The Reinforcement Learning for Structural Evolution method on a Docker Environment

Ivan Buccella and Matteo Maiorano
Department of Computer Science
University of Salerno
Italy

Abstract

The ReLeaSE (Reinforcement Learning for Structural Evolution) method is a new computational strategy for de novo design of molecules with desired properties. This strategy integrates two deep neural networks, one generative and one predictive, which are trained separately but used jointly to generate new targeted chemical libraries. ReLeaSE uses a simple representation of molecules only through SMILES (simplified molecular-input line-entry system) strings. The generative models are trained with a stack-augmented memory network to produce chemically feasible SMILES strings, and predictive models are derived to predict the desired properties of the generated de novo compounds. In the first stage of the method, the generative and predictive models are trained separately with a supervised learning algorithm. In the second stage, both models are jointly trained with the Reinforcement Learning approach to guide the generation of new chemical structures toward those with the desired physical and/or biological properties. The ReLeaSE method was used to design chemical libraries with a bias toward structural complexity or toward compounds with maximal, minimal or specific physical properties, such as melting point or hydrophobicity, or toward compounds with inhibitory activity toward Janus protein kinase 2. Based on these explanations, the main work focuses on the execution of the method by training a recurrent neural network to predict logP from SMILES; in addition, several optimizations to the code are applied in order to create a platform-independent execution using modern technologies like Docker and Docker Compose.

1 Introduction

The World Economic Forum has called the combination of big data and artificial intelligence (AI) the fourth industrial revolution, capable of radically transforming science. AI is revolutionizing several fields of medicine, such as radiology, pathology and other specialties, and has begun to be used in drug discovery through deep learning (DL) technologies. These technologies are being applied in different areas, such as molecular docking, transcriptomics, understanding reaction mechanisms, and molecular energy prediction. In drug discovery, a crucial step is to formulate a well-reasoned hypothesis on how to synthesize new compounds or select them from available chemical libraries based on structure-activity relationship (SAR) data. Automated design approaches to create compounds with specific properties have been the subject of research for the past 15 years. Although there are many synthetically feasible chemicals that could be considered as possible drug-like molecules, their huge number makes it impossible to systematically examine all of them and verify them through the construction and evaluation of each individual compound. In this document we present an execution of a novel method to generate de novo chemical compounds with desired physical, chemical and/or bioactive properties based on deep reinforcement learning (RL). RL, a subfield of artificial intelligence, is used to solve dynamic decision problems and involves analyzing possible actions, estimating the statistical relationship between actions and their possible outcomes, and determining a treatment regime that seeks to achieve the most desirable outcome. The integration of RL and neural networks was developed in the 1990s, but with the recent advancement of deep learning, which benefits from big data, new and powerful algorithmic approaches are emerging. RL, especially when combined with deep neural networks, is currently experiencing a renaissance and has recently enabled superior human performance. The execution of the method, called ReLeaSE (Reinforcement Learning for Structural Evolution), applies to the problem of designing chemical libraries with desired properties and proves to be a viable solution to this problem. This method is able to predict different properties like logP, Tm, and pIC50. In the field of machine learning, the partition coefficient (logP) can be used as a property of

a chemical compound to predict its toxicity or biological activity. For example, in some reinforcement learning applications, logP can be used as a feature to help the model predict the response of the biological system to a given chemical compound. In this way, the ReLeaSE model can learn to select the most promising molecules for further study or new drug development. For example, the model could be trained to select molecules with a high logP because they have a higher probability of being biologically active. Conversely, the model could be trained to exclude molecules with too high a logP because they might be too toxic. In addition, logP can be used to predict the distribution of the chemical compound in the body and thus to determine its bioavailability. For example, a compound with a high logP might have higher bioavailability because it has a greater tendency to accumulate in lipid membranes and thus be absorbed by the body. Conversely, a compound with a low logP might have lower bioavailability because it has a lower tendency to accumulate in fats. In some cases, the model can be used to predict the melting point (Tm) of a substance from its chemical structure; it can be useful in identifying new molecules that may have desirable properties. Tm (melting point) is the temperature at which a solid changes to a liquid state. In chemistry and medicinal chemistry, the melting point of a substance is often used as an identifying property because each substance has a specific melting point. In other cases, the model can be used to predict the activity of a series of chemical compounds on a particular enzyme, like the pIC50 values of the compounds which can be used as a metric to assess how accurately the model can predict the activity of each compound. The Inhibitory concentration (IC) is a measure of the efficacy of a chemical compound in inhibiting the activity of a particular biological target, such as an enzyme or protein. The pIC50 is a specific type of IC that represents the concentration of a compound that is required to inhibit the activity of the target by 50In the following sections we will look in detail at the use of ReLeaSe.

Related work. *The generative model in references (23, 37) is a "vanilla" RNN without augmented memory stack, which does not have the capacity to count and infer algorithmic patterns (34).*

2 The Method

The ReLeaSE method implements a deep RL approach for de novo design of novel chemical compounds with desired properties. ReLeaSE is distinguished from other similar approaches by its simple representation of molecules through the molecular string input system (SMILES) only during the generation and prediction phases of the method and the integration of these phases into a single workflow that also includes an RL module. The simplified molecular-input line-entry system (SMILES) is a specification in the form of a line notation for describing the structure of chemical species using short ASCII strings [1]. The project makes use of QSPR (Quantitative Structure-Property Relationship) a technique that uses mathematical models to predict the properties of a chemical compound based on its chemical structure. QSPR models are based on the assumption that there is a quantitative relationship between the structure of a chemical compound and its properties. To build a QSPR model, data on the structure and properties of a large number of known chemical compounds (training set) are used, which are then used to train the model. Once the model has been trained, it can be used to predict the properties of unknown chemical compounds (test set). QSPR models are very useful because they make it possible to predict the properties of a chemical compound without having to synthesize or test it experimentally. The training process consists of two stages: in the first stage the generative model is built. Since Regular RNNs such as long short-term memory (LSTM) (31) and gated recurrent unit (GRU) (32) are unable to solve the sequence prediction problems because of their inability to count, for the generative model, a special type of stack-augmented RNN (Stack-RNN) (30) has been used; that has found success in inferring algorithmic patterns. This Stack-RNN defines a new neuron or cell structure on top of the standard gated recurrent unit [2] cell (see Fig. 1 - A). It has two additional multiplicative gates referred to as the memory stack, which allow the Stack-RNN to learn meaningful long-range interdependencies. Stack memory is a differentiable structure onto and from which continuous vectors are inserted and removed. The objective of the Stack-RNN then is to learn hidden rules of forming sequences of letters that correspond to legitimate SMILES strings (SMILES strings as sentences composed of characters used in SMILES notation). The generative model has two modes of processing sequences—training and generating. At each time step, during the training mode, the generative network takes a current prefix of the training object and predicts the probability distribution of the next character (Fig. 1 - A).

Then, the next character is sampled from this predicted probability distribution and is compared to the ground truth. Afterward, on the basis of this comparison, the cross-entropy loss function is calculated, and parameters of the model are updated. At each time step, in generating mode, the generative network takes a prefix of already generated sequences and then, like in the training mode, predicts the probability distribution of the next character and samples it from this predicted

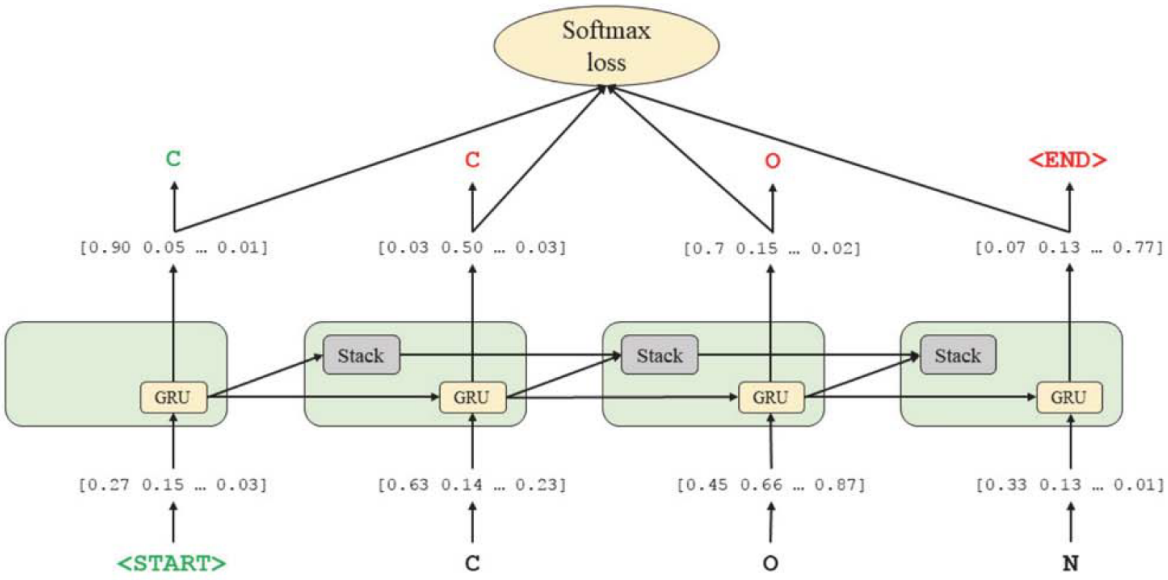


Figure 1: Training step of the generative Stack-RNN

distribution (Fig. 2 - B).

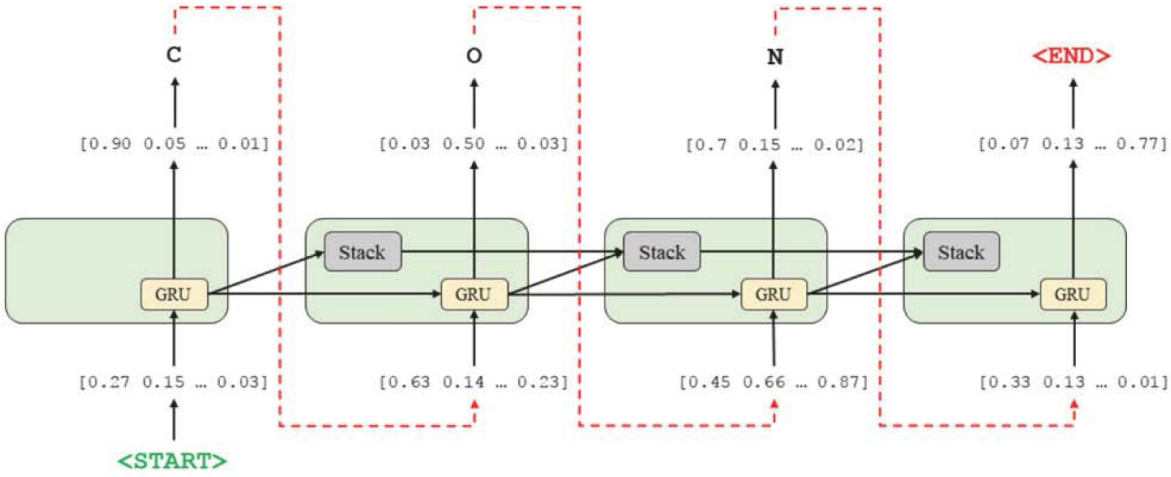


Figure 2: Generator step of the generative Stack-RNN

In the generative model, the model parameters are not updated. At the second stage, both generative and predictive models are combined into one RL system, as shown in figure 3.

In this system, the generative model G (Fig. 3) plays the role of an agent, whose action space is represented by the SMILES notation alphabet, and state space is represented by all possible strings in this alphabet.The predictive model plays the role of a critic estimating the agent's behavior by assigning a numerical reward to every generated molecule (that is, SMILES string). The predictive model P (Fig. 3) is a model for estimating physical, chemical, or biological properties of molecules. This property prediction model is a deep neural network, which consists of an embedding layer, an LSTM layer, and two dense layers. This network is designed to calculate user-specified property (activity) of the molecule taking a SMILES string as an input data vector. This model takes a SMILES string as an input and provides one real number, which is an estimated property value, as an output (Fig. 4). In a practical sense, this learning step is analogous to traditional quantitative structure–activity relationships (QSAR) models. However, unlike conventional QSAR, no numerical descriptors are needed, as the model distinctly learns directly from the SMILES notation as to how to relate the comparison between SMILES strings to that between target properties. The predictor uses, as shown in the figure 4, two unidirectional LSTM layers with hidden size of 128 each, with an embedding size of 128. The dense layer is a Multi-layer Perceptron with hidden size of 128 and ReLU activation function.

The reward (Fig. 3) is a function of the numerical property calculated by the predictive model. At

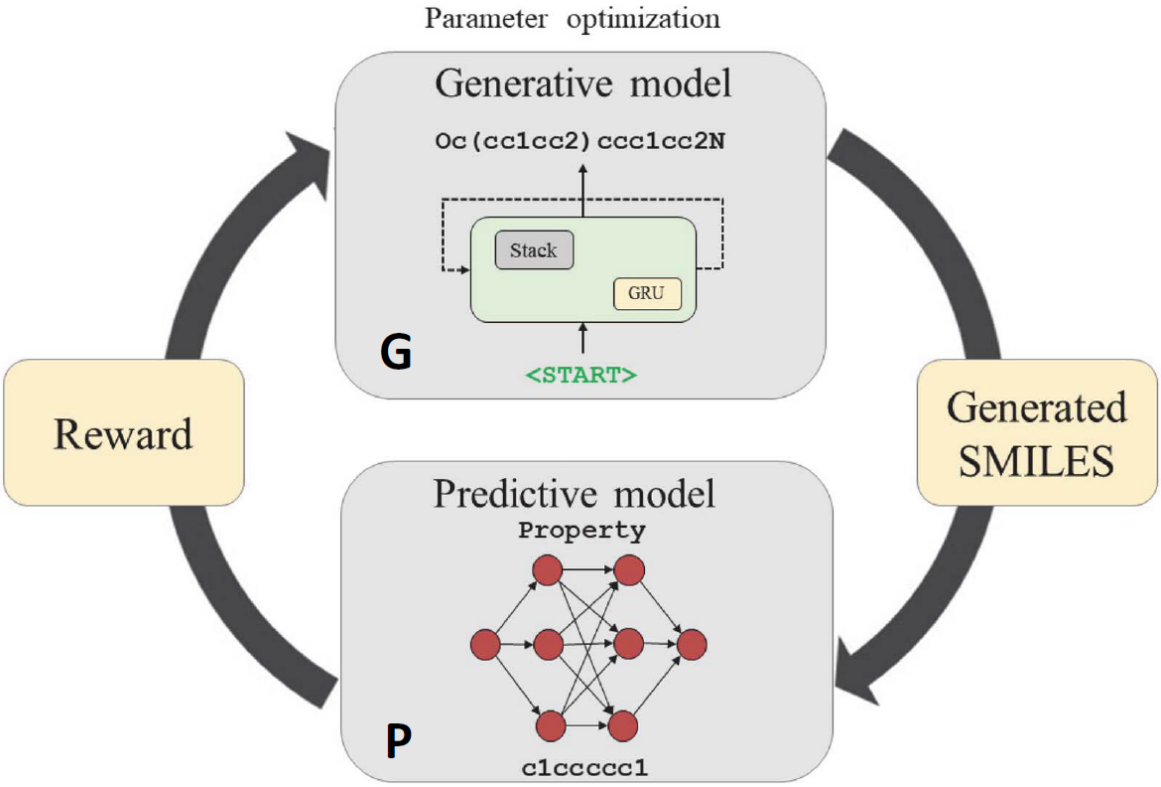


Figure 3: General pipeline of RL system for novel compound generation

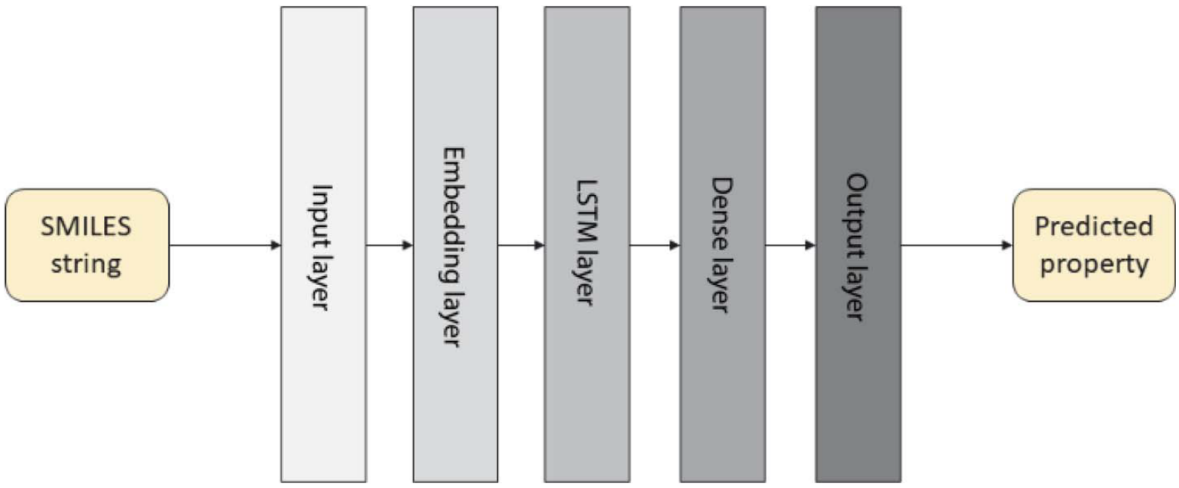


Figure 4: Scheme of predictive model.

this stage, the generative model is trained to maximize the expected reward. The absolute majority of compounds generated de novo by the ReLeaSE method are novel structures as compared to the data sets used to train generative models, and any traditional quantitative structure–activity relationships model could be used to evaluate their properties. As a proof of principle, the method has been tested on three diverse types of endpoints: physical properties, biological activity, and chemical substructure bias. The use of a flexible reward function enables different library optimization strategies where one can minimize,maximize, or impose a desired range to a property of interest in the generated compound libraries.

3 The execution

In this experiment we will optimize parameters of pretrained generative RNN to produce molecules with values of logP within drug-like regions according to Lipinsky rule. We use policy gradient algorithm with custom reward function to bias the properties of generated molecules. The following example uses real SMILES strings, contained in a given input file (we used the ChEMBL database of drug-like compounds). In this execution the generator produces a new SMILES string, which is then evaluated by the predictor. Based on the obtained prediction and the goal, it is assigned a numerical reward value and updated the parameters of the generator using policy gradient algorithm.

The policy gradient loss is defined as follow:

$$L(S|\theta) = -\frac{1}{n}\sum_{i=1}^{|S|}\sum_{j=1}^{length(s_i)} R_i \cdot \gamma^i \cdot \log p(s_i|s_0 \dots s_{i-1}\theta),$$

where R is the reward obtained at time step i. γ is the discount factor and $p(s_i|s_1, \cdots, s_{i-1}, \theta)$ is the probability of the next character given the prefix, which is obtained from the generator. The reward is the same for every time step and is equal to the reward for the whole molecule.

3.1 Prerequisites

- Linux OS or WSL 2 on a Windows 10 (or higher) machine. If you use WSL 2, follow the [official guide].
- A Modern NVIDIA GPU, compatible with [CUDA 11.3].
- Docker and Docker Compose (Application containers engine). Install it from [here].
- The Docker GPU Support enabled on the machine; check it out [here].
- The Nvidia Container Toolkit. Install it from [here].

Note that you do not need to install the CUDA Toolkit on the host system.
Note that you have to install the NVIDIA drivers on your system.

4 Installation

In this section is described how to implement and run the ReLeaSe method on a machine.

4.1 Instructions

As mentioned into the repository of the project, the following instructions need to be performed for running the container:

\$ git clone https://*github.com/IvanBuccella/SF2Bio*
HTTP_PORT=8888
\$ docker−compose build
\$ docker−compose up

4.2 The Dockerfile

During the installation and initialization phases of the project, some changes had to be made to the code to resolve errors that prevented it from starting. This could have been caused by several factors, such as the presence of outdated dependencies or incompatible package versions. These changes allowed the use of newer versions of the packages, which were more stable and compatible with the code. The result of this phase gave in output the construction of the Dockerfile which replicates the installation methodology described in the ReLeaSe method repository.

```
1 FROM nvidia/cuda:11.3.0−cudnn8−runtime−ubuntu20.04

2 RUN apt−get update
3 RUN apt−get install −y git
4 RUN apt−get install −y wget

5 RUN wget https://repo.anaconda.com/miniconda/ Miniconda3−latest−Linux−x86_64.sh
6 RUN bash Miniconda3−latest−Linux−x86_64.sh −b −p /miniconda
7 ENV PATH=\$PATH:/miniconda/pcondabin:/miniconda/bin

8 RUN conda update −n base −c defaults conda
9 RUN conda create −n release python=3.6

10 SHELL ["conda","run","−n","release","/bin/bash", "−c"]

11 COPY environment environment
12 RUN conda install −−yes −−file environment/conda.txt
13 RUN conda install −c rdkit rdkit nox cairo
14 RUN conda install pytorch=1.10.1 torchvision=0.11.2 −c pytorch −c conda−forge
15 RUN pip install −r environment/pip.txt

16 WORKDIR /home

17 RUN git clone https://github.com/isayev/ReLeaSE.git .

18 ENTRYPOINT ["conda", "run", "−n", "release", "jupyter", "notebook", "−−ip=0.0.0.0",
     "−−port=8888", "−−allow−root", "−−NotebookApp.token=''",
     "−−NotebookApp.password=''"]
```

As shown in the code above, build starts from the official Nvidia CUDA docker image with Ubuntu 20.04. To the initial image, several dependencies are added, like MiniConda which is a bootstrap version of Anaconda [Row 5 del codice]. Once Conda is installed, a new environment with Python is created [Row 11 del codice] and the required dependencies specified in the conda.txt file are installed [Row 12 del codice]. In addition, RDKit, Nox, Cairo, pytorch and torchvision are installed [Rows 13-14 del codice]. Then the pip dependencies listed in the pip.txt file, such as tensorboard, tensorflow are installed [Row 15 del codice]. Finally, the ReLeaSe code repository is cloned into the workdir of the image and the entrypoint of the jupyter notebook is defined.

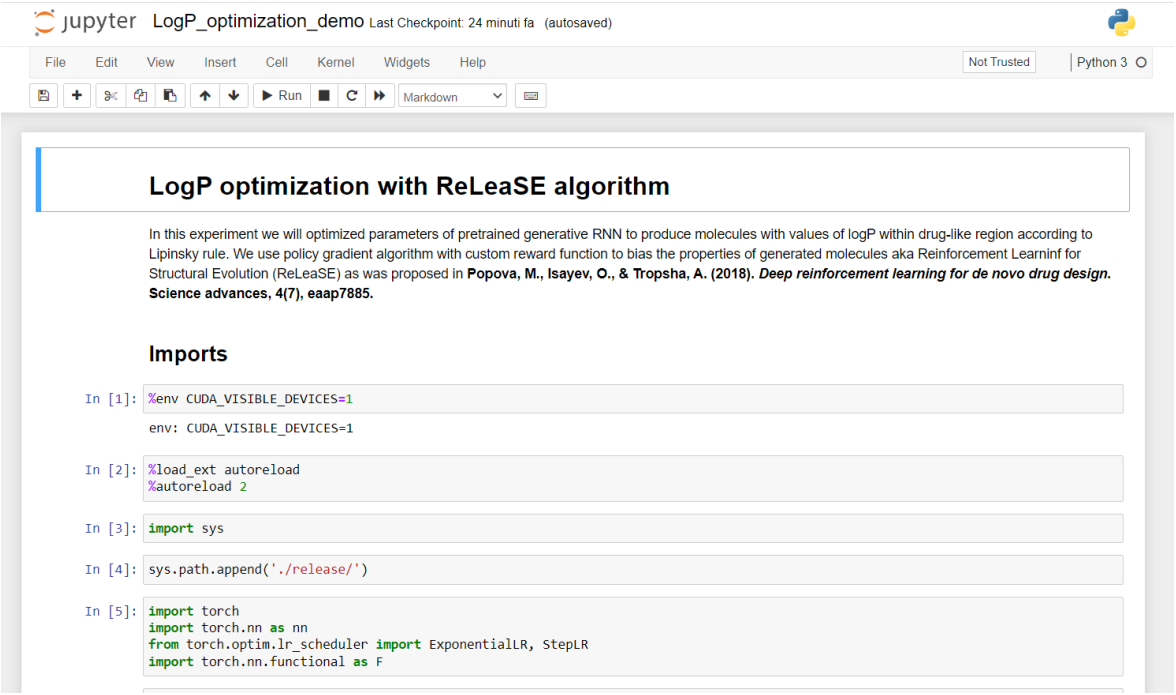
4.3 The Docker Compose

```
version: "3.8"

services:
  app:
    platform: linux/amd64
    build:
      context: .
       dockerfile: Dockerfile
    environment:
      − NVIDIA_DISABLE_REQUIRE=1
```

```
    ports:
      - \${HTTP_PORT}:8888
    deploy:
      resources:
        reservations:
          devices:
            - capabilities: [gpu]
```

As shown in the code above, the docker-compose file includes the GPU access [3] to the service container "app" including all the nvidia GPUs installed on the host machine. This access allows the container to use the nvidia GPUs shared for the execution; in particular this allows "pytorch" to find and to use the CUDA cores.

4.4 The running container



In the figure above is reported the configured jupyter notebook of the execution described in the following sections; it can be reached by visiting the URL into the host machine web browser.

5 Description of the execution

5.1 First stage - Setting up the generator

The real SMILES strings, aka the data, are read from the ChEMBL database; they are included into a .smi file:

```
gen_data_path = './data/chembl_22_clean_1576904_sorted_std_final.smi'
tokens = ['<', '>', '#', '%', ')', '(', '+', '-', '/', '.', '1', '0', '3', '2', '5', '4', '7',
          '6', '9', '8', '=', 'A', '@', 'C', 'B', 'F', 'I', 'H', 'O', 'N', 'P', 'S', '[', ']',
          '\\', 'c', 'e', 'i', 'l', 'o', 'n', 'p', 's', 'r', '\n']
gen_data = GeneratorData(training_data_path=gen_data_path, delimiter='\t',
                         cols_to_read =[0], keep_header=True, tokens=tokens)
```

Then the stack-augmented generative RNN is initialized:

```
hidden_size = 1500
stack_width = 1500
stack_depth = 200
layer_type = 'GRU'
lr = 0.001
optimizer_instance = torch.optim.Adadelta

my_generator = StackAugmentedRNN(input_size=gen_data.n_characters, hidden_size=hidden_size,
                    output_size=gen_data.n_characters, layer_type=layer_type,
                    n_layers=1, is_bidirectional=False, has_stack=True,
                    stack_width=stack_width, stack_depth=stack_depth,
                    use_cuda=use_cuda,
                    optimizer_instance=optimizer_instance, lr=lr)
model_path = './checkpoints/generator/checkpoint_biggest_rnn'
losses = my_generator.fit(gen_data, 1500000)
plt.plot(losses)
my_generator.evaluate(gen_data)
my_generator.save_model(model_path)
my_generator.load_model(model_path)
```

5.2 First stage - Build the Predictive Recurrent Neural Network

The stack-augmented predictive RNN is generated.

The data are read from the following code:

```
from openchem.data.utils import read_smiles_property_file
data = read_smiles_property_file('./data/logP_labels.csv',
                                 cols_to_read=[1, 2], keep_header=False)
smiles = data[0]
labels = data[1].astype('float32')
```

The imported data are prepared:

```
from openchem.data.utils import get_tokens
tokens, _, _ = get_tokens(smiles)
tokens = ['<', '>', '#', '%', ')', '(', '+', '-', '/', '.', '1', '0', '3', '2', '5', '4', '7',
          '6', '9', '8', '=', 'A', '@', 'C', 'B', 'F', 'I', 'H', 'O', 'N', 'P', 'S', '[', ']',
          '\\', 'c', 'e', 'i', 'l', 'o', 'n', 'p', 's', 'r', '\n']
tokens = ''.join(tokens) + ' '

from sklearn.model_selection import KFold, train_test_split

cross_validation_split = KFold(n_splits=5, shuffle=True)

data = list(cross_validation_split.split(smiles, labels))
```

The model parameters are configured as follow:

```
import torch
from openchem.utils.utils import identity
from openchem.modules.embeddings.basic_embedding import Embedding
model_object = Smiles2Label

n_hidden = 128
batch_size = 128
num_epochs = 26
lr = 0.005
```

```
model_params = {
    'use_cuda': True,
    'random_seed': 42,
    'world_size': 1,
    'task': 'regression',
    'data_layer': SmilesDataset,
    'use_clip_grad': False,
    'batch_size': batch_size,
    'num_epochs': num_epochs,
    'logdir': './checkpoints/logP/',
    'print_every': 1,
    'save_every': 5,
    'train_data_layer': None,
    'val_data_layer': None,
    'eval_metrics': r2_score,
    'criterion': nn.MSELoss(),
    'optimizer': Adam,
    'optimizer_params': {
        'lr': lr,
    },
    'lr_scheduler': ExponentialLR,
    'lr_scheduler_params': {
        'gamma': 0.98
    },
    'embedding': Embedding,
    'embedding_params': {
        'num_embeddings': len(tokens),
        'embedding_dim': n_hidden,
        'padding_idx': tokens.index('␣')
    },
    'encoder': RNNEncoder,
    'encoder_params': {
        'input_size': n_hidden,
        'layer': "LSTM",
        'encoder_dim': n_hidden,
        'n_layers': 2,
        'dropout': 0.8,
        'is_bidirectional': False
    },
    'mlp': OpenChemMLP,
    'mlp_params': {
        'input_size': n_hidden,
        'n_layers': 2,
        'hidden_size': [n_hidden, 1],
        'activation': [F.relu, identity],
        'dropout': 0.0
    }
}
```

The training is done from the following code:

```
import os
i = 0
models = []
results = []
for split in data:
    print('Cross␣validation,␣fold␣number␣' + str(i) + '␣in␣progress ...')
    train, test = split
    X_train = smiles[train]
```

```
    y_train = labels[train].reshape(-1)
    X_test = smiles[test]
    y_test = labels[test].reshape(-1)
    save_smiles_property_file(tmp_data_dir + str(i) + '_train.smi',
                              X_train, y_train.reshape(-1, 1))
    save_smiles_property_file(tmp_data_dir + str(i) + '_test.smi',
                              X_test, y_test.reshape(-1, 1))

    train_dataset = SmilesDataset(tmp_data_dir + str(i) + '_train.smi',
                                  delimiter=',', cols_to_read=[0, 1], tokens=tokens,
                                  flip=False,)
    train_dataset.target = train_dataset.target
    test_dataset = SmilesDataset(tmp_data_dir + str(i) + '_test.smi',
                                 delimiter=',', cols_to_read=[0, 1], tokens=tokens,
                                 flip=False)
    test_dataset.target = test_dataset.target
    model_params['train_data_layer'] = train_dataset
    model_params['val_data_layer'] = test_dataset
    model_params['logdir'] = log_dir + 'fold_' + str(i)
    ckpt_dir = model_params['logdir'] + '/checkpoint/'
    try:
        os.stat(ckpt_dir)
    except:
        os.mkdir(model_params['logdir'])
        os.mkdir(ckpt_dir)
    train_loader = create_loader(train_dataset,
                                 batch_size=model_params['batch_size'],
                                 shuffle=True,
                                 num_workers=4,
                                 pin_memory=True,
                                 sampler=None)
    val_loader = create_loader(test_dataset,
                               batch_size=model_params['batch_size'],
                               shuffle=False,
                               num_workers=1,
                               pin_memory=True)
    models.append(model_object(params=model_params).cuda())
    criterion, optimizer, lr_scheduler = build_training(models[i], model_params)
    results.append(fit(models[i], lr_scheduler, train_loader, optimizer, criterion,
        model_params, eval=True, val_loader=val_loader))

    i = i+1
```

The evaluation is done from the following code:

```
import numpy as np
rmse = []
auc_score = []
for i in range(5):
    test_dataset = SmilesDataset(tmp_data_dir + str(i) + '_test.smi',
                                 delimiter=',', cols_to_read=[0, 1], tokens=tokens,
                                 flip=False)
    test_dataset.target = test_dataset.target
    val_loader = create_loader(test_dataset,
                               batch_size= model_params['batch_size'],
                               shuffle=False,
                               num_workers=1,
                               pin_memory=True)
    metrics = evaluate(models[i], val_loader, criterion)
    rmse.append(np.sqrt(metrics[0]))
```

```
    auc_score.append(metrics[1])
```

And then the evaluated data are saved in order to be used in the next steps:

```
! mv ./checkpoints/logP/fold_0/checkpoint/epoch_25 ./checkpoints/logP/fold_0.pkl
! mv ./checkpoints/logP/fold_1/checkpoint/epoch_25 ./checkpoints/logP/fold_1.pkl
! mv ./checkpoints/logP/fold_2/checkpoint/epoch_25 ./checkpoints/logP/fold_2.pkl
! mv ./checkpoints/logP/fold_3/checkpoint/epoch_25 ./checkpoints/logP/fold_3.pkl
! mv ./checkpoints/logP/fold_4/checkpoint/epoch_25 ./checkpoints/logP/fold_4.pkl
```

5.3 First stage - Setting up the predictor

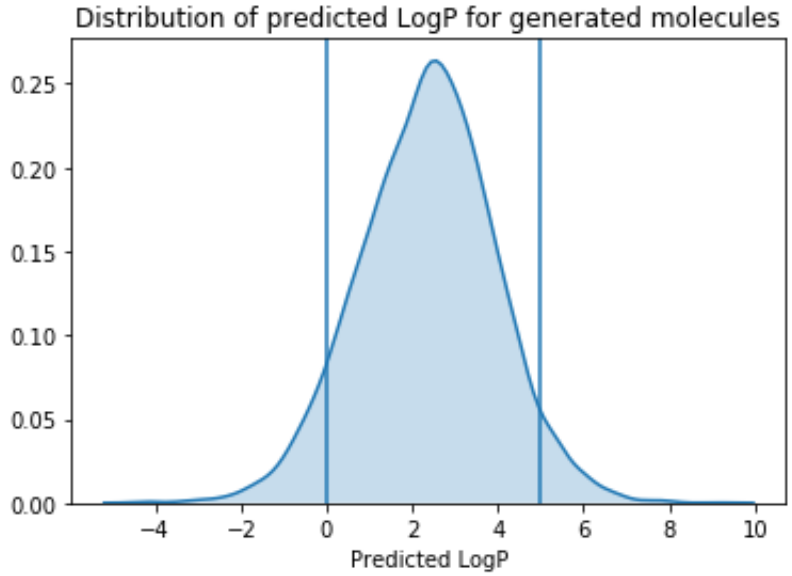
The path of the generated data from the Predictive RNN is configured:

```
path_to_params = './checkpoints/logP/model_parameters.pkl'
path_to_checkpoint = './checkpoints/logP/fold_'
```

The unbiased distribution of the property by using the previous generator and predictor created is produced:

```
my_predictor = RNNPredictor(path_to_params, path_to_checkpoint, predictor_tokens)
smiles_unbiased, prediction_unbiased = estimate_and_update(my_generator,
                    my_predictor,
                    n_to_generate=10000)
```

This phase produces in output:



5.4 Second stage - Biasing the distribution of the generator with reinforcement learning

A copy of the generator is created:

```
my_generator_max = StackAugmentedRNN(input_size=gen_data.n_characters,
                    hidden_size=hidden_size,
                    output_size=gen_data.n_characters,
                    layer_type=layer_type,
                    n_layers=1, is_bidirectional =False, has_stack=True,
```

```
                    stack_width=stack_width, stack_depth=stack_depth,
                    use_cuda=use_cuda,
                    optimizer_instance=optimizer_instance, lr=lr)

my_generator_max.load_model(model_path)
```

The reward function used here is the following:

$$R = \begin{cases} 11.0, & \text{if } 1.0 < \log P < 4.0 \\ 1.0, & \text{otherwise} \end{cases}$$

```
def get_reward_logp(smiles, predictor, invalid_reward=0.0):
    mol, prop, nan_smiles = predictor.predict([smiles])
    if len(nan_smiles) == 1:
        return invalid_reward
    if (prop[0] >= 1.0) and (prop[0] <= 4.0):
        return 11.0
    else:
        return 1.0
```

And then the Reinforcement Learning (combining the generator, the predictor and the reward function depending on the goal) is started:

```
RL_logp = Reinforcement(my_generator_max,
                        my_predictor,
                        get_reward_logp)

rewards = []
rl_losses = []

for i in range(n_iterations):
  for j in trange(n_policy, desc='Policy_gradient...'):
      cur_reward, cur_loss = RL_logp.policy_gradient(gen_data)
      rewards.append(simple_moving_average(rewards, cur_reward))
       rl_losses.append(simple_moving_average(rl_losses, cur_loss))

  plt.plot(rewards)
  plt.xlabel('Training_iteration')
  plt.ylabel('Average_reward')
  plt.show()
  plt.plot( rl_losses )
  plt.xlabel('Training_iteration')
  plt.ylabel('Loss')
  plt.show()

  smiles_cur, prediction_cur = estimate_and_update(
                                          RL_logp.generator,
                                          my_predictor,
                                          n_to_generate)

  print('Sample_trajectories:')
  for sm in smiles_cur[:5]:
      print(sm)
```

In this step is used an important function "estimate_and_update" which:

1. generates n_to_generate number of SMILES strings
2. filters invalid SMILES

3. predicts logP for valid SMILES

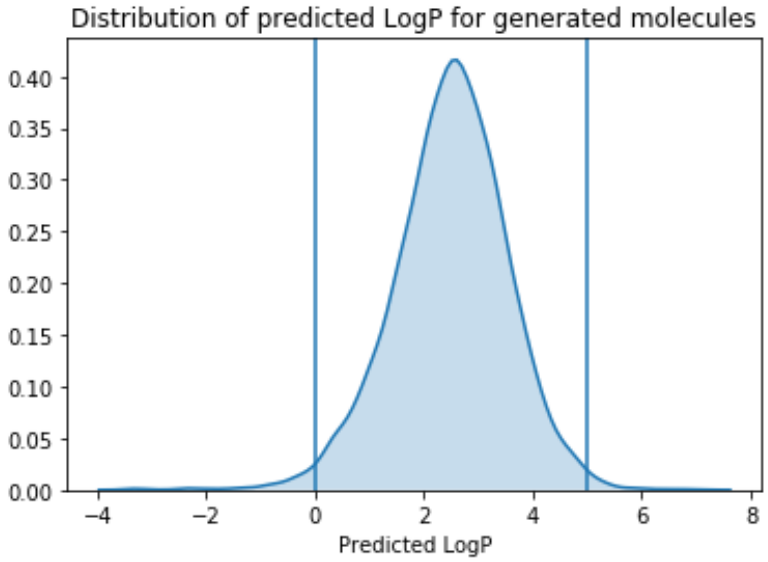
4. Returns valid SMILES and their predicted logPs

```
def estimate_and_update(generator, predictor, n_to_generate):
    generated = []
    pbar = tqdm(range(n_to_generate))
    for i in pbar:
        pbar.set_description("Generating molecules...")
        generated.append(generator.evaluate(gen_data, predict_len=120)[1:-1])

    sanitized = canonical_smiles(generated, sanitize=False, throw_warning=False)[:-1]
    unique_smiles = list(np.unique(sanitized))[1:]
    smiles, prediction, nan_smiles = predictor.predict(unique_smiles, use_tqdm=True)

    plot_hist(prediction, n_to_generate)

    return smiles, prediction
```

After reached the goal, the biased distribution of the property is produced:

```
smiles_biased, prediction_biased = estimate_and_update(RL_logp.generator,
                                                       my_predictor,
                                                       n_to_generate=10000)
```

This phase produces in output:



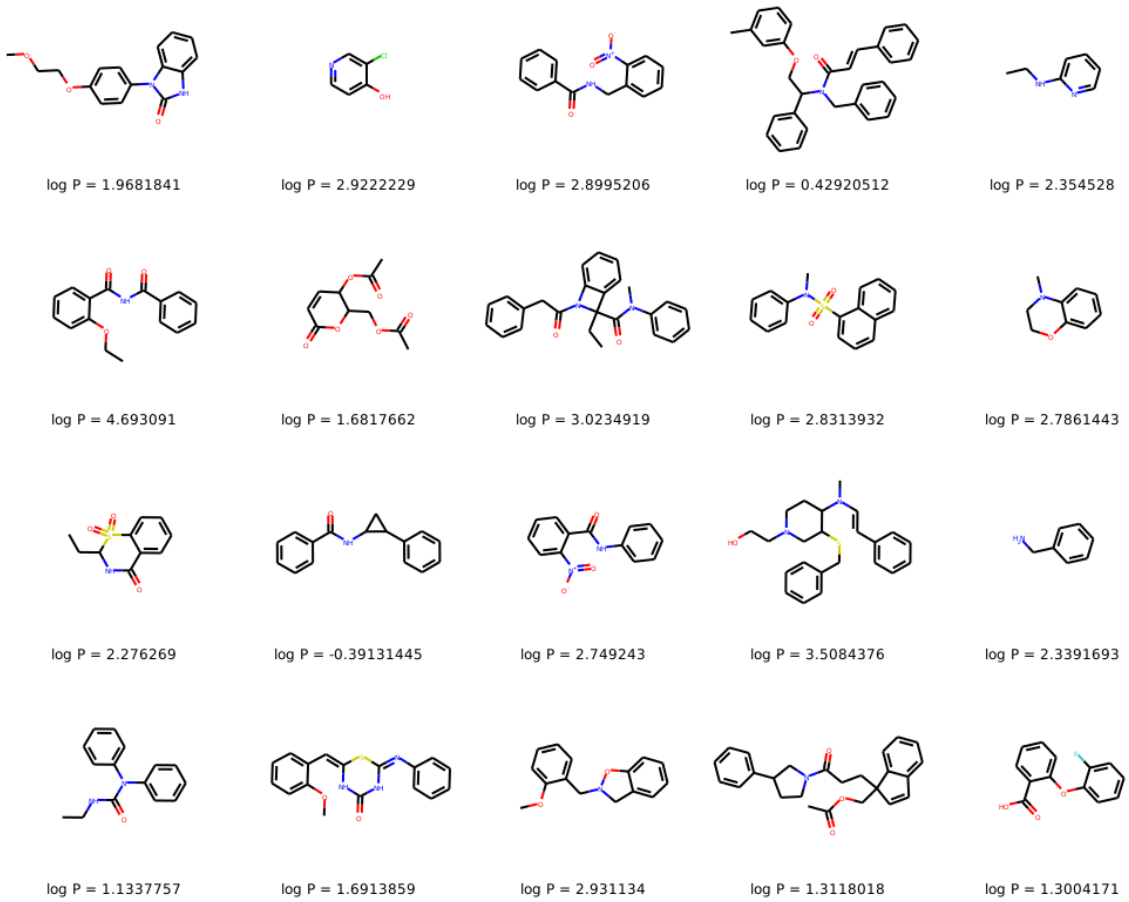
5.5 Output - Drawing random molecules

For drawing some random compounds from the biased library, the following code can be executed:

```
from rdkit.Chem import Draw
from rdkit.Chem.Draw import DrawingOptions
from rdkit.Chem import Draw
DrawingOptions.atomLabelFontSize = 50
DrawingOptions.dotsPerAngstrom = 100
DrawingOptions.bondLineWidth = 3
generated_mols = [Chem.MolFromSmiles(sm, sanitize=True) for sm in smiles_biased]
sanitized_gen_mols = [generated_mols[i] for i in np.where(np.array(generated_mols) != None)[0]]
```

```
n_to_draw = 20
ind = np.random.randint(0, len(sanitized_gen_mols), n_to_draw)
mols_to_draw = [sanitized_gen_mols[i] for i in ind]
legends = ['log P = ' + str(prediction_biased[i]) for i in ind]
Draw.MolsToGridImage(mols_to_draw, molsPerRow=5,
                     subImgSize=(200,200), legends=legends)
```

The output will be something like:



6 Conclusions

References

[1] Simplified molecular-input line-entry system

[2] J. Chung, C. Gulcehre, K. Cho, Y. Bengio, Empirical evaluation of gated recurrent neural networks on sequence modeling, (2014).

[3] Docker GPU support