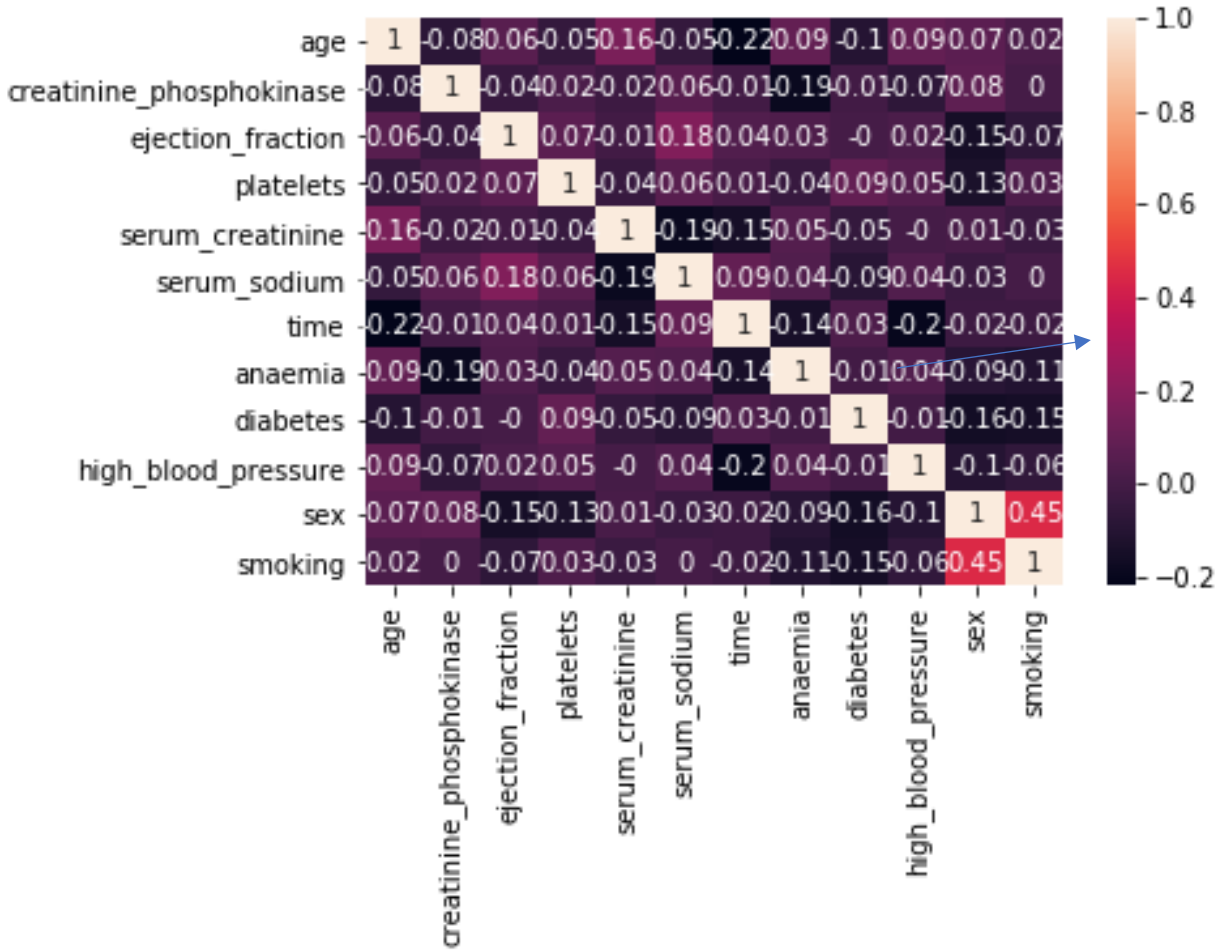
Neural Network Implementation
Metrics, Plots & Proofs

Shreyas Srinivasan

1. **Introduction to Dataset**: The dataset contains 12 predictor variables and 1 output variable DEATH_EVENT. Let us look at how the predictor variables are correlated with each other.



From the plot above, we see that the predictor variables are not strongly correlated amongst themselves. This is a good sign because it means that every attribute will be useful in predicting the output variable. Smoking and sex have the highest correlation (0.45).

Furthermore, note that the following attributes are categorical: "anaemia", "diabetes", "high_blood_pressure", "sex", "smoking", "DEATH_EVENT". Excluding these, we standardize all other variables (which are continuous).

2. **Introduction to RMSProp:** "Root Mean Square Prop" or RMSProp is an enhancement to the standard vanilla gradient descent which uses the exponential moving average of gradients to update weights during every iteration. The main advantage of using RMSProp is that it allows us to combine the power of momentum with bigger learning rates. Usually when momentum alone is used, it is easy to overshoot the global minimum, which is why we use smaller learning rates while dealing working with momentum. However, RMSProp smartly prevents the gradient from blowing up in each iteration, thus enabling us to use momentum with bigger learning rates to converge even faster.

   We start from the ground up to make sense of how this algorithm works. We are trying to learn the following function:

$$\hat{y}_\theta(\mathrm{x}) = \theta_0 x_0 + \theta_1 x_1 + \theta_2 x_2 + \cdots + \theta_\mathrm{n} x_n$$

   Where $\hat{y}$ is the prediction, x_i represent the n attributes (with x_0 always = 1) and θ_0 are the different weights. Since we have n attributes, we have n + 1 parameters (θ) that we need to learn. Intuitively, like in vanilla gradient descent, we need to minimize the errors to learn the best parameters. The error function is:

$$J(\theta) = \frac{1}{2m} \sum_{i=1}^{m} (\hat{y}_\theta(\mathrm{x}^{(i)}) - y^{(i)})^2$$

   To minimize the errors, we follow the standard approach of taking the first derivative. When we update the weights, we need to take the partial derivative with respect to the weight that is being updated. The partial derivative is:

$$\frac{\delta J(\theta)}{\delta \theta_j} = \frac{1}{m} \sum_{i=1}^{m} (\hat{y}_\theta(\mathrm{x}^{(i)}) - y^{(i)})\, x_j^{(i)}$$

   For RMSProp, we define the weight update as follows:

$$\theta_{j+1} = \theta_j - \frac{\alpha}{\sqrt{v_j + \varepsilon}} \left[\frac{\delta J(\theta)}{\delta \theta_j}\right]$$

   In the above equation, alpha is the learning rate which controls how fast the equation grows. v_j is a constant that needs to be calculated using the momentum, and epsilon prevents us from dividing by 0. v_j can be interpreted as moving average of the gradients, and is defined as:

$$v_j = \beta v_j + (1 - \beta) \left[\frac{\delta J(\theta)}{\delta \theta_j}\right]^2$$

   The default values - recommended by the authors of the original paper on RMSProp – for the parameters are $\alpha = 0.001$, $\beta = 0.9$ and $\varepsilon = 10^{-6}$.

3. **Comparing models fitted using different parameters:** The modelComparer class fits neural networks using the RMSProp optimizer on many sets of parameters and outputs the metrics in the form of a data frame. It can print the output to the console or write the output to an Excel file. The metrics it returns are the Scaled Training Error, Scaled Test Error, Training Accuracy, and Test Accuracy. A screenshot from the Excel sheet which tracks different trials is given below:

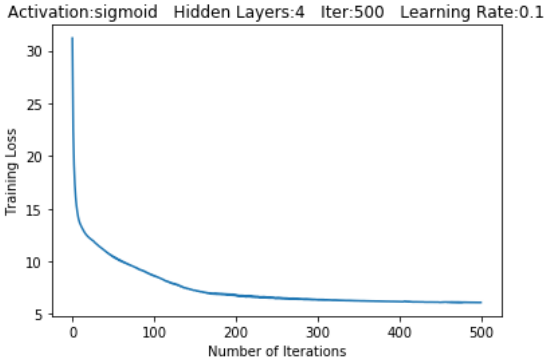
	Activation	Hidden Layers	Iterations	Learning Rate	Scaled Training Error	Scaled Test Error	Training Accuracy	Test Accuracy
1	sigmoid	4	500	0.1	0.027203492	0.078392221	0.941964286	0.8
2	sigmoid	4	500	0.25	0.0208093	0.111138434	0.955357143	0.76
3	sigmoid	6	500	0.25	0.013404205	0.087078573	0.96875	0.8
4	sigmoid	4	1000	0.25	0.026968115	0.099995562	0.933035714	0.76
5	tanh	4	500	0.1	0.04522267	0.071719457	0.910714286	0.866666667
6	tanh	4	500	0.25	0.101865809	0.117744708	0.8125	0.706666667
7	tanh	6	500	0.1	0.041136224	0.105889867	0.9375	0.72
8	tanh	6	500	0.25	0.07747564	0.126352619	0.767857143	0.653333333
9	relu	4	250	0.25	0.081391408	0.073010285	0.75	0.746666667
10	relu	4	250	0.1	0.079258	0.079584245	0.772321429	0.773333333
11	relu	4	500	0.1	0.082083326	0.088961783	0.772321429	0.786666667
12	relu	6	500	0.1	0.089388377	0.086995704	0.732142857	0.706666667

Observations:

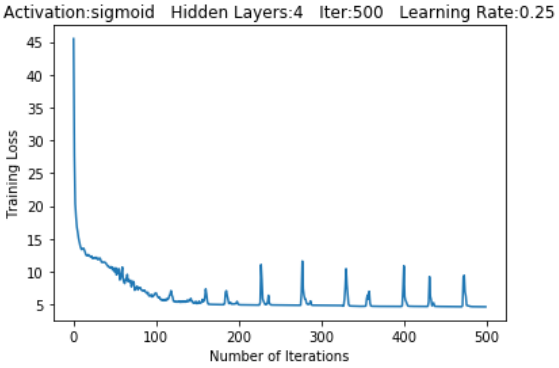
- As expected, a higher number of iterations generally helped us decreased Scaled Training Error, but at the cost of Scaled Test Error. The most likely cause is our model overfitting the training dataset, which then makes it difficult to generalize on other datasets.
- Overall, we see that tanh activation performed poorly on this dataset, while sigmoid performed the best. However, it is ironic that in this instance, the best model is trial 5 which utilizes tanh activation. Trial 5 gave us a Test Accuracy of 0.87 and a Training Accuracy of 0.91. Note that while Trials 1 and 2 had higher Training Accuracies of 0.94 and 0.96, they had lower Test Accuracies at 0.8 and 0.76. Once again, this tells us that a very high training accuracy is a sign of overfitting.
- The learning rate controls how fast the model learns. Intuitively, a higher learning rate means the model will have a higher Training Accuracy (and a lower Scaled Training Error). For example, in trials 1 and 2, we saw that increasing the learning rate while holding all other parameters constant increased Training Accuracy from 0.94 to 0.96. As discussed before, this could lead to overfitting.
- Increasing the number of hidden layers had different effects for different activation functions. For sigmoid, it increased Training Accuracy (trial 2 vs. 3), for tanh it increased Training Accuracy (trial 5 vs. 7), but for relu it decreased training accuracy (trial 11 vs. 12).

Generally, relu outperforms tanh and sigmoid since it overcomes the vanishing gradient problem. However, since we are not performing very deep learning in our case, it is reasonable to assume that tanh/sigmoid will do the job well. As expected, tanh had the best model and on average sigmoid consistently outperformed relu and tanh

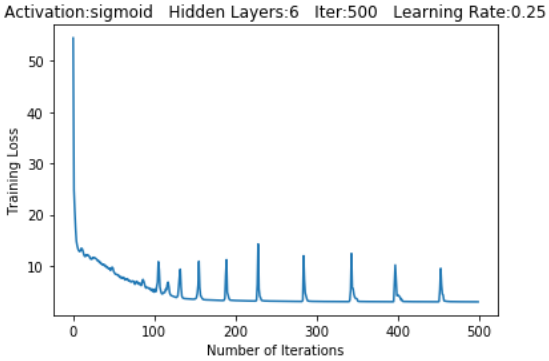
Plots for the above trials are given below:



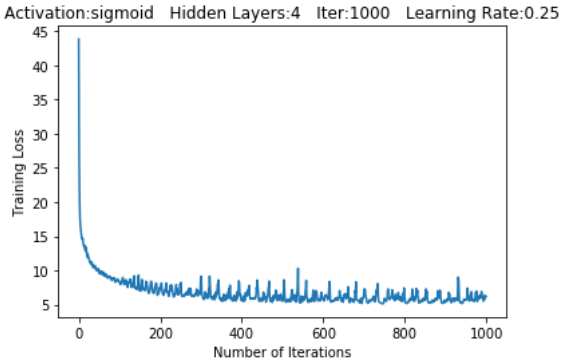
Trial 1



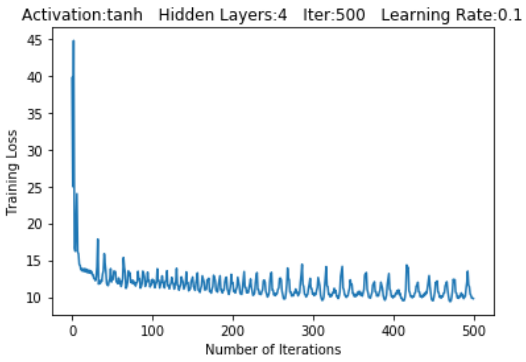
Trial 2



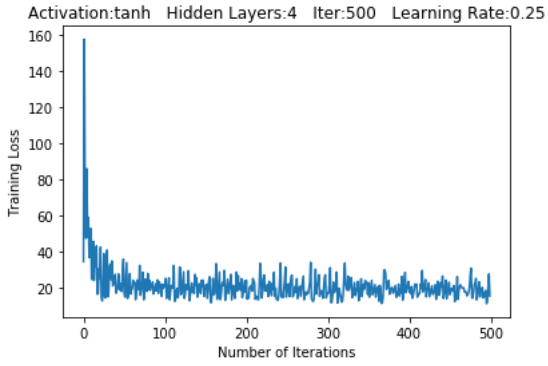
Trial 3



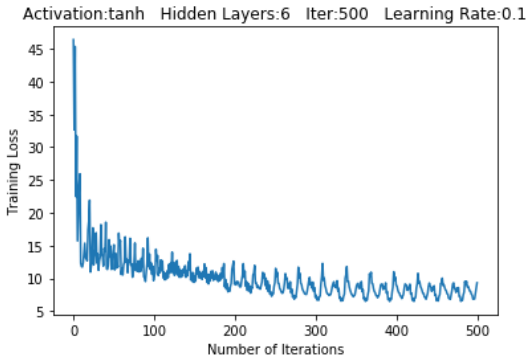
Trial 4



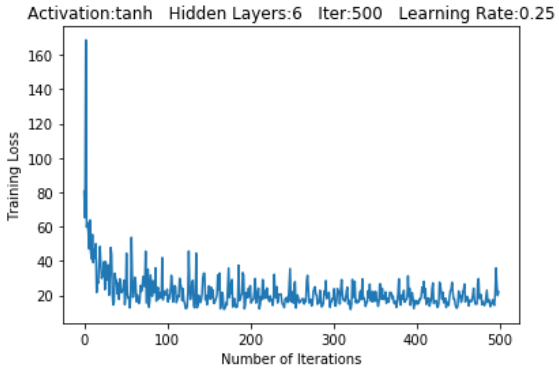
Trial 5



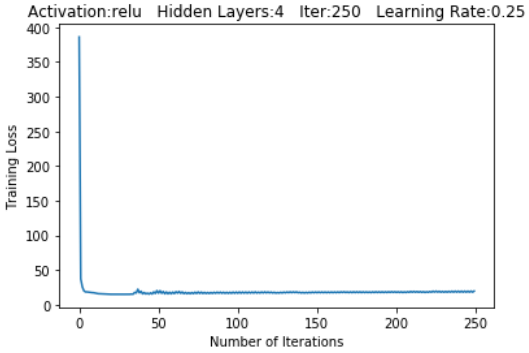
Trial 6



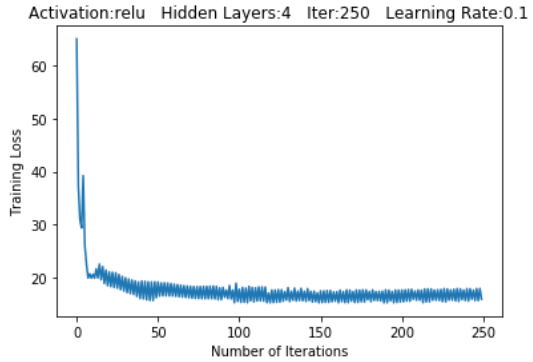
Trial 7



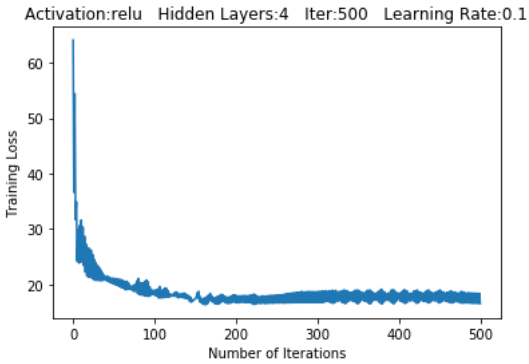
Trial 8



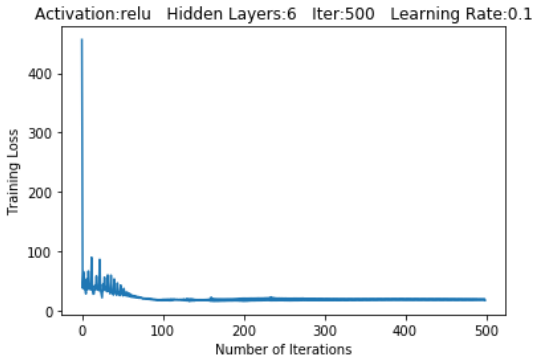
Trial 9



Trial 10



Trial 11



Trial 12

We see that the “grooves” tend to increase with the increase in number of hidden layers. Nevertheless, we see that most of the learning takes place within the first 100 iterations. After which, there are only incremental improvements to decrease error. In some places the weights do not converge, they oscillate around the optimum solution.

Overall, Trial 5 is the best model in this instance. We must keep in mind that weights are randomly initialized, and that plays a huge role in how well the model fits the data. If we run the modelComparer class again, we might get different results.