**Ramaiah Institute of Technology**

(An Autonomous Institute, Affiliated to VTU)

MSR Nagar, MSRIT post, Bangalore-54

A Dissertation Report on

GENETIC ALGORITHM

by

NIHARIKA.C 1MS15CS077

PAVITRA.K.R 1MS15CS086

KAVERI.B.W 1MS16CS406

ASHA.D.N 1MS16CS400

# *Bachelor of Engineering in Computer Science & Engineering*

Under the guidance of

SOUMYA.B.J

Asst. Professor

CSE Department

RIT



**DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING**

**M.S.RAMAIAH INSTITUTE OF TECHNOLOGY**

**(Autonomous Institute, Affiliated to VTU)**

**BANGALORE-560054**

[www.msrit.edu](http://www.msrit.edu), **2017**

**Times New Roman**

# INTRODUCTION

In [computer science](https://en.wikipedia.org/wiki/Computer_science) and [operations research](https://en.wikipedia.org/wiki/Operations_research), a **genetic algorithm** (**GA**) is

a [metaheuristic](https://en.wikipedia.org/wiki/Metaheuristic) inspired by the process of [natural selection](https://en.wikipedia.org/wiki/Natural_selection) that belongs to the larger class

of [evolutionary algorithms](https://en.wikipedia.org/wiki/Evolutionary_algorithm)(EA). Genetic algorithms are commonly used to generate high-

quality solutions to [optimization](https://en.wikipedia.org/wiki/Optimization_(mathematics)) and [search problems](https://en.wikipedia.org/wiki/Search_algorithm) by relying on bio-inspired operators

such as [mutation](https://en.wikipedia.org/wiki/Mutation_(genetic_algorithm)), [crossover](https://en.wikipedia.org/wiki/Crossover_(genetic_algorithm)) and [selection](https://en.wikipedia.org/wiki/Selection_(genetic_algorithm)).

A genetic algorithm requires:

1. A [genetic representation](https://en.wikipedia.org/wiki/Genetic_representation) of the solution domain,

2. A [fitness function](https://en.wikipedia.org/wiki/Fitness_function) to evaluate the solution domain.

Here, we optimize NEURAL NETWORKS through GENETIC ALGORITHM.

# Neutral Networks

An Artificial Neural Network (ANN) is a computational model that is inspired by the way

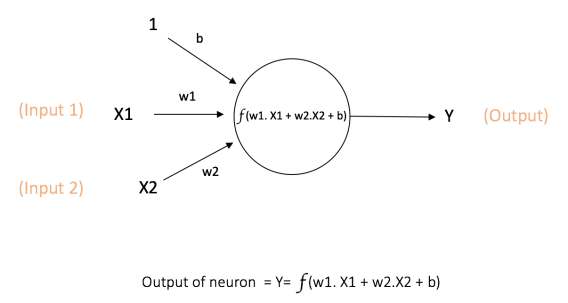
Biological neural networks in the human brain process information. Artificial Neural

Networks have generated a lot of excitement in Machine Learning research and industry,

thanks to many breakthrough results in speech recognition, computer vision and text

processing. In this blog post we will try to develop an understanding of a particular type of

Artificial Neural Network called the Multi Layer Perceptron.



The above network takes numerical inputs **X1** and **X2** and has weights **w1** and **w2** associated

with those inputs. Additionally, there is another input **1** with weight **b** (called the **Bias**)

associated with it.

The output **Y** from the neuron is computed as shown in the Figure 1. The function ***f***is non-linear

and is called the **Activation Function**. The purpose of the activation function is to introduce

non-linearity into the output of a neuron. This is important because most real world data is non

linear and we want neurons to learn these non linear representations.

There are several activation functions you may encounter in practice:

* **Sigmoid:**takes a real-valued input and squashes it to range between 0 and 1

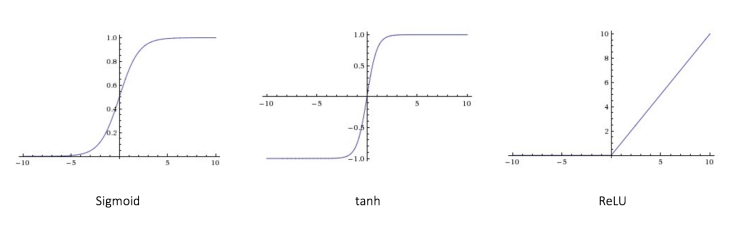
σ(x) = 1 / (1 + exp(−x))

* **tanh:** takes a real-valued input and squashes it to the range [-1, 1]

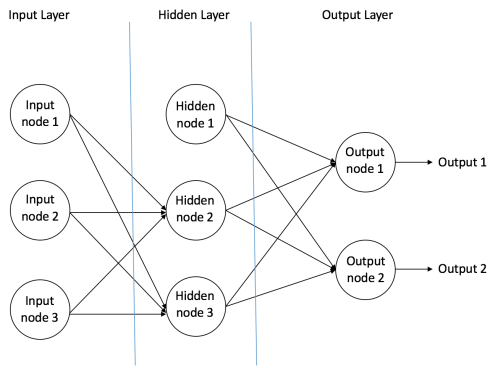
tanh(x) = 2σ(2x) − 1

* **ReLU**: ReLU stands for Rectified Linear Unit. It takes a real-valued input and thresholds it at zero (replaces negative values with zero)

F(x) = max (0, x)



#### **Feedforward Neural Network:**



1. **Input Nodes –** The Input nodes provide information from the outside world to the network and are together referred to as the “Input Layer”. No computation is performed in any of the Input nodes – they just pass on the information to the hidden nodes.
2. **Hidden Nodes –**The Hidden nodes have no direct connection with the outside world (hence the name “hidden”). They perform computations and transfer information from the input nodes to the output nodes. A collection of hidden nodes forms a “Hidden Layer”. While a feedforward network will only have a single input layer and a single output layer, it can have zero or multiple Hidden Layers.
3. **Output Nodes –**The Output nodes are collectively referred to as the “Output Layer” and are responsible for computations and transferring information from the network to the outside world.

# DATASET

Frameworks- Keras and tensor flow are used to directly download data from online.

Size of the dataset(in bytes):

C:\Users\Niharika\Desktop\data.PNG

Totally used number tuples = 70,000

Number of tuples used to train = 60,000

Number of tuples used to test = 10,000

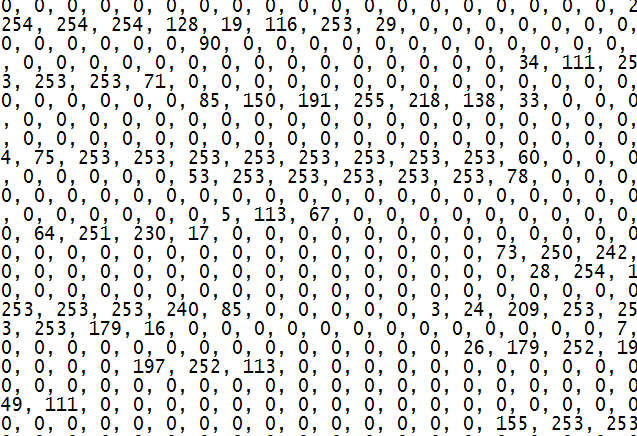


Fig 1.

Fig 1. Shows us the dataset where 0 is for black and 255 is for white.

This figure represents a MNIST handwritten number.

MNIST hand written digits:

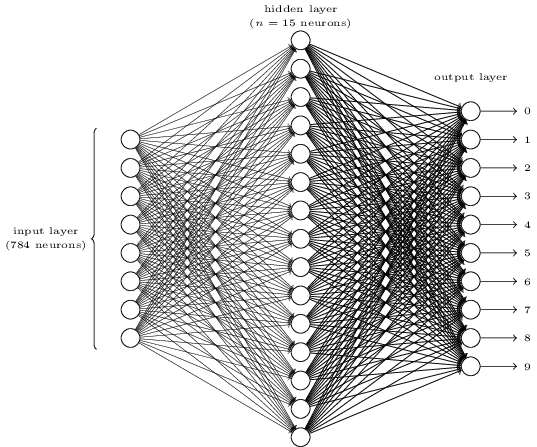


Here, each number is a tuple and each tuple has 784 attributes.

Each attribute is a pixel. That is, 1 digit is a 28\*28 matrix where each neuron holds a value between [0,255]. 0 for black and 255 for white. Suppose a neuron has a value of 170 it proves that, that neuron is a part of the edge of the number in the matrix.

The inference that we can draw from each attribute is that, each attribute holds a value through which we can optimize it and identify the number.

Each tuple. That is, all 784 pixels which are stored in 784 neurons are passed through several layers and the output is found. More the number of layers, more is the accuracy of the number.



# ALGORITHM

1. **[Start]**Generate random population of *n* chromosomes (suitable solutions for the problem)
2. **[Fitness]** Evaluate the fitness *f(x)*of each chromosome *x* in the population
3. **[New population]**Create a new population by repeating following steps until the new population is complete
   1. **[Selection]**Select two parent chromosomes from a population according to their fitness (the better fitness, the bigger chance to be selected)
   2. **[Crossover]** With a crossover probability cross over the parents to form a new offspring (children). If no crossover was performed, offspring is an exact copy of parents.
   3. **[Mutation]** With a mutation probability mutate new offspring at each locus (position in chromosome).
   4. **[Accepting]** Place new offspring in a new population
4. **[Replace]** Use new generated population for a further run of algorithm
5. **[Test]** If the end condition is satisfied, **stop**, and return the best solution in current population
6. **[Loop]** Go to step **2**

Genetic algorithm uses 3 operators to generate off strings. They are:

1. Selection

-Retains the best performing bit strings from one generation to the next

-parent1 = [1 0 1 0 0 1 1 0 0 0]

-parent2 = [1 0 0 1 0 0 1 0 1 0]

2. Crossover

-parent1 = [1 0 1 0 0 1 1 0 0 0]

-parent2 = [1 0 0 1 0 0 1 0 1 0]

-child = [1 0 0 0 0 0 1 0 0 0]

3. Mutation

- It chooses random features

-parent1 = [1 0 1 0 0 1 1 0 0 0]

-child = [0 1 0 1 0 1 0 0 0 1]

Every time we choose parents and generate off springs to form a new generation. Here, we form 10 generations. More the generations more is the accuracy.

Optimizer is the code that runs genetic algorithm. In this we initially create random population using Network and parameter choices. And it creates a list in pop (by appending pop). Each random population created is appended in pop.

Fitness function evaluates the value of neural network.

Evaluating evaluates the network accuracy. We use the neural network parameters and check the fitness in the GA. In evolve we select, crossover and mutate. Where as in breed we just crossover and mutate. Children will be bred from the program – networks.

In brute program we use brute force method. It selects neural networks without intelligently optimizing.

In main they are creating random and creating set. Then training plus printing the network.

In Train we get the data set, compile the model and builds the model. Fit function in the train program fits the model. Instead if print we store the output in log.txt file.

-Train and Network are used for neural networks. To take the different parameters and different attributes for each parameter. Find random and to train and print the network.

-Optimizer is the actual genetic algorithm code.

-Main is used to run the code.

-Brute is used for random selection of neural networks without intelligently optimizing.

# CODE

1. main.py

**"""Entry point to evolving the neural network. Start here."""**

**import logging**

**from optimizer import Optimizer**

**from tqdm import tqdm**

**# Setup logging.**

**logging.basicConfig(**

**format='%(asctime)s - %(levelname)s - %(message)s',**

**datefmt='%m/%d/%Y %I:%M:%S %p',**

**level=logging.DEBUG,**

**filename='log.txt'**

**)**

**def train\_networks(networks, dataset):**

**"""Train each network.**

**Args:**

**networks (list): Current population of networks**

**dataset (str): Dataset to use for training/evaluating**

**"""**

**pbar = tqdm(total=len(networks))**

**for network in networks:**

**network.train(dataset)**

**pbar.update(1)**

**pbar.close()**

**def get\_average\_accuracy(networks):**

**"""Get the average accuracy for a group of networks.**

**Args:**

**networks (list): List of networks**

**Returns:**

**float: The average accuracy of a population of networks.**

**"""**

**total\_accuracy = 0**

**for network in networks:**

**total\_accuracy += network.accuracy**

**return total\_accuracy / len(networks)**

**def generate(generations, population, nn\_param\_choices, dataset):**

**"""Generate a network with the genetic algorithm.**

**Args:**

**generations (int): Number of times to evole the population**

**population (int): Number of networks in each generation**

**nn\_param\_choices (dict): Parameter choices for networks**

**dataset (str): Dataset to use for training/evaluating**

**"""**

**optimizer = Optimizer(nn\_param\_choices)**

**networks = optimizer.create\_population(population)**

**# Evolve the generation.**

**for i in range(generations):**

**logging.info("\*\*\*Doing generation %d of %d\*\*\*" %**

**(i + 1, generations))**

**# Train and get accuracy for networks.**

**train\_networks(networks, dataset)**

**# Get the average accuracy for this generation.**

**average\_accuracy = get\_average\_accuracy(networks)**

**# Print out the average accuracy each generation.**

**logging.info("Generation average: %.2f%%" % (average\_accuracy \* 100))**

**logging.info('-'\*80)**

**# Evolve, except on the last iteration.**

**if i != generations - 1:**

**# Do the evolution.**

**networks = optimizer.evolve(networks)**

**# Sort our final population.**

**networks = sorted(networks, key=lambda x: x.accuracy, reverse=True)**

**# Print out the top 5 networks.**

**print\_networks(networks[:5])**

**def print\_networks(networks):**

**"""Print a list of networks.**

**Args:**

**networks (list): The population of networks**

**"""**

**logging.info('-'\*80)**

**for network in networks:**

**network.print\_network()**

**def main():**

**"""Evolve a network."""**

**generations = 10 # Number of times to evole the population.**

**population = 10 # Number of networks in each generation.**

**dataset = 'mnist'**

**nn\_param\_choices = {**

**'nb\_neurons': [4, 8, 16],**

**'nb\_layers': [1, 2, 3, 4],**

**'activation': ['relu', 'elu', 'tanh', 'sigmoid'],**

**'optimizer': ['rmsprop', 'adam', 'sgd', 'adagrad',**

**'adadelta', 'adamax', 'nadam'],**

**}**

**logging.info("\*\*\*Evolving %d generations with population %d\*\*\*" %**

**(generations, population))**

**generate(generations, population, nn\_param\_choices, dataset)**

**if \_\_name\_\_ == '\_\_main\_\_':**

**main()**

2. train.py

**from keras.datasets import mnist, cifar10**

**from keras.models import Sequential**

**from keras.layers import Dense, Dropout**

**from keras.utils.np\_utils import to\_categorical**

**from keras.callbacks import EarlyStopping**

**# Helper: Early stopping.**

**early\_stopper = EarlyStopping(patience=5)**

**def get\_cifar10():**

**"""Retrieve the CIFAR dataset and process the data."""**

**# Set defaults.**

**nb\_classes = 10**

**batch\_size = 64**

**input\_shape = (3072,)**

**# Get the data.**

**(x\_train, y\_train), (x\_test, y\_test) = cifar10.load\_data()**

**x\_train = x\_train.reshape(50000, 3072)**

**x\_test = x\_test.reshape(10000, 3072)**

**x\_train = x\_train.astype('float32')**

**x\_test = x\_test.astype('float32')**

**x\_train /= 255**

**x\_test /= 255**

**# convert class vectors to binary class matrices**

**y\_train = to\_categorical(y\_train, nb\_classes)**

**y\_test = to\_categorical(y\_test, nb\_classes)**

**return (nb\_classes, batch\_size, input\_shape, x\_train, x\_test, y\_train, y\_test)**

**def get\_mnist():**

**"""Retrieve the MNIST dataset and process the data."""**

**# Set defaults.**

**nb\_classes = 10**

**batch\_size = 128**

**input\_shape = (784,)**

**# Get the data.**

**(x\_train, y\_train), (x\_test, y\_test) = mnist.load\_data()**

**x\_train = x\_train.reshape(60000, 784)**

**x\_test = x\_test.reshape(10000, 784)**

**x\_train = x\_train.astype('float32')**

**x\_test = x\_test.astype('float32')**

**x\_train /= 255**

**x\_test /= 255**

**# convert class vectors to binary class matrices**

**y\_train = to\_categorical(y\_train, nb\_classes)**

**y\_test = to\_categorical(y\_test, nb\_classes)**

**return (nb\_classes, batch\_size, input\_shape, x\_train, x\_test, y\_train, y\_test)**

**def compile\_model(network, nb\_classes, input\_shape):**

**"""Compile a sequential model.**

**Args:**

**network (dict): the parameters of the network**

**Returns:**

**a compiled network.**

**"""**

**# Get our network parameters.**

**nb\_layers = network['nb\_layers']**

**nb\_neurons = network['nb\_neurons']**

**activation = network['activation']**

**optimizer = network['optimizer']**

**model = Sequential()**

**# Add each layer.**

**for i in range(nb\_layers):**

**# Need input shape for first layer.**

**if i == 0:**

**model.add(Dense(nb\_neurons, activation=activation, input\_shape=input\_shape))**

**else:**

**model.add(Dense(nb\_neurons, activation=activation))**

**model.add(Dropout(0.2)) # hard-coded dropout**

**# Output layer.**

**model.add(Dense(nb\_classes, activation='softmax'))**

**model.compile(loss='categorical\_crossentropy', optimizer=optimizer,**

**metrics=['accuracy'])**

**return model**

**def train\_and\_score(network, dataset):**

**"""Train the model, return test loss.**

**Args:**

**network (dict): the parameters of the network**

**dataset (str): Dataset to use for training/evaluating**

**"""**

**if dataset == 'cifar10':**

**nb\_classes, batch\_size, input\_shape, x\_train, \**

**x\_test, y\_train, y\_test = get\_cifar10()**

**elif dataset == 'mnist':**

**nb\_classes, batch\_size, input\_shape, x\_train, \**

**x\_test, y\_train, y\_test = get\_mnist()**

**model = compile\_model(network, nb\_classes, input\_shape)**

**model.fit(x\_train, y\_train,**

**batch\_size=batch\_size,**

**epochs=10, # using early stopping, so no real limit**

**verbose=1,**

**validation\_data=(x\_test, y\_test),**

**callbacks=[early\_stopper])**

**score = model.evaluate(x\_test, y\_test, verbose=0)**

**return score[1] # 1 is accuracy. 0 is loss.**

3. network.py

**"""Class that represents the network to be evolved."""**

**import random**

**import logging**

**from train import train\_and\_score**

**class Network():**

**"""Represent a network and let us operate on it.**

**Currently only works for an MLP.**

**"""**

**def \_\_init\_\_(self, nn\_param\_choices=None):**

**"""Initialize our network.**

**Args:**

**nn\_param\_choices (dict): Parameters for the network, includes:**

**nb\_neurons (list): [64, 128, 256]**

**nb\_layers (list): [1, 2, 3, 4]**

**activation (list): ['relu', 'elu']**

**optimizer (list): ['rmsprop', 'adam']**

**"""**

**self.accuracy = 0.**

**self.nn\_param\_choices = nn\_param\_choices**

**self.network = {} # (dic): represents MLP network parameters**

**def create\_random(self):**

**"""Create a random network."""**

**for key in self.nn\_param\_choices:**

**self.network[key] = random.choice(self.nn\_param\_choices[key])**

**def create\_set(self, network):**

**"""Set network properties.**

**Args:**

**network (dict): The network parameters**

**"""**

**self.network = network**

**def train(self, dataset):**

**"""Train the network and record the accuracy.**

**Args:**

**dataset (str): Name of dataset to use.**

**"""**

**if self.accuracy == 0.:**

**self.accuracy = train\_and\_score(self.network, dataset)**

**def print\_network(self):**

**"""Print out a network."""**

**logging.info(self.network)**

**logging.info("Network accuracy: %.2f%%" % (self.accuracy \* 100))**

**4. optimizer.py**

**from functools import reduce**

**from operator import add**

**import random**

**from network import Network**

**class Optimizer():**

**"""Class that implements genetic algorithm for MLP optimization."""**

**def \_\_init\_\_(self, nn\_param\_choices, retain=0.4,**

**random\_select=0.1, mutate\_chance=0.2):**

**"""Create an optimizer.**

**Args:**

**nn\_param\_choices (dict): Possible network paremters**

**retain (float): Percentage of population to retain after**

**each generation**

**random\_select (float): Probability of a rejected network**

**remaining in the population**

**mutate\_chance (float): Probability a network will be**

**randomly mutated**

**"""**

**self.mutate\_chance = mutate\_chance**

**self.random\_select = random\_select**

**self.retain = retain**

**self.nn\_param\_choices = nn\_param\_choices**

**def create\_population(self, count):**

**"""Create a population of random networks.**

**Args:**

**count (int): Number of networks to generate, aka the**

**size of the population**

**Returns:**

**(list): Population of network objects**

**"""**

**pop = []**

**for \_ in range(0, count):**

**# Create a random network.**

**network = Network(self.nn\_param\_choices)**

**network.create\_random()**

**# Add the network to our population.**

**pop.append(network)**

**return pop**

**@staticmethod**

**def fitness(network):**

**"""Return the accuracy, which is our fitness function."""**

**return network.accuracy**

**def grade(self, pop):**

**"""Find average fitness for a population.**

**Args:**

**pop (list): The population of networks**

**Returns:**

**(float): The average accuracy of the population**

**"""**

**summed = reduce(add, (self.fitness(network) for network in pop))**

**return summed / float((len(pop)))**

**def breed(self, mother, father):**

**"""Make two children as parts of their parents.**

**Args:**

**mother (dict): Network parameters**

**father (dict): Network parameters**

**Returns:**

**(list): Two network objects**

**"""**

**children = []**

**for \_ in range(2):**

**child = {}**

**# Loop through the parameters and pick params for the kid.**

**for param in self.nn\_param\_choices:**

**child[param] = random.choice(**

**[mother.network[param], father.network[param]]**

**)**

**# Now create a network object.**

**network = Network(self.nn\_param\_choices)**

**network.create\_set(child)**

**# Randomly mutate some of the children.**

**if self.mutate\_chance > random.random():**

**network = self.mutate(network)**

**children.append(network)**

**return children**

**def mutate(self, network):**

**"""Randomly mutate one part of the network.**

**Args:**

**network (dict): The network parameters to mutate**

**Returns:**

**(Network): A randomly mutated network object**

**"""**

**# Choose a random key.**

**mutation = random.choice(list(self.nn\_param\_choices.keys()))**

**# Mutate one of the params.**

**network.network[mutation] = random.choice(self.nn\_param\_choices[mutation])**

**return network**

**def evolve(self, pop):**

**"""Evolve a population of networks.**

**Args:**

**pop (list): A list of network parameters**

**Returns:**

**(list): The evolved population of networks**

**"""**

**# Get scores for each network.**

**graded = [(self.fitness(network), network) for network in pop]**

**# Sort on the scores.**

**graded = [x[1] for x in sorted(graded, key=lambda x: x[0], reverse=True)]**

**# Get the number we want to keep for the next gen.**

**retain\_length = int(len(graded)\*self.retain)**

**# The parents are every network we want to keep.**

**parents = graded[:retain\_length]**

**# For those we aren't keeping, randomly keep some anyway.**

**for individual in graded[retain\_length:]:**

**if self.random\_select > random.random():**

**parents.append(individual)**

**# Now find out how many spots we have left to fill.**

**parents\_length = len(parents)**

**desired\_length = len(pop) - parents\_length**

**children = []**

**# Add children, which are bred from two remaining networks.**

**while len(children) < desired\_length:**

**# Get a random mom and dad.**

**male = random.randint(0, parents\_length-1)**

**female = random.randint(0, parents\_length-1)**

**# Assuming they aren't the same network...**

**if male != female:**

**male = parents[male]**

**female = parents[female]**

**# Breed them.**

**babies = self.breed(male, female)**

**# Add the children one at a time.**

**for baby in babies:**

**# Don't grow larger than desired length.**

**if len(children) < desired\_length:**

**children.append(baby)**

**parents.extend(children)**

**return parents**

5. brute.py

**"""Iterate over every combination of hyperparameters."""**

**import logging**

**from network import Network**

**from tqdm import tqdm**

**# Setup logging.**

**logging.basicConfig(**

**format='%(asctime)s - %(levelname)s - %(message)s',**

**datefmt='%m/%d/%Y %I:%M:%S %p',**

**level=logging.DEBUG,**

**filename='brute-log.txt'**

**)**

**def train\_networks(networks, dataset):**

**"""Train each network.**

**Args:**

**networks (list): Current population of networks**

**dataset (str): Dataset to use for training/evaluating**

**"""**

**pbar = tqdm(total=len(networks))**

**for network in networks:**

**network.train(dataset)**

**network.print\_network()**

**pbar.update(1)**

**pbar.close()**

**# Sort our final population.**

**networks = sorted(networks, key=lambda x: x.accuracy, reverse=True)**

**# Print out the top 5 networks.**

**print\_networks(networks[:5])**

**def print\_networks(networks):**

**"""Print a list of networks.**

**Args:**

**networks (list): The population of networks**

**"""**

**logging.info('-'\*80)**

**for network in networks:**

**network.print\_network()**

**def generate\_network\_list(nn\_param\_choices):**

**"""Generate a list of all possible networks.**

**Args:**

**nn\_param\_choices (dict): The parameter choices**

**Returns:**

**networks (list): A list of network objects**

**"""**

**networks = []**

**# This is silly.**

**for nbn in nn\_param\_choices['nb\_neurons']:**

**for nbl in nn\_param\_choices['nb\_layers']:**

**for a in nn\_param\_choices['activation']:**

**for o in nn\_param\_choices['optimizer']:**

**# Set the parameters.**

**network = {**

**'nb\_neurons': nbn,**

**'nb\_layers': nbl,**

**'activation': a,**

**'optimizer': o,**

**}**

**# Instantiate a network object with set parameters.**

**network\_obj = Network()**

**network\_obj.create\_set(network)**

**networks.append(network\_obj)**

**return networks**

**def main():**

**"""Brute force test every network."""**

**dataset = 'mnist'**

**nn\_param\_choices = {**

**'nb\_neurons': [4, 8, 16, 32, 64],**

**'nb\_layers': [1, 2, 3, 4],**

**'activation': ['relu', 'elu', 'tanh', 'sigmoid'],**

**'optimizer': ['rmsprop', 'adam', 'sgd', 'adagrad',**

**'adadelta', 'adamax', 'nadam'],**

**}**

**logging.info("\*\*\*Brute forcing networks\*\*\*")**

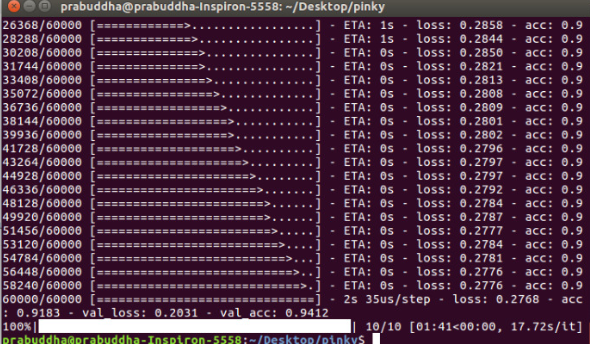
**networks = generate\_network\_list(nn\_param\_choices)**

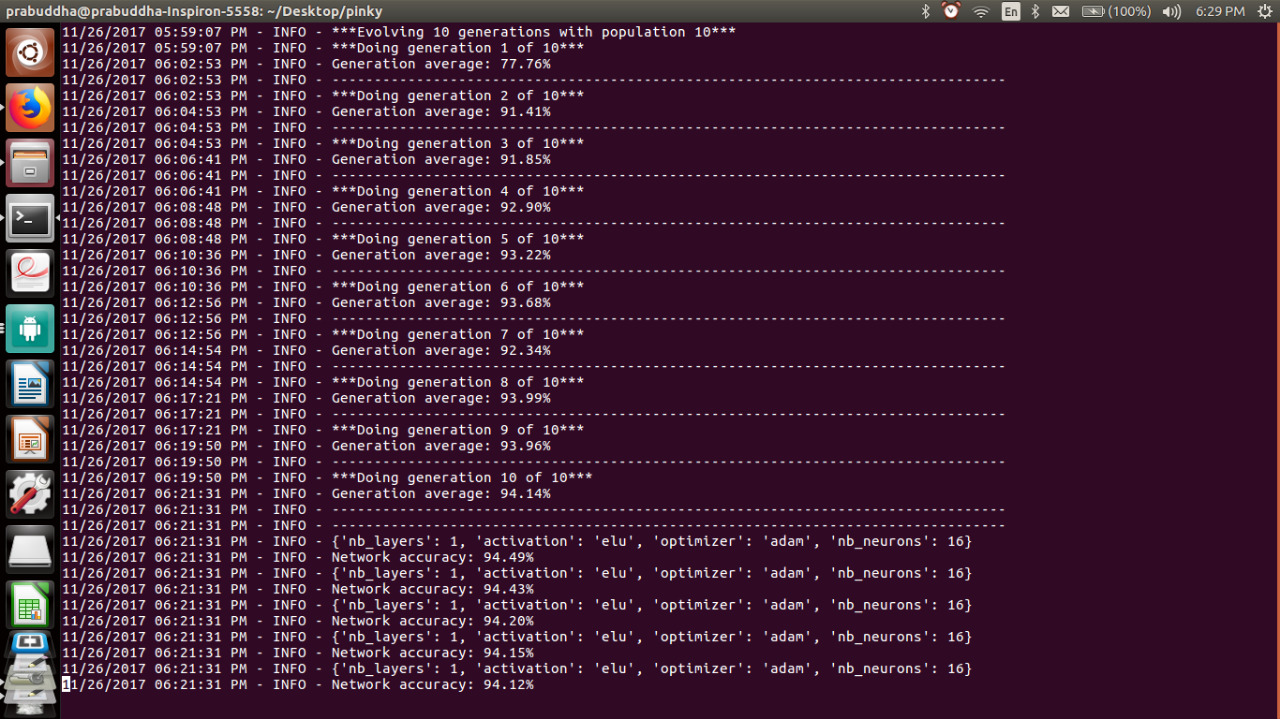
**train\_networks(networks, dataset)**

**if \_\_name\_\_ == '\_\_main\_\_':**

**main()**

# OUTPUT





Log.txt file:

**11/23/2017 04:56:03 PM - INFO - \*\*\*Evolving 10 generations with population 20\*\*\***

**11/23/2017 04:56:03 PM - INFO - \*\*\*Doing generation 1 of 10\*\*\***

**11/23/2017 04:58:13 PM - INFO - \*\*\*Evolving 2 generations with population 2\*\*\***

**11/23/2017 04:58:13 PM - INFO - \*\*\*Doing generation 1 of 2\*\*\***

**11/23/2017 05:02:43 PM - INFO - \*\*\*Evolving 10 generations with population 20\*\*\***

**11/23/2017 05:04:28 PM - INFO - \*\*\*Doing generation 1 of 10\*\*\***

**11/23/2017 05:06:35 PM - INFO - \*\*\*Evolving 10 generations with population 20\*\*\***

**11/23/2017 05:06:40 PM - INFO - \*\*\*Doing generation 1 of 10\*\*\***

**11/23/2017 05:07:20 PM - INFO - \*\*\*Evolving 10 generations with population 20\*\*\***

**11/23/2017 05:07:21 PM - INFO - \*\*\*Doing generation 1 of 10\*\*\***

**11/23/2017 05:18:53 PM - INFO - Generation average: 77.22%**

**11/23/2017 05:18:53 PM - INFO - --------------------------------------------------------------------------------**

**11/23/2017 05:18:53 PM - INFO - \*\*\*Doing generation 2 of 10\*\*\***

**11/23/2017 05:28:27 PM - INFO - Generation average: 91.31%**

**11/23/2017 05:28:27 PM - INFO - --------------------------------------------------------------------------------**

**11/23/2017 05:28:27 PM - INFO - \*\*\*Doing generation 3 of 10\*\*\***

**11/23/2017 05:35:57 PM - INFO - Generation average: 95.42%**

**11/23/2017 05:35:57 PM - INFO - --------------------------------------------------------------------------------**

**11/23/2017 05:35:57 PM - INFO - \*\*\*Doing generation 4 of 10\*\*\***

**11/23/2017 05:42:43 PM - INFO - Generation average: 96.42%**

**11/23/2017 05:42:43 PM - INFO - --------------------------------------------------------------------------------**

**11/23/2017 05:42:43 PM - INFO - \*\*\*Doing generation 5 of 10\*\*\***

**11/23/2017 06:00:52 PM - INFO - Generation average: 93.55%**

**11/23/2017 06:00:52 PM - INFO - --------------------------------------------------------------------------------**

**11/23/2017 06:00:52 PM - INFO - \*\*\*Doing generation 6 of 10\*\*\***

**11/23/2017 06:13:02 PM - INFO - Generation average: 94.05%**

**11/23/2017 06:13:02 PM - INFO - --------------------------------------------------------------------------------**

**11/23/2017 06:13:02 PM - INFO - \*\*\*Doing generation 7 of 10\*\*\***

**11/23/2017 06:20:19 PM - INFO - Generation average: 95.92%**

**11/23/2017 06:20:19 PM - INFO - --------------------------------------------------------------------------------**

**11/23/2017 06:20:19 PM - INFO - \*\*\*Doing generation 8 of 10\*\*\***

**11/23/2017 06:31:36 PM - INFO - Generation average: 94.34%**

**11/23/2017 06:31:36 PM - INFO - --------------------------------------------------------------------------------**

**11/23/2017 06:31:36 PM - INFO - \*\*\*Doing generation 9 of 10\*\*\***

**11/23/2017 06:41:20 PM - INFO - Generation average: 97.24%**

**11/23/2017 06:41:20 PM - INFO - --------------------------------------------------------------------------------**

**11/23/2017 06:41:20 PM - INFO - \*\*\*Doing generation 10 of 10\*\*\***

**11/23/2017 06:50:18 PM - INFO - Generation average: 97.36%**

**11/23/2017 06:50:18 PM - INFO - --------------------------------------------------------------------------------**

**11/23/2017 06:50:18 PM - INFO - --------------------------------------------------------------------------------**

**11/23/2017 06:50:18 PM - INFO - {'nb\_layers': 1, 'activation': 'relu', 'optimizer': 'nadam', 'nb\_neurons': 64}**

**11/23/2017 06:50:18 PM - INFO - Network accuracy: 97.76%**

**11/23/2017 06:50:18 PM - INFO - {'nb\_layers': 3, 'activation': 'relu', 'optimizer': 'nadam', 'nb\_neurons': 64}**

**11/23/2017 06:50:18 PM - INFO - Network accuracy: 97.53%**

**11/23/2017 06:50:18 PM - INFO - {'nb\_layers': 3, 'activation': 'relu', 'optimizer': 'nadam', 'nb\_neurons': 64}**

**11/23/2017 06:50:18 PM - INFO - Network accuracy: 97.52%**

**11/23/2017 06:50:18 PM - INFO - {'nb\_layers': 4, 'activation': 'relu', 'optimizer': 'nadam', 'nb\_neurons': 64}**

**11/23/2017 06:50:18 PM - INFO - Network accuracy: 97.50%**

**11/23/2017 06:50:18 PM - INFO - {'nb\_layers': 4, 'activation': 'relu', 'optimizer': 'nadam', 'nb\_neurons': 64}**

**11/23/2017 06:50:18 PM - INFO - Network accuracy: 97.48%**

**11/23/2017 07:27:31 PM - INFO - \*\*\*Evolving 10 generations with population 20\*\*\***

**11/23/2017 07:27:31 PM - INFO - \*\*\*Doing generation 1 of 10\*\*\***

**11/23/2017 07:28:39 PM - INFO - \*\*\*Evolving 10 generations with population 10\*\*\***

**11/23/2017 07:28:39 PM - INFO - \*\*\*Doing generation 1 of 10\*\*\***

**11/23/2017 07:31:02 PM - INFO - Generation average: 76.52%**

**11/23/2017 07:31:02 PM - INFO - --------------------------------------------------------------------------------**

**11/23/2017 07:31:02 PM - INFO - \*\*\*Doing generation 2 of 10\*\*\***

**11/23/2017 07:32:38 PM - INFO - Generation average: 91.23%**

**11/23/2017 07:32:38 PM - INFO - --------------------------------------------------------------------------------**

**11/23/2017 07:32:38 PM - INFO - \*\*\*Doing generation 3 of 10\*\*\***

**11/23/2017 07:33:49 PM - INFO - Generation average: 92.66%**

**11/23/2017 07:33:49 PM - INFO - --------------------------------------------------------------------------------**

**11/23/2017 07:33:49 PM - INFO - \*\*\*Doing generation 4 of 10\*\*\***

**11/23/2017 07:35:07 PM - INFO - Generation average: 93.93%**

**11/23/2017 07:35:07 PM - INFO - --------------------------------------------------------------------------------**

**11/23/2017 07:35:07 PM - INFO - \*\*\*Doing generation 5 of 10\*\*\***

**11/23/2017 07:36:42 PM - INFO - Generation average: 94.19%**

**11/23/2017 07:36:42 PM - INFO - --------------------------------------------------------------------------------**

**11/23/2017 07:36:42 PM - INFO - \*\*\*Doing generation 6 of 10\*\*\***

**11/23/2017 07:37:58 PM - INFO - Generation average: 94.27%**

**11/23/2017 07:37:58 PM - INFO - --------------------------------------------------------------------------------**

**11/23/2017 07:37:58 PM - INFO - \*\*\*Doing generation 7 of 10\*\*\***

**11/23/2017 07:39:53 PM - INFO - Generation average: 94.41%**

**11/23/2017 07:39:53 PM - INFO - --------------------------------------------------------------------------------**

**11/23/2017 07:39:53 PM - INFO - \*\*\*Doing generation 8 of 10\*\*\***

**11/23/2017 07:41:08 PM - INFO - Generation average: 94.28%**

**11/23/2017 07:41:08 PM - INFO - --------------------------------------------------------------------------------**

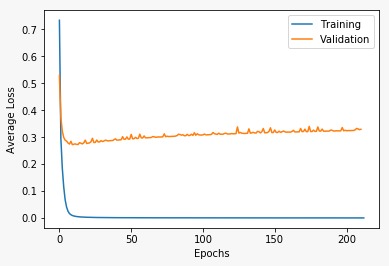
**11/23/2017 07:41:08 PM - INFO - \*\*\*Doing generation 9 of 10\*\*\***

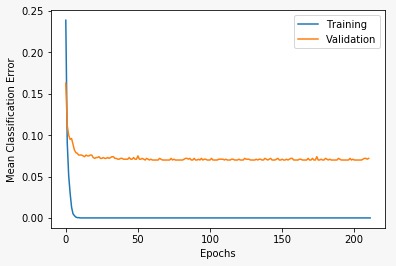
**11/23/2017 07:43:01 PM - INFO - Generation average: 92.72%**

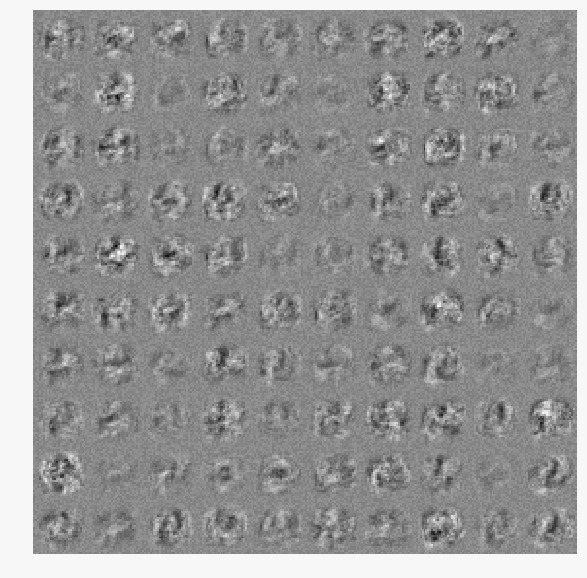
**11/23/2017 07:43:01 PM - INFO - --------------------------------------------------------------------------------**

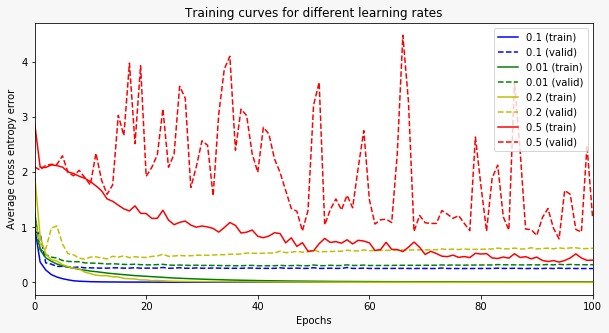
**11/23/2017 07:43:01 PM - INFO - \*\*\*Doing generation 10 of 10\*\*\***

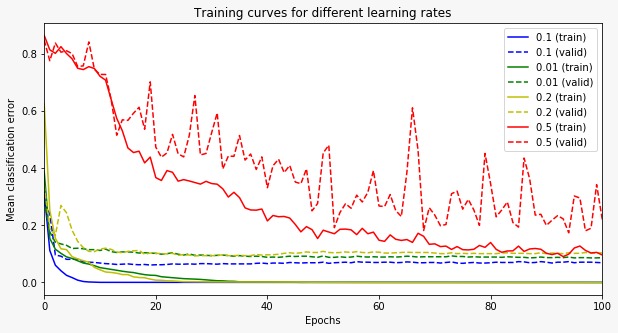
# GRAPHS

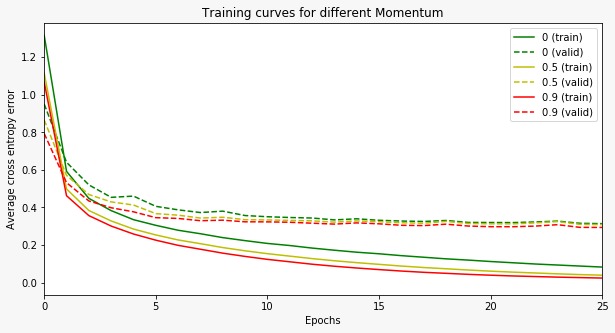


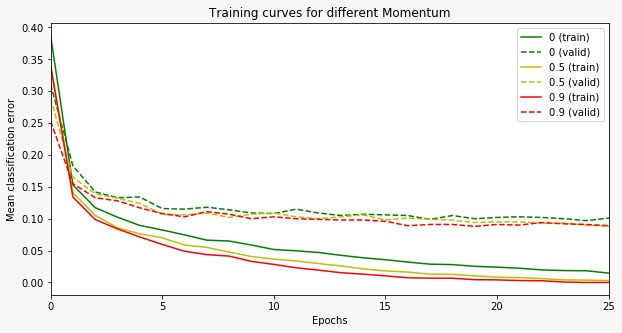


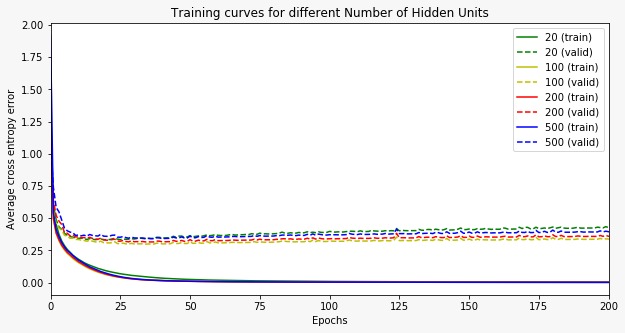


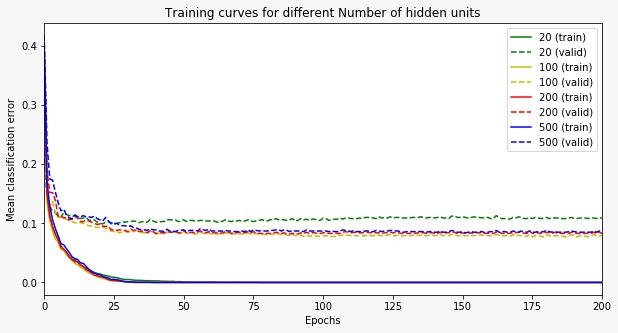


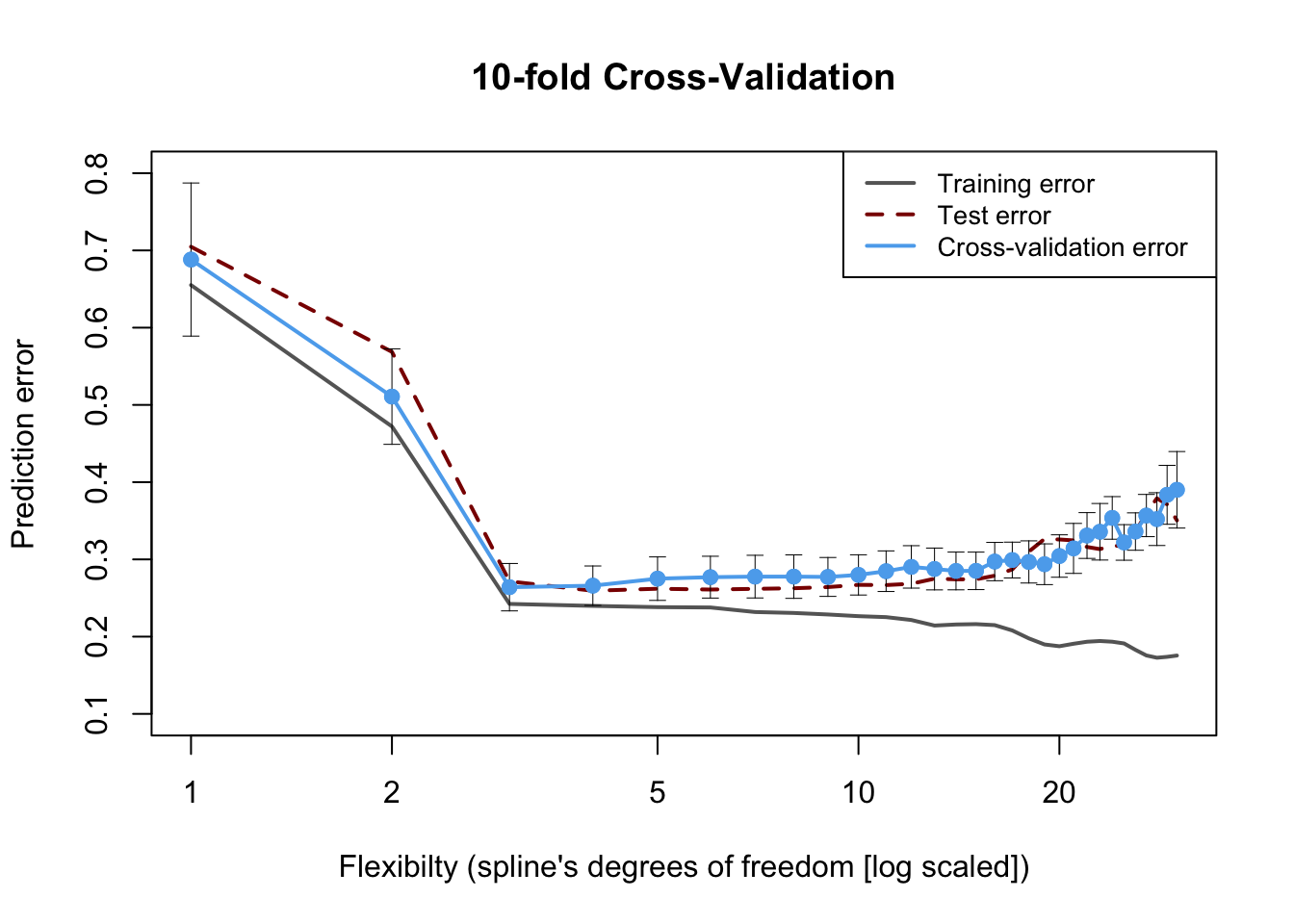


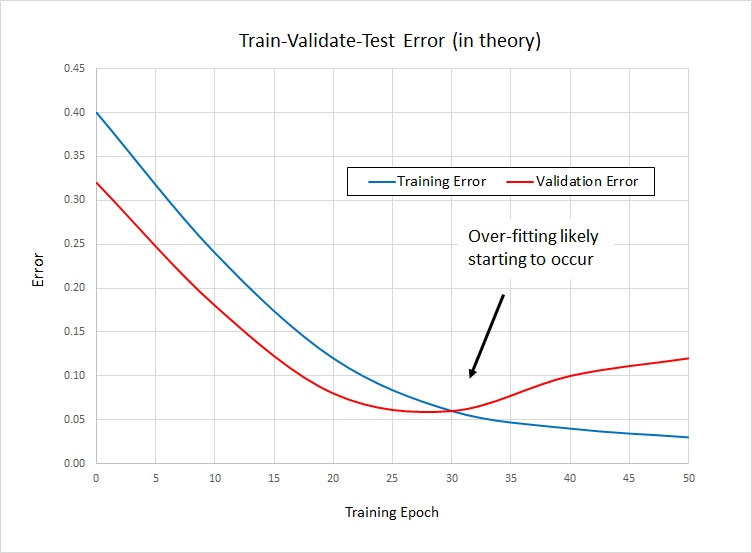












# DESCRIPTION AND SOCIAL IMPACT

The first 9 graphs are the training loss vs epochs, testing loss vs epochs, training vs varying learning rate etc. of the implementation of neural networks through Genetic algorithm.

Whereas the last 2 graphs are the graphs obtained by random selection code. And the graphs are plotted by using average validation accuracy per generation.

The increase in that is better through our implementation then the random selection code.

GENETIC ALGORITHM accuracy that we obtained is 97.48% which is much more higher than the compared code. Hence, proving it’s efficiency.

Socially By using this algorithm, we can predict the probable attributes that a parameters needs to have to have the best offspring as well as predicting the best outcome for the given problem.