TDT4137 - Exercise 3

Håkon Ødegård Løvdal

October 13, 2015

1 Perceptron

I created my perceptron in such a manner that it automatically initate the weights and threshold to random numbers in the interval $[-0.5, 0.5]$. The only user input possible is:

```
-debug D, -d D              Debug, default=0
-method M, -m M             Training set method, default=AND
-learningrate LR, -lr LR    Learning rate for training, default=0.1
-ephocs E, -e E             Max epochs for training, default=1000
```

From running the perceptron multiple times with diffrent weights and input data, I noticed that the weights changes during the epochs. Sometimes it does not converge, due to errors it ain't able to fix. Other times it converges, but also gets errors it ain't able to fix. This could be due a too high learning rate. When the applied learning rate is too high, the perceptron oscilliates around the solution, but never reaches the solution. It learns faster, but gets a much poorer accuracy. The inital weights makes a influence on how things turns out.

Example output

```
$ python3 perception.py -m AND -lr 0.1 -e 100 -d 0
Initiated perceptron with [-0.2, 0.1], 0.3
============
[-0.2, 0.1]
[-0.2, 0.1]
[-0.2, 0.1]
[-0.1, 0.2]
```

```
============
[-0.1, 0.2]
[-0.1, 0.2]
[-0.1, 0.2]
[0.0, 0.3]
============
[0.0, 0.3]
[0.0, 0.2]
[0.0, 0.2]
[0.1, 0.3]
============
[0.1, 0.3]
[0.1, 0.2]
[0.1, 0.2]
[0.1, 0.2]
============
[0.1, 0.2]
[0.1, 0.2]
[0.1, 0.2]
[0.1, 0.2]
Training converged with 5 iterations needed
```

```
$ python3 perception.py -m AND -lr 0.1 -e 100 -d 0
Initiated perceptron with [-0.3, -0.0], 0.0
============
[-0.3, -0.0]
[-0.3, -0.1]
[-0.3, -0.1]
[-0.2, 0.0]
============
[-0.2, 0.0]
[-0.2, -0.1]
[-0.2, -0.1]
[-0.1, 0.0]
============
[-0.1, 0.0]
[-0.1, -0.1]
[-0.1, -0.1]
```

```
[0.0, 0.0]
============
[0.0, 0.0]
[0.0, -0.1]
[-0.1, -0.1]
[0.0, 0.0]
============
[0.0, 0.0]
[0.0, -0.1]
[-0.1, -0.1]
[0.0, 0.0]
Training isn't able to correct existing errors
```

```
$ python3 perception.py -m OR -lr 0.1 -e 100 -d 0
Initiated perceptron with [0.4, -0.1], 0.1
============
[0.4, -0.1]
[0.4, 0.0]
[0.4, 0.0]
[0.4, 0.0]
============
[0.4, 0.0]
[0.4, 0.1]
[0.4, 0.1]
[0.4, 0.1]
============
[0.4, 0.1]
[0.4, 0.1]
[0.4, 0.1]
[0.4, 0.1]
Training converged with 3 iterations needed
```

```
$ python3 perception.py -m OR -lr 0.1 -e 100 -d 0
Initiated perceptron with [-0.3, 0.3], -0.1
============
[-0.3, 0.3]
[-0.3, 0.3]
```

```
[-0.2, 0.3]
[-0.2, 0.3]
============
[-0.2, 0.3]
[-0.2, 0.3]
[-0.1, 0.3]
[-0.1, 0.3]
============
[-0.1, 0.3]
[-0.1, 0.3]
[-0.1, 0.3]
[-0.1, 0.3]
============
[-0.1, 0.3]
[-0.1, 0.3]
[-0.1, 0.3]
[-0.1, 0.3]
Training isn't able to correct existing errors
```

2 PyBrain and feed-forward networks

The lowest number of hidden layers that produces a decent result is around 3-4. I find it rather intersting that the training varies quite alot during each run of the code. Sometimes it gives a very good and precise result, but other runs gives an output not even close to expected output. After the training, when I activate the net with the numbers 1-6 it does quite good. The numbers 7-8 are not that precise, but within an acceptable range. Also, sometimes, it outputs a good result with 1 hidden layer.

Through the hidden layers, the neural net has recreated perceptrons (computational neurons) to represent a transformation from input- to output-data. The hidden layers is a internal representation of the input data. Each layer in the hidden layer applies a function from the previous layer to produce an output. The hidden layers transforms the inputs into an output the output layer can use.

When I activate the net with different numbers than the one it trained with, like negativ-, decimal, and/or large numbers, it handles this quite poorly. See the example output for examples of this. What I notice is that the large numbers outputs numbers closer to 8. Similar large negative numbers outputs numbers closer to -8. Small negative numbers moves closer to 0.

Example output

```
Using 8 hidden layers
Start training
Finished training
Test function f(x)=x
f(1) = 0.979860
f(2) = 2.141948
f(3) = 3.022019
f(4) = 3.900924
f(5) = 4.906988
f(6) = 6.040428
f(7) = 7.091751
f(8) = 7.842596
f(2) = 2.141948
f(-8) = -10.639628
f(-0.234000) = -2.462520
f(1765345) = 8.714651
f(-34534) = -11.296342
```

```
Using 6 hidden layers
Start training
Finished training
Test function f(x)=x
f(1) = 0.985286
f(2) = 2.000066
f(3) = 2.985579
f(4) = 3.922041
f(5) = 4.896732
f(6) = 5.995403
f(7) = 7.065934
f(8) = 7.872399
f(2) = 2.000066
f(-8) = -6.784545
f(-0.234000) = -2.837673
f(1765345) = 8.834537
f(-34534) = -6.826734
```

```
Using 4 hidden layers
Start training
Finished training
Test function f(x)=x
f(1) = 1.196657
f(2) = 1.901461
f(3) = 2.939462
f(4) = 4.153277
f(5) = 5.188203
f(6) = 5.879415
f(7) = 6.286682
f(8) = 6.514659
f(2) = 1.901461
f(-8) = -1.085883
f(-0.234000) = 0.648394
f(1765345) = 6.806245
f(-34534) = -1.498718
```

```
Using 1 hidden layers
Start training
Finished training
Test function f(x)=x
f(1) = 1.167560
f(2) = 1.965237
f(3) = 3.099231
f(4) = 4.336313
f(5) = 5.350459
f(6) = 6.004519
f(7) = 6.363566
f(8) = 6.543403
f(2) = 1.965237
f(-8) = 0.250433
f(-0.234000) = 0.633517
f(1765345) = 6.703820
f(-34534) = 0.249485
```