Machine Learning - Exercise 5

MNIST database
Installing Theano and Lasagne

- You need first to install locally theano and lasagne in order to run the network used in this exercise.

- First, create a local directory where the libraries will be installed:

```bash
cmdir -p $HOME/.local/lib/python2.7/site-packages
```

- Add this directory to your Python path. Edit your $HOME/.bashrc config file (with emacs/vi/nano/whatever) and add the following line at the bottom:

```bash
export PYTHONPATH=$PYTHONPATH:$HOME/.local/lib/python2.7/site-packages
```

- Close the terminal and re-open it. Install theano and lasagne using easy-install:

```bash
easy-install --prefix=$HOME/.local theano
easy-install --prefix=$HOME/.local lasagne
```

- Test that it worked by typing in a Python interpreter:

```python
import lasagne
```

- If anything goes wrong, call me!
Understanding lasagne

- lasagne is a wrapper around theano, a tensor library allowing to perform computations both on CPU and GPU.
- It allows to define easily feedforward neural networks, including multi-layer perceptrons.

1. Read the doc at lasagne.readthedocs.org to understand what it does and how to define a neural network layer by layer. Focus on the MNIST tutorial http://lasagne.readthedocs.org/en/latest/user/tutorial.html, especially on the Multi-Layer Perceptron (MLP) part.
MNIST database

- MNIST is the simplest image recognition dataset, created by Yann LeCun to benchmark supervised learning algorithms. State-of-the-art is at 99.7% accuracy on the test set (convolutional deep networks).

http://yann.lecun.com/exdb/mnist/

- Each input is a 28*28 grayscale image representing digits between 0 and 9.
- The training set has 50,000 examples, the validation set 10,000 and the test set 10,000.
Simple MLP for MNIST

- The default MLP to learn the MNIST dataset is provided in the script `MLP.py`.

1. Run the script (may take a while) and read it to understand what it does (see the next slides, skip the part about drop_in and drop_out for now).

2. At the end of the script (after 50 epochs of learning), you will see two windows.
   - One shows the evolution of the training and validation accuracy over time. How do they evolve? Which one is higher and why?
   - The other shows 12 examples of digits in the test set that were misclassified. Are some of these errors acceptable?
Simple MLP for MNIST

- The MLP used in the script has a single hidden layer of 50 neurons, using the sigmoid/logistic transfer function.

- The learning method is Stochastic Gradient Descent (SGD), i.e. the backpropagation algorithm seen in the course but applied on minibatches of 500 examples.

- Two main differences with what you already saw:
  - The 10 output neurons use the softmax transfer function (see next slide).
  - The loss function (or objective function) is not the quadratic error function as previously, but the cross-entropy error function.
A softmax layer interprets the activity of each output neuron as the probability that the output of the network is one of the 10 digits.

The net activation of the output neurons has to be normalized so their sum is exactly one.

The prediction $\mathbf{o}$ of the network therefore varies between two presentations of the same input.

This is similar to the probabilistic interpretation in logistic regression.

$$o_k = P(o = k|x) = \frac{\text{net}_k}{\sum_{l=1}^{10} \text{net}_l}$$
Cross-entropy error function

- The target vector for the digit 3 is for example:

\[ t = [0, 0, 0, 1, 0, 0, 0, 0, 0, 0] \]

- Instead of minimizing the quadratic error/loss function as before:

\[ L = \frac{1}{2} (t - o)^2 \]

we minimize the categorical cross-entropy loss function (aka logistic loss or negative log-likelihood):

\[ L = - \sum_{k=1}^{10} t_k \cdot \log(o_k) \]

- This function is positive and has its minimum when the predictions are correct.

- This only changes the first step of the backpropagation algorithm, but converges better for softmax output units.
Goal of the exercise

- The goal is to find a neural network with 98% test accuracy after 50 epochs on MNIST by varying multiple parameters and methods.

- Within your experiments, you will particularly pay attention to the cases where the training accuracy becomes higher than the validation one. What happens here? Is it worth waiting longer?

- Keep in mind that time is limited: a deep network with 5 hidden layers (784-2500-2000-1500-1000-500-10) would obtain an accuracy of 99.65%, but computing a single epoch would take hours on your PC. Check the neural nets section at http://yann.lecun.com/exdb/mnist/ (not convolutional nets!).

- Quit with Ctrl+c if you think there will be no improvement.
Things to explore

1. Change the learning rate (e.g. 0.01, 0.05, 0.1, 0.5).

2. Change the number of neurons in the hidden layer (hidden_layers = [100]).


   
   transfer_function = lasagne.nonlinearities.rectify

Note: if you use ReLU units, you should change the initialization of the weights to :

   network = lasagne.layers.DenseLayer(
       incoming=network,
       num_units=nb_neurons,
       nonlinearity=transfer_function,
       W=lasagne.init.GlorotUniform(gain='relu'))
Things to explore

5. Change the learning rule. Instead of the regular SGD, use for example the Nesterov Momentum method:

   ```python
   updates = lasagne.updates.nesterov_momentum(train_loss, params,
                                             learning_rate=learning_rate, momentum=0.9)
   ```

6. Or the AdaDelta learning rule:

   ```python
   updates = lasagne.updates.adadelta(train_loss, params,
                                      learning_rate=1.0, rho=0.95, epsilon=1e-06)
   ```

   Note that AdaDelta has no learning rate, so let it to 1.0.

7. Or any of the update rules available in Lasagne...

   ```python
   ```

8. Comment on these update rules and overfitting by comparing the training and validation accuracies.
Things to explore

9. Apply L2- or L1-regularization to the weight updates to avoid overfitting
   http://lasagne.readthedocs.org/en/latest/modules/regularization.html:

   ```
   C = 0.0001
   train_loss += C * lasagne.regularization.regularize_network_params(
                   network, lasagne.regularization.l2)
   ```

10. Apply dropout regularization on each layer, including the input layer

   `drop_in` defines the percentage of input neurons which will be randomly dropped; `drop_out` is for the
   hidden neurons. `drop_in` is usually smaller than `drop_out`:

   ```
   # Dropout in the input layer
   drop_in = 0.1

   # Dropout in the hidden layers
   drop_out = 0.5
   ```