A combination of Genetic Algorithms and Neural Networks

Made By:-
Pulkit pandey(190020088)

Acknowledgement
We would like to express our sincere gratitude towards our Professor for the course Genetic Algorithms Dr. Nirupam Chakraborti for his constant support and guidance throughout the semester and our Project as well.

Abstract
Neural Networks and Genetic Algorithms are two very different techniques to conquer a specific problem of optimization and Learning. Both of them have their own weaknesses and strengths. Both of them have evolved in an entirely different but planned manner. In recent times, one can clearly notice the combination of the two to get some exceptionally good results. Tech giants such as Amazon include a combination of both in their product ALEXA to get exceptionally good results which cannot be achieved by either of the two.

Introduction
Neural Networks as well as Genetic Algorithms could be viewed as Machine Learning Techniques as both of them could be used for optimization and learning purposes. The core idea to development of both is as described below:-

1) **Genetic Algorithm** - In computer science and operations research, a genetic algorithm (GA) is a metaheuristic inspired by the process of natural selection that belongs to the larger class of evolutionary algorithms (EA). Genetic algorithms are commonly used to generate high-quality solutions to optimization and search problems by relying on bio-inspired operators such as mutation, crossover and selection.
2) **Neural Networks** - A neural network is a network or circuit of neurons, or in a modern sense, an artificial neural network, composed of artificial neurons or nodes. Thus a neural network is either a biological neural network, made up of real biological neurons, or an artificial neural network, for solving artificial intelligence (AI) problems.

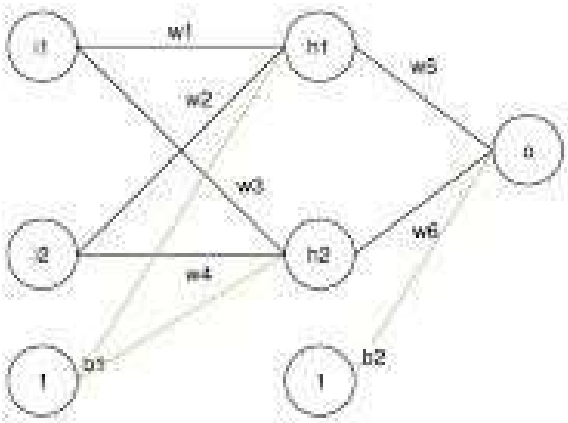
One can clearly visualise that both of them were developed keeping in mind the existing natural processes or natural organs.

In this term project, we will describe a genetic algorithm which could be used for the purpose of training Neural Networks. How? - Genetic Algorithms could be used to learn the best hyperparameters for a Neural Network. This makes it an immensely powerful tool as training a Neural Network is a tedious task. Achieving the trained model in a lesser time span is definitely going to be an achievement.

What is a Neural Network?
Neural Networks are parallel computing devices, which were developed as an attempt to develop a computerised model of the brain. One can conclude the objective of the same is to be able to perform the tedious computational task at an accelerated pace. The tasks mentioned include a variety of tasks such as Classification problems, Regression Problems, Speech Recognition, Pattern Recognition, Optimization and Clustering.

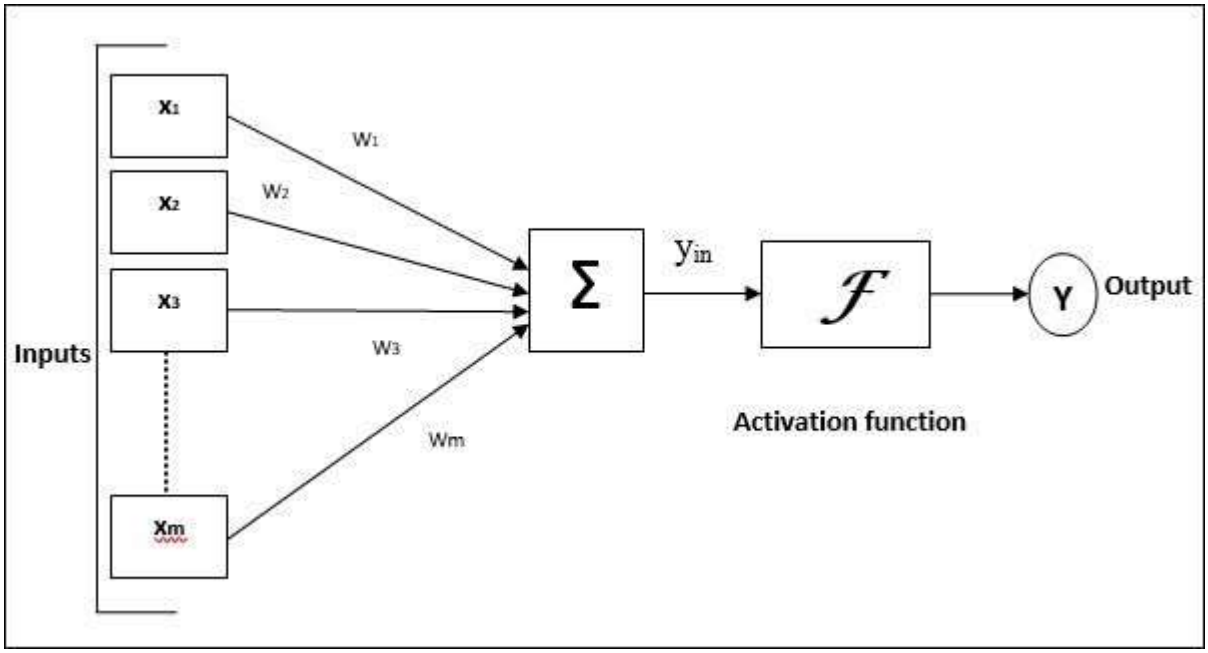
The fact cannot be denied that the Neural Networks are far away from the dimensions and performance of the Human Brain. On an average, a human brain consists of 10 to 15 billion neurons. However, Artificial Neural Networks contain rarely more than a thousand of the same.

Let us describe a simple Neural Network to get ourselves acquainted to the methodologies used for describing the same.



The first layer in the above diagram is called the “Input Layer”. This layer consists of the input feature vector. The middle layer is called the “Hidden Layer” followed by the output in the Last Layer. The numbers (w1, w2 etc) next to each connection is called weights which indicates the strength of a connection. The arrangement of neurons and the connection collectively is called the “Network Architecture” or its “Topology”.

From a mathematical point of view, the neural network is nothing but a function. This is a function which maps an input to an output.



General Model of Artificial Neural Network

From the diagram mentioned above, one can get an insight of what actually a Neural Network is and how it performs.

Let's assume x1, x2, x3 … xm to be input feature vectors and Y be the output.

Yin = x1w1 + x2w2 + x3w3 + …… + xmwm

The output can be calculated in the following manner

Y = F(Yin), where F is the activation function.

There are a variety of activation functions in use such as: sigmoid, gaussian, ReLU, hyperbolic tangent etc. These can be used in accordance with the problem one is dealing with.

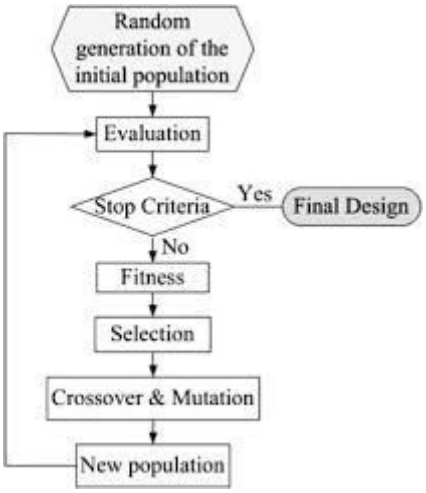
Here w's are the weights which is what makes the neural network learn things. Various optimization techniques such as Gradient Descent, Stochastic gradient Descent etc. have been devised to achieve the optimal weights for a particular optimization problem. We will describe these techniques in detail as we progress.

Genetic Algorithms

A genetic algorithm is a search heuristic that is inspired by Charles Darwin's theory of natural evolution. This algorithm reflects the process of natural selection where the fittest individuals are selected for reproduction in order to produce offspring of the next generation. These algorithms try to simulate the natural evolution processes with the purpose to optimize a set of parameters.

Natural selection starts with the selection of the fittest individual from a population. Offsprings are produced which inherit the characteristics of their parents and added to the upcoming generation. Considering the parents have better fitness, their offspring would therefore have a better chance of survival. This process keeps on iterating itself and at the end, a generation with the fittest individual will be found.

Original idea, as proposed by John Holland, was to encode the genetic information in a bit string of fixed length, popularly known as an individual. Basic Genetic Algorithm operators are the crossover, selection and mutation.



The above diagram illustrates the principle structure of a genetic algorithm. It starts with a random set of individuals, the initial population.

The processes which generate new individuals are: mutation and crossover. Let us describe these processes (including selection) in detail to get an insight of what these actually are.

Selection:

The intuitive idea behind the selection phase is to select the fittest individuals and let them pass their genes to the next generation. Two pairs of individuals are selected on the basis of their fitness scores. Individuals having a higher fitness is more likely to get selected. In the classical fitness-based roulette-wheel, the chance of an individual to be selected is based on its relative fitness in the population.

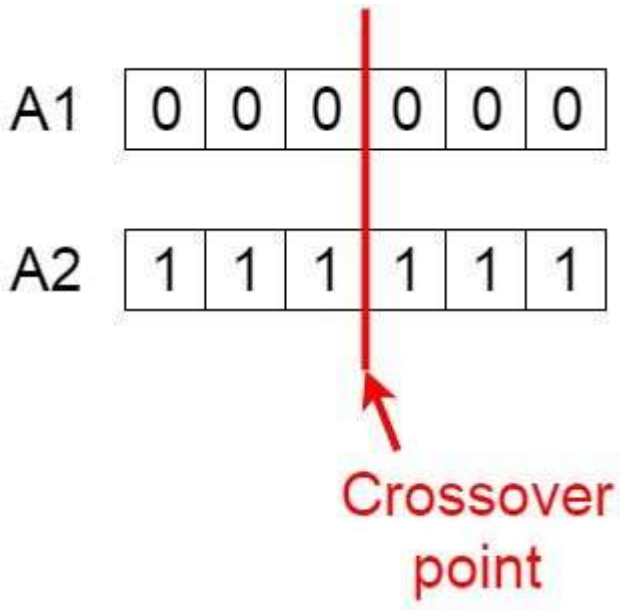
Chromosome	Fitness Value
A	8.2
B	3.2
C	1.4
D	1.2
E	4.2
F	0.3

As clearly visible, the chances of selection increase if one has a higher fitness level as it corresponds to a higher proportion of the wheel.

Crossover:

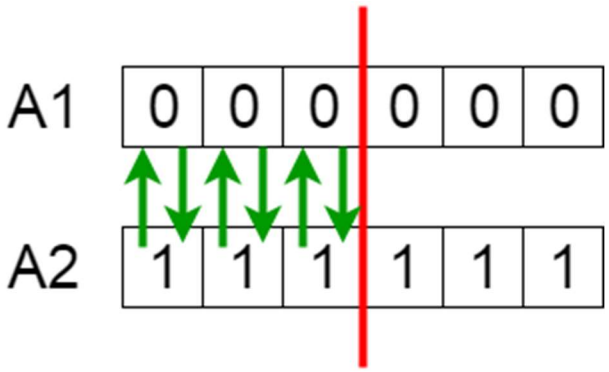
One may consider this as the most significant phase of a genetic algorithm For each pair of parents to be mated, a crossover point is chosen at random from within the genes.

Let's consider the following figure:

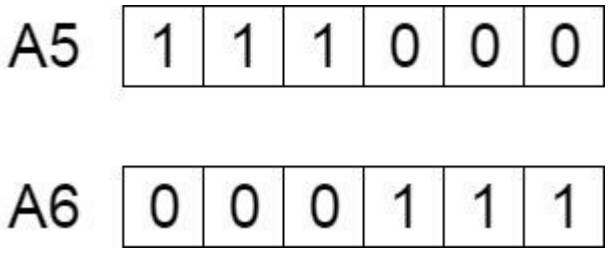


Choosing 3rd as a random crossover point.

Offsprings are created by exchanging the genes of parents among themselves until the crossover point is reached.

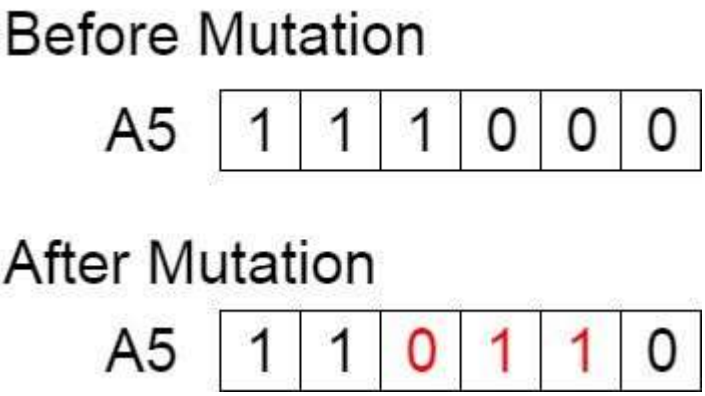


The new Offsprings will now be added to the population.



Mutation:

In a certain formed offspring, some of their genes can be subjected to a mutation with a low random probability. This implies that some of the bits in the bit string can be flipped.



Mutation occurs to maintain diversity within the population to prevent premature convergence.

Pseudocode for Genetic Algorithm

START

Generate the initial population

Compute fitness

REPEAT

Selection

Crossover

Mutation

Compute fitness

UNTIL population has converged

STOP

Genetic Algorithm application to train Neural Network in Python

```
from sklearn import datasets import numpy as np
import matplotlib.pyplot as plt from
sklearn.neural_network import MLPClassifier from
sklearn.metrics import accuracy_score from
sklearn.model_selection import train_test_split from
random import randint import random from
sklearn.metrics import mean_absolute_error as mae
iris = datasets.load_iris() X = iris.data y =
iris.target
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.33,
random_state=42)
 def initialize_population
(size_mlp):
  pop =  [[]]*size_mlp
  activation = ['identity','logistic', 'tanh', 'relu']
solver = ['lbfgs','sgd', 'adam']   pop =
[[random.choice(activation), random.choice(solver),
randint(2,100), randint(2,100)] for i in range(0, size_mlp)]
return pop
```

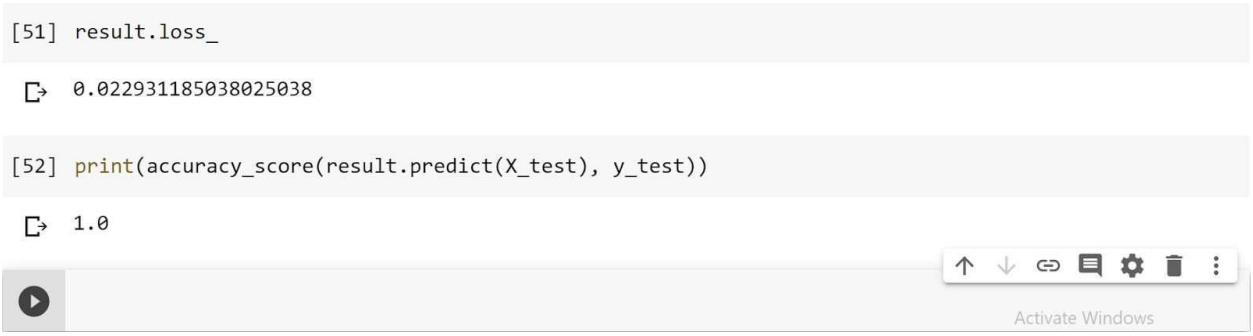
```
def crossover_mlp(mother_1, mother_2):
  child = [mother_1[0], mother_2[1], mother_1[2], mother_2[3]]
```

```
  return child
 def mutation_mlp(child,
prob_mut):
  for c in range(0, len(child)):
if np.random.rand() > prob_mut:
      k = randint(2,3)       child[c][k] =
int(child[c][k]) + randint(1, 10)   return child
 def function_fitness_mlp(pop, X_train, y_train, X_test,
y_test):
  fitness = []
j = 0
  for w in pop:
    clf = MLPClassifier(learning_rate_init=0.09, activation=w[0], solver
= w[1], alpha=1e-5, hidden_layer_sizes=(int(w[2]), int(w[3])), max_iter=
1000, n_iter_no_change=80)     try:
      clf.fit(X_train, y_train)       f =
accuracy_score(clf.predict(X_test), y_test)
fitness.append([f, clf, w])     except:
      pass
  return fitness
 def ag_mlp(X_train, y_train, X_test, y_test, num_epochs = 10, size_mlp=
10, prob_mut=0.8):
  pop = initialize_population(size_mlp)   fitness =
function_fitness_mlp(pop,  X_train, y_train, X_test, y_test)
pop_fitness_sort = np.array(list(reversed(sorted(fitness,key=lambda x:
x[0]))))
  for j in range(0, num_epochs):     length = len(pop_fitness_sort)
parent_1 = pop_fitness_sort[:,2][:length//2]     parent_2 =
pop_fitness_sort[:,2][length//2:]     child_1 =
[crossover_mlp(parent_1[i], parent_2[i]) for i in range(    0,
np.min([len(parent_2), len(parent_1)]))]     child_2 =
[crossover_mlp(parent_2[i], parent_1[i]) for i in range(    0,
np.min([len(parent_2), len(parent_1)]))]     child_2 =
mutation_mlp(child_2, prob_mut)
```

```
fitness_child_1 = function_fitness_mlp(child_1,X_train, y_train, X_test,
y_test)
    fitness_child_2 = function_fitness_mlp(child_2, X_train, y_train,
X_test, y_test)     pop_fitness_sort = np.concatenate((pop_fitness_sort,
fitness_child_1, fitness_child_2))     sort =
np.array(list(reversed(sorted(pop_fitness_sort,key=lambda x:
x[0])))))     pop_fitness_sort = sort[0
:size_mlp, :]     best_individual = sort[
0][1]
  return best_individual
 result = ag_mlp(X_train, y_train, X_test, y_test, num_epochs =
10, size_mlp=20, prob_mut=0.9)
 print (accuracy_score(result.predict(X_test),
y_test))
```

We get an accuracy score of 1.0 when using these basic operations of the Genetic Algorithms to train the model.

So, we can easily see the same being performed on in-built sci-kitlearn dataset to obtain an accuracy score of 1.0 with the loss value converging which shows that genetic algorithms, when combined with neural networks can help us achieve some very good results.



We also compared the obtained results from the basic GA operations on Neural Networks with the normal neural network procedures. **The code and results for the same are undermentioned:**

```
print(__doc__)
 import
warnings
 import matplotlib.pyplot as
plt
 from sklearn.neural_network import MLPClassifier
from sklearn.preprocessing import MinMaxScaler
from sklearn import datasets from
sklearn.exceptions import ConvergenceWarning

# different learning rate schedules and momentum parameters params = [{
'solver': 'sgd', 'learning_rate': 'constant', 'momentum': 0,
'learning_rate_init': 0.2},
         {'solver': 'sgd', 'learning_rate': 'constant', 'momentum': .9,
'nesterovs_momentum': False, 'learning_rate_init': 0.2},
         {'solver': 'sgd', 'learning_rate': 'constant', 'momentum': .9,
          'nesterovs_momentum': True, 'learning_rate_init': 0.2},
         {'solver': 'sgd', 'learning_rate': 'invscaling', 'momentum': 0,
          'learning_rate_init': 0.2},
         {'solver': 'sgd', 'learning_rate': 'invscaling', 'momentum': .9,
'nesterovs_momentum': True, 'learning_rate_init': 0.2},
         {'solver': 'sgd', 'learning_rate': 'invscaling', 'momentum': .9,
          'nesterovs_momentum': False, 'learning_rate_init': 0.2},
         {'solver': 'adam', 'learning_rate_init': 0.01}]
 labels = ["constant learning-rate", "constant with momentum"
,
         "constant with Nesterov's momentum",
         "inv-scaling learning-rate", "inv-scaling with momentum",
         "inv-scaling with Nesterov's momentum", "adam"]
```

```
plot_args = [{'c': 'red', 'linestyle': '-'},
```

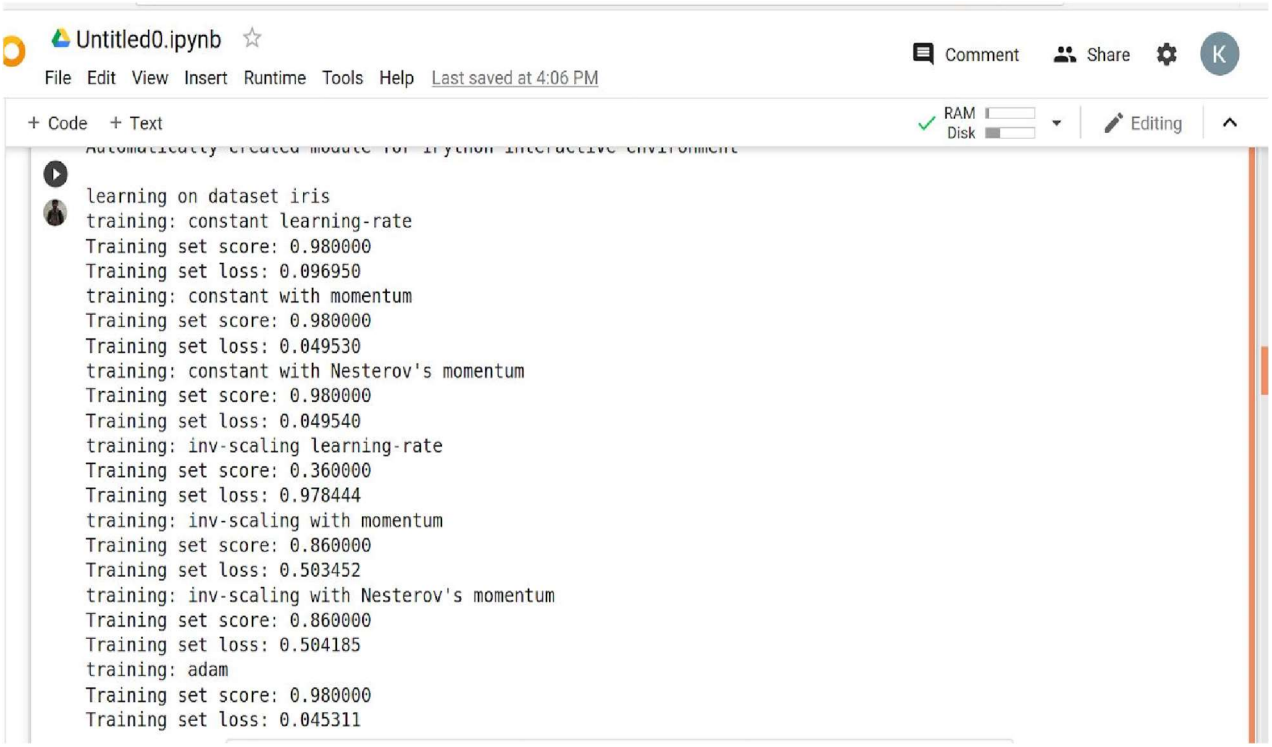
```
{'c': 'green', 'linestyle': '-'},
            {'c': 'blue', 'linestyle': '-'},
            {'c': 'red', 'linestyle': '--'},
            {'c': 'green', 'linestyle': '--'},
            {'c': 'blue', 'linestyle': '--'},
{'c': 'black', 'linestyle': '-'}]
  def plot_on_dataset(X, y, ax,
name):
   # for each dataset, plot learning for each learning
strategy    print("\nlearning on dataset %s" % name)
ax.set_title(name)

   X = MinMaxScaler().fit_transform(X)
mlps = []
   if name == "digits":
       # digits is larger but converges fairly
quickly        max_iter = 15    else:
       max_iter = 400

   for label, param in zip(labels, params):
print("training: %s" % label)        mlp =
MLPClassifier(random_state=0,
max_iter=max_iter, **param)

       # some parameter combinations will not converge as can be seen on
the
       # plots so they are ignored here        with
warnings.catch_warnings():            warnings.filterwarnings("ignore",
category=ConvergenceWarning,                                    module=
"sklearn")            mlp.fit(X, y)
```

```
mlps.append(mlp)        print("Training set score: %f" %
mlp.score(X, y))        print("Training set loss: %f" %
mlp.loss_)
   for mlp, label, args in zip(mlps, labels, plot_args):
       ax.plot(mlp.loss_curve_, label=label, **args)
  fig, axes = plt.subplots(2, 2, figsize=(15,
10))
# load / generate some toy datasets
iris = datasets.load_iris()
X_digits, y_digits = datasets.load_digits(return_X_y=True)
data_sets = [(iris.data, iris.target),
(X_digits, y_digits),
            datasets.make_circles(noise=0.2, factor=0.5, random_state=1),
datasets.make_moons(noise=0.3, random_state=0)]
 for ax, data, name in zip(axes.ravel(), data_sets, ['iris'
]):
   plot_on_dataset(*data, ax=ax, name=name)
 fig.legend(ax.get_lines(), labels, ncol=3, loc="upper center"
) plt.show()
```



We had the final loss value of our model:- 0.023 whereas the normal Implementation of Neural Network outputs 0.045 when using Adam optimization technique for convergence.

Sweet!

Looks like the method converged and provided some very good results when combined with genetic algorithms with a fair accuracy score and a final loss value. Also, this was implemented on the inbuilt iris dataset owing to the fact that we wanted to have an intuitive understanding of the working of the same.

References

1. Links:- https://towardsdatascience.com/gas-and-nns-6a41f1e8146 d

2. Research Papers:-
   Training Feedforward Neural Networks Using Genetic Algorithms by David J. Montana

   Combining Genetic Algorithms and Neural Networks: The Encoding Problem A Thesis Presented for the Master of Science Degree The University of Tennessee, Knoxville by: Philipp Koehn