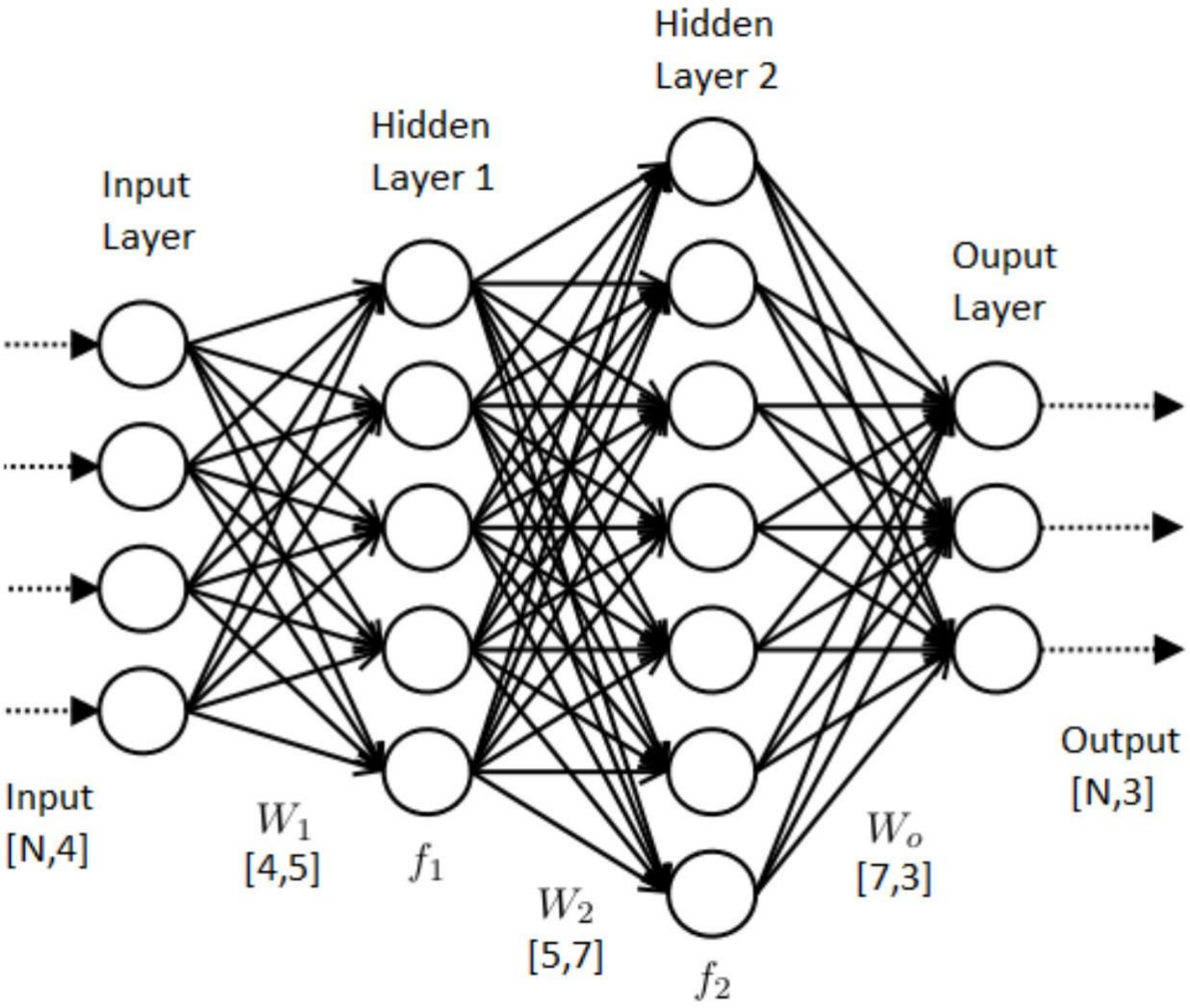
Part 2: Neural Network implementation



Author: Julen Arizaga

Contents

Problem statement

In this project we will implement a three layer neural network for a three class classification without using a GPU framework and with classes for each node type. Therefore we will have to implement a neural network from scratch, using only the knowledge obtained in class and without help of a high level framework as Tensorflow, Keras or Pytorch. Apart from this we are also asked to use categorical cross entropy, use sigmoid activation function for the hidden layers and softmax for the output one, and implement functions to calculate the loss and evaluate how well our model behaves.

Proposed solution

In order to implement a Neural Network from scratch, apart from the class nodes, several other functions and classes have been developed. Here the functions and classes made will be explained, with their input and output.

Helper functions

Those helper functions include several functions in order to make the code cleaner. The functions are the following:

- **Sigmoid:** Takes an input parameter (can be a vector), computes the sigmoid of it and returns it
- **Softmax:** Takes an input parameter, computes the softmax of each of its values and returns it
- **L2loss:** Takes the real Y, the predicted Y and the number of samples and calculates the average L2 loss. (Note: This function isn't used, it is for further development of the implementation)
- **CatCrossEntropyLoss:** Takes the real y, the predicted y and the number of samples and calculates the average Categorical Cross-Entropy loss.
- **Evaluate:** Takes as input the real Y and the predicted Y and calculates the mean accuracy. This function has been developed in order to work in a binary classification too.
- **Predict:** Takes as input the X (input features) and the trained model, computes the Y according to the input and the model and returns it
- **One_hot_encoding:** Takes as input the Y and the number of classes and returns Y as a one hot vector. (Note: This function only works for 3 classes)

Parameters handling

In order to initialize the parameters inside the neural network, two functions have been done:

Initialize_parameters

This function takes as inputs the number of neurons in layer 1, layer 2, the number of input features and the number of classes and initializes the weight matrices correctly. There is more than one approach to do this, but in this case, for each layer a weight matrix and a bias vector have been defined, initializing the weight matrices randomly and multiplying them by 0.01, in order the initial weights not to have repercussion in further updating. Finally all the parameters are ensembled in a dictionary and returned.

Update_parameters

This function takes as input the model, the learning_rate and the decay and updates the weights. In order to do so, the weights and its derivatives are taken from the model (they are stored there) and the computation of each weight is done. After this, the new weights are stored in the model, and this function doesn't have to return anything, as the weights are updated inside the model.

Classes developed

In order to implement the neural network with one class type for each node, several class types have been made with inheritance in order to ease the later work. Those are the classes developed:

Obj

This class is the parent class whose functions and attributes will be inherited by the other classes. The Obj class has two functions:

- **MakeGraph:** This function initializes the weights and stores them as an attribute and builds the neural network by concatenating multiple nodes. The nodes (called gates in the code) are stored in a list inside the class.
- **MakeGraphBinary:** This function does the same as the previous one, but in this case the last node will be a sigmoid node, used for binary classification instead of multiclass classification. This function is not used in the code, but is done for further development.

MultiplyGate

This class inherits the attributes of the Obj class and has two more functions:

- **Forward:** takes as input the activation (or features) of the previous layer, the weight matrix, the bias and the save parameter (used to differentiate between forward propagation for the training set and the validation set), computes the multiplication for the next layer and returns the product of the multiplication. Also stores the activation function, the weights and the bias (if save = True) for the backward propagation.
- **Backward:** takes as input the derivative of the previous step in the backward propagation and the number of samples, computes the derivative of the node with respect to the activation, the weight matrix and the bias vector of the forward propagation of the node and returns those derivatives.

SigmoidGate

This class inherits the attributes of the Obj class and has two more functions:

- **Forward:** takes as input the multiplication from the previous layer and the save parameter, computes the sigmoid function over this input and returns the activation function. Also stores the input multiplication (if save = True) for the backward propagation.
- **Backward:** takes as input the derivative of the previous step in the backward propagation, computes the derivative of the node with respect to the input multiplication of the forward propagation of the node and returns this derivative.

SoftmaxGate

This class inherits the attributes of the Obj class and has two more functions:

- **Forward:** takes as input the multiplication from the previous laye, computes the softmax function over this input and returns the predicted labels.
- **Backward:** takes as input the real and the predicted labels and returns dz. Note that dz in this case is the combination of the backward propagation of the Softmax and the Categorical Cross-Entropy.

ComputeGraph

This class inherits the attributes of the Obj class and is used to do the forward and the backward propagation. For this it uses the following functions:

- **Forward:** takes as input the input of the neural network and the save parameter and performs the forward propagation over all the nodes of the model. In order to do this it takes the weights of the model and goes through each node giving the adequate parameters, until reaching the last Softmax node. Finally it returns the predicted labels
- **Backward:** This function takes as input the real labels, the predicted labels and the number of samples and computes the backward propagation through all the nodes. Through the backward propagation, the derivatives will propagate till the input, and the derivatives of the weights and the bias will be obtained this way. This derivatives will be stored inside the model (as a dictionary), and the function will return this derivatives too.

Final function

In order to train the model, a function has been developed in order to do the forward and backward propagation several times, returning the trained model in addition to the plot of the loss through the iterations.

ComputeModel

This function takes as input the model, the input of the neural network of the training and validation sets, the labels of the training and validation sets, the batch size, the epochs, the learning rate and the decay.

This function first makes the batches with the training set and then, looping over the number of epochs and the number of batches, it performs the forward propagation, calculates and stores the loss, performs the backward propagation and finally updates the parameters. In addition to this, inside of the loop the validation loss is calculated too, by doing a forward pass with the validation set and calling the CatCrossEntropyLoss function (same as with the training set, but without backward propagation and save = False).

Finally, the average of the loss per epoch is computed and it is plotted. This function also returns the loss as a list, and the model fed is trained.

Implementation details

Through this assignment several implementation choices have been made, as the neural network has been implemented from scratch without using any other tool.

One design issue has been how to make the neural network as general as possible. In order to do this generalization and allow the user to build and customize the neural network, all the network has been implemented with the “layer1”, “layer2”, “classes” and “input_size” parameters in order to do multiclass classification (not only with 3 classes, but with user-specified classes) with different types of input features.

The most difficult problem faced during this assignment has been the dimensionality of the matrices. In order to do the implementation as efficient and as flexible as possible, a lot of matrix multiplications have to be done, and in order to do them, the dimensions have to be right, which isn’t as easy as it may seem when there are so many matrix multiplications.

Another problem faced has been the loss plot. As silly as it may seem, when plotting the loss per batch, it seemed not to decrease at all and only oscillate around the initial value. Because of this, it seemed the model was not being trained and there was some kind of error, when in reality the model was being trained and the only error was the loss plot. Finally, computing the average loss per epoch this problem has been solved.

Results and discussion

In order to prove that this self-made neural network really works for classification, two datasets from UCI have been downloaded and used. As the purpose of this step is to see that the neural network works properly, the hyperparameter tuning has not been a deep search, instead, some hyperparameters have been tried and the best models will be presented in the following part.

First, the obtention of the data will be discussed. As it is hard to obtain multiclass datasets of 3 classes from the UCI repository, two multiclass datasets have been downloaded, and then a mask has been added in order only to take 3 classes out of them. The two datasets are the following:

- Frogs_MFCCs: it has 23 numeric variables and 3 categorical columns
- Winequality-white: it has 12 numeric variables and 1 categorical column

In the first dataset all the numerical variables have been taken, and from the categorical columns only "Family". In the second dataset all the columns have been taken. One remark, the first dataset contained non usable rows in the UCI repository, so they have been removed manually.

First, after loading the datasets, the variables have been normalized in both cases. In order to perform a good evaluation, in both datasets the dataset has been separated into training set, validation set and test set.

Apart from this, as the model has to have the labels fed as one hot vector, they have been first modified (in order the labels to be "0", "1" and "2") and then fed to the one_hot_encoding function.

Finally, as we have all the appropriate vectors, we can use them to train and evaluate the model.

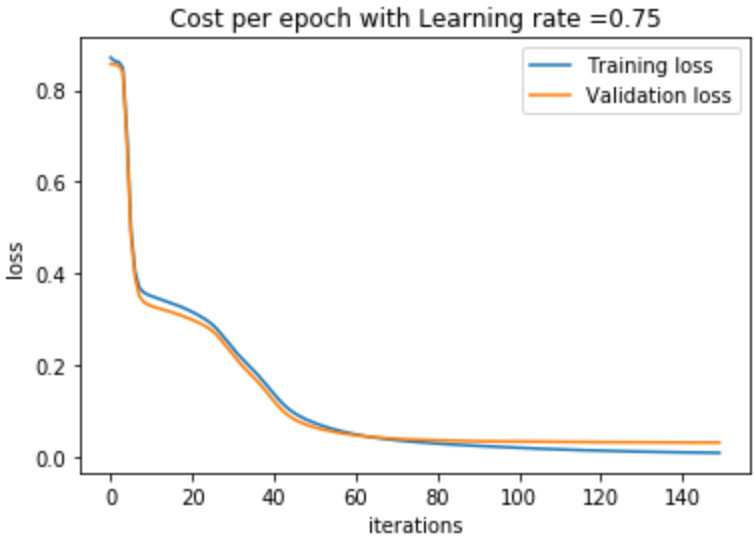
Results

1st dataset

After trying different combination of hyperparameters, those are the ones that give the best accuracy for this dataset:

- Layer 1: 20 neurons
- Layer 2: 10 neurons
- Batch_size: 128
- Epochs: 150
- Learning rate: 0.75
- Decay: 0.0001

Training the model with those parameters, the following loss graph is obtained:



As it can be seen in this graph, both the training loss and the validation loss decrease and neither of theme increases. The validation loss not increasing with the number of epochs means that the model is not overfitted, which is good notice.

Finally, evaluating our model in the test set we obtain the following accuracy:

```
0.9908835904628331
```

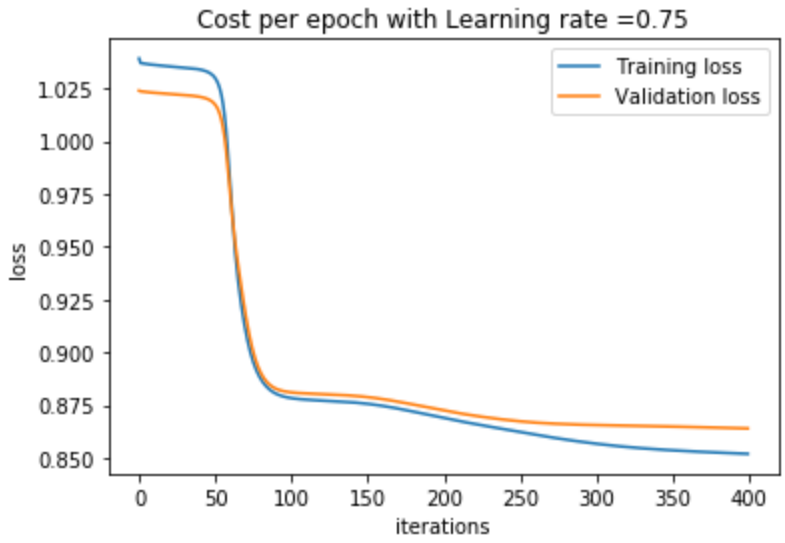
As it can be seen, this accuracy is very high, and the model predicts the labels nearly perfect. With this in mind, it can be said that the implementation of the neural network has been successful.

2nd dataset

After trying different combination of hyperparameters, those are the ones that give the best accuracy for this dataset:

- Layer 1: 15 neurons
- Layer 2: 10 neurons
- Batch_size: 256
- Epochs: 400
- Learning rate: 0.75
- Decay: 0.001

Training the model with those parameters, the following loss graph is obtained:



As it can be seen in this graph, both the training loss and the validation loss decrease and neither of these increases. The validation loss not increasing with the number of epochs means that the model is not overfitted, which is good notice.

Finally, evaluating our model in the test set we obtain the following accuracy:

```
0.5755237045203969
```

This accuracy is far lower than the one obtained in the previous dataset. After looking carefully at both datasets, this second dataset contains less features (nearly half) than the previous one, so this could be one reason of the low accuracy.

In order to conclude, after the evaluation of the model in two datasets, the implementation of the neural network behaves properly. The model doesn't achieve a high accuracy in the second dataset, but this is surely due to the dataset and not the implementation, because otherwise the model wouldn't achieve a 0.99 of accuracy in the first model.

References

https://archive.ics.uci.edu/ml/index.php