HW2

January 31, 2025

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`

1 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.

[1]:
```
# 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
```

1.1 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.

[16]:
```
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
```

```
        self.data_tensor = torch.tensor(data, dtype=torch.float)
        '''
        self.data_tensor = torch.from_numpy(data).float()

        Torch.tensor makes a copy of the numpy array and does not point to the␣
↪same memory
        as the numpy array whereas torch.from_numpy does so changes to numpy␣
↪affect tensor too
        '''

    def __len__(self):
        return self.data_tensor.shape[0] # number of elements num-el

    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␣
↪label.
        shape of x: (5,)
        shape of y: (1,)

        This is weird to me, but I'm thinking that this implements the return␣
↪for a given
        section of the tensor being retrieved in Python's indexing fashion so␣
↪that when
        a slice of the tensor is pulled, I'm returning particular information␣
↪for that slice

        Seems to pick a certain N row and pulls the 5 columns of data and 1␣
↪label for that datapoint
        """

        slice = self.data_tensor[idx] # use input idx to get chosen datapoint␣
↪or slice by user
        x = slice[:5] # get the first 5
        y = slice[5:6] # get the last one (should be the 6th but best to␣
↪specify)
        return x, y
```

[18]:
```
# testing the shapes and dtypes

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

```
# print(len(dataset))

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,)
```

1.2 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.

[67]:
```
# AUTOGRADED
class Model(nn.Module):
    def __init__(
            self,
            input_dim: int,
            output_dim: int,
            hidden_dims: list = [1024,256,64,16],
            activation: nn.Module = nn.ReLU
        ):
        super(Model, self).__init__()

        # Establish modifiable layers
        self.hidden_layers = nn.ModuleList()

        # Append a linear layer with the first hidden layer
        self.hidden_layers.append(nn.Linear(input_dim,hidden_dims[0]))

        # Create the subsequent hidden layers.
        for i in range(1, len(hidden_dims)):
            self.hidden_layers.append(nn.Linear(hidden_dims[i-1],␣
 ↪hidden_dims[i]))

        # Store the activation function (instantiated once and reused)
        self.activation = activation()

        # Final output layer maps from the last hidden dimension to output_dim.
        self.output_layer = nn.Linear(hidden_dims[-1], output_dim)


    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!
        # Forward pass through each hidden layer with activation.
        for layer in self.hidden_layers:
            # Only apply activation for Linear layers; dropout layers pass x␣
 ↪unchanged.
            if isinstance(layer, nn.Linear):
                x = self.activation(layer(x))
            else:
                x = layer(x)
        # Final linear transformation for the output.
        x = self.output_layer(x)
        return x
```

1.3 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.

[28]:
```
# DO NOT MODIFY THIS CELL!

# The first line is called a function decorator. It's a shorthand way to wrap a␣
 ↪function with another function.

# Remember that torch always keeps track of the computations so we can␣
 ↪calculate the gradients if we want to
# This can induce unnecessary overhead when we are not training!
# By using this function decorator, we are telling torch that we are not␣
 ↪interested in keeping track of gradients.
# 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 training and evaluating)
    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 as a normal python scalar

    average_loss = total_loss / n_samples
    return average_loss
```

1.4 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.

[36]:
```
# For train function, we use this decorator to make sure that torch keeps track␣
↪of the gradients.
# 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␣
↪optimizer
    lr_scheduler_name: Union[str, None] = None, # Name of the learning rate␣
↪scheduler class in torch.optim.lr_scheduler. If None, no scheduler is used
    lr_scheduler_config: dict = {}, # default setting. You can pass more␣
↪options to the scheduler

    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
    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 class from torch.optim
    # For example, getattr(optim, 'Adam') or optim.__getattribute__('Adam')␣
↪gives you optim.Adam
    # pass their config dictionaries using ** to unpack it as keyword arguments
    optimizer_cls = getattr(optim, opt_name)
    optimizer = optimizer_cls(model.parameters(), **opt_config)

    scheduler = None
    if lr_scheduler_name is not None:
        scheduler_cls = getattr(torch.optim.lr_scheduler, lr_scheduler_name)
        scheduler = scheduler_cls(optimizer, **lr_scheduler_config)

    epoch_pbar = tqdm(range(1, n_epochs+1), desc='Epochs', unit='epoch',␣
↪leave=False, ncols=100)

    for epoch in epoch_pbar:

        # Each epoch will be fast. No need for a progres bar inside the epoch␣
↪for train or test batches!
        # loop over training batches using the dataloader to traing the model
        model.train().to(device)
        train_loss_epoch = 0.0
        n_train_samples = 0

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

            # accumulate training loss (multiply by batch size for proper␣
↪averaging later)
            batch_size_actual = x.size(0)
            train_loss_epoch += loss.item() * batch_size_actual
            n_train_samples += batch_size_actual
```

```
        # After the epoch is done, evaluate the model on the training and␣
↪validation set

        # Compute average training loss for the epoch
        train_loss_epoch /= n_train_samples

        # Evaluate on the validation set
        val_loss_epoch = evaluate(model, val_loader, loss_fn=nn.
↪MSELoss(reduction='sum'), device=device)
        val_loss_epoch /= len(val_loader.dataset)

        # update the learning curve tracker and the learning rate scheduler

        # Learning Curve Tracker(assuming it takes training and validation loss)
        lct.update(train_loss_epoch, val_loss_epoch)

        # Update the learning rate scheduler if it is provided
        if scheduler is not None:
            scheduler.step()

        # Optionally update the progress bar description
        epoch_pbar.set_description(f"Epoch {epoch}: train loss␣
↪{train_loss_epoch:.4f} / val loss {val_loss_epoch:.4f}")

    return lct.get_losses()
```

1.5 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.

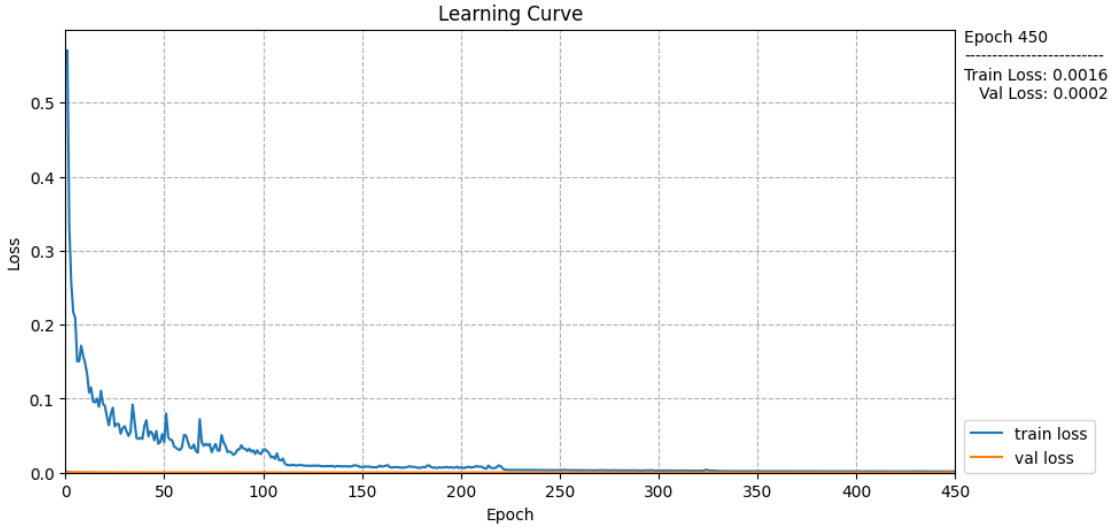
[30]:
```
# Load the data
if __name__ == '__main__':
    train_data = AirFoilDataset('data/train.npy')
    val_data = AirFoilDataset('data/val.npy')
```

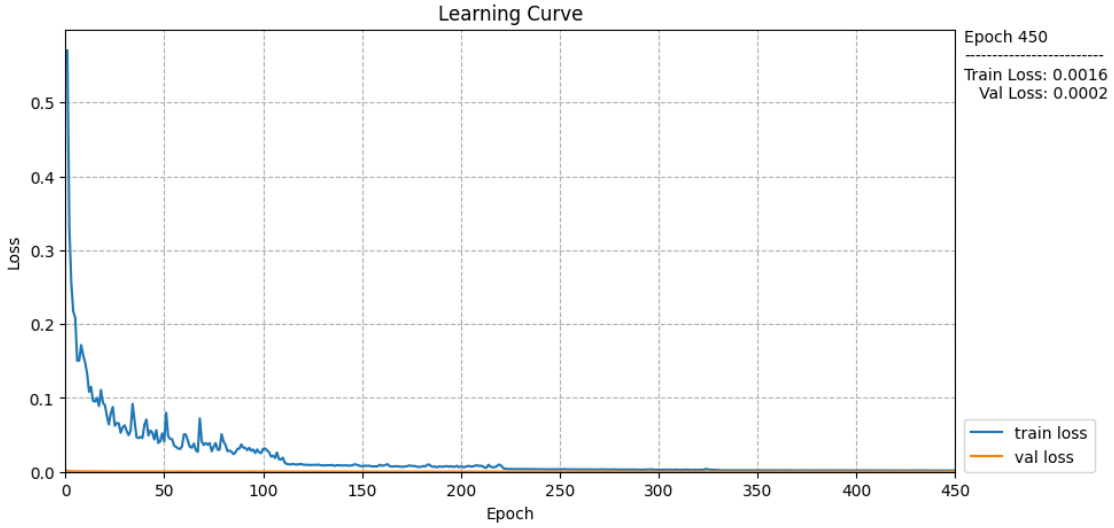
[145]:
```
# Model and training configuration
# You can iterate using this cell to find the best configuration

# Model configuration: Specify the input and output dimensions.
model_config = dict(
    input_dim=5,
    output_dim=1,
)

# Training configuration: Set the training hyperparameters.
train_config = dict(
    n_epochs=450,          # You can change this number based on your␣
 ↪experimentation.
    batch_size=32,         # Try different batch sizes; 32 is a common default.
    opt_name='Adam',       # Use Adam optimizer.
    opt_config={'lr': 0.00165},
    lr_scheduler_name='StepLR',  # If you want to use a scheduler, e.g.,␣
 ↪'StepLR'
    lr_scheduler_config={'step_size':110, 'gamma': 0.3},  # Provide␣
 ↪configuration if a scheduler is used.
)
```

[146]:
```
# train the model
if __name__ == '__main__':
    model = Model(**model_config).to(Device)
    losses = train(model, train_data, val_data, **train_config, plot_freq=50)
```





1.6 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.

Regarding my findings, I was originally experienced with much larger datasets from Systems and Toolchains so I started with 5 epochs and my loss never even made the chart, so I had a good laugh at myself as I started adding epochs up to the original 100 and immediately had my training loss at 0.32 with hidden dims of [4,3,2]. I had previously with computer vision tasks had good experience with running down only from the number of pixels to [256,64,16] in CNNs down to the

output_dims but I was afraid I would introduce way too much complexity for not enough data. I was having terrible results so I tried the [256,64,16] hidden dims and had far better results. After fooling with the lr only slightly, I thought that I might try [1024,256,64,16] introducing an even larger first layer, and that introduced incremental results as well so I really got to see how changing the number of neurons in an MLP had an effect.

While changing my learning rate had immediate impact, I instinctively knew that 1e-2 was too large for this data set of only 5 columns so I changed back to the 1e-3 and left it there. I then focused on batch size of 16, 32, 64, and 128, but all of these batch sizes performed worse than 32 and rather than introduce a non-power of 2, I decided to work with the learning rate scheduler.

Using the StepLR as suggested, I was able to adjust the step size and gamma after a large number of iterations to find that increasing the step size from the documentation's 30 all the way to 100 continually produced great results while increasing gamma had a detrimental impact after 0.3. Understanding that the step size only allows the LR to change after the every "step_size" number of epochs allows the optimizer to train with a higher LR for longer allowing the model to explore the loss surface and make significant progress while gradients are larger while later on allowing the model to fine-tune weights.

Finally, after reaching my maximum 0.43 test_loss with the learning rate scheduler, I began to finetune the LR itself, and I believe this was the best approach as it allowed me to see how the model reacted to the scheduling, allow me to finetune the effect on that scheduling with the LR after I had found the best parameters of the function if you will.

Finally, I felt the need to return to the Epochs because with such a large step size and watching the training_loss drop significantly after reaching 200, it felt like it had further to go, but I was stopping it, and I believe I was correct. I tried 250 epochs and 300 epochs and 350 (only because the plot did every 50 and the performance improvements had generally slowed signficantly) and my assumption proved that the model at least on the training loss could improve performance another tremendous amount from 0.0064 at 200 epochs to 0.0016 at 350 epochs. Finally, I tried 450 epochs because my step_size was 110, and my model would improve no longer

RESPONSE:

[147]:
```
"""
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 like int, float, str, list, dict, etc.
```

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

```
/tmp/ipykernel_21414/3889064912.py:22: FutureWarning: You are using `torch.load`
with `weights_only=False` (the current default value), which 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 future release, the default value for `weights_only` will be
flipped to `True`. This limits the functions that could be executed during
unpickling. Arbitrary objects will no longer be allowed to be loaded via this
mode unless they are explicitly allowlisted 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 the
loaded file. Please open an issue on GitHub for any issues related to this
experimental feature.
  state_dict = torch.load('state_dict.pth', map_location='cpu')
```

2 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.

[148]:
```
files_to_zip = ['HW2.ipynb', 'model_config.yaml', 'train_config.yaml',␣
↪'state_dict.pth']
output_zip = 'HW2_submission.zip'
zip_files(output_zip, *files_to_zip)
```