Learning Manual

MALLA REDDY UNIVERSITY

(Telangana State Private Universities Act No.13 of 2020 and G.O.Ms.No.14, Higher Education (UE) Department)

Name	
Roll No...............................	Branch ....
Year	Sem.....................

MALLA REDDY UNIVERSITY
(Telangana State Private Universities Act No.13 of 2020 and
G.O.Ms.No.14, Higher Education (UE) Department)

Maisammaguda, Kompally,
Medchal - Malkajgiri District
Hyderabad - 500100, Telangana State.
mruh@mallareddyuniversity.ac.in
www.mallareddyuniversity.ac.in

Certificate

School of Engineering

Certified that this is the bonafide record of practical work done by

Mr./Ms.. Roll. No.................. of

B.Tech .... year .... Semester for Academic year 20.... - 20.... in

.... Laboratory.

Date: Faculty Incharge

KNOWLEDGE EMPOWERMENT WORLD

MALLA REDDY UNIVERSITY

(Telangana State Private Universities Act No.13 of 2020 and

Deep Learning Lab

MR20-1CS0284

Learning Manual

B.Tech: III Year II Semester

(CSE-AI&ML)

(2023-24)

School of Engineering

www.mallareddyuniversity.ac.in

“ A thought beyond horizons of success committed for professional excellence ”

Vision

To be a world class university visualizing a great future for the young aspirants, with innovative nature, research culture and ethical sensitivities to meet the global challenges improving the quality of human life.

Mission

To impart value based futuristic higher education moulding students into globally competent empowered youth, rich in culture and ethics along with professional expertise.

To promote innovation, entrepreneurship, research and development for the broad purpose of fulfilling societal goals such as societal welfare and benefit.

Quality Policy

To pursue continual improvement of teaching learning process of Undergraduate, Post Graduate programs and Research Programs vigorously.

To provide state of the art infrastructure and expertise to impart the quality education.

To groom the students to become intellectually creative and professionally competitive.

To explore the opportunities in the professional fields.

Academic Best Practices

Industry Oriented Curriculum

Technology Training and Certifications

Institution to Corporate Exposure

Interactive Learning

International Career Guidance

Choice Based Flexible Curriculum

MALLA REDDY UNIVERSITY

(Telangana State Private Universities Act No.13 of 2020 and G.O.Ms.No.14, Higher Education (UE) Department)

Maisammaguda, Kompally, Hyderabad - 500 100
Telangana State

www.mallareddyuniversity.ac.in

CSE-AI & ML DEPARTMENT (III YEAR II SEMESTER)

DEEP LEARNING LAB
LIST OF EXERCISES / EXPERIMENTS

S No.	Name of the Experiment	Page No.
1	Getting familiar with the usage of Google colab and using GPU as processing unit.	
2	Study and implementation of feed forward NN.	
3	Study and implementation of back propagation.	
4	Implement Batch gradient descent, Stochastic gradient descent and mini batch gradient descent.	
5	Study the effect of batch normalization and dropout in neural network classifier PCA.	
6	Study of Singular value Decomposition for dimensionality reduction..	
7	Train a sentiment analysis model on IMDB dataset, use RNN layers.	
8	Perform Object detection USING CNN.	
9	Implementing Autoencoder for encoding the real-world data.	
10	Study and implementation of LSTM	
11	Implementation of GAN for generating Handwritten Digits images.	
12	Implementing word2vec for the rela-world data.	

1. Getting familiar with the usage of Google Colab and using GPU as processing unit

Introduction

Colaboratory by Google (Google Colab in short) is a Jupyter notebook based runtime environment which allows you to run code entirely on the cloud. This is necessary because it means that you can train large scale ML and DL models even if you don't have access to a powerful machine or a high-speed internet access. Google Colab supports both GPU and TPU instances, which makes it a perfect tool for deep learning and data analytics enthusiasts because of computational limitations on local machines. Since a Colab notebook can be accessed remotely from any machine through a browser, it's well suited for commercial purposes as well.

In this tutorial you will learn:

- Getting around in Google Colab
- Installing python libraries in Colab
- Downloading large datasets in Colab
- Training a Deep learning model in Colab
- Using TensorBoard in Colab

Creating your first .ipynb notebook in colab

Open a browser of your choice and go to colab.research.google.com and sign in using your Google account. Click on a new notebook to create a new runtime instance. In the top left corner, you can change the name of the notebook from "Untitled.ipynb" to the name of your choice by clicking on it. The cell execution block is where you type your code. To execute the cell, press shift + enter. The variable declared in one cell can be used in other cells as a global variable. The environment automatically prints the value of the variable in the last line of the code block if stated explicitly.

Training a sample tensorflow model

Training a machine learning model in Colab is very easy. The best part about it is not having to set up a custom runtime environment, it's all handled for you. For example, let's look at training a basic deep learning model to recognize handwritten digits trained on the MNIST dataset. The data is loaded from the standard Keras dataset archive. The model is very basic, it categorizes images as numbers and recognizes them.

Setup:

```
#import necessary libraries
import tensorflow as tf

#load training data and split into train and test sets
mnist = tf.keras.datasets.mnist

(x_train,y_train), (x_test,y_test) = mnist.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0
```

The output for this code snippet will look like this:

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz
11490434/11490434 [==============================] - 0s 0us/step
```

Next, we define the Google Colab model using Python:

```
#define model
model = tf.keras.models.Sequential([
                        tf.keras.layers.Flatten(input_shape=(28,28)),
                           tf.keras.layers.Dense(128,activation='relu'),
                           tf.keras.layers.Dropout(0.2),
                           tf.keras.layers.Dense(10)
])

#define loss function variable
loss_fn = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True)

#define optimizer,loss function and evaluation metric
model.compile(optimizer='adam',
          loss=loss_fn,
          metrics=['accuracy'])

#train the model
model.fit(x_train,y_train,epochs=5)
```

The expected output upon execution of the above code snippet is:

```
Epoch 1/5
1875/1875 [==============================] - 7s 3ms/step - loss: 0.2954 - accuracy: 0.9138
Epoch 2/5
1875/1875 [==============================] - 6s 3ms/step - loss: 0.1422 - accuracy: 0.9572
Epoch 3/5
1875/1875 [==============================] - 5s 3ms/step - loss: 0.1070 - accuracy: 0.9664
Epoch 4/5
1875/1875 [==============================] - 5s 3ms/step - loss: 0.0886 - accuracy: 0.9721
Epoch 5/5
1875/1875 [==============================] - 5s 3ms/step - loss: 0.0753 - accuracy: 0.9764
<keras.callbacks.History at 0x7f84a21a2760>
```

Next, we test the model accuracy on test set:

```
#test model accuracy on test set
model.evaluate(x_test,y_test,verbose=2)
```

The expected output upon execution of the above code snippet is:

```
313/313 - 1s - loss: 0.0809 - accuracy: 0.9744 - 655ms/epoch - 2ms/step
[0.0809035375714302, 0.974399983882904]
```

Next, we extend the base model to predict softmax output:

```
#extend the base model to predict softmax output
probability_model = tf.keras.Sequential([
                    model,
                    tf.keras.layers.Softmax()])
```

Installing packages in Google Colab

One can use the code cell in Colab not only to run Python code but also to run shell commands. Just add a ! before a command. The exclamation point tells the notebook cell to run the following command as a shell command. Most general packages needed for deep learning come pre-installed. In some cases, you might need less popular libraries, or you might need to run code on a different version of a library. To do this, you'll need to install packages manually.
The package manager used for installing packages is pip. To install a particular version of TensorFlow use this command:

```
!pip3 install tensorflow
```

Downloading a dataset

When you're training a machine learning model on your local machine, you're likely to have trouble with the storage and bandwidth costs that come with downloading and storing the dataset required for training a model. Deep learning datasets can be massive in size, ranging between 20 to 50 Gb. Downloading them is most challenging if you're living in a developing country, where getting high-speed internet isn't possible. The most efficient way to use datasets is to use a cloud interface to download them, rather than manually uploading the dataset from a local machine. Thankfully, Colab gives us a variety of ways to download the dataset from common data hosting platforms.
To download an existing dataset from Kaggle, we can follow the steps outlined below:

1. Go to your Kaggle Account and click on "Create New API Token". This will download a kaggle.json file to your machine.
2. Go to your Google Colab project file, and run the following commands:

```
! pip install -q kaggle
from google.colab import files

# choose the kaggle.json file that you downloaded
files.upload()
! mkdir ~/.kaggle
```

Downloading the dataset from any generic website

Depending on the browser that you're using, there are extensions available to convert the dataset download link into `curl` or `wget` format. You can use this to efficiently download the dataset.

1. For Firefox, there's the cliget browser extension: cliget – Get this Extension for Firefox (en-US)
2. For Chrome, there's the CurlWget extension: Ad Added CurlWget 52

These extensions will generate a curl/wget command as soon as you click on any download button in your browser.
You can then copy that command and execute it in your Colab notebook to download the dataset.

NOTE: By default, the Colab notebook uses Python shell. To run terminal commands in Colab, you will have to use "!" at the beginning of the command.

For example, to download a file from some.url and save it as some.file, you can use the following command in Colab:

```
!curl http://some.url --output some.file
```

NOTE: The curl command will download the dataset in the Colab workspace, which will be lost every time the runtime is disconnected. Hence a safe practice is to move the dataset into your cloud drive as soon as the dataset is downloaded completely.

Downloading the dataset from GCP or Google Drive

Google Cloud Platform is a cloud computing and storage platform. You can use it to store large datasets, and you can import that dataset directly from the cloud into Colab. To upload and download files on GCP, first you need to authenticate your Google account.

```
from google.colab import auth
auth.authenticate_user()
```

After that, install gsutil to upload and download files, and then init gcloud.

```
!curl https://sdk.cloud.google.com | bash
!gcloud init
```

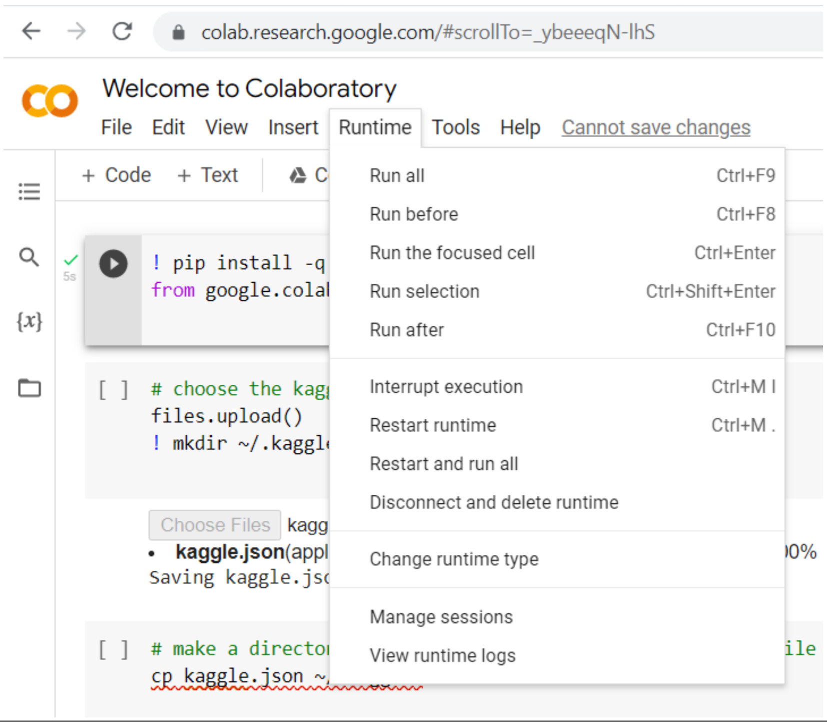
Once you have configured these options, you can use the following commands to download/upload files to and from Google Cloud Storage.

Initiating a runtime with GPU/TPU enabled

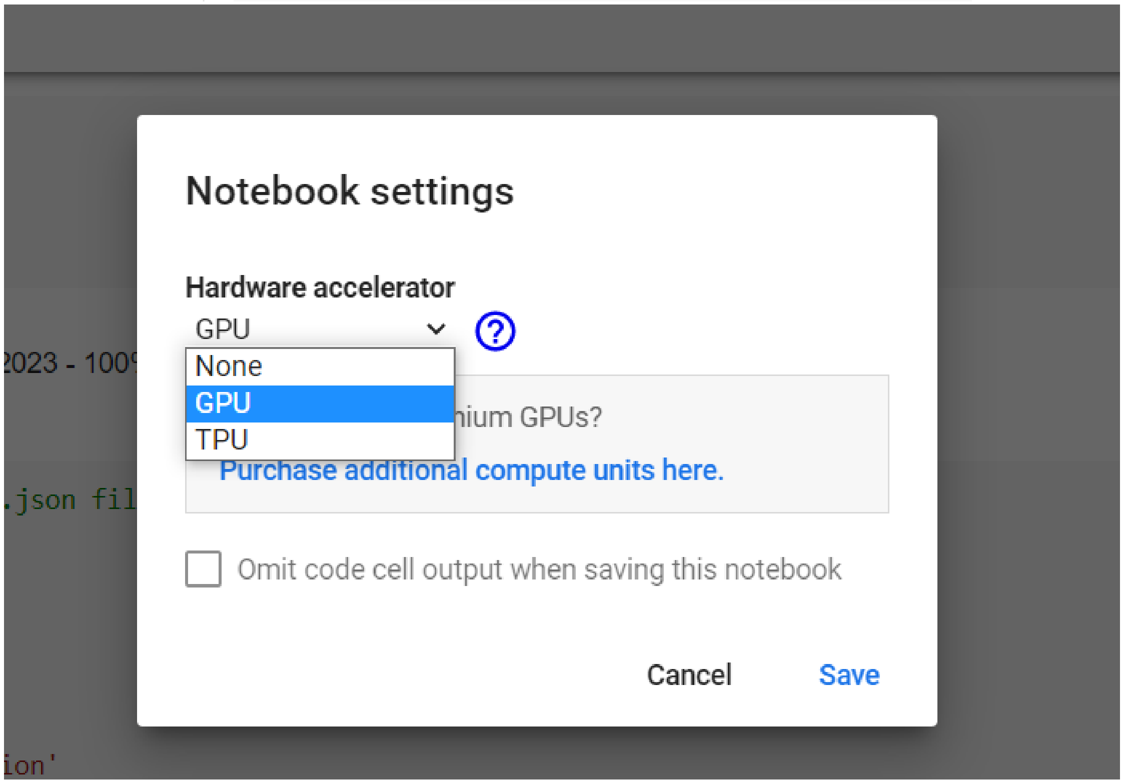
Deep learning is a computationally expensive process, a lot of calculations need to be executed at the same time to train a model. To mitigate this issue, Google Colab offers us not only the classic CPU runtime but also an option for a GPU and TPU runtime as well.

The CPU runtime is best for training large models because of the high memory it provides. The GPU runtime shows better flexibility and programmability for irregular computations, such as small batches and nonMatMul computations. The TPU runtime is highly-optimized for large batches and CNNs and has the highest training throughput. If you have a smaller model to train, I suggest training the model on GPU/TPU runtime to use Colab to its full potential.

To create a GPU/TPU enabled runtime, you can click on runtime in the toolbar menu below the file name. From there, click on "**Change runtime type**", and then select GPU or TPU under the Hardware Accelerator dropdown menu.


colab.research.google.com/#scrollTo=_ybeeeqN-lhS
Welcome to Colaboratory
File Edit View Insert Runtime Tools Help Cannot save changes
+ Code + Text
Run all Ctrl+F9
Run before Ctrl+F8
Run the focused cell Ctrl+Enter
Run selection Ctrl+Shift+Enter
Run after Ctrl+F10
Interrupt execution Ctrl+M I
Restart runtime Ctrl+M .
Restart and run all
Disconnect and delete runtime
Change runtime type
Manage sessions
View runtime logs
Choose Files

```
! pip install -q
from google.colal
# choose the kag
files.upload()
! mkdir ~/.kaggl
kaggle.json(appl
Saving kaggle.js
# make a directo
cp kaggle.json ~
```


Notebook settings
Hardware accelerator
GPU
None
GPU
TPU
ium GPUs?
Purchase additional compute units here.
Omit code cell output when saving this notebook
Cancel
Save

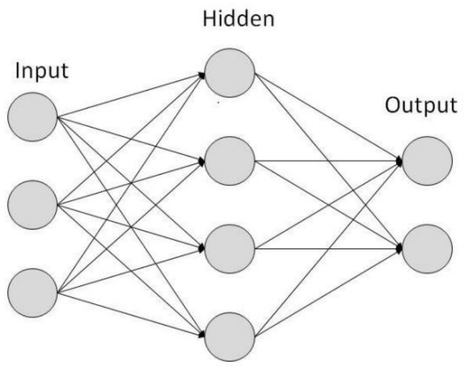
Exercise: Try yourself uploading a dataset and perform basic data pre-processing on it

2. Getting Study and implementation of feed forward Neural Network

What is a (Neural Network) NN?

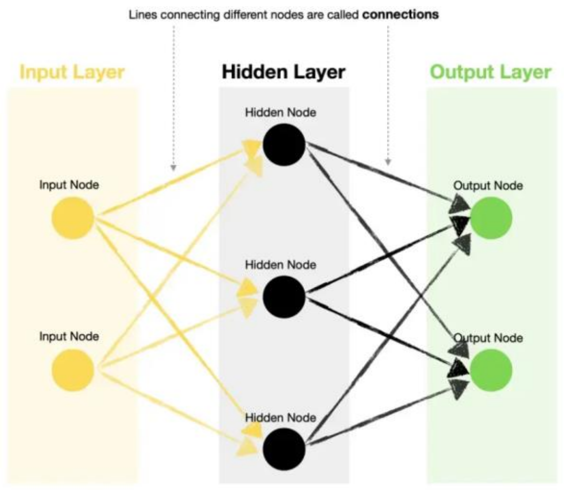
- Single neuron == linear regression without applying activation(perceptron)
- Basically a single neuron will calculate weighted sum of input(W.T*X) and then we can set a threshold to predict output in a perceptron. If weighted sum of input cross the threshold, perceptron fires and if not then perceptron doesn't predict.
- Perceptron can take real values input or boolean values.
- Actually, when w·x+b=0 the perceptron outputs 0.
- Disadvantage of perceptron is that it only output binary values and if we try to give small change in weight and bais then perceptron can flip the output. We need some system which can modify the output slightly according to small change in weight and bias. Here comes sigmoid function in picture.
- If we change perceptron with a sigmoid function, then we can make slight change in output. e.g. output in perceptron = 0, you slightly changed weight and bias, output becomes = 1 but actual output is 0.7. In case of sigmoid, output1 = 0, slight change in weight and bias, output = 0.7.
- If we apply sigmoid activation function then Single neuron will act as Logistic Regression. we can understand difference between perceptron and sigmoid function by looking at sigmoid function graph.

Simple NN graph:



A visual explanation of how Feed Forward NNs work

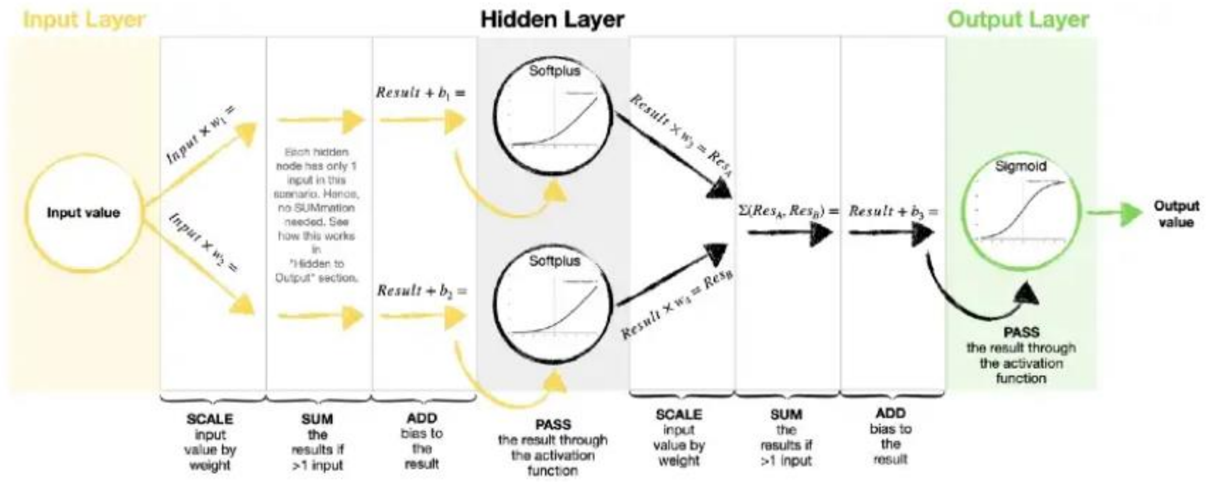
First, let’s familiarize ourselves with the basic structure of a Neural Network.



- **Input Layer** — contains one or more input nodes. For example, suppose you want to predict whether it will rain tomorrow and base your decision on two variables, humidity and wind speed. In that case, your first input would be the value for humidity, and the second input would be the value for wind speed.
- **Hidden Layer** — this layer houses hidden nodes, each containing an **activation function** (more on these later). *Note that a Neural Network with multiple hidden layers is known as Deep Neural Network.*
- **Output Layer** — contains one or more output nodes. Following the same weather prediction example above, you could choose to have only one output node generating a rain probability (where >0.5 means rain tomorrow, and ≤0.5 no rain tomorrow). Alternatively, you could have two output nodes, one for rain and another for no rain. Note, you can use a different **activation function** for output nodes vs. hidden nodes.
- **Connections** — lines joining different nodes are known as connections. These contain **kernels (weights)** and **biases**, the parameters that get optimized during the training of a neural network.

Parameters and activation functions

Let’s take a closer look at kernels (weights) and biases to understand what they do. For simplicity, we create a basic neural network with one input node, two hidden nodes, and one output node (1–2–1)


Input Layer
Hidden Layer
Output Layer
Input value
$Input \times w_1 =$
$Input \times w_2 =$
Each hidden node has only 1 input in this scenario. Hence, no SUMmation needed. See how this works in "Hidden to Output" section.
$Result + b_1 =$
$Result + b_2 =$
Softplus
Softplus
$Result \times w_3 = Res_A$
$Result \times w_4 = Res_B$
$\Sigma(Res_A, Res_B) =$
$Result + b_3 =$
Sigmoid
Output value
PASS the result through the activation function
SCALE input value by weight
SUM the results if >1 input
ADD bias to the result
PASS the result through the activation function
SCALE input value by weight
SUM the results if >1 input
ADD bias to the result

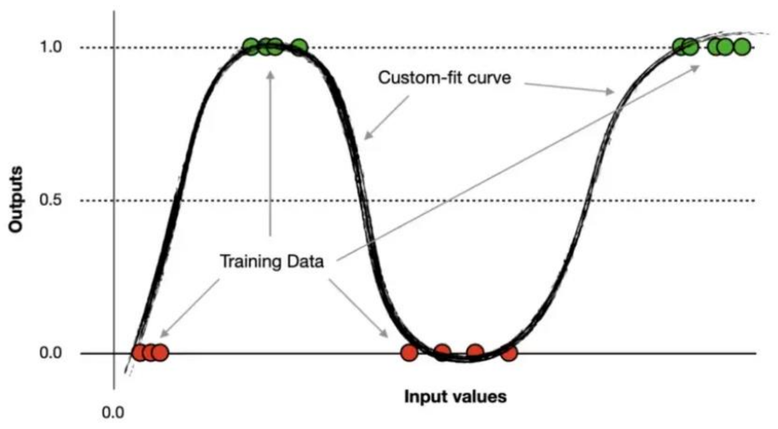
- **Kernels (weights)** — used to scale input and hidden node values. Each connection typically holds a different weight.
- **Biases** — used to adjust scaled values before passing them through an activation function.
- **Activation functions** — think of activation functions as standard curves (building blocks) used by the Neural Network to create a custom curve to fit the training data. Passing different input values through the network selects different sections of the standard curve, which are then assembled into a final custom-fit curve.

Loss functions, optimizers, and training

Training Neural Networks involves a complicated process known as backpropagation. I will not go through a step-by-step explanation of how backpropagation works since it is a big enough topic deserving a separate article. Instead, let me briefly introduce you to loss functions and optimizers and summarize what happens when we "train" a Neural Network.

- **Loss** — represents the "size" of error between the **true** values/labels and the **predicted** values/labels. The goal of training a Neural Network is to minimize this loss. The smaller the loss, the closer the match between the true and the predicted data. There are many **loss functions** to choose from, with Binary Crossentropy, Categorical Crossentropy, and Mean Squared Error being the most common.
- **Optimizers** — are the **algorithms used in backpropagation**. The goal of an optimizer is to find the optimum set of kernels (weights) and biases to minimize the loss. Optimizers typically use a gradient descent approach, which allows them to iteratively find the "best" possible configuration of weights and biases. The most commonly used ones are SGD, ADAM, and RMS Prop.

Training a Neural Network is basically fitting a custom curve through the training data until it can approximate it as well as possible. The graph below illustrates what a custom-fitted curve could look like in a specific scenario. This example contains a set of data that seem to flip between 0 and 1 as the value for input increases.



Setup

We’ll need the following data and libraries:

Australian weather data from Kaggle (license: Creative Commons, original source of the data: Commonwealth of Australia, Bureau of Meteorology).

Pandas and Numpy for data manipulation

Plotly for data visualizations

Tensorflow/Keras for Neural Networks

Scikit-learn library for splitting the data into train-test samples, and for some basic model evaluation

Example: Implementation of Feed Forward Neural Network.

Code along with outputs:

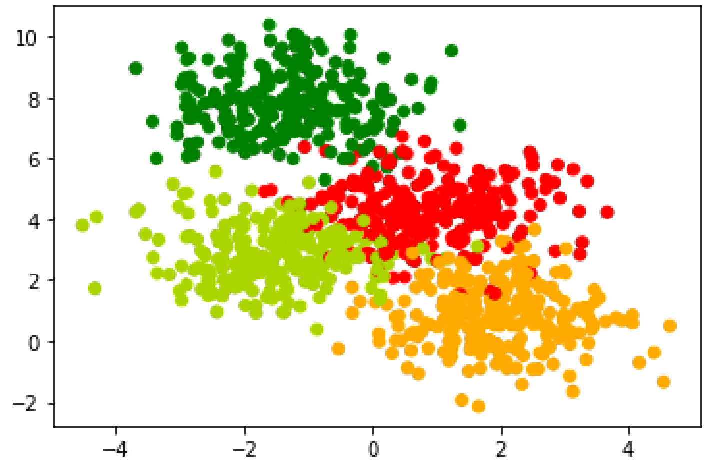
```
#Importing required libraries for the code implementation
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import matplotlib.colors
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, mean_squared_error
from tqdm import tqdm_notebook

from sklearn.preprocessing import OneHotEncoder
from sklearn.datasets import make_blobs

Creating sigmoid neuron
class SigmoidNeuron:
  #A class for sigmoid neuron

  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, initi
alise=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=epo
chs, 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)
#custom cmap

my_cmap = matplotlib.colors.LinearSegmentedCol
ormap.from_list("", ["red", "yellow", "green"])

Generate data
data, labels = make_blobs(n_samples=1000, center
s=4,n_features=2, random_state=0)
print(data.shape, labels.shape)
```

Output:

```
(1000, 2) (1000,)
```

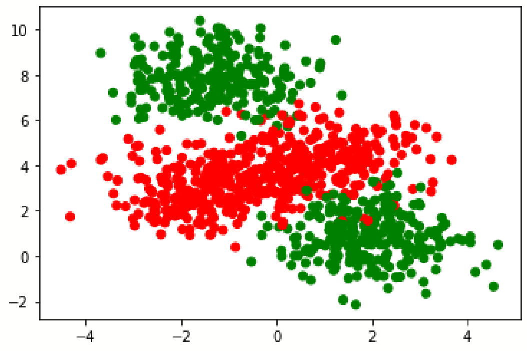
```
plt.scatter(data[:,0], data[:,1], c = labels, cmap = my_cmap)
plt.show()
```

Output:



```
labels_orig = labels
#converting the multi-class to binary class
labels = np.mod(labels_orig,2)

plt.scatter(data[:,0], data[:,1], c=labels, cmap=my_cmap)
plt.show()
```

Output:



```
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)
```

Output:

```
(750, 2) (250, 2)
```

Implementation of Feed Forward Neural Network code

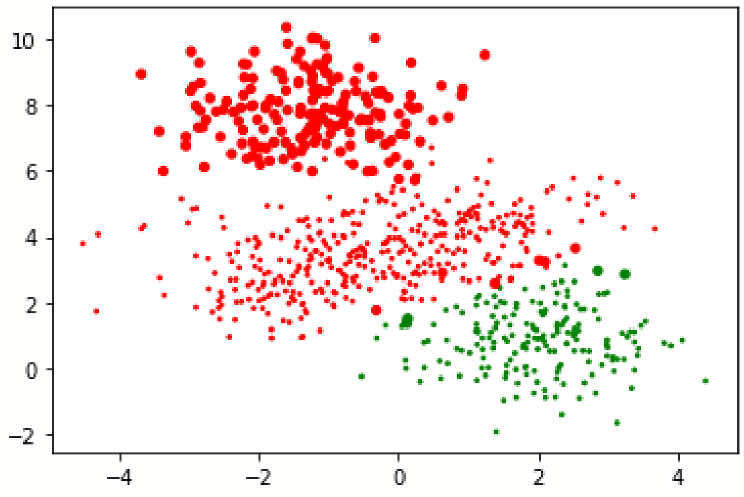
```
class FirstFFNetwork:

  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, initi
alise=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=epo
chs, 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)
```

```
Y_pred_train = ffn.predict(X_train)
Y_pred_binarised_train = (Y_pred_train >= 0.5).as
type("int").ravel()
Y_pred_val = ffn.predict(X_val)
Y_pred_binarised_val = (Y_pred_val >= 0.5).asty
pe("int").ravel()
accuracy_train = accuracy_score(Y_pred_binarise
d_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
))
```

Output:

```
Training accuracy 0.73
Validation accuracy 0.72
```

```
#visualize the results
plt.scatter(X_train[:,0], X_train[:,1], c=Y_pred_bin
arised_train, cmap=my_cmap, s=15*(np.abs(Y_pr
ed_binarised_train-Y_train)+.2))
plt.show()
```



Exercise: Try to improve the accuracy in the above example.

3. Study and implementation of back propagation.

Why use the back propagation algorithm?

Earlier we discussed that the network is trained using 2 passes: forward and backward. At the end of the forward pass, the network error is calculated, and should be as small as possible.

If the current error is high, the network didn't learn properly from the data. What does this mean? It means that the current set of weights isn't accurate enough to reduce the network error and make accurate predictions. As a result, we should update network weights to reduce the network error.

The back propagation algorithm is one of the algorithms responsible for updating network weights with the objective of reducing the network error. It's quite important.

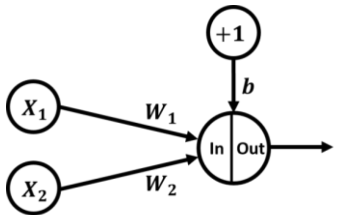
Here are some of the advantages of the back propagation algorithm:

- It's memory-efficient in calculating the derivatives, as it uses less memory compared to other optimization algorithms, like the genetic algorithm. This is a very important feature, especially with large networks.
- The back propagation algorithm is fast, especially for small and medium-sized networks. As more layers and neurons are added, it starts to get slower as more derivatives are calculated.
- This algorithm is generic enough to work with different network architectures, like convolutional neural networks, generative adversarial networks, fully-connected networks, and more.
- There are no parameters to tune the back propagation algorithm, so there's less overhead. The only parameters in the process are related to the gradient descent algorithm, like learning rate.

How back propagation algorithm works

How the algorithm works is best explained based on a simple network, like the one given in the next figure. It only has an input layer with 2 inputs (X1 and X2), and an output layer with 1 output. There are no hidden layers.

The weights of the inputs are W1 and W2, respectively. The bias is treated as a new input neuron to the output neuron which has a fixed value +1 and a weight b. Both the weights and biases could be referred to as parameters.



Code along with outputs:

```
import numpy
import matplotlib.pyplot

def sigmoid(sop):
    return 1.0/(1+numpy.exp(-1*sop))
def error(predicted, target):
    return numpy.power(predicted-target, 2)
def error_predicted_deriv(predicted, target):
    return 2*(predicted-target)
def sigmoid_sop_deriv(sop):
    return sigmoid(sop)*(1.0-sigmoid(sop))
def sop_w_deriv(x):
    return x
def update_w(w, grad, learning_rate):
    return w - learning_rate*grad
x1=0.1
x2=0.4
target = 0.7
learning_rate = 0.01

w1=numpy.random.rand()
w2=numpy.random.rand()

print("Initial W : ", w1, w2)

predicted_output = []
network_error = []

old_err = 0
for k in range(80000):
    # Forward Pass
    y = w1*x1 + w2*x2
    predicted = sigmoid(y)
    err = error(predicted, target)

    predicted_output.append(predicted)
    network_error.append(err)

    # Backward Pass
    g1 = error_predicted_deriv(predicted, target)

    g2 = sigmoid_sop_deriv(y)
    g3w1 = sop_w_deriv(x1)
    g3w2 = sop_w_deriv(x2)

    gradw1 = g3w1*g2*g1
    gradw2 = g3w2*g2*g1

    w1 = update_w(w1, gradw1, learning_rate)
    w2 = update_w(w2, gradw2, learning_rate)

    print(predicted)

matplotlib.pyplot.figure()
matplotlib.pyplot.plot(network_error)
matplotlib.pyplot.title("Iteration Number vs Error")
matplotlib.pyplot.xlabel("Iteration Number")
matplotlib.pyplot.ylabel("Error")

matplotlib.pyplot.figure()
matplotlib.pyplot.plot(predicted_output)
matplotlib.pyplot.title("Iteration Number vs Prediction")
matplotlib.pyplot.xlabel("Iteration Number")
matplotlib.pyplot.ylabel("Prediction")
```

Output:

Streaming output truncated to the last 5000 lines.

```
0.6999984864252651
0.6999984866522116
0.6999984868791244
0.699998487106003
0.6999984873328476

.

.

.

.

0.6999992843293953
0.6999992844367032
0.6999992845439951
0.6999992846512709
0.6999992847585306
```

Example: Execute the error plots for 80,000 epochs and show the saturated error output value. Also write a short note about your analysis with the output value.

Solution:

4. Implement Batch gradient descent, Stochastic gradient descent and mini batch gradient descent.

Objective:

To implement Batch gradient descent, Stochastic gradient descent and mini batch gradient descent.

Introduction to the Topic:

We will use very simple home prices data set to implement batch and stochastic gradient descent in python.

Batch gradient descent uses all training samples in forward pass to calculate cumulative error and then we adjust weights using derivatives. In stochastic GD, we randomly pick one training sample, perform forward pass, compute the error and immediately adjust weights.

So the key difference here is that to adjust weights, batch GD will use all training samples where as stochastic GD will use one randomly picked training sample.

Mini batch is intermediate version of batch GD and stochastic GD. In mini gradient descent you will use a batch of samples in each iteration. For example if you have total 50 training samples, you can take a batch of 10 samples, calculate cumulative error for those 10 samples and then adjust weights.

To summarize it, In SGD we adjust weights after every one sample. In Batch we adjust weights after going through all samples but in mini batch we do after every m samples (where m is batch size and it is 0 < m < n, where n is total number of samples).

Gradient descent allows you to find weights (w1,w2,w3) and bias in the following linear equation for housing price prediction.

$$price = w1 * area + w2 * bedrooms + bias$$

Example and solution:

```
import pandas as pd
import numpy as np
from matplotlib import pyplot as plt
%matplotlib inline
```

```
df = pd.read_csv("homeprices.csv")
df.sample(5)

output:
index,area,bedrooms,price
5,1170,2,38.0
14,2250,3,101.0
8,1310,3,50.0
10,1800,3,82.0
0,1056,2,39.07

from sklearn import preprocessing
sx = preprocessing.MinMaxScaler()
sy = preprocessing.MinMaxScaler()

scaled_X = sx.fit_transform(df.drop('price',axis='columns'))
scaled_y = sy.fit_transform(df['price'].values.reshape(df.shape[0],1))

scaled_X
scaled_y
scaled_y.reshape(20,)
```

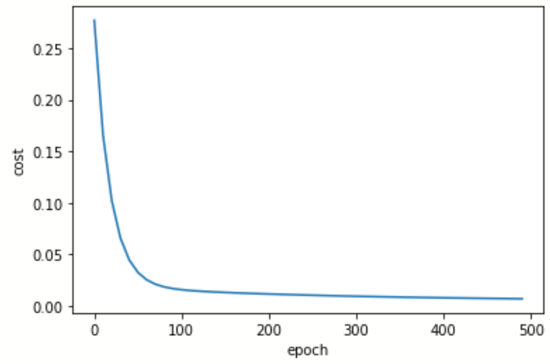
(1) Batch Gradient Descent Implementation

```
def batch_gradient_descent(X, y_true, epochs, learning_rate = 0.01):

    number_of_features = X.shape[1]
    # numpy array with 1 row and columns equal to number of features. In
    # our case number_of_features = 2 (area, bedroom)
    w = np.ones(shape=(number_of_features))
    b = 0
    total_samples = X.shape[0] # number of rows in X

    cost_list = []
    epoch_list = []

    for i in range(epochs):
        y_predicted = np.dot(w, X.T) + b

        w_grad = -(2/total_samples)*(X.T.dot(y_true-y_predicted))
        b_grad = -(2/total_samples)*np.sum(y_true-y_predicted)

        w = w - learning_rate * w_grad
        b = b - learning_rate * b_grad
        cost = np.mean(np.square(y_true-
y_predicted)) # MSE (Mean Squared Error)

        if i%10==0:
            cost_list.append(cost)
            epoch_list.append(i)

    return w, b, cost, cost_list, epoch_list

w, b, cost, cost_list, epoch_list = batch_gradient_descent(scaled_X,scaled
_y.reshape(scaled_y.shape[0],),500)
w, b, cost
```

Output:
(array([0.70712464, 0.67456527]), -0.23034857438407427, 0.0068641890429808105)

```
plt.xlabel("epoch")
plt.ylabel("cost")
plt.plot(epoch_list,cost_list)
```

Output:



```
def predict(area,bedrooms,w,b):
    scaled_X = sx.transform([[area, bedrooms]])[0]
    # here w1 = w[0] , w2 = w[1], w3 = w[2] and bias is b
    # equation for price is w1*area + w2*bedrooms + w3*age + bias
    # scaled_X[0] is area
    # scaled_X[1] is bedrooms
    # scaled_X[2] is age
    scaled_price = w[0] * scaled_X[0] + w[1] * scaled_X[1] + b
    # once we get price prediction we need to to rescal it back to origina
l value
```

```
    # also since it returns 2D array, to get single value we need to do va
lue[0][0]
    return sy.inverse_transform([[scaled_price]])[0][0]

predict(2600,4,w,b)
predict(1000,2,w,b)
predict(1500,3,w,b)
```

(2) Stochastic Gradient Descent Implementation

# we will use random libary to pick random training sample.

```
import random
random.randint(0,6)# randit gives random number between two numbers specified in the argument

def stochastic_gradient_descent(X, y_true, epochs, learning_rate = 0.01):

    number_of_features = X.shape[1]
    # numpy array with 1 row and columns equal to number of features. In
    # our case number_of_features = 3 (area, bedroom and age)
    w = np.ones(shape=(number_of_features))
    b = 0
    total_samples = X.shape[0]

    cost_list = []
    epoch_list = []

    for i in range(epochs):
        random_index = random.randint(0,total_samples-
1) # random index from total samples
        sample_x = X[random_index]
        sample_y = y_true[random_index]

        y_predicted = np.dot(w, sample_x.T) + b

        w_grad = -(2/total_samples)*(sample_x.T.dot(sample_y-y_predicted))
        b_grad = -(2/total_samples)*(sample_y-y_predicted)

        w = w - learning_rate * w_grad
        b = b - learning_rate * b_grad

        cost = np.square(sample_y-y_predicted)

        if i%100==0: # at every 100th iteration record the cost and epoch
value
```

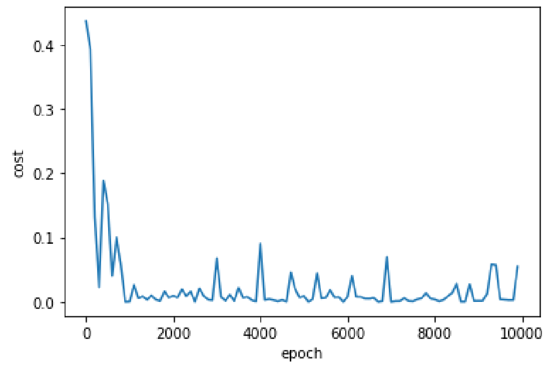
```
            cost_list.append(cost)
            epoch_list.append(i)

    return w, b, cost, cost_list, epoch_list

w_sgd, b_sgd, cost_sgd, cost_list_sgd, epoch_list_sgd = stochastic_gradien
t_descent(scaled_X,scaled_y.reshape(scaled_y.shape[0],),10000)
w_sgd, b_sgd, cost_sgd
```

Output:

```
(array([0.70999659, 0.67807531]), -0.23262199997984362,
0.011241941221627246)
```

```
w , b
plt.xlabel("epoch")
plt.ylabel("cost")
plt.plot(epoch_list_sgd,cost_list_sgd)
```

Output:



```
predict(2600,4,w_sgd, b_sgd)
predict(1000,2,w_sgd, b_sgd)
np.random.permutation(20)
```

Output:

```
array([16, 10, 7, 8, 3, 17, 15, 13, 2, 12, 4, 1, 18, 9, 14, 6, 5, 19, 0,
11])
```

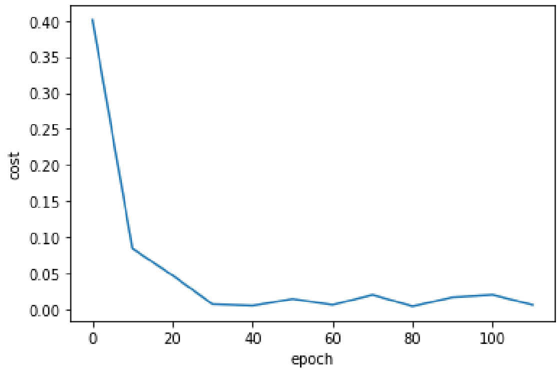
(3) Mini Batch Gradient Descent Implementation

```
def mini_batch_gradient_descent(X, y_true, epochs = 100, batch_size = 5, l
earning_rate = 0.01):

    number_of_features = X.shape[1]
    # numpy array with 1 row and columns equal to number of features. In
    # our case number_of_features = 3 (area, bedroom and age)
    w = np.ones(shape=(number_of_features))
    b = 0
    total_samples = X.shape[0] # number of rows in X

    if batch_size > total_samples: # In this case mini batch becomes same
as batch gradient descent
        batch_size = total_samples

    cost_list = []
    epoch_list = []

    num_batches = int(total_samples/batch_size)

    for i in range(epochs):
        random_indices = np.random.permutation(total_samples)
        X_tmp = X[random_indices]
        y_tmp = y_true[random_indices]

        for j in range(0,total_samples,batch_size):
            Xj = X_tmp[j:j+batch_size]
            yj = y_tmp[j:j+batch_size]
            y_predicted = np.dot(w, Xj.T) + b

            w_grad = -(2/len(Xj))*(Xj.T.dot(yj-y_predicted))
            b_grad = -(2/len(Xj))*np.sum(yj-y_predicted)

            w = w - learning_rate * w_grad
            b = b - learning_rate * b_grad

            cost = np.mean(np.square(yj-
y_predicted)) # MSE (Mean Squared Error)

        if i%10==0:
            cost_list.append(cost)
            epoch_list.append(i)

    return w, b, cost, cost_list, epoch_list

w, b, cost, cost_list, epoch_list = mini_batch_gradient_descent(
    scaled_X,
    scaled_y.reshape(scaled_y.shape[0],),
    epochs = 120,
    batch_size = 5
)
w, b, cost
```

Output:
```
(array([0.71015977, 0.67813327]), -0.23343143725261098,
0.0025898311387083403)
```

```
plt.xlabel("epoch")
plt.ylabel("cost")
plt.plot(epoch_list,cost_list)
```

Output:



```
def predict(area,bedrooms,w,b):
    scaled_X = sx.transform([[area, bedrooms]])[0]
    # here w1 = w[0] , w2 = w[1], w3 = w[2] and bias is b
    # equation for price is w1*area + w2*bedrooms + w3*age + bias
    # scaled_X[0] is area
    # scaled_X[1] is bedrooms
    # scaled_X[2] is age
    scaled_price = w[0] * scaled_X[0] + w[1] * scaled_X[1] + b
    # once we get price prediction we need to to rescal it back to origina
l value
    # also since it returns 2D array, to get single value we need to do va
lue[0][0]
    return sy.inverse_transform([[scaled_price]])[0][0]

predict(2600,4,w,b)
```

Output:
```
128.65424087579652
```

```
predict(1000,2,w,b)
```

```
Output:
29.9855861683337
```

Exercise:

Implement Gradient Descent for Neural Network (or Logistic Regression) by Predicting if a person would buy life insurance based on his age.

Solution:

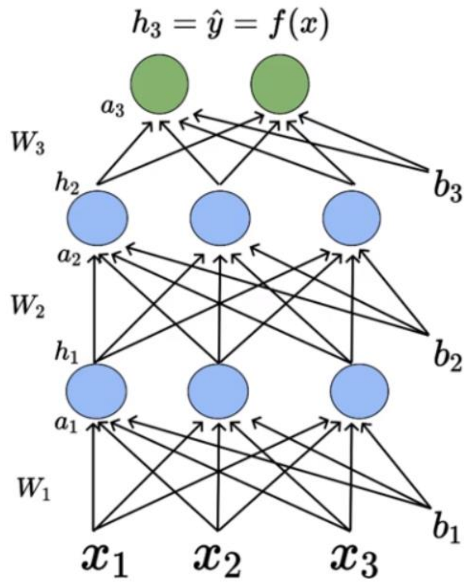
5. Study the effect of batch normalization and dropout in neural network classifier PCA.

Introduction

In this lab exercise, we will discuss why we need batch normalization and dropout in deep neural networks followed by experiments using Pytorch on a standard data set to see the effects of batch normalization and dropout.

Batch Normalization

By normalizing the inputs we are able to bring all the inputs features to the same scale. In the neural network, we need to compute the pre-activation for the first neuron of the first layer a_{11}. We know that pre-activation is nothing but the weighted sum of inputs plus bias. In other words, it is the dot product between the first row of the weight matrix W_1 and the input matrix X plus bias b_{11}.



The mathematical equation for pre-activation at each layer 'i' is given by,

$$a_1 = W_1 * x + b_1$$

The activation at each layer is equal to applying the activation function to the output of the pre-activation of that layer. The mathematical equation for the activation at each layer 'i' is given by,

$$h_i(x) = g(a_i(x))$$

where 'g' is called as the activation function

Activation Function

In order to bring all the activation values to the same scale, we normalize the activation values such that the hidden representation doesn't vary drastically and also helps us to get improvement in the training speed.

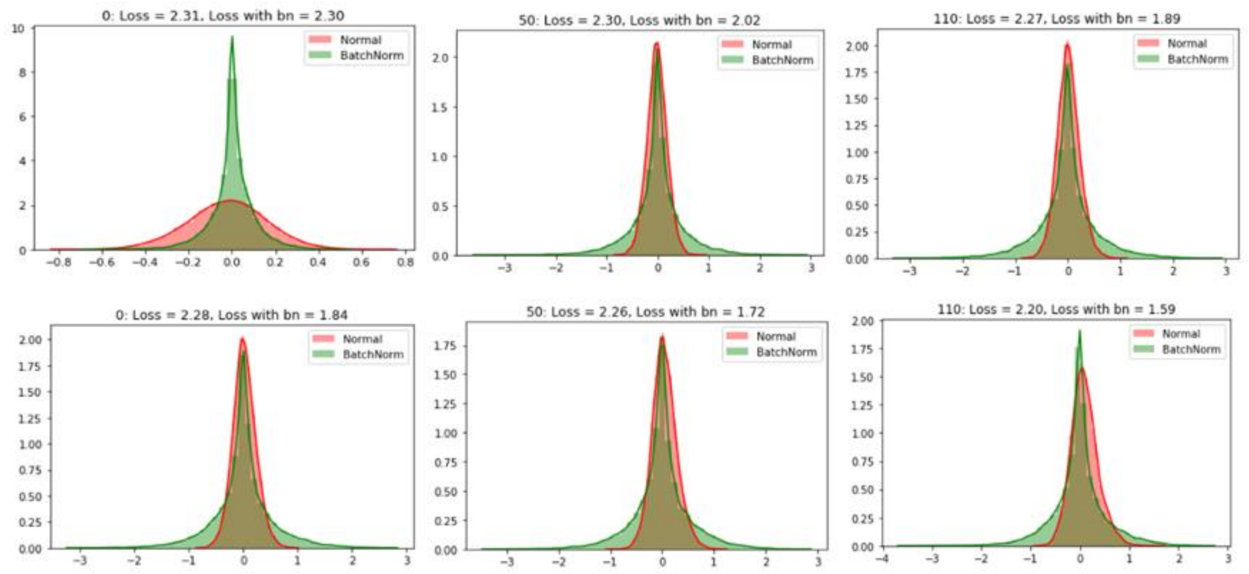
Why is it called batch normalization?

Since we are computing the mean and standard deviation from a single batch as opposed to computing it from the entire data. Batch normalization is done individually at each hidden neuron in the network.

$$h_{ij}^{norm} = \frac{h_{ij} - \mu_j}{\sigma_j}$$

To get a better insight into how batch normalization helps in faster converge of the network, we will look at the distribution of values across multiple hidden layers in the network during the training phase.

For consistency, we will plot the output of the second linear layer from the two networks and compare the distributions of the output from that layer across the networks. The results look like this:



From the graphs, we can conclude that the distribution of values without batch normalization has changed significantly between iterations of inputs within each epoch which means that the subsequent layers in the network without batch normalization are seeing a varying distribution of input data. But the change in the distribution of values for the model with batch normalization seems to be slightly negligible.

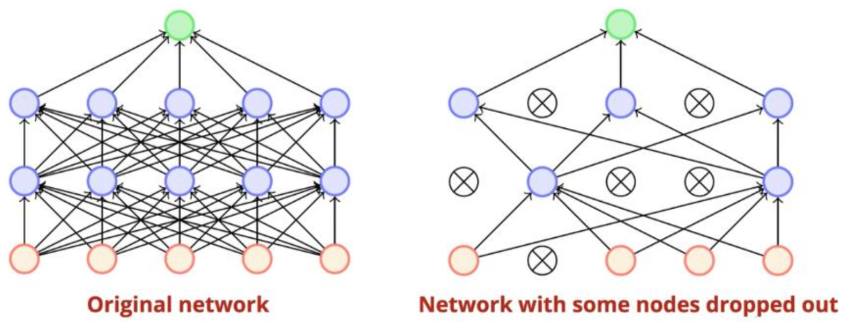
Dropout

In this section of the lab, we discuss the concept of dropout in neural networks specifically how it helps to reduce overfitting and generalization error. After that, we will implement a neural network with and without dropout to see how dropout influences the performance of a network using Pytorch.

Dropout is a regularization technique that “drops out” or “deactivates” few neurons in the neural network randomly in order to avoid the problem of overfitting.

The idea of Dropout

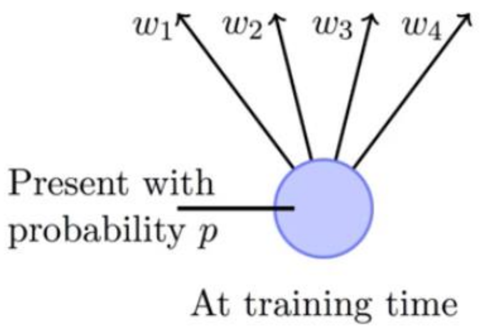
Training one deep neural network with large parameters on the data might lead to overfitting. Can we train multiple neural networks with different configurations on the same dataset and take the average value of these predictions?.



But creating an ensemble of neural networks with different architectures and training them wouldn’t be feasible in practice. Dropout to the rescue.

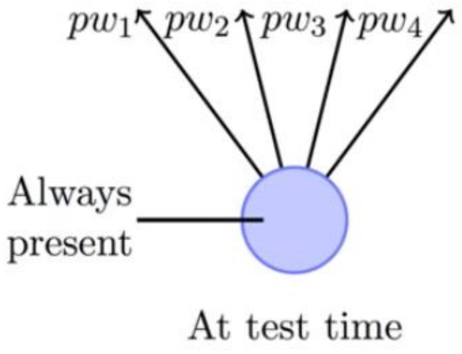
Dropout deactivates the neurons randomly at each training step instead of training the data on the original network, we train the data on the network with dropped out nodes. In the next iteration of the training step, the hidden neurons which are deactivated by dropout changes because of its probabilistic behavior. In this way, by applying dropout i.e…deactivating certain individual nodes at random during training we can simulate an ensemble of neural network with different architectures.

Dropout at Training time



In each training iteration, each node in the network is associated with a probability p whether to keep in the network or to deactivate it (dropout) out of the network with probability 1-p. That means the weights associated with the nodes got updated only p fraction of times because nodes are active only p times during training.

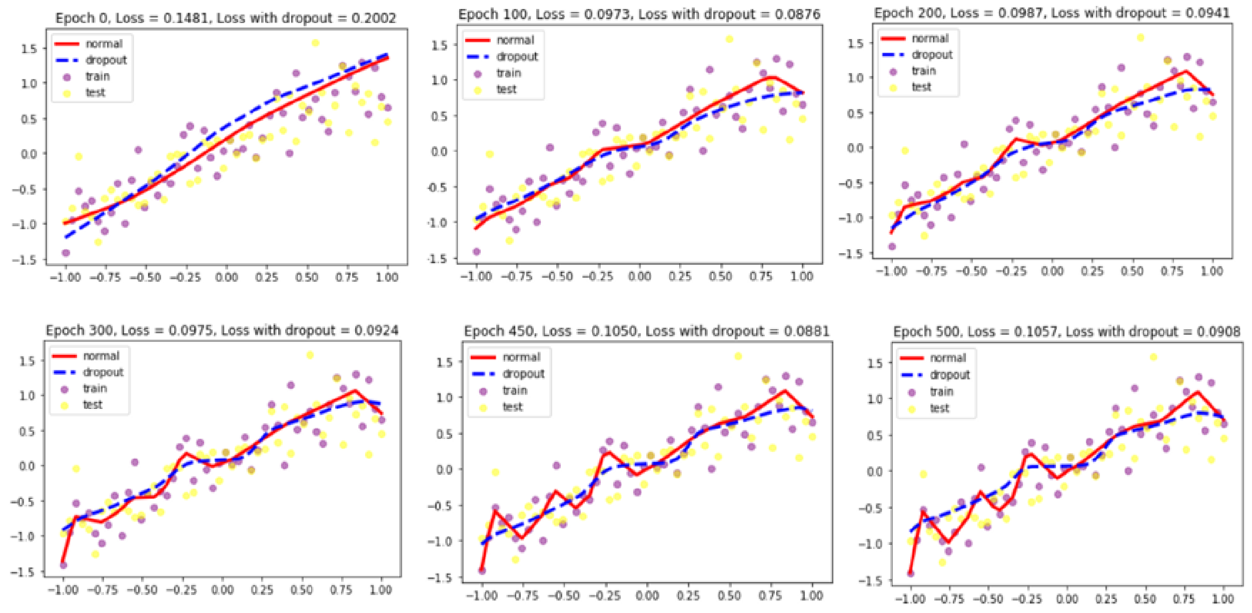
Dropout at Test time



During test time, we consider the original neural network with all activations present and scale the output of each node by a value p. Since each node is activated the only p times.

To show the overfitting, we will train two networks — one without dropout and another with dropout. The network without dropout has 3 fully connected hidden layers with ReLU as the activation function for the hidden layers and the network with dropout also has similar architecture but with dropout applied after first & second Linear layer.

In this example, I have used a dropout fraction of 0.5 after the first linear layer and 0.2 after the second linear layer. Once we train the two different models i.e…one without dropout and another with dropout and plot the test results, it would look like this:



From the above graphs, we can conclude that as we increase the number of epochs the model without dropout is overfitting the data. The model without dropout is learning the noise associated with the data instead of generalizing for the data. We can see that the loss associated with the model without drop increases as we increase the number of epochs unlike the loss associated with the model with dropout.

Example:

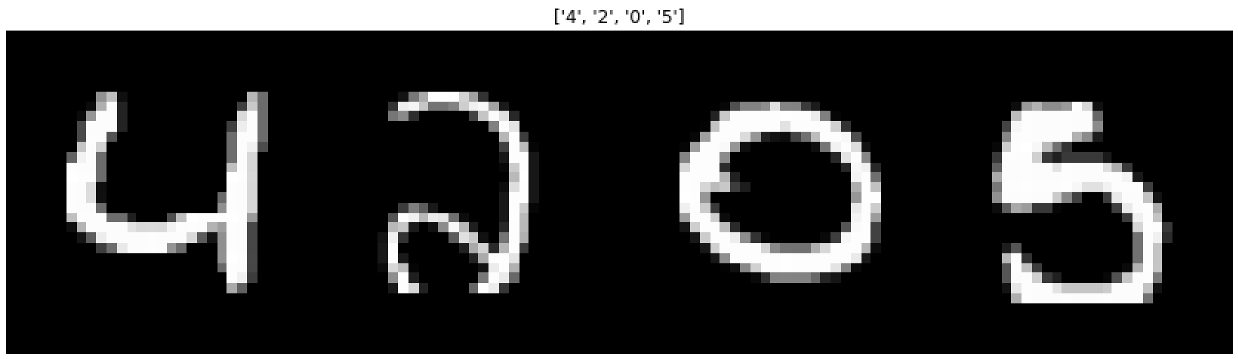
Outline

1. Load dataset and visualise
2. Add batch normalization layers
3. Comparison with batch normalization layers
4. Add dropout layer
5. Comparison with dropout layer

```
import torch

import matplotlib.pyplot as plt
import numpy as np

import torchvision
import torchvision.transforms as transforms

import torch.nn as nn
import torch.optim as optim
import seaborn as sns
```

Dataset and visualisation

```
trainset = torchvision.datasets.MNIST(root='./data', train=True,
                                        download=True,
                                        transform=transforms.ToTensor())
def imshow(img, title):

    plt.figure(figsize=(batch_size * 4, 4))
    plt.axis('off')
    plt.imshow(np.transpose(img, (1, 2, 0)))
    plt.title(title)
    plt.show()
def show_batch_images(dataloader):
    images, labels = next(iter(dataloader))

    img = torchvision.utils.make_grid(images)
    imshow(img, title=[str(x.item()) for x in labels])

    return images, labels
images, labels = show_batch_images(trainloader)
```

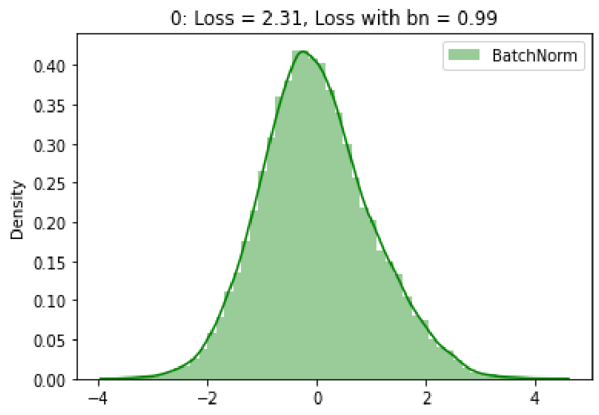


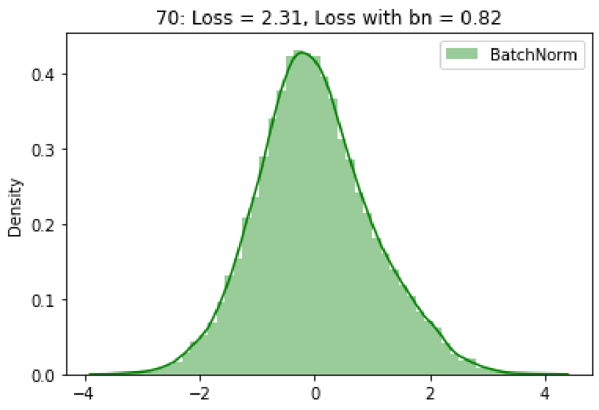
Batch Normalisation

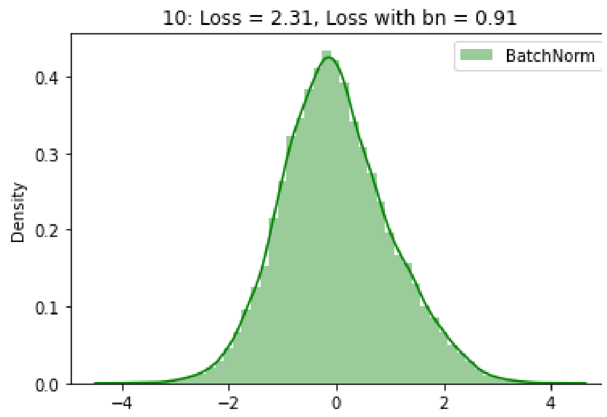
```
class MyNetBN(nn.Module):
    def __init__(self):
        super(MyNetBN, self).__ini
t__()
        self.classifier = nn.Seque
ntial(
            nn.Linear(784, 48),
            nn.BatchNorm1d(48),
            nn.ReLU(),
            nn.Linear(48, 24),
            nn.BatchNorm1d(24),
            nn.ReLU(),
            nn.Linear(24, 10)
        )

    def forward(self, x):
        x = x.view(x.size(0), -1)
        x = self.classifier(x)
```

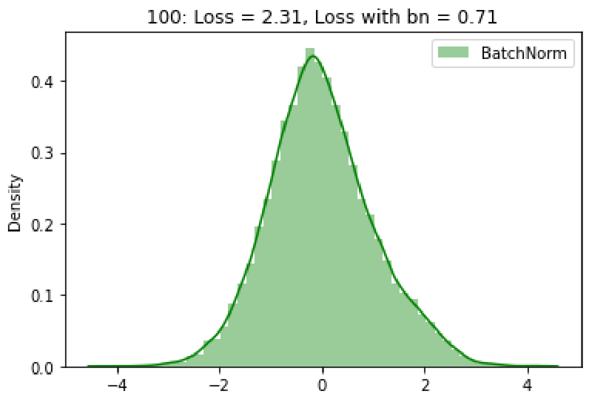
```
return x
model_bn = MyNetBN()
print(model_bn)
batch_size = 512
trainloader = torch.utils.data.Dat
aLoader(trainset, batch_size=batch
_size, shuffle=True)
loss_fn = nn.CrossEntropyLoss()
opt_bn = optim.SGD(model_bn.parame
ters(), lr=0.01)


oss_arr = []
loss_bn_arr = []

max_epochs = 2

for epoch in range(max_epochs):
for i, data in enumerate(trainload
er, 0):

inputs, labels = data

# training steps for bn model
opt_bn.zero_grad()
outputs_bn = model_bn(inputs)
loss_bn = loss_fn(outputs_bn, labe
ls)
loss_bn.backward()
opt_bn.step()

loss_bn_arr.append(loss_bn.item())

if i % 10 == 0:

inputs = inputs.view(inputs.size(0
), -1)


model_bn.eval()



b = model_bn.classifier[0](inputs)
b = model_bn.classifier[1](b)
b = b.detach().numpy().ravel()

sns.distplot(b, kde=True, color='g
', label='BatchNorm')
plt.title('%d: Loss = %0.2f, Loss
with bn = %0.2f' % (i, loss.item()
, loss_bn.item()))
plt.legend()
plt.show()
plt.pause(0.5)


model_bn.train()


print('----------------------')


plt.plot(loss_bn_arr, 'g', label='
BatchNorm')
plt.legend()
plt.show()
```

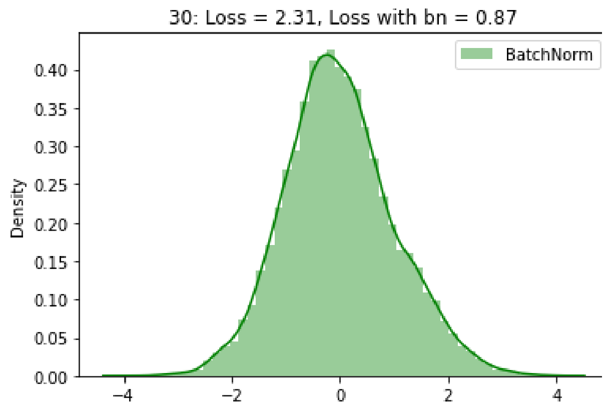
Outputs:

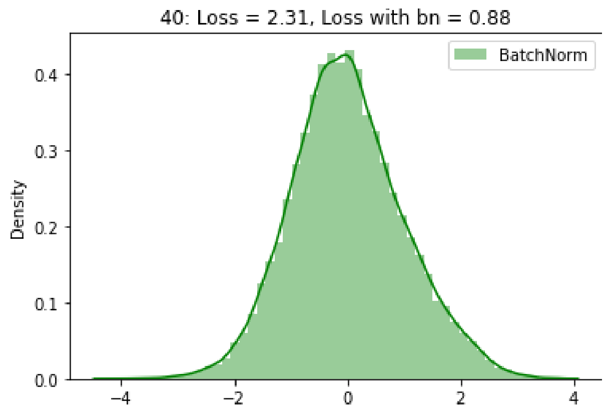

0: Loss = 2.31, Loss with bn = 0.99
BatchNorm
Density

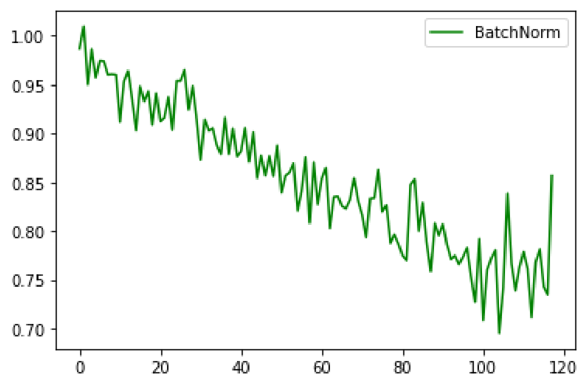

70: Loss = 2.31, Loss with bn = 0.82
BatchNorm
Density

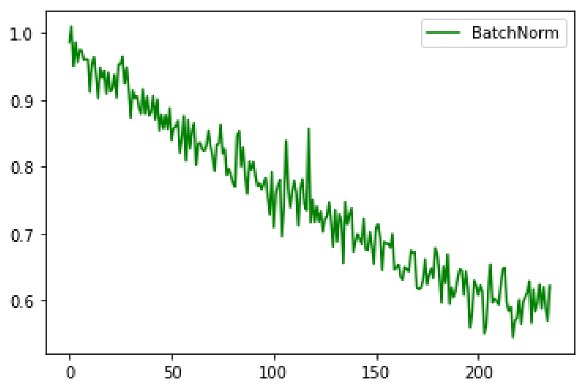

10: Loss = 2.31, Loss with bn = 0.91
BatchNorm
Density

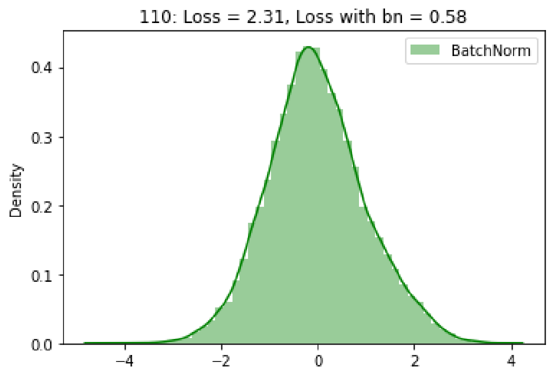

100: Loss = 2.31, Loss with bn = 0.71
BatchNorm
Density


30: Loss = 2.31, Loss with bn = 0.87
BatchNorm
Density


40: Loss = 2.31, Loss with bn = 0.88
BatchNorm
Density







Exercise

Write down the dropout for the above program

6. Study of Singular value Decomposition for dimensionality reduction.

Introduction:

Reducing the number of input variables for a predictive model is referred to as dimensionality reduction. Fewer input variables can result in a simpler predictive model that may have better performance when making predictions on new data.
Perhaps the more popular technique for dimensionality reduction in machine learning is Singular Value Decomposition, or SVD for short. This is a technique that comes from the field of linear algebra and can be used as a data preparation technique to create a projection of a sparse dataset prior to fitting a model.

Dimensionality Reduction and SVD

Dimensionality reduction refers to reducing the number of input variables for a dataset. If your data is represented using rows and columns, such as in a spreadsheet, then the input variables are the columns that are fed as input to a model to predict the target variable. Input variables are also called features.
We can consider the columns of data representing dimensions on an n-dimensional feature space and the rows of data as points in that space. This is a useful geometric interpretation of a dataset. Having a large number of dimensions in the feature space can mean that the volume of that space is very large, and in turn, the points that we have in that space (rows of data) often represent a small and non-representative sample.
This can dramatically impact the performance of machine learning algorithms fit on data with many input features, generally referred to as the “curse of dimensionality.” Therefore, it is often desirable to reduce the number of input features. This reduces the number of dimensions of the feature space, hence the name “dimensionality reduction.”
A popular approach to dimensionality reduction is to use techniques from the field of linear algebra. This is often called “*feature projection*” and the algorithms used are referred to as “*projection methods*.” Projection methods seek to reduce the number of dimensions in the feature space whilst also preserving the most important structure or relationships between the variables observed in the data.

Singular Value Decomposition, or SVD, might be the most popular technique for dimensionality reduction when data is sparse. Sparse data refers to rows of data where many of the values are zero. This is often the case in some problem domains like recommender systems where a user has a rating for very few movies or songs in the database and zero ratings for all other cases. Another common example is a bag of words model of a text document, where the document has a count or frequency for some words and most words have a 0 value.

Examples of sparse data appropriate for applying SVD for dimensionality reduction:

- Recommender Systems
- Customer-Product purchases
- User-Song Listen Counts
- User-Movie Ratings
- Text Classification

- One Hot Encoding
- Bag of Words Counts
- TF/IDF

SVD can be thought of as a projection method where data with m-columns (features) is projected into a subspace with m or fewer columns, whilst retaining the essence of the original data.

The SVD is used widely both in the calculation of other matrix operations, such as matrix inverse, but also as a data reduction method in machine learning.

Example and Code:

```
# compare svd number of components with logistic regression algorithm for classification
from numpy import mean
from numpy import std
from sklearn.datasets import make_classification
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RepeatedStratifiedKFold
from sklearn.pipeline import Pipeline
from sklearn.decomposition import TruncatedSVD
from sklearn.linear_model import LogisticRegression
from matplotlib import pyplot

# get the dataset
def get_dataset():
	X, y = make_classification(n_samples=1000, n_features=20, n_informative=15,
n_redundant=5, random_state=7)
	return X, y

# get a list of models to evaluate
def get_models():
	models = dict()
	for i in range(1,20):
		steps = [('svd', TruncatedSVD(n_components=i)), ('m', LogisticRegression())]
		models[str(i)] = Pipeline(steps=steps)
	return models

# evaluate a give model using cross-validation
def evaluate_model(model, X, y):
	cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
	scores = cross_val_score(model, X, y, scoring='accuracy', cv=cv, n_jobs=-1,
error_score='raise')
	return scores

# define dataset
X, y = get_dataset()
# get the models to evaluate
models = get_models()
# evaluate the models and store results
```

```
results, names = list(), list()
for name, model in models.items():
        scores = evaluate_model(model, X, y)
        results.append(scores)
        names.append(name)
        print('>%s %.3f (%.3f)' % (name, mean(scores), std(scores)))
# plot model performance for comparison
pyplot.boxplot(results, labels=names, showmeans=True)
pyplot.xticks(rotation=45)
pyplot.show()
```

Output:

```
>1 0.542 (0.046)
>2 0.626 (0.050)
>3 0.719 (0.053)
>4 0.722 (0.052)
>5 0.721 (0.054)
>6 0.729 (0.045)
>7 0.802 (0.034)
>8 0.800 (0.040)
>9 0.814 (0.037)
>10 0.814 (0.034)
>11 0.817 (0.037)
>12 0.820 (0.038)
>13 0.820 (0.036)
>14 0.825 (0.036)
>15 0.865 (0.027)
>16 0.865 (0.027)
>17 0.865 (0.027)
>18 0.865 (0.027)
>19 0.865 (0.027)
```

Exercise: Draw the whisker plot for the above solution for creating the distribution of accuracy scores for each configured number of dimensions and also find out the predicted class by using combination of SVD transform and logistic regression model.

Solution:

7. Train a sentiment analysis model on IMDB dataset, use RNN layers.

Sentiment analysis probably is one the most common applications in Natural Language processing. I don’t have to emphasize how important customer service tool sentiment analysis has become. So here we are, we will train a classifier movie reviews in IMDB data set, using Recurrent Neural Networks. Recurrent Neural Networks (RNNs) are a type of neural network that are well suited for natural language processing tasks, such as sentiment analysis. To perform sentiment analysis on IMDB movie reviews using an RNN, you would need to first pre-process the text data by tokenizing the reviews and creating numerical representations of the words, such as word embeddings. Then you would train the RNN on the pre-processed data, using the review text as input and the corresponding sentiment (positive or negative) as the output. Once the model is trained, you can use it to predict the sentiment of new reviews by inputting the text into the trained model and interpreting the output.

A recurrent neural network (RNN) is a class of artificial neural networks where connections between nodes form a directed graph along a temporal sequence. This allows it to exhibit temporal dynamic behavior. Derived from feedforward neural networks, RNNs can use their internal state (memory) to process variable length sequences of inputs. This makes them applicable to tasks such as unsegmented, connected handwriting recognition or speech recognition.

The term “recurrent neural network” is used indiscriminately to refer to two broad classes of networks with a similar general structure, where one is finite impulse and the other is infinite impulse. Both classes of networks exhibit temporal dynamic behavior. A finite impulse recurrent network is a directed acyclic graph that can be unrolled and replaced with a strictly feedforward neural network, while an infinite impulse recurrent network is a directed cyclic graph that cannot be unrolled.

Both finite impulse and infinite impulse recurrent networks can have additional stored states, and the storage can be under direct control by the neural network. The storage can also be replaced by another network or graph, if that incorporates time delays or has feedback loops. Such controlled states are referred to as gated state or gated memory, and are part of long short-term memory networks (LSTMs) and gated recurrent units. This is also called Feedback Neural Network (FNN).

Sentiment analysis on the IMDB dataset using a Recurrent Neural Network (RNN) in Python:

Sentiment analysis is the process of determining the sentiment or emotion expressed in a piece of text, such as a review or a tweet. One way to perform sentiment analysis is by using a Recurrent Neural Network (RNN) in Python. Here is an example of how to do sentiment analysis on the IMDB dataset using an RNN:

```
import numpy as np
from keras.datasets import imdb
from keras.preprocessing import sequence
from keras.models import Sequential
from keras.layers import Embedding, LSTM, Dense
```

```
# Load the IMDB dataset
(x_train, y_train), (x_test, y_test) = imdb.load_data(num_words=5000)

# Pad the sequences to a fixed length
max_length = 500
x_train = sequence.pad_sequences(x_train, maxlen=max_length)
x_test = sequence.pad_sequences(x_test, maxlen=max_length)

# Define the model architecture
model = Sequential()
model.add(Embedding(5000, 128, input_length=max_length))

model.add(LSTM(128, dropout=0.2, recurrent_dropout=0.2))
model.add(Dense(1, activation='sigmoid'))

# Compile the model
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])

# Train the model
model.fit(x_train, y_train, batch_size=32, epochs=15, validation_data=(x_test, y_test))
```

In this example, we first load the IMDB dataset using the **imdb.load_data** function. The dataset consists of 25,000 labeled movie reviews for training and 25,000 for testing. The reviews are already preprocessed and encoded as sequences of integers, where each integer represents a specific word in a vocabulary of 5,000 words.
Next, we use the **sequence.pad_sequences** function to pad the sequences to a fixed length of 500, to make sure that all the input sequences have the same length.
Then, we define the architecture of the RNN model using the Keras library. The model includes an Embedding layer, which maps the input sequences to a high-dimensional space, an LSTM layer, which processes the input sequences, and a Dense layer, which is used for classification.
After that, we compile the model by specifying the loss function, the optimizer, and the evaluation metric.
Finally, we train the model using the **fit** function. The training process consists of multiple iterations over the training data, called epochs, and at the end of each epoch, the model's performance is evaluated using the validation data (x_test, y_test).
Once the model is trained, it can be used to classify new reviews as positive or negative by calling the **predict** method and passing in the review as a padded sequence of integers.

Exercise:
Perform sentiment analysis using a Recurrent Neural Network (RNN) with python code using simple data set.

8. Perform Object detection USING CNN.

Object Detection is the process of finding real-world object instances like car, bike, TV, flowers, and humans in still images or Videos. It allows for the recognition, localization, and detection of multiple objects within an image which provides us with a much better understanding of an image as a whole. It is commonly used in applications such as image retrieval, security, surveillance, and advanced driver assistance systems (ADAS). Image classification is straight forward, but the differences between object localization and object detection can be confusing, especially when all three tasks may be just as equally referred to as object recognition.

Image classification involves assigning a class label to an image, whereas object localization involves drawing a bounding box around one or more objects in an image. Object detection is more challenging and combines these two tasks and draws a bounding box around each object of interest in the image and assigns them a class label. Together, all of these problems are referred to as object recognition.

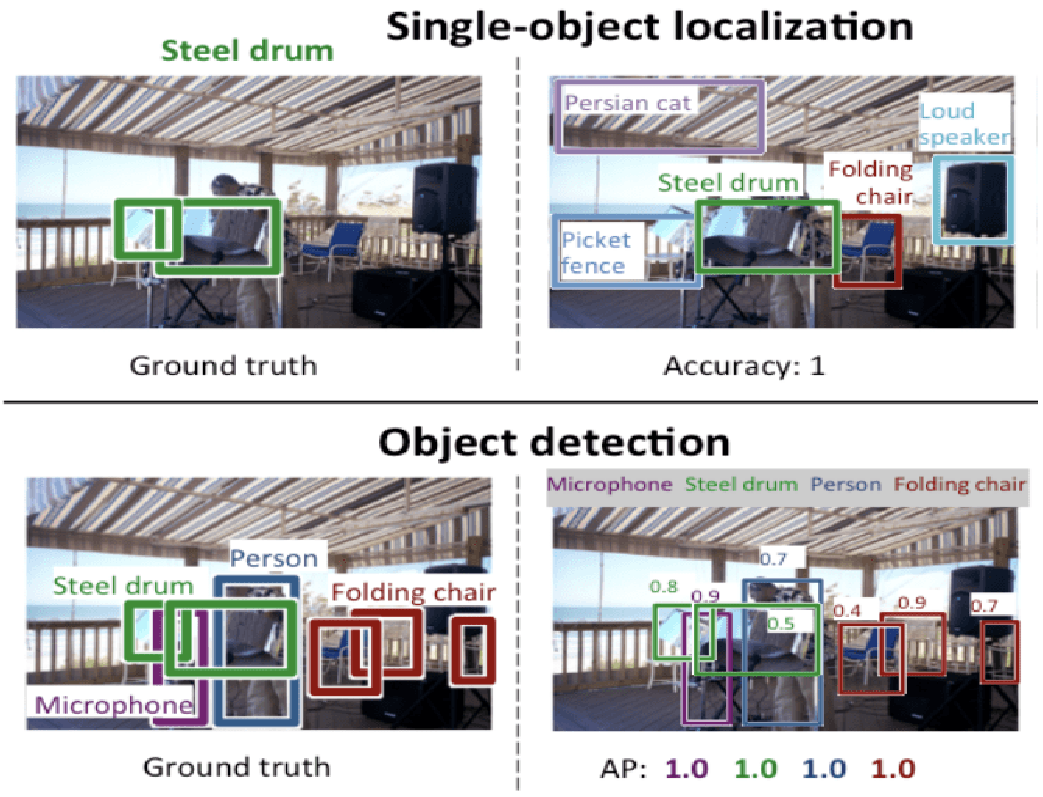
As such, we can distinguish between these three computer vision tasks:

- Image Classification: Predict the type or class of an object in an image.
    - *Input*: An image with a single object, such as a photograph.
    - *Output*: A class label (e.g. one or more integers that are mapped to class labels).
- Object Localization: Locate the presence of objects in an image and indicate their location with a bounding box.
    - *Input*: An image with one or more objects, such as a photograph.
    - *Output*: One or more bounding boxes (e.g. defined by a point, width, and height).
- Object Detection: Locate the presence of objects with a bounding box and types or classes of the located objects in an image.
    - *Input*: An image with one or more objects, such as a photograph.
    - *Output*: One or more bounding boxes (e.g. defined by a point, width, and height), and a class label for each bounding box.

Object Detection can be done via multiple ways:

- Feature-Based Object Detection
- Viola Jones Object Detection
- SVM Classifications with HOG Features
- Deep Learning Object Detection

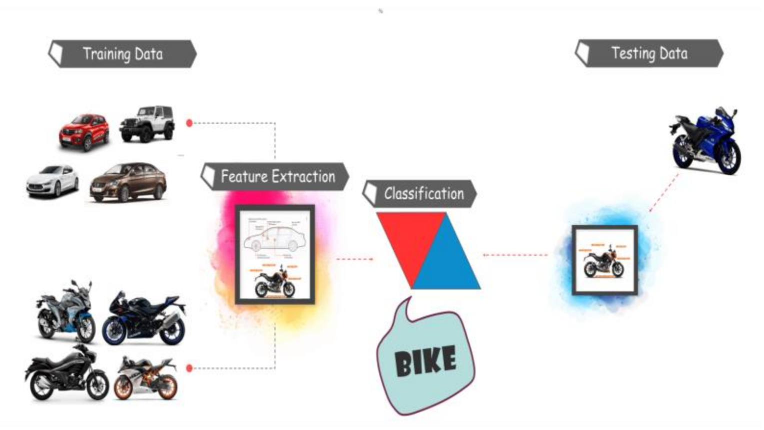
Below is an example comparing single object localization and object detection, taken from the ILSVRC paper. Note the difference in ground truth expectations in each case.



Object Detection Workflow

Every Object Detection Algorithm has a different way of working, but they all work on the same principle.

Feature Extraction: They extract features from the input images at hands and use these features to determine the class of the image. Be it through MatLab, Open CV, Viola Jones or Deep Learning.



Example: Object Identification using CNN on COCO dataset

SOLUTION:

Setting up the Environment

Now to Download TensorFlow and TensorFlow GPU you can use pip or conda commands:

```
# For CPU
pip install tensorflow
# For GPU
pip install tensorflow-gpu
```

For all the other libraries we can use pip or conda to install them. The code is provided below

```
pip install --user Cython
pip install --user contextlib2
pip install --user pillow
pip install --user lxml
pip install --user jupyter
pip install --user matplotlib
```

Next, we have Protobuf: Protocol Buffers (Protobuf) are Google's language-neutral, platform-neutral, extensible mechanism for serializing structured data, – think of it like XML, but smaller, faster, and simpler. You need to Download Protobuf version 3.4 or above for this demo and extract it.

Now you need to Clone or Download TensorFlow's Model from Github. Once downloaded and extracted rename the "models-masters" to just "models".

Now for simplicity, we are going to keep "models" and "protobuf" under one folder "Tensorflow".

Next, we need to go inside the Tensorflow folder and then inside research folder and run protobuf from there using this command:

```
"path_of_protobuf's bin"./bin/protoc object_detection/protos/
```

To check whether this worked or not, you can go to the protos folder inside models>object_detection>protos and there you can see that for every proto file there's one python file created.

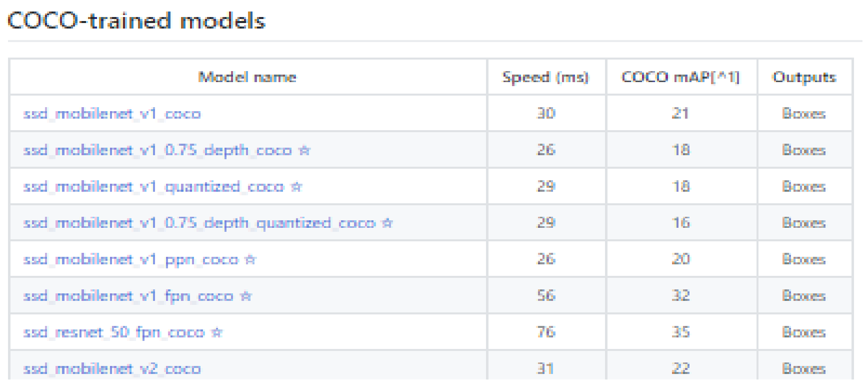
First of all, we need to import all the libraries

```
import numpy as np
import os
import six.moves.urllib as urllib
import sys
import tarfile
import tensorflow as tf
import zipfile
```

```
 from collections import defaultdict
from io import StringIO
from matplotlib import pyplot as plt
from PIL import Image
 sys.path.append("..")
from object_detection.utils import ops as utils_ops
 from utils import label_map_util
 from utils import visualization_utils as vis_util
```

Next, we will download the model which is trained on the COCO dataset. COCO stands for Common Objects in Context, this dataset contains around 330K labeled images. Now the model selection is important as you need to make an important tradeoff between Speed and Accuracy. Depending upon your requirement and the system memory, the correct model must be selected.

Inside “models>research>object_detection>g3doc>detection_model_zoo” contains all the models with different speed and accuracy(mAP).



COCO-trained models

Model name	Speed (ms)	COCO mAP[^1]	Outputs
ssd_mobilenet_v1_coco	30	21	Boxes
ssd_mobilenet_v1_0.75_depth_coco ☆	26	18	Boxes
ssd_mobilenet_v1_quantized_coco ☆	29	18	Boxes
ssd_mobilenet_v1_0.75_depth_quantized_coco ☆	29	16	Boxes
ssd_mobilenet_v1_ppn_coco ☆	26	20	Boxes
ssd_mobilenet_v1_fpn_coco ☆	56	32	Boxes
ssd_resnet_50_fpn_coco ☆	76	35	Boxes
ssd_mobilenet_v2_coco	31	22	Boxes

Next, we provide the required model and the frozen inference graph generated by Tensorflow to use.

```
MODEL_NAME = 
'ssd_mobilenet_v1_coco_2017_11_17'
MODEL_FILE = MODEL_NAME + '.tar.gz'
DOWNLOAD_BASE = '<a
href="http://download.tensorflow.org/m
odels/object_detection/">http://downloa
d.tensorflow.org/models/object_detectio
n/</a>'
 PATH_TO_CKPT = MODEL_NAME +
'/frozen_inference_graph.pb'
 PATH_TO_LABELS = os.path.join('data',
'mscoco_label_map.pbtxt')
 NUM_CLASSES = 90
```

This code will download that model from the internet and extract the frozen inference graph of that model.

```
opener = urllib.request.URLopener()
opener.retrieve(DOWNLOAD_BASE +
MODEL_FILE, MODEL_FILE)
tar_file = tarfile.open(MODEL_FILE)
for file in tar_file.getmembers():
 file_name =
os.path.basename(file.name)
 if 'frozen_inference_graph.pb' in
file_name:
   tar_file.extract(file, os.getcwd())
 detection_graph = tf.Graph()
```

```
with detection_graph.as_default():
  od_graph_def = tf.GraphDef()
  with tf.gfile.GFile(PATH_TO_CKPT, 'rb') as
fid:
    serialized_graph = fid.read()

od_graph_def.ParseFromString(serialized
_graph)
    tf.import_graph_def(od_graph_def,
name='')
```

Next, we are going to load all the labels

```
label_map =
label_map_util.load_labelmap(PATH_TO_
LABELS)
categories=label_map_util.convert_label_
map_to_categories(label_map,
max_num_classes=NUM_CLASSES,
use_display_name=True)
category_index =
label_map_util.create_category_index(cat
egories)
```

Now we will convert the images data into a numPy array for processing.

```
def
load_image_into_numpy_array(image):
  (im_width, im_height) = image.size
  return
np.array(image.getdata()).reshape(
    (im_height, im_width,
3)).astype(np.uint8)
```

The path to the images for the testing purpose is defined here. Here we have a naming convention “image[i]” for i in (1 to n+1), n being the number of images provided.

```
PATH_TO_TEST_IMAGES_DIR =
'test_images'
TEST_IMAGE_PATHS = [
os.path.join(PATH_TO_TEST_IMAGES_DIR,
'image{}.jpg'.format(i))
```

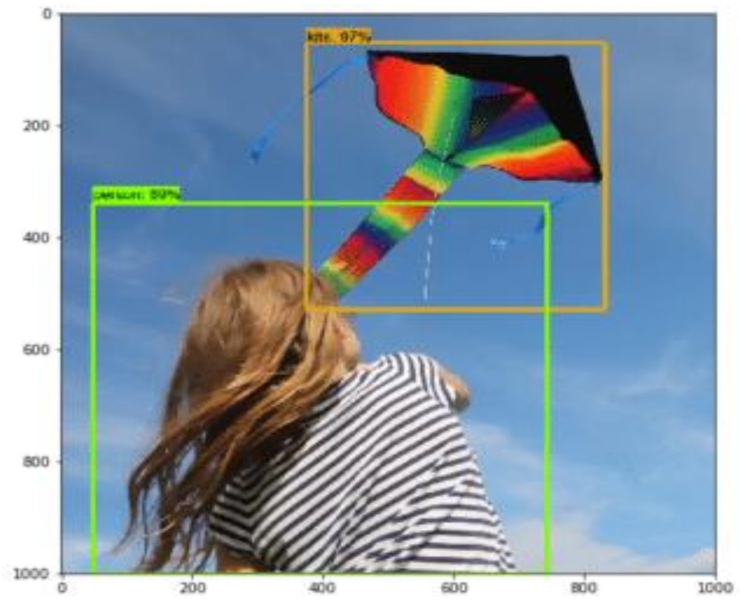
This code runs the inference for a single image, where it detects the objects, make boxes and provide the class and the class score of that particular object.

```
def
run_inference_for_single_image(image,
graph):
  with graph.as_default():
    with tf.Session() as sess:
    # Get handles to input and output
tensors
      ops =
tf.get_default_graph().get_operations()
      all_tensor_names = {output.name for
op in ops for output in op.outputs}
      tensor_dict = {}
      for key in [
          'num_detections', 'detection_boxes',
'detection_scores',
          'detection_classes',
'detection_masks'
     ]:
        tensor_name = key + ':0'
        if tensor_name in all_tensor_names:
          tensor_dict[key] =
tf.get_default_graph().get_tensor_by_na
me(
            tensor_name)
      if 'detection_masks' in tensor_dict:
        # The following processing is only for
single image
        detection_boxes =
tf.squeeze(tensor_dict['detection_boxes'],
[0])
        detection_masks =
tf.squeeze(tensor_dict['detection_masks']
, [0])
        # Reframe is required to translate
mask from box coordinates to image
coordinates and fit the image size.
        real_num_detection =
tf.cast(tensor_dict['num_detections'][0],
tf.int32)
```

```
    detection_boxes =
tf.slice(detection_boxes, [0, 0],
[real_num_detection, -1])
    detection_masks =
tf.slice(detection_masks, [0, 0, 0],
[real_num_detection, -1, -1])
    detection_masks_reframed =
utils_ops.reframe_box_masks_to_image_
masks(
      detection_masks, detection_boxes,
image.shape[0], image.shape[1])
      detection_masks_reframed =
tf.cast(

tf.greater(detection_masks_reframed,
0.5), tf.uint8)
    # Follow the convention by adding
back the batch dimension
      tensor_dict['detection_masks'] =
tf.expand_dims(
      detection_masks_reframed, 0)
    image_tensor =
tf.get_default_graph().get_tensor_by_na
me('image_tensor:0')
     # Run inference
      output_dict = sess.run(tensor_dict,
      feed_dict={image_tensor:
np.expand_dims(image, 0)})
     # all outputs are float32 numpy
arrays, so convert types as appropriate
    output_dict['num_detections'] =
int(output_dict['num_detections'][0])
    output_dict['detection_classes'] =
output_dict[

'detection_classes'][0].astype(np.uint8)
    output_dict['detection_boxes'] =
output_dict['detection_boxes'][0]
    output_dict['detection_scores'] =
output_dict['detection_scores'][0]
    if 'detection_masks' in output_dict:
    output_dict['detection_masks'] =
output_dict['detection_masks'][0]
    return output_dict
```

Our Final loop, which will call all the functions defined above and will run the inference on all the input images one by one, which will provide us the output of images in which objects are detected with labels and the percentage/score of that object being similar to the training data.

```
for image_path in TEST_IMAGE_PATHS:
 image = Image.open(image_path)
 # the array based representation of the
image will be used later in order to
prepare the
 # result image with boxes and labels on
it.
 image_np =
load_image_into_numpy_array(image)
 # Expand dimensions since the model
expects images to have shape: [1, None,
None, 3]
 image_np_expanded =
np.expand_dims(image_np, axis=0)
 # Actual detection.
 output_dict =
run_inference_for_single_image(image_n
p, detection_graph)
 # Visualization of the results of a
detection.

vis_util.visualize_boxes_and_labels_on_i
mage_array(
   image_np,
   output_dict['detection_boxes'],
   output_dict['detection_classes'],
   output_dict['detection_scores'],
   category_index,

instance_masks=output_dict.get('detectio
n_masks'),
   use_normalized_coordinates=True,
   line_thickness=8)
   plt.figure(figsize=IMAGE_SIZE)
   plt.imshow(image_np)
```



Exercise:

Objective: Implement object detection using CNN using The GTSRB dataset consists of 43 traffic sign classes and nearly 50,000 images.

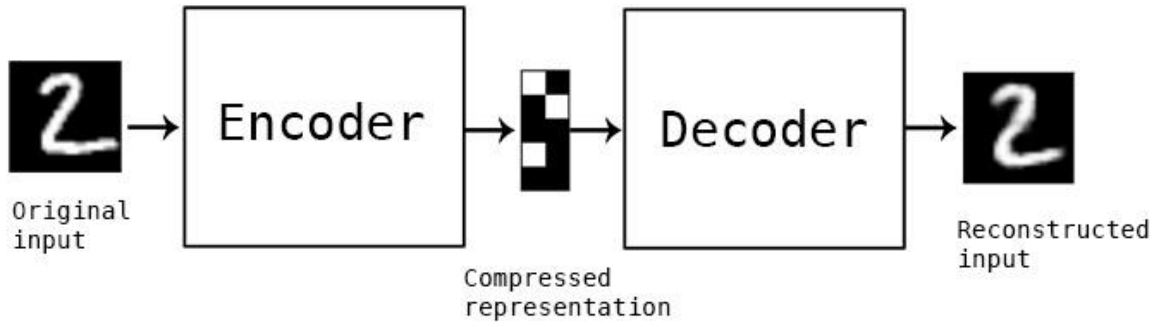
9. Implementing Autoencoder for encoding the real-world data.

Introduction

Building Auto encoders in Keras

- a simple auto encoder based on a fully-connected layer
- a sparse auto encoder
- a deep fully-connected auto encoder
- a deep convolutional auto encoder
- an image denoising model
- a sequence-to-sequence auto encoder
- a variational auto encoder

What are autoencoders?



"Auto encoding" is a data compression algorithm where the compression and decompression functions are 1) data-specific, 2) lossy, and 3) *learned automatically from examples* rather than engineered by a human. Additionally, in almost all contexts where the term "auto encoder" is used, the compression and decompression functions are implemented with neural networks.

1) Auto encoders are data-specific, which means that they will only be able to compress data similar to what they have been trained on. This is different from, say, the MPEG-2 Audio Layer III (MP3) compression algorithm, which only holds assumptions about "sound" in general, but not about specific types of sounds. An auto encoder trained on pictures of faces would do a rather poor job of compressing pictures of trees, because the features it would learn would be face-specific.

2) Auto encoders are lossy, which means that the decompressed outputs will be degraded compared to the original inputs (similar to MP3 or JPEG compression). This differs from lossless arithmetic compression.

3) Auto encoders are learned automatically from data examples, which is a useful property: it means that it is easy to train specialized instances of the algorithm that will perform well on a specific type of input. It doesn't require any new engineering, just appropriate training data.

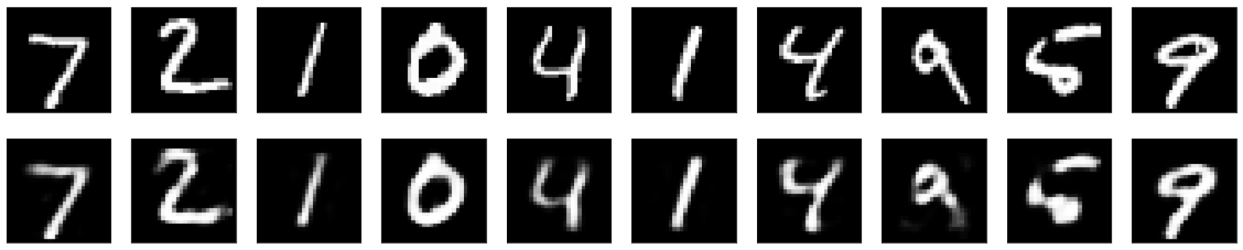
To build an auto encoder, you need three things: an encoding function, a decoding function, and a distance function between the amount of information loss between the compressed representation of your data and the decompressed representation (i.e. a "loss" function). The encoder and decoder will be chosen to be parametric functions (typically neural networks), and to be differentiable with respect to the distance function, so the parameters of the encoding/decoding functions can be optimizing to minimize the reconstruction loss, using Stochastic Gradient Descent. It's simple! And you don't even need to understand any of these words to start using auto encoders in practice.

Code:

```
import keras
from keras import layers
# This is the size of our encoded representations
encoding_dim = 32
 # 32 floats -
> compression of factor 24.5, assuming the input is 784 floats
# This is our input image
input_img = keras.Input(shape=(784,))
# "encoded" is the encoded representation of the input
encoded = layers.Dense(encoding_dim, activation='relu')(input_img)
# "decoded" is the lossy reconstruction of the input
decoded = layers.Dense(784, activation='sigmoid')(encoded)
# This model maps an input to its reconstruction
autoencoder = keras.Model(input_img, decoded)
# Let's also create a separate encoder model:
# This model maps an input to its encoded representation
encoder = keras.Model(input_img, encoded)
# This is our encoded (32-dimensional) input
encoded_input = keras.Input(shape=(encoding_dim,))
# Retrieve the last layer of the autoencoder model
decoder_layer = autoencoder.layers[-1]
# Create the decoder model
decoder = keras.Model(encoded_input, decoder_layer(encoded_input))
autoencoder.compile(optimizer='adam', loss='binary_crossentropy')
from keras.datasets import mnist
import numpy as np
(x_train, _), (x_test, _) = mnist.load_data()
x_train = x_train.astype('float32') / 255.
```

```
x_test = x_test.astype('float32') / 255.
x_train = x_train.reshape((len(x_train), np.prod(x_train.shape[1:])))
x_test = x_test.reshape((len(x_test), np.prod(x_test.shape[1:])))
print(x_train.shape)
print(x_test.shape)
autoencoder.fit(x_train, x_train,
                epochs=50,
                batch_size=256,
                shuffle=True,
                validation_data=(x_test, x_test))
# Encode and decode some digits
# Note that we take them from the *test* set
encoded_imgs = encoder.predict(x_test)
decoded_imgs = decoder.predict(encoded_imgs)

# Use Matplotlib (don't ask)
import matplotlib.pyplot as plt

n = 10  # How many digits we will display
plt.figure(figsize=(20, 4))
for i in range(n):
    # Display original
    ax = plt.subplot(2, n, i + 1)
    plt.imshow(x_test[i].reshape(28, 28))
    plt.gray()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)

    # Display reconstruction
    ax = plt.subplot(2, n, i + 1 + n)
    plt.imshow(decoded_imgs[i].reshape(28, 28))
    plt.gray()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)
plt.show()
```

Output:



Example: As we cannot limit ourselves to a single layer as encoder or decoder. Implement same method by using stack of layers with Convolutional autoencoder

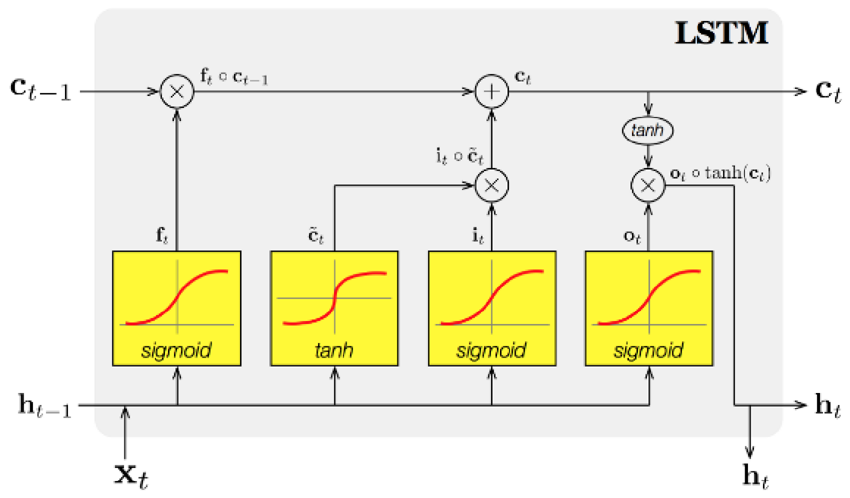
Solution:

10. Study and implementation of LSTM

Introduction

Long Short Term Memory (LSTM) : Long short-term memory (LSTM) units (or blocks) are a building unit for layers of a recurrent neural network (RNN). A RNN composed of LSTM units is often called an LSTM network. A common LSTM unit is composed of a cell, an input gate, an output gate and a forget gate. The cell is responsible for "remembering" values over arbitrary time intervals; hence the word "memory" in LSTM. Each of the three gates can be thought of as a "conventional" artificial neuron, as in a multi-layer (or feed forward) neural network: that is, they compute an activation (using an activation function) of a weighted sum. Intuitively, they can be thought as regulators of the flow of values that goes through the connections of the LSTM; hence the denotation "gate". There are connections between these gates and the cell.

The expression long short-term refers to the fact that LSTM is a model for the short-term memory which can last for a long period of time. An LSTM is well-suited to classify, process and predict time series given time lags of unknown size and duration between important events. LSTMs were developed to deal with the exploding and vanishing gradient problem when training traditional RNNs.



Gating variables

$$\mathbf{f}_t = \sigma\left(\mathbf{W}_f[\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_f\right)$$
$$\mathbf{i}_t = \sigma\left(\mathbf{W}_i[\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_i\right)$$
$$\mathbf{o}_t = \sigma\left(\mathbf{W}_o[\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_o\right)$$

Candidate (memory) cell state

$$\tilde{\mathbf{c}}_t = \tanh\left(\mathbf{W}_c[\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_c\right)$$

Cell & Hidden state

$$\mathbf{c}_t = \mathbf{f}_t \circ \mathbf{c}_{t-1} + \mathbf{i}_t \circ \tilde{\mathbf{c}}_t$$
$$\mathbf{h}_t = \mathbf{o}_t \circ \tanh(\mathbf{c}_t)$$

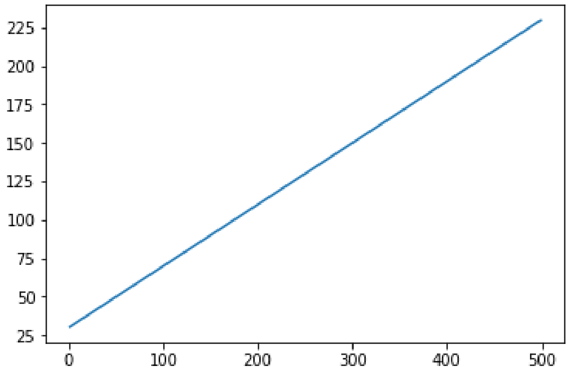
Example

Recurrent neural networks have a wide array of applications. These include time series analysis, document classification, and speech and voice recognition. In contrast to feed forward artificial neural networks, the predictions made by recurrent neural networks are dependent on previous predictions.

Draw a straight line. Let us see, if LSTM can learn the relationship of a straight line and predict it.

First let us create the dataset depicting a straight line.

```
x = numpy.arange (1,500,1)
y = 0.4 * x + 30
plt.plot(x,y)
```



```
trainx, testx = x[0:int(0.8*(len(x)))], x[int(0.8*(len(x))):]
trainy, testy = y[0:int(0.8*(len(y)))], y[int(0.8*(len(y))):]
train = numpy.array(list(zip(trainx,trainy)))
test = numpy.array(list(zip(trainx,trainy)))
```

Now that the data has been created and split into train and test. Let’s convert the time series data into the form of supervised learning data according to the value of look-back period, which is essentially the number of lags which are seen to predict the value at time ‘t’.

So a time series like this −

```
time variable_x
t1  x1
```

```
t2  x2
 :   :
 :   :
T   xT
```

When look-back period is 1, is converted to −

```
x1   x2
x2   x3
 :    :
 :    :
xT-1 xT
```

In [404]:

```
def create_dataset(n_X, look_back):
   dataX, dataY = [], []
   for i in range(len(n_X)-look_back):
      a = n_X[i:(i+look_back), ]
      dataX.append(a)
      dataY.append(n_X[i + look_back, ])
   return numpy.array(dataX), numpy.array(dataY)

look_back = 1
trainx,trainy = create_dataset(train, look_back)
testx,testy = create_dataset(test, look_back)

trainx = numpy.reshape(trainx, (trainx.shape[0], 1, 2))
testx = numpy.reshape(testx, (testx.shape[0], 1, 2))
```

Now we will train our model.

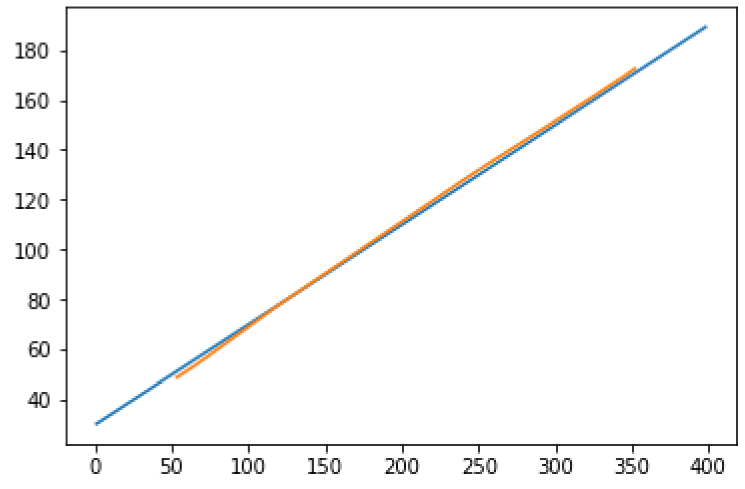
Small batches of training data are shown to network, one run of when entire training data is shown to the model in batches and error is calculated is called an epoch. The epochs are to be run 'til the time the error is reducing.

In [ ]:

```
from keras.models import Sequential
from keras.layers import LSTM, Dense

model = Sequential()
model.add(LSTM(256, return_sequences = True, input_shape = (trainx.shape[1], 2)))
model.add(LSTM(128,input_shape = (trainx.shape[1], 2)))
model.add(Dense(2))
model.compile(loss = 'mean_squared_error', optimizer = 'adam')
model.fit(trainx, trainy, epochs = 2000, batch_size = 10, verbose = 2, shuffle = False)
model.save_weights('LSTMBasic1.h5')
model.load_weights('LSTMBasic1.h5')
predict = model.predict(testx)
```

Now let’s see what our predictions look like.

```
plt.plot(testx.reshape(398,2)[:,0:1], testx.reshape(398,2)[:,1:2])
plt.plot(predict[:,0:1], predict[:,1:2])
```



Exercise

The International Airline Passengers prediction problem. This is a problem where, given a year and a month, the task is to predict the number of international airline passengers in units of 1,000. The data ranges from January 1949 to December 1960, or 12 years, with 144 observations.

Solution:

11. Implementation of GAN for generating Handwritten Digits images.

GANs consist of two neural networks, one trained to generate data and the other trained to distinguish fake data from real data (hence the “adversarial” nature of the model).

Discriminative vs Generative Models

If you’ve studied neural networks, then most of the applications you’ve come across were likely implemented using discriminative models. Generative adversarial networks, on the other hand, are part of a different class of models known as generative models.
Discriminative models are those used for most supervised **classification** or **regression** problems. As an example of a classification problem, suppose you’d like to train a model to classify images of handwritten digits from 0 to 9. For that, you could use a labeled dataset containing images of handwritten digits and their associated labels indicating which digit each image represents.
During the training process, you’d use an algorithm to adjust the model’s parameters. The goal would be to minimize a loss function so that the model learns the **probability.**

The Architecture of Generative Adversarial Networks

Generative adversarial networks consist of an overall structure composed of two neural networks, one called the **generator** and the other called the **discriminator**.
The role of the generator is to estimate the probability distribution of the real samples in order to provide generated samples resembling real data. The discriminator, in turn, is trained to estimate the probability that a given sample came from the real data rather than being provided by the generator.
These structures are called generative adversarial networks because the generator and discriminator are trained to compete with each other: the generator tries to get better at fooling the discriminator, while the discriminator tries to get better at identifying generated samples.

Exercise:
With this sample example, students are asked to perform Handwritten Digits Generator with a GAN.

```
$ conda install -c pytorch torchvision=0.5.0
```

Importing Libraries

```
import torch
from torch import nn

import math
import matplotlib.pyplot as plt
import torchvision
import torchvision.transforms as transforms

torch.manual_seed(111)
```

Device Selection

```
device = ""
if torch.cuda.is_available():
    device = torch.device("cuda")
else:
    device = torch.device("cpu")
```

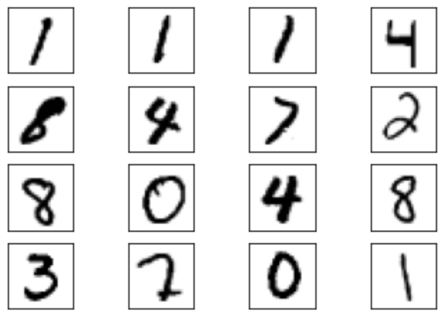
Preparing the Training Data

```
transform = transforms.Compose(
    [transforms.ToTensor(), transforms.Normalize((0.5,), (0.5,))]
)
train_set = torchvision.datasets.MNIST(
    root=".", train=True, download=True, transform=transform
)

batch_size = 32
train_loader = torch.utils.data.DataLoader(
    train_set, batch_size=batch_size, shuffle=True
)
```

To plot some samples of the training data

```
real_samples, mnist_labels = next(iter(train_loader))
for i in range(16):
    ax = plt.subplot(4, 4, i + 1)
    plt.imshow(real_samples[i].reshape(28, 28), cmap="gray_r")
    plt.xticks([])
    plt.yticks([])
```

output:



Implementing the Discriminator and the Generator

```
class Discriminator(nn.Module):
    def __init__(self):
        super().__init__()
        self.model = nn.Sequential(
            nn.Linear(784, 1024),
            nn.ReLU(),
            nn.Dropout(0.3),
            nn.Linear(1024, 512),
            nn.ReLU(),
            nn.Dropout(0.3),
            nn.Linear(512, 256),
            nn.ReLU(),
            nn.Dropout(0.3),
            nn.Linear(256, 1),
            nn.Sigmoid(),
        )

    def forward(self, x):
        x = x.view(x.size(0), 784)
        output = self.model(x)
        return output

discriminator = Discriminator().to(device=device)


class Generator(nn.Module):
    def __init__(self):
        super().__init__()
        self.model = nn.Sequential(
            nn.Linear(100, 256),
            nn.ReLU(),
            nn.Linear(256, 512),
            nn.ReLU(),
            nn.Linear(512, 1024),
            nn.ReLU(),
            nn.Linear(1024, 784),
            nn.Tanh(),
        )

    def forward(self, x):
        output = self.model(x)
        output = output.view(x.size(0), 1, 28, 28)
        return output

generator = Generator().to(device=device)
```

Training the Models

```
lr = 0.0001
num_epochs = 50
loss_function = nn.BCELoss()

optimizer_discriminator = torch.optim.Adam(discriminator.parameters(), lr=lr)
optimizer_generator = torch.optim.Adam(generator.parameters(), lr=lr)
```

The training loop

```
for epoch in range(num_epochs):
    for n, (real_samples, mnist_labels) in enumerate(train_loader):
        # Data for training the discriminator
        real_samples = real_samples.to(device=device)
        real_samples_labels = torch.ones((batch_size, 1)).to(
            device=device
        )
        latent_space_samples = torch.randn((batch_size, 100)).to(
            device=device
        )
        generated_samples = generator(latent_space_samples)
        generated_samples_labels = torch.zeros((batch_size, 1)).to(
            device=device
        )
        all_samples = torch.cat((real_samples, generated_samples))
        all_samples_labels = torch.cat(
            (real_samples_labels, generated_samples_labels)
        )

        # Training the discriminator
        discriminator.zero_grad()
        output_discriminator = discriminator(all_samples)
        loss_discriminator = loss_function(
            output_discriminator, all_samples_labels
        )
        loss_discriminator.backward()
        optimizer_discriminator.step()

        # Data for training the generator
        latent_space_samples = torch.randn((batch_size, 100)).to(
            device=device
        )

        # Training the generator
        generator.zero_grad()
        generated_samples = generator(latent_space_samples)
        output_discriminator_generated = discriminator(generated_samples)
        loss_generator = loss_function(
            output_discriminator_generated,
real_samples_labels
        )
        loss_generator.backward()
        optimizer_generator.step()

        # Show loss
        if n == batch_size - 1:
            print(f"Epoch: {epoch} Loss D.:
{loss_discriminator}")
            print(f"Epoch: {epoch} Loss G.:
{loss_generator}")
```

Exercise

Check the Samples Generated by the GAN. Also write your experience with what you did and output you got.

12. Implementing word2vec for the real-world data.

The basic process for training a word2vec model is as follows:

1. Collect a large dataset of text data. This data is typically in the form of a list of sentences or a collection of documents.
2. Tokenize the text data into individual words. This is often done using a tokenizer, such as the NLTK library in Python.
3. Build a vocabulary of all the words in the dataset. This vocabulary will be used to create the word vectors.
4. Train the model. The training process involves iterating over the dataset, and for each word, predicting the words that are likely to appear in its context. This process is done using a neural network with a single hidden layer.
5. Save the model. After the model is trained, it can be saved for later use.
6. Using the model: Once the word2vec model is trained, it can be used to perform several operations such as finding the similarity between words, finding the odd one out, finding the analogy and more.

The NLTK (Natural Language Toolkit) library in Python provides a convenient way to train a word2vec model using its built-in tokenizer and utilities. Here is an example of how to train a word2vec model using NLTK:

```
import nltk
from nltk.tokenize import sent_tokenize, word_tokenize
from gensim.models import Word2Vec

# Collect the dataset (text data)
text = "This is some sample text data. It can be any kind of text data, such as a book, a news article, or a webpage."

# Tokenize the text into sentences
sentences = sent_tokenize(text)

# Tokenize the sentences into words
sentences = [word_tokenize(sent) for sent in sentences]

# Train the word2vec model
model = Word2Vec(sentences, min_count=1)

# Access vector for one word
vector = model['text']

# Perform cosine similarity between two words
similarity = model.similarity('text', 'data')

# Perform a king of operation
word_vec = model.most_similar(positive=['text'], negative=['data'], topn=1)

# Save the model
```

model.save("word2vec.model")
In this example, the text data is tokenized into sentences and words using the **sent_tokenize** and **word_tokenize** functions from NLTK. Then, the **Word2Vec** class from the gensim library is used to train the model on the tokenized data. The **min_count** parameter is used to specify the minimum number of occurrences of a word in the dataset for it to be included in the vocabulary.
Additionally, as you see in the last 3 lines, we are using the model to access vector for one word, perform cosine similarity between two words and find the most similar word based on the provided positive and negative words
It is important to note that the word2vec algorithm works better with large datasets, so for better performance, it is recommended to use a larger dataset of text data when training the model.

Exercise

Ex 1: use pre-trained word2vec models from Google or Wikipedia

Ex 2: Write Python code that uses the gensim library to train a word2vec model on a pre-defined dataset: