Problem 1 : Decision Trees

Dataset

- There are 12 festures total
- Categorial $[8]$: `[team, opp , host, month, day_match , toss, bat_first, format]`
- These take `[19 , 19 , 19, 12 , 3 , 2 , 2 , 2]` distinct values respectively
- Continuous $[4]$: `[year, fow, score, rpo]`
- Out of the above, `[team, opp , host, month, day_match]` need label encoding
- After `OneHOT` encoding there are total 75 Categorial features and 4 continuous features.
- Training data size: 7827
- Validation data size: 870
- Testing data size: 967

Splitting a node (Categorial Feature)

- We want to decide the parameter of split and also the threshold.

$$H(Y) = \sum_{y} -P(y) * log(P(y))$$

$$H(Y|X_j = x_j) = \sum_{y} -P(y|X_j = x_j) * log(P(y|X_j = x_j))$$

$$H(Y|X_j) = \sum_{x_j} P(X_j = x_j) * H(Y|X_j = x_j)$$

$$I(Y|X_j) = H(Y) - H(Y|X_j)$$

- In `Scratch` , while deciding split at a node, `all features` are considered (here the number of features is already too small) and feature corresponding to the minimum $I(Y|X_j)$, or equivalently maximum $H(Y|X_j)$ is used to split. {As $H(Y)$ is same for every X_j}
- For Continuous Features, above is done by splitting across the median

- Problem faced while using Ordinal Endoding is that: In some cases, say a feature X_j has possible values $x_1, x_2 \ldots x_k$ and a possible split by this feature leaves 0 examples with the feature $x_p, 1 \leq p \leq k$, in this case $H(Y|X_j = x_p)$ is considered ∞ and a split is never made using this feature.

- We stop splitting in one of these cases only:

1. Pure leaf is reached. All examples have the same Y label
2. `MAX_DEPTH` is reached
3. There is no valid feature left to split. `Valid` explained in the above paragraph.

Note: (3) was never encountered on the given dataset

- Prediction: On a test example, on reaching the leaf $Y_{predicted}$ is the majority label in X_{train} limited to that leaf only.

(a) Decision Tree: Ordinal Encoding

- Note that here there was no use of Validation set in training as there were no hyper-parameters in this implementation. Both $(X_{validation}, Y_{validation})$ and (X_{test}, Y_{test}) can be treated as test sets.

- Accuracy (%)

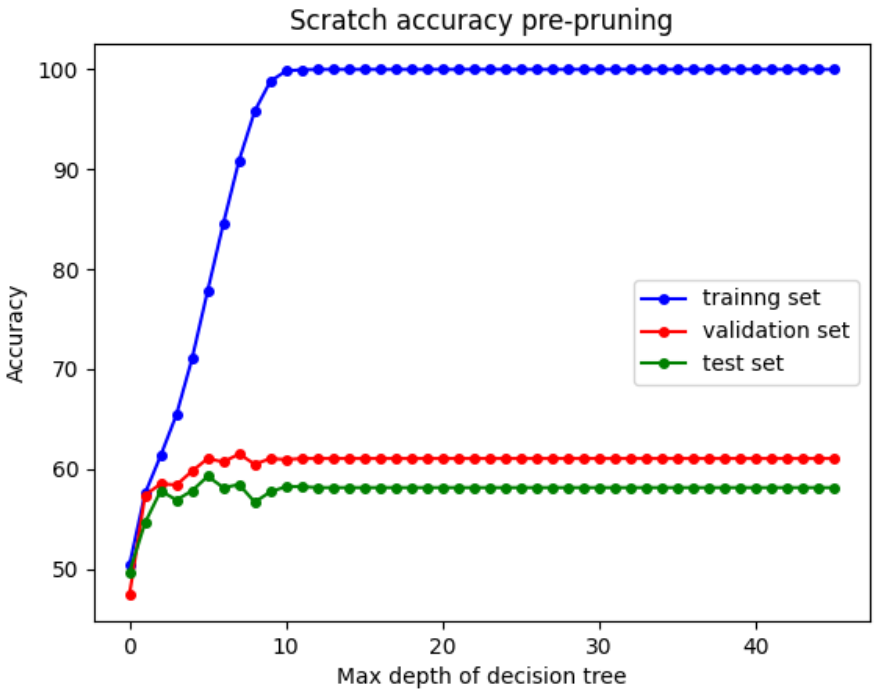
Set	Only win	Only loss
Training	50.339	49.661
Validation	47.356	52.644
Testing	49.638	50.362

- Accuracy by Decision Tree (%)

max_depth	Training Set	Validation Set	Test Set
0	50.339	47.356	49.638
5	77.871	61.034	59.255
10	99.847	60.920	58.221
15	100.000	61.034	58.118
20	100.000	61.034	58.118

max_depth	Training Set	Validation Set	Test Set
25	100.000	61.034	58.118

- In both $(X_{validation}, Y_{validation})$ and (X_{test}, Y_{test}), Decision Tree does $\approx 10\%$ better than `Only win` and `Only loss` prediction.
- Notice that `max_depth = 0` exactly matches with `Only Win` cases. This is because the training set has more `Win` samples than `Loss` samples
- With increasing `max_depth`, accuracy values soon saturates. In Ordinal Endoding we use `k-way splits` and hence the tree had more width and less height, compared to `OneHot` Encoding case.
- Validation set attains maximum accuracy at `max_depth = 5` and later-on it remains constant (ignoring fluctuations) irrespective of `max_depth`
- Given sufficient `max_depth`, the Decision Tree is able to `perfectly` `pack in` the training set. This is `overfitting`, clearly indicated by the much lower Validation Accuracies



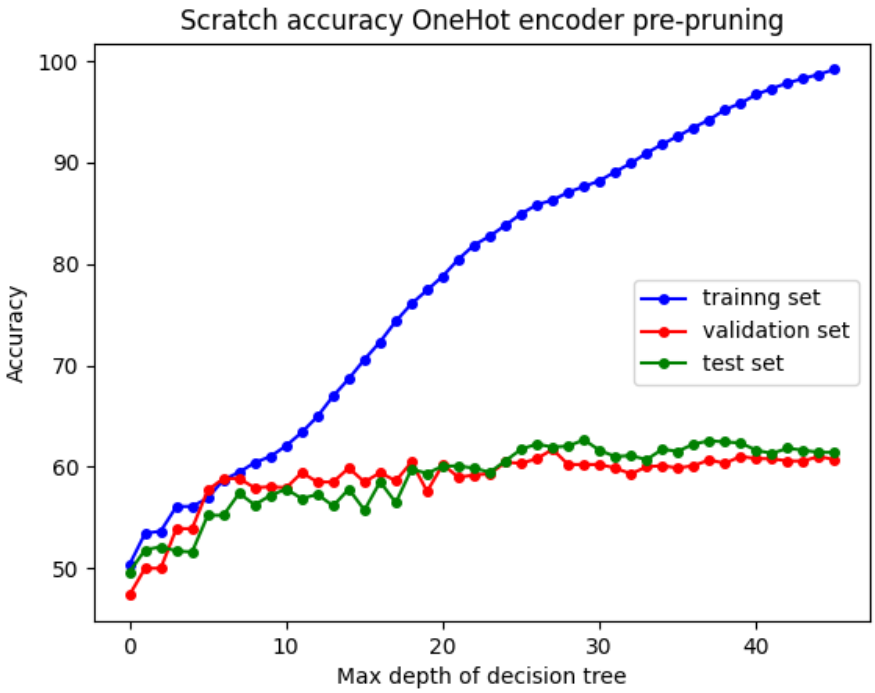
(b) Decision Tree One Hot Encoding

- Note that here there was no use of Validation set in training. Both $(X_{validation}, Y_{validation})$ and (X_{test}, Y_{test}) can be treated as test sets.

- Accuracy (%) : Pre-Pruning

`max_depth`	Training Set	Validation Set	Test Set
0	50.339	47.356	49.638
5	56.906	57.701	55.222
10	62.080	57.931	57.808
15	70.589	58.506	55.739
20	78.804	60.230	60.083
25	84.950	60.345	61.737
35	92.603	59.885	61.531
45	99.182	60.690	61.427

- For the same `max_depth` (before saturation), `Ordinal encoded` tree offers better training accuracy than `OneHot encoded` tree. This is because the former has more width and packs more information about the (X_{train}, Y_{train}) in the same depth.
- Accuracy values are almost the same as `Ordinal encoded` trees.
- Validation set accuracy achieves maximum at `max_depth = 40` approximately which can also be seen in the below graph (ignoring fluctuation)



(c) Decision Tree Post Pruning

- In a single iteration of pruning, all non-leaf node in the Decision Tree are `hypotheticaly`, pruned, i.e. all nodes in the subtree of the chosen node are removed, except the node itself. Accuracy on Validation Set is calculated. The node corresponding to maximum `increase` on validation accuracy is used to actually prune. This is continued till pruning of any node, `strictly decreases` the validation accuracy.
- Implemented a fast algorithm for pruning that took $\mathcal{O}(x)$ time per iteration, where x is the current number of nodes in the tree. This does not need to iterate over the entire validation set in each iteration. Verified it against the naive method for correctness.
- Note: Post-pruning results might vary slightly with each run. At any stage during purning, there can be many `nodes`, at which pruning will offer the exact same maximum improvement in accuracy improvement on (X_{val}, Y_{val}). The `node` to prune at, is `randomly` chosen in such cases. For example, $8, 8, 8, 6, 6, 3, 1$ giving the number of more examples in X_{val} thare are correctly classified by pruning.

`max_depth`	Intial Size	Final Size	Number of Iterations
15	1569	185	61
25	3395	367	190
35	4431	529	292
45	5469	637	320

Training Set Accuracy (%)

`max_depth`	Pre-pruning	Post Pruning
15	70.589	63.448
25	84.950	66.398
35	92.603	68.724
45	99.182	70.167

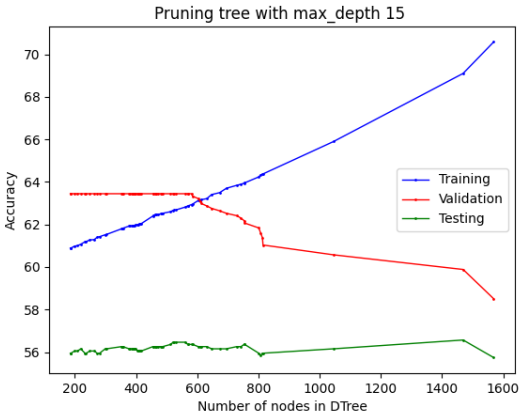
Validation Set Accuracy (%)

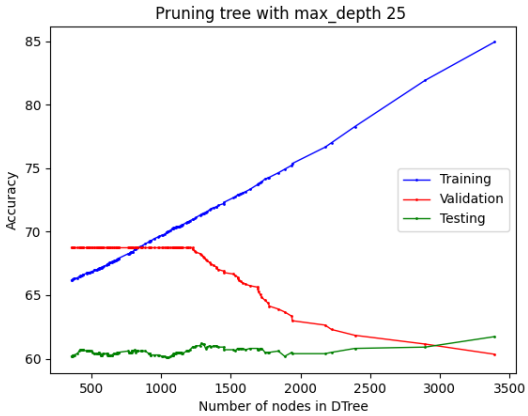
`max_depth`	Pre-pruning	Post Pruning
15	58.506	63.448
25	60.345	68.736
35	59.885	72.529
45	60.690	74.138

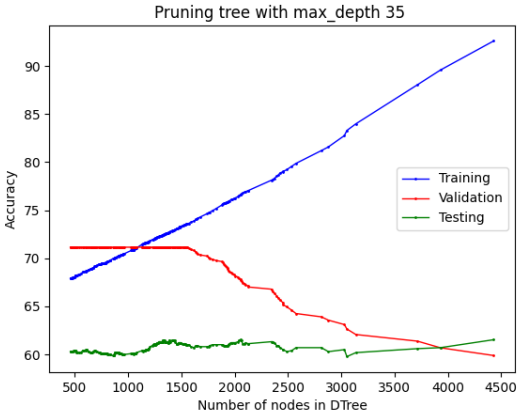
Test Set Accuracy (%)

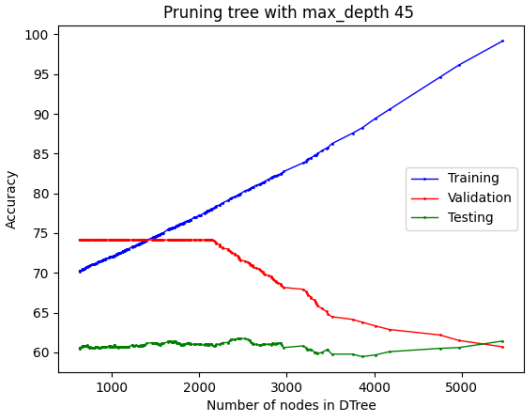
`max_depth`	Pre-pruning	Post Pruning
15	55.739	55.946
25	61.737	59.876
35	61.531	60.290
45	61.427	60.703

- Training set accuracy clearly decreases on further pruning. This is reducing overfitting. Decision Tree, is an unstable learner and without pruning is hightly susecptible to overfitting.
- Validation set accuracy increases with pruning, by design of the algorithm
- Validation set accuracy becomes constant in the later part during pruning, the tree size just keeps reducing. Pruning is continued at this stage to save computation during prediction.
- Test set accuracy, in the last 3 cases infact `decreases` post pruning. This indicated that `excessive` pruning can lead to `loss` of information and increasing `bias`
- In the below graph, It can be seen that Test Set accuracy attains a maximum in between (the values in above table are at the `end of pruning` but we can't use Test set accuracies to influence out training process
- Difference in Test set accuracies post pruning, against `max_depth` is not large in the last 3 rows. Both Post Pruning Test and Validation accuracies incease with `max_depth`





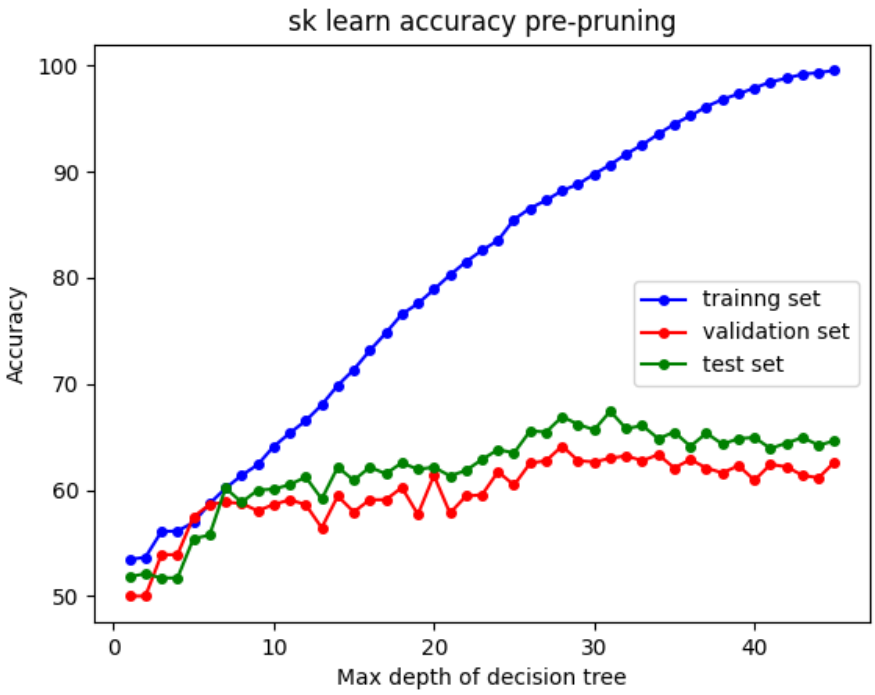




(d) Decision Tree sci-kit learn

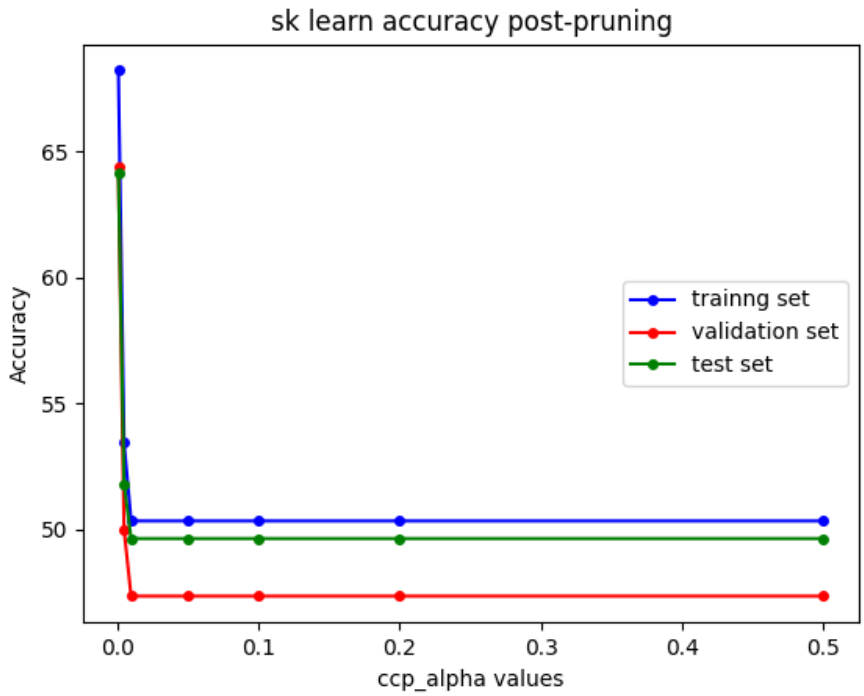
- Note that Validation Set was not used in training, so it can be treated as test set.
- `max_depth` vs Accuracy (%)

`max_depth`	Training Set	Validation Set	Test Set
15	71.343	57.931	60.910
25	85.499	60.460	63.495
35	94.493	62.069	65.460
45	99.540	62.644	64.633



- `ccp_alpha` vs Accuracy (%)

ccp_alpha	Training Set	Validation Set	Test Set
0.001	68.213	64.368	64.116
0.01	50.339	47.356	49.638
0.1	50.339	47.356	49.638
0.2	50.339	47.356	49.638



- The best validation/ test performance is at `max_depth = 35` corresponding to a an accuracy of $\approx 65.460\%$
- The best validation/ test performance is at `ccp_alpha = 0.001` corresponding to a an accuracy of $\approx 64.386\%$
- In `scratch` the best best pre-pruning validation accuracy corresponds to `max_depth = 45` that has a test accuracy of $\approx 61.427\%$. The best post-pruning validation also corresponds to `max_depth = 45` that has a test accuracy of ≈ 60.703 %.
- SK-learn's model performs somewhat better.

(e) Grid Search

- `sk learn`'s `GridSearchCV` by default does `k-fold cross validation` on the input data. Changed this mode to train on (X_{train}, Y_{train}) cross validation using (X_{val}, Y_{val})
- "The out-of-bag (OOB) error is the average error for each calculated using predictions from the trees that do not contain in their respective bootstrap sample. This allows the RandomForestClassifier to be fit and validated whilst being trained." : Source (https://scikit-learn.org)
- Grid Search is exhaustive and tests every possible combination of the hyper-paramters from the input search space
- The search space was

```
PARAM_GRID = {
n_estimators: [50 , 100 , 150 , 200 , 250 , 300 , 350],
max_features: [0.1 , 0.3 , 0.5 , 0.7 , 0.9 ,  1.0],
min_samples_split: [2 , 4 , 6 , 8 , 10]
}
```

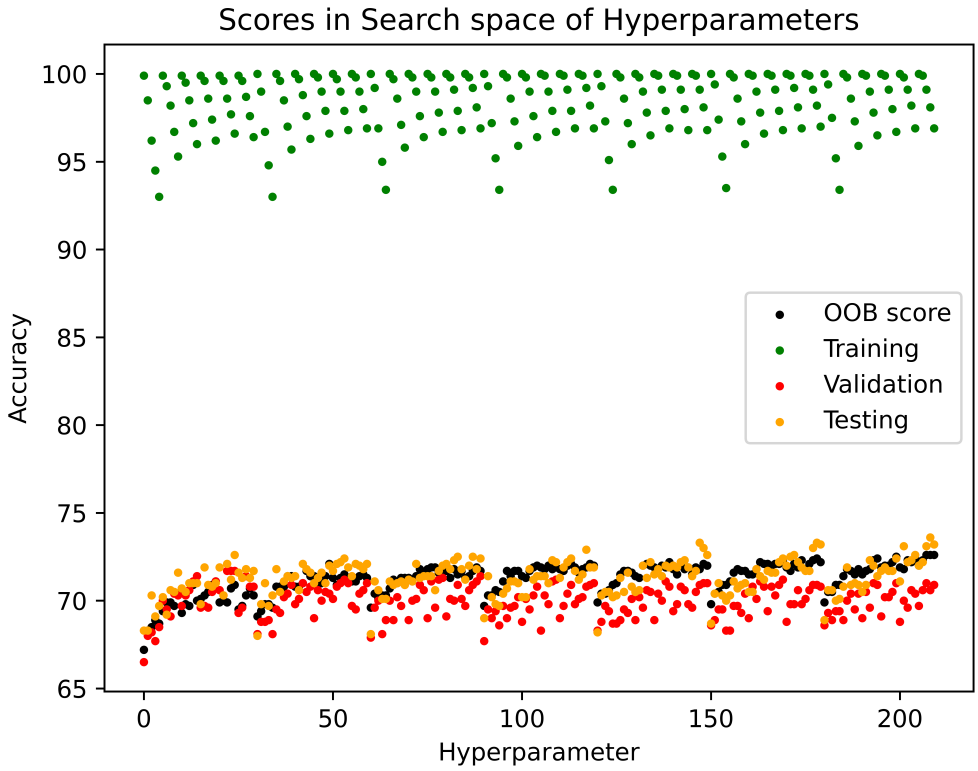
- Final Hyper-parameters obtained:

```
- max_features: 0.7
- min_samples_split = 8
- n_estimators = 100
```

- The exhaustive search takes a lot of time. The above `forests` consist of total $15,400$ different trees to train and botstrapping.
- Best `Out-Of-Bag (OOB)` accuracy = 71.795%

Set	Accuracy (%)
Training	97.675
Validation	97.126
Testing	70.734

- Note that the above results came out different, with slight change in accuracies (1% - 2%) each time the program runs, giving a different set of best hyper-parameters
- These acuracies ($\approx 70\%$ test) are higher than the previous ones ($\approx 65\%$ by sk-learn and $\approx 60\%$ by scratch) , but note that these are `Random Forestes` and get the advantage of `ensembling` which reduces learning `variance` compared to simple decision trees.
- Various Data points in the search space are plotted here


Scores in Search space of Hyperparameters
Accuracy
Hyperparameter
OOB score
Training
Validation
Testing
100
95
90
85
80
75
70
65
0
50
100
150
200

Problem 2 : Neural Networks

Terminology

- Total number of layers: L namely l_1, l_2..... l_L
- l_0 is the input layer. It outputs n^0 paramters which are the input features (including the constant $x_0 = 1$ for the `bias` term)
- $l_1, l_2 . . . . . l_{(L-1)}$ are hidden layers
- l_L is the output layer
- Layer l_i has $n^{(i)}$ perceptrons
- Output of each perceptron in l_i (hence total n_l in number) are fed into each perceptron in l_{i+1}
- $\theta_k^{(l)}$: paramter of k^{th} $(1 \leq k \leq n_l)$ perceptron in the l^{th} $(1 \leq l \leq L)$ layer. It is a **vector** of size n_{l-1}
- $o_k^{(l)}$ is output of k^{th} $(1 \leq k \leq n_l)$ perceptron in the l^{th} $(1 \leq l \leq L)$ layer$(0 \leq o_k^{(l)} \leq 1)$
- $o^{(l)} = (o_1^{(l)}, o_2^{(l)}, ... o_{n^{(l)}}^{(l)})^T$ is a vector of size $n^{(l)}$
- $net_k^{(l)}$ is the net linear combination, inside k^{th} $(1 \leq k \leq n_l)$ perceptron in the l^{th} ($1 \leq l \leq L$) layer given as

$$net_k^{(l)} = \sum_{j=1}^{n^{(l-1)}} o_j^{(l-1)} \theta_{kj}^{(l)} = (\theta_k^{(l)})^T o^{(l-1)}$$

- The perceptron output is given as:

$$o_k^{(l)} = g(net_k^{(l)})$$

Backpropagation: Derivation

- When g is the **sigmoid**, we get

$$\frac{\partial o_k^{(l)}}{\partial net_k^{(l)}} = o_k^{(l)} * (1 - o_k^{(l)})$$

- Also we define

$$\frac{\partial J}{\partial net_k^{(l)}} = -\delta_k^{(l)}$$

- Keep in mind that $\delta_k^{(l)}$ is a scalar
- For gradient descent to obtain optimal $\theta_k^{(l)}$'s we'll need $\nabla_{\theta_k^{(l)}} J$
- Using chain rule we get

$$\nabla_{\theta_k^{(l)}} J = \nabla_{net_k^{(l)}} J * \nabla_{\theta_k^{(l)}} net_k^{(l)}$$

- which gives us

$$\nabla_{\theta_k^{(l)}} J = -\delta_k^{(l)} * \nabla_{\theta_k^{(l)}} net_k^{(l)}$$

- Noticing that

$$\nabla_{\theta_k^{(l)}} net_k^{(l)} = o^{(l-1)}$$

- We finally get

$$\boxed{\nabla_{\theta_k^{(l)}} J = -\delta_k^{(l)} * o^{(l-1)}}$$

- Now to obtain $\delta_k^{(l)}$

$$\nabla_{net_k^{(l)}} J = \nabla_{o_k^{(l)}} J \frac{\partial o_k^{(l)}}{\partial net_k^{(l)}}$$

- Taking g as the sigmoid, we get

$$-\delta_k^{(l)} = \nabla_{net_k^{(l)}} J = \nabla_{o_k^{(l)}} J * o_k^{(l)} * (1 - o_k^{(l)})$$

- $o_k^{(l)}$ affects J through $net_j^{(l+1)}$ for $j = 1, 2 \ldots n^{(l+1)}$ which gives us

$$\nabla_{o_k^{(l)}} J = \sum_{j=1}^{n^{(l+1)}} \nabla_{net_j^{(l+1)}} J * \frac{\partial net_j^{(l+1)}}{\partial o_k^{(l)}}$$

- simplifying to

$$\nabla_{o_k^{(l)}} J = \sum_{j=1}^{n^{(l+1)}} -\delta_j^{(l+1)} * \theta_{jk}^{(l+1)}$$

- finally giving us the iteratve rule for Back Propagation:

$$\boxed{\delta_k^{(l)} = [\sum_{j=1}^{n^{(l+1)}} \delta_j^{(l+1)} * \theta_{jk}^{(l+1)}] * o_k^{(l)} * (1 - o_k^{(l)})}$$

- and

$$\boxed{\nabla_{\theta_k^{(l)}} J = -\delta_k^{(l)} * o^{(l-1)}}$$

- $o_k^{(l)}$ values are first obtained during forward propagation, then $\delta_k^{(l)}$ values during backward propagation
- Base case: $o_k^{(0)}$ are the values of input features
- $\delta_k^{(l)}$ values are first obtained from the last layer during back propagation as shown below.
- Note that in our implemetation, in the last layer, $net_k^{(L)}$ is not passed into the **sigmoid**, rather, $o_k^{(L)}$ is generated as:

$$o_k^{(L)} = \frac{e^{net_k^{(L)}}}{\sum_j e^{net_j^{(L)}}}$$

- When solved, it gives:

$$\delta_k^{(L)} =$$

$$1 - o_k^L \;\; if \;\; k = \bar{k}$$

$$-o_k^L \;\; otherwise$$

- Here we derived for just a single training example with true label $\bar{k}$. For a batch of greater size, these results are to be averaged over each example in the (mini) batch

Data:

- Train data class frequencies (total 10000):

$$1971, 1978, 1952, 2008, 2091$$

- Test (total 1000)

$$229, 198, 199, 187, 187$$

- Both the datas are `balanced`. Each class has an almost equal population

(a) Stochastic Gradient Descent: Hidden Layer Architecture {100 , 50}

- Used constant learning rate. Note that n below includes the bias term $x_0 = 1.0$.
- H denotes hidden layer architecture

$$M = 32$$
$$\eta = 0.01$$
$$n = 1025$$
$$H = 100, 50$$
$$K_{prev} = 400$$

- Each of $\theta_{kj}^{(l)}$ is initialised from Random Normal Distribution with mean 0 and vairance σ^2 that was adjusted for convergence. $\sigma^2 \approx 0.05$ worked for all.
- We move over (X_{train}, Y_{train}) in a robin fashion, in batch sizes of M. One complete traversal over the data is considered an `epoch`.
- J_{mini} is Loss funciton is the `averaged` over all examples in the mini batch.
- Error of a mini match J_{mini}, averaged over 1 `epoch` is called J_{avg}
- Experimeted with value of gradient (max norm) after iteration but even after the cost J_{avg} falls subtantially, the max norm of gradient is no-where close to say, 10^{-4}. The J_{avg} cost, though does fall in this time.

Convergence Criteria

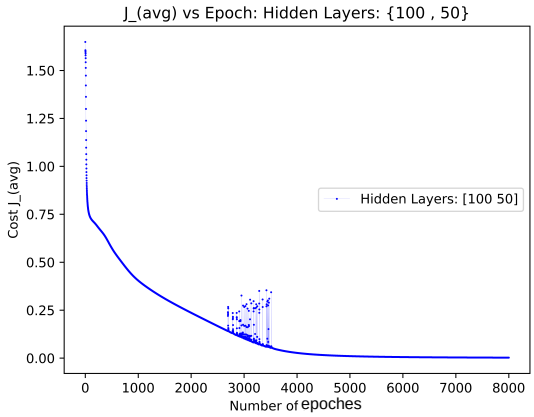
- It was found that J_{avg} keeps on decreasing even after 8000 epoches. Setting a criteria like difference of J_{avg} in 2 successive epoches be less than 10^{-4} for convergence `gives too early` stopping. In this example, the final `Loss` after 8000 epoches was

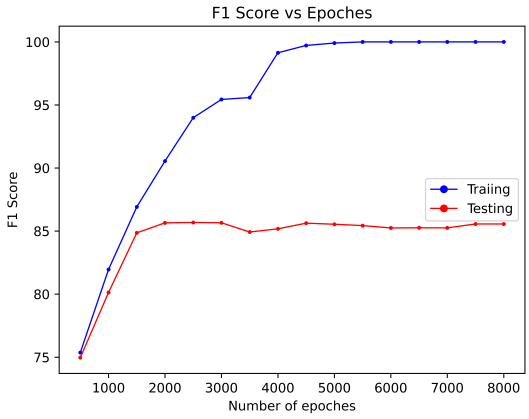
  $$J_{avg} = 0.002427077703840211$$

  it's difference with the previous epoch was

  $$J_{avg}^{7999} - J_{avg}^{8000} = 7.4 * 10^{-7}$$

- The amount of time taken though for this is infeasible to be run on ever other architecture below. In the following graph we can see the convergence of J_{avg} and saturation of training accuracy to 100%



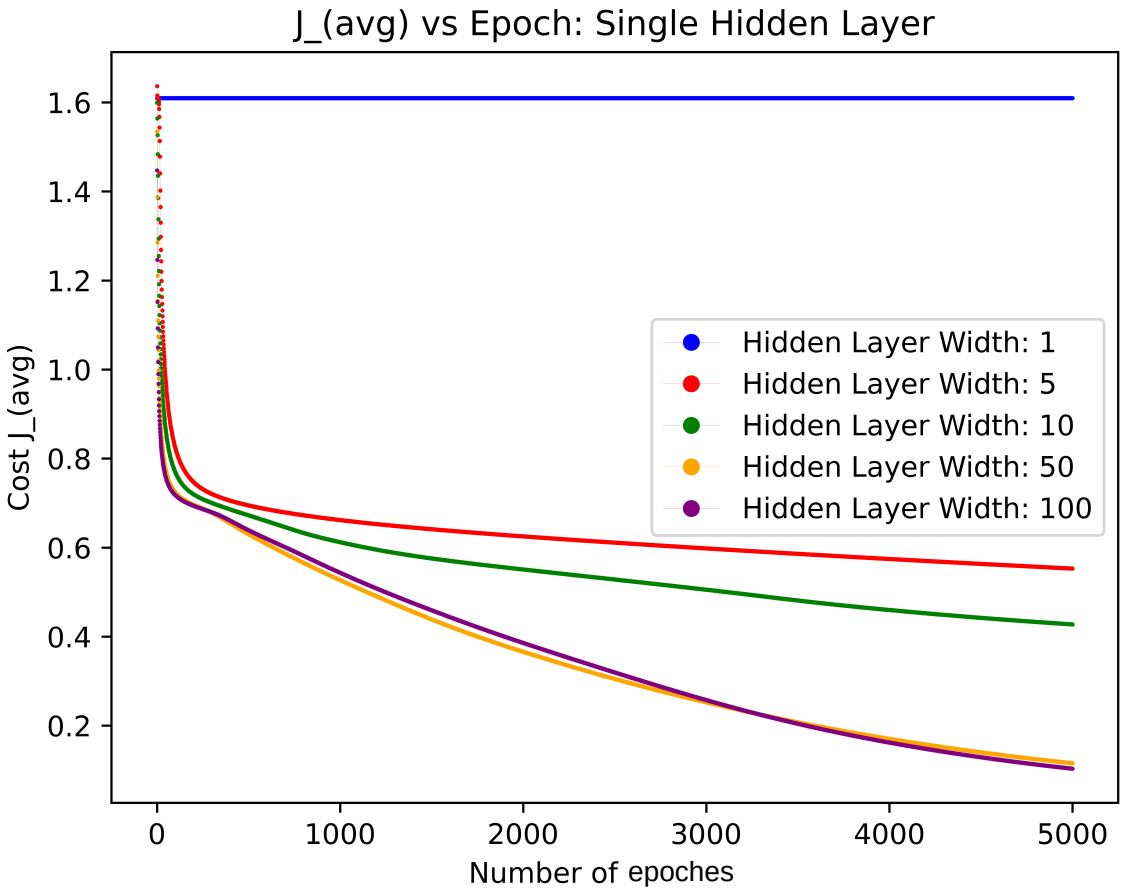


- F1 Score vs Number of Epoches (%)

Epoches	Training	Testing
500	75.367	74.967
1000	81.956	80.129
1500	86.914	84.859
2000	90.552	85.654
2500	93.989	85.681
3000	95.437	85.659
3500	95.587	84.926
4000	99.139	85.178
4500	99.720	85.628
5000	99.910	85.542
5500	100.000	85.436
6000	100.000	85.248
6500	100.000	85.260
7000	100.000	85.254
7500	100.000	85.562
8000	100.000	85.563

- As clearly indicated by the `Loss` value shoun above, this achieves 100% training accuracy.

(b) Varying Layer Width on Single Hidden Layer Architecture



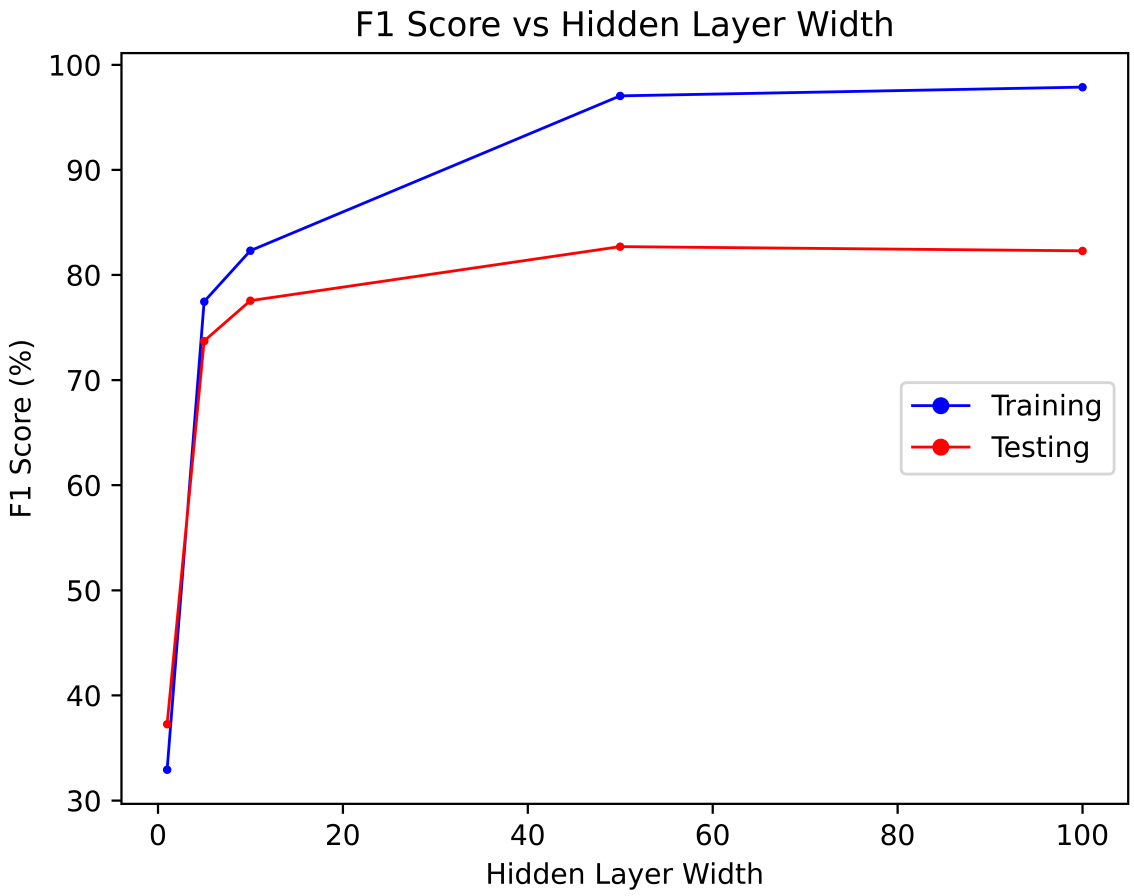
- Using `sk learn :: classification_report()`, `weighted` average of `Precision`, `Recall` and `F1_socre` are reported, but they aren't far from `unweighted` values as the count of each class is almost the same i.e. both `train` and `test` data are `balanced`.
- After training for 5000 epoches, we get:
- Training Data Scores (%):

Width	Precision	Recall	F1 Score
1	100.000	19.710	32.930
5	78.531	76.890	77.459

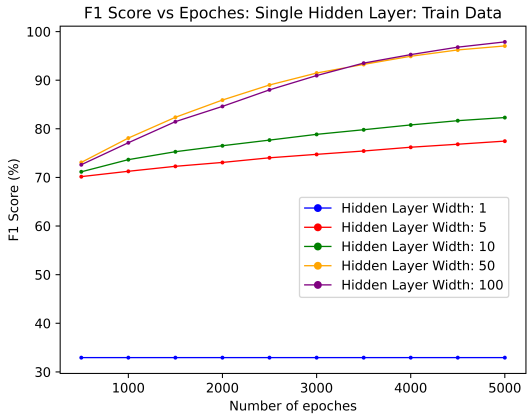
Width	Precision	Recall	F1 Score
10	83.148	82.000	82.306
50	97.064	97.040	97.035
100	97.878	97.860	97.866

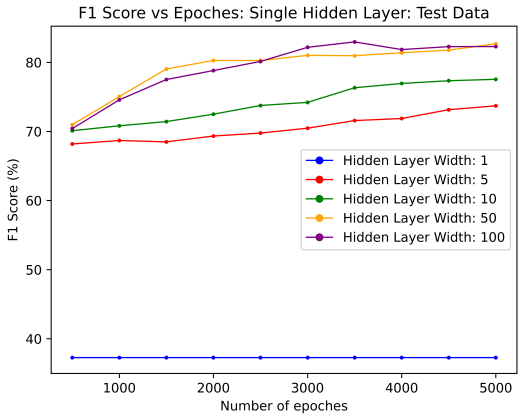
- Testing Data Scores (%):

Width	Precision	Recall	F1 Score
1	100.000	22.900	37.266
5	74.694	73.200	73.713
10	78.272	77.300	77.551
50	83.175	82.800	82.691
100	82.624	82.200	82.292



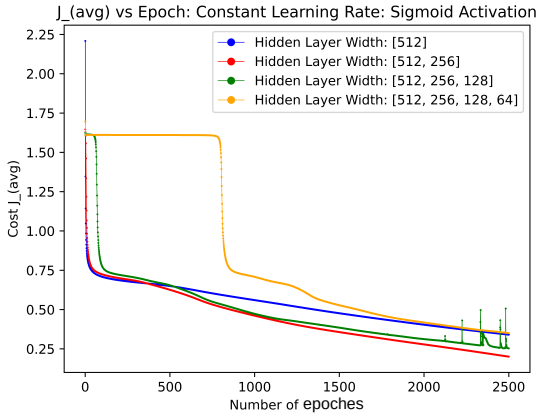
- The `F1 Score` for 1 should be ignored. This classifies all the samples into the same class. This gives `zero division error` for calculation of `F1 Score`
- As expected, wider layer is able to better `absorb` the information in (X_{train}, Y_{train}) and gives better Scores
- Stopping Criteria: As explained in (a), we need to limit the number of epoches. I tried monitoring the Training and Test Set F1 Score after each 1000 epoches. Here are the results.

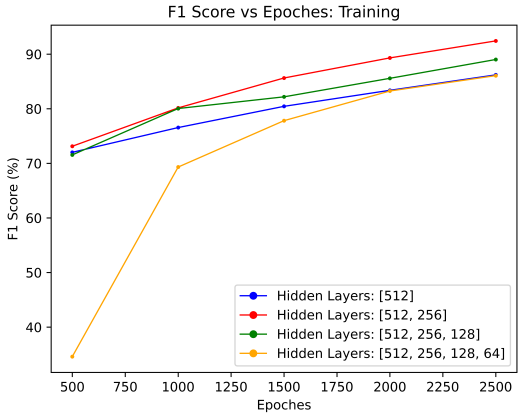




- While the Training Set Scores still seem to increase, the Test Set scores (which I intend to use as a Validation set here) look like getting saturated. This can be used a stopping criteria. A more clear version, can be seen in (a) "F1 Score vs Epoches" there it's much clear that the Test (Validation) set F1 Scores saturate much earlier than Training Set F1 Scores

(c) Varying Network Depth





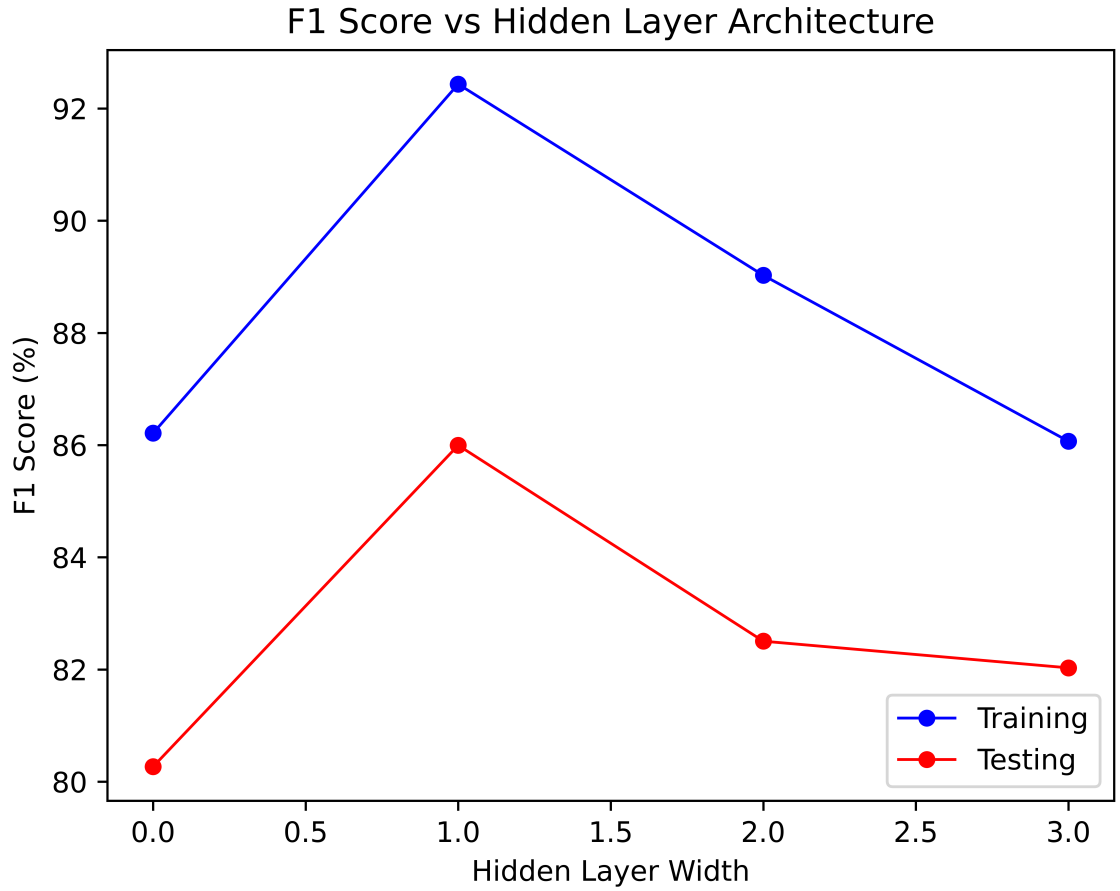
- In $[512, 256, 128, 64]$ there's no change in J_{avg} for atleast the first 500 epoches. This is because it's a deeper network than the rest so the X_{train} takes `longer` (more epoches) to propagate from l_0 to l_L and Y_{train} information takes `longer` to propagate from l_L to l_0. The same pattern was obtained on running it repeatedly.
- Further there's role of the `sigmoid` whoch gets saturated `too soon` on both sides of 0 hence losing information while propagation. This drawback is handles well by ReLU (shown later)
- After training for 2500 epoches, we get:
- Training Data Scores (%)

Layers	Precision	Recall	F1 Score
$[512]$	86.706	86.080	86.213
$[512, 256]$	92.449	92.440	92.434
$[512, 256, 128]$	89.308	88.990	89.026
$[512, 256, 128, 64]$	86.346	85.930	86.069

- Testing Data Scores (%)

Layers	Precision	Recall	F1 Score
$[512]$	80.654	80.200	80.268
$[512, 256]$	86.051	86.000	85.996
$[512, 256, 128]$	82.687	82.500	82.504
$[512, 256, 128, 64]$	82.095	82.000	82.029

F1 Score vs Network Depth

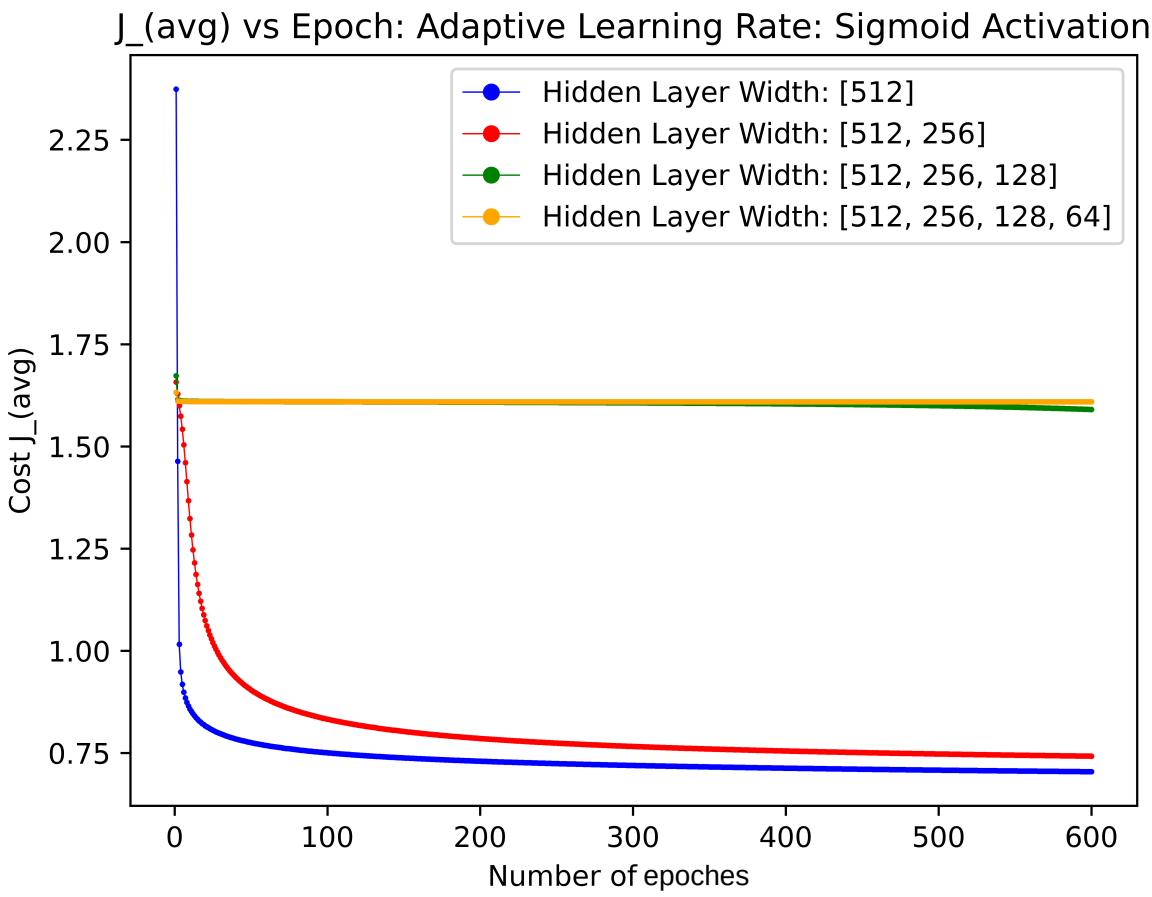


- Having trained all for the same number of epoches, we see $[512, 256]$ offers the best `F1 Score`. $[512]$ loses probably because of the simplicity (bias) while $[512, 256, 128, 64]$ because of the loss of information with depth as explained above.
- Note that given a sufficient number of epoches, all these should be abble to attain much better Training scores (see (a)). But that time was infeasible to train all these networks

(d) Adaptive Learning Rate

$$\eta_e = \frac{\eta_0}{\sqrt{e}}$$

where e is the current **epoch** number



- After training for 500 epoches
- Training Data Scores (%)

Layers	Precision	Recall	F1 Score
$[512]$	70.473	69.080	69.604
$[512, 256]$	67.617	66.460	66.939
$[512, 256, 128]$	100.000	20.910	34.588
$[512, 256, 128, 64]$	100.000	20.910	34.588

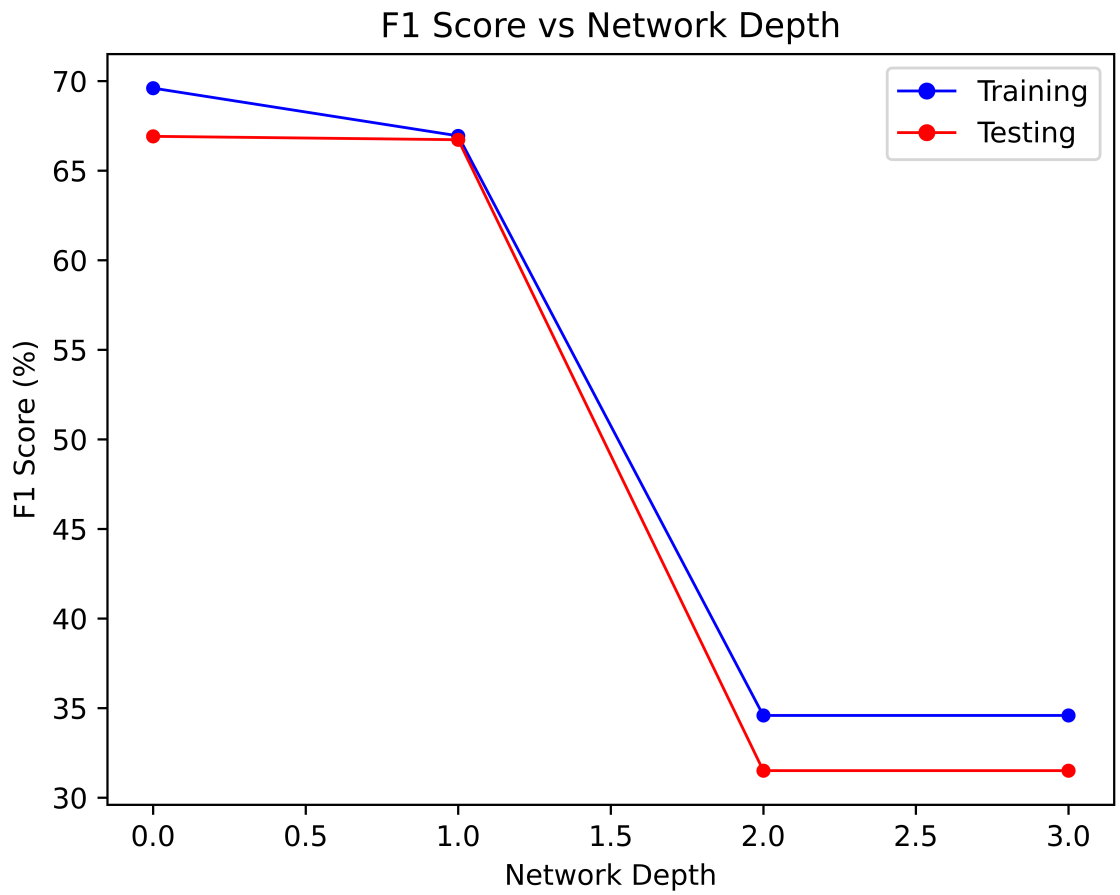
- Testing Data Scores (%)

Layers	Precision	Recall	F1 Score
$[512]$	67.101	66.800	66.913
$[512, 256]$	66.950	66.600	66.724

Layers	Precision	Recall	F1 Score
$[512, 256, 128]$	100.000	18.700	31.508
$[512, 256, 128, 64]$	100.000	18.700	31.508

- F1 Scores for last 2 rows should be ignored. These classfiers predicted all examples into the same class, F1 Score calculation gives `zero division error`

F1 Score vs Network Depth



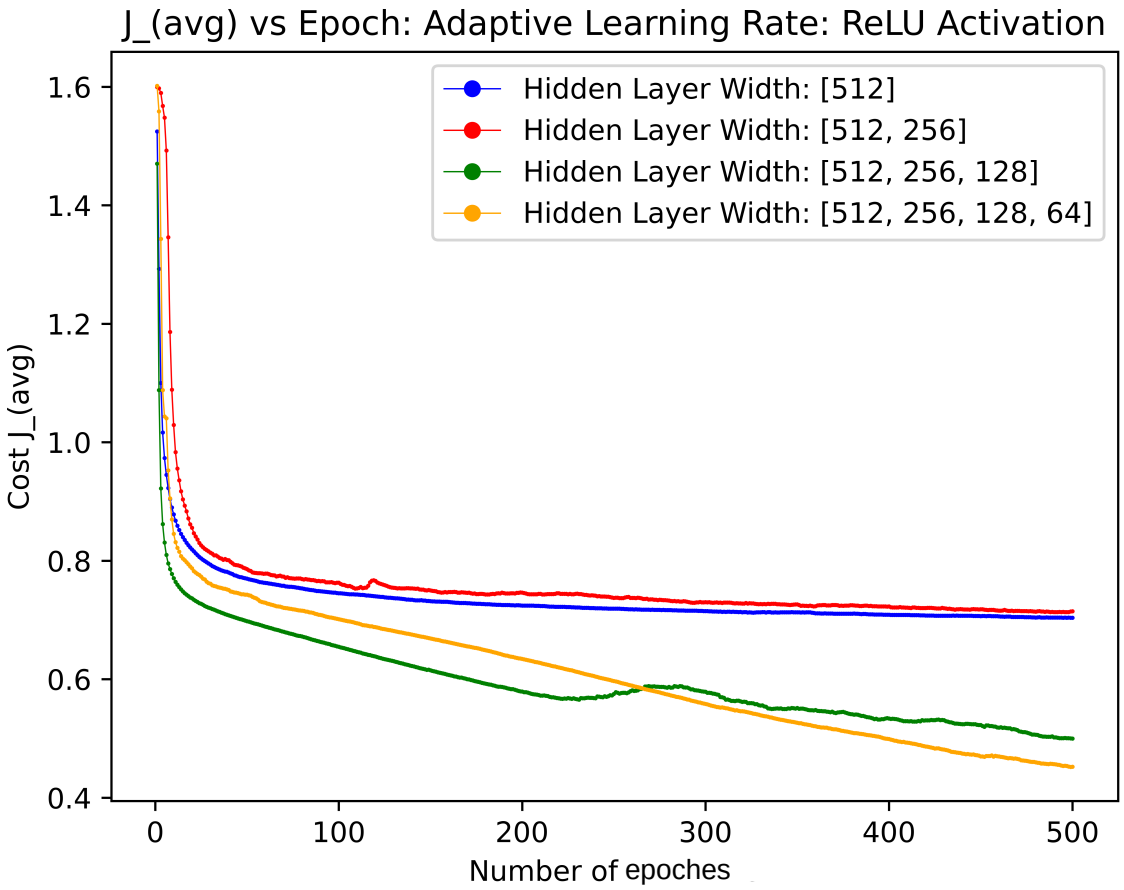
- While it looks like, from the graph of J_{avg} vs Epoch that J_{avg} converges, this is because the `step size` becomes too small beacuse η becomes too small. These values as shown in (e) after 500 iterations are much smaller than in the constant η case.

- A relative comparison of speed of convergence using adapted η is given later in (e)

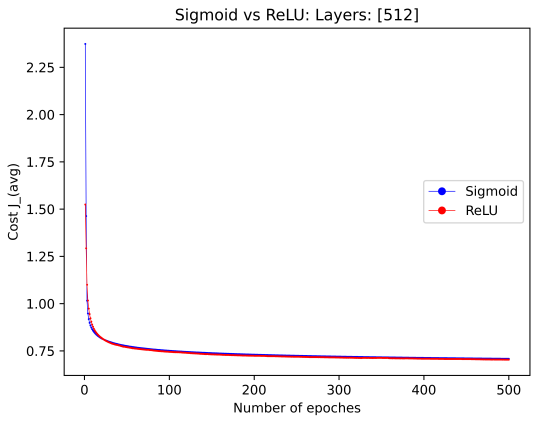
- The problem here is that, the given $\eta = \frac{\eta_0}{\sqrt{e}}$ falls to quickly, definitely not in sync with how J_{avg} changes here. Which is why we see $[512, 256, 128, 64]$ and $[512, 256, 128]$ perform poorly.

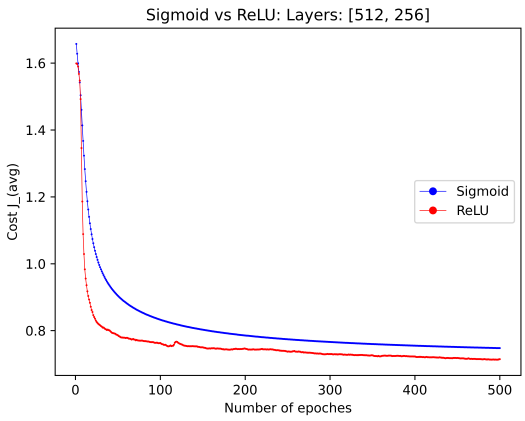
(e) ReLU activation: Scratch

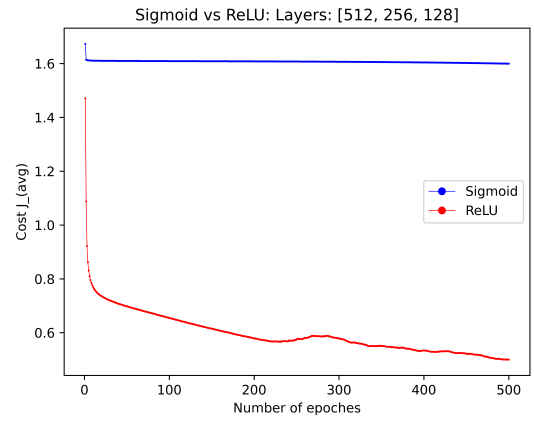
- The gradient for non negative values is 0, and the slope for positive values is 1.0.
- Initialisation is important. Even during descent, $||\theta^{(l)}||$ may become zero and the descent gets stuck.
- It took 17 trials for $[512, 256, 128, 64]$ to actually start Training correctly

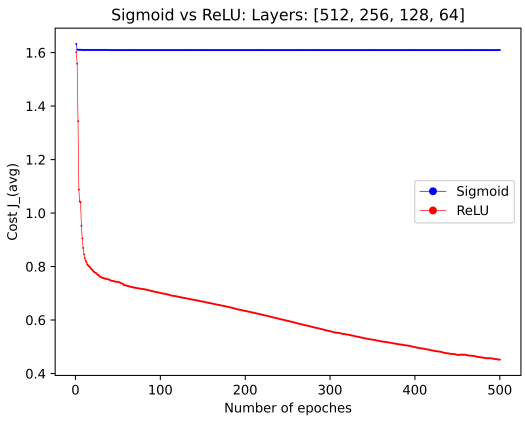


Relative Convergence: Sigmoid vs ReLU









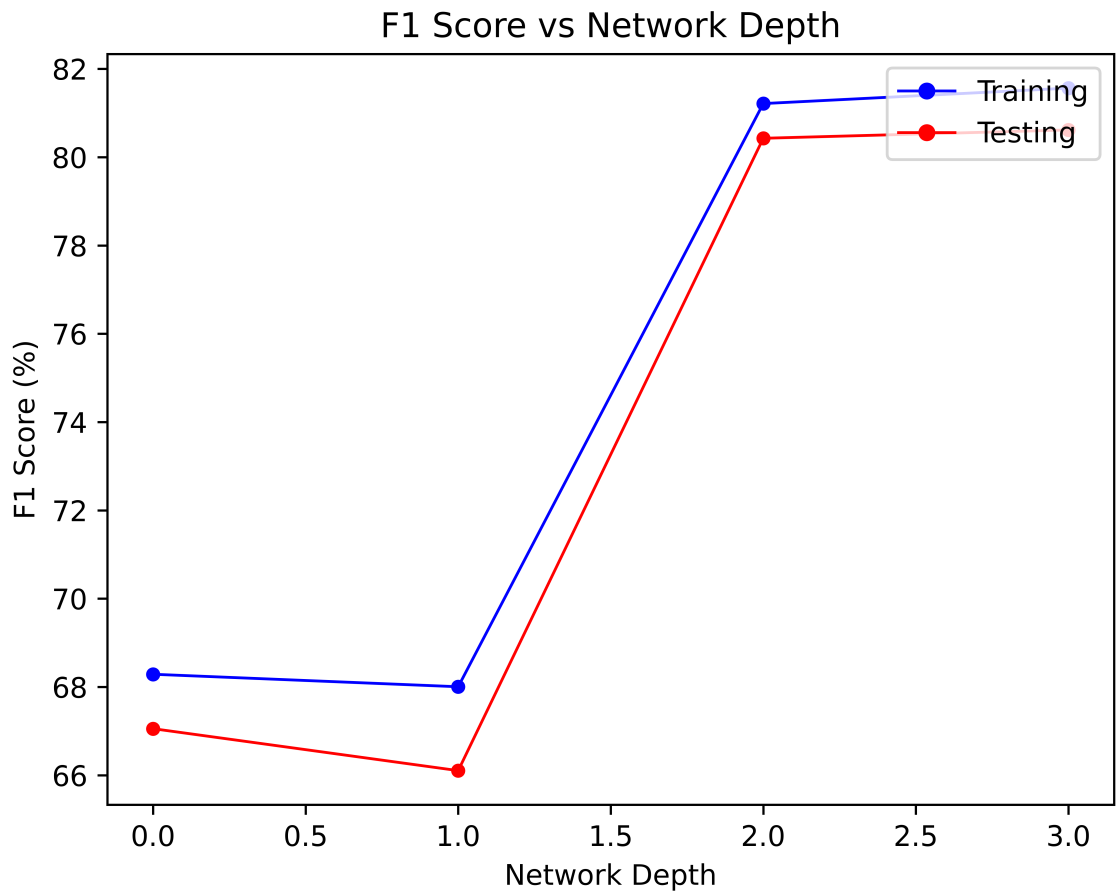
- Clearly ReLU outperforms Sigmoid in every case. The `loss of information` in deeper networks in Sigmoid is not a problem in ReLU as it `saturates` on only one side of 0, as long as θ is in a certain region of space (atleast few components. Which is why it needs multiple trials to be initilised in the correct region), there is no information loss (on that particular perceptron), while also preserving `non linearity` (otherwise the entire Neural Network would just be a linear classifier)
- After training for 500 epoches
- Training Data Scores (%)

Layers	Precision	Recall	F1 Score
$[512]$	68.997	67.800	68.287
$[512, 256]$	68.604	67.650	68.004
$[512, 256, 128]$	81.366	81.090	81.214
$[512, 256, 128, 64]$	81.908	81.430	81.564

- Testing Data Scores (%)

Layers	Precision	Recall	F1 Score
$[512]$	67.377	66.900	67.054
$[512, 256]$	66.098	66.200	66.104
$[512, 256, 128]$	80.486	80.400	80.428
$[512, 256, 128, 64]$	80.724	80.600	80.617

F1 Score vs Network Depth



- ReLU obtains the same J_{avg} values much faster than using `sigmoid`

Speed of Convergence: Sigmoid vs ReLU

- After 500 epoches, Loss values

Layers	Constant η	Adaptive η: Sigmoid	Adaptive η: ReLU
$[512]$	0.6497	0.7081	0.7039

Layers	Constant η	Adaptive η: Sigmoid	Adaptive η: ReLU
$[512, 256]$	0.6257	0.7477	0.7145
$[512, 256, 128]$	0.6544	1.5996	0.5000
$[512, 256, 128, 64]$	1.6103	1.6093	0.4521

- Adaptive η using Sigmoid Activation does worse than constant η in all but the last row. This was justified in (d).
- ReLU also suffers the same problem in the first 2 rows but does far better (much faster convergence) in the last 2 rows.

F1 Score: Sigmoid vs ReLU

- F1 Scores After 500 epoches on Test Set

Layers	Sigmoid	ReLU
$[512]$	66.913	67.054
$[512, 256]$	66.724	66.104
$[512, 256, 128]$	31.508	80.428
$[512, 256, 128, 64]$	31.508	80.617

- ReLU does far better than Sigmoid in the last 2 cases. In the first, ReLU does better and in the second, Sigmoid does better but the difference is very slight.

(f) ReLU activation: SK Learn

- Training Settings:

```
clf = MLPClassifier(hidden_layer_sizes= hidden_layer_arch , activation=
'relu',
                        solver = 'sgd', alpha=0, batch_size = 32 ,
learning_rate='invscaling' , max_iter= 1000 , tol = 1e-6)
```

Layers	`n_iter_`	Training time (hrs)
$[512]$	1000	1.01

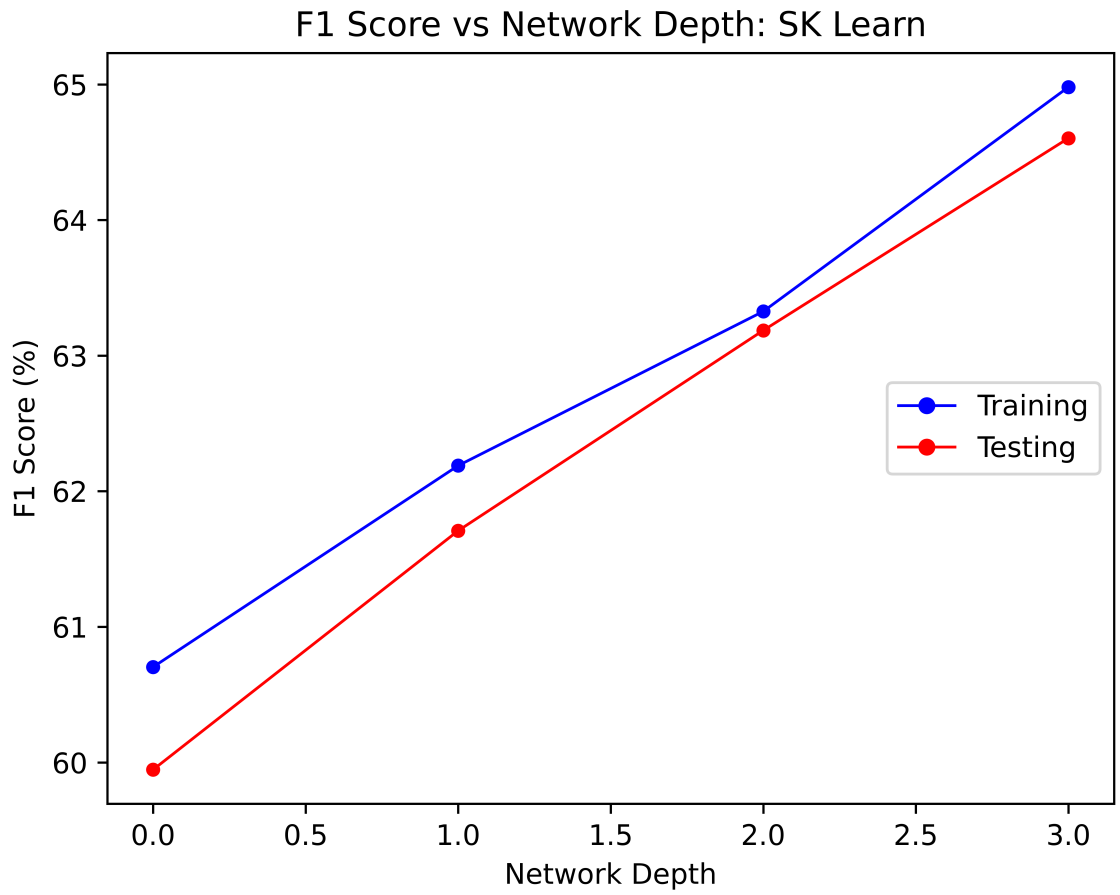
Layers	`n_iter_`	Training time (hrs)
$[512, 256]$	1000	3.00
$[512, 256, 128]$	1000	2.84
$[512, 256, 128, 64]$	1000	2.90

- All of them hit the `max_iter = 1000` while training but training times are similar to those allowed in (e)
- Training Data Scores (%)

Layers	Precision	Recall	F1 Score
$[512]$	63.402	59.240	60.704
$[512, 256]$	63.245	61.480	62.189
$[512, 256, 128]$	63.908	62.880	63.326
$[512, 256, 128, 64]$	65.539	64.570	64.980

- Testing Data Scores (%)

Layers	Precision	Recall	F1 Score
$[512]$	62.355	58.700	59.947
$[512, 256]$	62.361	61.300	61.709
$[512, 256, 128]$	63.413	63.100	63.186
$[512, 256, 128, 64]$	64.720	64.500	64.603



- Test Data F1 Score (%): ReLU: Scratch vs SK Learn

Layers	Scratch	SK Learn
$[512]$	67.054	59.947
$[512, 256]$	66.104	61.709
$[512, 256, 128]$	80.428	63.186
$[512, 256, 128, 64]$$	80.617	64.603

- Clearly, `scratch` outperforms `sk learn` in all 4 cases. But this could be due to differing convergence criterial too. I also trained `sk learn` with `tol = 1e-4` in which case it convergs before hitting `max_iter`. The Scores in those cases were atmost 1% - 2% different from these.