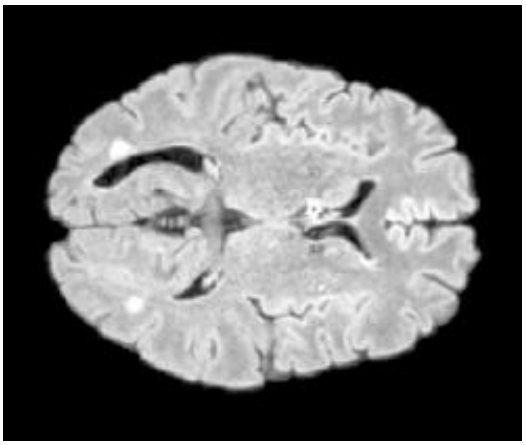
Classification of brain MRI voxels

Introduction

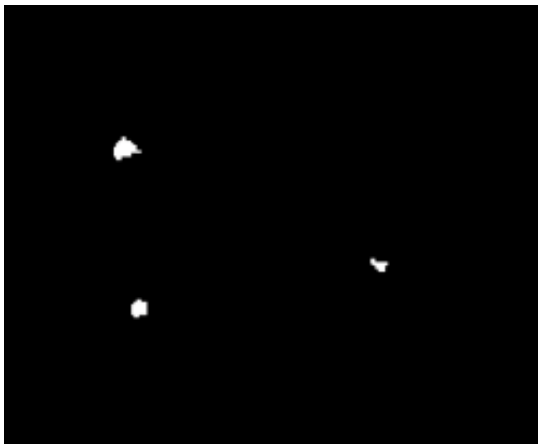
In this exercise, you will implement the backpropagation algorithm for fully connected neural network and apply it to the task of brain MRI voxels classification, when voxel can belong or not to multiple sclerosis lesion.

Multiple Sclerosis is one of the most common non-traumatic neurological diseases in young adults. It is a chronic inflammatory disease in which the immune system attacks the central nervous system and damages myelin, myelin producing cells and underlying nerve fibers.

Multiple sclerosis lesions appear as hyper-intense regions in FLAIR MRI modality images:



FLAIR MRI modality



Lesions segmentation mask

Dataset

Dataset for this exercise contains patches of 32 x 32 pixels. These patches are extracted from the axial view of FLAIR MRI modality. Patch considered to be positive if the central pixel (x=16, y=16) belongs to multiple sclerosis lesion, and negative if the central pixel belongs to the healthy region of the brain. As you can see, the dataset is noisy, because of not precise segmentations (welcome to the real world). The dataset is already divided to train and validation.

Tasks

You will implement fully connected neural network that has 3 layers: an input layer, a hidden layer and an output layer. Input vector consists of pixel values of patch. Since the patches are of size 32 x 32, this gives us 1024 input layer units. **Output layer is a single neuron** and returns probability of the patch to be positive. You are not restricted on a number of neurons in the hidden layer.

Note: the project is a competition. You should be able to receive at least 89% accuracy. The grades will be determined on the basis of your algorithm performances,

comparing to the other students on the test set (which you don't get). The evaluation metric is the mean accuracy of your prediction on the test set. You will implement the network from scratch in Python, **Deep Learning libraries are not allowed**.

Implementation guide:

- Initialize the weights and biases, for example using a normal distribution with mean = 0 and std = 1.
- Do the following for the number of epochs:
  - Sample a random mini-batch (choose the appropriate size of mini-batch) from the training set. Prepare input vectors (pixels of patch) and labels (0 - negative patch, 1 - positive patch).
  - Forward propagate the input vectors of mini-batch through the network (use matrix / vector multiplication whenever you can) and cache forward pass variables (to compute the backward pass it is very helpful to have some of the variables that were used in the forward pass).
  - Compute the loss (for example MSE - mean squared error) and accuracy (if label == round(output): accuracy = 1; else: accuracy == 0) using result of the forward propagation and labels. The loss and accuracy should be an average of the losses on all samples in the mini-batch.
  - Compute the gradients of the training loss with the respect to the weights and biases using backpropagation equations:

$$\delta^L = \nabla_a C \odot \sigma'(z^L)$$

$$\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l)$$

$$\frac{\partial C}{\partial b^l_j} = \delta^l_j$$

$$\frac{\partial C}{\partial w^l_{j\,k}} = a^{l-1}_k \delta^l_j$$

  - Update the weights and biases using calculated gradients and step size.
  - Forward propagate the validation examples and compute the loss and accuracy for them. Use validation loss to decide if the training need to be stopped.
- Visualize a learning curve for training set and validation set: plot loss and accuracy as function of epochs.

Note: as an engineers, you have to choose the appropriate hyper-parameters: the learning rate and the batch size, along with other important decisions: the number of neuron in the hidden layer (more neurons leads to more parameters to train, is it good or bad? Adjust the algorithm to the data! Is it big?), the loss function, the non-linearity etc.

Submissions details:

You will submit the 'zip' file with:

- Neural network implementation code.
- A PDF file which summarize:
    - Loss and accuracy plots for training and validation.
    - The chosen hyper-parameters, loss function, hidden layer size, activation functions
    - Short description of the method.
- A json file which stores a dictionary with the following pattern:

```
trained_dict = {'weights': (W1, W2),
                'biases': (b1, b2),
                'nn_hdim': nn_hdim,
                'activation_1': 'activation_1',
                'activation_2': 'activation_2',
                'IDs': (id_1, id_2)}
```

Where:

- W1, W2, b1, b2 - your trained parameters - numpy arrays of shapes:
    - Wi – (input_size, output_size)
    - bi – (1, output_size)
- nn_hdim – number of neurons in the hidden layer (int).
- 'activation_1' – chosen activation function for the hidden layer. Can be 'sigmoid', 'tanh', 'ReLU', ' final_act' (string).
- 'activation_2' – chosen activation function for the output layer. Can be 'sigmoid', 'tanh', 'ReLU', 'final_act' (string).
- 'IDs' – your ID's (int)

Note:

- 'final_act' is for custom activation function and if you use ine, you should mention it in your submission and provide the code as follows:

```
def final_act(x):
    """
    insert activation function description
    """
    return 1 / (1 + np.exp(-x/10))
```

- Imput images should be normalized to range [0, 1]. You can use matplotlib library to load the images:

```
import matplotlib.image as mpimg
img = mpimg.imread(img_path)
```

Submission date: 03.06.2020

'zip' file name: project_id1_id2 **Good luck!**