its parameters W and wrt. to its input (4-dim tensor). [5 pts]

Express these derivates in vector/matrix/tensor format whenever possible. This will help you in implementing the update operations more efficiently. As in `MatLab`, most matrix operations are highly optimized in `numpy` via `BLAST`, so do take advantage of this whenever possible.

(b) Implement and train the model in (P1:a) [10 pts]

(c) Implement and train the model in (P1:b) [10 pts]

(d) Implement and train the model in (P1:c) [10 pts]

(e) Implement and train the model in (P1:d) [10 pts]
Hint: Naive implementations of the convolution and max-pooling layers will be very slow, so after checking that your implementation is doing the right thing, try to optimize by relying mostly on matrix/tensor operations as recommended earlier. Methods you might find useful: `np.tensordot`, `np.pad`, `np.reshape`.

Final notes:
1) It's always a good idea to check your implementations of forward passes and gradients, layer by layer to avoid compounding errors. You can use the TF counterparts for that.
2) Expect the experiments in the second part (your implementations) to take considerably longer to train. Budget your time accordingly & start early!
3) Have fun & experiment, experiment, experiment!

Reporting results

For **both parts** and all experiments, please report:

- for each experiment **training and testing errors** (throughout training),
- as well as their values at the end of the optimization (complete the table bellow)

Experiment	P1:a	P1:b	P1:c	P1:d	P2:b	P2:c	P2:d
Training Error Rate							
Testing Error Rate							

- for each experiment provide the **confusion matrix** for all classes.

Please **save the trained models** for all experiments and provide code to load the models with these saved parameters. Note the misclassification errors for these saved models should match the ones you reported in the table above.