hw3 report

hw3: Regression

group 10

Introduction

This project aims to forecast dengue fever cases in Taiwan by treating it as a time-series regression task. The primary objective is to build a model that predicts the 7-day forward mean of confirmed dengue cases for each city. The model is trained on historical data sourced from the Taiwan Centers for Disease Control and is optimized by minimizing the Mean Squared Error between its predictions and the actual case counts.

Implementation Details

This homework is implemented in a Jupyter Notebook (running on kaggle) using PyTorch. The process can be broken down into the following:

- **Feature Engineering:**

  Most of the feature extraction and data preprocessing procedures were already provided in the starter code. In addition to these provided steps, I implemented **four additional time-series features**: `roll7_mean`, `roll14_mean`, `roll7_sum`, and `roll14_sum`.

  These features represent the mean and total number of dengue cases over the previous 7-day and 14-day windows for each city. They were created using `groupby` (to compute values per city), `shift(1)` (to avoid using future data), and `rolling()` (to calculate moving averages and sums).

```
# make 4 new columns: roll7_mean, roll14_mean, roll7_sum, roll14_sum (use DataFrame.groupby)
feat["roll7_mean"]  = feat.groupby("res_city")["cases"].transform(lambda s: s.shift(1).rolling(7,  min_periods=1).mean
())
feat["roll14_mean"] = feat.groupby("res_city")["cases"].transform(lambda s: s.shift(1).rolling(14, min_periods=1).mean
())
feat["roll7_sum"]   = feat.groupby("res_city")["cases"].transform(lambda s: s.shift(1).rolling(7,  min_periods=1).sum
())
feat["roll14_sum"]  = feat.groupby("res_city")["cases"].transform(lambda s: s.shift(1).rolling(14, min_periods=1).sum
())
```

- **Dataset Splitting:**

  Most of the dataset splitting procedures were already implemented in the starter code. I summarized them as follows:

  - **Train/Test Split:** Data from years 1998-2021 is used for training, while data from 2022-2025 is reserved for the test set.
  - **Validation Split:** A `temporal_holdout` function is used, which partitions the training data by taking the last `valid_rows` (default: 1000) as the validation set.

  After that, we need to write the member function of `DengueDataset` class. Specifically, `__getitem__` and `__len__`. Here, based on the dataloader we use (e.g. train v.s. test), labels may not be provided. Therefore, an `if` - `else` statement is used to conditionally return either `(X, y)` pairs or `X` alone.

```python
class DengueDataset(Dataset):
    def __init__(self, X: np.ndarray, y: np.ndarray | None = None):
        X = _to_float32_np(X)
        self.X = torch.from_numpy(X)
        if y is None:
            self.y = None
        else:
            y = _to_float32_np(y)
            self.y = torch.from_numpy(y)

    def __len__(self):
        # get the length of the dataset (for DataLoader)
        return len(self.X)

    def __getitem__(self, idx):
        # get item by index
        if self.y is None: return self.X[idx]
        else: return self.X[idx], self.y[idx]
```

- **Model Selection:**

For simplicity and ease of debugging, I chose a simple yet effective Feed-Forward Neural Network, implemented as the `Regressor`.

```
class Regressor(nn.Module):
    # build ur model here
    def __init__(self, input_dim: int):
        super().__init__()
        self.model = nn.Sequential(
            nn.Linear(input_dim, 16),
            nn.ReLU(),
            # nn.Linear(16, 16),
            # nn.ReLU(),
            nn.Linear(16, 1)
        )
    def forward(self, X):
        return self.model(X).squeeze(-1)
```

The architecture consists of:

1. An input layer matching the number of features.
2. One hidden layer with 16 neurons and a `ReLU` activation function.
3. An output layer with a single neuron to produce the regression value.

- **Training Loop:**
  - The model is trained using the `MSELoss` criterion and the `Adam` optimizer.

```
# give criterion and optimizer
criterion = nn.MSELoss()

optimizer = torch.optim.Adam(
    model.parameters(),
    lr=lr,
    weight_decay=weight_decay
)
```

- The following summarizes the implementation details of the training and validation process:

  First, the input data and labels are transferred to the designated device (CPU or GPU). The model is set to **training mode** (`model.train()`), where it iteratively computes predictions, evaluates the loss using the chosen criterion, performs **backpropagation**, and updates parameters through the optimizer.

  After each epoch, the model switches to **evaluation mode** (`model.eval()`) with gradient computation disabled (`torch.no_grad()`), to calculate the **validation loss** on the hold-out set. Both training and validation losses are averaged across batches. (This technique is what professor mentioned in the class: **mini-batch GD** !)

```
for epoch in range(1, n_epochs+1):
    model.train()
    losses = []
    for xb, yb in train_loader:
        # calculate loss and backprop
        xb, yb = xb.to(device), yb.to(device)
        yb_hat = model(xb) # (B, 1)
        loss = criterion(yb_hat, yb)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        losses.append(loss.item())
        step += 1


    train_loss = float(np.mean(losses)) if losses else math.inf

    # validation
    model.eval()
    v_losses = []
    with torch.no_grad():
        for xb, yb in valid_loader:
            # run the validation step
            xb, yb = xb.to(device), yb.to(device)
            yb_hat = model(xb).squeeze()
            loss = criterion(yb_hat, yb)
            v_losses.append(loss.item())
    # calculate validation loss
    valid_loss = float(np.mean(v_losses)) if v_losses else math.inf
```

Parameter Configuration and Training

- **Hyperparameter Settings:**

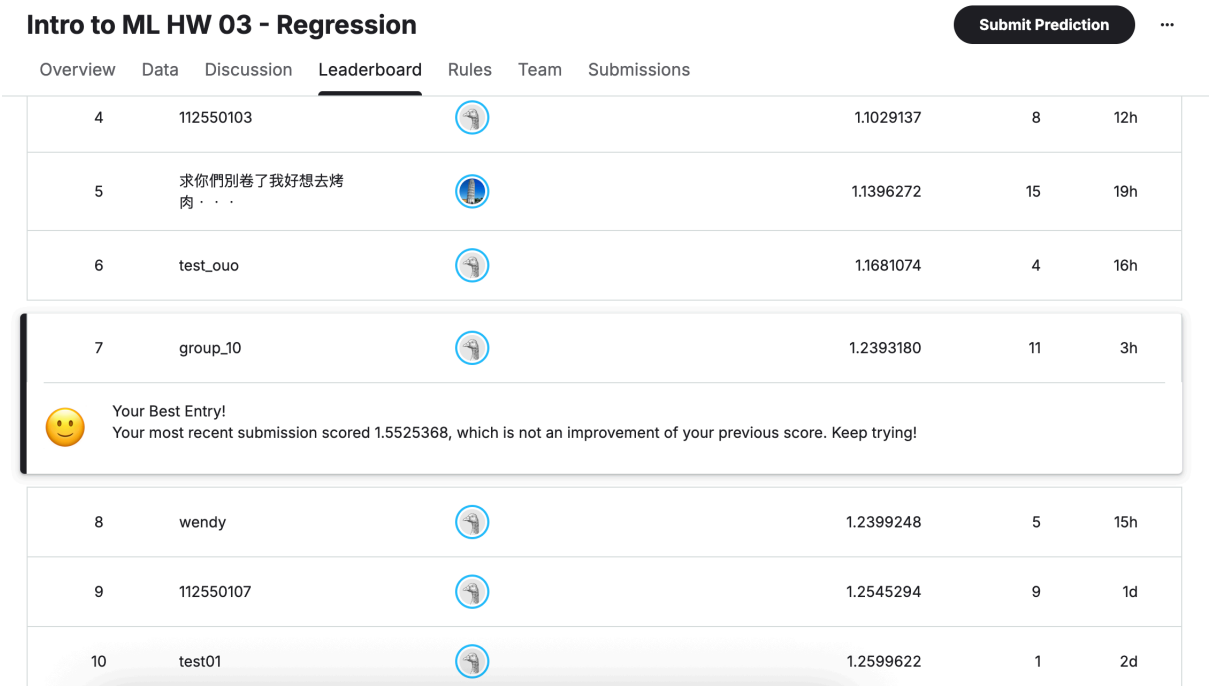
  Key hyperparameters are managed using an `ArgumentParser` for easy configuration. The specific values used in the main loop are:

  - **Learning Rate (`lr`):** `8e-4`
  - **Weight Decay:** `1e-6`
  - **Batch Size:** `512`
  - **Max Epochs:** `1000`
  - **Early Stopping:** `100`

```python
def build_arg_parser():
    # you can try different settings here
    parser = argparse.ArgumentParser()
    parser.add_argument("--seed",              type=int,   default=42)
    parser.add_argument("--valid_rows",        type=int,   default=1000, help="rows used for validation")
    parser.add_argument("--test_year_start",   type=int,   default=2022, help="first year to predict")
    parser.add_argument("--test_year_end",     type=int,   default=2025, help="last year to predict")
    parser.add_argument("--batch_size",        type=int,   default=512)
    parser.add_argument("--epochs",            type=int,   default=1000)
    parser.add_argument("--early_stop",        type=int,   default=100)
    parser.add_argument("--lr",                type=float, default=8e-4)
    parser.add_argument("--weight_decay",      type=float, default=1e-6)
    parser.add_argument("--num_workers",       type=int,   default=2)
    parser.add_argument("--tensorboard",                   default=True, action="store_true")
    return parser
```

For simplicity and ease of debugging, I initially chose a simple Feed-Forward Neural Network for the regression task. Surprisingly, even such a lightweight model performed very well, achieving a Kaggle score of **1.3506460** on the **first attempt without any hyperparameter tuning**. This suggests that the problem can be effectively modeled without requiring an overly complex architecture.

Specifically, the model consists of one hidden layer with 16 neurons and a ReLU activation function. I also experimented with deeper networks (two hidden layers) or broader networks (24 or 32 neurons), but these variations did not yield better performance compared to the single-layer configuration. However, after tuning **learning rate** to 8e-4, eventually the score of **1.2393180** can be achieved.



In addition, I explored several advanced techniques such as:

- **Dropout regularization** to prevent overfitting,
- **Learning rate scheduling** to improve convergence.

However, given the dataset's size and the simplicity of the target relationship, these techniques provided limited improvement over the baseline network.

Summary

In summary, a straightforward regression pipeline was implemented, combining proper data preprocessing, feature engineering, and a simple feed-forward neural network. Despite its simplicity, the model achieved strong performance. This demonstrates that well-designed features and a clean training setup can often outperform more complex architectures in practical regression tasks.