Module 5: The Deep Learning & Hugging Face Ecosystem, Lesson 2: Training-a-Simple-Model

Hands-On: Training a Weather Model on Five Years of Orlando Data

Overview

This demo takes the abstract pieces from Lecture 2 and puts them on a live screen: a real CSV file, a real Keras model, a real training loop with early stopping, and a real test-set evaluation against a baseline. The whole workflow runs end to end in about ten minutes.

- You will load five years of daily Orlando weather observations and engineer two cyclical features that encode the seasonal pattern.
- You will split the data chronologically — never randomly — so that the model is evaluated on dates it has never seen.
- You will train a small feed-forward network with two hidden layers, watch validation loss, and stop early when it stops improving.
- You will compare the test MAE to a baseline that just predicts "tomorrow looks like today" — and prove that the model has actually learned something useful.

At the end of the demo the model and its scaler are saved to disk. Lecture 3 picks them up from there.

Before You Start

- Have Python 3.10+ and pip installed.
- Install dependencies: pip install tensorflow scikit-learn pandas numpy joblib.
- Open the file lecture2_train_weather_model.py from https://github.com/AI-Native-Engineer-Course/AI-Native-Engineer/tree/main/module 5/lecture 2.
- Have orlando_weather.csv from the same directory in GitHub. The file is 1,826 rows — five years of daily observations.
- Open a terminal in the same directory. Have the lecture slide deck open on a second screen so you can refer back to the train/val/test split diagram.

> **REMEMBER**
> Training is not where the work is. The work is in feature engineering and disciplined data splitting. A good model on bad splits will lie to you. A good split will surface a mediocre model and let you fix it. Watch how much of this script is data preparation versus actual model fitting — and notice how the model fitting is almost an afterthought.

During Hands-On — Four Steps at a Glance

- STEP 1 — Load the CSV and engineer cyclical day-of-year features. ~2 min
- STEP 2 — Chronological 80/10/10 split, then standard-scale fit on the training set only. ~2 min
- STEP 3 — Build, compile, and train the network with early stopping. ~4 min
- STEP 4 — Evaluate against a persistence baseline, then save the model and the scaler. ~2 min

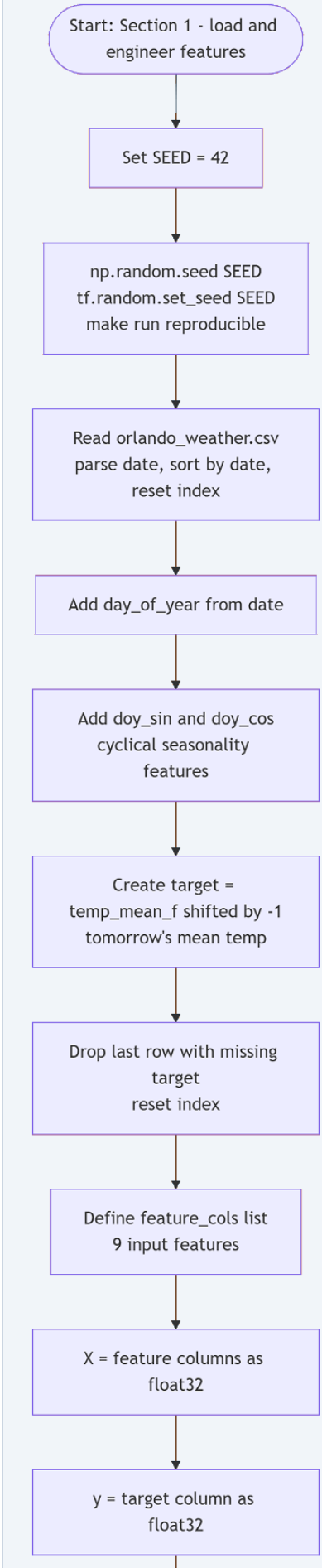
Step 1 — Load the Data and Engineer Cyclical Features

WHAT TO LEARN

A neural network does not understand that December 31 and January 1 are neighbors on the calendar. If you pass the day-of-year as a plain integer, the model thinks day 1 and day 365 are as far apart as possible. The fix is two columns: sin(2π·doy/365) and cos(2π·doy/365). Together they encode the cyclical structure that the model can actually learn from.

Open the script and walk through the load step:

```
df = pd.read_csv('orlando_weather.csv', parse_dates=['date'])
df['doy'] = df['date'].dt.dayofyear
df['doy_sin'] = np.sin(2 * np.pi * df['doy'] / 365.0)
df['doy_cos'] = np.cos(2 * np.pi * df['doy'] / 365.0)
```


Start: Section 1 - load and engineer features
Set SEED = 42
np.random.seed SEED
tf.random.set_seed SEED
make run reproducible
Read orlando_weather.csv
parse date, sort by date,
reset index
Add day_of_year from date
Add doy_sin and doy_cos
cyclical seasonality
features
Create target =
temp_mean_f shifted by -1
tomorrow's mean temp
Drop last row with missing
target
reset index
Define feature_cols list
9 input features
X = feature columns as
float32
y = target column as
float32

```
Flow chart of the program’s beginning.
```

WHAT YOU SHOULD GET

A DataFrame of 1,826 rows with columns for temperature, precipitation, wind, humidity, pressure, plus the two engineered cyclical features. The model figures out, on its own, that July is hot and January is cool.

Step 2 — Chronological Split and Scaling

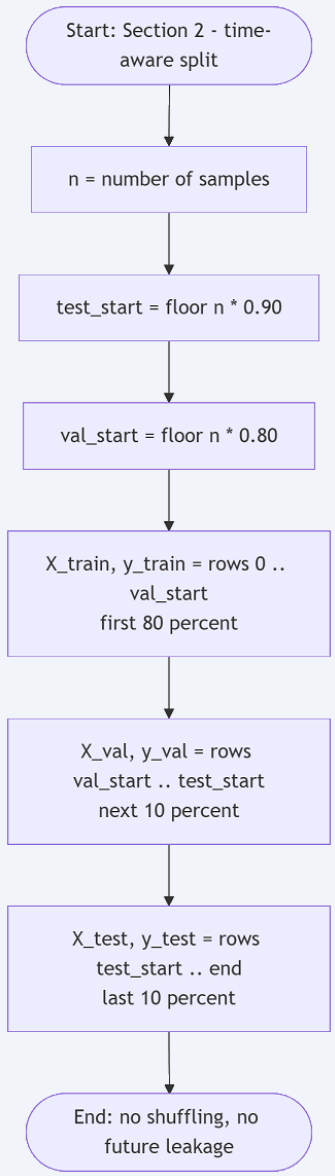
WHAT TO LEARN

In tabular ML it is tempting to shuffle the data and split randomly. For time series that is a leak. Future days end up in the training set and the model learns information it would never have at prediction time. The correct split is chronological: oldest 80% for training, next 10% for validation, most recent 10% for the final test. And the scaler must be fit on the training set alone — never on the test set.

The relevant lines in the script:

```
n = len(df)
train_end = int(0.80 * n)
val_end   = int(0.90 * n)
train = df.iloc[:train_end]
val   = df.iloc[train_end:val_end]
test  = df.iloc[val_end:]

scaler = StandardScaler().fit(train[FEATURES])
X_train = scaler.transform(train[FEATURES])
X_val   = scaler.transform(val[FEATURES])
X_test  = scaler.transform(test[FEATURES])
```



Flow chart for section 2.

WHAT YOU SHOULD GET

Three arrays with shapes that approximately match the 80/10/10 ratio. The scaler is fit only on X_train. Iif you reverse the order of these two lines — fit the scaler on the whole dataset before splitting — you will get a slightly lower test MAE and a model that quietly cheats every time it runs in production.

Step 3 — Build, Compile, and Train

WHAT TO LEARN

The network itself is almost boring. Two hidden Dense layers with ReLU, one dropout for regularization, one output neuron with no activation (we are predicting a temperature, not a probability). Adam optimizer, MSE loss, EarlyStopping with patience 15. That is the entire pattern you will reuse for ninety percent of tabular regression problems for the rest of your career.

The model definition is short — read it line by line with the room:

```
model = Sequential([
    Input(shape=(len(FEATURES),)),
    Dense(64, activation="relu"),
    Dropout(0.15),
    Dense(32, activation="relu"),
    Dense(1),  # regression output, no activation
])
model.compile(optimizer="adam", loss="mse", metrics=["mae"])

early = EarlyStopping(patience=15, restore_best_weights=True)
history = model.fit(
    X_train, y_train,
    validation_data=(X_val, y_val),
    epochs=200, batch_size=32,
    callbacks=[early], verbose=2,
)
```

Run the script in the terminal:

```
python3 lecture2_train_weather_model.py
```

WHAT YOU SHOULD GET

Training logs scroll past. Watch validation MAE on each epoch — it drops, levels out, then EarlyStopping kicks in well before 200 epochs and restores the best weights. The whole training run takes under a minute on a CPU. No GPU required. Tthis is what most "real" deep learning looks like — small networks, modest data, fast iteration.

Step 4 — Evaluate Against the Baseline and Save

WHAT TO LEARN

A test MAE in isolation tells you nothing. Three point five degrees Fahrenheit sounds bad until you compare it to the dumbest possible model — predicting that tomorrow will be

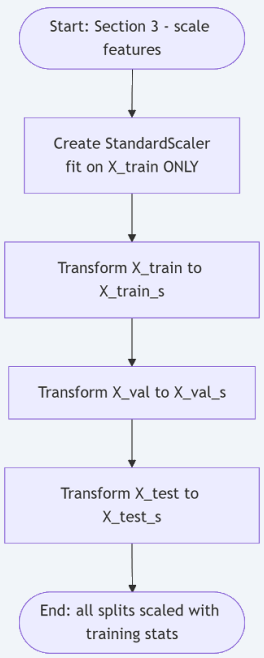
exactly the same as today. That persistence baseline scores around 4.84°F on this data. If your trained model cannot beat it, your model is decorative, not useful.

The evaluation block is the most important part of the script:

```
test_loss, test_mae = model.evaluate(X_test, y_test, verbose=0)
baseline_mae = np.mean(np.abs(test["temp_max_f"].shift(1) - test["temp_max_f"]))
print(f"Test MAE:    {test_mae:.2f} °F")
print(f"Baseline:    {baseline_mae:.2f} °F")
print(f"Improvement: {baseline_mae - test_mae:.2f} °F")
```

Then the model and scaler are persisted for Lecture 3:

```
model.save("weather_model.keras")
joblib.dump(scaler, "weather_scaler.pkl")
with open("feature_columns.json", "w") as f:
    json.dump(FEATURES, f)
```



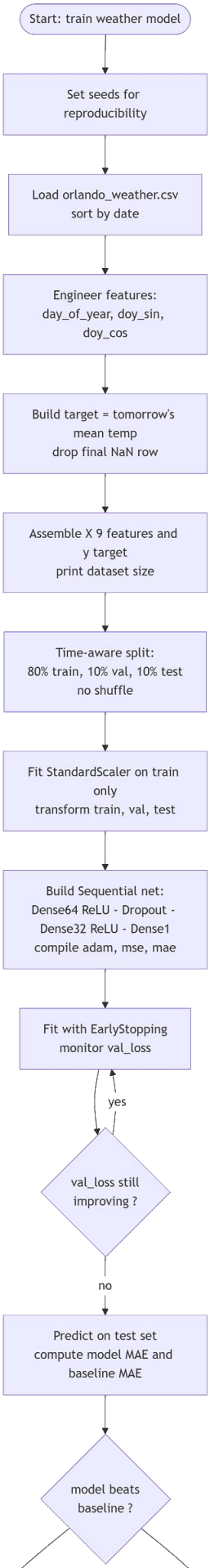
WHAT YOU SHOULD GET

Three files written to disk and three printed lines: a test MAE around 3.50°F, a baseline around 4.84°F, and an improvement of about 1.34°F. That improvement is the entire point of the lecture. Tomorrow you can write a one-line predictor that says "same as today" — or you can spend ten minutes training a network that is a degree and a third more accurate, every day, forever.

Wrap-Up — The Pattern, The Assignment Handoff

REMEMBER

The pipeline is data load, feature engineer, chronological split, scale on train only, build small, train with early stopping, evaluate against a baseline, save. Eight steps. They are the same eight steps for almost every supervised tabular problem you will ever face. The architecture changes, the data changes, the loss function changes — but the discipline stays the same. Internalize this template and you will write trustworthy models for the rest of your career.


Start: train weather model
Set seeds for reproducibility
Load orlando_weather.csv sort by date
Engineer features: day_of_year, doy_sin, doy_cos
Build target = tomorrow's mean temp drop final NaN row
Assemble X 9 features and y target print dataset size
Time-aware split: 80% train, 10% val, 10% test no shuffle
Fit StandardScaler on train only transform train, val, test
Build Sequential net: Dense64 ReLU - Dropout - Dense32 ReLU - Dense1 compile adam, mse, mae
Fit with EarlyStopping monitor val_loss
yes
val_loss still improving ?
no
Predict on test set compute model MAE and baseline MAE
model beats baseline ?

Full Program Flowchart

Key takeaways:

- Feature engineering matters more than model size. Two cyclical features encode a year of seasonal structure that no number of hidden units would discover on its own.
- Chronological splits prevent leakage. Shuffled splits on time-series data produce optimistic, misleading results.
- Fit the scaler on training data only. Information from the test set should never touch the training pipeline at any stage.
- Always compare to a dumb baseline. Persistence forecasting is free and surprisingly hard to beat — but a trained model that beats it is the model worth deploying.
- Early stopping is your friend. It prevents overfitting and saves wall-clock time.

Source File Reference

Script: code/lecture2_train_weather_model.py. Data: data/orlando_weather.csv (1,826 daily observations, five years). Outputs: weather_model.keras, weather_scaler.pkl, feature_columns.json. These three artifacts are the input to Lecture 3.