50.039 Theory and Practice of Deep Learning
Final Report – Group 13
Draught Prediction using Meteorological Data of Soil Conditions

Gan Chin Song, 1006283
Shelen Go, 1006236
Kripashree Suryaprakash, 1006507

GitHub: https://github.com/jsonggan/2510_deep_learning
Computer Science and Design (CSD)
Singapore University of Technology and Design

19 April 2025

1. Introduction

Droughts pose significant challenges to agriculture, water resources, and ecosystems across the United States. Accurate and timely prediction of drought levels can enable proactive measures to mitigate their impact, benefiting stakeholders in resource management and planning. This project aims to leverage deep learning techniques to predict future drought levels in US counties using historical meteorological data.

The investigation aims to determine whether a deep learning model can accurately predict drought levels in 2017–2018 across U.S. counties, based solely on meteorological data from 2000–2016 and 2019–2020, as well as soil data, which is non-time-series information. This task is critical for developing early warning systems and could have broader implications for global drought prediction if successful.

2. Datasets

We used the US Drought Meteorological Data dataset, sourced from Kaggle (https://www.kaggle.com/datasets/cdminix/us-drought-meteorological-data/data). The dataset is divided into two primary components: soil data and time-series weather and drought data, both linked by the FIPS (Federal Information Processing Standards) code, which uniquely identifies US counties. This structure enables spatial and temporal analysis of drought patterns.

2.1 Soil Data

The soil dataset contains static information characterizing soil properties for each US county, indexed by the FIPS code. The dataset includes 31 features, capturing geographical, topographical, land use, and soil quality metrics:

1. Geographical Features:

- lat, lon: Latitude and longitude.
- elevation

2. Topographical Features:
   - slope1 to slope8: Distribution of slope gradients (in percentage ranges), affecting runoff and soil erosion.
   - aspectN, aspectE, aspectS, aspectW, aspectUnknown: Proportions of land facing north, east, south, west, or unknown directions.
3. Land Use Features:
   - WAT_LAND: Water-covered land.
   - NVG_LAND: Barren/very sparsely vegetated land.
   - URB_LAND: Urban land (infrastructure and residential land).
   - GRS_LAND: Grassland/scrub/woodland.
   - FOR_LAND: Forested land.
   - CULTRF_LAND
   - CULTIR_LAND: Irrigated cultivated land.
   - CULT_LAND: Total cultivated land.
4. Soil Quality Features:
   - SQ1 to SQ7: Soil quality indices, values from 0 to 7.

2.2 Time-Series Weather and Drought Data

The time-series dataset captures daily weather and drought-related measurements for each US county, also indexed by the FIPS code and date. Each entry represents a drought level at a specific point in time, accompanied by:

1. Meteorological Indicators: 18 weather variables (e.g., precipitation, temperature, humidity).

Indicator	Description	TS	Earth Skin Temperature (C)
WS10M_MIN	Minimum Wind Speed at 10 Meters (m/s)	WS50M_RANGE	Wind Speed Range at 50 Meters (m/s)
QV2M	Specific Humidity at 2 Meters (g/kg)	WS50M_MAX	Maximum Wind Speed at 50 Meters (m/s)
T2M_RANGE	Temperature Range at 2 Meters (C)	WS10M_MAX	Maximum Wind Speed at 10 Meters (m/s)
WS10M	Wind Speed at 10 Meters (m/s)	WS10M_RANGE	Wind Speed Range at 10 Meters (m/s)
T2M	Temperature at 2 Meters (C)	PS	Surface Pressure (kPa)
WS50M_MIN	Minimum Wind Speed at 50 Meters (m/s)	T2MDEW	Dew/Frost Point at 2 Meters (C)
T2M_MAX	Maximum Temperature at 2 Meters (C)	T2M_MIN	Minimum Temperature at 2 Meters (C)
WS50M	Wind Speed at 50 Meters (m/s)	T2MWET	Wet Bulb Temperature at 2 Meters (C)
		PRECTOT	Precipitation (mm day-1)

Figure 2.1. Weather and drought dataset features

2. Score Column: drought severity score which available only once per week, resulting in NaN values for the rest of the days.
   This score is represented as a **decimal number** ranging from 0 to 5, indicating the drought level. Each score corresponds to one of the following six categorical labels:
   - None (no drought)
   - D0 (abnormally dry)
   - D1 (moderate drought)
   - D2 (severe drought)
   - D3 (extreme drought)
   - D4 (exceptional drought)

2.3 Dataset Characteristics

1. Regression Task: Since the drought score is represented as a decimal rather than a categorical (integer) value, treating it as a regression task is more appropriate. For example, a score of 1.5 and a score of 1.99 may reflect different levels of severity and could contribute differently to the decision-making process.
2. Data Splits: The dataset is pre-divided into:
   - 80% Training Datasets (2.2 GB)
   - 10% Validation Datasets (258.35 MB)
   - 10% Testing Datasets (258.43 MB)
3. Granularity and Linkage: Weekly drought score at the county level, linked via the FIPS code, enables integration of soil and weather data for comprehensive drought analysis.

4. Feature Dimensionality: The soil data contributes 31 static features per county, while the time-series data adds number of sequence x 18 meteorological indicators per entry, creating a high-dimensional input space.

2.4 Dataset Challenges

The dataset presents several challenges:

1. Data Imbalance: The distribution of drought classes is uneven, with severe drought levels (e.g., D3, D4) underrepresented compared to no drought (None) or mild conditions (D0).
2. Missing Values: NaN values in the score column (due to weekly reporting) and potentially other weather variables disrupt training.
3. High-Dimensional Input: The combination of 31 soil features and number of sequence x 18 meteorological indicators results in a high-dimensional dataset.

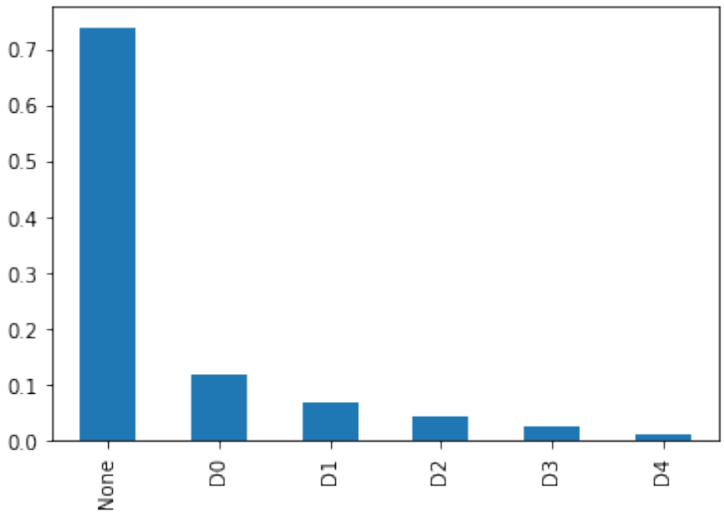


Figure 2.2. Datasets imbalanced 2123

2.5 Data Observation

All the graphs in this section can be reproduced by running observation.ipynb available in the GitHub repository.

The figure below shows the trend of drought scores from the year 2000 to 2017. As you can see, droughts do not consistently occur in a specific month of the year. However, once they begin, they tend to persist for several months. Some years experienced severe droughts, while in other years, droughts did not occur at all.

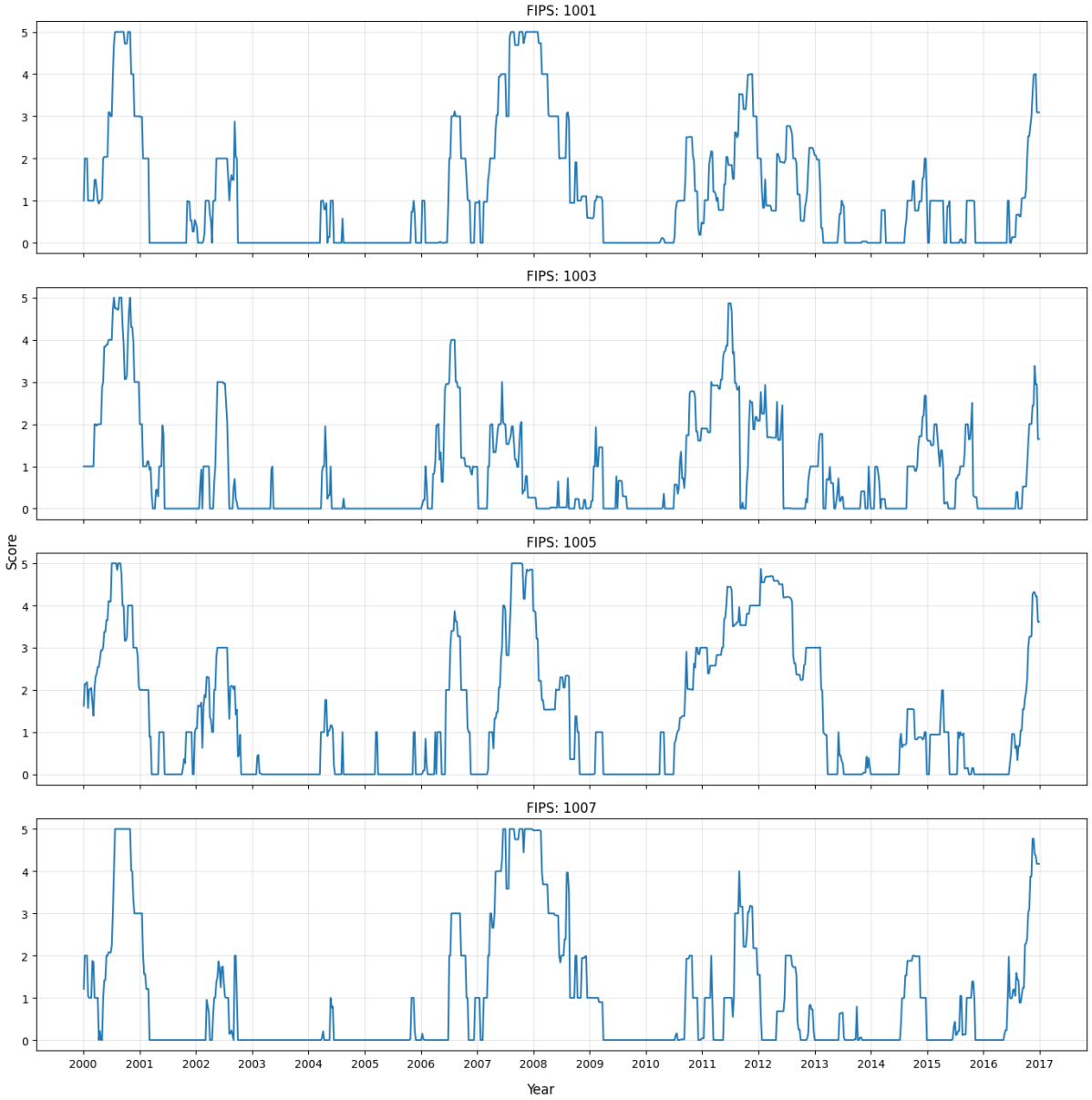


Fig 2.5.1 Four graphs of drought score vs. year

The last three digits of a FIPS code represent a specific area, while the prefix denotes a state in the U.S. As shown in the screenshot below, certain states—such as Nevada—suffer more from droughts compared to others, such as New York.

Top 5 highest-average-score FIPS areas

fips	avg_score
32001	2.433521
32027	2.421126
32031	2.250204
32029	2.215673
32019	2.208064

Top 5 lowest-average-score FIPS areas

fips	avg_score
36023	0.144355
36007	0.136731
36053	0.124738
36077	0.102989
36017	0.100410

Figure 2.5.2: Top 5 FIPS areas with the highest and lowest average drought scores

The following graph further illustrates that some regions or states are naturally more prone to drought than others.

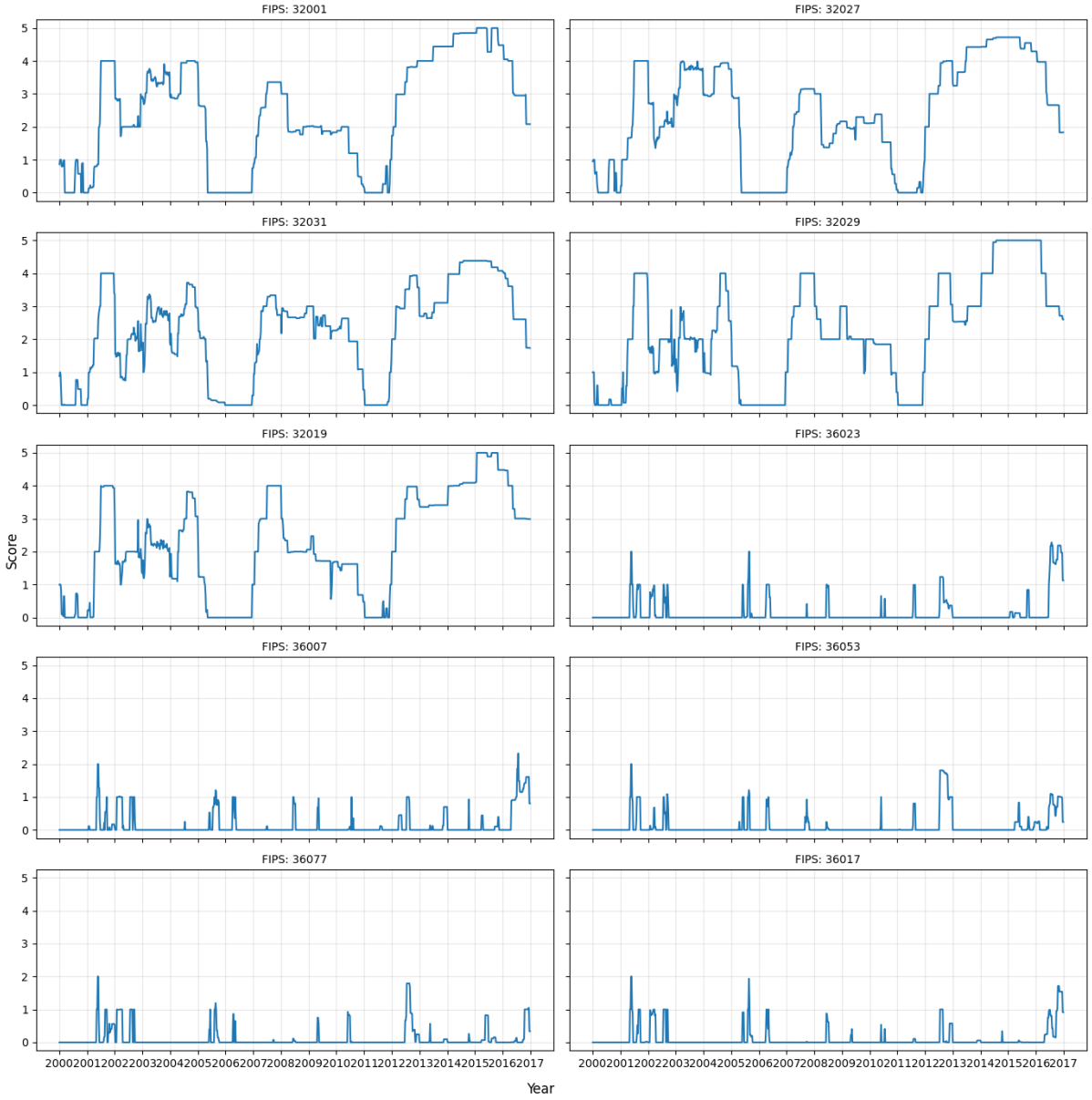


Figure 2.5.3: Drought score vs. year for the top 5 highest and lowest average score FIPS areas

3 Pre-processing Strategy

3.1 Date conversion and feature engineering by adding seasonal features

First, the 'date' column was converted to datetime format to extract temporal features like the day of the year, and the data was sorted by 'fips' (county identifier) and 'date' in ascending order to ensure chronological consistency within each county for time-series modeling. Feature engineering was applied by transforming the day of the year into season_sin and season_cos features using $\sin(2\pi \times (\text{day\_of\_year} - 1) / 365)$ and $\cos(2\pi \times (\text{day\_of\_year} - 1) / 365)$, capturing the cyclical nature of seasons. Finally, the time-series dataframes were

merged with soil_df on 'fips' using a left join, adding 31 static soil features per county.

```
#Merge soil data and add seasonal features
for df in [train_df, val_df, test_df]:
    df['date'] = pd.to_datetime(df['date'])
    df.sort_values(['fips', 'date'], inplace=True)
    day_of_year = df['date'].dt.dayofyear
    df['season_sin'] = np.sin(2 * np.pi * (day_of_year - 1) / 365)
    df['season_cos'] = np.cos(2 * np.pi * (day_of_year - 1) / 365)
train_df = train_df.merge(soil_df, on='fips', how='left')
val_df = val_df.merge(soil_df, on='fips', how='left')
test_df = test_df.merge(soil_df, on='fips', how='left')
```

Figure 3.1. Code of the preprocessing dataset which involves date conversion and feature engineering

3.2 Normalization with feature scaling

Features with different ranges (e.g., T2M in degrees Celsius vs. elevation in meters) can cause gradient imbalances, leading to unstable training or slow convergence.

Standardization (zero mean, unit variance) ensures all features contribute equally by normalizing them to the same scale, preventing larger-scale features (e.g., elevation) from dominating smaller ones (e.g., season_sin, ranging from -1 to 1). The high-dimensional input (n days × 18 variables, plus 31 soil features, plus 2 seasonal features) benefits from this, ensuring stable training.

This normalization allows the model to learn from both meteorological and soil data without scale-induced bias.

```
#Normalization with feature scaling
train_features = train_df[weather_features + soil_features].values
feature_mean = np.nanmean(train_features, axis=0)
feature_std = np.nanstd(train_features, axis=0)
feature_std = np.where(feature_std == 0, 1.0, feature_std)
for df in [train_df, val_df, test_df]:
    df[weather_features + soil_features] = (
        df[weather_features + soil_features].values - feature_mean
    ) / feature_std

dynamic_features = weather_features + ['season_sin', 'season_cos']
static_features = soil_features
target = 'score'
```

Figure 3.2.1 Preprocessing code to normalize the dataset with feature scaling

4 Models Evaluated

We evaluated a total of 3 different models include LSTM, GRU and Transformer and evaluated using MAE for the ease of comparison with other Kaggle notebooks result.

User	*Model/Notebook*	Macro F1 Mean	MAE Mean
@cdminix	LSTM Baseline	0.639	0.277
@epistoteles	Ridge Regression (default features)	0.579	0.255
@epistoteles	Ridge Regression (MiniROCKET features)	0.444	0.372

Fig. 4.1 Kaggle Notebook Result

The best result is achieved using a transformer, with a MAE of 0.1949, which is lower than the best result available on Kaggle (0.277).

4.1. LSTM

The best result from our LSTM model achieved a test MAE of **0.4199.**

4.1.1 Pre-processing Strategy

Using the same steps as explained in section 3, then for the sequence input feed into the model, it was prepared as follows:

Sequence input preparation

In the initial iteration, both the features from the weather dataset (dynamic features) and soil dataset (static features) are feed in together as a combination of features without separating them when feeding it into the model. This cause some issues:

- The model is seeing the same soil values repeated every timestep as for the same fips, it would have the same soil features data.
- It tries to learn time-dependent patterns in features that never change (e.g., same soil data every time step).
- This confuse the model and make learning slower and less accurate.

Hence, the features are separated into dynamic features and static features. Refer to figure 4.1.1 for the code of the DroughtDataset which prepares the input sequence.

```python
class DroughtDataset(Dataset):
    def __init__(self, df, dynamic_features, static_features, target, seq_length=30):
        self.dynamic_features = dynamic_features
        self.static_features = static_features
        self.target = target
        self.seq_length = seq_length
        self.data = {}
        self.indices = []
        df = df.sort_values(['fips', 'date']).copy()
        for fips, fips_df in df.groupby('fips'):
            X_dynamic = fips_df[dynamic_features].values.astype(np.float32)
            X_static = fips_df[static_features].iloc[0].values.astype(np.float32)
            y = fips_df[target].values.astype(np.float32)
            self.data[fips] = {'X_dynamic': X_dynamic, 'X_static': X_static, 'y': y}
            for i in range(len(fips_df) - seq_length):
                if not np.isnan(y[i + seq_length]):
                    self.indices.append((fips, i))

    def __len__(self):
        return len(self.indices)

    def __getitem__(self, idx):
        fips, start_idx = self.indices[idx]
        X_dynamic = self.data[fips]['X_dynamic'][start_idx:start_idx + self.seq_length]
        X_static = self.data[fips]['X_static']
        y = self.data[fips]['y'][start_idx + self.seq_length]
        return (torch.from_numpy(X_dynamic), torch.from_numpy(X_static), torch.tensor(y, dtype=torch.float32))
```

Figure 4.1.1. Code of the DroughDataset class which prepares the input sequence to the model

This prepares sequential input data by grouping records by location (fips) and sorting by date. For each location, it extracts sequences of dynamic features over a defined length (seq_length) and pairs them with static features (which remain constant per location). Only sequences with non-missing target values at the prediction point are kept. Each data point returned includes a dynamic sequence, static features, and the corresponding target value for model training.

4.1.2 Model Implementation

The LSTM architecture consist of:

- *self.lstm:* Input: (batch_size, seq_length, dynamic_input_size), Output: (batch_size, seq_length, hidden_size) but only the last timestep's output is used (lstm_out[:, -1, :])
- *self.fc_static:* Input: (batch_size, static_input_size), Output: (batch_size, hidden_size) after ReLU
- *self.fc_final:* Concatenates lstm_out and static_out → (batch_size, hidden_size * 2)

Initial Model Configuration (Without Dropout and Early Stopping)

We began by testing the model without dropout or early stopping to establish a baseline. The hyperparameters explored were:

- Learning rate: 0.001
- Hidden size: 64 (indicating the number of units in the hidden layers)
- Number of layers: 2
- Sequence lengths: 30, 90, 180 (the number of time steps or input features in each sequence)
- Batch sizes: 32, 512, 2048 (the number of samples processed before updating weights)
- Number of epochs: 10 (full passes through the training data)

Results Without Regularization

The best test MAE achieved was 0.4227, obtained with a sequence length of 90 and a batch size of 2048.

Training Strategy

To enhance the model's performance, we implemented a structured training strategy that integrates an optimizer, a learning rate scheduler, regularization techniques such as dropout, and early stopping. (The dropout of the model is implemented inside the LSTM layer itself)

For optimization, we employed *AdamW*, an adaptive optimizer that adjusts learning rates based on gradient statistics while incorporating decoupled weight decay to mitigate overfitting. To further enhance convergence, we integrated PyTorch's *ReduceLROnPlateau* scheduler, which adaptively lowers the learning rate when the validation MAE plateaus, allowing for finer learning rate control during training. Additionally, Mean Squared Error (MSE) was used as the loss function to guide the model's learning process during training.

Hyperparameter Tuning with Dropout, Scheduler and Early Stopping

Building on the baseline, we introduced dropout and early stopping, fixing the batch size at 2048 (the best performers from the initial tests). The test MAE is evaluated based on the model parameters that achieved the lowest validation MAE. We then tuned the following hyperparameters:

1. Configuration 1:
   - Sequence length: 90
   - Dropout rate: 0.2
   - Learning rate: 0.001
   - Weight decay: 1e-5

Result: Test MAE of 0.4222. This setup generalized well, showing stable performance with minimal overfitting. (Refer to figure 4.1.2.1 for the graph visualization of the training and validation loss as well as the validation MAE graph)

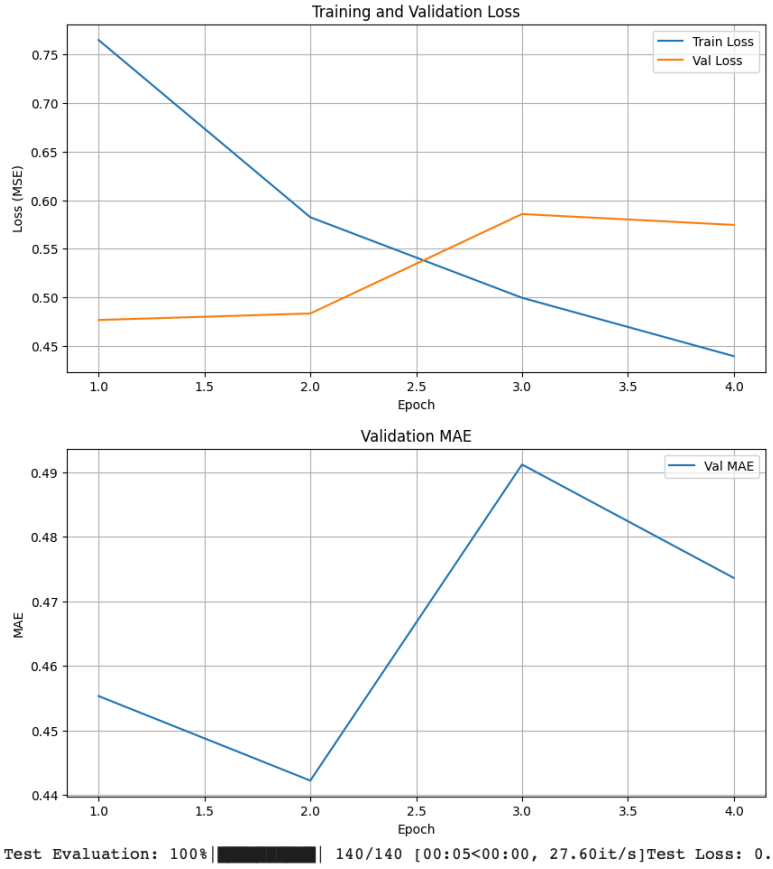
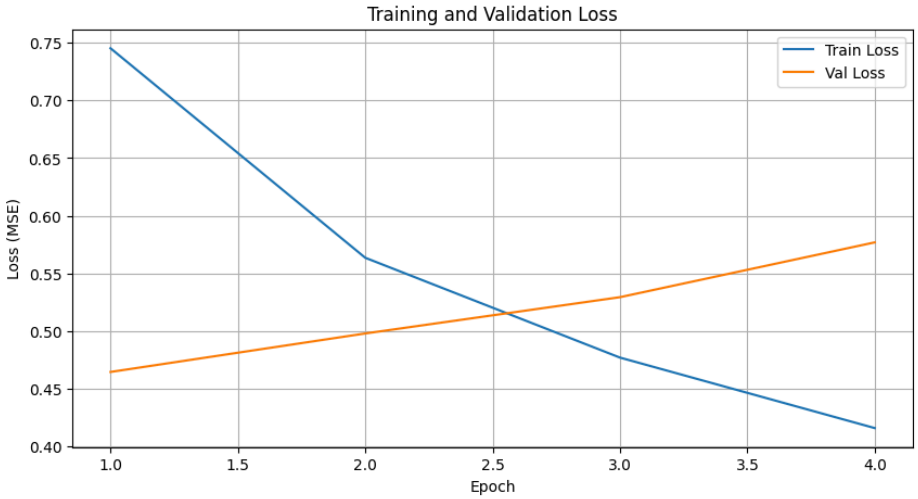


Figure 4.1.2.1. Training and validation graph as well as the validation MAE graph for the model configuration 1

2. Configuration 2:
   - Sequence length: 90
   - Dropout rate: 0.1 (reduced from 0.2)
   - Other parameters unchanged (learning rate 0.001, weight decay 1e-5)

Result: Test MAE of 0.4219. Slightly better than Configuration 1, but it exhibited more overfitting, suggesting that lower dropout allowed the model to memorize training data more. (Refer to figure 4.1.2.2 for the graph visualization of the training and validation loss as well as the validation MAE graph)

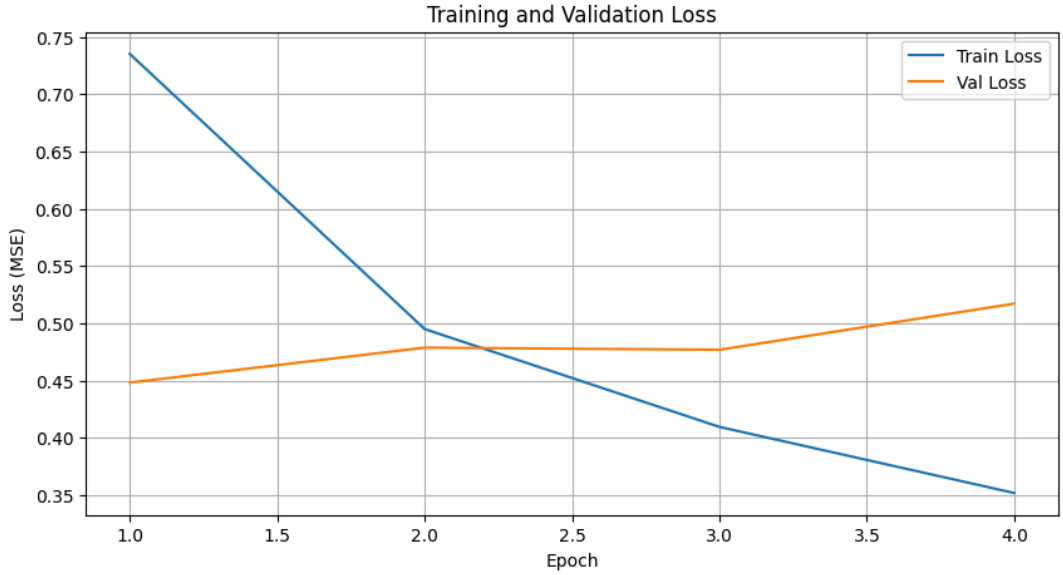


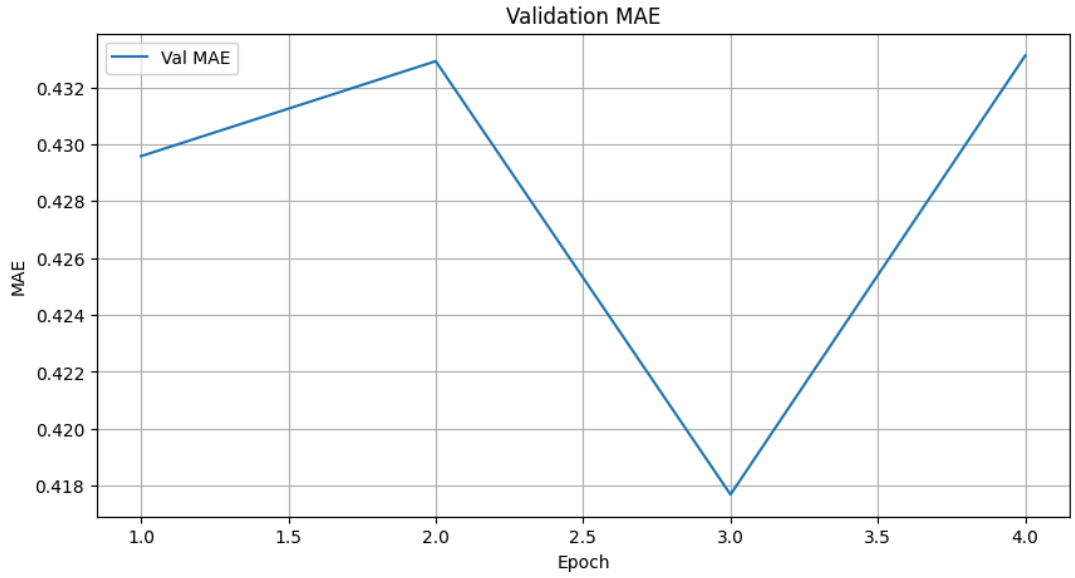

```
Test Evaluation: 100%|██████████| 140/140 [00:05<00:00, 24.93it/s]Test Loss: 0.4939, Test MAE: 0.4219
```

Figure 4.1.2.2 Training and validation graph as well as the validation MAE graph for the model configuration 2

3. Configuration 3: (best results)
   - Sequence length: 180 (increased from 90)
   - Dropout rate: 0.2
   - Learning rate: 0.001
   - Weight decay: 1e-5

Result: Test MAE of 0.4199, the best performance so far. The longer sequence length provided richer context, improving predictions, though mild overfitting persisted. (Refer to figure 4.1.2.3 for the graph visualization of the training and validation loss as well as the validation MAE graph)





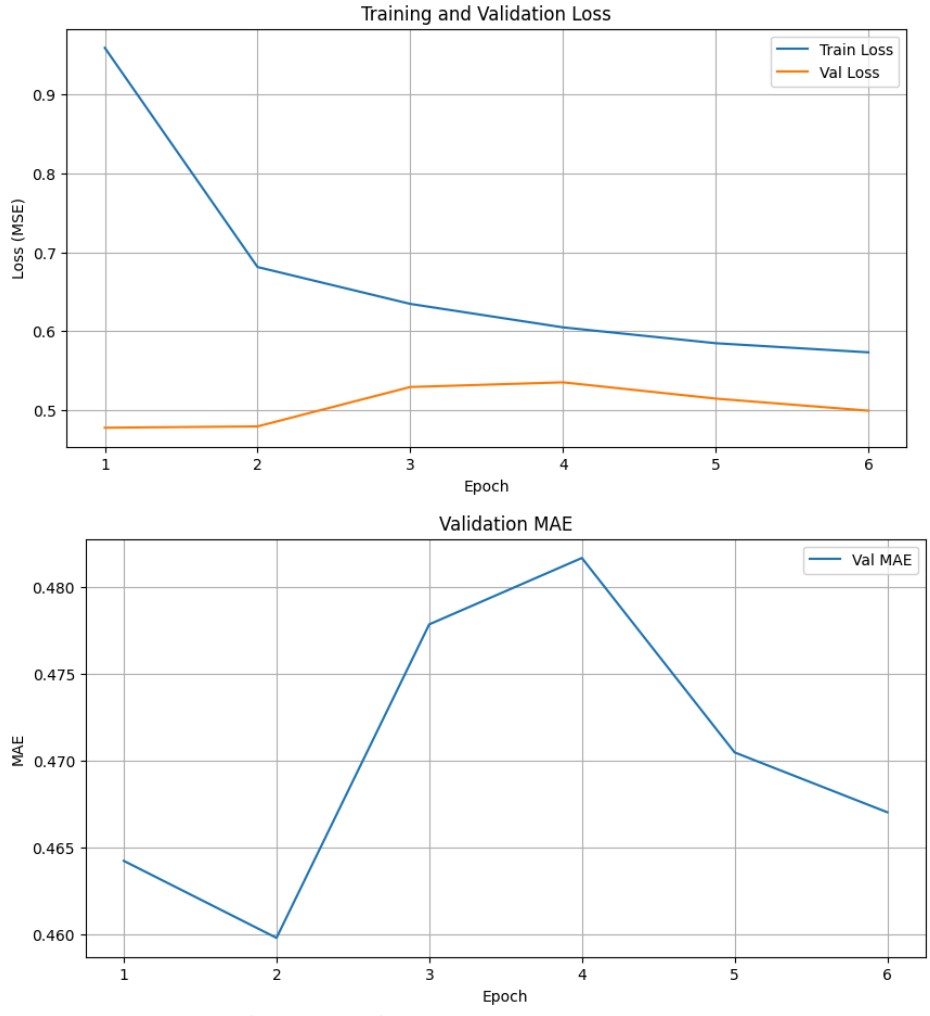

```
Test Evaluation: 100%|██████████| 120/120 [00:05<00:00, 23.77it/s]
Test Loss: 0.4965, Test MAE: 0.4199
```

Figure 4.1.2.3 Training and validation graph as well as the validation MAE graph for the model configuration 3

4. Configuration 4:
   - Sequence length: 180
   - Dropout rate: 0.4 (increased to combat overfitting)
   - Learning rate: 0.0001 (reduced for finer adjustments)
   - Weight decay: 1e-5

Result: Test MAE of 0.4824. Overfitting was eliminated, but performance dropped significantly, due to excessive regularization and a learning rate too small to effectively train the model. (Refer to figure 4.1.2.4 for the graph visualization of the training and validation loss as well as the validation MAE graph)

Figure 4.1.2.3 Training and validation graph as well as the validation MAE graph for the model configuration 3




```
Test Evaluation: 100%|██████████| 120/120 [00:05<00:00, 21.43it/s]Test Loss: 0.5727, Test MAE: 0.4824
```

Figure 4.1.2.4 Training and validation graph as well as the validation MAE graph for the model configuration 4

Results of LSTM

By incorporating dropout and early stopping, we tuned hyperparameters to achieve a best MAE of **0.4199** (sequence length 180, dropout 0.2 which is the configuration 3), though

Configuration 2 (MAE 0.4219) offered a more robust alternative. Meanwhile, configuration 4 shows no sign of overfitting, it swings too far toward generalization, sacrificing too much accuracy due to excessive regularization and a tiny learning rate, resulting in underfitting. The model parameter of the best results (configuration 3) can be accessed through this Google Drive link: https://drive.google.com/drive/folders/10bdCuT-0ztD-lIT4Z2f5lZgc5gUcGeIq?usp=share_link and follow the instructions in LSTM.ipynb to use it.

4.2 GRU

The best result from our LSTM model achieved a test MAE of **0.3794**.

4.2.1. Pre-processing Strategy

The pre-processing strategy for GRU implemented as mentioned at section 3 and the input sequence preparation was implemented the same was as for LSTM (as shown in section 4.1.1)

4.2.2. Model Implementation

The initial implementation of the GRU model is as follows (**Version 1**):

- self.gru: Input: (batch_size, seq_length, dynamic_input_size + static_input_size), Output: Tuple (output, h_n) which consist of:
  - output: (batch_size, seq_length, hidden_size) → Not used
  - h_n: (num_layers, batch_size, hidden_size)
  - Only the last layer's final hidden state is used: h_n[-1] → (batch_size, hidden_size)
- self.fc: Input: (batch_size, hidden_size), Output: (batch_size, 1) → Final prediction per sample

GRU receives a concatenation of dynamic and static inputs repeated over the sequence and the dropout is applied inside the GRU layer itself. After the GRU and fully connected layer, the output is (batch_size,) (squeezed from (batch_size, 1)).

Training Strategy

The model is trained the same way as how LSTM training was implemented where we integrates an optimizer (*AdamW*), a learning rate scheduler (*ReduceLROnPlateau*), regularization techniques such as dropout, and early stopping. Additionally, Mean Squared Error (MSE) was used as the loss function to guide the model's learning process during training.

Next, the model was trained with four different dropout configurations: no dropout, dropout rates of 0.2, 0.4, and 0.5. The goal was to identify which model configuration overfits and which one generalizes best to unseen data, as evaluated by test set performance.

The hyperparameters used are (these hyperparameters remained the same on all configuration below and the results achieved are done without seeding):

- learning rate = 0.001
- weight decay = 1e-5
- hidden size = 64
- number of layers = 2
- batch size = 2048
- sequence length = 90
- number of epochs = 10

1. Configuration 1: 0 dropout rate

The model overfits the training data. The training loss decreases consistently from 0.6977 (Epoch 1) to 0.3395 (Epoch 4), while the validation loss increases from 0.4576 (Epoch 1) to 0.6037 (Epoch 4). The test loss is the highest among all models at 0.5513, with a test **MAE of 0.4297**, indicating poor generalization.

2. Configuration 2: 0.2 dropout rate

The model shows improved generalization compared to no dropout, with a test loss of 0.4707 and a test **MAE of 0.3942**. However, the validation loss still increases after the first epoch, suggesting some overfitting.

3. Configuration 3: 0.4 dropout rate

The model performs better than no dropout but worse than dropout 0.2 and 0.5, with a test loss of 0.4874 and a **test MAE of 0.4050**.

4. Configuration 4: 0.5 dropout rate

This model achieves the best generalization, with the lowest **test MAE of 0.3934** and a test loss of 0.4717, which is very close to the minimum achieved by dropout 0.2. The validation loss increases from 0.4548 (Epoch 1) to 0.5444 (Epoch 4), but the strong test performance indicates effective regularization.

Results of Initial GRU Implementation

The model with a dropout rate of 0.5 demonstrates the best generalization. It achieves the **lowest test MAE (0.3934)** and a test loss (0.4717) that is nearly identical to the lowest test loss achieved by dropout 0.2 (0.4707). The slight edge in MAE, which directly measures prediction error, makes dropout 0.5 the superior model in terms of average error on the test set. (Detailed results & implementation can be viewed on Initial_GRU.ipynb file on GitHub).

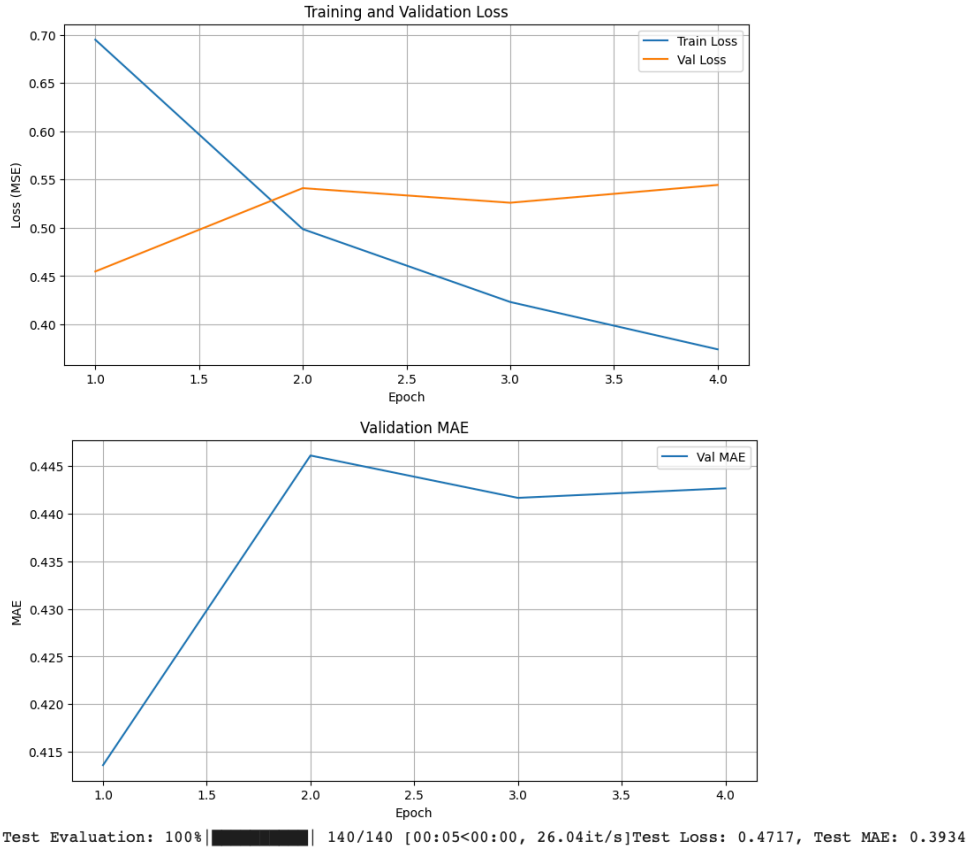


Figure 4.2.2.1 Training and validation loss graph as well as validation MAE graph for the configuration with dropout rate of 0.5

The model parameter for the figure 4.2.2.1 can be accessed through this google drive link: https://drive.google.com/drive/folders/1yOJeVe3FfRFRPtee8qFZwjHFmuDe-SWe?usp=share_link and follow the instructions in Initial_GRU.ipynb to use it.

GRU Model Alternatives and Modifications

Version 2 (Refer to the Final_GRU.ipynb file in the github for the detailed results)

The GRU model was modified by adding a layer norm and dropout layer on top of the dropout that is implemented inside the GRU layer. With the same hyperparameters settings, the model was tested with different dropout settings which are 0, 0.2, 0.3 and 0.5. The results didn't give a significant improvement compared to the previous GRU architecture. (Refer to the GRU.ipynb file in the github for the detailed results).

Version 3 (Refer to the Final_GRU.ipynb file in the github for the detailed results)

The GRU architecture was modified to apply Dropout as a separate layer outside the GRU, rather than relying solely on the built-in dropout within the GRU layer. Layer Normalization was retained in this configuration. Using the same hyperparameter settings, the model was evaluated with various dropout rates: 0, 0.2, 0.3, 0.4, and 0.5. Among these, the best performance was achieved at a dropout rate of 0.2, resulting in a test **MAE of 0.3794**.

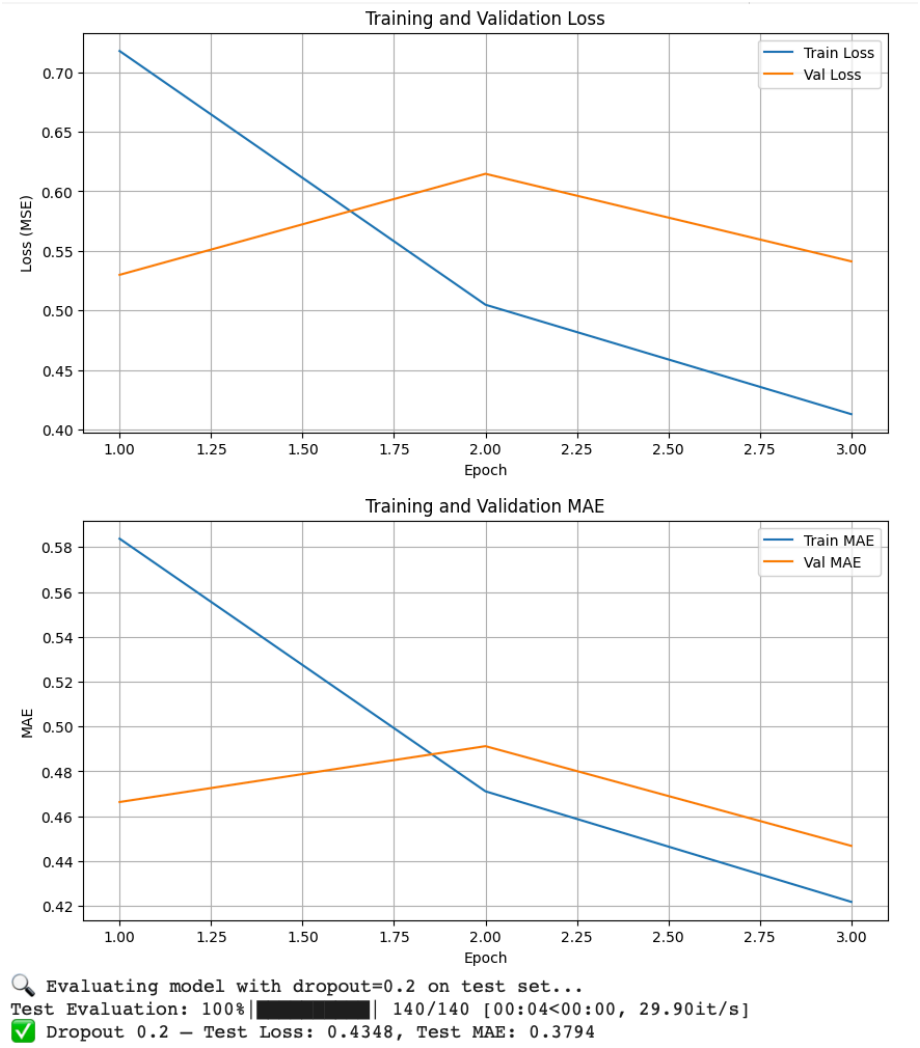


Figure 4.2.2.2 Training and validation loss graph as well as validation MAE graph for the GRU model version 2 with dropout rate of 0.2

The model parameter for the figure 4.2.2.2 can be accessed through this google drive link: https://drive.google.com/drive/folders/1i0JNR7Uo5V0KWmJTqKcTjb93dZfJaxeX?usp=share_link and follow the instructions in Final_GRU.ipynb to use it.

Final GRU Model Implementation (from the best results which is the version 3)

The GRU architecture consists of:

- self.gru: Input is the concatenation of dynamic and static features → (batch_size, seq_length, dynamic + static input size); Output is the hidden state h_n, with only the last layer's hidden state h_n[-1] used → (batch_size, hidden_size)
- self.norm: Applies Layer Normalization to h_n[-1] → (batch_size, hidden_size)
- self.dropout: Applies Dropout after normalization → (batch_size, hidden_size)
- self.fc: Final linear layer for regression → (batch_size, 1), output is squeezed to (batch_size,)

4.3 Transformer

The best result achieved by the Transformer model is a Mean Absolute Error (MAE) of 0.1949 for the 6-week drought score prediction, which is lower than the best model available on Kaggle (0.277). You can obtain the model parameter here https://drive.google.com/file/d/1nS6-S7frEegDAi3tXEfM7WjEIoSZwEwH/view?usp=sharing, and follow the instructions in transformer.ipynb to use it.

4.3.1 Preprocessing Strategy

The preprocessing method contains what we have mention in the previous section 3.1 Date conversion and feature engineering by adding seasonal feature.

The following section highlights the different preprocessing method that we try in transformer model.

4.3.1.1 Autoregressive features

score_past = score[t − 7 days] is added as a cheap yet powerful lag feature (l 92).
These hand-crafted signals give the Transformer strong periodic and memory cues without making it recurrent.

4.3.1.2 Linear interpolation within each county

All 18 weather variables plus score_past are linearly interpolated inside their own county blocks.

```
def interpolate_group(df, cols):
    arr = df[cols].values
    idx = np.arange(len(df))
    for j in range(arr.shape[1]):
        col = arr[:, j]
        mask = ~np.isnan(col)
        if mask.any():
            arr[:, j] = np.interp(idx, idx[mask], col[mask])
    df[cols] = arr
    return df
```

Fig 4.3.1 Interpolation code

# fips	date	# score
1001	2000-06-06	2.0391
1001	2000-06-07	
1001	2000-06-08	
1001	2000-06-09	
1001	2000-06-10	
1001	2000-06-11	
1001	2000-06-12	
1001	2000-06-13	3.0946

Fig 4.3.2 Drought score example

The drought score is only available once per week, and since drought conditions tend to change gradually - serious droughts don't occur overnight - a smooth interpolation works well for estimating missing values between known points.

Short gaps caused by missing satellite pixels or station outages are filled without blending statistics across counties, which helps preserve the realistic high-frequency variance in the data.

We also experimented with simply dropping rows where the score was missing, but this led to more than twice the error, with MAE values rising above **0.4**.

4.3.1.3 Robust scaling (median / IQR)

For each dynamic and static column, the median and interquartile range (IQR) are computed using only the training split, and then applied consistently to the train, validation, and test splits.

```python
def robust_fit(values: np.ndarray):
    median = np.nanmedian(values)
    q75 = np.nanpercentile(values, 75)
    q25 = np.nanpercentile(values, 25)
    iqr = max(q75 - q25, 1e-6)
    return median, iqr

def robust_transform(values: np.ndarray, median: float, iqr: float):
    return (values - median) / iqr
```

Fig 4.3.3 Code to normalize data with only NumPy

Robust scaling using the median and IQR is significantly more stable than mean/standard deviation (μ/σ) when extreme outliers are present, such as in drought spike events. It helps prevent exploding gradients and naturally clips outliers without using hard thresholds.

Prior to applying robust scaling, the model suffered from exploding gradients, which often led to the loss becoming NaN during training.

4.3.1.4 Sequence framing

The custom DroughtDataset applies a rolling look-back window across each FIPS code, pairing it with a six-week forecast horizon.
This look-back window serves as an important hyperparameter that significantly affects the model's accuracy and performance, as demonstrated in later sections.

4.3.2 Model Implementation

4.3.2.1 Dual-Branch Input Embedding

The sequence of **dynamic features** is passed through a linear layer via `self.embed = nn.Linear(dyn_in, hidden)`, which maps each 21-dimensional input vector to a shared hidden space.

Static soil and topographic features are first normalized and passed through.

```
self.fc_sta = nn.Sequential(
    nn.LayerNorm(sta_in),
    nn.Linear(sta_in, hidden),
    nn.ReLU()
)
```

This projects the 31 static features into the same hidden space and allows nonlinear interactions.

The two branches are processed independently until their latent representations are concatenated just before the prediction head.

4.3.2.2 Positional Encoding

To preserve temporal order in the dynamic input sequence, fixed sinusoidal positional encodings are added:
`x_dyn = self.embed(x_dyn) + self.pos[:, :x_dyn.size(1)]`
These help the model understand the position of each day in the sequence.

4.3.2.3 Transformer Encoder

The encoded dynamic sequence is passed through a multi-head self-attention stack using PyTorch's
`self.enc = nn.TransformerEncoder(enc_layer, layers)`

Only the final time step's output is retained using `h_dyn = self.enc(x_dyn)[:, -1]`

4.3.2.4 Fusion & Output Head

The dynamic and static hidden vectors are concatenated and passed through a fully connected head:

```
self.fc_out = nn.Sequential(
    nn.Linear(hidden*2, hidden),
```

```
    nn.ReLU(),
    nn.Dropout(dropout),
    nn.Linear(hidden, horizon)
)
```
This outputs a 6-dimensional vector representing weekly drought score predictions for the next 6 weeks. (horizon = 6)

4.3.3 Training Strategy

4.3.3.1 Optimizer & Scheduler

AdamW is used for optimization, with the learning rate following a One-Cycle policy. Training minimizes the mean absolute error (MAE), computed in PyTorch as `loss = F.l1_loss(pred, y)`

Before applying this strategy, the model suffered from overfitting due to an unadjusted learning rate. The One-Cycle policy helps accelerate early learning while gracefully annealing to a stable minimum. The model typically reaches its best weights around the first or third epoch.

4.3.3.2 Mixed Precision & Gradient Scaling

To improve efficiency, training uses PyTorch's Automatic Mixed Precision (AMP):

```
with autocast():
    pred = model(Xd, Xs)
    loss = F.l1_loss(pred, y)
```

Combined with GradScaler, this approach reduces memory usage and speeds up training. Since we're limited by computing resources, reducing memory consumption allows training on Google Colab without crashing.

4.3.3.3 Regularization Techniques

Dropout is applied both in the Transformer encoder and the MLP output head.

Gradient clipping is applied with `nn.utils.clip_grad_norm_(model.parameters(), GRAD_CLIP)`. Initially, the model suffered from exploding gradients, and this technique helps prevent unstable updates.

Stochastic Weight Averaging (SWA) is optionally enabled using:

```
swa_model = AveragedModel(model)
swa_model.update_parameters(model)
```

SWA helps smooth out the final weights and often improves generalization. It is typically enabled after epoch 4. While the model often achieves its best performance around the first or third epoch and tends to overfit afterward, SWA can still be beneficial—especially with specific hyperparameters or longer training runs.

4.3.3.4 Early Stopping & Checkpointing

Validation MAE is tracked every epoch, and the best model is saved via:
`torch.save({"model": model.state_dict()}, CKPT_PATH)`

Training stops early if performance doesn't improve for three consecutive epochs. This allows us to experiment with more model designs or hyperparameters given our limited computing resources.

4.3.4 Evaluation Protocol

4.3.4.1 Inference Procedure

During evaluation, predictions are made in mixed precision with no gradients:

```
with torch.no_grad(), autocast():
    pred = model(Xd, Xs)
```

All predictions are collected and compared to ground truth using total MAE.

4.3.4.2 Final Results

After loading the best checkpoint with:

```
ckpt = torch.load(CKPT_PATH)
model.load_state_dict(ckpt["model"])
```

the model is evaluated on the test set.

4.3.4.3 Hyper-parameter tuning

Because of limited time and GPU capacity, we tuned only a small subset of the model's hyper-parameters.
The following settings remained fixed in every experiment:
EPOCHS = 10
HORIZON = 6 (weeks to predict)
HIDDEN = 256
LAYERS = 3
HEADS = 8
LR = 3e-4
GRAD_CLIP = 1.0
USE_SWA = True
SWA_START_EP = 4

We then performed a grid search over batch size, sequence length (the rolling window), and dropout rate.

Results on the test set are summarised below:

Batch size	Sequence Length (window size)	Dropout	Test MAE
1024	240	0.4	0.2358
1024	180	0.4	0.2321
1024	90	0.4	0.2047
2048	45	0.4	0.2007
2048	21	0.4	0.1962
2048	7	0.4	**0.1949**
2048	7	0.5	0.1982
2048	3	0.4	0.2009

- Window size matters. Performance improves sharply as the look-back window is shortened from 240 days to 7 days, suggesting that the most recent week carries the bulk of the predictive signal.

- Batch size helps. Doubling the batch from 1024 to 2048 consistently lowers MAE, likely because larger batches stabilise the moving-average statistics used by SWA.

- Dropout sweet-spot. Increasing dropout from 0.4 to 0.5 degraded performance, indicating the model was already well-regularised.

- Best configuration. The combination (batch = 2048, sequence length = 7 days, dropout = 0.4) achieved the lowest Test MAE of 0.1949.

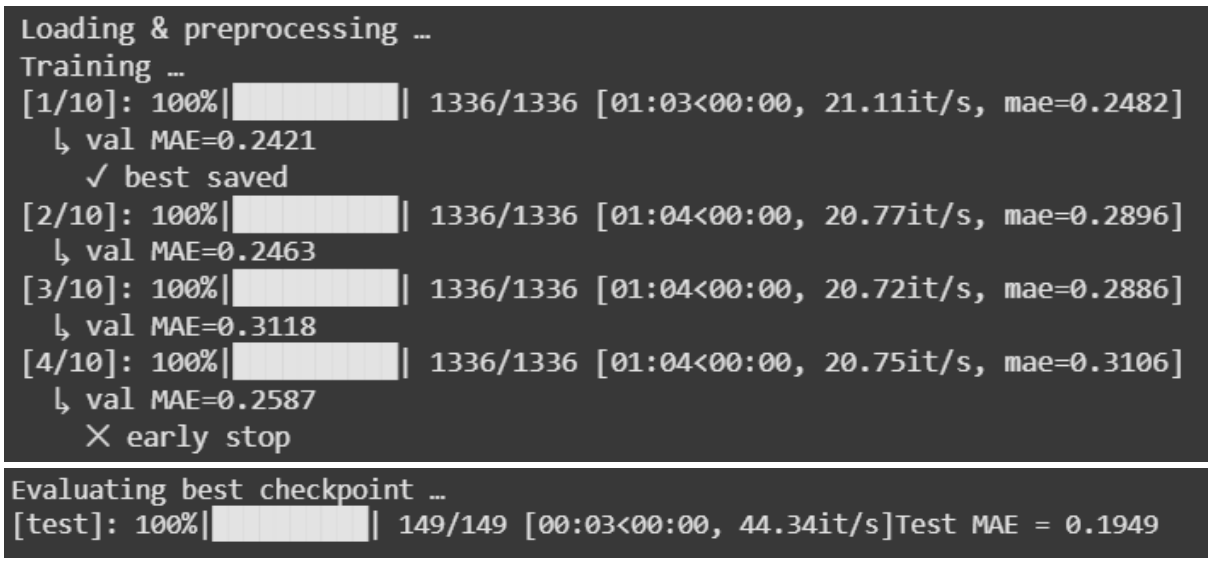


Figure 4.3.4 Screenshot of the best result

5 How to Run the Files

All models are run in Google Colab. Each individual file contains instructions on how to execute the code, as well as guidance on how to modify key parameters or hyperparameters.

6 Members' Contribution

Members	Contribution
Gan Chin Song	Preprocessing, Transformer model, Writing Report, Presentation
Shelen Go	Preprocessing, LSTM, GRU, Writing Report, Presentation
Kripashree Suryaprakash	