# BAND NN: A Deep Learning Framework for Energy Prediction of Organic Small Molecules

The paper proposes a machine learning approach to predict the energies of molecules at par with those of density functional theory calculations. The method is inspired from the classical molecular force fields by building four different neural networks for bonds (B), angles (A), nonbonds (N), dihedrals (D) to predict energies independently and a sum of all these predictions is estimated as the total energy of the molecule. As a supervised learning task, the model is improved via the loss function computed on the training data's ground truths.

$$E_{\text{total}} = E_{\text{bonded}} + E_{\text{nonbonded}},$$

$$E_{\text{bonded}} = E_{\text{bonds}} + E_{\text{angles}} + E_{\text{dihedrals}}.$$

## The method:

Firstly, a subset of ANI-1 dataset was used. The subset of size of 100,000 molecules was synthesized and tuned for generating features (discussed below). Four different feed forward neural networks were built based on the molecule descriptors for bonds, angles, nonbonds and dihedrals. The estimated energy was computed by sum of the predictions of each of the models and the loss was minimized by computing the gradients.

## Data extraction and features generation:

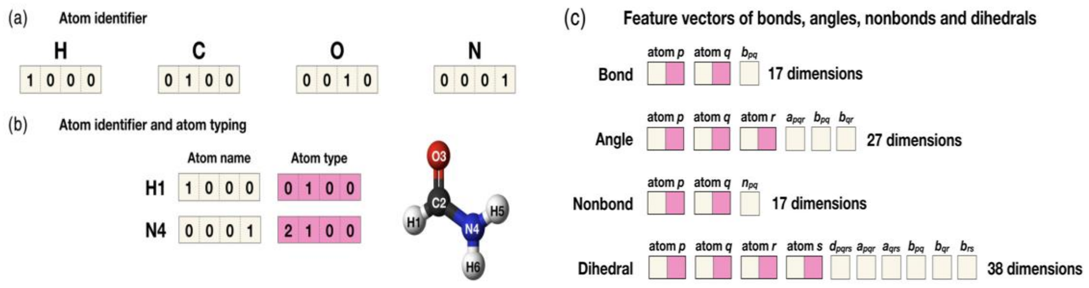

(a) Atom identifier
H 1 0 0 0
C 0 1 0 0
O 0 0 1 0
N 0 0 0 1
(b) Atom identifier and atom typing
Atom name
Atom type
H1 1 0 0 0 0 1 0 0
N4 0 0 0 1 2 1 0 0
O3 C2 H1 N4 H5 H6
(c) Feature vectors of bonds, angles, nonbonds and dihedrals
Bond atom p atom q b_pq 17 dimensions
Angle atom p atom q atom r a_pqr b_pq b_qr 27 dimensions
Nonbond atom p atom q n_pq 17 dimensions
Dihedral atom p atom q atom r atom s d_pqrs a_pqr a_qrs b_pq b_qr b_rs 38 dimensions

*Figure 1 (a), (b) and (c) shows the feature vector composition*

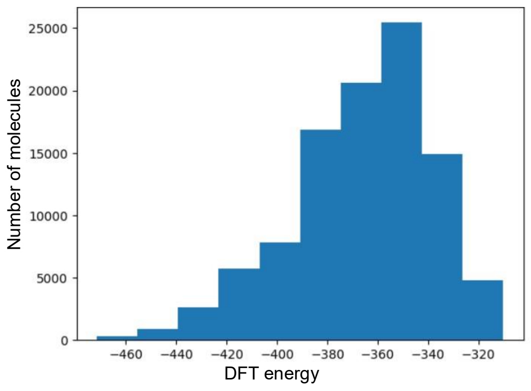


*Figure 2 histogram plot of the dataset against DFT energies*

There is no specific theory behind the choice of the selection of these 100,000 molecules. The histogram (Fig. 2) gives an idea about the distribution of the selected molecules on different energies. Since the data seems to be populated in the range of -360 eV to -340 eV, it can be expected for the model to learn better around these data points.

To generate features, the python script featurizer.py from the GitHub repository BANDNN (https://github.com/devalab/BAND-NN.git) was directly implemented in the workflow. Features vectors are defined as follows:

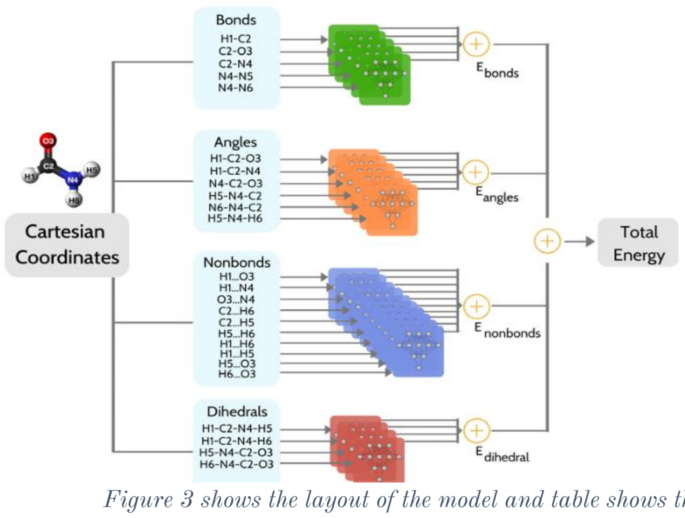
- Atom is represented by an eight-dimensional vector, first four representing the name of the atom and the rest representing the atom type in terms of how many of the C, N, O, H atoms are connected to it,
- Bond feature vector: atomic representation of the bonded pairs + bond length (17-dimensional vector),
- Angle feature vector: atomic representation of the three atoms + two bond lengths + bond angle (27- dimensional),
- Non bond feature vector: atomic representation of between two nonbonded atoms (within cutoff radius of 6 Angstroms) + internuclear distance (17-dimensional),
- Dihedral feature vectors: atomic representation of the four consecutive atoms + dihedral angle + two angles + three bond lengths (34-dimensional).

The Fig 1 shows a summary of the feature vector representation of the same.

## The model and training

The model was built using the nn.Module of the PyTorch library. A simple Artificial Neural Network architecture was made. The layout of the framework followed is provided in the Fig 3. The code snippet below shows the outline of the model developed for training and evaluation. The hidden layers use ReLU activation function. Adam optimizer ($\beta_1 = 0.9$, $\beta_2 = 0.999$) with learning rate of 0.01 was used for an effective gradient descent computation. Mean squared error loss function was employed.

An 80:20 split for training and testing data was done on the dataset. The model was trained for nine epochs with each epoch of containing batches of shuffled feature variables and targets of size 32.



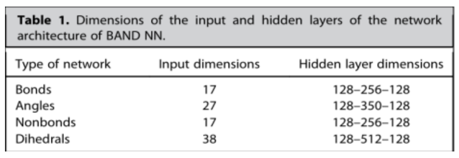
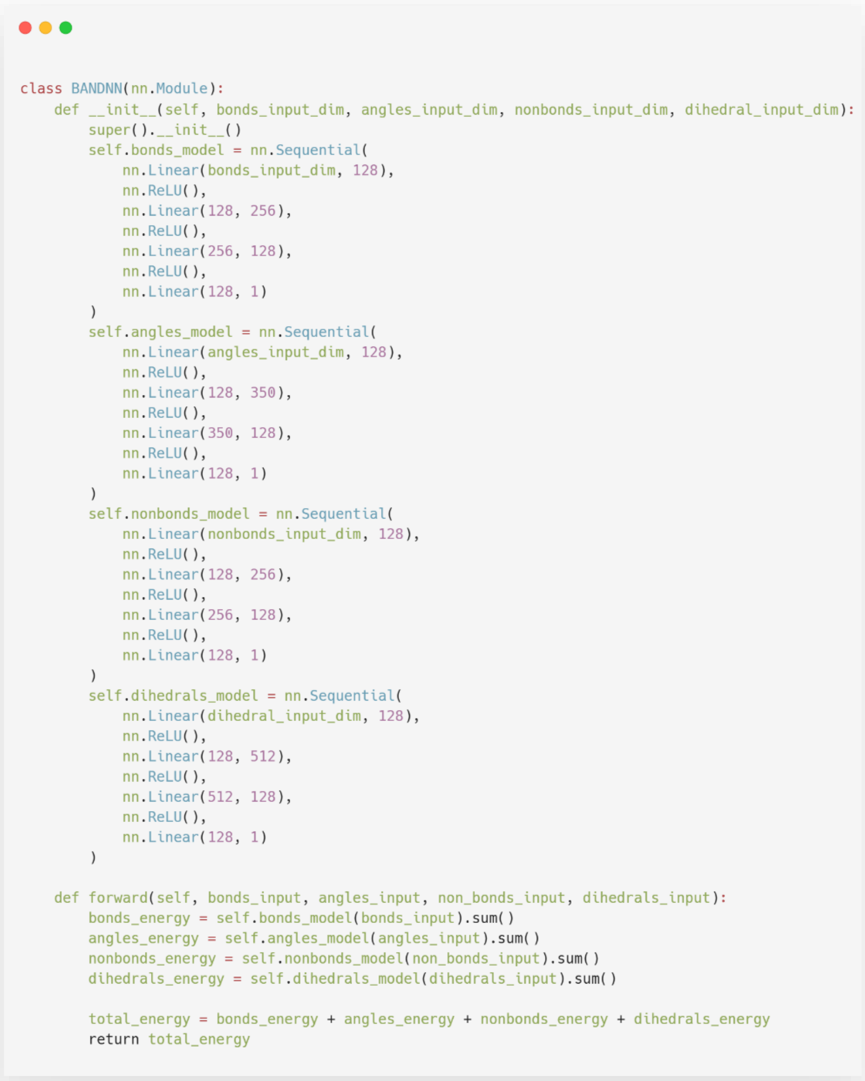


**Table 1.** Dimensions of the input and hidden layers of the network architecture of BAND NN.

| Type of network | Input dimensions | Hidden layer dimensions |
|---|---|---|
| Bonds | 17 | 128–256–128 |
| Angles | 27 | 128–350–128 |
| Nonbonds | 17 | 128–256–128 |
| Dihedrals | 38 | 128–512–128 |

*Figure 3 shows the layout of the model and table shows the hidden layer dimensions*



```
class BANDNN(nn.Module):
    def __init__(self, bonds_input_dim, angles_input_dim, nonbonds_input_dim, dihedral_input_dim):
        super().__init__()
        self.bonds_model = nn.Sequential(
            nn.Linear(bonds_input_dim, 128),
            nn.ReLU(),
            nn.Linear(128, 256),
            nn.ReLU(),
            nn.Linear(256, 128),
            nn.ReLU(),
            nn.Linear(128, 1)
        )
        self.angles_model = nn.Sequential(
            nn.Linear(angles_input_dim, 128),
            nn.ReLU(),
            nn.Linear(128, 350),
            nn.ReLU(),
            nn.Linear(350, 128),
            nn.ReLU(),
            nn.Linear(128, 1)
        )
        self.nonbonds_model = nn.Sequential(
            nn.Linear(nonbonds_input_dim, 128),
            nn.ReLU(),
            nn.Linear(128, 256),
            nn.ReLU(),
            nn.Linear(256, 128),
            nn.ReLU(),
            nn.Linear(128, 1)
        )
        self.dihedrals_model = nn.Sequential(
            nn.Linear(dihedral_input_dim, 128),
            nn.ReLU(),
            nn.Linear(128, 512),
            nn.ReLU(),
            nn.Linear(512, 128),
            nn.ReLU(),
            nn.Linear(128, 1)
        )

    def forward(self, bonds_input, angles_input, non_bonds_input, dihedrals_input):
        bonds_energy = self.bonds_model(bonds_input).sum()
        angles_energy = self.angles_model(angles_input).sum()
        nonbonds_energy = self.nonbonds_model(non_bonds_input).sum()
        dihedrals_energy = self.dihedrals_model(dihedrals_input).sum()

        total_energy = bonds_energy + angles_energy + nonbonds_energy + dihedrals_energy
        return total_energy
```

## Results and comments

The model when trained for 20 epochs at learning rate of 0.1, an average mean squared error on the training data of 1079.3 was observed however, a significant decrease in the this was seen when trained on learning rate of 0.01 for 9 epochs (~600). This further decreased when to 568.8 when trained for 20 epochs. The codes of the pipeline and training is available in the GitHub repository BANDNN_PyTorch (https://github.com/adithyamauryakr/BANDNN_PyTorch.git).

Upon evaluation on the testing data, it was seen that the model trained on learning rate of 0.1 showed an r2 score of -0.24 which and on learning rate of 0.01 gave r2 score of 0.348 which is quite some improvement from the former. Due to resources and data limitations, the model could not be further improvised. However, there is a huge scope for improvement via hyperparameter tuning (for which a sample code snippet is provided in the appendix) and

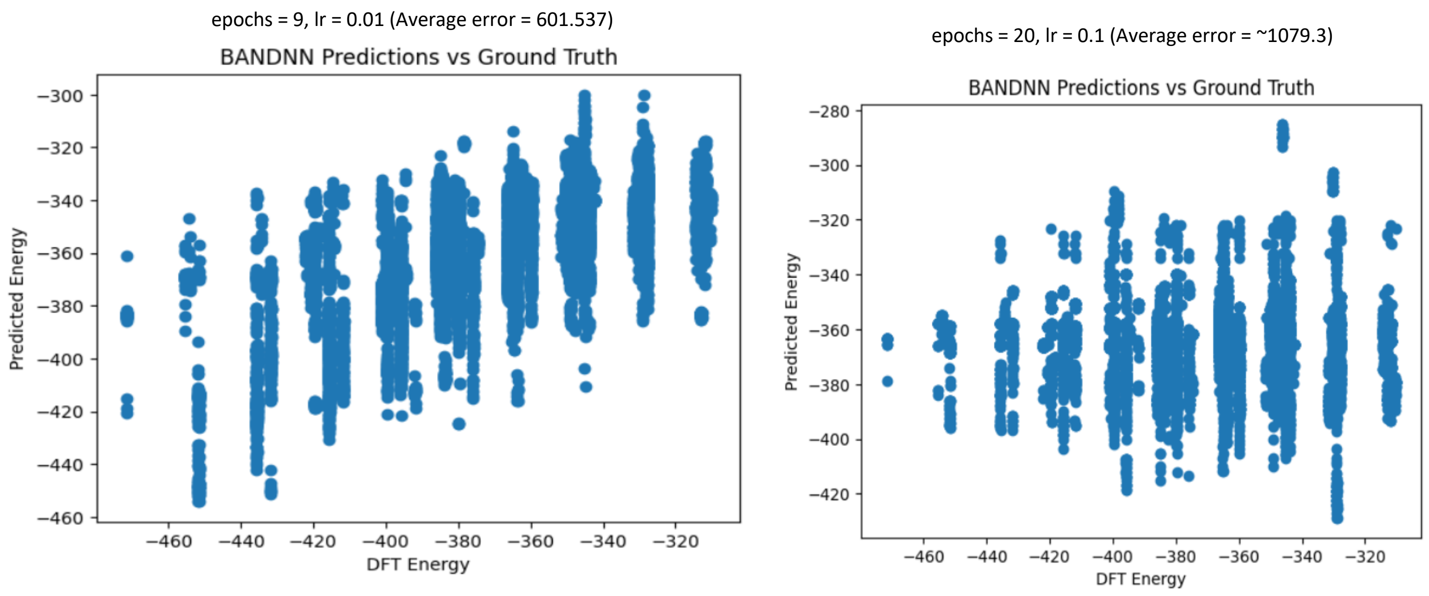


*Figure 4 parity plot of the model performance on the testing data*

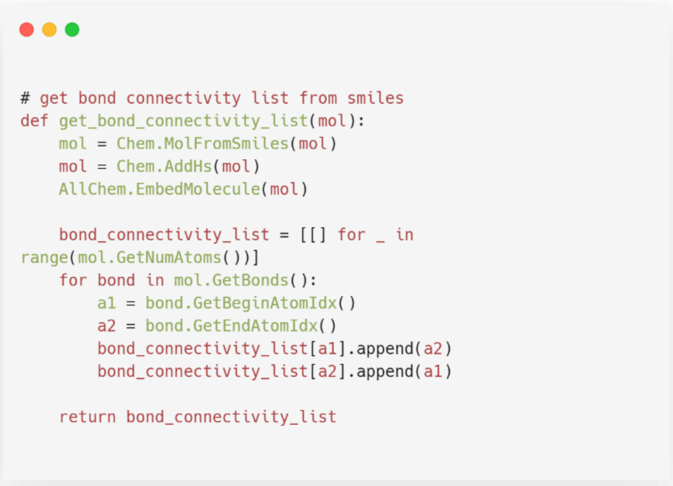
increasing the dataset size. Feedback and guidance on this approach is highly appreciated for further improvements. For applications such as geometry optimization, a model with low error (as close to zero) would be ideal since it captures the essence of the objective we intend for. The original paper does show a success in geometry optimization through the BANDNN model through along with other vital applications. This is valuable as the machine learning models that have been proposed earlier use atomic positions as input of already DFT optimized structures. This makes those models ineffective for energy predictions during the course of the geometry optimization.

## References

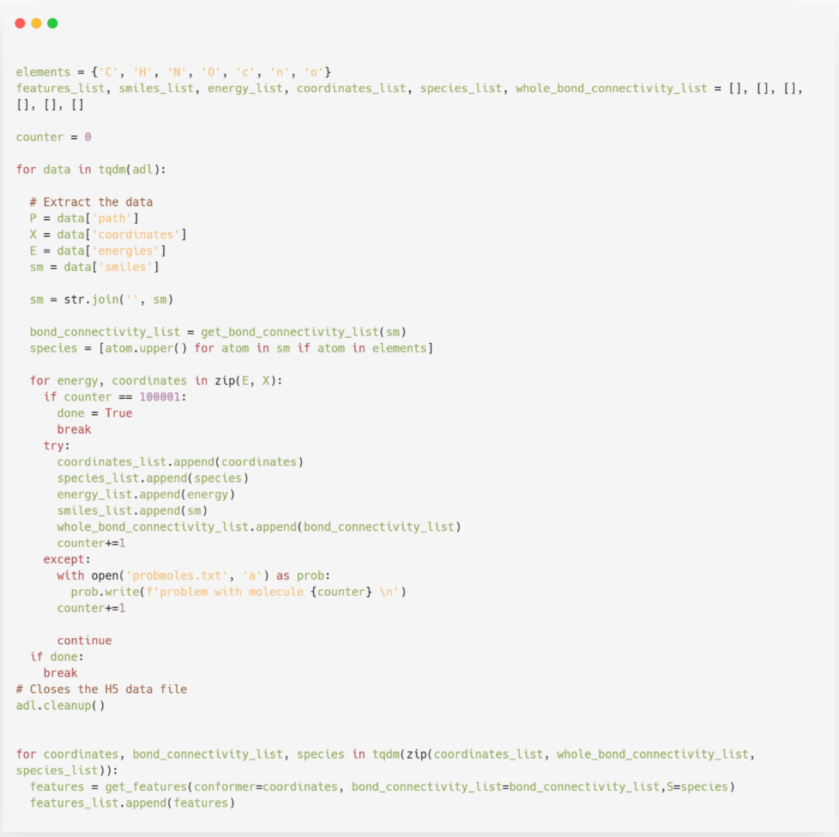
1. Laghuvarapu, S., Pathak, Y., & Priyakumar, U. D. (2019). BAND NN: a deep learning framework for energy prediction and geometry optimization of organic small molecules. *Journal of Computational Chemistry*, *41*(8), 790–799. https://doi.org/10.1002/jcc.26128
2. https://github.com/devalab/BAND-NN.git

# Appendix

1. Data extraction and featurization

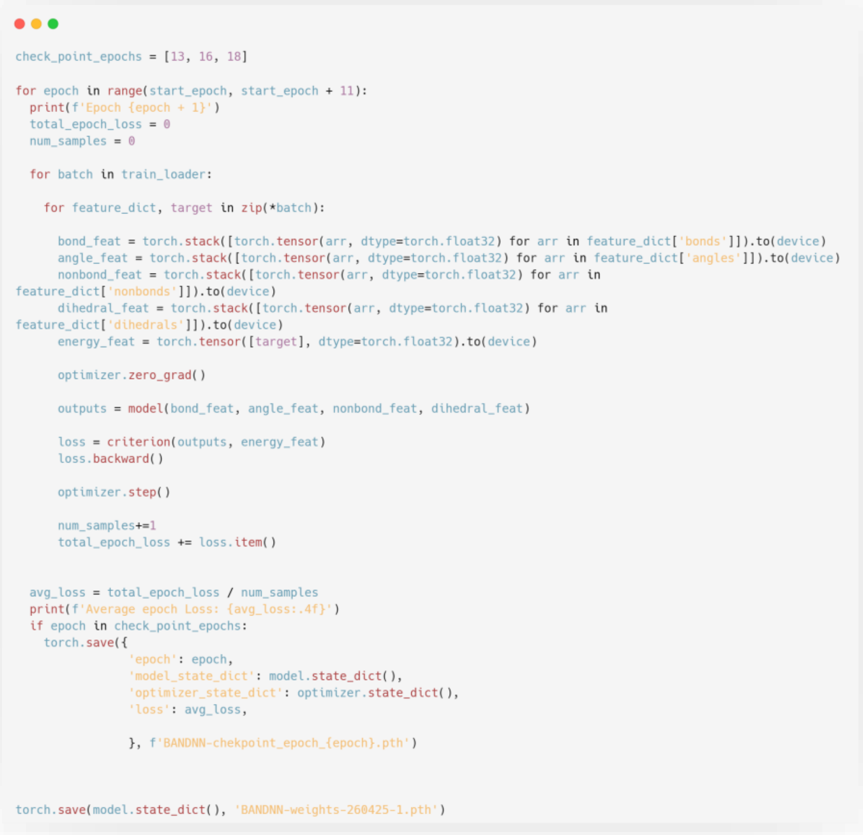


```python
# get bond connectivity list from smiles
def get_bond_connectivity_list(mol):
    mol = Chem.MolFromSmiles(mol)
    mol = Chem.AddHs(mol)
    AllChem.EmbedMolecule(mol)

    bond_connectivity_list = [[] for _ in
range(mol.GetNumAtoms())]
    for bond in mol.GetBonds():
        a1 = bond.GetBeginAtomIdx()
        a2 = bond.GetEndAtomIdx()
        bond_connectivity_list[a1].append(a2)
        bond_connectivity_list[a2].append(a1)

    return bond_connectivity_list
```



```python
elements = {'C', 'H', 'N', 'O', 'c', 'n', 'o'}
features_list, smiles_list, energy_list, coordinates_list, species_list, whole_bond_connectivity_list = [], [], [],
[], [], []

counter = 0

for data in tqdm(adl):

  # Extract the data
  P = data['path']
  X = data['coordinates']
  E = data['energies']
  sm = data['smiles']

  sm = str.join('', sm)

  bond_connectivity_list = get_bond_connectivity_list(sm)
  species = [atom.upper() for atom in sm if atom in elements]

  for energy, coordinates in zip(E, X):
    if counter == 100001:
      done = True
      break
    try:
      coordinates_list.append(coordinates)
      species_list.append(species)
      energy_list.append(energy)
      smiles_list.append(sm)
      whole_bond_connectivity_list.append(bond_connectivity_list)
      counter+=1
    except:
      with open('probmoles.txt', 'a') as prob:
        prob.write(f'problem with molecule {counter} \n')
      counter+=1

      continue
  if done:
    break
# Closes the H5 data file
adl.cleanup()


for coordinates, bond_connectivity_list, species in tqdm(zip(coordinates_list, whole_bond_connectivity_list,
species_list)):
  features = get_features(conformer=coordinates, bond_connectivity_list=bond_connectivity_list,S=species)
  features_list.append(features)
```

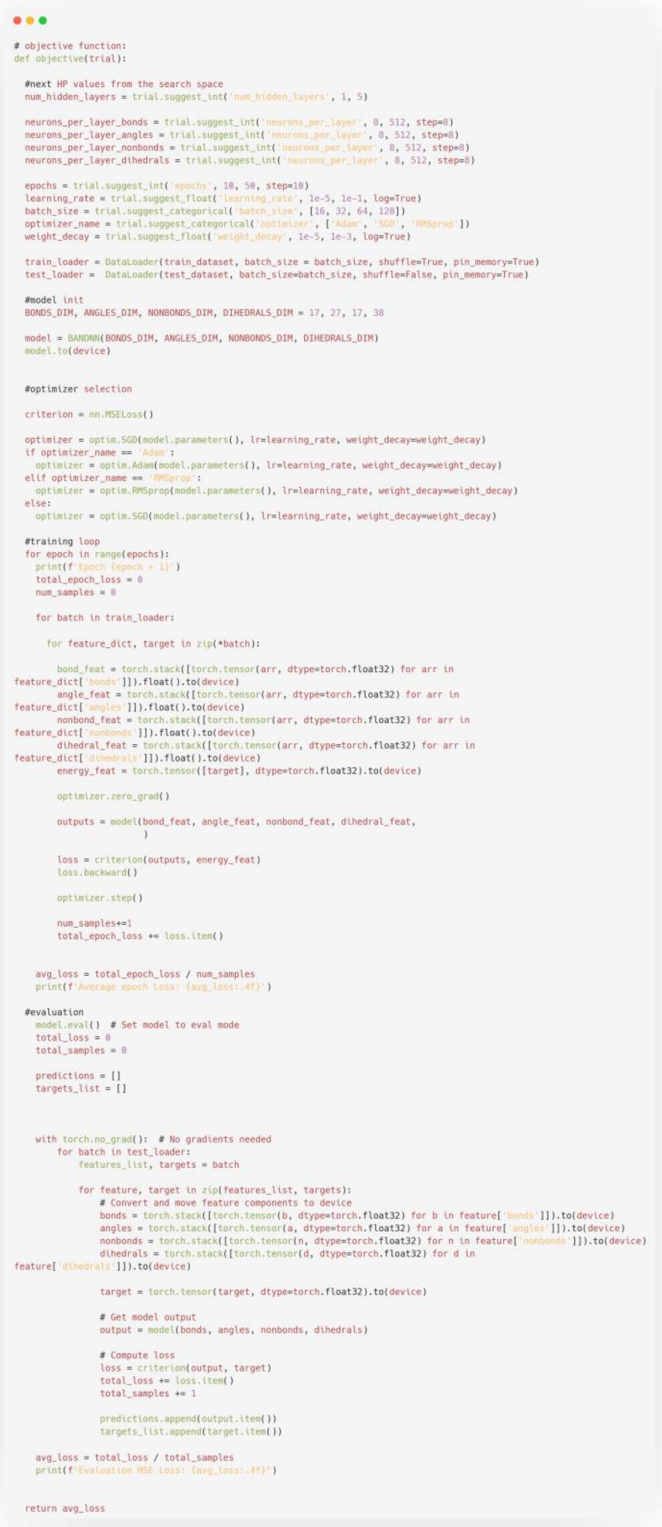
2. Training loop:



```python
check_point_epochs = [13, 16, 18]

for epoch in range(start_epoch, start_epoch + 11):
  print(f'Epoch {epoch + 1}')
  total_epoch_loss = 0
  num_samples = 0

  for batch in train_loader:

    for feature_dict, target in zip(*batch):

      bond_feat = torch.stack([torch.tensor(arr, dtype=torch.float32) for arr in feature_dict['bonds']]).to(device)
      angle_feat = torch.stack([torch.tensor(arr, dtype=torch.float32) for arr in feature_dict['angles']]).to(device)
      nonbond_feat = torch.stack([torch.tensor(arr, dtype=torch.float32) for arr in
feature_dict['nonbonds']]).to(device)
      dihedral_feat = torch.stack([torch.tensor(arr, dtype=torch.float32) for arr in
feature_dict['dihedrals']]).to(device)
      energy_feat = torch.tensor([target], dtype=torch.float32).to(device)

      optimizer.zero_grad()

      outputs = model(bond_feat, angle_feat, nonbond_feat, dihedral_feat)

      loss = criterion(outputs, energy_feat)
      loss.backward()

      optimizer.step()

      num_samples+=1
      total_epoch_loss += loss.item()


  avg_loss = total_epoch_loss / num_samples
  print(f'Average epoch Loss: {avg_loss:.4f}')
  if epoch in check_point_epochs:
    torch.save({
                'epoch': epoch,
                'model_state_dict': model.state_dict(),
                'optimizer_state_dict': optimizer.state_dict(),
                'loss': avg_loss,

                }, f'BANDNN-chekpoint_epoch_{epoch}.pth')



torch.save(model.state_dict(), 'BANDNN-weights-260425-1.pth')
```

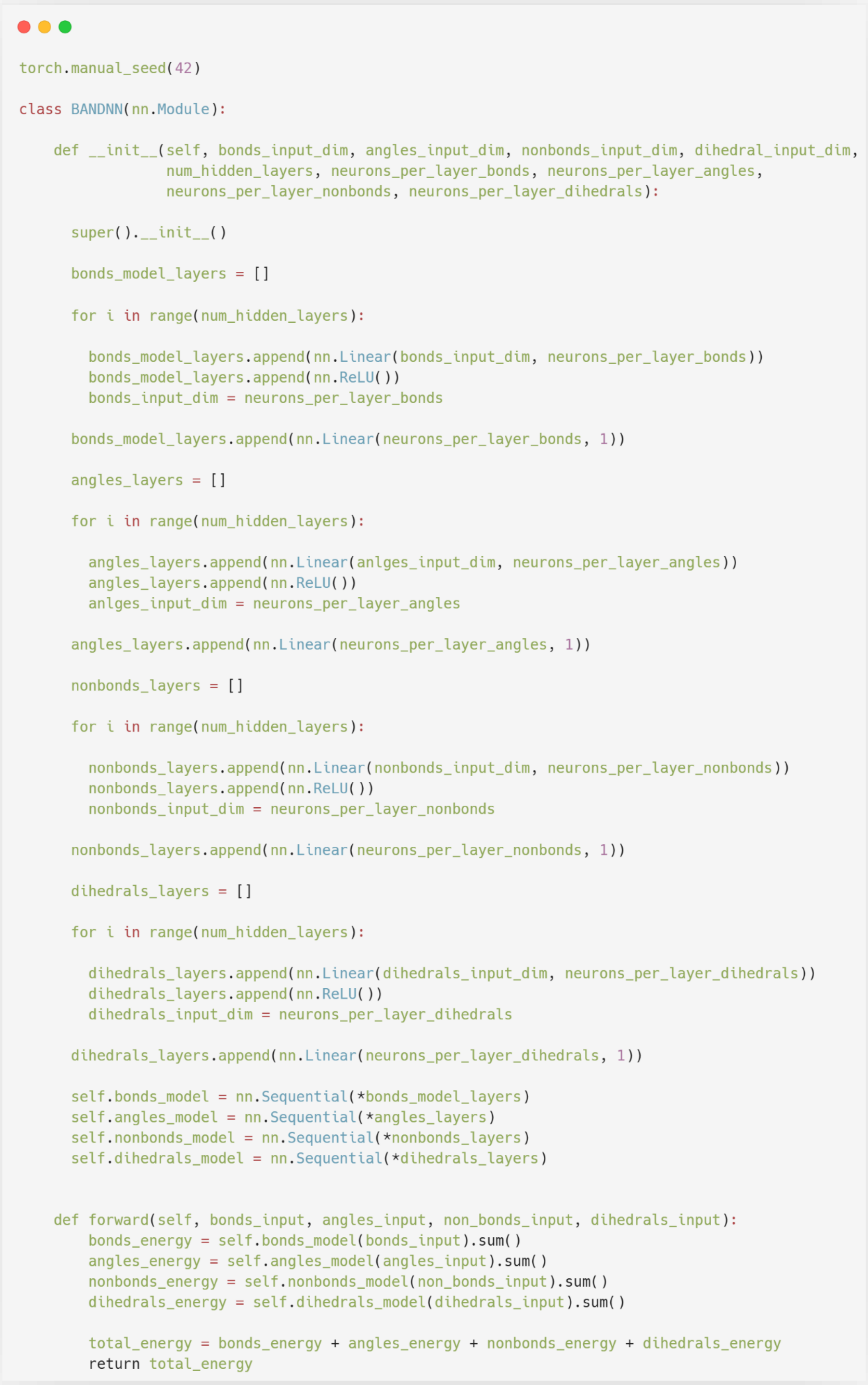
Hyper parameter tuning:

a. Objective function:



```python
# objective function:
def objective(trial):

  #next HP values from the search space
  num_hidden_layers = trial.suggest_int('num_hidden_layers', 1, 5)

  neurons_per_layer_bonds = trial.suggest_int('neurons_per_layer', 8, 512, step=8)
  neurons_per_layer_angles = trial.suggest_int('neurons_per_layer', 8, 512, step=8)
  neurons_per_layer_nonbonds = trial.suggest_int('neurons_per_layer', 8, 512, step=8)
  neurons_per_layer_dihedrals = trial.suggest_int('neurons_per_layer', 8, 512, step=8)

  epochs = trial.suggest_int('epochs', 10, 50, step=10)
  learning_rate = trial.suggest_float('learning_rate', 1e-5, 1e-1, log=True)
  batch_size = trial.suggest_categorical('batch_size', [16, 32, 64, 128])
  optimizer_name = trial.suggest_categorical('optimizer', ['Adam', 'SGD', 'RMSprop'])
  weight_decay = trial.suggest_float('weight_decay', 1e-5, 1e-3, log=True)

  train_loader = DataLoader(train_dataset, batch_size = batch_size, shuffle=True, pin_memory=True)
  test_loader =  DataLoader(test_dataset, batch_size=batch_size, shuffle=False, pin_memory=True)

  #model init
  BONDS_DIM, ANGLES_DIM, NONBONDS_DIM, DIHEDRALS_DIM = 17, 27, 17, 38

  model = BANDNN(BONDS_DIM, ANGLES_DIM, NONBONDS_DIM, DIHEDRALS_DIM)
  model.to(device)


  #optimizer selection

  criterion = nn.MSELoss()

  optimizer = optim.SGD(model.parameters(), lr=learning_rate, weight_decay=weight_decay)
  if optimizer_name == 'Adam':
    optimizer = optim.Adam(model.parameters(), lr=learning_rate, weight_decay=weight_decay)
  elif optimizer_name == 'RMSprop':
    optimizer = optim.RMSprop(model.parameters(), lr=learning_rate, weight_decay=weight_decay)
  else:
    optimizer = optim.SGD(model.parameters(), lr=learning_rate, weight_decay=weight_decay)

  #training loop
  for epoch in range(epochs):
    print(f'Epoch {epoch + 1}')
    total_epoch_loss = 0
    num_samples = 0

    for batch in train_loader:

      for feature_dict, target in zip(*batch):

        bond_feat = torch.stack([torch.tensor(arr, dtype=torch.float32) for arr in
feature_dict['bonds']]).float().to(device)
        angle_feat = torch.stack([torch.tensor(arr, dtype=torch.float32) for arr in
feature_dict['angles']]).float().to(device)
        nonbond_feat = torch.stack([torch.tensor(arr, dtype=torch.float32) for arr in
feature_dict['nonbonds']]).float().to(device)
        dihedral_feat = torch.stack([torch.tensor(arr, dtype=torch.float32) for arr in
feature_dict['dihedrals']]).float().to(device)
        energy_feat = torch.tensor([target], dtype=torch.float32).to(device)

        optimizer.zero_grad()

        outputs = model(bond_feat, angle_feat, nonbond_feat, dihedral_feat,
                        )

        loss = criterion(outputs, energy_feat)
        loss.backward()

        optimizer.step()

        num_samples+=1
        total_epoch_loss += loss.item()


    avg_loss = total_epoch_loss / num_samples
    print(f'Average epoch Loss: {avg_loss:.4f}')

  #evaluation
    model.eval()  # Set model to eval mode
    total_loss = 0
    total_samples = 0

    predictions = []
    targets_list = []



    with torch.no_grad():  # No gradients needed
        for batch in test_loader:
            features_list, targets = batch

            for feature, target in zip(features_list, targets):
                # Convert and move feature components to device
                bonds = torch.stack([torch.tensor(b, dtype=torch.float32) for b in feature['bonds']]).to(device)
                angles = torch.stack([torch.tensor(a, dtype=torch.float32) for a in feature['angles']]).to(device)
                nonbonds = torch.stack([torch.tensor(n, dtype=torch.float32) for n in feature['nonbonds']]).to(device)
                dihedrals = torch.stack([torch.tensor(d, dtype=torch.float32) for d in
feature['dihedrals']]).to(device)

                target = torch.tensor(target, dtype=torch.float32).to(device)

                # Get model output
                output = model(bonds, angles, nonbonds, dihedrals)

                # Compute loss
                loss = criterion(output, target)
                total_loss += loss.item()
                total_samples += 1

                predictions.append(output.item())
                targets_list.append(target.item())

    avg_loss = total_loss / total_samples
    print(f"Evaluation MSE Loss: {avg_loss:.4f}")


  return avg_loss
```

b. Model architecture:



```python
torch.manual_seed(42)

class BANDNN(nn.Module):

    def __init__(self, bonds_input_dim, angles_input_dim, nonbonds_input_dim, dihedral_input_dim,
                 num_hidden_layers, neurons_per_layer_bonds, neurons_per_layer_angles,
                 neurons_per_layer_nonbonds, neurons_per_layer_dihedrals):

      super().__init__()

      bonds_model_layers = []

      for i in range(num_hidden_layers):

        bonds_model_layers.append(nn.Linear(bonds_input_dim, neurons_per_layer_bonds))
        bonds_model_layers.append(nn.ReLU())
        bonds_input_dim = neurons_per_layer_bonds

      bonds_model_layers.append(nn.Linear(neurons_per_layer_bonds, 1))

      angles_layers = []

      for i in range(num_hidden_layers):

        angles_layers.append(nn.Linear(anlges_input_dim, neurons_per_layer_angles))
        angles_layers.append(nn.ReLU())
        anlges_input_dim = neurons_per_layer_angles

      angles_layers.append(nn.Linear(neurons_per_layer_angles, 1))

      nonbonds_layers = []

      for i in range(num_hidden_layers):

        nonbonds_layers.append(nn.Linear(nonbonds_input_dim, neurons_per_layer_nonbonds))
        nonbonds_layers.append(nn.ReLU())
        nonbonds_input_dim = neurons_per_layer_nonbonds

      nonbonds_layers.append(nn.Linear(neurons_per_layer_nonbonds, 1))

      dihedrals_layers = []

      for i in range(num_hidden_layers):

        dihedrals_layers.append(nn.Linear(dihedrals_input_dim, neurons_per_layer_dihedrals))
        dihedrals_layers.append(nn.ReLU())
        dihedrals_input_dim = neurons_per_layer_dihedrals

      dihedrals_layers.append(nn.Linear(neurons_per_layer_dihedrals, 1))

      self.bonds_model = nn.Sequential(*bonds_model_layers)
      self.angles_model = nn.Sequential(*angles_layers)
      self.nonbonds_model = nn.Sequential(*nonbonds_layers)
      self.dihedrals_model = nn.Sequential(*dihedrals_layers)


    def forward(self, bonds_input, angles_input, non_bonds_input, dihedrals_input):
        bonds_energy = self.bonds_model(bonds_input).sum()
        angles_energy = self.angles_model(angles_input).sum()
        nonbonds_energy = self.nonbonds_model(non_bonds_input).sum()
        dihedrals_energy = self.dihedrals_model(dihedrals_input).sum()

        total_energy = bonds_energy + angles_energy + nonbonds_energy + dihedrals_energy
        return total_energy
```