Please complete the `NotImplemented` parts of the code cells and write your answers in the markdown cells designated for your response to any questions asked. The tag `# AUTOGRADED` (all caps, with a space after `#`) should be at the beginning of each autograded code cell, so make sure that you do not change that. You are also not allowed to import any new package other than the ones already imported. Doing so will prevent the autograder from grading your code.

For the code submission, run the last cell in the notebook to create the submission zip file. If you are working in Colab, make sure to download and then upload a copy of the completed notebook itself to its working directory to be included in the zip file. Finally, submit the zip file to Gradescope.

After you finish the assignment and fill in your code and response where needed (all cells should have been run), save the notebook as a PDF using the `jupyter nbconvert --to pdf HW2.ipynb` command (via a notebook code cell or the command line directly) and submit the PDF to Gradescope under the PDF submission item. If you cannot get this to work locally, you can upload the notebook to Google Colab and create the PDF there. You can find the notebook containing the instruction for this on Canvas.

If you are running the notebook locally, make sure you have created a virtual environment (using `conda` for example) and have the proper packages installed. We are working with `python=3.10` and `torch>=2`.

Files to be included in submission:

- `HW2.ipynb`
- `model_config.yaml`
- `train_config.yaml`
- `state_dict.pth`

Build and train a neural network for regression

The problem you are asked to solve is Airfoil Self-Noise prediction. Namely, given 5 features (Frequency in Hertz, Angle of attack in degrees, Chord length in meters, Free-stream velocity in meters per second, and Suction side displacement thickness in meters), your model is supposed to accurately predict the Scaled sound pressure level, in decibels. The datasets have been preprocessed for you and can be found as `train.npy` and `val.npy` in the `data` folder. You have to implement your custom dataset, model, and train function. We have also provided helper functions for you to keep track of model performance during

training. Please make use of them, and try to understand their code as you may need to implement similar functions in the future.

In [103…

```
# DO NOT MODIFY THIS CELL OR ADD ANY IMPORTS IN OTHER CELLS!

from typing import Union, Tuple, List, Sequence
import numpy as np
import torch
from torch import nn, optim
from torch.utils.data import Dataset, DataLoader
from tqdm import tqdm
from HW2_utils import save_yaml, load_yaml, zip_files, Learning_Curve_Tracker

if torch.cuda.is_available():
    Device = 'cuda'
elif torch.backends.mps.is_available():
    Device = 'mps'
else:
    Device = 'cpu'

print(f'Device is {Device}')
```

```
Device is cpu
```

Implement the dataset class (20)

First, you will implement a subclass of `torch.utils.data.Dataset` to define a custom dataset class. To do so, you will need to implement three methods for the subclass:

- `__init__` defines the dataset using the path to the data file (for example `data/train.npy` or `data/val.npy`). Your code should load the data using `np.load` and save it as attributes to be refereced in other methods that you implement. You can apply transformations like changing the dtype of data when saving them as attributes, which might be convenient.
- `__len__` should return a non-negative integer that is the total number of data points. This will be used by the dataloader to count and batch the data.
- `__getitem__` should return a single data sample (containing input, output pairs for this problem) using the index passed. Generally, the `__getitem__` method defines the behavior of an object when indexed using square brackets (like `a[i]`).

Both datasets are of shape `(N, 6)` where N is the number of samples. The first five indexes of the last dimension contain the input features and the last one contains the output.

In [103…

```
class AirFoilDataset(Dataset):

    def __init__(
            self,
            data_path: str,
```

```
            ):
        super().__init__()
        data = np.load(data_path)
        # process the data as torch tensors with the correct dtype and shape
        # NotImplemented
        self.inputs = torch.tensor(data[:, :5], dtype=torch.float32)  # input featu
        self.targets = torch.tensor(data[:, 5], dtype=torch.float32).unsqueeze(1)
    def __len__(self):
        # NotImplemented
        return self.inputs.shape[0]

    def __getitem__(
            self,
            idx: int,
            ) -> Tuple[torch.FloatTensor, torch.FloatTensor]: # (5,), (1,)
        """
        Returns a tuple of (x, y) where x is the input data and y is the target lab
        shape of x: (5,)
        shape of y: (1,)
        """
        # NotImplemented
        return self.inputs[idx], self.targets[idx]
```

In [103…

```
# testing the shapes and dtypes

data_path = './data/train.npy'
dataset = AirFoilDataset(data_path)

for idx in np.random.randint(0, len(dataset), 5):
    x, y = dataset[idx]
    assert x.dtype == torch.float32
    assert y.dtype == torch.float32
    assert x.shape == (5,)
    assert y.shape == (1,)
```

Implement the model (30)

Implement your model class. Try to make use of modules like `nn.Sequential`, `nn.ModuleList`, and `nn.ModuleDict` to define a neural network with a modifiable number of layers.

In [103…

```
# AUTOGRADED
class Model(nn.Module):
    def __init__(
            self,
            input_dim: int,
            output_dim: int,
            # NotImplemented
            # hidden_dims: List[int] = [64, 32],
            # hidden_dims: List[int] = [128, 64, 32],
            hidden_dims: List[int] = [256, 128, 64, 32],
            # hidden_dims: List[int] = [512, 256, 128, 64, 32],
            # activation: nn.Module = nn.ReLU,
```

```
            activation: str = "ReLU",
        ):
        super().__init__()
        # NotImplemented

        # activation function map to the string
        activation_map = {
            "ReLU": nn.ReLU(),
            "Sigmoid": nn.Sigmoid(),
            "Tanh": nn.Tanh(),
            "LeakyReLU": nn.LeakyReLU(),
            "ELU": nn.ELU(),
            "PReLU": nn.PReLU(),
            "SELU": nn.SELU(),
            "GELU": nn.GELU(),
        }

        activation_fn = activation_map[activation]

        layers = []
        dims = [input_dim] + hidden_dims + [output_dim]

        for i in range(len(dims) - 1):
            layers.append(nn.Linear(dims[i], dims[i+1]))
            if i < len(dims) - 2:
                layers.append(activation_fn)

        self.model = nn.Sequential(*layers)

    def forward(
            self,
            x: torch.FloatTensor, # (batch_size, input_dim)
            ) -> torch.FloatTensor: # (batch_size, output_dim)
        # you can modify properties of the data before passing it through the model
        # NotImplemented

        return self.model(x)
```

Helper functions for tracking model performance

Before moving on to training, we provide an evaluation function for you to use during training. At the end of each epoch, use this function to calculate the loss on your training and validation dataset. Also, we provide a class to keep track of your losses with an option to plot the learning curve in real-time during training in the util file.

In [103…

```
# DO NOT MODIFY THIS CELL!

# The first line is called a function decorator. It's a shorthand way to wrap a fun

# Remember that torch always keeps track of the computations so we can calculate th
# This can induce unnecessary overhead when we are not training!
# By using this function decorator, we are telling torch that we are not interested
# This can make the code run faster.
```

```
@torch.inference_mode() # this is a function decorator
def evaluate(
    model: nn.Module,
    dataloader: DataLoader,
    loss_fn = nn.MSELoss(reduction='sum'),
    device = Device,
    ):

    # Set the model to evaluation mode and move to the correct device
    # (because some layers like dropout or batchnorm have different behavior when t
    model.eval().to(device)

    total_loss = 0.
    n_samples = len(dataloader.dataset)
    for x, y in dataloader:

        # move data to the correct device and calculate the predictions
        x, y = x.to(device), y.to(device)
        y_pred = model(x)
        # calculate the loss
        total_loss += loss_fn(y_pred, y).item() # use .item() to extract the loss a

    average_loss = total_loss / n_samples
    return average_loss
```

Helper functions for evaluation and tracking model performance

Before moving on to training, we provide an evaluation function for you to use during training. At the end of each epoch, use this function to calculate the loss on your training and validation dataset. Also, we provide a class to keep track of your losses with an option to plot the learning curve in real-time during training in the util file.

In [103…

```
# For train function, we use this decorator to make sure that torch keeps track of
# Although this is the default behavior, it's good practice to make it explicit.
@torch.enable_grad()
def train(
    model: nn.Module,
    train_data: Dataset,
    val_data: Dataset,

    # training hyperparameters:
    n_epochs: int,
    batch_size: int,
    opt_name: str, # Name of the optimizer class in torch.optim
    opt_config: dict = {}, # default setting. You can pass more options to the opti
    lr_scheduler_name: Union[str, None] = None, # Name of the learning rate schedul
    lr_scheduler_config: dict = {}, # default setting. You can pass more options to

    device = Device,
    plot_freq = None,
    ):
```

```
    loss_fn = nn.MSELoss(reduction='mean')

    # initialize a learning curve tracker
    lct = Learning_Curve_Tracker(n_epochs, plot_freq)

    # create dataloaders
    # NotImplemented
    train_loader = DataLoader(train_data, batch_size=batch_size, shuffle=True)
    val_loader = DataLoader(val_data, batch_size=batch_size, shuffle=False)

    # define your optimizer and learning rate scheduler.
    # use the getattr fuction or the .__getattribute__ method to get the optimizer
    # For example, getattr(optim, 'Adam') or optim.__getattribute__('Adam') gives y
    # pass their config dictionaries using ** to unpack it as keyword arguments
    # NotImplemented

    optimizer = getattr(optim, opt_name)(model.parameters(), **opt_config)
    lr_scheduler = getattr(optim.lr_scheduler, lr_scheduler_name)(optimizer, **lr_s

    model.to(device)

    epoch_pbar = tqdm(range(1, n_epochs+1), desc='Epochs', unit='epoch', leave=Fals

    for epoch in epoch_pbar:

        model.train()
        total_loss = 0.0
        total_samples = 0.0

        # Each epoch will be fast. No need for a progress bar inside the epoch for
        # loop over training batches using the dataloader to traing the model
        # NotImplemented

        for x, y in train_loader:
            optimizer.zero_grad()
            x, y = x.to(device), y.to(device)
            y_pred = model(x)
            loss = loss_fn(y_pred, y)
            loss.backward()
            optimizer.step()

            batch_size = x.shape[0]
            total_loss += loss.item() * batch_size
            total_samples += batch_size

        # After the epoch is done, evaluate the model on the training and validatio
        # NotImplemented

        model.eval()
        val_loss = 0.0
        with torch.no_grad():
            for x, y in val_loader:
                x, y = x.to(device), y.to(device)
                y_pred = model(x)
                val_loss += ((y_pred - y) ** 2).sum().item()
```

```
        train_loss = total_loss / total_samples
        val_loss = val_loss / len(val_loader.dataset)

        # val_loss = evaluate(model=model, dataloader=val_loader, device=device)

        # update the learning curve tracker and the learning rate scheduler
        # NotImplemented

        lct.update(train_loss, val_loss)
        if lr_scheduler:
            lr_scheduler.step()

        # pass

    return lct.get_losses()
```

train your model (10)

You have find a good set of hyperparameters for your model and your trianing. You will submit the successful config and state_dict. 10 points of your score depends on your model's performance on the test dataset, which will be evaluated by the autograder. Please run the final cell to save the model config and state to include them in your submission to Gradescope. Your score based on test loss will be:

- `loss <= 0.035`: 15 points (5 extra points)
- `0.035 < loss <= 0.05`: 10 points
- `0.05 < loss <= 0.07`: 5 points
- `loss > 0.07`: 0 points

Hyperparameters you can explore:

- model configuration: Try changing the model size like number of layers or hidden dimensions.
- optimizer: Look into the online documentation for different choices for the optimizer, as well as their hyperparameters and regularization options.
- learning rate scheduler: Look into the online documentation for different choices of schedulers for the learning rate of the optimizer and its hyperparameters, and how to use it.
- training hyperparameters: You can also try increasing the number of epochs or the batch size. Training the model for more epochs may resolve underfitting. Bigger batch size may also help with training stability.

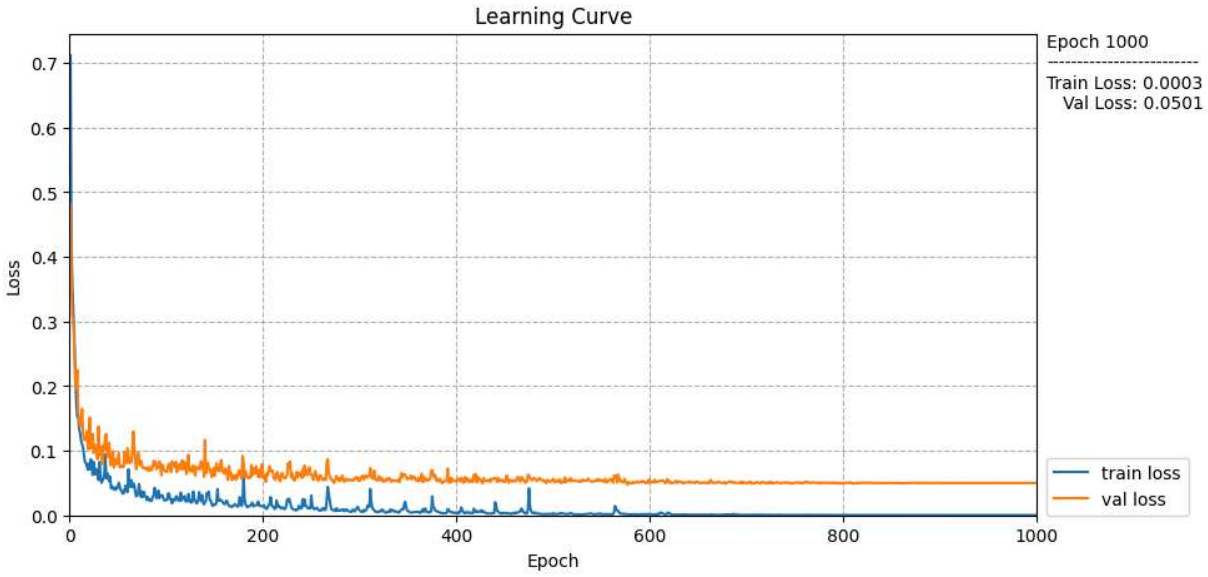
In [103…
```
# Load the data
if __name__ == '__main__':
    train_data = AirFoilDataset('data/train.npy')
    val_data = AirFoilDataset('data/val.npy')
```

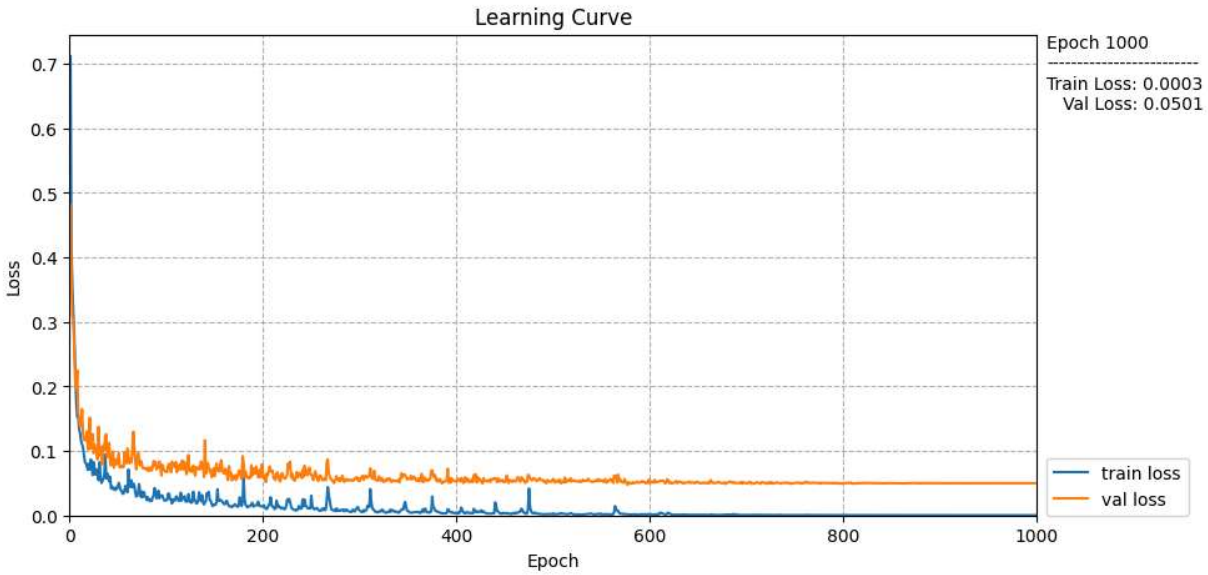
In [104…
```
# Model and training configuration
# You can iterate using this cell to find the best configuration

model_config = dict(
    # NotImplemented
    input_dim = 5,
    output_dim = 1,
    # hidden_dims = [64, 32],
    # hidden_dims = [128, 64, 32],
    hidden_dims = [256, 128, 64, 32],
    # hidden_dims = [512, 256, 128, 64, 32],
    # activation = "ReLU"
    # activation = "Sigmoid"
    # activation = "Tanh"
    activation = "LeakyReLU"
    # activation = "GELU"
)

train_config = dict(
    # NotImplemented
    # n_epochs = 50,
    # n_epochs = 100,
    n_epochs = 1000,
    # n_epochs = 200,
    batch_size = 32,
    # batch_size = 64,
    # opt_name = 'Adam',
    opt_name = 'AdamW',
    # opt_name = 'RMSprop',
    # opt_name = 'SGD',
    opt_config = {'lr': 1e-3, 'weight_decay': 1e-4},
    # opt_config = {'lr': 3e-4, 'weight_decay': 5e-4},
    # lr_scheduler_name = 'StepLR',
    # lr_scheduler_config = {'step_size': 20, 'gamma': 0.5},
    # lr_scheduler_name = 'MultiStepLR',
    # lr_scheduler_config = {'milestones': [20, 40, 60, 80], 'gamma': 0.5},
    lr_scheduler_name = 'CosineAnnealingLR',
    lr_scheduler_config = {'T_max': 1000, 'eta_min': 1e-6},
    device='cuda' if torch.cuda.is_available() else 'cpu',
)
```

In [104…
```
# train the model
if __name__ == '__main__':
    model = Model(**model_config).to(Device)
    losses = train(model, train_data, val_data, **train_config, plot_freq=50)
```





Explain your findings (10)

Please explain how you searched for your hyperparameters, and what you learned about the effect of each in the next markdown cell.

RESPONSE:

To optimize the model's performance, I experimented with various hyperparameters using a trial-and-error approach. The best configuration found was:

- `lr=1e-3 with weight_decay': 1e-4` , `AdamW` , `hidden_dims=[256, 128, 64, 32]` , `batch_size=32` , `n_epochs=150`
- `CosineAnnealingLR` with `T_max=1000` , `eta_min=1e-6`
- `LeakyReLU` activation

1. Hidden Layer Dimensions

Increasing the number of hidden layers improved performance significantly but slightly increased computational cost. The addition of `256` neurons in the first layer enhanced feature extraction.

2. Optimizer Choice

- **AdamW** performed best, providing stable updates and fast convergence.
- **RMSprop** is better suited for recurrent models.
- **SGD** required more tuning and worked better for larger datasets.

3. Learning Rate Scheduler

- **CosineAnnealingLR** resulted in smoother convergence and reduced overfitting.
- **ReduceLROnPlateau** was tested but did not yield better results.
- **MultiStepLR** produced slightly higher loss.

4. Training Parameters

- **Batch Size**: Smaller batches provided faster updates but led to instability. Larger batches produced smoother gradients but reduced generalization. **32** was the optimal choice.
- **Epochs**: Fewer epochs led to underfitting, while too many caused overfitting. Increasing epochs to **1000** allowed the model to learn complex patterns effectively.

This configuration achieved the lowest validation loss while maintaining stability and avoiding overfitting.

In [104…

```
"""
RUN THIS CELL TO SAVE CONFIGS AND MODEL STATE FOR YOUR SUBMISSION
"""
def load_model(
        model_class,
        config: dict,
        state_dict: dict,
        ):
    model: nn.Module = model_class(**config).cpu()
    model.load_state_dict(state_dict)
    return model

if __name__ == '__main__':
    save_yaml(model_config, 'model_config.yaml')
    save_yaml(train_config, 'train_config.yaml')
    torch.save(model.cpu().state_dict(), 'state_dict.pth')

    # TESTING IF MODEL CAN BE LOADED WITHOUT ERRORS
    model = load_model(
        model_class = Model,
```

```
        config = load_yaml('model_config.yaml'),
        state_dict = torch.load('state_dict.pth', map_location='cpu')
    )
    print('Model can be loaded successfully!')

    # You may encounter errors when loading the model config from the yaml file.
    # If so, make sure all arguments are defined as basic python data structures li
```

```
Model can be loaded successfully!
```

```
C:\Users\RyanWu\AppData\Local\Temp\ipykernel_34304\3889064912.py:22: FutureWarning:
You are using `torch.load` with `weights_only=False` (the current default value), wh
ich uses the default pickle module implicitly. It is possible to construct malicious
pickle data which will execute arbitrary code during unpickling (See https://github.
com/pytorch/pytorch/blob/main/SECURITY.md#untrusted-models for more details). In a f
uture release, the default value for `weights_only` will be flipped to `True`. This
limits the functions that could be executed during unpickling. Arbitrary objects wil
l no longer be allowed to be loaded via this mode unless they are explicitly allowli
sted by the user via `torch.serialization.add_safe_globals`. We recommend you start
setting `weights_only=True` for any use case where you don't have full control of th
e loaded file. Please open an issue on GitHub for any issues related to this experim
ental feature.
  state_dict = torch.load('state_dict.pth', map_location='cpu')
```

Zip submission files

You can run the following cell to zip the generated files for submission.

If you are on Colab, make sure to download and then upload a completed copy of the notebook to the working directory so the code can detect and include it in the zip file for submission.

In [104…

```
files_to_zip = ['HW2.ipynb', 'model_config.yaml', 'train_config.yaml', 'state_dict.
output_zip = 'HW2_submission.zip'
zip_files(output_zip, *files_to_zip)
```