Lipophilicity of Peptides and Proteins

Definition

- Lipophilicity refers to the tendency of a compound, such as a peptide or protein, to dissolve in fats, oils, and lipids, as opposed to water. It's a measure of a molecule's affinity for a lipophilic (fat-loving) environment versus a hydrophilic (water-loving) one.

Significance in Peptides and Proteins

- Peptides and proteins are chains of amino acids, and their lipophilicity is determined by the nature and sequence of these amino acids. The side chains of some amino acids are hydrophobic and tend to increase the lipophilicity of the molecule, while others are hydrophilic and decrease it.

What the Lipophilicity Value Signifies

- Balance Between Hydrophobic and Hydrophilic Interactions: The lipophilicity value of a peptide or protein gives insight into how it will interact with other molecules. Proteins with higher lipophilicity tend to interact more with lipid membranes and less with aqueous environments.

- Solubility and Stability: Lipophilicity affects a molecule's solubility, stability, and overall conformation in different environments. This is crucial in understanding how a protein or peptide behaves in biological systems.

- Transport and Bioavailability: For drugs, lipophilicity is a key factor in determining how well they are absorbed, distributed, and reach their target site in the body. Lipophilic drugs tend to cross cell membranes more easily.

Role in Drug Discovery

- Target Interaction: Drugs are often designed to interact with specific proteins. Understanding the lipophilicity of these target proteins helps in designing drugs that have the right balance of lipophilic and hydrophilic properties for optimal interaction.

- Improving Drug Efficacy and Safety: Drugs with appropriate lipophilicity are more likely to reach their target, bind effectively, and exert their therapeutic effect while minimizing off-target effects and toxicity.

- Formulation and Delivery: Knowledge of lipophilicity aids in drug formulation, ensuring that the drug remains stable, soluble, and effective throughout its shelf life and inside the body.

- Predicting Drug Behavior: Lipophilicity is used in pharmacokinetic modeling to predict how a drug will be absorbed, distributed, metabolized, and excreted - crucial for understanding its overall behavior in the body.

Conclusion

- Integral in Rational Drug Design: Lipophilicity is a crucial parameter in the rational design of peptides and proteins as therapeutic agents. It influences how well a drug can reach its target, its interactions with the target, and its overall pharmacokinetic and pharmacodynamic profiles.

- Balancing Act: In drug discovery, achieving the right balance of lipophilicity is key to developing effective and safe drugs. Too lipophilic, and the drug may be poorly soluble and have high toxicity; too hydrophilic, and it may fail to cross cell membranes to reach its target.

---------------------------------------------------

What we are trying to achieve

Applying PyTorch for modeling and then using the Intel OpenVINO toolkit for inference is a feasible approach for a wide range of machine learning tasks, including those related to bioinformatics and molecular biology. However, the process of modeling lipophilicity of proteins and using OpenVINO for inference involves several steps:

1. Data Preparation and Feature Extraction: You need a dataset that represents proteins and their lipophilic properties. This could include molecular descriptors, 3D structures, and known lipophilicity measurements. Feature extraction in the context of proteins might involve calculating various physicochemical properties or using techniques like molecular fingerprints.

2. Model Development in PyTorch: With the prepared dataset, you can develop a machine learning model using PyTorch. This might involve a regression model if you're predicting a continuous measure of lipophilicity, or a classification model for categorizing proteins based on their lipophilicity. Deep learning approaches, especially those leveraging convolutional neural networks (CNNs), can be effective if you are working with 3D structural data of proteins.

3. Training and Validation: Train your model on a portion of your data and validate its performance using a separate validation set. It’s crucial to ensure that the model generalizes well and isn't just memorizing the training data.

4. Model Conversion for OpenVINO: Once you have a trained PyTorch model, you'll need to convert it into a format compatible with OpenVINO. OpenVINO typically uses the ONNX (Open Neural Network Exchange) format or its own Intermediate Representation (IR) format. You can convert the PyTorch model to ONNX and then use OpenVINO's Model Optimizer to convert it to IR.

5. Inference with OpenVINO: With the model converted to OpenVINO’s format, you can now run inference efficiently, especially on Intel hardware. OpenVINO is designed to optimize performance by utilizing Intel CPUs, GPUs, and other hardware accelerators.

6. Analysis and Interpretation: Finally, analyze the output from the inference process. In the context of lipophilicity, this might involve interpreting how the model's predictions correlate with known lipophilic properties and understanding the biological significance of these predictions.

----------------------------------Notebook------------------

%pip install -q "openvino>=2023.1.0"

By running this command in a Jupyter notebook, you instruct the notebook environment to install the specified version (or newer) of the OpenVINO toolkit, while keeping the output minimal. It's a common way to set up the necessary software environment within a notebook before running code that depends on those packages.

pip install rdkit-pypi

By running **pip install rdkit-pypi**, you are installing RDKit into your Python environment, making it available for use in cheminformatics tasks like molecule manipulation, chemical file reading and writing, calculating molecular descriptors, molecular similarity, etc. This is particularly useful for scientists and developers working in the field of computational chemistry, drug discovery, and materials science.

from rdkit import Chem

from rdkit.Chem import AllChem

from rdkit.Chem import PandasTools

from rdkit import RDConfig

import torch

import torch.nn as nn

import torch.optim as optim

import numpy as np

from sklearn.model\_selection import train\_test\_split

from sklearn.metrics import mean\_squared\_error

import os

import pandas as pd

The code you've provided includes import statements from various Python libraries, each serving different purposes in a programming context, especially in the fields of cheminformatics and machine learning. Here's an explanation of each import:

1. `from rdkit import Chem`:

- Imports the `Chem` module from the RDKit library. RDKit is a widely used toolkit for cheminformatics. The `Chem` module provides classes and functions for handling and manipulating chemical structures.

2. `from rdkit.Chem import AllChem`:

- Imports the `AllChem` module from RDKit's `Chem` package. `AllChem` provides a wide range of functions for chemical informatics, including molecule conversion, substructure searching, and molecular descriptor calculation.

3. `from rdkit.Chem import PandasTools`:

- Imports the `PandasTools` module from RDKit, which provides functions to integrate RDKit with pandas DataFrames. It is useful for managing chemical data within pandas, including rendering molecule structures within the DataFrame.

4. `from rdkit import RDConfig`:

- Imports the `RDConfig` module from RDKit. `RDConfig` contains configuration variables for RDKit, such as directory paths to data files and environmental settings.

5. `import torch`:

- Imports PyTorch, a popular deep learning library. PyTorch provides a flexible platform for building and training neural networks, with strong GPU acceleration support.

6. `import torch.nn as nn`:

- Imports the `nn` module from PyTorch as `nn`. This module provides the building blocks for creating neural networks, like layers, activation functions, and loss functions.

7. `import torch.optim as optim`:

- Imports the `optim` module from PyTorch as `optim`. This module includes optimization algorithms like SGD, Adam, etc., used for training neural networks.

8. `import numpy as np`:

- Imports the NumPy library as `np`. NumPy is fundamental for scientific computing in Python, offering powerful data structures for efficient computation with arrays and matrices.

9. `from sklearn.model\_selection import train\_test\_split`:

- Imports the `train\_test\_split` function from scikit-learn's `model\_selection` module. This function is used to easily split datasets into training and test sets.

10. `from sklearn.metrics import mean\_squared\_error`:

- Imports the `mean\_squared\_error` function from scikit-learn's `metrics` module. This function is used to calculate the mean squared error (MSE) between actual and predicted values, a common metric for regression models.

11. `import os`:

- Imports Python's built-in `os` module, which provides functions for interacting with the operating system, like file path manipulation, directory management, and environment variable access.

12. `import pandas as pd`:

- Imports the pandas library as `pd`. Pandas is an essential data analysis and manipulation library, offering powerful data structures like DataFrames for handling tabular data.

Together, these imports suggest that the code is likely involved in a cheminformatics project, possibly involving machine learning or deep learning for chemical data analysis or predictive modeling. The use of RDKit indicates chemical data manipulation, while PyTorch and scikit-learn are indicative of machine learning model development and evaluation.

# Replace 'path\_to\_file.tsv' with the actual file path

file\_path = 'logd74.tsv'

# Load the dataset

df = pd.read\_csv(file\_path, delimiter='\t')

The code snippet you've provided is performing the following tasks:

1. Defining a File Path:

- `file\_path = 'logd74.tsv'`: This line sets the variable `file\_path` to the string `'logd74.tsv'`. This string is assumed to be the name of a file that contains the data you want to work with. The comment above the line suggests that `'logd74.tsv'` should be replaced with the actual path to the file you intend to load.

2. Loading a Dataset with pandas:

- `df = pd.read\_csv(file\_path, delimiter='\t')`: This line uses the pandas library (imported as `pd` in the previous imports you mentioned) to load a dataset from the file located at `file\_path`.

- `pd.read\_csv()`: This function is a common way to read data into a pandas DataFrame from a CSV (Comma-Separated Values) file. Despite its name, it can read files with various delimiters, not just commas.

- `delimiter='\t'`: This parameter specifies that the delimiter in the file is a tab character (`\t`). This is typical of TSV (Tab-Separated Values) files, which is consistent with the file extension `.tsv` in `logd74.tsv`.

- The resulting DataFrame, `df`, will contain the data from `logd74.tsv`, with each row corresponding to a line in the file, and columns determined based on the tab delimiter.

In summary, this code is intended to read a dataset from a TSV file named `logd74.tsv` into a pandas DataFrame for further analysis or processing. The dataset is expected to be in a tab-separated format.

def smiles\_to\_fp(smiles, n\_bits=2048):

    mol = Chem.MolFromSmiles(smiles)

    fp = AllChem.GetMorganFingerprintAsBitVect(mol, radius=2, nBits=n\_bits)

    return np.array(fp)

df['fp'] = df['SMILES'].apply(smiles\_to\_fp)

The provided code defines a function `smiles\_to\_fp` and then applies this function to a column in a pandas DataFrame (`df`). Here's a breakdown of what each part of the code is doing:

### Function Definition: `smiles\_to\_fp`

1. Purpose:

- This function converts a SMILES (Simplified Molecular Input Line Entry System) string to a fingerprint representation of the molecule. SMILES is a notation that encodes the structure of a chemical compound.

2. Parameters:

- `smiles`: A string representing a molecule in SMILES notation.

- `n\_bits=2048`: The number of bits in the fingerprint. 2048 is set as the default value.

3. Function Process:

- `mol = Chem.MolFromSmiles(smiles)`: Converts the SMILES string into an RDKit molecule object. This is necessary because further chemical computations require a molecule object, not just a string.

- `fp = AllChem.GetMorganFingerprintAsBitVect(mol, radius=2, nBits=n\_bits)`: Generates a Morgan fingerprint for the molecule. Morgan fingerprints are a type of circular fingerprint used in cheminformatics for representing the presence or absence of particular substructures in the molecule.

- `radius=2` specifies the radius of the circular neighborhoods considered around each atom when generating the fingerprint.

- `nBits=n\_bits` sets the length of the bit vector (fingerprint). A longer bit vector can capture more information but takes up more memory.

- `return np.array(fp)`: Converts the fingerprint, which is a bit vector, into a NumPy array and returns it. This is often done for compatibility with machine learning libraries, which typically work with NumPy arrays.

### Applying the Function to a DataFrame

- `df['fp'] = df['SMILES'].apply(smiles\_to\_fp)`:

- This line applies the `smiles\_to\_fp` function to each element in the 'SMILES' column of the DataFrame `df`.

- The result of the function (a NumPy array representing the fingerprint) for each SMILES string is stored in a new column `'fp'` in the DataFrame.

- Essentially, this transforms each SMILES representation in the DataFrame into its corresponding fingerprint representation and stores these fingerprints in the DataFrame for further analysis, which could include machine learning tasks.

In summary, this code is used to process chemical data: it takes molecular structures represented as SMILES strings, converts them into a numerical fingerprint format suitable for computational analysis, and stores these fingerprints in the DataFrame for future use. This is a common preprocessing step in cheminformatics, particularly when preparing data for machine learning models.

X = np.array(list(df['fp']))

y = df['logD7.4'].values

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

The code snippet you provided is performing data preparation steps typically seen in the context of machine learning, particularly for setting up a supervised learning task. Here's a breakdown of what each line is doing:

### Preparing the Feature Matrix (X) and Target Vector (y)

1. `X = np.array(list(df['fp']))`:

- This line creates the feature matrix `X`.

- `df['fp']` accesses the 'fp' (fingerprint) column in the pandas DataFrame `df`. This column contains the molecular fingerprints, likely created from SMILES strings as per your previous code snippet.

- `list(df['fp'])` converts the pandas Series into a list where each element is a fingerprint (which itself is a NumPy array).

- `np.array(...)` converts this list of arrays into a 2D NumPy array. Each row in `X` corresponds to a molecule, represented by its fingerprint.

2. `y = df['logD7.4'].values`:

- This line creates the target vector `y`.

- `df['logD7.4']` accesses the 'logD7.4' column in the DataFrame, which presumably contains the target variable for your machine learning model (e.g., a physicochemical property of the molecules).

- `.values` converts the pandas Series into a NumPy array.

### Splitting the Dataset into Training and Test Sets

3. `X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)`:

- This line uses the `train\_test\_split` function from scikit-learn to split the dataset into training and test sets.

- `X` and `y` are the input feature matrix and target vector, respectively.

- `test\_size=0.2` specifies that 20% of the data should be used as the test set, with the remaining 80% as the training set.

- `random\_state=42` sets the seed for the random number generator used in splitting the dataset. This is done for reproducibility, ensuring that the same split is generated each time the code is run.

- The function returns four arrays: `X\_train` (training features), `X\_test` (test features), `y\_train` (training targets), and `y\_test` (test targets).

In summary, this code is preparing a dataset for a supervised learning task. It organizes the features (`X`) and the target (`y`) and splits them into training and test sets, ensuring that the data is ready for training and evaluating a machine learning model. This kind of data preparation is crucial for building reliable and robust predictive models.

class Net(nn.Module):

    def \_\_init\_\_(self):

        super(Net, self).\_\_init\_\_()

        self.fc1 = nn.Linear(2048, 1024)

        self.fc2 = nn.Linear(1024, 512)

        self.fc3 = nn.Linear(512, 1)

    def forward(self, x):

        x = torch.relu(self.fc1(x))

        x = torch.relu(self.fc2(x))

        return self.fc3(x)

model = Net()

The code you've provided defines a neural network model using PyTorch, a popular deep learning library in Python. Here's an explanation of what each part of the code is doing:

### Defining the `Net` Class

1. Class Declaration:

- `class Net(nn.Module)`: This line declares a new class `Net`, which is a subclass of `nn.Module`. In PyTorch, `nn.Module` is the base class for all neural network modules, and your custom models should also subclass it.

2. Constructor (`\_\_init\_\_` method):

- `super(Net, self).\_\_init\_\_()`: This line initializes the parent class (`nn.Module`). It's necessary for proper initialization of various PyTorch internals.

- `self.fc1 = nn.Linear(2048, 1024)`: Defines the first fully connected (linear) layer of the network with 2048 input features and 1024 output features.

- `self.fc2 = nn.Linear(1024, 512)`: Defines the second fully connected layer with 1024 input features (matching the output of the previous layer) and 512 output features.

- `self.fc3 = nn.Linear(512, 1)`: Defines the third and final fully connected layer with 512 input features and 1 output feature. This output size suggests that the network is designed for a regression task (predicting a single continuous value).

3. Forward Method:

- `def forward(self, x)`: This method defines how the input data `x` flows through the network (i.e., the forward pass).

- `x = torch.relu(self.fc1(x))`: Applies the first linear layer (`fc1`) to the input `x`, and then applies the ReLU activation function. ReLU (Rectified Linear Unit) is a common activation function used in neural networks, particularly in hidden layers.

- `x = torch.relu(self.fc2(x))`: Applies the second linear layer (`fc2`) followed by another ReLU activation.

- `return self.fc3(x)`: Finally, applies the third linear layer (`fc3`). There is no activation function applied here, which is typical for the output layer in regression tasks.

### Instantiating the Model

- `model = Net()`: This line creates an instance of the `Net` class, effectively creating the neural network model. This instance, `model`, can then be used for training and making predictions.

### Summary

- The `Net` class defines a simple feedforward neural network with three fully connected layers. The network appears to be designed for a regression task, as indicated by the single unit in the output layer and the lack of an activation function at the output.

- The use of ReLU activation functions in the hidden layers is a standard choice for introducing non-linearity into the model, allowing it to learn more complex patterns in the data.

- This network can be trained using a dataset (like the one prepared in your previous code snippets) to perform tasks such as predicting a property of a molecule based on its fingerprint representation.

optimizer = optim.Adam(model.parameters(), lr=0.001)

criterion = nn.MSELoss()

X\_train\_tensor = torch.tensor(X\_train, dtype=torch.float32)

y\_train\_tensor = torch.tensor(y\_train, dtype=torch.float32)

epochs = 100

for epoch in range(epochs):

    optimizer.zero\_grad()

    outputs = model(X\_train\_tensor)

    loss = criterion(outputs.squeeze(), y\_train\_tensor)

    loss.backward()

    optimizer.step()

    print(f'Epoch {epoch+1}/{epochs}, Loss: {loss.item()}')

The code you've provided is setting up and running the training process for a neural network using PyTorch. Here's a step-by-step explanation of what each part does:

### Setting Up the Optimizer and Loss Function

1. Optimizer:

- `optimizer = optim.Adam(model.parameters(), lr=0.001)`: This line creates an optimizer object, which is used to update the weights of the network during training. Specifically, it uses the Adam optimization algorithm (a popular choice due to its efficiency in handling sparse gradients and adaptive learning rates) with a learning rate (`lr`) of 0.001. The `model.parameters()` function call provides the optimizer with the parameters (weights and biases) of the `model` that it will update.

2. Loss Function:

- `criterion = nn.MSELoss()`: This line sets up the loss function to be used during training. Here, Mean Squared Error (MSE) loss is used, which is a common choice for regression tasks. The MSE loss calculates the average squared difference between the predicted values and the actual values, providing a measure of model accuracy.

### Preparing the Data for Training

3. Data Conversion to Tensors:

- `X\_train\_tensor = torch.tensor(X\_train, dtype=torch.float32)`: Converts the training features (`X\_train`) into a PyTorch tensor. This is necessary because PyTorch models expect input data in the form of tensors.

- `y\_train\_tensor = torch.tensor(y\_train, dtype=torch.float32)`: Similarly, converts the training labels/targets (`y\_train`) into a PyTorch tensor.

### Training Loop

4. Training Loop:

- The loop `for epoch in range(epochs):` iterates over the number of epochs specified (100 in this case). An epoch is a complete pass over the entire training dataset.

- Inside the loop:

- `optimizer.zero\_grad()`: Clears old gradients from the last step (otherwise, gradients would accumulate from other epochs).

- `outputs = model(X\_train\_tensor)`: Passes the training data through the model to get the predicted outputs.

- `loss = criterion(outputs.squeeze(), y\_train\_tensor)`: Computes the loss by comparing the model's predictions (`outputs`) with the actual values (`y\_train\_tensor`). The `squeeze()` method is used to remove any extra dimensions (of size 1) from the output tensor to match the shape of `y\_train\_tensor`.

- `loss.backward()`: Performs backpropagation, calculating the gradients of the loss function with respect to the model parameters.

- `optimizer.step()`: Updates the model's parameters based on the calculated gradients.

- `print(...)`: Prints the epoch number and the current loss, providing a view of the training progress.

### Summary

- This code trains the neural network `model` on the training data (`X\_train\_tensor`, `y\_train\_tensor`) for 100 epochs using the Adam optimizer and MSE loss.

- During each epoch, it calculates the loss, performs backpropagation to compute gradients, and updates the model's weights.

- The print statement at the end of each epoch helps to monitor the training process by showing the loss after each epoch, which is typically expected to decrease as training progresses.

torch.save(model.state\_dict(), 'lipophilicity\_model.pth')

The code `torch.save(model.state\_dict(), 'lipophilicity\_model.pth')` is used to save the trained state of a PyTorch model to a file. Here's a breakdown of what this specific line is doing:

1. `torch.save(...)`:

- This is a function provided by PyTorch for serializing and saving Python objects. In the context of PyTorch models, it's commonly used to save the trained model weights.

2. `model.state\_dict()`:

- `model` is your PyTorch neural network model (likely the one you've trained in the previous steps).

- The `state\_dict()` is a Python dictionary object that maps each layer to its parameter tensor. In simple terms, the state dictionary contains all the weights and biases of the model.

- Saving the `state\_dict` is a lightweight and efficient way to store a model, as it only includes the parameters of the model, not the entire model. This is beneficial because it allows for more flexibility; for instance, you can load these parameters into a model with a different architecture, as long as the parameter shapes match.

3. Saving the Model to a File:

- `'lipophilicity\_model.pth'`: This is the filename where the model's state dictionary will be saved. The `.pth` extension is a convention for PyTorch model files, but it's essentially just a file format for saving Python objects.

- The file will be saved in the current working directory of your Python script or Jupyter notebook, unless you specify a different path.

### Summary

- This line of code is saving the trained model's weights and biases to a file named `'lipophilicity\_model.pth'`.

- This saved model can later be loaded to make predictions or to continue training without having to start the training process from scratch. This is particularly useful for large models or models that take a long time to train.

import torch

import openvino.runtime as ov

from pathlib import Path

# Load your PyTorch model

model = Net()  # Assuming 'Net' is your model class

model.load\_state\_dict(torch.load('lipophilicity\_model.pth'))

model.eval()

The provided code is performing the following actions, primarily focused on loading a trained PyTorch model:

### 1. Importing Required Libraries

- `import torch`:

- This imports the PyTorch library, which is a popular deep learning framework. PyTorch is used for building and training neural networks, among other tasks.

- `import openvino.runtime as ov`:

- This imports the OpenVINO™ toolkit's runtime, which is used for optimizing and deploying deep learning models, particularly for inference. OpenVINO™ is designed to accelerate model inference, especially on Intel hardware.

- `from pathlib import Path`:

- This imports the `Path` class from the `pathlib` module, a Python standard library module used for handling filesystem paths in an object-oriented way.

### 2. Loading a Pre-trained PyTorch Model

- `model = Net()`:

- This line creates an instance of the `Net` class. `Net` is assumed to be a class defining a neural network model, as you might have defined in a previous part of your code. The `Net()` constructor call initializes a new instance of this model.

- `model.load\_state\_dict(torch.load('lipophilicity\_model.pth'))`:

- `torch.load('lipophilicity\_model.pth')`: This function loads the saved model state (weights and biases) from the file `'lipophilicity\_model.pth'`. This file should contain the model's state dictionary, as saved previously using `torch.save`.

- `model.load\_state\_dict(...)`: This method loads the state dictionary into the `model`. This effectively restores the model to the state it was in when the state dictionary was saved, allowing you to use the model with the trained parameters.

- `model.eval()`:

- This puts the model in evaluation mode. In evaluation mode, certain operations behave differently than in training mode, particularly those related to dropout and batch normalization layers. For example, dropout layers will not drop activations, and batch normalization layers will use running statistics instead of batch statistics. This is essential for consistent performance of the model during inference.

### Summary

The code is setting up a PyTorch neural network model (presumably named `Net`) by loading its pre-trained weights and biases from a file, and then switching the model to evaluation mode. This is a typical procedure for preparing a trained model for inference tasks, such as making predictions on new data.

pip install onnx

The command `pip install onnx` is used to install the ONNX (Open Neural Network Exchange) library in a Python environment. Here's a breakdown of what this command does:

1. `pip`:

- `pip` is the package installer for Python. It's used to install and manage Python packages from the Python Package Index (PyPI) and other package repositories.

2. `install`:

- This is a command used with `pip` to indicate that the following term is a package to be installed.

3. `onnx`:

- `onnx` stands for Open Neural Network Exchange, which is an open format used to represent deep learning models.

- ONNX provides an interoperable format for AI models. It allows models trained in one framework (like PyTorch or TensorFlow) to be exported and used in another (like Caffe2, Microsoft Cognitive Toolkit, or OpenVINO).

- This is particularly useful for model deployment and inference across various platforms and devices, and for utilizing optimizations for specific hardware.

### Summary

By running `pip install onnx`, you are installing the ONNX library into your Python environment. This enables you to work with ONNX models, such as converting models from popular deep learning frameworks to the ONNX format, or loading and running ONNX models for inference purposes. This is especially relevant in machine learning and deep learning workflows where model portability and cross-platform compatibility are important.

import torch.onnx

dummy\_input = torch.randn(1, 2048)  # Adjust the size according to your model input

torch.onnx.export(model, dummy\_input, "model.onnx", opset\_version=11)

The provided code is using PyTorch to export a trained neural network model to the ONNX (Open Neural Network Exchange) format. Here's a breakdown of each step:

### Importing `torch.onnx`

- `import torch.onnx`:

- This line imports the `onnx` module from PyTorch, which provides necessary functions to export PyTorch models to the ONNX format. ONNX is a popular open-format used to represent deep learning models and allows for model interchange between various deep learning frameworks.

### Preparing a Dummy Input

- `dummy\_input = torch.randn(1, 2048)`:

- This creates a dummy input tensor using PyTorch's `randn` function, which generates a tensor with random numbers drawn from a standard normal distribution.

- The size of the tensor `(1, 2048)` should match the input size that the model expects. In this case, it seems to be a single input (hence `1`) with 2048 features. This dummy input is used during the export process to trace the model's operations.

### Exporting the Model to ONNX Format

- `torch.onnx.export(model, dummy\_input, "model.onnx", opset\_version=11)`:

- `torch.onnx.export(...)` is the function used to export the PyTorch model to ONNX format.

- `model` is the PyTorch model that you want to export.

- `dummy\_input` is the input tensor that will be passed through the model. This is necessary because ONNX needs to trace the operations performed by the model.

- `"model.onnx"` is the filename where the ONNX model will be saved. The `.onnx` file extension is standard for models in this format.

- `opset\_version=11` specifies the version of the ONNX operator set to use. Different versions of the operator set might have different capabilities. It's important to choose a version supported by the frameworks and tools you plan to use with the ONNX model.

### Summary

This code is exporting a PyTorch neural network model to the ONNX format, enabling the model to be used in different deep learning frameworks that support ONNX. This is useful for deployment or for running inference in environments where PyTorch is not the preferred framework. The use of a dummy input allows ONNX to understand the operations and layer connections within the model.

# Create OpenVINO Core object instance

core = ov.Core()

# Read the ONNX model

ov\_model = core.read\_model("model.onnx")

# (Optional) Perform any necessary optimizations

compiled\_model = core.compile\_model(ov\_model, "CPU")

The provided code is using the OpenVINO toolkit, a library developed by Intel for optimizing and deploying deep learning models, especially for inference. Here's a breakdown of each step in the code:

### 1. Creating an OpenVINO Core Object Instance

- `core = ov.Core()`:

- This line creates an instance of the OpenVINO Core object. The Core object is a central entity in the OpenVINO runtime that allows you to work with models and perform various operations, like reading, loading, and compiling models for inference.

### 2. Reading the ONNX Model

- `ov\_model = core.read\_model("model.onnx")`:

- Here, the `read\_model` method of the Core object is used to read the ONNX model saved previously in the file `"model.onnx"`.

- This method loads the model into an OpenVINO model object (`ov\_model`). The model is now in a format that OpenVINO can work with, but it is not yet optimized for inference.

### 3. Optional Optimizations and Compiling the Model

- `compiled\_model = core.compile\_model(ov\_model, "CPU")`:

- This line compiles the loaded model for a specific hardware target, in this case, a CPU.

- The `compile\_model` method optimizes the model for the specified hardware, potentially improving performance during inference. This step is crucial for leveraging hardware-specific optimizations that OpenVINO offers, particularly for Intel CPUs, GPUs, and other accelerators.

- The resulting `compiled\_model` is an optimized, executable representation of the original deep learning model. It can be used to run inference efficiently on the specified hardware (CPU in this instance).

### Summary

This code demonstrates how to use OpenVINO to load, optimize, and compile a deep learning model (originally in ONNX format) for efficient inference on a CPU. This process is integral to deploying deep learning models in production, especially in scenarios where inference speed and efficiency are critical. OpenVINO is particularly effective when used with Intel hardware, offering significant performance improvements.

MODEL\_DIR = '/content/sample\_data'  # Specify your directory as a string

MODEL\_NAME = "lipophilicity\_openvino"

# Ensure that `ov\_model` is the original OpenVINO model object

# Save the OpenVINO model to disk

ov.save\_model(ov\_model, MODEL\_DIR + "/" + f"{MODEL\_NAME}.xml")

The code snippet you've provided is performing the task of saving an OpenVINO model to disk. Here's a breakdown of what each part of the code is doing:

### 1. Setting Directory and Model Name

- `MODEL\_DIR = '/content/sample\_data'`:

- This line sets a variable `MODEL\_DIR` to the string `'/content/sample\_data'`. This string represents a file path where the model will be saved. The path appears to be structured for a Google Colab environment, as indicated by the `/content` prefix.

- `MODEL\_NAME = "lipophilicity\_openvino"`:

- Here, `MODEL\_NAME` is set to `"lipophilicity\_openvino"`. This is the name that will be given to the saved model file.

### 2. Saving the OpenVINO Model

- `ov.save\_model(ov\_model, MODEL\_DIR + "/" + f"{MODEL\_NAME}.xml")`:

- `ov.save\_model(...)`: This function is used to save an OpenVINO model to disk. The model must be an OpenVINO model object, which is indicated by the comment in the code.

- `ov\_model`: This is the OpenVINO model object that you want to save. It should already be loaded or created in a previous step of your code.

- `MODEL\_DIR + "/" + f"{MODEL\_NAME}.xml"`: This is the path where the model will be saved. It concatenates the directory path (`MODEL\_DIR`), a forward slash (acting as a directory separator), and the model name with the `.xml` extension. The `.xml` extension is used for OpenVINO models, which are saved in XML format.

- In the final form, it creates a path like `/content/sample\_data/lipophilicity\_openvino.xml`.

### Summary

This code saves an OpenVINO model to a specified directory with a specified name. The saved model file will be in XML format, which is standard for OpenVINO models. This is useful for persisting trained models, sharing them, or deploying them in different environments where OpenVINO is used for inference. The model can later be loaded from this file for performing inference tasks.

# Load OpenVINO model on device

compiled\_model = core.compile\_model(ov\_model, device.value)

compiled\_model

The code snippet you've provided is part of a process for compiling an OpenVINO model for a specific hardware device using the OpenVINO toolkit. Here's a breakdown of what it's doing:

### Loading and Compiling the OpenVINO Model for a Specific Device

1. `compiled\_model = core.compile\_model(ov\_model, device.value)`:

- `core.compile\_model(...)`: This method is called on an OpenVINO Core object (`core`) that you would have previously instantiated. The method compiles the model for optimized inference on a specified hardware device.

- `ov\_model`: This represents the OpenVINO model object you want to compile. This model would have been previously loaded or converted into OpenVINO's format.

- `device.value`: This specifies the hardware device you want to compile the model for. The `device` here is expected to be an object or variable that holds information about the target device for model compilation. The `.value` attribute is expected to contain the device identifier string (e.g., `'CPU'`, `'GPU'`, `'MYRIAD'` for VPU, etc.). The actual hardware device used will depend on what is available and compatible in your system and what `device.value` is set to.

2. `compiled\_model`:

- After the model is compiled, it is stored in the variable `compiled\_model`. This compiled model is an optimized version of your original model, tailored for efficient inference on the specified hardware device.

### Summary

This code is part of a workflow in OpenVINO to optimize and compile a deep learning model for efficient inference on a specific hardware device. The compiled model (`compiled\_model`) can then be used to perform inference tasks with improved performance, taking advantage of hardware-specific optimizations offered by OpenVINO. The actual device for compilation is determined by `device.value`, which should be set to the identifier of the desired inference hardware.

# Assuming you have a function to convert SMILES to fingerprints

def smiles\_to\_fp(smiles, n\_bits=2048):

    mol = Chem.MolFromSmiles(smiles)

    fp = AllChem.GetMorganFingerprintAsBitVect(mol, radius=2, nBits=n\_bits)

    return np.array(fp)

# Example SMILES string

smiles = "C[C@H](N)C(=O)O"  # Replace with your SMILES string

# Prepare input tensor

fp = smiles\_to\_fp(smiles)

input\_tensor = torch.tensor(fp, dtype=torch.float32).unsqueeze(0)  # Adding batch dimension

# Convert PyTorch tensor to NumPy array

input\_numpy = input\_tensor.numpy()

# Create OpenVINO tensor from NumPy array

ov\_input\_tensor = ov.Tensor(input\_numpy)

# Run model inference

result = compiled\_model([ov\_input\_tensor])[0]

# Postprocess and display the result

predicted\_lipophilicity = result[0]  # Assuming the model outputs a single value

print(f"Predicted Lipophilicity: {predicted\_lipophilicity}")

The provided code snippet is executing a series of steps to perform model inference using OpenVINO, based on input derived from a chemical structure represented as a SMILES string. Here's a detailed breakdown:

### 1. Defining a Function to Convert SMILES to Fingerprints

- `def smiles\_to\_fp(smiles, n\_bits=2048)`:

- This function, `smiles\_to\_fp`, is defined to convert a SMILES (Simplified Molecular Input Line Entry System) string to a molecular fingerprint.

- `mol = Chem.MolFromSmiles(smiles)`: Converts the SMILES string into an RDKit molecule object.

- `fp = AllChem.GetMorganFingerprintAsBitVect(mol, radius=2, nBits=n\_bits)`: Generates a Morgan fingerprint for the molecule, which is a type of circular fingerprint used in cheminformatics.

- `return np.array(fp)`: Converts the fingerprint to a NumPy array and returns it.

### 2. Preparing the Input for the Model

- `smiles = "C[C@H](N)C(=O)O"`:

- This line defines a SMILES string representing a specific molecule.

- `fp = smiles\_to\_fp(smiles)`:

- Converts the SMILES string to its fingerprint representation using the defined function.

- `input\_tensor = torch.tensor(fp, dtype=torch.float32).unsqueeze(0)`:

- Converts the fingerprint array to a PyTorch tensor with the appropriate data type (`float32`).

- `.unsqueeze(0)` adds an additional dimension to the tensor, effectively creating a batch dimension. This is necessary because models typically expect inputs in batch format, even if there's only one item in the batch.

### 3. Preparing the Input for OpenVINO Inference

- `input\_numpy = input\_tensor.numpy()`:

- Converts the PyTorch tensor to a NumPy array. This is necessary for compatibility with OpenVINO, which accepts NumPy arrays as input.

- `ov\_input\_tensor = ov.Tensor(input\_numpy)`:

- Creates an OpenVINO tensor from the NumPy array. This tensor is compatible with the OpenVINO runtime and can be used for inference.

### 4. Running Model Inference

- `result = compiled\_model([ov\_input\_tensor])[0]`:

- Runs inference on the `compiled\_model` using the prepared OpenVINO tensor.

- The `[0]` at the end extracts the first (and presumably only) output from the result.

### 5. Postprocessing and Displaying the Result

- `predicted\_lipophilicity = result[0]`:

- Extracts the predicted lipophilicity value from the result. This assumes that the model outputs a single value representing the lipophilicity.

- `print(f"Predicted Lipophilicity: {predicted\_lipophilicity}")`:

- Prints the predicted lipophilicity, displaying the result of the model inference.

### Summary

This code is a complete workflow for taking a chemical structure in SMILES format, converting it into a format suitable for a neural network model (a molecular fingerprint), and then using that fingerprint to perform inference with an OpenVINO-compiled model to predict a property of the molecule (in this case, lipophilicity). The process includes data conversion steps necessary to interface between different libraries and frameworks (RDKit, PyTorch, and OpenVINO).

from rdkit import Chem

from rdkit.Chem import Draw

import numpy as np

import torch

import openvino.runtime as ov

# Load your PyTorch model (assuming it's already trained and saved)

model = Net()  # Replace with your model class

model.load\_state\_dict(torch.load('lipophilicity\_model.pth'))

model.eval()

# OpenVINO setup (assuming you have already converted your model)

core = ov.Core()

ov\_model = core.read\_model('/content/sample\_data/lipophilicity\_openvino.xml')

compiled\_model = core.compile\_model(ov\_model, "CPU")

def predict\_and\_visualize(smiles):

    # Convert SMILES to fingerprint

    fp = smiles\_to\_fp(smiles)

    input\_tensor = torch.tensor(fp, dtype=torch.float32).unsqueeze(0)

    input\_numpy = input\_tensor.numpy()

    # Create OpenVINO tensor

    ov\_input\_tensor = ov.Tensor(input\_numpy)

    # Run model inference

    result = compiled\_model([ov\_input\_tensor])[0]

    predicted\_lipophilicity = result[0]

    # Visualize molecule

    mol = Chem.MolFromSmiles(smiles)

    img = Draw.MolToImage(mol)

    return predicted\_lipophilicity, img

# Example usage

smiles\_list = ["C[C@H](N)C(=O)O", "CCO", "CCN(CC)CC"]  # Replace with your SMILES strings

for smiles in smiles\_list:

    lipophilicity, img = predict\_and\_visualize(smiles)

    print(f"SMILES: {smiles}, Predicted Lipophilicity: {lipophilicity}")

    display(img)

The code snippet you've provided sets up a complete workflow for predicting a chemical property (lipophilicity) of molecules represented by SMILES (Simplified Molecular Input Line Entry System) strings, using both a PyTorch model and an OpenVINO-optimized model. It also includes a visualization of the molecules. Here's a breakdown of what each part of the code is doing:

### 1. Importing Necessary Libraries

- The code imports necessary modules from RDKit (a cheminformatics software), NumPy, PyTorch, and OpenVINO.

### 2. Loading and Setting Up the PyTorch Model

- The PyTorch model (`Net`) is loaded with its trained state and set to evaluation mode. This model is assumed to be trained to predict the lipophilicity of molecules based on their fingerprints.

### 3. Setting Up the OpenVINO Model

- An OpenVINO Core object is created, and the pre-converted ONNX model is read and compiled for inference on a CPU. This step optimizes the model for efficient execution on the specified hardware.

### 4. Defining the Prediction and Visualization Function

- `def predict\_and\_visualize(smiles)`:

- This function takes a SMILES string as input.

- The SMILES string is converted to a molecular fingerprint, which is then converted into a tensor compatible with both PyTorch and OpenVINO.

- Inference is run on the compiled OpenVINO model using the fingerprint.

- The molecular structure represented by the SMILES string is visualized using RDKit's drawing tools.

- The function returns the predicted lipophilicity and the image of the molecule.

### 5. Running Predictions on a List of SMILES Strings

- The code iterates over a list of SMILES strings (`smiles\_list`), predicts the lipophilicity for each molecule using the `predict\_and\_visualize` function, and prints the results. It also displays the visual representation of each molecule.

### Summary

This code integrates cheminformatics (RDKit), machine learning (PyTorch), and model optimization and inference (OpenVINO) to predict a chemical property (lipophilicity) from molecular structures (SMILES). It demonstrates a sophisticated use case involving the intersection of chemistry and artificial intelligence, showcasing the capabilities of these libraries for tasks in computational chemistry and drug discovery.

import streamlit as st

from rdkit import Chem

from rdkit.Chem import Draw, AllChem

from PIL import Image

import numpy as np

import torch

import openvino.runtime as ov

# Define the function to convert SMILES to fingerprints

def smiles\_to\_fp(smiles, n\_bits=2048):

    mol = Chem.MolFromSmiles(smiles)

    fp = AllChem.GetMorganFingerprintAsBitVect(mol, radius=2, nBits=n\_bits)

    return np.array(fp)

# Load the OpenVINO model (update the path as needed)

model\_path = 'lipophilicity\_openvino.xml'  # Update this path

core = ov.Core()

compiled\_model = core.compile\_model(model\_path, "CPU")

# Define the prediction function

def predict\_lipophilicity(smiles):

    fp = smiles\_to\_fp(smiles)

    input\_tensor = torch.tensor(fp, dtype=torch.float32).unsqueeze(0)

    input\_numpy = input\_tensor.numpy()

    # Create OpenVINO tensor from NumPy array

    ov\_input\_tensor = ov.Tensor(input\_numpy)

    # Run model inference

    result = compiled\_model([ov\_input\_tensor])[0]

    return result[0]

# Streamlit User Interface

st.title('Lipophilicity Prediction App')

st.write('Select a SMILES string to predict its lipophilicity and visualize the molecule.')

# Example SMILES strings

# smiles\_options = ["C[C@H](N)C(=O)O", "CCO", "CCN(CC)CC", ...]  # Add your SMILES strings here

smiles\_options = [

    "C[C@H](N)C(=O)O", "CCO", "CCN(CC)CC", "CC(=O)O", "C1=CC=C(C=C1)C(=O)O",

    "C1CCC(CC1)N", "CC(C(=O)O)N", "C1CCCCC1", "C1=CC=CC=C1", "C1=CN=C(N=C1)N",

    "C1CC1", "C1=CC=C(C=C1)O", "C1=CN=CN1", "C1=CC=C(C=C1)N", "C1=CC=CC=C1N",

    "C1CCC(CC1)O", "C1=CC=C(C=C1)Cl", "C1=CN=C(N=C1)N", "C1CCNCC1", "C1=CC=C(C=C1)F"

]

# Dropdown for SMILES selection

selected\_smiles = st.selectbox("Select a SMILES String", smiles\_options)

# Button to make a prediction

if st.button('Predict Lipophilicity'):

    predicted\_lipophilicity = predict\_lipophilicity(selected\_smiles)

    st.write(f"Predicted Lipophilicity: {predicted\_lipophilicity}")

    # Visualize the molecule

    mol = Chem.MolFromSmiles(selected\_smiles)

    mol\_image = Draw.MolToImage(mol)

    st.image(mol\_image, caption='Molecular Structure')

The code snippet you've provided is for a web application built using Streamlit, a Python framework for creating web apps quickly and easily. This particular app predicts the lipophilicity of molecules based on their SMILES (Simplified Molecular Input Line Entry System) strings and visualizes their structures. Here's a detailed explanation of its components and functionality:

### 1. Importing Necessary Libraries

- The code imports necessary libraries, including Streamlit for the web app interface, RDKit for cheminformatics functions, PIL for image processing, NumPy for numerical operations, PyTorch for tensor operations, and OpenVINO for optimized model inference.

### 2. Defining Functions

- SMILES to Fingerprint Conversion:

- `smiles\_to\_fp`: This function converts a SMILES string to a molecular fingerprint, which is a numerical representation used for machine learning models.

- Load and Compile the OpenVINO Model:

- The OpenVINO model is loaded and compiled for inference on a CPU. This model is expected to be trained for predicting lipophilicity based on molecular fingerprints.

- Predict Lipophilicity Function:

- `predict\_lipophilicity`: This function takes a SMILES string, converts it to a fingerprint, creates a tensor suitable for model input, and performs inference using the compiled OpenVINO model to predict the lipophilicity.

### 3. Streamlit User Interface

- App Title and Description:

- The app is given a title ("Lipophilicity Prediction App") and a brief description using Streamlit's `st.title` and `st.write` functions.

- SMILES String Selection:

- A dropdown menu is created using `st.selectbox`, allowing users to select a SMILES string from a predefined list (`smiles\_options`). These strings represent different chemical compounds.

- Prediction and Visualization:

- When the user clicks the 'Predict Lipophilicity' button (`st.button`), the app calls the `predict\_lipophilicity` function with the selected SMILES string, displays the predicted lipophilicity, and visualizes the molecular structure.

- The molecular structure is visualized using RDKit's `Draw.MolToImage` function, and the image is displayed in the app with `st.image`.

### Summary

This Streamlit web application allows users to select a chemical compound represented by a SMILES string, predicts its lipophilicity using a pre-trained machine learning model (optimized with OpenVINO for CPU-based inference), and visualizes the molecular structure of the compound. The app demonstrates an interesting use case of combining cheminformatics, machine learning, and web development for a scientific application.

What is the webapp doing

The described web application is a tool designed for predicting the lipophilicity of chemical compounds and visualizing their molecular structures. It uses a combination of cheminformatics, machine learning, and web development technologies. Here's a summary of its functionality and components:

### Core Functionalities of the Web App

1. Predicting Lipophilicity:

- Users can input a chemical compound in the form of a SMILES (Simplified Molecular Input Line Entry System) string. SMILES is a notation that encodes the structure of chemical compounds as text strings.

- The application then predicts the lipophilicity of the chosen compound. Lipophilicity, which is the ability of a chemical compound to dissolve in fats, oils, and non-polar solvents (as opposed to water), is an important property in pharmacology and chemistry.

2. Visualizing Molecular Structures:

- The app also provides a visual representation of the molecular structure of the input compound. This is helpful for users to understand and verify the chemical structure of the compound they are analyzing.

### Technical Components

1. Streamlit for Web Interface:

- The app is built using Streamlit, a Python library that simplifies the process of creating and deploying web applications. Streamlit is particularly popular in data science for quickly turning data scripts into shareable web apps.

2. RDKit for Cheminformatics:

- RDKit, a collection of cheminformatics and machine learning tools, is used for processing chemical information. In this app, it converts SMILES strings to molecular fingerprints (numerical representations) and generates images of the molecular structures.

3. OpenVINO for Model Inference:

- Intel's OpenVINO toolkit is employed for running the machine learning model that predicts lipophilicity. The model seems to be pre-trained and optimized for efficient performance on CPUs, which is a typical use case for OpenVINO.

4. PyTorch and NumPy for Data Handling:

- PyTorch is used for handling tensor operations, and NumPy for numerical operations. These libraries are integral in transforming the molecular data into a format suitable for the machine learning model.

### User Interaction

- Users interact with the app through a simple and intuitive interface. They can select a compound from a dropdown menu of predefined SMILES strings or potentially input their own (depending on app configuration). After selection, the app displays the predicted lipophilicity and a visual of the compound's structure.

### Conclusion

This web application demonstrates an innovative intersection of chemistry, machine learning, and web technology. It serves as a practical tool for chemists, pharmacologists, or anyone interested in the study of chemical compounds, allowing them to quickly predict and visualize important properties of molecules.