Lab Manual
For

NEURAL NETWORKS LAB
(CM703PC)

IV B. TECH I SEMESTER
(R20 - AUTONOMOUS)

DEPARTMENT OF CSE
(ARTIFICIAL INTELLIGENCE AND MACHINE LEARNING)

ACE
Engineering College
An Autonomous Institution
Ghatkesar, Hyderabad - 501 301, Telangana.
Approved by AICTE & Affiliated to JNTUH
NBA Accredited B.Tech Courses, Accorded NACC A-Grade with 3.20 CGPA

ACE

Engineering College

An Autonomous Institution

Ghatkesar, Hyderabad - 501 301, Telangana.
Approved by AICTE & Affiliated to JNTUH
NBA Accredited B.Tech Courses, Accorded NACC A-Grade with 3.20 CGPA

DEPARTMENT OF CSE

(ARTIFICIAL INTELLIGENCE AND MACHINE LEARNING)

Lab Manual

Neural Networks Lab
(CM703PC)

IV B. TECH I SEMESTER

Institute Vision:

To be a leading Technical Institute to prepare high quality Engineers to cater the needs of the stakeholders in the field of Engineering and Technology with global competence fundamental comprehensive analytical skills, critical reasoning, research aptitude, entrepreneur skills, ethical values and social concern.

Institute Mission:

Imparting Quality Technical Education to young Engineers by providing the state-of-the-art laboratories, quality instructions by qualified and experienced faculty and research facilities to meet the requirements of stakeholders in real time usage and in training them to excel in competitive examinations for higher education and employment to interface globally emerging techno-informative challenges in the growth corridor of techno-excellence.

Department Vision:

To be an epicenter of excellence in education by offering thrust courses, research and services for the students and make them to succeed in professional **competitive examinations** globally with an attitude of entrepreneurial skills, ethical values and social concern.

Department Mission:

Imparting quality Technical Education to young Computer Engineer by providing them

M1: Impart quality technical Education with State of-the-art laboratories, Analytical and Technical Skills with International standards by qualified and experienced faculty

M2: Prepare for competitive examinations for higher studies / Employment

M3: Develop professional attitude, Research aptitude, Critical Reasoning and technical consultancy by providing training in cutting edge technologies.

M4: Endorse and Nurture knowledge, Life-long learning, Entrepreneurial practices, ethical values and social concern

Program Educational Objectives (PEOs):

PEO 1: To prepare the students for successful careers in Computer Science and Engineering and fulfill the need by providing training to excel in competitive examinations for higher education and employment.

PEO 2: To provide students a broad-based curriculum with a firm foundation in Computer Science and Engineering, Applied Mathematics & Sciences. To impart high quality technical skills for designing, modeling, analyzing and critical problem solving with global competence.

PEO 3: To inculcate professional, social, ethical, effective communication skills and entrepreneurial practice among their holistic growth.

PEO 4: To provide Computer Science and Engineering students with an academic environment and members associated with student related to professional bodies for multi-disciplinary approach and for lifelong learning.

PEO 5: To develop research aptitude among the students in order to carry out research in cutting edge technologies, solve real world problems and provide technical consultancy services.

Program Outcomes:

Program Outcomes	Statement
PO1	An ability to apply knowledge of mathematics, science, and engineering and knowledge of Fundamental Principles.
PO2	An ability to Identify, formulate and solve engineering problems.
PO3	An ability to design a system, component, or process to meet desired needs in Computer Science and Engineering within realistic constraints such as economic, environmental, social, political, ethical, health and safety, manufacturability and sustainability, Design and Modeling.
PO4	An ability to design and conduct experiments, as well as to analyze and interpret data, Experimentation & Interpret/Engineering Analysis.
PO5	An ability to use the techniques, skills and modern Computer Science and Engineering tools necessary for system design with embedded engineering

	practice.
PO6	Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.
PO7	The broad education necessary to understand the impact of engineering solutions in a global, economic, environmental, and societal context.
PO8	An understanding of professional and ethical responsibility.
PO9	An ability to function on multidisciplinary teams.
PO10	An ability to communicate effectively.
PO11	Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.
PO12	A recognition of the need for, and an ability to engage in life-long learning.

Program Specific Outcomes:

Program Specific Outcomes	Statement
PSO1	To prepare the students ready for industry usage by providing required training in cutting edge technologies.
PSO2	An Ability to use the core concepts of computing and optimization techniques to develop more efficient and effective computing mechanisms.
PSO3	Prepare the graduates to demonstrate a sense of societal and ethical responsibility In their professional endeavors and will remain informed and involved as full participants in the profession and our society.

ACE Engineering College

An AUTONOMOUS Institution

Ghatkesar, Medchal (Dist), Hyderabad, Telangana State – 501 301

(NBA Accredited B.Tech Courses Accredited NAAC with A Grade 3.20 CGPA)

Phone: 9133308533, 468, website: www.aceec.ac.in

DEPARTMENT OF CSE (ARTIFICIAL INTELLIGENCE & MACHINE LEARNING)

NEURAL NETWORKS LAB

B.Tech .IV Year I Sem.	**L T P C**
CM703PC	**0 0 2 1**

LIST OF EXPERIMENTS:

1. Implementation of Perceptron Model in Python
2. Implementation of Sigmoid Model in Python
3. Implement 2 layer Feed Forward Network(FFN) for non separable 2 class problem in Python and compare the performance with the sigmoid Model
4. Introduction to Frameworks- Pytorch
5. Implement Fully Connected Neural Network(FCNN) for MNIST dataset in Pytorch
6. Analyze vanishing gradient problem by using various activation functions
7. Analyze the performance of FCNN using various Optimization methods
8. Apply dropout regularization technique to enhance the performance of FCNN
9. Implement Convolution Neural Network(CNN) on CIFAR10 data set
10. Implement Recurrent Neural network for text classification

1. Program to implementation Perceptron Model in Python

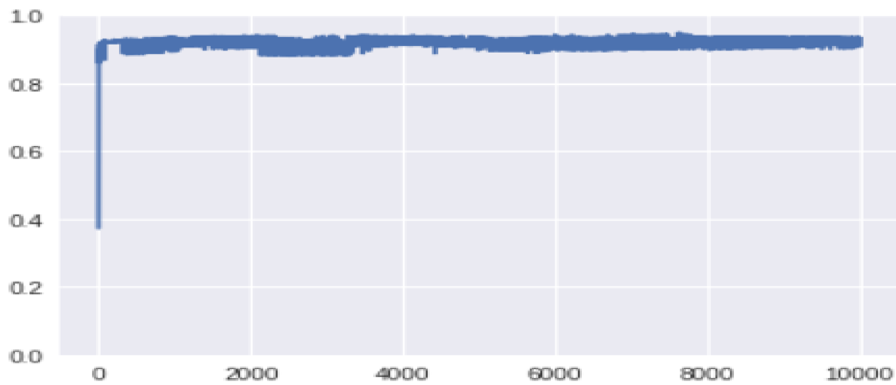
```
import sklearn.datasets
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
 #Perceptron Model
class Perceptron:
  def __init__ (self):
    self.w = None
    self.b = None
  def model(self, x):
    return 1 if (np.dot(self.w, x) >= self.b) else 0
  def predict(self, X):
    Y = []
    for x in X:
      result = self.model(x)
      Y.append(result)
    return np.array(Y)
  def fit(self, X, Y, epochs = 1, lr = 1):
    self.w = np.ones(X.shape[1])
    self.b = 0
    accuracy = {}
    max_accuracy = 0
    wt_matrix = []
    for i in range(epochs):
      for x, y in zip(X, Y):
        y_pred = self.model(x)
        if y == 1 and y_pred == 0:
          self.w = self.w + lr * x
          self.b = self.b - lr * 1
        elif y == 0 and y_pred == 1:
          self.w = self.w - lr * x
          self.b = self.b + lr * 1
      wt_matrix.append(self.w)
      accuracy[i] = accuracy_score(self.predict(X), Y)
      if (accuracy[i] > max_accuracy):
        max_accuracy = accuracy[i]
        chkptw = self.w
        chkptb = self.b
    self.w = chkptw
    self.b = chkptb
    print(max_accuracy)
    plt.plot(accuracy.values())
    plt.ylim([0, 1])
    plt.show()
    return np.array(wt_matrix)
# Loading the dataset and splitting the dataset
breast_cancer = sklearn.datasets.load_breast_cancer()
```

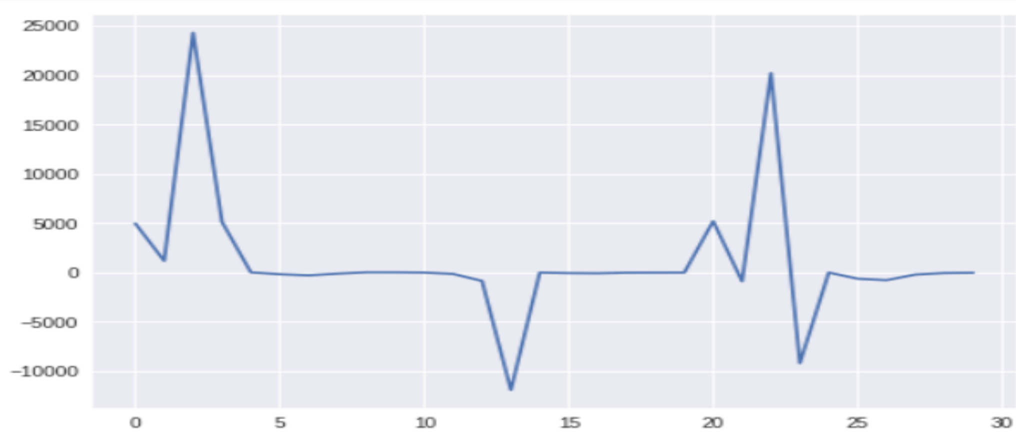
```
X = breast_cancer.data
Y = breast_cancer.target
data = pd.DataFrame(breast_cancer.data, columns=breast_cancer.feature_names)
data['class'] = breast_cancer.target
X = data.drop('class', axis=1)
Y = data['class']
X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0.1, stratify = Y,
                        random_state=1)
X_train = X_train.values
X_test = X_test.values
# Model Training & Prediction
perceptron = Perceptron()
wt_matrix = perceptron.fit(X_train, Y_train, 10000, 0.5)
Y_pred_test = perceptron.predict(X_test)
print(accuracy_score(Y_pred_test, Y_test))
plt.plot(wt_matrix[-1,:])
plt.show()
```

Output:

```
0.9298245614035088
```

```
0.94921875
```





2. Program to implementation Sigmoid Model in Python

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits import mplot3d
import matplotlib.colors
import pandas as pd
from sklearn.preprocessing import StandardScaler, MinMaxScaler
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, mean_squared_error
from tqdm import tqdm_notebook

class SigmoidNeuron:

  def __init__(self):
    self.w = None
    self.b = None

  def perceptron(self, x):
    return np.dot(x, self.w.T) + self.b

  def sigmoid(self, x):
    return 1.0/(1.0 + np.exp(-x))

  def grad_w(self, x, y):
    y_pred = self.sigmoid(self.perceptron(x))
    return (y_pred - y) * y_pred * (1 - y_pred) * x

  def grad_b(self, x, y):
    y_pred = self.sigmoid(self.perceptron(x))
    return (y_pred - y) * y_pred * (1 - y_pred)

  def fit(self, X, Y, epochs=1, learning_rate=1, initialise=True, display_loss=False):
    # initialise w, b
    if initialise:
      self.w = np.random.randn(1, X.shape[1])
      self.b = 0
    if display_loss:
      loss = {}
    for i in tqdm_notebook(range(epochs), total=epochs, unit="epoch"):
      dw = 0
      db = 0
      for x, y in zip(X, Y):
        dw += self.grad_w(x, y)
        db += self.grad_b(x, y)
      self.w -= learning_rate * dw
      self.b -= learning_rate * db
      if display_loss:
        Y_pred = self.sigmoid(self.perceptron(X))
        loss[i] = mean_squared_error(Y_pred, Y)
    if display_loss:
```

```
      plt.plot(loss.values())
      plt.xlabel('Epochs')
      plt.ylabel('Mean Squared Error')
      plt.show()

  def predict(self, X):
    Y_pred = []
    for x in X:
      y_pred = self.sigmoid(self.perceptron(x))
      Y_pred.append(y_pred)
    return np.array(Y_pred)

#Fit for toy data
X = np.asarray([[2.5, 2.5], [4, -1], [1, -4], [-3, 1.25], [-2, -4], [1, 5]])
Y = [1, 1, 1, 0, 0, 0]
sn = SigmoidNeuron()
sn.fit(X, Y, 1, 0.25, True)
```

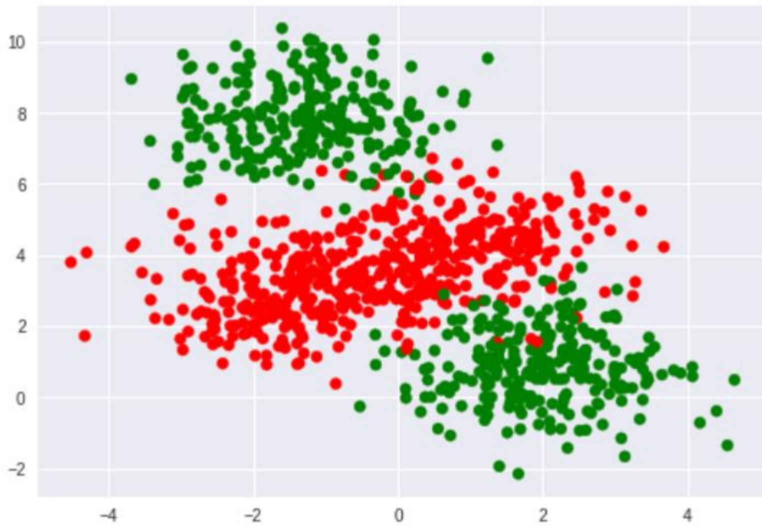
3. Implement 2 layer Feed Forward Network(FFN) for non separable 2 class problem in Python and compare the performance with the sigmoid Model

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.colors
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, mean_squared_error
from tqdm import tqdm_notebook
from sklearn.preprocessing import OneHotEncoder
from sklearn.datasets import make_blobs

class SigmoidNeuron:

  def __init__(self):
    self.w = None
    self.b = None

  def perceptron(self, x):
    return np.dot(x, self.w.T) + self.b

  def sigmoid(self, x):
    return 1.0/(1.0 + np.exp(-x))

  def grad_w_mse(self, x, y):
    y_pred = self.sigmoid(self.perceptron(x))
    return (y_pred - y) * y_pred * (1 - y_pred) * x

  def grad_b_mse(self, x, y):
    y_pred = self.sigmoid(self.perceptron(x))
    return (y_pred - y) * y_pred * (1 - y_pred)

  def grad_w_ce(self, x, y):
    y_pred = self.sigmoid(self.perceptron(x))
    if y == 0:
      return y_pred * x
    elif y == 1:
      return -1 * (1 - y_pred) * x
    else:
      raise ValueError("y should be 0 or 1")

  def grad_b_ce(self, x, y):
    y_pred = self.sigmoid(self.perceptron(x))
    if y == 0:
      return y_pred
    elif y == 1:
      return -1 * (1 - y_pred)
    else:
      raise ValueError("y should be 0 or 1")
```

```
def fit(self, X, Y, epochs=1, learning_rate=1, initialise=True, loss_fn="mse", display_loss=False):

  # initialise w, b
  if initialise:
    self.w = np.random.randn(1, X.shape[1])
    self.b = 0

  if display_loss:
    loss = {}

  for i in tqdm_notebook(range(epochs), total=epochs, unit="epoch"):
    dw = 0
    db = 0
    for x, y in zip(X, Y):
      if loss_fn == "mse":
        dw += self.grad_w_mse(x, y)
        db += self.grad_b_mse(x, y)
      elif loss_fn == "ce":
        dw += self.grad_w_ce(x, y)
        db += self.grad_b_ce(x, y)

    m = X.shape[1]
    self.w -= learning_rate * dw/m
    self.b -= learning_rate * db/m

    if display_loss:
      Y_pred = self.sigmoid(self.perceptron(X))
      if loss_fn == "mse":
        loss[i] = mean_squared_error(Y, Y_pred)
      elif loss_fn == "ce":
        loss[i] = log_loss(Y, Y_pred)

  if display_loss:
    plt.plot(loss.values())
    plt.xlabel('Epochs')
    if loss_fn == "mse":
      plt.ylabel('Mean Squared Error')
    elif loss_fn == "ce":
      plt.ylabel('Log Loss')
    plt.show()

 def predict(self, X):
  Y_pred = []
  for x in X:
    y_pred = self.sigmoid(self.perceptron(x))
    Y_pred.append(y_pred)
  return np.array(Y_pred)

my_cmap = matplotlib.colors.LinearSegmentedColormap.from_list("", ["red","yellow","green"])
```

#Generate data

```
data, labels = make_blobs(n_samples=1000, centers=4, n_features=2, random_state=0)
print(data.shape, labels.shape)
labels_orig = labels
labels = np.mod(labels_orig, 2)
plt.scatter(data[:,0], data[:,1], c=labels, cmap=my_cmap)
plt.show()
X_train, X_val, Y_train, Y_val = train_test_split(data, labels, stratify=labels, random_state=0)
print(X_train.shape, X_val.shape)
```



#SN classification

```
sn = SigmoidNeuron()
sn.fit(X_train, Y_train, epochs=1000, learning_rate=0.5, display_loss=True)
Y_pred_train = sn.predict(X_train)
Y_pred_binarised_train = (Y_pred_train >= 0.5).astype("int").ravel()
Y_pred_val = sn.predict(X_val)
Y_pred_binarised_val = (Y_pred_val >= 0.5).astype("int").ravel()
accuracy_train = accuracy_score(Y_pred_binarised_train, Y_train)
accuracy_val = accuracy_score(Y_pred_binarised_val, Y_val)

print("Training accuracy", round(accuracy_train, 2))
print("Validation accuracy", round(accuracy_val, 2))
plt.scatter(X_train[:,0], X_train[:,1], c=Y_pred_binarised_train, cmap=my_cmap,
s=15*(np.abs(Y_pred_binarised_train-Y_train)+.2))
plt.show()
```

#Our First FF Network

```
class FirstFFNetwork:

  def __init__(self):
    self.w1 = np.random.randn()
    self.w2 = np.random.randn()
    self.w3 = np.random.randn()
    self.w4 = np.random.randn()
    self.w5 = np.random.randn()
    self.w6 = np.random.randn()
```

```
    self.b1 = 0
    self.b2 = 0
    self.b3 = 0

  def sigmoid(self, x):
    return 1.0/(1.0 + np.exp(-x))

  def forward_pass(self, x):
    self.x1, self.x2 = x
    self.a1 = self.w1*self.x1 + self.w2*self.x2 + self.b1
    self.h1 = self.sigmoid(self.a1)
    self.a2 = self.w3*self.x1 + self.w4*self.x2 + self.b2
    self.h2 = self.sigmoid(self.a2)
    self.a3 = self.w5*self.h1 + self.w6*self.h2 + self.b3
    self.h3 = self.sigmoid(self.a3)
    return self.h3

  def grad(self, x, y):
    self.forward_pass(x)

    self.dw5 = (self.h3-y) * self.h3*(1-self.h3) * self.h1
    self.dw6 = (self.h3-y) * self.h3*(1-self.h3) * self.h2
    self.db3 = (self.h3-y) * self.h3*(1-self.h3)

    self.dw1 = (self.h3-y) * self.h3*(1-self.h3) * self.w5 * self.h1*(1-self.h1) * self.x1
    self.dw2 = (self.h3-y) * self.h3*(1-self.h3) * self.w5 * self.h1*(1-self.h1) * self.x2
    self.db1 = (self.h3-y) * self.h3*(1-self.h3) * self.w5 * self.h1*(1-self.h1)

    self.dw3 = (self.h3-y) * self.h3*(1-self.h3) * self.w6 * self.h2*(1-self.h2) * self.x1
    self.dw4 = (self.h3-y) * self.h3*(1-self.h3) * self.w6 * self.h2*(1-self.h2) * self.x2
    self.db2 = (self.h3-y) * self.h3*(1-self.h3) * self.w6 * self.h2*(1-self.h2)


  def fit(self, X, Y, epochs=1, learning_rate=1, initialise=True, display_loss=False):

    # initialise w, b
    if initialise:
      self.w1 = np.random.randn()
      self.w2 = np.random.randn()
      self.w3 = np.random.randn()
      self.w4 = np.random.randn()
      self.w5 = np.random.randn()
      self.w6 = np.random.randn()
      self.b1 = 0
      self.b2 = 0
      self.b3 = 0

    if display_loss:
      loss = {}

    for i in tqdm_notebook(range(epochs), total=epochs, unit="epoch"):
      dw1, dw2, dw3, dw4, dw5, dw6, db1, db2, db3 = [0]*9
```

```
    for x, y in zip(X, Y):
      self.grad(x, y)
      dw1 += self.dw1
      dw2 += self.dw2
      dw3 += self.dw3
      dw4 += self.dw4
      dw5 += self.dw5
      dw6 += self.dw6
      db1 += self.db1
      db2 += self.db2
      db3 += self.db3

    m = X.shape[1]
    self.w1 -= learning_rate * dw1 / m
    self.w2 -= learning_rate * dw2 / m
    self.w3 -= learning_rate * dw3 / m
    self.w4 -= learning_rate * dw4 / m
    self.w5 -= learning_rate * dw5 / m
    self.w6 -= learning_rate * dw6 / m
    self.b1 -= learning_rate * db1 / m
    self.b2 -= learning_rate * db2 / m
    self.b3 -= learning_rate * db3 / m

    if display_loss:
      Y_pred = self.predict(X)
      loss[i] = mean_squared_error(Y_pred, Y)

  if display_loss:
    plt.plot(loss.values())
    plt.xlabel('Epochs')
    plt.ylabel('Mean Squared Error')
    plt.show()

 def predict(self, X):
  Y_pred = []
  for x in X:
    y_pred = self.forward_pass(x)
    Y_pred.append(y_pred)
  return np.array(Y_pred)

ffn = FirstFFNetwork()
ffn.fit(X_train, Y_train, epochs=2000, learning_rate=.01, display_loss=True)
Y_pred_train = ffn.predict(X_train)
Y_pred_binarised_train = (Y_pred_train >= 0.5).astype("int").ravel()
Y_pred_val = ffn.predict(X_val)
Y_pred_binarised_val = (Y_pred_val >= 0.5).astype("int").ravel()
accuracy_train = accuracy_score(Y_pred_binarised_train, Y_train)
accuracy_val = accuracy_score(Y_pred_binarised_val, Y_val)
plt.scatter(X_train[:,0], X_train[:,1], c=Y_pred_binarised_train, cmap=my_cmap,
s=15*(np.abs(Y_pred_binarised_train-Y_train)+.2))
plt.show()
```

Output:

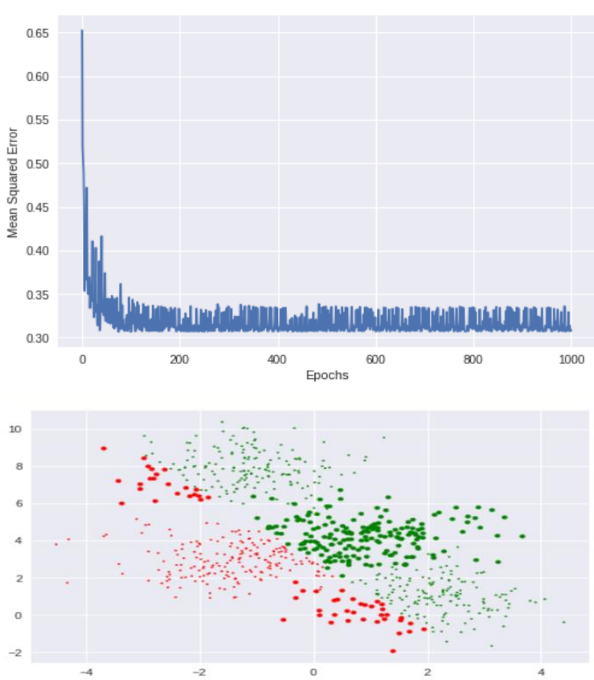


Fig:1 Sigmoid Neuron O/P

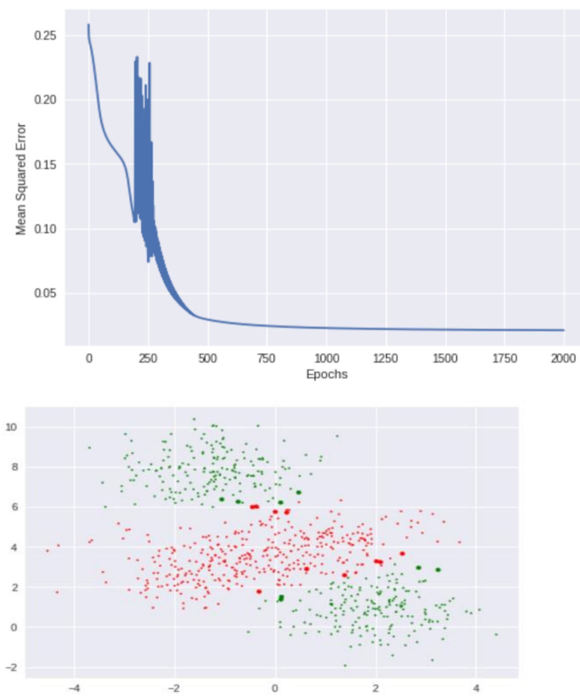


Fig:2 Feed Forward Neuron O/P

4. Introduction to Frameworks- Pytorch

```
import torch
import numpy as np
import matplotlib.pyplot as plt
```

#Initialise tensors

```
x = torch.ones(3, 2)
print(x)
x = torch.zeros(3, 2)
print(x)
x = torch.rand(3, 2)
print(x)
```

Output

```
tensor([[1., 1.],
        [1., 1.],
        [1., 1.]])
tensor([[0., 0.],
        [0., 0.],
        [0., 0.]])
tensor([[0.3102, 0.8892],
        [0.1277, 0.3445],
        [0.0322, 0.5172]])
```

```
x = torch.empty(3, 2)
print(x)
y = torch.zeros_like(x)
print(y)
```

Output:

```
tensor([[4.4765e-35, 0.0000e+00],
        [1.2773e-01, 3.4447e-01],
        [8.9683e-44, 0.0000e+00]])
tensor([[0., 0.],
        [0., 0.],
        [0., 0.]])
```

```
x = torch.linspace(0, 1, steps=5)
print(x)
tensor([0.0000, 0.2500, 0.5000, 0.7500, 1.0000])
x = torch.tensor([[1, 2],
                  [3, 4],
                  [5, 6]])
print(x)
```

Output:

```
tensor([[1, 2],
        [3, 4],
        [5, 6]])
```

```
## Slicing tensors
```

```
print(x.size())
print(x[:, 1])
print(x[0, :])
```
Output:
```
torch.Size([3, 2])
tensor([2, 4, 6])
tensor([1, 2])
```

```
y = x[1, 1]
print(y)
print(y.item())
```
Output:
```
tensor(4)
4
```

```
## Reshaping tensors
print(x)
y = x.view(2, 3)
print(y)
```
Output:
```
tensor([[1, 2],
        [3, 4],
        [5, 6]])
tensor([[1, 2, 3],
        [4, 5, 6]])
```

```
y = x.view(6,-1)
print(y)
```
Output:
```
tensor([[1],
        [2],
        [3],
        [4],
        [5],
        [6]])
```

#Simple Tensor Operations

```
x = torch.ones([3, 2])
y = torch.ones([3, 2])
z = x + y
print(z)
z = x - y
print(z)
z = x * y
print(z)
```
Output:
```
tensor([[2., 2.],
        [2., 2.],
        [2., 2.]])
tensor([[0., 0.],
        [0., 0.],
```

```
       [0., 0.]])
tensor([[1., 1.],
       [1., 1.],
       [1., 1.]])


z = y.add(x)
print(z)
print(y)
```

Output:

```
tensor([[2., 2.],
       [2., 2.],
       [2., 2.]])
tensor([[1., 1.],
       [1., 1.],
       [1., 1.]])


z = y.add_(x)
print(z)
print(y)
```

Output:

```
tensor([[2., 2.],
       [2., 2.],
       [2., 2.]])
tensor([[2., 2.],
       [2., 2.],
       [2., 2.]])
```

```
#CUDA support

print(torch.cuda.device_count())
print(torch.cuda.device(0))
print(torch.cuda.get_device_name(0))
```

Output:

```
<torch.cuda.device object at 0x7f785e6ec2b0>
Tesla T4


cuda0 = torch.device('cuda:0')
a = torch.ones(3, 2, device=cuda0)
b = torch.ones(3, 2, device=cuda0)
c = a + b
print(c)
print(a)
```

Output:

```
tensor([[2., 2.],
       [2., 2.],
       [2., 2.]], device='cuda:0')

tensor([[1., 1.],
```

```
[1., 1.],
[1., 1.]], device='cuda:0')
```

```
%%time
for i in range(10):
  a = np.random.randn(10000,10000)
  b = np.random.randn(10000,10000)
  np.add(b, a)
```

Output:

CPU times: user 54.7 s, sys: 4.26 s, total: 59 s
Wall time: 58.9 s

```
%%time
for i in range(10):
  a_cpu = torch.randn([10000, 10000])
  b_cpu = torch.randn([10000, 10000])
  b_cpu.add_(a_cpu)
```

Output:

CPU times: user 1.88 ms, sys: 2.87 ms, total: 4.75 ms
Wall time: 29.6 ms

```
%%time
for i in range(10):
  a = torch.randn([10000, 10000], device=cuda0)
  b = torch.randn([10000, 10000], device=cuda0)
  b.add_(a)
```

Output:

CPU times: user 1.88 ms, sys: 2.87 ms, total: 4.75 ms
Wall time: 29.6 ms

```
%%time
for i in range(10):
  a = np.random.randn(10000,10000)
  b = np.random.randn(10000,10000)
  np.matmul(b, a)
```

```
%%time
for i in range(10):
  a = torch.randn([10000, 10000], device=cuda0)
  b = torch.randn([10000, 10000], device=cuda0)
  torch.matmul(a, b)
```

Output:

CPU times: user 24.1 ms, sys: 18.9 ms, total: 42.9 ms

Wall time: 154 ms

#Autodiff

```
x = torch.ones([3, 2], requires_grad=True)
print(x)
```
Output:
```
tensor([[1., 1.],
        [1., 1.],
        [1., 1.]], requires_grad=True)
```

```
y = x + 5
print(y)
```
Output:
```
tensor([[6., 6.],
        [6., 6.],
        [6., 6.]], grad_fn=<AddBackward0>)
```

```
z = y*y + 1
print(z)
```
Output:
```
tensor([[37., 37.],
        [37., 37.],
        [37., 37.]], grad_fn=<AddBackward0>)
```

```
t = torch.sum(z)
print(t)
```
Output:
```
tensor(222., grad_fn=<SumBackward0>)
```

```
t.backward()
print(x.grad)
```
Output:
```
tensor([[12., 12.],
        [12., 12.],
        [12., 12.]])
```

$$t = \sum_i z_i, z_i = y_i^2 + 1, y_i = x_i + 5$$

$$\frac{\partial t}{\partial x_i} = \frac{\partial z_i}{\partial x_i} = \frac{\partial z_i}{\partial y_i}\frac{\partial y_i}{\partial x_i} = 2y_i \times 1$$

At x = 1, y = 6, $\frac{\partial t}{\partial x_i} = 12$

```
x = torch.ones([3, 2], requires_grad=True)
y = x + 5
r = 1/(1 + torch.exp(-y))
print(r)
s = torch.sum(r)
s.backward()
print(x.grad)
```

Output:

```
tensor([[0.9975, 0.9975],
        [0.9975, 0.9975],
        [0.9975, 0.9975]], grad_fn=<MulBackward0>)
tensor([[0.0025, 0.0025],
        [0.0025, 0.0025],
        [0.0025, 0.0025]])
```

```
x = torch.ones([3, 2], requires_grad=True)
y = x + 5
r = 1/(1 + torch.exp(-y))
a = torch.ones([3, 2])
r.backward(a)
print(x.grad)
```

Output:

```
tensor([[0.0025, 0.0025],
        [0.0025, 0.0025],
        [0.0025, 0.0025]])
```

```
%%time
learning_rate = 0.001
N = 10000000
epochs = 200

w = torch.rand([N], requires_grad=True)
b = torch.ones([1], requires_grad=True)

# print(torch.mean(w).item(), b.item())

for i in range(epochs):

  x = torch.randn([N])
  y = torch.dot(3*torch.ones([N]), x) - 2

  y_hat = torch.dot(w, x) + b
```

```
  loss = torch.sum((y_hat - y)**2)

  loss.backward()

  with torch.no_grad():
    w -= learning_rate * w.grad
    b -= learning_rate * b.grad

    w.grad.zero_()
    b.grad.zero_()

#   print(torch.mean(w).item(), b.item())
```

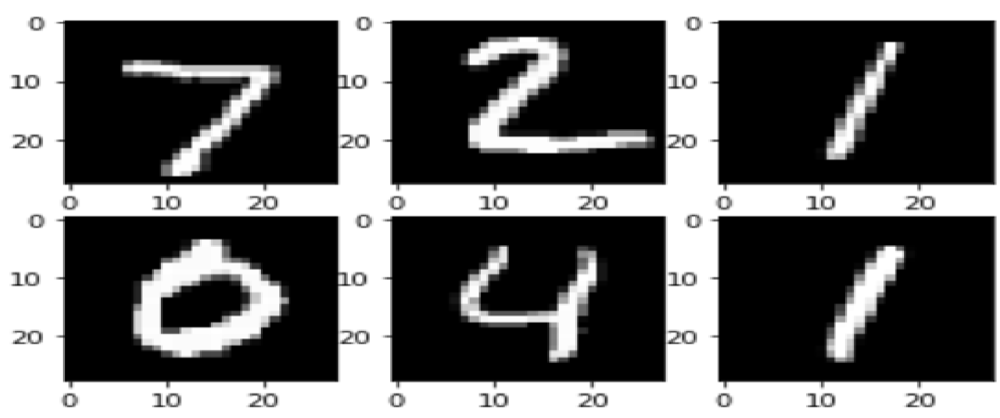
Output:
CPU times: user 36.7 s, sys: 443 ms, total: 37.2 s
Wall time: 37.2 s

```
%%time
learning_rate = 0.001
N = 10000000
epochs = 200

w = torch.rand([N], requires_grad=True, device=cuda0)
b = torch.ones([1], requires_grad=True, device=cuda0)

# print(torch.mean(w).item(), b.item())

for i in range(epochs):

  x = torch.randn([N], device=cuda0)
  y = torch.dot(3*torch.ones([N], device=cuda0), x) - 2

  y_hat = torch.dot(w, x) + b
  loss = torch.sum((y_hat - y)**2)

  loss.backward()

  with torch.no_grad():
    w -= learning_rate * w.grad
    b -= learning_rate * b.grad

    w.grad.zero_()
    b.grad.zero_()

  #print(torch.mean(w).item(), b.item())
```
Output:
CPU times: user 467 ms, sys: 305 ms, total: 772 ms
Wall time: 784 ms

5. Implementation of FCNN in Pytorch on MNIST data

```
import torch
import torch.nn as nn
import torchvision
import torchvision.transforms as transforms
import matplotlib.pyplot as plt
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
input_size = 784 # 28x28
hidden_size = 500
num_classes = 10
num_epochs = 2
batch_size = 100
learning_rate = 0.001
# Import MNIST dataset
train_dataset = torchvision.datasets.MNIST(root='./data',
                                           train=True,
                                           transform=transforms.ToTensor(),
                                           download=True)

test_dataset = torchvision.datasets.MNIST(root='./data',
                                          train=False,
                                          transform=transforms.ToTensor())
# Data loader
train_loader = torch.utils.data.DataLoader(dataset=train_dataset,
                                           batch_size=batch_size,
                                           shuffle=True)

test_loader = torch.utils.data.DataLoader(dataset=test_dataset,
                                          batch_size=batch_size,
                                          shuffle=False)
# Fully connected neural network with one hidden layer
class NeuralNet(nn.Module):
    def __init__(self, input_size, hidden_size, num_classes):
        super(NeuralNet, self).__init__()
        self.input_size = input_size
        self.l1 = nn.Linear(input_size, hidden_size)
        self.relu = nn.ReLU()
        self.l2 = nn.Linear(hidden_size, num_classes)

    def forward(self, x):
        out = self.l1(x)
        out = self.relu(out)
        out = self.l2(out)
        # no activation and no softmax at the end
        return out

model = NeuralNet(input_size, hidden_size, num_classes).to(device)
# Loss and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)
# Train the model
n_total_steps = len(train_loader)
```

```
for epoch in range(num_epochs):
    for i, (images, labels) in enumerate(train_loader):
        # origin shape: [100, 1, 28, 28]
        # resized: [100, 784]
        images = images.reshape(-1, 28*28).to(device)
        labels = labels.to(device)

        # Forward pass
        outputs = model(images)
        loss = criterion(outputs, labels)

        # Backward and optimize
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        if (i+1) % 100 == 0:
            print (f'Epoch [{epoch+1}/{num_epochs}], Step [{i+1}/{n_total_steps}], Loss:
{loss.item():.4f}')
  # Test the model
# In test phase, we don't need to compute gradients (for memory efficiency)
with torch.no_grad():
    n_correct = 0
    n_samples = 0
    for images, labels in test_loader:
        images = images.reshape(-1, 28*28).to(device)
        labels = labels.to(device)
        outputs = model(images)
        # max returns (value ,index)
        _, predicted = torch.max(outputs.data, 1)
        n_samples += labels.size(0)
        n_correct += (predicted == labels).sum().item()

    acc = 100.0 * n_correct / n_samples
    print(f'Accuracy of the network on the 10000 test images: {acc} %')
```



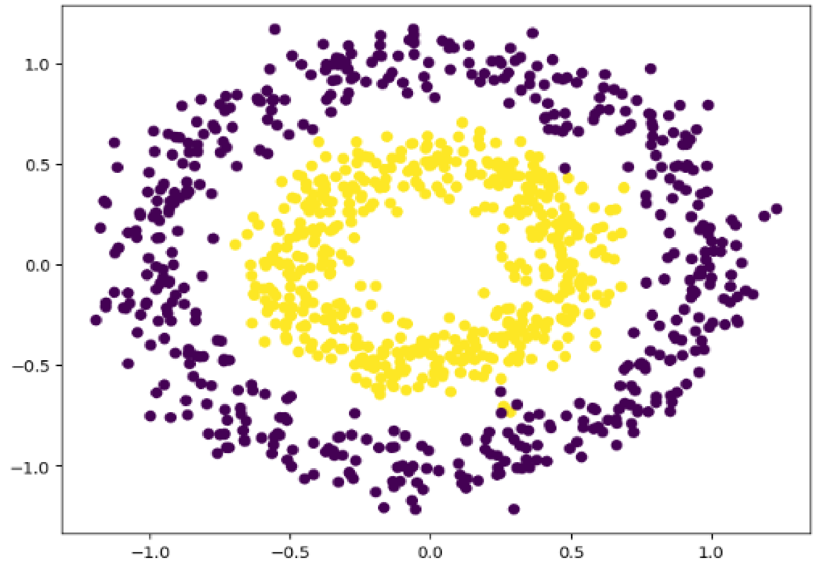
Output:
Accuracy of the network on the 10000 test images: 97.22 %

6. Analyze vanishing gradient problem by using various activation functions

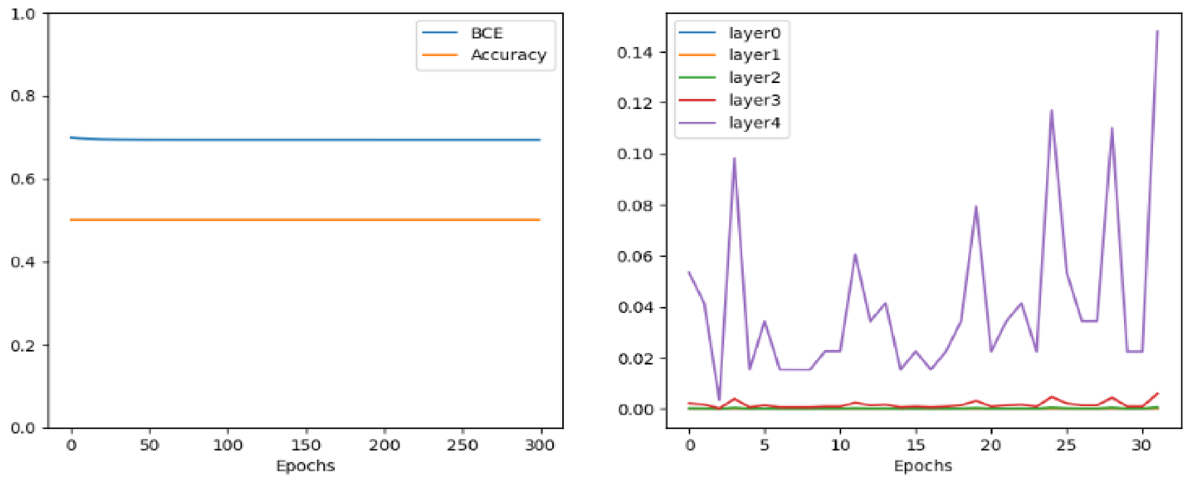
```
from sklearn.datasets import make_circles
import matplotlib.pyplot as plt
import torch
import torch.nn as nn
import torch.optim as optim

# Make data: Two circles on x-y plane as a classification problem
X, y = make_circles(n_samples=1000, factor=0.5, noise=0.1)
X = torch.tensor(X, dtype=torch.float32)
y = torch.tensor(y.reshape(-1, 1), dtype=torch.float32)
plt.figure(figsize=(8,6))
plt.scatter(X[:,0], X[:,1], c=y)
plt.show()

# Binary classification model
class Model(nn.Module):
    def __init__(self, activation=nn.ReLU):
        super().__init__()
        self.layer0 = nn.Linear(2,5)
        self.act0 = activation()
        self.layer1 = nn.Linear(5,5)
        self.act1 = activation()
        self.layer2 = nn.Linear(5,5)
        self.act2 = activation()
        self.layer3 = nn.Linear(5,5)
        self.act3 = activation()
        self.layer4 = nn.Linear(5,1)
        self.act4 = nn.Sigmoid()

    def forward(self, x):
        x = self.act0(self.layer0(x))
        x = self.act1(self.layer1(x))
        x = self.act2(self.layer2(x))
        x = self.act3(self.layer3(x))
        x = self.act4(self.layer4(x))
        return x

# train the model and produce history
def train_loop(model, X, y, n_epochs=300, batch_size=32):
    loss_fn = nn.BCELoss()
    optimizer = optim.Adam(model.parameters(), lr=0.0001)
    batch_start = torch.arange(0, len(X), batch_size)

    bce_hist = []
    acc_hist = []
    grad_hist = [[],[],[],[],[]]

    for epoch in range(n_epochs):
        # train model with optimizer
        model.train()
```

```
        layer_grad = [[],[],[],[],[]]
        for start in batch_start:
            X_batch = X[start:start+batch_size]
            y_batch = y[start:start+batch_size]
            y_pred = model(X_batch)
            loss = loss_fn(y_pred, y_batch)
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()
            # collect mean absolute value of gradients
            layers = [model.layer0, model.layer1, model.layer2, model.layer3, model.layer4]
            for n,layer in enumerate(layers):
                mean_grad = float(layer.weight.grad.abs().mean())
                layer_grad[n].append(mean_grad)
        # evaluate BCE and accuracy at end of each epoch
        model.eval()
        with torch.no_grad():
            y_pred = model(X)
            bce = float(loss_fn(y_pred, y))
            acc = float((y_pred.round() == y).float().mean())
        bce_hist.append(bce)
        acc_hist.append(acc)
        for n, grads in enumerate(layer_grad):
            grad_hist[n].append(sum(grads)/len(grads))
        # print metrics every 10 epochs
        if epoch % 10 == 9:
            print("Epoch %d: BCE=%.4f, Accuracy=%.2f%%" % (epoch, bce, acc*100))
    return bce_hist, acc_hist, layer_grad

# pick different activation functions and compare the result visually
for activation in [nn.Sigmoid, nn.Tanh, nn.ReLU, nn.ReLU6, nn.LeakyReLU]:
    model = Model(activation=activation)
    bce_hist, acc_hist, grad_hist = train_loop(model, X, y)

    fig, ax = plt.subplots(1, 2, figsize=(12, 5))
    ax[0].plot(bce_hist, label="BCE")
    ax[0].plot(acc_hist, label="Accuracy")
    ax[0].set_xlabel("Epochs")
    ax[0].set_ylim(0, 1)
    for n, grads in enumerate(grad_hist):
        ax[1].plot(grads, label="layer"+str(n))
    ax[1].set_xlabel("Epochs")
    fig.suptitle(str(activation))
    ax[0].legend()
    ax[1].legend()
    plt.show()
```

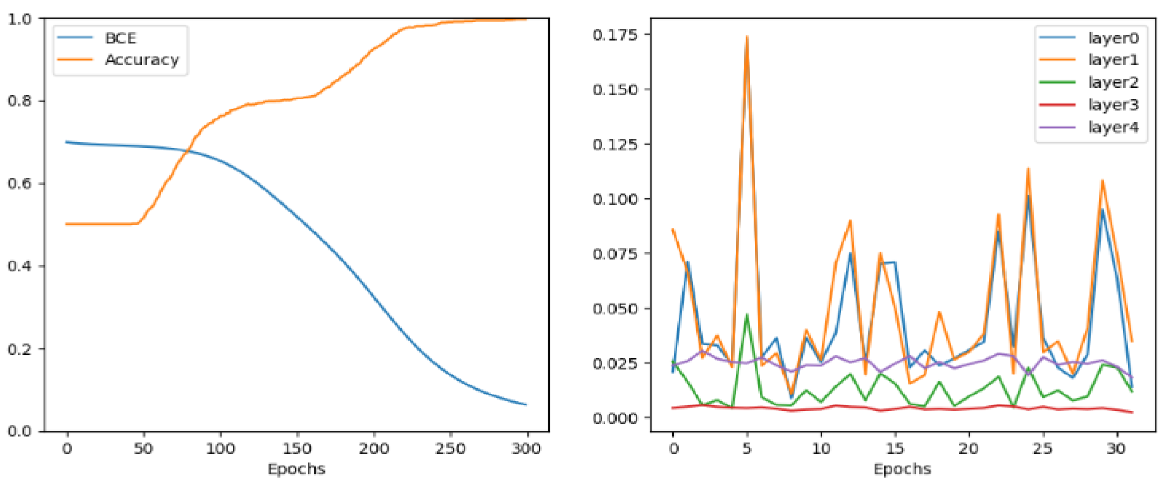
Input & Output:



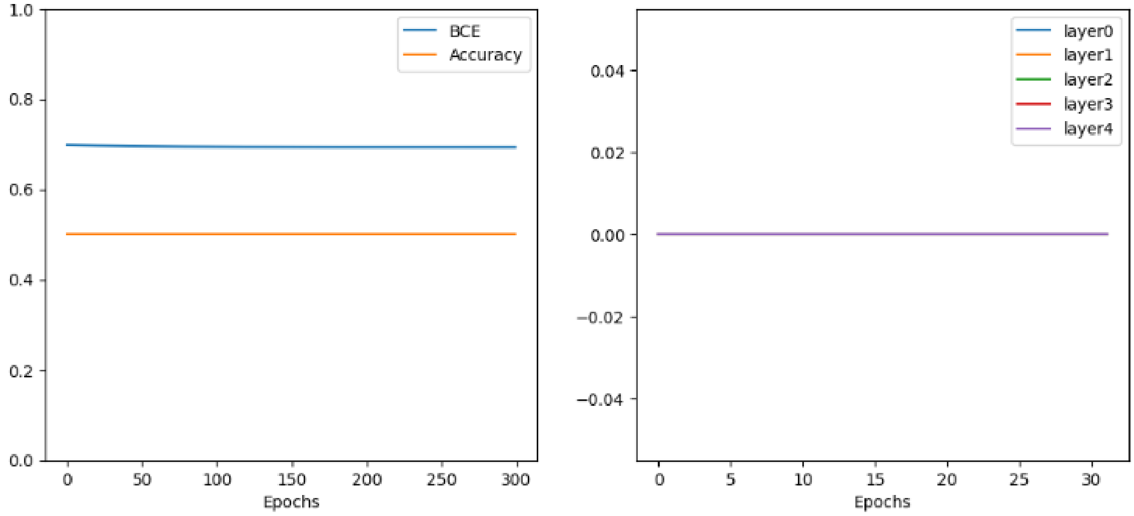
<class 'torch.nn.modules.activation.Sigmoid'>



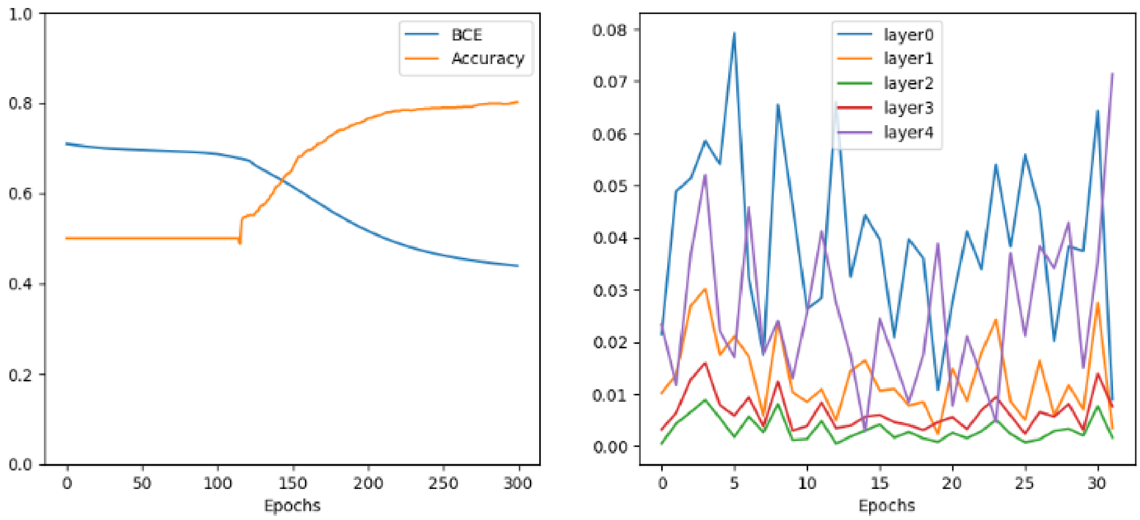
<class 'torch.nn.modules.activation.Tanh'>



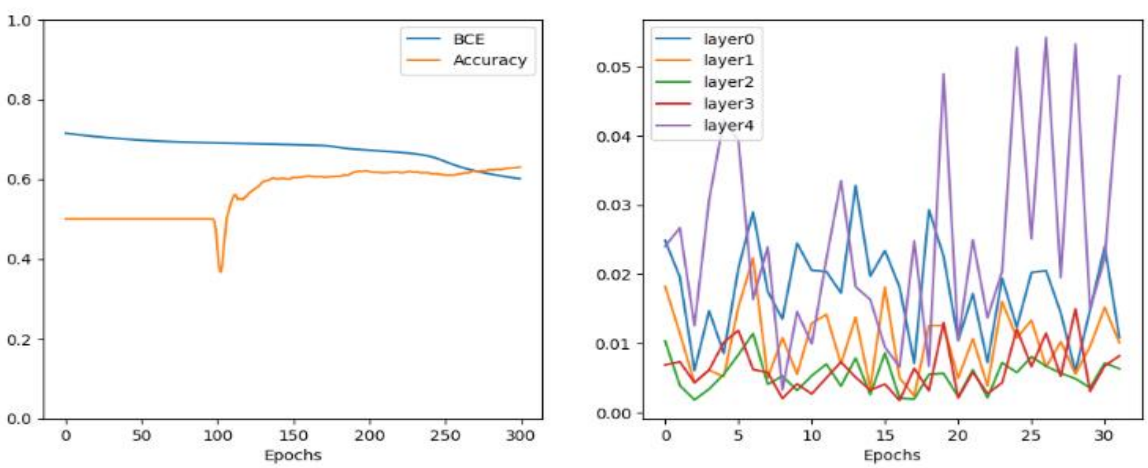
<class 'torch.nn.modules.activation.ReLU'>



<class 'torch.nn.modules.activation.ReLU6'>



<class 'torch.nn.modules.activation.LeakyReLU'>



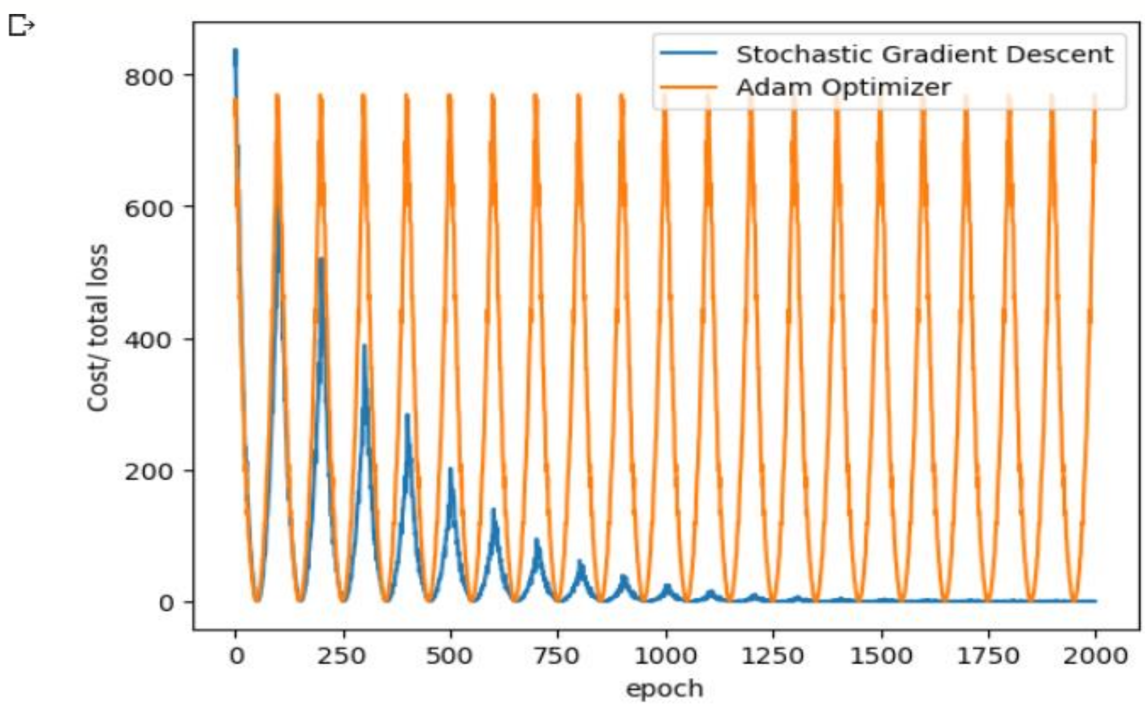
7.

7. Analyze the performance of FCNN using various Optimization methods

```
import matplotlib.pyplot as plt
import numpy as np
import torch
from torch.utils.data import Dataset, DataLoader

# Creating our dataset class
class Build_Data(Dataset):
    # Constructor
    def __init__(self):
        self.x = torch.arange(-5, 5, 0.1).view(-1, 1)
        self.func = -5 * self.x + 1
        self.y = self.func + 0.4 * torch.randn(self.x.size())
        self.len = self.x.shape[0]
    # Getting the data
    def __getitem__(self, index):
        return self.x[index], self.y[index]
    # Getting length of the data
    def __len__(self):
        return self.len

# Create dataset object
data_set = Build_Data()

model = torch.nn.Linear(1, 1)
criterion = torch.nn.MSELoss()

# Creating Dataloader object
trainloader = DataLoader(dataset = data_set, batch_size=1)

# define optimizer
optimizer = torch.optim.Adam(model.parameters(), lr=0.01)

loss_SGD = []
n_iter = 20

for i in range(n_iter):
    for x, y in trainloader:
        # making a prediction in forward pass
        y_hat = model(x)
        # calculating the loss between original and predicted data points
        loss = criterion(y_hat, y)
        # store loss into list
        loss_SGD.append(loss.item())
        # zeroing gradients after each iteration
        optimizer.zero_grad()
        # backward pass for computing the gradients of the loss w.r.t to learnable parameters
        loss.backward()
        # updating the parameters after each iteration
        optimizer.step()
```

```
model = torch.nn.Linear(1, 1)
loss_Adam = []
for i in range(n_iter):
    for x, y in trainloader:
        # making a prediction in forward pass
        y_hat = model(x)
        # calculating the loss between original and predicted data points
        loss = criterion(y_hat, y)
        # store loss into list
        loss_Adam.append(loss.item())
        # zeroing gradients after each iteration
        optimizer.zero_grad()
        # backward pass for computing the gradients of the loss w.r.t to learnable parameters
        loss.backward()
        # updating the parameters after each iteration
        optimizer.step()

plt.plot(loss_SGD,label = "Stochastic Gradient Descent")
plt.plot(loss_Adam,label = "Adam Optimizer")
plt.xlabel('epoch')
plt.ylabel('Cost/ total loss')
plt.legend()
plt.show()
```

Output:



8. Apply various regularization techniques to enhance the performance of FCNN

Using Dropouton the Hidden layers

```
import numpy as np
import pandas as pd
import torch
import torch.nn as nn
import torch.optim as optim
from sklearn.preprocessing import LabelEncoder
from sklearn.model_selection import StratifiedKFold

# Read data
data = pd.read_csv("sonar.csv", header=None)
X = data.iloc[:, 0:60]
y = data.iloc[:, 60]

# Label encode the target from string to integer
encoder = LabelEncoder()
encoder.fit(y)
y = encoder.transform(y)

# Convert to 2D PyTorch tensors
X = torch.tensor(X.values, dtype=torch.float32)
y = torch.tensor(y, dtype=torch.float32).reshape(-1, 1)

# Define PyTorch model, with dropout at hidden layers
class SonarModel(nn.Module):
    def __init__(self):
        super().__init__()
        self.layer1 = nn.Linear(60, 60)
        self.act1 = nn.ReLU()
        self.dropout1 = nn.Dropout(0.2)
        self.layer2 = nn.Linear(60, 30)
        self.act2 = nn.ReLU()
        self.dropout2 = nn.Dropout(0.2)
        self.output = nn.Linear(30, 1)
        self.sigmoid = nn.Sigmoid()

    def forward(self, x):
        x = self.act1(self.layer1(x))
        x = self.dropout1(x)
        x = self.act2(self.layer2(x))
        x = self.dropout2(x)
        x = self.sigmoid(self.output(x))
        return x

# Helper function to train the model and return the validation result
def model_train(model, X_train, y_train, X_val, y_val,
                n_epochs=300, batch_size=16):
    loss_fn = nn.BCELoss()
    optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.8)
```

```
    batch_start = torch.arange(0, len(X_train), batch_size)

    model.train()
    for epoch in range(n_epochs):
        for start in batch_start:
            X_batch = X_train[start:start+batch_size]
            y_batch = y_train[start:start+batch_size]
            y_pred = model(X_batch)
            loss = loss_fn(y_pred, y_batch)
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()

    # evaluate accuracy after training
    model.eval()
    y_pred = model(X_val)
    acc = (y_pred.round() == y_val).float().mean()
    acc = float(acc)
    return acc

# run 10-fold cross validation
kfold = StratifiedKFold(n_splits=10, shuffle=True)
accuracies = []
for train, test in kfold.split(X, y):
    # create model, train, and get accuracy
    model = SonarModel()
    acc = model_train(model, X[train], y[train], X[test], y[test])
    print("Accuracy: %.2f" % acc)
    accuracies.append(acc)

# evaluate the model
mean = np.mean(accuracies)
std = np.std(accuracies)
print("Baseline: %.2f%% (+/- %.2f%%)" % (mean*100, std*100))
import numpy as np
import pandas as pd
import torch
import torch.nn as nn
import torch.optim as optim
from sklearn.preprocessing import LabelEncoder
from sklearn.model_selection import StratifiedKFold

# Read data
data = pd.read_csv("sonar.csv", header=None)
X = data.iloc[:, 0:60]
y = data.iloc[:, 60]

# Label encode the target from string to integer
encoder = LabelEncoder()
encoder.fit(y)
y = encoder.transform(y)
```

```
# Convert to 2D PyTorch tensors
X = torch.tensor(X.values, dtype=torch.float32)
y = torch.tensor(y, dtype=torch.float32).reshape(-1, 1)

# Define PyTorch model
class SonarModel(nn.Module):
    def __init__(self):
        super().__init__()
        self.layer1 = nn.Linear(60, 60)
        self.act1 = nn.ReLU()
        self.layer2 = nn.Linear(60, 30)
        self.act2 = nn.ReLU()
        self.output = nn.Linear(30, 1)
        self.sigmoid = nn.Sigmoid()

    def forward(self, x):
        x = self.act1(self.layer1(x))
        x = self.act2(self.layer2(x))
        x = self.sigmoid(self.output(x))
        return x

# Helper function to train the model and return the validation result
def model_train(model, X_train, y_train, X_val, y_val,
                n_epochs=300, batch_size=16):
    loss_fn = nn.BCELoss()
    optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.8)
    batch_start = torch.arange(0, len(X_train), batch_size)

    model.train()
    for epoch in range(n_epochs):
        for start in batch_start:
            X_batch = X_train[start:start+batch_size]
            y_batch = y_train[start:start+batch_size]
            y_pred = model(X_batch)
            loss = loss_fn(y_pred, y_batch)
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()

    # evaluate accuracy after training
    model.eval()
    y_pred = model(X_val)
    acc = (y_pred.round() == y_val).float().mean()
    acc = float(acc)
    return acc

# run 10-fold cross validation
kfold = StratifiedKFold(n_splits=10, shuffle=True)
accuracies = []
for train, test in kfold.split(X, y):
    # create model, train, and get accuracy
    model = SonarModel()
```

```
    acc = model_train(model, X[train], y[train], X[test], y[test])
    print("Accuracy: %.2f" % acc)
    accuracies.append(acc)

# evaluate the model
mean = np.mean(accuracies)
std = np.std(accuracies)
print("Baseline: %.2f%% (+/- %.2f%%)" % (mean*100, std*100))
```

Output: Using Dropout

```
Accuracy: 0.71
Accuracy: 1.00
Accuracy: 0.71
Accuracy: 0.90
Accuracy: 0.86
Accuracy: 0.81
Accuracy: 0.90
Accuracy: 0.86
Accuracy: 0.80
Accuracy: 0.90
Baseline: 84.62% (+/- 8.48%)
```

Output: Without Dropout

```
Accuracy: 0.90
Accuracy: 0.81
Accuracy: 0.81
Accuracy: 0.76
Accuracy: 0.81
Accuracy: 0.90
Accuracy: 0.95
Accuracy: 0.90
Accuracy: 0.65
Accuracy: 0.80
Baseline: 83.07% (+/- 8.42%)
```

9. Implement Convolution Neural Network(CNN) on CIFAR10 data set

Dataset Link:

http://archive.ics.uci.edu/ml/datasets/Connectionist+Bench+(Sonar,+Mines+vs.+Rocks)

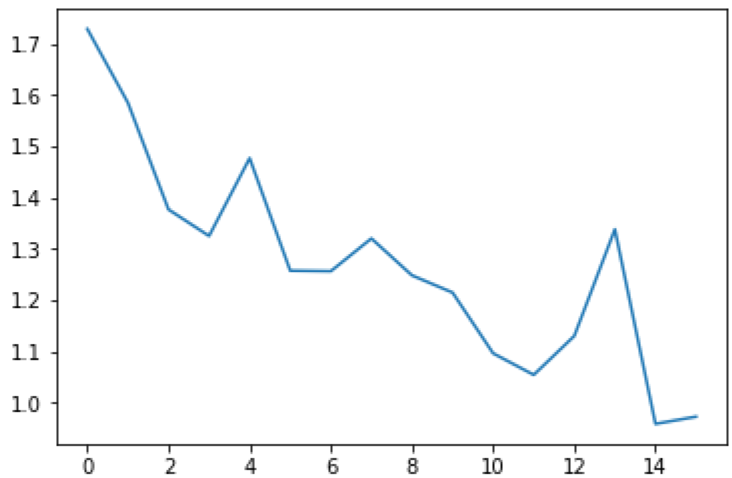
```
import torch
import torch.optim as optim
import matplotlib.pyplot as plt
import torch.nn as nn
import numpy as np
import torchvision
import torchvision.transforms as transforms

#download CIFAR data set
trainset = torchvision.datasets.CIFAR10(root='./data', train=True,
                                        download=True,
                                        transform=transforms.ToTensor())
classes = ('plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship',
'truck')
trainloader = torch.utils.data.DataLoader(trainset, batch_size=4, shuffle=True)
dataiter = iter(trainloader)
images, labels = next(dataiter)
print(images.shape)
print(images[1].shape)
print(labels[1].item())

# Visualize the data
def imshow(img):
    npimg = img.numpy()
    plt.imshow(np.transpose(npimg, (1, 2, 0)))
    plt.show()
imshow(torchvision.utils.make_grid(images))
print(' '.join(classes[labels[j]] for j in range(4)))


#Single Convolutional Layer

class LeNet(nn.Module):
    def __init__(self):
        super(LeNet, self).__init__()
        self.cnn_model = nn.Sequential(
            nn.Conv2d(3, 6, 5),         # (N, 3, 32, 32) -> (N,  6, 28, 28)
            nn.Tanh(),
            nn.AvgPool2d(2, stride=2),  # (N, 6, 28, 28) -> (N,  6, 14, 14)
            nn.Conv2d(6, 16, 5),        # (N, 6, 14, 14) -> (N, 16, 10, 10)
            nn.Tanh(),
            nn.AvgPool2d(2, stride=2)   # (N,16, 10, 10) -> (N, 16, 5, 5)
        )
        self.fc_model = nn.Sequential(
            nn.Linear(400,120),         # (N, 400) -> (N, 120)
            nn.Tanh(),
```

```
        nn.Linear(120,84),        # (N, 120) -> (N, 84)
        nn.Tanh(),
        nn.Linear(84,10)          # (N, 84)  -> (N, 10)
    )

  def forward(self, x):
    x = self.cnn_model(x)
    x = x.view(x.size(0), -1)
    x = self.fc_model(x)
    return x

batch_size = 128
trainset = torchvision.datasets.CIFAR10(root='./data', train=True, download=True,
transform=transforms.ToTensor())
trainloader = torch.utils.data.DataLoader(trainset, batch_size=batch_size, shuffle=True)
testset = torchvision.datasets.CIFAR10(root='./data', train=False, download=True,
transform=transforms.ToTensor())
testloader = torch.utils.data.DataLoader(testset, batch_size=batch_size, shuffle=False)

def evaluation(dataloader):
  total, correct = 0, 0
  for data in dataloader:
    inputs, labels = data
    outputs = net(inputs)
    _, pred = torch.max(outputs.data, 1)
    total += labels.size(0)
    correct += (pred == labels).sum().item()
  return 100 * correct / total


net = LeNet()

loss_fn = nn.CrossEntropyLoss()
opt = optim.Adam(net.parameters())

%%time
loss_arr = []
loss_epoch_arr = []
max_epochs = 16

for epoch in range(max_epochs):

  for i, data in enumerate(trainloader, 0):

    inputs, labels = data

    opt.zero_grad()

    outputs = net(inputs)
    loss = loss_fn(outputs, labels)
    loss.backward()
    opt.step()

    loss_arr.append(loss.item())

  loss_epoch_arr.append(loss.item())

  print('Epoch: %d/%d, Test acc: %0.2f, Train acc: %0.2f' % (epoch, max_epochs,
```

```
    evaluation(testloader), evaluation(trainloader)))

plt.plot(loss_epoch_arr)
plt.show()
```

Output:

```
Epoch: 0/16, Test acc: 38.39, Train acc: 38.13
Epoch: 1/16, Test acc: 43.67, Train acc: 43.74
Epoch: 2/16, Test acc: 46.30, Train acc: 46.62
Epoch: 3/16, Test acc: 49.37, Train acc: 50.37
Epoch: 4/16, Test acc: 50.15, Train acc: 51.86
Epoch: 5/16, Test acc: 52.14, Train acc: 54.40
Epoch: 6/16, Test acc: 52.72, Train acc: 56.28
Epoch: 7/16, Test acc: 53.53, Train acc: 57.73
Epoch: 8/16, Test acc: 54.44, Train acc: 58.82
Epoch: 9/16, Test acc: 54.61, Train acc: 59.97
Epoch: 10/16, Test acc: 55.91, Train acc: 61.58
Epoch: 11/16, Test acc: 55.41, Train acc: 61.88
Epoch: 12/16, Test acc: 55.28, Train acc: 63.09
Epoch: 13/16, Test acc: 56.54, Train acc: 64.56
Epoch: 14/16, Test acc: 56.37, Train acc: 64.63
Epoch: 15/16, Test acc: 56.54, Train acc: 66.50
```




```
CPU times: user 7min 39s, sys: 9.16 s, total: 7min 48s
Wall time: 7min 49s
```

10. Implement RNN for Text Classification

Data Set: https://www.kaggle.com/datasets/rp1985/name2lang

```
from io import open
import os, string, random, time, math
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np
from sklearn.model_selection import train_test_split
import torch
import torch.nn as nn
import torch.optim as optim
from IPython.display import clear_output
languages = []
data = []
X = []
y = []


#Dataset Loading

with open('/content/name2lang.txt', 'r') as f:
    for line in f:
        line = line.split(',')
        name = line[0].strip()
        lang = line[1].strip()
        if not lang in languages:
            languages.append(lang)
        X.append(name)
        y.append(lang)
        data.append((name, lang))
         n_languages = len(languages)

#Train-test split

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=0, stratify=y)

#Encoding names and language

all_letters = string.ascii_letters + " .,;'"
n_letters = len(all_letters)

def name_rep(name):
    rep = torch.zeros(len(name), 1, n_letters)
    for index, letter in enumerate(name):
        pos = all_letters.find(letter)
        rep[index][0][pos] = 1
    return rep

def lang_rep(lang):
    return torch.tensor([languages.index(lang)], dtype=torch.long)
```

```
def dataloader(npoints, X_, y_):
    to_ret = []
    for i in range(npoints):
        index_ = np.random.randint(len(X_))
        name, lang = X_[index_], y_[index_]
        to_ret.append((name, lang, name_rep(name), lang_rep(lang)))
    return to_ret

def eval(net, n_points, k, X_, y_):

    data_ = dataloader(n_points, X_, y_)
    correct = 0

    for name, language, name_ohe, lang_rep in data_:

        output = infer(net, name)
        val, indices = output.topk(k)

        if lang_rep in indices:
            correct += 1

    accuracy = correct/n_points


def train(net, opt, criterion, n_points):

    opt.zero_grad()
    total_loss = 0

    data_ = dataloader(n_points, X_train, y_train)

    for name, language, name_ohe, lang_rep in data_:

        hidden = net.init_hidden()

        for i in range(name_ohe.size()[0]):
            output, hidden = net(name_ohe[i], hidden)

        loss = criterion(output, lang_rep)
        loss.backward(retain_graph=True)

        total_loss += loss

    opt.step()

  return total_loss/n_points

def infer(net, name):
    net.eval()
    name_ohe = name_rep(name)
    hidden = net.init_hidden()

    for i in range(name_ohe.size()[0]):
        output, hidden = net(name_ohe[i], hidden)

    return output
```

```
#Basic network and testing inference

class RNN_net(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(RNN_net, self).__init__()
        self.hidden_size = hidden_size
        self.i2h = nn.Linear(input_size + hidden_size, hidden_size)
        self.i2o = nn.Linear(input_size + hidden_size, output_size)
        self.softmax = nn.LogSoftmax(dim=1)

    def forward(self, input_, hidden):
        combined = torch.cat((input_, hidden), 1)
        hidden = self.i2h(combined)
        output = self.i2o(combined)
        output = self.softmax(output)
        return output, hidden

    def init_hidden(self):
        return torch.zeros(1, self.hidden_size)

def train_setup(net, lr = 0.01, n_batches = 100, batch_size = 10, momentum = 0.9, display_freq=5):

    criterion = nn.NLLLoss()
    opt = optim.SGD(net.parameters(), lr=lr, momentum=momentum)

    loss_arr = np.zeros(n_batches + 1)

    for i in range(n_batches):
        loss_arr[i+1] = (loss_arr[i]*i + train(net, opt, criterion, batch_size))/(i + 1)

        if i%display_freq == display_freq-1:
            clear_output(wait=True)

            print('Iteration', i, 'Top-1:', eval(net, len(X_test), 1, X_test, y_test), 'Top-2:',
eval(net, len(X_test), 2, X_test, y_test), 'Loss', loss_arr[i])
            plt.figure()
            plt.plot(loss_arr[1:i], '-*')
            plt.xlabel('Iteration')
            plt.ylabel('Loss')
            plt.show()
            print('\n\n')

n_hidden = 128
net = RNN_net(n_letters, n_hidden, n_languages)
train_setup(net, lr=0.0005, n_batches=100, batch_size = 256)
```

Output:

```
Iteration 99 Top-1: 0.6428927680798004 Top-2: 0.7713216957605985 Loss 1.4926992654800415
```