Artificial Neural Networks and Deep Architectures, DD2437

Short report on lab assignment 1

Learning and generalisation in feed-forward networks — from perceptron learning to backprop

Chiachen Ho, Angelos Stais and Ruxue Zeng

September 15, 2021

1 Main objectives and scope of the assignment

Our major goals in the assignment were

- to design and apply networks in classification, function approximation, and generalization tasks
- to identify key limitations of single-layer networks
- to configure and monitor the behavior of learning algorithms for single- and multi-layer perceptrons networks
- to recognize risks associated with backpropagation and minimize them for robust learning of multi-layer perceptrons.

2 Methods

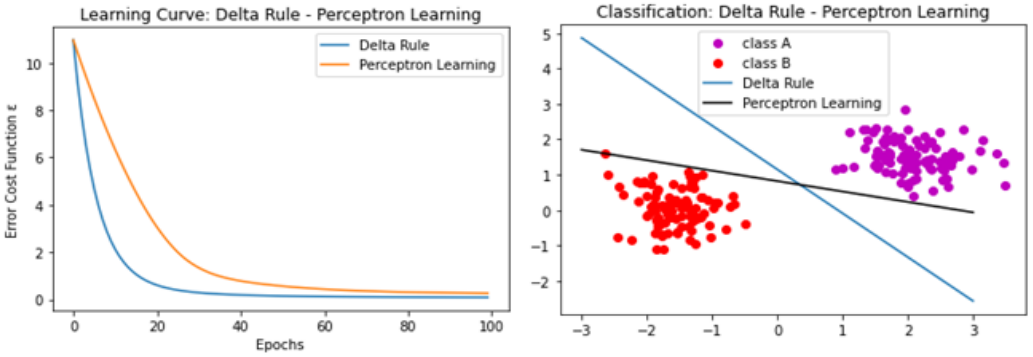
We have used two different languages in this lab:

- Python: with libraries NumPy, Matplotlib, sklearn
- Matlab: no specific tool boxes have been used

3 Results and discussion - Part I

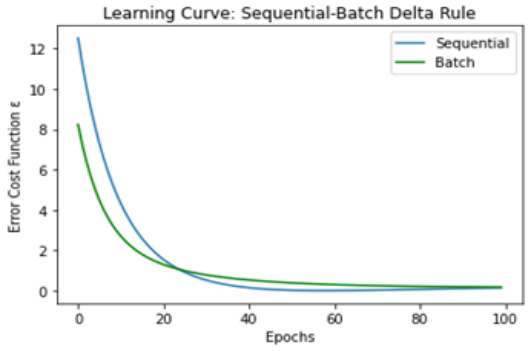
3.1 Classification with a single-layer perceptron

Linearly Separable Dataset: Delta Rule in Batch Mode – Perceptron Learning



η (learning rate) of 0.0001 seems to be an appropriate value for both algorithms to converge at around 100-200 epochs depending on the generated dataset. If a greater value is applied the error diverges. If a smaller value is applied the convergence of the algorithms is slowed down.

From the comparing graphs, it can be concluded that Delta Rule is preferable because converges a lot faster for the same learning rate and also because the decision boundary of Perceptron Learning is close to the limit points which leads to a higher possibility of misclassification of some data.

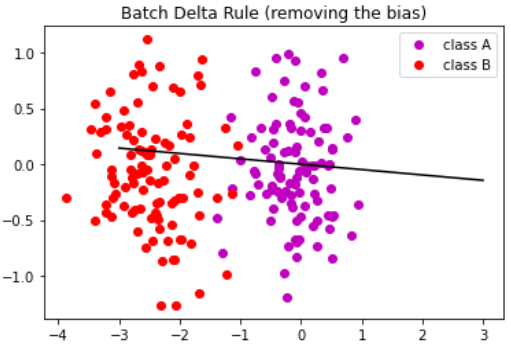


Linearly Separable Dataset: Sequential – Batch Learning with Delta Rule

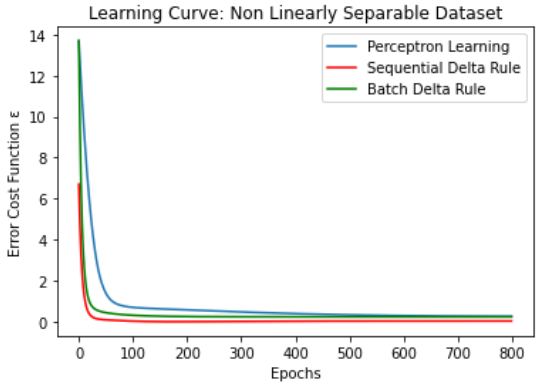
From the comparing graph, it can be concluded that the two versions of Delta Rule achieve similar results with Sequential Delta Rule converging a little quicker to a lower mean square error.

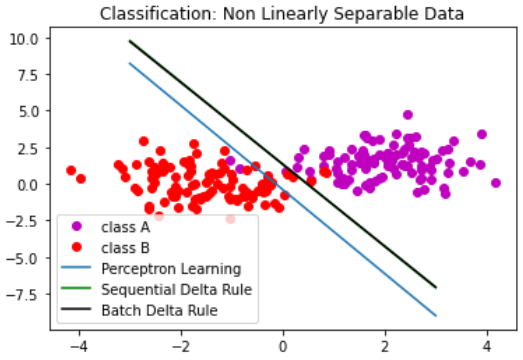
Linearly Separable Dataset: Removing Bias in Batch Delta Rule

Having removed the bias the perceptron is not able to classify the samples if one of the classes has a mean which is almost the origin, even after as many as 1000 epochs. The error will not converge to 0 and the perceptron is not able to correctly classify the sample from one or both of the classes.



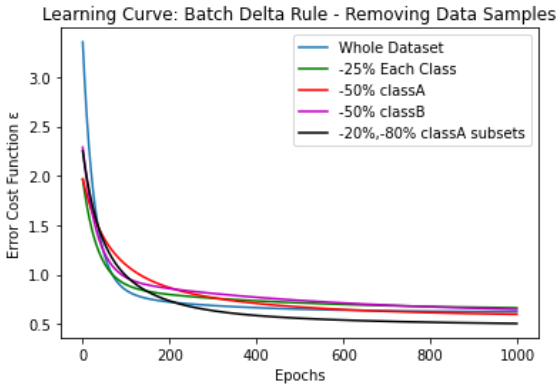
Non Linearly Separable Dataset: Delta Rule in Batch, Sequential Mode – Perceptron Learning





From the above graphs, it can be concluded that Perceptron Learning is significantly slower than both versions of Delta Rule. Also Sequential and Batch Delta Rule converge to lower mean square error. All the algorithms misclassify some data as expected.

Different Versions of Non-Linearly Separable Dataset



It can be concluded that if we remove some samples from each class the mean square error remains more or less close to having the whole dataset, the algorithm converges a little slower as we remove more samples and the decision boundary moves a little bit to corresponding direction. In the last scenario, the decision boundary is totally different. Indeed we have almost removed all the samples from class A which had a positive abscissa so the dataset is now almost linearly separable.

3.2 Classification and regression with a two-layer perceptron

3.1.1 Classification of linearly non-separable data

In this part, we want to build a two-layer perceptron network with Backprop using generalized Delta rule to separate linearly non-separable data.,And we want to examine how well it performs in separating the two classes. We choose to implement the batch learning process, because it gives better accuracy than the sequential approach, and it is less time-consuming according to our experiments.

First of all, we want to analyze its performance by varying the number of hidden nodes, for example for *nhidden* = 3, 8, 15, 20, and 30. ($\eta = 0.01$)

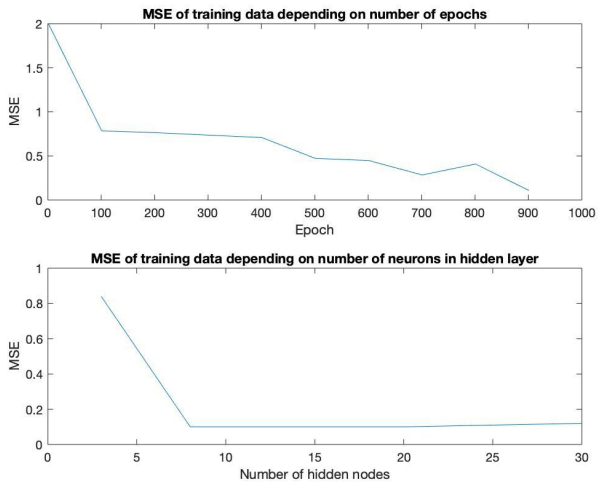


Figure - MSE of Training Data depending on epochs and *nhidden*

According to the figure on the left-hand side, we observe that the MSE (*Mean Square Error*) of training data keeps decreasing when the number of epochs increases, so we choose **epochs = 1000**. When it depends on the *nhidden*, it keeps decreasing.

Unfortunately, we can't have a number of hidden nodes to perfectly separate all the available data, because two classes are overlapped. We remark that the MSE has been stuck on its local minimum after 10 hidden nodes, and it is not equal to 0. So that means we won't obtain a perfect separation for all the nodes.

Now we want to study how the number of training samples could affect the performance of our network. For that, we create two data set based on our original data:

- dataset 1: 80% of data for training and 20% for validation
- dataset 2: 20% of data for training and 80% for validation

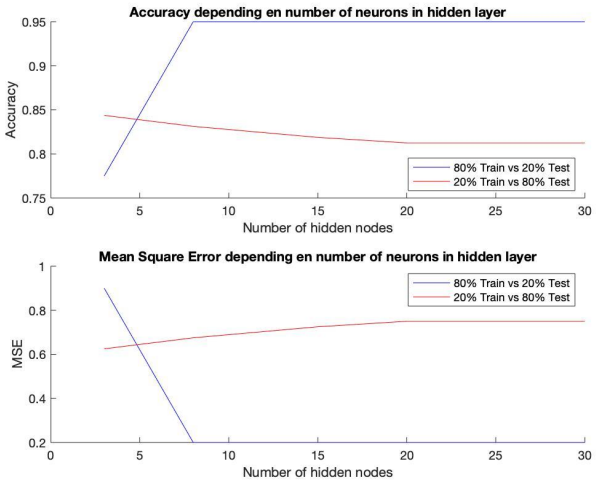


Figure - Comparison between accuracy of 80% Training vs 20% Training dataset

Clearly, according to the figure on the right-hand side, the network trained with dataset 1 has better performance with *nhidden*>5 (higher accuracy and lower MSE). Furthermore, we notice that unlike the network on dataset 1, the performance of the network trained with dataset 2 doesn't vary much. A network trained with a few data samples has more limits on performance.

And finally, in this part, let's compare the decision boundary between the network with 3 hidden nodes and 30 hidden nodes.

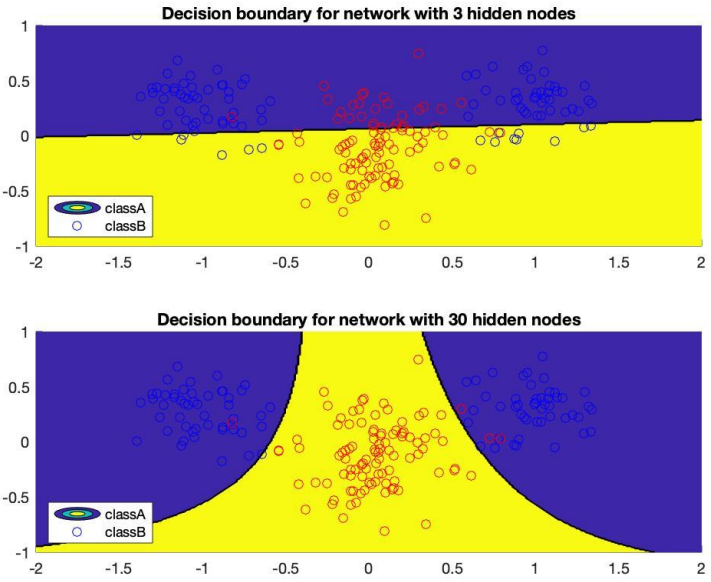


Figure - Decision boundary with *nhidden*=3 and *nhidden*=30

The figure below shows that with 3 hidden nodes, our model is very simple: the decision boundary is like a linear function. But as the number of hidden nodes increases, the decision boundary tends to convert to the non-linear transfer function we used. And as a result, we have a better network.

3.1.2 Function approximation

We have used our two-layer perceptron network to classify data, now, we want to approximate the well-known bell-shaped Gauss function. By analyzing the learning curve, we choose **epochs = 200** and $\eta = 0.001$.

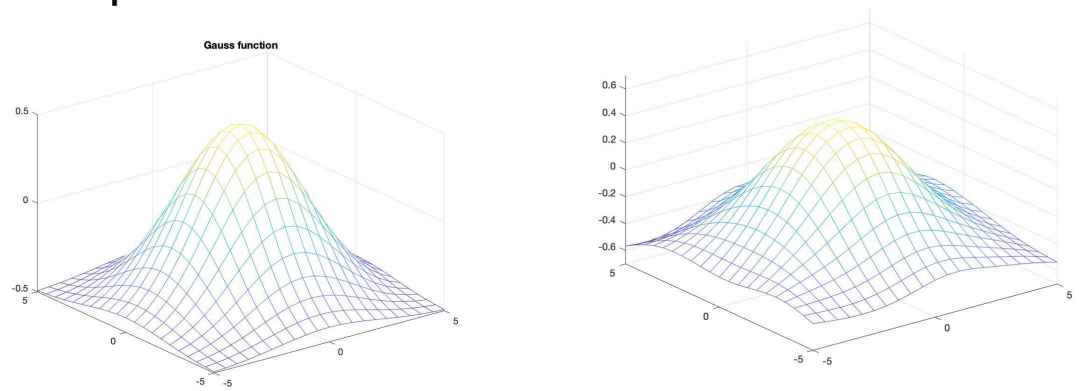


Figure - (L) Gauss Function - (R) Aproximated function by backprop network

If we compare the Gauss function with the function we approximated, we do not know how good (or how bad) our approximation is. Therefore, we are going to evaluate our network generalization.

We split our data into two subsets: 80% for training data and 20% for validation data.
By varying the number of nodes in the hidden layer (*nhidden* = 1, 3, 5, 10, 25), we approximate 10 times for each number of hidden nodes, and we plot the evolution of the mean of its MSE.

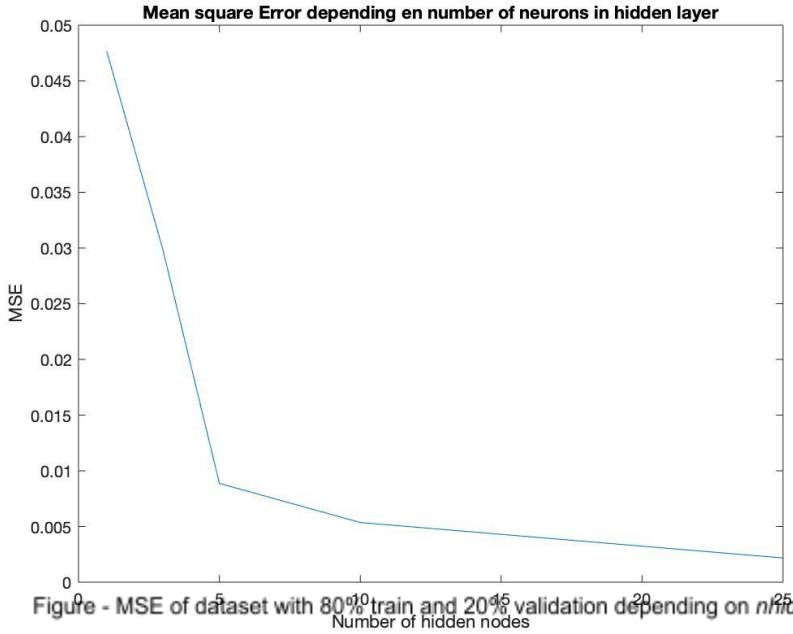


Figure - MSE of dataset with 80% train and 20% validation depending on *nhidden*

Apparently, the MSE of approximation keeps decreasing when we have more hidden nodes. When we have very few numbers of hidden nodes, the MSE is very high, we have an underfitting phenomenon. The speed of convergence is very quick, with a number of hidden nodes superior to 20, the MSE stuck on its local minimum. We also observe that the speed of approximation is very quick with *nhidden* > 20. But, we remark that we could potentially have an overfitting phenomenon.

Then, we decided to change the split of our dataset: 20% for training and 80% for validation.
Regarding the evolution of MSE, I choose the network with 20 hidden nodes.
With 20 hidden nodes, we trained 10 times with each dataset, and we got the mean of its results.

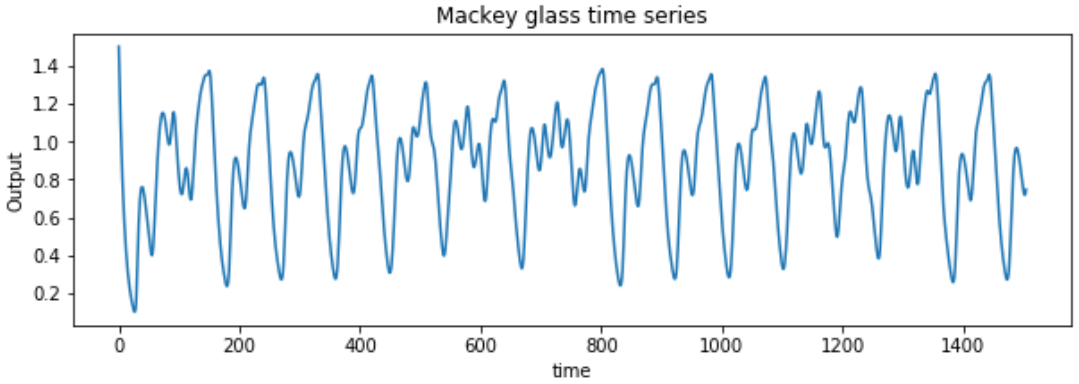
	Variance	Bias	MSE
80% Train VS 20% Valid	0.0459	0.0060	0.0020
20% Train VS 80% Valid	0.0480	0.0171	0.0077

According to the table, dataset 2 has an MSE and bias more important, which means the network with dataset 1 has a better approximation. A model trained with a smaller subset has a poor sensibility. Furthermore, I think which part of the input space training data situated could affect the generalization too because the correlation between samples is important.

4 Results and discussion - Part II

4.1 Three-layer perceptron for time series prediction - model selection, validation

The plot of Mackey glass time series



Model : MLPRegressor (framework : sklearn)
Settings:

- solver: lbfgs (The default solver 'adam' works pretty well on relatively large datasets in terms of both training time and validation score. For small datasets, however, 'lbfgs' can converge faster and perform better.)
- alpha : 0.001
- max_iter : 10000
- Validation fraction: 0.1
- number of simulations: 10
- early stop : True

Validation score (n1: number of nodes in hidden layer1, n2: number of nodes in hidden layer2)

n1/n2	2	5	15
2	0.430	0.802	0.814
4	0.571	**0.933**	**0.967**

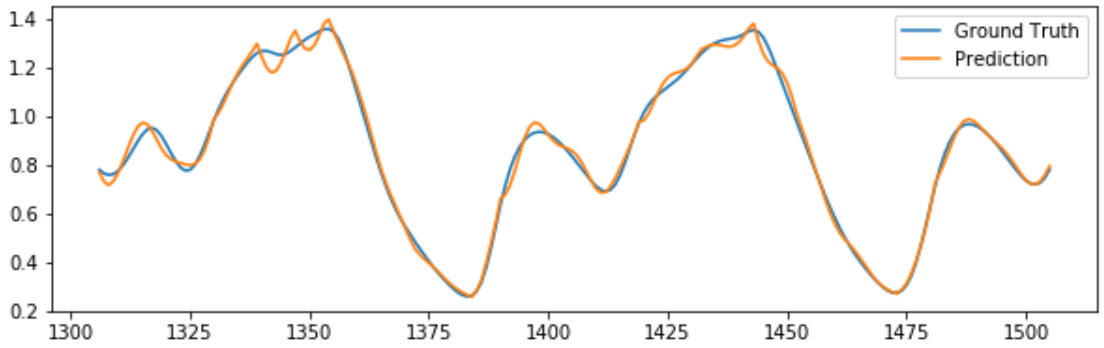
As expected, models with more nodes give better results as shown in the table.

MSE on the test set

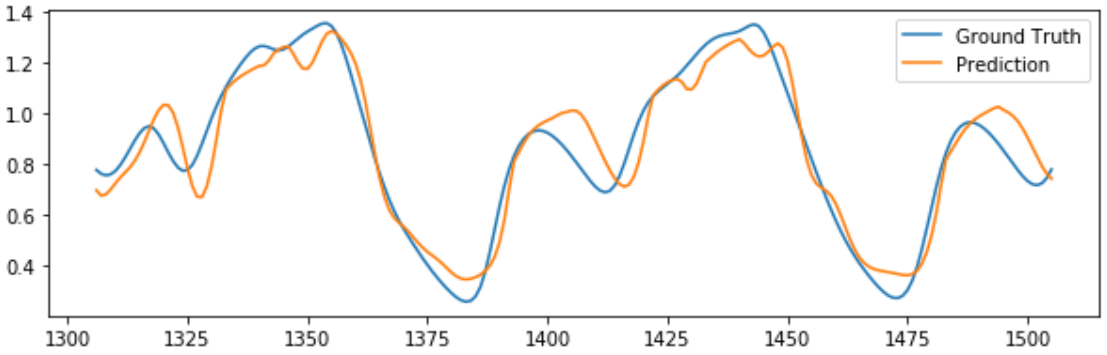
- Best model [4,15] : 0.0012
- Worst model [2,5] : 0.0093

Plot of prediction on test set:

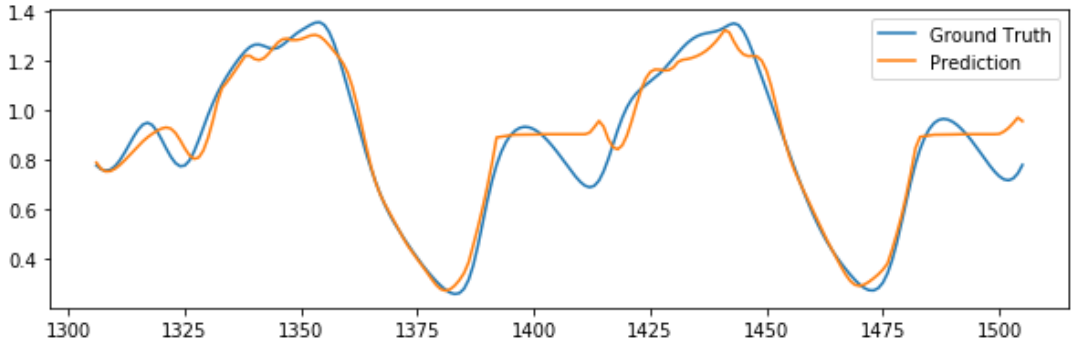
Best model:



Worst model:



Best model (solver: sgd)
MSE: 0.0052



4.2 Three-layer perceptron for noisy time series prediction with penalty regularisation

The effect of different numbers of nodes in h2 has on the performance testing on Val Score is the mean value obtained from 10 runs with s.d. value included in parenthesis (n2: number of nodes in hidden layer2)

Sigma=0.05

n2/alpha	10e-1	10e-2	10e-5
2	0.227 (0.01)	0.282 (0.23)	0.365 (0.22)
5	0.495 (0.02)	0.414 (0.19)	0.445 (0.15)
15	0.536 (0.01)	**0.545 (0.03)**	0.544 (0.02)

Sigma=0.3

n2/alpha	10e-1	10e-2	10e-5
2	0.213 (0.17)	0.141(0.11)	0.169 (0.14)
5	0.440 (0.10)	0.409 (0.05)	0.443 (0.11)
15	**0.500 (0.06)**	0.481(0.08)	0.465 (0.07)

Observation:

Each result is the average mean obtained from 10 runs. One thing to be noted here is that when n2 is small, the standard deviation is large, which suggests that when the number of nodes is small, the result heavily relies on the initialization of the model weights.
When the amount of noise is increased, all models perform worse compared with themselves with the same parameter settings. When the amount of noise is small, the effect of alpha does not seem to be significant when n2 is large. And the model performs worst when the amount of noise is big and n2 is small but with a large alpha value. In this case, the model becomes too simple to handle noisy data well. Overall, with noises introduced in the training set, models with larger alpha have better results. This is because the model has a better regularization capability; therefore performs better on noisy data.

5 Final remarks

Generally, this lab covers the fundamental notions of perceptron networks, we assimilate more profoundly all the theories that we have acquired during the course.

In the first part of the assignment, the programming of the backprop network allows us to learn how the steepest gradient works. We had the chance to understand better the algorithms and formulas that we had seen during the lecture. We compared their different variations according to their performance on the datasets.

In the second part of the lab, we got to learn how to search for the best model with the grid-search method. Along with the concepts we learn from the course, we were able to observe certain patterns and formulate some reasonings behind them. This ability is quite useful when we try to do in-depth research in the field of deep learning.