EE 782: Assignment 1 Report

Team Members:

1) Abhishek Sandeep Tanpure (17D070018)

2) Anwesh Mohanty (170070009)

3) Rohan Bansal (170070058)

INTRODUCTION:

As mentioned in the problem statement, training a deep neural network is a process of trial and error. Based on the performance of any model first on the training dataset and then the validation dataset, we obtain a good estimate of the hyperparameters which will give us good accuracy on the final test dataset. These “optimal” hyperparameters are usually obtained by performing a wide variety of experiments on the dataset. Hence in this assignment, we have identified certain hyperparameters that we think are one of the most important ones that are used in any deep learning model and aim to design experiments to extract the optimal value of these hyperparameters.

HYPERPARAMETER SELECTION:

1. The first parameter that we choose is the **learning rate** of the model. The learning rate is one of the most essential hyperparameters as it decides the rate of change of parameters. For a high learning rate, we might shoot past the global minima, and for a low learning rate, we will take a lot of time to converge at the local minima. Hence it is crucial to select the learning rate such that the chances of the model encountering the aforementioned problems are minimized. We perform this experiment by using different learning rates in different optimizers (we use SGD, RMSProp, and Adam as these are the main ones and others are slight modifications of these). We also check if the size of the training dataset affects the overall result.
2. Since we are looking at the learning rate, it is also beneficial to look at different **learning rate schedulers** as we might want to vary the learning rate in different regions of learning i.e. a high learning rate during the initial stages when we are far away from the global minima and a slow learning rate when we are close to it. Hence we provide a small study on different learning rate schedulers and report the performance for different **decay rates** in each individual scheduler as well.
3. Regularization is a key part of the training of any machine learning model. It helps prevent overfitting as well as ensures that the model weights do not

assume arbitrarily high and undesirable values. We explore the performance of the model under different popular **regularizers** like the Lasso (L1), Ridge (L2), and the Elastic Net (L1L2) regularizers. To measure the performance of the model under these regularizers, we report the accuracy of the model under different values of **weight decay**. This also gives us the performance of the different regularizers used for a fixed value of weight decay.

4. Next, we experiment with the **keep probability/dropout** applied in each of the convolution layers. The dropout value of p indicates that 100p% of the neurons are dropped during training. This induces a sort of regularization in the training process and helps in preventing overfitting. We analyze this for different values of dropout and as well vary the size of the dataset used for training to gauge its effect on the overall learning. Following this, we also show the model performance for a combination of dropout with L2 regularizer.

LOSS FUNCTIONS & METRICS:

Identifying the correct loss function and metrics to judge the model performance is almost as important as, if not more, hyperparameter tuning. From the provided papers in the problem statement and some other resources, we chose a total of **seven** loss functions for our model. These are-

1. Standard Dice Loss
2. Focal Loss
3. IOU Loss
4. Combo Loss
5. Weighted Cross-Entropy
6. Dice + Weighted Cross-Entropy
7. Tversky Loss (Extra from our side)

Based on the values of optimal hyperparameters that we obtained (using the standard dice loss and dice metric), we evaluate our model using other loss functions and check if we can replace the standard dice loss function with any other loss function without compromising on accuracy.
Given the time constraints, we were unable to check the performance with different metrics other than the dice metric.

EXPERIMENT DESIGN STRATEGY:

Since we are trying a wide range of hyperparameters, for each hyperparameter we used a different experimentation strategy keeping all other parameters constant. For each of these we have written a script to write all the data into a csv file and then plot the graphs using that data. The ‘runme.py’ file has to be used to run the code:

1) Learning rate: Here we vary the learning rate from 10e-5 to 10e-1 for training data sizes of 100%, 80% , 60%, 40% and 20% of the entire training data. This is done thrice for each optimizer ‘Adam’, ‘Rmsprop’ and ‘SGD’. The aim here is to see just the variation with learning rate for different optimizers and varying data size. Other hyperparameters are kept as follows:
   i) LR decay = 0;
   ii) Dropout = 0;
   iii) No Regularization;
   iv) Dice Loss function;

2) Learning rate scheduler: In this we compare for categories of Lr scheduler namely ‘No decay’, ‘Inverse Time decay’, ‘Exp. time decay with steps’ and ‘Exp. time decay without steps’, for training data sizes of 100%, 80% , 60%, 40% and 20% of the entire training data. This is done thrice for each optimizer ‘Adam’, ‘Rmsprop’ and ‘SGD’. The aim here is to see just the variation with learning rate decays for different optimizers and varying data size. Other hyperparameters are kept as follows:
   i) Learning rate = 10e-4;
   ii) Dropout = 0;
   iii) No Regularization;
   iv) Dice Loss function;

3) Regularization: In this first we compare different values of lambdas for weight decay for training data sizes of 100%, 80% , 60%, 40% and 20% of the entire training data. Secondly we compare different dropout probabilities for training data sizes of 100%, 80% , 60%, 40% and 20% of the entire training data. Thirdly we take different regularizers like ‘L1’, ‘L2’ and ‘elastic net’ and vary the dropout probabilities and lambdas for weight decay. The aim here is to see just the variation with various types of regularizers, dropout values, weight decay and data size. Other hyperparameters are kept as follows:
   i) Learning rate = 10e-4;
   ii) Optimizer = Adam;
   ii) LR decay = 0;
   iv) Dice Loss function

RESULTS:

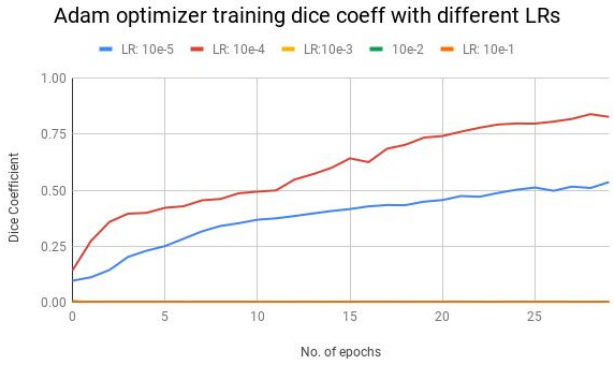
1) Learning Rate (Optimizer, LR scheduler)

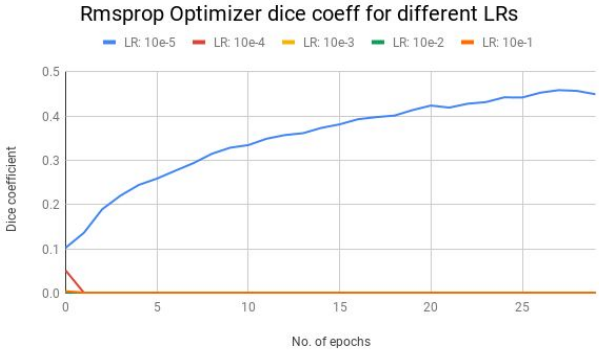
I) Optimizers:

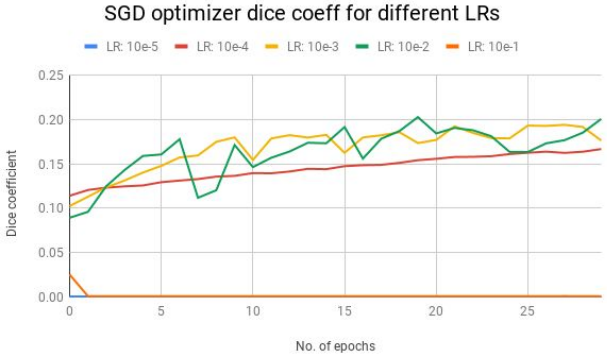
i) Variation with Learning rates:

We have used 3 different types of Optimizers which include ‘Adam’, ‘Rmsprop’, and ‘SGD’. For each of the optimizers, the learning rate was varied from 10e-5 to 10e-1. The training Dice Coefficient with each epoch has been shown in plots below.

From the first plot it can be seen that the dice coefficient has almost saturated with 30 epochs for a learning rate of 10e-4, whereas for the learning rate of 10e-5 the training is slower and takes much more time for saturation. One can also observe the low dice coefficient for Learning rates greater than 10e-3. We suppose that due to high learning rates the gradients overshoot the minima thus giving very high losses and hence the low dice coefficient. The ‘Val’ dice scores have also been shown in table 1).







The same conclusion applies to ‘Rmsprop’, however for a learning rate of 10e-5 the training dice coefficient had saturated to a value of around 0.45 which is quite less than that obtained by Adam.

Lastly, for SGD optimizer we observed a similar performance for learning rates of 10e-5, 10e-4, and 10e-3. But this time the dice coefficient saturated to only 0.20.

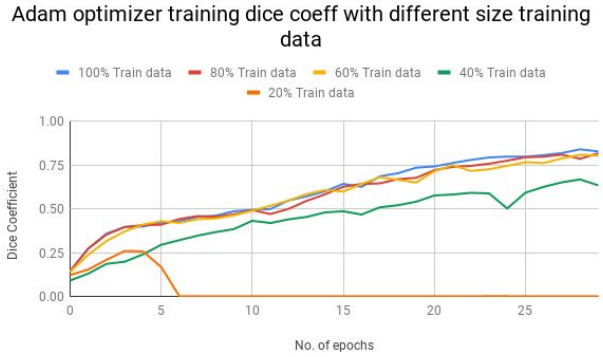
Inferences :

Rmsprop is the most sensitive to learning rates among all the optimizers. Since only a change from 10e-5 to 10e-4 reduced the performance drastically. Whereas SGD is the least sensitive to changes in learning rates since the variation of learning rates from 10e-5 to 10e-3 did not have an impact on the model performance.

The ‘Val’ dice for SGD and RmsProp with a change in Learning rate is also shown in table 1). Thus from all these, we conclude different optimizers perform optimally on different learning rates. However, most of these perform poorly for learning rates above 10e-3. For the segmentation task using UNET, ‘Adam’ optimizer gives the best performance, followed by ‘Rmsprop’ and then ‘SGD’.

ii) Variation with the amount of training data:

1) Adam Optimizer: From Fig4) we can infer that as long as we take at least 60% of the training data, the performance remains the same i.e the training dice coefficient saturates around 0.80. However, a clear decrement in the performance can be observed for 40% of training data, with the training dice reaching only 0.60. In fact, the model failed to learn anything for 20% of the train data.



2) Rmsprop: There was a steady decline in performance with a decrease in the amount of training data from 100% to 20%. For Rmsprop as well 20% of training data did not manage to learn the model at all.
3) SGD: The decrease in training data did not have much impact until 60% of data. After which the performance suddenly dropped to a very low value.

Learning Rate --->		0.00001	0.0001	0.001	0.01	0.1
Optimizer	%age data					
ADAM	100%	0.479	0.642	0.001	0.001	0.001
	80%	0.464	0.624	0.001	0.001	0.001
	60%	0.399	0.598	0.001	0.001	0.001
	40%	0.371	0.443	0.001	0.001	0.001
	20%	0.308	0.001	0.001	0.091	0.001
SGD	100%	0.0006	0.177	0.217	0.221	0.001
	80%	0.0004	0.149	0.207	0.198	0.001
	60%	0.093	0.116	0.188	0.206	0.001
	40%	0.099	0.0007	0.0009	0.0009	0.001

	20%	0.092	0.117	0.0009	0.001	0.247
RMSPROP	100%	0.448	0.001	0.001	0.001	0.001
	80%	0.001	0.001	0.0745	0.001	0.089
	60%	0.35	0.001	0.001	0.074	0.001
	40%	0.35	0.001	0.001	0.001	0.001
	20%	0.266	0.001	0.001	0.001	0.001

`Table 1: Validation dice for variation of learning rate with optimizer

Inference:

The best optimizer to use when we have less data is 'Adam' since even for 20% train data the dice coefficient on the validation set is 0.334, whereas the dice coefficient is lower for the other two optimizers.

II) LR scheduler:

We implement three types of learning rate schedulers using Keras.

1. Inverse time decay

$$lr = \frac{initial\ lr}{1+\frac{decay\ rate * steps}{decay\ steps}}$$

2. Exponential time decay without steps

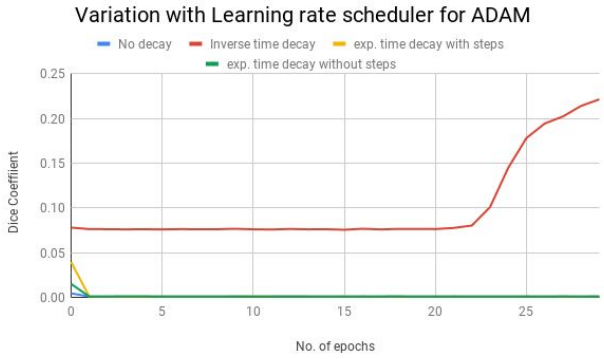
$$lr = initial\ lr * (decay\ rate)^{\frac{steps}{decay\ steps}}$$

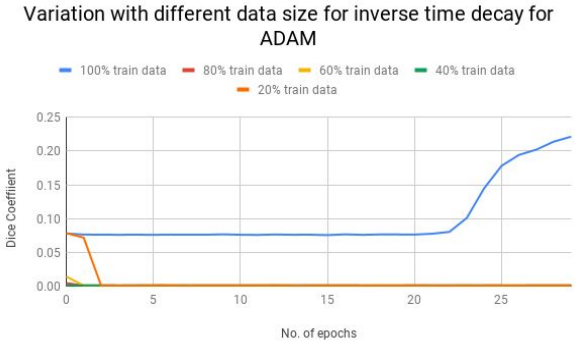
3. Exponential time decay with steps

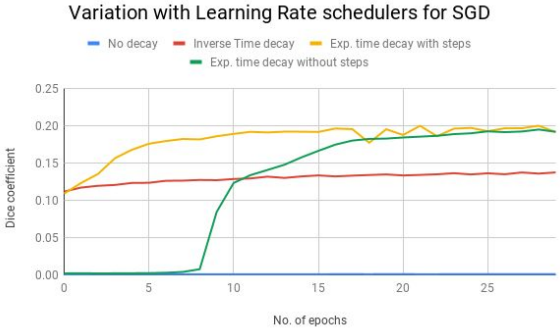
$$lr = initial\ lr * (decay\ rate)^{\frac{1}{[\frac{decay\ steps}{steps}]}}$$

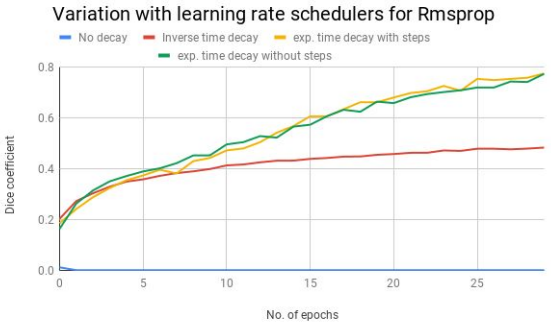
where [.] is the greatest integer function.

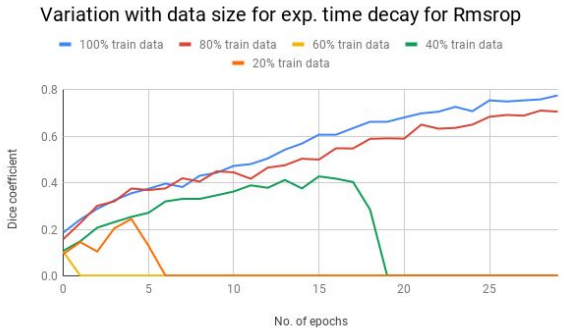
The learning rate has been chosen appropriately such that variation of decay can be observed distinctly.











Learning rate decay		No decay	Inverse Time Decay	Exponential decay without steps	Exponential decay with steps
Optimizer	%age data				
ADAM	100%	0.001	0.206	0.001	0.001
	80%	0.001	0.001	0.001	0.001
	60%	0.6	0.001	0.001	0.001

	40%	0.001	0.001	0.001	0.001
	20%	0.001	0.001	0.001	0.001
SGD	100%	0.0009	0.140	0.215	0.218
	80%	0.205	0.113	0.200	0.202
	60%	0.194	0.110	0.001	0.207
	40%	0.167	0.145	0.0009	0.200
	20%	0.0007	0.122	0.149	0.001
RMSPROP	100%	0.001	0.448	0.596	0.634
	80%	0.001	0.434	0.6	0.596
	60%	0.440	0.379	0.001	0.001
	40%	0.001	0.331	0.001	0.395
	20%	0.295	0.001	0.001	0.001

Validation dice for variation of decay with optimizer

i) Variation with type of decay

1. ADAM : Only Inverse time decay works well. Only Inverse time decay could provide the necessary decay to prevent overshoot.
2. SGD : Exponential decay with and without steps work equally well. Inverse time decay provides a bit more decay to slow the training.
3. RMSProp : Same observation as SGD. Exponential decay with and without steps work equally well. Inverse time decay provides a bit more decay to slow the training.

ii) Variation with training data size

1. ADAM : The model overshoots except 100% training data. It is because of a larger update step when training with less data. Less data leads to higher loss which results in a larger update step for a particular learning rate.
2. SGD : The model performs well for 100% and 80% data but overshoots for 40% data. This can be explained by the same explanation as ADAM.

3. RMSprop : For 20% data, model overshoots after 5 epochs. For 40% data, model overshoots after 18 epochs. For 80% and 100% data models perform well. Again it can be explained by the same explanation as RMSprop.

Thus, we witness similar trends in all of the optimizers for variation in training data size.

Inferences:

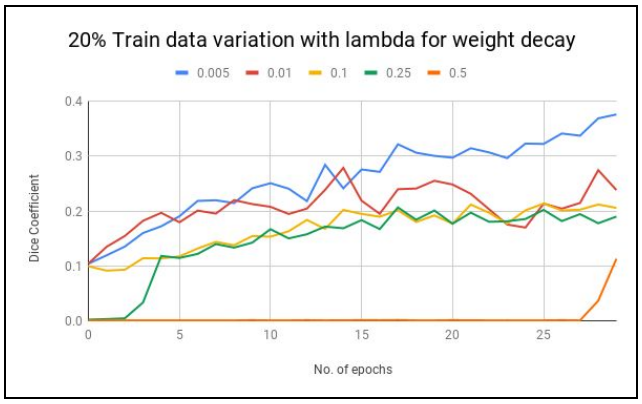
- ADAM optimizer is more vulnerable to overshoot, so we need a higher decay rate for ADAM optimizer.
- SGD and RMSProp do well for all the decay types.
- If the overshoot problem still persists, we may try to increase the training data size and lower the update step. But that may lead to overfitting, so there is trade off at work.
- These observations help us in choosing the right learning rate scheduler for any optimizer. The choice obviously depends on the initial learning rate and decay rate too.

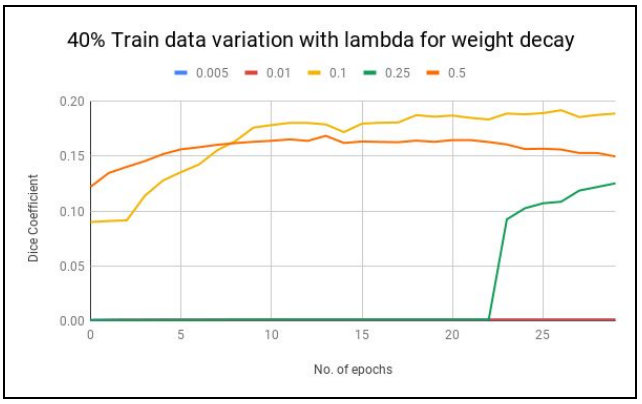
2) Regularizers:

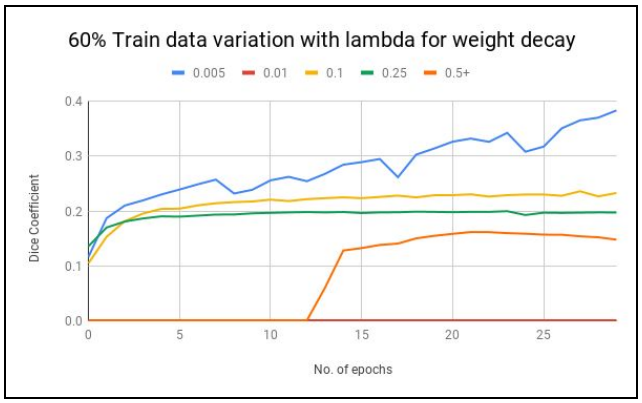
a) For this part we add a L2 regularization term with all the convolution layers and introduce a kernel regularizer = bias regularizer = lambda. Tuning lambda will give us different constraints on the parameters in the model and subsequently a different value of loss and hence varying accuracy. Therefore we choose 5 different values of lambda, {0.005, 0.01, 0.1, 0.25, 0.5}. We also check the dependence of the final accuracy on the size of the training dataset used for different values of lambda. The final plots and inferences obtained from experimentation are shown below-

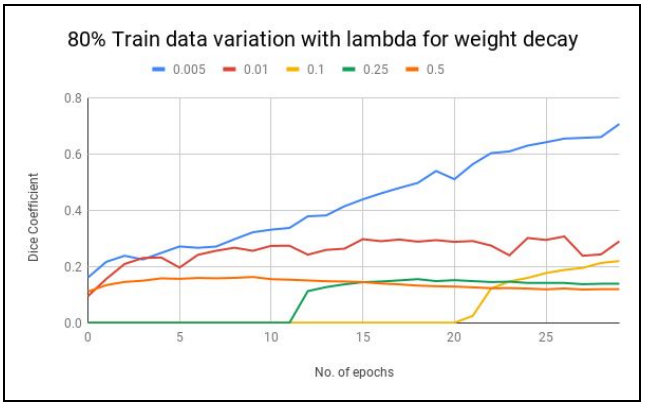
Lambda / Regularisation	0.005	0.01	0.1	0.25	0.5
L1	0.251	0.164	0.001	0.092	0.092
L2	0.55	0.001	0.21	`0.132	0.122
Elastic net	0.145	0.216	0.001	0.092	0.092

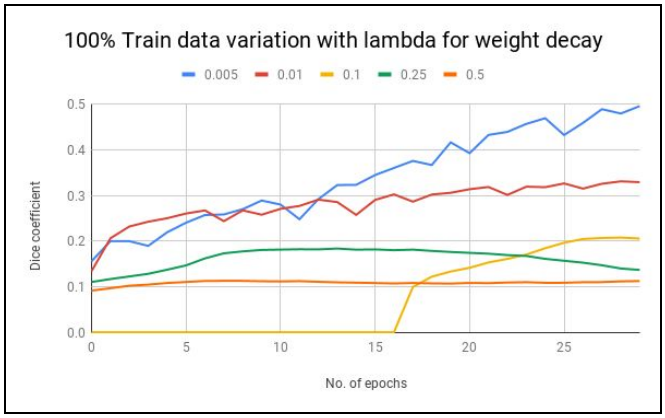
Validation dice for Regularisation variation











Inferences:

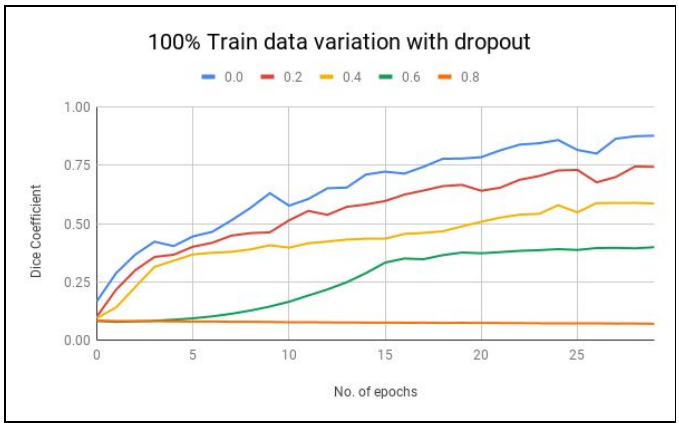
- For a fixed quantity of training samples, there is a clear trend of which values of regularization perform better.For example when we only use 20% of the training dataset, the accuracy values follow the following trend: 0.005 > 0.01 > 0.1 > 0.25 > 0.5. But the same trends are not observed for any other size of the training dataset i.e. for **different sizes of training dataset used we get a different set of regularizing values which perform better**.
- Small regularizing values like 0.005 and 0.01 give almost a consistent performance apart from certain instances like when 40% of the training data is used. There is a steady increase in the accuracy when using these values which

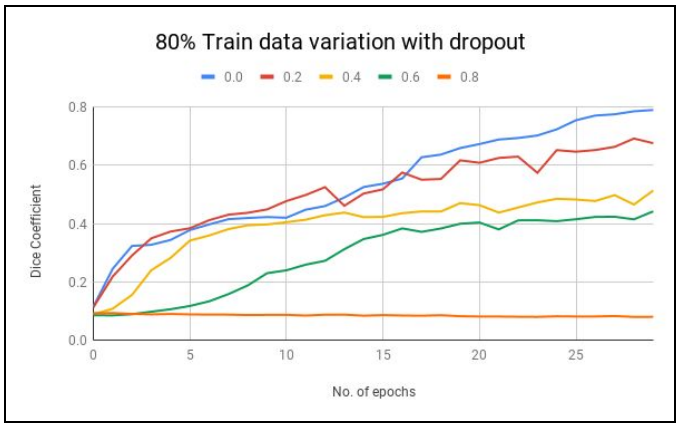
suggests that we should keep the regularizer hyperparameter close to these **low values (0.005-0.01) to achieve good results**.

- For large values of regularizers, the performance saturates around 0.1 to 0.2 irrespective of the size of the training dataset used as can be seen in all of the above plots. Hence **high values (>0.1) are not suitable for this model**.
- The validation accuracy table for different sizes of training dataset and lambda values is shown below. There are some anomalous values in between possibly due the model getting stuck in a local minima maybe. The overall trend is hard to predict but one observable trend is that **generally for a fixed size of the training set, the accuracy decreases as the lambda is increased**.

Lambda / %age data	0.005	0.01	0.1	0.25	0.5
100%	0.445	0.337	0.200	0.136	0.114
80%	0.569	0.249	0.217	0.127	0.113
60%	0.339	0.001	0.217	0.193	0.154
40%	0.001	0.001	0.202	0.145	0.167
20%	0.329	0.242	0.198	0.197	0.112

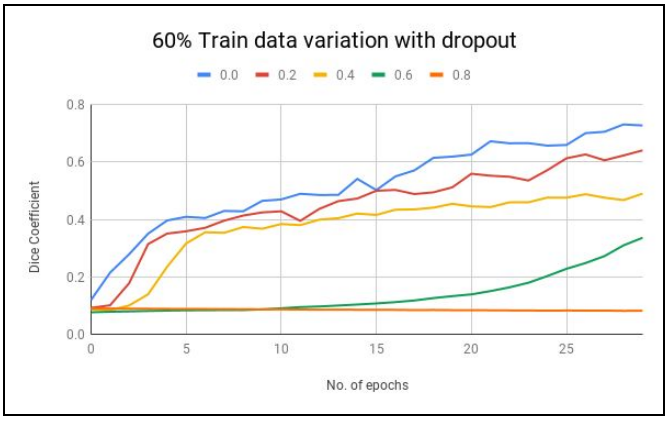
Validation dice for variation of regularisation with %age data

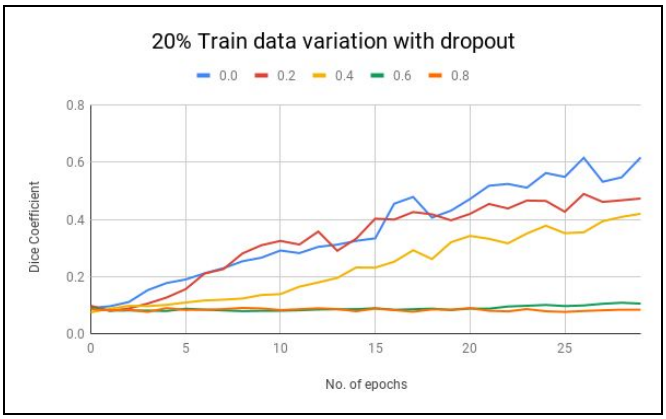




b) Like previous methods we chose 5 different values (0, 0.2, 0.4, 0.6, and 0.8), while keeping other hyperparameters constant, for the keep probability to use for dropout in each convolution layer in our U-Net model. The performance of the model is judged for

different sizes of training datasets (but the test dataset is left unchanged). Based on the





experiments carried out, the following plots and inferences were obtained-

Inferences:

- As the size of the training dataset is reduced, the performance for high values of dropout decreases radically. For example, there is a steady increase in the dice coefficient for dropout=0.6 when 100% or 80% of training data is used for training. But when the percentage of training data is reduced to <=60%, the performance of the model falls noticeably as evidenced by the fall in the rate of change of the dice coefficient.
- For very high values of dropout (here 0.8) and another fixed set of parameters, the model is unable to learn from the training dataset since a large number of nodes are dropped randomly for training. Hence for the chosen set of parameters, the model is not able to improve from the initial dice coefficient of around 0.1 achieved in the starting epochs.

- From the plots at different dropout probabilities, it can be seen that the performance for 0, 0.2 and 0.4 is comparable to some extent with the best performance occurring at value 0. This indicates that the model is performing better without dropout which might imply that the model is not undergoing over-fitting at any point and the complexity of the model can further be increased to get further results. Also the accuracy is dependent on the training dataset as evidenced by the fall in accuracy as the size of the training sample is reduced. The value of 0.6 works slowly compared to smaller values when we have a large number of training samples, but gives disappointing results when the training samples are further reduced. 0.8 dropout value doesn't give good results for any size of training samples. Hence overall as **dropout increases, training accuracy decreases and as the size of the training sample decreases the accuracy with/without dropout decreases.**
- The highest value of the dice coefficient obtained during 30 epochs on the validation set is reported in the table below. Notice that these values are consistent with the trends observed with the training dataset which implies that the training is happening normally and that we can choose the **dropout probability as 0 for further experiments as it will give us the best results.**

Dropout / %age data	0	0.2	0.4	0.6	0.8
100%	0.651	0.463	0.250	0.207	0.001
80%	0.575	0.351	0.235	0.164	0.001
60%	0.550	0.377	0.235	0.121	0.001
40%	0.469	0.356	0.214	0.149	0.001
20%	0.240	0.287	0.137	0.095	0.001

Validation dice for variation of dropout with %age data

c) As observed in the previous part, here also we get a similar trend. The dropout probability in the upsampling and downsampling blocks are varied using values from the set {0,0.2,0.4,0.6} and the dropout values in the UNet architecture is varied from the set {0,0.3,0.5,0.7}. For this we report only the final table of the maximum validation dice coefficient as the training set shows a similar trend as compared to the previous part.

Lambda in Upsample/Downsample Block / Lambda in Unet	0	0.2	0.4	0.6
0	0.651	0.510	0.316	0.152
0.3	0.456	0.329	0.204	0.118
0.5	0.260	0.198	0.107	0.073
0.7	0.008	0.008	0.006	0.002

Validation dice for different dropout in different layers

d) Dropout is typically used to prevent overfitting i.e. either when we have too much data or when the model is very complex and has several redundant features. So ideally if we have enough data (assuming that enough data means that overfitting is not happening), then dropout is **not** required.
To check if we need dropout or not for a particular quantity of training samples, look at the variation of the accuracy of the validation set. If the accuracy on the validation set is increasing with more samples then we can consider more data. If the accuracy has saturated or started decreasing, then it may suggest that we are overfitting and need to start using dropout.

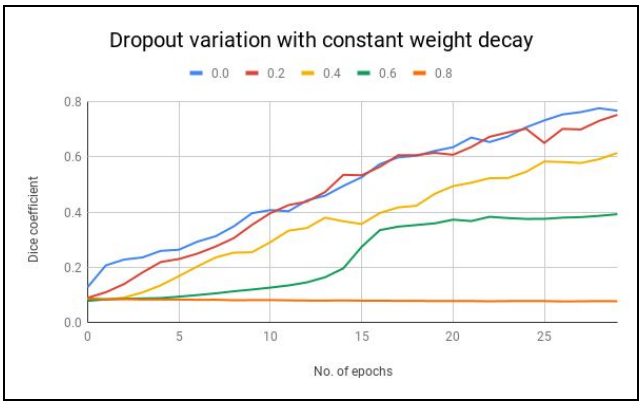
e) For this part, we introduce both dropout and weight decay and report the model performance. The validation dice coefficient for different coefficients are reported below-

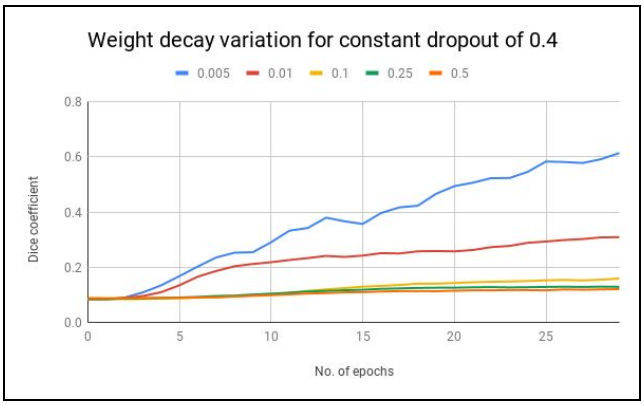
Lambda / Dropout	0.005	0.01	0.1	0.25	0.5
0	0.590	0.279	0.225	0.126	0.121
0.2	0.504	0.219	0.238	0.127	0.113
0.4	0.262	0.319	0.147	0.124	0.116
0.6	0.198	0.294	0.153	0.126	0.118
0.8	0.001	0.116	0.270	0.175	0.153

Validation dice for weight decay with dropout

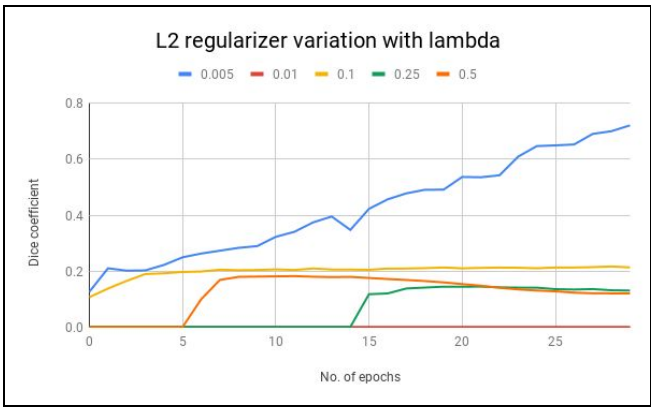
Inferences:

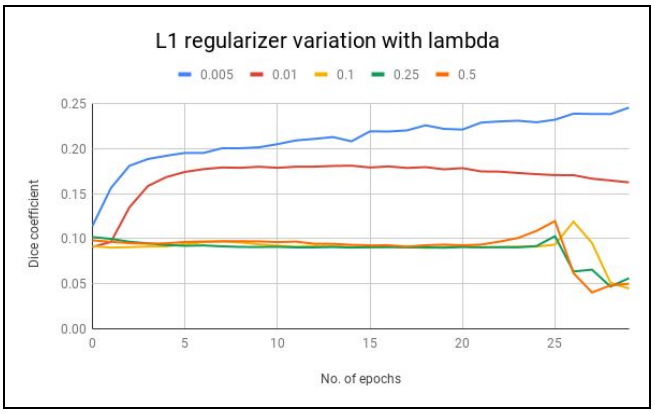
- A general trend observed in **validation** is that for a **fixed value of dropout and varying values of lambda, the accuracy first increases to reach the maximum for a certain value (or starts from a maximum eg. lambda=0.005 and dropout=0) and then decreases as lambda is further increased**. For example for dropout of 0.4, the maximum accuracy is achieved for lambda=0.01 and for dropout=0.8, it is obtained at lambda=0.1.
- The trend of accuracy dependence during **validation** on dropout is hard to predict. For **smaller values of dropout (like 0.005), the trend is consistent** with that obtained in the previous part i.e. **the accuracy decreases as dropout increases**. But **as lambda increases, the trend in dropout becomes unpredictable with the accuracy even almost saturating or even increasing at higher values of dropout** (look at the column of 0.25 and 0.5).
- The trend during **training** is consistent with the previous results i.e. for a **fixed dropout, the training accuracy decreases with increasing lambda** and **for a fixed lambda the training accuracy decreases with increasing dropout**. We attach two plots below to further explain this point.

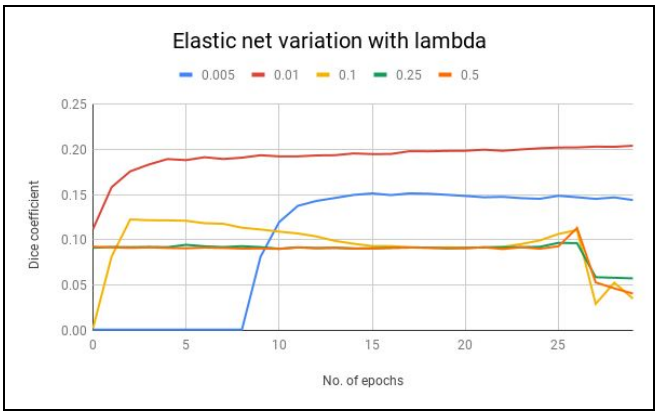


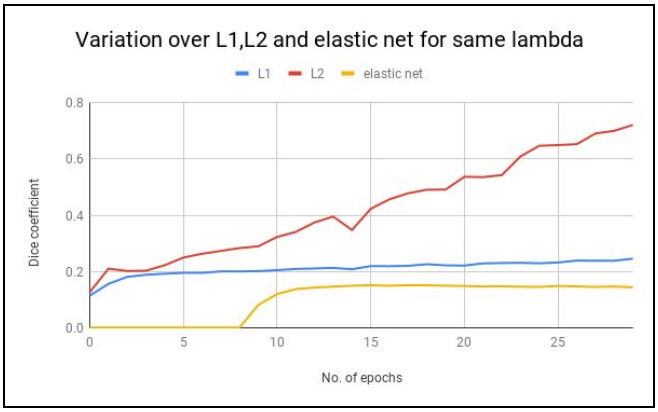


f),g) For these two parts, we explore the use of other regularizers and compare the benefits against the L2 regularizer used in the previous parts. We chose the following set of regularizer values to perform our experiments {0.005, 0.01, 0.1, 0.25, 0.5}. The plots for different regularizers is shown below and also a plot with all three together is shown-





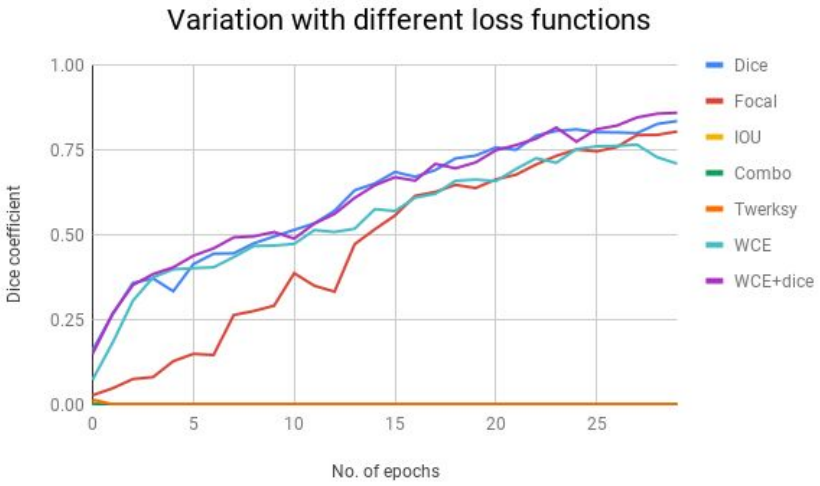


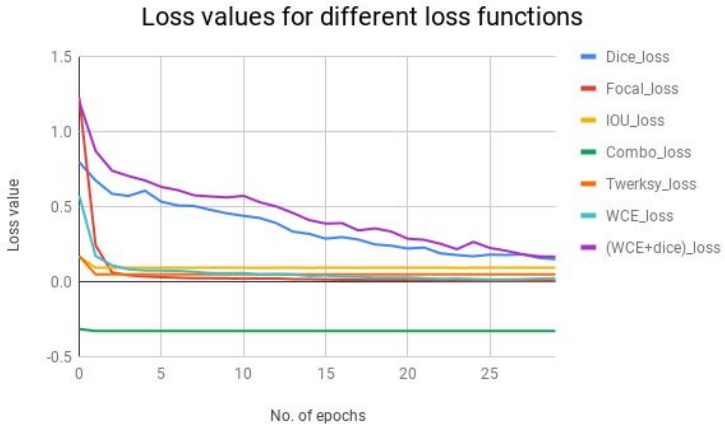


The L2 plot is as observed previously with the training accuracy saturating for most of the values around 0.2 except for lambda=0.005. For this value the training occurs properly and we get a good result as the training dice coefficient increases. But for the same values of lambda, **elastic net and L1 regularizer introduce a saturating effect** for almost all values of lambda. The maximum dice coefficient doesn't exceed 0.25 for these two regularizers. In the overall comparison graph, notice that as the number of epochs increases the L2 regularizer outperforms the other two by quite a large margin. Hence the **elastic net and L1 regularizer are not suitable for our current model**.

3) Loss functions:

In this section, we use the optimal values of the hyperparameters obtained in the previous sections and evaluate the model performance against different loss functions given our obtained optimal hyperparameters. The loss functions used for the simulations are ‘Dice’, ‘Focal’, ‘IOU’, ‘Combo’, ‘Twersky’, ‘Weighted Cross-Entropy’ and ‘Weighted Cross-Entropy + Dice’. Some of these losses are not mentioned in the paper for the assignment. However, we thought that it would be a good exercise to see their performances on the task at hand. Since these losses have been thought of lately and have shown good performance in certain segmentation tasks.





Loss Function	Validation dice
Standard Dice Loss	0.556
Focal Loss	0.627
IOU Loss	0.001
Combo Loss	0.001
Tversky Loss	0.001
Weighted Cross-Entropy	0.578
Dice+Weighted Cross-Entropy	0.662

Validation dice for variation of the loss function

From the above two plots and the table, it can be observed the best performance on training data was obtained for ‘Dice’, ‘Focal’, ‘WCE’, and ‘WCE+dice’. Notice that the best performance on the validation set was obtained for the combined loss function ’Dice + Weighted Cross-Entropy’. Dice loss and Cross-Entropy loss are the two most widely used loss functions and thus one might expect an even better performance using a combination of both the losses.

The other losses have shown good performance in segmentation tasks. However, we infer that the poor performance for the other losses is due to the fact that all other hyperparameters have been tuned to obtain the best performance on the basic loss used for all the simulation i.e the dice loss. Clearly, a better weight initialization, a different optimizer, or a different model architecture can give good results when using these loss functions.