Deep Learning - Neural Networks

Aristotle Universtity Thessaloniki - School of Informatics

Assignment 1: Multilayer Perceptron

Antoniou, Antonios - 9482

aantonii@ece.auth.gr

GitHub repository can be found here

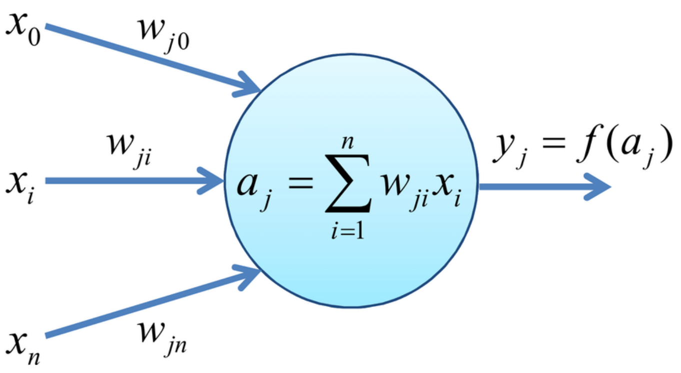
Introduction

In the first assignment of the Deep Learning course, our goal is to make a Neural Network **(NN)**, of any architecture, built for classification of problems with many classes (i.e. not a binary classifier). The selected one was a Multilayer Perceptron **(MLP)**, that is the simplest and most easily digestible of all, so we can dive deeper into the mathematic side of the project and have time to develop it all from scratch.
Before we start dissecting the algorithm into its components, we need to make sure we reminded ourselves of the most important terms, briefly.

Terminology recap

What is a Perceptron?

A Perceptron is an entity simulating the function of a **neuron** in the human brain. In that sense, it simply accumulates information from sources before it (for which it has different **"weights"**, or degrees of **significance**), it processes it (that is, applies a certain arithmetic operation on all the collected data), and passes the output to the neurons that are connected to it afterwards. We can look at a Perceptron like this:



We refer to $f(\cdot)$ as the **activation function** of the Perceptron. It is used to keep the output on a relatively predictable range and filter out insignificant inputs, but also filter in significant ones.
Note: Oftentimes, as well as on this project, the x_0 and $w_{j,0}$ parameters belong to the **bias**: An *offset* to the inputs that either trainable, or given explicitely by the designer.

Multilayer perceptrons

MLP's are simply layers of Perceptrons connected with each other, to make a structure that's capable of solving problems, or approximating functions that are non-linear and complex in general. They are:

- Fully connected: All neurons in layer i communicate with all neurons in layer $i+1$.
- Feedforward: Information, other than errors, travels to the next layers and never comes back.

We will delve into the details, by explaining the structure of the MLP factory that was developed for this report.

Coding and training an MLP

We split the classifier into its basic entities, that we turned into Python classes. This admittedly made the program a little less efficient, and the parameter changes a little longer to write. However, efficient libraries for Deep Learning have already been written. The reasons behind that Object-Oriented approach are pretty simple. First, it's a habit! Secondly, it makes the algorithm written a lot more readable, easier to debug and fully understand, since that's the bottom line of the project in question.
Having gotten that out of the way, we will now briefly explain the functionality of each of the 3 classes that the classifier is comprised of.

Neuron

This is the modelling of the Perceptron, explained above. **It will also be reffered to as a *Node*,** like it belongs to a directed graph. For the Neuron to be capable of being trained there is a myriad of attributes and inbetween values to be kept and updated during each iteration of testing.
First and foremost, a Node needs to know if it belongs to the input, output or any hidden layer. In the first case, it only broadcasts the value given to it to every Node in the first hidden layer. In the latter case, it needs to keep track of the target value given to it, and compare it to the one that it output. Output Nodes calculate their error e, and δ value, that are kept as attributes:

```
def get_error(self):
    return self._d - self._y
```

```
delta = e * self.df(node.u)
```

As you can see, the Neuron has also stored a `callable` for activation function `f` and its derivative `df`, along with:
$y = f(u)$, for which
$u = \underline{w} \cdot \underline{x}$
The $\underline{w}$ vector is kept internally, for each Neuron, along with the weights before the current iteration of training, for reasons we will explore later on. For the sake of consistency, the rest of the class attributes will be listed here, and also explained later:

- ID's for the layer it lies in, and an its own ID inside the layer,
- The learning rate η
- The target d (allocated and initialized in case of an output neuron)
- The number of inputs n_{in}

Layer

The Layer class is a wrapper around the array of Nodes that make it up. It contains the number of Nodes of the current and the previous layer, essentially kept so that the data can be passed on to the constructor of each Node. It also contains the type t of layer it is (once again, input, output, or hidden) and the layer ID it comes with.
All in all, its function is to simply organize Neurons into arrays, so they can process information both during predicting and training in a more easy-to-track way.

Network

It encapsulates all the attributes and routines an MLP needs to be constructed, trained, tested and used. Apart from the attributes engrained in the array of Layers (and subsequently the arrays of Nodes), it also has:

- The α momentum factor,
- The resulting accuracy,
- The total error e', and
- The array of the resulting δ values for each Neuron of the Network, for a single sample during training, $sdeltas$.

The class implements the *predict()*, *train()* and *test()* functions.

Predict

Function signature:

```
def predict(self, x)
```

We create an array $y = x$. For the first layer, we simply assign:

```
for i in range(len(x)):
    self.layers[0].nodes[i].y = x[i]
```

For the rest of them, we use the same y vector to place the result in. The output of layer i is the input of layer $i + 1$, until the output layer is reached. As usual, for each Neuron of the layer, u is calculated first, then y is extracted. The function returns the entirety of $\underline{y}$, so we can keep all the probability values that the output Neurons concluded in. If *classes* is the array of available classification options, this means that in the occassion that *index_of(max(y)) = i*, then `classes[i]` is the decision the Network has made.

Train

Function signature:

```
def train(self, x, d, batch_size, epochs, minJ)
```

For reference, here is the whole implementation of the function (without the implementation of the auxiliary routines constructed to make it more readable):

```
for iter in range(epochs):
    perm = np.random.permutation(size)
    x, d = x[perm], d[perm]
    curr_idx = 0
    self.e = 0
    for b in range(0, size, batch_size):
        b_size = min(batch_size, size - b)
        for p in range(b_size):
            curr_x = x[curr_idx]
            curr_d = d[curr_idx]
            set_targets(out_layer, nout, curr_d)
            self.predict(curr_x)
            self.update_output_errors(nout, out_layer)
            self.update_hidden_deltas(out_layer, last_idx)
            curr_idx += 1
        last_hidden = self.layers[last_idx - 1]
        update_output_weights(last_hidden, out_layer, nout, self.eta, self.alpha)
        self.update_hidden_weights(last_hidden, out_layer, last_idx)
        self.reset_errors_and_deltas(nout, out_layer, last_idx)
```

The routine is given the inputs x, and target outputs d of the dataset it will be trained on. We also supply:

- The *batch size*, which is the number of samples the Network makes a prediction and calculates e and δ values for,
- The *epochs*, which is the number of iterations of training throughout all samples,
- The *minJ* parameter, which is the value of the *loss function* at which the Network stops training, because we deem it reached a level of correctness we are satisfied with,
- The *min_acc* parameter, which denotes the accuracy level that is good enough for the classifier.

Remarks:

1. As one could guess, the batch size is kept consistent until we've reached the last batch. In that case, the batch-specific size is either the same, or the remainder of the samples, after we've gone through the rest
2. We will break down the training procedure into epochs. Every epoch requires the same processes and calculations, so all we need is to describe the progression of the algorithm during just one.
3. During each epoch, the initial inputs and targets are shuffled using the same permutation of indices, so we can keep the Network from overfitting.

When the Network is first made, the weights are randomly initialized. However, since symmetry has to be avoided, the weights are given by a random generator following a **uniform distribution between -2 and 2**.
For every sample p, the output nodes are informed about the target values they are supposed to output. The `predict()` function is called, and directly every output Node's error is calculated, together with δ, like shown above. For the rest, hidden, nodes, we need to calculate δ once again,

but differently this time.
Let's take Node $n_i^{(l)}$ as an example. The notation means we're dealing with the Node with $ID = i$, on layer l:

- $$\delta = \dot{f}(u) \cdot \sum_{j=1}^{N(l+1)} (\delta_j^{l+1} \cdot w_{j,i})$$

When it comes to the implementation in Python, that's what we need the $sdeltas$ array, that keeps track of δ each Neuron calculates, layer by layer, so that the error can be propagated to the previous layer, up until we've reached the first hidden nodes. That is **Back Propagation (BK)**:

```
for j in range(next_layer.n):
    delta += next_layer_deltas[j] * next_layer.nodes[j].w[i + 1]
return dfu * delta
```

We use `w[i+1]`, since `w[0]` is **reserved** for the **bias** of each Neuron.
As can be seen in the last 3 lines of code, every time we've examined enough samples, it's time to update the weights of the Network, then reset all the values that relate to errors: e and δ for all Nodes.
Regarding the weight update, once we've gone through `b_size` of samples, summing up errors and δ's we calculate for Neuron i, on layer l:

- $$new\_w_{i,j} = w_{i,j} + \eta \cdot \delta_i^l + y_j^{l-1} + \alpha \cdot previous\_w_{i,j}$$

This is what we need `wprev` for, the array of weights before the current iteration, that we talked about earlier:

```
for j in range(node_i.n_in):
    wprev = node_i.wprev[j + 1]
    node_i.wprev[j + 1] = node_i.w[j + 1]
    node_i.w[j + 1] += eta * node_i.delta * last_hidden.nodes[j].y + alpha * wprev
```

Testing

Lastly, there is a decision to make on the division of the data. Generally speaking, we give 60% of the data to the Network to train and keep 40% for testing. We have to pay special attention to pre-processing: the training data is shuffled before every epoch, but we need to make sure we keep a balance between the samples of different classes the Network trains on. There is a significant possibility of **overfitting**, with a bias towards the class that had the most training samples.
Here is an example of how that was handled using the Iris dataset:

```
samples, targets = split_into_classes(x, y, 3, species)
x_train, x_test, y_train, y_test =
    split_trainset_testset(samples, targets, train_fraction, species)
```

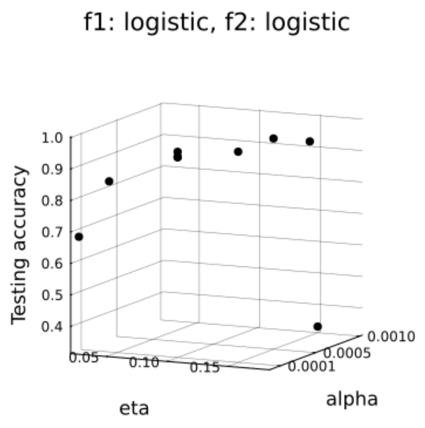
Here, the *split_into_classes* function uses the *species* dictionary to split the classes of the dataset into different arrays inside *samples* .
Afterwards, *split_trainset_testset* , takes approximately a percentage of each sample class equal to *train_fraction* and places them all into *x_train* (and the respective targets into *y_train*). The rest goes into the tests.

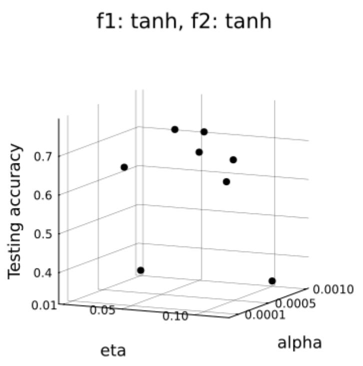
Acknowledgement

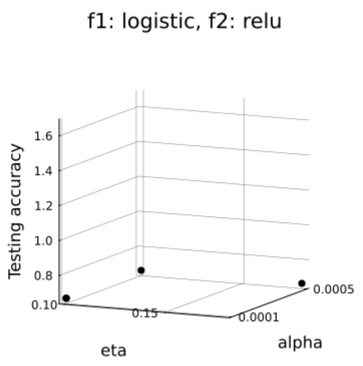
Any of the proposed models seem to be performing sub-par for non-linearly separable datasets. The loss function is minimized, however it soon reaches a plateau and takes considerable time in order to "escape" that region. All in all, the MLP works pretty well with linearly separable classes, such as Iris.

Training with the Iris dataset

The main activation functions to have been implemented were **tanh**, **logistic** and **relu**. The dataset was through of series of tests with combinations of activation functions and hyperparameters (that can be also found in tabular form in the GitHub repository). `f1` is the activation function on the hidden layer, and `f2` is the respective function of the output layer.







Clearly, the **logistic function** performs better and is a little more **robust** towards the different hyperparameters the Network is given to train with. It took only 3 experiments to see that using the Iris dataset with relu on the output is overkill and doesn't result in convergence.

Now that we know that the logistic function performs better, the next step is experimenting with **early stopping**. The next experiments show the number of epochs it took before the Network reaches an accuracy of **95%** on the training dataset, and what that meant for the testing accuracy percentage:

eta	alpha	train acc.	test acc.	epochs
0.05	0.0001	0.967	0.967	81
0.05	0.0002	0.956	0.967	60
0.05	0.0003	0.956	0.933	44
0.05	0.0004	0.967	0.967	37
0.05	0.0005	0.956	0.933	29
0.1	0.0001	0.956	0.95	59
0.1	0.0002	0.956	0.933	44
0.1	0.0003	0.956	0.933	29
0.1	0.0004	0.956	0.95	30
0.1	0.0005	0.967	0.967	29

The rest of the table can be found on ***acc95.csv***. The two values of η, paired with the five combinations of α, are enough to show that, even if convergence is quick, a greater value for the momentum factor can result to less accurate predictions, provided that the learning rate isn't sufficiently greater.

KNN and Nearest Centroid Classifiers

A more aggressive division of data was used (80% for training and 20% for testing) for two KNN Classifiers (with number of neighbors equal to 1 and equal to 3), and a Nearest Centroid Classifier.

Below are the results of the tests:

```
>> Fit model for Nearest Centroid Classifier : 0.0010 sec.
 >>> NearestCentroidClassifier : 66.667 accuracy.
```

```
>> Fit model for number of neighbors : 1 [0.0010 sec.]
>> Fit model for number of neighbors : 3 [0.0000 sec.]

 >>> KNeighborsClassifier with n_neighbors=1 : 70.000 accuracy.
 >>> KNeighborsClassifier with n_neighbors=3 : 70.000 accuracy.
```

The average MLP training time (without early stopping, and one hidden layer) is 2.9 seconds. While this still isn't much, it is entire orders of magnitude larger than the training times above. However, the MLP averages an accuracy above 95% on a larger testing dataset.

The differences in time can be better viewed on the Dry Bean dataset, that consists of 2500 samples. The simpler classifiers still take less than a second to train:

```
>> Fit model for Nearest Centroid Classifier : 0.0010 sec.
 >>> NearestCentroidClassifier : 61.431 accuracy.
```

An MLP with one hidden layer and the `logistic` function (that performs much better than `tanh`), takes over 3 seconds for a single epoch (with disappointing results):

```
- Epoch 0 [3.3 secs.]: e = 0.81, accuracy = 0.154
- Epoch 1 [4.1 secs.]: e = 0.78, accuracy = 0.149
- Epoch 2 [3.2 secs.]: e = 0.78, accuracy = 0.144
- Epoch 3 [3.5 secs.]: e = 0.78, accuracy = 0.146
```

Note: More details on the training of the `Dry Bean` and `Red Wine` datasets can be found in the `logs` directory of the repository.

Thank you for your time!

Antoniou, Antonios