Ted Holmberg

Fall 2019: CSCI 6522: Programming Assignment #2:

Artificial Neural Net (ANN)

Abstract:

This project tasked to build several Artificial Neural Networks (ANNs) to recognize one of the 03 (three) classes of flowers based on given 04 attributes or features of the flowers.

Source Code: https://github.com/scalemailted/MachineLearning2-HW3

Table of Contents:

I. Problem Statement

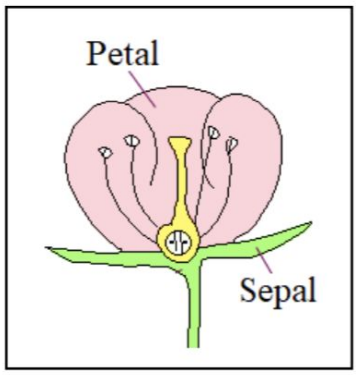
Data Set Information:
The data set contains 3 classes of 50 instances each, where each class refers to a type of iris plant. One class is linearly separable from the other 2; the latter are NOT linearly separable from each other.

Data Set Characteristics: Multivariate	**Attribute Characteristics:** Real	**Associated Task:** Classification
Number of Instances: 150	**Number of Attributes:** 4	**Missing Values:** No

Predicted attribute: class of iris plant.

Input 04 features are:
- Sepal length in cm
- Sepal width in cm
- Petal length in cm
- Petal width in cm

Output 03 classes are:
- Iris Setosa
- Iris Versicolour
- Iris Virginica

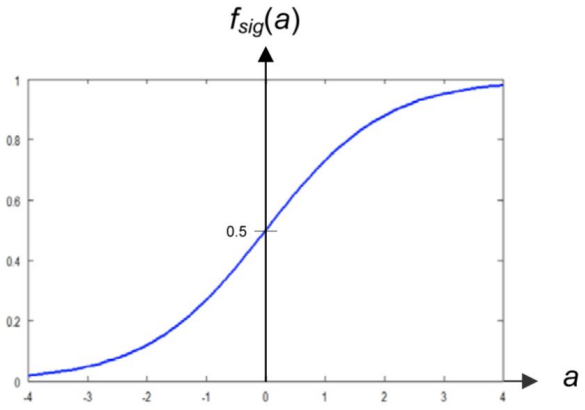


Full description of the problem including the data set is available here:
https://archive.ics.uci.edu/ml/datasets/Iris

Objective:
This project requires that a predictor be constructed that given 4 Input features (sepals and petals), it can determine what type of flower it is from the three output classes.

II. Approach

Logistic Regression:
Classification problems, or predictors for qualitative/discrete data relies on logistic-based activation functions. This is because, it's important to quickly select to a singular choice of one class from the group of possibilities.



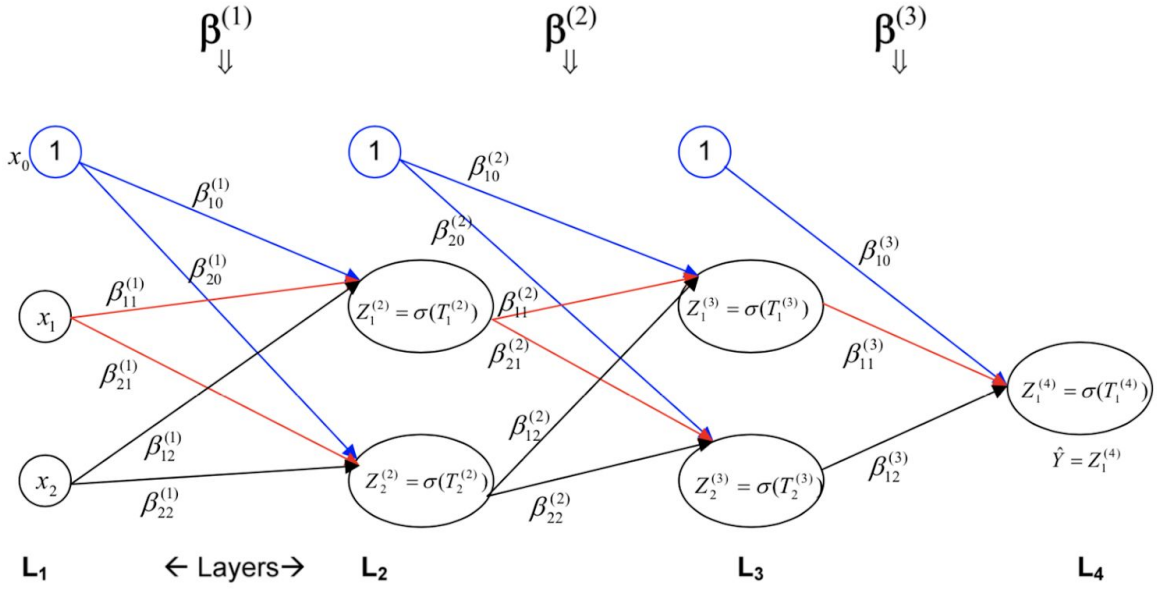
Sigmoid function,

$$f_{sig}(a)=\frac{1}{1+e^{-a}}$$

Octave code:

```
a=-4.0:0.1:4.0;
y=(1./(1+exp(-a)));
plot(a, y, 'LineWidth', 3);
```

(Feed-Forward) Artificial Neural Network (ANN):
The central idea for ANN is to extract linear combinations of the inputs as derived features, and then model the target as a nonlinear function of these features. The feed-forward neural network which is comprised of multi-layers of logistic regression models.



Neural Network Algorithm Explanation for this Project:
Step 1: Load Feature Data (X) into Input Layer
Step 2: Initialize Nodes in Hidden Layers via Tabular Data Structures (rows/columns)
Step 3: Randomize a Set of Beta Values
Step 4: Forward Propagate Feature Data through ANN
Step 5: Calculate Mean Square Error of Output Layer
Step 6: Check Exit Condition
Step 7: Back Propagate => Calculate delta errors
Step 8: Back Propagate => Update Weights
Step 9: Update Beta values
Step 10: GO Step 4

Technology & Implementation:
Python 3.7 programming language selected to implement the project for its wide, popular usage in machine learning applications. Many libraries and modules make designing projects easy. Additionally, python is portable and open source. The Python dependencies for this project require: Numpy and Sklearn.

Data Preprocessing:
All project data contained in a comma-separated-value file: 'iris-data.txt', such that the initial four columns are the input features and the last column is the flower classification. Numpy is used to load the feature data into the software as an ndarray named X. Sklearn is used to load the output data in with OneHotEncoder which converts a string label into a binary tuple with a number of bits equal to the number of classes in output. Thus:

Iris-setosa = (1,0,0)
Iris-versicolor = (0,1,0)
Iris-virginica = (0,0,1)

III. ANN Predictor

This section defines the necessary components for a feed-forward neural network.

Input Layer

The input later is four nodes. A node for each feature. Represented as real values.

Output Layer

The output layer is three nodes. A node for each output class

Hidden Layers

Each of the hidden layers will have arbitrary units ranging from 2 to 20.

```
def create_layers(inputs, hiddens, outputs):
    layers = [];
    layers.append(inputs);
    for i in range(hiddens):
        nodes = random.randint(2,20);
        layers.append(nodes);
    layers.append(outputs);
    return layers
```

Batching

All Observations from training dataset may be batched together and fed into the ANN all at one time in the form of a matrix. Thus linear algebra operations may be used to process results at a given node.This allows entire dataset to be used in a single feed of the ANN for training.

Setup Input (X) / Output (Y) into ANN

```
    def set_data(self, X, Y):
        self.X = X;
        self.Y = Y.T;
        self.Nx, self.P = X.shape;
        self.Ny, self.K = Y.shape;
```

Initialize Random Beta Values (B) into ANN

```
def init_betas(self):
    self.B = np.empty( len(self.layers)-1, dtype=object )
    for i in range( len(self.B) ):
        self.B[i] = 1.4 * np.random.rand( self.layers[i]+1, self.layers[i+1] ) - 0.7;
```

Initialize Transmitter (T) Between Layers

```
def init_transmitters(self):
        self.T = np.empty(len(self.layers), dtype=object)
        for i in range(len(self.layers)):
            self.T[i] = np.ones( self.layers[i]  );
```

Initialize Layers (Z) of ANN

```
    def init_Zs(self):
        self.Z = np.empty( len(self.layers), dtype=object)
        for i in range( len(self.layers) ):
            self.Z[i] = np.zeros( self.layers[i]+1  );
        self.Z[0] = (np.append(self.X, np.ones([self.Nx,1]), axis=1)).T
```

Forward Propagation

$$Z^{(1)} = X$$
$$T^{(2)} = \beta^{(1)^T} Z^{(1)}$$
$$Z^{(2)} = f_{sig}.(T^{(2)}) \text{ and add } Z_0^{(2)}$$
$$T^{(3)} = \beta^{(2)^T} Z^{(2)}$$
$$Z^{(3)} = f_{sig} \cdot \left(T^{(3)}\right) \text{ and add } Z_0^{(3)}$$
$$T^{(4)} = \beta^{(3)^T} Z^{(3)}$$
$$Z^{(4)} = \hat{Y} = f_{sig}(T^{(4)})$$

This is the algorithm for Forward Propagation.
Z(1) is the input layer.
Z(4) is the output layer.
Z(2,3) are the hidden layers.
Z layers use sigmoid activation function to fit result into likely classifications. All possibilities are calculated at each layer and batched together.
The T(2,3,4) are the linear equation with the Beta values and the intermediary features values.
The more hidden layers, the more non-linearity can be modeled in the ANN.

```
  def forward_propagation(self):
        for i in range( len(self.layers)-1 ):
            self.T[i+1] = self.B[i].T @ self.Z[i];
            if (i+1)<len(self.layers)-1:
                t = 1/(1+np.exp(-self.T[i+1]))
                ones = np.ones(self.Nx)
                self.Z[i+1] = (np.column_stack([t.T,ones])).T
            else:
                self.Z[i+1] = (1/(1+np.exp(-self.T[i+1])))
```

The forward propagation algorithm uses the sigmoid activation function. Forward propagation requires that the feature values are modeled as input layer and transmitted to each hidden layer until they are processed into output layer. The intermediary values are held in Transmission (T) Matrices, Layer Matrices (Z) where Z[0] is Input layer, Z[-1] is Output layer, and Z[0:-1] are hidden layers. A forward propagation converts the batch of inputs into a batch of class likelihoods given as a binary string scoped between [0.1).

IV. ANN Training Methods

There are four different approaches for training an ANN to optimize its ability to accurately predict.

1. Back Propagation

This approach to minimize uses gradient descent. Because of the compositional form of the model, the gradient can be easily derived using the chain rule for differentiation. This can be computed by a forward and backward sweep over the network, keeping track only of quantities local to each unit.

Delta Errors

```
def get_delta(self):
    self.d[-1] = (self.Z[-1]-self.Y) * self.Z[-1] * (1 - self.Z[-1])
    for i in reversed( range(1, len(self.layers)-1) ):
        W = self.Z[i][:-1] * (1 - self.Z[i][:-1]);
        D = self.d[i+1];
        self.d[i] = np.ones([self.Nx,self.layers[i]])
        for m in range(self.Nx):
            self.d[i][m] = (W.T[m] * np.sum( (D.T[m]  * self.B[i][:-1] ), 1  ));
        self.d[i] = self.d[i].T
```

Apply Weights

```
def update_weights(self):
    for i in range( len(self.layers)-1 ):
        W = self.Z[i][:-1].T;
        V1 = np.zeros( [self.layers[i],self.layers[i+1]] );
        V2 = np.zeros( [1,self.layers[i+1]] );
        D = self.d[i+1].T;
        for m in range(self.Nx):
            V1 = V1 + (W[m].reshape(-1,1) @ D[m].reshape(1,-1));
            V2 = V2 + D[m]
            self.B[i][:-1] = self.B[i][:-1] - (self.alpha/self.Nx) * V1;
            self.B[i][ -1] = self.B[i][ -1] - (self.alpha/self.Nx) * V2;
```

Apply weights updates the Beta values using the gradient descent algorithm. The step size for gradient descent is controlled by alpha. Nx is the number of rows in the input data.

2. Back Propagation and Momentum

Momentum in Gradient Descent (GD)

- The method of **momentum** is designed to accelerate learning, especially in the face of high curvature, small but consistent gradients, or noisy gradients.
- The momentum algorithm accumulates an **exponentially decaying moving average** of the past gradients and continues to move in their direction.

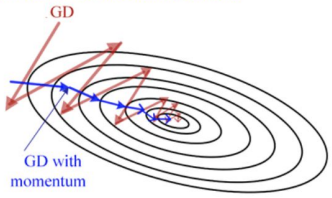


Figure 1: The heavily zig-zak red path indicates the path of GD without the momentum. The blue path cutting across the contours shows the path followed by the momentum learning rule, which is a less zig-zak path that helps faster convergence.

… Momentum in Gradient Descent (GD)

- Formally, the momentum algorithm introduces a variable $\boldsymbol{v}$ that plays the role of velocity—it is the direction and speed at which the parameters move through parameter space.
- The velocity is set to an exponentially decaying average of the negative gradient. Momentum in physics is mass times velocity. In the momentum learning algorithm, we assume unit mass, so the velocity vector v may also be regarded as the momentum of the particle.
- A hyperparameter $\boldsymbol{\gamma} \in [0, 1)$ determines how quickly the contributions of previous gradients exponentially decay.
- For GD without momentum, we have

$$\beta(t+1) = \beta(t) - \alpha \frac{\partial E}{\partial \beta(t)}$$

- For GD with momentum, we use:

$$v(t) = v(t-1)\gamma - \alpha \frac{\partial E}{\partial \beta(t)} \quad \text{where, velocity } \boldsymbol{v}(0) = 0, \text{ when iteration } \boldsymbol{t} = 1$$

$$\beta(t+1) = \beta(t) + v(t)$$

Initializing Momentum

```
        def init_momentum(self):
            self.velocity = np.empty(len(self.layers)-1, dtype=object)
            for i in range(len(self.velocity)):
                self.velocity[i] = np.zeros( [self.layers[i]+1,self.layers[i+1]] );
```

Calculating Velocity

```
def update_weights(self):
    for i in range( len(self.layers)-1 ):
        W = self.Z[i][:-1].T;
        V1 = np.zeros( [self.layers[i],self.layers[i+1]] );
        V2 = np.zeros( [1,self.layers[i+1]] );
        D = self.d[i+1].T;
        for m in range(self.Nx):
        V1 = V1 + (W[m].reshape(-1,1) @ D[m].reshape(1,-1));
        V2 = V2 + D[m]
        self.velocity[i][:-1] *= self.gamma
        self.velocity[i][:-1] -= (self.alpha/self.Nx) * V1;
        self.velocity[i][ -1] *= self.gamma
        self.velocity[i][ -1] -= (self.alpha/self.Nx) * V2;
        self.B[i][:-1] = self.B[i][:-1] + self.velocity[i][:-1];
        self.B[i][ -1] = self.B[i][ -1] + self.velocity[i][ -1];
```

Gamma

Gamma is a parameter that controls how much momentum to apply.

3. Genetic Algorithm

A genetic algorithm can be used to evolve the set of beta values used to process the feed-forward prediction in the ANN. The general process of applying a genetic algorithm requires that we model Beta in such a way that we can perform calculate a fitness, perform a mutation, and perform a crossover. An Object-Oriented approach is used. Beta is modeled as a class in the following way

Model: Beta Object

```
class Betas:
    def __init__(self, layers):
        self.layers = np.empty( len(layers)-1, dtype=object )
        self.score = float('inf');
        self.accuracy = 0;
        for i in range( len(self.layers) ):
            self.layers[i] = 4 * np.random.rand( layers[i]+1,layers[i+1] ) - 2.0;
```

Determine Beta Fitness

Each Beta object should be able to calculate its fitness, we define a Betas fitness with its mean square error. The predicted value of X is determined by a feed-forward of the ANN. So two methods can be created to optimize this. The first sets up the ANN for the test dataset, resizing the T & Z matrices.

```
    def setup_mse(self, X,Y, ann):
        ann.set_data(X,Y)
        ann.setup_training()
```

The second method actually calculates the fitness using a reference to the ANN. This Beta overwrites the B in the ANN and then runs the test method on it. Every Beta object may calculate its own score in this manner.

```
    def get_mse(self, X,Y, ann):
        ann.B = self.layers;
        ann.test(X,Y);
        self.score = ann.test_log[-1];
```

Beta Crossovers

Crossover operations are performed by the Genetic Algorithm by invoking the swap methods within a Beta object. Swap has two parameters. The index of the hidden layer to swap out, and the collect of beta values at a given layer to be swapped. The algorithm will invoke the swap method between two selected Beta objects

```
    def swap(self, index, otherBetas):
        temp = self.layers[index];
        self.layers[index] = otherBetas.layers[index];
        otherBetas.layers[index] = temp;
```

Beta Mutations

Mutations are implemented by randomly selecting a hidden layer matrix, and within that randomly selecting a value and then replace it with a new random value.

```
    def mutate(self):
        ilayer = np.random.randint( len(self.layers) );
        rows, cols = self.layers[ilayer].shape;
        i = np.random.randint(rows);
        j = np.random.randint(cols);
        self.layers[ilayer][i][j] = 4 * np.random.random() - 2.0;
```

Genetic Algorithm:

The Genetic Algorithm is also implemented using an Object-Oriented approach. The initial properties of the Genetic Algorithm include the population size, elite rate, crossover rate, mutation rate, and the population list.

```
class GeneticAlgorithm:
    def __init__(self, config):
        self.population_size = 200;
        self.elite_rate = .10;
        self.cross_over = .80;
        self.mutation   = .05;
        self.population = [];
```

Genetic Algorithm: Initialize Population

To initialize a starting population, a for-loop for population size which simply instantiates a Beta object and appends it to the population. The Beta constructor randomizes its starting values.

```
    def initialize_population(self ):
        for i in range( self.population_size ):
            self.population.append( Betas( self.layers) );
```

Genetic Algorithm: Calculate Fitness

The genetic algorithm must iterate through each beta in the population list, and invoke it to calculate its MSE, then sort the population list using the score.

```
    def calculate_fitness(self):
        for b in self.population:
            x,y,ann = self.data;
            b.get_mse(x,y,ann);
        self.population.sort( key=lambda b: b.score  )
```

Genetic Algorithm: Elite Population

A deep copy of the elite portion of the population is made to carry over to the next generation.

```
   def getElite(self):
        ielite = int(self.population_size * self.elite_rate)
        self.pop_elite = copy.deepcopy( self.population[0:ielite]);
```

Genetic Algorithm: Non-elite Population

Non-elite population size must be determined, from that crossovers and mutations occur on current generation using a random sampling that preferences the most fit Betas.

```
def getNonelite(self):
  nonelite_size = self.population_size - int(self.population_size * self.elite_rate)
  crossover_size = int(nonelite_size * self.cross_over);
  self.pop_nonelite = [];
  for i in range(0,crossover_size,2):
      self.getCrossovers();
  while (len(self.pop_nonelite) < nonelite_size ):
      self.getRandoms();
```

Genetic Algorithm: Get Crossovers

This is a helper method that randomly selects two Beta objects from the population and swaps a layer's value between them.

```
    def getCrossovers(self):
        #print('get crossovers')
        b1 = self.selectOne();
        b2 = self.selectOne();
        index = np.random.randint( len(self.layers)-1 );
        b1.swap(index,b2);
        self.pop_nonelite.append(b1);
        self.pop_nonelite.append(b2);
```

Genetic Algorithm: Select One

This is a helper method that uses a roulette random sampling to select a Beta object from population, prefencing the most fit.

```
    def selectOne(self):
        max = sum([b.score for b in self.population])
        selection_probs = [b.score/max for b in self.population][::-1]
        index = np.random.choice(len(self.population)
        return self.population[index, p=selection_probs)]
```

Genetic Algorithm: Mutate Non-Elite Population

This performs the random mutations on the non-elite portion of the next generation. The mutation rate determines how many objects are randomly selected from the list. The Beta object's mutate method is then invoked.

```
    def mutatePopulation(self):
        #print('mutate population');
        mutation_size = int( len(self.pop_nonelite) * self.mutation)
        for i in range(mutation_size):
            index = np.random.randint( len(self.pop_nonelite) );
            self.pop_nonelite[index].mutate();
```

Genetic Algorithm: Get Random

This is a helper method that instantiates a new Beta object and appends it to the non-elite list.

```
  def getRandoms(self):
       #print('get randoms');
       b = Betas(self.layers);
       self.pop_nonelite.append(b)
```

Genetic Algorithm: Execute the Algorithm

This is the main logic of the genetic algorithm. The population is initialized. The fitness for each member is calculated and sorted. While-loop controls how many generations occur. For each generation, elite portion are selected and saved, then the remaining non-elite portion is generated. These two are combined to form the new generation which then has its fitness calculated and its ranked

```
def execute(self):
   self.initialize_population();
   self.calculate_fitness();
   counter = 0;
   while self.population[0].score < self.target_value or counter < self.max_gen:
           self.getElite();
           self.getNonelite();
           self.population = self.pop_elite + self.pop_nonelite;
           self.calculate_fitness();
           counter += 1;
```

4. Combining Methods

Training an ANN may actually use all three of these methods combined. For instance instead of starting the Backpropagation (BP) algorithm with Momentum on a random Beta set. First, fInd quality Beta set using a Genetic Algorithm, to further fit with BP. This approach proved to provide the best results in this project

Genetic Algorithm + Backpropagation + Momentum

V. Models (Implementations)

This project evaluated the performance of several ANNs. All models are listed below along with specifications.

ANN01:

Artificial Neural Network with one hidden layer. Training uses Backpropagation with no Momentum. *Source code: ANN01.py*

```
config = {
    'layers': create_layers(inputs=4,hiddens=1,outputs=3),
    'alpha': 0.2,
    'target_mse': 0.0001,
    'max_epoch': 2000,
}
```

ANN03:

Artificial Neural Network with three hidden layers. Training uses Backpropagation with no momentum. *Source code: ANN03.py*

```
config = {
    'layers': create_layers(inputs=4,hiddens=3,outputs=3),
    'alpha': 0.2,
    'target_mse': 0.0001,
    'max_epoch': 2000,
}
```

ANN07

Artificial Neural Network with seven hidden layers. Training uses Backpropagation with no momentum. *Source code: ANN07.py*

```
config = {
    'layers': create_layers(inputs=4,hiddens=7,outputs=3),
    'alpha': 0.2,
    'target_mse': 0.0001,
    'max_epoch': 2000,
}
```

ANN03Mo
Artificial Neural Network with three hidden layers. Training uses Backpropagation with momentum. *Source code: ANN03Mo.py*

ANN03Mom, picks a suitable value of the momentum hyperparameter ∈ [0,1).
gamma = 0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9

A 10% bonus for finding the best gamma value , and must show graph for at least 10 different values of using 10 FCV to find the best (see results section)

```
config = {
    'layers': create_layers(inputs=4,hiddens=3,outputs=3),
    'alpha': 0.2,
    'target_mse': 0.0001,
    'max_epoch': 2000,
    'gamma': 0.0
}
```

ANN03GA
Artificial Neural Network with three hidden layers. Training uses Genetic Algorithm. *Source code: ANN03GA.py*

For ANN03GA, the units of a chromosome are the parameters (i.e., the weights or, βs) of the ANN (in real-value).

- Your GA must apply elite, cross and mutation operations.
- A crossover position can be in between two βs or could be a β. In case, it is a β, a split ratio of the β will be chosen randomly.
- For mutation operation, a chromosome unit β, will be selected and will be either incremented or decremented (decided randomly) 10% of the current value.
- Your population size will be 200, elite rate 10%, crossover rate 80%, and mutation rate 5%. The GA generation will be the epoch of the ANN03GA.
- The MSE value computed from a chromosome will be the fitness of that chromosome.

ANN03GABP

Artificial Neural Network with three hidden layers. Training uses Genetic Algorithm followed by Backpropagation with momentum. *Source code: ANN03GABP.py*

```
config = {
    'layers': create_layers(inputs=4,hiddens=3,outputs=3),
    'alpha': 0.2,
    'target_mse': 0.0001,
    'max_epoch': 2000,
    'gamma': 8.0
}
```

```
config_GA = {
    'target_value':0.005,
    'max_generations':1000,
}
```

You will get a 5% bonus marks for including backpropagation (BP) in ANN03GA, called this version ANN03GABP.

VI. Testing

To test performance of each ANN model, 10 Fold Cross Validation is used that allows for all data in the dataset to be used for both the training and testing. Since this approach generates 10 different error and accuracy, the average among all 10 folds is then calculated as the final result. Each epoch during training and testing is recorded and displayed in the Results section.

10 Fold Cross Validation

Python library, sklearn, easily implements 10-Fold Cross Validation

```
folds = KFold(n_splits=10, shuffle=True)
folds.get_n_splits(X)
ann = NeuralNet(config);
for train_index, test_index in folds.split(X):
    X_train, X_test = X[train_index], X[test_index]
    Y_train, Y_test = Y[train_index], Y[test_index]
    ann.train(X_train, Y_train);
    ann.test(X_test, Y_test);
```

Average Error

Mean Square Error is calculated for each epoch of each fold. The average across all 10 folds in then calculated.

```
train_err = np.mean(np.array(ann.error_log).T,axis=1)
test_err = np.mean(np.array(ann.beta_error_log).T,axis=1)
```

Average Accuracy

Python library, sklearn, calculates the accuracy score between actual value and predicted value.

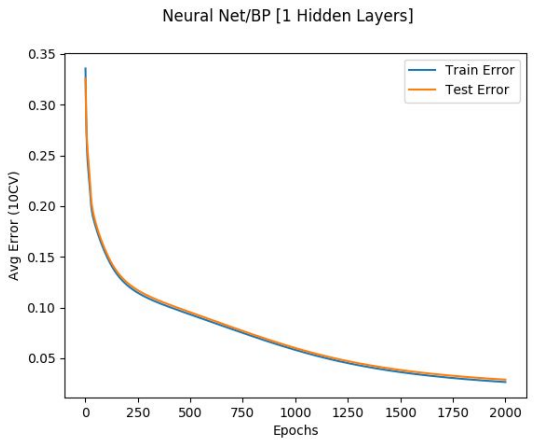
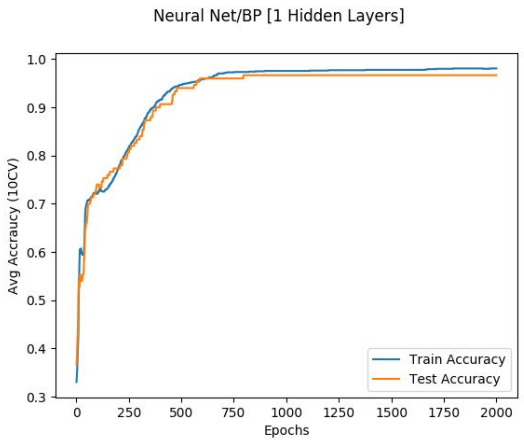
```
def get_accuracy(self):
        y = self.Y.T
        y_pred = np.zeros_like(self.Z[-1].T)
        y_pred[np.arange(len(self.Z[-1].T)), self.Z[-1].T.argmax(1)] = 1
        score = accuracy_score(y, y_pred)
        return score
```

VII. Results

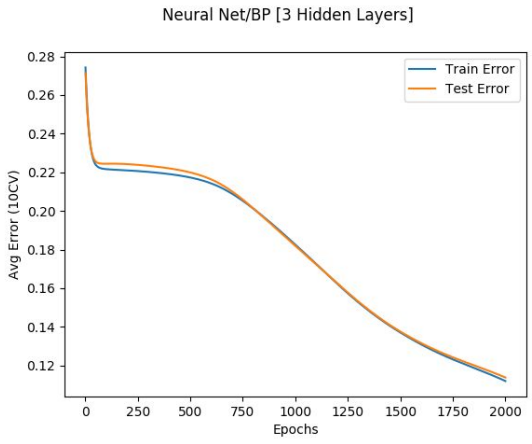
- In a table show, 10 *minimum* test-error (i.e., MSE) collected from 10 FCV and their average for each of the ANNs.
- For each of the ANNs, plot graphs of the training and test
    - (a) average MSE of the 10-folds and
    - (b) average classification accuracy,

  for epochs, at least ranging from 1 to 2000.

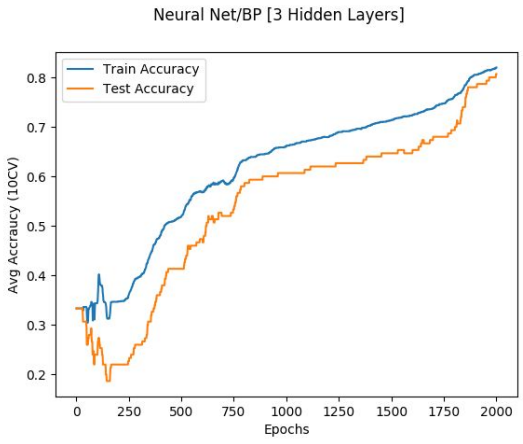
ANN01:

	
Average MSE of the 10-folds	*Average classification accuracy*

ANN03



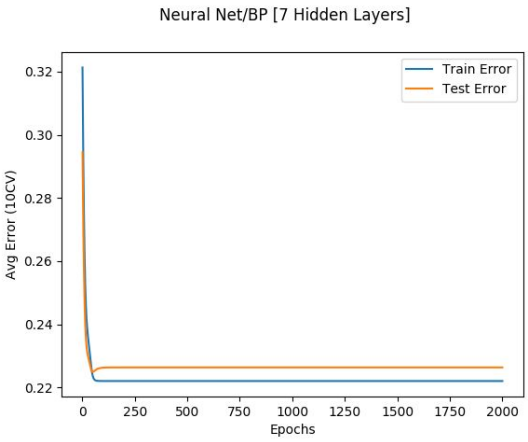
Average MSE of the 10-folds



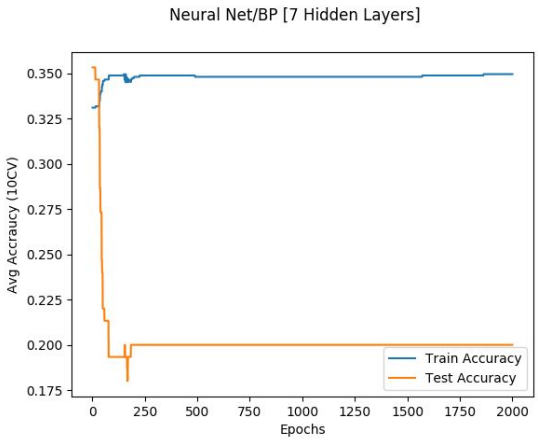
Average classification accuracy

ANN07

This model is overfit, it performs terribly.

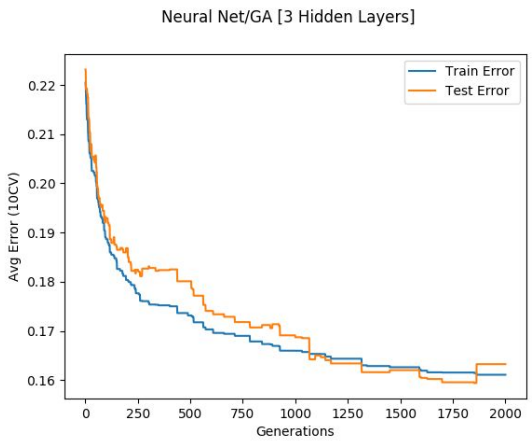


Average MSE of the 10-folds

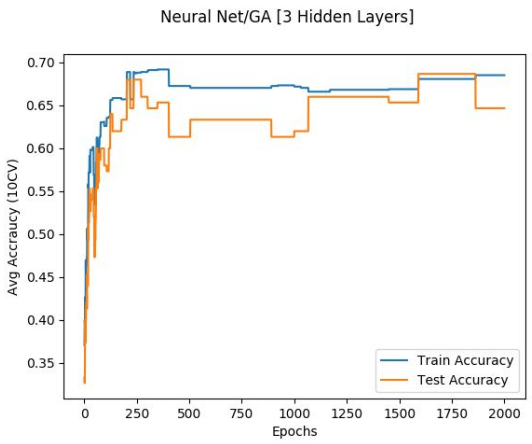


Average classification accuracy

ANN03GA



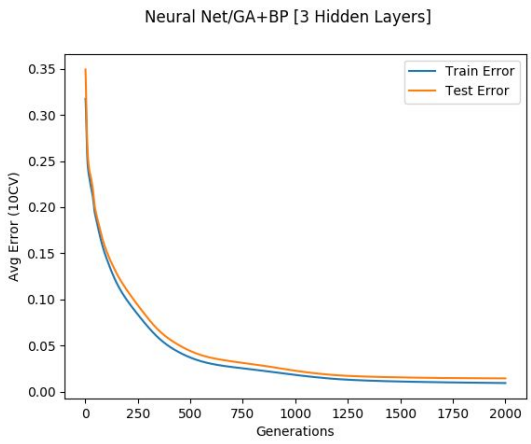
Average MSE of the 10-folds



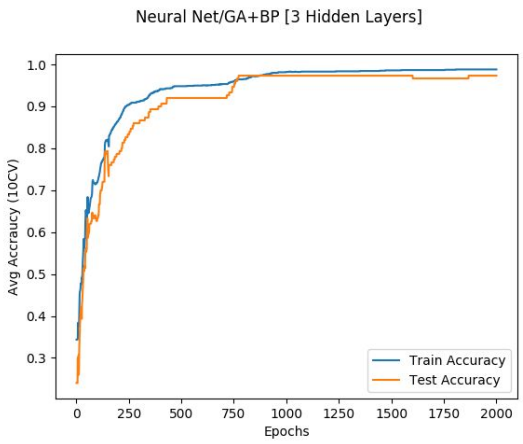
Average classification accuracy

ANN03GABP

1000 generation in GA
2000 Epochs in BP
Momentum of 0.8



Average MSE of the 10-folds

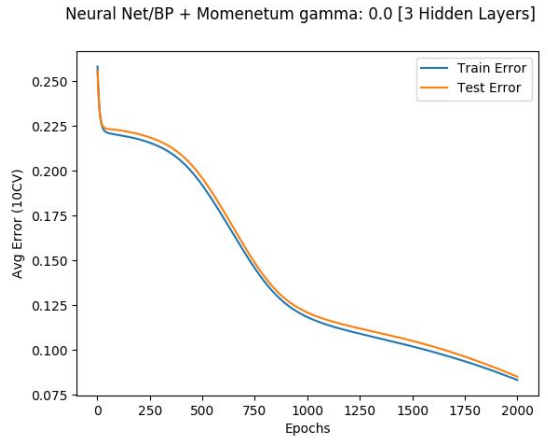
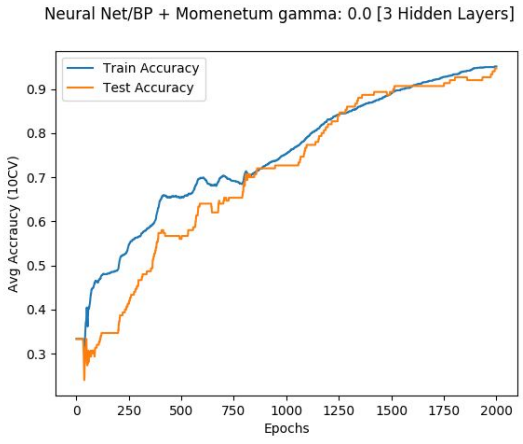


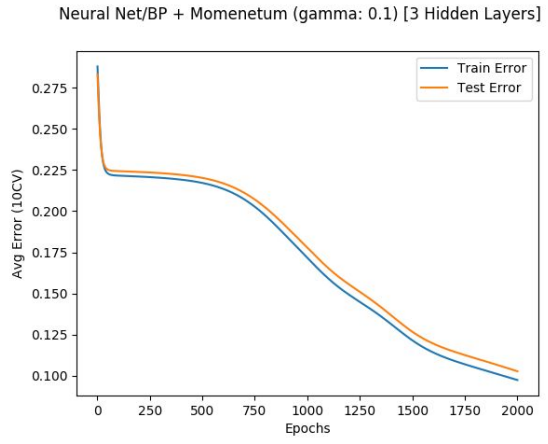
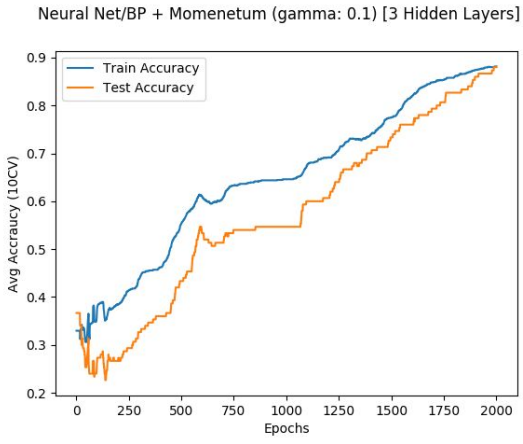
Average classification accuracy

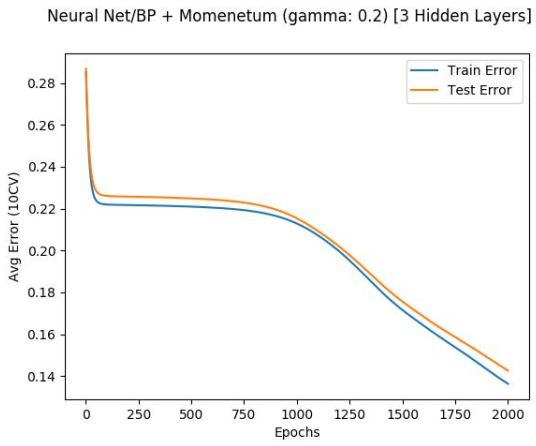
ANN03MO

10 different variants of this model was evaluated in an attempt to find the best value for gamma. Each gamma value is incremented by a tenth: 0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9. The higher value of gamma yielded better results.

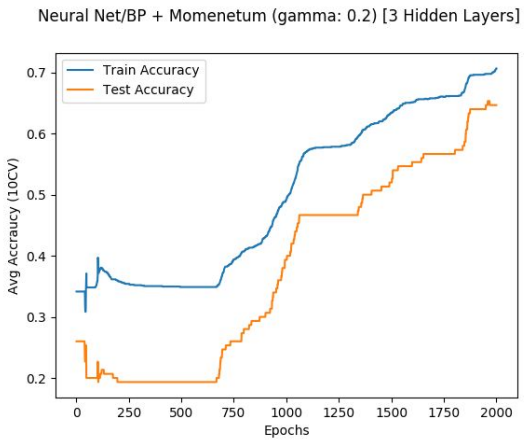
In my implementation, the plots for gamma = 0.8 appear to be the best.

<table>
<tr>
<td> 
</td>
<td> 
</td>
</tr>
<tr>
<td>Average MSE of the 10-folds</td>
<td>Average classification accuracy</td>
</tr>
</table>

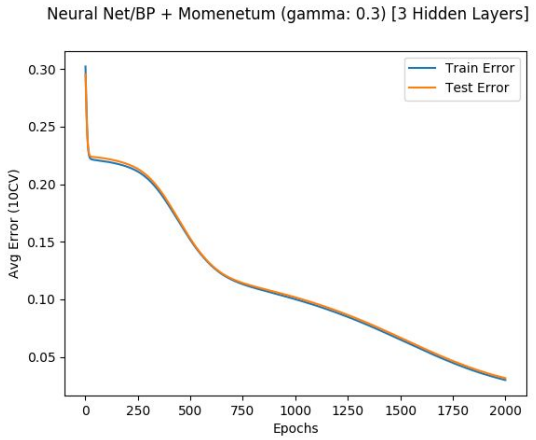
<table>
<tr>
<td> 
</td>
<td> 
</td>
</tr>
<tr>
<td>Average MSE of the 10-folds</td>
<td>Average classification accuracy</td>
</tr>
</table>



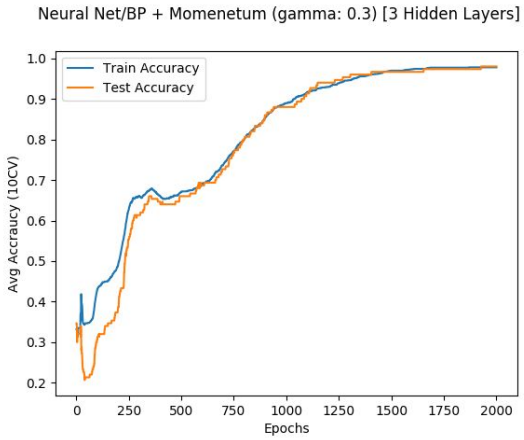
Average MSE of the 10-folds



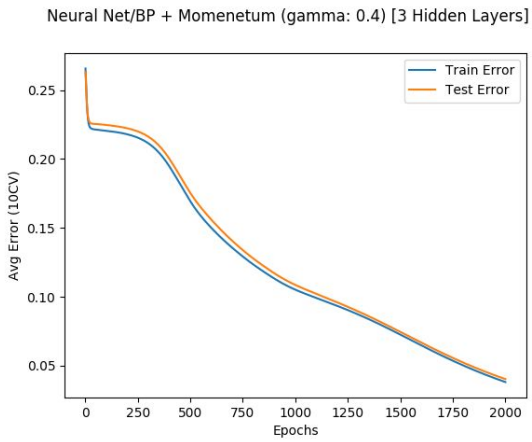
Average classification accuracy



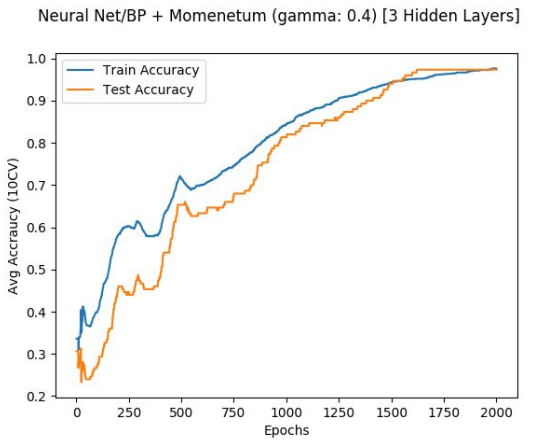
Average MSE of the 10-folds



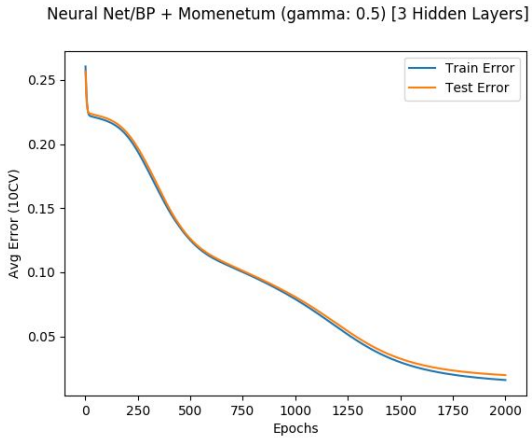
Average classification accuracy



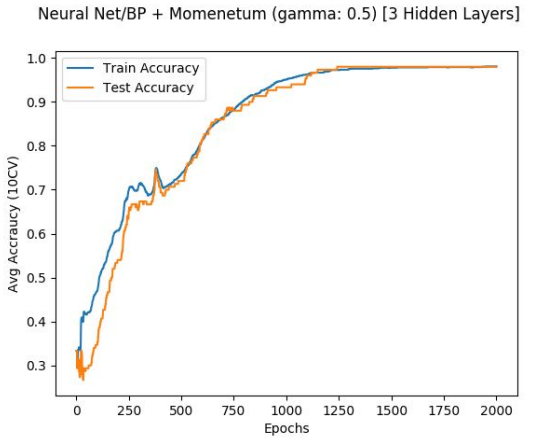
Average MSE of the 10-folds



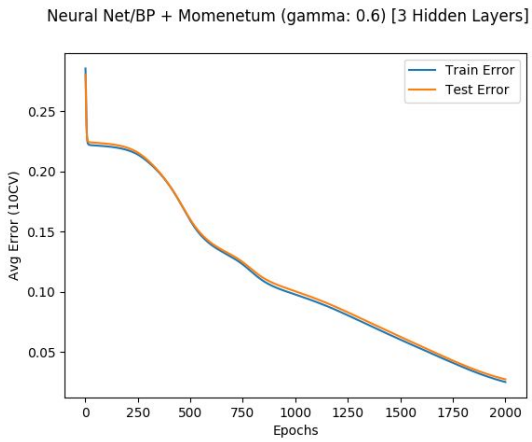
Average classification accuracy



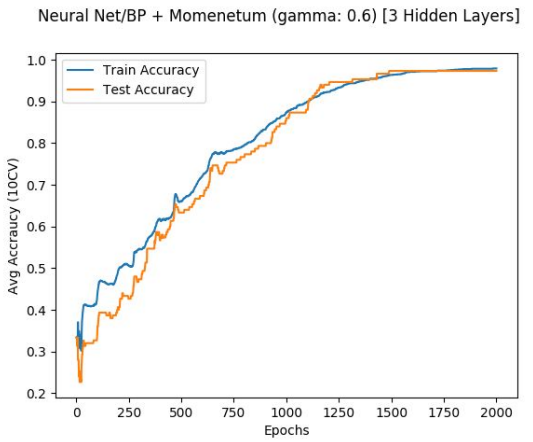
Average MSE of the 10-folds



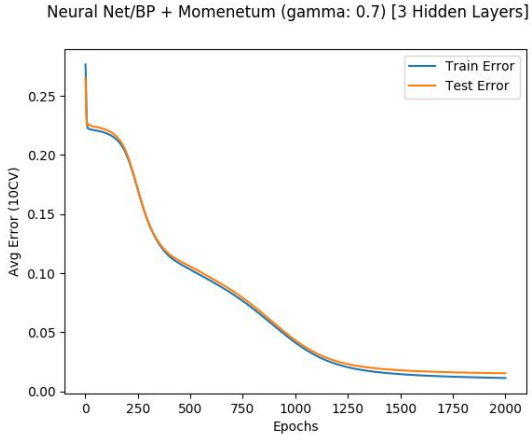
Average classification accuracy



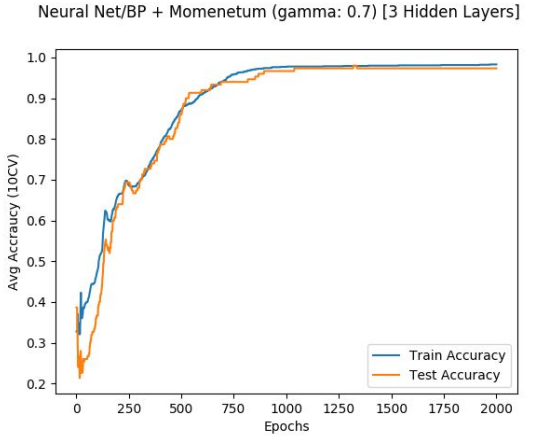
Average MSE of the 10-folds



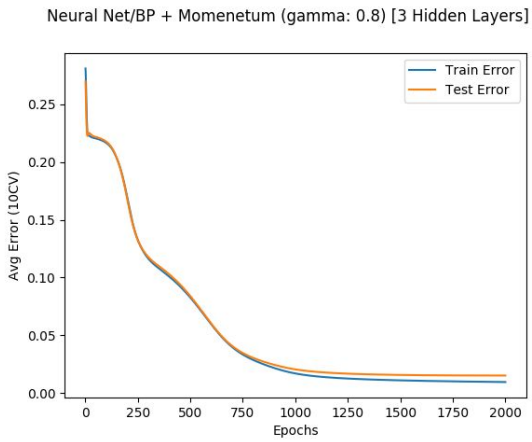
Average classification accuracy



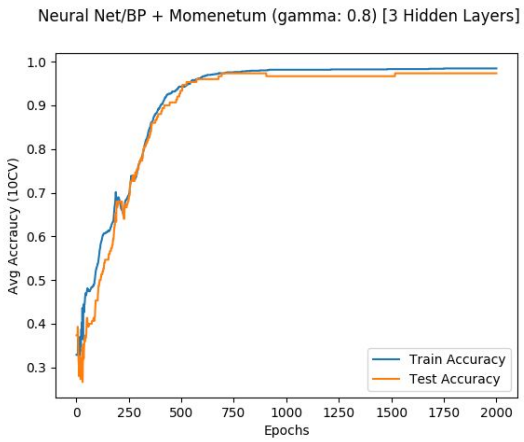
Average MSE of the 10-folds



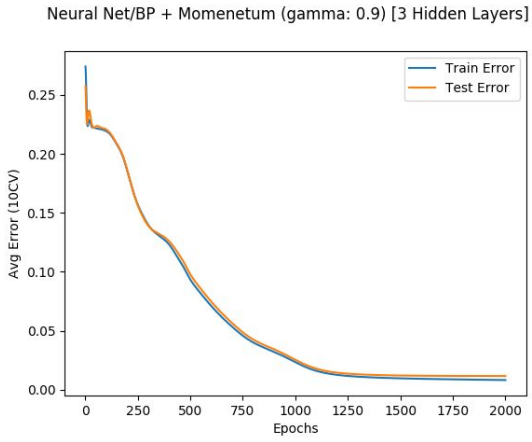
Average classification accuracy



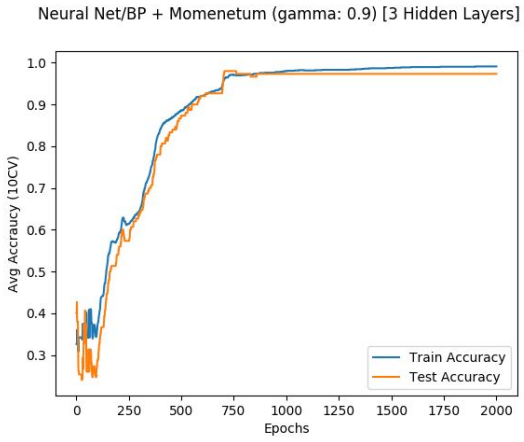
Average MSE of the 10-folds



Average classification accuracy



Average MSE of the 10-folds



Average classification accuracy

Fall 2019: CSCI 6522 Homework #3
Neural Net

Instructions

- All work must be your own (other than the instructor provided codes and hints to be used). You are NOT to work in teams on this assignment.

- Format: Your solution must be typed. Submit as a single compressed file (via Moodle) containing all the related files in it. Name it as PA2_<Your_name>. Provide hardcopy (see Reporting section.)

- The top/cover page of the report should have the title, “Fall 2019: CSCI 6522, Adv. ML-II Programming Assignment #2”. Then your, “Name:_____
  ID:_____

- Total marks=100+bonus(15)=115.

PART (A) [20 × 3 = 60 marks]

Description

This programming assignment is to build several Artificial Neural Networks (ANNs) to recognize one of the 03 (three) classes of flowers based on given 04 attributes or features of the flowers. The details are:

- You will build 03 different batch versions of ANNs having a hidden layer(s): 1, 3, 7 and call them ANN01, ANN03, ANN07, respectively.

- Each of the hidden layers will have arbitrary units ranging from 2 to 20.

- You are welcome to generate any additional useful features from the given datasets to be used as an input feature.

PART (B) [20 +20 = 40 marks]

Description

- Based on the constructed ANN03 from part-A, build a new ANN by (i) adding momentum (call it ANN03Mom) and (ii) replacing the backpropagation of the ANN03 by a Genetic Algorithm (call it ANN03GA).
- For ANN03Mom, pick a suitable value of the momentum hyperparameter $\gamma \in [0,1)$. You will get a **10% bonus** for finding the best value of γ, but you must show the graph for at least 10 different values of using 10 FCV to find the best γ.

- For ANN03GA, the units of a chromosome are the parameters (i.e., the weights or, β s) of the ANN (in real-value).
    - Your GA must apply elite, cross and mutation operations.
    - A crossover position can be in between two β s or could be a β. In case, it is a β, a split ratio of the β will be chosen randomly.
    - For mutation operation, a chromosome unit β, will be selected and will be either incremented or decremented (decided randomly) 10% of the current value.
    - Your population size will be 200, elite rate 10%, crossover rate 80%, and mutation rate 5%. The GA generation will be the epoch of the ANN03GA.
    - The MSE value computed from a chromosome will be the fitness of that chromosome.
    - You will get a 5% bonus marks for including backpropagation (BP) in ANN03GA. We will call this version ANN03GABP.

Data

Full description of the problem including the data set is available here:
https://archive.ics.uci.edu/ml/datasets/Iris

- Input 04 features are:
    - Sepal length in cm
    - Sepal width in cm
    - Petal length in cm
    - Petal width in cm

- Output 03 classes are: o
    - Iris Setosa

    - Iris Versicolour
    - Iris Virginica

- Check Moodle for a copy of the datasets and related information.

Operations to Perform

- Train and Test your 05 (or, 06) ANNs using the training datasets using 10-fold cross-validations (10 FCVs).
- Exit condition for the ANN is to reach (at least) 2000 epochs.

Submission of the Report

Submit (both softcopy and hardcopy) a report that includes:

- In a table show, 10 *minimum* test-error (i.e., MSE) collected from 10 FCV and their average for each of the ANNs.
- For each of the ANNs, plot graphs of the training and test
    - (a) average MSE of the 10-folds and
    - (b) average classification accuracy,

  for epochs, at least ranging from 1 to 200.
- A readme file describing how to run your program.
- Program code with necessary comments.

Information

- Check the lecture note of chapter #04 to review the existing ideas and results.
- **You must follow the ANN code provided by the instructor in the class.** Extend the code for this assignment problem (You may convert the given code or idea of the code to a different programming language, and then you can extend it further).