11-364 Presentation Grid search

Jineet Doshi

Brownlee 9

2/8/2017

Hyper-parameters to tune

- Weight initialization techniques

(Gaussian distribution, uniform, random)?

- Learning rate η

(slow learning vs overshooting convergence)

- Mini batch size M

(suboptimal updates to weights in Stochastic Gradient Descent vs CPU power)

- Regularization parameter λ

(curve smoothing vs overfitting)

- Number of neurons in a layer

(accuracy vs performance)

- Number of epochs

(accuracy vs CPU power, time)

- Dropout rate

(accuracy vs overfitting)

Parameter tuning is extremely important for ANY machine learning model!

Image courtesy: http://www.3hatscommunications.com/2016/choices-wordpress/

Grid Search to the rescue!

- Special package in scikit-learn library (GridSearchCV)
- Provides accuracy rates for every possible combination of the hyper parameter values provided.
- Implementation: Define the range of each hyper parameter in a python dictionary and pass it to a GridSearchCV model.
- Useful for all machine learning algorithms. Not just Deep Learning.
- Integrates well with Keras running on Theano or Tensorflow.

Experiment performed:

Pima Indians Diabetes Dataset: 768 samples

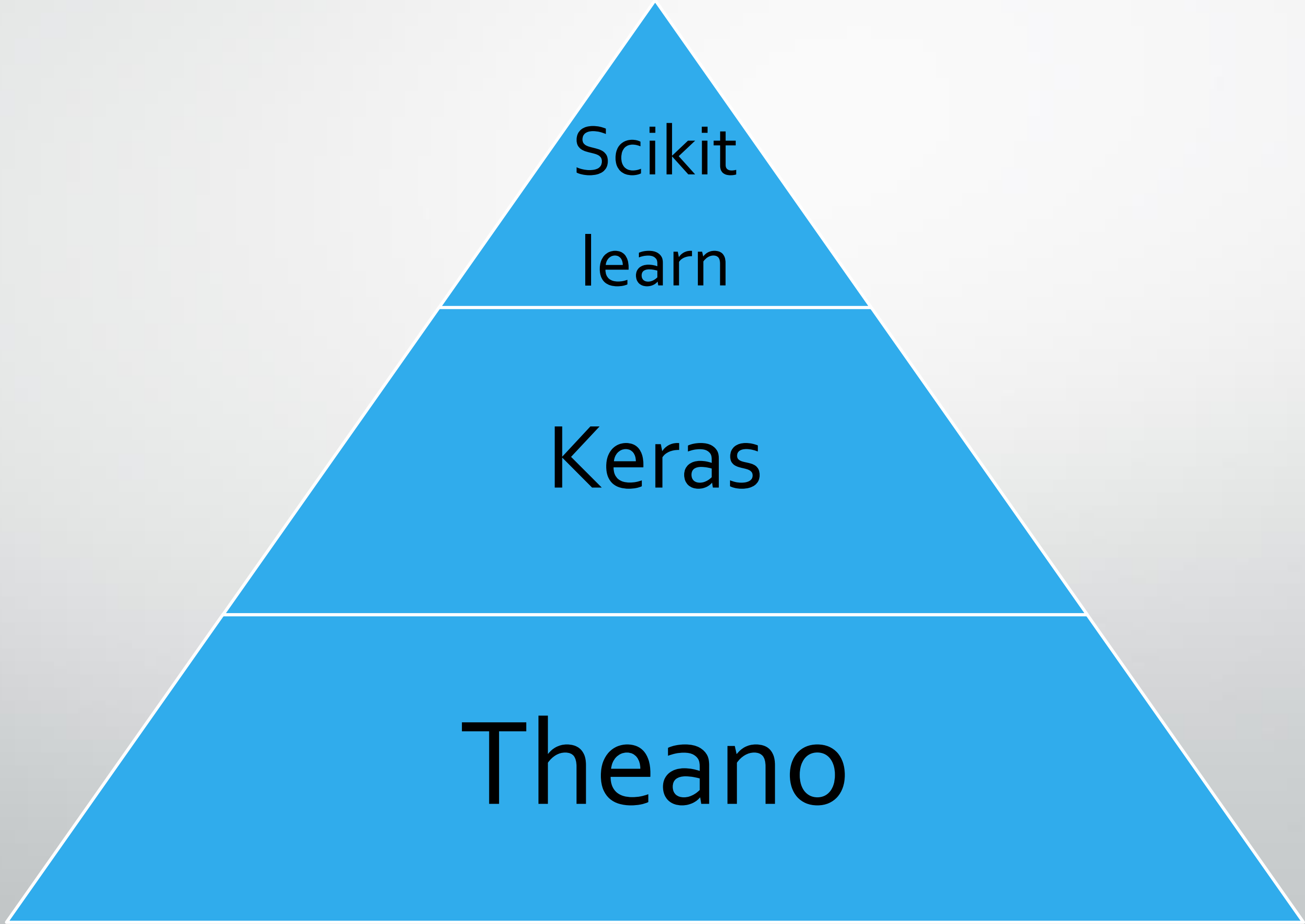
8 Features:

1. Number of times pregnant
2. Plasma glucose concentration a 2 hours in an oral glucose tolerance test
3. Diastolic blood pressure
4. Triceps skin fold thickness
5. 2-Hour serum insulin
6. Body mass index
7. Diabetes pedigree function
8. Age

<u>Task</u>: Classify whether diabetes present or not

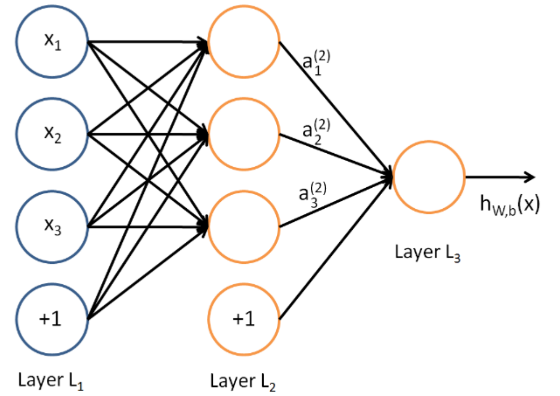
Experiment performed:

Stack:



Experiment performed:

Network (hidden layer with 8 neurons):



Code written for GridSearch:

```
optimizers = ['rmsprop', 'adam']
init = ['glorot_uniform', 'normal', 'uniform']
epochs = [50, 100, 150]
batches = [5, 10, 20]
param_grid = dict(optimizer=optimizers, nb_epoch=epochs, batch_size=batches, init=init)
grid = GridSearchCV(estimator=model, param_grid=param_grid)
grid_result = grid.fit(X, Y)
```

Experiment performed:

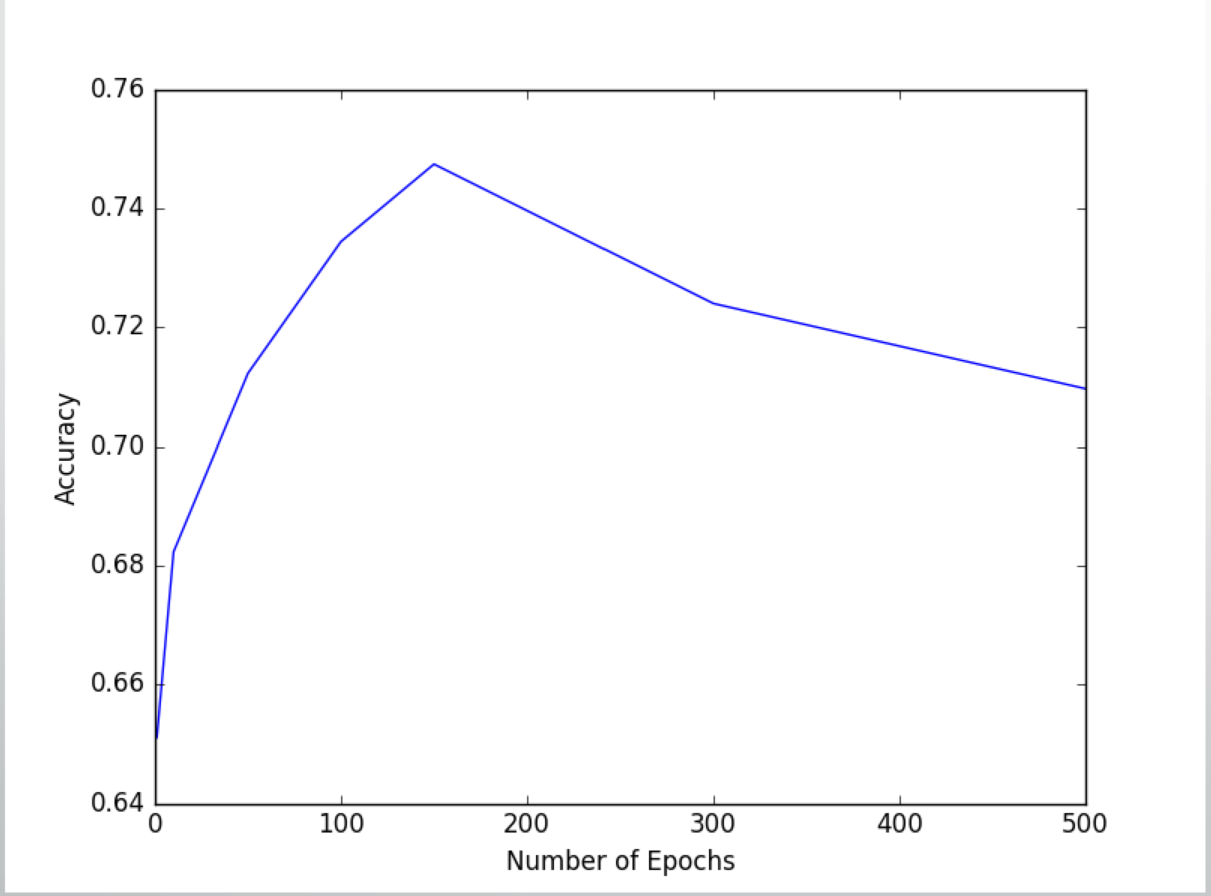
Results (screenshot from my terminal):

Best: 75% accuracy with weight initializations = Gaussian, Optimizer = rmsprop, Epochs = 150, Batch size = 5

```
Using Theano backend.
Best: 0.748698 using {'init': 'normal', 'optimizer': 'rmsprop', 'nb_epoch': 150, 'batch_size': 5}
0.661458 with: {'init': 'glorot_uniform', 'optimizer': 'rmsprop', 'nb_epoch': 50, 'batch_size': 5}
0.667969 with: {'init': 'glorot_uniform', 'optimizer': 'adam', 'nb_epoch': 50, 'batch_size': 5}
0.667969 with: {'init': 'glorot_uniform', 'optimizer': 'rmsprop', 'nb_epoch': 100, 'batch_size': 5}
0.710938 with: {'init': 'glorot_uniform', 'optimizer': 'adam', 'nb_epoch': 100, 'batch_size': 5}
0.697917 with: {'init': 'glorot_uniform', 'optimizer': 'rmsprop', 'nb_epoch': 150, 'batch_size': 5}
0.730469 with: {'init': 'glorot_uniform', 'optimizer': 'adam', 'nb_epoch': 150, 'batch_size': 5}
0.707031 with: {'init': 'normal', 'optimizer': 'rmsprop', 'nb_epoch': 50, 'batch_size': 5}
0.718750 with: {'init': 'normal', 'optimizer': 'adam', 'nb_epoch': 50, 'batch_size': 5}
0.716146 with: {'init': 'normal', 'optimizer': 'rmsprop', 'nb_epoch': 100, 'batch_size': 5}
0.725260 with: {'init': 'normal', 'optimizer': 'adam', 'nb_epoch': 100, 'batch_size': 5}
0.748698 with: {'init': 'normal', 'optimizer': 'rmsprop', 'nb_epoch': 150, 'batch_size': 5}
0.710938 with: {'init': 'normal', 'optimizer': 'adam', 'nb_epoch': 150, 'batch_size': 5}
0.667969 with: {'init': 'uniform', 'optimizer': 'rmsprop', 'nb_epoch': 50, 'batch_size': 5}
0.703125 with: {'init': 'uniform', 'optimizer': 'adam', 'nb_epoch': 50, 'batch_size': 5}
0.716146 with: {'init': 'uniform', 'optimizer': 'rmsprop', 'nb_epoch': 100, 'batch_size': 5}
0.710938 with: {'init': 'uniform', 'optimizer': 'adam', 'nb_epoch': 100, 'batch_size': 5}
0.747396 with: {'init': 'uniform', 'optimizer': 'rmsprop', 'nb_epoch': 150, 'batch_size': 5}
0.746094 with: {'init': 'uniform', 'optimizer': 'adam', 'nb_epoch': 150, 'batch_size': 5}
0.644531 with: {'init': 'glorot_uniform', 'optimizer': 'rmsprop', 'nb_epoch': 50, 'batch_size': 10}
0.680990 with: {'init': 'glorot_uniform', 'optimizer': 'adam', 'nb_epoch': 50, 'batch_size': 10}
0.695313 with: {'init': 'glorot_uniform', 'optimizer': 'rmsprop', 'nb_epoch': 100, 'batch_size': 10}
0.667969 with: {'init': 'glorot_uniform', 'optimizer': 'adam', 'nb_epoch': 100, 'batch_size': 10}
0.566406 with: {'init': 'glorot_uniform', 'optimizer': 'rmsprop', 'nb_epoch': 150, 'batch_size': 10}
0.660156 with: {'init': 'glorot_uniform', 'optimizer': 'adam', 'nb_epoch': 150, 'batch_size': 10}
0.677083 with: {'init': 'normal', 'optimizer': 'rmsprop', 'nb_epoch': 50, 'batch_size': 10}
0.701823 with: {'init': 'normal', 'optimizer': 'adam', 'nb_epoch': 50, 'batch_size': 10}
0.696615 with: {'init': 'normal', 'optimizer': 'rmsprop', 'nb_epoch': 100, 'batch_size': 10}
0.731771 with: {'init': 'normal', 'optimizer': 'adam', 'nb_epoch': 100, 'batch_size': 10}
0.743490 with: {'init': 'normal', 'optimizer': 'rmsprop', 'nb_epoch': 150, 'batch_size': 10}
0.734375 with: {'init': 'normal', 'optimizer': 'adam', 'nb_epoch': 150, 'batch_size': 10}
0.718750 with: {'init': 'uniform', 'optimizer': 'rmsprop', 'nb_epoch': 50, 'batch_size': 10}
0.717448 with: {'init': 'uniform', 'optimizer': 'adam', 'nb_epoch': 50, 'batch_size': 10}
0.717448 with: {'init': 'uniform', 'optimizer': 'rmsprop', 'nb_epoch': 100, 'batch_size': 10}
0.704427 with: {'init': 'uniform', 'optimizer': 'adam', 'nb_epoch': 100, 'batch_size': 10}
0.740885 with: {'init': 'uniform', 'optimizer': 'rmsprop', 'nb_epoch': 150, 'batch_size': 10}
0.708333 with: {'init': 'uniform', 'optimizer': 'adam', 'nb_epoch': 150, 'batch_size': 10}
0.680990 with: {'init': 'glorot_uniform', 'optimizer': 'rmsprop', 'nb_epoch': 50, 'batch_size': 20}
0.686198 with: {'init': 'glorot_uniform', 'optimizer': 'adam', 'nb_epoch': 50, 'batch_size': 20}
0.688802 with: {'init': 'glorot_uniform', 'optimizer': 'rmsprop', 'nb_epoch': 100, 'batch_size': 20}
0.691406 with: {'init': 'glorot_uniform', 'optimizer': 'adam', 'nb_epoch': 100, 'batch_size': 20}
0.670573 with: {'init': 'glorot_uniform', 'optimizer': 'rmsprop', 'nb_epoch': 150, 'batch_size': 20}
0.700521 with: {'init': 'glorot_uniform', 'optimizer': 'adam', 'nb_epoch': 150, 'batch_size': 20}
0.696615 with: {'init': 'normal', 'optimizer': 'rmsprop', 'nb_epoch': 50, 'batch_size': 20}
0.712240 with: {'init': 'normal', 'optimizer': 'adam', 'nb_epoch': 50, 'batch_size': 20}
0.703125 with: {'init': 'normal', 'optimizer': 'rmsprop', 'nb_epoch': 100, 'batch_size': 20}
0.718750 with: {'init': 'normal', 'optimizer': 'adam', 'nb_epoch': 100, 'batch_size': 20}
0.718750 with: {'init': 'normal', 'optimizer': 'rmsprop', 'nb_epoch': 150, 'batch_size': 20}
0.740885 with: {'init': 'normal', 'optimizer': 'adam', 'nb_epoch': 150, 'batch_size': 20}
```

Experiment performed:

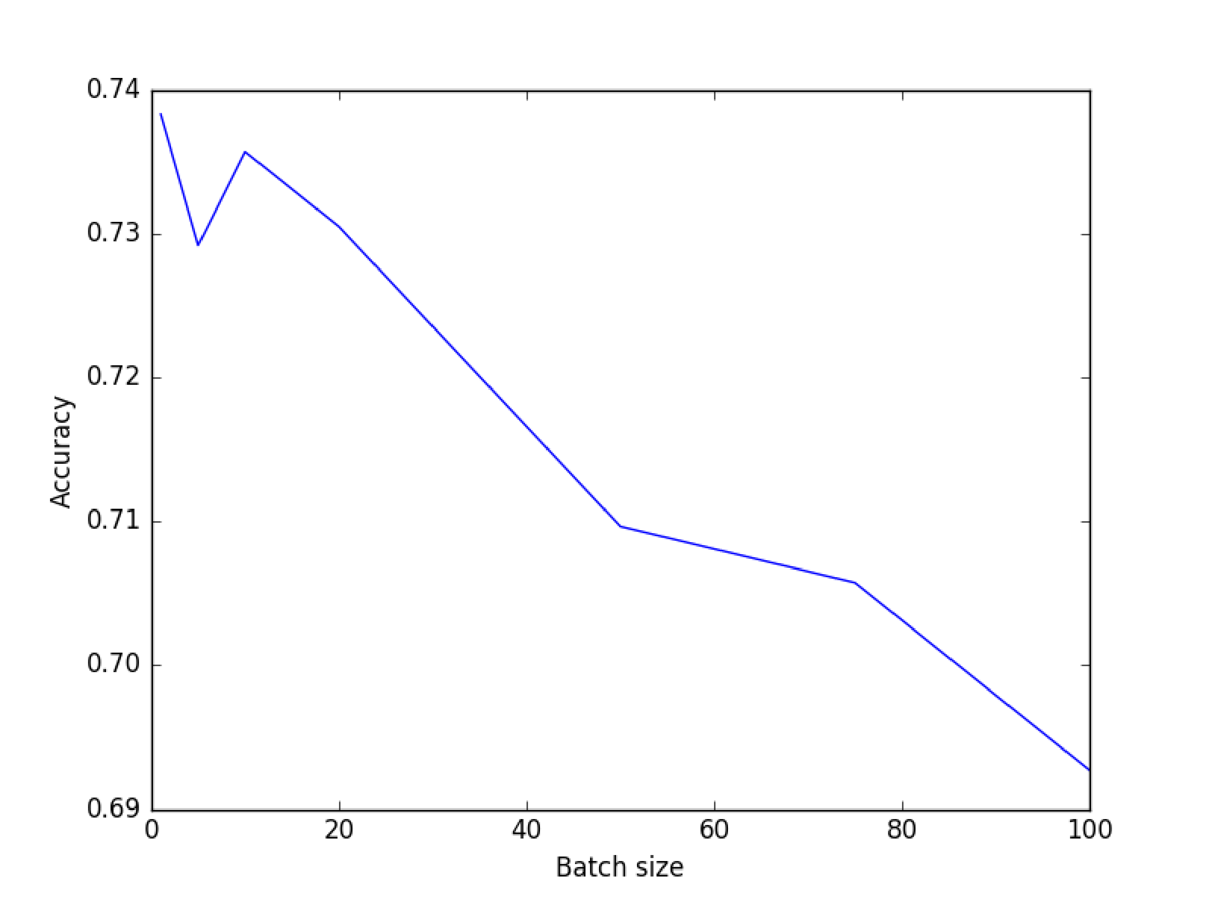
Effect of number of epochs on accuracy



Plot using Matplotlib

Experiment performed:

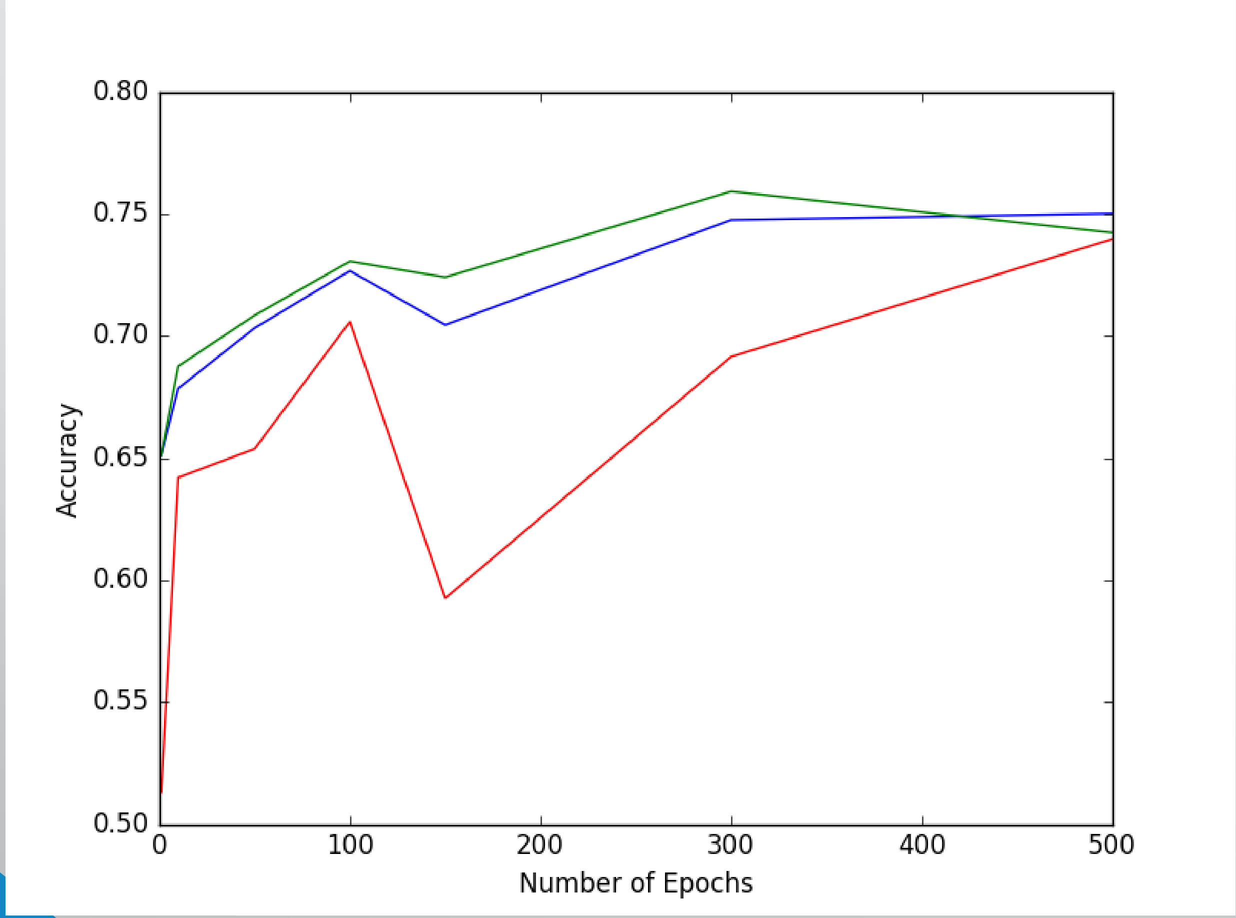
Effect of mini batch size on accuracy



Plot using Matplotlib

Experiment performed:

Effect of weight initializations on accuracy



Plot using Matplotlib

Red – Glorot uniform
Blue – Normal
Green - Uniform

Warning!

For every combination of the hyper parameter values, Grid Search generates a new model and tests accuracy using 3 fold cross validation. Hence, if training data is large or number of parameters to tune are more, this can be extremely time consuming!
Make sure you have enough computing power (read: AWS instances or a supercomputer) before running Grid Search on large data or with more parameters.

Current research in Hyper parameter tuning

- Surrogate based Optimization

http://www.jmlr.org/proceedings/papers/v28/bardenet13.pdf

- Gradient based Reversible learning

https://arxiv.org/pdf/1502.03492.pdf

Takeaways

- Hyper parameter tuning is very important for success of the model
- Lot of options and limited computing power/time. GridSearch is great for datasets with less features and when less number of hyper-parameters are to be tuned. For more complex datasets or networks, more computing power is needed.
- Area of active research. Lot of new ideas being proposed.