KU Leuven

Data Mining and Neural Networks

Assignment 1

Zachary Jones
Master of Staistics

January 6, 2019

Contents

1 The Perceptron and Beyond

1.1

A perceptron classifies data by minimizing a cost function through a gradient descent approach, allowing two disparate datasets that can be seperated by a straight line to be classified into two seperate categories. Linear regression refers to the approximation of a straing line that has a closed form solution found by minimizing the mean squared error between a hypothetical straight line and the individual data points.

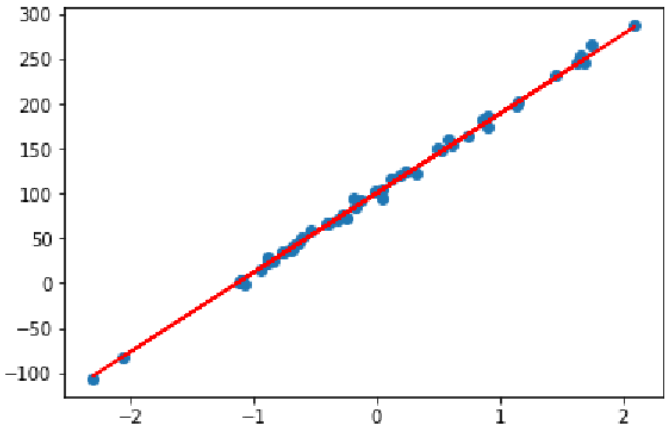


Figure 1: Perceptron trained by minimizing the MSE of generated data with gaussian noise

The perceptron works by first initializing weights for a straight line $\hat{(y)} = \theta X$ where $\hat{(y)}$ is the estimate of the value of the datapoint for each X and θ corresponds to the vector of weights applied to each dimension of the data. Then, after calculating the *cost function*, a measure of how seperate the hypothetical straight line is from the actual data, the weights are updated by the rate of change of the cost function with respect to the weights in order to find a minima. This is repeated until either a sufficient number of updates have occurred, there is little change in the values during, or some error threshhold is met. It then classifies the data using an output function which is dependent on the weights. By chosing the cost function to be the mean squared error and the output function to simply be the result of the weights acting on the datapoints, a perceptron can perform a linear regression, as seen in figure 1

1.2

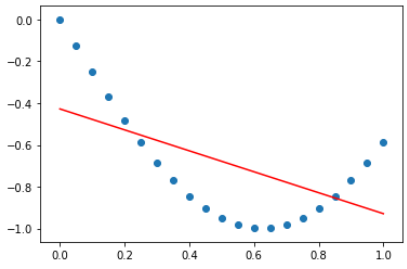


Figure 2: Plot of noiseless data generated with attempted linear regression

Clearly since sin is decidedly nonlinear the data will not be fittable by a linear regression, as illustrated in figure 2. One approach that may be viable while still allowing for the use of linear regression is a transformation of variables to act as coefficients for the truncated taylor polynomial for $sin(x) = x - \frac{x^3}{3!}$....
A linear regressor of the form $\hat{y} = \theta_0 + \theta_1 c_1 + \theta_2 c_2$... where the c_i's are the odd polynomial values of x would allow for a better fit. Since a linear regression is essentially a truncated version of the above, we can argue that it is similar to using a polynomial fit with weights that have been severely penalized past θ_1, and is underfitted.

1.3

Broadcasting the values of each x_i onto the coefficients $[1.39199167, -1.927428]$ yields the activation values for neurons one and two respectively.

```
array([[-0.        , -0.        ],
       [-0.06076021, -0.08800269],
       [-0.12152042, -0.17600538],
       [-0.18228063, -0.26400806],
       [-0.24304084, -0.35201075],
       [-0.30380105, -0.44001344],
       [-0.36456126, -0.52801613],
       [-0.42532147, -0.61601882],
       [-0.48608168, -0.7040215 ],
       [-0.54684189, -0.79202419],
       [-0.6076021 , -0.88002688],
       [-0.66836231, -0.96802957],
       [-0.72912252, -1.05603226],
       [-0.78988273, -1.14403494],
       [-0.85064294, -1.23203763],
       [-0.91140315, -1.32004032],
       [-0.97216336, -1.40804301],
       [-1.03292357, -1.4960457 ],
       [-1.09368378, -1.58404838],
       [-1.15444399, -1.67205107],
       [-1.2152042 , -1.76005376]])
```

The activations for each neuron.

1.4

As we can see in figure 3, when we plot the values of x^1 and x^2, we uncover a quadratic relationship between both the x_i^1's and x_i^2's and their respecive y values.
For the next part, we calculate the output values of our classifier using the weights manually uncoverid. All of the values have been calculated using an equation

$$y = \begin{pmatrix} V_1 & V_2 \end{pmatrix} \begin{pmatrix} tanh(W_i^1 x_i + \beta_1) \\ tanh(W_i^2 x_i + \beta_1) \end{pmatrix} + \beta_2$$

with V_1 and V_2 being the second layer weights, W_i^1 and W_i^2 the weights from the previous section, and $\beta_{(1,2)}$ the intercepts.

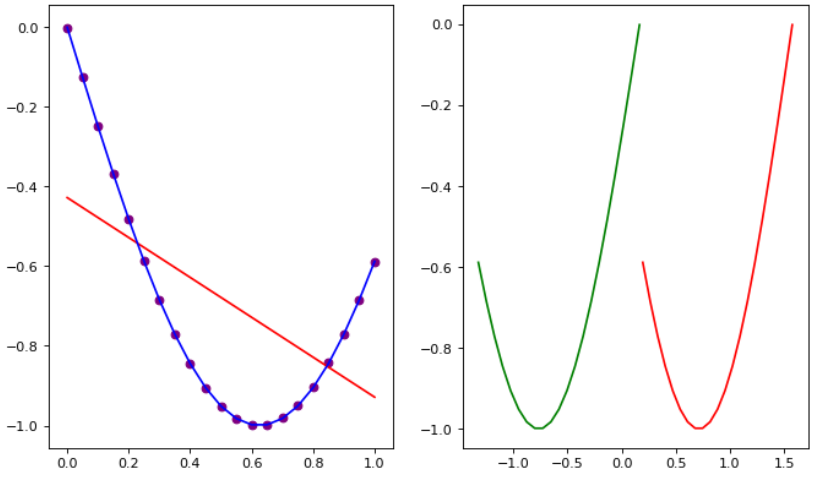


Figure 3: On the left hand side, a comparison of the outputs of the neural network manually calculated (purple) with the actual value of the function (blue) and a linear regression solution. On the right hand side, the weights x^1 (red) and x^2 (green).

2 Backpropagation in Feedforward Neural Networks

2.1

Comparing the different algorithms in figure 4, we find that the trainbfg is the most effective at approximating the $sin(x^2)$ function across both noisy and noisless datasets. It is a quasi-newtonian method for unconstrained nonlinear optimization problems where both the Hessian matrix is approximated and updated iteratively and the step size is found through a local optimization.

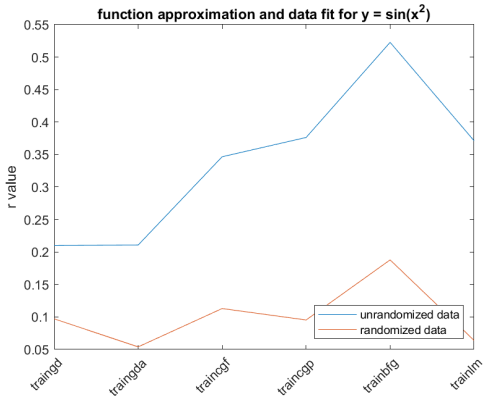


Figure 4: A comparison of the R value for different algorithms on data with noise and without

2.2

We can also examine the different levels of noise across all algorithms in order to determine the effect of noise on the training data, as well as see which algorithms are more robust to noisy data. By comparing the plots of several algorithms we are able to determine that the "trainlm" function actually works better for our noisy data than the others and focus our efforts on that one, as seen in 5

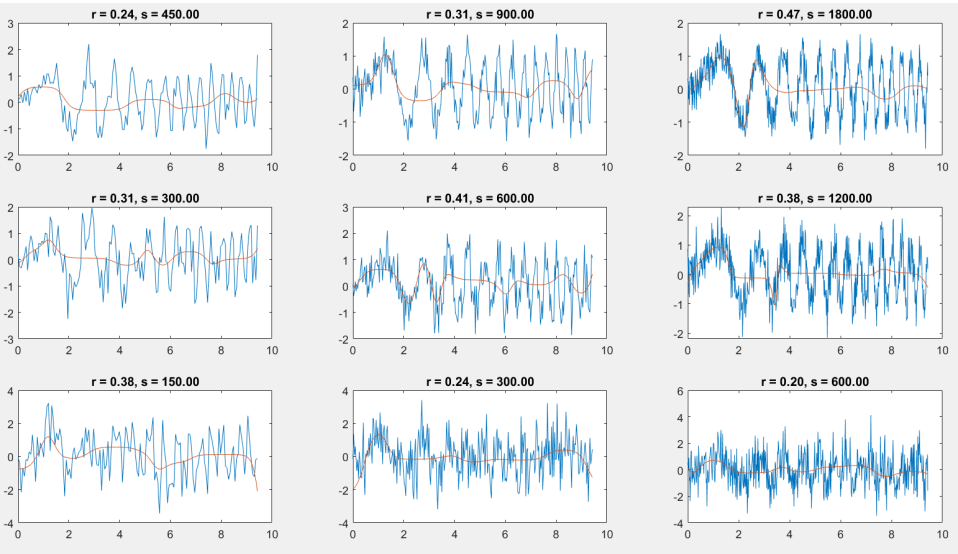


Figure 5: The trainlm function on sample sizes ranging from 75 to 300 and noise levels ranging from 1/3 to 1

Here we notice that as the noise incrases to a mid-range value we see a higher correlation coefficient and a slightly better fit of the data. This may be an indicator that the algorithm is "overfitting" on low noise data. As expected, we also see a slightly better fit for higher sample sizes. Unfortunately, almost all instances of the trainlm function, we also see a fair amount of underfitting, especially on extremely noisy data where the noise, on the same order of magnitude as the function itself, may be obfusicating the underlying function too severely.

2.3

Since this is a deterministic nonlinear function, it is hopeful that the training and validation sets can both be representative of the function as a whole, so random indices were chosen such that approximately 1/3 of the data was in the training, validation, and test sets. Here the data was simply split into even thirds, however a good representative sample could also be achieved by randomly selecting approximately 1/3 of the data for each subset.

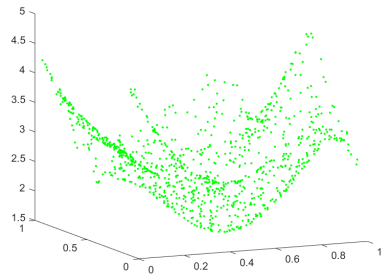


Figure 6: The surface to be approximated by a MLP

In figure 6 we see the full function to be approximated. Note it is highly nonlinear and has several "saddle points". It is fortunate, though, that there appear to be no gratuitous local minima that may interfere with our solution.

Since the 'trainlm' function was most effective on datasets with little noise in the previous example, this algorithm was chosen, and the transfer function chosen was a simple tanh function, as it allows for nonlinear values to be approximated. Because the universal approximation theorem states that any function can be approximated with one hidden layer and a finite number of neurons, the neural network chosen was with one hidden layer and 39 neurons, which had an MSE of 5.80×10^{-7}

In addition to the mean squared error, it can be seen from figure 7, the neural network performed admirably. Since the MSE is slow low, the error level curves are not of high interest. To improve the performance might be to add the squares of the x values as inputs along with the x's themselves, which may help more easily uncover nonlinear features.

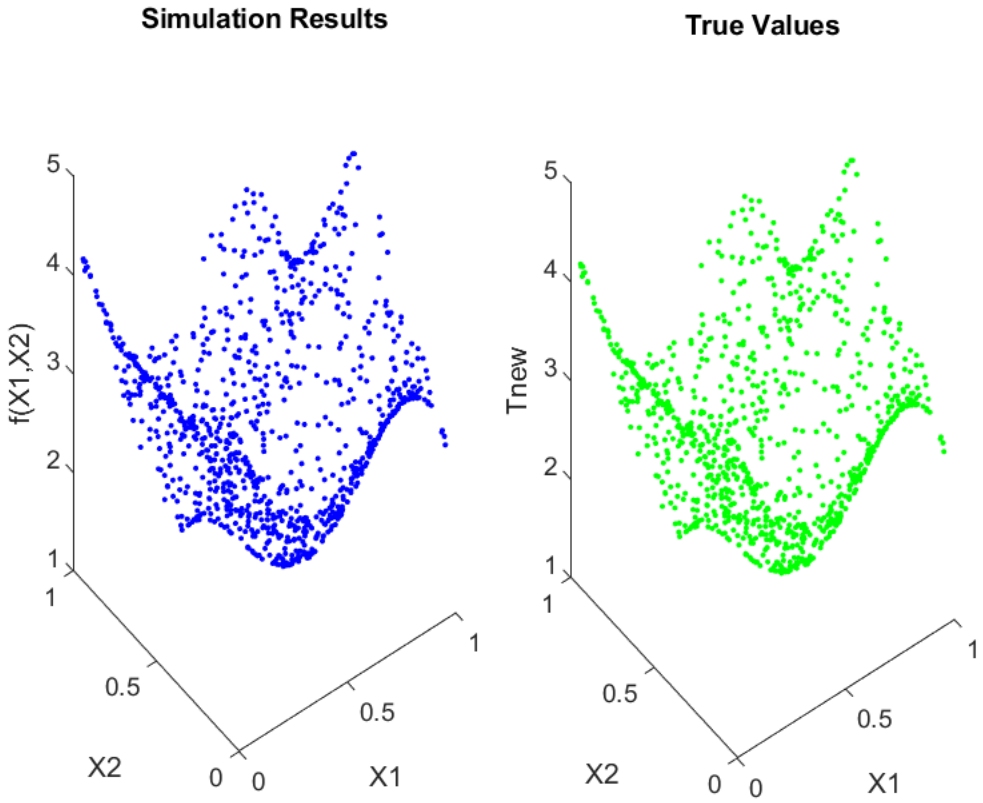
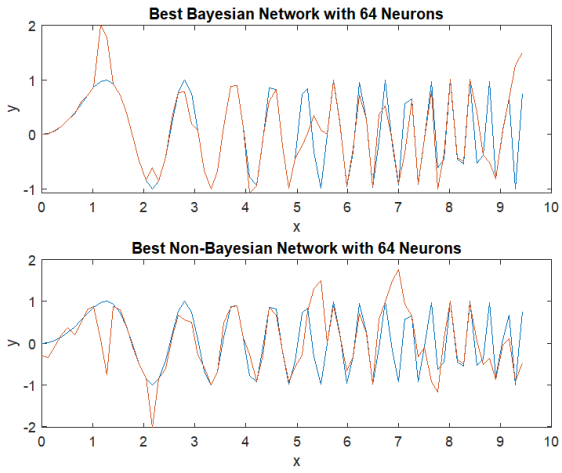
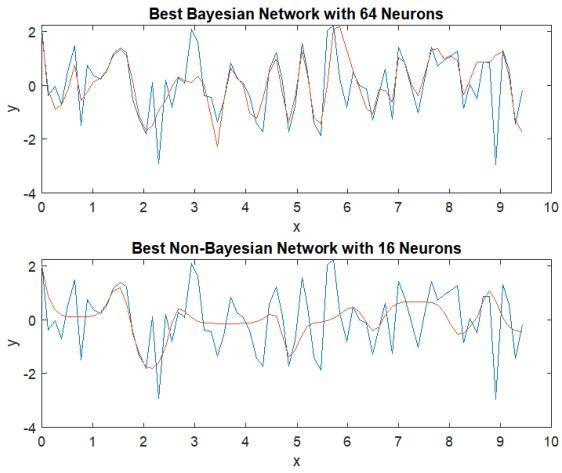


Figure 7: Left, the results of the MLP. Right, the original test surface

3 Bayesian Inference



(a) trainbr and trainlm compared with no noise



(b) trainbr and trainlm compared on data with gaussian noise of variance 1

Figure 8: Side by Side view of best possible bayesian and non-bayesian algorithms

In this example, the depth of a neural network with one hidden layer and neurons from 2 to 2^6 were tested. After finding the number of neurons that minimized the MSE on a randomly chosen validation set, We compared the best found nonbayesian network using the trainlm algorithm, which was shown to be effective against noisy data, and compared it to the solution given by using the trainbr algorithm. It can see clearly from figure 8 that the trainbr algorithm is more effective than the trainlm network for lower noise data, and the MSE reflects that with a value of 8.4×10^{-1} as opposed to the trainlm model with an MSE of 1.0×100^. This is because the bayesian neural network ignores interactions between the model and the bayesian Neural Net only needs enough data to reach a reasonable probability estimate, so it performs better on smaller datasets. Whereas the trainlm performs several iterative optimizations and needs a larger dataset in order to find several minima and account for interaction effects.
There is overfitting, however, on the noisy data. The bayesian classifier matches the noisy data much more closely than the model found using the trainlm function, however it is not more "correct". The optimal solution is actually closer to the one found by trainlm, as the line follows the general trend of the noisy data more than the individual peaks and troughs.

4 The Curse of Dimensionality

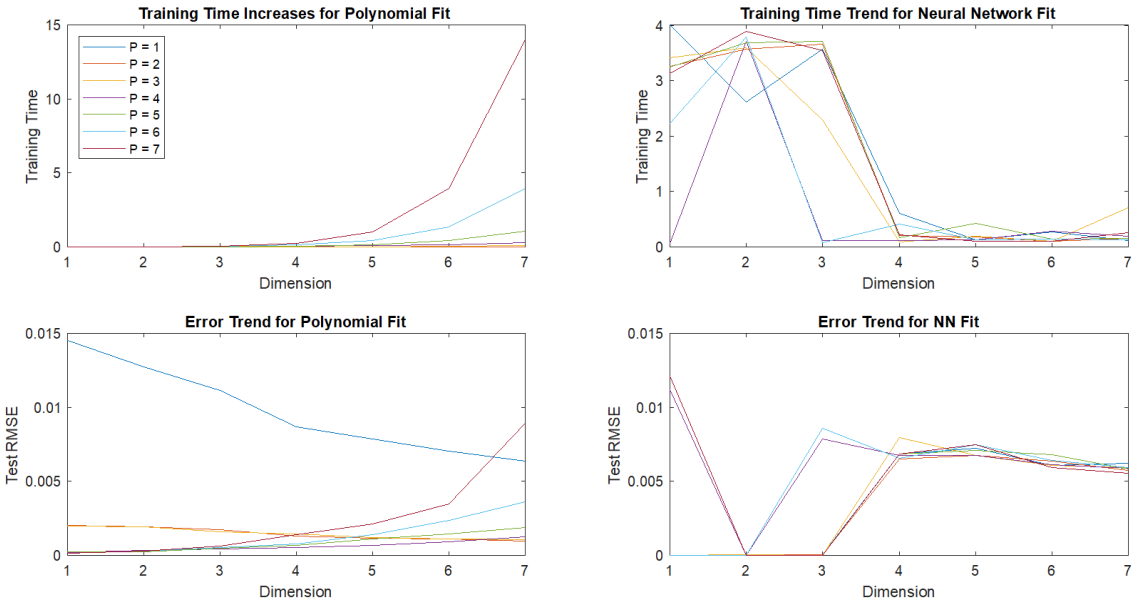


Figure 9: Training time and error comparisons for both polynomial fit and Neural Network architectures

The clear overall message illustarted by 9 is that there is a tradeoff between time, error, and dimension of the data. At low dimension all degrees of polynomial are extremely computationally cheap to fit, as there are fewer parameters (given by the formula $\frac{(d+p)!}{d!p!}$ and they combine linearly. Not only are all polynomials easier to train, all except those of degree 1 have low error in addition. An interesting trend in the graph of the error trend for polynomial fit is that of the first degree polynomial. As the dimensionality increases, the liklihood of sampling data from a far away region where the magnitude of the sinc function is closer to zero and better approximated by a linear function incrases, and so the MSE of the linear classifier decreases. The other dimensions work as intended.

The neural network has a less general trend, but rather a critical point at 4 dimensions. After D=4 the neural network training time and error "level off" or potentially slowly change. This is likely due to a combination of the sparsity of higher dimensional datasets decreasing, or even removing, several weights between the inputs and the hidden layer of the neural network. After four dimensions, the sparsity of the weight matrix to be optimized by backpropagation levels off and there is little difference in training time as many of the new entries at higher dimension are zero.