ChatGPT

Day 4, Session 2: LSTM Modeling for NDVI-based Drought Monitoring

In this hands-on session, we will build a Long Short-Term Memory (LSTM) model to monitor and forecast drought conditions using Sentinel-2 NDVI time series from agricultural zones in the Philippines. By the end of this 2.5-hour module, you will know how to preprocess NDVI time series data, train an LSTM to predict future NDVI values, and interpret the results in the context of drought risk.

Learning Objectives

- **Understand NDVI for drought monitoring:** Explain why NDVI (Normalized Difference Vegetation Index) can serve as a proxy indicator for vegetation health and drought stress in crops.
- **Time series preprocessing:** Learn to normalize NDVI data and convert a continuous time series into supervised learning samples using sliding windows (e.g. using 12 months of data to predict the next month's NDVI).
- **Build an LSTM model:** Define a simple LSTM-based neural network (using PyTorch, and optionally discuss Keras) that takes a sequence of NDVI values as input and outputs a predicted NDVI.
- **Train and tune the model:** Compile the LSTM model, train it on historical NDVI sequences, monitor the training process (loss curves), and apply techniques like early stopping to prevent overfitting.
- **Evaluate predictions:** Compare the LSTM's predicted NDVI against actual values, visualize the results, compute error metrics (RMSE/MAE), and reason about model performance — especially how well it captures drought-related drops in NDVI.

Case Context: Mindanao Drought Risk and NDVI

The Philippine regions of **Northern Mindanao** (which includes Bukidnon province) and **Soccsksargen** (which includes South Cotabato province) are important agricultural zones that have experienced periodic droughts. For example, during the strong El Niño of 2015–2016, Mindanao's prime corn-growing areas suffered severe drought. Satellite-derived NDVI data from NASA indicated that vegetation vigor was *"much lower than normal"* in these areas, and that **South Cotabato** (Soccsksargen) and **Bukidnon** (Northern Mindanao) were among the worst-hit regions [1]. In other words, NDVI clearly reflected the impact of drought: healthy crops typically have high NDVI (~0.6–0.9), whereas drought-stressed or barren fields show NDVI values dropping below ~0.4 [2].

Why NDVI? NDVI (Normalized Difference Vegetation Index) is a satellite-derived index measuring vegetation greenness and vigor. It is calculated from visible and near-infrared reflectance and ranges from -1 to +1 (with higher values indicating denser, healthier vegetation). NDVI is widely used for monitoring agricultural drought because it can *"identify drought-related stress to vegetation"* by tracking how green (or not) the crops are [3]. In drought conditions, vegetation greenness declines due to water stress, pests, or heat, causing NDVI values to drop noticeably. This makes NDVI an excellent proxy for assessing and even forecasting drought impacts on crops.

In this session, we focus on **NDVI-only prediction** for simplicity – using just the NDVI time series to predict future NDVI. If our model can forecast a significant NDVI drop for the next month, that's a warning sign of potential drought stress emerging. (In practice, you might also incorporate climate data like rainfall or SPEI, but here we'll develop the core methodology with NDVI alone.)

Module 1: Load and Inspect NDVI Time Series Data

First, let's load the NDVI dataset and understand its structure. The dataset consists of **monthly NDVI values over 3+ years** (e.g., 2018–2021) for one or more agricultural locations in Mindanao (such as a site in Bukidnon and another in South Cotabato). The data has been pre-cleaned, so we can directly read it from a CSV file or NumPy array.

Assume we have a CSV file `mindanao_ndvi.csv` where each column is a location's NDVI time series and each row is a monthly observation. The first column might be a date or month index (we can generate one if not). For example, the CSV might look like:

```
Month, NDVI_Bukidnon, NDVI_SouthCotabato
2018-01, 0.55, 0.50
2018-02, 0.57, 0.52
...       ...   ...
```

Let's load the data using pandas and inspect it:

```python
import pandas as pd

# Load NDVI time series data
data = pd.read_csv('mindanao_ndvi.csv')
print(data.head())          # show first few rows
print(data.columns)         # show column names (locations)
print(data.shape)           # e.g., (N_months, N_locations+1)
```

After loading, we might have a DataFrame where each location's NDVI is in a column. For instance, `NDVI_Bukidnon` and `NDVI_SouthCotabato` could be two columns. We should convert these columns into NumPy arrays for easier handling in PyTorch later:

```python
ndvi_bukidnon = data['NDVI_Bukidnon'].values
ndvi_southcot = data['NDVI_SouthCotabato'].values
time = data['Month'].values  # time labels (if present)
print("Bukidnon NDVI series length:", len(ndvi_bukidnon))
```

Now, **plot the NDVI time series** for each location to visualize their behavior over time:

```python
import matplotlib.pyplot as plt

plt.figure(figsize=(8,4))
plt.plot(time, ndvi_bukidnon, label='Bukidnon')
plt.plot(time, ndvi_southcot, label='South Cotabato')
plt.title('Monthly NDVI Time Series (2018–2021)')
plt.xlabel('Time')
plt.ylabel('NDVI')
plt.legend()
plt.show()
```

Simulated NDVI time series for two locations (Bukidnon and South Cotabato) from 2017–2020. Each line represents monthly NDVI values. Notice the seasonal oscillations (peaks during wet season, dips during dry months) and an anomalous sharp drop (shaded in orange) corresponding to a drought period. This visual pattern highlights how NDVI captures vegetation stress during droughts: values plunge below the usual range when crops are water-stressed.

Take a close look at the NDVI plots. You will likely observe **seasonal patterns** – for example, NDVI rising and falling in a yearly cycle following the rainy and dry seasons. Healthy vegetation greenness peaks in the wet season (higher NDVI) and declines during dry periods. A drought might appear as an especially pronounced or extended dip in NDVI compared to normal seasonal lows.

Think-Through: Examine the NDVI curves. Do you see regular peaks and troughs each year? How does an extreme drought manifest in the NDVI trend compared to a normal seasonal drop? *(Hint: During a drought, NDVI may drop lower and recover more slowly than usual. For instance, if NDVI typically bottoms out around 0.5 in a regular dry season, a drought year might see it fall below 0.4 – a clear stress signal.)* Also consider: *Why is NDVI a reasonable proxy for drought?* Think about what NDVI measures (vegetation greenness) and how a lack of rainfall would affect that greenness.

Mini-Challenge: Using the loaded data, plot the NDVI time series for **each location separately** (or overlay them as above). Annotate on the plot (mentally or using code) any known drought period. For example, mark the period corresponding to early 2019 if that was a drought in the region. How low did the NDVI drop relative to its usual range? Identify the **minimum NDVI value** in each series and the month it occurred. This will give you a sense of the worst drought impact in each location.

(Feel free to explore basic statistics: e.g., compute the mean NDVI for wet vs dry season months, or the standard deviation. Understanding data variability will help when setting up your prediction model.)

Module 2: Normalize and Preprocess the Time Series

Before feeding NDVI data into an LSTM, we need to preprocess it for training. There are two key steps here: **normalization** and **framing the data into sequences**.

2.1 Normalize NDVI Values

Neural networks train more effectively when input features are on a similar scale. Even though NDVI is already bounded (-1 to +1), in practice your NDVI values might lie in a narrower range (e.g., 0.3 to 0.8 for vegetated areas). We'll apply **min-max normalization** to scale the NDVI values of each series to [0, 1]. This ensures the model doesn't get biased by the absolute scale of values from different locations.

For each location's NDVI series: - Compute the minimum and maximum NDVI. - Transform each value v into `v_norm = (v - min) / (max - min)`.

We'll store the min and max so we can invert the normalization later to interpret predictions in real NDVI units.

```
# Normalize the NDVI series for each location
def min_max_scale(series):
    min_val, max_val = series.min(), series.max()
    scaled = (series - min_val) / (max_val - min_val)
    return scaled, min_val, max_val

ndvi_buk_scaled, min_buk, max_buk = min_max_scale(pd.Series(ndvi_bukidnon))
ndvi_sco_scaled, min_sco, max_sco = min_max_scale(pd.Series(ndvi_southcot))
```

(If the series is already largely between 0 and 1, scaling might not change values dramatically, but it will ensure both locations have a 0-1 range individually. In our example, Bukidnon's NDVI might naturally range 0.45–0.75 and South Cotabato 0.40–0.70; after scaling, both span 0–1.)

Why normalize? Training an LSTM involves gradient-based optimization. Features on different scales can lead to unstable gradients and slow convergence. Normalizing NDVI puts all inputs on an even footing, making training more stable. It also helps when combining data from multiple locations — e.g., one site might have consistently higher NDVI than another due to denser vegetation, but after scaling each, the patterns (not absolute values) drive the learning. Keep in mind we will later *de-normalize* the outputs to get actual NDVI predictions.

2.2 Create Sliding Window Sequences

The LSTM needs **examples of sequences** with corresponding targets to learn from. We will turn each NDVI time series into multiple training samples using a sliding window approach. The idea is to take a window of `N` time steps (months) as input and use the **next** time step as the target to predict. By sliding this window along the series, we generate many (input, output) pairs.

For example, if we choose `N = 12` (12 months = 1 year of data) to predict the 13th month: - **Input:** NDVI at months [t, t+1, ..., t+11] (12 values) - **Target:** NDVI at month t+12 (the next value)

We then slide t from the start of the series until the end, each time picking up the next 12-month chunk and the following value. This gives us a supervised learning dataset derived from the time series [4].

Let's implement this. We'll use 12 months as the sequence length (you can adjust this as needed):

```
import numpy as np

SEQ_LENGTH = 12  # using one year of past data to predict the next month

def create_sequences(series, seq_length):
    X = []
    y = []
    for i in range(len(series) - seq_length):
        x_seq = series[i : i+seq_length]
        y_val = series[i+seq_length]
        X.append(x_seq)
        y.append(y_val)
    return np.array(X), np.array(y)

# Create sequences for each location
X_buk, y_buk = create_sequences(ndvi_buk_scaled.values, SEQ_LENGTH)
X_sco, y_sco = create_sequences(ndvi_sco_scaled.values, SEQ_LENGTH)

print("Bukidnon sequences shape:", X_buk.shape, "-> targets shape:",
y_buk.shape)
print("South Cotabato sequences shape:", X_sco.shape, "-> targets shape:",
y_sco.shape)
```

If the NDVI series had, say, 48 months of data, `X_buk` will have shape (36, 12) because from 48 months we can get 36 windows of length 12 (window starting at month 0 gives target month 12, ... window starting at month 35 gives target month 47). Each input sequence is length 12, and each target is a single value. We should also add a feature dimension to these sequences for the LSTM. Currently each `x_seq` is a 1D array of length 12 (univariate time series). We need to reshape it to (sequence_length, num_features), where num_features = 1 (since NDVI is the only feature):

```
# Add feature dimension for LSTM (sequence_length, num_features)
X_buk = X_buk.reshape(-1, SEQ_LENGTH, 1)
X_sco = X_sco.reshape(-1, SEQ_LENGTH, 1)
print("After adding feature dimension:", X_buk.shape)  # e.g., (36, 12, 1)
```

Now, what about combining data from multiple locations? We have two options: - **Train one model per location:** Treat each site separately. (This is simpler, but with limited data per site, the model might underfit. Still, it allows the model to specialize to each location's patterns.) - **Train one model on all locations' data combined:** We can pool all sequences from all locations into one training set. This assumes that different locations share similar NDVI dynamics (seasonality, response to drought). The model can then learn a more general mapping from a year's NDVI to next-month NDVI, applicable to multiple areas.

To maximize training data, we'll take the second approach here and combine sequences from both locations for training. For clarity, we'll create combined arrays `X_all` and `y_all`:

```
# Combine sequences from both locations
X_all = np.vstack([X_buk, X_sco])   # stack all sequences
y_all = np.concatenate([y_buk, y_sco])
print("Combined dataset shape:", X_all.shape, y_all.shape)
```

Before training, we need to split the data into **training and testing** sets. Since this is time series data, we should use a chronological split rather than a random split (to avoid training on future data). For example, we can train on the first 80% of the timeline and test on the last 20%. However, because we combined sequences from two series, a simple chronological split is tricky – instead, let's apply the split on each location before combining:

- Choose a cut-off date (e.g., end of 2020) as the division. Sequences that end before that date will be used for training, and sequences that end on or after that date will be for testing.
- In practice, for each location, we can say the last 6 months (for example) are the test period. Any sequence that goes into those last 6 months should be part of the test set.

For example, if our data runs from Jan 2018 to Dec 2021, we could reserve Jan–Jun 2021 as test. That means any window that predicts a month in Jan–Jun 2021 is a test sample. We can implement that by index calculations or simply slicing the arrays after creating them (since sequences are in order). For simplicity, let's assume 20% of sequences for test:

```
# Simple approach: 80/20 split on the combined sequence dataset
n_samples = X_all.shape[0]
train_size = int(0.8 * n_samples)
X_train, X_test = X_all[:train_size], X_all[train_size:]
y_train, y_test = y_all[:train_size], y_all[train_size:]
print("Train samples:", X_train.shape[0], "| Test samples:", X_test.shape[0])
```

(Note: A more rigorous approach is to ensure the test set covers the most recent months for each location. For example, take the final 6 sequences from `X_buk` and `X_sco` separately as test. The above combined splitting ignores location boundaries, but since the data from each location was appended, the latter portion of `X_all` corresponds to later sequences of South Cotabato in our concatenation. In a real scenario, handle each time series split individually to avoid leakage.)

Now we have our training set (`X_train, y_train`) and test set (`X_test, y_test`), ready for modeling.

Let's double-check one sample to ensure our preprocessing is correct:

```
# Example: print first training sample (after normalization)
print("Example input sequence (normalized NDVI):", X_train[0].flatten())
print("Example target (normalized NDVI):", y_train[0])
```

This should output something like:

```
Example input sequence: [0.5  0.53 0.58 0.60 ... 0.55]  (12 values)
Example target: 0.59
```

This means, for the first training example, the model will see a year's worth of NDVI values (normalized) and try to predict the next month's NDVI (normalized).

Think-Through: Why did we choose a 12-month window? What might happen if we use a much shorter window (say 3 months) or a much longer one (24 months)? Consider the patterns the model can learn: with 3 months, it might capture short-term trends but miss annual cycles; with 24 months, it sees two years of data, which could include two wet/dry seasons, but it also means fewer training examples and potentially more noise to learn. *Also, imagine if the NDVI had an even longer-term trend (e.g., a slow upward trend due to improved farming) – would the window size affect the model's ability to learn that?*

Additionally, think about how you would modify the above approach if you wanted to predict not just the next month but *the next 3 months* of NDVI. This is a **multi-step forecasting** scenario. One approach could be to have the LSTM output 3 values instead of 1 (so `output_size = 3`). Another approach is to recursively use the model: predict 1 month ahead, append it to the sequence, slide window forward, predict the next, and so on. Each method has pros and cons – how might error compound when predicting multiple steps into the future?

Mini-Challenge: Write a function `create_sequences(series, seq_length, forecast_horizon)` that generalizes the window creation to handle a forecast horizon > 1. For example, `forecast_horizon=3` would mean each target is a sequence of 3 values (months t+12, t+13, t+14 if seq_length=12). This is an extra challenge – you can assume for now we stick to horizon=1, but it's a good mental exercise for extending the method.

Module 3: Define the LSTM Model

With our data prepared, let's define the LSTM network that will learn from these sequences. We will use **PyTorch** to build the model (you can also use TensorFlow/Keras; an example is noted later). Since this is a regression task (predicting a continuous NDVI value), our model will output a single number for each input sequence.

Model Architecture: We'll create a simple LSTM model with: - An **LSTM layer** that processes the input sequence and produces an output (hidden state) at each time step. - A **fully connected (Dense) layer** that takes the LSTM's final output and maps it to our target NDVI prediction.

Key parameters for the LSTM: - `input_size`: Number of features in each time step (for us, 1 feature = NDVI). - `hidden_size`: Number of hidden units in the LSTM. This is a hyperparameter – more units can capture more complex patterns but also risk overfitting if data is limited. - `num_layers`: We can stack multiple LSTM layers (the output of one LSTM goes into the next). We'll start with 1 layer for simplicity. - `output_size`: Number of outputs. For one-step prediction, this is 1.

In PyTorch, an LSTM layer by default returns two things: the output for **each time step** in the sequence, and the final hidden (and cell) states. We only need the final output (after the last time step), which represents the LSTM's understanding after seeing the whole sequence. We will feed that into the Dense layer to get our prediction.

Let's define the model class:

```
import torch
import torch.nn as nn

class LSTMForecast(nn.Module):
    def __init__(self, input_size=1, hidden_size=50, num_layers=1,
output_size=1):
        super(LSTMForecast, self).__init__()
        self.lstm = nn.LSTM(input_size, hidden_size, num_layers,
batch_first=True)
        self.fc   = nn.Linear(hidden_size, output_size)
    def forward(self, x):
        # x shape: (batch, sequence_length, input_size)
        lstm_out, (h_n, c_n) = self.lstm(x)
        # lstm_out: outputs for ALL timesteps (shape: [batch, seq_len,
hidden_size] since batch_first=True)
        # We want the output at the final timestep for each sequence:
        last_out = lstm_out[:, -1, :]    # shape: [batch, hidden_size]
        out = self.fc(last_out)          # shape: [batch, output_size]
        return out
```

Here, we set default `hidden_size=50` which is a moderate number of units. You can adjust this. `batch_first=True` means our input tensors will be shaped as (batch, seq_len, features), which is exactly how we prepared `X_train`. The forward pass returns `out`, the prediction for each sequence in the batch.

Now, initialize the model:

```
model = LSTMForecast(input_size=1, hidden_size=50, num_layers=1, output_size=1)
print(model)
```

This will print a summary like:

```
LSTMForecast(
  (lstm): LSTM(1, 50, batch_first=True)
  (fc): Linear(50 -> 1)
)
```

This confirms we have an LSTM that takes 1 feature and outputs 50 features internally, then a linear layer to 1 output.

Understanding the LSTM model: The LSTM's hidden state size (50) is essentially the dimensionality of the learned representation of the sequence. Think of it as the number of "features" the LSTM will internally use to summarize the sequence's information at each time step. More hidden units => potentially capturing more nuanced patterns, but also more parameters to train. We have not added any regularization (like dropout) here, but PyTorch's LSTM can apply dropout between layers if `num_layers>1`.

Also, note we used only the last output (`last_out = lstm_out[:, -1, :]`). This is a typical *many-to-one* LSTM setup, where a sequence yields one prediction. The LSTM processed the entire sequence of 12 NDVI values and condensed the relevant info into the final hidden state, which we assume carries what we need to predict the next value.

(In contrast, if we wanted to output a sequence – many-to-many – we could have a different setup or take the full `lstm_out` *and apply a linear layer at each timestep. But here we just want the next single value, so many-to-one is appropriate.)*

TensorFlow/Keras equivalent (optional): If you prefer Keras, defining a similar model is straightforward:

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM, Dense

model_keras = Sequential([
    LSTM(50, input_shape=(SEQ_LENGTH, 1)),  # 50 hidden units, returns final
output by default
    Dense(1)
])
model_keras.compile(loss='mse', optimizer='adam')
model_keras.summary()
```

This would show an LSTM with 50 units followed by a Dense with 1 output. Keras abstracts some of the details, but conceptually it's the same architecture.

Think-Through: *What would happen if we stacked two LSTM layers instead of one?* (Hint: You'd need `num_layers=2` in PyTorch or add another `LSTM(...)` layer in Keras with `return_sequences=True` for the first one. Stacking can allow the model to learn hierarchical temporal features – e.g., lower layer might capture short-term fluctuations, upper layer more long-term effects – but it also makes the model more complex.) Also consider *why we include a Dense layer after the LSTM output*. Could we directly use the

LSTM's last hidden state as the prediction? (We could if we set hidden_size=1. But generally, the Dense layer is a learnable linear mapping that can combine the hidden features into the exact output scale we need. It's like the final step of translation from internal representation to the actual NDVI value.)

Mini-Challenge: Try modifying the model in one of two ways: (a) Increase or decrease the `hidden_size` (e.g., try 16 or 100) – how do you expect this to affect learning and results? (b) Add a second LSTM layer (in PyTorch, set `num_layers=2` and maybe use a smaller hidden size in each). Don't forget if stacking LSTMs in PyTorch, you can still access only the final output as we did. If you know how, you could even add a **Dropout** layer between LSTM layers or after the LSTM to prevent overfitting. This challenge will let you see the effect of model capacity on a small dataset – sometimes a simpler model is easier to train on limited data.

Module 4: Compile and Train the LSTM Model

Now we have a model and data; it's time to train the LSTM to learn the mapping from past NDVI sequence to next-month NDVI.

4.1 Compile the Model (Loss and Optimizer)

For a regression problem like this, a common loss function is **Mean Squared Error (MSE)** – we want to minimize the squared difference between predicted NDVI and actual NDVI for the next month. We'll use MSE as our loss. Another option is Mean Absolute Error (MAE), which is more robust to outliers (large errors), but MSE is standard for optimization.

We also need an **optimizer** to update the model's weights. We'll use **Adam**, a good default optimizer that usually converges faster than plain SGD.

In PyTorch, we don't have an explicit `compile` step like Keras, but we set up the loss and optimizer as:

```
criterion = nn.MSELoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.01)
```

We chose a learning rate of 0.01; this might be adjusted based on training behavior (we can lower it if the training loss oscillates or decrease it later for fine-tuning).

(If using Keras, our `model_keras.compile(loss='mse', optimizer='adam')` *already did this.)*

4.2 Training Loop

We will train for a number of **epochs**. An epoch means going through all training samples once. Given our dataset is not huge, we can start with, say, 100 epochs and see if the model converges. We'll also implement **early stopping** to halt training if the validation loss stops improving, to avoid overfitting.

First, let's prepare the data for training: - Convert `X_train` and `y_train` to PyTorch tensors (if not already). We already shaped `X_train` as (batch_size, seq_len, 1). We might also want to use a **DataLoader** to batch and shuffle training samples each epoch for efficiency. - However, given the relatively small size, we can even feed the whole batch at once. But to mimic a realistic training, we'll use mini-batches.

Example: use all training data as one batch:

```
X_train_t = torch.tensor(X_train, dtype=torch.float32)
y_train_t = torch.tensor(y_train, dtype=torch.float32).unsqueeze(1)  # make it
(batch,1)
```

(We unsqueeze `y_train` *to have shape [batch, 1] matching the model output shape.)*

We will also create a validation split from the training data to monitor performance. Alternatively, since we set aside test data, we could use that for early stopping monitoring (though typically one uses a separate validation set). For simplicity, let's use a small portion of training data as validation (e.g., 10%):

```
val_size = int(0.1 * X_train_t.shape[0])
X_val_t = X_train_t[-val_size:]
y_val_t = y_train_t[-val_size:]
X_train_t = X_train_t[:-val_size]
y_train_t = y_train_t[:-val_size]
print("Training on", X_train_t.shape[0], "samples; Validating on",
X_val_t.shape[0], "samples.")
```

Now, the training loop with early stopping:

```
num_epochs = 100
patience = 5  # epochs to wait for improvement before stopping
best_val_loss = float('inf')
epochs_no_improve = 0

for epoch in range(1, num_epochs+1):
    model.train()  # put model in training mode
    optimizer.zero_grad()
    # Forward pass
    pred = model(X_train_t)            # predicted normalized NDVI (shape
[batch,1])
    loss = criterion(pred, y_train_t)  # MSE loss on training batch
    # Backpropagation and optimization
    loss.backward()
    optimizer.step()

    # Validation
    model.eval()  # switch to eval mode
    with torch.no_grad():
        val_pred = model(X_val_t)
        val_loss = criterion(val_pred, y_val_t)
    # Print progress
    if epoch % 10 == 0:
        print(f"Epoch {epoch}: Train loss = {loss.item():.4f}, Val loss =
{val_loss.item():.4f}")
    # Check early stopping criteria
    if val_loss.item() < best_val_loss:
        best_val_loss = val_loss.item()
        epochs_no_improve = 0
        # Optionally save the model weights here if needed
    else:
        epochs_no_improve += 1
        if epochs_no_improve >= patience:
            print(f"Early stopping on epoch {epoch} (no improvement in val loss
in last {patience} epochs).")
            break
```

This loop will train the model and monitor validation loss. We zero the gradients, do a forward pass to get predictions for all training samples, compute loss, then `backward()` to compute gradients and `step()` to update weights. On validation data (which the model hasn't directly optimized on), we just forward-pass and compute `val_loss`.

The `print` statement every 10 epochs gives an idea of how loss is decreasing. Early stopping will stop the loop if validation loss doesn't improve for 5 consecutive epochs (you might adjust patience or disable early stopping if you want to train all epochs regardless).

(In Keras, you would instead do `model.fit(X_train, y_train, epochs=100, batch_size=..., validation_data=(X_val, y_val), callbacks=[EarlyStopping(patience=5, restore_best_weights=True)] )` *to get a similar effect.)*

Note: Keep an eye on the losses. If training loss decreases but validation loss starts increasing, that's overfitting – the model is memorizing training patterns that don't generalize. Early stopping helps avoid wasting time in that regime.

Also, if the training loss plateaus and is not decreasing much, you might try lowering the learning rate or increasing training epochs.

4.3 Visualizing the Training Curve

After training, it's informative to plot the training and validation loss over epochs to see how the model learned. We could have recorded the loss values in arrays during the loop. For demonstration, suppose we did store them:

```
epochs = range(1, epoch+1)
plt.plot(epochs, train_losses, label='Training Loss')
plt.plot(epochs, val_losses, label='Validation Loss')
plt.xlabel('Epoch')
plt.ylabel('MSE Loss')
plt.legend()
plt.show()
```

You would typically see the training loss steadily decreasing. The validation loss might mirror it initially and then possibly diverge (increase) if overfitting begins. Ideally, both go down and level off, and we stop at the point where adding more epochs would not improve validation.

Think-Through: If you find that the LSTM is not improving much (loss remains high) or is very unstable (jumping around), what could be the issues? Consider possibilities such as: not enough data for the model to learn (leading to underfitting), or too high a learning rate causing gradient overshooting, or maybe the sequence length chosen doesn't capture the pattern. How might you address each scenario? For instance, *if underfitting:* increase model capacity (more hidden units or layers) or provide more data; *if overfitting:* reduce capacity or use regularization or just stop earlier; *if learning issues:* tune learning rate, ensure data is normalized, etc.

Mini-Challenge: Try training the model on **only one location's data** (e.g., just Bukidnon) and see how the performance differs from training on combined data. Does the model do better or worse when specializing on one area? Also, as an experiment, change the `SEQ_LENGTH` (try 6 or 24) in the preprocessing and retrain – how does that affect the validation loss? This will help you understand the role of the input sequence length in capturing the NDVI patterns.

(Ensure you re-generate sequences and reshape if you change SEQ_LENGTH. You might find that with only 6 months of input, the model struggles if the NDVI has a strong annual cycle that it can't fully see in 6 months.)

Module 5: Evaluate the Model and Visualize Predictions

Now for the moment of truth: how well does our trained LSTM model predict NDVI, and by extension, how well can it signal drought on the horizon? We will use the **test set** (the data we set aside and never used for training) to evaluate model performance. This will give an unbiased indication of how the model generalizes to new data.

5.1 Generate Predictions on Test Data

We prepared `X_test` and `y_test` earlier. Each `X_test` sequence corresponds to some period in time (likely the last several months of our dataset) and `y_test` is the actual next-month NDVI for those periods.

To get predictions: 1. Feed the `X_test` sequences into the model (in evaluation mode, with no gradient needed). 2. The model outputs normalized NDVI predictions. We then invert the normalization to get them back to actual NDVI values. 3. Compare with the actual `y_test` (which is also normalized in our arrays, so we need to invert those too for meaning).

Let's do that (assuming our model and min/max values are available):

```
model.eval()
with torch.no_grad():
    y_pred_norm = model(torch.tensor(X_test, dtype=torch.float32))
y_pred_norm = y_pred_norm.numpy().flatten()  # convert to 1D NumPy array
y_true_norm = y_test  # already as NumPy array

# Invert normalization for meaningful NDVI values:
# We need to apply the inverse of min_max_scale we did for each location.
# Careful: if test contains sequences from both locations, we should invert per
location for each sample.
# Simpler approach: if using one model for both, it's okay to interpret
normalized values generically 0-1 ~ min to max.
# For illustration, let's invert assuming the average min/max or using one
location's scale (not ideal but just example).
y_pred = y_pred_norm * (max_buk - min_buk) + min_buk  # here assume Bukidnon
scale
y_true = y_true_norm * (max_buk - min_buk) + min_buk
```

The above assumes a single scale; if we had separate scales per location, we would need to know which location each test sample belongs to (since each was normalized separately originally). For a robust evaluation, we might actually evaluate the model per location: e.g., take the model and feed it Bukidnon's last few sequences (normalized with Bukidnon's min/max) and invert with Bukidnon's min/max; then do South Cotabato similarly. This ensures correctness. But to keep things straightforward, imagine we evaluate on one location at a time:

```
# Example: Evaluate on Bukidnon test sequences only
X_test_buk = X_buk[-6:]  # suppose last 6 windows are test for Bukidnon
y_test_buk = y_buk[-6:]
model.eval()
with torch.no_grad():
    pred_buk_norm = model(torch.tensor(X_test_buk, dtype=torch.float32))
pred_buk_norm = pred_buk_norm.numpy().flatten()
# invert norm for Bukidnon
pred_buk = pred_buk_norm * (max_buk - min_buk) + min_buk
true_buk = y_test_buk * (max_buk - min_buk) + min_buk
```

Now we have `pred_buk` vs `true_buk` in actual NDVI values. We can do the same for South Cotabato.

5.2 Plot Actual vs Predicted NDVI

Visualization will help us see if the model is capturing the right trend and magnitude. We can plot the predicted and actual NDVI over the test time period.

Continuing the example for Bukidnon:

```
# Create a time index for test months, e.g., if test months are Jan-Jun 2021:
test_months = pd.date_range("2021-01", periods=len(true_buk), freq='M')
plt.figure(figsize=(6,4))
plt.plot(test_months, true_buk, marker='o', label='Actual NDVI')
plt.plot(test_months, pred_buk, marker='o', label='Predicted NDVI',
linestyle='--')
plt.title('Bukidnon NDVI: Actual vs Predicted (Test Period)')
plt.xlabel('Time')
plt.ylabel('NDVI')
plt.legend()
plt.show()
```

Ideally, the predicted line should track the actual line closely. If our model is good, it will follow the rises and falls of NDVI with minimal lag. If there was a drought-induced drop in NDVI during the test period, we want the model to have predicted a drop as well.

Beyond visual inspection, we should compute some error metrics: - **Root Mean Square Error (RMSE):** `sqrt(mean((pred - true)^2))` - **Mean Absolute Error (MAE):** `mean(|pred - true|)`

These give a sense of average prediction error. RMSE penalizes large errors more, while MAE is more interpretable in terms of actual NDVI units.

```
from math import sqrt
mse = np.mean((pred_buk - true_buk)**2)
rmse = sqrt(mse)
mae = np.mean(np.abs(pred_buk - true_buk))
print(f"Test RMSE: {rmse:.4f}, MAE: {mae:.4f}")
```

For example, you might get something like RMSE = 0.05 NDVI (which in NDVI terms is quite small, as NDVI ranges 0-1) and MAE = 0.04. These errors correspond to, say, a predicted NDVI being off by ~0.04 on average. Whether that's significant depends on context: an NDVI drop from 0.6 to 0.5 is meaningful for drought, so a 0.04 error might be acceptable or might mean slight mis-timing.

Interpreting Results: If the model predictions closely match the actual NDVI, great – it means the LSTM learned the temporal patterns well. Often, the model will get the general shape right (e.g., it knows NDVI should decline during dry months and recover after rains) but might **smooth out extremes**. It's common for neural network forecasts to under-predict peaks or over-predict troughs, effectively dampening the variability. This is partly because the model is trying to minimize average error (MSE) – a big spike that occurs rarely is hard to predict exactly, so the model might play it safe and predict a more moderate value.

For instance, if an actual NDVI jumps from 0.4 to 0.7 (a big greening after rain), the model might predict a rise to only 0.6. The error isn't huge, but it's noticeable. Conversely, if NDVI crashes due to a sudden drought, the model might not fully anticipate the depth of the drop if that pattern wasn't evident in the training data.

Think-Through: Look at any points where the prediction error is large. Are those corresponding to sudden changes in NDVI? Why might the model have struggled there? Consider that our model had no direct knowledge of rainfall or climate events – it's guessing the next NDVI purely from recent NDVI. If a drought hits (NDVI plummets) with little warning in the NDVI trend beforehand, the model likely cannot predict that well (it might have predicted a value closer to the previous month, resulting in a big error). This hints that incorporating rainfall or drought indices could improve the model, but our exercise shows what NDVI-alone can do and where it might fall short.

Another question: How far ahead could we reasonably forecast NDVI with this approach? One month ahead is usually okay because vegetation changes aren't completely random month-to-month. But forecasting, say, 6 months ahead in one shot would be much harder because errors compound and NDVI could be influenced by external factors (like an upcoming El Niño) not signaled in the past NDVI alone.

Mini-Challenge: Use the trained model to **forecast beyond the test set**. For example, if our data ended in Dec 2021, try to predict NDVI for Jan–Jun 2022 *even though we don't have actual values to compare*. You can do this by taking the last available 12 months from 2021 as input, predicting Jan 2022, then appending that prediction and sliding the window to predict Feb 2022, and so on (recursive forecasting). Plot the resulting forecasted NDVI curve for 6 months. While this would be an extrapolation, it's a useful exercise to see how the model behaves in truly unseen territory. Do the predictions seem reasonable (e.g., do they stay within [0,1] and follow a plausible seasonal pattern)? This will also illustrate the concept of **forecast uncertainty** – the further out we predict, generally the less confident we are.

Finally, think about how you would deploy such a model in practice. You could run this LSTM each month with the latest NDVI data to predict the next month's NDVI for these agricultural zones. If the forecast NDVI is significantly lower than the historical average for that time of year, that could be an early warning of drought stress, prompting authorities to take preemptive action (like crop insurance payouts, drought advisories, etc.).

You have now completed building, training, and evaluating an LSTM model for NDVI-based drought monitoring! This workflow – from data prep to model to evaluation – is a powerful template for many Earth observation time series forecasting tasks.

Keep experimenting with the model and data; for instance, you might try incorporating additional inputs (like soil moisture or rainfall) or predicting vegetation indices at different time scales. But even with NDVI alone, you've seen how deep learning can extract temporal patterns to provide actionable insights in environmental monitoring. Congrats on getting through this advanced session!

---

[1] Index
https://ipad.fas.usda.gov/highlights/2016/03/Philippines/Index.htm

[2] Understanding Vegetation Indices Used in Precision Agriculture - Alabama Cooperative Extension System
https://www.aces.edu/blog/topics/crop-production/understanding-vegetation-indices-used-in-precision-agriculture/

[3] Normalized Difference Vegetation Index (NDVI) – Integrated Drought Management Programme
https://www.droughtmanagement.info/normalized-difference-vegetation-index-ndvi/

4 Using PyTorch to Train an LSTM Forecasting Model | by Michael Rowe | Medium
https://medium.com/@mike.roweprediger/using-pytorch-to-train-an-lstm-forecasting-model-e5a04b6e0e67