Machine Learning Final Project Report Spring 2024

Jedrick Regala Zablan

CHEM 277B

1. INTRODUCTION

The advancement of machine learning models has revolutionized the technological landscape, broadening the possibilities for learning and analysis across diverse fields. In the field of computational chemistry, machine learning models facilitate a better understanding of the molecular stability and atomic interactions which can be critical in industrial applications of chemistry such as drug discovery. Furthermore, there is potential for optimizing performance to minimize temporal and memory overhead to allow for a more effective research workflow. Deep learning methods such as Artificial Neural Networks (ANN) have become foundational in computational chemistry as they can handle complex spatial and sequential data analysis. These neural networks allow for the model to identify and learn patterns when modeling molecular interactions without needing to program specific chemical properties.

"Neural network potentials for chemistry: concepts, applications and prospects" by Silvan Käser et al. (2022) provides a comprehensive overview of the use and application of Artificial Neural Networks in computational chemistry. The paper discusses the methods to use an ANN model for the potential energy surfaces (PES) which are significant for molecular simulations. It presents how to construct these models, optimize the training process, and further strengthen them using transfer learning to enhance the model's accuracy.

Another notable publication in machine learning and computational chemistry is "The Rise of Neural Networks for Materials and Chemical Dynamics," by Maksim Kulichenko et al. (2021). The paper expands beyond neural network models to examine the different advancements for modeling chemical processes and material dynamics, providing new advantages and challenges in developing accurate, transferable, and interpretable models. It highlights active learning, transfer learning, and other methods that improve performance when training large data sets on electron structures.

Both publications highlight the potential for neural networks to advance the understanding of chemical properties, improve computational overhead, and enhance prediction accuracy. These papers lay the foundation for the model developed in this project and the analysis of that model's performance.

In this project, a supervised learning Artificial Neural Network model was developed and applied to the ANI-1 dataset. This report will present the methodology used to develop this model and how the construction of that model and its hyperparameters impact the model's prediction performance. Detailed exploration of the model in this report will prompt a deeper understanding of neural networks and their implications in chemical simulations.

2. METHODOLOGY

2.1 Data Processing

The dataset utilized for this project was the ANI-1 dataset which provided the properties of organic molecules that are predominantly composed of carbon, hydrogen, nitrogen, and oxygen. The full ANI-1 dataset was used to train the ANN model with molecules with more than four heavy atoms. The dataset was split into four different ANI subsets (s01 to s04) for further analysis which allowed for examining molecules with less heavy atoms such as one heavy atom.

An Atomic Environment Vector (AEV) computer was initialized, using the "TorchANI" library, to calculate features within the chemical environment of the atoms in the molecule which are then used as inputs for the ANN model. The parameters of the AEV computer followed the recommendations of the literature (Chem. Sci., 2017, 8, 3192). After the initialization, the data was split into training, validation, and test sets with an 80%, 10%, and 10% split respectively. By splitting this dataset, there are 691918 training samples, 86489 validation samples, and 86489 testing samples. To further optimize efficiency and memory management, batching was used with a batch size of 8192.

2.2 Model

A neural network architecture was developed and named "AtomicNet" which was designed to use the atomic environments to predict molecular properties. Each predominant atom type (hydrogen, carbon, nitrogen, oxygen) is initialized as a separate instance of the AtomicNet class. The AEV computer generates a vector with size 384 as the input layer of this model. For the hidden layer, linear transformation reduces the dimensionality from 384 to 128 and then a ReLU activation function is used. The output layers reduced the dimensionality to 1 which should represent the chemical property of the atom.

2.3 Training Scheme and Hyperparameters

The training scheme of the ANN model was encapsulated by the “ANITrainer” class which trained and evaluated the performance of the model. The “ANITrainer” class used model, batch size, learning rate, epochs, and L2 regularization as important parameters to adjust the training of the model, and it used an Adam optimizer.

The “train” method in this class initializes the data loaders for the training and validation data. For each epoch (number specified in the parameters), the method processes the different batches to compute the predictions and the loss using Mean Squared Error (MSE) then performs backpropagation. To account for the difference in batch sizes, the loss of each batch is adjusted based on the batch importance. Lastly, to minimize overfitting, an early stop was implemented when the validation loss did not show improvement. The processes within this method are timed using the “timeit” decorator from the “time” package to record the duration of the training session which will be used to compare the performance of different hyperparameters and changes in architecture.

The “evaluate” method in this class assesses the performance of the ANN model and its training abilities by predicting the energies and comparing them to the actual values. The species, coordinates, and true energy values are extracted from the batched data. The species and coordinates are used to predict the energies by passing them through the ANN model. The loss is calculated by computing the difference between the predicted and true energy values scaled by the batch importance. To present the loss, Mean Absolute Error (MAE) and Root Mean Square Error (RMSE) are calculated and plotted with a hartree2kcalmol conversion factor.

2.4 List of Public Packages

1. numpy
2. tqdm
3. torch
4. torch.nn as nn
5. torchani
6. DataLoader from torch.utils.data

7. matplotlib.pyplot
8. time
9. warnings

3. SUMMARY OF RESULTS

To analyze the functionality of the model, the ANN model was first trained on the subsets of the data. The model was trained on a dataset containing one heavy atom (s01) and then a dataset containing four heavy atoms (s04). This demonstrated the ability of the model to train and predict different molecular structures. Baseline hyperparameters of 8192 batch size, 1e-3 learning rate, 100 epochs, and 1e-5 L2 regularization were used for all runs. It was evident that on runs with the GPU and CPU, the model was able to achieve acceptable loss however the GPU performed better in run time than the CPU. This was especially evident for the four heavy atoms where the GPU performs training almost 39 times faster than the CPU. The performance results for the one heavy atom and four heavy atoms are listed in Table 1.

Table #1: GPU Versus CPU Performance on ANN Model

Number of Heavy Atoms	Processing Unit	Run Time (seconds)	Test MAE (kcal/mol)	Test RMSE (kcal/mol)
1 Heavy Atom	CPU	35.2741 seconds	1.97 kcal/mol	2.88 kcal/mol
1 Heavy Atom	GPU	19.9933 seconds	1.69 kcal/mol	2.18 kcal/mol
4 Heavy Atoms	CPU	6114.0713 seconds	1.75 kcal/mol	2.91 kcal/mol
4 Heavy Atoms	GPU	157.0019 seconds	1.92 kcal/mol	2.87 kcal/mol

The ANN model was then trained on the full ANI-1 dataset to train on at least four heavy atoms. The model was initialized with the baseline hyperparameters of 8192 batch size, 1e-3 learning rate, 50 epochs, and 1e-5 L2 regularization. Different combinations of hyperparameters were changed to find the optimal combination that would reduce the MAE below 2 kcal/mol and an RMSE below 3 kcal/mol while maintaining a reasonably low runtime (under 1548 seconds). It was evident that 10000 batch size, 1e-3 learning rate, 50 epochs, and 1e-7 L2 regularization were the optimal hyperparameters for this model because of its ability to prevent underfitting. These hyperparameters achieved an MAE of 1.56 kcal/mol and an RMSE of 2.21 kcal/mol in 511.1501 seconds. This provided a good balance between performance and minimizing run time overhead. These same parameters except with 100 epochs had a lower MAE of 1.18 kcal/mol and RMSE of 1.68 kcal/mol. However, the run time was twice as long, taking the model 984.6864 seconds. Therefore, 50 epochs would be more effective. The difference in performance between different hyperparameters can be seen in Table 2 which presents how these changes impacted the performance results. Figures of the learning curve and linear fit for the optimal hyperparameters (10000 batch size, 1e-3 learning rate, 50 epochs, and 1e-7 L2 regularization) are presented in Figures 1 and 2.

Table #2: Model MAE Using Different Hyperparameters

Hidden Layers	Batch Size	Learning Rate	Epochs	L2 Regularization	Run Time (Seconds)	Test MAE (kcal/mol)	Test RMSE (kcal/mol)
[384, 128, 1]	8192	1e-3	50	1e-5	519.7056 seconds	3.01 kcal/mol	4.58 kcal/mol
[384, 128, 1]	8192	1e-3	100	1e-5	1010.4456 seconds	2.35 kcal/mol	4.35 kcal/mol
[384, 128, 1]	10000	1e-3	100	1e-5	989.1252 seconds	2.08 kcal/mol	3.96 kcal/mol
[384, 128, 1]	10000	1e-5	100	1e-5	968.7207 seconds	3.36 kcal/mol	6.87 kcal/mol
[384, 128, 1]	10000	1e-3	100	1e-7	984.6854 seconds	1.18 kcal/mol	1.68 kcal/mol
[384, 128, 1]	10000	1e-3	50	1e-7	511.1501 seconds	1.73 kcal/mol	2.26 kcal/mol

Figure 1: Learning Curve of Optimal Model

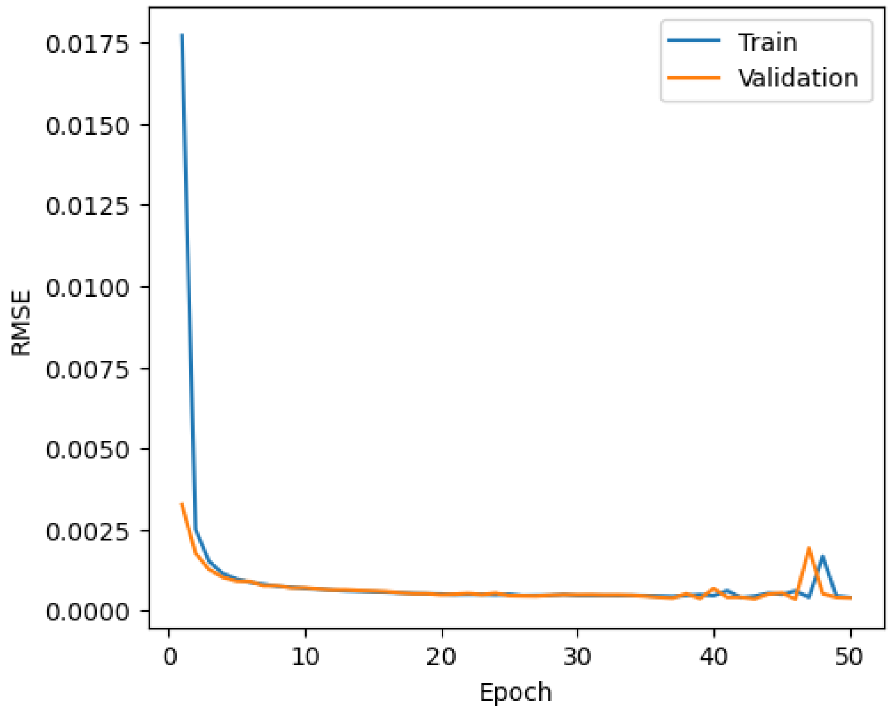
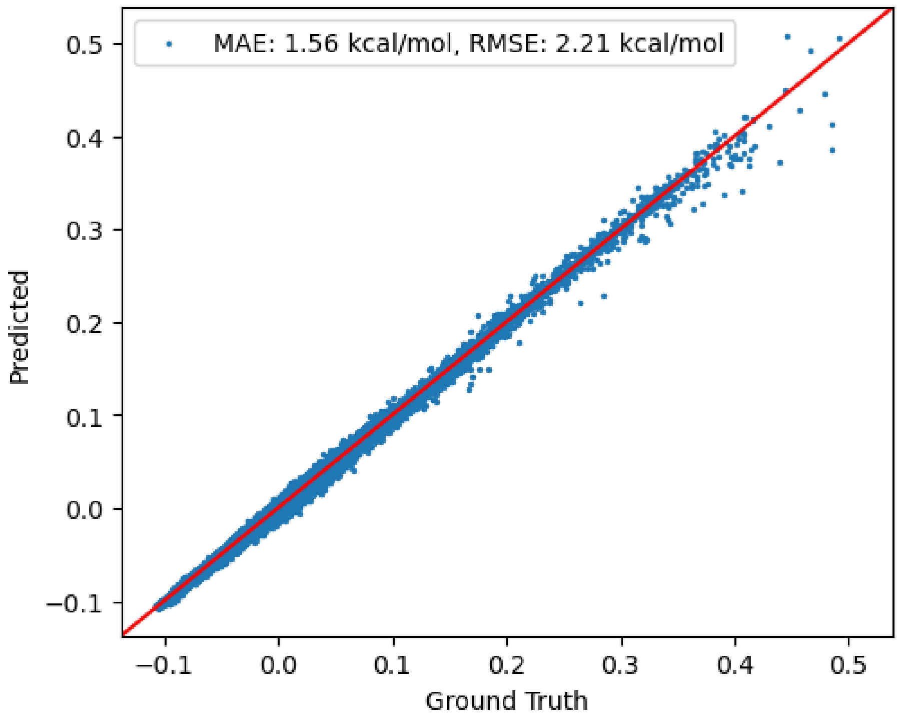


Figure 2: Linear Fit Line of Predicted Versus Ground Truth Values



4. DISCUSSION

The implementation of the Artificial Neural Network (ANN) model for the ANI-1 dataset in this project emphasizes the significance and potential of deep learning techniques in computational chemistry. The ANN model developed in this project was designed to predict molecules energies based on different atomic environments using calculations from the initialized Atomic Environment Vector computer. The results of this model were promising as the model was able to achieve a lower Mean Absolute Error and Root Mean Square Error than the established threshold of the 2 kcal/mol and 3 kcal/mol respectively. This low error displays the model's ability to predict across different molecular structures and sizes, proving its reliability for energy predictions. This is essential in real-world applications such as drug discovery because predictions need to be precise as small changes can have significant impacts on the outcomes of drug research.

In terms of training, the techniques proved to be effective because of the low error values. The use of the Adam optimizer, higher batch size, and lower L2 regularization value were effective in minimizing the underfitting and reducing the influence of noise present in the baseline hyperparameters. The batching techniques in the training were especially effective in improving the performance and computation efficiency without increasing the run time overhead.

The current ANN model provides a strong foundation for exploring the applications of machine learning models in computational chemistry. A future extension to this model would be to modify it to handle other types of molecular data to improve the performance in predicting molecular energies. In addition, it would be worthwhile to examine different neural network architectures to compare how the different models, such as Convolutional Neural Networks (CNN) and Graph Neural Networks (GNN) can predict energies while capturing more complex relationships in the data. The project displayed the effectiveness of ANN models in computation chemistry and exemplifies the continually advancing field of machine learning models.

5. REFERENCES

1. Käser, Silvan, Luis Itza Vazquez-Salazar, Markus Meuwly, and Kai Töpfer. "Neural Network Potentials for Chemistry: Concepts, Applications and Prospects." *Department of Chemistry, University of Basel*. First published 21 Dec. 2022.

2. Kulichenko, Maksim, Justin S. Smith, Benjamin Nebgen, Ying Wai Li, Nikita Fedik, Alexander I. Boldyrev, Nicholas Lubbers, Kipton Barros, and Sergei Tretiak. "The Rise of Neural Networks for Materials and Chemical Dynamics." *Journal of Physical Chemistry Letters*, 2021, vol. 12, no. 26, pp. 6227–6243. https://doi.org/10.1021/acs.jpclett.1c01357.

CHEM 277B Final Project Spring 2024

By Jedrick Regala Zablan

In [1]:
```
!pwd
```

```
/global/scratch/users/jedrickzablan/jedrickzablan/final
```

In [2]:
```
import warnings
warnings.filterwarnings("ignore", category=UserWarning)
import numpy as np
from tqdm import tqdm
import torch
import torch.nn as nn
import torchani
from torch.utils.data import DataLoader
import matplotlib.pyplot as plt
import time
```

Use GPU

In [3]:
```
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
print(device)
```

```
cpu
```

Data Processing

AEV Computer Initialization

AEV: Atomic Environment Vector (atomic features)

Ref: Chem. Sci., 2017, 8, 3192

In [4]:
```
def init_aev_computer():
    Rcr = 5.2
    Rca = 3.5
    EtaR = torch.tensor([16], dtype=torch.float, device=device)
    ShfR = torch.tensor([
        0.900000, 1.168750, 1.437500, 1.706250,
        1.975000, 2.243750, 2.512500, 2.781250,
        3.050000, 3.318750, 3.587500, 3.856250,
        4.125000, 4.393750, 4.662500, 4.931250
    ], dtype=torch.float, device=device)


    EtaA = torch.tensor([8], dtype=torch.float, device=device)
    Zeta = torch.tensor([32], dtype=torch.float, device=device)
    ShfA = torch.tensor([0.90, 1.55, 2.20, 2.85], dtype=torch.float, device=device)
    ShfZ = torch.tensor([
        0.19634954, 0.58904862, 0.9817477, 1.37444680,
        1.76714590, 2.15984490, 2.5525440, 2.94524300
    ], dtype=torch.float, device=device)

    num_species = 4
    aev_computer = torchani.AEVComputer(
        Rcr, Rca, EtaR, ShfR, EtaA, Zeta, ShfA, ShfZ, num_species
    )
    return aev_computer

aev_computer = init_aev_computer()
aev_dim = aev_computer.aev_length
print(aev_dim)
```

```
384
```

Prepare dataset & split

In [5]:
```
def load_ani_dataset(dspath):
    self_energies = torch.tensor([
        0.500607632585, -37.8302333826,
        -54.5680045287, -75.0362229210
    ], dtype=torch.float, device=device)
    energy_shifter = torchani.utils.EnergyShifter(None)
    species_order = ['H', 'C', 'N', 'O']

    dataset = torchani.data.load(dspath)
    dataset = dataset.subtract_self_energies(energy_shifter, species_order)
    dataset = dataset.species_to_indices(species_order)
    dataset = dataset.shuffle()
    return dataset

dataset = load_ani_dataset("./ani_gdb_s01_to_s04.h5")
```

In [6]:
```
# Use dataset.split method to do split
train_data, val_data, test_data = dataset.split(0.8, 0.1, 0.1)

# Show amount of training data vs total data
print(f'Total data amount: {len(dataset)}')
print(f'Training data amount: {len(train_data)}')
print(f'Validation data amount: {len(val_data)}')
print(f'Testing data amount: {len(test_data)}')
```

```
Total data amount: 864898
Training data amount: 691918
Validation data amount: 86489
Testing data amount: 86489
```

Batching

In [7]:
```
batch_size = 8192
train_data_loader = train_data.collate(batch_size).cache()
val_data_loader = val_data.collate(batch_size).cache()
test_data_loader = test_data.collate(batch_size).cache()
```

In [8]:
```
class AtomicNet(nn.Module):
    def __init__(self):
        super().__init__()
        self.layers = nn.Sequential(
            nn.Linear(384, 128),
            nn.ReLU(),
            nn.Linear(128, 1)
        )

    def forward(self, x):
        return self.layers(x)

net_H = AtomicNet()
net_C = AtomicNet()
net_N = AtomicNet()
net_O = AtomicNet()

# ANI model requires a network for each atom type
# use torchani.ANIModel() to compile atomic networks
atom_net = [net_H, net_C, net_N, net_O]

ani_net = torchani.ANIModel(atom_net)
model = nn.Sequential(
    aev_computer,
    ani_net
).to(device)
```

In [9]:
```
train_data_batch = next(iter(train_data_loader))

loss_func = nn.MSELoss()
species = train_data_batch['species'].to(device)
coords = train_data_batch['coordinates'].to(device)
true_energies = train_data_batch['energies'].to(device).float()
_, pred_energies = model((species, coords))
loss = loss_func(true_energies, pred_energies)
print(loss)
```

```
tensor(0.4350, grad_fn=<MseLossBackward0>)
```

In [10]:
```
val_data_batch = next(iter(val_data_loader))

loss_func = nn.MSELoss()
species = val_data_batch['species'].to(device)
coords = val_data_batch['coordinates'].to(device)
true_energies = val_data_batch['energies'].to(device).float()
_, pred_energies = model((species, coords))
loss = loss_func(true_energies, pred_energies)
print(loss)
```

```
tensor(0.4393, grad_fn=<MseLossBackward0>)
```

In [11]:
```
test_data_batch = next(iter(test_data_loader))

loss_func = nn.MSELoss()
species = test_data_batch['species'].to(device)
coords = test_data_batch['coordinates'].to(device)
true_energies = test_data_batch['energies'].to(device).float()
_, pred_energies = model((species, coords))
loss = loss_func(true_energies, pred_energies)
print(loss)
```

```
tensor(0.4400, grad_fn=<MseLossBackward0>)
```

AtomicNet Model Using Torchani

In [12]:
```
class AtomicNet(nn.Module):
    def __init__(self):
        super().__init__()
        self.layers = nn.Sequential(
            nn.Linear(384, 128),
            nn.ReLU(),
            nn.Linear(128, 1)
        )

    def forward(self, x):
        return self.layers(x)

net_H = AtomicNet()
net_C = AtomicNet()
net_N = AtomicNet()
net_O = AtomicNet()

# ANI model requires a network for each atom type
# use torchani.ANIModel() to compile atomic networks
atom_net = [net_H, net_C, net_N, net_O]

ani_net = torchani.ANIModel(atom_net)
model = nn.Sequential(
    aev_computer,
    ani_net
).to(device)
```

In [13]:
```
model
```

Out[13]:
```
Sequential(
  (0): AEVComputer()
  (1): ANIModel(
    (0): AtomicNet(
      (layers): Sequential(
        (0): Linear(in_features=384, out_features=128, bias=True)
        (1): ReLU()
        (2): Linear(in_features=128, out_features=1, bias=True)
      )
    )
    (1): AtomicNet(
      (layers): Sequential(
        (0): Linear(in_features=384, out_features=128, bias=True)
        (1): ReLU()
        (2): Linear(in_features=128, out_features=1, bias=True)
      )
    )
    (2): AtomicNet(
      (layers): Sequential(
        (0): Linear(in_features=384, out_features=128, bias=True)
        (1): ReLU()
        (2): Linear(in_features=128, out_features=1, bias=True)
      )
    )
    (3): AtomicNet(
      (layers): Sequential(
        (0): Linear(in_features=384, out_features=128, bias=True)
        (1): ReLU()
        (2): Linear(in_features=128, out_features=1, bias=True)
      )
    )
  )
)
```

ANITrainer Class Definition with Timed Training and Evaluation

In [14]:
```python
def timeit(f):
    def timed(*args, **kw):
        ts = time.time()
        result = f(*args, **kw)
        te = time.time()
        print(f'func: {f.__name__} took: {te-ts:.4f} sec on {device}')
        return result
    return timed

class ANITrainer:
    def __init__(self, model, batch_size, learning_rate, epoch, l2):
        self.model = model

        num_params = sum(item.numel() for item in model.parameters())
        print(f"{model.__class__.__name__} - Number of parameters: {num_params}")

        self.batch_size = batch_size
        self.optimizer = torch.optim.Adam(model.parameters(), learning_rate, weight_de
        self.epoch = epoch
        self.loss_function = nn.MSELoss()
    @timeit
    def train(self, train_data, val_data,
              early_stop=True, draw_curve=True, verbose=True):
        self.model.train()

        # init data loader
        print("Initialize training data...")
        train_data_loader = train_data.collate(batch_size).cache()
        val_data_loader = val_data.collate(batch_size).cache()

        # record epoch losses
        train_loss_list = []
        val_loss_list = []
        lowest_val_loss = np.inf

        if verbose:
            iterator = range(self.epoch)
        else:
            iterator = tqdm(range(self.epoch), leave=True)

        for i in iterator:
            train_epoch_loss = 0.0
            # compute energies
            for train_data_batch in train_data_loader:
                species = train_data_batch['species'].to(device)
                coordinates = train_data_batch['coordinates'].to(device)
                true_energies = train_data_batch['energies'].to(device).float()

                batch_importance = species.shape[0] / len(val_data)

                self.optimizer.zero_grad()
                _, pred_energies = self.model((species, coordinates))

            # compute loss
                loss = self.loss_function(pred_energies, true_energies) * batch_import
                loss.backward()

            # do a step
                self.optimizer.step()

                train_epoch_loss += loss.item() * batch_importance

            val_epoch_loss, _, _ = self.evaluate(val_data, draw_plot=False)
            train_loss_list.append(train_epoch_loss / len(train_data_loader))
            val_loss_list.append(val_epoch_loss / len(val_data_loader))

            if early_stop and val_epoch_loss < lowest_val_loss:
                lowest_val_loss = val_epoch_loss
                weights = self.model.state_dict()

        if draw_curve:
            # Plot train loss and validation loss
            fig, ax = plt.subplots(1, 1, figsize=(5, 4), constrained_layout=True)
            # If you used MSELoss above to compute the loss
            # Calculate the RMSE for plotting
            epochs = np.arange(1, self.epoch + 1)
            train_rmse = [np.sqrt(loss) for loss in train_loss_list]
            val_rmse = [np.sqrt(loss) for loss in val_loss_list]

            ax.plot(epochs, train_rmse, label='Train')
            ax.plot(epochs, val_rmse, label='Validation')
            ax.legend()
            ax.set_xlabel("Epoch")
            ax.set_ylabel("RMSE")

        if early_stop:
            self.model.load_state_dict(weights)

        return train_loss_list, val_loss_list


    def evaluate(self, data, draw_plot=False):

        # init data loader
        data_loader = data.collate(batch_size).cache()
        total_loss = 0.0

        # init energies containers
        true_energies_all = []
        pred_energies_all = []

        with torch.no_grad():
            for batch_data in data_loader:
                species = batch_data['species'].to(device)
                coordinates = batch_data['coordinates'].to(device)
                true_energies = batch_data['energies'].to(device).float()

                batch_importance = species.shape[0] / len(data)

                # Predict energies using the model
                _, pred_energies = self.model((species, coordinates))

                # Compute loss
                loss = self.loss_function(pred_energies, true_energies) * batch_import
                total_loss += loss.item() * batch_importance

                # store true and predicted energies
                true_energies_all.append(true_energies.detach().cpu().numpy().flatten(
                pred_energies_all.append(pred_energies.detach().cpu().numpy().flatten(
        true_energies_all = np.concatenate(true_energies_all)
        pred_energies_all = np.concatenate(pred_energies_all)

        # Report the mean absolute error (MAE) and root mean square error (RMSE)
        # The unit of energies in the dataset is hartree
        # please convert it to kcal/mol when reporting
        # 1 hartree = 627.5094738898777 kcal/mol
        # MAE = mean(|true - pred|)
        # RMSE = sqrt(mean( (true-pred)^2 ))
        hartree2kcalmol = 627.5094738898777
        mae = np.mean(np.abs(true_energies_all - pred_energies_all)) * hartree2kcalmol
        rmse = np.sqrt(np.mean((true_energies_all - pred_energies_all) ** 2)) * hartre

        if draw_plot:
            fig, ax = plt.subplots(1, 1, figsize=(5, 4), constrained_layout=True)
            ax.scatter(true_energies_all, pred_energies_all, label=f"MAE: {mae:.2f} kc
            ax.set_xlabel("Ground Truth")
            ax.set_ylabel("Predicted")
            xmin, xmax = ax.get_xlim()
            ymin, ymax = ax.get_ylim()
            vmin, vmax = min(xmin, ymin), max(xmax, ymax)
            ax.set_xlim(vmin, vmax)
            ax.set_ylim(vmin, vmax)
            ax.plot([vmin, vmax], [vmin, vmax], color='red')
            ax.legend()

        return total_loss, mae, rmse
```

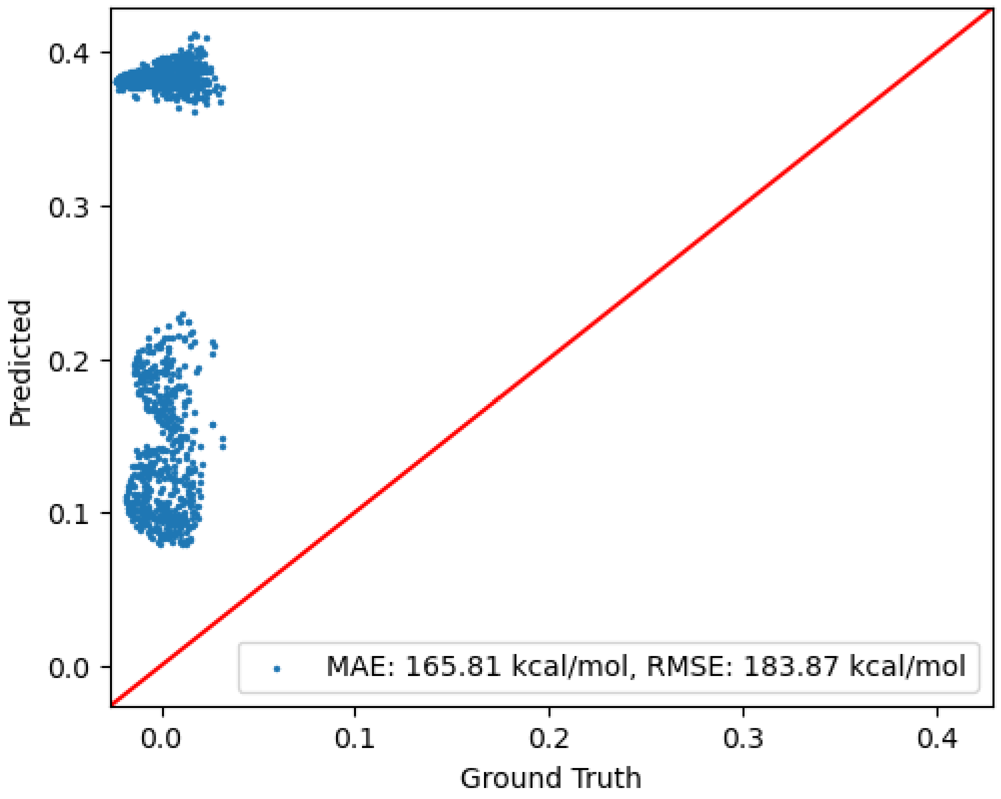
Training and Outputs for Different Ani Datasets

1 Heavy Atom Model

In [19]:

```
batch_size = 8192
epochs_count = 100

# Load dataset with 1 heavy atom
# Then do a train/val/test = 80/10/10 split
dataset_heavy = load_ani_dataset("./ani_gdb_s01.h5")
train_data, val_data, test_data = dataset_heavy.split(0.8, 0.1, 0.1)

print(f'Train/Total: {len(train_data)}/{len(dataset)}')

# Define the model
model = nn.Sequential(
    aev_computer,
    ani_net
).to(device)

# Initiate the trainer and evaluate on test_dataset with draw_plot=True
trainer = ANITrainer(model, batch_size, 1e-3, epochs_count, 1e-5)
loss, mae, rmse = trainer.evaluate(test_data, draw_plot=True)
```

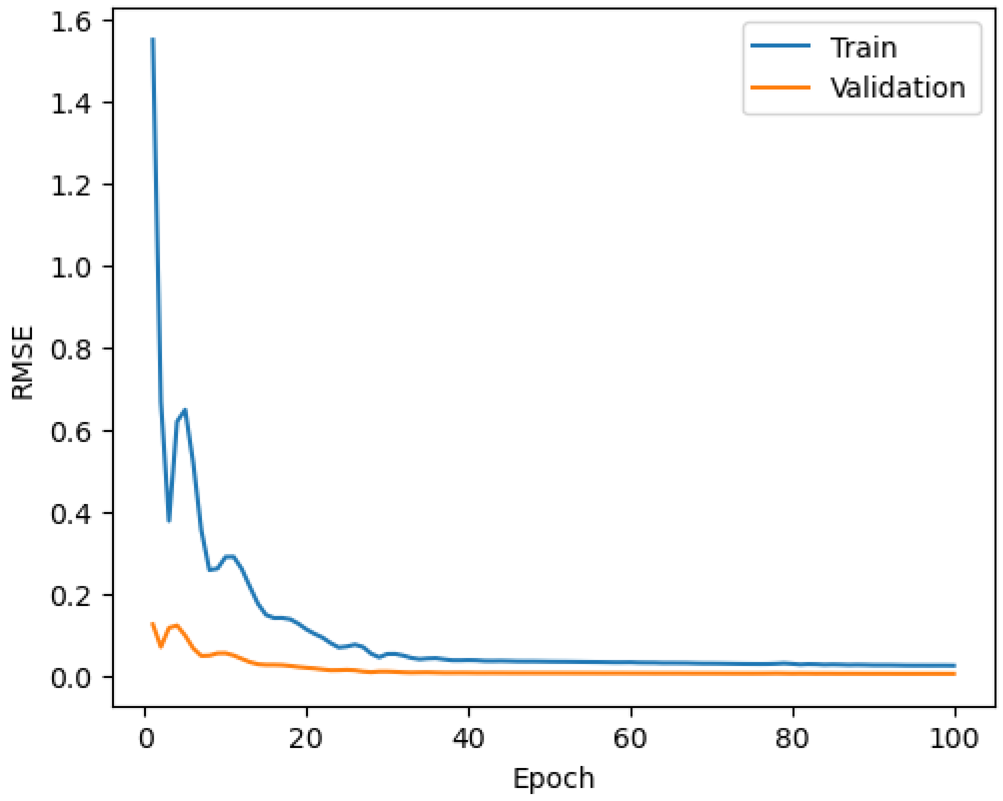
```
Train/Total: 8640/864898
Sequential - Number of parameters: 197636
```

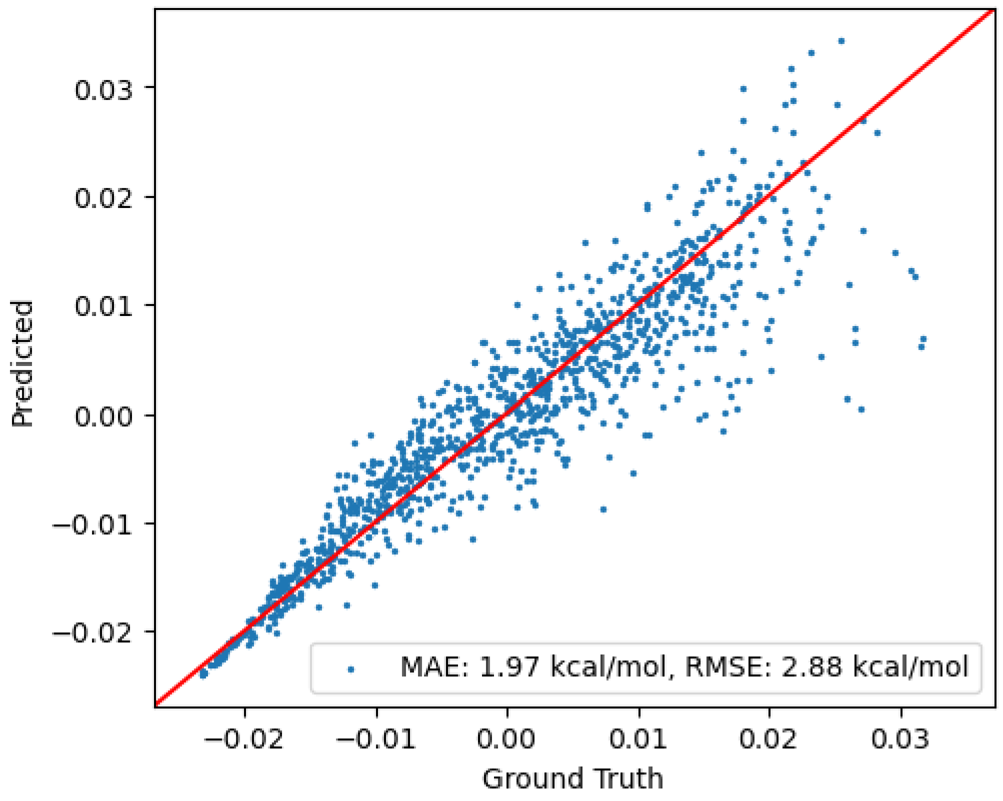


1 Heavy Atom Training, Output Visualization and Results

In [20]:
```
#CPU
train_losses, val_losses = trainer.train(train_data, val_data, verbose=True)
loss, mae, rmse = trainer.evaluate(test_data, draw_plot=True)
```

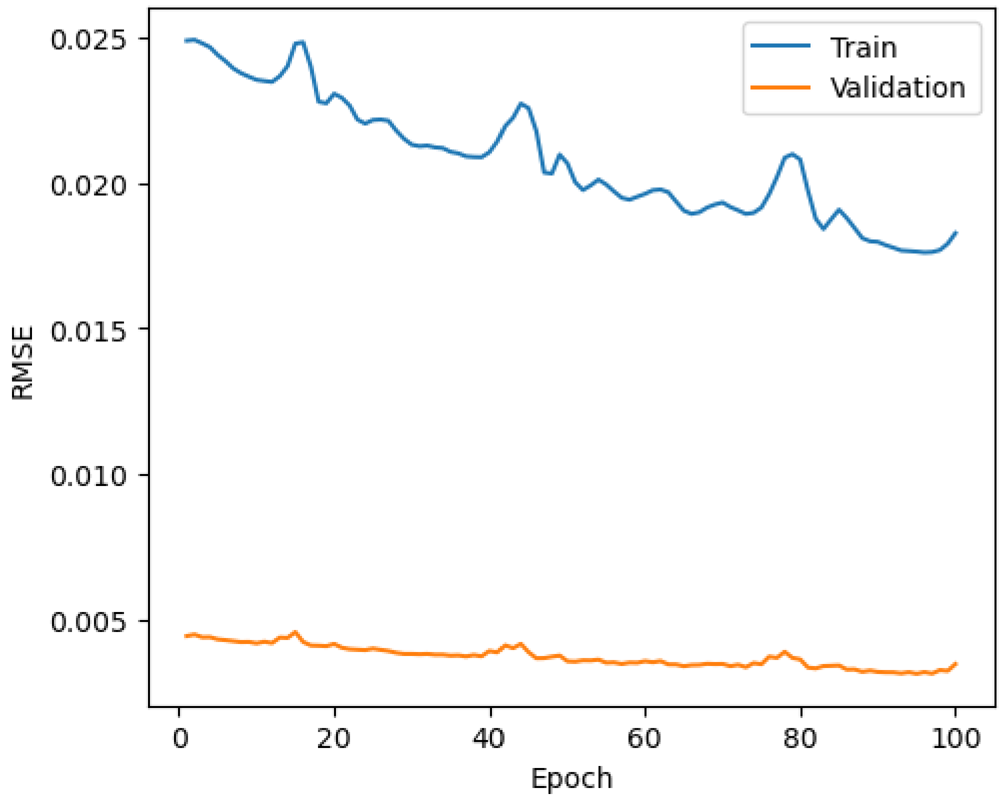
```
Initialize training data...
func: train took: 35.2741 sec on cpu
```

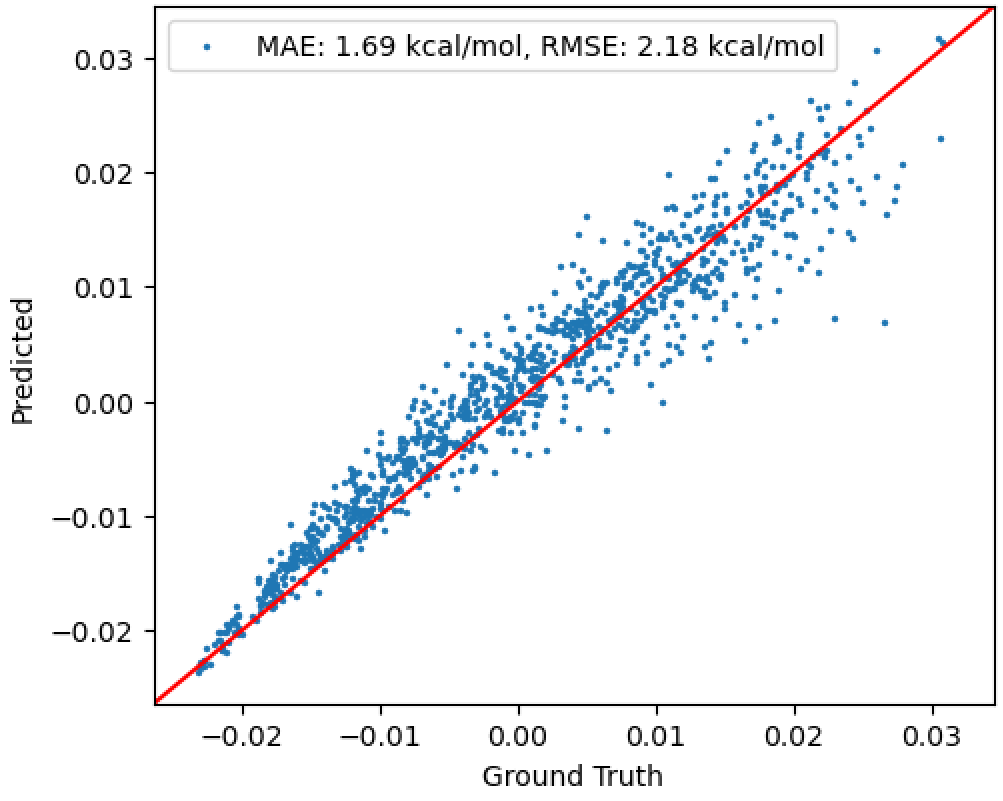




In [17]:
```
#GPU
train_losses, val_losses = trainer.train(train_data, val_data, verbose=True)
loss, mae, rmse = trainer.evaluate(test_data, draw_plot=True)
```

```
Initialize training data...
func: train took: 19.9933 sec on cuda
```

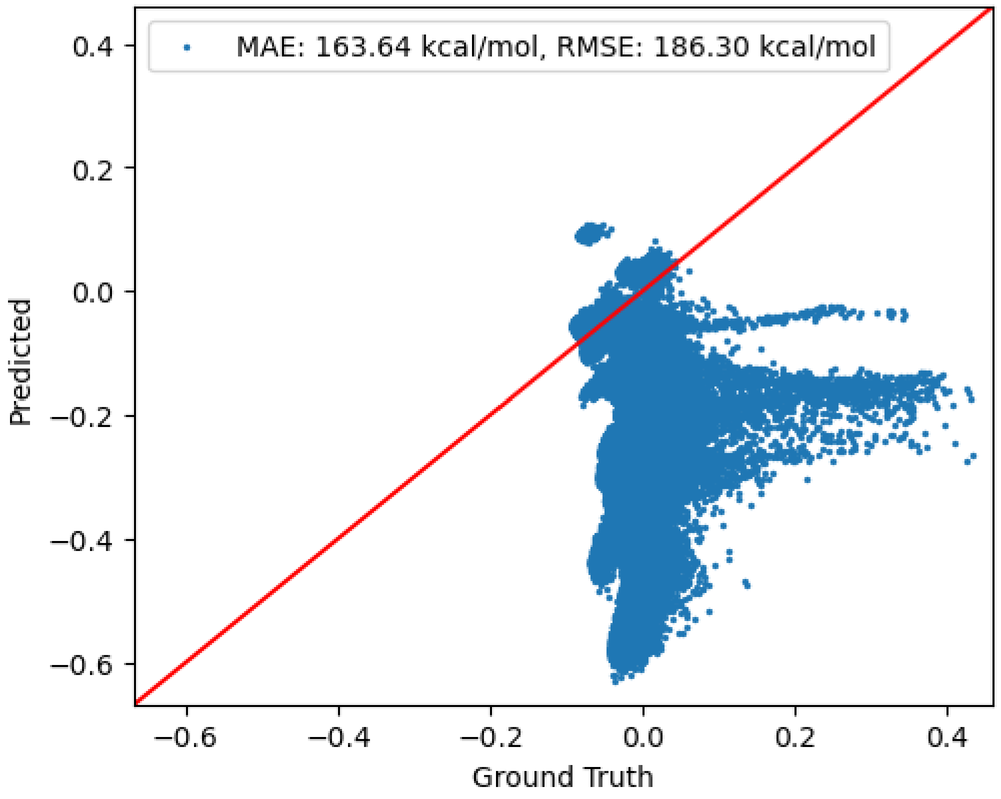




N > 1 Heavy Atoms Model

```
In [15]: batch_size = 8192
         epochs_count = 100

         # Load dataset with n (different from 1) heavy atom
         # Then do a train/val/test = 80/10/10 split
         dataset_heavy = load_ani_dataset("./ani_gdb_s04.h5")
         train_data, val_data, test_data = dataset_heavy.split(0.8, 0.1, 0.1)
         print(f'Train/Total: {len(train_data)}/{len(dataset)}')


         print(f'Train/Total: {len(train_data)}/{len(dataset)}')

         # Define the model
         model = nn.Sequential(
             aev_computer,
             ani_net
         ).to(device)

         # Initiate the trainer and evaluate on test_dataset with draw_plot=True
         trainer = ANITrainer(model, batch_size, 1e-3, epochs_count, 1e-5)
         loss, mae, rmse = trainer.evaluate(test_data, draw_plot=True)
```

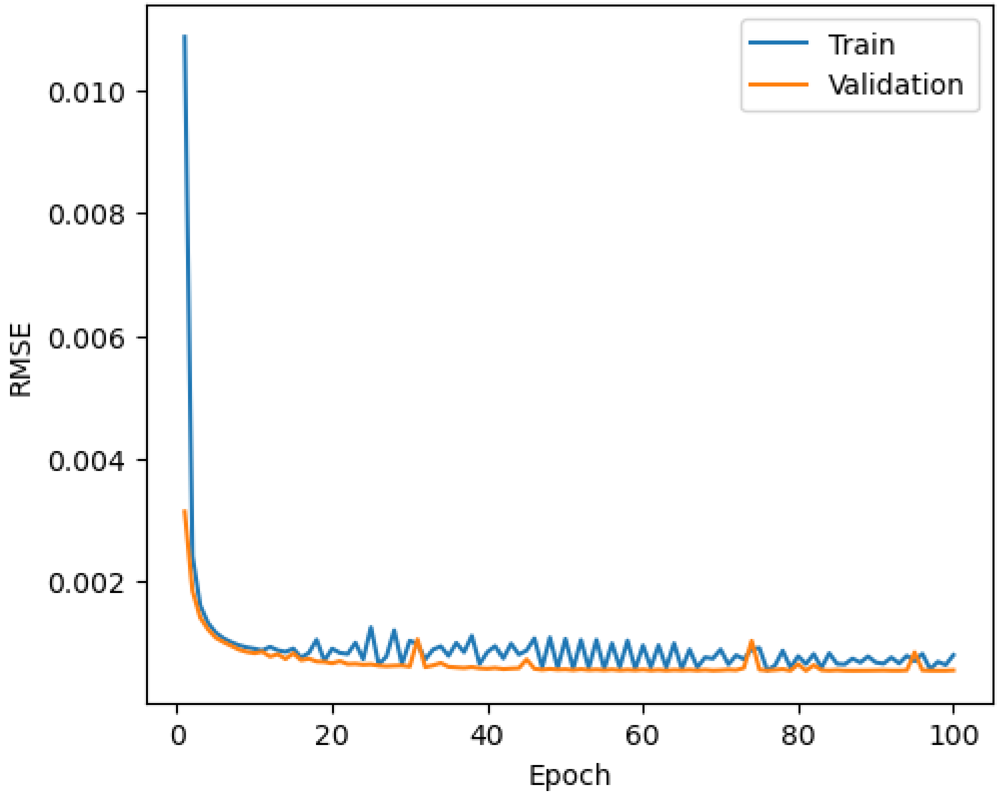
```
Train/Total: 521548/864898
Train/Total: 521548/864898
Sequential - Number of parameters: 197636
```

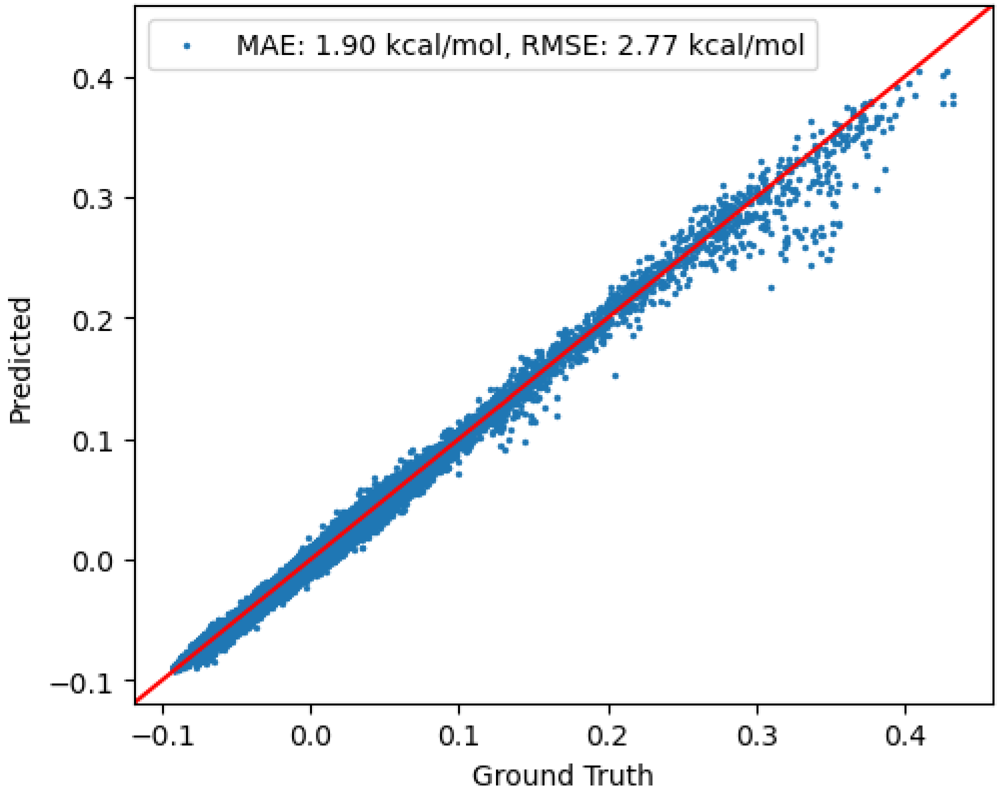


N > 1 Heavy Atoms Training, Output Visualization and Results

In [16]:
```
#CPU
train_losses, val_losses = trainer.train(train_data, val_data, verbose=True)
loss, mae, rmse = trainer.evaluate(test_data, draw_plot=True)
```

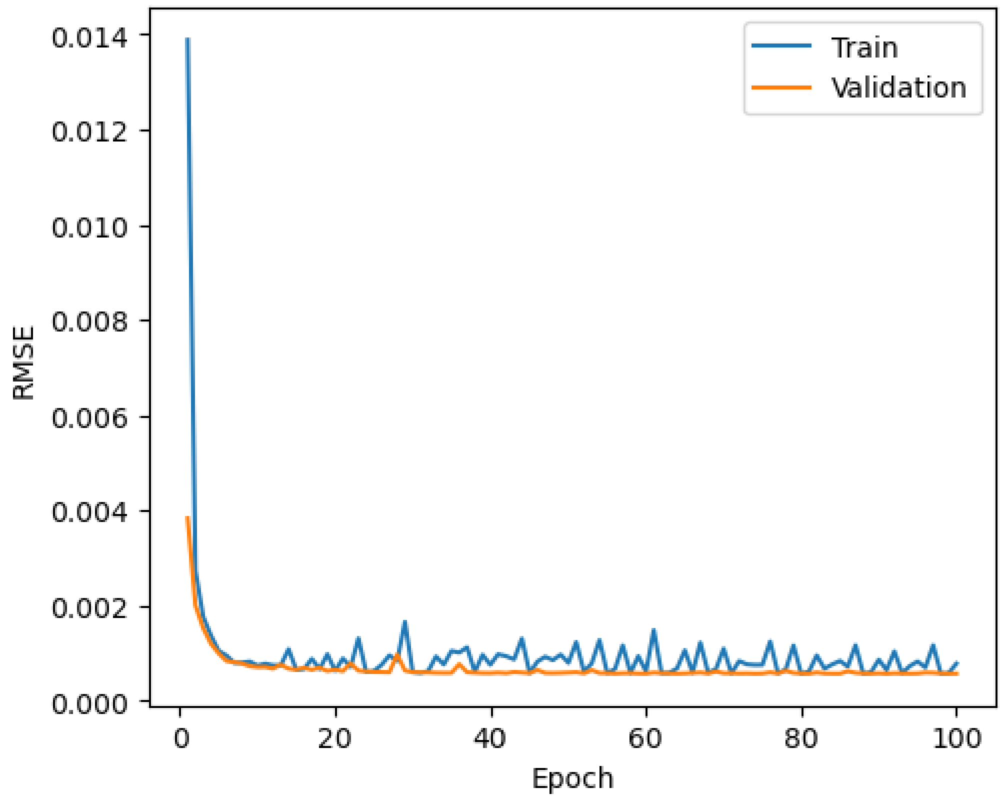
```
Initialize training data...
func: train took: 6114.0713 sec on cpu
```

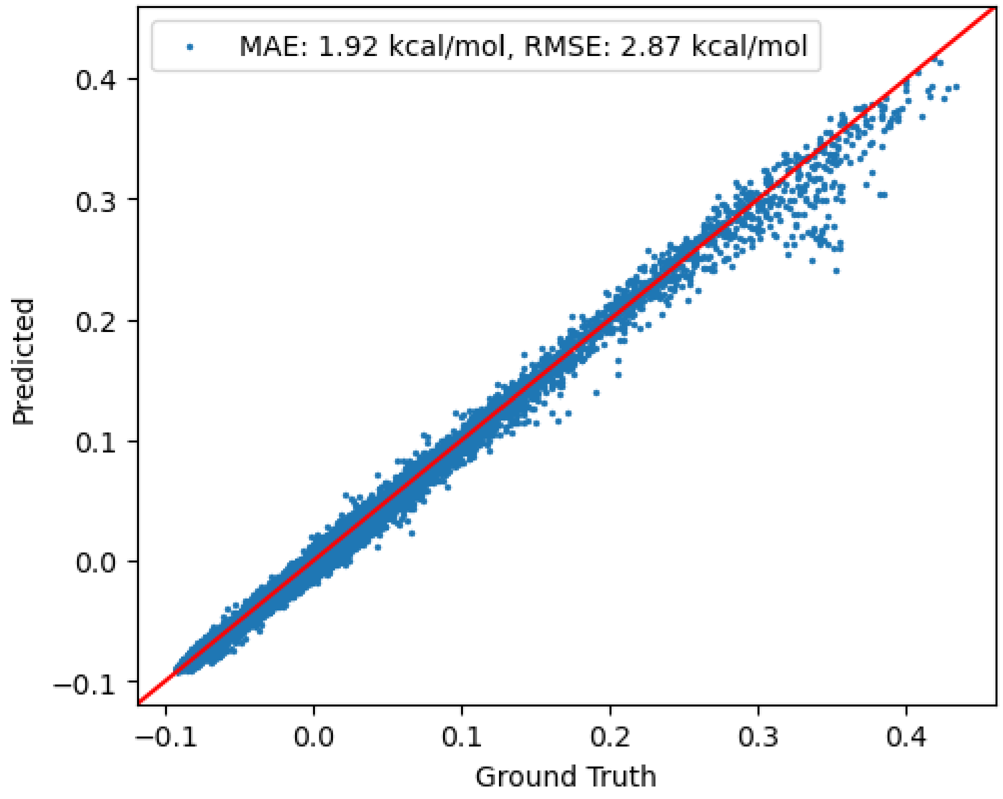




In [19]:
```
#GPU
train_losses, val_losses = trainer.train(train_data, val_data, verbose=True)
loss, mae, rmse = trainer.evaluate(test_data, draw_plot=True)
```

```
Initialize training data...
func: train took: 826.5780 sec on cuda
```





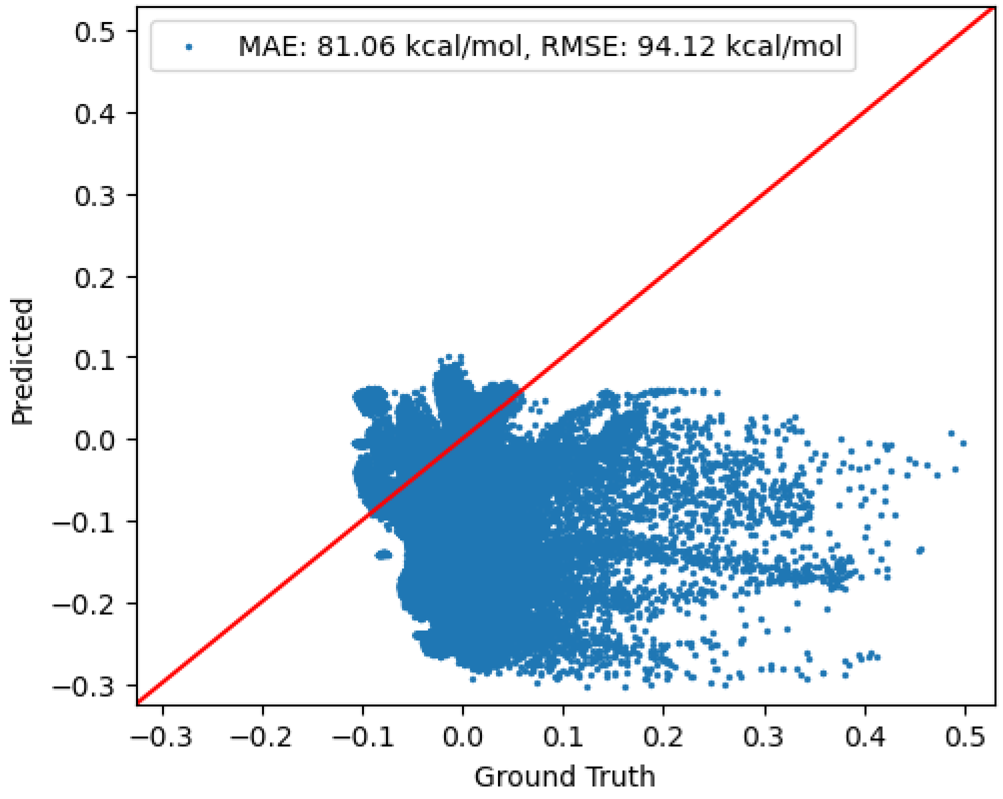
Assessment of GPU Performance versus CPU Performance

The use of the GPU improves performance for 1 heavy atom and multiple heavy atoms. For 1 heavy atom, the GPU trained on the model in 19.9933 seconds while the CPU trained on the model in 35.2741 seconds. For multiple heavy atoms, the GPU trained on the model in 157.0019 seconds while the CPU trained on the model in 6114.0713 seconds. All runs had a RMSE that was less than 3 kcal/mol and an MAE less than 2 kcal/mol.

Full Ani Dataset Model

In [24]:
```
# Then do a train/val/test = 80/10/10 split
dataset = load_ani_dataset("./ani_gdb_s01_to_s04.h5")
train_data, val_data, test_data = dataset.split(0.8,0.1,None)
print(f'Train/Total: {len(train_data)}/{len(dataset)}')

# Define the model
net_H = AtomicNet()
net_C = AtomicNet()
net_N = AtomicNet()
net_O = AtomicNet()
ani_net = torchani.ANIModel([net_H, net_C, net_N, net_O])
model = nn.Sequential(
    aev_computer,
    ani_net
).to(device)

batch_size = 10000
epochs_count = 50

trainer = ANITrainer(model, batch_size, 1e-3, epochs_count, 1e-7)
loss, mae, rmse = trainer.evaluate(test_data, draw_plot=True)
```

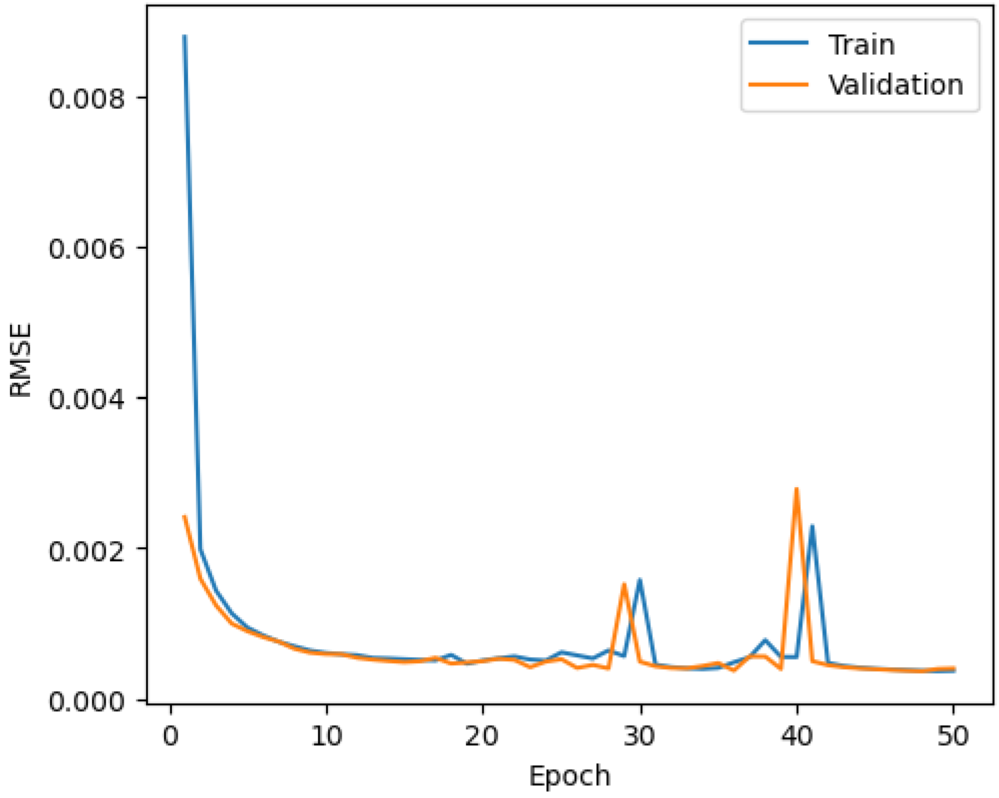
```
Train/Total: 691918/864898
Sequential - Number of parameters: 197636
```

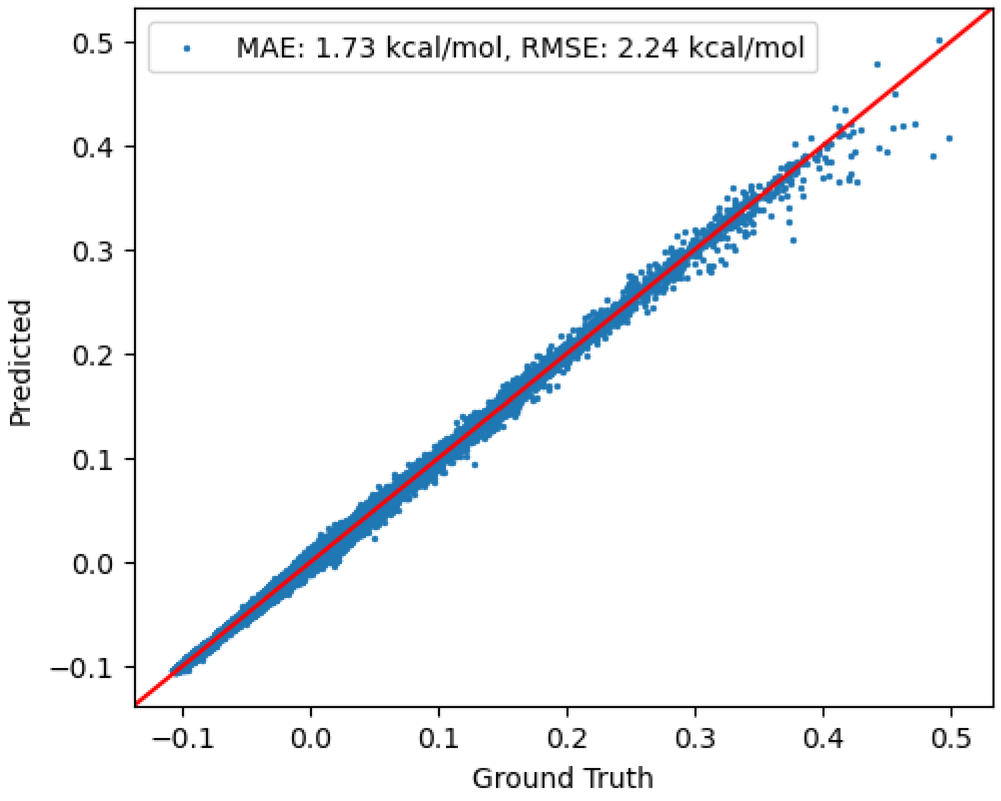


Full Ani Dataset Train

```
In [25]: #GPU
         train_losses, val_losses = trainer.train(train_data, val_data, verbose=False)
         loss, mae, rmse = trainer.evaluate(test_data, draw_plot=True)
```

```
Initialize training data...
100%|██████████| 50/50 [08:07<00:00,  9.75s/it]
func: train took: 511.1501 sec on cuda
```

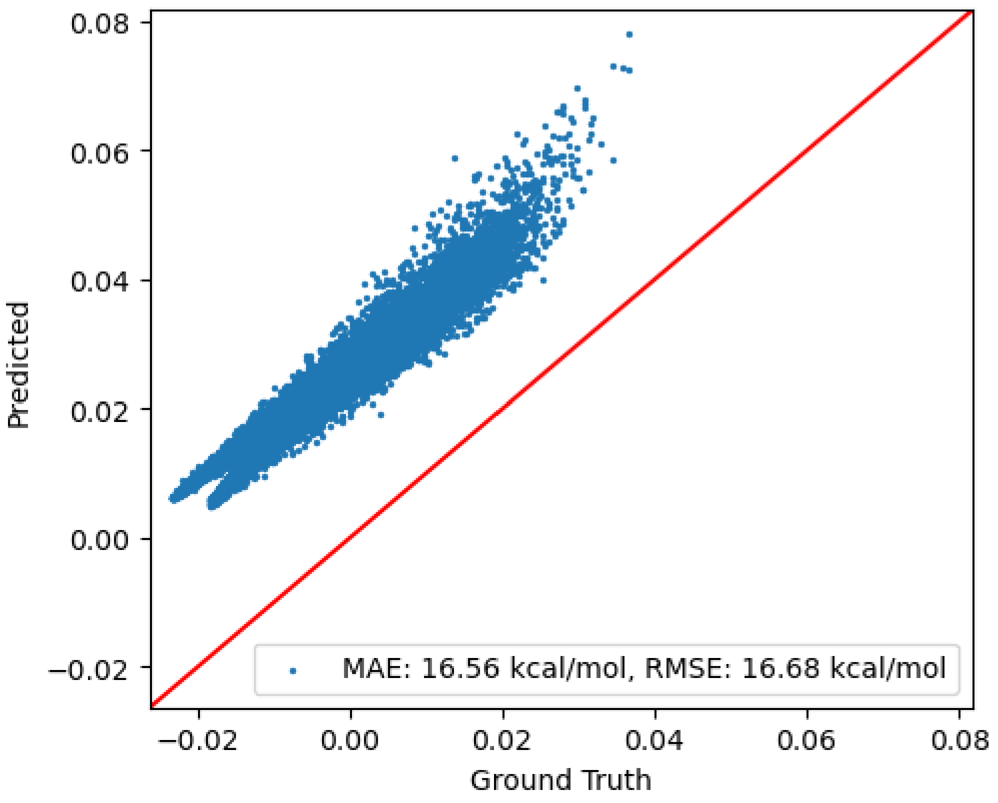




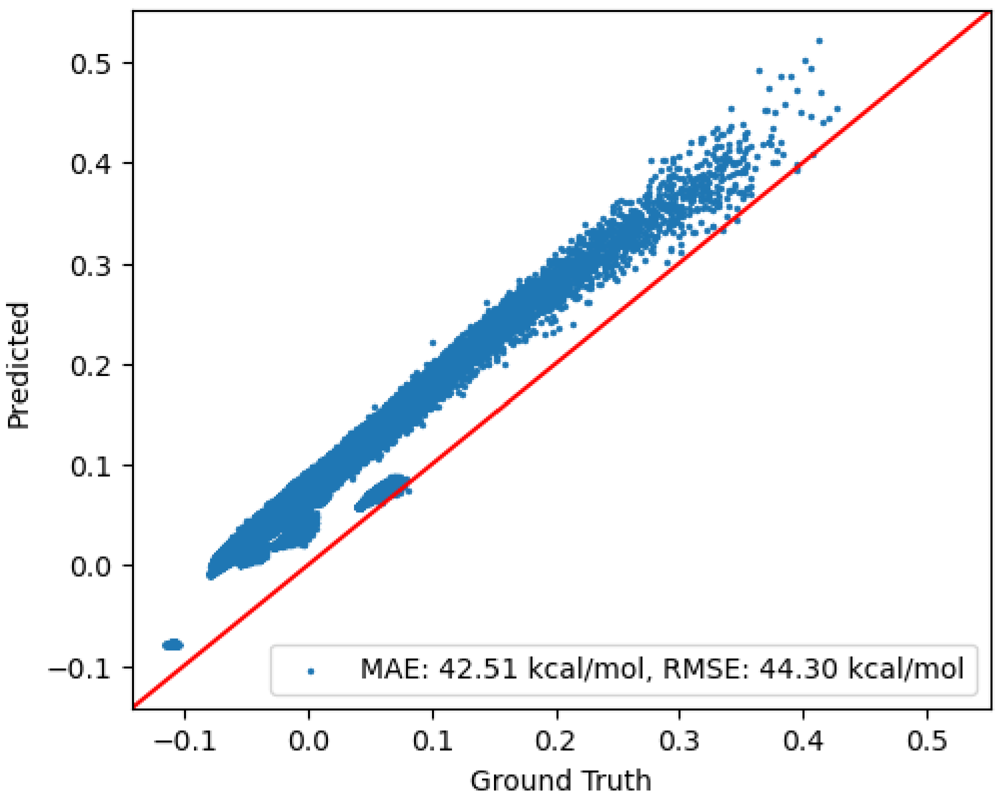
Full Dataset Output Visualization and Results

```
In [26]: for i in range(1,5):
             print(f'Test model trained on s01_to_s04 on s0{i} data')
             dataset = load_ani_dataset(f"./ani_gdb_s0{i}.h5")
             train_data, val_data, test_data = dataset.split(0.8,0.1,None)
             loss, mae, rmse = trainer.evaluate(train_data, draw_plot=True)
```

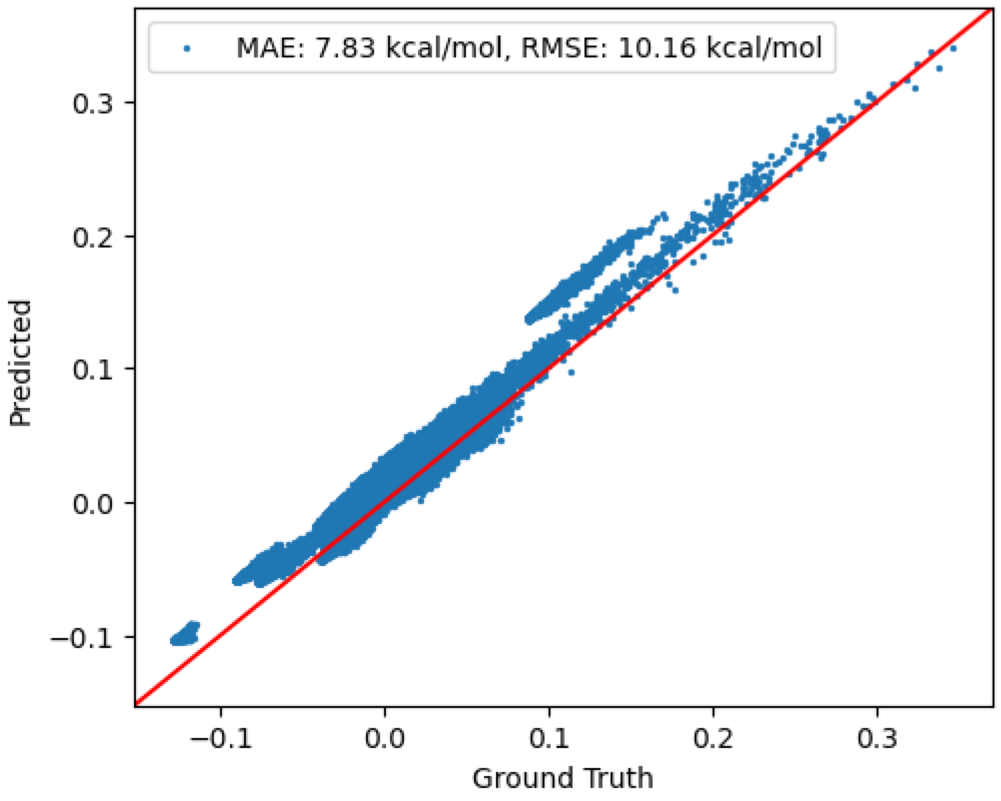
```
Test model trained on s01_to_s04 on s01 data
Test model trained on s01_to_s04 on s02 data
Test model trained on s01_to_s04 on s03 data
Test model trained on s01_to_s04 on s04 data
```

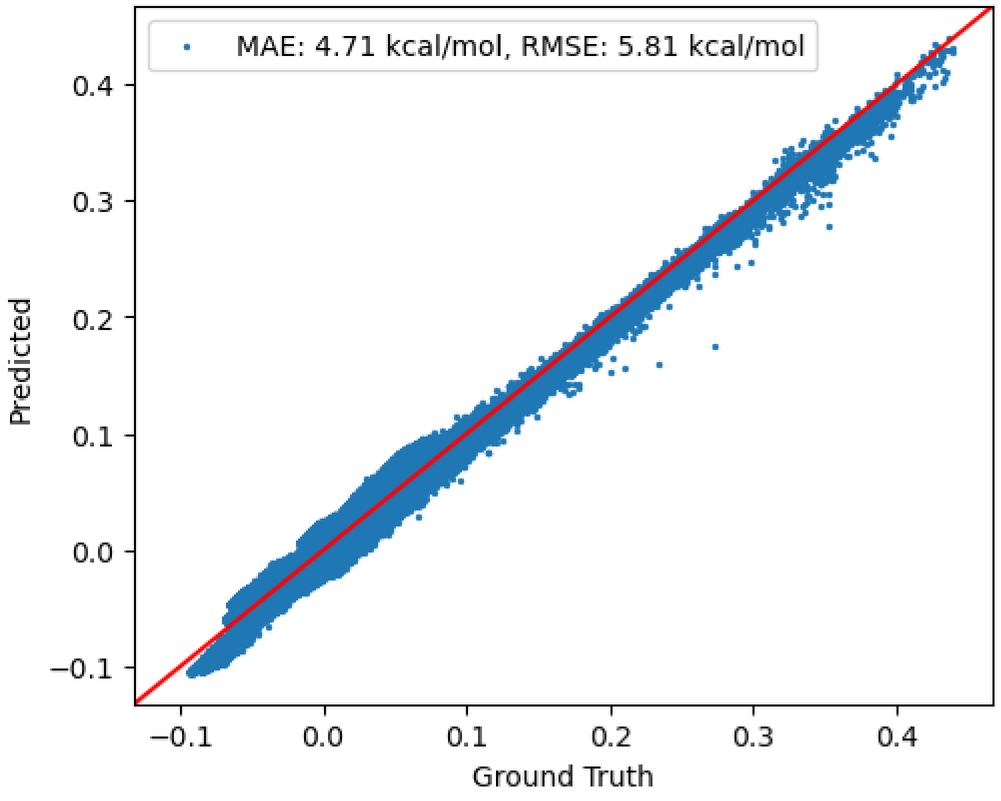


0.08
0.06
0.04
0.02
0.00
−0.02
Predicted
MAE: 16.56 kcal/mol, RMSE: 16.68 kcal/mol
−0.02
0.00
0.02
0.04
0.06
0.08
Ground Truth



0.5
0.4
0.3
0.2
0.1
0.0
−0.1
Predicted
MAE: 42.51 kcal/mol, RMSE: 44.30 kcal/mol
−0.1
0.0
0.1
0.2
0.3
0.4
0.5
Ground Truth





Assessment of Full Dataset Performance

A baseline of 8192 batch size, 1e-3 learning rate, 50 epochs, 1e-5 L2 regularization.This baseline had an MAE of 3.01 kcal/mol and an RMSE of 4.58 kcal/mol with a time of 519.7056 seconds. Steps were taken to improve the model by changing the hyperparameters. Through testing different changes in hyperparameters, it appeared that the graph was underfitting, so changes were made accordingly.

It was found that the optimal hyperparameters were 10000 batch size, 1e-3 learning rate, 50 epochs, 1e-7 L2 regularization which achieved an MAE of 1.76 kcal/mol and an RMSE of 2.24 kcal/mol in 511.1501 seconds. This provided a good balance between performance and minimizing run time overhead. These same parameters except with 100 epochs had a lower MAE of 1.18 kcal/mol and an RMSE of 1.68 kcal/mol. However, the run time was twice as long, taking the model 984.6864 seconds.