Project: Classification of Brain MRI Voxels

Course: Medical Image Processing (2)

1. The Purpose of the Assignment:

The purpose of this exercise is to implement a backpropagation algorithm with fully connected neural network in order to classify brain MRI voxels (input) to two categories (output):

(1) **Positive** to multiple sclerosis lesion.

(2) **Negative** to multiple sclerosis lesion.

2. Data:

a. 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.

b. Data Set Processing:

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

```
def create_dataset(train_or_val):
```

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 load and pre-process (training or validation).

Since the patches are of size 32X32, we will represent them by 1024 input vectors. 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 two (numpy) arrays:

(1) x = vectorized patches pixels' values [1024XN].

(2) y = labels [1XN] of each patch (Pos. or Neg.) in accordance.

N – the number of images in the given set of data.

A suggestion for a slight improvement of the results (will be presented in section 5c), 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.

3. Network Design:

a. 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. $\geq 0.5 = Pos; else\ Neg$) .

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

b. 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 major advantage of ReLU is the non-saturation of its gradient, which greatly accelerates the convergence of stochastic gradient descent

compared to the sigmoid/tanh functions[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 its input value to closer to 1 or closer to 0. Therefore, we decided to use it in our case: 1 for Positive and 0 for Negative.

c. Architecture:

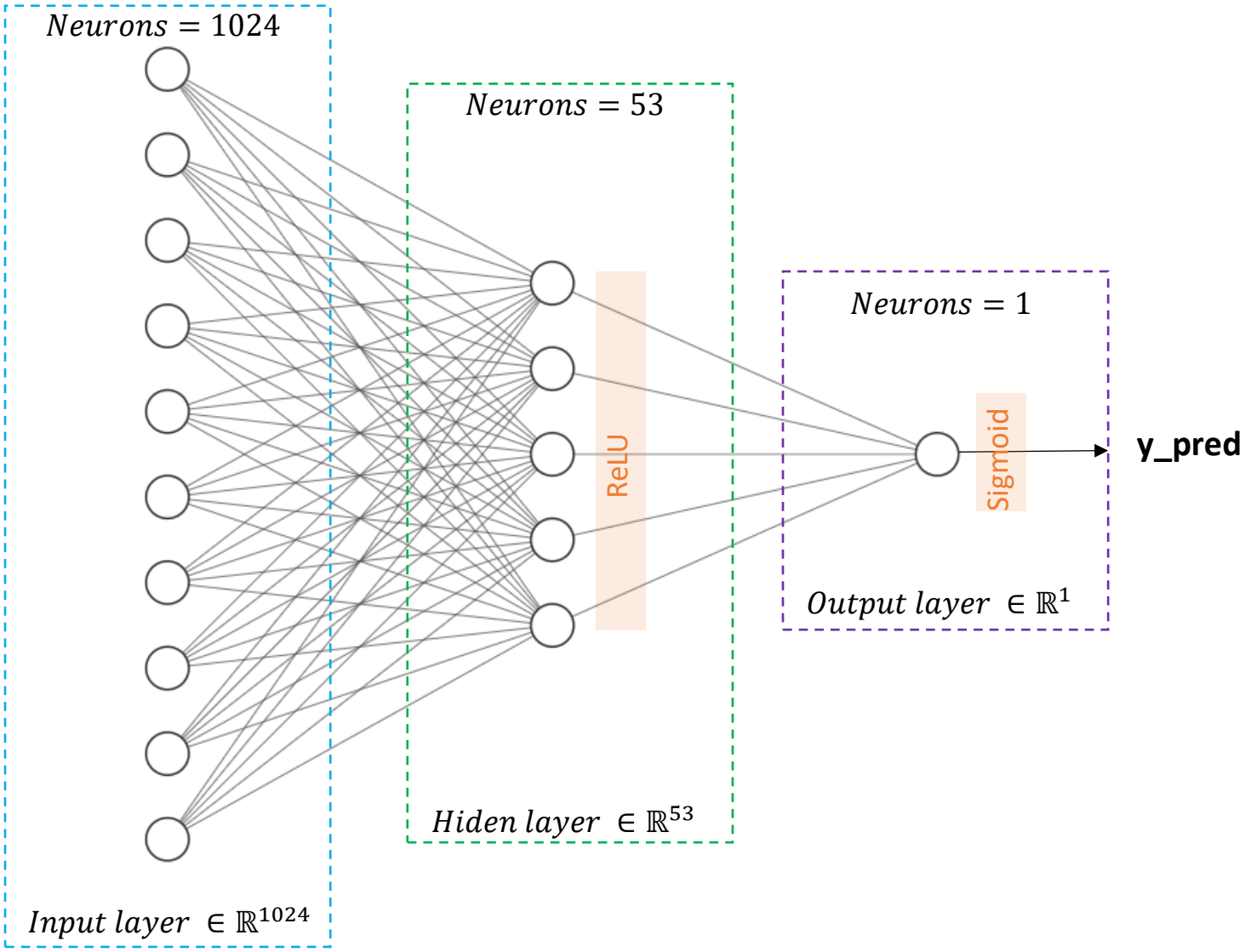

Figure 1 - Network architecture diagram

[1] Alex Krizhevsky et Al., ImageNet Classification with Deep CNNs, link.

d. Loss Function

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

$$MSE_{loss} = \frac{1}{2n}\sum_{1}^{n}(\hat{y} - y)^2$$

where n is the batch_size , ŷ is the prediction output vector and y is the correct output vector. As for the backward propagation we used the following derivative:

$$\frac{\partial MSE_{loss}(\hat{y})}{\partial \hat{y}} = \frac{1}{n}(\hat{y} - y)$$

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.

e. Hyper-parameters

Hyper-parameter	Value	Description
Hidden_size	53	The number of neurons in the hidden layer.
Batch_size	117	The number of images to be forwarded parallel in every iteration.
Learning rate	0.0385	The learning rate for the update parameters.
regularization	1.865e-06	Controls the effect of the regularization method on the lost and weight parameters update.

All values were set using an optimization technique (further explanation in subsection 4c).

4. Algorithm Description:

a. Network Class:

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

Initialization

```
def __init__(self, input_size, hidden_layer_size, output_size, std=1e-4):
```

Initially, we set the weight relatively randomly, but to be more accurate, we used a normal distribution with mean=0 and $std = \sqrt{\frac{2}{1024}}$. The value of the std was chosen according to the recommendation[2] in CS231n course[3].

The biases were all set to 0.

Forward Propagation

```
def forward(self, x):
```

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:

$$\underline{h^i} = f_i(\underline{\underline{W^i}} \cdot \underline{x} + \underline{b^i})$$

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

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

```
def get_accuracy(self, x, y):
    return (np.round(self.forward(x)) == y).mean()
```

according to the formula requested in the assignment.

Loss Calculation

```
def calc_loss(self, y_pred, y, reg = 0):
```

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

[2] Based on Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification by He et al. for ReLU based networks.

[3] CS231n: Convolutional Neural Networks for Visual Recognition

Backpropagation

```
def backward(self, x, y_train, y_pred, reg = 0):
```

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

```
def update_parameters(self, lr):
```

This method updates the values of the weights and biases parameters according to the pre-calculated derivatives and the learning rate:

$$params = params + \frac{\partial params}{\partial x} * learning\_rate$$

based on the formula given in the lecture.

Train

```
def train(self, x, y, x_val, y_val,
          learning_rate,
          reg, num_iters,
          batch_size, verbose=False):
```

This method initiates the training parameters of the network and starts. Given the training data, validate data and hyper-parameters values, the method preforms in every iteration (i.e. epoch) the following steps:

1. Randomly chooses a batch from the training data.
2. Forwards the batch trough the network.
3. Calculates the loss between the prediction vector and the expected result.
4. Backpropagates and updates the weight parameters accordingly.
5. Saves loss history of both the training and validation data.

   The accuracy history is being saved every epoch. Finally, returns the loss and accuracy history for both the training and validation sets, and the weight parameters values.

b. Training and Validation Script:

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

The script code contains the methods regarding the hyper-parameters optimization (explained in the next subsection), the create_dataset method for loading the pre-processed data, and the main() function itself.

As for the main(), we first load and pre-process both train and validation data, then printing their statistics. Right after that, we define the hyper-parameter values and some general parameters.

When the flag for 'search_parameters' is on, the code will go into the optimization section. Otherwise the model will preform one single train according to the manually set parameters.

Last, if the 'verbose' flag is on, we will print the loss and accuracy graph to the screen. In the end, we save the network parameters to a .JSON file.

c. Hyper-Parameters Optimization:

We used a random search technique for choosing the hyper-parameters values of [lr, reg, hidden, batch_size].

In train.py we defined the following methods:

```
def generate_random_hyperparams(lr_min, lr_max, reg_min, reg_max, h_min, h_max, b_min, b_max):
def random_search_hyperparams(lr_values, reg_values, h_values, b_values):
```

The first function generates random values inside a given range for all four parameters. The latter function chooses from a given set of values a random combination for the parameters.

The optimization process is as followed:

First, we used the "generate_random_hyperparams" method to generate values in a relative broad range. We did so for twenty iterations and then deduced which values gave the best result as for the validation accuracy percentage.

Then, we updated the randomization range accordingly and repeated the above process twice more.

After doing so, we had limited range for the parameter's values that produced the best results, from which we chose three values for each parameter.

Next, we used the "random_search_hyperparams" method to generate random combinations of those values and did so for twenty more times.

Finally, we chose the best result of the last twenty iterations.

The three ranges sets we used were:

```
# Use generate_random function with 1000 iterations
lr, reg, hidden_size, batch_size = generate_random_hyperparams(-5, -1, -5, 0, 1, 100, 1, 256) #1000 iters
lr, reg, hidden_size, batch_size = generate_random_hyperparams(-3, -1, -6, -4, 8, 92, 32, 256) #1000 iters
lr, reg, hidden_size, batch_size = generate_random_hyperparams(-2, -1, -6, -5, 40, 60, 100, 230)  # 1000 iters
```

**The 'lr' (learning rate) and 'reg' range are for the exponential degree.*

And the final values we used for the randomized combination were:

```
random_search_hyperparams(    [0.038446182351894766, 0.038446182351894766, 0.031557371805925306],
                              [1.8653823506313295e-06, 2.47040676891957e-06, 4.2812448183270965e-06],
                              [50, 53, 55], [117,200,180])
```

5. Results:

a. Loss & Accuracy graphs:

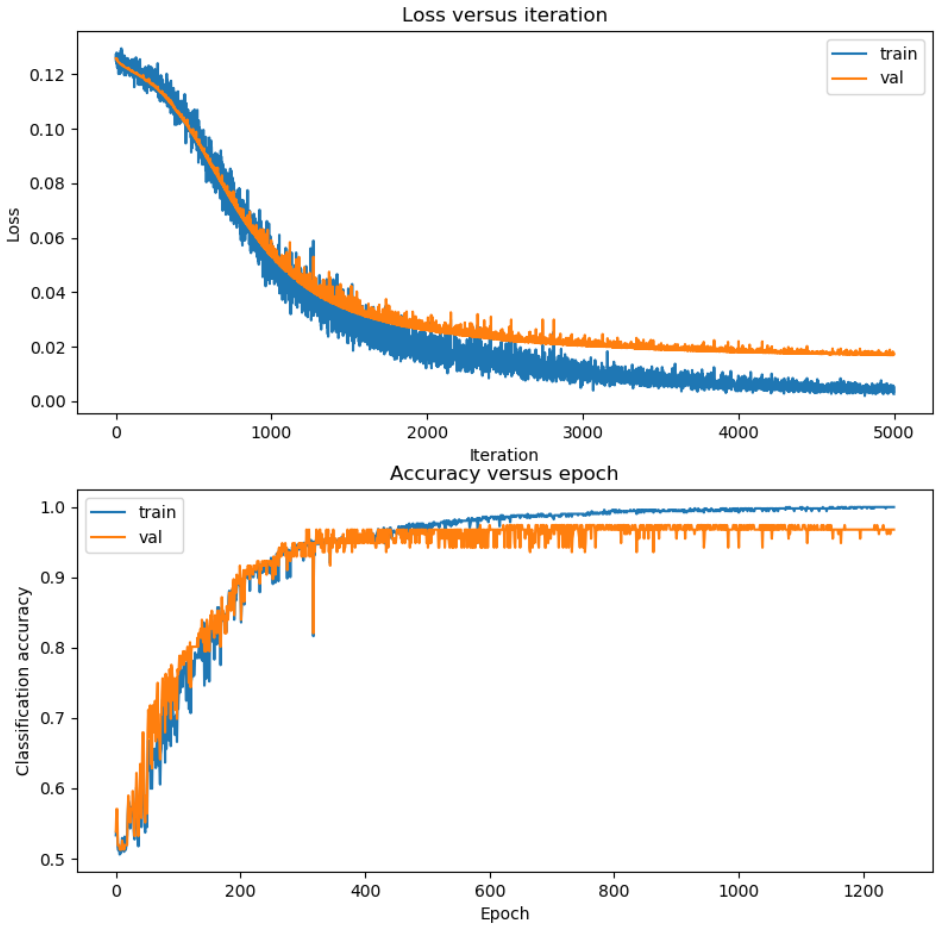

Figure 2 – loss versus #iteration (top) and accuracy versus #epoch (bottom)

As expected, the validation set yield higher loss value and smaller accuracy result than the training set and in a reasonable range.
We can see that as for the accuracy graph the validation set does not improve from around epoch 500, however the training set's accuracy continue in converging to almost 100%. This would be due to a slightly overfit process happening in the second half of the epochs.
Still, the validation loss is monotonically decreasing, thereby we conclude that the overfitting effect is minor.

b. Results Summary:

Result	Value
Train loss	0.0041
Validate loss	0.017
Train accuracy	0.998
Validate accuracy	0.974

We achieved a result of **97.4%** accuracy for the validation data.
Those are pretty good results and if no major overfitting occurred, we expect to see the same results also for the test set.

c. Suggestions for improvement

As mentioned at the begging, we achieved better results when pre-process with an equalization method the data set distribution as follows:

$$\#\, Normalize\ the\ data\colon subtract\ the\ mean\ image$$
$$mean\_image\ =\ np.mean(x, axis = 0)$$
$$x\ -=\ mean\_image$$

Using the above method, we got **99.35%** accuracy for the validation set!

Hereby is the loss and accuracy graph:

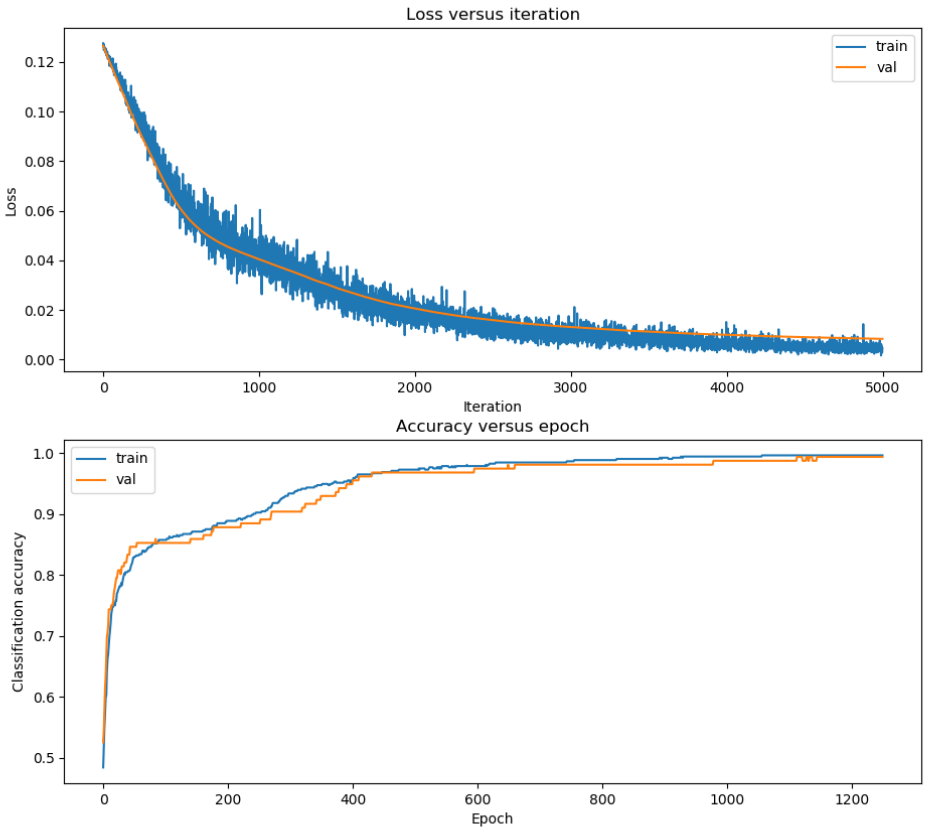

Figure 3- loss versus #iteration (top) and accuracy versus #epoch (bottom) with equal distbute data sets