ANN Models Applied to ANI-1 Dataset

Introduction

In the field of theoretical chemistry, the ability to perform computational experiments rapidly and with high accuracy is greatly beneficial. This is enabled with the power of Machine Learning (ML), which has enabled various applications from prediction to classification by exploiting and learning the feature-based nature of datasets. ML has emerged as an efficient and progressive alternative to more resource-intensive methods such as Coupled Cluster Singles and Doubles (CCSD).

In past instances, ML has been applied to the field of chemistry to predict molecular properties including calculating potential energy surfaces and atomic charge models [1]. Deep learning models are trained on diversified and extensive amounts of data in order to learn patterns within molecular structures which are then able to predict properties with accuracies that are comparable to conventional methods such as Density Functional Theory (DFT). Evidently, ML is powerful for modern experiments, yet it is limited by access to diversified and extensive amounts of representative data.

In 2017, Smith et al. introduced a dataset of 20M off-equilibrium DFT calculations for 57,454 small organic molecules called the ANI dataset [2]. Intended for the development of transferable deep learning potentials to represent molecular structure, this computational database would serve as a benchmark and further serve to demonstrate the power of ML in computational chemistry. The ANI dataset is based on a modified version of the Behler and Parrinello symmetry function and is intended to allow the development of single-atom atomic environment vectors. These vectors enable molecular prediction in both configurational and conformational space, setting a precedent for its time. The ANI-1 dataset was trained on a subset of the GDB database and contains quantum mechanical properties for four heavy chemical elements, including C, H, N, and O [3]. Information about these atoms' energies, atomic forces, multipole moments, etc is provided. Smith et al. later demonstrated how a deep neural network which was trained on the ANI-1 dataset could successfully learn accurate and transferable organic molecule energy potentials comparable to conventional DFT calculations. .

In this experiment, I have used the ANI-1 dataset to train several Artificial Neural Networks (ANN) in order to determine the optimal model architecture when predicting molecular atom energies. I implement regularization techniques, hyperparameter tuning, and architecture changes in order to optimize prediction accuracy on the data.

Methods

The goal of this experiment was to improve upon previous predictive architectures for the ANI-1 dataset and to devise an optimal configuration of hyperparameters, regularization, and atomic net architecture. I iterated over six different artificial neural network configurations, each trained on a dataset containing four heavy atoms, H, C, N, O. Each model was written using Pytorch.

I started by setting up an atomic environment vector (AEV) computer and defining the number of species. The dimensions of this AEV computer were 384. The next step was to prepare the dataset and split it into training, validation, and test sets. The dataset was first loaded in as a .h5 file and then was split into 80% testing, 10% validation, and 10% testing. Next, the dataset was divided into a batch size of 8192, and data was populated into loaders for the train, validation, and test sets. I then defined the initial atomic network with an input of 384 neurons, a hidden layer of 128 neurons, and an output dimension of 1. To train my models and evaluate their performance, I then created an ANITrainer class that iterated through each epoch and through each batch within the train data. In each training batch, the framework computes the predicted energies, computes the batch loss, and then performs a gradient descent step. Furthermore, in each epoch, the model performs a loss evaluation on the validation dataset and appends both the validation and training loss to later visualize the relationship between these two metrics.

Within the evaluation function of the ANITrainer class, I converted the molecular energies from hartree to kcal/mol and then computed the mean absolute error (MAE). The target MAE for an optimal architecture is below 2 kcal/mol. I tried six different architecture and hyperparameter configurations which can be seen in Fig 1. All models were trained for 30 epochs. For each iteration, I plotted the validation MAE, the test MAE, and the graphical relationship between loss vs epoch for the training and validation sets.

Iteration	Hidden Layers	Learn Rate	Batch Size	L2	Dropout	Val MAE	Smallest Loss	Activation
original	1 [384, 128, 1]	1e-3	8192	1e-5	no	3.99	1.729e-5	ReLU
1	1 [384, 128, 1]	1-5	8192	1e-5	no	1.30	1.177e-5	ReLU
2	1 [384, 200, 1]	1e-5	8192	1e-5	no	4.13	.000133	ReLU
3	2 [384, 128, 64, 1]	1e-5	8192	1e-5	no	3.91	.000162	ReLU
4	1 [384, 128, 1]	1e-5	8192	1e-5	no	28.78	.00536	Sigmoid
5	1 [384, 128, 1]	1e-5	8192	1e-5	yes	1244.18	4.3699	Sigmoid
6	2 [384, 128, 64, 1]	1e-5	512	1e-5	no	31.18	.00541	ReLU

Fig 1. ANN iterations to determine optimal architecture. Highlighted squares show changes across iterations. Optimal iteration was Iteration 1.

Results and Discussion

The performance for my initial model was not optimal with an MAE of 3.99 kcal/mol and a loss of around 1.729e-5. Furthermore, the validation and training loss oscillated drastically beginning at batch 15, which can be seen in Fig 2. To improve this performance, in Iteration 1 I decided to decrease the learning rate from 1e-3 to 1e-5, which led to smooth loss curves and an optimal MAE of 1.30 kcal/mol. To see if the loss and MAE values could be further optimized, I performed five more iterations that included increasing the hidden layer size, increasing the number of hidden layers to two, changing the activation function, adding dropout, and decreasing the batch size. Iterations 4, 5, and 6 led to significantly worse performance, yielding MAE values of 28.78 kcal/mol, 1245.78 kcal/mol, and 31.18 kcal/mol respectively. The MAE distribution for Iteration 5 and its mismatch to the data can be seen in Fig 3. This suggests that using the sigmoid activation function, adding dropout as a regularization technique, and decreasing the batch size are not effective techniques in this case.

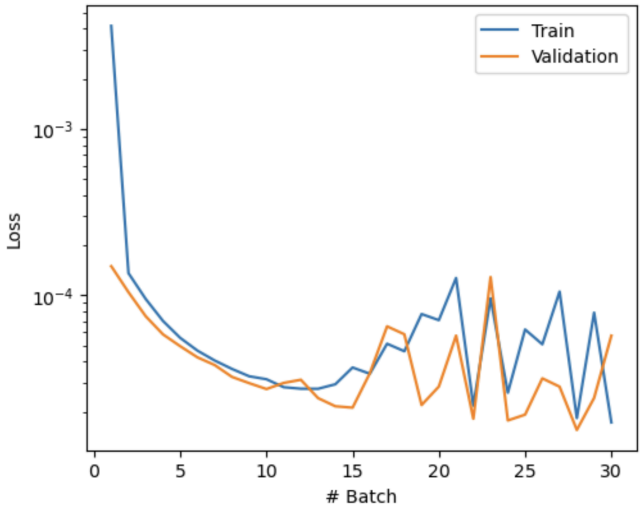


Fig 2. Loss vs batch for training and validation sets in iteration 1

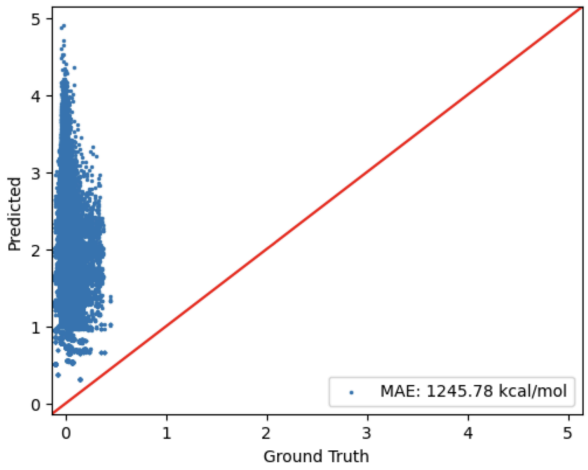


Fig 3. MAE for Iteration 5 with dropout and sigmoid activation

Dropout may have been ineffective in this case because of underfitting. After the dropout of random information units, the model may not have been complex enough to learn from the data that was left. Another reason that dropout was ineffective could be because important information was lost as a result of the dropout, leading to inefficient training. Decreasing the batch size may not have been effective as it may have made the model more sensitive to noise leading to higher variance and slower convergence. Furthermore, smaller batch sizes may not be

utilizing the complete computational capabilities of the gpu. Adding additional hidden layers and changing their size likely failed to optimize the model due to similar reasons. It's possible that adding more hidden layers and features led to dimensional complexity, causing a need for more training data to successfully cover the input space.

On the other hand, decreasing the learning rate to 1e-5 likely led to the most optimal performance because it helped stabilize the learning process. Previously, the model had oscillated as it trained and was not able to converge successfully. Decreasing the learning rate helped smoothen the training while also avoiding overshooting by ensuring the minimum is approached gradually. Furthermore, tuning this hyperparameter may have helped prevent overfitting to noise in the data and also may have helped ensure that the model could escape shallow local minima. The resulting MAE and training vs loss for the optimal final architecture can be seen in Fig 4. Ultimately, Iteration 1 which had a decreased learning rate and standard architecture of one hidden layer led to the best results of 1.30 kcal/mol and a loss of 1.177e-5.

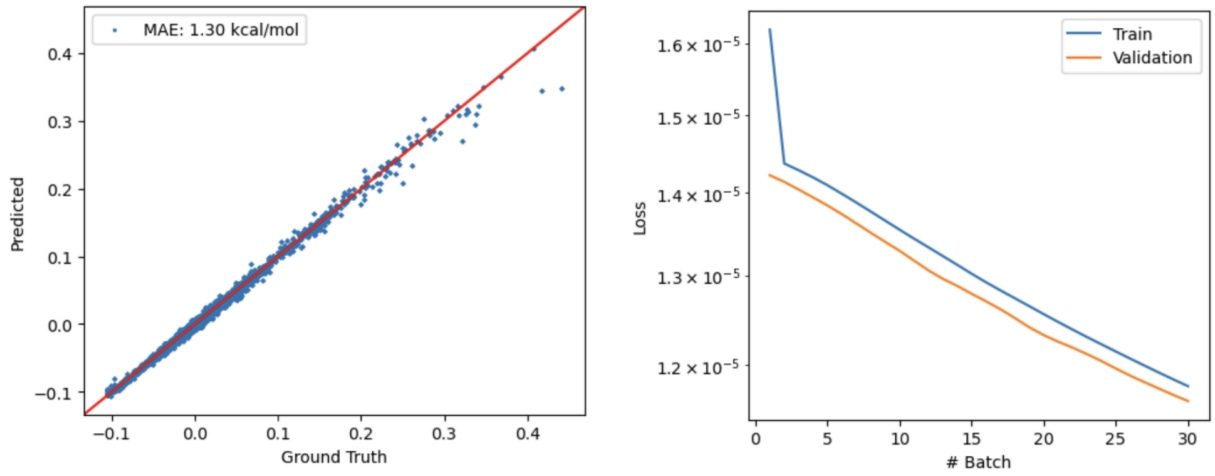


Fig 4. Validation MAE and Loss vs Batch for final optimal model architecture (Iteration 1)

Conclusion

This experiment demonstrated the viability of using an artificial neural network to predict properties of heavy atoms as an alternative to resource-intensive conventional methods. The MAE in the most optimal architecture was below 2 kcal/mol, and the loss also reached a

minimum. Ultimately, a low learning rate led to the stabilization of training and validation loss, ensuring the generalizability of the model. This highlights that the hyperparameter tuning and iterative architecture modifications led to the emergence of a high-accuracy atomic network under a dataset representing diverse molecular structures and four heavy atoms. In the future, training on a dataset with more heavy atoms and as a result greater configurational diversity in molecular structure would lead to more efficient training results. Ultimately, this demonstrates the efficacy of using an ANN to predict molecular energies of a varied set of structures and suggests that this model will have predictive accuracy on new test sets.

References

1. Smith, Justin S., et al. “The Ani-1ccx and Ani-1x Data Sets, Coupled-Cluster and Density Functional Theory Properties for Molecules.” Nature News, Nature Publishing Group, 1 May 2020, www.nature.com/articles/s41597-020-0473-z.
2. Smith, Justin S., Roman Zubatyuk, et al. “The Ani-1ccx and Ani-1x Data Sets, Coupled-Cluster and Density Functional Theory Properties for Molecules.” *Nature News*, Nature Publishing Group, 1 May 2020, www.nature.com/articles/s41597-020-0473-z.
3. Smith, J. S., O. Isayev, et al. “Ani-1: An Extensible Neural Network Potential with DFT Accuracy at Force Field Computational Cost.” Chemical Science, The Royal Society of Chemistry, 8 Feb. 2017, pubs.rsc.org/en/content/articlelanding/2017/sc/c6sc05720a.

Final Project

This notebook is adapted from here:
https://aiqm.github.io/torchani/examples/nnp_training.html

Checkpoint 1: Data preparation

1. Create a working directory: `/global/scratch/users/[USER_NAME]/[DIR_NAME]`. Replace the [USER_NAME] with yours and specify a [DIR_NAME] you like.
2. Copy the Jupyter Notebook to the working directory
3. Download the ANI dataset `ani_dataset_gdb_s01_to_s04.h5` from bCourses and upload it to the working directory

In [1]:
```
import warnings
warnings.filterwarnings("ignore", category=UserWarning)
import numpy as np
from tqdm import tqdm
import torch
import torch.nn as nn
import torchani

print(torch.__version__)
print(torchani.__version__)
```

```
2.2.2
2.2.4
```

Use GPU

In [2]:
```
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
print(device)
```

```
cuda
```

Set up AEV computer

AEV: Atomic Environment Vector (atomic features)

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

In [3]:
```
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=
    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 [4]:
```
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")
# Use dataset.split method to do split
train_data, val_data, test_data = dataset.split(.8, .1, .1)
```

Batching

In [5]:
```
batch_size = 8192
# use dataset.collate(...).cache() method to do batching
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()
```

Torchani API

In [6]:
```
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 torch.ANIModel() to compile atomic networks
ani_net = torchani.ANIModel([net_H, net_C, net_N, net_O])
model = nn.Sequential(
    aev_computer,
    ani_net
).to(device)
```

In [7]:
```
# train_data_batch = next(iter(train_data_loader))

for train_data_batch in 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)
    break
```

```
tensor(0.0355, device='cuda:0', grad_fn=<MseLossBackward0>)
```

Checkpoint 2

In [8]:
```
coords.shape[0]
train_epoch_loss = 0.0

loss_func = nn.MSELoss()
batch_loss = loss_func(true_energies, pred_energies)
batch_importance = coords.shape[0] / len(train_data)
train_epoch_loss += batch_loss.detach().cpu().item() * batch_importance

print(train_epoch_loss)
```

```
0.0004199363558073355
```

In [9]:
```
from torch.utils.data import DataLoader
import matplotlib.pyplot as plt
```

In [10]:
```
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_para

        self.batch_size = batch_size
        self.optimizer = torch.optim.Adam(model.parameters(), learning_rate,
        self.epoch = epoch

    def train(self, train_data, val_data, early_stop=True, draw_curve=True):
        self.model.train()

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

        # definition of loss function: MSE is a good choice!
        loss_func = nn.MSELoss()

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

        for i in tqdm(range(self.epoch), leave=True):
            train_epoch_loss = 0.0
            for train_data_batch in train_data_loader:

                # compute energies
                species = train_data_batch['species'].to(device)
                coords = train_data_batch['coordinates'].to(device)
                true_energies = train_data_batch['energies'].to(device).floa
```

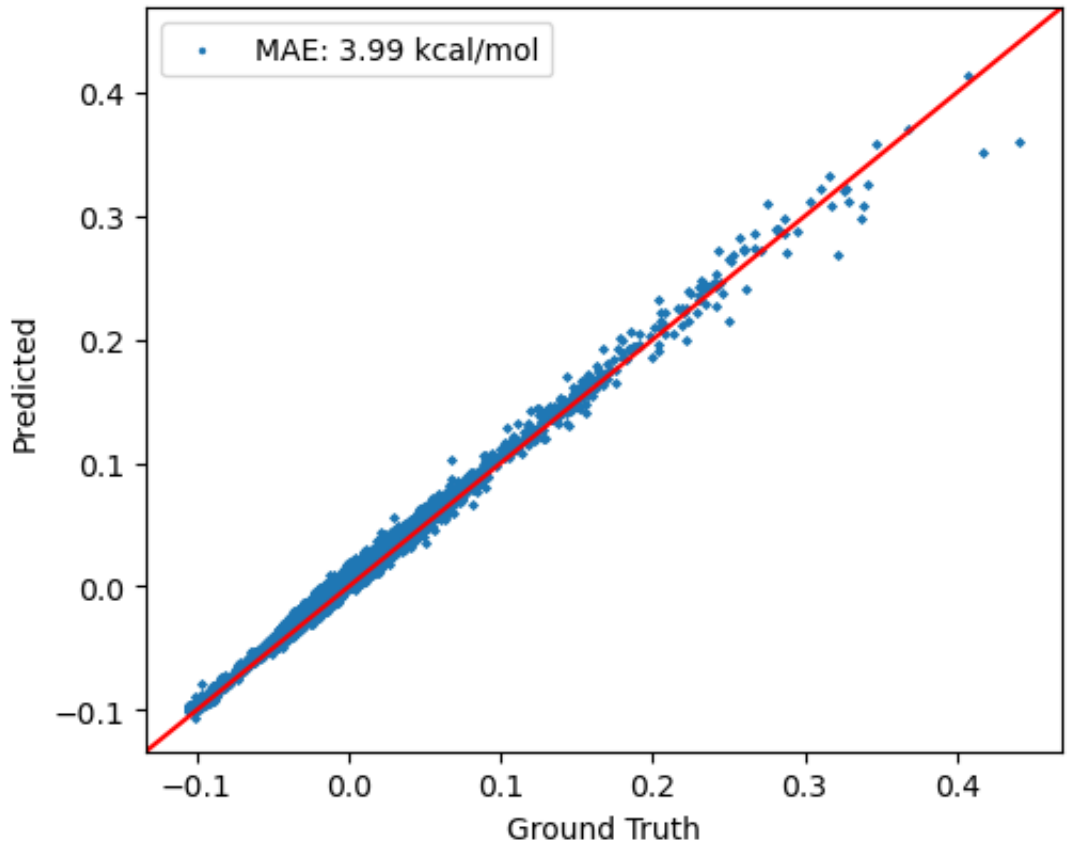
```
                _, pred_energies = model((species, coords))

                # compute loss
                batch_loss = loss_func(true_energies, pred_energies)

                # do a step
                self.optimizer.zero_grad()
                batch_loss.backward()
                self.optimizer.step()

                batch_importance = coords.shape[0] / len(train_data)
                train_epoch_loss += batch_loss.detach().cpu().item() * batch

            # use the self.evaluate to get loss on the validation set
            if (i == (self.epoch - 1)):
                val_epoch_loss = self.evaluate(val_data, draw_plot = True)
            else:
                val_epoch_loss = self.evaluate(val_data)

            # append the losses
            train_loss_list.append(train_epoch_loss)
            val_loss_list.append(val_epoch_loss)

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

        if draw_curve:
            fig, ax = plt.subplots(1, 1, figsize=(5, 4), constrained_layout=
            ax.set_yscale("log")
            # Plot train loss and validation loss
            ax.plot(range(1, len(train_loss_list) + 1), train_loss_list, lab
            ax.plot(range(1, len(val_loss_list) + 1), val_loss_list, label='
            ax.legend()
            ax.set_xlabel("# Batch")
            ax.set_ylabel("Loss")

        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()

        # init loss function
        loss_func = nn.MSELoss()
        total_loss = 0.0
```

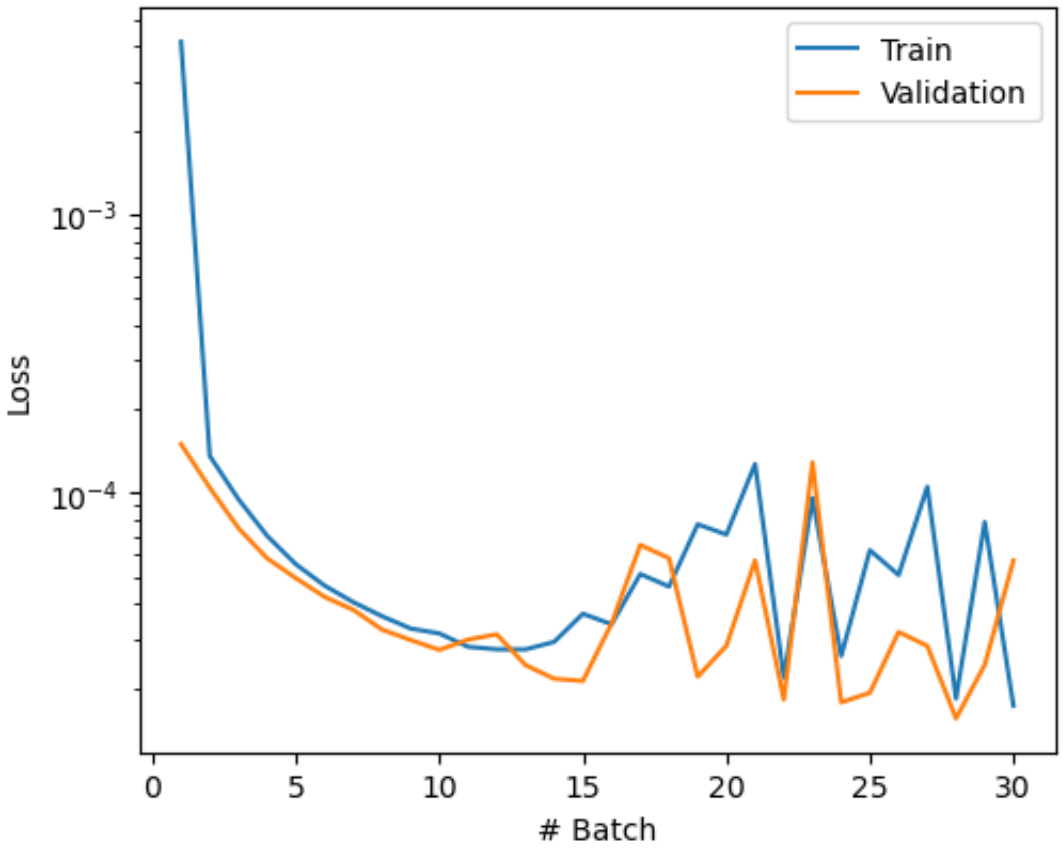
```
        if draw_plot:
            true_energies_all = []
            pred_energies_all = []

        with torch.no_grad():
            for batch_data in data_loader:

                # compute energies
                species = train_data_batch['species'].to(device)
                coords = train_data_batch['coordinates'].to(device)
                true_energies = train_data_batch['energies'].to(device).floa
                _, pred_energies = model((species, coords))


                # compute loss
                batch_loss = loss_func(true_energies, pred_energies)

                batch_importance = coords.shape[0] / len(data)
                total_loss += batch_loss.detach().cpu().item() * batch_impor

                if draw_plot:
                    true_energies_all.append(true_energies.detach().cpu().nu
                    pred_energies_all.append(pred_energies.detach().cpu().nu

        if draw_plot:
            true_energies_all = np.concatenate(true_energies_all)
            pred_energies_all = np.concatenate(pred_energies_all)
            # Report the mean absolute error
            # The unit of energies in the dataset is hartree
            # please convert it to kcal/mol when reporting the mean absolute
            # 1 hartree = 627.5094738898777 kcal/mol
            # MAE = mean(|true - pred|)
            hartree2kcalmol = 627.5094738898777 # changed
            mae = np.mean(np.abs(true_energies_all * hartree2kcalmol - pred_
            fig, ax = plt.subplots(1, 1, figsize=(5, 4), constrained_layout=
            ax.scatter(true_energies_all, pred_energies_all, label=f"MAE: {m
            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()
            #print(mae)

        return total_loss
```

In [13]:
```
ani_net = torchani.ANIModel([net_H, net_C, net_N, net_O])
model = nn.Sequential(
    aev_computer,
    ani_net
).to(device)
```

In [14]:
```
model = model

trainer = ANITrainer(model, learning_rate=1e-3, batch_size = 8192, epoch = 3

train_losses, val_losses = trainer.train(train_data, val_data)
```

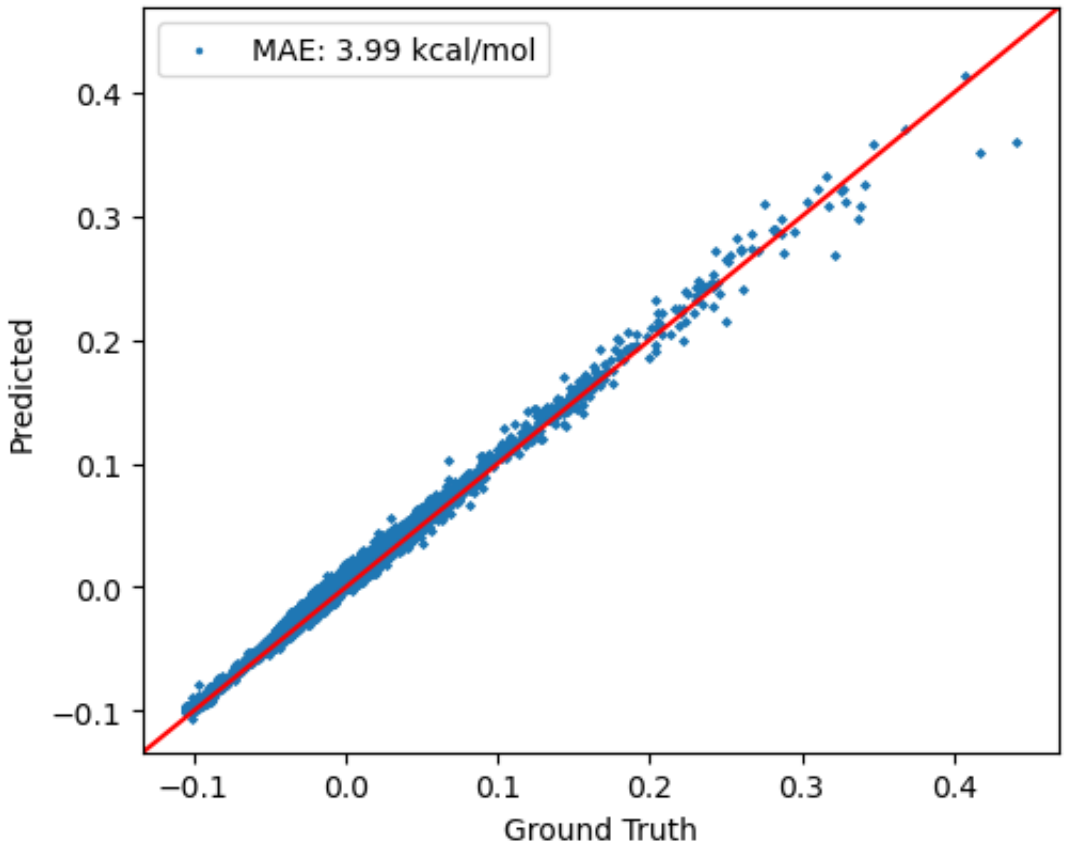
```
Sequential - Number of parameters: 197636
Initialize training data...
100%|██████████| 30/30 [04:47<00:00,  9.57s/it]
```





In [15]:
```
trainer.evaluate(test_data, draw_plot = True)
```

Out[15]:
```
5.7331837363884256e-05
```



In [16]:
```
print(train_losses)
```

```
[0.004150044536737322, 0.00013599351319533524, 9.514818989270177e-05, 7.0278
52081559144e-05, 5.559908003337367e-05, 4.661247391919525e-05, 4.06657865127
73615e-05, 3.621267203060398e-05, 3.270527230855875e-05, 3.1402471820477994e
-05, 2.812861814916419e-05, 2.755122388931103e-05, 2.7531479899193634e-05,
2.9291981469117514e-05, 3.693555873841545e-05, 3.3873674977873154e-05, 5.129
2386025111436e-05, 4.623098497962414e-05, 7.731075727434039e-05, 7.103601808
456633e-05, 0.00012720442190070373, 2.1829907562654332e-05, 9.59621303833095
9e-05, 2.6061130264811148e-05, 6.244133583872179e-05, 5.088604436045897e-05,
0.00010542876427378199, 1.8358793291461095e-05, 7.876913400413208e-05, 1.729
4839518349943e-05]
```

Checkpoint 3

The lowest loss after checkpoint 2 was 1.7294839518349943e-05, and the MAE was 3.99 kcal/mol for both validation and test data. Now I will optimize the loss and MAE by implementing regularization techniques and performing hyperparameter tuning. My approaches include: modifying the architecture in each atomic net, using more hidden layers, varying the hidden layer size, adding dropouts, changing activation function

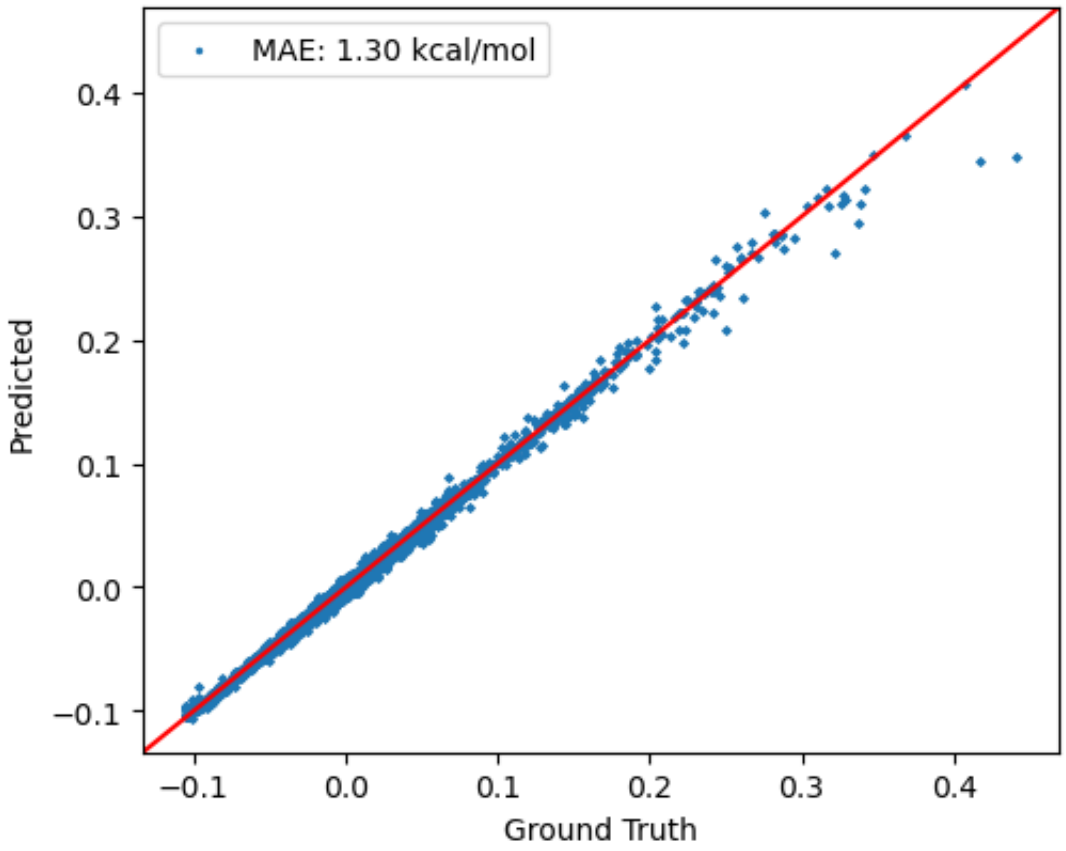
Iteration 1

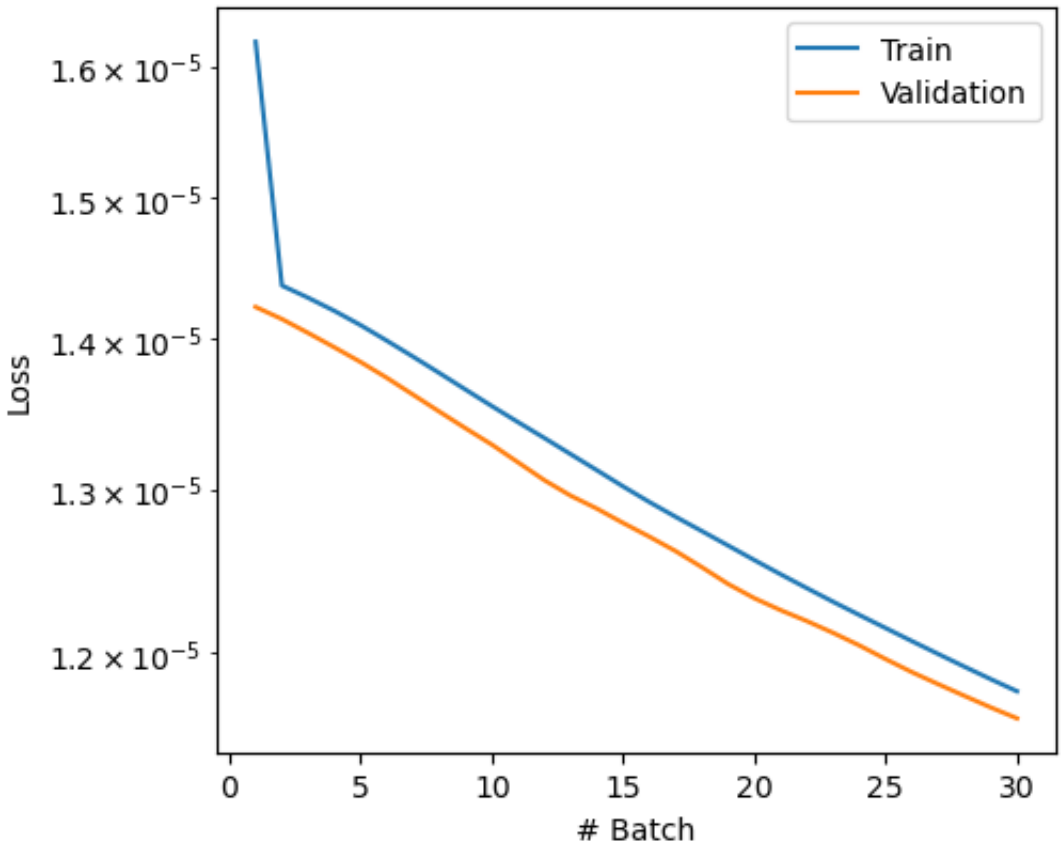
Here I decreased the learning rate from 1e-3 to 1e-5

In [17]:
```
ani_net = torchani.ANIModel([net_H, net_C, net_N, net_O])
model = nn.Sequential(
    aev_computer,
    ani_net
).to(device)
```

In [18]:
```
model = model

# tried decreasing the learning rate
trainer = ANITrainer(model, learning_rate=1e-5, batch_size = 8192, epoch = 3

train_losses, val_losses = trainer.train(train_data, val_data)
```

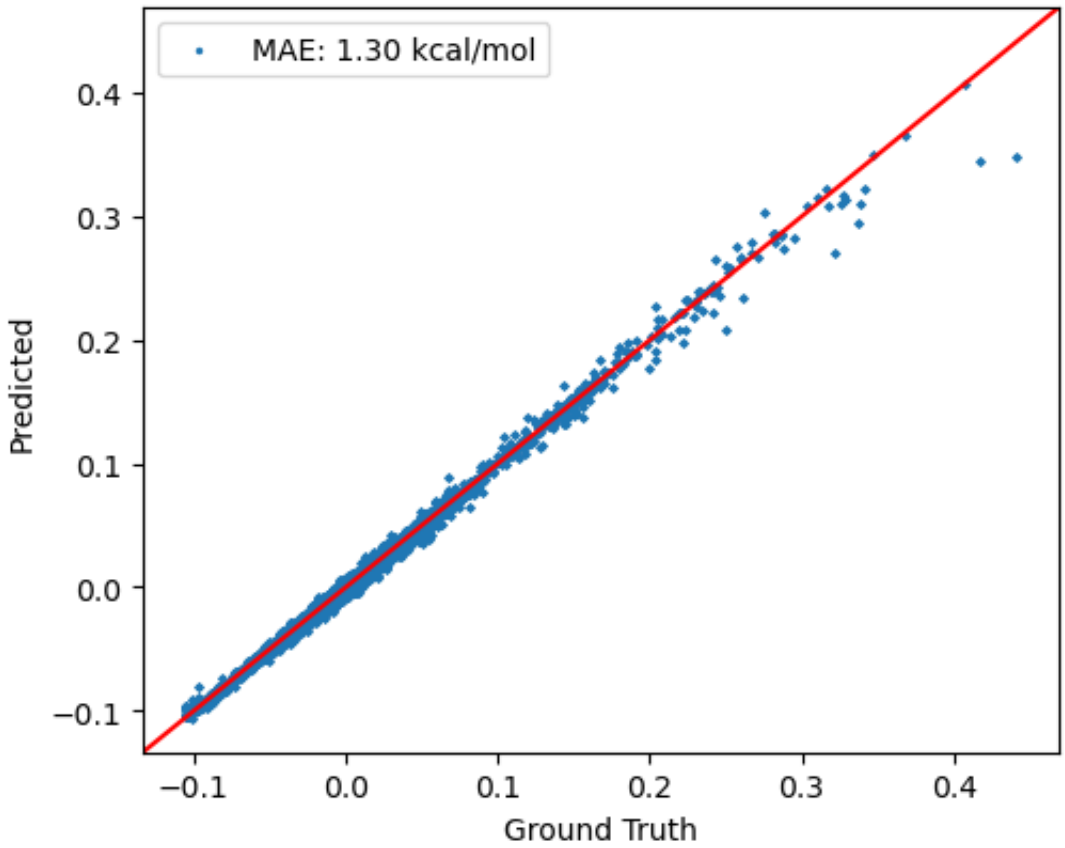
```
Sequential - Number of parameters: 197636
Initialize training data...
100%|██████████| 30/30 [04:55<00:00,  9.85s/it]
```





In [19]:
```
trainer.evaluate(test_data, draw_plot = True)
```

Out[19]:
```
1.1618574713498447e-05
```



In [20]:
```
print(train_losses)
```

```
[1.6198787425184687e-05, 1.437100855824389e-05, 1.4284719865594101e-05, 1.41
93105494001543e-05, 1.409366437732297e-05, 1.3986305857150458e-05, 1.3877033
926014824e-05, 1.3766084715054987e-05, 1.3654304865283119e-05, 1.35442844073
60258e-05, 1.3437554973404667e-05, 1.3333548492603435e-05, 1.322821489480700
8e-05, 1.3124391283397342e-05, 1.3020091951700905e-05, 1.2920777155121306e-0
5, 1.2827279716368306e-05, 1.2737543308203968e-05, 1.2647523866215924e-05,
1.2557485454078932e-05, 1.2470214182097147e-05, 1.2385469280158987e-05, 1.23
02969178580005e-05, 1.2223159963160752e-05, 1.2145029925637953e-05, 1.206752
8187063236e-05, 1.1991593831423151e-05, 1.1917112312281973e-05, 1.1844411974
322837e-05, 1.1774181918935158e-05]
```

This gives a very optimal loss of 1.1774181918935158e-05, which is even less than what we had achieved in checkpoint 2. Furthermore, we have a validation and test MAE of 1.30 kcal/mol which is signinifcantly less than the MAE from iteration 1 of 3.99. Since the MAE for this iteration is less tha 2 kcal/mol, we can see that this model architecture is efficiently and accuratly predicts molecular preoperties.

Altering Hidden Layers and Number of Neurons

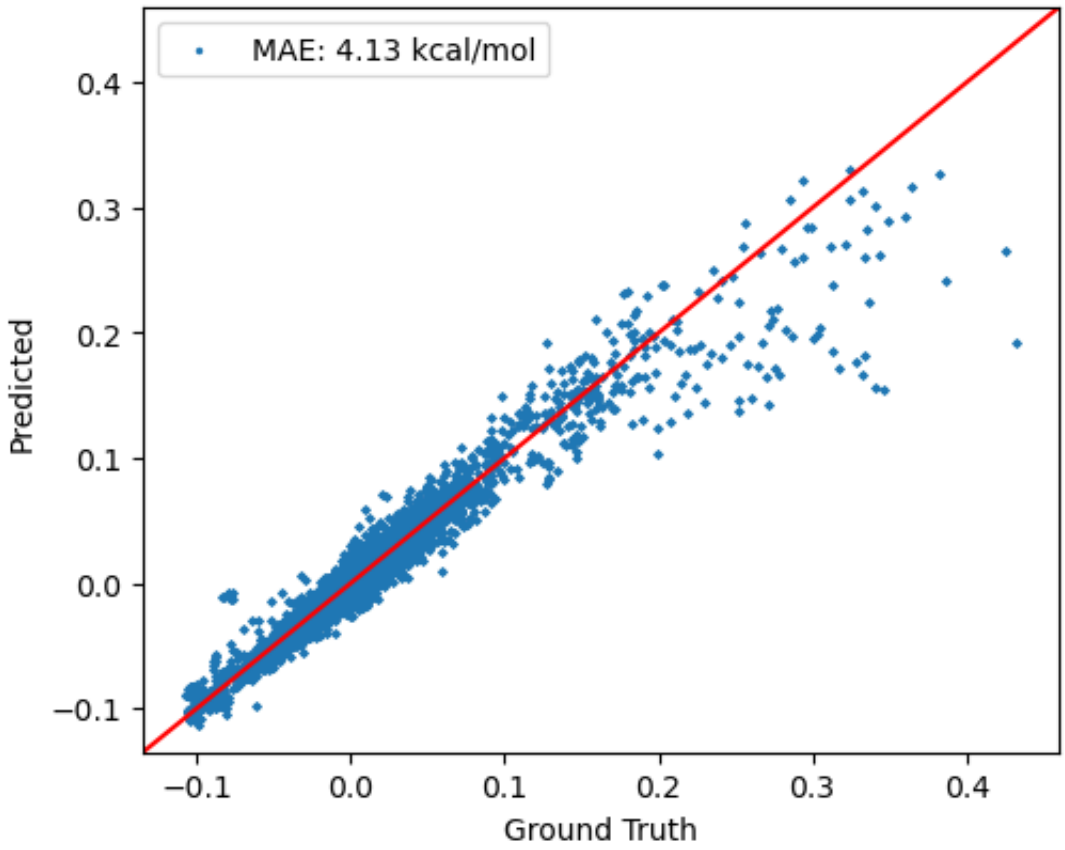
Iteration 2

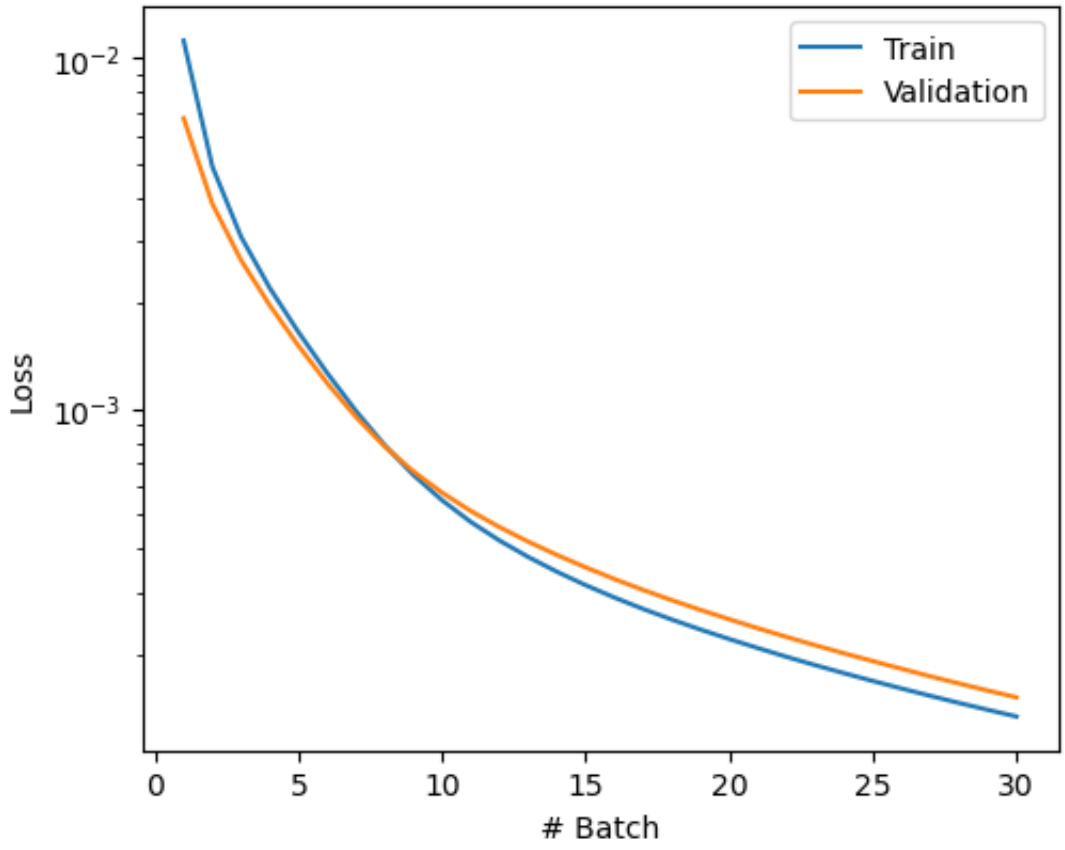
Here I change the hidden layer size to be 200 neurons.

In [27]:
```
# change hidden layer to 200 neurons
class AtomicNet(nn.Module):
    def __init__(self):
        super().__init__()
        self.layers = nn.Sequential(
            nn.Linear(384, 200),
            nn.ReLU(),
            nn.Linear(200, 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 torch.ANIModel() to compile atomic networks
ani_net = torchani.ANIModel([net_H, net_C, net_N, net_O])
model = nn.Sequential(
    aev_computer,
    ani_net
).to(device)
```

In [28]:
```
model = model

trainer = ANITrainer(model, learning_rate=1e-5, batch_size = 8192, epoch = 3

train_losses, val_losses = trainer.train(train_data, val_data)
```

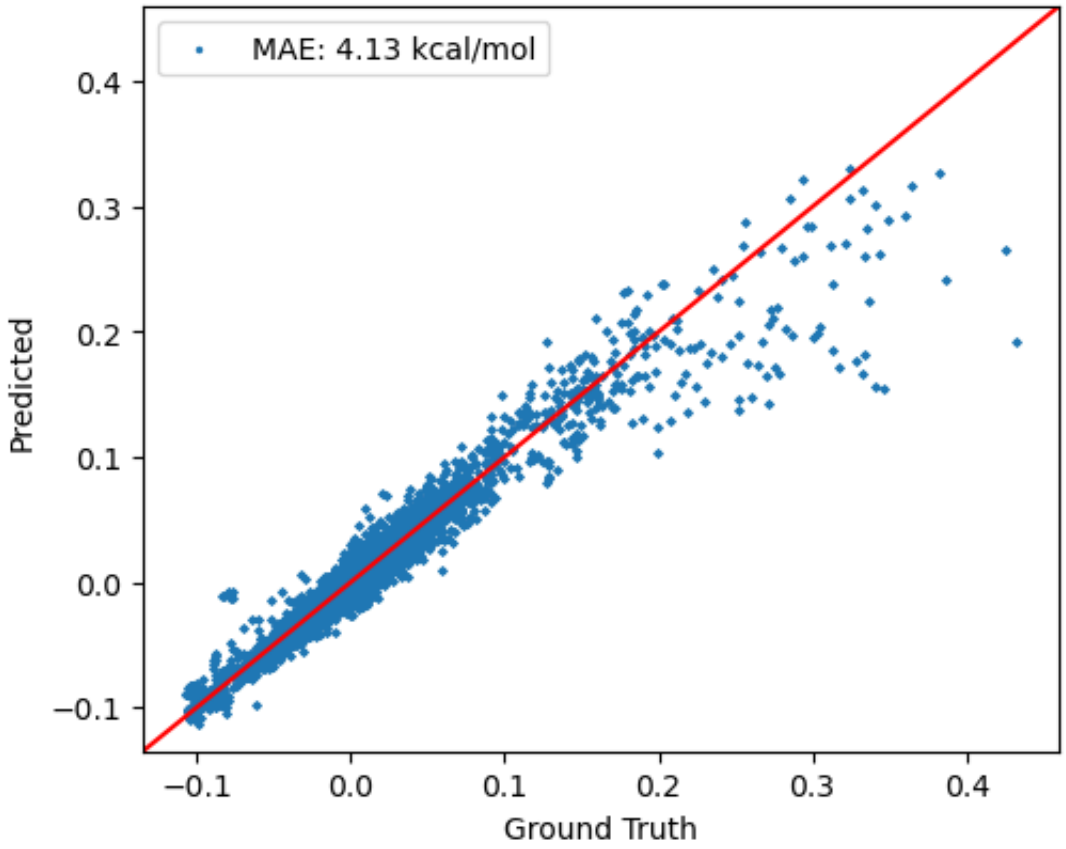
```
Sequential - Number of parameters: 308804
Initialize training data...
100%|██████████| 30/30 [05:15<00:00, 10.50s/it]
```


MAE: 4.13 kcal/mol
Predicted
Ground Truth
0.4
0.3
0.2
0.1
0.0
−0.1
−0.1
0.0
0.1
0.2
0.3
0.4


Train
Validation
Loss
Batch
10^{-2}
10^{-3}
0
5
10
15
20
25
30

In [29]:
```
trainer.evaluate(test_data, draw_plot = True)
```

Out[29]:
```
0.0001510852903961897
```



In [30]:
```
print(train_losses)
```

```
[0.011198594086009303, 0.004904332405735425, 0.0030942654196581304, 0.002212
1939635551344, 0.001655802852550606, 0.0012665693233772789, 0.00098804594648
36882, 0.0007894237959531604, 0.0006494236399167477, 0.0005498852287692776,
0.0004771340878089664, 0.0004222732428181937, 0.0003792264057511114, 0.00034
41798974062128, 0.0003151930724736412 5, 0.0002907010556164493, 0.00026971169
850084837, 0.0002515513059471340 3, 0.0002355815441843788, 0.0002213984195992
1885, 0.000208615520156008, 0.0001970005347301850 6, 0.0001865073006893595 7,
0.0001769377814453241 2, 0.0001681686502599739, 0.0001600868728579345 2, 0.000
1525658371898371, 0.0001456089711597646, 0.0001391919152983602, 0.0001332223
0234034257]
```

We can see that the lowest loss here is 0.00013322230234034257, which is not as optimale as Iteration 1. Both the validation and test MAE are 4.13.

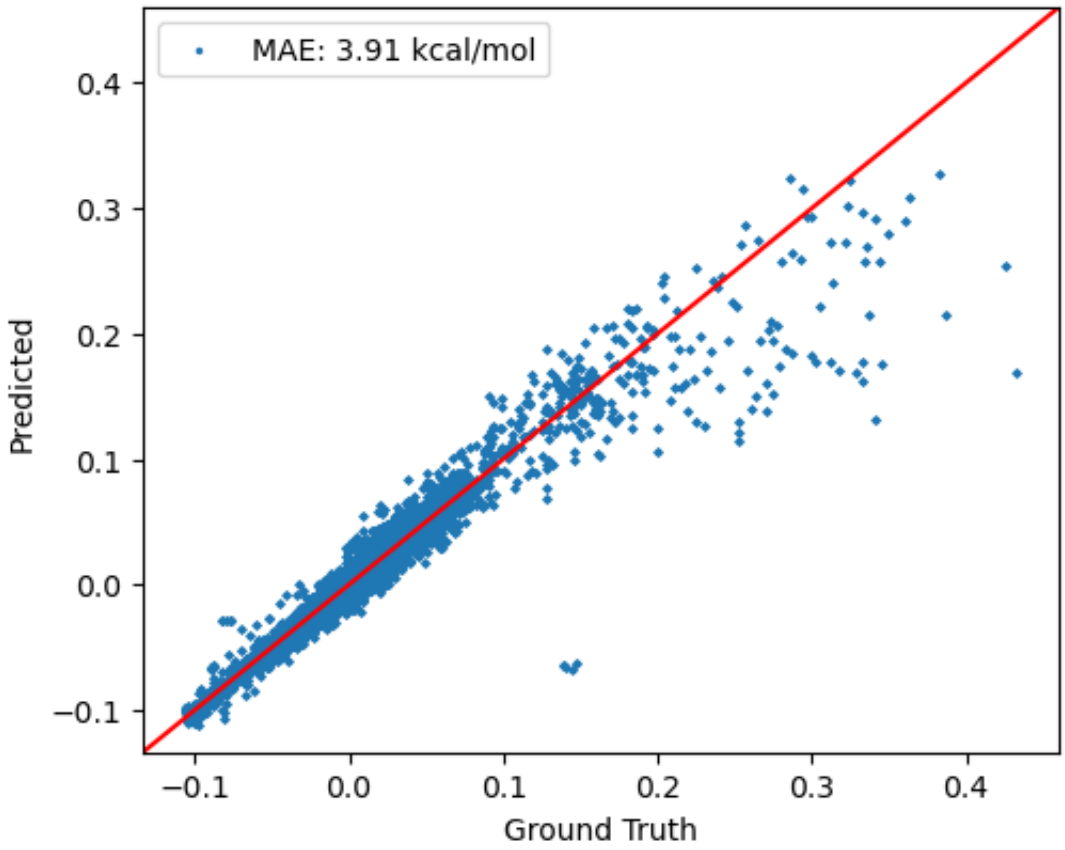
Iteration 3

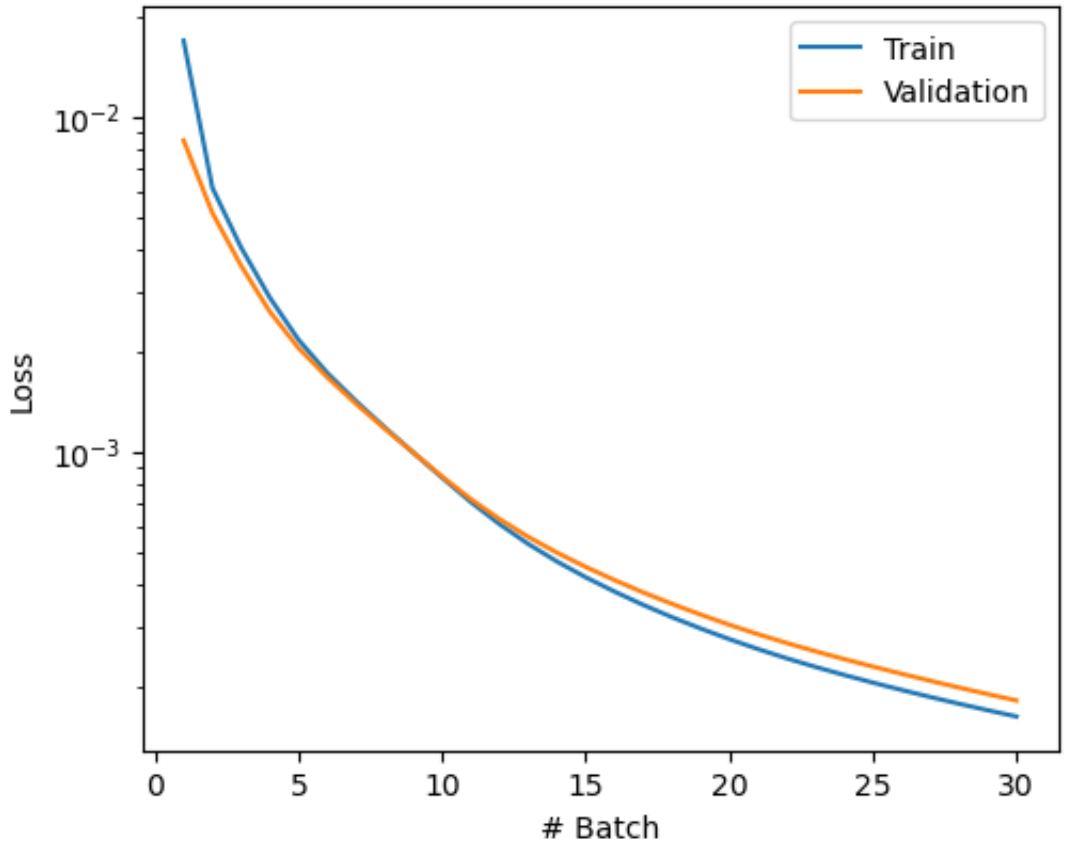
Here I add another hidden layer for total of 2 hidden layers

In [31]:
```
# change to 2 hidden layers
class AtomicNet(nn.Module):
    def __init__(self):
        super().__init__()
        self.layers = nn.Sequential(
            nn.Linear(384, 128),
            nn.ReLU(),
            nn.Linear(128, 64),
            nn.ReLU(),
            nn.Linear(64, 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 torch.ANIModel() to compile atomic networks
ani_net = torchani.ANIModel([net_H, net_C, net_N, net_O])
model = nn.Sequential(
    aev_computer,
    ani_net
).to(device)
```

In [32]:
```
model = model

trainer = ANITrainer(model, learning_rate=1e-5, batch_size = 8192, epoch = 3

train_losses, val_losses = trainer.train(train_data, val_data)
```

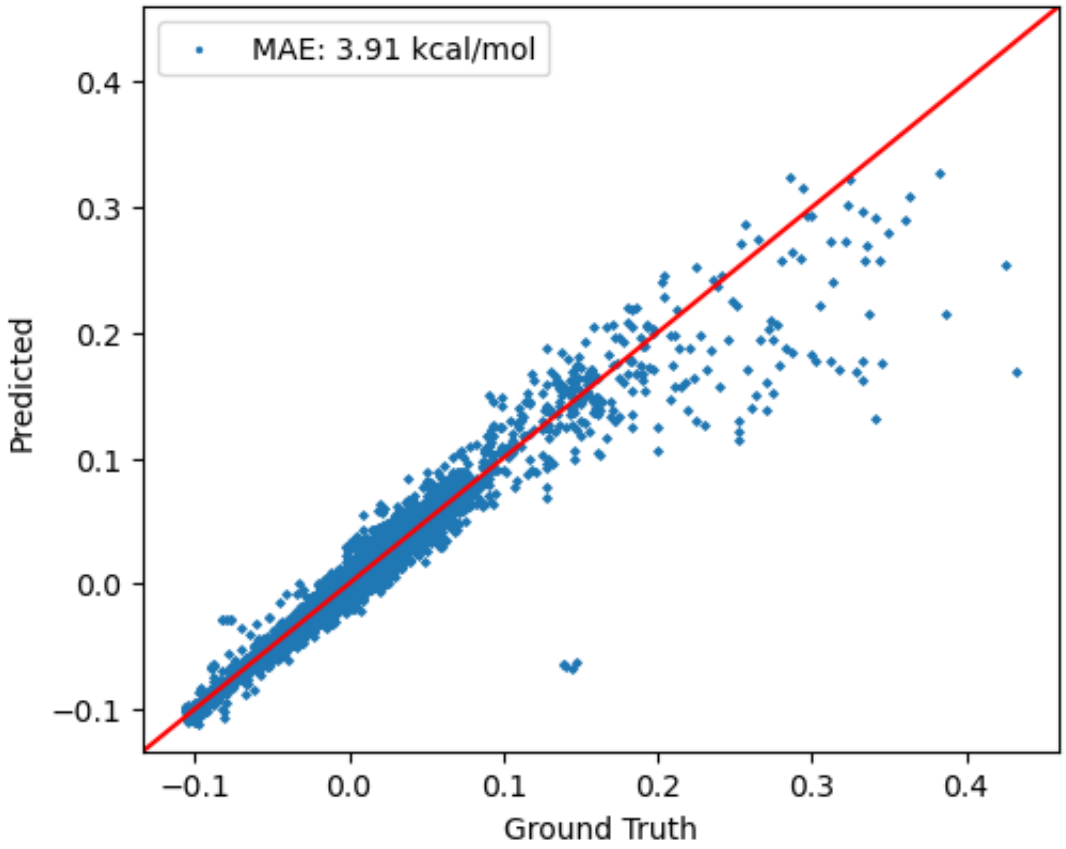
```
Sequential - Number of parameters: 230404
Initialize training data...
100%|██████████| 30/30 [04:56<00:00,  9.88s/it]
```


MAE: 3.91 kcal/mol
Predicted
Ground Truth
0.4
0.3
0.2
0.1
0.0
−0.1
−0.1
0.0
0.1
0.2
0.3
0.4


Train
Validation
Loss
Batch
10^{-2}
10^{-3}
0
5
10
15
20
25
30

In [33]:
```
trainer.evaluate(test_data, draw_plot = True)
```

Out[33]:
```
0.00018131680013287472
```



In [34]:
```
print(train_losses)
```

```
[0.016902633900835327, 0.006151463566138277, 0.004080630323489097, 0.0028912
382810755764, 0.002164501003997744, 0.0017229005873061597, 0.001419564669807
1768, 0.0011846522954481072, 0.0009959653117935375, 0.0008389344439036669,
0.0007092813084363018, 0.0006091732172968584, 0.0005317302496779389, 0.00047
127280169100423, 0.0004229998762752826, 0.00038336311315306966, 0.0003498510
0296456504, 0.0003215336028427507, 0.00029742407860066213, 0.000276422869111
1021, 0.0002581598130783221 7, 0.00024226185530607145, 0.0002283875115168195
8, 0.0002160682457105 9532, 0.00020500398995561942, 0.00019495014930457887,
0.0001857012934934007 6, 0.00017727921520860123, 0.00016954677151862892, 0.00
016239312862845572]
```

Similar to iterration 2, iteration 3 leads to a loss of 0.00016239312862845572. This is again not as optimal as iteration 1. The validation and test MAE are 3.91, which is slightly better than the MAE found in iteration 2.

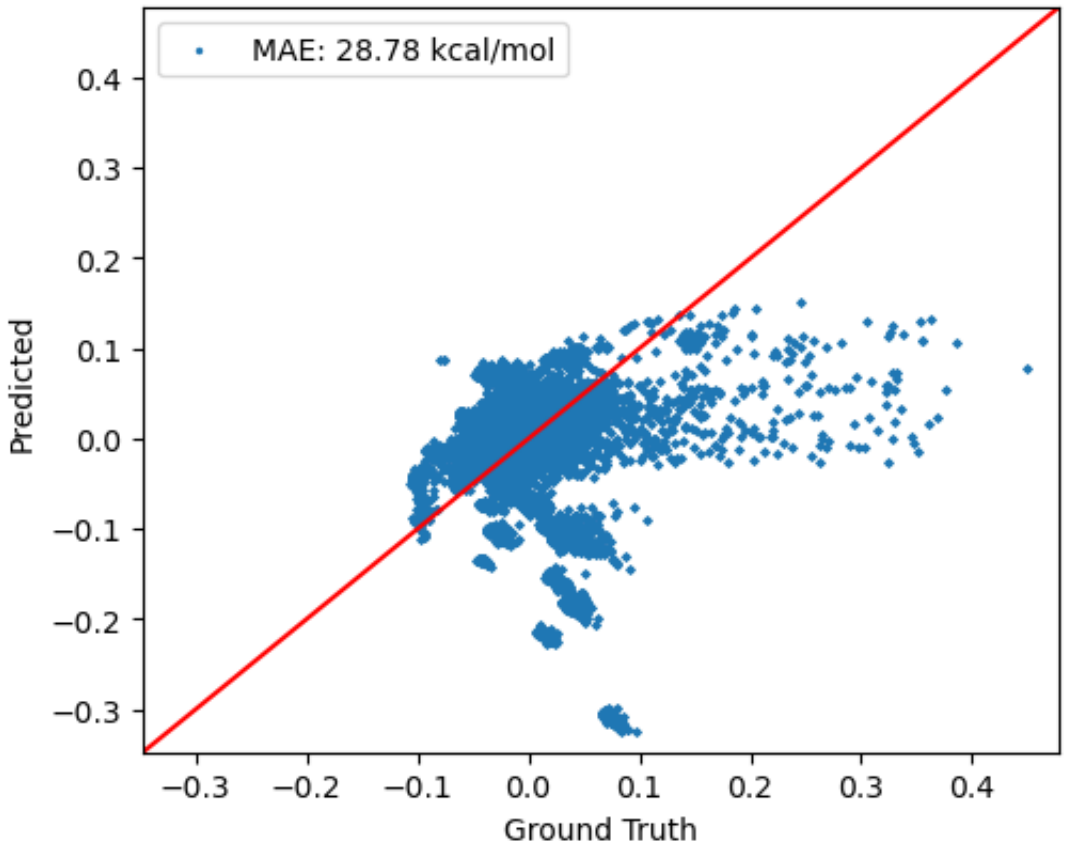
Iteration 4

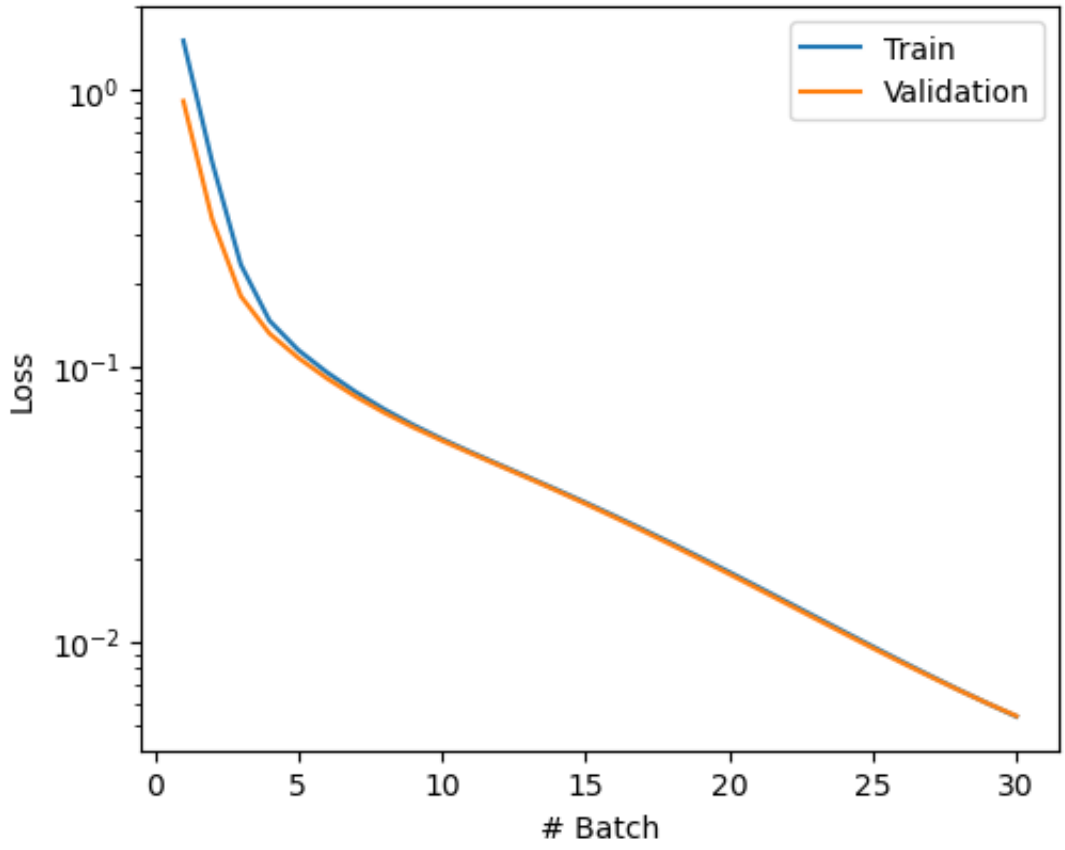
Now I will change activation function from ReLU to Sigmoid.

In [16]:
```
# change activation function
class AtomicNet(nn.Module):
    def __init__(self):
        super().__init__()
        self.layers = nn.Sequential(
            nn.Linear(384, 128),
            nn.Sigmoid(),
            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 torch.ANIModel() to compile atomic networks
ani_net = torchani.ANIModel([net_H, net_C, net_N, net_O])
model = nn.Sequential(
    aev_computer,
    ani_net
).to(device)
```

In [17]:
```
model = model

trainer = ANITrainer(model, learning_rate=1e-5, batch_size = 8192, epoch = 3

train_losses, val_losses = trainer.train(train_data, val_data)
```

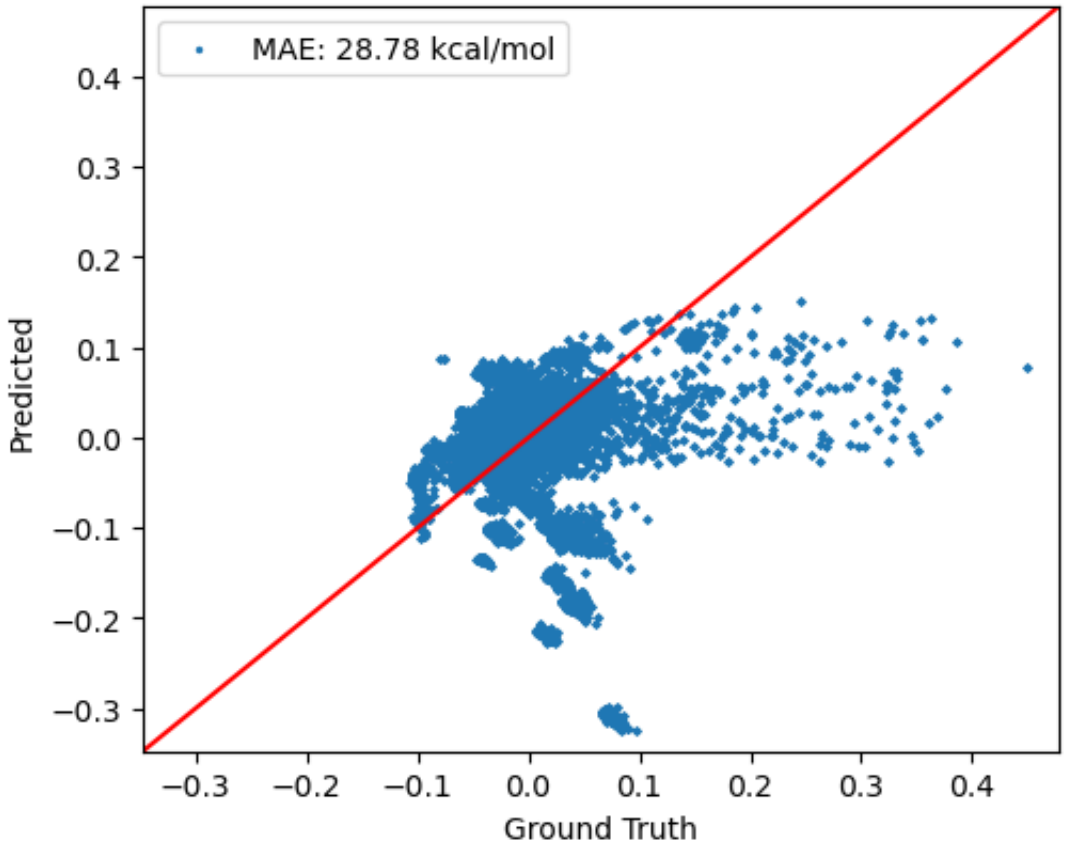
```
Sequential - Number of parameters: 197636
Initialize training data...
100%|██████████| 30/30 [05:15<00:00, 10.53s/it]
```


MAE: 28.78 kcal/mol
Predicted
Ground Truth
0.4
0.3
0.2
0.1
0.0
−0.1
−0.2
−0.3
−0.3
−0.2
−0.1
0.0
0.1
0.2
0.3
0.4


Train
Validation
Loss
Batch
10^{0}
10^{-1}
10^{-2}
0
5
10
15
20
25
30

In [18]:
```
trainer.evaluate(test_data, draw_plot = True)
```

Out[18]:
```
0.00538318540836897
```



In [19]:
```
print(train_losses)
```

```
[1.5109391065437816, 0.5500195810874008, 0.2331469871181487, 0.1459520590001
274, 0.11423158201673402, 0.09482943795281365, 0.0804987498352487, 0.0696012
8407505019, 0.06115456678750528, 0.05439721283410198, 0.048772662073493796,
0.04390613837408042, 0.039564907185506995, 0.035615733834595716, 0.031988203
731551416, 0.02864782305148718, 0.025578042204748432, 0.02276968319424918,
0.020215203202925903, 0.017906103204565784, 0.01583196588893969, 0.013980424
235496773, 0.012337452546964849, 0.010887857790499698, 0.00961575704842344,
0.008504947504170511, 0.0075393399237242194, 0.006703351922989068, 0.0059820
55371318673, 0.005361442608532155]
```

This led to a worse performance than all the previous iterations. The loss is 0.005361442608532155 and the MAE is a very high 28.78 kcal/mol.

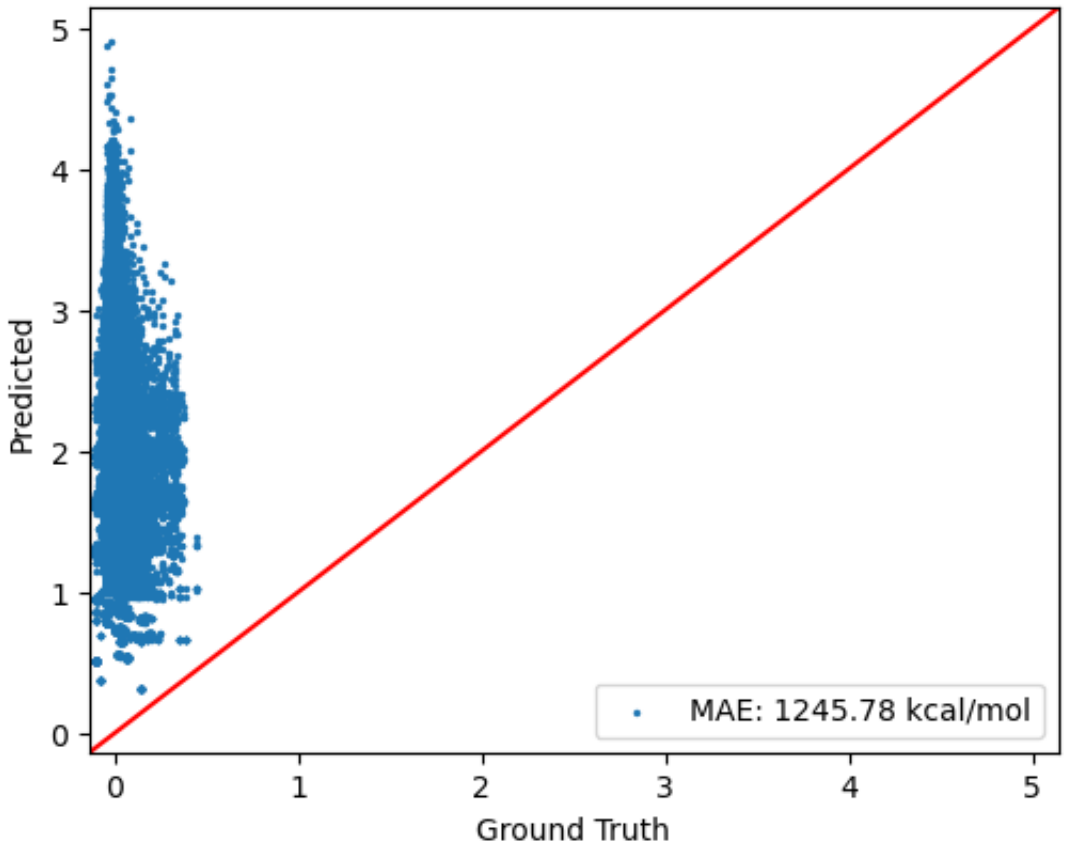
Iteration 5

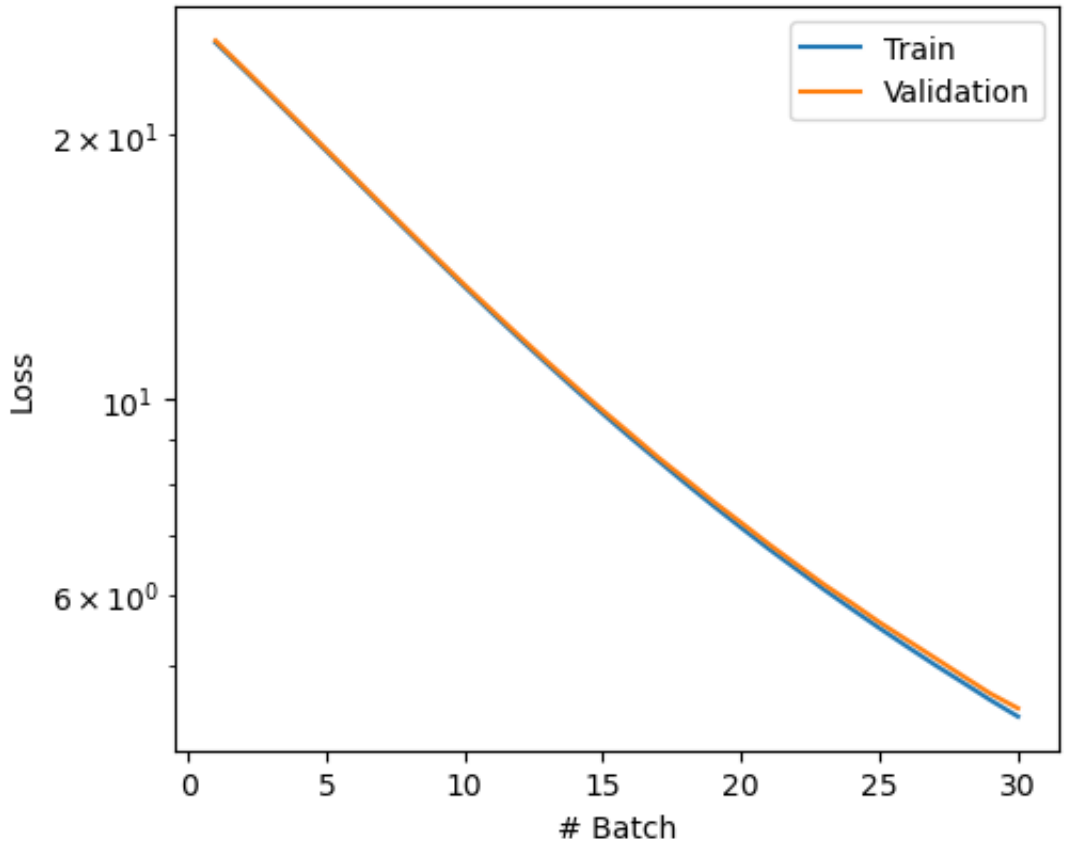
Here I add dropout with the Sigmoid activation function.

In [20]:
```
# Add Dropout
class AtomicNet(nn.Module):
    def __init__(self):
        super().__init__()
        self.layers = nn.Sequential(
            nn.Linear(384, 128),
            nn.Dropout(p=0.15),
            nn.Sigmoid(),
            nn.Linear(128, 1),
            nn.Dropout(p=0.15),
            nn.Sigmoid(),
        )

    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 torch.ANIModel() to compile atomic networks
ani_net = torchani.ANIModel([net_H, net_C, net_N, net_O])
model = nn.Sequential(
    aev_computer,
    ani_net
).to(device)
```

In [21]:
```
model = model

trainer = ANITrainer(model, learning_rate=1e-5, batch_size = 8192, epoch = 3

train_losses, val_losses = trainer.train(train_data, val_data)
```

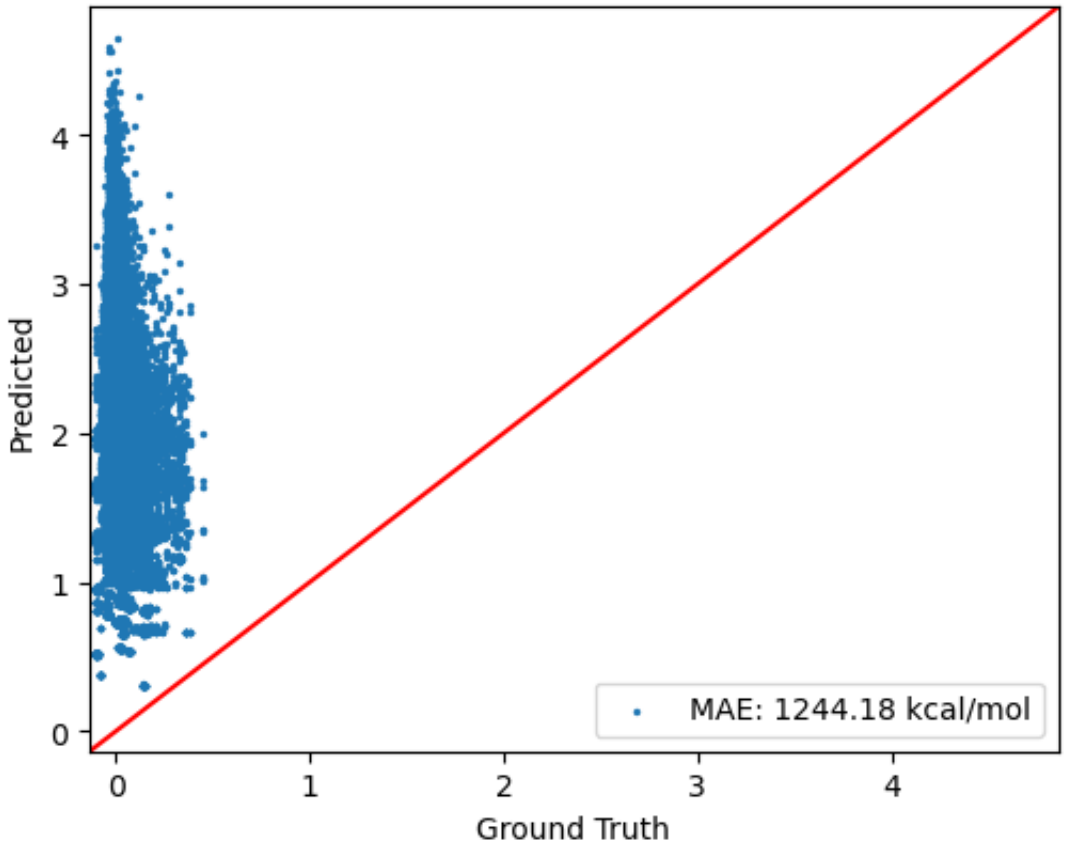
```
Sequential - Number of parameters: 197636
Initialize training data...
100%|██████████| 30/30 [04:55<00:00,  9.85s/it]
```


Predicted
Ground Truth
MAE: 1245.78 kcal/mol
0
1
2
3
4
5


Loss
Batch
Train
Validation
2×10^1
10^1
6×10^0
0
5
10
15
20
25
30

In [22]:
```
trainer.evaluate(test_data, draw_plot = True)
```

Out[22]:
```
4.452102335629386
```



In [23]:
```
print(train_losses)
```

```
[25.42728441695278, 23.7217000573818, 22.106758819508894, 20.58712591213645
7, 19.163246333882157, 17.8357531841696, 16.600765391986602, 15.457164777947
526, 14.405716801323448, 13.43582456236555, 12.53967674584036, 11.7194286971
87189, 10.966857420888148, 10.273054157884143, 9.63960918208587, 9.054474303
075532, 8.526406179845244, 8.029856606249867, 7.575496110684058, 7.157020076
723713, 6.769853723063323, 6.419699803022335, 6.087143596358469, 5.785909187
165259, 5.504668085305071, 5.244672303925321, 5.001379956716937, 4.778309161
511738, 4.56298850804464, 4.369976060625383]
```

Similar to the last iteration, iteration 5 did not reduce the loss and performed worse than the previous iterations. The loss was 4.369976060625383 and the MAE is the highest it has been, 1244.18 kcal/mol. Evidently, dropout is not an effective regularization technique for this dataset.

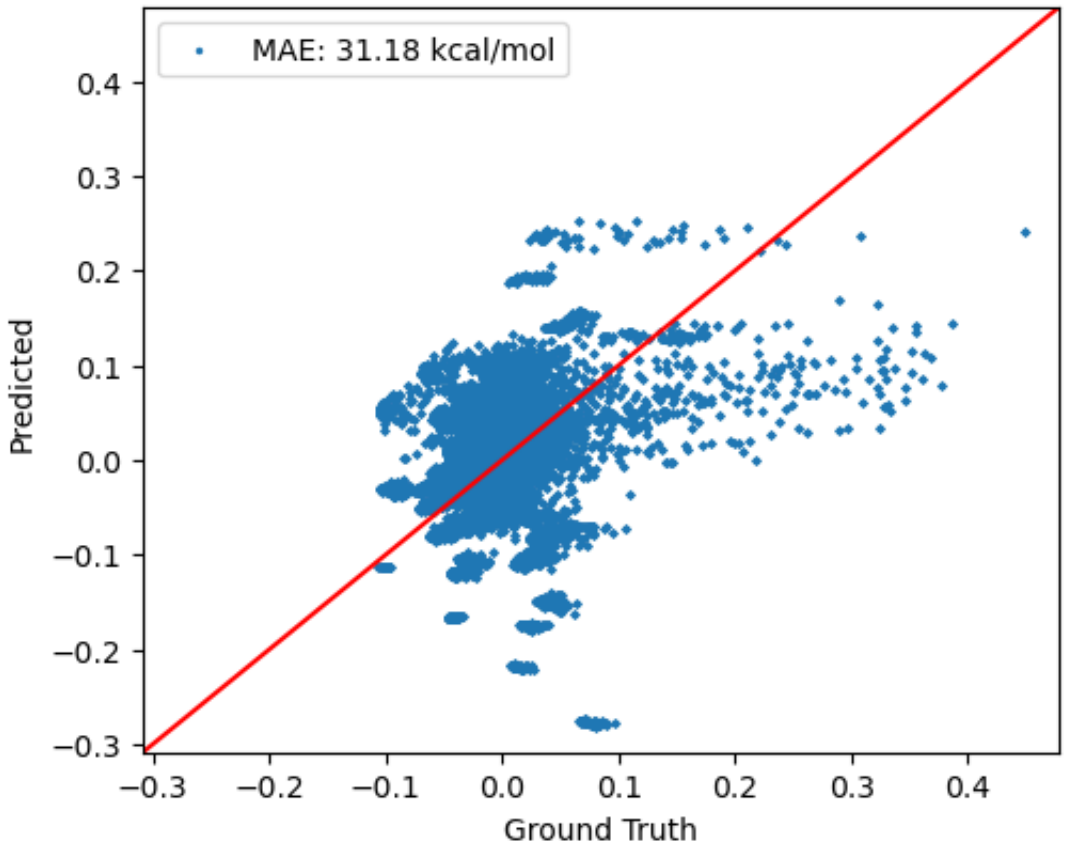
Iteration 6

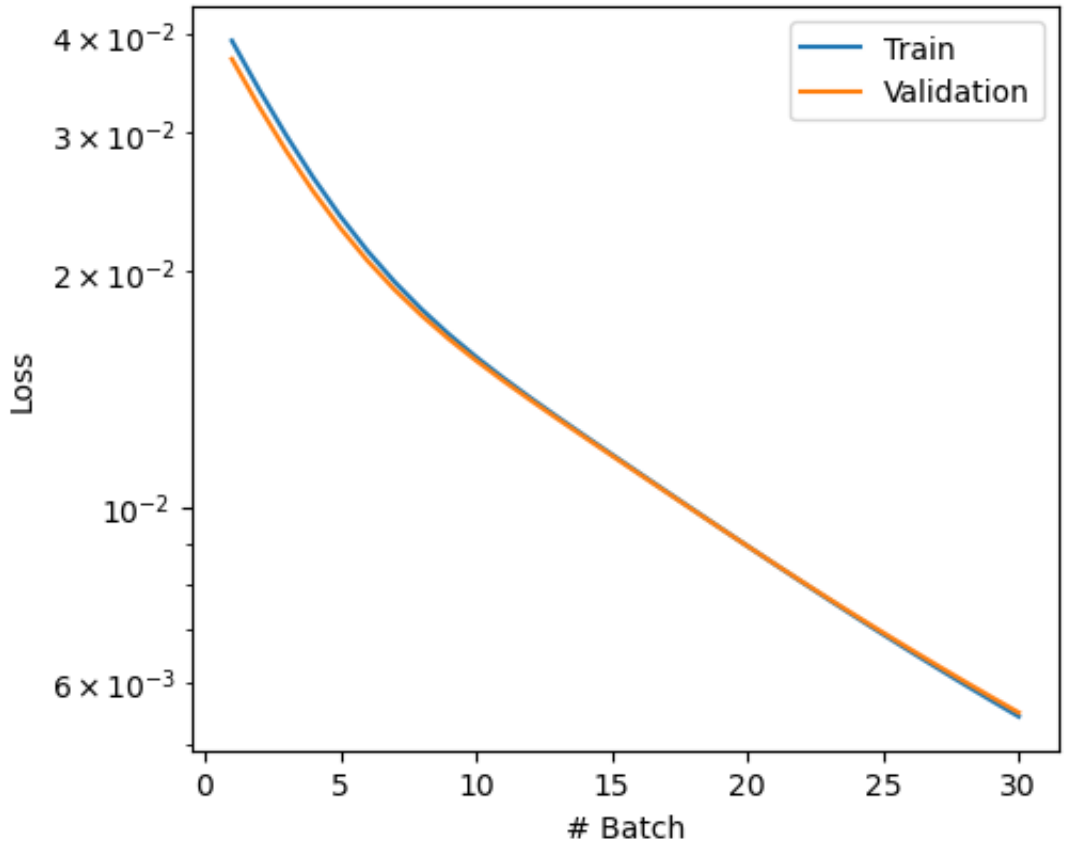
Adding an additional hidden layer and decreasing the batch size and decreasing the learning rate

In [24]:
```
# change to 2 hidden layers
class AtomicNet(nn.Module):
    def __init__(self):
        super().__init__()
        self.layers = nn.Sequential(
            nn.Linear(384, 128),
            nn.ReLU(),
            nn.Linear(128, 64),
            nn.ReLU(),
            nn.Linear(64, 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 torch.ANIModel() to compile atomic networks
ani_net = torchani.ANIModel([net_H, net_C, net_N, net_O])
model = nn.Sequential(
    aev_computer,
    ani_net
).to(device)
```

In [25]:
```
model = model

trainer = ANITrainer(model, learning_rate=1e-6, batch_size = 512, epoch = 30

train_losses, val_losses = trainer.train(train_data, val_data)
```

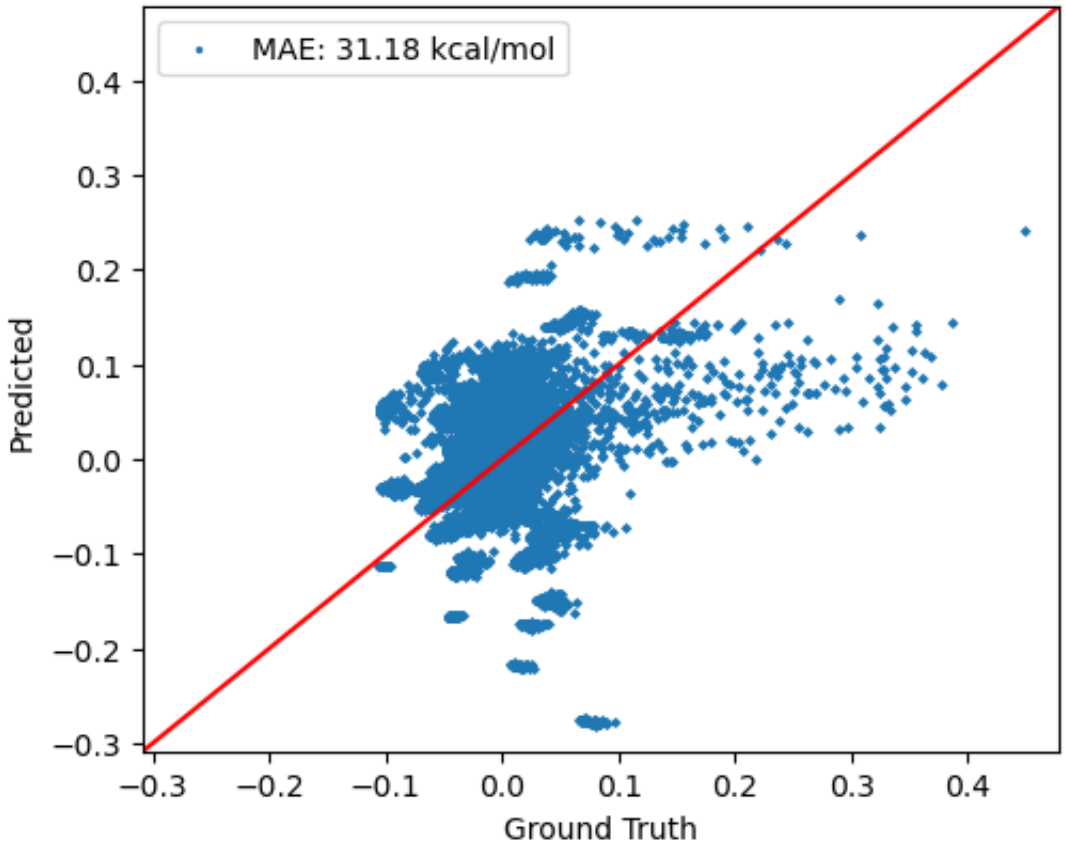
```
Sequential - Number of parameters: 230404
Initialize training data...
100%|██████████| 30/30 [05:12<00:00, 10.42s/it]
```


MAE: 31.18 kcal/mol
Predicted
Ground Truth
0.4
0.3
0.2
0.1
0.0
−0.1
−0.2
−0.3
−0.3
−0.2
−0.1
0.0
0.1
0.2
0.3
0.4


Train
Validation
Loss
Batch
4×10^{-2}
3×10^{-2}
2×10^{-2}
10^{-2}
6×10^{-3}
0
5
10
15
20
25
30

In [26]:
```
trainer.evaluate(test_data, draw_plot = True)
```

Out[26]:
```
0.005478806803251806
```



In [27]:
```
print(train_losses)
```

```
[0.03929618941770233, 0.034070429020977566, 0.02978522476788199, 0.026300331
83955821, 0.02348309789126103, 0.021210439898901142, 0.019370547304545196,
0.017865402058884282, 0.01661317994193476, 0.015548132281081628, 0.014619789
697872967, 0.013791077278980724, 0.013036152717752358, 0.012337521710276028,
0.011683877044643815, 0.01106817112568961, 0.010486652314335971, 0.009937363
379443571, 0.009418975679104711, 0.008930188727469417, 0.008469817711170233,
0.008036897714337686, 0.007630460965456753, 0.007249356522645049, 0.00689223
2608928757, 0.006557624200998794, 0.006244065058841623, 0.00595021616373908
5, 0.005674777854223976, 0.005416390709110152]
```

From iteration 6, we can see that that the loss only decreased to 0.005416390709110152 which is still not as optimal as iteraiton 1. The MAE of 31.18 kcal/mol is again not optimal.

Conclusions

From 6 iterations of modifying the atomic net architecture and tuning the hyperparameters, I can see that what helped optimize the loss most was the learning rate. Decreasing the learning rate helped stabilize oscillation in the loss functions that were seen when the learning rate was 1e-3. Furthermore, decreasing the learning rate to 1e-5 helped decrease the overall loss to its lowest point of 1.1774181918935158e-05 and led to a test/val MAE of 1.30 kcal/mol.

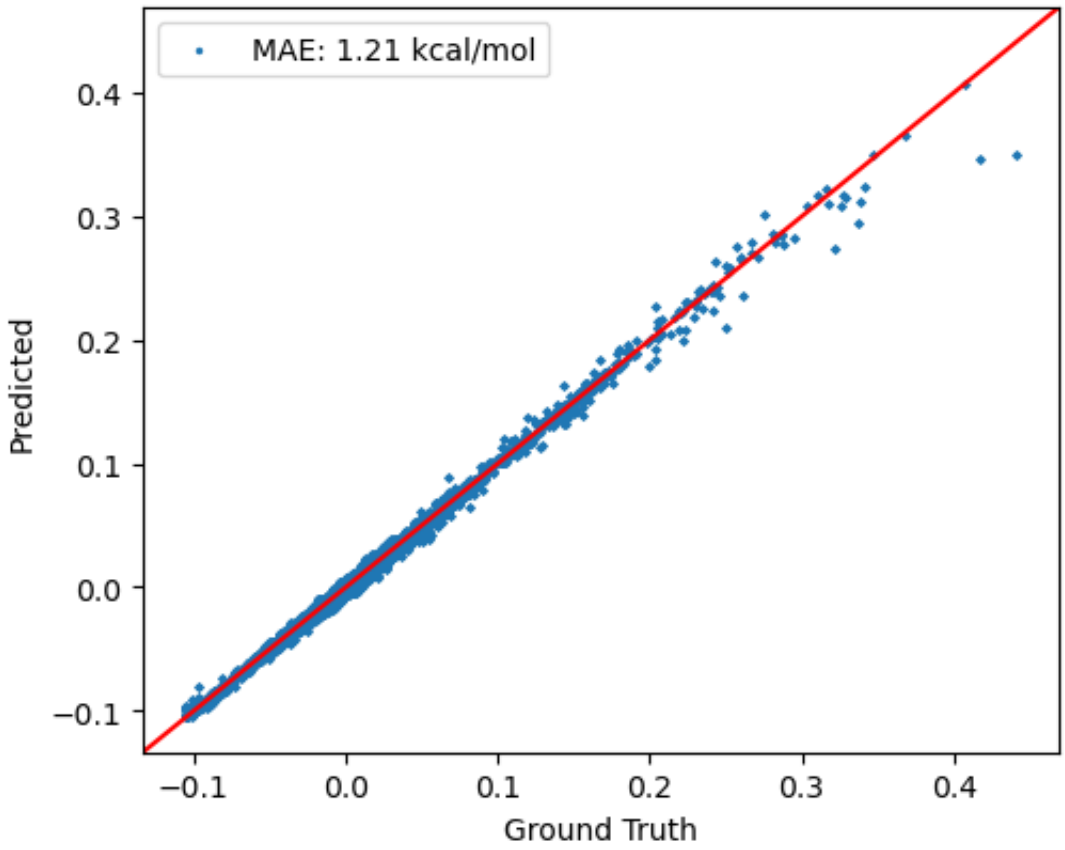
Checkpoint 4

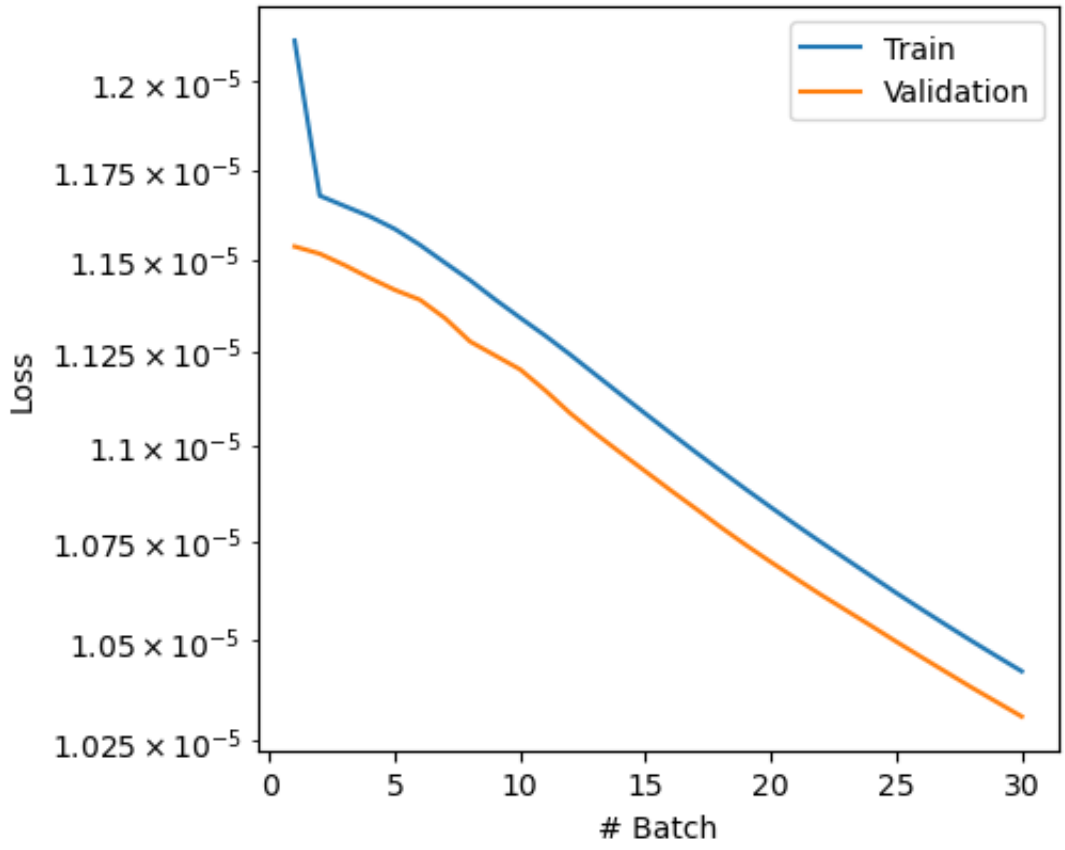
The best model architecture was from Iteration 1, having a learning rate of 1e-5, batch size of 8192, l2 of 1e-5 and 30 epochs.

In [21]:
```
ani_net = torchani.ANIModel([net_H, net_C, net_N, net_O])
model = nn.Sequential(
    aev_computer,
    ani_net
).to(device)
```

In [22]:
```
model = model

# tried decreasing the learning rate
trainer = ANITrainer(model, learning_rate=1e-5, batch_size = 8192, epoch = 3

train_losses, val_losses = trainer.train(train_data, val_data)
```

```
Sequential - Number of parameters: 197636
Initialize training data...
100%|██████████| 30/30 [04:47<00:00,  9.58s/it]
```


MAE: 1.21 kcal/mol
Predicted
Ground Truth
0.4
0.3
0.2
0.1
0.0
−0.1
−0.1
0.0
0.1
0.2
0.3
0.4


Train
Validation
Loss
Batch
1.2×10^{-5}
1.175×10^{-5}
1.15×10^{-5}
1.125×10^{-5}
1.1×10^{-5}
1.075×10^{-5}
1.05×10^{-5}
1.025×10^{-5}
0
5
10
15
20
25
30

In [ ]: