ML Applications

Sarthak Toshniwal

July 26, 2024

1. Supervised Learning

Supervised Learning is a subset of Machine Learning where we have labeled data where each training example is mapped to an output label. The goal of Supervised Learning is to learn a mathematical curve from the training dataset which predicts outputs for new labels to some degree of accuracy.

1.1 Feature Vectors and Target Labels

Each training example has to be modified such that it contributes significantly to the performance of the model and has high predictive power.

Feature – A variable or property of the data that is being used for analysis.

We can identify the relevant features by plotting their dependence with the target values and then pick the features with highest correlation.

We can also engineer new features that are a mathematical function of the given features and which have high predictive power.

After the identification of important features, each data point is assembled into one vector called a feature vector which is the input to our model and each feature vector has a labeled output called a target label.

1.2 Feature Engineering and Scaling

The identifications of relevant features and engineering of new features which are relevant to the problem at hand is known as Feature Engineering.

Feature Scaling involves scaling of features by techniques like mean normalization or Z-score normalization which enhances the numerical stability of computations and ensures that all features contribute equally. Empirical observations suggest that it also improves the model performance and also gives faster and assured convergence in optimization algorithms like Gradient Descent.

1.3 Cost Function

Cost function, also known as loss or objective function, is a mathematical function used to measure the performance of the model. It quantifies the difference between predicted and target values for each data point in the dataset. The main aim during training is to minimize this function which improves model's accuracy.

This is done using optimization algorithms like Gradient Descent.

There are many types of cost functions and we choose one for our model depending on the problem at hand.

Some of the common loss functions are –

1. Mean Squared Error (MSE) – Average of sum of the square of the difference between predicted and target values. Mainly used for regression tasks.

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(Y_i - \hat{Y}_i)^2$$

MSE	= mean squared error
n	= number of data points
Y_i	= observed values
$\hat{Y}_i$	= predicted values

2. Mean Absolute Error (MAE) – Average of sum of absolute difference between predicted and target values. Also used for regression tasks.

$$\text{MAE} = \frac{1}{n}\sum_{j=1}^{n}|y_j - \hat{y}_j|$$

3. Cross-Entropy Loss (log loss) – Measure of difference between the probability distributions of true labels and predicted probabilities. Mainly used for classification tasks.

$$LogLoss = -\frac{1}{n}\sum_{i=0}^{n}[y_i log(\hat{y}_i) + (1 - y_i)\log(1 - \hat{y}_i)]$$

1.4 Optimization Algorithms

Minimizing the cost functions while training a model is done using optimization algorithms like Gradient Descent, Adam Optimizer etc.

The most common algorithm is Gradient Descent.

Gradient Descent:

We assume the output label to be a function of the features. The expression would turn out to be a sum involving all the features multiplied by some coefficients called weights. These weights are what known as tunable parameters which means we can alter them to improve our model's accuracy.

There is another thing called the learning rate which determines the amount by which every tunable parameter changes in every iteration of Gradient Descent.

Steps of Gradient Descent:

1. Initialization – Start with some random values for the parameters.

2. Cost function – Calculate the current value of cost function.

3. Update Parameters – Adjust the parameters to reduce the cost. For Gradient Descent this means decreasing the weight in the direction of the gradient of cost function with respect to that weight.

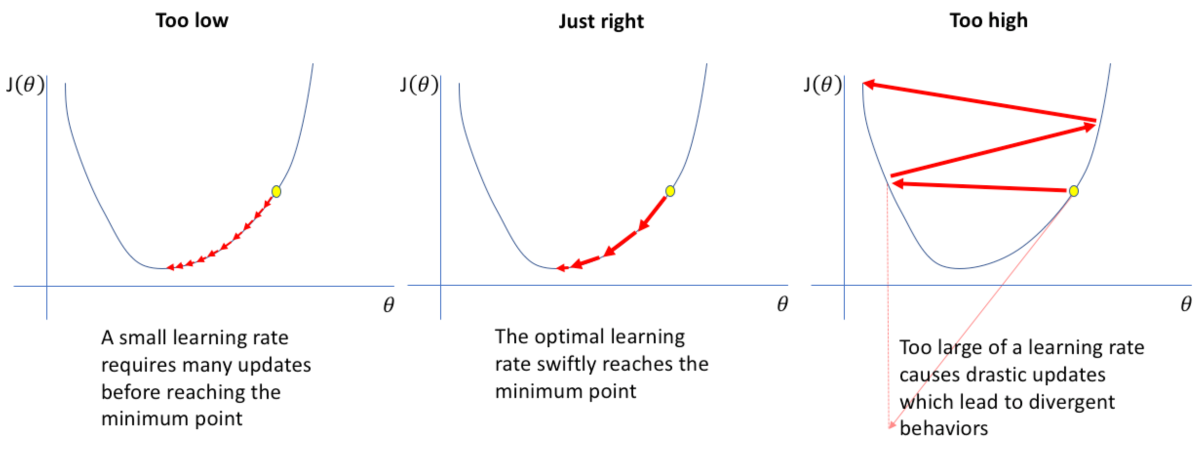
$$\theta_j \leftarrow \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta)$$

In the above figure, J is the cost function, α the learning rate, and θ_j the weight.

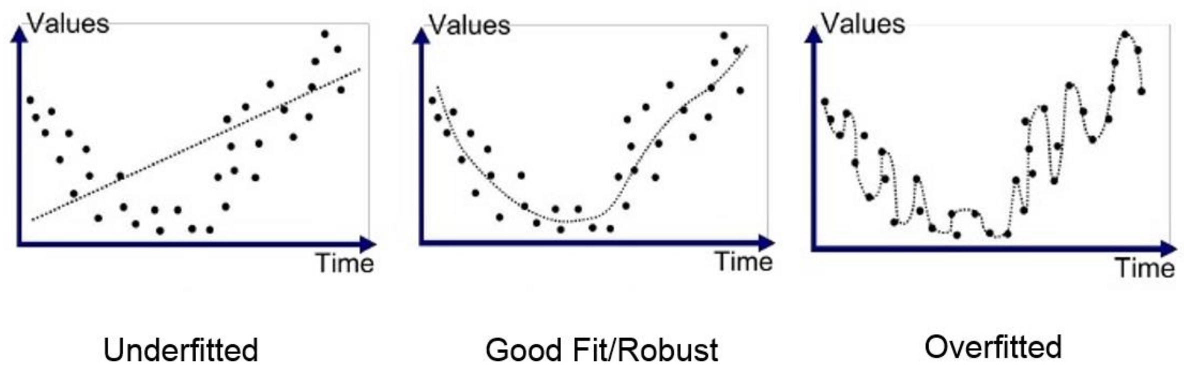
Repeat steps 2 and 3 until convergence that is the loss function no longer decreases significantly.

Some things to keep in mind while performing Gradient Descent:

1. Set a low learning rate. This helps the model converge faster and prevents any oscillations.


Too low
Just right
Too high
J(θ)
θ
A small learning rate requires many updates before reaching the minimum point
The optimal learning rate swiftly reaches the minimum point
Too large of a learning rate causes drastic updates which lead to divergent behaviors

2. Use Regularization to prevent overfitting to data



3. Can also modify batch size, use momentum and advanced optimizers like Adam which are offered by many Machine Learning libraries like TensorFlow and PyTorch.

1.5 Regularization

Regularization is a technique in Machine Learning which helps to prevent overfitting to training data. Regularization adds a penalty term to the cost function.

Regularization Techniques:

1. L1 Regularization – Adds absolute values of weights to cost function.

Cost = Loss + $\lambda \sum |\theta j|$

L1 Regularization can help drive some features to zero which removes unnecessary features.

2. L2 Regularization – Adds squares of weights to the cost function.

Cost = Loss + $\lambda \sum \theta j^2$

L2 Regularization shrinks all the weights of unnecessary features to values close to zero.

In above expressions, λ is known as the regularization parameter.

We can also use both L1 and L2 Regularization simultaneously.

1.6 Regression and Classification

A problem in Supervised Learning is mainly of two types, either of Regression or of Classification.

Regression:

The main task in regression is to identify a mathematical relationship between features and output labels.

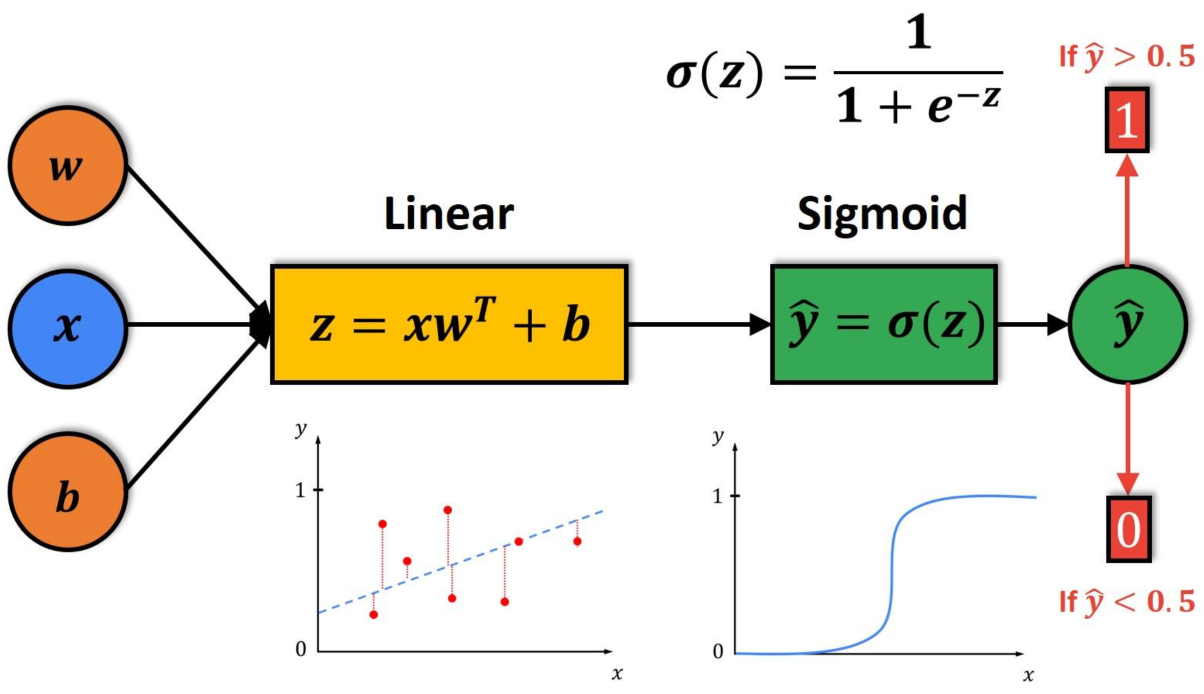
Regression can be Linear, Multiple Linear, or Polynomial regression.

Classification:

The main task in classification is to classify an input into predefined categories based on its features. Classification can be Binary or Multi-Class Classification.

Usually, a decision boundary or a classification threshold is used to classify inputs.

For binary classification, we mainly use Logistic Regression whose overall structure is as follows.



1.6 Train-Test-Validation Split

It is good practice to break the available data into training data that is used to train the model, test data which is used to check the model's accuracy and performance, and finally validation data that checks if the model hasn't overfit the training data.

The split ratio depends from model to model but a 80:10:10 split is considered good.

2. Advanced Learning Algorithms

2.1 Neural Networks

Neural networks are the foundation of deep learning. Their architecture is similar to the neurons in a biological brain.

Components of a Neural Network:

1. Neuron (Node) – The very basic unit of a neural network that performs computation on the input and applies an activation function to produce an output.

2. Layers – Neurons are organized into layers within a neural network.

There are three types of layers namely:

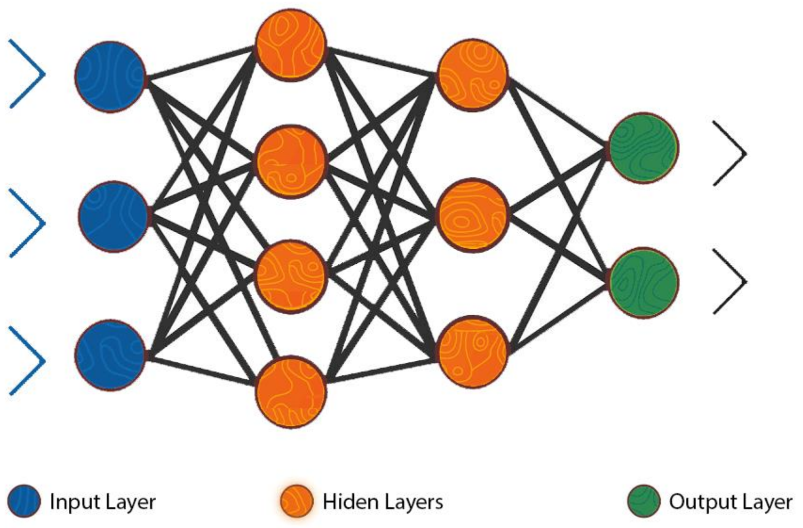
- Input layer – Receives input.
- Hidden layer(s) – Intermediate layers between input and output layer of a neural network which perform computations.
- Output layer – Receives the final output of a network.

3. Weights and Biases – Each connection between any two neurons of adjacent layers has a weight associated to it which determines the strength of the influence of one neuron on

another. Biases are constant parameters added to the input in the neurons to help the model fit the data better.

4. Activation Function – After multiplying features with respective weights and adding the bias term, this weighted sum of inputs is put into an activation function like sigmoid, tanh (hyperbolic tangent), ReLU (Rectified Linear Unit), Leaky ReLU, and Softmax (mainly used in the output layer in a classification problem).

5. Feedforward and Backpropagation – Neural networks use feedforward propagation to pass data through the network from input to output. Backpropagation is used to update weights and biases during training by propagating the error backward through the network, adjusting weights to minimize the difference between predicted and actual outputs.



Above illustration depicts how the neurons are fired in a neural network ultimately producing the final output.

3. Unsupervised Learning

In Unsupervised Learning the data we have does not have any target labels.

There are many ways to deal with unlabeled data, the most common being the K-means clustering. In this algorithm our main goal is to create clusters of closely related data points in space. There is an underlying assumption that close points in space are closely related to each other, hence points within a cluster must have the same output label.

3.1 The K-means clustering algorithm

The main aim of this algorithm is to map all of the data points into K mutually exclusive clusters with the underlying assumption that points in the same cluster are similar and points of two different clusters are dissimilar.

Steps:

1. Initialization – Choose the number K of clusters that you want to make and also initialize the K cluster centroids randomly. A centroid is a point representing the center of a cluster.

2. Assign points to clusters – A datapoint which is closest to a particular centroid is assigned to the cluster whose center is that centroid, breaking ties arbitrarily. Here the term close is with respect to the distance of the feature vector of the data point from the cluster centroid in a suitable vector space.

3. Update cluster centroids – Compute the centroids of all the clusters by taking the average of the features of all the data points in that cluster and update the value of the centroid to this average vector.

Repeat steps 2 and 3 until convergence that is there is no change in points among clusters and the centroids have stabilized or after a certain number of iterations.

4. Reinforcement Learning

4.1 Q-Learning algorithm

The main agenda of a Q-Learning algorithm is to learn an action-value function denoted as Q(s, a) where s denotes the state and a denotes the action. By exploring the environment using an exploration-exploitation strategy, it tries to converge to the optimal policy by having knowledge of the state and action space.

The meaning of the word policy is the action the agent may take when it is in a specific state.

Q(s, a) represents the expected cumulative reward for taking action ‘a’ in state ‘s’ and following the optimal policy thereafter until it reaches the terminal state.

Steps:

1. Initialization – Initialize Q(s, a) to some default value.

2. For each episode repeat:

- Initialize state s
- Repeat until you reach the terminal state:
    - Choose action ‘a’ using an exploration-exploitation strategy (like the ε-greedy strategy)
    - Take action ‘a’, observe reward ‘r’ and next state ‘s`’
    - Update Q(s, a) using the Bellman Equation
    - Move to the next state ‘s`’

The above algorithm converges to optimal Q-values Q*(s, a) under the conditions of sufficient exploration.

Bellman Equation

$$Q_{i+1}(s,a) = R + \gamma \max_{a'} Q_i(s',a')$$

R is the reward that you get right away by taking action ‘a’ at state ‘s’.

ϒ (gamma) is the discount factor that makes future rewards less valuable than immediate rewards.

ε-greedy strategy

This is a kind of exploration-exploitation strategy in which we set a parameter ε known as the exploration rate. Exploration involves trying new actions apart from the ones that yield high rewards based on past experience.

ε is the probability of choosing a random action. Usually, it’s a small number.

5. Applications in Drug Discovery

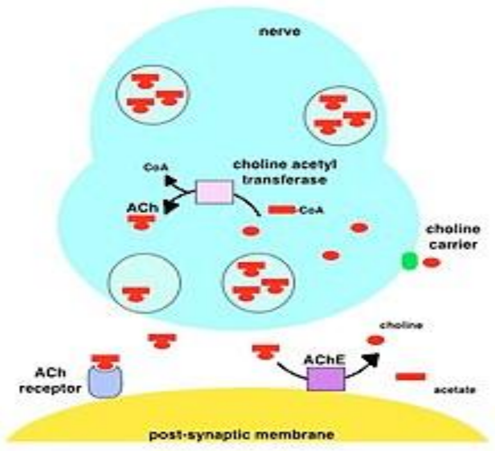
With the basic knowledge of Machine Learning, we can implement a model that accelerates the process of Drug Design and Discovery. For the purpose of this project a generative model which creates new possible lead compounds which are effective against Alzheimer’s Disease has been implemented.

5.1 Alzheimer’s Disease

Alzheimer’s disease is a neurodegenerative condition that progresses subtly from its onset. Symptoms in the clinical setting encompass memory deterioration, speech difficulties, apraxia, agnosia, deficits in visual spatial abilities, executive function disturbances, and shifts in personality and behaviour, etc. The etiology is unknown so far. Alzheimer’s disease has the characteristics of long course of disease, many causes and complicated pathology. There are other irregularities of neurotransmitters in the center in addition to the drop-in acetylcholine levels in the brain.

The aggregation of A, the disturbance of metal ion metabolism, the imbalance of calcium balance, the rise in free radicals, and the onset of inflammation are the primary causes of Alzheimer’s disease. In view of the above causes, the therapeutic targets of Alzheimer’s disease mainly include acetylcholinesterase (AChE), metal ions, Beta Amyloid Peptide (β-AP), monoamine oxidase (MAO), free radicals, tau protein, N-methyl-D-aspartate (NMDA) receptor and other related targets.

Acetylcholine (ACh) is the first neurotransmitter discovered by human beings, and its mediated neurotransmission is the basis of nervous system function. Sudden interruption of ACH mediated neurotransmission is fatal, and its gradual loss is associated with progressive deterioration of cognitive, autonomic and neuromuscular functions. AChE is an efficient hydrolase that catalyses the hydrolysis of ACh, which regulates the concentration of ACh at synapses and then terminates ACh-mediated neurotransmission.



The above figure shows how Acetylcholinesterase (AChE) catalyses the hydrolysis of acetylcholine (ACh), which regulates the concentration of ACh at synapses and then terminates ACh mediated neurotransmission. AChE is a very efficient hydrolase that breaks about 5000 molecules of ACh per second!

This problem can be resolved by discovering drugs which inhibit the activity of AChE in the brain by binding to them. This class of compounds known as AChE inhibitors are not new in the pharma industry and some compounds are already known which inhibit the activity of AChE but their side-effects are inevitable. Therefore, there is an increasing demand for developing active compounds with stronger inhibitory function and minimal side effects.

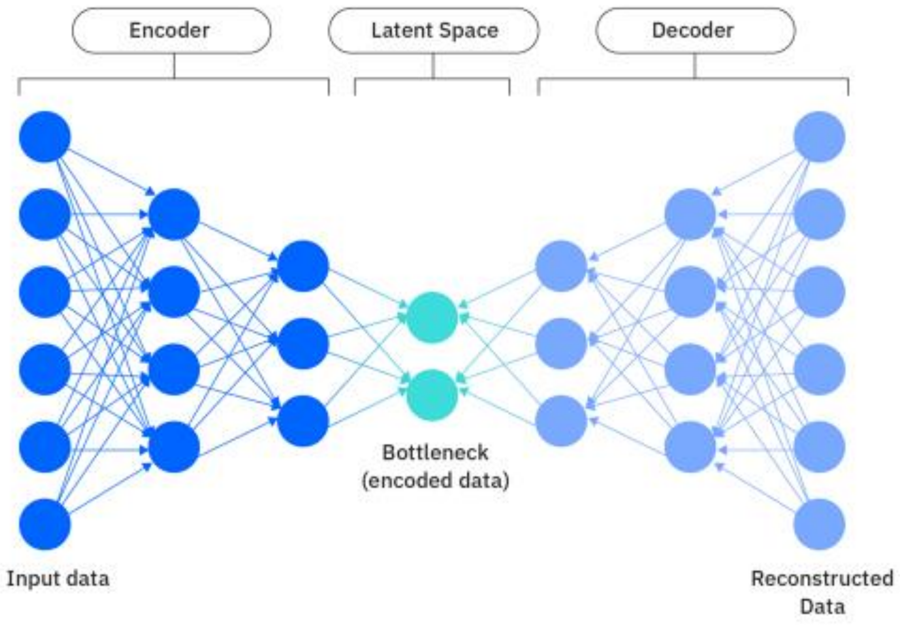
With the advent of Machine Learning and Artificial Intelligence, it is possible to generate molecules with desired properties by feeding some valid examples known as training data.

5.2 Variational Autoencoders

Our model's framework is based upon Variational Autoencoders.

Variational autoencoders are deep learning models composed of an encoder that learns to isolate the important latent variables from training data and a decoder that then uses those latent variables to reconstruct the input data.

VAEs encode a continuous, probabilistic representation of the latent space which is nothing but the collection of latent variables of a specific set of input data. On a fundamental level, the function of an autoencoder is to effectively extract the data's most salient information-its latent variables-and discard irrelevant noise. This enables a VAE to not only accurately reconstruct the exact original input, but also use variational inference to generate new data samples that resemble the original input data.



In training, the encoder network passes input data from the training data set through a "bottleneck" before it reaches the decoder. The decoder network, in turn, is then responsible for reconstructing the original input by using only the vector of latent variables.

After each training epoch, optimization algorithms such as gradient descent are used to adjust model weights in a way that minimizes the difference between the original data input and the decoder's output. Eventually, the encoder learns to allow through the information most conducive to accurate reconstruction and the decoder learns to effectively reconstruct it.

Autoencoder structure

Though different types of autoencoders add or alter certain aspects of their architecture to better suit specific goals and data types, all autoencoders share three key structural elements:

The **encoder** extracts latent variables of input data *x* and outputs them in the form of a vector representing latent space *z*. In a typical "vanilla" autoencoder, each subsequent layer of the encoder contains progressively fewer nodes than the previous layer; as data traverses each encoder layer, it's compressed through the process of "squeezing" itself into fewer dimensions.

Other autoencoder variants instead use regularization terms, like a function that enforces *sparsity* by penalizing the number of nodes that are activated at each layer, to achieve this dimensionality reduction.

The **bottleneck**, or **"code,"** is both the output layer of the encoder network and the input layer of the decoder network. It contains the latent space: the fully compressed, lower-dimensional embedding of the input data. A sufficient bottleneck is necessary to help ensure that the decoder cannot simply copy or memorize the input data, which would nominally satisfy its training task but prevent the autoencoder from learning.

The **decoder** uses that latent representation to reconstruct the original input by essentially reversing the encoder: in a typical decoder architecture, each subsequent layer contains a progressively larger number of active nodes.

VAEs are probabilistic models. VAEs encode latent variables of training data not as a fixed discrete value **z**, but as a continuous range of possibilities expressed as a probability distribution ***p*(*z*)**.

In Bayesian statistics, this learned range of possibilities for the latent variable is called the prior distribution. In variational inference, the generative process of synthesizing new data points, this prior distribution is used to calculate the posterior distribution, ***p*(*z*|*x*).** In other words, the value of observable variables **x,** given a value for latent variable **z**.

For each latent attribute of training data, VAEs encode two different latent vectors: a vector of means, "***μ***," and a vector of standard deviations, "***σ***." In essence, these two vectors represent the range of possibilities for each latent variable and the expected variance within each range of possibilities.

By randomly sampling from within this range of encoded possibilities, VAEs can synthesize new data samples that, while unique and original unto themselves, resemble the original training data.

Two important elements of VAEs are:

1) Reconstruction loss –

VAEs use reconstruction loss, also called reconstruction error, as a primary loss function in training. Reconstruction error measures the difference (or "loss") between the original input data and the reconstructed version of that data output by the decoder. Multiple algorithms, including cross-entropy loss or mean-squared error (MSE), can be used as the reconstruction loss function.

The autoencoder architecture creates a bottleneck that allows only a subset of the original input data to pass through to the decoder. At the onset of training, which typically begins with a randomized initialization of model parameters, the encoder has not yet learned which parts of the data to weigh more heavily. As a result, it will initially output a suboptimal latent representation, and the decoder will output a fairly inaccurate or incomplete reconstruction of the original input.

By minimizing reconstruction error through some form of gradient descent over the parameters of the encoder network and decoder network, the weights of the autoencoder model will be adjusted in a way that yields a more useful encoding of latent space (and thus a more accurate reconstruction). Mathematically, the goal of the reconstruction loss function is to optimize $p_\theta(z|x)$, in which θ represents the model parameters that enforce the accurate reconstruction of input **x** given latent variable **z**.

However, the goal of a variational autoencoder is not to reconstruct the original input; it's to generate new samples that resemble the original input. For that reason, an additional optimization term is needed which is called the Kullback-Leibler divergence.

2) Kullback-Leibler divergence –

VAEs incorporate a regularization term called Kullback-Leibler divergence, or KL divergence which is responsible for preventing the bottleneck part to overfit the training data.

To generate new data samples, the VAE must be able to sample from anywhere in the latent space between the original data points. For this to be possible, the latent space must exhibit two types of regularity:

- **Continuity:** Nearby points in latent space should yield similar content when decoded.
- **Completeness:** Any point sampled from the latent space should yield meaningful content when decoded.

A simple way to implement both continuity and completeness in latent space is to help ensure that it follows a standard normal distribution, called a Gaussian distribution. But minimizing only reconstruction loss doesn't incentivize the model to organize the latent space in any particular way, because the "in-between" space is not relevant to the accurate reconstruction of the original data points. This is where the KL divergence regularization term comes into play.

KL divergence is a metric used to compare two probability distributions. Minimizing the KL divergence between the learned distribution of latent variables and a simple Gaussian distribution whose values range from 0 to 1 forces the learned encoding of latent variables to follow a normal distribution. This allows for smooth interpolation of any point in latent space, and thereby the generation of new data points.

With all of the basics covered we can now implement our model in code!

5.3 PED Model Implementation[1]

The first step involves installing and importing all the libraries that we will be working with.

[87…

```
!pip install chembl_webresource_client
!pip install rdkit
!pip install numpy pandas openpyxl
!pip install torch
```

[2]:

```
# Import necessary libraries for data preparation and model implementation
import numpy as np
import pandas as pd
from chembl_webresource_client.new_client import new_client
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from rdkit import Chem
```

Next, we read the data and extract the SMILES notation of all the compounds in a single list.

[55…

```
# Data preparation
df = pd.read_excel('Drugs_ChEMBL.xlsx', usecols=[0])
df_split = df[df.columns[0]].str.split(';', expand=True)
smiles = df_split.iloc[:, 31].tolist()
smiles = [item for item in smiles if item is not None]
# Removing the double quotes at the beginning and end of smiles notation
smiles = [x[1:-1] for x in smiles if x]
```

[56…

```
df_split
```

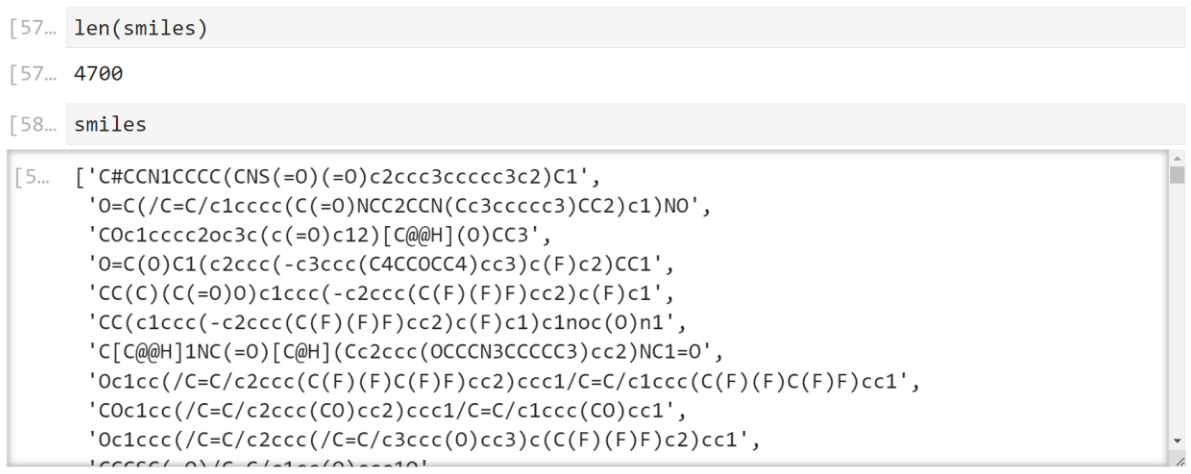
[56…

	0	1	2	3	4	5	6	7	8	9	...	28	29	30
0	CHEMBL3943366	""	""	"Small molecule"	""	"342.46"	"7"	"8"	"2.46"	"49.41"	...	"-1.75"	"NEUTRAL"	"C19H22N2O2S
1	CHEMBL5268446	""	""	""	""	"393.49"	"6"	"9"	"2.85"	"81.67"	...	"-0.79"	"BASE"	"C23H27N3O3
2	CHEMBL5269968	""	""	""	""	"232.24"	"1"	"2"	"1.78"	"59.67"	...	"1.20"	"NEUTRAL"	"C13H12O4
3	CHEMBL193634	""	""	"Small molecule"	""	"340.39"	"4"	"8"	"4.50"	"46.53"	...	"-0.27"	"ACID"	"C21H21FO3
4	CHEMBL194692	""	""	"Small molecule"	""	"326.29"	"4"	"8"	"4.87"	"37.30"	...	"-0.71"	"ACID"	"C17H14F4O2
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
4744	CHEMBL38	"TRETINOIN"	"ABEREL|ACTICIN|ALL-TRANS RETINOIC ACID|ALL-TR...	"Small molecule"	"4.0"	"300.44"	"849"	"2273"	"5.60"	"37.30"	...	"2.31"	"ACID"	"C20H28O2
4745	CHEMBL242948	"POLAPREZINC"	"Carnosine|POLAPREZINC|Z 103|Z-103"	"Small molecule"	"2.0"	"226.24"	"31"	"114"	"-1.13"	"121.10"	...	"0.01"	"ZWITTERION"	"C9H14N4O3
4746	CHEMBL169	"URSOLIC ACID"	"MALOL|MICROMEROL|NSC-167406|NSC-4060|PRUNOL|U...	"Small molecule"	"2.0"	"456.71"	"213"	"948"	"7.09"	"57.53"	...	"3.19"	"ACID"	"C30H48O3
4747	CHEMBL15192	"LAPACHONE"	".BETA.-LAPACHONE|ARQ 501|ARQ-501|BETA LAPACHO...	None	None	None	None	None	None	None	...	None	None	None
4748	CHEMBL563	"FLURBIPROFEN"	"ANSAID|BTS 18	None	None	None	None	None	None	None	...	None	None	None

4749 rows × 38 columns

[1]The actual paper implements a very vast model that takes all of the compounds in the Zinc15 database as training examples and the neural network is itself very complex. So, a simple version of the model has been implemented here that takes the drugs used to treat Alzheimer's disease as training examples. The database of the same was taken from https://www.kaggle.com/code/nicolascafuri/ml-for-alzheimer-s-drug-discovery-in-progress#2.-QSAR:-Unveiling-the-Significance-of-Molecular-Structure-in-Drug-Design.

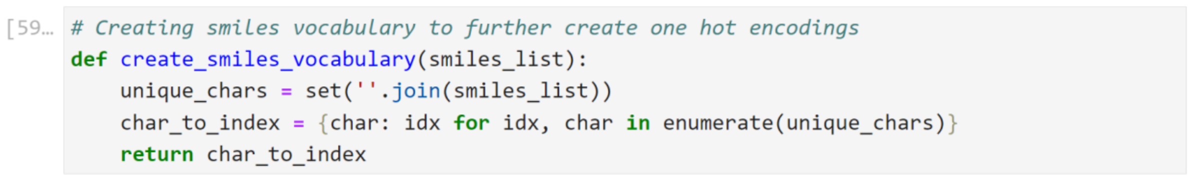
This is what our 'smiles' list looks like after taking care of all the null data. It contains 4700 data points.



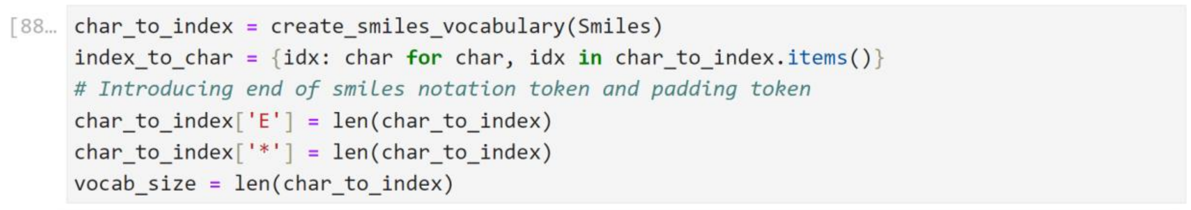
```
[57… len(smiles)
[57… 4700
[58… smiles
[5…  ['C#CCN1CCCC(CNS(=O)(=O)c2ccc3ccccc3c2)C1',
      'O=C(/C=C/c1cccc(C(=O)NCC2CCN(Cc3ccccc3)CC2)c1)NO',
      'COc1cccc2oc3c(c(=O)c12)[C@@H](O)CC3',
      'O=C(O)C1(c2ccc(-c3ccc(C4CCOCC4)cc3)c(F)c2)CC1',
      'CC(C)(C(=O)O)c1ccc(-c2ccc(C(F)(F)F)cc2)c(F)c1',
      'CC(c1ccc(-c2ccc(C(F)(F)F)cc2)c(F)c1)c1noc(O)n1',
      'C[C@@H]1NC(=O)[C@H](Cc2ccc(OCCCN3CCCCC3)cc2)NC1=O',
      'Oc1cc(/C=C/c2ccc(C(F)(F)C(F)F)cc2)ccc1/C=C/c1ccc(C(F)(F)C(F)F)cc1',
      'COc1cc(/C=C/c2ccc(CO)cc2)ccc1/C=C/c1ccc(CO)cc1',
      'Oc1ccc(/C=C/c2ccc(/C=C/c3ccc(O)cc3)c(C(F)(F)F)c2)cc1',
```

The most important aspect of any generative model is the data that is given to it. A model cannot take a molecular structure as input, rather we have to convert our training examples into tensors for fast computation. This can be done by one hot encoding all of the smiles notation of every data point which will be given as input to the PED model.

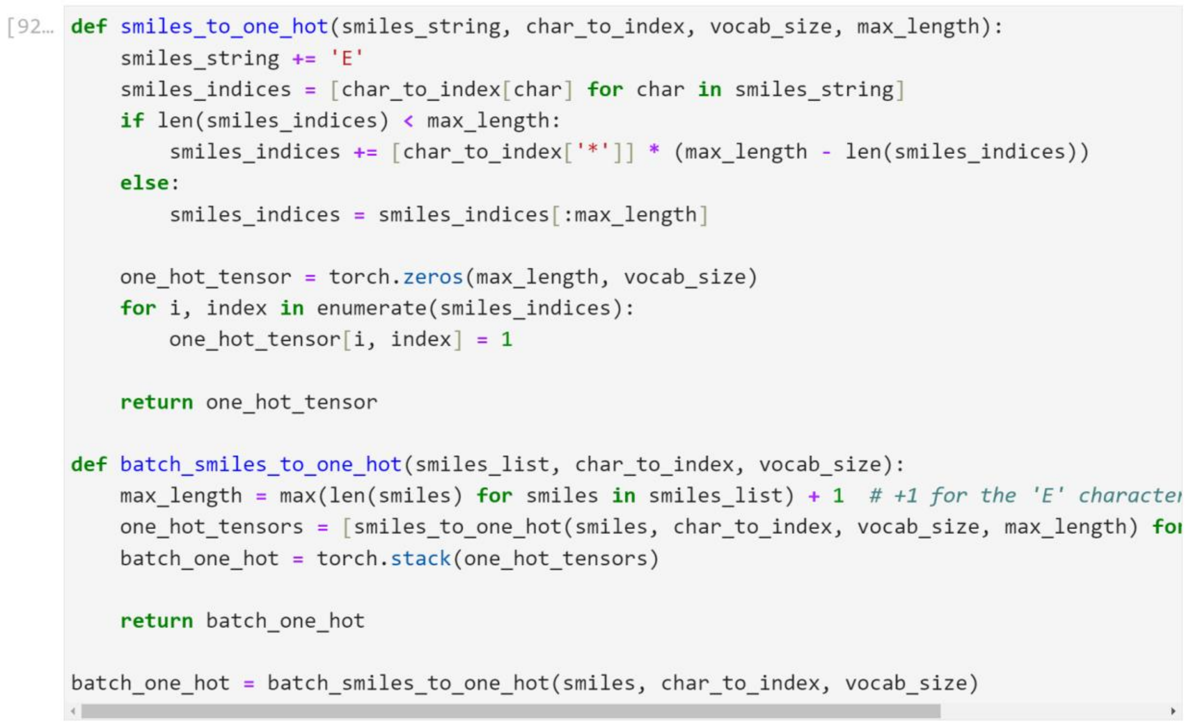
But first we need to create a vocabulary of all the unique characters that appear in the smiles notation of the training examples.



```
[59… # Creating smiles vocabulary to further create one hot encodings
     def create_smiles_vocabulary(smiles_list):
         unique_chars = set(''.join(smiles_list))
         char_to_index = {char: idx for idx, char in enumerate(unique_chars)}
         return char_to_index
```



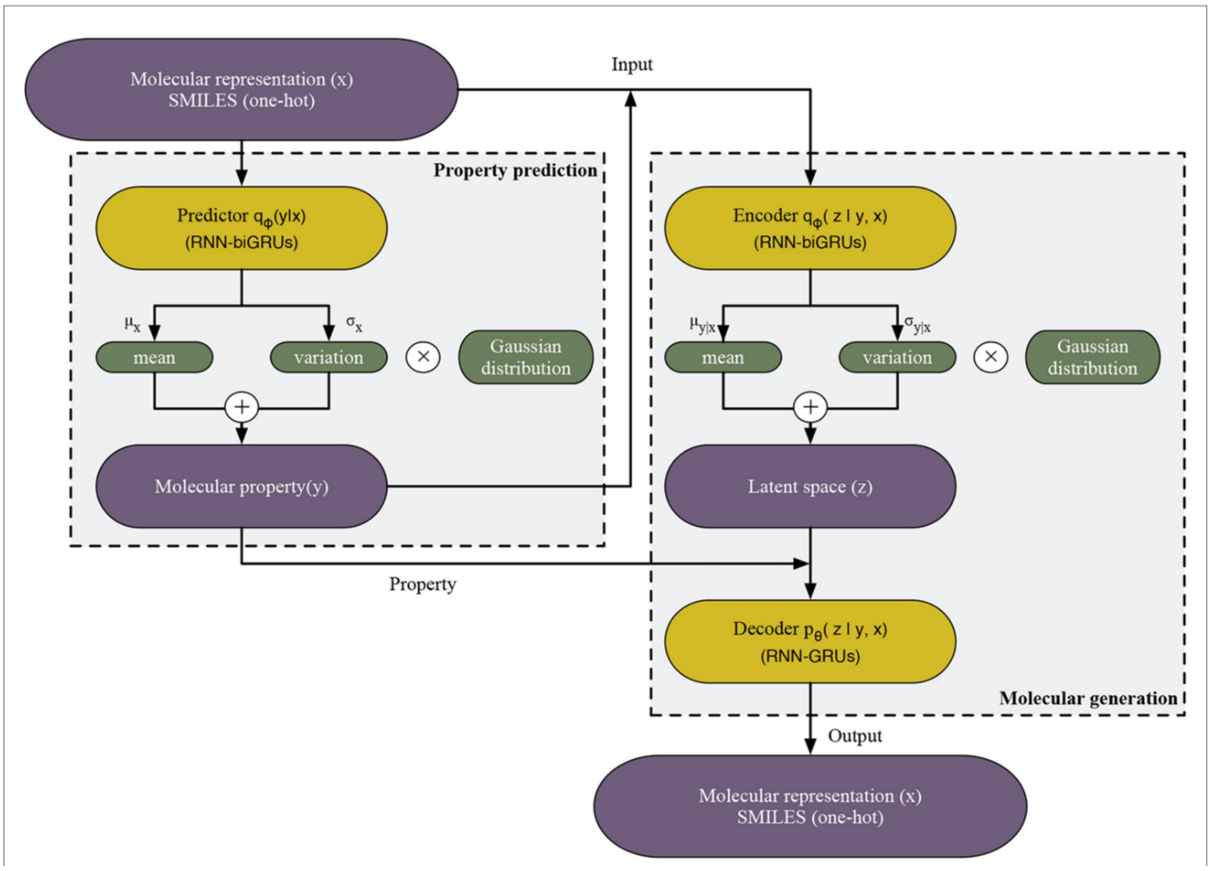
```
[88… char_to_index = create_smiles_vocabulary(Smiles)
     index_to_char = {idx: char for char, idx in char_to_index.items()}
     # Introducing end of smiles notation token and padding token
     char_to_index['E'] = len(char_to_index)
     char_to_index['*'] = len(char_to_index)
     vocab_size = len(char_to_index)
```



```
[92… def smiles_to_one_hot(smiles_string, char_to_index, vocab_size, max_length):
         smiles_string += 'E'
         smiles_indices = [char_to_index[char] for char in smiles_string]
         if len(smiles_indices) < max_length:
             smiles_indices += [char_to_index['*']] * (max_length - len(smiles_indices))
         else:
             smiles_indices = smiles_indices[:max_length]

         one_hot_tensor = torch.zeros(max_length, vocab_size)
         for i, index in enumerate(smiles_indices):
             one_hot_tensor[i, index] = 1

         return one_hot_tensor

     def batch_smiles_to_one_hot(smiles_list, char_to_index, vocab_size):
         max_length = max(len(smiles) for smiles in smiles_list) + 1  # +1 for the 'E' characte
         one_hot_tensors = [smiles_to_one_hot(smiles, char_to_index, vocab_size, max_length) fo
         batch_one_hot = torch.stack(one_hot_tensors)

         return batch_one_hot

     batch_one_hot = batch_smiles_to_one_hot(smiles, char_to_index, vocab_size)
```

The code cell above creates one hot encodings of all the training examples.

Next, we implement the PED class which defines the neural network that we train in the model.



The above figure (Figure 1[1]) illustrates the workflow of the PED model.

The Predictor takes one hot encoded SMILES input(x) and predicts molecular properties(y) and creates a distribution from which a sample is drawn.

The Encoder encodes the predicted molecular property(y) and the original SMILES input(x) into a latent space representation(z) and a sample is further drawn from this distribution.

The Decoder then decodes the latent space representation back into a new molecular representation.

The model consists of three gated recurrent unit (GRU) networks: the predictor network, the encoder network and the decoder network. The predictor and encoder are made up of bidirectional GRUs network, while the decoder is a unidirectional GRU.

[68…
```python
# Define the PEDModel class
class PEDModel(nn.Module):
    def __init__(self, input_dim, hidden_dim, output_dim, vocab_size):
        super(PEDModel, self).__init__()
        self.hidden_dim = hidden_dim

        # Define the predictor (bidirectional GRU)
        self.predictor_gru = nn.GRU(input_dim, hidden_dim, bidirectional=True, batch_first=

        # Define the encoder (bidirectional GRU)
        self.encoder_gru = nn.GRU(hidden_dim * 2, hidden_dim, bidirectional=True, batch_fir

        # Define the decoder (unidirectional GRU)
        self.decoder_gru = nn.GRU(hidden_dim, hidden_dim, batch_first=True)

        # Dense layers for the latent variables
        self.hidden2mean = nn.Linear(hidden_dim * 2, hidden_dim)
        self.hidden2logvar = nn.Linear(hidden_dim * 2, hidden_dim)

        # Output layers
        self.hidden2property = nn.Linear(hidden_dim * 2, output_dim)
        self.hidden2output = nn.Linear(hidden_dim, vocab_size)
        self.softmax = nn.Softmax(dim=-1)

    def forward(self, x):
        # Predictor forward pass
        predictor_output, _ = self.predictor_gru(x)
        predictor_output = predictor_output[:, -1, :]  # Take the output of the last time

        # Encoder forward pass
        encoder_output, _ = self.encoder_gru(predictor_output.unsqueeze(1))
        encoder_output = encoder_output[:, -1, :]  # Take the output of the last time step

        # Latent variables
        mu = self.hidden2mean(encoder_output)
        logvar = self.hidden2logvar(encoder_output)
        std = torch.exp(0.5 * logvar)
        z = mu + std * torch.randn_like(std)

        # Property prediction
        y_pred = self.hidden2property(encoder_output)

        return y_pred, mu, logvar, z
```

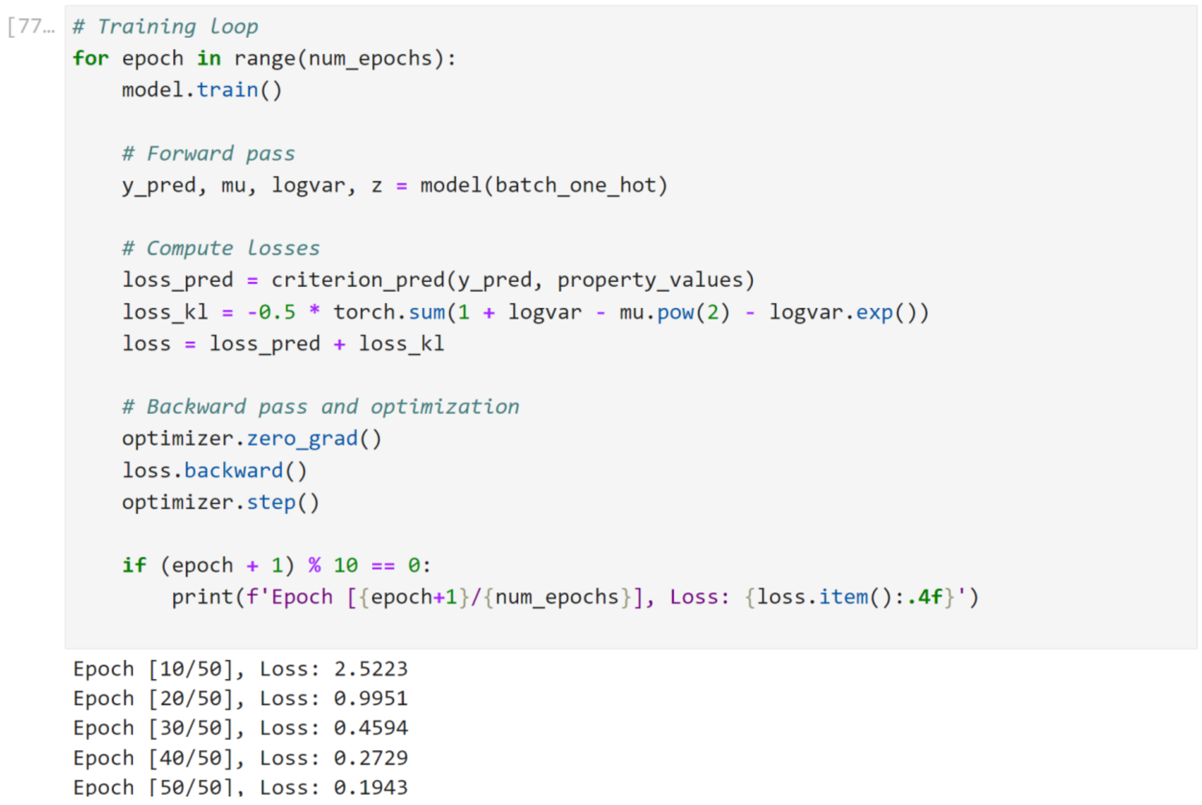
The next few cells define the hyperparameters of the model and

[3]:
```python
# Using device GPU to train the model faster
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
```

[9]:
```python
# Example property values for the list of SMILES strings
property_values = torch.rand(len(smiles), 1)
```

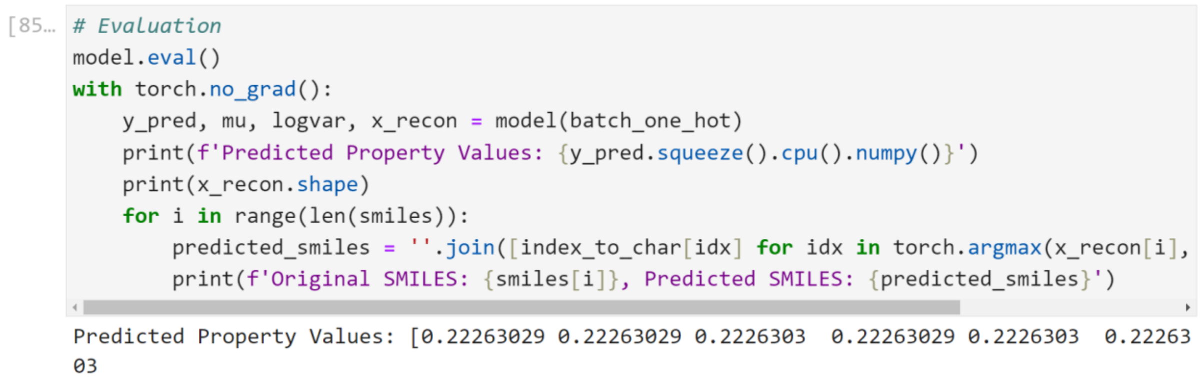
[10…
```python
# Initialize model, loss functions, and optimizer
model = PEDModel(input_dim, hidden_dim, output_dim, vocab_size).to(device)
criterion_pred = nn.MSELoss()
criterion_recon = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=learning_rate)

batch_one_hot = batch_one_hot.to(device)
property_values = property_values.to(device)
```

Next, we train the model and evaluate its performance.



```
[77… # Training Loop
     for epoch in range(num_epochs):
         model.train()

         # Forward pass
         y_pred, mu, logvar, z = model(batch_one_hot)

         # Compute losses
         loss_pred = criterion_pred(y_pred, property_values)
         loss_kl = -0.5 * torch.sum(1 + logvar - mu.pow(2) - logvar.exp())
         loss = loss_pred + loss_kl

         # Backward pass and optimization
         optimizer.zero_grad()
         loss.backward()
         optimizer.step()

         if (epoch + 1) % 10 == 0:
             print(f'Epoch [{epoch+1}/{num_epochs}], Loss: {loss.item():.4f}')
```

```
Epoch [10/50], Loss: 2.5223
Epoch [20/50], Loss: 0.9951
Epoch [30/50], Loss: 0.4594
Epoch [40/50], Loss: 0.2729
Epoch [50/50], Loss: 0.1943
```

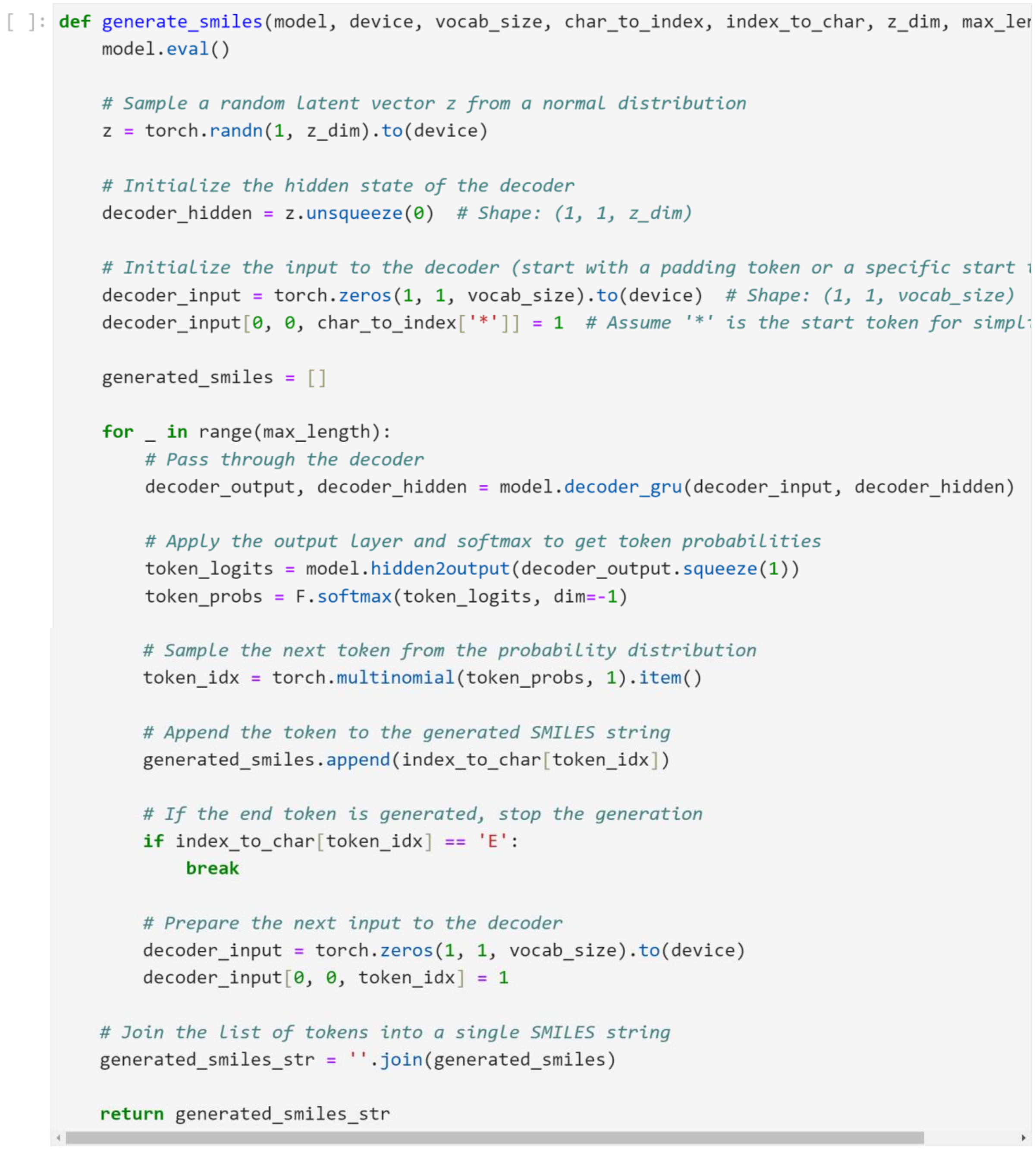


```
[85… # Evaluation
     model.eval()
     with torch.no_grad():
         y_pred, mu, logvar, x_recon = model(batch_one_hot)
         print(f'Predicted Property Values: {y_pred.squeeze().cpu().numpy()}')
         print(x_recon.shape)
         for i in range(len(smiles)):
             predicted_smiles = ''.join([index_to_char[idx] for idx in torch.argmax(x_recon[i],
             print(f'Original SMILES: {smiles[i]}, Predicted SMILES: {predicted_smiles}')
```

```
Predicted Property Values: [0.22263029 0.22263029 0.2226303  0.22263029 0.2226303  0.22263
03
```

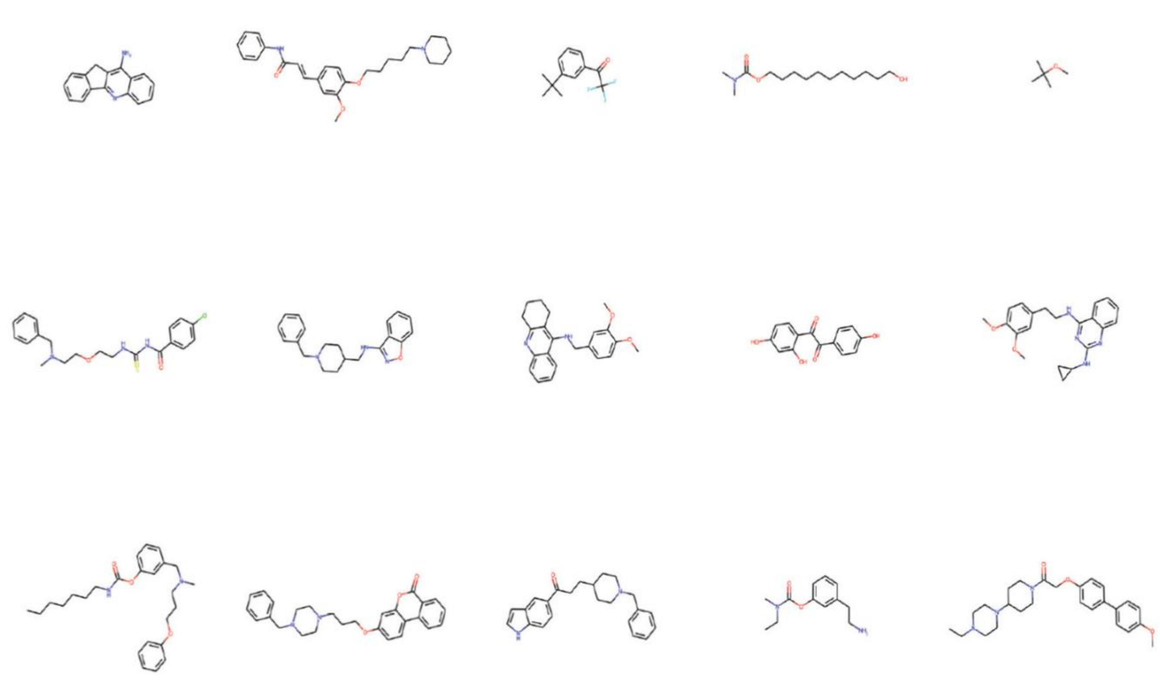
After we get the predicted property values, we can generate new SMILES notations with desired properties from the latent space and the convert these SMILES notations to their molecular structures and evaluate their docking scores with AChE using molecular docking softwares. The compounds with the highest docking scores are the possible lead compounds which can further be tested in a laboratory to evaluate their efficacy and side effects.

The code to generate new SMILES notations from the generative model's latent space is given below.



```python
def generate_smiles(model, device, vocab_size, char_to_index, index_to_char, z_dim, max_le
    model.eval()

    # Sample a random latent vector z from a normal distribution
    z = torch.randn(1, z_dim).to(device)

    # Initialize the hidden state of the decoder
    decoder_hidden = z.unsqueeze(0)  # Shape: (1, 1, z_dim)

    # Initialize the input to the decoder (start with a padding token or a specific start
    decoder_input = torch.zeros(1, 1, vocab_size).to(device)  # Shape: (1, 1, vocab_size)
    decoder_input[0, 0, char_to_index['*']] = 1  # Assume '*' is the start token for simpl

    generated_smiles = []

    for _ in range(max_length):
        # Pass through the decoder
        decoder_output, decoder_hidden = model.decoder_gru(decoder_input, decoder_hidden)

        # Apply the output layer and softmax to get token probabilities
        token_logits = model.hidden2output(decoder_output.squeeze(1))
        token_probs = F.softmax(token_logits, dim=-1)

        # Sample the next token from the probability distribution
        token_idx = torch.multinomial(token_probs, 1).item()

        # Append the token to the generated SMILES string
        generated_smiles.append(index_to_char[token_idx])

        # If the end token is generated, stop the generation
        if index_to_char[token_idx] == 'E':
            break

        # Prepare the next input to the decoder
        decoder_input = torch.zeros(1, 1, vocab_size).to(device)
        decoder_input[0, 0, token_idx] = 1

    # Join the list of tokens into a single SMILES string
    generated_smiles_str = ''.join(generated_smiles)

    return generated_smiles_str
```

Some of the possible lead compounds that we get from the model are (Figure 2[1]) –



Donepezil is a compound that selectively inhibits AChE and is widely used in the treatment of Dementia of Alzheimer's type. Comparing the molecules predicted by our model to Donepezil is another benchmark which validates the correctness of the model.

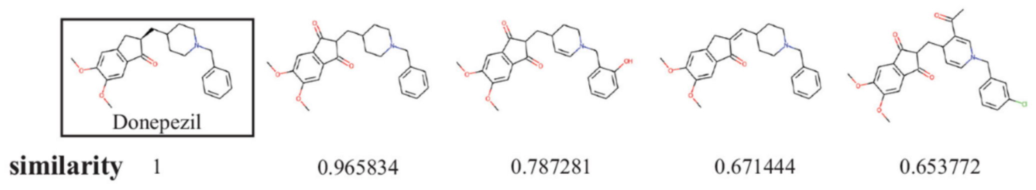


FIGURE 11
The most similar molecules to Donepezil among the generated molecules.

Figure 3 – Molecules generated by the model similar to Donepezil[1]

Note that we get above results only by training the model on the entire Zinc15 database and then evaluating their docking scores which takes a lot of time and technical resources like GPUs.

This is how AI and ML is creating a significant impact in the pharmaceutical industry as we now don't need to synthesize the drug molecules from scratch. Also, these models better extract all the patterns in the data and accelerate the process of drug discovery.

But still there are some things that we have to take care of like ensuring that the data being fed into the model is authentic and is properly pre-processed into a form on which our generative models can perform fast computations to evaluate results. Although data is at the core of the functioning of our model, other things like the type of model that we are using, tuning the model's hyperparameters, deciding how complex our model should be, what optimisation techniques it should use while training to learn the patterns in the data are very important to take care of.

Finally, having an optimal model is great but we should also ensure that it is not harming our society in any way. Our model should be fair in every respect and should not learn to bias on the basis of race, caste, gender, social and cultural values. Their working and decision making should be transparent to its users. Also, our model should take care of the privacy and personal data of the user. This ensures the safety of data and also helps in building user's trust.

With such things taken care of we are sure to progress in this field towards a better future!

References –

1) PED Research Paper - Liu D, Song T, Na K, Wang S. PED: a novel predictor-encoder-decoder model for Alzheimer drug molecular generation. Frontiers in Artificial Intelligence. 2024 Apr 16; 7:1374148.

[1]Figure 1,2 and 3 are taken as references from the above cited research paper.

2) Variational Autoencoder Images and Theory - https://www.ibm.com/think/topics/variational-autoencoder

3) AChE Mechanism - https://en.wikipedia.org/wiki/File:AChe_mechanism_of_action.jpg