Module 5: The Deep Learning & Hugging Face Ecosystem, Lesson 3: Using a Trained Model

Hands-On: From Prediction to Decision — A Weather Advisor in Three Layers

Overview

A trained model is not a product. It is a function that takes numbers and returns numbers. To turn it into something useful you need three layers of code wrapped around it: an inference layer that loads the artifacts and shapes the input, a decision layer that turns the prediction into a recommendation a person can act on, and a confidence layer that flags outputs you do not trust.

- You will load the Keras model, the scaler, and the feature manifest that were saved in Lecture 2.
- You will write a predict_tomorrow_temp() function that does the boring-but-critical work: build the feature vector, scale it correctly, run the forward pass, return a number with units.
- You will write an advise() function that converts a number into three concrete decisions: heat-risk level, watering recommendation, activity suggestion.
- You will add an out-of-distribution check that refuses to make recommendations when the input is too far from anything the model has seen during training.

Before You Start

- Have the three artifacts from Lecture 2 sitting in the same directory: weather_model.keras, weather_scaler.pkl, feature_columns.json. If you skipped Lecture 2 in your prep, run python3 lecture2_train_weather_model.py first.
- Install runtime dependencies: pip install tensorflow scikit-learn joblib numpy.
- Open lecture3_weather_advisor.py from https://github.com/AI-Native-Engineer-Course/AI-Native-Engineer/tree/main/module 5/lecture 3
- Open a terminal in the same directory.
- Have today’s real Orlando weather pulled up on a phone or browser tab — at the end of the demo you will type in actual current values and let the model recommend an action.

> **REMEMBER**
> In production, ML models live behind interfaces, not in notebooks. Every model that has ever survived contact with real users is wrapped in code that validates inputs, enforces units, and refuses to answer when the input looks wrong. This hands-on builds that wrapper from scratch so you see exactly where each responsibility lives.

During Hands-On — Four Steps at a Glance

- STEP 1 — Load the model, the scaler, and the feature manifest at startup, not per call. ~2 min
- STEP 2 — Build the inference function with correct feature ordering and unit handling. ~3 min
- STEP 3 — Build the decision layer — heat, watering, activity. ~4 min
- STEP 4 — Add the out-of-distribution confidence flag and run live with today's weather. ~3 min

Step 1 — Load Once, Predict Many Times

WHAT TO LEARN

Loading a Keras model takes about a second. Loading a scaler takes milliseconds. If you load them on every prediction, you have built a slow application. Load once at process startup, hold the references in module-level variables, and let every request reuse them. This pattern is identical to opening a database connection pool — same principle, same payoff.

Open the script. The top of the file looks like this:

```
import json, joblib, numpy as np
from tensorflow import keras

MODEL    = keras.models.load_model("weather_model.keras")
SCALER   = joblib.load("weather_scaler.pkl")
with open("feature_columns.json") as f:
    FEATURES = json.load(f)
```

Flowchart for first section of code

WHAT YOU SHOULD GET

Three module-level constants. MODEL is a Keras model object. SCALER is a fitted StandardScaler. FEATURES is the list of column names, in the exact order the model was trained on. That order matters — passing features in the wrong order silently produces wrong predictions with no error message. The feature_columns.json file is what protects you from that.

Step 2 — The Inference Function

WHAT TO LEARN

Every inference function has the same skeleton: accept human inputs in human units, assemble them into the vector the model expects in the exact order it expects, scale them with the same scaler that was fit during training, run the forward pass, and return a result with units attached. Anything else — JSON I/O, HTTP handling, logging — wraps around this core. Get this function right and the rest of your application is plumbing.

Walk through predict_tomorrow_temp() in the script. Notice the cyclical-feature engineering happens here too, because it has to match what the model saw during training:

```
def predict_tomorrow_temp(observation, date):
    doy = date.timetuple().tm_yday
    row = {
        **observation,
        "doy_sin": np.sin(2 * np.pi * doy / 365.0),
        "doy_cos": np.cos(2 * np.pi * doy / 365.0),
    }
    x = np.array([[row[f] for f in FEATURES]])
    x_scaled = SCALER.transform(x)
    return float(MODEL.predict(x_scaled, verbose=0)[0][0])
```

Flowchar for section 2.

WHAT YOU SHOULD GET

A single float — the predicted maximum temperature for tomorrow, in degrees Fahrenheit. Test it with today's values. The number should land within a few degrees of the actual forecast you can pull up online. That is the test MAE from Lecture 2 made real.

Step 3 — The Decision Layer

WHAT TO LEARN

Users do not want a temperature. They want to know whether to water the lawn, whether to take the kids to the park, whether to hydrate aggressively. The decision layer is where you encode the domain rules that turn a number into an action. These rules are NOT

learned by the model. They are written by you, the engineer, and they are the most important code in the application because they are what the user actually sees.

Read the advise() function with the room. It maps the predicted temperature into three named recommendations:

```
def advise(temp_f, recent_precip_in):
    # Heat risk
    if   temp_f >= 95: heat = "EXTREME HEAT — avoid outdoor exertion 11am–4pm"
    elif temp_f >= 90: heat = "HIGH HEAT — hydrate, schedule outdoor work for early morning"
    elif temp_f >= 85: heat = "MODERATE HEAT — typical Florida day, hydrate as usual"
    elif temp_f >= 70: heat = "MILD — great day for outdoor activities"
    else:              heat = "COOL — light jacket for the morning"

    # Watering decision
    if recent_precip_in >= 0.5:  water = "Skip watering — soil is wet enough"
    elif temp_f >= 90:           water = "Water in the early morning to reduce evaporation"
    else:                        water = "Standard watering schedule is fine"

    return {"heat": heat, "watering": water}
```

Flowchart for section 3 code.

WHAT YOU SHOULD GET

A dictionary of human-readable recommendations. Notice every branch returns a sentence, not a number. The application boundary is the moment numbers become language. If you find yourself returning floats to a UI, you have stopped too early.

Step 4 — The Out-of-Distribution Guardrail

WHAT TO LEARN

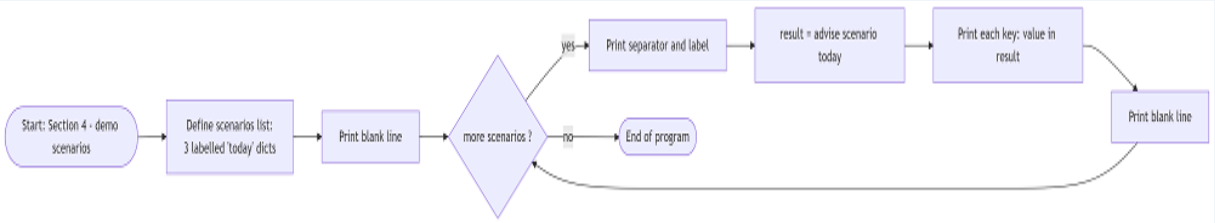
The model was trained on Orlando weather. If you feed it the inputs from a winter day in Minneapolis, it will still produce a number — confidently, and wrongly. The model has no way to say "I do not know." Your wrapper has to say it for the model. The standard pattern:

> compute the z-score of the scaled input. If any feature is more than three standard deviations from the training mean, flag the prediction as low-confidence and refuse to issue a recommendation.

Find the confidence check in the script:

```
def is_in_distribution(x_scaled, threshold=3.0):
    return bool(np.all(np.abs(x_scaled) < threshold))

x_scaled = SCALER.transform(x)
if not is_in_distribution(x_scaled):
    return {"confidence": "low", "advice": "Inputs are outside the model's
training range — refusing to recommend"}
```

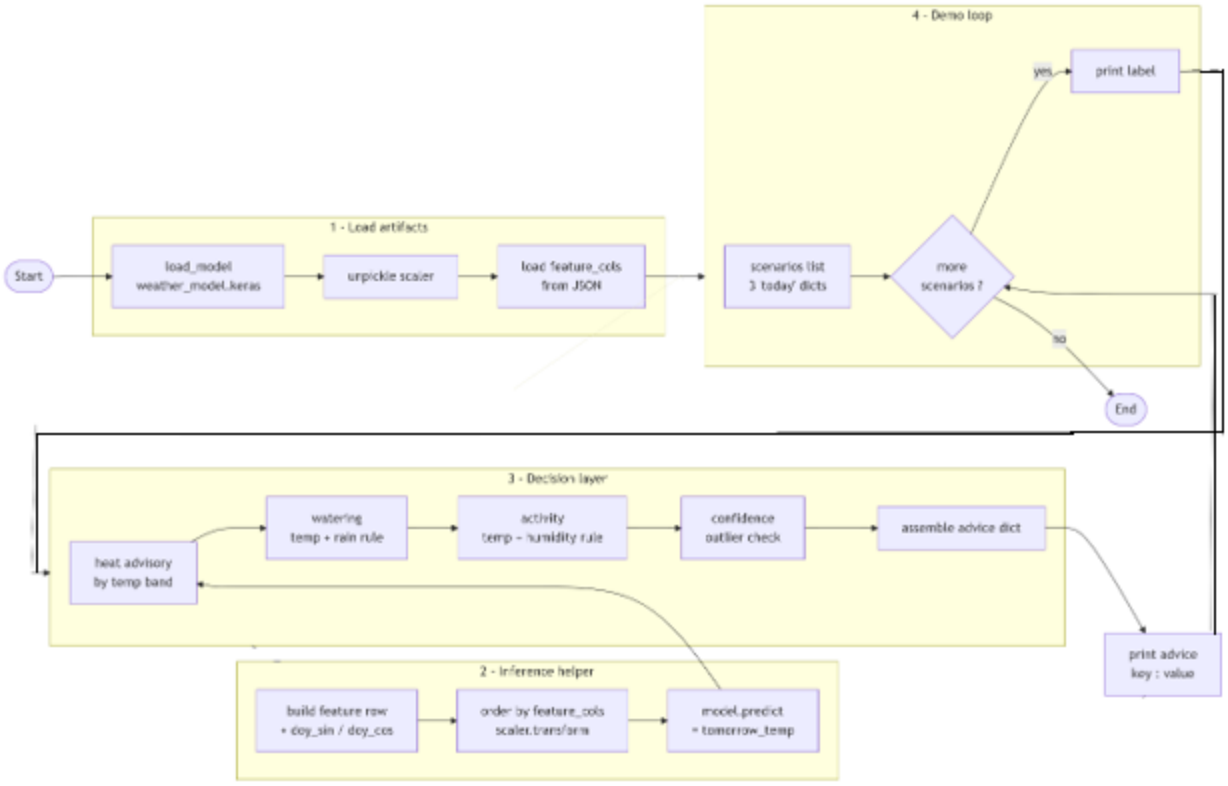


Flowchart for part 4.

Run the script with today's real values:

```
python3 lecture3_weather_advisor.py
```

> **WHAT YOU SHOULD GET**
>
> A predicted temperature, a heat-risk recommendation, and a watering recommendation — all derived from one forward pass and a handful of if-statements. Then run it again with deliberately extreme inputs (humidity = 200, wind = 80) and watch the confidence guard refuse to issue a recommendation. That refusal is the difference between a model that ships and a model that gets pulled after the first user complaint.



Entire program flowchart.

Wrap-Up — The Pattern, The Assignment Handoff

REMEMBER

Every production ML feature looks like this: load once, infer with the right feature order and the right units, decide using rules you wrote and can audit, refuse when the input is out of distribution. The model is one line in that pipeline. The other three lines are what makes the model trustworthy. Build them every time. There are no exceptions.

Key takeaways:

- Load model and scaler once at startup; reuse them on every request.
- Persist the feature column order with the model — never trust dict iteration order to match training.
- The decision layer is where domain knowledge lives. The model does not know what "watering" means; you do.
- Always add an out-of-distribution check. A confident wrong answer is worse than a refusal.

- Return language to the UI, not numbers. The application boundary is where floats become sentences.

Source File Reference

Script: code/lecture3_weather_advisor.py. Required input artifacts in the same directory: weather_model.keras, weather_scaler.pkl, feature_columns.json (all produced by Lecture 2). The script is self-contained, about 90 lines, and runs in under three seconds end to end.