Machine Learning - Exercise 5

MNIST database

Installing Theano and Keras

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

```
pip2 install Theano --user

pip2 install keras --user
```

- Then tell `keras` to use `Theano` as a backend. Open `~/.keras/keras.json` (warning: hidden file!) with a text editor and modify it to:

```
{
    "image_dim_ordering": "th",
    "floatx": "float32",
    "epsilon": 1e-07,
    "backend": "theano"
}
```

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

```
>>> import keras
Using Theano backend.
```

Understanding Keras

- `keras` 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 http://keras.io to understand what it does and how to define a neural network layer by layer.

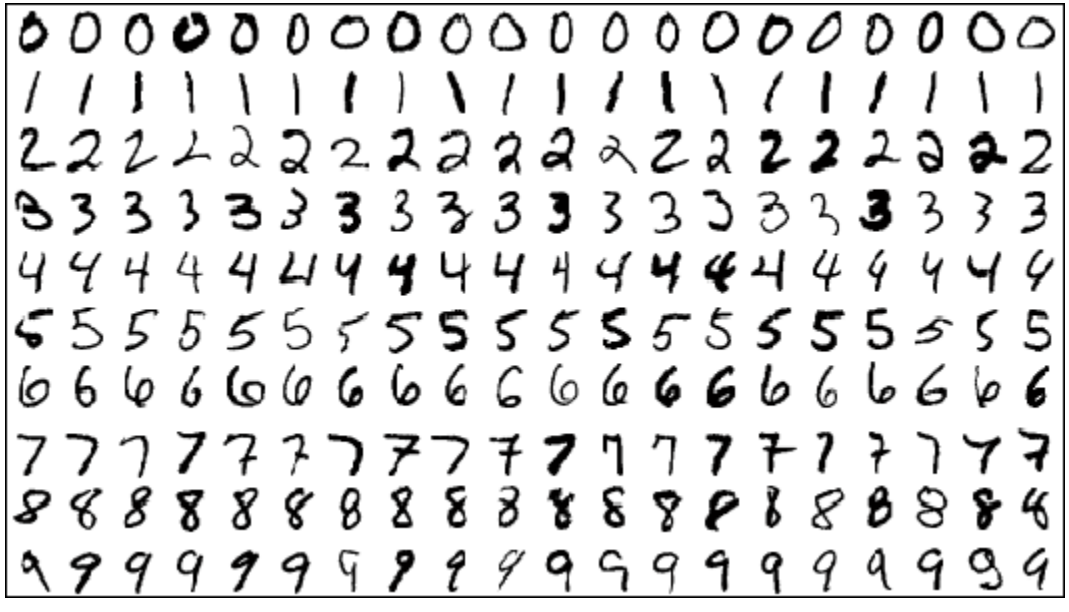
Focus on the MNIST tutorial https://keras.io/getting-started/sequential-model-guide/, especially on the Multi-Layer Perceptron (MLP) part.

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 60.000 examples, 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).

2. At the end of the script (after 20 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 scripts starts with the usual imports:

```
# Usual imports
from __future__ import print_function
import numpy as np
import matplotlib
matplotlib.use('TKAgg')
import matplotlib.pyplot as plt
```

as well as all the relevant Keras objects:

```
# Import Keras objects
from keras.datasets import mnist
from keras.models import Sequential
from keras.layers import Dense, Dropout, Activation, BatchNormalization
from keras.optimizers import SGD, Adam, RMSprop, Adadelta
from keras.regularizers import l2, l1
from keras.callbacks import History
from keras.utils import np_utils
```

Simple MLP for MNIST

We then load the MNIST dataset by using the built-in method of Keras:

```
# Load the data, shuffled and split between train and test sets
(X_train, y_train), (X_test, y_test) = mnist.load_data()
```

We reshape the data and remove the mean:

```
# Reshape and normalize the data
X_train = X_train.reshape(X_train.shape[0], 784).astype('float32') / 255.
X_test = X_test.reshape(X_test.shape[0], 784).astype('float32') / 255.
X_mean = np.mean(X_train, axis=0)
X_train -= X_mean
X_test -= X_mean
print(X_train.shape[0], 'train samples')
print(X_test.shape[0], 'test samples')
```

and transform the labels (0, 1, 2, 3, ...) into binary vectors (0000000001, 0000000010, ...)

```
# Convert class vectors to binary class matrices
Y_train = np_utils.to_categorical(y_train, nb_classes)
Y_test = np_utils.to_categorical(y_test, nb_classes)
```

Simple MLP for MNIST

- The MLP used in the script has a single hidden layer of 100 neurons, using the sigmoid/logistic transfer function.
- The learning method is Stochastic Gradient Descent (SGD).

```
def create_mlp():
    model = Sequential()

    # Hidden layer with 100 neurons, taking inputs from the 784 input pixels
    model.add(Dense(100, input_shape=(784,))) # Weights
    model.add(Activation('sigmoid')) # Transfer function

    # Softmax output layer
    model.add(Dense(nb_classes))
    model.add(Activation('softmax'))

    # Learning rule
    optimizer = SGD(lr=0.01)

    # Loss function
    model.compile(
        loss='categorical_crossentropy', # loss
        optimizer=optimizer, # learning rule
        metrics=['accuracy'] # show accuracy
    )
    return model
```

Simple MLP for MNIST

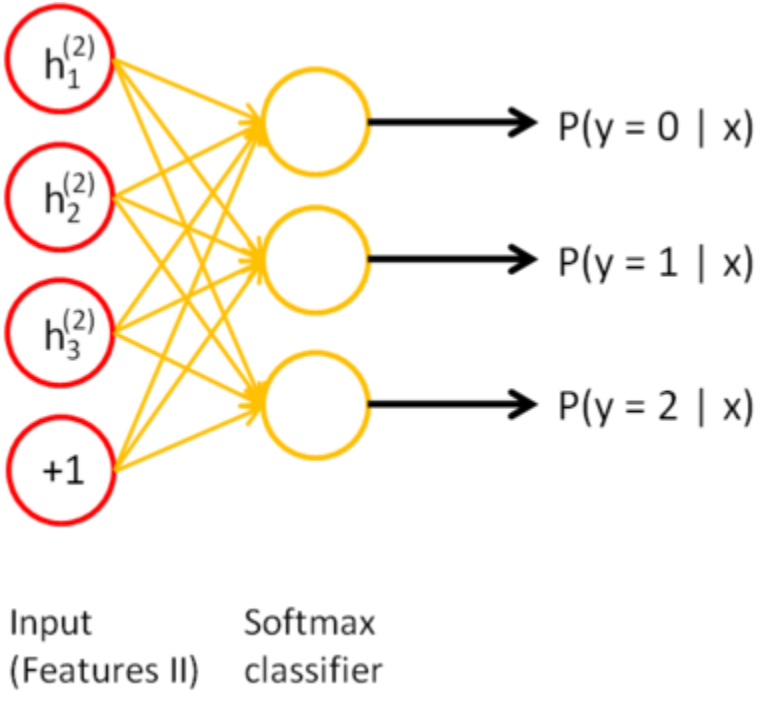
- A neural network is simply defined by **adding** functions to a `Sequential` model (directed acyclic graph).
- The model must be **compiled** at the end in order to be functional.
- Two main differences with what you already did:
  - The 10 output neurons use the softmax transfer function.

```
model.add(Activation('softmax'))
```

  - The loss function (or objective function) is not the quadratic error function as in the course, but the categorical cross-entropy.

```
model.compile(
    loss='categorical_crossentropy', # loss
)
```

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 $\mathbf{o}$ of the network therefore varies between two presentations of the same input.
- This is similar to the probabilistic interpretation in logistic regression.

$$o_k = P(\mathbf{o} = k|\mathbf{x}) = \frac{\text{net}_k}{\sum_{l=1}^{10} \text{net}_l}$$

Cross-entropy error function

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

$$\mathbf{t} = [0, 0, 0, 1, 0, 0, 0, 0, 0, 0]$$

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

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

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.

Simple MLP for MNIST

The model is simply created by calling the method. You can call `summary()` to visualize the different layers created and the number of free parameters:

```
# Create the model
model = create_mlp()
# Print a summary of the network
model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_1 (Dense)              (None, 100)               78500
_________________________________________________________________
activation_1 (Activation)    (None, 100)               0
_________________________________________________________________
dense_2 (Dense)              (None, 10)                1010
_________________________________________________________________
activation_2 (Activation)    (None, 10)                0
=================================================================
Total params: 79510
```

Simple MLP for MNIST

Training the network on the data is done by calling the `fit()` method on the model:

```
# Train for 20 epochs using minibatches
history = History()
try:
    model.fit(X_train, Y_train,
        batch_size=50,
        nb_epoch=20,
        validation_split=0.1,
        callbacks=[history])

except KeyboardInterrupt:
    pass
```

Here we use:

- minibatches of 50 samples.
- a maximum number of 20 epochs.
- 10 % of the training set is used for validation.
- a `History` callback to record the evolution of the training/validation loss and accuracy.

Simple MLP for MNIST

The model can be tested after training by calling `evaluate()` on the test set:

```
score = model.evaluate(X_test, Y_test, verbose=0)
print('\nTest loss:', score[0])
print('Test accuracy:', score[1])
```

After that is only the code to display the two windows.

Goal of the exercise

- The goal is to find a neural network with 98% test accuracy after 20 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 yet!).
- Escape the learning procedure with `Ctrl+c` if you think there will be no improvement anymore, the test accuracy will be computed anyway.

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 hidden layers, for example three hidden layers having resp. 100, 50 and 25 neurons.
4. Change the transfer function of the hidden neurons. See https://keras.io/activations/ for the different possibilities in `keras`. Check in particular the Rectifier Linear Unit (ReLU):

```
model.add(Activation('relu'))
```

Things to explore

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

```
optimizer = SGD(lr=0.1, decay=1e-6, momentum=0.9, nesterov=True)
```

6. Or the AdaDelta learning rule:

```
optimizer = Adadelta(lr=1.0, rho=0.95, epsilon=1e-08, decay=0.0)
```

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

7. Or any of the update rules available in Keras... https://keras.io/optimizers/

Things to explore

8. Apply **L2- or L1-regularization** to the weight updates to avoid overfitting https://keras.io/regularizers/:

```
C = 0.0001
model.add(Dense(50, W_regularizer=l2(C)))
```

9. Apply **dropout** regularization on each layer.

```
model.add(Dropout(0.2))
```

10. Add **Batch normalization** after each transfer function:

```
model.add(Dense(100, input_shape=(784,))) # Weights
model.add(Activation('relu')) # Transfer function
model.add(BatchNormalization()) # Batch normalization
```