**🧮 NumPy (math engine)**

Main things we’ll use:

* np.random.randn(...) → generate random numbers for test matrices, initial values.
* np.array([...]) → build arrays.
* arr.shape, arr.dtype → check array dimensions and types.
* a @ b → matrix multiplication (core of NNs).
* np.mean(...), np.std(...) → compute averages/variances (useful for sanity checks).

**📊 Pandas (data handling like Excel-in-Python)**

Main things we’ll use:

* pd.DataFrame → table structure for our weather data.
* df['col'] → access a column.
* df.shift(1) → create lag features (y(t–1)).
* df.rolling(6).mean() → rolling averages (6-hour mean temp).
* df.dropna() → remove rows with missing values.
* df.loc[:'2024-04-30'] → slice data by date.

**🎓 Scikit-learn (sklearn) (data prep + metrics)**

Main things we’ll use:

* from sklearn.preprocessing import StandardScaler
  + sc.fit(X\_train) → compute mean/std.
  + sc.transform(X) → scale features to mean 0, std 1.
* from sklearn.metrics import mean\_absolute\_error, mean\_squared\_error
  + mae = mean\_absolute\_error(y\_true, y\_pred)
  + rmse = sqrt(mean\_squared\_error(...))

**🧠 TensorFlow / Keras (training engine)**

Main things we’ll use:

* keras.Input(shape=(n\_features,)) → define input layer.
* layers.Dense(32, activation='relu') → dense (fully connected) hidden layer.
* layers.Dense(1) → output layer for regression.
* model.compile(optimizer='adam', loss='mae') → set up training.
* model.fit(X\_train, y\_train, epochs=20, validation\_data=(X\_val,y\_val)) → train the net.
* model.predict(X\_test) → make predictions.

**🔢 QKeras (quantized NN layers)**

Main things we’ll use:

* from qkeras import QDense, QActivation → quantized versions of Keras layers.
* QDense(32, kernel\_quantizer=quantized\_bits(8,0)) → force weights to 8-bit.
* QActivation(quantized\_relu(8)) → quantized activation function.

**⚙️ hls4ml (translate NN → FPGA code)**

Main things we’ll use:

* hls4ml.utils.config\_from\_keras\_model(model) → auto-generate config template.
* edit config to set precision (ap\_fixed<16,6>) and I/O type (io\_stream).
* hls4ml.converters.convert\_from\_keras\_model(model, hls\_config=..., output\_dir=...) → make the HLS project.
* hls\_model.compile() → simulate NN in Python, compare vs Keras.
* hls\_model.build(csim=True, synth=True) → run C simulation and synthesis in Vivado/Vitis HLS.

✅ So:

* **NumPy** → arrays + math.
* **Pandas** → tabular weather data + feature engineering.
* **Scikit-learn** → scaling + error metrics.
* **TensorFlow/Keras** → train neural net.
* **QKeras** → train quantized NN.
* **hls4ml** → export NN to FPGA code.

Features:  
 **lags** → past returns, past vol, past option greeks

 **rolling stats** → moving averages, rolling volatility, RSI, implied vs historical vol

 **cyclical encodings** → day of week (trading days have rhythm: Monday ≠ Friday), time of day (open vs close behavior)

 **exogenous features** → market indices (SPY, VIX), rates, macro data

 **target** → what you want to forecast (option price in 1 hour? implied vol change tomorrow? up/down movement?)

# convert Nas into 0

for c in ['prcp','snow','tsun','wpgt']:

    if c in df.columns:

        df[c] = df[c].fillna(0.0)         # “no event” default

for c in ['dwpt','rhum','pres','wspd']:

    if c in df.columns:

        df[c] = df[c].interpolate('time').ffill().bfill()  # smooth + fill ends

# wind direction is circular; simplest practical fill:

if 'wdir' in df.columns:

    df['wdir'] = df['wdir'].interpolate('time').ffill().bfill()

Keras is a big toolbox, but you only need a few pieces at a time. Here’s a clean “what’s in the box” map for **tf.keras** (the Keras that ships with TensorFlow).

**Model APIs (how you build nets)**

* **Sequential**: stack layers linearly.
* **Functional**: wire DAGs (multi-input/output, skip connections).
* **Subclassing**: write class MyModel(keras.Model) for full control.

**Layers (the building blocks)**

* **Core**: Dense, Activation, Dropout, BatchNormalization, LayerNormalization, Flatten, Reshape, Concatenate, Add, Multiply.
* **Convolution/Pooling**: Conv1D/2D/3D, SeparableConv2D, DepthwiseConv2D, MaxPooling\*, AveragePooling\*, Global\*Pooling\*.
* **Recurrent**: SimpleRNN, LSTM, GRU (+ Bidirectional, TimeDistributed).
* **Attention**: Attention, AdditiveAttention, MultiHeadAttention.
* **Embeddings & NLP prep**: Embedding, TextVectorization, CategoryEncoding, StringLookup.
* **Image prep**: Rescaling, Resizing, RandomFlip/Rotation/Zoom/Contrast.
* **Masking**: Masking (handle variable-length sequences).
* **Custom**: subclass keras.layers.Layer to make your own.

**Losses (what you optimize)**

* Regression: MeanSquaredError, MeanAbsoluteError, Huber.
* Classification: BinaryCrossentropy, CategoricalCrossentropy, SparseCategoricalCrossentropy.
* Advanced: KLDivergence, CosineSimilarity, custom callables.

**Metrics (how you judge)**

* Regression: MAE, MSE, RootMeanSquaredError.
* Classification: Accuracy, Precision, Recall, AUC, F1Score (via keras.metrics.F1Score in newer TF or implement from Precision/Recall).
* Custom metrics: any function or keras.metrics.Metric subclass.

**Optimizers (how weights update)**

* SGD (+ momentum, nesterov), RMSprop, Adam, AdamW, Adagrad, Adadelta, Nadam, FTRL.
* Learning-rate schedules: ExponentialDecay, CosineDecay, PiecewiseConstantDecay, custom schedule.

**Regularization & initialization**

* **Regularizers**: l1, l2, l1\_l2, activity regularizers.
* **Constraints**: MaxNorm, NonNeg, UnitNorm.
* **Initializers**: Glorot/Xavier, He, Orthogonal, RandomNormal/Uniform, Zeros/Ones.

**Training loop (high-level)**

* model.compile(optimizer=..., loss=..., metrics=[...], jit\_compile=True/False)
* model.fit(X, y, validation\_data=..., epochs, batch\_size, class\_weight, sample\_weight)
* model.evaluate(...), model.predict(...)
* Works with NumPy arrays, tf.data.Dataset, Python generators, or keras.utils.Sequence.

**Callbacks (automation & monitoring)**

* EarlyStopping, ModelCheckpoint, ReduceLROnPlateau, LearningRateScheduler
* TensorBoard, CSVLogger, TerminateOnNaN, BackupAndRestore
* Custom callbacks (subclass keras.callbacks.Callback).

**Data pipelines**

* **tf.data**: Dataset.from\_tensor\_slices, map/batch/prefetch for streaming.
* **Preprocessing layers** in-model (so transforms export with the model).

**Distributed & performance**

* **Mixed precision**: tf.keras.mixed\_precision.set\_global\_policy("mixed\_float16") (on GPUs).
* **Distribution strategies**: MirroredStrategy (multi-GPU), MultiWorkerMirroredStrategy, TPUStrategy.
* tf.function/jit\_compile to XLA-compile graphs.

**Saving & loading**

* Whole model: model.save("path") (SavedModel or H5), keras.models.load\_model(...)
* Weights only: model.save\_weights(...), model.load\_weights(...)
* Config: model.get\_config(), keras.models.model\_from\_json(...)
* Export for mobile/edge: convert to **TFLite** via tf.lite.TFLiteConverter.

**Transfer learning**

* **Applications**: tf.keras.applications (ResNet, MobileNet, EfficientNet, Inception, etc.)
* Freeze base, add new head, fine-tune selected layers.

**Customization (advanced)**

* **Custom layers/models**: subclass Layer/Model and implement call.
* **Custom training loop**: tf.GradientTape() for full control.
* **Custom losses/metrics/regularizers/initializers/constraints**.

**Utilities & quality-of-life**

* model.summary(), keras.utils.plot\_model(model, show\_shapes=True)
* keras.utils.set\_random\_seed(seed) for reproducibility
* Masking & sample/class weights for imbalanced data
* keras.saving.register\_keras\_serializable for custom objects

**Ecosystem (adjacent but separate)**

* **KerasCV** (vision building blocks), **KerasNLP** (tokenizers, Transformers)
* **TensorFlow Model Optimization** (quantization/pruning)
* **TensorFlow Serving** (deploy), **TensorBoard** (viz)

**Minimal patterns you’ll use 95% of the time**

**Build → compile → fit → evaluate → save**

model = keras.Sequential([

layers.Input((n\_features,)),

layers.Dense(64, activation="relu"),

layers.Dense(32, activation="relu"),

layers.Dense(1)

])

model.compile(optimizer="adam", loss="mse", metrics=["mae"])

model.fit(X\_tr, y\_tr, validation\_data=(X\_va, y\_va), epochs=20, batch\_size=128)

print(model.evaluate(X\_te, y\_te))

model.save("model.h5")

**Functional API (skip connections)**

inp = keras.Input((n\_features,))

x = layers.Dense(64, activation="relu")(inp)

y = layers.Dense(64, activation="relu")(x)

out = layers.Dense(1)(layers.Concatenate()([x, y])) # tiny skip concat

model = keras.Model(inp, out)

**Common callbacks**

cbs = [

keras.callbacks.EarlyStopping(patience=5, restore\_best\_weights=True),

keras.callbacks.ModelCheckpoint("best.keras", save\_best\_only=True),

keras.callbacks.ReduceLROnPlateau(patience=2)

]

model.fit(..., callbacks=cbs)