Classification and Representation Learning

Tutorial 3 : MNIST and Deep Learning

Hoel Le Capitaine

Academic year 2017-2018

Setting up ...

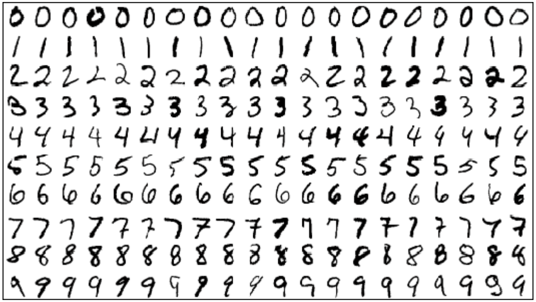
- you first need to install `theano` and `lasagne` in order to run (deep) networks
- you can do so by using `pip`, `easy-install` or any python package manager
- test your installation by using `import lasagne`

What is 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 multilayer perceptrons.
- 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 MultiLayer Perceptron (MLP) part.

MNIST data

- MNIST is the simplest image recognition dataset, created by Yann LeCun to benchmark supervised learning algorithms. Stateof the art is at 99.7% accuracy on the test set (convolutional deep networks).
- 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
2. At the end of training, plot the validation accuracy over time (epoch). How this accuracy evolves?

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 crossentropy error function.

Softmax layer

- 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 of the network therefore varies between two presentations of the same input.
- This is similar to the probabilistic interpretation in logistic regression

Cross-entropy

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

$$L = \frac{1}{2}(\mathbf{y} - \mathbf{o})^2$$

we minimize the categorical crossentropy loss function (aka logistic loss or negative loglikelihood):

$$L = -\sum_{k=1}^{10} y_k \log(o_k)$$

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.

Exercise

- The goal is to find a neural network with 98% test accuracy after 50 epochs on MNIST by varing 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 .
3. Add more hiddenlayers (e.g. create 3 hidden layers having resp. 100, 50 and 25 neurons.)
4. Change the transfer function of the hidden neurons. See `http://lasagne.readthedocs.org/en/latest/modules/nonlinearities.html` for the different possibilities in lasagne. Check in particular the Rectifier Linear Unit (ReLU):

   ```
   transfer_function = lasagne.nonlinearities.rectify
   ```

   Note: if you use ReLU units, you should change the initiliation 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

1. Change the learning rule. Instead of the regular SGD, use for example the Nesterov Momentum method (`http://lasagne.readthedocs.org/en/latest/modules/updates.html#lasagne.updates.nesterov_momentum`):
   ```
   updates = lasagne.updates.nesterov_momentum(train_loss,
   params, learning_rate=learning_rate, momentum=0.9)
   ```
2. Or the AdaDelta learning rule `http://lasagne.readthedocs.org/en/latest/modules/updates.html#lasagne.updates.adadelta`:
   ```
   updates = lasagne.updates.adadelta(train_loss, params,
   learning_rate=1.0, rho=0.95, epsilon=1e06)
   ```
   Note that AdaDelta has no learning rate, so let it to 1.0.
3. Or any of the update rules available in Lasagne...
   `http://lasagne.readthedocs.org/en/latest/modules/updates.html`

Things to explore

1. Apply L2 or L1regularization 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)
```

2. Apply dropout regularization on each layer, including the input layer http://lasagne.readthedocs.org/en/latest/modules/layers/noise.html#lasagne.layers.DropoutLayer.
   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
```

3. When you are done, save your model as a npz-file (as explained in the **mlp.py** file), and send it to me (mailto:hoel.lecapitaine@ls2n.fr) by mail before 5th, November