ITI106 Assignment

21A466A Ong Joo Kiat Kenneth
21A471K Shee Jing Le

1. Introduction and Background, Conclusions

TensorFlow is a deep learning library inspired by how the brain works. Deep learning is a branch of Machine Learning which is a branch of Artificial Intelligence. As we branch into deep learning, the amount of automation increases and along with it, the sophistication. An end-to-end open-source platform, developed by google, it allowed developers to easily build and deploy ML powered applications. Functioning like a brain, it has multiple layers of processing between the input and output layers. In this project, we will focus on fully-connected Feedforward Neural Networks (FNN).

2. Data Pre-processing

We are using the MAGIC Gamma Telescope Data Set. The data is generated through a Monte Carlo process to simulate registration of high energy gamma particles in a ground-based atmospheric Cherenkov gamma telescope using the imaging technique.
Attribute Information:
1. fLength: continuous # major axis of ellipse [mm]
2. fWidth: continuous # minor axis of ellipse [mm]
3. fSize: continuous # 10-log of sum of content of all pixels [in #phot]
4. fConc: continuous # ratio of sum of two highest pixels over fSize [ratio]
5. fConc1: continuous # ratio of highest pixel over fSize [ratio]
6. fAsym: continuous # distance from highest pixel to center, projected onto major axis [mm]
7. fM3Long: continuous # 3rd root of third moment along major axis [mm]
8. fM3Trans: continuous # 3rd root of third moment along minor axis [mm]
9. fAlpha: continuous # angle of major axis with vector to origin [deg]
10. fDist: continuous # distance from origin to center of ellipse [mm]
11. class: g,h # gamma (signal), hadron (background)

For 4, 5: Since they are in ratio, no need for scaling. For 9: Since in degrees in major axis with vector to origin, reasonable scaling 0-90 Others: min-max scaling (split train-test/validation set before applying scaling)

Using the train set, we found out the best way to scale and apply the scaling to the validation and test set after splitting the train-validation-test into 50-25-25.

We have chosen a classification set of data, so this is a 1D array. Unlike an image which is a 2D array, there is no need to flatten.

The simple classification accuracy is not meaningful for this data, since classifying a background event as signal is worse than classifying a signal event as background. For comparison of different classifiers an ROC curve has to be used. The relevant points on this curve are those, where the probability of accepting a background event as signal is below one of the following thresholds: 0.01, 0.02, 0.05, 0.1, 0.2 depending on the required quality of the sample of the accepted events for different experiments. Thus for the performance measurement metrics, we chose AUC. AUC is the Area Under the ROC Curve. The ROC curve is plotted as True positive (Y-axis) to False positive (X-axis). The AUC places emphasis on minimising false positives while maximising true positives . By maximising AUC, we optimise the model.

3. Experiments

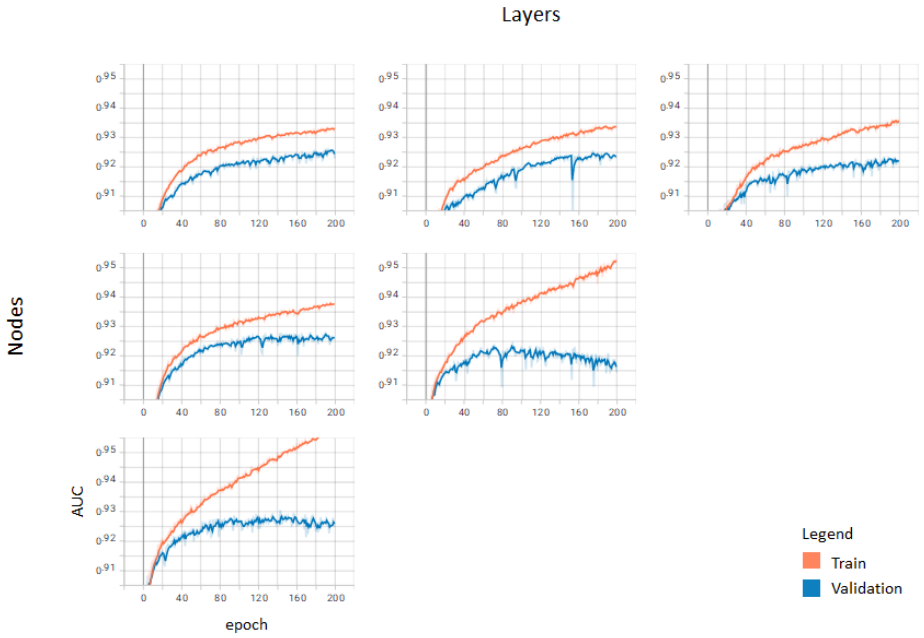
3.1 Number of layers and neuron units

We run initial experiments in Table(1), each model will be run with an “adam” optimizer, without regularization, and “ReLu” activation functions for the hidden layers. Each experiment will be run for 200 epochs.

Nodes\Layers	3	5	7
16	x	x	x
32	x	x	
64	x		

Table(1). Experiments with number of layers and nodes

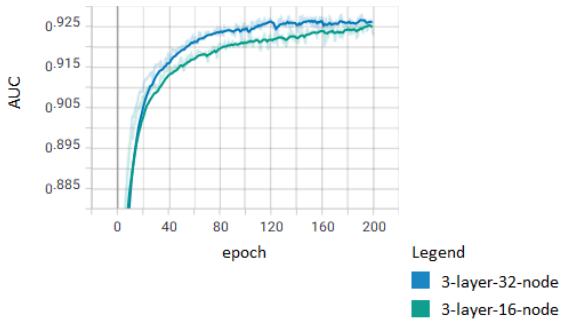
We expect a model with a higher number of nodes and layers to have a higher risk of overfitting. However, due to the increased capacity, there is also a chance for learning a more accurate representation of the hidden features (if the data/problem allows).



Figure(1). Results with different number of layers and nodes

First of all, the evidence of overfitting is very strong. Overfitting increases when either number of nodes or number of layers increases.

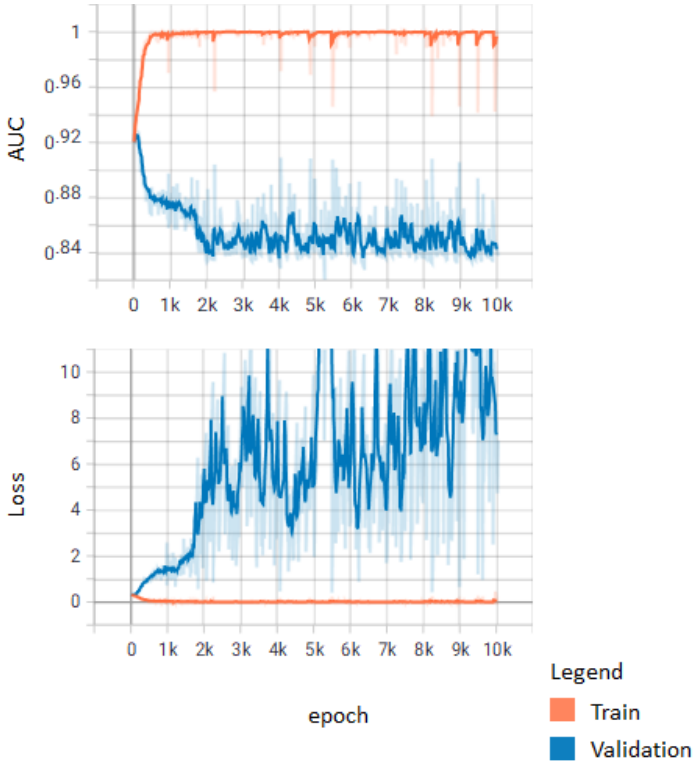
For this set of data, it would seem that models with 16 nodes are slower to converge as compared to models with 32 nodes. Figure(2) shows 2 results from 3-layer-16nodes and 3-layer-32nodes in a single graph. It can be seen that the model with 32 nodes converged (AUC value plateaus) and the model with 16 nodes have not actually converged after 200 epochs.



Figure(2). Results comparison with 3-layer-32-node model and 3-layer-16-node model

While at 200 epoch, the 32 node model performs better than the 16 node model, as the 16 node model has not converged yet, a comparison of the performance of these 2 models will be done with early stopping to assure convergence in section 4.

To investigate if a more complex model/representation will be able to better generalize[2][3] to the problem, a model with 7-layers and 64-nodes will be allowed to run for 10k epochs. Results are shown in figure(3).

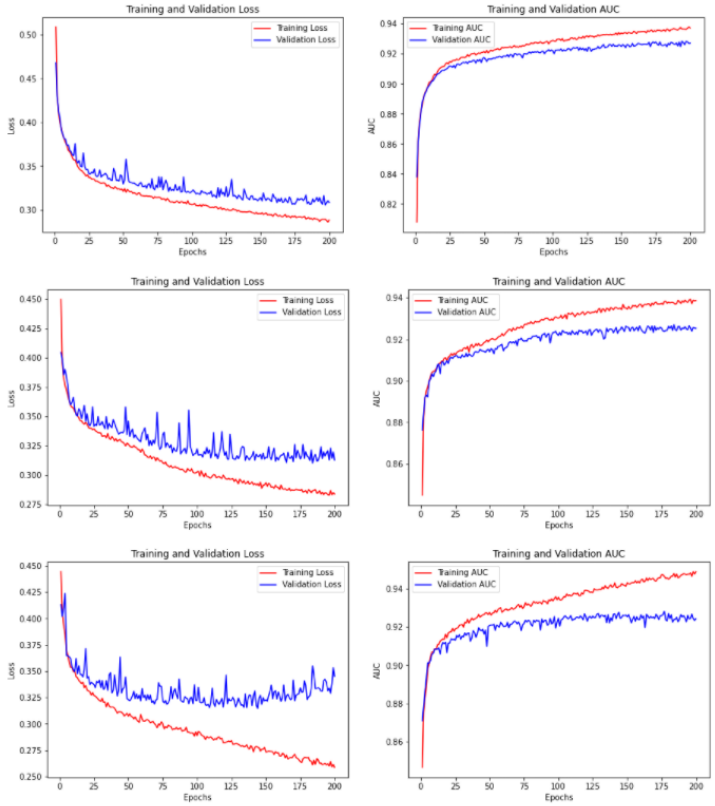


Figure(3). Results of 7-layer-64-node model for 10k epochs

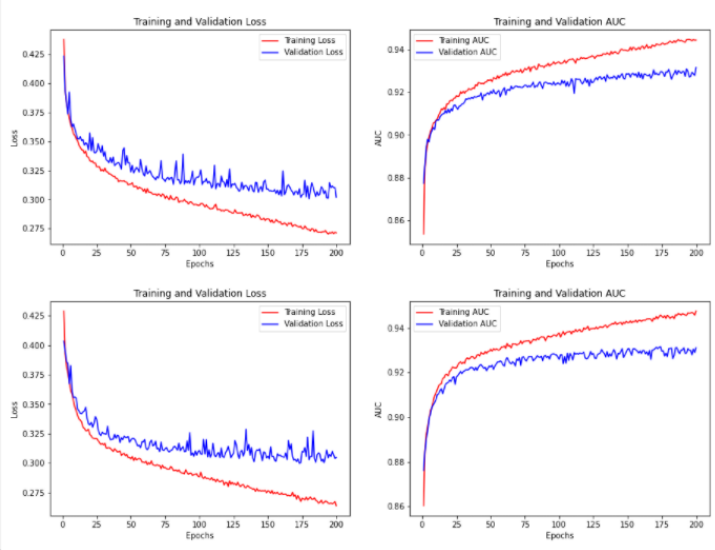
Overfitting is to be expected with a larger model, however, the deep double descent (epoch based)[2] is not being observed after 10k epochs. Further investigation is needed to investigate if: 1. there are not enough epochs to reach the second dip; 2. the model is not large enough to exhibit this phenomenon. It is shown[2] that the model has to be large enough, and the number of epochs at which the phenomenon occurs also depends on the size of the model; 3. the dataset itself does not have a more accurate representation beyond what we found.

3.1.1 Layers vs Neurons

Keeping all else constant, we tested the layers change only vs the neurons change only. we ran 10 hidden layers with 32 neurons and 1 hidden layers with 10 multiples of 16 neurons. At 3 layers, declining loss function is showing but for neurons, it only happens at the last loop of 160 neurons. This shows a parameter of about 2500 for layers and 5000 for neurons for optimal results at 200 epochs.



Figure(4). 1-3 hidden layers in ascending order

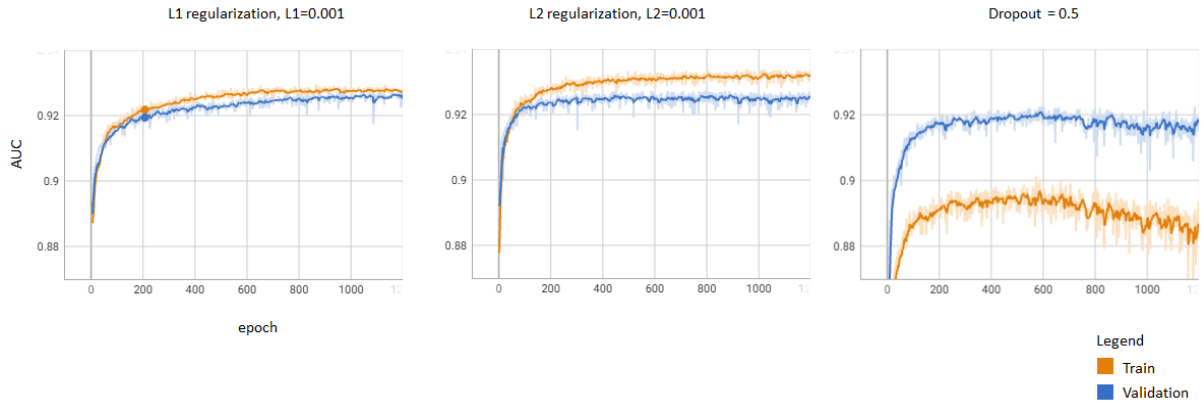


Figure(5). 1 hidden layer, 144 and 160 Neurons in ascending order

3.2 Regularization

Overfitting occurs when the model fits the training data too much and does not accurately predict unknown data (testing data).

To investigate the effect of regularization, we choose the 7-layer-64-node model because it exhibits the strongest overfitting. 3 types of regularization are used, L1, L2 and dropout.

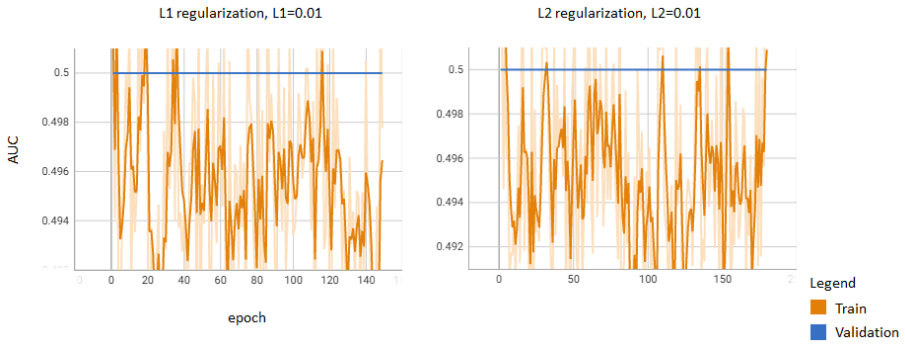


Figure(6). Effects of different types of regularization

Both L1 and L2 exhibit similar effects of regularization, with L1 having a slightly better performance than L2, in both preventing overfitting as well as the results (AUC). Due to the nature of "dropout" regularization, the results from the validation set will be higher than the results from the train set. It is because the results from the train set are calculated with the nodes dropped-out, which means that the results are calculated with an incomplete model, whereas, the results from the validation set is calculated from the full model with all nodes re-activated. Hence, the performance of reducing overfitting cannot be compared in this way.

All 3 regularization methods resulted in performance (AUC) lower, even though not much, than that of lower smaller models. However, all 3 methods succeeded in preventing overfitting of a model which performs very much otherwise.

One thing of note is that when we choose the regularization rates as 0.01 (which is the default for both L1 and L2), the model failed to learn properly. It has also been noted that it also depends on the weights that have been initialized. There was an instance of initialization that the model managed to converge properly even with the rate of 0.01. As the regularization in this section is being applied to all layers, it remains to be investigated if there is a more elegant solution other than reducing the regularization.

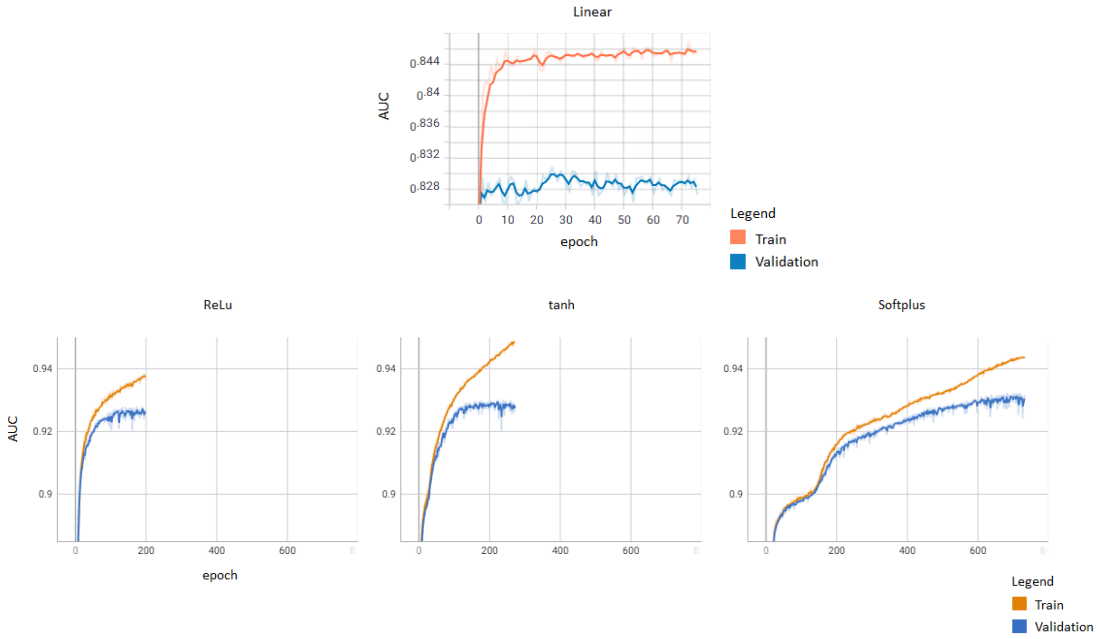


Figure(7). Regularization causing model to not learn

3.3 Activation functions

The Sigmoid is used for the output layer of activation function for binary classification. Up until here, the activation function for the hidden layers has been ReLu, and it has been performing in

a stable way. To investigate the effects of activation functions, here we use a 3-layer-32-node model with the following activation functions: ReLu, Linear, Softplus, tanh.



Figure(8). Effects of different activation function

When the activation function is linear, the performance metric (AUC) converges to a lower value at about 0.83. This shows that the dataset has to be represented beyond a linear relationship with the features. By having a linear activation function, the model becomes linear; the model will be in the form of y = sigmoid(Σwx + c), where w is a scalar which is a function of multiple weights in the model.

With ReLu as the baseline, we experimented with tanh as well as softplus, which is meant to be an improvement over ReLu. Both of the experiments were early-stopped to make sure convergence. Since the model used is a 3-layer-32-nodes model, there are no signs of overfitting, therefore that will not be a factor in the results.

While tanh (and sigmoid) is mainly used for the output layer for classification uses, when used as the activation function for the inner layers, it performs quite similarly (at least for this model and dataset) to the more popular ReLu.

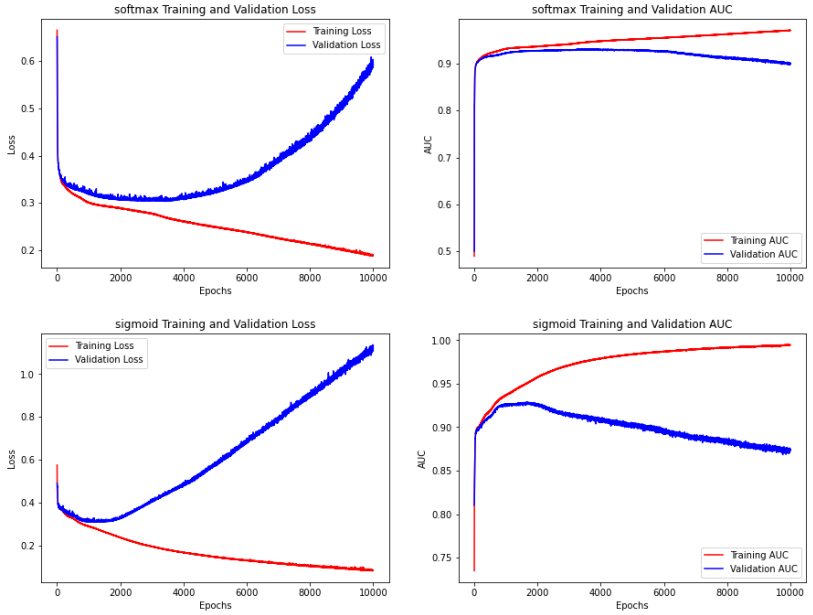
Softplus, being proposed as an improvement over ReLu, does seem to help achieve higher performance, at the expense of the learning speed. This might be due to the less steep curve at around x=0, but this needs to be verified. Another interesting point to note is the elbow at around 150 epoch where the curve changes. Further studies have to be done.

Simple testing of 9 activation functions with 2 hidden layers at 32 and 16 nodes, input layer at 64 nodes, total of 3329 parameters, 300 epochs show that the loss function on the validation set does hit a minimum quite early in the epochs. Softmax and Sigmoid show potential to produce better results given more epochs. Further extension of these 2 activation functions to 10000 epochs does not yield better validation AUC than running a simple tanh, softsign, elu.

A surprising find is ELU does perform better than Relu. Looking at the documentation, the difference between Relu and Elu is the negative inputs. Our features do have negative inputs. The trade off here is the time needed. Notice relu needs 139 epochs which elu needs 251 epochs.

```
['softmax', 299, 0.3302077651023865, 0.9170950651168823, 0.3387663662433624, 0.9125328063964844]
['relu', 139, 0.2814963161945343, 0.9399942755699158, 0.30799442529678345, 0.9280372858047485]
['tanh', 160, 0.28282997012138367, 0.9397354125976562, 0.30146265029907227, 0.9302423596382141]
['sigmoid', 295, 0.34381136298179626, 0.9089289903640747, 0.3506508469581604, 0.9063679575920105]
['linear', 111, 0.45040372014045715, 0.8459786176681519, 0.47102829813957214, 0.8277087211608887]
['softplus', 297, 0.3201388418674469, 0.9212273955345154, 0.3324286937713623, 0.9142693281173706]
['softsign', 251, 0.2732381224632263, 0.9437079429626465, 0.29396021366119385, 0.9340373873710632]
['selu', 287, 0.2768750786781311, 0.9426751136779785, 0.2966492176055908, 0.9327892661094666]
['elu', 251, 0.2749277353286743, 0.9424397945404053, 0.29095324873924255, 0.934605598449707]
```

Figure(9). 300 epochs chosen based on least validation loss, left to right - activation function, epochs, loss, AUC, validation loss, validation AUC



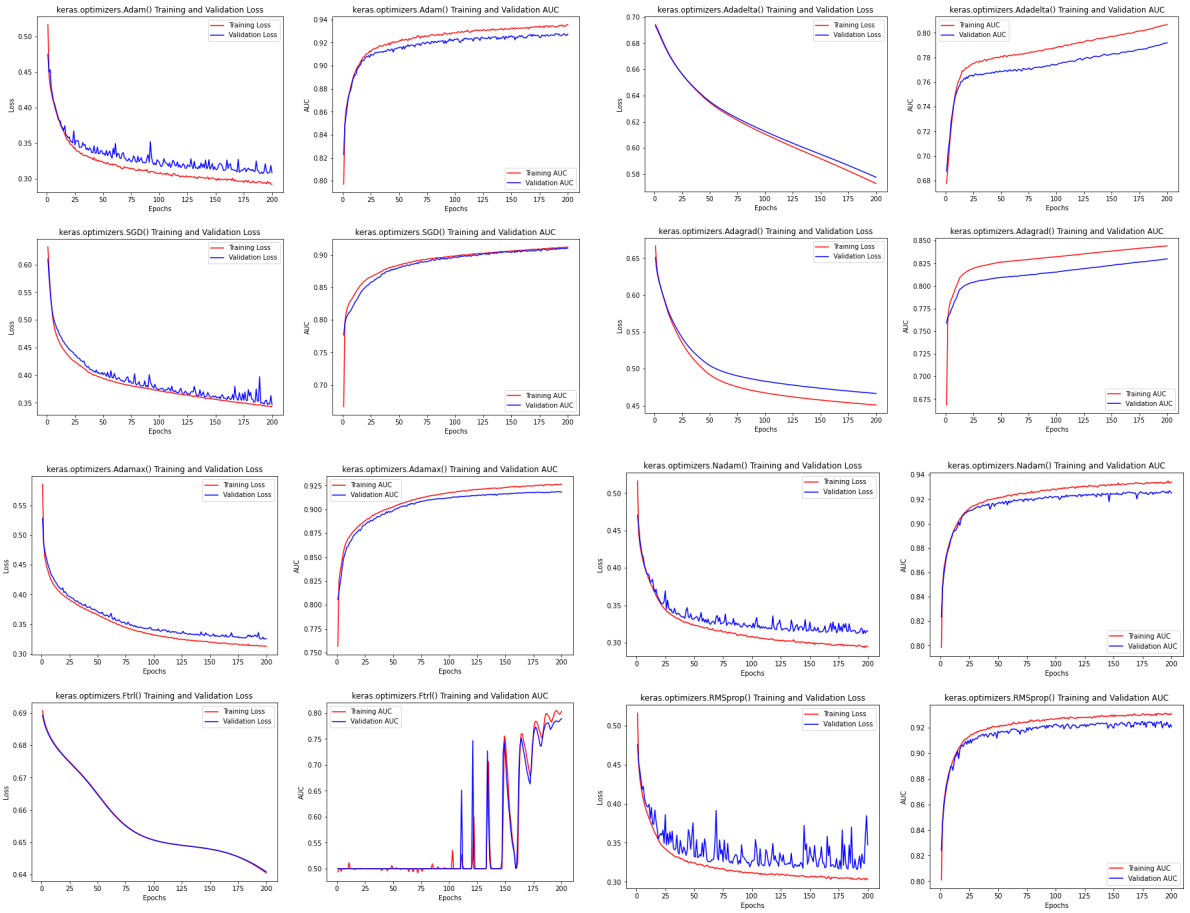

```
['softmax', 3416, 0.2699788808822632, 0.9445530772209167, 0.30319273471832275, 0.9307671785354614]
['sigmoid', 1123, 0.2828584909439087, 0.9393510818481445, 0.30895891785621643, 0.9262967109680176]
```

Figure(10). 10,000 epochs chosen based on least validation loss, left to right - activation function, epochs, loss, AUC, validation loss, validation AUC

3.4 Optimization algorithms

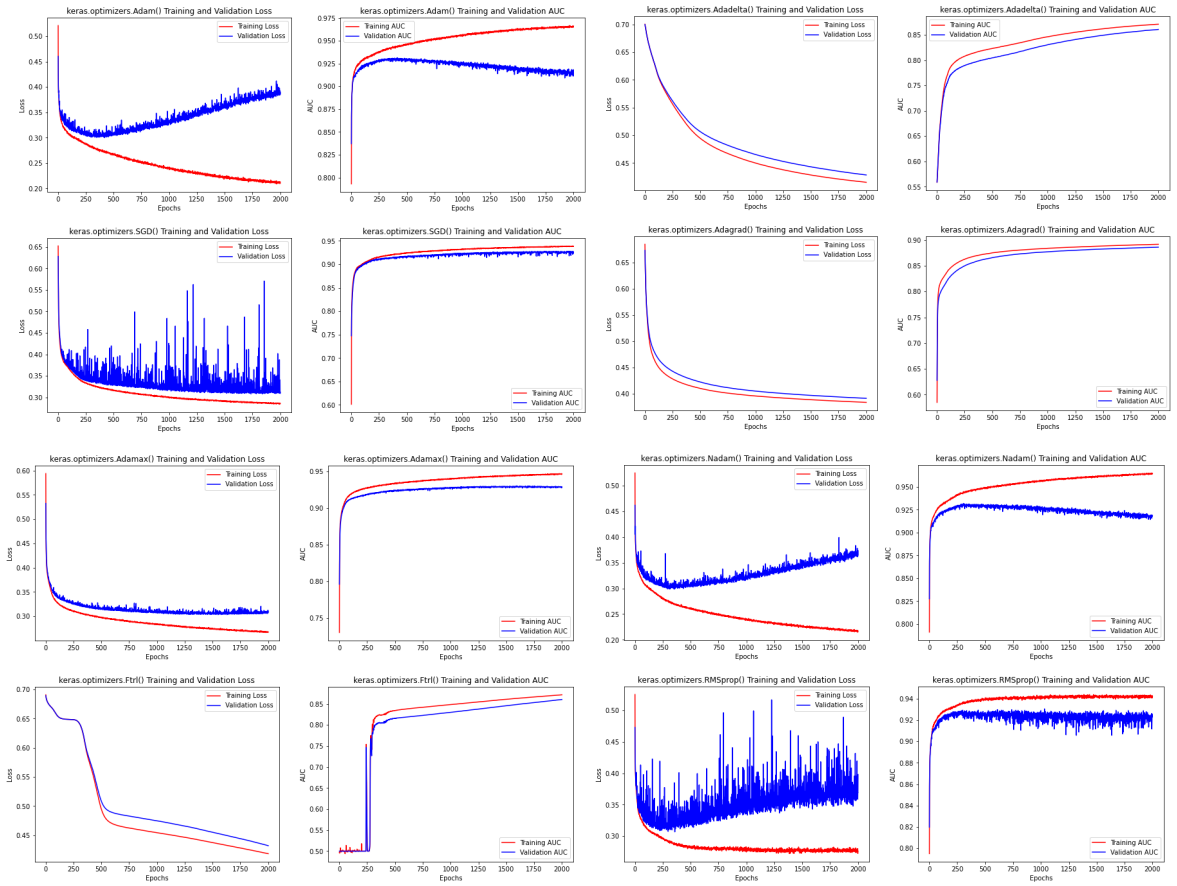
The 2 most popular optimizers are SGD and ADAM. Initial testing is done with ADAM with recommended parameters because of its speed. But that does not mean it is the best optimizer.

Using 1 hidden layer and 32 neurons at input and hidden layer, 1441 parameters, we tested the 8 optimizers using default hyperparameters. At 200 epochs, SGD showed promising results. there are no overfitting and there the results showed consistency in improvement. The next most consistent optimiser is the ADAM. As epochs increase, there is some divergence between training and validation set. However, results are still improving - we will try with more epochs with early stopping.



Figure(11). optimizer comparison <u>200 epochs</u>, 1 hidden layer, 32 neurons, 1441 parameters

Using 1 hidden layer and 32 neurons at input and hidden layer, 1441 parameters, 2000 epochs we tested the 8 optimizers using default hyperparameters. From this run, we can see most optimisers fall off at certain points of the epoch at around 200-500 epochs. The 4 remaining are SGD, Adadelta, adagrad and Ftrl.



Figure(12). optimizer comparison 2000 epochs, 1 hidden layer, 32 neurons, 1441 parameters

We need to determine the best optimizer - the one which can reduce the val loss to the minimum. Further testing will be conducted with activation function elu, 2 hidden layers at 32 and 16 nodes, input layer at 64 nodes, total of 3329 parameters, 300 epochs. elu loss function is minimised at 251 epochs, so 300 is sufficient. But to our surprise, it is not - there are continual signs of improvement with multiple val_loss minimums at close to 300 epochs.

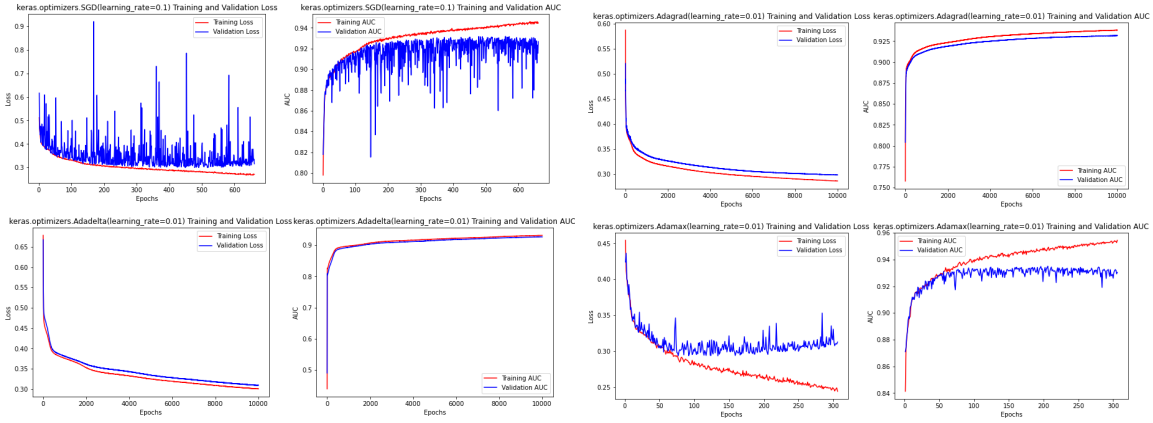
```
['keras.optimizers.Adam()', 275, 0.2729845345020294, 0.9438936710357666, 0.29020050168037415, 0.9347242712974548]
['keras.optimizers.SGD()', 289, 0.3424530029296875, 0.9115883708000183, 0.34851184487342834, 0.909023106098175]
['keras.optimizers.Adadelta()', 299, 0.46493035554885864, 0.8318040370941162, 0.4861174523830414, 0.8115432262420654]
['keras.optimizers.Adagrad()', 299, 0.41619932651519775, 0.8715313076972961, 0.42967647314071655, 0.8606337308883667]
['keras.optimizers.Adamax()', 299, 0.3017013669013977, 0.9312753677368164, 0.3104044198989868, 0.9261410236358643]
['keras.optimizers.Ftrl()', 287, 0.6484513878822327, 0.5, 0.6484495401382446, 0.5]
['keras.optimizers.Ftrl()', 289, 0.6484513878822327, 0.5, 0.6484495401382446, 0.5]
['keras.optimizers.Nadam()', 295, 0.2695693075656891, 0.945112943649292, 0.29146087169647217, 0.9342374205589294]
['keras.optimizers.RMSprop()', 279, 0.28180378675460815, 0.9401530623435974, 0.2951964735984802, 0.9339743256568909]
```

Figure(13). optimizer comparison 300 epochs, 2 hidden layer, 32/16 neurons, 3329 parameters, chosen based on least validation loss, left to right - activation function, epochs, loss, AUC, validation loss, validation AUC

```
['keras.optimizers.Adam()', 252, 0.27401888370513916, 0.9431732296943665, 0.2889091372489929, 0.9357287883758545]
['keras.optimizers.SGD()', 498, 0.33022576570510864, 0.9168603420257568, 0.34072843194007874, 0.9106914401054382]
['keras.optimizers.Adadelta()', 499, 0.4579402804374695, 0.8370460271835327, 0.4788220524787903, 0.8176506161689758]
['keras.optimizers.Adagrad()', 499, 0.41154783964157104, 0.874293327331543, 0.4237872362136841, 0.8647060394287109]
['keras.optimizers.Adamax()', 495, 0.2913331687450409, 0.9363157749176025, 0.30388039350509644, 0.9286714196205139]
['keras.optimizers.Ftrl()', 269, 0.6484511494636536, 0.5, 0.6484493017196655, 0.5]
['keras.optimizers.Ftrl()', 328, 0.6484509706497192, 0.5, 0.6484493017196655, 0.5]
['keras.optimizers.Nadam()', 287, 0.27079641819000244, 0.9440401196479797, 0.2938912808895111, 0.9338741898536682]
['keras.optimizers.RMSprop()', 343, 0.27781951427459717, 0.9413385391235352, 0.2986997365951538, 0.932842493057251]
```

Figure(14). optimizer comparison 500 epochs, 2 hidden layer, 32/16 neurons, 3329 parameters, chosen based on least validation loss, left to right - activation function, epochs, loss, AUC, validation loss, validation AUC

Applying the early stopping and at the same time increasing the learning rate by 10x to speed make the model faster (lesser epochs), we can see the SGD and Adamax hit early stopping at an early stage. Looking at the graphs, it does show very high volatility for validation loss when the model is running.

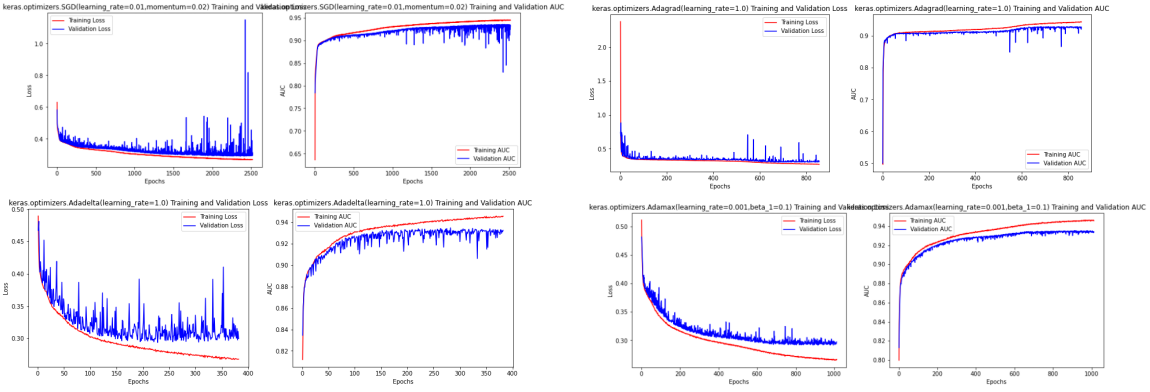



```
['keras.optimizers.SGD(learning_rate=0.1)', 448, 0.28574708104133606, 0.9385145902633667, 0.2990104854106903, 0.9306257367134094]
['keras.optimizers.Adadelta(learning_rate=0.01)', 9992, 0.3008441925048828, 0.931368350982666, 0.3090488910675049, 0.9266956448554993]
['keras.optimizers.Adagrad(learning_rate=0.01)', 9998, 0.2864679992198944, 0.9380747079849243, 0.29842278361320496, 0.9316802620887756]
['keras.optimizers.Adamax(learning_rate=0.01)', 133, 0.2732990086078644, 0.9430289268493652, 0.29344069957733154, 0.9338343739509583]
```

Figure(15). optimizer comparison 10000 epochs, 2 hidden layer, 32/16 neurons, 3329 parameters, early stopping (patience=100), chosen based on least validation loss, left to right - activation function, epochs, loss, AUC, validation loss, validation AUC

Currently, none of the 4 optimizers outperform the ADAM optimizer of validation AUC 0.9357. However, let's give this a last try before we focus on fine tuning the ADAM. For SGD, momentum helps to push it past the local minimum.

For Adadelta and Adagrad, based on the white paper, the learning rate starts at 1.0. For Adamax, reducing the decay rate helps to reduce the volatility. For SGD, we added momentum but it pushed up the volatility at the end. What turned out well was that we no longer need 10,000 epoch, all except Adagrad showed improvement and the results of Adamax is at 0.935 which is close to Adam of 0.9357.




```
['keras.optimizers.SGD(learning_rate=0.01,momentum=0.02)', 2362, 0.27167749404907227, 0.9435696601867676, 0.29094311594963074, 0.9343922734260559]
['keras.optimizers.Adadelta(learning_rate=1.0)', 227, 0.2805197834968567, 0.9398901462554932, 0.2938108742237091, 0.9337601661682129]
['keras.optimizers.Adagrad(learning_rate=1.0)', 793, 0.27656564116477966, 0.9425830245018005, 0.3014266788959503, 0.929570198059082]
['keras.optimizers.Adamax(learning_rate=0.001,beta_1=0.1)', 914, 0.26837992668151855, 0.9451494216918945, 0.291659951210022, 0.9350026249885559]
```

Figure(16). optimizer comparison 10000 epochs, 2 hidden layer, 32/16 neurons, 3329 parameters, early stopping (patience=100), chosen based on least validation loss, left to right - activation function, epochs, loss, AUC, validation loss, validation AUC

What we discovered during our testing is Adamax performs better than Adam. The difference between Adam and the Adamax optimizer, which is essentially a generalization of the L2 norm into the L-infinity norm. And as we continue to test, we can conclude that starting weights play a big part. This is because the validation AUC fluctuates around 0.934-0.936 but because the initialisation weights are not always the same, there is a small difference.

3.4.1 Other testing - Decay vs Mini batch

Decay is a big part of Adamax, we wanted to look at how decay fair against mini-batch. In figure X below, we see that beta=0.9 delivers the best results, while adjusting the beta2 values from the default of 0.99 creates some volatility and poorer validation AUC. Setting decay to 0, increase batch size on the other hand, does have the same effect as increasing decay. But also notice that the number of epochs needed to reach the optimal loss function also increases. Neither batches from Adamax nor SGD produce better results than decay.

```
['keras.optimizers.Adamax(learning_rate=0.001,beta_1=0.1)', 1014, 0.2679750621318817, 0.9456824660301208, 0.2928064167499542,
0.9352034330368042]
['keras.optimizers.Adamax(learning_rate=0.001,beta_1=0.9)', 1078, 0.26331827044487, 0.94691401720047, 0.2865518629550934, 0.9
376118779182434]
['keras.optimizers.Adamax(learning_rate=0.001,beta_1=0.1,beta_2=0.1)', 159, 0.334629088640213, 0.9164049029350281, 0.32647144
79446411, 0.9212926030158997]
['keras.optimizers.Adamax(learning_rate=0.001,beta_1=0.1,beta_2=0.5)', 257, 0.3522186279296875, 0.9096965193748474, 0.3244532
9427719116, 0.9195135831832886]
```

Figure(17). beta1 and beta2 comparison 10000 epochs, 2 hidden layer, 32/16 neurons, 3329 parameters, early stopping (patience=100), chosen based on least validation loss, left to right - activation function, epochs, loss, AUC, validation loss, validation AUC

```
['batchsize=16', 732, 0.27128106355667114, 0.9452048540115356, 0.29580220580101013, 0.9336374402046204]
['batchsize=32', 639, 0.27809444069862366, 0.9414600729942322, 0.2924884259700775, 0.9336254000663757]
['batchsize=64', 1234, 0.2709418535232544, 0.9442221522331238, 0.2915864884853363, 0.9351465106010437]
['batchsize=128', 1579, 0.273025780916214, 0.9433059096336365, 0.29257458448410034, 0.934185802936554]
['batchsize=256', 2272, 0.27646711468696594, 0.9415880441665649, 0.2919830083847046, 0.9342402219772339]
```

Figure(18). batch size comparison 10000 epochs, 2 hidden layer, 32/16 neurons, 3329 parameters, early stopping (patience=100), chosen based on least validation loss, left to right - activation function, epochs, loss, AUC, validation loss, validation AUC, Adamax(learning_rate=0.001,beta_1=0.0)

```
['batchsize=16', 1211, 0.27808433771133423, 0.9416564702987671, 0.2954791784286499, 0.9319961071014404]
['batchsize=32', 1283, 0.2917487919330597, 0.9359638094902039, 0.3064590096473694, 0.9272301197052002]
['batchsize=64', 2166, 0.2916964292526245, 0.9355582594871521, 0.3077380061149597, 0.9272100329399109]
['batchsize=128', 4001, 0.29169246554374695, 0.9359790682792664, 0.30819469690322876, 0.9268787503242493]
['batchsize=256', 7040, 0.29487717151641846, 0.9345530867576599, 0.3092406988143921, 0.9265961050987244]
```

Figure(19). batch size comparison 10000 epochs, 2 hidden layer, 32/16 neurons, 3329 parameters, early stopping (patience=100), chosen based on least validation loss, left to right - activation function, epochs, loss, AUC, validation loss, validation AUC, SGD(default)

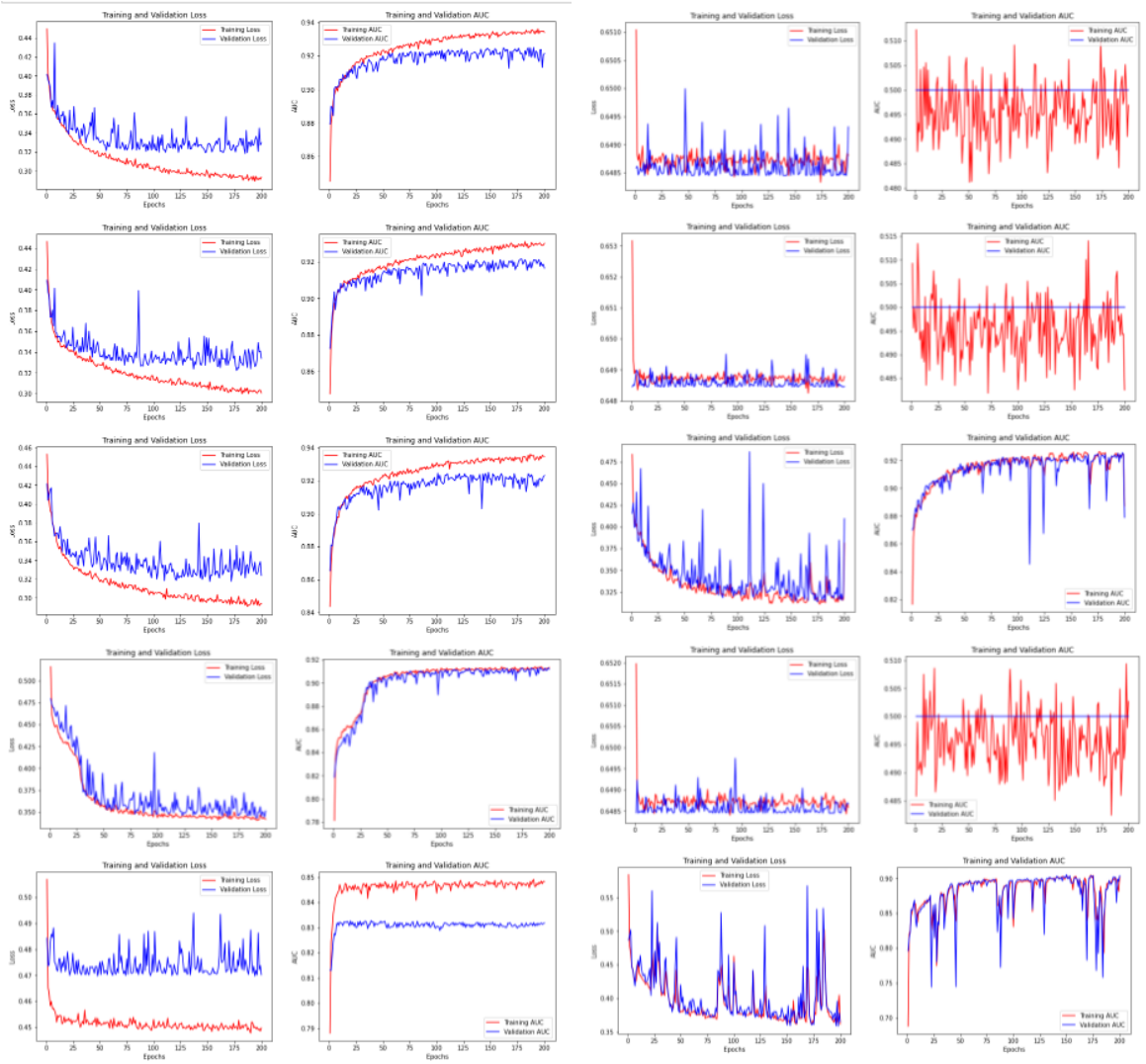
3.4.2 Learning rate

Accidental discovery. While testing layers, keeping neurons at 32 per layer, Activation function Relu, Optimizer Adam (learning rate = 0.01), Hidden layers in ascending order of 1 to 10. ADAM's default learning rate is 0.001, we accidentally set it to 0.01.

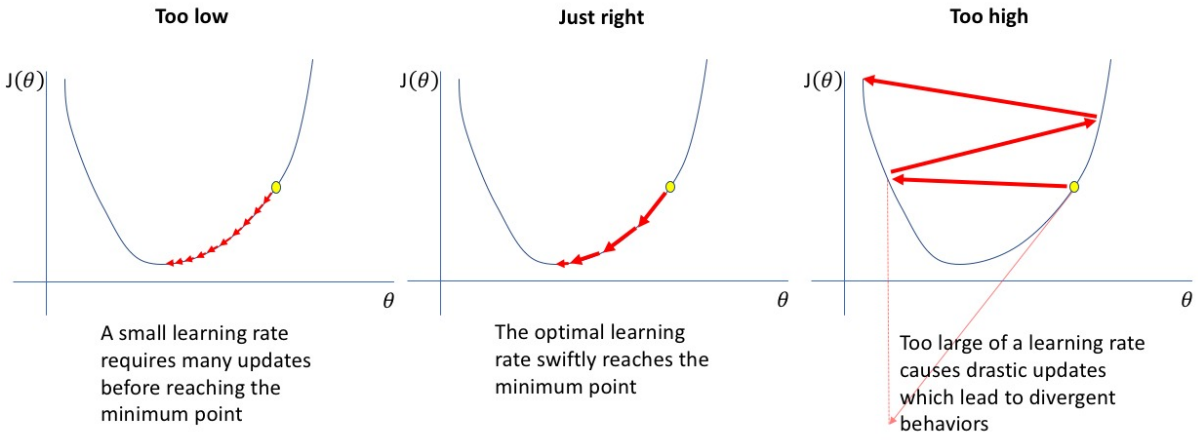
Over here we discovered 2 things:

1. Contrary to the understanding that more layers/parameters will cause overfitting, at the 10 layer, we do not have overfitting. One possible explanation is the additional layers allow the model to have enough variability to break down the model allowing it to follow the high learning rate to fit into the model.

2. Results at 6,7,9 hidden layers show that the AUC fluctuates at around 0.5 which means there is no result. This is accompanied by a very high loss function of more than 0.6. This occurrence is likely due to the random starting weights set but due to the high learning rate, it starts bouncing to and fro the minimal loss (global or local) but is unable to reach the minimum.

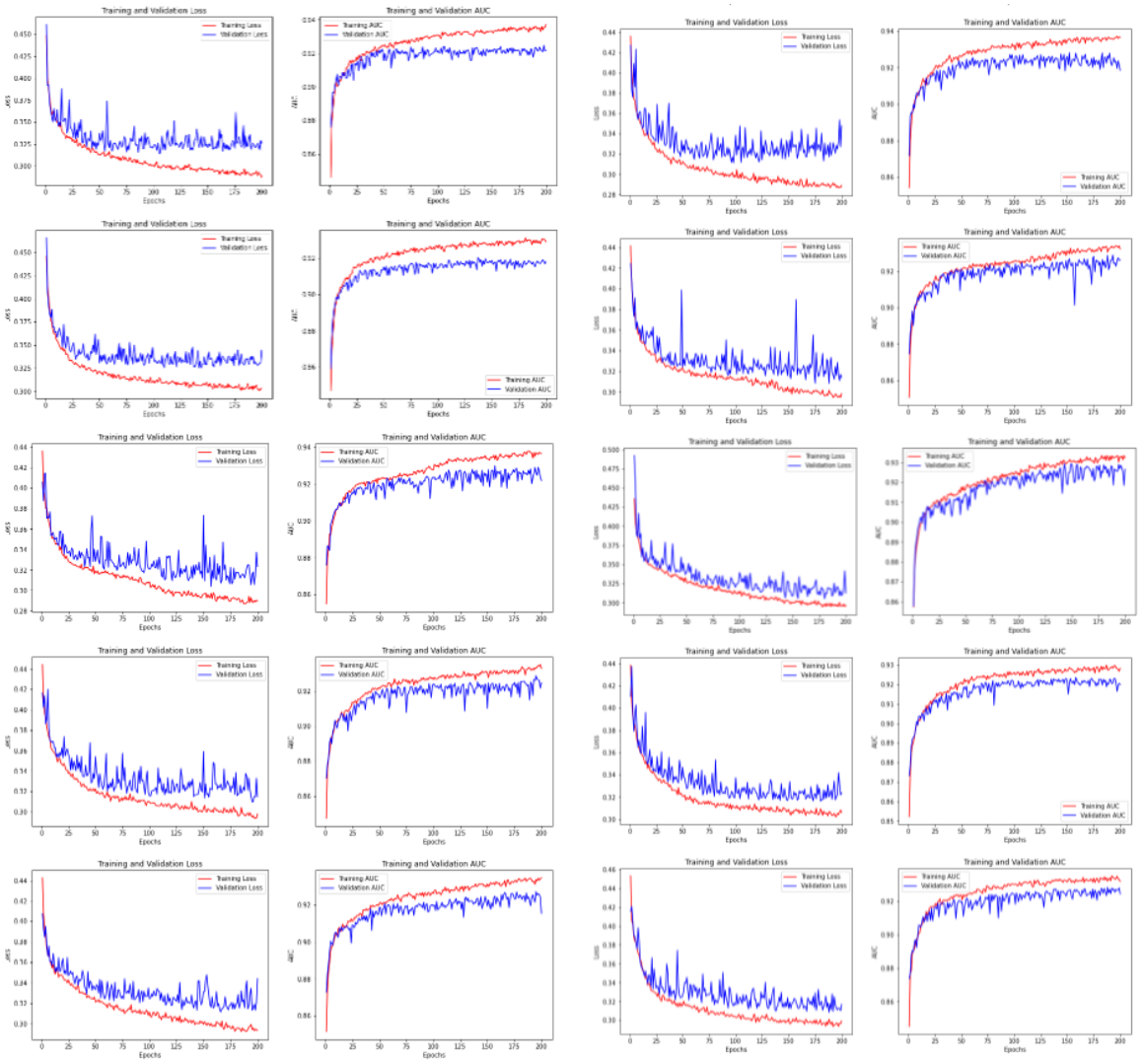


Figure(20). Neurons at 32 per layer, Activation function Relu, Optimizer Adam (learning rate = 0.01), Hidden layers in ascending order of 1 to 10



Figure(21). Too high learning rate explanation

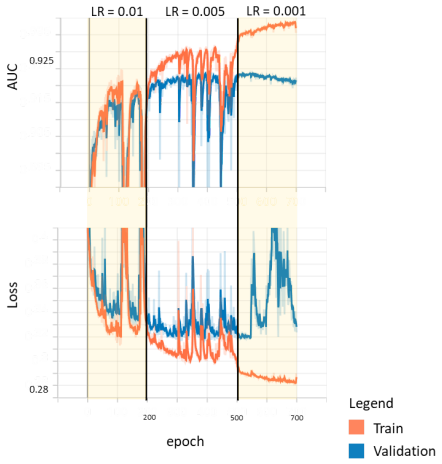
Also accidentally, testing with the number of neurons with the wrong learning rate, Activation function Relu, Optimizer Adam(learning rate 0.01), 1 hidden layer with multiples of 16 nodes from 1 to 10, we see similar results as that of increasing nodes. However, over here, the results don't go into an unstable array of 0.5 AUC beyond the 5th test. This shows that the learning rate affects the layers but don't affects the nodes.



Figure(22). Activation function Relu, Optimizer Adam(learning rate 0.01), 1 hidden layer with multiples of 16 nodes from 1 to 10

3.4.3 Investigation: manual decay of learning rate for special case of 10-layers-32-nodes-model

We want to further investigate the special case of 10-layers-32-nodes-model where instead of overfitting with a large model, the training curve and the validation curve seems to be very close, indicating possibly underfitting, or (more optimistically) could mean generalization. If it is indeed generalization, a decay of learning rate should help it converge to a generalization minimal. However, disappointingly, as shown in Figure.(23), as the learning rate decays, the model starts to overfit as seen in the model learned with the smaller learning rate at the start. Which indicates it is underfitting with a larger learning rate.



Figure(23). Manual decay of learning rate: 0.01, 0.005, 0.001

3.5 Initialization of weights

As already discussed in section 3.2, a different initialization of weights, a different starting point in the optimization curve, will affect how regularization behaves, and in our case, it could cause regularization to totally fail the model learning.

During testing, initialisation was unstable because of the changing layers and neurons. We tried to save our initial weights whenever we could to facilitate consistency of testing across the factors.

We would also expect initialization of weights to affect the "minimum point" at which the model would find at the end of learning. To experiment on this, we select the 3-32-adam-ReLu, and run the dataset with multiple starting points, initialization of weights. It was found out that there are no significant visual differences between the curves and results ("AUC") on the model. It seems that (from all other experiments we ran), that this dataset is not complicated enough for a significant "generalization" and it also seems to be noisy with maximum achievable AUC at only about 0.93.

4. Choosing a model

Based on the five points of interest (as discussed at the beginning of the document), we want to choose a model which is the best for each of the use-cases. While AUC provides some sort of an idea to the performance of the model, AUC does not have information on the shape of the ROC curve; e.g. the slope of the curve at each different region, which will result in different curves even with the same AUC. To choose a model for each of the use-case we will take a look at the FPR at each of the use-case.

Model\FPR	0.2	0.1	0.05	0.02	0.01
3-16-adam-ReLu	0.8965	0.7211	0.5748	0.3996	0.2906
3-16-adam-softplus	**0.9225**	0.7713	0.5566	0.3909	0.2832
3-32-adam-ReLu	0.9092	0.7515	0.5813	0.4042	**0.3364**
3-32-adam-softplus	0.9160	0.7717	0.5754	0.3973	0.2524
3-64,32,16-adamax-elu	0.9222	**0.7856**	**0.5819**	**0.4158**	0.3146

Table(2). Best TPR for each FPR point highlighted

There seems to be a trend at which models with “softplus” as the activation function provide better performance at higher FPR rates, while “ReLu” provides better performance at lower FPR rates. At FPR=0.1, 3-16-adam-softplus and 3-32-adam-softplus have very similar performance, there for simplicity’s sake, we choose 3-16-adam-softplus for TPR rates = 0.2, 0.1 and 3-32-adam-ReLu for TPR rates = 0.05, 0.02 and 0.01.

5. Evaluations

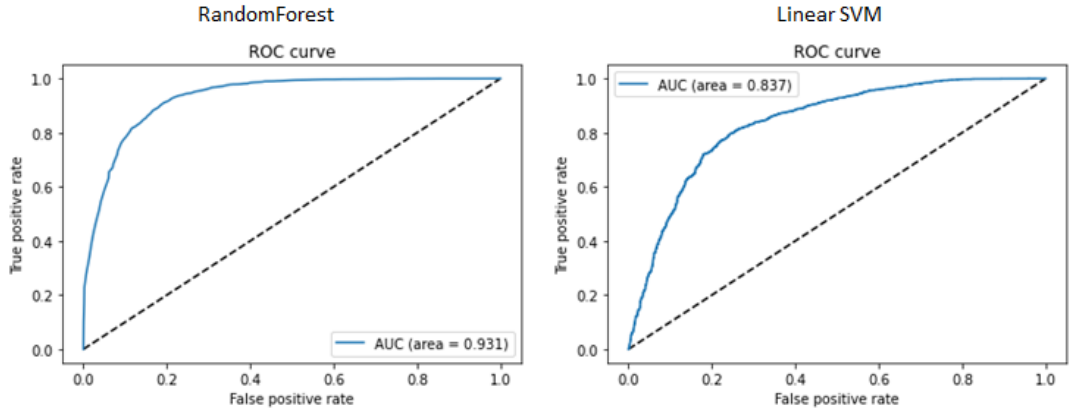
Finally, we run the chosen models on the test set.

Model\FPR	0.2	0.1	0.05	0.02	0.01
3-16-adam-softplus	**0.9166**	**0.7820**	0.5929	0.3801	0.3023
3-32-adam-ReLu	0.9118	0.7619	**0.5939**	0.4204	0.3334
3-64,32,16-adamax-elu	0.9118	0.7613	0.5718	**0.4232**	**0.3473**

Table(3). Results on the test set

6. Comparison between deep learning and classical machine learning algorithms

To make comparisons between DNN and classical machine learning algorithms, we will run a Randomforest ensemble as well as a Linear SVM on the dataset.



Figure(24). ROC of 2 classical ML algorithms

There are 2 main observations to be made. First is that for linear SVM, which assumes that the 2 classes (of the data) can be linearly separated, have an AUC of 0.837 which is similar to the results from when the activation function is changed to linear for a FNN model of 3-layer-32-nodes. This reiterates the points that 1. with a linear activation function, the FNN model is linear and 2. the data cannot be represented by a linear model. Secondly, the RandomForest model performs very similarly to what we have tested with FNN so far.

Model\FPR	0.2	0.1	0.05	0.02	0.01
RandomForest	0.9153	0.7811	0.5871	0.3971	0.3051
LinearSVM	0.7350	0.4904	0.2799	0.1226	0.0620
3-16-adam-softplus	0.9166	0.7820	0.5929	0.3801	0.3023

3-32-adam-ReLu	0.9118	0.7619	0.5939	0.4204	0.3334
3-64,32,16-adamax-elu	0.9118	0.7613	0.5718	0.4232	0.3473

Table(4). TPR of model at different FPR points

7. Conclusion

With advances in computing power, using deep neural networks is faster and easier than ever before, it would seem that deep neural networks might totally make classical machine learning algorithms obsolete. While this is true for problems which involve complex data such as image, language or signal processing, which classical machine learning will not be able to match the performance of deep neural networks, problems such as the one presented in this report can still be learned successfully with a comparable performance. The biggest disadvantage that deep neural networks have, in my opinion, is the lack of interpretation of the learned model. The hidden layers of a deep neural network, is typically labelled as a black box (hence the name hidden layers), which signifies the lack of interpretation of whatever is happening. This interpretation is important especially in manufacturing where traceability and accountability is very important; some things cannot be left to the whims of a black box model. For example, in the recent case of the Boeing 737 MAX crashes which is the cause of a software design (or a lack of), or the earlier case of Samsung Note 7 battery failures which most likely is a cause of QA problems. Without accountability to find the true root cause and the ability to tune manufacturing processes(or models) to the user's exact requirements, we might never see another Boeing plane in the sky or another Samsung phone. However, with advancements in research in interpretation methods (i.e. LRP methods) for deep neural networks and symbolic neural networks, this is set to be changed.

References

[1]https://archive.ics.uci.edu/ml/datasets/magic+gamma+telescope
[2]Nakkiran, P., Kaplun, G., Bansal, Y., Yang, T., Barak, B., & Sutskever, I., Deep Double Descent: Where Bigger Models and More Data Hurt. *ArXiv,abs/1912.02292*, 2020.
[3]Harrison E., Alethea P., Yuri B., Igor B., Vedant M., Grokking: Generalization Beyond Overfitting on Small Algorithmic Datasets. *OpenAI, 1st Mathematical Reasoning in General Artificial Intelligence Workshop, ICLR,* 2021.
[4]Activation Functions — ML Glossary documentation. (2021). Retrieved 19 November 2021, from https://ml-cheatsheet.readthedocs.io/en/latest/activation_functions.html
[5]Understanding AUC - ROC Curve. (2021). Retrieved 19 November 2021, from https://towardsdatascience.com/understanding-auc-roc-curve-68b2303cc9c5
[6]Activation Functions — ML Glossary documentation. (2021). Retrieved 19 November 2021, from https://ml-cheatsheet.readthedocs.io/en/latest/activation_functions.html