15ECE368-Introduction to Soft Computing.

Term Work Report.

CB.EN.U4ECE17303	Aditya S
CB.EN.U4ECE17307	Ashwinkumar JS
CB.EN.U4ECE17330	Madisetty Eshwar
CB.EN.U4ECE17135	Kolluru Harsha

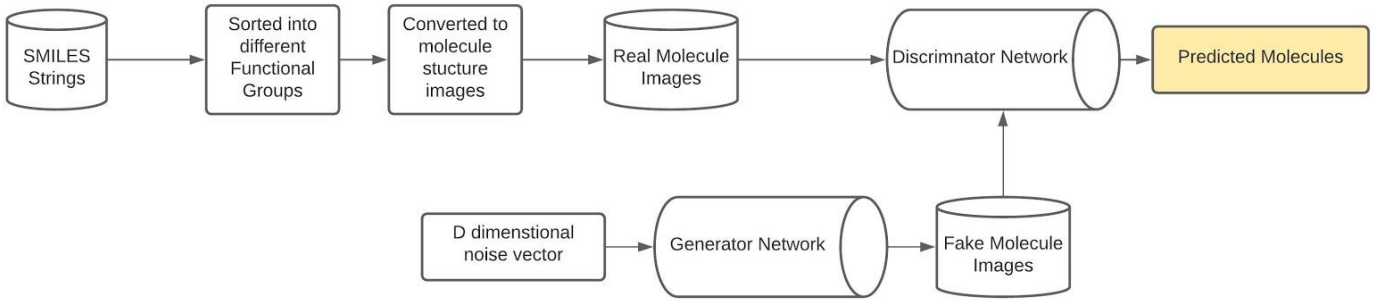
Index

Molecules Generation to Aid Drug Design

Abstract :

In this project we have trained a Generative recurrent neural network to create new molecules which are similar to existing drugs and which may help as a vaccine for new viruses and diseases. We compared RNN with LSTM cells or Generative Adversarial Networks (GAN) to learn patterns from SMILES (simplified molecular-input line-entry system) Strings, and use them to generate new molecular structures.Which aid in reducing time taken for the process of drug design.

Block Diagram:



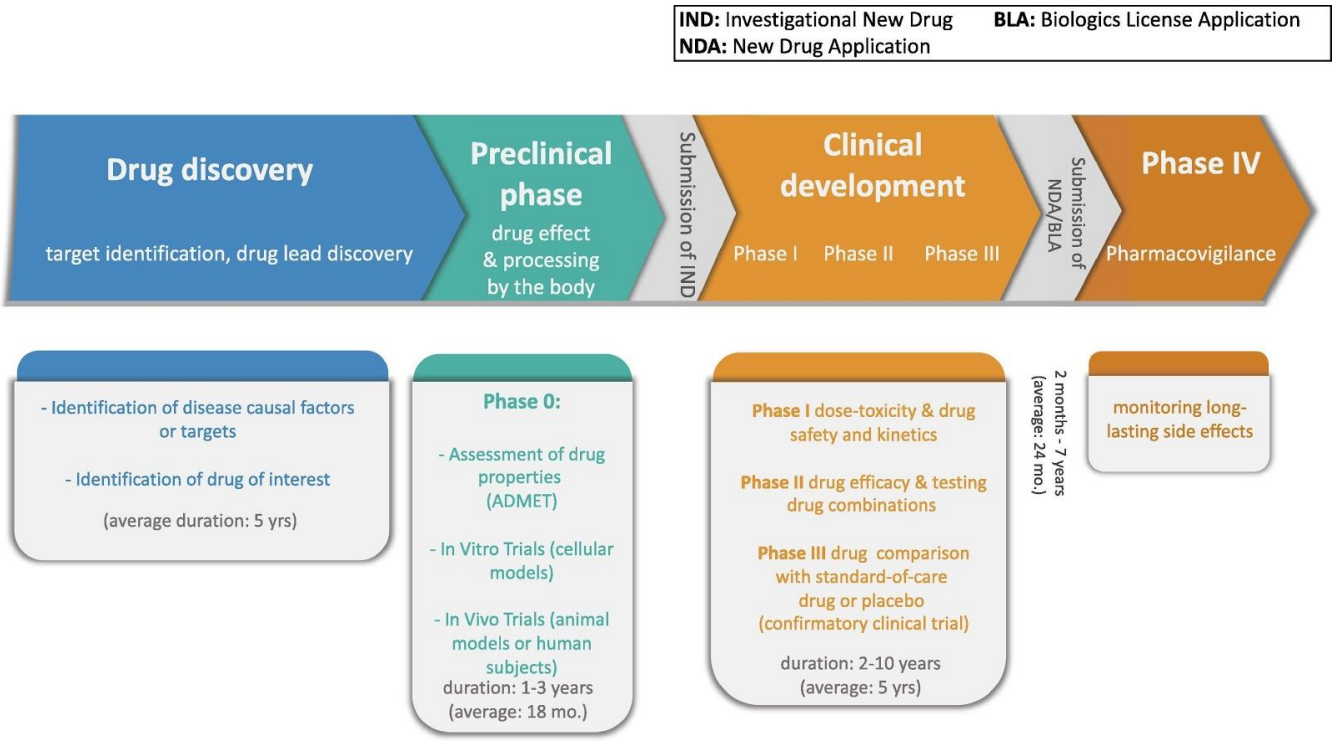
Need for the project:

The current pandemic situation inspired us to choose this project, Development of new drugs is a time-consuming and costly process. Indeed, in order to ensure both the patients' safety aand drug effectiveness, prospective drugs must undergo a competitive and long procedure. Drug development is roughly split into four major stages, called phases. Phase 0 comprises basic research/drug discovery and preclinical tests, which aim at assessing the efficiency and body processing of the drug candidate.

The last three stages are clinical trials: study of dose-toxicity, short-lived side effects, and kinetic relationships (Phase I); determination of drug performance (Phase II); and comparison of the molecule to the standard-of-care (Phase III). An optional Phase IV can be post-drug marketing to monitor long-lasting side effects and drug combination with other therapies. The picture depicts the whole drug development timeline. This pipeline takes at least 5 years to be completed , and can last up to 15 years . The minimal amount of time covers the setup of preclinical and clinical tests (Phases 0 up to III), that is, the time to ponder upon and write down the study design, to recruit and select patients, to analyze the results, and so on, let alone to perform the wet-lab experiments.

Clinical development time (that is, from Phase I) has steadily increased. For drugs approved in 2005–2006, the average clinical development time was 6.4 years, whereas it increased up to 9.1 years for 2008–2012 drug candidates. This might denote an issue in assessing drug effects and benefits. Conversely, the high failure rate of drug development pipelines, often at late stages of clinical testing, has always been a critical issue . Clinical trials occurred between 1998 and 2008 (in Phases II and III), have reported a failure rate of 54%. Main reasons for failure were the lack of efficacy (57% of the failing drug candidates), and safety concerns (17%). Among safety concerns were increased risk of death or of serious side effects, which were still the main reasons of failure in Phases II and III in 2012, and in 2019. For drug pipelines starting in 2007–2009, the gap in estimated success rates in 2012 was particularly steep between Phase II (first patient-dose) and Phase III (first clinical trial-dose). This means that Phase II, which is related to drug performance assessment, is particularly discriminatory: only 14% of the drug candidates that reached Phase II, compared to 64% of the drug pipelines reaching Phase III, were

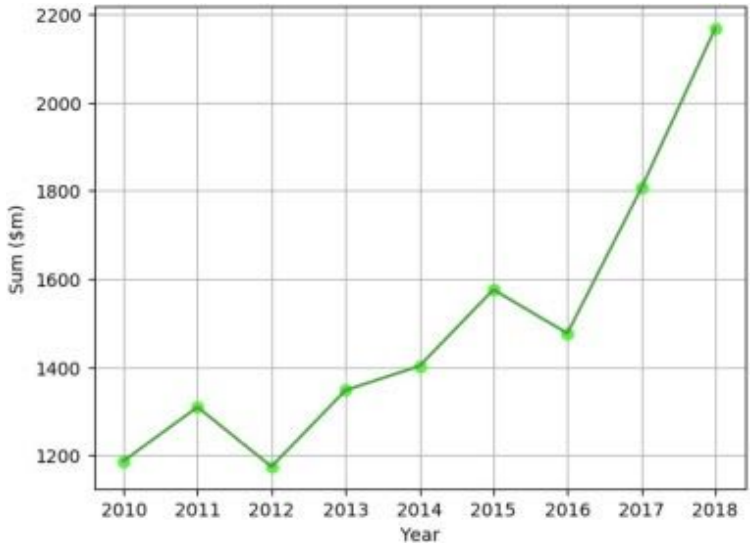
eventually marketed . This can still be observed for drug pipelines starting in 2015–2017 . 25% of the drug candidates that reached Phase II, compared to 62% of the drug pipelines reaching Phase III, were approved (estimation made in 2019).



Representation of the four stages of drug development, along with Phase IV, which occurs after the start of drug marketing.
Source:Machine learning applications in drug development
Clémence Rédaab,Emilie Kaufmannc ,Andrée Delahaye-Duriezade

Meanwhile, total capitalized expected cost of drug development was estimated at $868 million for approved drugs in 2006, with an average clinical development cost of $487 million, according to the public Pharma projects database . There are large variations due to drug type ($479 million for a HIV drug, compared to $936 million for a rheumatoid arthritis drug) . The recent study of a cohort of 12 large pharmaceutical labs led by shows that total development cost per approved drug has skyrocketed from 2016 to 2018 (from $1,477 to $2,168 million), and almost increased two-fold in eight years (from 2010 to 2018). It is worth noticing that, in the meantime, the number of discovered molecules that have reached the late clinical test stage for this cohort dropped by 22%. Moreover, clinical trials pose a barrier to rapid drug development. The cost and time involved in patient recruitment has been increasing.

There is a high failure rate and consequent financial loss in product development. Drug efficiency assessment might not be carried to term because of prematurely-stopped clinical testing, due to the lack of funding. These factors contribute to the decrease in approved drugs.



Evolution of average development cost in a cohort of 12 major pharmaceutical labs, in millions of dollars, between 2010 and 2018

Source:Machine learning applications in drug development
Clémence Rédaab,Emilie Kaufmannc ,Andrée Delahaye-Duriezade

Application Area of the project:

This project will be most applicable to Pharmaceutical Industries. This method can exponentially reduce the time for drug synthesis. The same algorithm can be altered for drug testing, DNA matching, etc. which involves generation of new compounds / strings.

This type of approach will be more useful in the time where conventional methods fail to meet the time limit in generating new medicines for a pandemic, like SARS, EBOLA, COVID-19. By factoring this new methodology we can cut the man hours exponentially in synthesizing new medicines.

Dataset Used:

We have used the open source dataset available at PubChem**.** To be precise we have used the CID-SMILES dataset. This dataset consists of 2 Billion unique chemical compounds spread across 21 different groups mostly relating to organic chemistry. Due to the lack of computational resources we have used only 2 Million compounds.

Algorithm Used:

Progressive Generative Adversarial Networks (PGAN)

Generative modeling is an unsupervised learning task in machine learning that involves automatically discovering and learning the regularities or patterns in input data in such a way that the model can be used to generate or output new examples that plausibly could have been drawn from the original dataset.

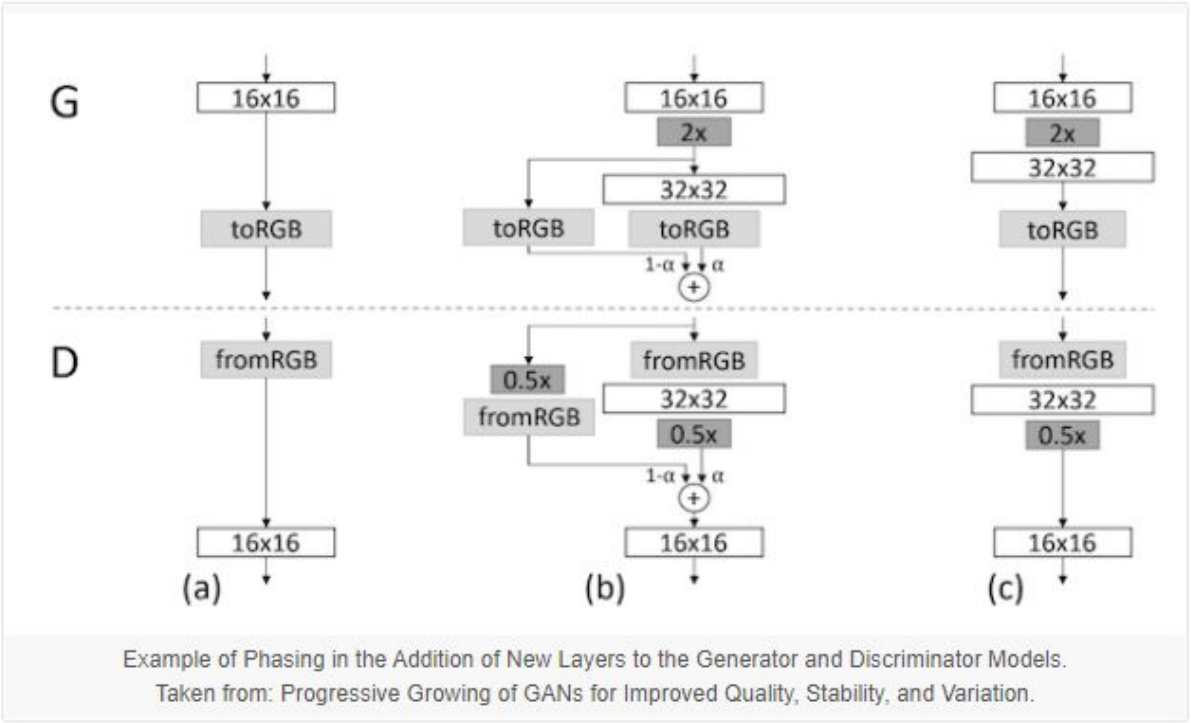
GANs are a clever way of training a generative model by framing the problem as a supervised learning problem with two sub-models: the generator model that we train to generate new examples, and the discriminator model that tries to classify examples as either real (from the domain) or fake (generated). The two models are trained together in a zero-sum game, adversarial, until the discriminator model is fooled about half the time, meaning the generator model is generating plausible examples.

GANs are an exciting and rapidly changing field, delivering on the promise of generative models in their ability to generate realistic examples across a range of problem domains, most notably in image-to-image translation tasks such as translating photos of summer to winter or day to night, and in generating photorealistic photos of objects, scenes, and people that even humans cannot tell are fake.

We also have trained a generative model by using LSTM inorder to compare it with GANs performance and we found out that GAN outperformed the LSTM model which will be explained in detail below.

In the case of the LSTM the input data is directly fed to it in SMILES format. But In the case of GAN the SMILES are converted to the molecular structures with the help of RDkit and those Images are fed to the Discriminator. So the possibility of Unstable compounds is very low in case of GAN compared to LSTM.

In this project we have used a specific GAN known as Progressive GAN (PGAN) which proved to be better in understanding the low level details of the input images. By using the concept of “Greedy Training”, adding layers and increasing the resolutions of the images after every iteration we can make the model to learn low level features and then make the neural network to decide which features are most important features in the subsequent iterations.



Example of Phasing in the Addition of New Layers to the Generator and Discriminator Models.
Taken from: Progressive Growing of GANs for Improved Quality, Stability, and Variation.

Advantages of Our model Compared to the Existing Work

Initially people had used Recurrent Neural Networks for generating new structures or molecules or compounds. The main disadvantage of the RNN based approach is that they are heavy and require large datasets. In this project we have simulated the result by using both RNN based approach and Adversarial approach (GAN). In the RNN side we have used LSTM and Progressive GAN for the GAN based approach. Recent studies have shown the main defects of using RNN based Neural Networks, people have started to use Transformer that is way lighter than the RNN counterparts for generating time-series analysis. On the other hand GAN undertakes a totally new category, it holds its position for having both generative and discriminative models in a single structure, before GAN came into existence, people have been using Auto-Encoders in one form or the other. The Auto-Encoders fail to address large datasets and complex pipelines. By learning the positives from all these methods we have used an architecture which prioritizes generative models and also makes sures that it can address low level features. The PGAN does exactly the same thing, being a dynamic neural network it can learn features along the way without much fine tuning of the hyper parameters, since the model will be trained for days.

In this project we have trained the LSTM model for 6 hours in a computer with specifications

CPU	Intel i5 8250U (4 cores)
RAM	8GB DDR3
GPU	NVIDIA MX150 (2GB VRAM)
Time Taken	6 hours
Number of Compounds Used	20k

The Progressive GAN requires more computationally expensive resources. It has been trained in two different phases. In the first phase we have trained the model using a couple of functional groups (Alcohol Aliphatic Compounds, Halogen Aromatic Compounds and Amine Cyclic Compounds). This model has been trained on an GCP (Google Cloud Processing) instance for approximately 9 days.

CPU	Intel Xeon (6 cores)
RAM	12GB DDR3
GPU	NVIDIA TESLA K80 (12GB VRAM)
Time Taken	~9 days
Number of Compounds Used	10M

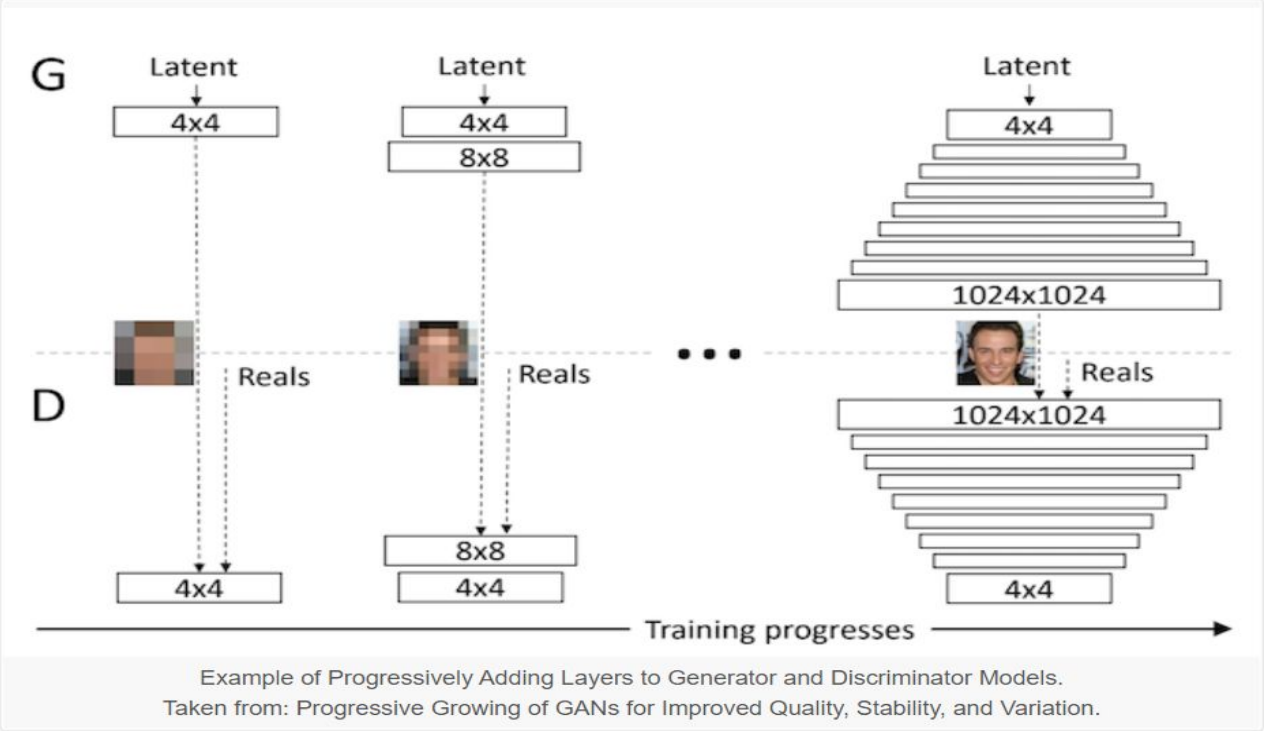
The weights from the phase 1 are obtained and then reused for phase 2 (Transfer Learning). This time due to lack of credits in our GCP account we have trained it on our local system. In this phase we have used all the available compounds in the datasets *(Alcohol, Alcohol Aliphatic, Amine, Alcohol Aromatic, Aldehyde, Aldehyde Aliphatic, Aldehyde Aromatic, AcidChloride, AcidChloride Aliphatic, AcidChloride Aromatic, Amine Aliphatic, Amine Primary, Amine Aromatic, Amine Cyclic, Amine Primary Aromatic, Amine Secondary, Amine Primary Aliphatic, CarboxylicAcid, Halogen, CarboxylicAcid Aliphatic, Amine Tertiary, Amine Tertiary Aliphatic, Halogen Aromatic, Halogen Aliphatic, Amine Secondary Aliphatic, CarboxylicAcid Aromatic, CarboxylicAcid AlphaAmino, Amine Secondary Aromatic, Amine Tertiary Aromatic, BoronicAcid, BoronicAcid Aliphatic, Azide, BoronicAcid Aromatic, Azide Aliphatic, Azide Aromatic, Halogen NotFluorine, Halogen NotFluorine Aromatic, Halogen NotFluorine Aliphatic, Nitro, Nitro Aromatic, Halogen Bromine, Halogen Bromine Aromatic, Halogen Bromine Aliphatic, TerminalAlkyne, Nitro Aliphatic, SulfonylChloride, SulfonylChloride Aromatic, Isocyanate, Isocyanate Aromatic, Isocyanate Aliphatic, Halogen Bromine BromoKetone, SulfonylChloride Aliphatic).* We again trained this model on top of the weights from phase 1 for 12hrs.

CPU	Intel i7 7700HQ
RAM	24GB DDR3
GPU	NVIDIA GTX 1060 (6GB VRAM)
Time Taken	12 Hours
Number of Compounds Used	20M

By using the Transfer Learning based approach we can be confident that our model can work on all scenarios.Since we have trained on only a couple of compounds initially we got pretty good results. Our model weights can be used to generate other chemical compounds outside our datasets, just by training on the new datasets for a couple of hours.

Justification of our Algorithm:

The main reason we have used LSTM over other RNN is because simple RNN is not complex enough to learn the features. Many research papers have proven that LSTM produces marginally better results compared to GRU (Gated Recurrent Units) on a performance trade off. Since we have initially decided to use the GAN based approach as our final approach, we need to get the best possible accuracy from the RNN based approach, that's why we chose LSTM. Coming to the GAN part, by going through many research papers we have learned that Progressive GAN gives the best results in this kind of problem. We tried the same and got good results from the model. We have also thought about using a Transformer based approach using multi-attention heads but due to the lack of time we haven't tried that. Overall this PGAN based approach gave good results when compared to the LSTM especially considering the stability of the molecules produced.



Example of Progressively Adding Layers to Generator and Discriminator Models.
Taken from: Progressive Growing of GANs for Improved Quality, Stability, and Variation.

CODE:

Get Functional groups

This part of the code takes the SMILES dataset as the input and outputs a csv file which contains the structure, functional group of the given compound.

```
#Importing all the necessary libraries

# rdkit is a library built on python which has all the essential functions necessary
for the drug synthesis
# rdkit is available only in anaconda because of its complex interface with compiled
C, this module cannot be downloaded in PIP
from rdkit import Chem
from rdkit.Chem import AllChem
from rdkit.Chem import Draw
from rdkit.Chem import FunctionalGroups

from time import time
import csv
```

```
start = time()
csvfile = open("fg_smils.csv", 'a')                 # open target csv file
header = ["pubchemId","functionalGroup", "SMILES"]  # These are the header for the
csv file to be created
writer = csv.writer(csvfile)
writer.writerow(header)                             # creates the csv file
smilesfile = "CID-SMILES"
```

```
fgs = FunctionalGroups.BuildFuncGroupHierarchy()    # rdkit function which returns
the functional group of the element
suppl = Chem.SmilesMolSupplier(smilesfile, delimiter=",",
        smilesColumn=1, nameColumn=0, titleLine=False )

mols = [x for x in suppl if x is not None]          # creating an array consisting
of all the functional groups available in the dataset
del suppl

print("processing %s with %d valid compounds" % (filename, (len(mols))))
def getFGs(fgs, res):                                 # this function generates all
the functional groups
    if not fgs:
```

```
        return
    for x in fgs:
        patt = x.pattern
        tmp = [m for m in mols if m.HasSubstructMatch(patt)]
        # if there are functional groups then check its children also
        if len(tmp):
            res[x.label] = {'mols': tmp, 'pattern': patt}
            getFGs(x.children, res)
        # end if
    # end for
    return
```

```
getFGs(fgs, zbbAllFGs)
len(zbbAllFGs)                                  # returns the length of all
the functional groups available
totalFGs = 0
for fgName in sorted(zbbAllFGs.keys()):
    totalFGs += len(zbbAllFGs[fgName]['mols'])
    for ml in zbbAllFGs[fgName]['mols']:
        id = ml.GetProp('_Name')
        smiles = Chem.MolToSmiles(ml)
        row = [id, fgName, smiles]
        writer.writerow(row)
    csvfile.flush()                             # writes all the preprocessed
information into a csv file
```

```
del mols
del zbbAllFGs
print("Total number of functional groups: %d" % (totalFGs))

csvfile.close()
end = time() - start
print(" Total time taken: ", end)               # shows the total time taken,
in our case it took approx ~7hrs
```

Draw molecule structure

This part of the code takes the csv file generated by the get_function_grops as the input and outputs the images of the functional groups using the built-in functions of rdkit

```
# Imports the necessary library

from time import time
import pandas as pd
import matplotlib.pyplot as plt
import os
from rdkit import Chem
from rdkit.Chem import AllChem
from rdkit.Chem import Draw
from rdkit.Chem import FunctionalGroups

from time import time
import csv
start = time()
```

```
baseImgDir = "./images/" # the sub direcotry where the generated images will be
stored
```

```
fgs = FunctionalGroups.BuildFuncGroupHierarchy()

# A template that stores the functional groups along with its sub direcotry
def makeTemplateDic(fgs, tempDic):
    for x in fgs:
        tempDic[x.label] = x.pattern
        makeTemplateDic(x.children, tempDic)
        # end if
    # end for
    return
```

```
templateDic = {}
makeTemplateDic(fgs, templateDic)
```

```
pd_iterator = pd.read_csv("./fg_smils.csv",  chunksize=20000)   # an iterator method
to iterte through th4e csv file, 20k lines at a time
```

```
## Get number of elements in each functional group.
fg_count = {}
for rowsDf in pd_iterator:
    rows = rowsDf.functionalGroup.value_counts()
    for row in rows.iteritems():
        if not row[0] in fg_count:
            fg_count[row[0]] = row[1]
        else:
            fg_count[row[0]] += row[1]
print(fg_count)                                                    # prints the total
number of compounds in every functional group
```

```
# the functional groups that we will be working on
usedfgs =
['Amine.Cyclic','Alcohol.Aliphatic','Halogen.Aromatic','CarboxylicAcid.AlphaAmino']
imgcount =
{'Amine.Cyclic':0,'Alcohol.Aliphatic':0,'Halogen.Aromatic':0,'CarboxylicAcid.AlphaAm
ino':0}

for rowsDf in pd_iterator:
    fgs = rowsDf.functionalGroup.unique()
    for ufg in fgs:
        if not ufg in usedfgs:
            continue
        if imgcount[ufg] >= 50000: # dont create more than 50K images per fg
            continue
        AllChem.Compute2DCoords(templateDic[ufg])
        rows = rowsDf[rowsDf['functionalGroup'] == ufg]
        for _, row in rows.iterrows():
            pubId = row["pubchemId"]
            smiles = row["SMILES"]
            mol = Chem.MolFromSmiles(smiles)
# converts the csv data into rdkit data type.
            AllChem.GenerateDepictionMatching2DStructure(mol,templateDic[ufg])
# generates the 2d structure based on the input compound
            dstdir = baseImgDir + ufg
            if not os.path.exists(dstdir):
                os.mkdir(dstdir)                                   # creates a new folder
for every functional group for easy access
            dstfile = dstdir +'/' + str(pubId) + ".png"
```

```
            Draw.MolToFile(mol,dstfile, size=(128, 128))      #rdkit inbuiilt
function which draws the molecular structure based on the string input
            imgcount[ufg] +=1                                 #counts the total
images processed.
```

PROGRESSIVE GAN

Progressive GAN involves many different libraries, which in reality cannot be built by one person. We have opted for help from the github community and came across many repositories, we have used mixture of many people work along with the necessary code from our side.
The repository which helped us the most was the one developed by the pyTorch team. since pytorch has the ability to use the CUDA cores we have used machines which have access to the NVIDIA GPUs, leaving behind the AMD counterparts.
Since our local machines are little powerful compared to the colab ones, we have used our local machines to train the model and also we have used cloud computing to do some of the heavy lifting.

```
# import the necessary libraries
import torch.optim as optim

from .base_GAN import BaseGAN
from .utils.config import BaseConfig
from .networks.progressive_conv_net import GNet, DNet
```

```
class ProgressiveGAN(BaseGAN):
    """
    Implementation of NVIDIA's progressive GAN.
    """

    def __init__(self,
                 dimLatentVector=512,
                 depthScale0=512,
                 initBiasToZero=True,
                 leakyness=0.2,
                 perChannelNormalization=True,
                 miniBatchStdDev=False,
                 equalizedlR=True,
                 **kwargs):
 if not 'config' in vars(self):
```

```
            self.config = BaseConfig()

        self.config.depthScale0 = depthScale0
        self.config.initBiasToZero = initBiasToZero
        self.config.leakyReluLeak = leakyness
        self.config.depthOtherScales = []
        self.config.perChannelNormalization = perChannelNormalization
        self.config.alpha = 0
        self.config.miniBatchStdDev = miniBatchStdDev
        self.config.equalizedlR = equalizedlR

        BaseGAN.__init__(self, dimLatentVector, **kwargs)
```

```
    def getNetG(self):

        gnet = GNet(self.config.latentVectorDim,
                    self.config.depthScale0,
                    initBiasToZero=self.config.initBiasToZero,
                    leakyReluLeak=self.config.leakyReluLeak,
                    normalization=self.config.perChannelNormalization,
                    generationActivation=self.lossCriterion.generationActivation,
                    dimOutput=self.config.dimOutput,
                    equalizedlR=self.config.equalizedlR)

        # Add scales if necessary
        for depth in self.config.depthOtherScales:
            gnet.addScale(depth)

        # If new scales are added, give the generator a blending layer
        if self.config.depthOtherScales:
            gnet.setNewAlpha(self.config.alpha)

        return gnet
```

```
def addScale(self, depthNewScale):
        """
        Add a new scale to the model. The output resolution becomes twice
        bigger.
        """
        self.netG = self.getOriginalG()
        self.netD = self.getOriginalD()

        self.netG.addScale(depthNewScale)
```

```
        self.netD.addScale(depthNewScale)

        self.config.depthOtherScales.append(depthNewScale)

        self.updateSolversDevice()
```

```
def updateAlpha(self, newAlpha):
        """
        Update the blending factor alpha.
        Args:
            - alpha (float): blending factor (in [0,1]). 0 means only the
                             highest resolution in considered (no blend), 1
                             means the highest resolution is fully discarded.
        """
        print("Changing alpha to %.3f" % newAlpha)

        self.getOriginalG().setNewAlpha(newAlpha)
        self.getOriginalD().setNewAlpha(newAlpha)

        if self.avgG:
            self.avgG.module.setNewAlpha(newAlpha)

        self.config.alpha = newAlpha
```

```
def getSize(self):
        """
        Get output image size (W, H)
        """
        return self.getOriginalG().getOutputSize()
```

LSTM Model:

We have created a LSTM model inorder to compare the results we obtained from GAN Here we have used the softmax as our activation function in the last layer because we are doing a multi class classification problem where the class corresponds to the english characters (elements)

Since LSTM involves a higher dimensional feature vector space, Adam is the most appropriate optimizer to use because of its adaptive nature and uses momentum. We have tested out many optimizer and this hypothesis is valid.

The loss is Categorical cross entropy because of the input shape we have opted for, we could have used KL Divergence but the difference in performance was not that significant. Another reason is that our GAN model also uses a loss function that is similar to categorical cross entropy.

In this model we have used LSTM layer twice, this was done to capture the high level temporal features, this proved to be an effective technique but requires more computational resources.

```
import tensorflow as tf
import numpy as np
from keras.models import Sequential
from keras.layers import Dense,LSTM,Dropout
from keras.layers.wrappers import TimeDistributed
import pandas as pd
```

```
def dimX(x,ts):      # get the dimentions of the input feature vectors
    x=np.asarray(x)
    newX=[]
    for i, c in enumerate(x):
        newX.append([])
        for j in range(ts):
            newX[i].append(c)
    return np.array(newX)
```

```
def dimY(Y,ts):               # get the dimentions of the output vector
    temp = np.zeros((len(Y), ts, len(chars)), dtype=np.bool)
    for i, c in enumerate(Y):
        for j, s in enumerate(c):

            temp[i, j, char_idx[s]] = 1
```

```
    return np.array(temp)
```

```
def prediction(preds):          # convert the categorical class sparse type
    y_pred=[]
    for i,c in enumerate(preds):
        y_pred.append([])
        for j in c:
            y_pred[i].append(np.argmax(j))
    return np.array(y_pred)
```

```
def seq_txt(y_pred):                    # make the output into a sequential array
    newY=[]
    for i,c in enumerate(y_pred):
        newY.append([])
        for j in c:
            newY[i].append(idx_char[j])

    return np.array(newY)
```

```
def smiles_output(s):                       # return the output in the smiles format
    smiles=np.array([])
    for i in s:
        j=''.join(str(k) for k in i)
        smiles=np.append(smiles,j)
    return smiles
```

```
import pandas as pd
data = pd.read_csv('lig_data.csv')          # read the input csv file
data=data.reindex(np.random.permutation(data.index))  # randomly shuffle the dataset
Y=data.SMILES
```

```
Y.head()
X=data.iloc[:,1:7]
X=X.values
X=X.astype('int')
type(X)
```

```
maxY=Y.str.len().max()             # total number of training examples
y=Y.str.ljust(maxY, fillchar='|')

ts=y.str.len().max()
```

```
chars = sorted(list( set("".join(y.values.flatten()))))    # calculating the total
number of characters in a molecule
print('total chars:', len(chars))
char_idx= dict((c, i) for i, c in enumerate(chars))
idx_char = dict((i, c) for i, c in enumerate(chars))
```

```
y_dash=dimY(y,ts)

x_dash=dimX(X,ts)

# creating the structure.

model = Sequential()
# time distributed layer is of 3dimention which holds the information of temporal
domain
# this time distributed dense layer is mostly used in RNN architecture where the
processing is required in all the time domains
model.add(TimeDistributed(Dense(x_dash.shape[2]),
input_shape=(x_dash.shape[1],x_dash.shape[2])))
model.add(LSTM(216, return_sequences=True))
model.add(Dropout(0.3))
model.add(LSTM(216, return_sequences=True))
model.add(TimeDistributed(Dense(y_dash.shape[2], activation='softmax')))
model.compile(loss='categorical_crossentropy', optimizer='adam')
```

```
model.fit(x_dash,y_dash,epochs=50)
```

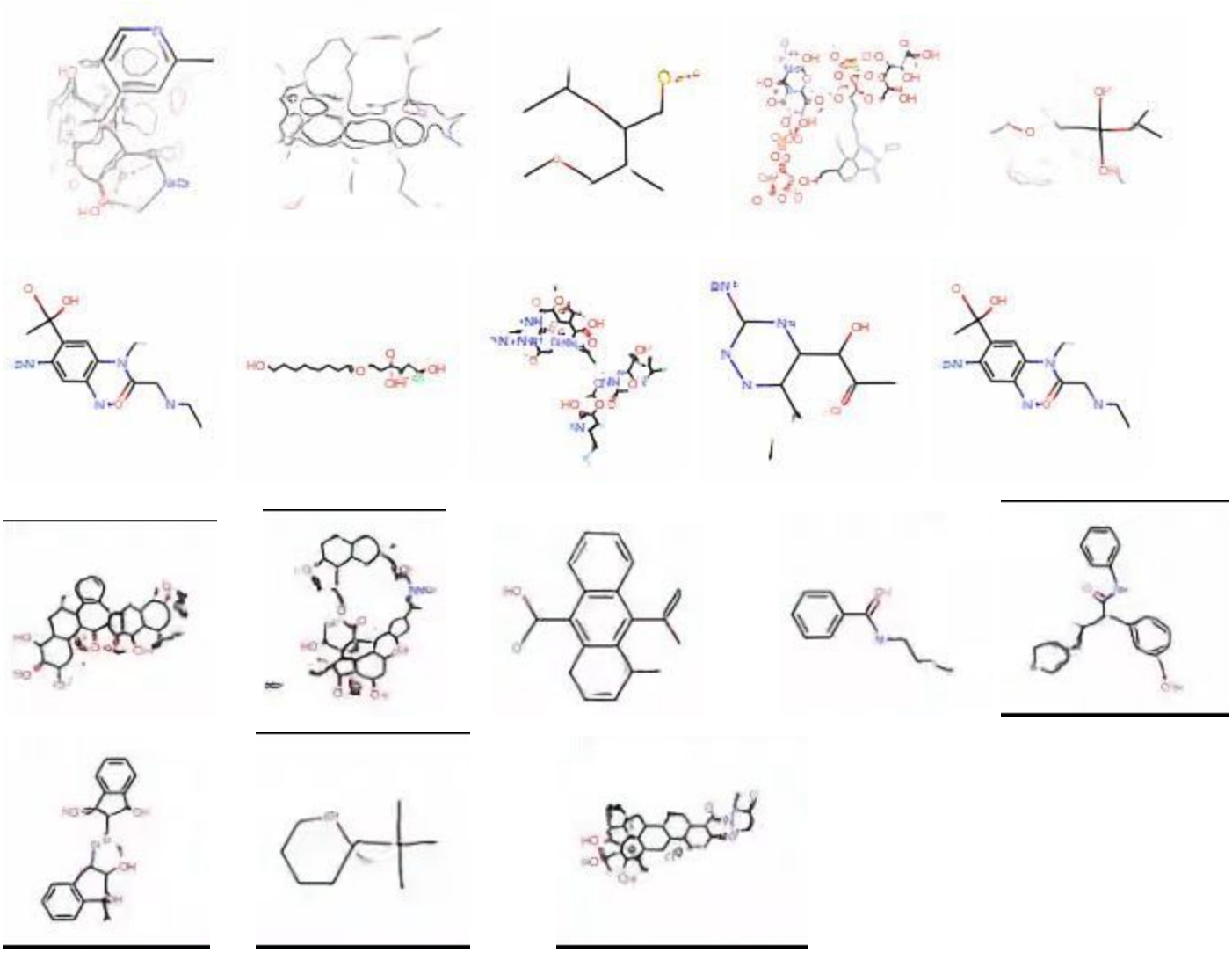
```
x_pred=[[0,0,0,1,0,0],
        [0,1,0,0,1,0],
        [0,1,0,0,0,1],[1,0,0,0,0,0]]
x_pred=dimX(x_pred,ts)
preds=model.predict(x_pred)
y_pred=prediction(preds)
y_pred=seq_txt(y_pred)
s=smiles_output(y_pred)
print(s)
```

Result Analysis:

From the results we obtained from both PGAN and LSTM models we learned two things.GAN produced more stable compounds compared to the LSTM based approach and GAN based approach produced compounds with more complexity compared to the LSTM based ones. From this we can know that PGAN has learnt intricate features of each and every compound, even though it requires more computational resources it proves that it is a viable option in the field of drug synthesis. The main advantage of using a PGAN over other GAN architectures is that it involves a dynamic learning approach that captures the right set of features at every stage which gives it an edge over cycle-GAN and style-GAN.

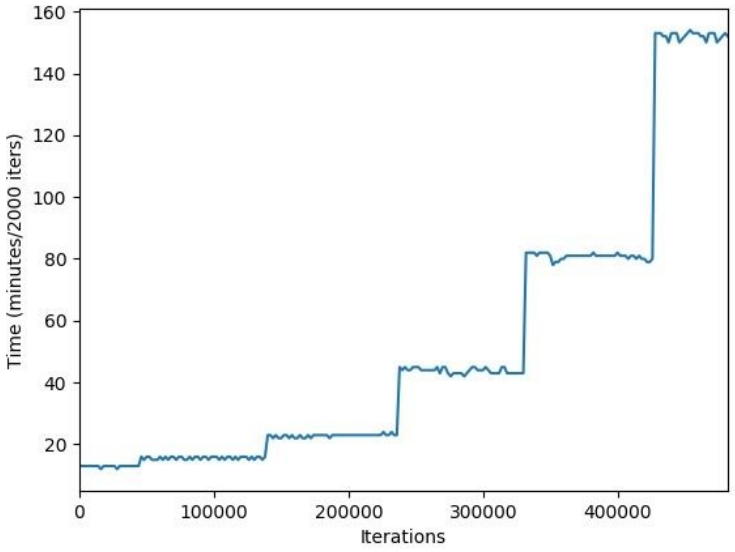
This model can be used in real world scenarios to generate molecules with particular requirements (with minimum tweaks). The compounds generated from this model are mostly stable which can exponentially decrease the man power required to validate each and every compound, thereby alleviating the pressure on the molecular biologist and biochemists.. In this era of artificial Intelligence utilizing such powerful tools makes the future an interesting place to look forward too.

Result obtained from the GAN model:



The outputs generated from the gan are of shape 128 x 128 x 3. We downsampled the shape of the input to match the output shape. The PGAN works on the principle of slow dynamic learning. The model initiates by taking the input of 4 x4 model using few layers, after a certain epoch (when the threshold loss has been reached) the layers and the input shape is increased based on the expression 2**(2*x) where x is the scale factor. Finally the model's output shape will be of 128 x 128. This accuracy has been reached after training the model for approximately 9 days.The compounds generated by the GAN are more stable and inert.

This feature has been learned from the input dataset. By cross validating the molecules generated by the GAN with the internet, we have found some pre-existing compounds.
We are happy with the results obtained from the model, with more training time and computational resources this model can be improved further.



Training time vs iterations

The graph indicates the training time compared to the number of iterations, its like a step function because of the dynamic nature of PGAN. This shows that new layers are added and the input shape is increased for every certain number of iterations. PGAN learns the implicit details of the feature space and tries to project that in the output, this what makes it different from other GAN architecture.

We can't show any validation results like confusion matrix, ROC Curve because we are utilizing the generator part of the GAN model. If we were using the discriminator part, it would be plausible to include the confusion matrix. We have tried to validate the structure by cross referencing with the tool available in the rdkit toolkit. It validates the stability of the molecule, and gives us an output. By passing the generated molecules to the rdkit validation function our loss (not stable compounds) were around 20% which is great enough to be used in real world scenario

Results Obtained from LSTM model:

```
x_pred=[[0,0,0,1,0,0],
        [0,1,0,0,0,0],
        [0,0,0,0,0,1],[1,0,0,0,0,0]]
x_pred=dimX(x_pred,ts)
preds=model.predict(x_pred)
y_pred=prediction(preds)
y_pred=seq_txt(y_pred)
s=smiles_output(y_pred)
print(s)
```

```
['CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCOOOO|||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||',
 'CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||',
 'CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC||||||||||||||||||||||||||||||||||||||||||||||||||',
 'CCCCC=CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC|||||||||||||||||||||||||||||||||||||||||||||||||||||||||']
```

```
x_pred=[[0,0,0,1,0,0],
        [0,1,0,0,0,0],
        [0,0,0,0,0,1],[1,0,0,0,0,0]]
x_pred=dimX(x_pred,ts)
preds=model.predict(x_pred)
y_pred=prediction(preds)
y_pred=seq_txt(y_pred)
s=smiles_output(y_pred)
print(s)
```

```
['CC(=O)NC1=C(CC=CC(CC(1CCC(CC)CCCCCC))OO||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||',
 'CC1=CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||',
 'NC(=CCCCCCCCCCCCCCCC)CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC||||||||||||||||||||||||||||||||||||||||||||||||',
 'C[S](=O)(=O)C1=CC=C(C=C1)C2=C(CCCCCCCCCCCCCCCCCCCC|||||||||||||||||||||||||||||||||||||||||||||||||||||||']
```

```
x_pred=[[0,0,0,1,0,0],
        [0,1,0,0,1,0],
        [0,1,0,0,0,1],[1,0,0,0,0,0]]
x_pred=dimX(x_pred,ts)
preds=model.predict(x_pred)
y_pred=prediction(preds)
y_pred=seq_txt(y_pred)
s=smiles_output(y_pred)
print(s)
```

```
['CC(=O)NC1=C(CC=CC(CC(1CCC(CC)CCCCCC))OO|||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||',
 'CCC1CCCCCCCCC1CCCCCCCCCCCCCCCCCCCCCC|||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||',
 'CC1=CC=CCCCCCCCCCCCCCCCC=CCCCCCCCCCCCCCC|||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||',
 'C[S](=O)(=O)C1=CC=C(C=C1)C2=C(CCCCCCCCCCCCCCCCCCC|||||||||||||||||||||||||||||||||||||||||||||||||||||']
```

From the output we can see that the LSTM captures some of the double bond essence but failed in the stability criteria, this could be fixed with increasing the dataset and distributing the dataset normally. changes to the architectures can also be made by using different techniques like K-Fold, ensemble and element embeddings. Making the model dynamic by introducing the concept of Reinforcement learning could have increased the performance. In our Future work, we will try to incorporate all the above mentioned concepts to create an appropriate test platform.

Inference based on the results:

1) GAN is more powerful in generating complex molecular structures compared to that of an RNN based approach
2) RNN based approach can be used in places where we want to generate compounds with very long chains, like cellulose, hemicellulose, lignin, etc.
3) The compounds generated by the RNN based approach may not be stable compared to their counterparts (GAN based approach)
4) Compounds generated using GAN are more stable and can learn complex intricate features.
5) Research has shown that PGAN works well in 3D feature space, which led us to feed the input in image format, resulting in more stable compounds.
6) Convolutional layers have proved a viable option to extract the features from the input data.
7) All the results are of 128 x 128 shape. By increasing the resolution in the input vector shape we can obtain more detailed images.
8) These models have learnt the importance of double bond in places where it's completely necessary.
9) By modifying the model structure we can use this approach in discovering the intermediate compounds in a cross-chemical reaction between two complex molecules, helping us in better understanding of the world of atoms and molecules.
10) From this experiment we have learned that including more feature extraction in the model results in learning more fine elements of the input space.
11) We have fed the input images (GAN approach) without normalizing the color space, this led to an interesting discovery. The model has learnt the color code for each and every functional group. This makes the outputs more interesting and easy to validate

Outcome:

From this project we have understood the working of GAN in a more explicit manner, we have also learned about the ideology behind the Drug Synthesis, stability of chemical compounds, generation of molecules, SMILES notation of the dataset, working of LSTM, working of Neural Networks, and Dynamic way of training of neural networks.

We have also learned about the speciality of Progressive GAN, LSTM, and importance of feature extraction. In this project we have referred to a lot of research papers and learned about different approaches to drug synthesis. In our future work we would like to try out new architectures like transformers, CTC loss.

References:

1 *Clémence Réda, Emilie Kaufmann, Andrée Delahaye-Duriez,Machine learning applications in drug development, Computational and Structural Biotechnology Journal, Volume 18, 2020, Pages 241-252, ISSN 2001-0370, https://doi.org/10.1016/j.csbj.2019.12.006. (http://www.sciencedirect.com/science/article/pii/S2001037019303988)*

2. *Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, & Yoshua Bengio. (2014). Generative Adversarial Networks.*

3. *Gupta, Anvita & Müller, Alex & Huisman, Berend & Fuchs, Jens & Schneider, Petra & Schneider, Gisbert. (2017). Generative Recurrent Networks for De Novo Drug Design. Molecular Informatics. 37. 10.1002/minf.201700111.*

4.*Hongming Chen, Ola Engkvist, Yinhai Wang, Marcus Olivecrona, Thomas Blaschke, The rise of deep learning in drug discovery,Drug Discovery Today,Volume 23, Issue 6, 2018,Pages 1241-1250,ISSN 1359-6446, https://doi.org/10.1016/j.drudis.2018.01.039. (http://www.sciencedirect.com/science/article/pii/S1359644617303598)*

5. *Tero Karras, Timo Aila, Samuli Laine, & Jaakko Lehtinen. (2018). Progressive Growing of GANs for Improved Quality, Stability, and Variation*.