Universitetet i Oslo

INF4490

Biologically inspired computing

Multilayer Perceptron

William Arild Dahl
Username: williada

16.10.2018

UiO : Universitetet i Oslo

1 Introduction

In this assignment, we were expected to implement, train and validate a simple multilayer perceptron. The miltilayer perceptron is a two layer neural network, with an input layer consisting of 40 neurons, a hidden layer with up to 12 neurons and an output layer with 8 neurons. The neural network is trained to classify electromyographic (EMG) signals corresponding to various hand motions. The neural network implemented is classified as a supervised machine learning model. This is because we are using data with correct classification.

When I mention some algorithm from the book I refer to *S. Marsland: Machine learning: An Algorithmic Perspective*. All code is written in Python (3.6.4) and the `sklearn` library is used to create the confusion matrix and to calculate accuracy score. Running the network is done by running the `movements.py` file. For kfold implementation run the `kfold.py` file:

```
python3 ./movements.py
python3 ./kfold.py
```

2 Algorithms and Implementation

Our neural network uses backpropagation where the general idea is that errors on the output layer is propagated backwards to update the weights is the network. We were quite free to choose which algorithms we wanted to use for activation functions, calculating errors and updating weights. To compute the activation of each neuron `j` in the hidden layer I choose to use a sigmoid function (4.5):

$$a_\zeta = g(h_\zeta) = \frac{1}{1 + exp(-\beta h_\zeta)}$$

Where:
h_ζ is the sum of weights multiplied by the input values from the input layer,
β is some positive parameter.

To compute the activation for each neuron in the output layer, I chose to just use output value as it is.

The error rate for output layer is calculated with(4.14):

$$\delta_o(\kappa) = (y_\kappa - t_\kappa)$$

Were:
κ is the neuron,
y is the actual output,
t is the target output.
The error rate for the hidden layer is calculated with(4.9):

$$\delta_h(\zeta) = a_\zeta(1 - a_\zeta)\sum_{k=1}^{N}\omega_\zeta\delta_o(k)$$

For updating the weights on the outer layer I use (4.10)

$$\omega_\zeta\kappa \leftarrow \omega_\zeta\kappa - \eta\delta_o(\kappa)\alpha_\zeta$$

As to what percentile correct guesses would be 'well classifications' is very dependent on the application of the network. Since we are implementing a prosthetic hand controller (PHC), I would guess that the poor fella using this prosthetic hand, would not be to happy with classifications below or very close to 100% as his/hers day would be quite miserable as this number decreases. But for a delivery on a 10 credit course I suppose there is allowed some more room for error.

The condition I implemented to end training is a limit on how many iterations without improvement we will allow before we revert back to the best point so far. I choose to do this so that the network will not run for too long and provide it enough time to explore some more solutions than simply ending it at a prefixed percentile correctness. I choose to update and save the best weights during runtime, so every time the network performs better on the validation data the new best weights are stored. It is also worth to mention that I do evaluation for each epoch/iteration over training data. This decreases performance, but increases precision, as the network will not easily overfit.

3 Example runs

I chose to test my network with $n = 3,6,9$ and 12 hidden nodes. I ran each of them 5 times, and here are the best results from those runs:

```
-------------------------------
Number of hidden nodes: 12
-------------------------------
Confusion matrix:

[[ 8  0  0  0  0  0  0  0]
 [ 0 13  0  0  0  0  0  0]
 [ 0  0 14  0  0  0  1  0]
 [ 0  1  0  9  2  0  1  1]
 [ 0  0  0  0  8  0  0  0]
 [ 0  1  0  0  1 10  3  0]
 [ 0  0  0  0  0  0 17  0]
 [ 0  2  0  0  0  0  0 19]]

-------------------------------
Correctness score: 88.29%
-------------------------------


-------------------------------
Number of hidden nodes: 9
-------------------------------
Confusion matrix:

[[16  0  0  0  1  0  0  0]
 [ 0 12  0  0  0  0  0  0]
 [ 0  0 12  0  0  0  0  0]
 [ 0  0  0  6  2  0  4  1]
 [ 0  0  0  0 12  0  1  0]
 [ 0  4  0  0  0  5  5  0]
 [ 0  0  1  0  0  0 16  0]
 [ 0  0  0  0  0  0  0 13]]

-------------------------------
Correctness score: 82.88%
-------------------------------
```

```
-------------------------------
Number of hidden nodes: 6
-------------------------------
Confusion matrix:

[[12  0  0  0  0  0  0  1]
 [ 0 18  0  0  0  0  0  0]
 [ 0  0 10  2  0  1  0  2]
 [ 0  0  0 11  1  0  0  0]
 [ 1  0  0  1 13  0  0  0]
 [ 0  0  0  0  0 13  0  0]
 [ 0  0  0  1  0  8  0  0]
 [ 0  0  0  5  0  0  0 11]]

-------------------------------
Correctness score: 79.28%
-------------------------------

-------------------------------
Number of hidden nodes: 3
-------------------------------
Confusion matrix:

[[ 0  2  0  4  2  0  0  4]
 [ 0 20  0  0  0  0  0  0]
 [ 0  1  0  0  0 12  2  0]
 [ 0  0  0  9  1  1  0  0]
 [ 0  0  0  0 16  0  0  0]
 [ 0  0  0  0  0  6  4  0]
 [ 0  0  0  0  0  1 13  0]
 [ 0  5  0  0  1  1  0  6]]

-------------------------------
Correctness score: 63.06%
-------------------------------
```

We can see, the correctness score decreases as the number of hidden nodes decreases. As mentioned above, the percentile limit for 'well classification' can be debated, but I wold say that the score should be atleast 80%, and that is possible with 9 hidden nodes or more. When we look at the confusion tables there are some classes that the network has problems with, but it seems to differ depending on the number of nodes. For the $n = 12$ node network, it seems like it has a slight problem with category g):(pronation), and mistakes it for f)(radial deviation). This also seems to hold true for $n = 9$ nodes aswell. For the $n = 3$ and $n = 6$ node networks, the category that causes most problems is f)(radial deviation). $n = 6$ mistakes it for g):(pronation) and $n = 3$ mistakes it for c)(flexion).

4 K-fold

For the kfold implementation i choose $k = 5$. Normally k value is set to 10, but since the amount of data available for us is quite small(compared to other real life datasets), I think 5 is sufficient. Below is the most successful confusion matrix with hidden nodes set to 12. Here are the Confusion matrices and correctness score for each of the folds.

```
-------------------------------
Fold number: 0
Correctness score: 91.01%
-------------------------------
Fold number: 1
Correctness score: 94.38%
-------------------------------
Fold number: 2
Correctness score: 92.13%
-------------------------------
Fold number: 3
Correctness score: 91.01%
-------------------------------
Fold number: 4
Correctness score: 93.26%
-------------------------------
Average correctness: 92.36%
Standard deviation: 1.31
-------------------------------
```

As with the iterative $n = 12$ network, it has problems with categorizing g) correctly. However the predicted class is now d) (extension). However it performs a lot better than the non K-fold implementation. Since the weights are initiated randomly we can experience a very different result after 1 iteration of training. Some times the network will be as successful as a middle 80's correct predictions after the first training iteration, and other times, the network only performs a low 60. But this is expected due to the randomness of our network initialization.

END.