**Project: Classification of Brain MRI Voxels**

**Medical Image Processing (2)**

1. The Purpose of the Assignment:

The purpose of this exercise, is to implement a backpropagation algorithm for fully connected neural network in order to classify brain MRI voxels (input) to 2 categories (output): (1) **Positive** to multiple sclerosis lesion or (2) **Negative** to multiple sclerosis lesion.

1. Data:
   1. Given Data Set:

In order to train (and validate) the neural network, we were given 668 patches (extracted from the axial view of FLAIR MRI modality) of 32X32 pixels, divided as follows:

* + 1. Training set - 512 patches (256 labeled as Positive and 256 labeled as Negative)
    2. Validation set – 156 patches (78 labeled as Positive and 78 labeled as Negative)

We noticed that the data is divided in a ratio of approx. 75% to the training set and 25% to the validation set, which is a legitimate division, according to many references.

* 1. Data Set Processing:

In our implementation, we made a simple Pre-Processing to the given data, using the function:



As an input, create\_dataset gets one of the following strings: (1) "training" or (2) "validation", in order to indicate which of the sets to pre-process (training or validation).

Since the patches are of size 32X32, we will represent them by 1024 input layer units. Initially, we will load each patch (in a matrix representation) into an array in gray scale. Afterwards, the pixels of each patch will be normalized to a range of [0, 1]. Finally, we will convert the representation of the patches into 1024 sized vectors.

The output of the function, which in practice will be used as the input to our neural network, is 2 (numpy) arrays: (1) x = vectorized patches pixels' values and (2) y = labels of each patch (Pos. or Neg.) in accordance.

A suggestion for a slight improvement of the results (will be presented in section 5), is to subtract the mean of the pixels' values of all the patches in the set from the pixel values in x. Since the test data will be given only as normalized values of the pixels, we decided not to implement it in this function.

1. Network Design:
   1. Layers:

Our fully connected neural network has 3 layers:

* + 1. Input Layer - vector in size 1024, which consists of pixel values of a patch.
    2. One hidden layer – 53 neurons.
    3. Output layer - a single neuron which returns the probability of the patch to be positive (i.e. 1=Pos. or 0=Neg.).

The number of neurons in the hidden layer, in addition to other hyper-parameters values, was chosen on a basis of a optimization process we made, and will be explained in subsection 3e.

* 1. Activation Functions:

We used the following activation functions:

* + 1. ReLU 🡪 Hidden layer. During our work on the project, we have read about which activation function is commonly used in Hidden layers. In most references, the recommended function is ReLU. The biggest advantage of ReLU is indeed non-saturation of its gradient, which greatly accelerates the convergence of stochastic gradient descent compared to the sigmoid/tanh functions[[1]](#footnote-1), which is an important attribute in the hidden layer.
    2. Sigmoid 🡪 Output layer. The sigmoid is a common function when a decision between two cases must be taken. By its definition, the sigmoid maps it input value to 1 or 0. Therefore, we decided to use it in our case: 1 for Positive and 0 for Negative.
  1. Architecture:   
     Draw a scheme…
  2. Loss Function

As suggested in the implementation guide, we used the MSE (Mean Squared Error) in order to compute the loss:

MSE formula…

Ask Paster about the derivative of MSE…

We got good and satisfying results using the MSE, and we decided to keep it as our method to calculate the loss of the neural network.

* 1. Parameters and Hyper-Parameters:

Paster, explain about the choice of the hyper-parameters and describe the "optimization process" you've done when search\_parameters==True…

1. Algorithm Description:
   1. Network Class:

In model.py, we defined a class for our model (network), with the following functions:

**Initialization**



Initially, we set the weight relatively randomly, but to be more accurate, with a normal distribution with mean=0 and std=. The value of the std was chosen based on Paster: write the name of the stanford's course…. The biases were all set to 0.

**Forward Propagation**



The forward propagation function, receives a mini-batch as an input, and computes the correlated predictions for each patch. The formula used is due to the lecture:

when fi is the activation function of the i's layer.

Also, the calculation of the accuracy is done by the forward function:



according to the formula requested in the assignment.

**Loss Calculation**



The function calculates the loss by MSE. Detailed explanation is given in subsection 3d.

**Backpropagation**



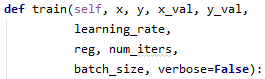
Based on the formulas that was shown in the lecture, the backpropagation function computes gradients of expressions through recursive application of chain rule.

**Weight and Biases Update**



Explanation…

**Train**



Explanation…

* 1. Training and Validation Script:

In train.py, we execute the training process of our network (i.e. model), along with the validation process.

Paster: explain the flow shortly (main etc.)…

1. Results:
   1. Loss & Accuracy:

Paster: graph plots only + add loss of validation…

* 1. Results Summary:

Paster: summarize the results…

Optional: (1) To add a chapter – how one can improve the results even more; (2) According to the results of the validation loss, we can finish the training earlier (after ~2500 epochs) to get the highest accuracy 🡪 ~97.6%...

1. Alex Krizhevsky et Al., ImageNet Classiﬁcation with Deep CNNs, [link](http://www.cs.toronto.edu/~fritz/absps/imagenet.pdf). [↑](#footnote-ref-1)