Fine Tuning ANI Architecture To Achieve DFT Accuracy

Introduction

It's one thing to train a neural network to predict the potential energy of one molecular system, it's another to have that model be able to calculate neural network potentials (NNPs) for a diverse set of complex chemical environments. Excitingly, deep learning techniques are making it possible to achieve such feats of transferability. Behler and Parrinello achieved small scale transferability by using symmetry functions, exploiting the fact that properties of atoms in a molecule are predominantly influenced by its neighbors (i.e., chemical locality).[1] Ten years later, in 2017, Smith et al. improved upon the symmetry function design with atomic environment vectors (AEV) that enable transferability at a much larger scale [2]. Their ANI-1 design uses the assumption that total energy can be decomposed into atomic energies to create a separate atomic network for each atom type (H, C, N, O) and thus predict the total atomic energy. Symmetry functions compute atomic features (i.e., radial and angular environments of atoms), which make up atomic environment vectors that serve as input to the atomic networks. The atomic networks' outputs, the individual energies, are summed to form the predicted total atomic energy of a molecule, which is then compared to the true energy calculated through density functional theory (DFT). Furthermore, the high accuracy of ANI-1 compared to semi-empirical QM methods (DFTB and PM6) serves as a significant advancement in expediting computationally expensive QM methods without sacrificing much accuracy.[2] Given that ANI-1 is a potential revolutionary solution to DFT and even *ab initio* level theory, the purpose of this research is to study the ANI-1 model and explore what architectural changes and what efforts to hyperparameter tune atomic nets could improve error. This research aims to achieve < 1.0 kcal/mol MAE.

Methods

The AEVs were initialized according to Smith et al. configurations. I loaded their data (from the GDB dataset) on small organic molecules containing 4 atom types (H, C, N, O).[3] The data was broken into 80% training, 10% validation, and 10% test sets. To compare my finalized model to Smith et al., who calculated RMSE for molecules with 4 (and more) heavy atoms, I used data for molecules with up to 4 heavy atoms (C, N, O). The atomic nets for each atom type were compiled using the torchani [4] package in Python. More generally, atomic nets were built, trained, and evaluated using Pytorch [5]. The AEVs served as inputs to the compiled atomic nets.

The code for training the ANI model allowed for manipulation of L2 regularization, batch size, and learning rate. The ADAM optimization algorithm was used to adjust weights and biases during back propagation. All models start with 384 input nodes (this is the input_dim of AEV) and end with 1 (predicted total energy). First, a baseline model was created for comparison purposes. Next, I tried various architectural manipulations for resblock type neural nets, including implementing skip connections, changing activation functions, adding dropout layers, reducing the size of hidden layers, and increasing the number of hidden layers. Hyperparameters remained constant during this period of architectural changes. Architectural manipulations that reduced MAE were combined into one of the final models.

Both loss per batch and loss per epoch were captured as MSE in training and validation loss curves, where the batch loss curve was not adjusted for the mismatch in number of batches between validation and training. After every training, a model was evaluated on the validation set to calculate MAE and create a prediction error plot. Once a model was in its final form, the model was evaluated on the test set in addition to the validation set. Hyperparameters of the final models were then tuned to achieve the lowest MAE possible given the time constraint of running models on 1 hour sessions imposed by SAVIO (Berkeley Computing Lab).

Results and Discussion

ANI Model Name	**Test Variable**	**Hidden Layers**	**Hidden Layer Size**	**Activation Function**	**Batch Size**	**Lear n-ing Rate**	**Epoch**	**L2**	**MAE (kcal/ mol)**
simplemodel	Baseline model	1	128	ReLU	8192	1e-3	20	1e-5	1.82
resmodel	Skip connection	2	128	ReLU	8192	1e-3	20	1e-5	2.87
plain_model	No Skip connection	2	128	ReLU	8192	1e-3	20	1e-5	3.39
resdropoutmodel	Skip + Dropout	2	128	ReLU	8192	1e-3	20	1e-5	4.62
moreblocksmodel	Adding resblock	4	128	ReLU	8192	1e-3	20	1e-5	1,82
Lsizeblocksmodel	Increasing hidden	2	192	ReLU	8192	1e-3	20	1e-5	1.79

	size of resblock								
Ssizeblocksmodel	Decreasing hidden size of resblock	2	64	ReLU	8192	1e-3	20	1e-5	2.63
leakyresmodel	Changing Activation	2	128	Leaky ReLU	8192	1e-3	20	1e-5	2.47
tanhresmodel	Changing Activation	2	128	Tanh	8192	1e-3	20	1e-5	2.94

Table 1: Preliminary Stage Models. Highlights changes made to find best NN architectures for ANI.

As seen in Table 1, the biggest improvements to MAE for the basic resmodel came from adding an additional resblock, increasing the hidden layer size of the resblock, and changing the activation function to leaky ReLU. Unfortunately, I couldn't use bigger hidden layer sizes for a NN with 2 resblocks because of the time constraints associated with SAVIO (I couldn't train within a 1 hour time frame). I had to opt for smaller size hidden layers, which luckily also happened to improve upon error of the basic resmodel. The Python code attached to this research will show learning curve plots for the preliminary stage models with significant oscillations, but these subsided during the final stage with the use of decreased learning rates and batch sizes.

Final Model One, which I call bestresmodel, incorporated the best practices I discovered from manipulating architectures. It uses an additional resblock and leaky ReLU. Unsurprisingly, Smith et al. also settled on 4 hidden layers for some of their models. Dropout, which did not improve MAE in the preliminary stage model, actually helped reduce error in the bestresmodel possibly because dropout works better over longer training periods. Final Model Two is my baseline model set to the same hyperparameters as Final Model One except for the number of epochs. Final Model Three is the Lsizeblocksmodel, which achieved a lower MAE than the simplemodel during the preliminary stage.

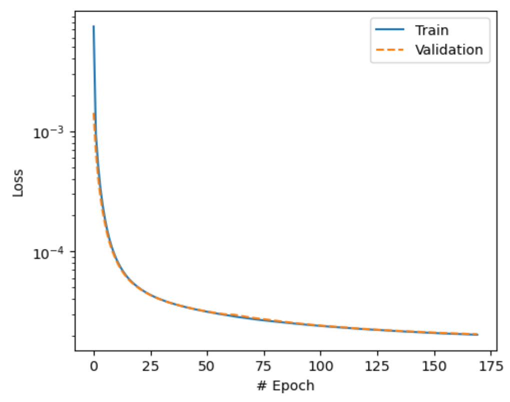
ANI Model Name	**D.O.**	**Skip Conn**	**Hidden**	**Hidden**	**Activation**	**Batch Size**	**Learning**	**Epoch**	**L2**	**MAE Val,**	**Train Time**

			Laye rs	Laye r Size	Funct ion		Rate			Test	(min:sec)
Final Model One/bestresmodel	yes	yes	4	32	Leaky ReLU	1024	1e-5	170	1e-5	1.10, 1.10	50:10
Final Model Two/simplemodel/ baseline	no	no	1	128	ReLU	1024	1e-5	200	1e-5	0.92, 0.91	45:11
Final Model Three/Lsizeblocks model	no	yes	2	192	ReLU	1024	1e-5	170	1e-5	0.91, 0.91	51:36

Table 2: Final Stage Models. Models with less hidden layers take the cake.

As seen in Table 2, for the final models' hyperparameters, batch size and learning rate were reduced to prolong training, and epochs were then increased accordingly. It was found that an L2 regularization value of 1e-4 negatively impacts the models' performance (this data is not present in the Python code). I configured the epochs of the models to max out at 50 minutes of training whilst trying to plateau the learning curves. Given the time constraint of training within 1 hour, the simplemodel outperformed the more complex bestresmodel. It can be reasonably assumed that without the time constraint and over longer training intervals, the bestresmodel could achieve sub 1.0 MAE or outperform the simplemodel, especially with greater hidden layer sizes.

The time-based relationship between the simplemodel versus the bestresmodel alludes to well-known commentary on machine learning. I had to decrease the capability of the bestresmodel in terms of hidden layer sizes to have it train within a fixed time frame, and the capability was decreased so much that it underperforms compared to the basic neural nets used to compile the simplemodel ANI. This showcases that sometimes in machine learning simpler architectures are more time efficient or more effective in general than more complex architectures.



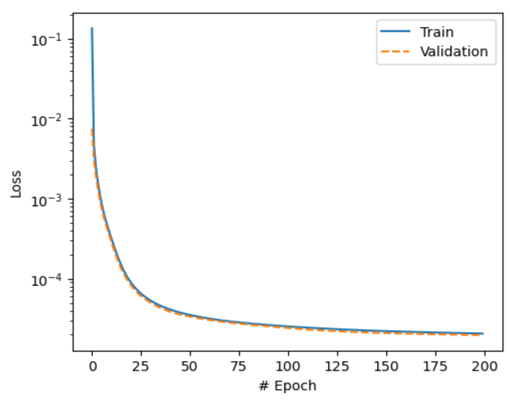
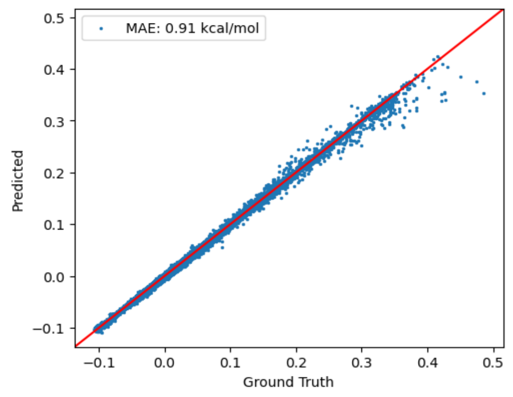


Figure 1: Final Model Two (right) & Three (left) Learning Curves.



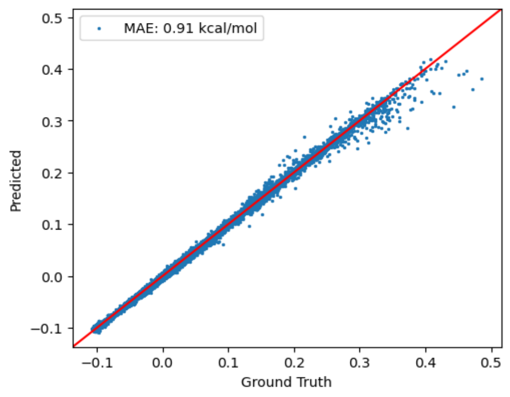


Figure 2: Prediction Error Plots For Final Models Two (right) & Three (left). Ground Truth represents true energies (DTF) and Predicted represents predicted energies.

Final Model Two and Final Model Three successfully achieved < 1 MAE kcal/mol. They do so with no significant signs of overfitting or underfitting as Figure 1 shows qualitatively (the validation and training curves are tightly close together). Within models, in Table 2, each model calculates the same value or very close to the same MAE for validation and testing, which also indicates very little overfitting or underfitting is present. Across models, in Figure 2, we see testing MAE is the same for both Final Models Two and Three; Final Models Two and Three perform very similarly with 0.91 MAE on the test set. Although MAE is not directly convertible to RMSE, 0.91 MAE is much better than Smith et al.'s 26.8 RMSE kcal/mol on organic molecules with up to 4 heavy atoms. A 0.91 MAE also seems on par or better than Smith et al.'s final fitness of 1.3 RMSE kcal/mol, which they achieved by using a greater portion of the dataset.

The last two final models worked surprisingly well and outperformed Smith et al.'s usage of 3 to 4 hidden layers. It can be argued the baseline model is the best model in terms of achieving the lowest MAE in the *shortest amount of time*. However, based on the downward trajectory of the learning curves for Final Model Three, this model would be the best in terms of having the lowest MAE if allowed to train for more epochs (i.e., not under time constraint).

Conclusion

The three major findings of this study are: 1) by using ANI, it's possible to achieve < 1 MAE when predicting total energies of a large variety of organic molecules up to 4 heavy atoms (i.e., large scale transferability); 2) it's possible to achieve < 1 MAE with very simple neural network structures; and 3) although increasing the number of hidden layers and using dropout is useful, simpler NN architectures outperformed the deeper design when constrained by time. These findings showcase that ANI could be used to predict energies very fast and at very high accuracy, which is acceptable in comparison to DFT accuracy. ANI models were trained on DFT to copy Smith et al. but could be used to predict *ab initio* methods too. This research did not exhaust the list of architectural changes and hyperparameter tuning needed to create the most accurate ANI model possible. Future improvements to this research might include training for more epochs, trying batch normalization, and using varying hidden layer sizes (i.e., pyramidal structure). Also, other NN architectures, such as graph neural networks, could prove more accurate because they are specifically designed for locality-centric problems like predicting total energy of a molecule.

References

1. Behler, J.; Parrinello, M. Generalized Neural-Network Representation of High-Dimensional Potential-Energy Surfaces. *Phys. Rev. Lett.* **2007**, *98* (14), 146401. https://doi.org/10.1103/PhysRevLett.98.146401.
2. Smith, J. S.; Isayev, O.; Roitberg, A. E. ANI-1: An Extensible Neural Network Potential with DFT Accuracy at Force Field Computational Cost. *Chem. Sci.* **2017**, *8* (4), 3192–3203. https://doi.org/10.1039/C6SC05720A.
3. Smith, J. S.; Isayev, O.; Roitberg, A. E. ANI-1, A Data Set of 20 Million Calculated off-Equilibrium Conformations for Organic Molecules. *Sci Data* **2017**, *4* (1), 170193. https://doi.org/10.1038/sdata.2017.193.
4. Gao, X.; Ramezanghorbani, F.; Isayev, O.; Smith, J. S.; Roitberg, A. E. TorchANI: A Free and Open Source PyTorch-Based Deep Learning Implementation of the ANI Neural Network Potentials. *J. Chem. Inf. Model.* **2020**, *60* (7), 3408–3415. https://doi.org/10.1021/acs.jcim.0c00451.
5. Paszke, A.; Gross, S.; Massa, F.; Lerer, A.; Bradbury, J.; Chanan, G.; Killeen, T.; Lin, Z.; Gimelshein, N.; Antiga, L.; Desmaison, A.; Kopf, A.; Yang, E.; DeVito, Z.; Raison, M.; Tejani, A.; Chilamkurthy, S.; Steiner, B.; Fang, L.; Bai, J.; Chintala, S. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems*; Wallach, H., Larochelle, H., Beygelzimer, A., Alché-Buc, F. d', Fox, E., Garnett, R., Eds.; Curran Associates, Inc., **2019**; Vol. 32.

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 torch.nn.functional as F
import torchani
import matplotlib.pyplot as plt
```

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=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(0.8, 0.1, 0.1)
```

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 [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
```

Checkpoint 2: Code For Training Network

In [5]:
```
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(), lr=learning_rate, weight
        self.epoch = epoch
        self.device = device

    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(self.batch_size).cache()

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

        # record epoch and batch losses
        train_loss_list = []
        val_loss_list = []
        batch_loss_list_train = []
        all_batch_loss_val = []
        lowest_val_loss = np.inf

        for epoch in tqdm(range(self.epoch), desc="Training Epochs"):
            train_epoch_loss = 0.0
            for train_data_batch in train_data_loader:
                # compute energies
                species = train_data_batch['species'].to(self.device)
                coordinates = train_data_batch['coordinates'].to(self.device)
                true_energies = train_data_batch['energies'].to(self.device).float()
                _, pred_energies = self.model((species, coordinates))

                # compute loss and append batch loss
                batch_loss = loss_func(pred_energies, true_energies)
                batch_loss_list_train.append(batch_loss.detach().cpu().numpy())

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

                batch_importance = len(train_data_batch) / len(train_data_loader)
                train_epoch_loss += batch_loss.detach().cpu().item() * batch_importanc
            # use the self.evaluate to get loss on the validation set
            val_epoch_loss, epoch_batch_loss_list_val = self.evaluate(val_data)
            all_batch_loss_val.extend(epoch_batch_loss_list_val)

            # 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=True)
            ax.set_yscale("log")
            # Plot train loss and validation loss
            ax.plot(np.arange(len(batch_loss_list_train)), batch_loss_list_train, labe
            #val_batches_scaled = np.linspace(0, len(batch_loss_list_train) - 1, num=l
            ax.plot(np.arange(len(all_batch_loss_val)), all_batch_loss_val, label='Val
            ax.legend()
            ax.set_xlabel("# Batch")
            ax.set_ylabel("Loss")

            fig, ax = plt.subplots(1, 1, figsize=(5, 4), constrained_layout=True)
            # Plot train loss and validation loss
            ax.set_yscale("log")
            ax.plot(np.arange(len(train_loss_list)), train_loss_list, label='Train')
            ax.plot(np.arange(len(train_loss_list)), val_loss_list, label='Validation'
            ax.legend()
            ax.set_xlabel("# Epoch")
            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(self.batch_size).cache()

        # init loss function and record batch loss
        loss_func = nn.MSELoss()
        total_loss = 0.0
        batch_loss_list_val = []

        if draw_plot:
            true_energies_all = []
            pred_energies_all = []

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

                # compute energies
                species = batch_data['species'].to(self.device)
                coordinates = batch_data['coordinates'].to(self.device)
                true_energies = batch_data['energies'].to(self.device).float()
                _, pred_energies = self.model((species, coordinates))

                # compute loss and append batch loss list
                batch_loss = loss_func(pred_energies, true_energies)
                batch_loss_list_val.append(batch_loss.detach().cpu().numpy())

                batch_importance = len(batch_data) / len(data_loader)
                total_loss += batch_loss.detach().cpu().item() * batch_importance

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

        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
            # converted it to kcal/mol when reporting the mean absolute error
            # 1 hartree = 627.5094738898777 kcal/mol
            # MAE = mean(|true - pred|)
            hartree2kcalmol = 627.5094738898777
            mae = np.mean(np.abs(true_energies_all - pred_energies_all)) * hartree2kca
            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, batch_loss_list_val
```

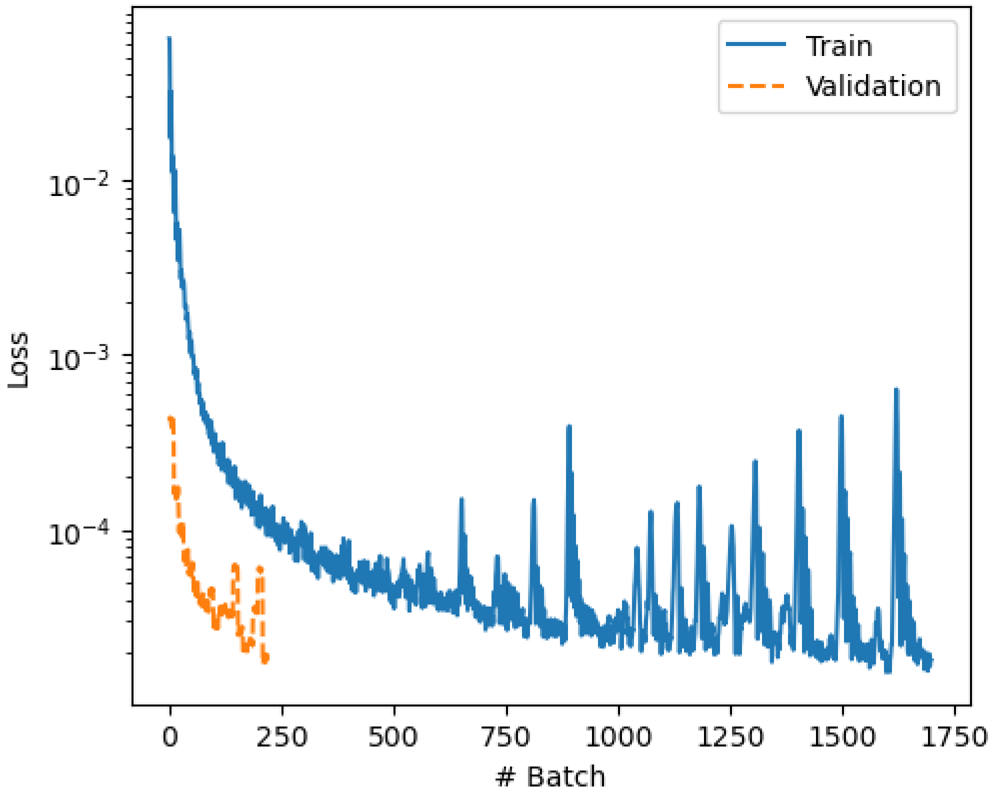
Baseline Model

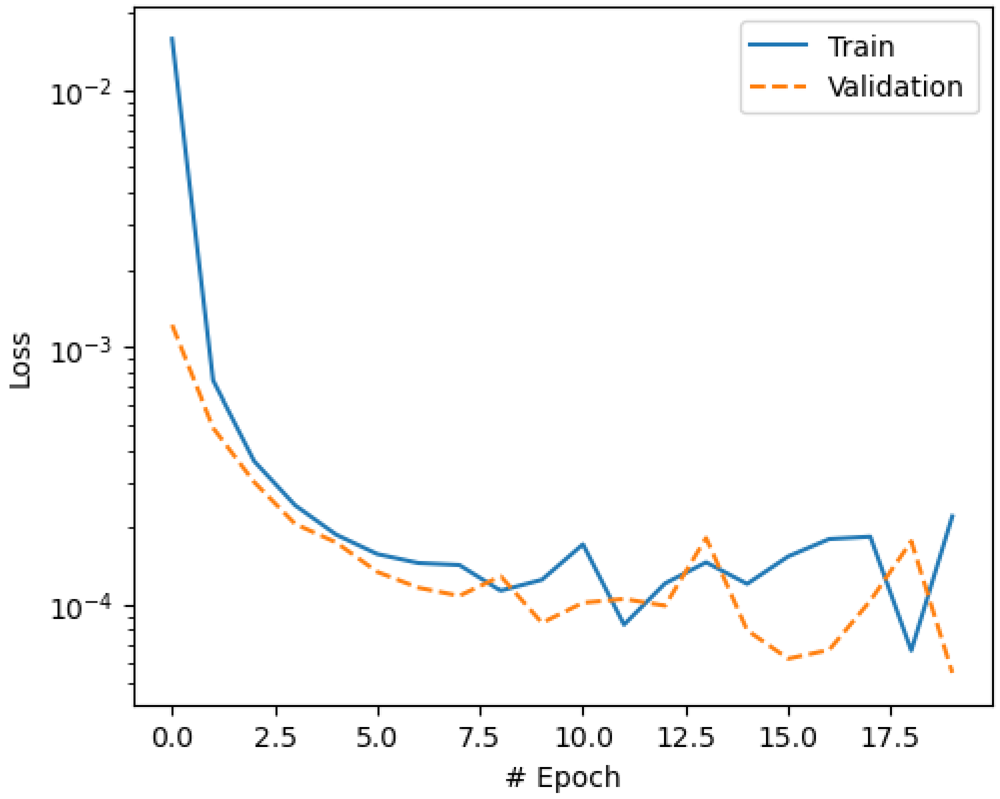
In [6]:
```
class AtomicNet(nn.Module):
    def __init__(self):
        super().__init__()
        self.layers = nn.Sequential(
            nn.Linear(384, 128),
            nn.ReLU(),
            nn.Linear(128, 1)
        )

    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])
simplemodel = nn.Sequential(
    aev_computer,
    ani_net
).to(device)
```

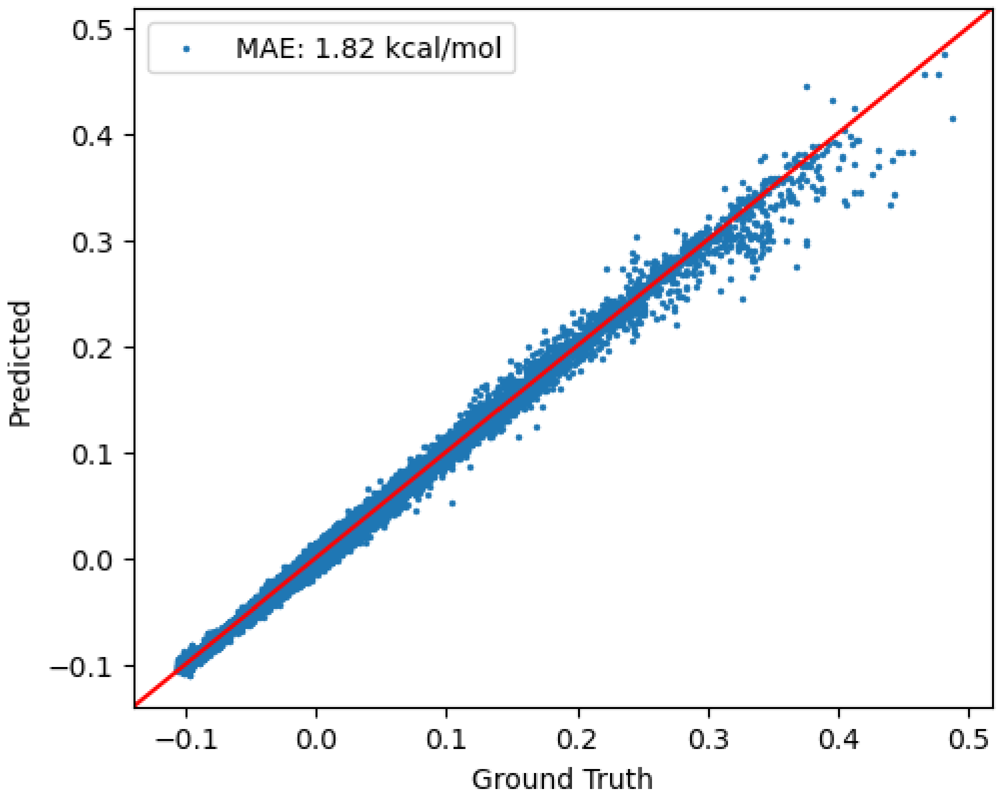
In [8]:
```
#Example

trainer = ANITrainer(simplemodel, batch_size=8192, learning_rate=1e-3, epoch=20, l2=1e

train_loss, val_loss = trainer.train(train_data, val_data)
```

```
Sequential - Number of parameters: 197636
Initialize training data...
Training Epochs: 100%|██████████| 20/20 [03:16<00:00,  9.82s/it]
```





```
In [9]: total_loss, batch_loss = trainer.evaluate(val_data, draw_plot=True)
```



Checkpoint 3 - Architectures and Regularization

Residual Connection, Dropout, More hidden layers, Greater Size of Hidden Layers, etc

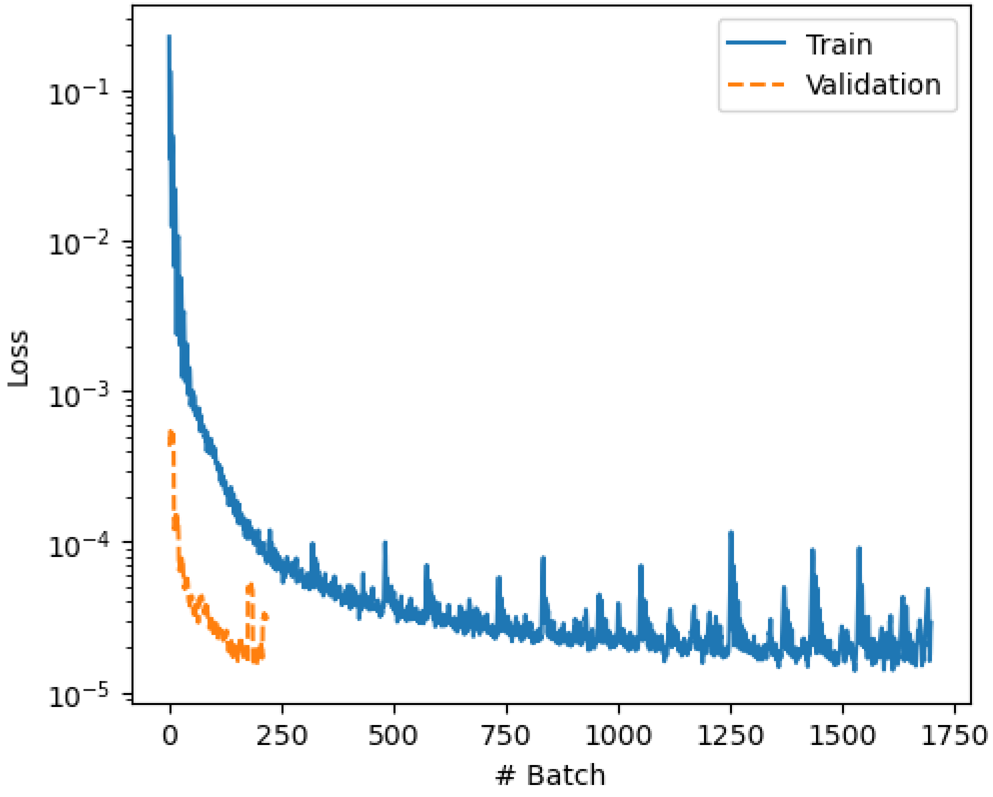
Baseline Model = 1.82 MAE

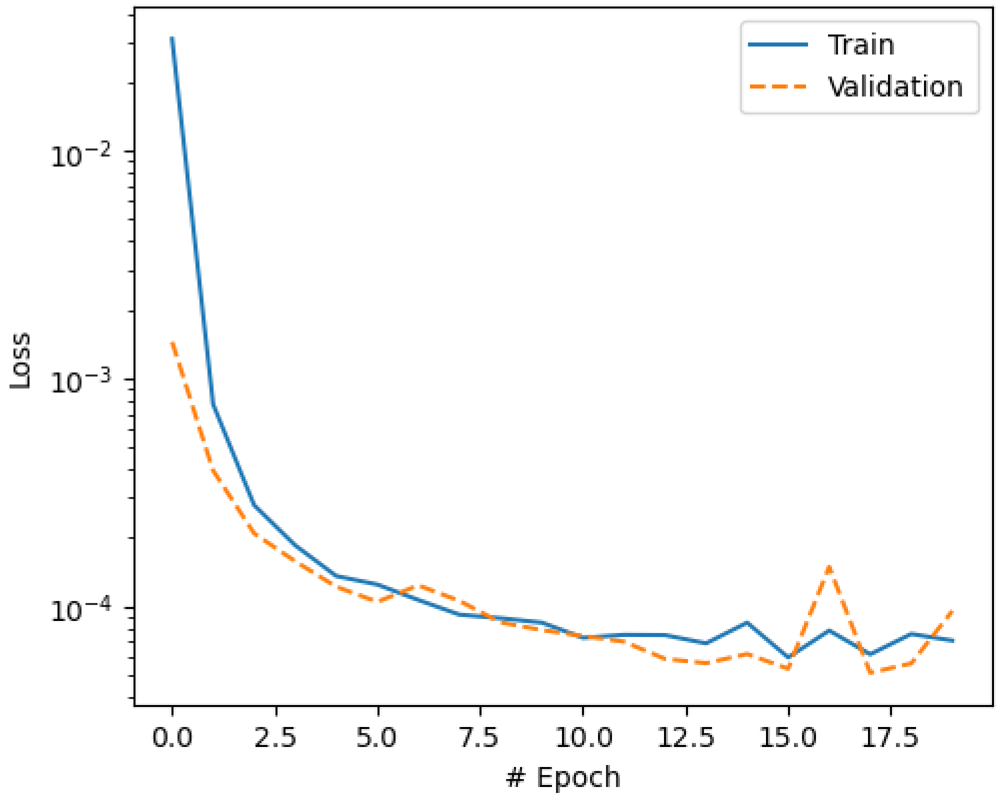
#1 Skip Connection vs no Skip connection

In [12]:
```
class ResidualBlock(nn.Module):
    def __init__(self, input_dim, output_dim):
        super(ResidualBlock, self).__init__()
        self.fc1 = nn.Linear(input_dim, output_dim)
        self.fc2 = nn.Linear(output_dim, output_dim)
        self.activation = nn.ReLU()

    def forward(self, x):
        identity = x

        out = self.fc1(x)
        out = self.activation(out)
        out = self.fc2(out)
        out += identity  # Residual connection
        out = self.activation(out)
        return out

class ResNetAtomic(nn.Module):
    def __init__(self):
        super(ResNetAtomic, self).__init__()
        self.initial_fc = nn.Linear(384, 128)

        self.res_blocks = nn.Sequential(
            ResidualBlock(128, 128)
        )

        self.final_fc = nn.Linear(128, 1)

    def forward(self, x):
        x = F.relu(self.initial_fc(x))
        x = self.res_blocks(x)
        x = self.final_fc(x)
        return x


net_H = ResNetAtomic()
net_C = ResNetAtomic()
net_N = ResNetAtomic()
net_O = ResNetAtomic()

# 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])
resmodel = nn.Sequential(
    aev_computer,
    ani_net
).to(device)
```

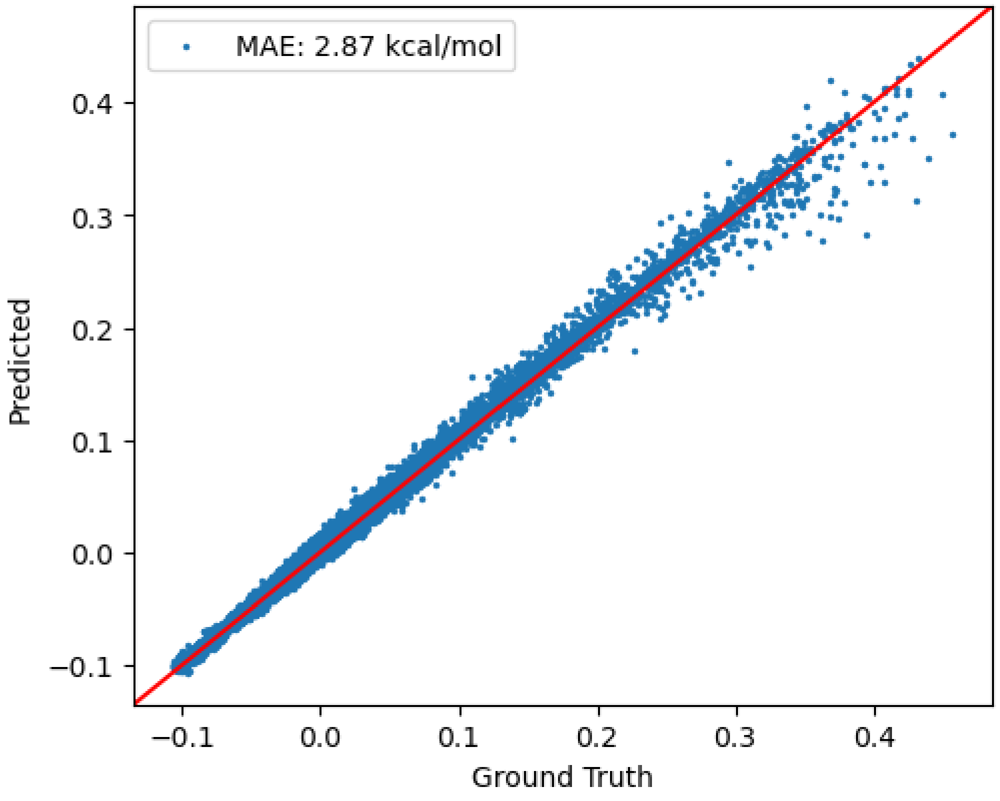
In [13]:
```
trainer = ANITrainer(resmodel, batch_size=8192, learning_rate=1e-3, epoch=20, l2=1e-5)

train_loss, val_loss = trainer.train(train_data, val_data)
```

```
Sequential - Number of parameters: 329732
Initialize training data...
Training Epochs: 100%|██████████| 20/20 [03:38<00:00, 10.92s/it]
```





In [14]:
```
total_loss, batch_loss = trainer.evaluate(val_data, draw_plot=True)
```

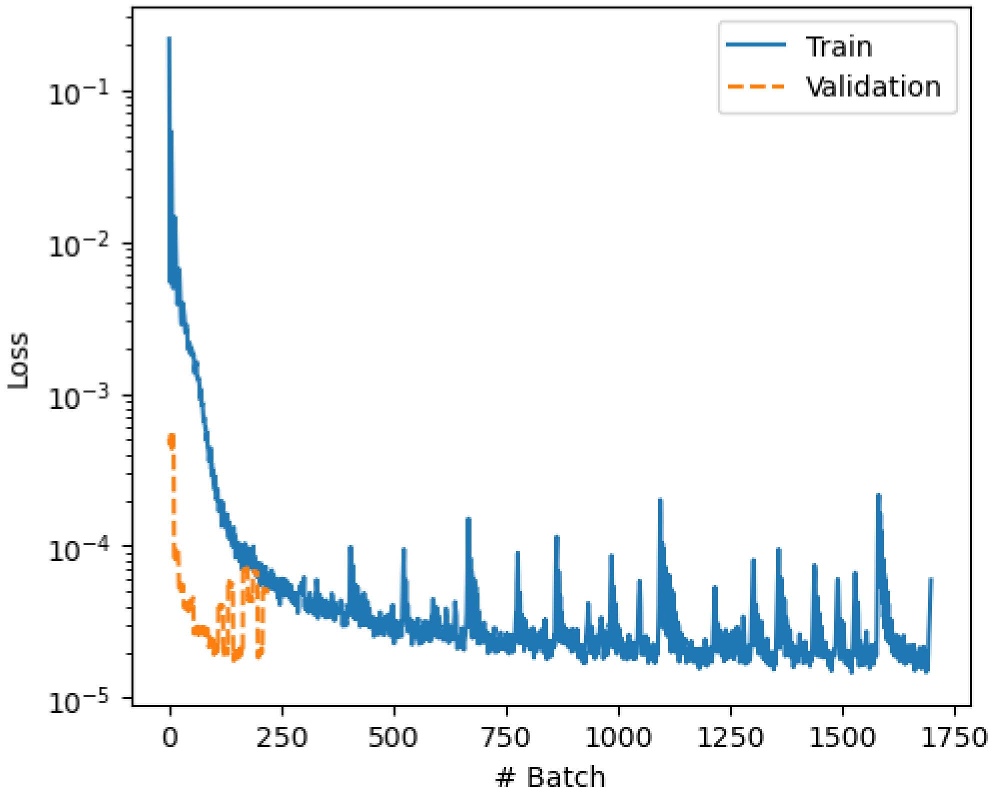


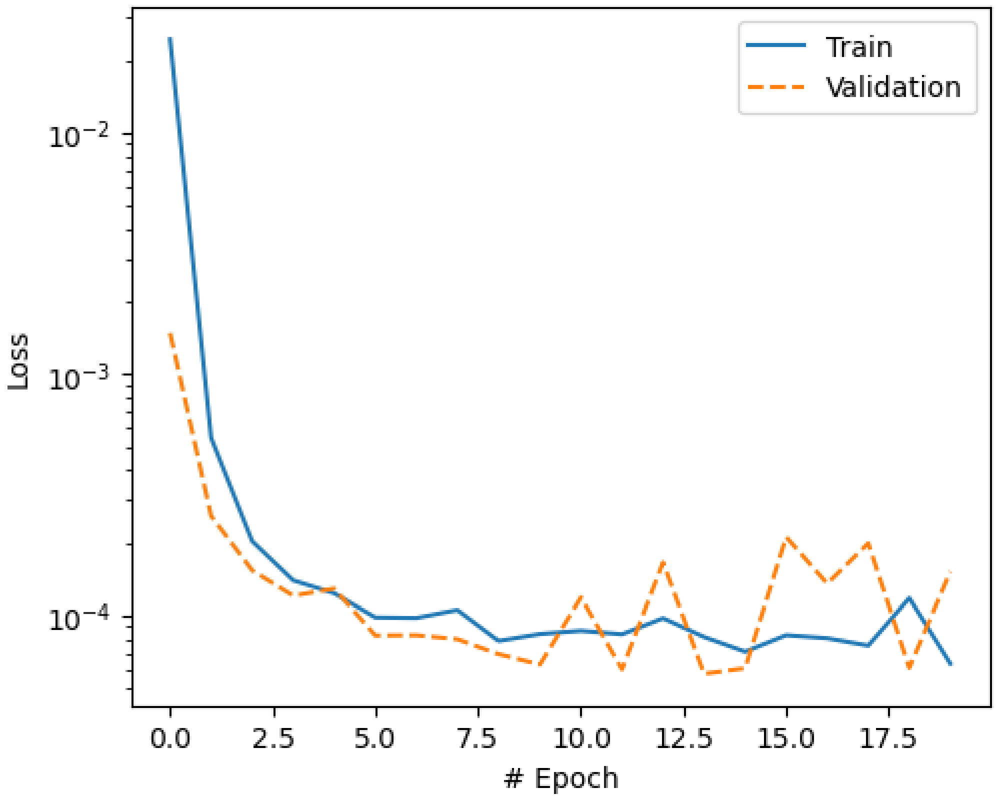
Now with NO skip Connection

In [24]:
```
class PlainBlock(nn.Module): #removed skip connection
    def __init__(self, input_dim, output_dim):
        super(PlainBlock, self).__init__()
        self.fc1 = nn.Linear(input_dim, output_dim)
        self.fc2 = nn.Linear(output_dim, output_dim)
        self.activation = nn.ReLU()

    def forward(self, x):
        x = self.fc1(x)
        x = self.activation(x)
        x = self.fc2(x)
        x = self.activation(x)
        return x

class PlainNetAtomic(nn.Module):
    def __init__(self):
        super(PlainNetAtomic, self).__init__()
        self.initial_fc = nn.Linear(384, 128)

        # Using PlainBlock which does not have residual connections
        self.blocks = nn.Sequential(
            PlainBlock(128, 128)
        )

        self.final_fc = nn.Linear(128, 1)

    def forward(self, x):
        x = F.relu(self.initial_fc(x))
        x = self.blocks(x)
        x = self.final_fc(x)
        return x

net_H = PlainNetAtomic()
net_C = PlainNetAtomic()
net_N = PlainNetAtomic()
net_O = PlainNetAtomic()

# Assuming you need a combined model for all atoms without residual connections
plain_ani_net = torchani.ANIModel([net_H, net_C, net_N, net_O])
plain_model = nn.Sequential(
    aev_computer,
    plain_ani_net
).to(device)
```

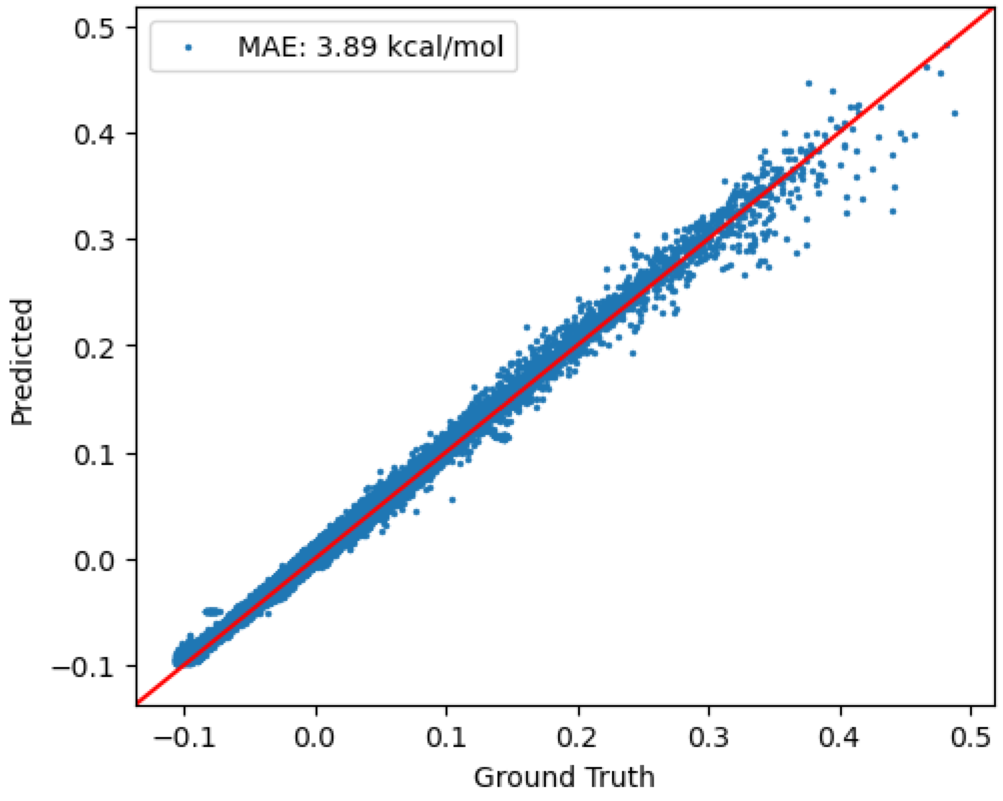
In [25]:
```
trainer = ANITrainer(plain_model, batch_size=8192, learning_rate=1e-3, epoch=20, l2=1e

train_loss, val_loss = trainer.train(train_data, val_data)
```

```
Sequential - Number of parameters: 329732
Initialize training data...
Training Epochs: 100%|██████████| 20/20 [03:36<00:00, 10.82s/it]
```





```
In [26]: total_loss, batch_loss = trainer.evaluate(val_data, draw_plot=True)
```

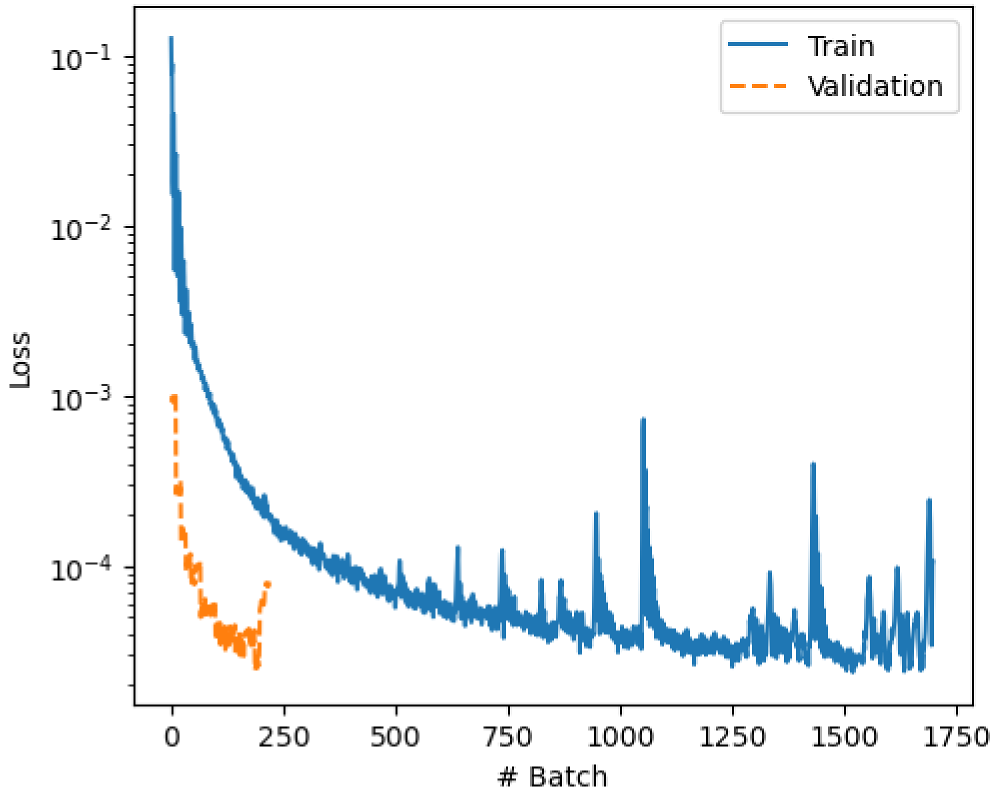


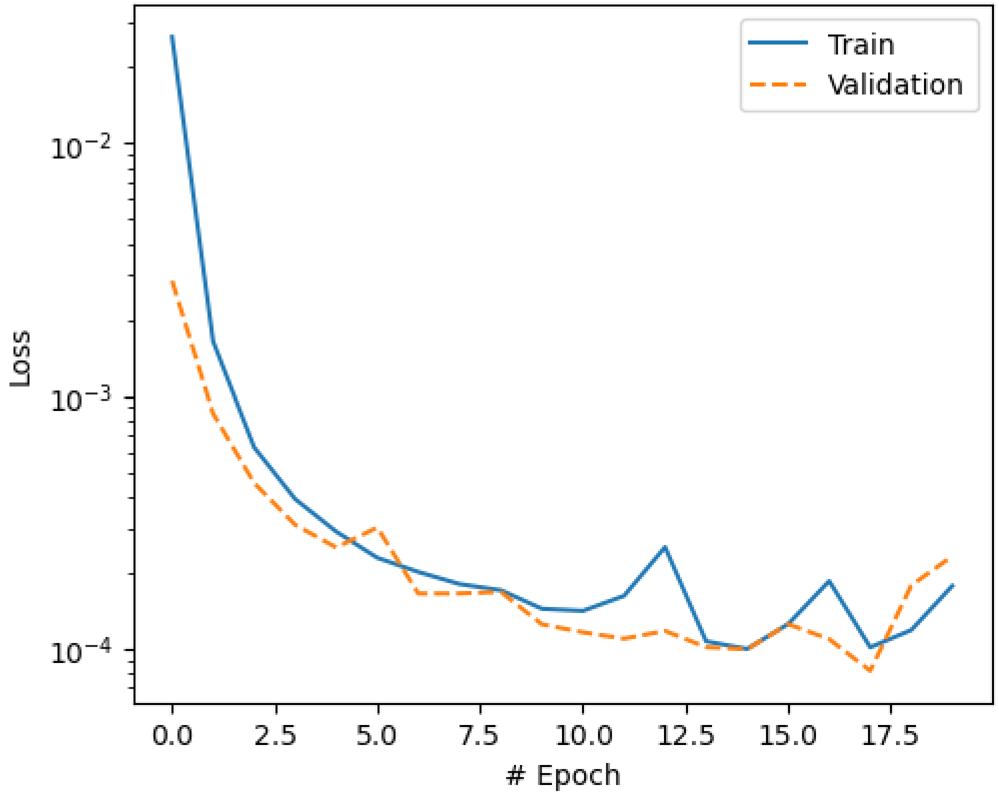
#2 Skip connection + Dropout layer

In [9]:
```python
class ResidualBlockPlusDropout(nn.Module):
    def __init__(self, input_dim, output_dim):
        super(ResidualBlockPlusDropout, self).__init__()
        self.fc1 = nn.Linear(input_dim, output_dim)
        self.fc2 = nn.Linear(output_dim, output_dim)
        self.activation = nn.ReLU()
        self.dropout = nn.Dropout(p = 0.50)

    def forward(self, x):
        identity = x

        out = self.fc1(x)
        out = self.activation(out)
        out = self.dropout(out)
        out = self.fc2(out)
        out += identity  # Residual connection
        out = self.activation(out)
        return out

class ResPlusDropoutAtomic(nn.Module):
    def __init__(self):
        super(ResPlusDropoutAtomic, self).__init__()
        self.initial_fc = nn.Linear(384, 128)

        self.res_blocks = nn.Sequential(
            ResidualBlockPlusDropout(128, 128)
        )

        self.final_fc = nn.Linear(128, 1)
    def forward(self, x):
        x = F.relu(self.initial_fc(x))
        x = self.res_blocks(x)
        x = self.final_fc(x)
        return x


net_H = ResPlusDropoutAtomic()
net_C = ResPlusDropoutAtomic()
net_N = ResPlusDropoutAtomic()
net_O = ResPlusDropoutAtomic()

# 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])
resdropoutmodel = nn.Sequential(
    aev_computer,
    ani_net
).to(device)
```

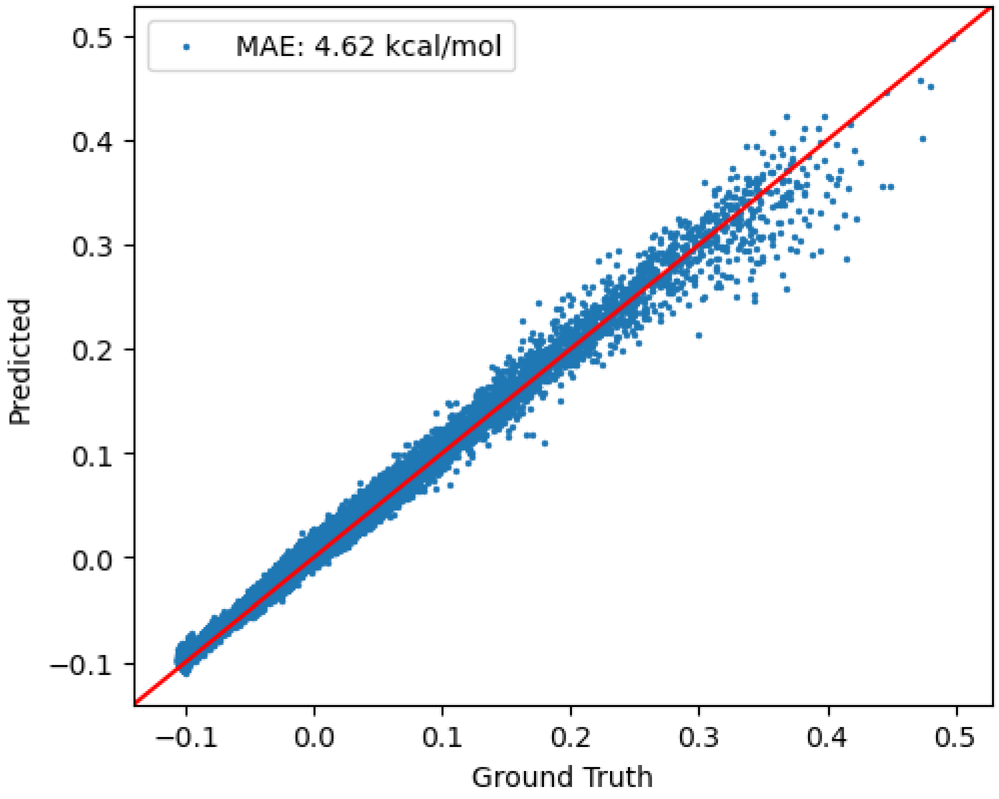
In [10]:
```
trainer = ANITrainer(resdropoutmodel, batch_size=8192, learning_rate=1e-3, epoch=20, l

train_loss, val_loss = trainer.train(train_data, val_data)
```

```
Sequential - Number of parameters: 329732
Initialize training data...
Training Epochs: 100%|██████████| 20/20 [03:47<00:00, 11.35s/it]
```





In [11]:
```
total_loss, batch_loss = trainer.evaluate(val_data, draw_plot=True)
```

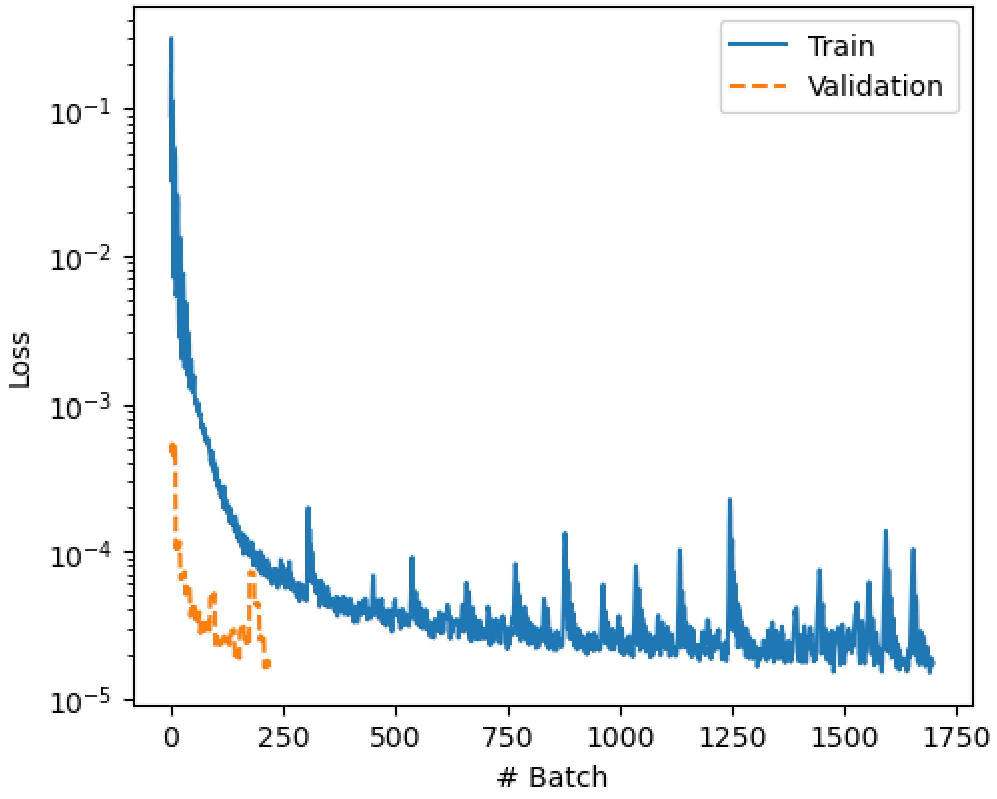


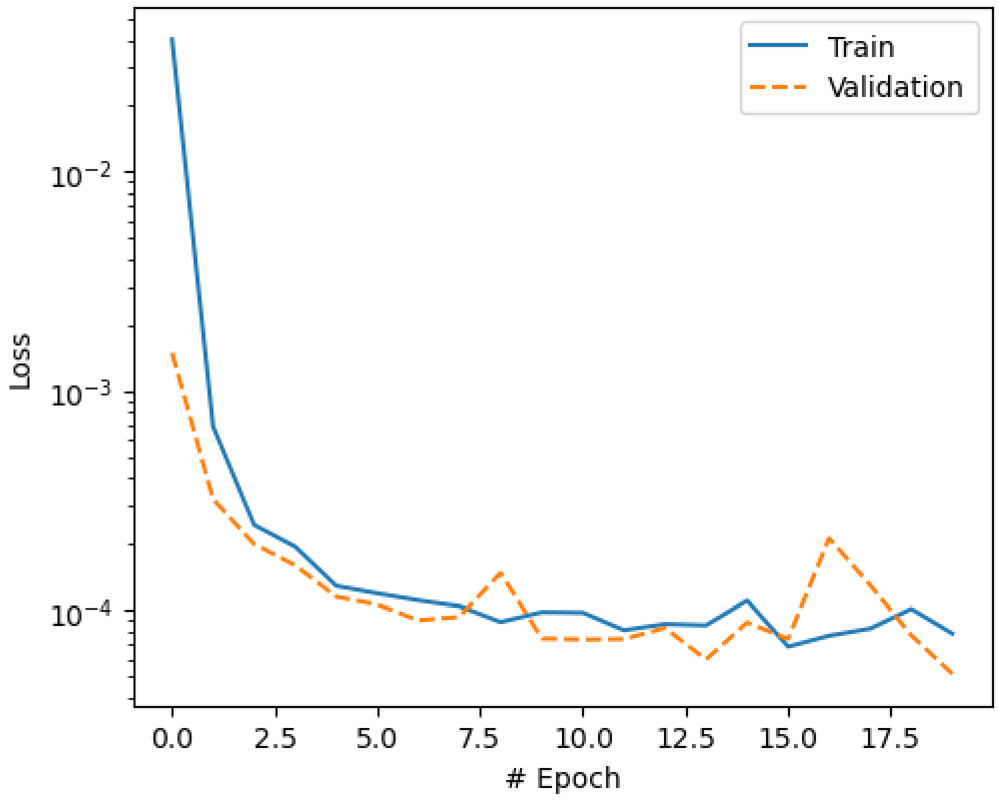
#3 More Resblocks? Adding one more residual block

In [30]:
```
class ResidualBlock(nn.Module):
    def __init__(self, input_dim, output_dim):
        super(ResidualBlock, self).__init__()
        self.fc1 = nn.Linear(input_dim, output_dim)
        self.fc2 = nn.Linear(output_dim, output_dim)
        self.activation = nn.ReLU()

    def forward(self, x):
        identity = x

        out = self.fc1(x)
        out = self.activation(out)
        out = self.fc2(out)
        out += identity  # Residual connection
        out = self.activation(out)
        return out

class MoreBlocksAtomic(nn.Module):
    def __init__(self):
        super(MoreBlocksAtomic, self).__init__()
        self.initial_fc = nn.Linear(384, 128)

        self.res_blocks = nn.Sequential(
            ResidualBlock(128, 128),
            ResidualBlock(128, 128)
        )

        self.final_fc = nn.Linear(128, 1)

    def forward(self, x):
        x = F.relu(self.initial_fc(x))
        x = self.res_blocks(x)
        x = self.final_fc(x)
        return x


net_H = MoreBlocksAtomic()
net_C = MoreBlocksAtomic()
net_N = MoreBlocksAtomic()
net_O = MoreBlocksAtomic()

# 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])
moreblocksmodel = nn.Sequential(
    aev_computer,
    ani_net
).to(device)
```

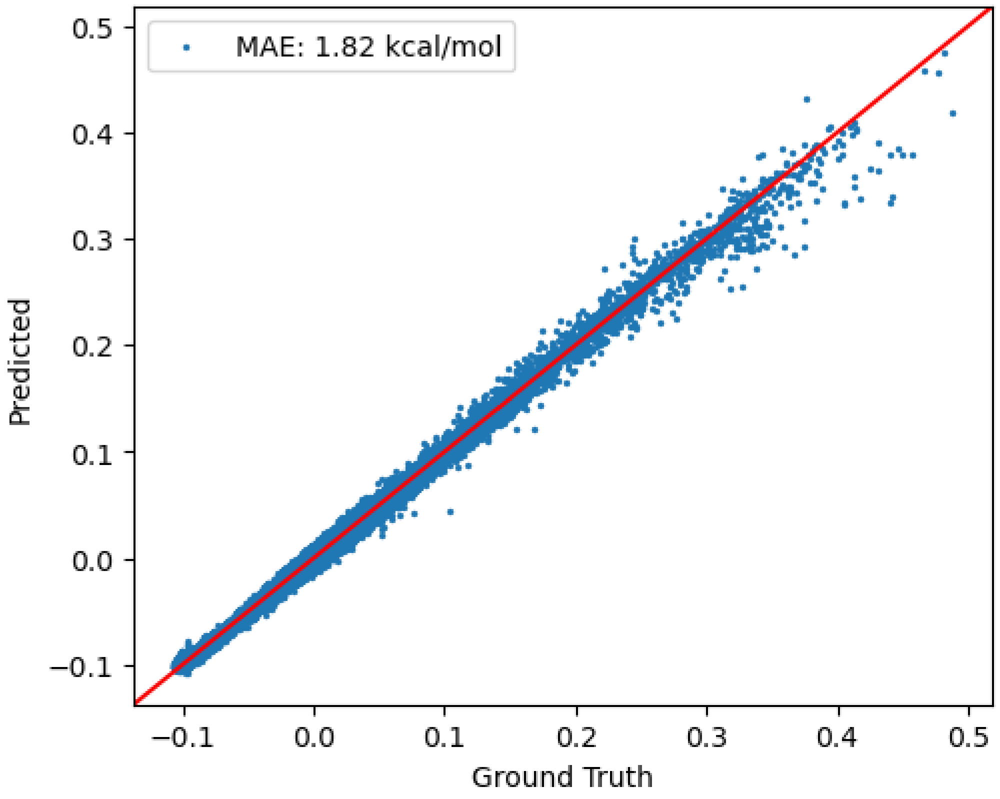
In [31]:
```
trainer = ANITrainer(moreblocksmodel, batch_size=8192, learning_rate=1e-3, epoch=20, l

train_loss, val_loss = trainer.train(train_data, val_data)
```

```
Sequential - Number of parameters: 461828
Initialize training data...
Training Epochs: 100%|██████████| 20/20 [04:03<00:00, 12.17s/it]
```





```
In [32]: total_loss, batch_loss = trainer.evaluate(val_data, draw_plot=True)
```

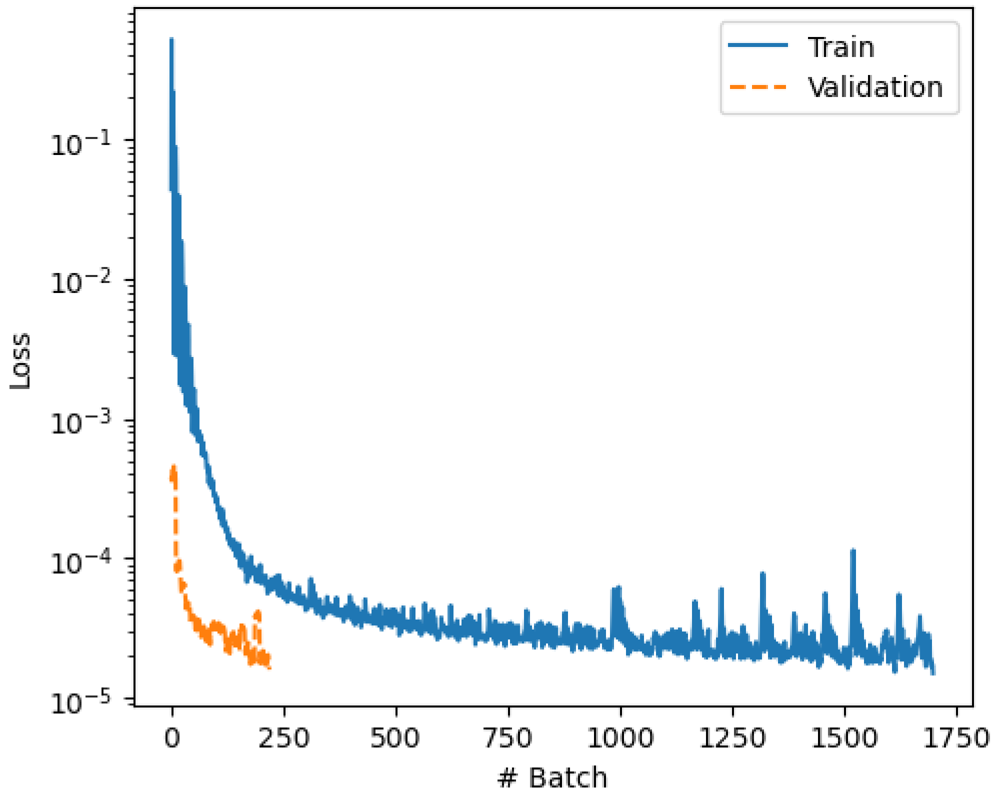


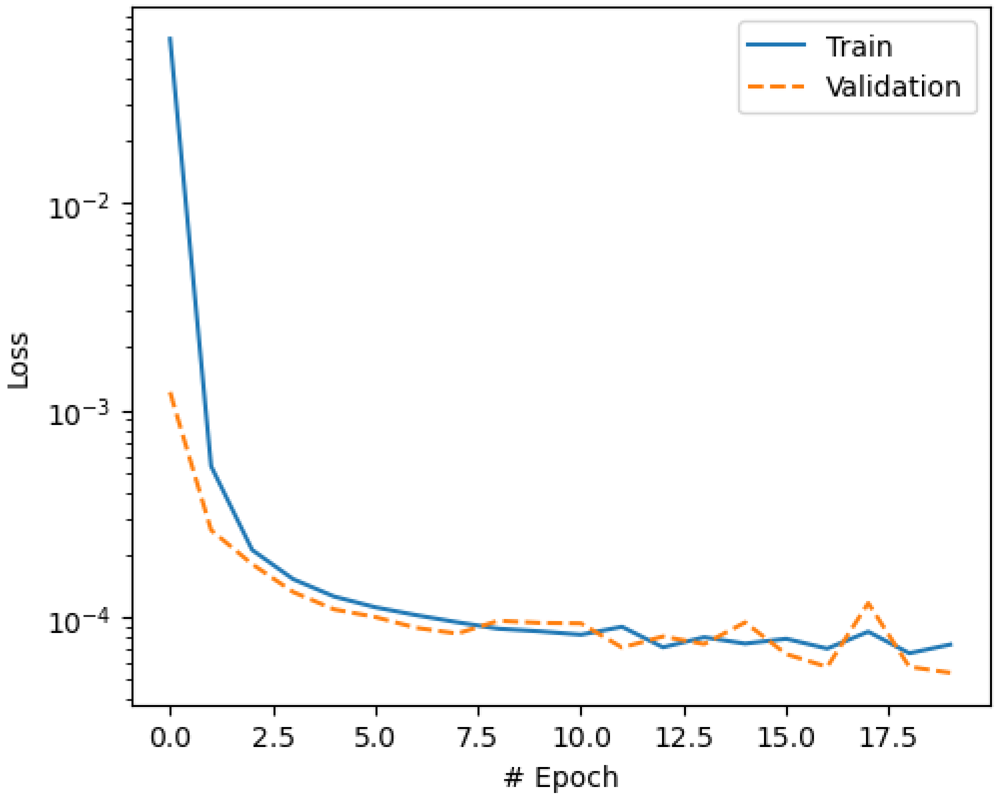
#4 Increasing and decreasing hidden layer size of resblock

In [24]:
```
class ResidualBlock(nn.Module):
    def __init__(self, input_dim, output_dim):
        super(ResidualBlock, self).__init__()
        self.fc1 = nn.Linear(input_dim, output_dim)
        self.fc2 = nn.Linear(output_dim, output_dim)
        self.activation = nn.ReLU()

    def forward(self, x):
        identity = x

        out = self.fc1(x)
        out = self.activation(out)
        out = self.fc2(out)
        out += identity  # Residual connection
        out = self.activation(out)
        return out

class LSizeBlocksAtomic(nn.Module):
    def __init__(self):
        super(LSizeBlocksAtomic, self).__init__()
        self.initial_fc = nn.Linear(384, 192)

        self.res_blocks = nn.Sequential(
            ResidualBlock(192, 192)
        )

        self.final_fc = nn.Linear(192, 1)
    def forward(self, x):
        x = F.relu(self.initial_fc(x))
        x = self.res_blocks(x)
        x = self.final_fc(x)
        return x


net_H = LSizeBlocksAtomic()
net_C = LSizeBlocksAtomic()
net_N = LSizeBlocksAtomic()
net_O = LSizeBlocksAtomic()

# 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])
Lsizeblocksmodel = nn.Sequential(
    aev_computer,
    ani_net
).to(device)
```

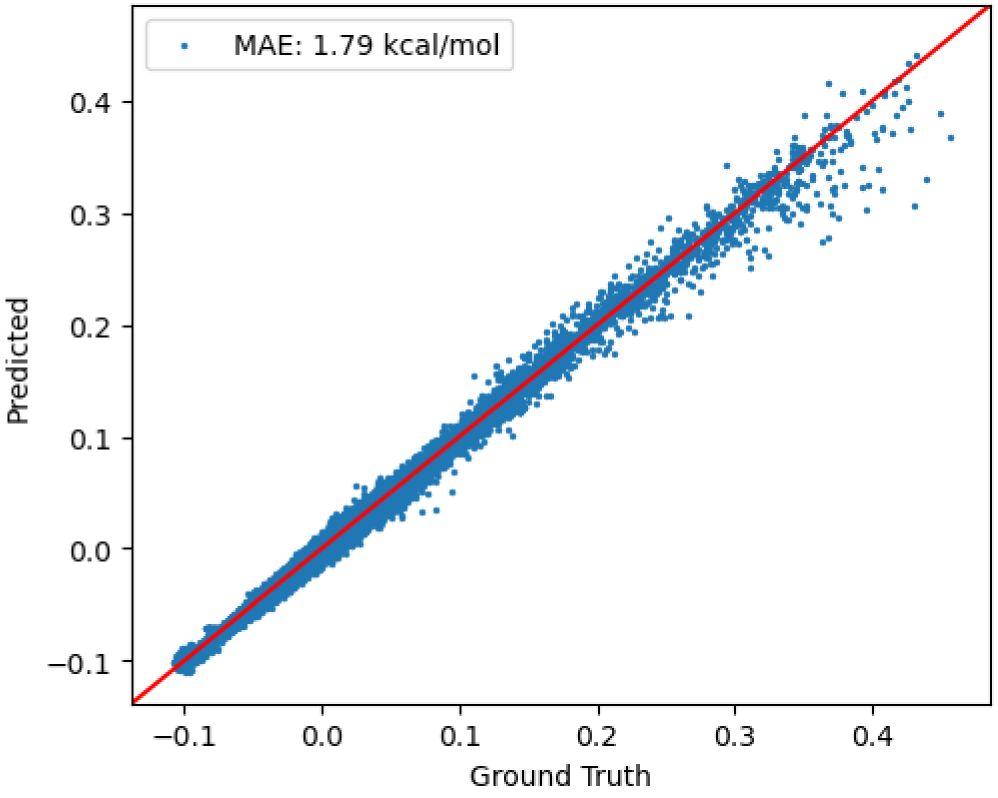
In [25]:
```
trainer = ANITrainer(Lsizeblocksmodel, batch_size=8192, learning_rate=1e-3, epoch=20,

train_loss, val_loss = trainer.train(train_data, val_data)
```

```
Sequential - Number of parameters: 592900
Initialize training data...
Training Epochs: 100%|██████████| 20/20 [04:22<00:00, 13.11s/it]
```





```
In [26]: total_loss, batch_loss = trainer.evaluate(val_data, draw_plot=True)
```

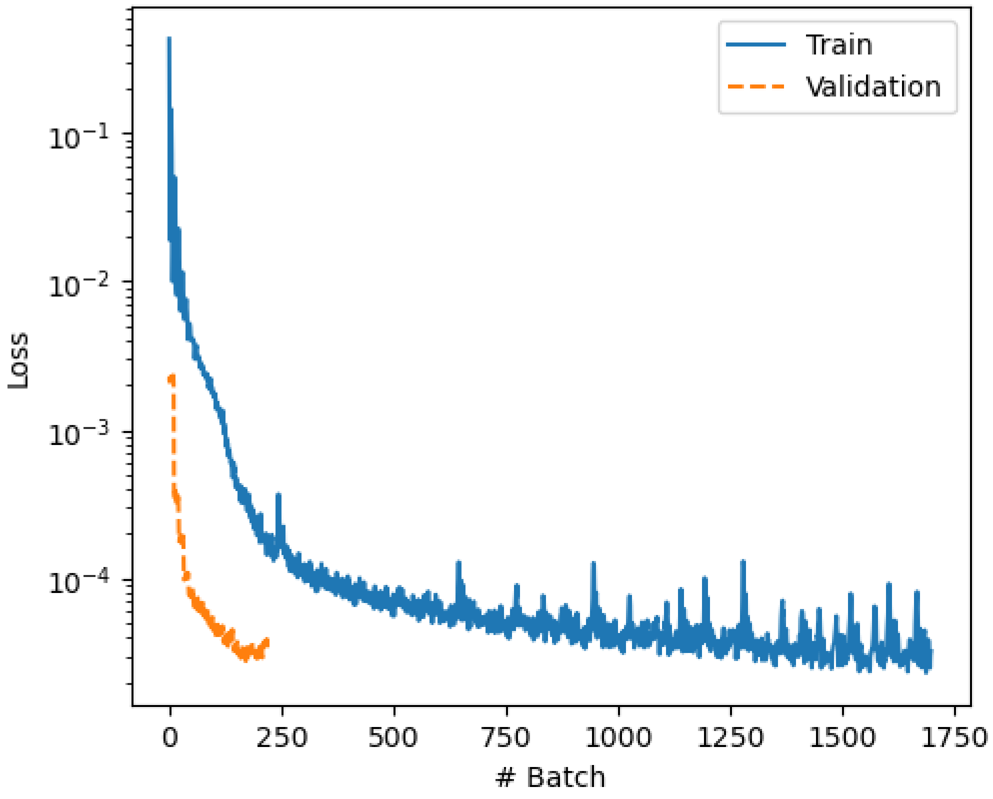


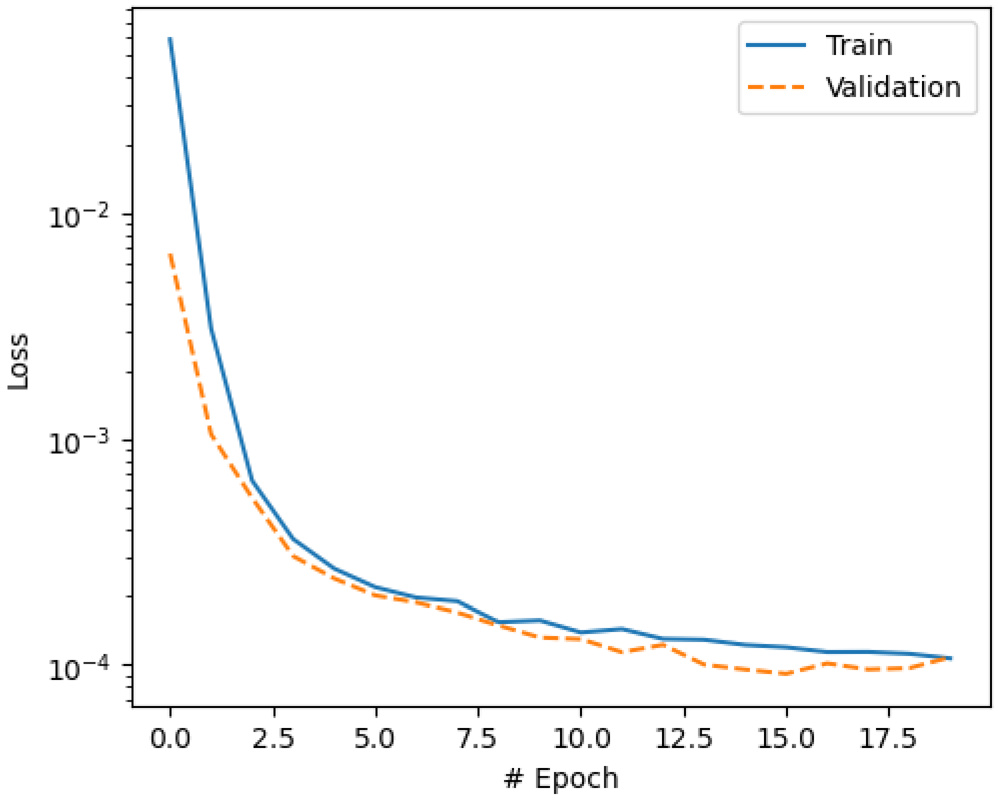
```
In [36]: class ResidualBlock(nn.Module):
             def __init__(self, input_dim, output_dim):
                 super(ResidualBlock, self).__init__()
                 self.fc1 = nn.Linear(input_dim, output_dim)
                 self.fc2 = nn.Linear(output_dim, output_dim)
```

```
        self.activation = nn.ReLU()

    def forward(self, x):
        identity = x

        out = self.fc1(x)
        out = self.activation(out)
        out = self.fc2(out)
        out += identity  # Residual connection
        out = self.activation(out)
        return out

class SSizeBlocksAtomic(nn.Module):
    def __init__(self):
        super(SSizeBlocksAtomic, self).__init__()
        self.initial_fc = nn.Linear(384, 64)

        self.res_blocks = nn.Sequential(
            ResidualBlock(64, 64)
        )

        self.final_fc = nn.Linear(64, 1)

    def forward(self, x):
        x = F.relu(self.initial_fc(x))
        x = self.res_blocks(x)
        x = self.final_fc(x)
        return x


net_H = SSizeBlocksAtomic()
net_C = SSizeBlocksAtomic()
net_N = SSizeBlocksAtomic()
net_O = SSizeBlocksAtomic()

# 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])
Ssizeblocksmodel = nn.Sequential(
    aev_computer,
    ani_net
).to(device)
```

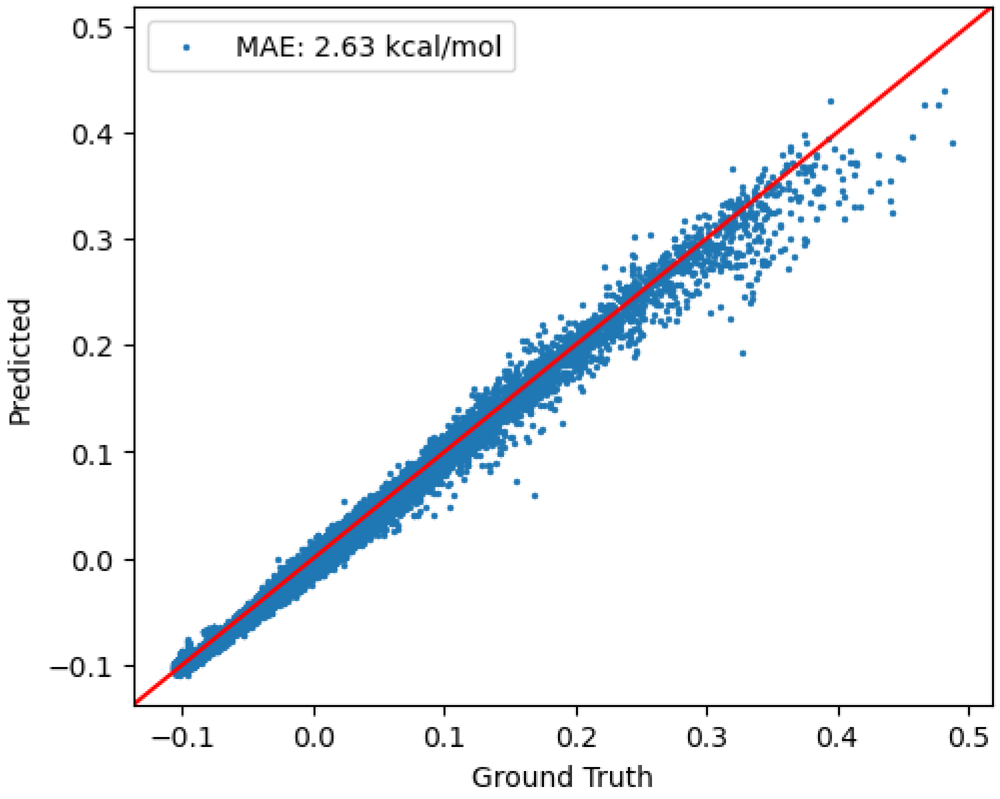
In [37]:
```
trainer = ANITrainer(Ssizeblocksmodel, batch_size=8192, learning_rate=1e-3, epoch=20,

train_loss, val_loss = trainer.train(train_data, val_data)
```

```
Sequential - Number of parameters: 132100
Initialize training data...
Training Epochs: 100%|██████████| 20/20 [03:05<00:00,  9.26s/it]
```





```
In [38]: total_loss, batch_loss = trainer.evaluate(val_data, draw_plot=True)
```

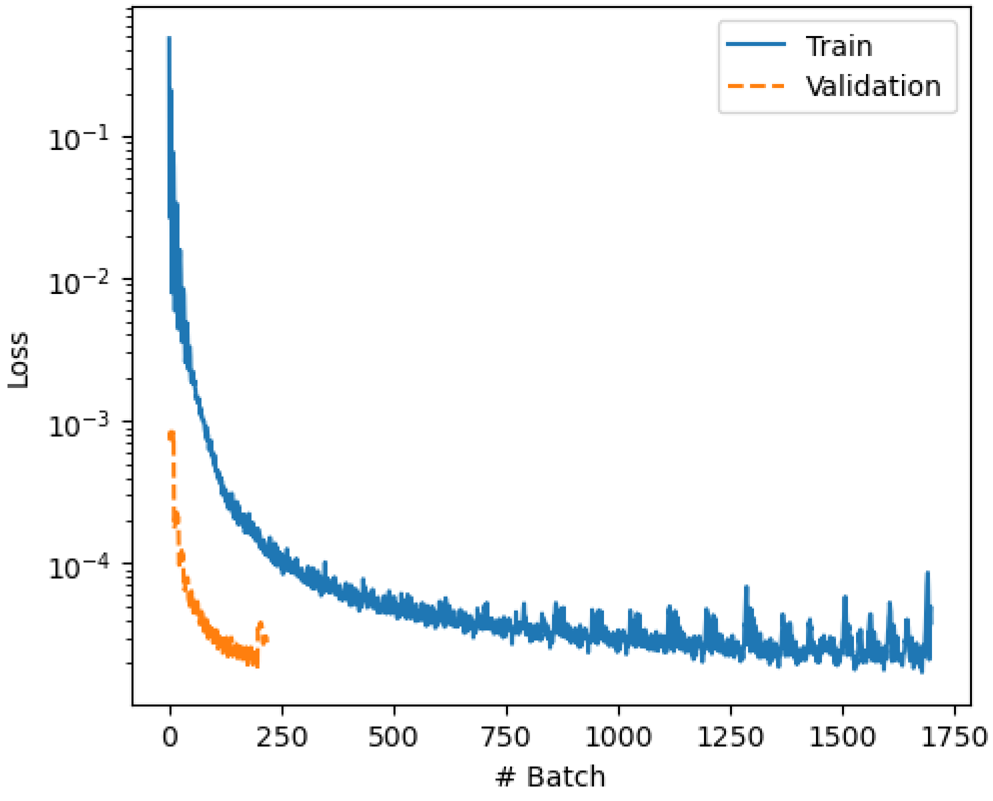


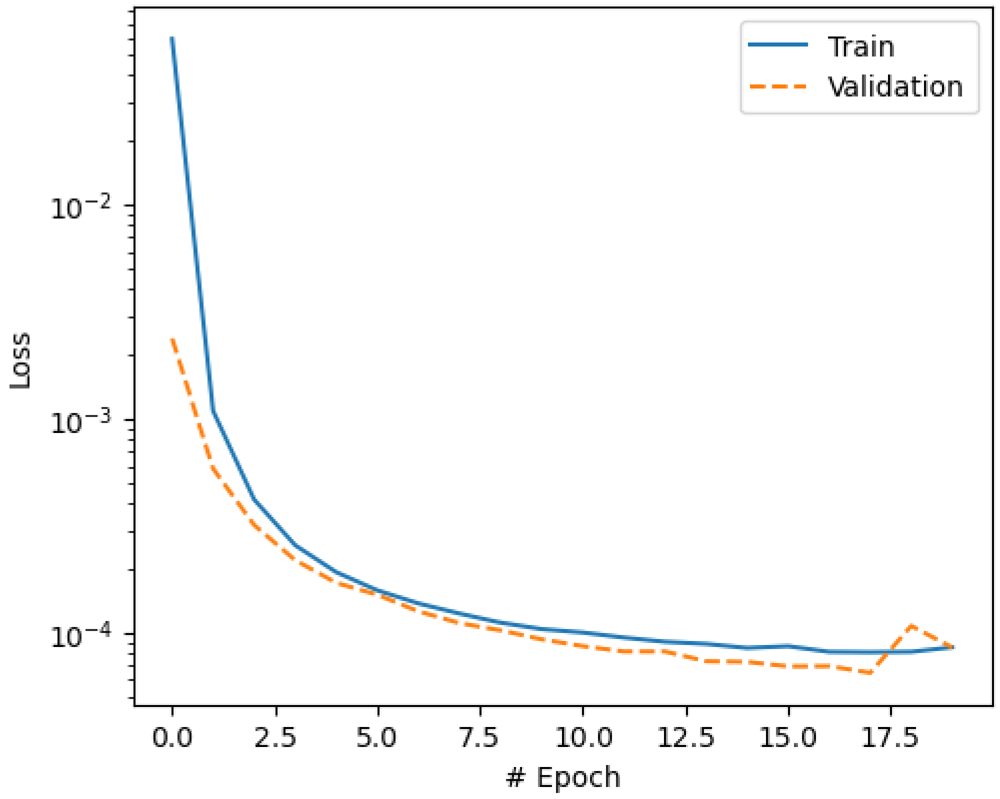
#5 Changing activations

In [9]:
```python
class ResidualBlock(nn.Module):
    def __init__(self, input_dim, output_dim):
        super(ResidualBlock, self).__init__()
        self.fc1 = nn.Linear(input_dim, output_dim)
        self.fc2 = nn.Linear(output_dim, output_dim)
        self.activation = nn.LeakyReLU() # now leaky instead of normal relu

    def forward(self, x):
        identity = x

        out = self.fc1(x)
        out = self.activation(out)
        out = self.fc2(out)
        out += identity  # Residual connection
        out = self.activation(out)
        return out

class LeakyResAtomic(nn.Module):
    def __init__(self):
        super(LeakyResAtomic, self).__init__()
        self.initial_fc = nn.Linear(384, 128)

        self.res_blocks = nn.Sequential(
            ResidualBlock(128, 128)
        )

        self.final_fc = nn.Linear(128, 1)

    def forward(self, x):
        x = F.leaky_relu(self.initial_fc(x))
        x = self.res_blocks(x)
        x = self.final_fc(x)
        return x


net_H = LeakyResAtomic()
net_C = LeakyResAtomic()
net_N = LeakyResAtomic()
net_O = LeakyResAtomic()

# 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])
leakyresmodel = nn.Sequential(
    aev_computer,
    ani_net
).to(device)
```

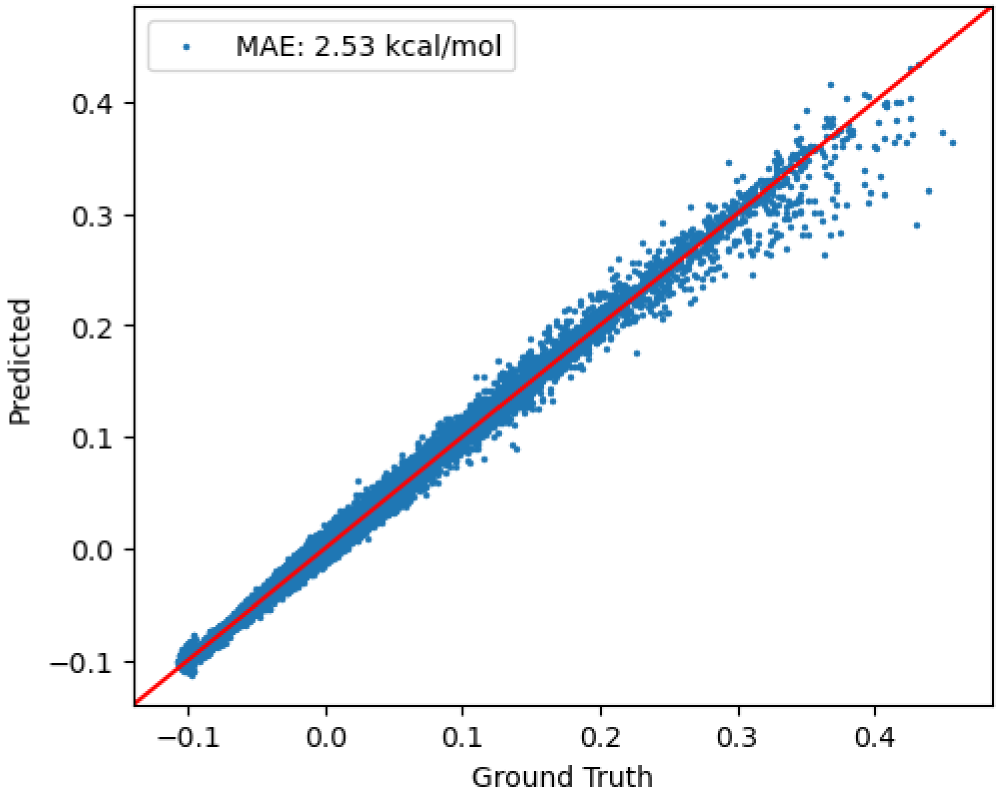
In [10]:
```
trainer = ANITrainer(leakyresmodel, batch_size=8192, learning_rate=1e-3, epoch=20, l2=

train_loss, val_loss = trainer.train(train_data, val_data)
```

```
Sequential - Number of parameters: 329732
Initialize training data...
Training Epochs: 100%|██████████| 20/20 [03:47<00:00, 11.37s/it]
```





```
In [11]: total_loss, batch_loss = trainer.evaluate(val_data, draw_plot=True)
```

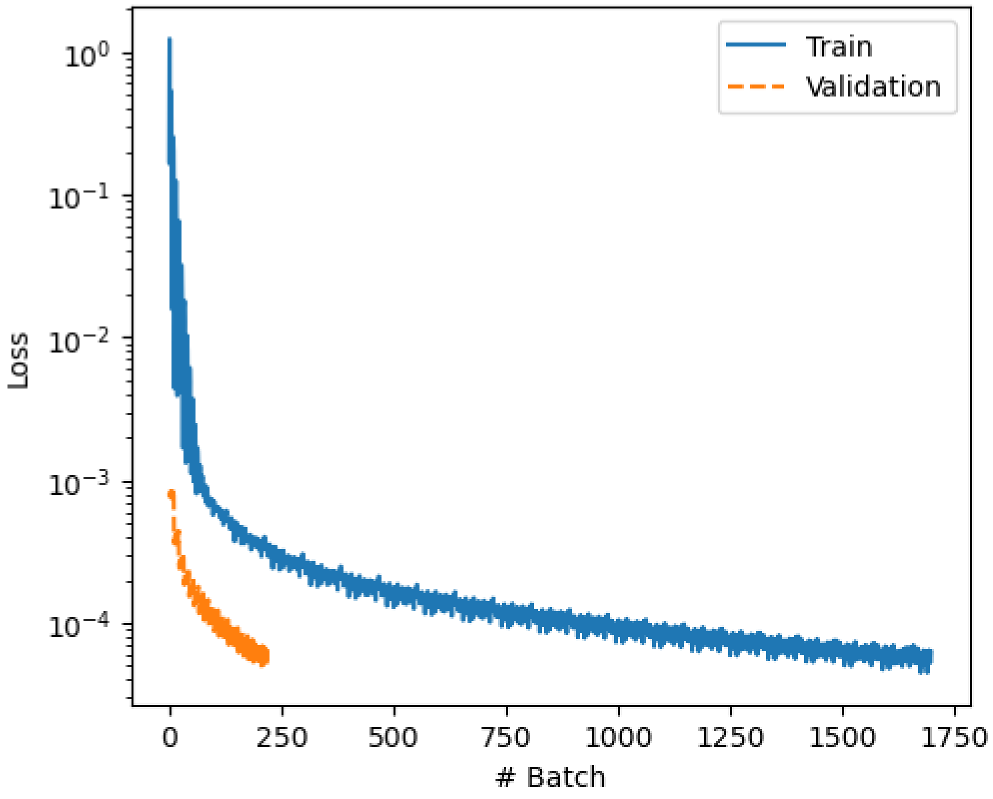


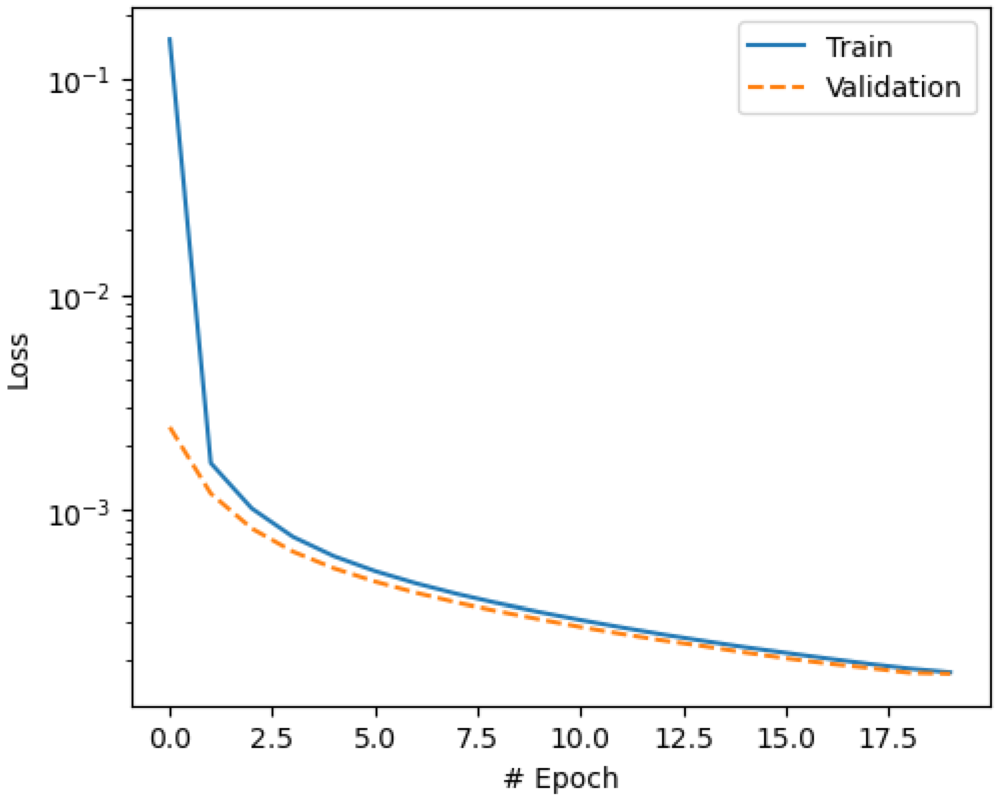
```
In [12]: class ResidualBlock(nn.Module):
             def __init__(self, input_dim, output_dim):
                 super(ResidualBlock, self).__init__()
                 self.fc1 = nn.Linear(input_dim, output_dim)
                 self.fc2 = nn.Linear(output_dim, output_dim)
```

```
        self.activation = nn.Tanh() # now tanh instead of normal relu

    def forward(self, x):
        identity = x

        out = self.fc1(x)
        out = self.activation(out)
        out = self.fc2(out)
        out += identity  # Residual connection
        out = self.activation(out)
        return out

class TanhResAtomic(nn.Module):
    def __init__(self):
        super(TanhResAtomic, self).__init__()
        self.initial_fc = nn.Linear(384, 128)

        self.res_blocks = nn.Sequential(
            ResidualBlock(128, 128)
        )

        self.final_fc = nn.Linear(128, 1)

    def forward(self, x):
        x = F.tanh(self.initial_fc(x))
        x = self.res_blocks(x)
        x = self.final_fc(x)
        return x


net_H = TanhResAtomic()
net_C = TanhResAtomic()
net_N = TanhResAtomic()
net_O = TanhResAtomic()

# 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])
tanhresmodel = nn.Sequential(
    aev_computer,
    ani_net
).to(device)
```

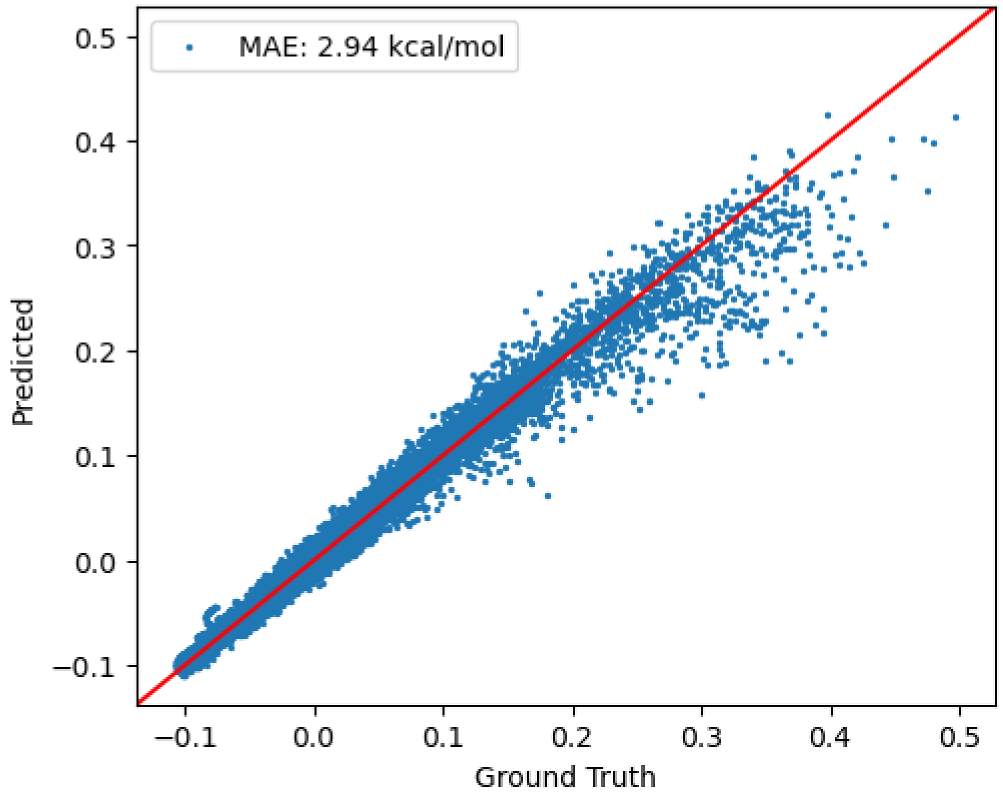
In [13]:
```
trainer = ANITrainer(tanhresmodel, batch_size=8192, learning_rate=1e-3, epoch=20, l2=1

train_loss, val_loss = trainer.train(train_data, val_data)
```

```
Sequential - Number of parameters: 329732
Initialize training data...
Training Epochs: 100%|██████████| 20/20 [03:50<00:00, 11.51s/it]
```





```
In [14]: total_loss, batch_loss = trainer.evaluate(val_data, draw_plot=True)
```



Checkpoint 4 - Final Models

Final Model One/bestresmodel combines best architectures

Combining architectures

1. Skip connection

2. Adding additional resblock

3. Dropout

4. Reducing size of hidden layer

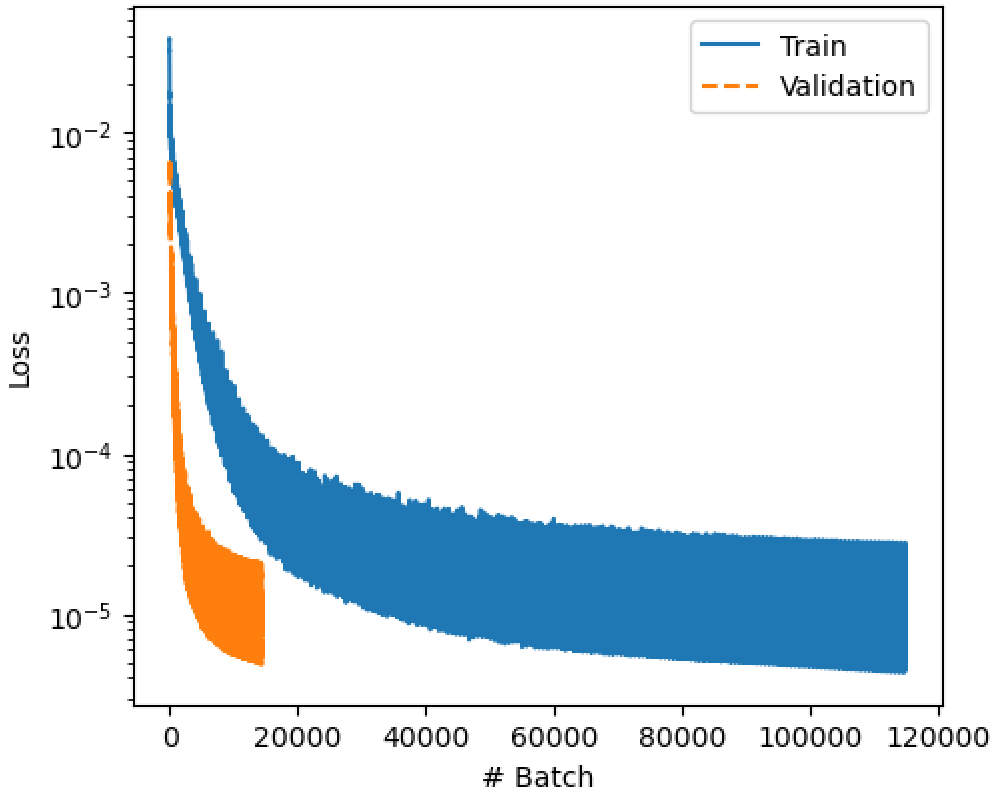
5. Leaky ReLU

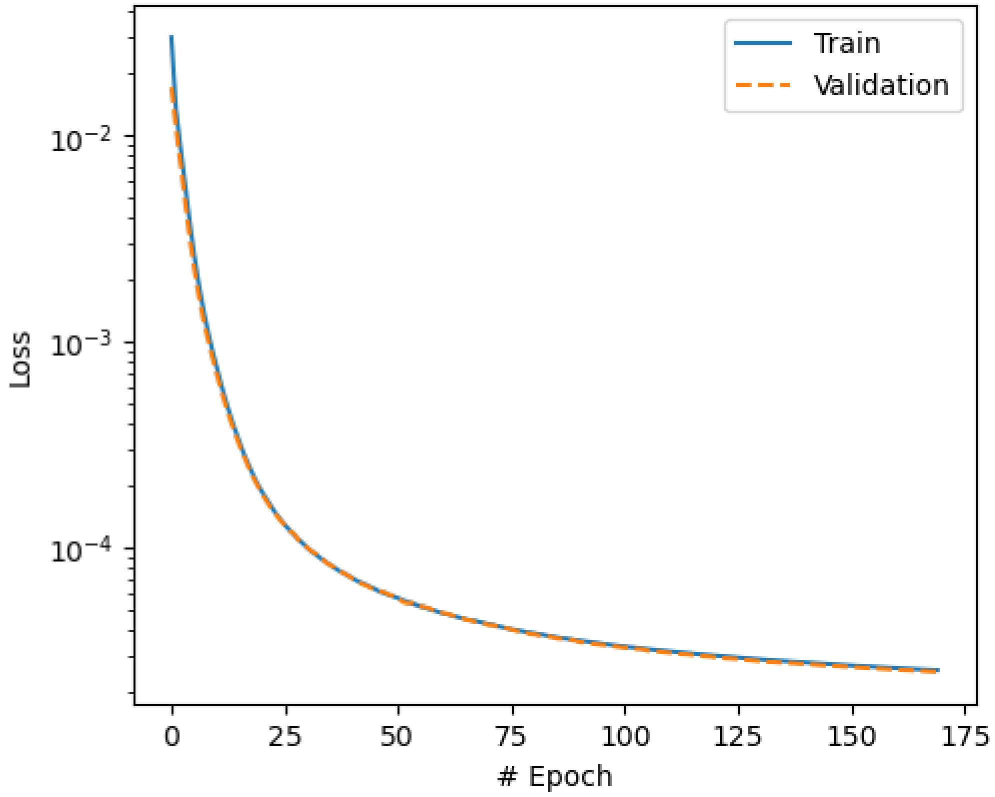
In [6]:
```
class ResidualBlock(nn.Module):
    def __init__(self, input_dim, output_dim):
        super(ResidualBlock, self).__init__()
        self.fc1 = nn.Linear(input_dim, output_dim)
        self.fc2 = nn.Linear(output_dim, output_dim)
        self.activation = nn.LeakyReLU() # leakyrelu activation function
        self.dropout = nn.Dropout(p = 0.50) # dropout

    def forward(self, x):
        identity = x

        out = self.fc1(x)
        out = self.activation(out)
        out = self.dropout(out)
        out = self.fc2(out)
        out += identity  # residual connection
        out = self.activation(out)
        return out


class BestResAtomic(nn.Module):
    def __init__(self):
        super(BestResAtomic, self).__init__()
        self.initial_fc = nn.Linear(384, 32)

        self.res_blocks = nn.Sequential( # 2 res blocks
            ResidualBlock(32, 32), # 4 hidden layers in total
            ResidualBlock(32, 32),
        )

        self.final_fc = nn.Linear(32, 1)

    def forward(self, x):
        x = F.leaky_relu(self.initial_fc(x))
        x = self.res_blocks(x)
        x = self.final_fc(x)
        return x


net_H = BestResAtomic()
net_C = BestResAtomic()
net_N = BestResAtomic()
net_O = BestResAtomic()

# 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])
bestresmodel = nn.Sequential(
    aev_computer,
    ani_net
).to(device)
```

Learning rate and batch size reduced to prolong training, epochs increased accordingly

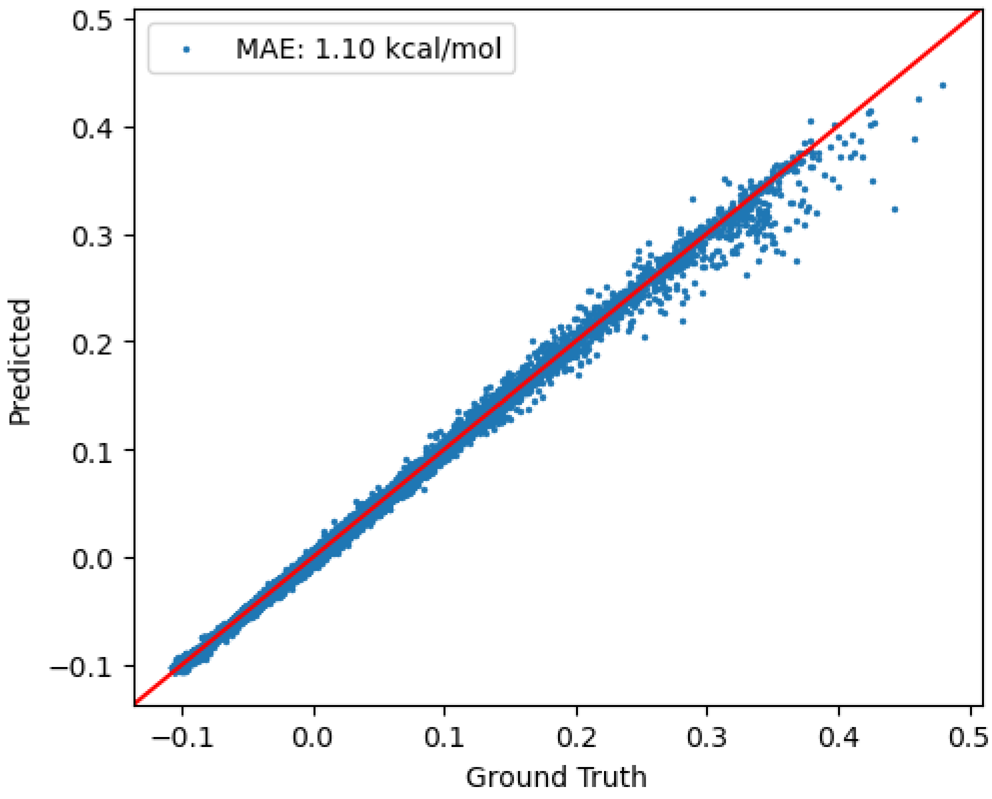
In [7]:
```
trainer = ANITrainer(bestresmodel, batch_size=1024, learning_rate=1e-5, epoch=170, l2=

loss1, loss2 = trainer.train(train_data, val_data)
```

```
Sequential - Number of parameters: 66308
Initialize training data...
Training Epochs: 100%|██████████| 170/170 [50:10<00:00, 17.71s/it]
```

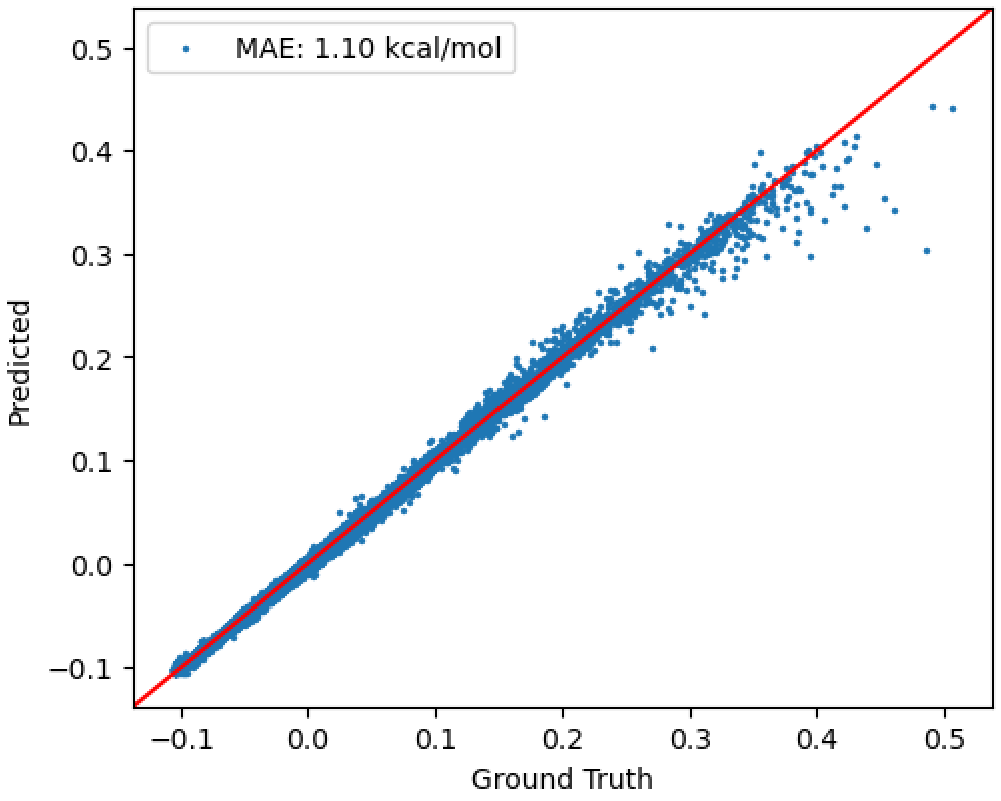




In [9]:
```
loss3, loss4 = trainer.evaluate(val_data, draw_plot=True)
```



In [8]:
```
loss5, loss6 = trainer.evaluate(test_data, draw_plot=True)
```



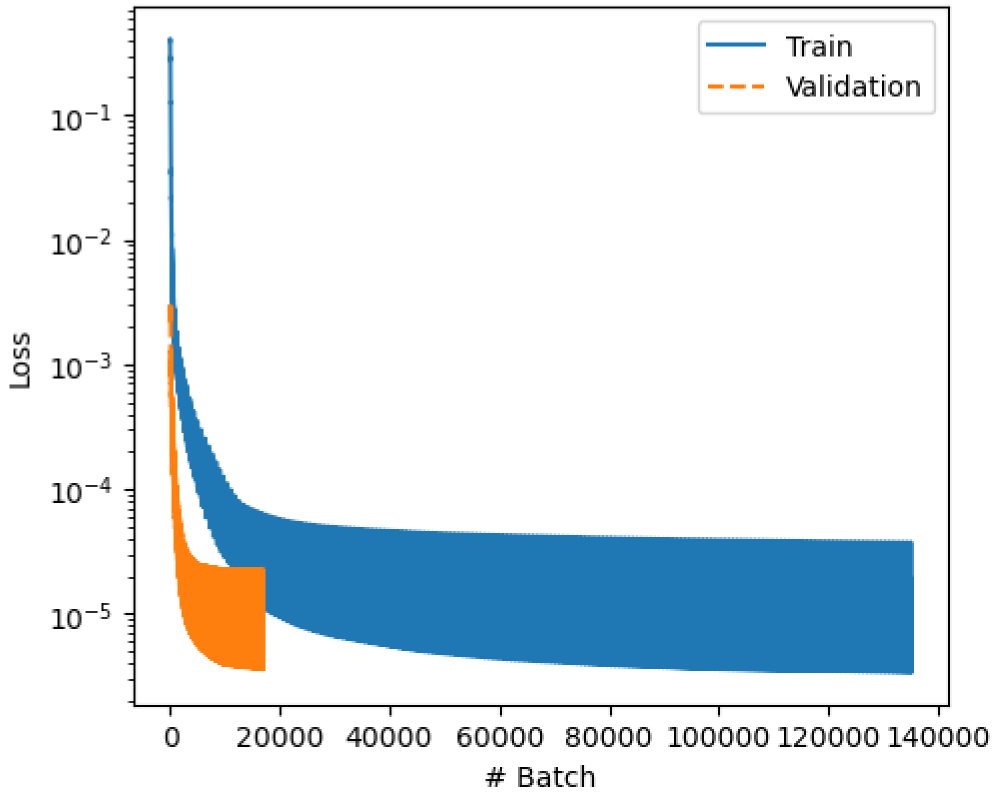
Final Model Two is baseline model with same hyperparameters as FInal Model One except

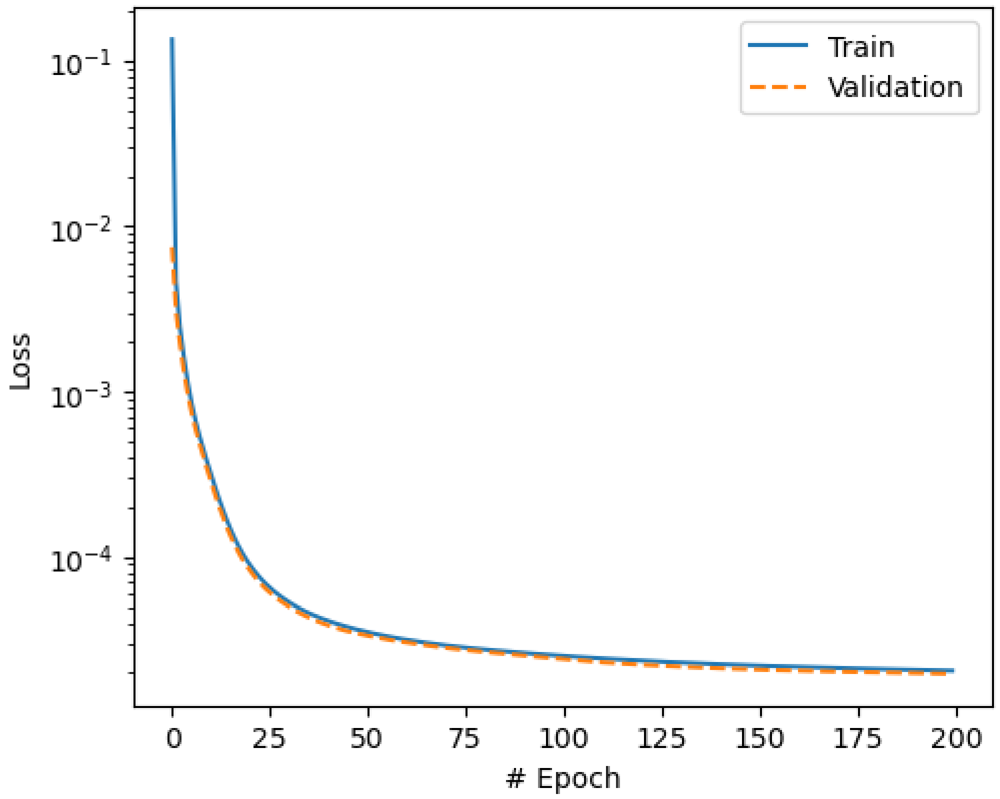
for number of epochs

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

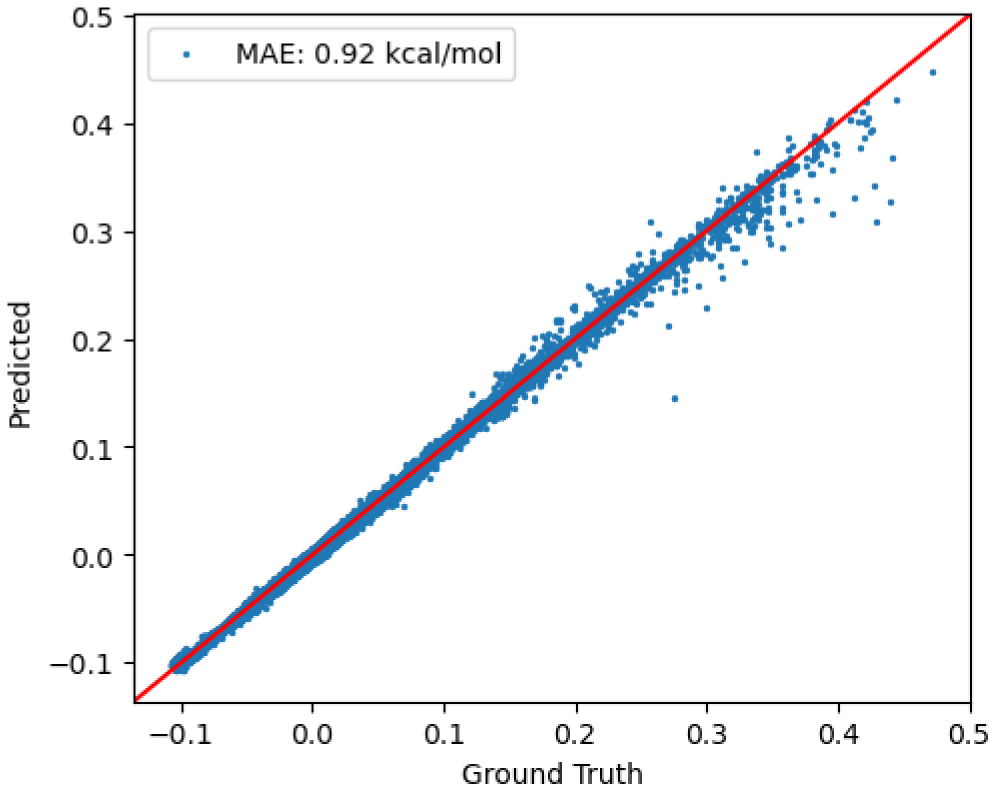
In [9]:
```
trainer = ANITrainer(simplemodel, batch_size=1024, learning_rate=1e-5, epoch=200, l2=1

loss1, loss2 = trainer.train(train_data, val_data)
```

```
Sequential - Number of parameters: 197636
Initialize training data...
Training Epochs: 100%|██████████| 200/200 [45:11<00:00, 13.56s/it]
```

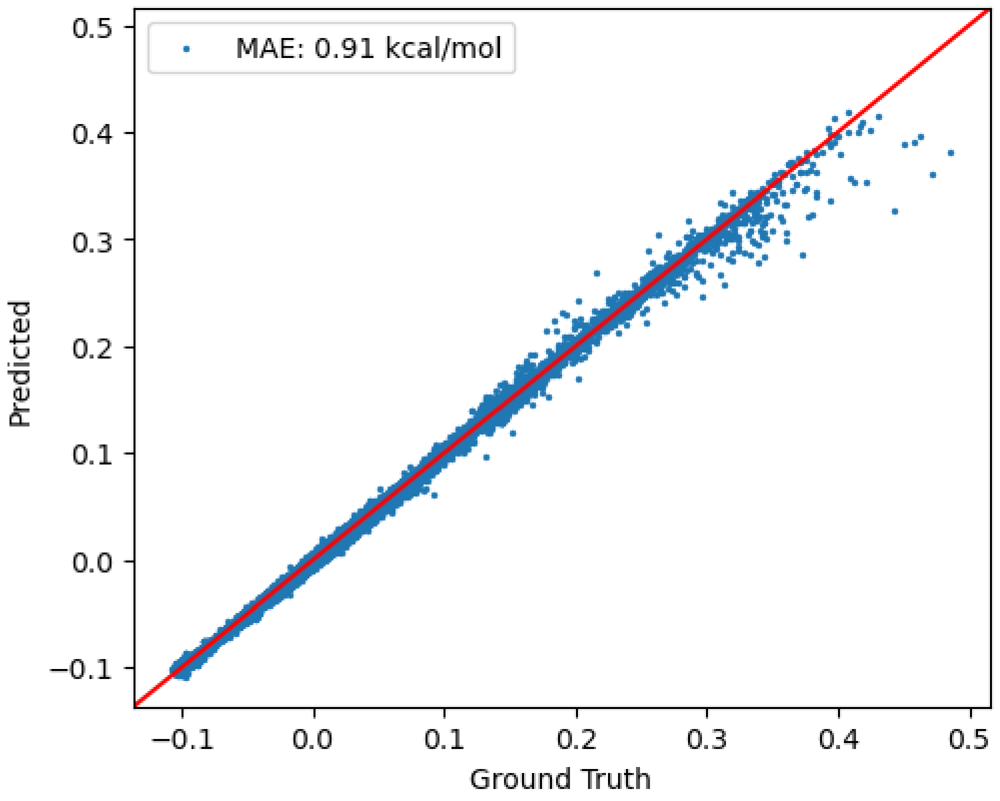




```
In [10]: loss3, loss4 = trainer.evaluate(val_data, draw_plot=True)
```



In [11]:
```
loss5, loss6 = trainer.evaluate(test_data, draw_plot=True)
```

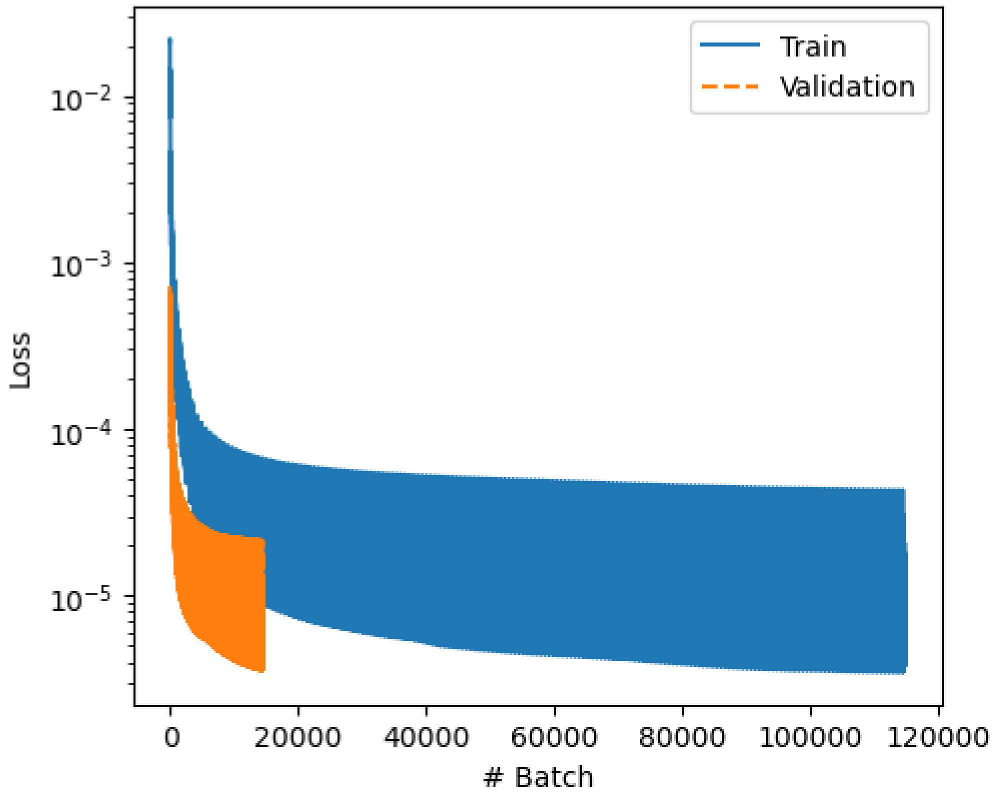


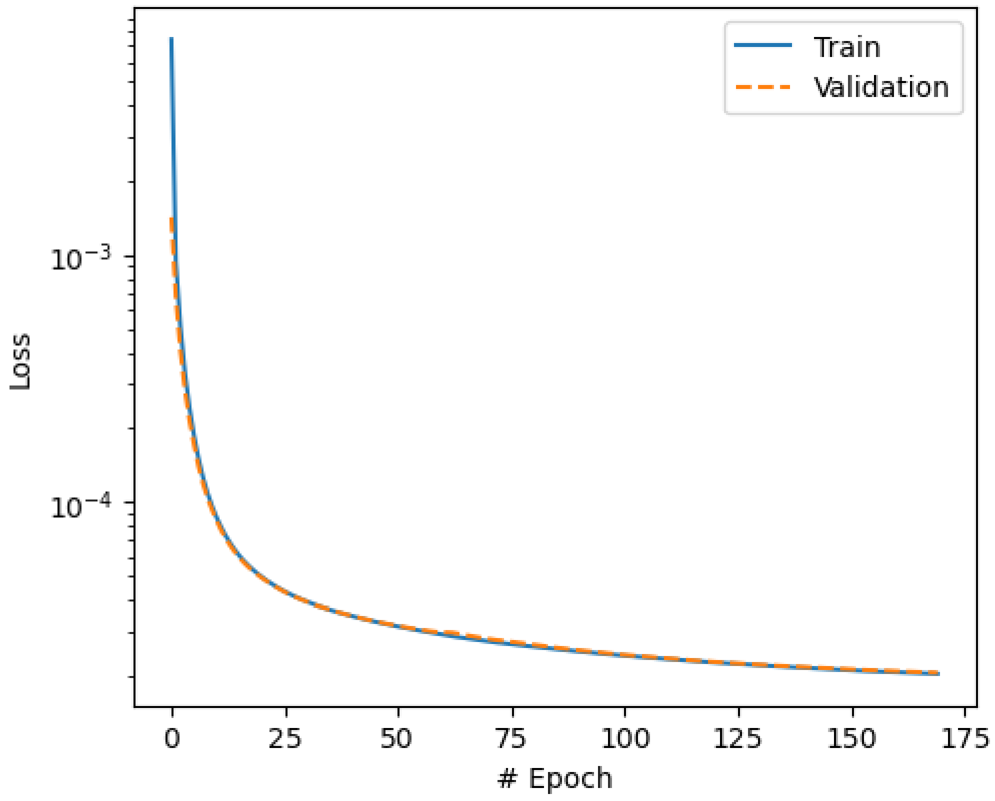
Final Model 3 is Lsizeblocksmodel with same hyperparameters as other final models

In [6]:
```
class ResidualBlock(nn.Module):
    def __init__(self, input_dim, output_dim):
        super(ResidualBlock, self).__init__()
        self.fc1 = nn.Linear(input_dim, output_dim)
        self.fc2 = nn.Linear(output_dim, output_dim)
        self.activation = nn.ReLU()

    def forward(self, x):
        identity = x

        out = self.fc1(x)
        out = self.activation(out)
        out = self.fc2(out)
        out += identity  # Residual connection
        out = self.activation(out)
        return out

class LSizeBlocksAtomic(nn.Module):
    def __init__(self):
        super(LSizeBlocksAtomic, self).__init__()
        self.initial_fc = nn.Linear(384, 192)

        self.res_blocks = nn.Sequential(
            ResidualBlock(192, 192)
        )

        self.final_fc = nn.Linear(192, 1)

    def forward(self, x):
        x = F.relu(self.initial_fc(x))
        x = self.res_blocks(x)
        x = self.final_fc(x)
        return x


net_H = LSizeBlocksAtomic()
net_C = LSizeBlocksAtomic()
net_N = LSizeBlocksAtomic()
net_O = LSizeBlocksAtomic()

# 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])
Lsizeblocksmodel = nn.Sequential(
    aev_computer,
    ani_net
).to(device)
```

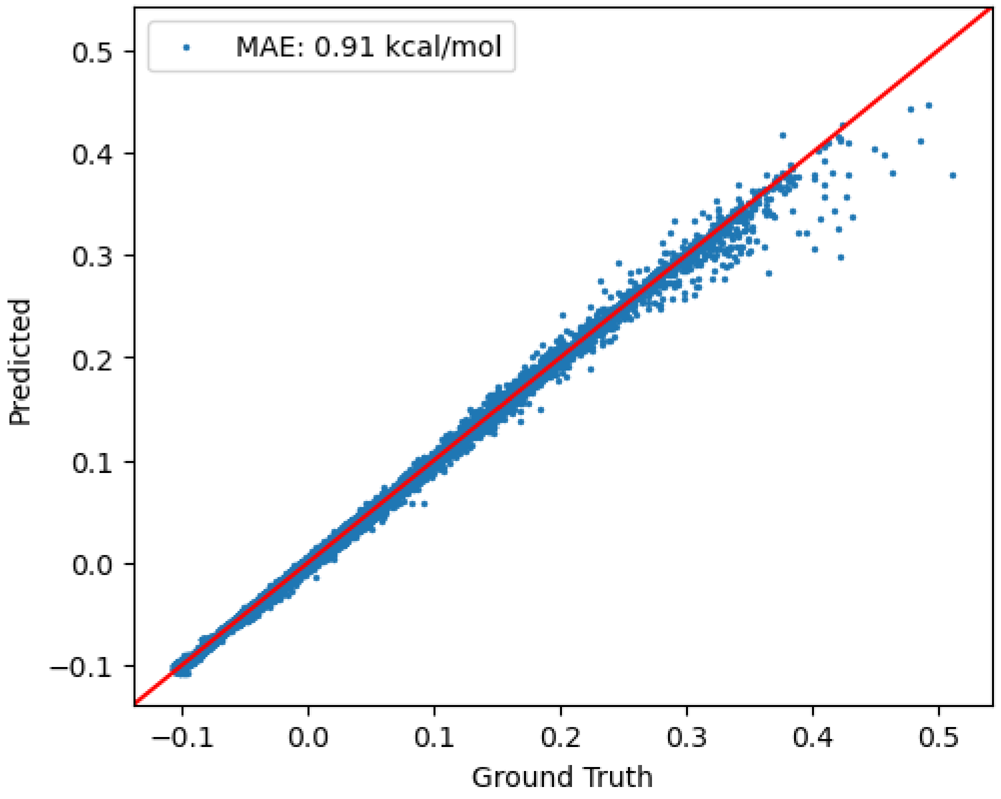
In [7]:
```
trainer = ANITrainer(Lsizeblocksmodel, batch_size=1024, learning_rate=1e-5, epoch=170,

loss1, loss2 = trainer.train(train_data, val_data)
```

```
Sequential - Number of parameters: 592900
Initialize training data...
Training Epochs: 100%|██████████| 170/170 [51:26<00:00, 18.15s/it]
```

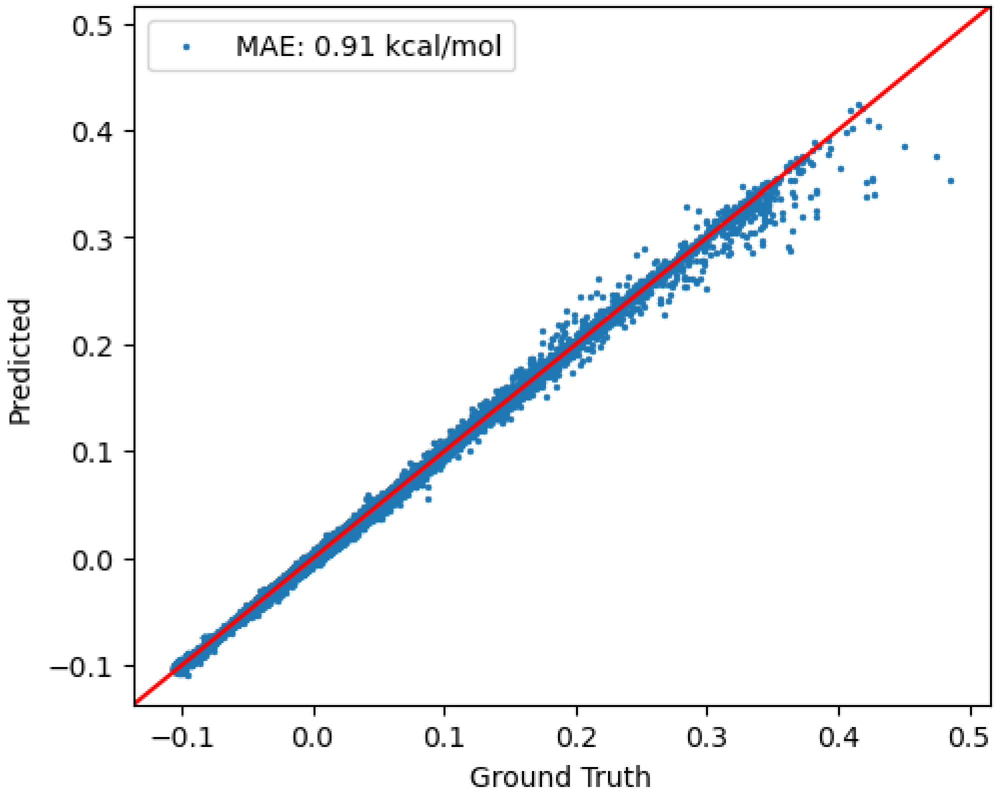




```
In [8]: loss3, loss4 = trainer.evaluate(val_data, draw_plot=True)
```



```
In [9]: loss5, loss6 = trainer.evaluate(test_data, draw_plot=True)
```


MAE: 0.91 kcal/mol
Predicted
Ground Truth