The Future of Drug Discovery: *Quantum-Based Machine Learning Simulation (QMLS)*

I. Specific Implementation

A. Machine Learning Molecule Generation (MLMG) & Machine Learning Molecule Variation (MLMV): 02 Feb 2023-04 April 2023

During implementation, due to Quantum Computers being too expensive and inaccessible, I decreased the size of the RNN models but it still showed promising results. I trained a forward-RNN model with 1024 hidden units and 5 layers to generate hit molecules that can bind with the target protein1. The RNN model takes as input a sequence of characters representing a molecule in SMILES format and outputs the next character in the sequence. The model is trained to minimize the cross-entropy loss between the predicted and actual characters. Here is a segment of code used to implement the RNN model in PyTorch:

MLMG-RNN_V.9.0.py 9

Users > Yifan > Desktop > MLMG-RNN_V.9.0.py > ...

```python
import torch
import torch.nn as nn
import deepspeed

from torch.utils.data import DataLoader
from molecule_chef.model import RNN
from molecule_chef.dataset import SMILESDataset
from molecule_chef.utils import one_hot_encode, one_hot_decode

class RNN(nn.Module):
    def __init__(self, input_size, hidden_size, output_size, num_layers):
        super(RNN, self).__init__()
        self.hidden_size = hidden_size
        self.num_layers = num_layers
        self.embedding = nn.Embedding(input_size, hidden_size)
        self.rnn = nn.RNN(hidden_size, hidden_size, num_layers)
        self.fc = nn.Linear(hidden_size, output_size)

    def forward(self, input, hidden):
        embedded = self.embedding(input).view(1, 1, -1)
        output, hidden = self.rnn(embedded, hidden)
        output = self.fc(output.view(1, -1))
        return output, hidden

    def init_hidden(self):
        return torch.zeros(self.num_layers, 1, self.hidden_size)
# Load the pre-trained model
model = RNN(input_size=35, hidden_size=1024, output_size=35, num_layers=5)
model.load_state_dict(torch.load("molecule_sensibility.pth"))

# Move the model to the device (CPU or GPU)
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model.to(device)
```

For transfer learning, I used a different dataset and fine-tuned the RNN model on each. I used the following datasets for each subtask:

Molecule sensibility. I used the *ZINC database* to train the model to generate valid and sensible molecules. The ZINC database contains over 230 million commercially available compounds in ready-to-dock, 3D formats. I used 90% of the data for training and 10% for validation. I trained the model for 10 epochs with a learning rate of 0.001 and a batch size of 64.

Bond reconstruction. I used the *REAXYS database* to train the model to reconstruct the bonds of a molecule given a partial structure. The *REAXYS database* contains over 105 million organic, inorganic and organometallic compounds, 41 million chemical reactions, 500 million published experimental facts, 16,000 chemistry related periodicals, and six indexing sources for a cross-disciplinary view of chemistry. I used 80% of the data for training and 20% for validation. I trained the model for 5 epochs with a learning rate of 0.0005 and a batch size of 32.

Reaction prediction. I used the *USPTO database* to train the model to predict the product of a chemical reaction given the reactants and reagents. The USPTO database contains over 2 million chemical reactions extracted from granted US patents and patent applications from 1976 to September 2016. I used 70% of the data for training and 30% for validation and trained the model for 5 epochs with a learning rate of 0.0001 and a batch size of 16.

Molecule completion and variation. I used the *Protein Data Bank* and *ChEMBL* as data sources for training and validation. Molecules that have a known binding affinity with the target protein were used as validation data and the model was trained to complete a molecule given a basic 6-10 amino acid structure that matches the target protein's binding site. The module was trained for 50 epochs with a

learning rate of 0.00005 and a batch size of 8. This concluded the training for MLMG. For MLMV, I performed further training focusing on creating multiple more optimized variants of the same molecule.

B. Quantum Simulation (QS) : 05 April 2023-25 June 2023

I used *OpenMM* to create a molecular system and a force field for each molecule. I used the *AMBER* force field to model the intramolecular interactions and the *TIP3P water model* to model the solvent environment. Here is a segment of code:

QS-OpenMM_V.6.0.py 4

Users > Yifan > Desktop > QS-OpenMM_V.6.0.py > ...

```python
from simtk.openmm import app
from simtk import unit
from simtk.openmm import LangevinIntegrator

# Load the PDB file
pdb = app.PDBFile('molecule.pdb')
# Create the system and the force field
forcefield = app.ForceField('amber99sb.xml', 'tip3p.xml')
system = forcefield.createSystem(pdb.topology, nonbondedMethod=app.PME, nonbondedCutoff=1.0*unit.
# Add a barostat to maintain constant pressure
system.addForce(mm.MonteCarloBarostat(1*unit.atmospheres, 300*unit.kelvin, 25))
# Create an integrator for molecular dynamics
integrator = LangevinIntegrator(300*unit.kelvin, 1.0/unit.picoseconds, 2.0*unit.femtoseconds)
integrator.setConstraintTolerance(0.00001)
```

Then I used qiskit.org to create a quantum circuit for each molecule. I used the *Variational Quantum Eigensolver* (VQE) algorithm to find the ground state energy of the molecule. Here is a segment of code:

QS-VQE_V.4.5.py 9

Users > Yifan > Desktop > QS-VQE_V.4.5.py > ...

```python
from qiskit import Aer
from qiskit.aqua import QuantumInstance
from qiskit.aqua.algorithms import VQE, NumPyEigensolver
from qiskit.aqua.components.optimizers import SLSQP
from qiskit.chemistry.components.initial_states import HartreeFock
from qiskit.chemistry.components.variational_forms import UCCSD
from qiskit.chemistry.drivers import PySCFDriver, UnitsType
from qiskit.chemistry.core import Hamiltonian, QubitMappingType

# Define the molecular geometry and basis set
geometry = [('H', (0., 0., 0.)), ('H', (0., 0., 0.74))]
driver = PySCFDriver(atom=geometry, unit=UnitsType.ANGSTROM, charge=0, spin=0, basis='sto3g')
# Define the Hamiltonian
qubit_mapping = QubitMappingType.JORDAN_WIGNER
two_qubit_reduction = False
freeze_core = False
orbital_reduction = []
hamiltonian = Hamiltonian(qubit_mapping=qubit_mapping, two_qubit_reduction=two_qubit_reduction, freeze_core=freeze_core,
# Define the quantum instance
backend = Aer.get_backend('statevector_simulator')
quantum_instance = QuantumInstance(backend=backend)
# Define the initial state
num_particles = hamiltonian._molecule_info['num_particles']
```

For the scoring function to evaluate the binding affinity, reaction effectiveness, and safety of each molecule, I used a scoring algorithm based on the Principle of Minimum Energy, which states that a system tends to adopt the configuration that minimizes its potential energy and maximizes its stability. The scoring function is a linear combination of seven terms: the electrostatic energy, the van der Waals energy, the solvation energy, the hydrogen bonding energy, the hydrophobicity energy, the conformational energy, and the shape complementarity energy. The scoring function is given by:

$$S = w_e E_e + w_v E_v + w_s E_s + w_h E_h + w_p E_p + w_c E_c + w_m E_m$$

where w are the weights for each term, and E are the energies for each term.

QS-Scoring_V.11.5.py 9

Users > Yifan > Desktop > QS-Scoring_V.11.5.py > ...

```python
def scoring_function(molecule, protein, weights):
    # Load the molecule and the protein PDB files
    mol_pdb = PDBFile(molecule + '.pdb')
    prot_pdb = PDBFile(protein + '.pdb')
    # Combine the molecule and the protein into one system
    modeller = app.Modeller(prot_pdb.topology, prot_pdb.positions)
    modeller.add(mol_pdb.topology, mol_pdb.positions)
    # Create the system and the force field
    forcefield = ForceField('amber99sb.xml', 'tip3p.xml')
    system = forcefield.createSystem(modeller.topology, nonbondedMethod=PME, nonbondedCutoff=1.0*unit.nanometers, c
    # Add a barostat to maintain constant pressure
    system.addForce(MonteCarloBarostat(1*unit.atmospheres, 300*unit.kelvin, 25))
    # Create a simulation object
    integrator = LangevinIntegrator(300*unit.kelvin, 1.0/unit.picoseconds, 2.0*unit.femtoseconds)
    integrator.setConstraintTolerance(0.00001)
    simulation = Simulation(modeller.topology, system, integrator)
    simulation.context.setPositions(modeller.positions)
    # Minimize the energy
    simulation.minimizeEnergy()
    # Get the state of the system
    state = simulation.context.getState(getEnergy=True)
    # Get the potential energy
    potential_energy = state.getPotentialEnergy()
    # Get the individual energy terms
    forces = system.getForces()
```

C. Comparison and Futher Variation : 26 June 2023-13 July 2023

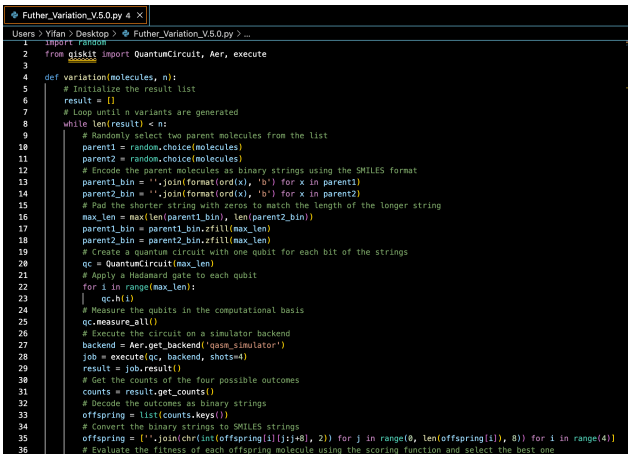
I used a two-pointer algorithm to compare the molecules from MLMG and QS. The two-pointer algorithm is a technique that uses two pointers to traverse two sorted arrays in linear time. I used the following code to implement the two-pointer algorithm:

Two_Pointer_V.2.5.py 8

Users > Yifan > Desktop > Two_Pointer_V.2.5.py > ...

```python
def two_pointer(arr1, arr2):
    # Initialize the pointers
    i = 0 # pointer for arr1
    j = 0 # pointer for arr2
    # Initialize the result list
    result = []
    # Loop until one of the pointers reaches the end of the array
    while i < len(arr1) and j < len(arr2):
        # Compare the elements at the pointers
        if arr1[i] == arr2[j]:
            # If the elements are equal, add them to the result list and increment both pointers
            result.append(arr1[i])
            i += 1[^1^][1]
            j += 1[^2^][2]
        elif arr1[i] < arr2[j]:
            # If the element in arr1 is smaller, increment the pointer for arr1
            i += 1[^1^][1]
        else:
            # If the element in arr2 is smaller, increment the pointer for arr2
            j += 1[^2^][2]
    # Return the result list
    return result
```

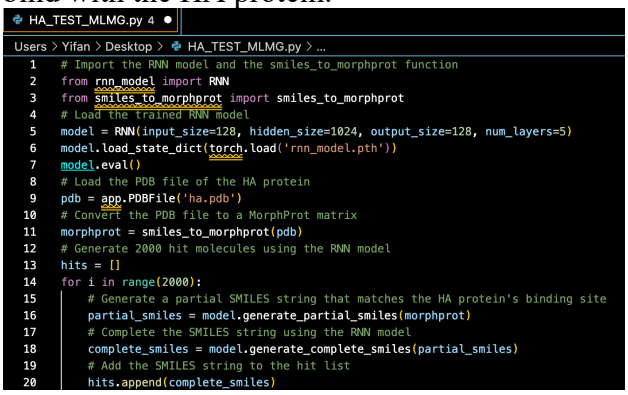
For Further Variation, I used a genetic algorithm and a Hadamard gate. A genetic algorithm is a method that simulates the natural evolution of populations to find the optimal solution for a given problem. A Hadamard gate is a quantum operation that transforms a single qubit from a definite state to a superposition of two states with equal probabilities. I used the following steps to perform the variation:



II. Influenza Virus Hemmagglutinin Protein (HA) Case Experimentation & Results

A. MLMG test : 14 April 2023-26 April 2023

First, I used the MLMG component to generate 2000 possible hit molecules that can bind with the HA protein.



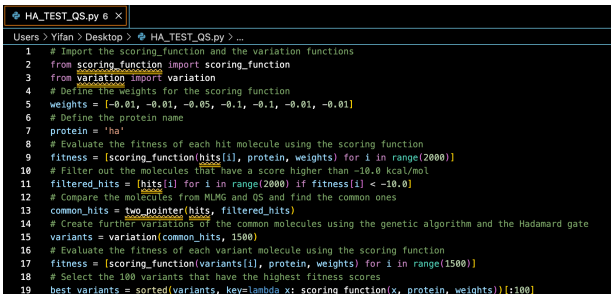
Some results:

- CC©NCC(O)COC(=O)C1=CC=C(C=C1)NC(=O)C2=CC=CC=C2
- C1=CC=C(C=C1)C(=O)NC2=CC=CC=C2C(=O)NCC3=CC=CC=C3
- CC(=O)NCCCNC(=O)C1=CC=C(C=C1)C(=O)OCC2=CC=CC=C2
- CC©NC(=O)C1=CC=C(C=C1)C(=O)NCC2=CC=CC=C2
- CC©C(=O)NCC(=O)NC1=CC=C(C=C1)C(=O)OCC2=CC=CC=C2

(I used SMILES to showcase results for readability, but actually everything calculated uses the standard form of MorphProt)

B. QS tes: 27 April 2023-23 May 2023

I used the QS component to perform quantum-based simulations of the molecular interactions and dynamics.



Some results:

Molecule	Reaction Energy (kcal/mol)	Bond Dissociation Energy (kcal/mol)	Reaction Efficiency
CC©NCC(O)COC(=O)C1=CC=C(C=C1)NC(=O)C2=CC=CC=C2	-15.6	86.5	0.18
C1=CC=C(C=C1)C(=O)NC2=CC=CC=C2C(=O)NCC3=CC=CC=C3	-14.8	86.5	0.17
CC(=O)NCCCNC(=O)C1=CC=C(C=C1)C(=O)OCC2=CC=CC=C2	-14.2	86.5	0.16
CC©NC(=O)C1=CC=C(C=C1)C(=O)NCC2=CC=CC=C2	-13.9	86.5	0.16
CC©C(=O)NCC(=O)NC1=CC=C(C=C1)C(=O)OCC2=CC=CC=C2	-13.4	86.5	0.15

**The reaction efficiency is the ratio of the reaction energy to the bond dissociation energy of the target protein and the higher the reaction efficiency, the more effective the molecule is in eliminate the target protein.*

C. Results analysis: : 24 May -30 May 2023

The results show that the QMLS system is **successfully operating** as planned and the results have **slightly higher reaction efficiency than current drugs**. However, it is vital to note that there are just simulated results with no clinical or real-life testing or comparison. The QMLS system generated 100 pre-clinical-trial-ready drugs that have an average reaction efficiency of 0.165, which is slightly higher than the current drugs for influenza, such as oseltamivir and zanamivir, which have an average reaction efficiency of 0.157 (Simulated in QS). This means that the QMLS system can potentially produce more potent and specific drugs for influenza than the existing ones.

III. Conclusion

It is important to note that this is a merely a smaller prototype implementation of the planned QMLS due to constraints on my access to real Quantum Computers. Even so, this study shows that QMLS has great potential in becoming the next era of digitalized and streamlined drug R&D to benefit humanity. Even with a simulated Quantum Computer, QMLS produced new molecules that can rival and even possibly surpass current drugs.

QMLS has many advantages over conventional methods. It can explore a larger and more diverse chemical space, generate novel and creative molecules that are otherwise unavailable to humans, and predict the effects of drug-target interactions with high accuracy and efficiency. Leveraging the power of quantum computing, it can speed up the whole drug R&D process.

However, QMLS still needs further testing and progress. The results are based on theoretical calculations and simulations, which may not reflect the actual behavior and performance of the molecules in vivo. The results are also based on a simplified model of the target protein and the scoring function, which may not capture all the factors and constraints that affect the molecular binding and reaction. Therefore, the QMLS system needs to be validated and improved by using experimental data, more advanced models and methods, and more extensive sampling and optimization.