Neural Networks

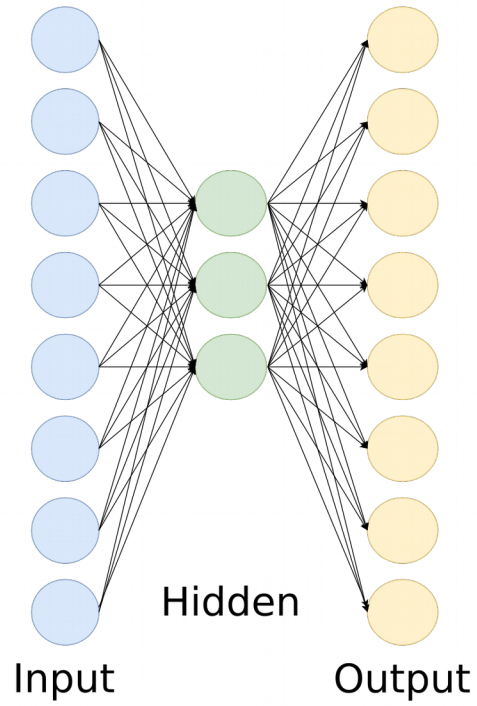
Assignment for the
Advanced Concepts of Machine Learning Course
by Kurt Driessen

Introduction

This report describes the developed network and how to use it. In addition to that it contains information on the learning performance of the network, several measurements and an interpretation of the learned weights. For the development of the code an overall of 17 hours has been spent. The report, benchmarking and evaluation took another two hours.

The Neural Network

The assignment is to produce and train a 3-layer network. On the input and output layer 8 nodes should be used and the hidden layer is supposed to have 3 nodes. Biases should be used as well. For the training all input nodes will be set to zero, except for one. The output is supposed to look like the input. All eight possibilities should be used for training.



For the implementation of the network an important decision was made in the beginning. While a neural network, especially of larger size, can and should be handled with code iterating over the different layers, this implementation is based on a different approach. The updates of the layers are done manually, so that every single step can be observed.

This does not mean, that every input node is handled manually, but simply that the training data is fed in as a vector rather than a matrix of multiple examples.

Initially basic functions like logging or formatting are set up. Furthermore the learning examples are generated and the weights are initialized to random values. In the core of the implementation a loop over the learning examples is executed as many times as the network should learn it. In this loop the forward propagation, the backpropagation and the weight updates take place. These make heavy use of the numpy package and utilize the util package for the sigmoid function and its derivative. The code is a straight forward implementation of the underlying mathematics, where the following abbreviations are used : w = weight, b = bias weights, a = activation, d = delta. An added number refers to the layer of the element.

There are some special quirks that might not show up in other implementations of this neural network. For example this implementation separates a bias vector which is added during forward propagation. This can be done, as the activation of the bias is always one and only the weight changes. The bias therefor is simply added to the final results and moves can move those results either to the left or the right on the coordinate system. As the sigmoid function, also called squashing function, maps all the activations to a range from 0 to 1 moving this result along the abscissa is often a hard requirement. In the given implementation the bias is not part of the weights, but separated to a vector that contains the weights of the bias. Those are added to the activation before they are passed into the sigmoid function. Mathematical identically to adding a 1 to the input vector and keeping track of the weights of the bias as well.

Analysis

First of all the influence of the random initialization of the weights was tested. Those were then fixed using the same seed for the following tests. Next up it was tested, whether shuffling the learning examples had any influence on convergence and finally different learning rates were tested for their convergence. The interpretation follows afterwards.

This PDF document uses images for the following graphs. The graphs have however been created as HTML/JS pages that offer interactive investigation tools like moving around or zooming. If readability is an issue, or more detailed information is desired, please take a look at the graph files as described in the README.MD and explore the data yourself.

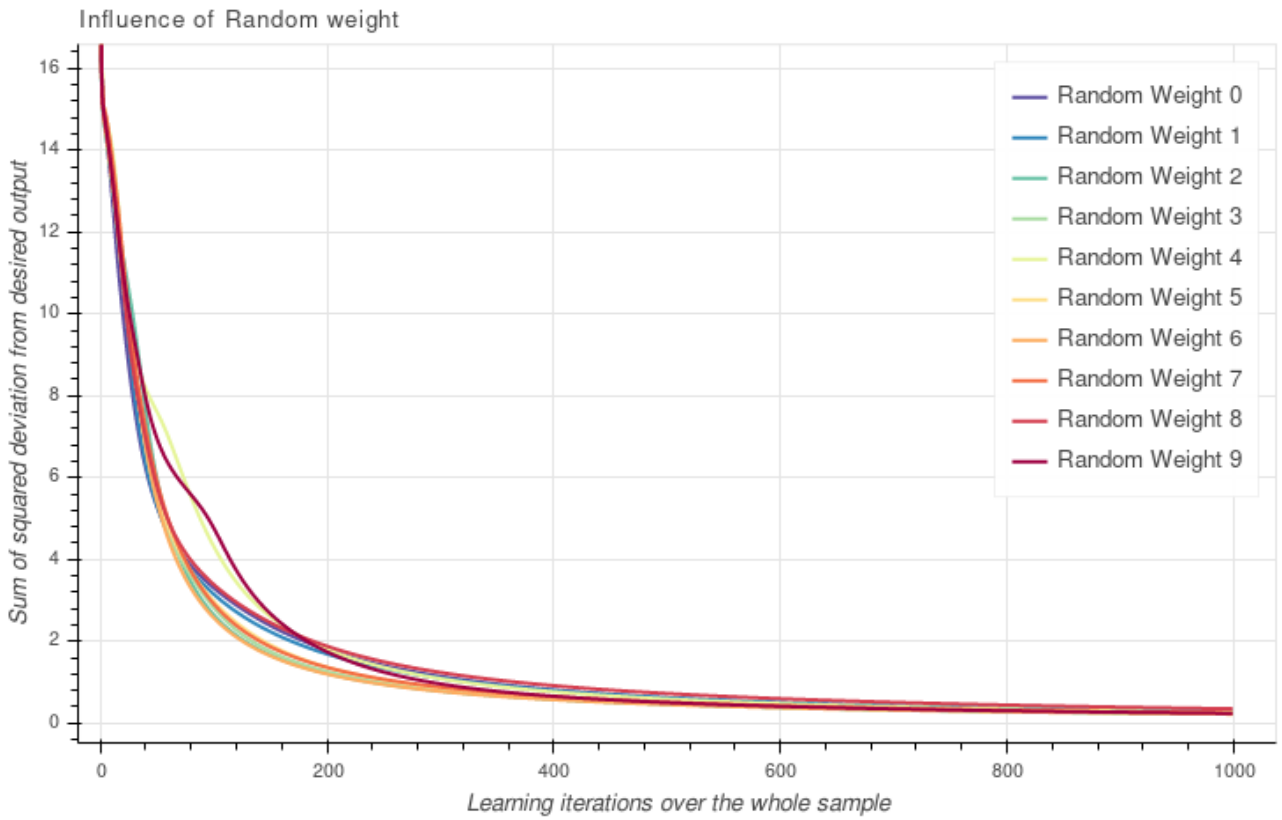


Illustration 1: A learning rate of 0.7 was used for this analysis. While the initialization of the weights has some influence, especially with little learning iterations, the differences are minimal later on.

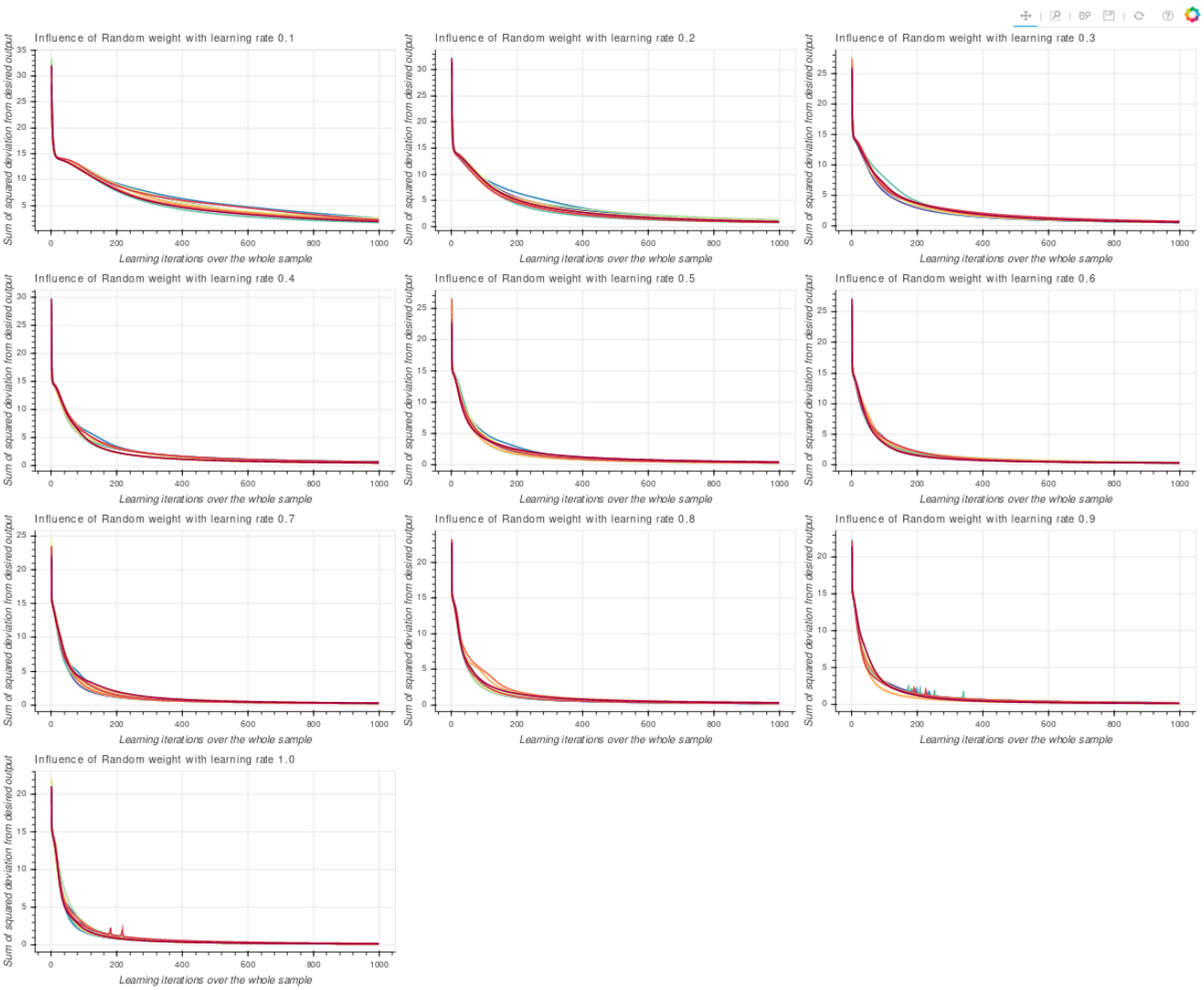


Illustration 2: This graphic shows the influence of the learning rate on the effect of the random weights. With a smaller learning rate, the effect is stretched and a steeper learning rate makes the random initial weights get closer pretty fast. An interesting aspect are the 'spikes' with high learning rates. With some setups the error spikes up again for some iterations. This could be related to over-fitting.

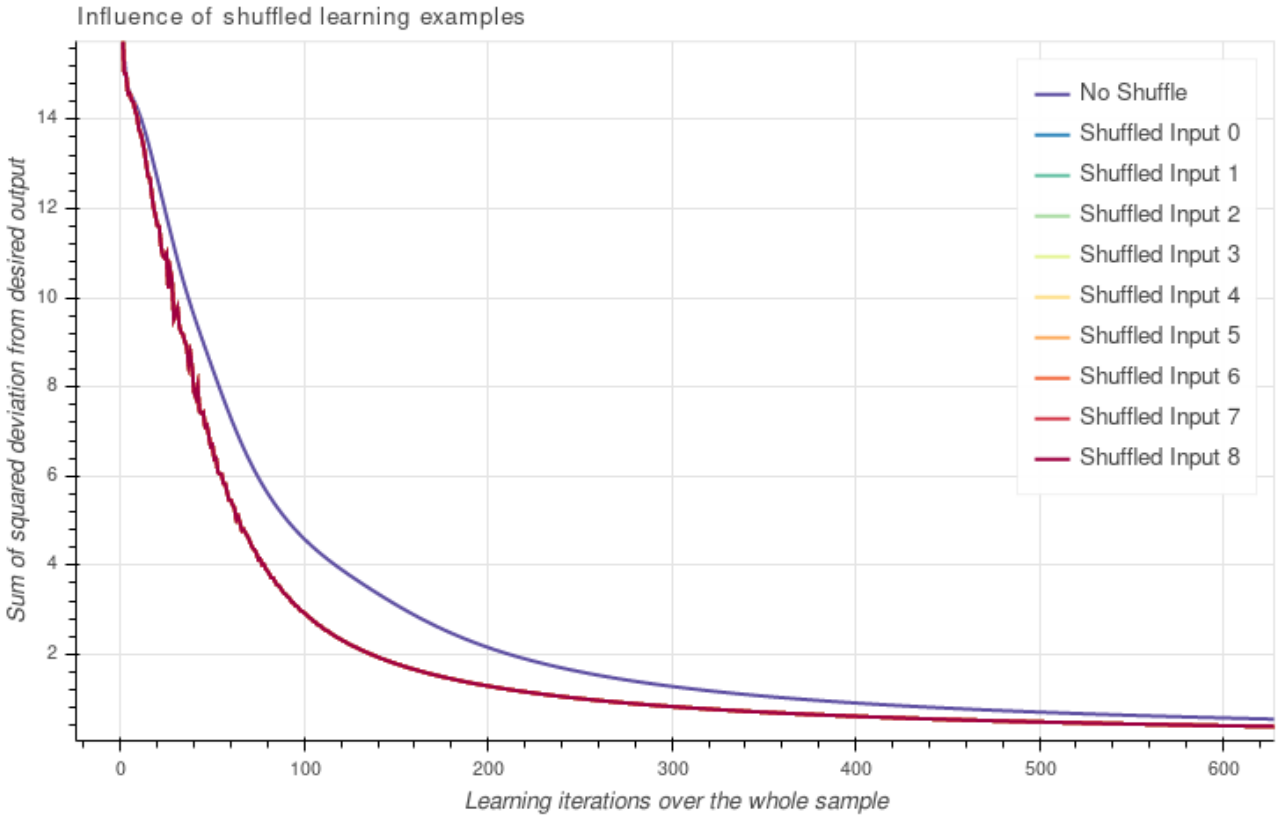


Illustration 3: This graph shows the influence of shuffling the learning examples in every learning iteration. One can quickly see, that the shuffle improves the convergence speed, but make the learning more spiky and unstable. Another very interesting observation is, that the shuffled inputs are all on top of each other. While there were no random weights used, one would assume that the different shuffling of the input would still result in different lines.

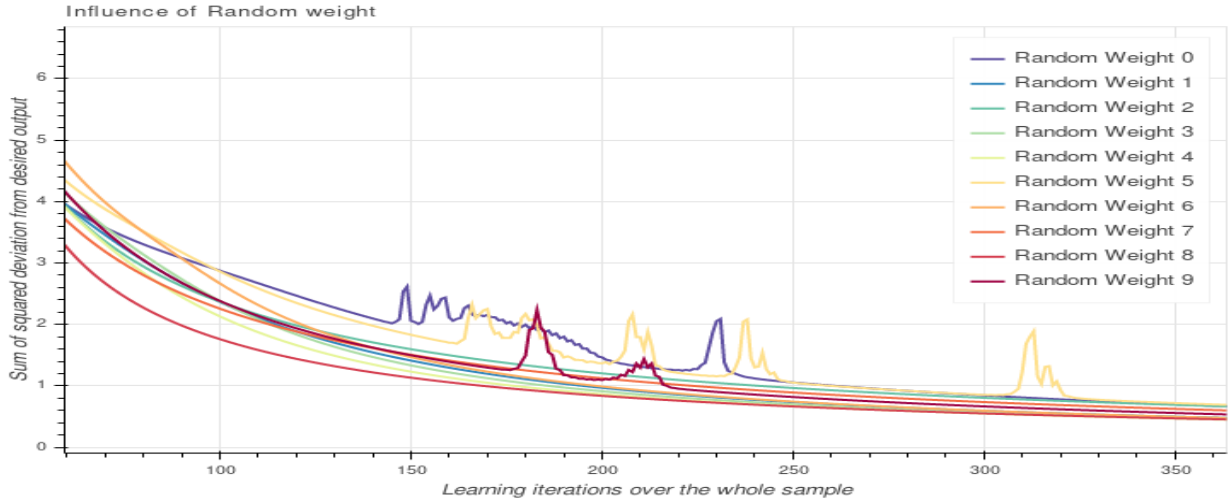


Illustration 4: A more detailed visualization of the spikes that occur with high learning rates. Here a learning rate of 0.9 was used and with several different random weights the issues occurred. Maybe the initialization of the weights can be adjusted to reduce such issues.

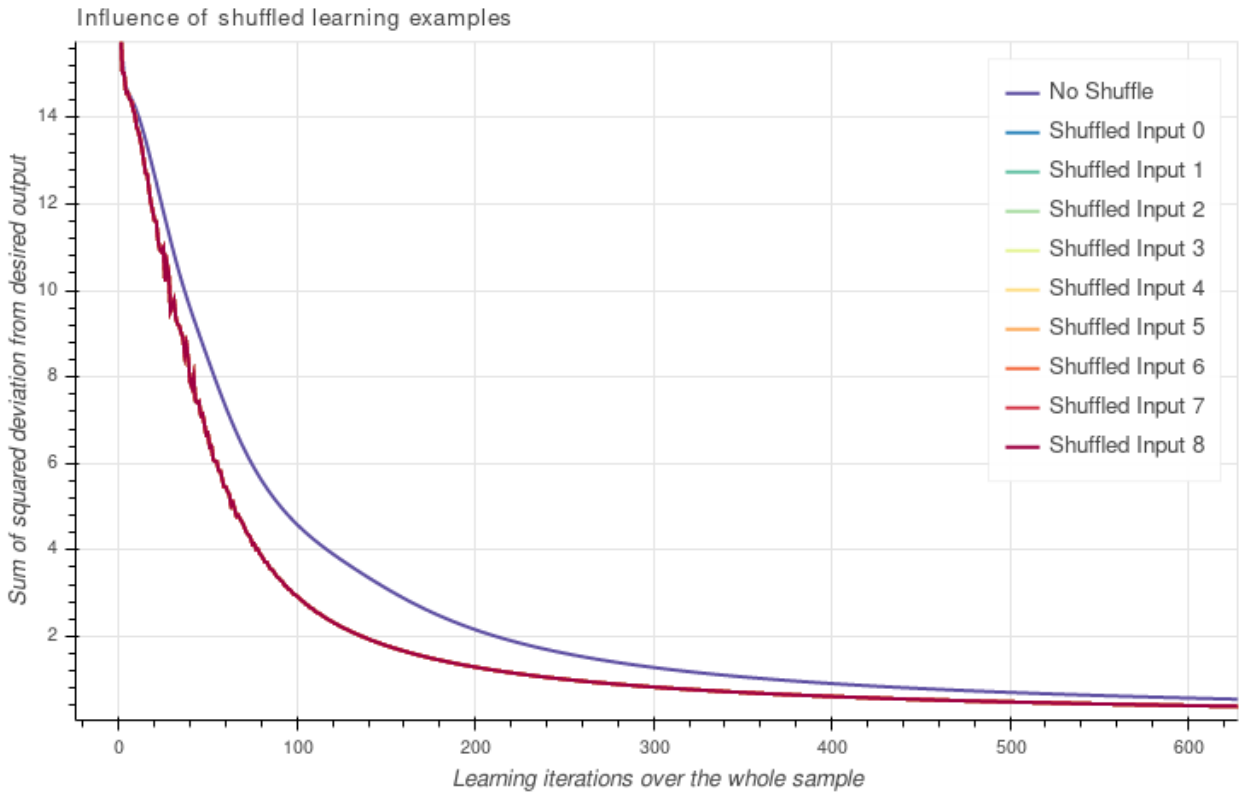
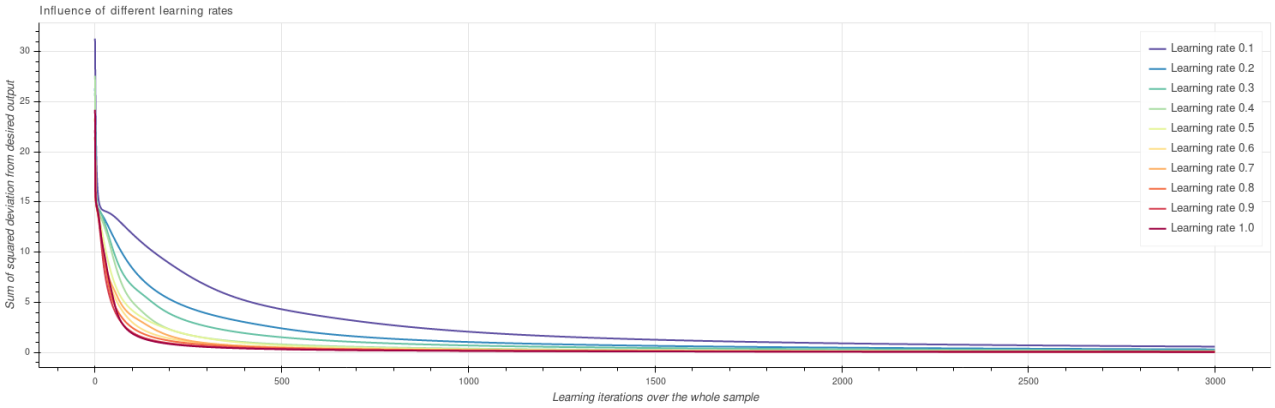


Illustration 5: This graph shows the influence of shuffling the learning examples in every learning iteration. One can quickly see, that the shuffle improves the convergence speed, but make the learning more spiky and unstable. Another very interesting observation is, that the shuffled inputs are all on top of each other. While there were no random weights used, one would assume that the different shuffling of the input would still result in different lines.



Text 1: A comparison of different learning rates using the same weight and no input shuffling. A higher learning rate converges faster. Learning rates higher than one on the other hand, often do not converge at all. Based on the previous examinations it is also to note, that a high learning rate introduces spikes to the lines, that could easily indicate very different results. For this project a learningrate of 0.7 offered a fast convergence and still stable results.

To conclude the results on the one hand several assumptions were met and on the other hand some observations were made, that require additional research. It can be measured, that a higher learning rate increases the convergence speed, and that learningrates bigger than one or smaller than zero do not converge. Also the random initialization of the weights has some influence on the convergence, but with additional training that is diminished. Finally the shuffling of the learning examples increased the conversion speed as well. Those were the expected results and they indicate the proper working of the neural net.

The spikes that are introduced at high learning rates, as well as the alignment of the different shuffled-input graphs are observations that require further research to be explained. In addition to that the analysis was still pretty limited in scope and worked with some fixed values. The random weights were always initialized in the range of -1 to 1 and the measurements on the effect of shuffling the learning examples have only been done for a fixed learning rate.

In general working with the network showed, that convergence was stable with proper characterization, never did any issues occur that would crash the program or result in wildly different results.

A measurement of 100 calls to the neural network with a learning rate of 0.7, 1,000 iterations, random initial weights and shuffling of the learning examples resulted in an average time of 1.04743 seconds, for 100 iterations 0.11034 seconds and for 10,000 iterations 10.231 seconds.

Evaluating the weights and activations of the nodes did not offer any big insights. At first I assumed that the activation of the hidden layer was an indication of the position of the input node. The first hidden node would have a higher activation if the 1 was at the first or second input node, but that turned out to be not true. Another assumption was a correlation between the weights, but that turned out to be wrong as well. Finally I came to the conclusion that the weights can be used as some kind of analogy to how much a specific node ‘likes’ each of their input nodes and the activation of hidden layers are just intentionally developed by the network.