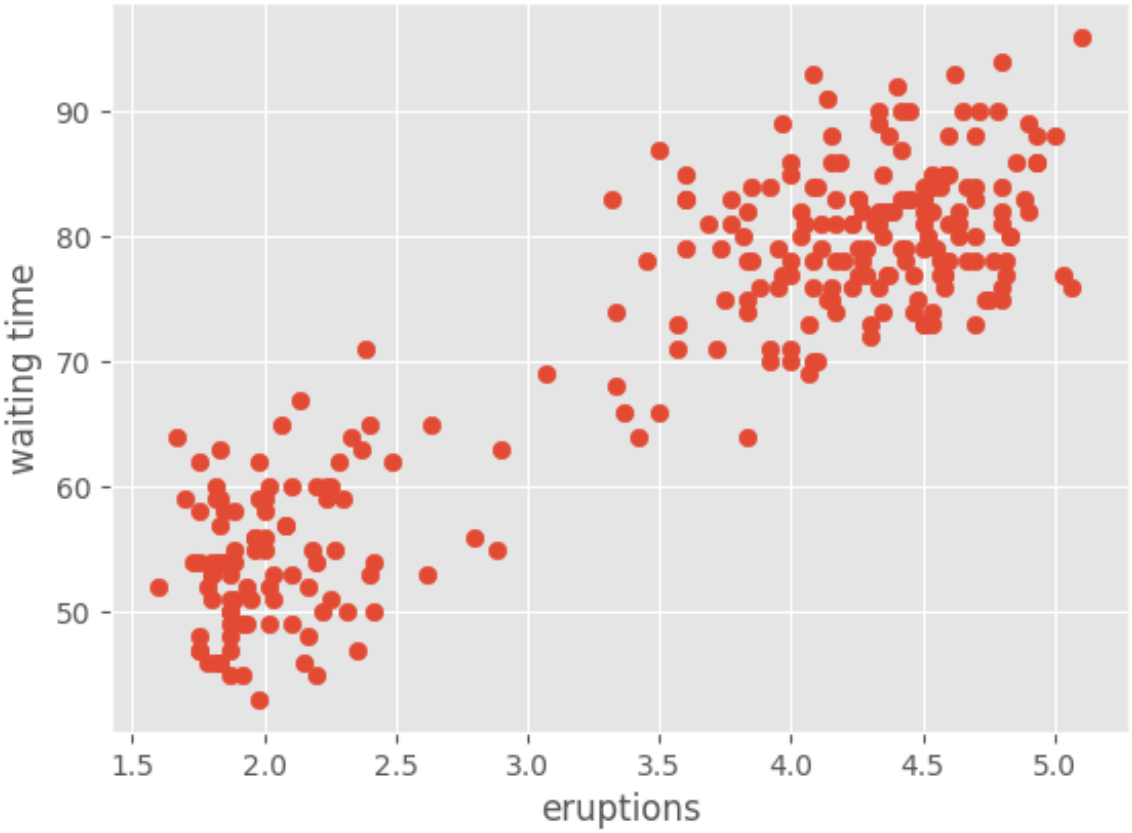
# ID5030 :: Quiz-2

Varun V (AE14B050)

## Solution 1: Clustering Old Faithful Geyser Data:

### (a) Clustering Models:

There are a number of algorithms at our disposal when we want to cluster data. Each of them works well with a specific kind of clustering job while are marginally worse at other kinds. Let us first plot our current data to determine the kind of clustering algorithm we may use for best results.




From a direct look at the data we see the number of cluster are most probably 2 and they have a well-defined boundary albeit a few outlying cases.

We shall use the Scikit-learn library functions for quick and efficient implementation of the clustering algorithms.

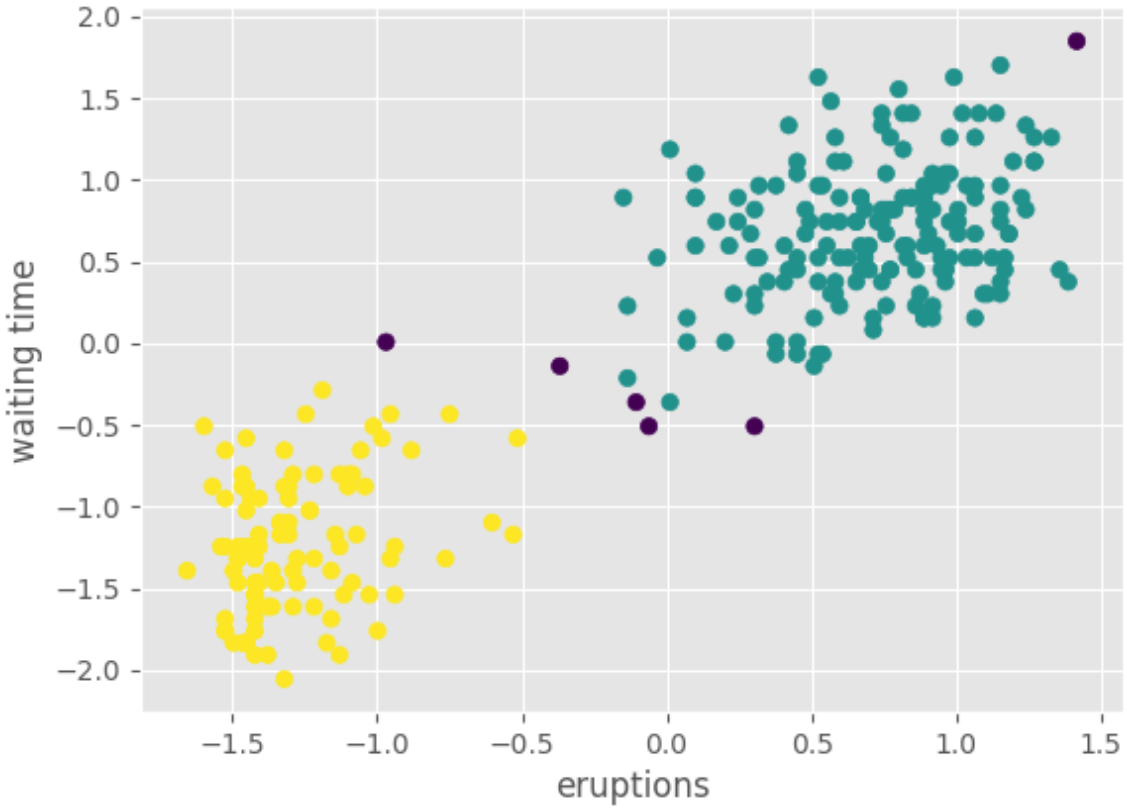
The most basic clustering algorithms we use are as follows:

- Density Based Clustering: This method assumes the clusters are high density areas while the spare points are generally noise or outliers which it ignores. DBSCAN is a popular density based clustering algorithm.
- Distribution Based Clustering: Popular probability distribution functions are fit to the data and boundaries are defined using the parameters of the fit functions. This work really well when there exists and underlying distribution, sometimes almost perfectly clustering data. EM algorithm of Gaussian mixtures is generally used.
- Centroid Based Clustering: Here we try to find central vectors which cluster data. These algorithms require the number of clusters we are trying to find, a drawback, but are fast compared to many other algorithms. They also tend to find equal sized clusters which means they perform very poorly on density or distribution based clustering. K-means algorithm and its variants are used.

**(b)Simple Clustering Program:**

**Ref:** [http://scikit-learn.org/stable/modules/generated/sklearn.cluster.DBSCAN.html#sklearn.cluster.DBSCAN](http://scikit-learn.org/stable/modules/generated/sklearn.cluster.DBSCAN.html#sklearn.cluster.DBSCAN).

From our observations we use the density based spatial clustering algorithm modified for noise. We obtain a clear clustered output with some of the data points as outliers (in violet).




**Code:**

```
# -*- coding: utf-8 -*-
"""
Created on Sat Apr 14 14:07:30 2018

@author: Varun

Q1 :  clustering
"""
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
#import tensorflow as tf
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler,MinMaxScaler
from sklearn.cluster import DBSCAN,AgglomerativeClustering,KMeans
#from sklearn.mixture import BayesianGaussianMixture,GaussianMixture
from matplotlib import style
style.use('ggplot')

"""
Importing the data and standard scaling it
"""
data = pd.read_csv('oldfaithful.csv')
scaler = StandardScaler()
data.iloc[:,:] = scaler.fit_transform(data.iloc[:,:])

"""
plot for intial observations
"""
#plt.xlabel(data.columns[0])
```

```
#plt.ylabel(data.columns[1]+' time')
#plt.scatter(data.iloc[:,0],data.iloc[:,1])

"""
DBSCAN clustering algorithm implementation
"""
clusterer = DBSCAN(eps = 0.4,min_samples = 10)#use eps = 0.52 for no outliers
db = clusterer.fit(data)
labels = db.labels_ #labels for the data points we fit the algorithm to

"""
plot
"""
plt.xlabel(data.columns[0])
plt.ylabel(data.columns[1]+' time')
scatterplot = plt.scatter(data.iloc[:,0],data.iloc[:,1], c = labels)
```

### (c) Naïve Bayes Classifier:

**Code:**

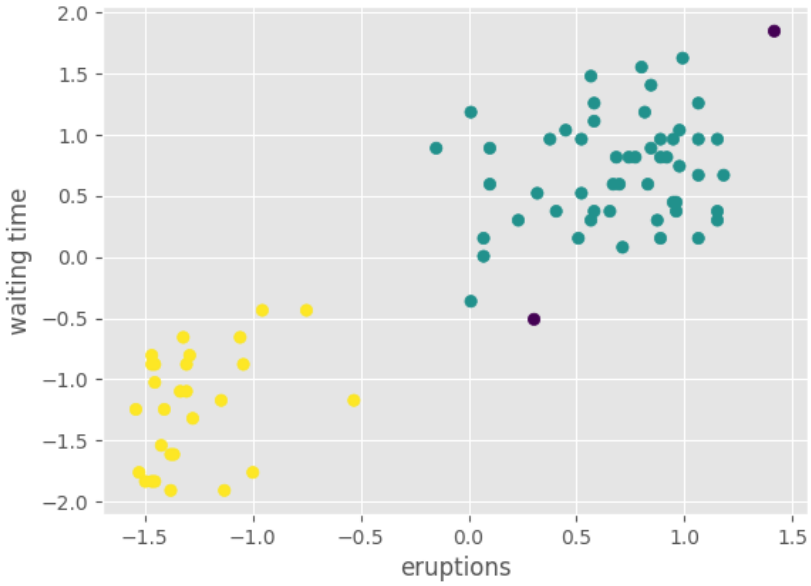
```
# -*- coding: utf-8 -*-
"""
Created on Sat Apr 14 14:07:30 2018

@author: Varun

Q1 :  clustering
"""
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
#import tensorflow as tf
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler,MinMaxScaler
from sklearn.cluster import DBSCAN,AgglomerativeClustering,KMeans
#from sklearn.mixture import BayesianGaussianMixture,GaussianMixture
from sklearn.naive_bayes import GaussianNB
from matplotlib import style
style.use('ggplot')

"""
Importing the data and standard scaling it
"""
data = pd.read_csv('oldfaithful.csv')
scaler = StandardScaler()
data.iloc[:,:] = scaler.fit_transform(data.iloc[:,:])

"""
plot for intial observations
"""
#plt.xlabel(data.columns[0])
#plt.ylabel(data.columns[1]+' time')
#plt.scatter(data.iloc[:,0],data.iloc[:,1])

"""
DBSCAN clustering algorithm implementation
"""
clusterer = DBSCAN(eps = 0.4,min_samples = 10)#use eps = 0.52 for no outliers
db = clusterer.fit(data)
labels = db.labels_ #labels for the data points we fit the algorithm to

"""
plot
"""
#plt.xlabel(data.columns[0])
#plt.ylabel(data.columns[1]+' time')
#scatterplot = plt.scatter(data.iloc[:,0],data.iloc[:,1], c = labels)
"""
```

```
Gaussian Naive bayes classifier - as data is continuous
"""
Xtrain,Xtest,ytrain,ytest = train_test_split(data,labels,test_size = 0.3)
classifier = GaussianNB()
classifier.fit(Xtrain,ytrain)
ypred = classifier.predict(Xtest)
accuracy = classifier.score(Xtest,ytest)

plt.xlabel(Xtest.columns[0])
plt.ylabel(Xtest.columns[1]+' time')
scatterplot = plt.scatter(Xtest.iloc[:,0],Xtest.iloc[:,1], c = ypred)#prediction data
#scatterplot = plt.scatter(Xtest.iloc[:,0],Xtest.iloc[:,1], c = ytest) #true data
```

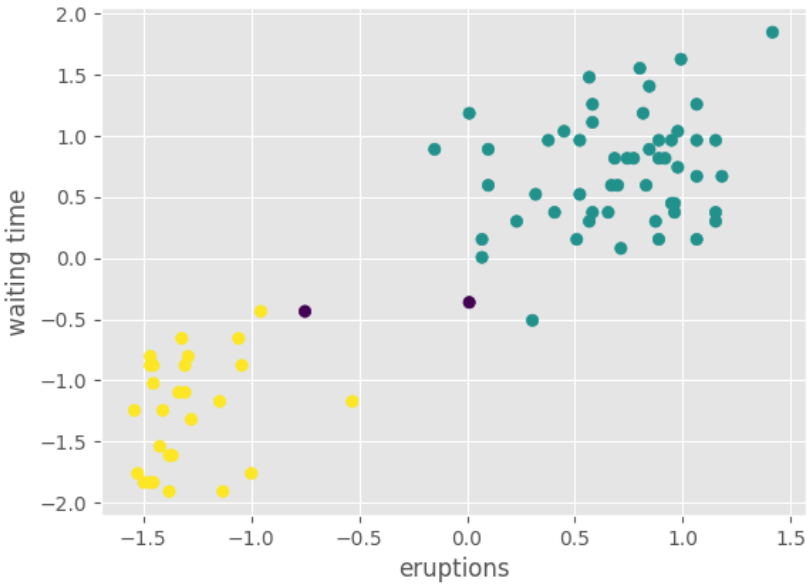
We use the Gaussian Naïve Bayes classifier as our data is continuous. Fractional and even negative. The Gaussian estimation handles these use cases for us.

Below are the plots for the 30% test data, both true and predicted. We do not neglect the outliers to fit our classifier. As observed the predictions are highly accurate with a prediction accuracy of 95.122%.

True labels:




NB predicted labels:



**(d) Maximum Likelihood Gaussians:**

Following a detailed derivation on Gaussian estimator parameters.

→ The probability distribution (density function) of a normal gaussian variable is given by

$$f(x_j) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(\frac{-1}{2}\frac{(x_j - \mu)^2}{\sigma^2}\right) \quad \ldots\ldots (1)$$

here, $\sigma$ is the true variance and $\mu$ is the mean.

→ Given that the variable is independently sampled and is identically distributed, we can define a likelihood function $L$,

$$L(\vec{x}) = \prod_{j=1}^{n} f(x_j), \text{ where } n \text{ is the total number of samples.}$$

$$L(\vec{x}) = \left[\frac{1}{2\pi\sigma^2}\right]^{n/2} \exp\left(\frac{-1}{2\sigma^2}\sum_{j=1}^{n}(x_j - \mu)^2\right) \quad \ldots (2)$$

→ The loglikelihood of $L$, defined as '$l$' can then be written as,

$$l = \log(L) = \frac{n}{2}\log\left(\frac{1}{2\pi\sigma^2}\right) + \left[\frac{-1}{2\sigma^2}\sum_{j=1}^{n}(x_j - \mu)^2\right]$$

$$= -\left[\frac{n}{2}\log(2\pi) + n\log\sigma + \frac{1}{2\sigma^2}\sum_{j=1}^{n}(x_j - \mu)^2\right]$$

→ The maximum log likelihood estimators are simply the values of $\mu, \sigma$ that are estimated from the data with the constraint that they maximize $l(\vec{x})$.

→ $\max_{\mu,\sigma^2} \{ L(\vec{x}) \}$ occurs when $\frac{\partial l}{\partial \mu} = 0$, $\frac{\partial l}{\partial \sigma^2} = 0$

(i) $$\frac{\partial l}{\partial \mu} = \frac{\partial}{\partial \mu}\left[\frac{-n}{2}\ln 2\pi - \frac{n}{2}\ln(\sigma^2) - \frac{1}{2\sigma^2}\sum_{j=1}^{n}(x_j-\mu)^2\right] = 0$$

$$\Rightarrow \frac{-1}{2\sigma^2}\frac{\partial}{\partial \mu}\left[\sum_{j=1}^{n}(x_j-\mu)^2\right] = \frac{-1}{2\sigma^2}(-2)\sum_{j=1}^{n}(x_j-\mu) = 0$$

$$\Rightarrow \sum_{j=1}^{n}(x_j-\mu) = 0 \Rightarrow \boxed{\frac{1}{n}\sum_{j=1}^{n}x_j = \hat{\mu}}$$ the $\hat{}$ here represents estimated value from data.

(ii) $$\frac{\partial l}{\partial \sigma^2} = \frac{\partial}{\partial \sigma^2}\left[\frac{-n}{2}\ln(2\pi) - \frac{n}{2}\ln(\sigma^2) - \frac{1}{2\sigma^2}\sum_{j=1}^{n}(x_j-\mu)^2\right] = 0$$

$$\Rightarrow \frac{\partial}{\partial \sigma^2}\left[\frac{-n}{2}\ln\sigma^2\right] - \frac{1}{2}\frac{\partial}{\partial \sigma^2}\left(\frac{1}{\sigma^2}\right)\left[\sum_{j=1}^{n}(x_j-\mu)^2\right] = 0$$

$$\Rightarrow \frac{-n}{2\sigma^2} + \frac{1}{2}\left(\frac{1}{\sigma^2}\right)^2\left[\sum_{j=1}^{n}(x_j-\mu)^2\right] = 0$$

$$\Rightarrow \boxed{\hat{\sigma}^2 = \frac{1}{n}\left[\sum_{j=1}^{n}(x_j-\mu)^2\right]}$$

→ Hence we obtain the maximum likelihood gaussian estimators for each variable.

→ We extend these results for two variable ( bivariate) cases

$$\rightarrow \quad z = \frac{(x_1-\mu_1)^2}{\sigma_1^2} - \frac{2\rho(x_1-\mu_1)(x_2-\mu_2)}{\sigma_1\sigma_2} + \frac{(x_2-\mu_2)^2}{\sigma_2^2}$$

$$\rho = \frac{\text{Covariance}(x_1,x_2)}{\sigma_1\sigma_2} = \text{correlation}(x_1,x_2)$$

$$p(x_1,x_2) = \frac{1}{2\pi\sigma_1\sigma_2\sqrt{1-\rho^2}} \exp\left[-\frac{z}{2(1-\rho^2)}\right]$$

where $\hat{\mu}_1 = \frac{1}{n}\sum_{j=1}^{n} x_{1j}$ , $\hat{\mu}_2 = \frac{1}{n}\sum_{j=1}^{n} x_{2j}$

$$\hat{\sigma}_1^2 = \frac{1}{n}\sum_{j=1}^{n}[x_{1j}-\hat{\mu}_1]^2 \;,\; \hat{\sigma}_2^2 = \frac{1}{n}\sum_{j=1}^{n}[x_{2j}-\hat{\mu}_2]^2$$

$$\text{Covariance}(x_1,x_2) = \hat{\sigma}_{12} = \hat{\sigma}_{21} = \frac{1}{n}\sum_{j=1}^{n}[x_{1j}-\hat{\mu}_1][x_{2j}-\hat{\mu}_2]$$

We use these above Bivariate relations in our code below to estimate our Gaussian distribution over the given data.

**Code:**

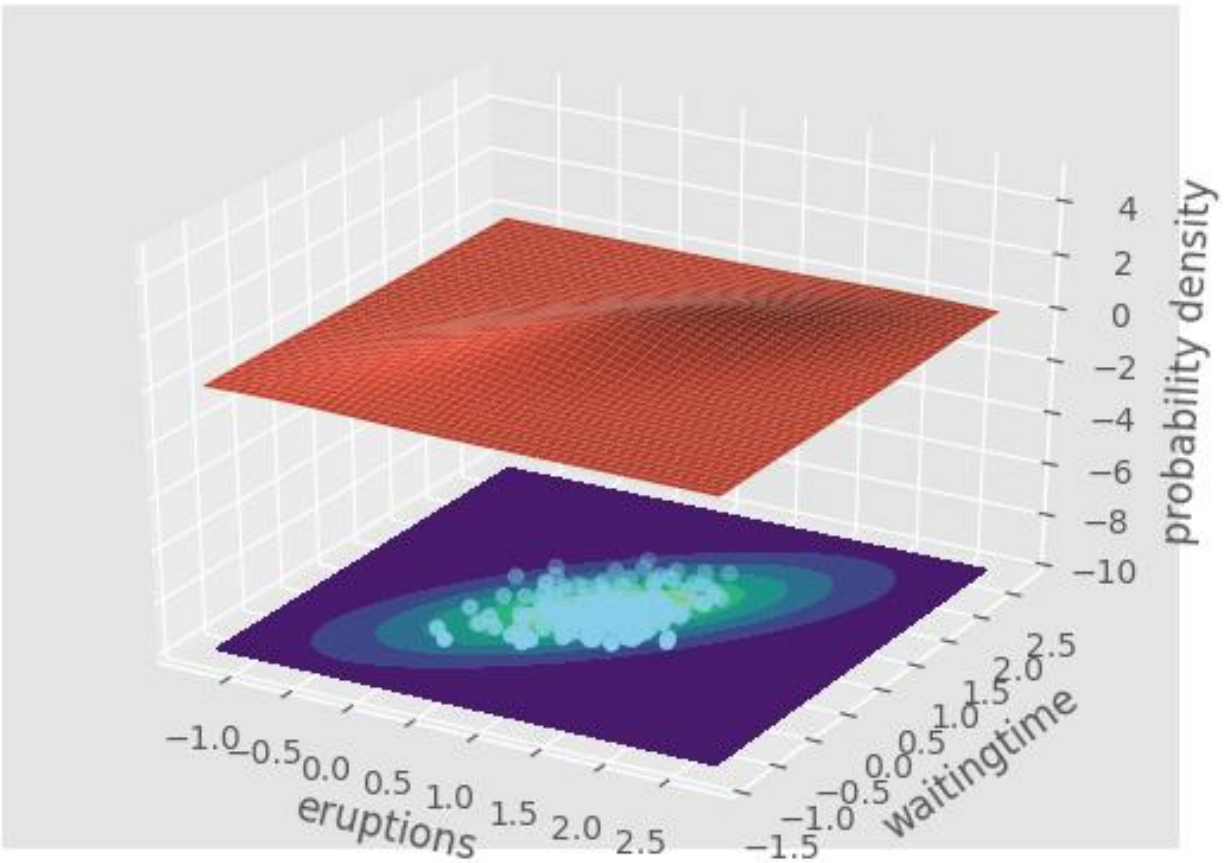
```
# -*- coding: utf-8 -*-
"""
Created on Sat Apr 14 14:07:30 2018

@author: Varun

Q1 :  clustering
"""
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
#import tensorflow as tf
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler,MinMaxScaler
from sklearn.cluster import DBSCAN,AgglomerativeClustering,KMeans
#from sklearn.mixture import BayesianGaussianMixture,GaussianMixture
from sklearn.gaussian_process import GaussianProcessRegressor
from sklearn.naive_bayes import GaussianNB
from matplotlib import style
from matplotlib.patches import Ellipse
from matplotlib import cm
from mpl_toolkits.mplot3d import Axes3D
```

```
style.use('ggplot')

"""
Importing the data and standard scaling it
"""
data = pd.read_csv('oldfaithful.csv')
scaler = StandardScaler()
data.iloc[:,:] = scaler.fit_transform(data.iloc[:,:])

"""
plot for intial observations
"""
#plt.xlabel(data.columns[0])
#plt.ylabel(data.columns[1]+' time')
#plt.scatter(data.iloc[:,0],data.iloc[:,1])

"""
DBSCAN clustering algorithm implementation
"""
clusterer = DBSCAN(eps = 0.4,min_samples = 10)#use eps = 0.52 for no outliers
db = clusterer.fit(data)
labels = db.labels_ #labels for the data points we fit the algorithm to

"""
plot
"""
#plt.xlabel(data.columns[0])
#plt.ylabel(data.columns[1]+' time')
#scatterplot = plt.scatter(data.iloc[:,0],data.iloc[:,1], c = labels)
"""
Gaussian Naive bayes classifier - as data is continuous
"""
Xtrain,Xtest,ytrain,ytest = train_test_split(data,labels,test_size = 0.3)
classifier = GaussianNB()
classifier.fit(Xtrain,ytrain)
ypred = classifier.predict(Xtest)
accuracy = classifier.score(Xtest,ytest)

"""
plots for comparing true labels to predicted labels
"""
#plt.xlabel(Xtest.columns[0])
#plt.ylabel(Xtest.columns[1]+' time')
#scatterplot = plt.scatter(Xtest.iloc[:,0],Xtest.iloc[:,1], c = ypred)#prediction data
#scatterplot = plt.scatter(Xtest.iloc[:,0],Xtest.iloc[:,1], c = ytest) #true data
"""
plt.xlabel(data.columns[0])
plt.ylabel(data.columns[1]+' time')
scatterplot = plt.scatter(data.iloc[:,0],data.iloc[:,1], c = labels)
plt.show()
"""

"""
Bivariate gaussian estimator
"""
class0 = data[labels==0]
class1 = data[labels==1]
mean0,mean1 = np.mean(class0),np.mean(class1)
std0,std1 = np.std(class0),np.std(class1)
rho0 = np.corrcoef(class0,rowvar = False)[0,1]
rho1 = np.corrcoef(class1,rowvar = False)[0,1]
X0,X1,Y0,Y1 = np.linspace(mean0[0]-2,mean0[0]+2,50),np.linspace(mean1[0]-
2,mean1[0]+2,50),np.linspace(mean0[1]-2,mean0[1]+2,50),np.linspace(mean1[1]-2,mean1[1]+2,50)
X0,Y0 = np.meshgrid(X0,Y0)
X1,Y1 = np.meshgrid(X1,Y1)
Z1 = (((X1 - mean1[0])**2)/std1[0]) + (((Y1 - mean1[1])**2)/std1[1]) - ((2*rho1*(X1-
mean1[0])*(Y1-mean1[1]))/(std1[0]*std1[1]))
P1 = (1/(2*np.pi*std1[0]*std1[1]*(np.sqrt(1-rho1**2))))*np.exp(-Z1/(2*(1-rho1**2)))

Z0 = (((X0 - mean0[0])**2)/std0[0]) + (((Y0 - mean0[1])**2)/std0[1]) - ((2*rho0*(X0-
mean0[0])*(Y0-mean0[1]))/(std0[0]*std0[1]))
```

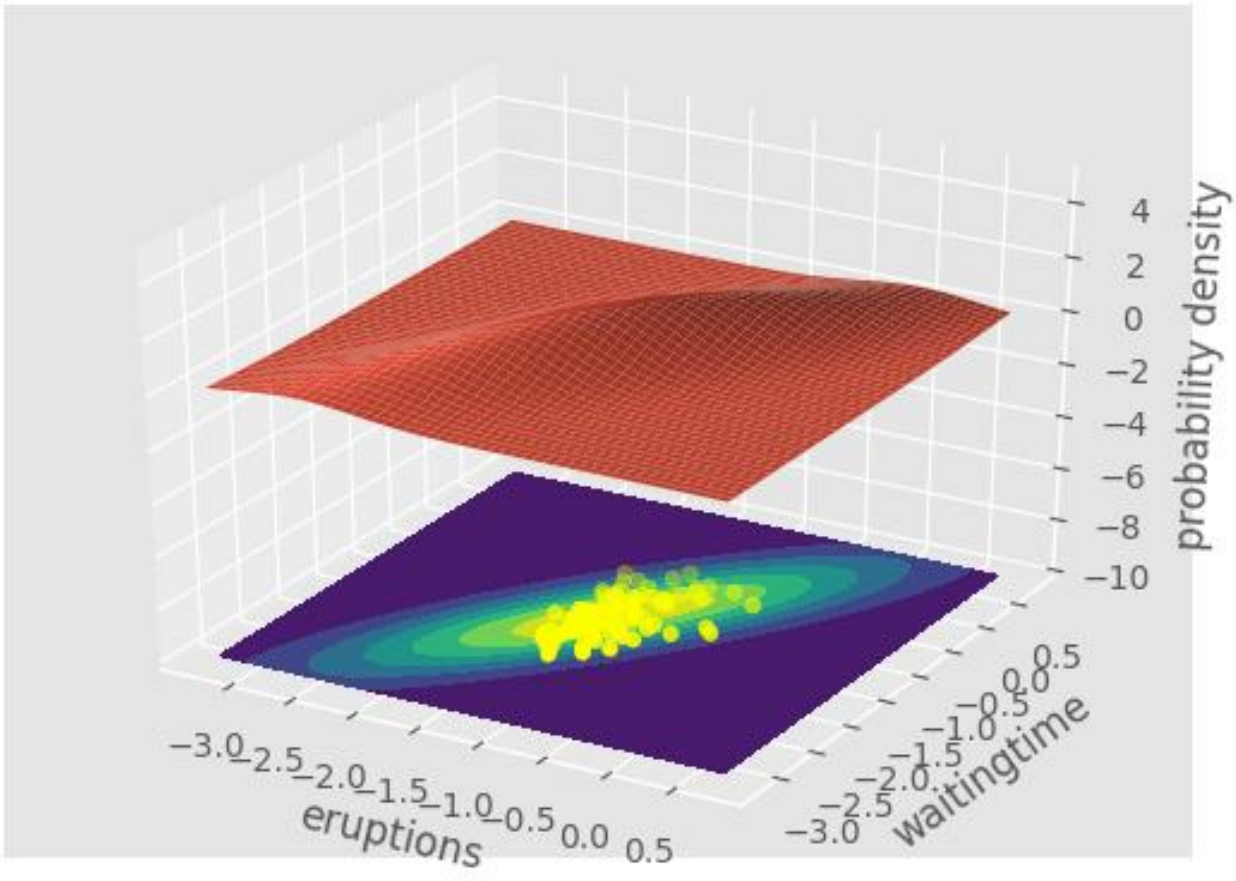
```
P0 = (1/(2*np.pi*std0[0]*std0[1]*(np.sqrt(1-rho0**2))))*np.exp(-Z0/(2*(1-rho0**2)))
"""
plot for class0
"""
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
surf = ax.plot_surface(X0,Y0,P0)
contour = ax.contourf(X0,Y0,P0,offset=-10)
scatter = ax.scatter(class0.iloc[:,0],class0.iloc[:,1],-10,c = 'blue')
        ax.set_zlim(-10, 5)
        ax.set_zlabel('probability density')
        ax.set_xlabel(Xtest.columns[0])
        ax.set_ylabel(Xtest.columns[1]+'time')
        plt.show()

        """
        plot for class1
        """

        fig = plt.figure()
        ax = fig.add_subplot(111, projection='3d')
        surf = ax.plot_surface(X1,Y1,P1)
        contour = ax.contourf(X1,Y1,P1,offset=-10)
        scatter = ax.scatter(class1.iloc[:,0],class1.iloc[:,1],-10,c = 'yellow')
        ax.set_zlim(-10, 5)
        ax.set_zlabel('probability density')
        ax.set_xlabel(Xtest.columns[0])
        ax.set_ylabel(Xtest.columns[1]+'time')
        plt.show()
```

Using the theory mentioned above we plot the following Gaussian distribution using our estimators, added are contour maps with the distribution of points for ease of observation.

(i) Blue labels - long eruption times cluster – class0:




(ii) Yellow labels - short eruption times cluster – class1:




**(e)MAP and Bayesian Estimates:**

## Solution 2: Applications of Inception module:

### (a) Advantages:

For a general CNN we need to decide at each layer what kind of filters we need to use, which primarily involves selecting the side of the filter. Some filters may perform better than other and it might be difficult to cross check each filters' effectiveness as the upcoming and previous filters used may influence the current filter when back propagating. We therefore try to model an architecture which not only implements different sizes of filters but also involves stride-1 pooling layers along with convolution filters and we let back propagation decide which of them is to be used best and most.

### (b) Improvements:

More filters obviously mean a lot more computations and weights to go through. The improvement of the inception module with dimensionality reduction tries to address this problem by reducing the depth of the inputs (or dimensionality) by applying 1x1 filters which are very fast to the inputs and then using larger convolution filters to get the output.

This dimensionality reduction is a very important factor, let us look at the number of operations for 128, 5x5x256 filters over an input layer of 28x28x256 (basic MNIST). With padding we are looking at,

5x5x256x28x28x128 = 642,252,800 Operations

Let us compute the number of operations for dimensionality reduced model,

Reducing the dimensionality by 4 with 1x1 filters, which reduces our input to 28x28x64,

1x1x256x28x28x64 = 12,845,056 Operations

We now apply the 5x5x64 filters to the reduced input

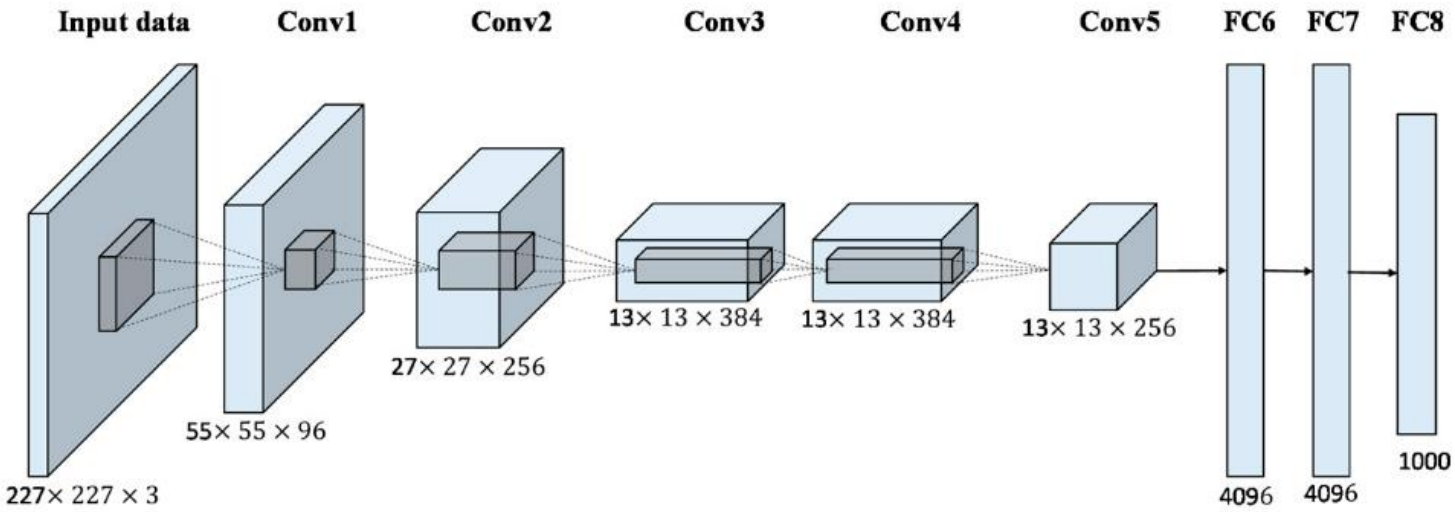
5x5x64x28x28x128 = 160,563,200 Operations

Total operations for reduced inception module = 173,408,256 Operations

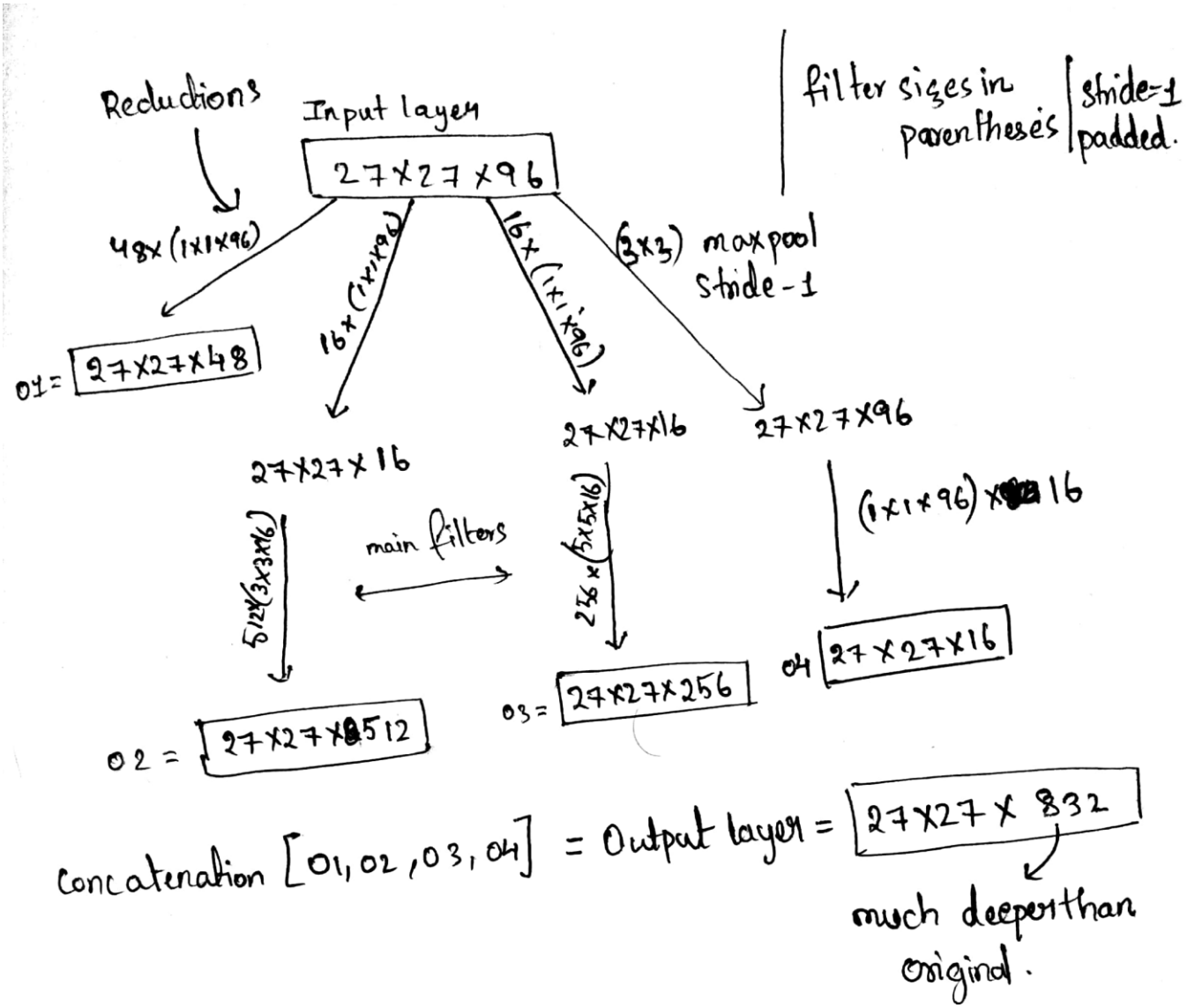
We see about ¼ reduction in the number of operations for the reduced model. This might seems a paltry improvement but that is only because of the very small reduction we are using. Higher orders of reduction give much better improvement in the computational performance with almost better than an order improvement over the naïve model when the reduction is increased to 1/16.

### (c)Modification over AlexNet:

Let us look at the alexnet architecture in the image given below:




Let us try to create and dimensionality reduced inception module to the normalized, max pooled output of Convolution layer 1 (27x27x96) and apply convolution layer 2 and obtain a (27x27x256). This step uses 5x5 filters in the AlexNet, which we shall modify to get a much deeper image. We employ filters in the same ratios as applied in the GoogleNet architecture.




**(d)Application in Encoder-Decoder Network:**

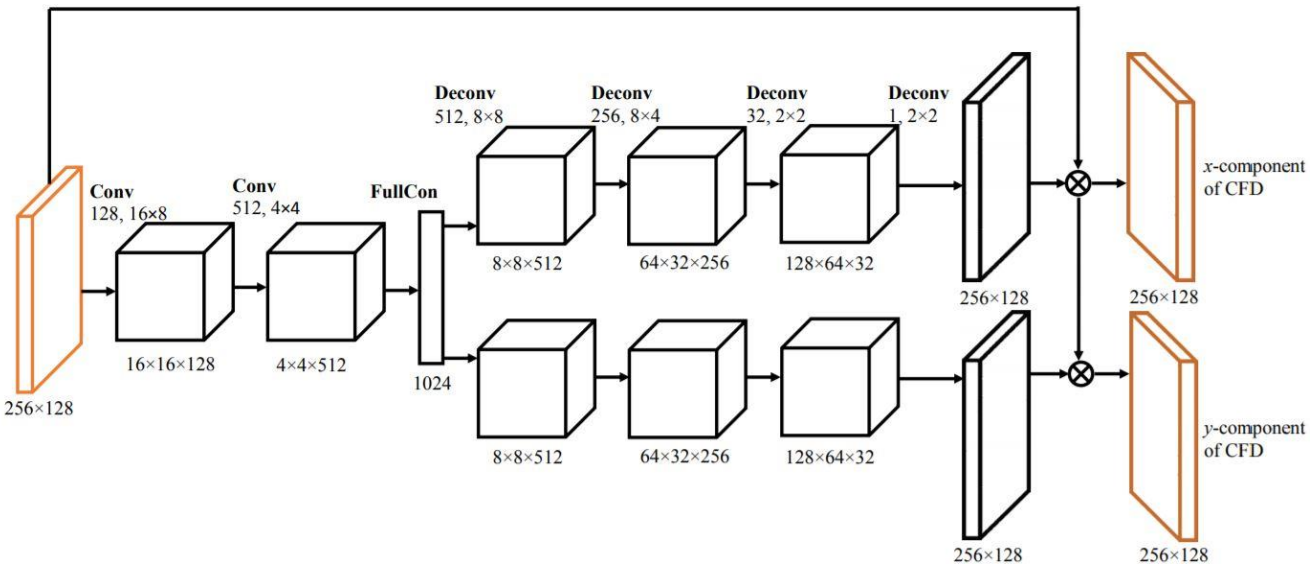
The primary use of encoder-decoder networks is to estimate continuous or seemingly unrelated variables, which is not a use case for a generic CNN.

Its major applications include Machine translations from one language to another as the languages have totally different construction rules and the final encoded vector helps in gauging the core meaning of a particular phrase or word which will then be converted to that another language with the decoder network.

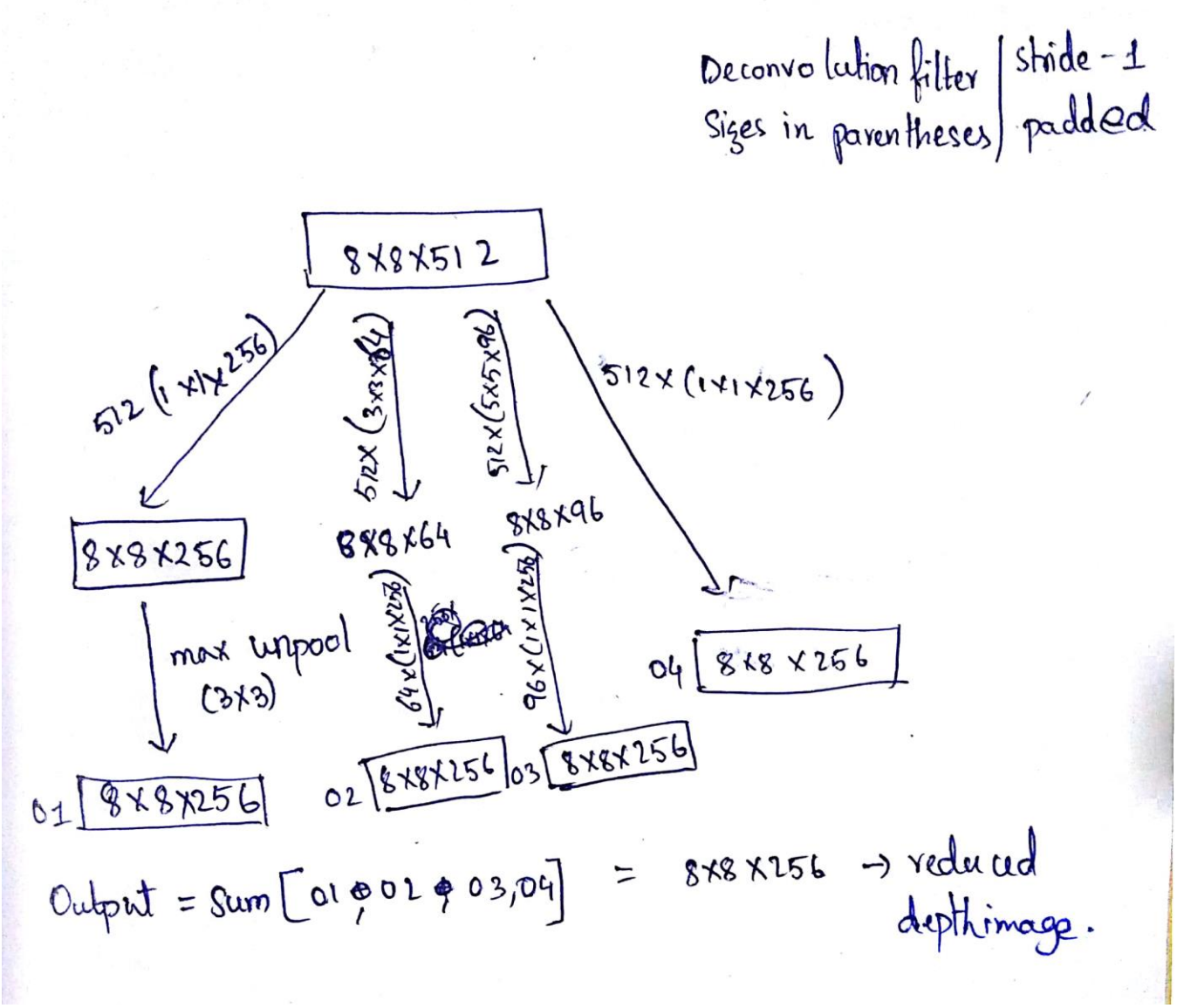
It also sees applications in CFD where it is proven to be orders faster than traditional methods like Lattice Boltzmann method when computing continuous vector fields of the flow.

**Ref:** 1. https://arxiv.org/pdf/1409.3215.pdf

2.https://autodeskresearch.com/publications/convolutional-neural-networks-steady-flow-approximation




Implementation of the Inception module is conceptually the same as in the normal CNNs when dealing with the encoder side, the decoder side however is a total inverse of the CNN implements which is drawn below. The example used is the second Deconvolution layer from the image above,



**Solution 3: RK-4 RNN implementation:**

6. RNN to calculate RK-4 ODE.

ODE : $\frac{dy}{dt} = f(y) = f(y(t)) = f(y,t) \mid y(t=0) = y_0$

→ for each time step of $\Delta t$, the Range-kutta 4 methods are given a

$$y_{n+1} = y_n + \frac{\Delta t}{6}\left[k_1 + 2k_2 + 2k_3 + k_4\right]$$

$$k_1 = f(t_n, y_n)$$

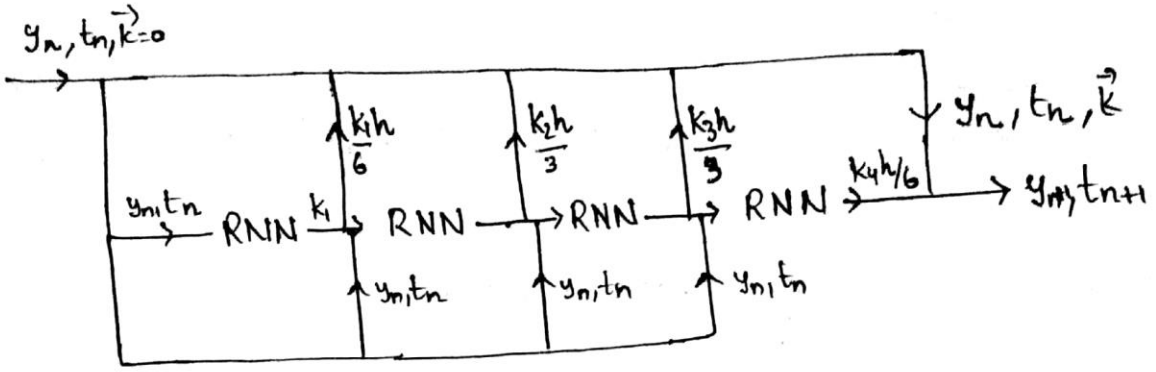
$$k_2 = f\left(t_n + \frac{\Delta t}{2}, y_n + \frac{\Delta t k_1}{2}\right)$$

$$k_3 = f\left(t_n + \frac{\Delta t}{2}, y_n + \frac{\Delta t k_2}{2}\right)$$

$$k_4 = f(t_n + \Delta t, y_n + \Delta t k_3)$$

→ Let us define a hidden state $H_m$ for our network, such that it corresponds to steps of concurrent calculations of k

→ We shall use the following Network to compute each time step of $y(t)$.




→ here at each layer, we use a vanilla RNN to compute the next $k$ in the network.

↳ The inputs at each step is $[y_n \; t_n \; k_1 \; k_2 \; k_3 \; k_4]$ vector where all the $k_i$ are zero, if it's the first step or one of them as a value previously computed.

↳ The output at each step is a $[k_1 \; k_2 \; k_3 \; k_4]$ vector where again only one of them is non-zero after each RNN implementation.

→ The hidden state $H_m$ will be representative of which part of the Calculation we are currenty in.

A simple example of how this works is as follows, let the hidden states cycle, with them being

$$H_1 = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}, \quad H_2 = \begin{bmatrix} 1 \\ h/2 \\ 0 \\ 0 \\ 0 \end{bmatrix}, \quad H_3 = \begin{bmatrix} 1 \\ 0 \\ h/2 \\ 0 \\ 0 \end{bmatrix}, \quad H_4 = \begin{bmatrix} 1 \\ 0 \\ 0 \\ h \\ 0 \end{bmatrix}, \quad H_5 = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

These can be used to compute the 'tn parts' of variable of the function f in each step,

$$\vec{t}_n = \begin{bmatrix} t_n \\ t_n + h/2 \\ t_n + h/2 \\ t_n + h \end{bmatrix}^T = \begin{bmatrix} t_n & 1 & 1 & 1 & 0 \end{bmatrix} \begin{bmatrix} 1 & 1 & 1 & 1 \\ 0 & h/2 & 0 & 0 \\ 0 & 0 & h/2 & 0 \\ 0 & 0 & 0 & h \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

$$\downarrow$$

$$W_{TH}$$

→ We shall create a weight matrix $W_{HH}$ to cycle through the hidden states,

$$W_{HH} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 1/2 & -1 & -1 & -1/2 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

, this is ~~pretty~~ easy to calculate by solving the matrix and relevant variables ~~and~~ ~~set~~ over each step and setting the rest to zero

$$H_{n+1} = W_{HH} \cdot H_n$$

→ After each iteration our RNN hidden S. is cycled back and we ~~get~~ get an out put $y_{n+1}$ according to the algorithm of RK4. This can be repeated for however many time intervals.

→ It is important to note that the hidden states and weight matrices are only representative of what is happening in a real case these tend to be a lot more complicated and bigger weight matrices ~~and~~ and a lot more variables are used in the general sense.

---