Statistical Methods for Machine Learning Assignment 3

Kristina Aluzaite (ljg630)
Morten Winther Olsson (hkv518)

18. March 2014

Introduction

III.1 Neural Networks

III.1.1 Neural network implementation

Code is done, see "src/Neuron.py". The random data is generated by `samples = map(lambda x:  (x, (math.sin(x)/x)+np.random.normal(0, .004, 1)[0]), (np.random.rand(1000)*32).tolist())` which should yield data similar to the provided `sinc` data. Results are in "Neuron_stdout". The backpropagation-wise and numerically calculated gradient differ by more than the assignment text says it should, but still very little. We presume the larger error is due to using floats of limited bit-length in the program.

As for the derivative:

$h(a) = \frac{a}{1+|a|} = a \cdot (1 + |a|)^{-1}$

Using the product and chain rule gives the derivative.

For the case of $a \leq 0$:

$h(a) = a \cdot (1 - a)^{-1}$

and:

$h'(a) = (1 - a)^{-1} - \frac{a}{(1-a)^2} \cdot -1 = \frac{1-a}{(1-a)^2} + \frac{a}{(1-a)^2} = \frac{1}{(1-a)^2}$

and since $a \leq 0$, $-a = |a|$ so $h'(a) = \frac{1}{(1+|a|)^2}$

For the case of $a \geq 0$:

$h(a) = a \cdot (1 + a)^{-1}$

and:

$h'(a) = (1 + a)^{-1} - \frac{a}{(1+a)^2} \cdot 1 = \frac{1+a}{(1+a)^2} - \frac{a}{(1+a)^2} = \frac{1}{(1+a)^2}$

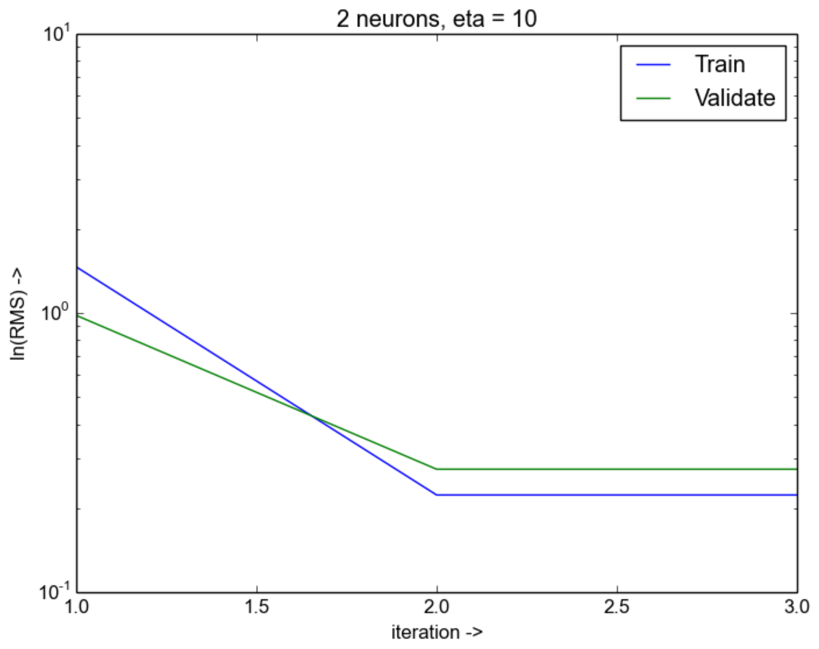
and since $a \geq 0$, $a = |a|$ so $h'(a) = \frac{1}{(1+|a|)^2}$

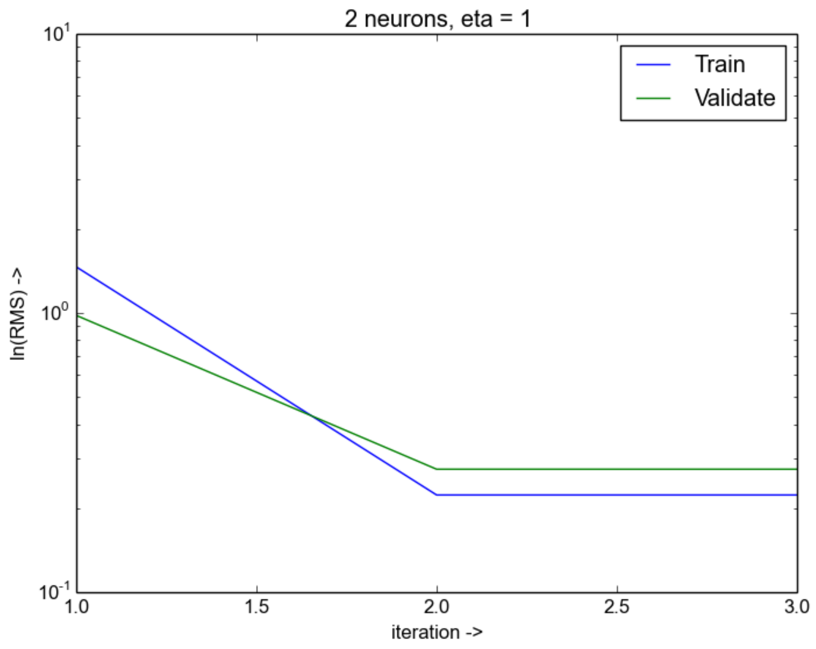
The derivative includes the case $a = 0$ in both cases, and therefore holds continuously across that point.

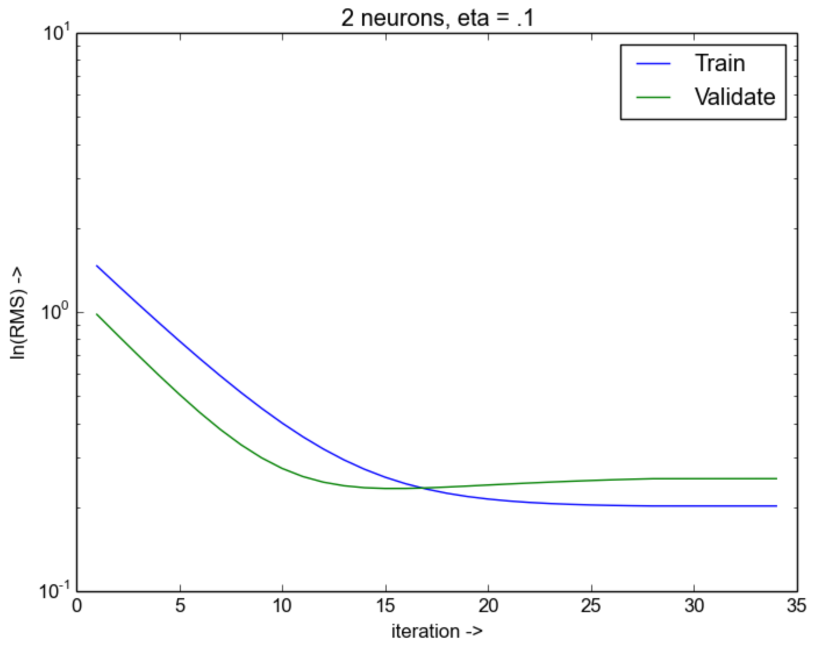
III.1.2 Neural network training

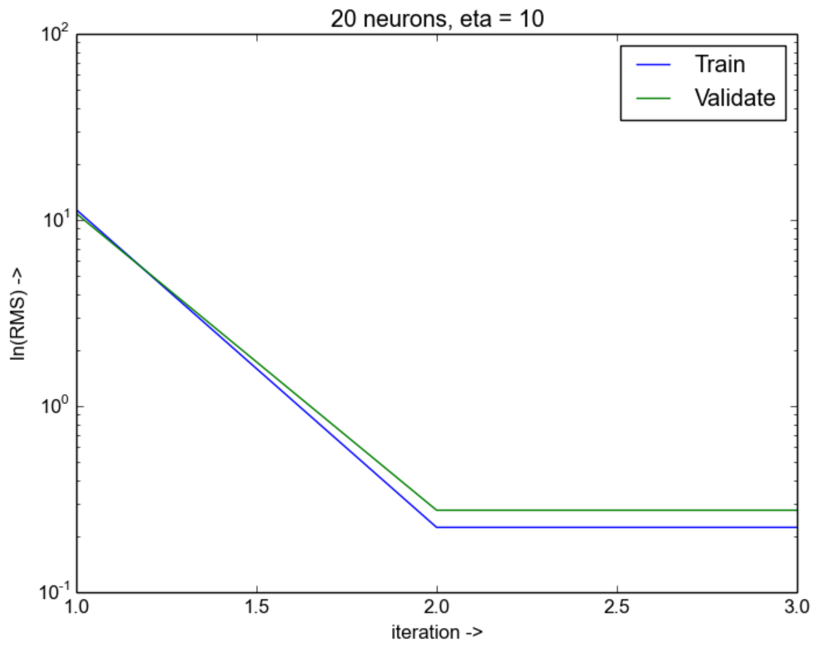
See “Neuron_stdout”. We need to stop gradient descent before overfitting (weights growing to extremes). For small learning rates, it is hard to tell if any progress is made at all, in terms of lowering the error, since the change is so small. However, it is possible to acurately approach the “ideal” compromise between low error and least overfitting, while for larger learning rates, the network might be overfitted without warning during a single application. In our case it seems the best learning rate lies somewhere between 0.01 and 1.0. We chose (on gut instinct, (mostly) ignoring validation data outputs) 15 iterations with a 0.1 learning rate for the 2 neuron network, and 30 iterations with a 0.01 learning rate for the 20 neuron network.

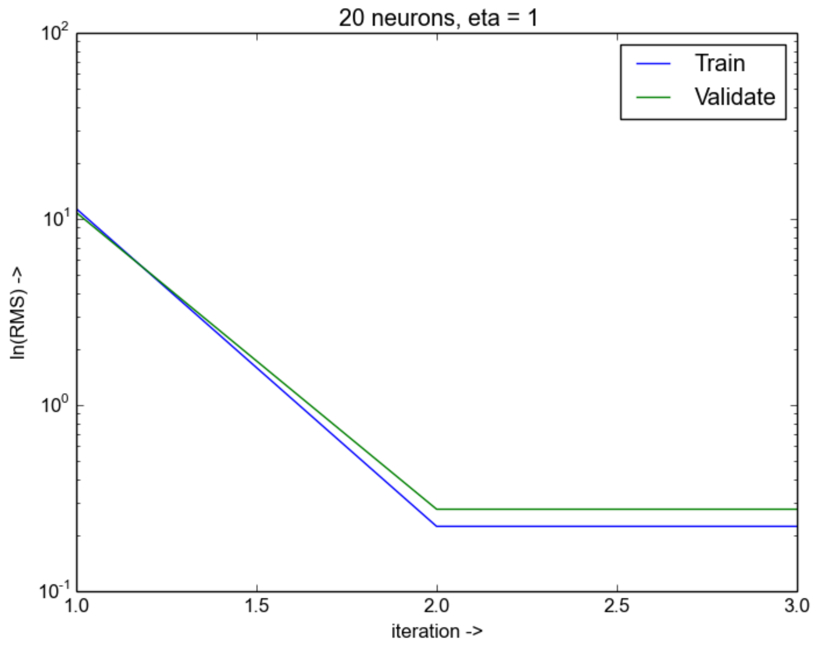
Plots:

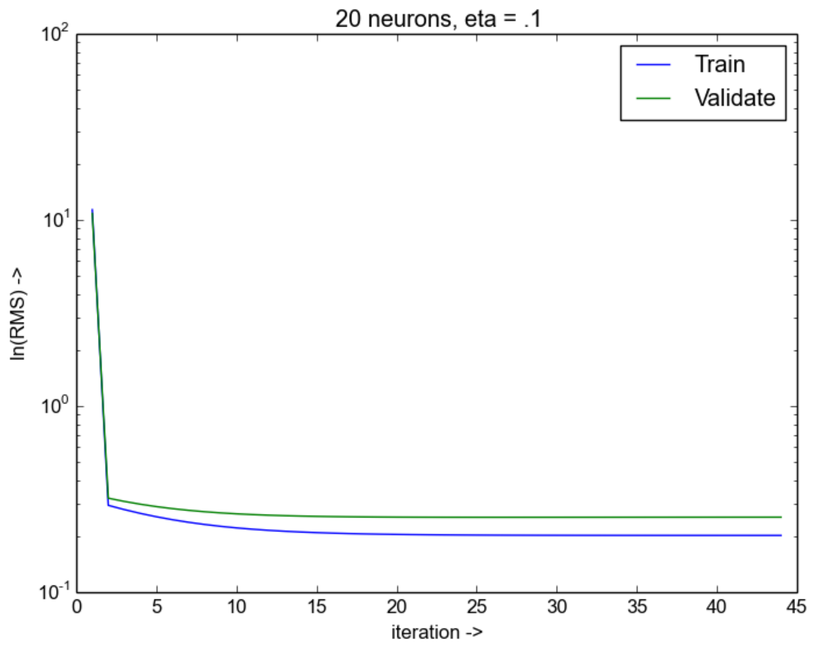


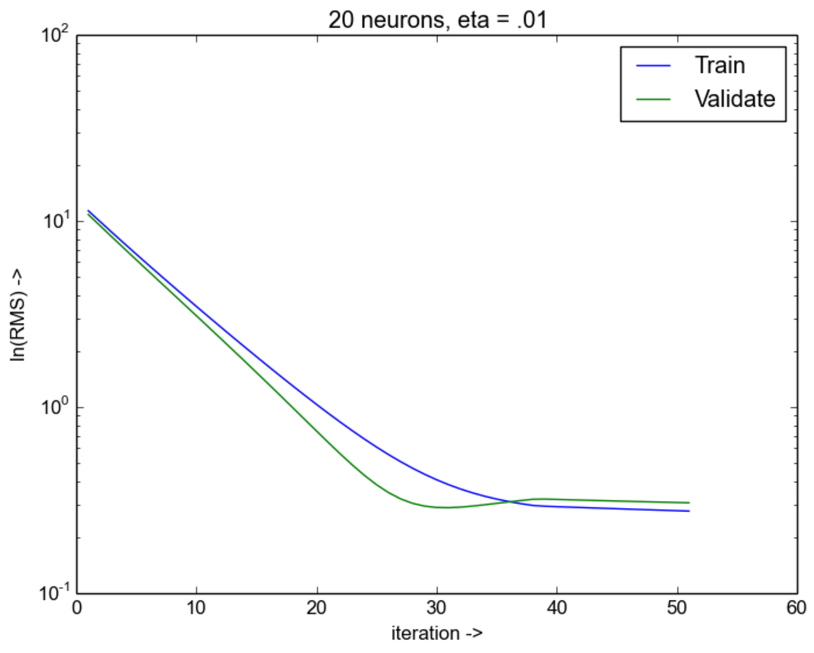

2 neurons, eta = 1
Train
Validate
ln(RMS) ->
iteration ->
10^1
10^0
10^{-1}
1.0
1.5
2.0
2.5
3.0


2 neurons, eta = .1
Train
Validate
ln(RMS) ->
iteration ->
10^1
10^0
10^{-1}
0
5
10
15
20
25
30
35

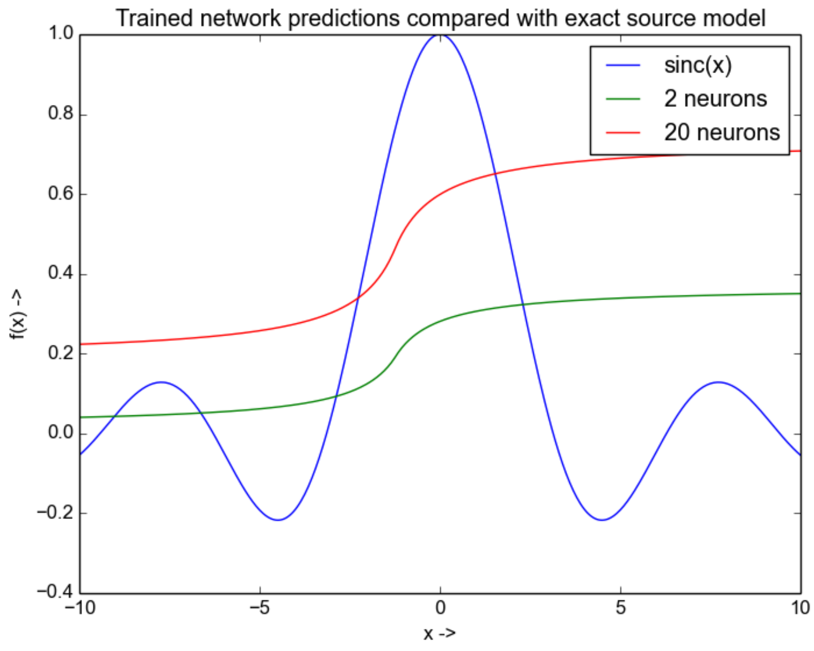

20 neurons, eta = 10
Train
Validate
ln(RMS) ->
iteration ->


20 neurons, eta = 1
Train
Validate
ln(RMS) ->
iteration ->


20 neurons, eta = .1
Train
Validate
ln(RMS) ->
iteration ->
10^{2}
10^{1}
10^{0}
10^{-1}
0
5
10
15
20
25
30
35
40
45


20 neurons, eta = .01
Train
Validate
ln(RMS) ->
iteration ->
10^{2}
10^{1}
10^{0}
10^{-1}
0
10
20
30
40
50
60

Plot of combined test of the trained networks[1]:



III.2 Support Vector Machines

Done, see “src/svm_prob.py”.

III.2.1 Data normalization

Done, see the end of “SVM_stdout”.

III.2.2 Model selection using grid-search

Done, see “src/svm_prob.py” and “SVM_stdout”. We used the libSVM as advertized in the assignment document. For cross-validation, we defined a higher-order function “`partition`”, which, given the number of desired `folds`, return a function that, given a list, returns a list of `folds` tuples of two lists made from the original list, where the first member of each tuple contain members whose index modulo `folds` is not the index of the tuples place in the overall list, and the second tuple member is a list of the remaining members left out from the

[1]Yeah, I know the predictions are about as close to the “right” values as american right-wing christian fundamentalists, but I'm too tired to care any more, I need sleep :) ... Zzz...

first list.[2]

It is important to perform numeric input normalization when using SVMs. It places the values of attributes on the same scale, avoiding the bias that would result from attributes of large original scales. In addition, it makes the data more separable, resulting in more accurate classification and less error. The effects of normalization get more pronounced in higher degree polynomial functions.

For the raw data we found the best hyper parameters were $C = 100$, $\gamma = 0.0001$, for normalized data $C = 10$, $\gamma = 0.1$. The SVM built on raw data predicted the test classification correctly in just over 80pct of the cases, while normalization increased that figure to just over 90pct, as can be seen in "`SVM_stdout`".

III.2.3 Inspecting the kernel expansion

.1 III.2.3.1 Support vectors

During training, `libSVM` automatically outputs the number of free and bounded support vectors as "`nSV`" and "`nBSV`" respectively. For the final normalized SVM, nSV=56 and nBSV=1, as can be seen in "`SVM_stdout`". Lower values of C generally yield more bounded SVs but the same amount of free SVs. C is an upper bound on the Lagrange multipliers, and bounded support vectors are those whose multiplier reaches that bound, while support vectors in general are all those that have a Lagrange multiplier $a \geq 0$. If a support vector is not bounded, then it lies on the margin. Thus, if there are no bounded SVs, the data is perfectly separated with a hard margin, which might suggest overfitting. Lowering C reduces model complexity, and allows more (potentially mis-classified) vectors on the "wrong side" of the margin.

.2 III.2.3.2 Scaling behavior

If different features of the data are on different scale, numerically larger ones will carry more weight in the separation process. For large data sets, this "advantage" can accumulate, meaning that the SVM model will tend to favor the larger feature, to the point of ignoring all others in the infinite limit[3].

[2] Just look at the code, please, it's not THAT complex, just hard to explain :) ...
[3] I think?