**Advanced AI Forecasting with TensorFlow and Natural Language Processing**

**Week 3 Instructor Notes — Architecture Overview (365D vs 1D)**

**General Note:** These notebooks run end-to-end on Colab Free (which typically provides T4 or GPUs). If you have access to A100 or L4 via Colab Pro or Pay As You Go, they’ll also run and usually train faster. While the A100/L4 are subject to availability on the free tier, students and educators are eligible for a Colab Pro subscription for free. Visit <https://colab.research.google.com/signup> to sign up.

**What changes between the notebooks**

* **365D model:** Conv1D(64) front-end → **4×** Bidirectional(GRU(250)) (stacked) → SimpleAttention → Dense(1).
* **1D model:** **No** Conv1D → **2×** Bidirectional(GRU(250)) → SimpleAttention → Dense(1).
* **Lookback window** (sequence length **T**, e.g., 365 or 1) comes from user input and sets the time dimension of the model inputs.

**Layer-by-layer: what/why/units**

**Input shape**

* (batch, T, features) — for these notebooks, features are columns from the price history (e.g., Close, Volume, etc.).

**Conv1D(64)** *(365D only)*

* **What:** 1-D convolution over time; learns short-range temporal filters.
* **Why here:** With long windows (e.g., 365 days), conv filters capture local motifs (spikes, mini-trends) and denoise before recurrent modeling.
* **Units meaning:** 64 = number of learned temporal filters (output channels). Each filter scans across time to produce a feature map.

**Bidirectional(GRU(250)) (stacked)**

* **What:** Gated Recurrent Unit (GRU) processes the sequence; bidirectional reads it forward and backward to capture patterns that depend on both past and near-future context within the window.
* **Why:** are lighter than LSTMs yet effective for financial time series.
  + **365D:** deeper stack (4 layers) to model richer, multi-scale dynamics across a long year-long window.
  + **1D:** shallower stack (2 layers) is sufficient because there’s minimal temporal depth to exploit.
* **Units meaning:** 250 = hidden size **per direction**. A BiGRU outputs 2×250 = 500 features at each time step (when return\_sequences=True).

**SimpleAttention(units=k)**

* **What:** Learnable, additive attention that scores each time step, produces weights (softmax over time), and forms a **weighted sum** of GRU outputs.
* **Why:** Instead of averaging or taking the last time step, attention lets the model emphasize the most informative days in the window.
* **Units meaning -** **“k” in SimpleAttention:**
  + **What it is:** The size of the hidden layer inside the attention scorer. Think of it as how many knobs the attention has to combine features at each timestep.
  + **What it changes:** Higher **k** ⇒ more expressive scoring (can focus more sharply on specific timesteps) but more parameters and overfitting risk; lower **k** ⇒ simpler, faster, less prone to overfit.
  + **Rule of thumb:** With GRU outputs ~500 features per timestep, start with **k = 64–128**. If attention looks too uniform, try bumping **k**; if validation worsens, lower **k** or add regularization.
  + **Parameter impact:** Grows roughly linearly with **k**. Doubling **k** roughly doubles the attention block’s parameters.
  + **1-day lookback (T=1):** The attention weight is trivially 1, so **k** won’t change the weighting behavior—only the parameter count.
  + **Relation to Bahdanau:** Same idea as Bahdanau’s “alignment size,” adapted for an encoder-only setup (no decoder query).

**Dense(1)**

* **What:** Final linear layer.
* **Why:** Maps the attention-derived context vector to a single regression target (e.g., next-day price).
* **Units meaning:** 1 = one scalar prediction.

**Why the 1-day lookback omits Conv1D**

* A Conv1D needs a **temporal neighborhood** (kernel size ≥ 2) to extract patterns.
* With **T = 1**, there’s **no** local structure to convolve; any convolution would collapse to a trivial linear transform. Skipping Conv1D avoids useless computation and parameters.

**How the SimpleAttention relates to Bahdanau attention**

* The SimpleAttention uses additive scoring: score\_t = vᵀ tanh(W h\_t) → softmax over time → weighted sum of h\_t.
* Classic Bahdanau attention computes e\_t = vᵀ tanh(W\_h h\_t + W\_s s\_{t-1}), where s\_{t-1} is the decoderstate (query).
* Because this model is encoder-only (no decoder/query), the layer adapts Bahdanau attention by dropping the decoder query term and scoring each encoder hidden state self-referentially. Functionally, it’s additive self-attention pooling over time tailored to a single-output regression head.

**Shapes at a glance (typical)**

* After BiGRU stack (with return\_sequences=True): (batch, T, 500).
* After SimpleAttention: (batch, 500) (time collapsed via weighted sum).
* After Dense(1): (batch, 1) prediction.

**Quick mental model to open the session**

* Goal: Forecast next-day close from ~19 features (OHLCV + engineered: moving averages, Bollinger bands, returns, lags, ranks).
* Pipeline: load → feature engineer → scale per feature → build sliding windows (lookback T) → train/val/test split → BiGRU(+attention) model → train with callbacks → predict → inverse scale → plot/save.
* Attention: additive, encoder-only; learns weights over timesteps so the model emphasizes the most informative days.
* 365D model: adds Conv1D + deeper BiGRUs to capture local motifs and long-range patterns in long sequences.
* 1D model: skips Conv1D (no temporal neighborhood at T=1) and uses a lighter BiGRU stack.

**Variable & shape cheat‑sheet (applies to both)**

* **features (list)**: 19 columns: 'Close', 'Volume', 'Open', 'High', 'Low', 'MA20', 'MA50', 'MA200', 'STD20', 'STD50', 'Return1', 'Return5', 'Return20', 'Bollinger\_Upper', 'Bollinger\_Lower', 'Range', 'Close\_Open', 'Lag1', 'Rank20'.
  + **Why feature engineering:** enriches signal and reduces noise by exposing trend (MAs), volatility (STDs/Bollinger), momentum (returns), relative position (ranks), and short memory (lags), improving learning stability and sample efficiency over raw OHLCV alone.
* **scaled\_data**: shape (T, 19) after per-column MinMax scaling.
  + **Why**: puts features on comparable ranges, stabilizes training, speeds convergence.
* **lookback**: integer window length (days) per sample.
  + **Why**: controls temporal context the model sees; larger = more history, smaller = more reactive.
* **X, y**: **X** shape (N, lookback, 19), **y** shape (N,), with N = T − lookback.
  + **What**: X is overlapping sliding windows from scaled\_data; y is the target Close at the prediction horizon (e.g., next day).
* **Splits**: 70% train / 20% val / 10% test in chronological order.
  + **Why**: time-aware split avoids lookahead leakage and mirrors real forecasting.
* **Model input**: Input(shape=(lookback, 19)).
  + **What**: Keras expects sequences of length lookback with 19 features; batch dimension is implicit at runtime.

**Tip**: Remind learners that the code assumes 'Close' is the *first* feature when building **y** and when inverse transforming predictions. That’s why the scaler dictionary stores scalers['Close'] separately for de‑normalization.

**Cell‑by‑cell walkthrough (code cells only)**

**Install & pin dependencies**

**What it does**

* Upgrades pip; installs pinned versions: numpy==2.0.2, tensorflow==2.18.0, scikit-learn==1.6.1, pandas==2.2.2, matplotlib, seaborn, tqdm, plus ipywidgets.
* Adds a shared baseline of extras: transformers==4.53.1, tokenizers, newsapi-python==0.2.7, requests, beautifulsoup4.

**Why it matters**

* Pinning locks APIs/behavior (critical for TF/NumPy) and makes runs reproducible across cohorts and Colab runtimes.
* Extra packages aren’t used in these two notebooks, but keeping a consistent Week 3 environment avoids version drift when students hop between labs.

**Instructor cues**

* After installing TensorFlow (or tokenizers), Colab may require a runtime restart. Remind learners: *Runtime → Restart runtime*, then re-run imports.
* The next cell re-imports everything; tell learners that errors like “module not found” usually mean they skipped the restart.
* ipywidgets enables nicer progress bars/controls; quiet flags (--quiet) reduce noise so students see only meaningful logs.

**Practical tips / quick fixes**

* If GPU is enabled in Colab, TF will pick it up automatically; mismatched CUDA/cuDNN warnings typically clear after a restart.
* If transformers/tokenizers throw build or ABI errors, restart first; if it persists, Factory reset runtime and re-run from top.

**Note on extra dependencies:** There may be install warnings or version-conflict messages for packages like transformers, tokenizers, or newsapi-python (and occasional CUDA/cuDNN notices). These libraries aren’t used in this notebook, so such messages don’t affect execution or results—you can safely ignore them. Only TensorFlow, NumPy, pandas, scikit-learn, matplotlib, seaborn, and tqdm are required for this lab.

**Imports & Colab widget support**

**What it does**

* Enables output.enable\_custom\_widget\_manager() for ipywidgets.
* Imports Python stdlib (os, random, datetime), **NumPy**, **pandas**, **matplotlib**, **sklearn** (MinMaxScaler, r2\_score), TensorFlow/Keras core + layers (Input, Layer, Dense, Conv1D, GRU, Bidirectional, LSTM) and callbacks (EarlyStopping, ReduceLROnPlateau).
* Imports ipywidgets and pickle (for potential serialization).

**Why it matters**

* Establishes all primitives used later: scaling, GRUs, attention, callbacks.

**Instructor cues**

* LSTM is imported but not used. It’s included to support student experimentation (e.g., swapping GRUs for LSTMs) and to avoid import errors if learners modify the model.

**GPU & TF version check**

**What it does**

* Prints tf.\_\_version\_\_ and tf.config.list\_physical\_devices('GPU')**.**
* **Why it matters**
* Confirms hardware acceleration. If the GPU list is empty, set expectations (training will be slower).

**Instructor cues**

* Briefly explain that determinism on GPU can still vary slightly despite fixed seeds.

**Mount Google Drive**

**What it does**

* drive.mount('/content/drive', force\_remount=True).

**Why it matters**

* Centralizes dataset path and places outputs (plots, CSVs, weights) into Drive for persistence across sessions.

**Gotchas**

* Learners must authorize Drive access. Re‑mounts will prompt again.

**Directories & user input**

**What it does**

* Defines base folders: '/content/drive/MyDrive/Nvidia\_Stock\_Market\_History' and a Training subfolder.
* **Prompts for**:
  + lookback (default 20; for the 365D notebook set this to 365, for the 1D notebook set to 1).
  + base\_name for naming the output folder (default Nvidia\_Stock\_Training).
  + graph\_base\_name to prefix plot files.

**Why it matters**

* The lookback directly controls sequence length and therefore memory/time cost and model capacity needs.

**Instructor cues**

* Tie the 1D vs 365D lesson goal to this input. Long lookbacks ≠ always better; they just capture different signal regimes.

**Reproducibility settings**

**What it does**

* Sets a single SEED=42 across Python, NumPy, and TensorFlow and sets TF\_DETERMINISTIC\_OPS=1.

**Why it matters**

* Makes runs more reproducible. Not 100% deterministic on every GPU op, but stable enough for instruction.

**Instructor cues**

* Makes runs more reproducible by ensuring that random number generation (e.g., for weight initialization, data shuffling, and augmentations) produces the same sequence of values each time the notebook is run. This allows students to see consistent results when following along. Not 100% deterministic on every GPU op, but stable enough for instruction.

**Load & feature engineer**

**What it does**

* Reads CSV, parses Date, sorts chronologically, resets index.
* Engineers 19 features:
  + MAs (MA20, MA50, MA200), rolling stds (STD20, STD50), short/medium/long returns (1, 5, 20 days), Bollinger bands, Range, Close\_Open, Lag1, Rank20` (rolling percentile rank).
* Drops NaNs produced by rolling computations.

**Why it matters**

* Feature engineering extracts meaningful patterns and relationships from raw OHLCV data, creating a richer, more informative, and more stationary-like feature space that helps the network learn faster, generalize better, and improve predictive accuracy.

**Instructor cues**

* Clarify that we predict next‑step Close (aligned later) rather than multi‑step horizons.

**Per‑feature MinMax scaling**

**What it does**

* Builds a scalers: Creates a Dict[str, MinMaxScaler] mapping each feature to its own fitted scaler.
* Scales each feature column individually to the [0, 1] range.
* Uses np.hstack to horizontally stack the scaled columns into a (T, 19) NumPy array, where each row is a time step and each column is a scaled feature, ready for model input.

**Why it matters**

* Per‑feature scaling preserves units for clean inverse transforms of predictions using scalers['Close'] .

**Gotchas**

* Data leakage: we scale on the full dataset here for simplicity. In a production setup, fit scalers on train only and transform val/test with those.

**Build sliding windows (sequences)**

**What it does**

* Loops from i = lookback … len(scaled\_data)-1:
  + Appends scaled\_data[i-lookback:i] to **X**.
  + Appends the current Close (column 0) to **y**.
* Converts to arrays → X.shape = (N, lookback, 19), y.shape = (N,).

**Why it matters**

* Casts the problem as sequence‑to‑one regression with fixed window length.

**Instructor cues**

* Emphasize that column order matters; 'Close' must be first in features, as it current is.

**Train/val/test split (time‑aware)**

**What it does**

Splits the dataset sequentially by index: 70% train, 20% validation, 10% test.

**Why it matters**

* Maintains **temporal causality** by preserving the time order so the model only sees “past” data when training, preventing leakage of future information.

**Gotchas**

* In real-world finance tasks, splitting by explicit date ranges is safer; ratio-based splits are fine here for demonstration.

**Type hint line (no‑op)**

**What it does**

* X\_train: np.ndarray explicitly declares that the variable X\_train is expected to hold a NumPy array, providing a type annotation for readability, documentation, and static analysis tools without altering the variable’s behavior at runtime.

**Why it matters**

* Makes code more self-explanatory, helps catch type mismatches early with linters or IDEs, and improves readability for learners and collaborators.

**Custom SimpleAttention layer**

**What it does**

* Implements a custom attention layer in Keras that learns which time steps in the sequence are most important for the task.  
  • **Step-by-step**:  
    1. **Score calculation:**
  + score = tanh(W1(hidden\_states)) — applies a small feedforward network to each time step, producing intermediate attention scores of shape (batch, time, units).

2. **Attention weights:**

* + attention\_weights = softmax(V(score), axis=1) — projects scores down to a single value per time step (batch, time, 1) and normalizes across all time steps so weights sum to 1.

3. **Weighted sum (context vector):**

* context = sum(attention\_weights \* hidden\_states, axis=1) — combines time steps into a single vector (batch, hidden\_dim), giving higher influence to important time steps.  
  • Returns this learned weighted summary of the sequence for downstream layers.

**Why it matters**

* Unlike pooling (which averages or picks the last time step), attention dynamically learns where to look in the sequence, adapting to patterns where relevance is unevenly distributed over time.
* Enables the model to capture long-term dependencies without “forgetting” earlier important steps.

**Instructor cues**

* Explain that softmax makes the weights act like probabilities over time steps.
* Use an analogy: attention is like skimming a book and focusing on the key sentences instead of reading every word equally carefully.
* Clarify that this is *not* transformer self-attention. It’s a simpler additive attention mechanism designed for RNN/LSTM outputs.

**Build the model (the only real difference)**

**Build the model — 365D window (Conv1D + 4×BiGRU + Attention)**

**What it does**

* **Input:** Input(shape=(lookback, 19))— sequence length × features**.**
* Local pattern extractor: Conv1D(64, kernel\_size=3, activation="relu") to capture short-term motifs (e.g., weekly/monthly).
* Temporal modeling:
  + One Bidirectional(GRU(550, return\_sequences=True, dropout=0.2)) layers to learn multi-scale dependencies across a long window.
    - dropout=0.2 in each GRU.
      * Dropout is a regularization technique used in neural networks to reduce overfitting by randomly “dropping out” (i.e., setting to zero) a fraction of the neurons’ outputs during training.
      * In this case, dropout=0.2 in each GRU means that at every training step, 20% of the output units within each Gated Recurrent Unit are randomly ignored.
      * This prevents the GRU from relying too heavily on specific neurons, encouraging it to learn more robust and generalized representations of the sequential data.
  + A second Bidirectional(GRU(350, return\_sequences=True, dropout=0.2, kernel\_regularizer=regularizers.l2(1e-5))
    - The second BIGRU layer not only has dropout=0.2 but it has kernel\_regularizer=regularizers.l2(1e-5))
      * This allows you to add weight regularization to layers (e.g., Dense, Conv1D, GRU) to help reduce overfitting.
* Sequence summarization: SimpleAttention(128) learns weights over time steps and produces a single context vector.
* Output: Dense(1) for scalar regression (next-step Close).
* Compile: optimizer="adam", loss="mse", metrics ["mae","mse"].
* Summary: model.summary() prints layer shapes/params.

Why it matters

* Long lookbacks (e.g., 365) benefit from Conv1D for local patterns and deeper GRUs for long-range structure; attention highlights the most relevant moments.

Instructor cues

* Emphasize return\_sequences=True so attention can see every time step.
* Show param growth with model.summary() and connect it to capacity vs. overfitting.

**Build the model — 1D window (2×BiGRU + Attention)**

**What it does**

* **Input:** Input(shape=(lookback, 19)).
* **Temporal modeling:** Two stacked Bidirectional(GRU(250, return\_sequences=True, dropout=0.2)) layers (no Conv1D).
* **Sequence summarization:** SimpleAttention(128) reduces the (very short) sequence to a context vector.
* **Output:** Dense(1) for scalar regression.
* **Compile:** optimizer="adam", loss="mse", metrics ["mae","mse"].
* **Summary:** model.summary( ).

**Why it matters**

* With lookback = 1, the time dimension is trivial; shallower stacks avoid unnecessary parameters and reduce overfitting risk while training faster.

**Instructor cues**

* Contrast parameter counts vs. the 365D model to illustrate architectural right-sizing.
* Note that both models rely on attention, but the **benefit is greater** when the sequence has meaningful length.
* Explain that Conv1D cannot be used meaningfully with a 1-day lookback because convolution requires a sliding window across multiple time steps; with only one time step, there’s no temporal neighborhood to convolve over.

**Build a model config string for filenames**

**What it does**

* Iterates model.layers and encodes a short descriptor per layer, e.g. C1D64, BiGRU250, BAtt, D1.
* Appends \_Lookback{lookback} → yields something like C1D64\_BiGRU250\_BiGRU250\_BiGRU250\_BiGRU250\_BAtt\_D1\_Lookback365 (365D) or BiGRU250\_BiGRU250\_BAtt\_D1\_Lookback1 (1D).

**Why it matters**

* Produces **human‑readable run IDs** for saving plots, weights, and predictions.

**Train with callbacks**

**What it does**

* Sets EarlyStopping on val\_loss with patience=8, restore\_best\_weights=True.
  + Monitors val\_loss, stops training if no improvement for 8 epochs, and restores the best weights to avoid keeping overfit parameters.
* Sets ReduceLROnPlateau on val\_loss with factor=0.5, patience=3.
  + Monitors val\_loss, halves the learning rate if it stalls for 3 epochs to encourage finer convergence after plateaus.
* model.fit(..., epochs=200, batch\_size=32, validation\_data=(X\_val, y\_val), callbacks=[...])
* Trains for up to 200 epochs with batch\_size=32, validating after each epoch, and using the callbacks to adaptively control training length and learning rate.

**Why it matters**

* Early stopping guards against overfitting and wasted epochs
* Learning rate reduction helps the optimizer escape flat spots and refine weights near the minimum.

**Instructor cues**

* For the 365D architecture, expect slower epochs and earlier overfitting.
* Encourage students to watch val\_loss curves closely.
* Show how ReduceLROnPlateau messages appear in the log when the LR changes.
* Highlight that both callbacks make training adaptive, rather than rigidly fixed for 200 epochs.

**Predictions, inverse transform, plots, R²**

**What it does**

* Builds all\_X as overlapping sliding windows from the fully scaled dataset, matching the lookback length used in training.
* **Predict**: Feeds all\_X to the trained model, producing scaled predictions.
* **Inverse transform**: Uses scalers['Close'] to map predictions back to the original price scale.
* **Align**: Matches predictions\_inverse with the corresponding dates and actual closing prices from df.
* **Evaluate**: Computes r2\_score to measure how well predictions match the actual prices.
* **Visualize**: Plots Actual vs Predicted prices and the training/validation loss curves.

**Why it matters**

* **Shows the full inference pipeline**: normalized inputs → model prediction → inverse scaling → evaluation → visualization.
* Provides both quantitative (R²) and qualitative (plots) insights into performance.

**Gotchas**

* R² can be misleading for non-stationary financial series. It measures fit, not profitability or predictive usefulness.
* For deeper evaluation, consider MAE/RMSE and error distribution plots in future iterations.
* **Ensure lookback alignment**: misalignment between actual and predicted series will distort metrics and visuals.

**Optional saving prompts**

**What it does**

* Asks whether to save CSV of predictions and plots to the Drive subfolder named from the model config string.
* Optionally saves model weights via model.save\_weights('...weights.h5').

**Why it matters**

* Encourages reproducible experiments: consistent filenames that encode architecture + lookback.

**Instructor cues**

* If automating, replace input() prompts with fixed flags and programmatic paths.

**365D vs 1D: how to frame the differences in class**

* **Signal horizon**: 1‑day windows focus **on ultra‑short‑term noise**; 365‑day windows allow the model to consider seasonality/long trends.
* **Capacity & inductive bias**:
  + Conv1D in 365D captures local motifs quickly; the GRU stack then models longer context.
  + Fewer layers in 1D reduce unnecessary capacity when there’s practically no sequence to model.
* **Compute**: 365D sequences are ~365× larger along the time axis → higher memory & runtime; hence a stronger but costlier model.

**Talking points & common questions**

* **Why GRU over LSTM?** Similar capability for these horizons with fewer parameters; faster to train.
* **Why MinMaxScaler?** Keeps outputs in [0,1] for stable RNN training; easy to invert for 'Close'.
* **Why attention?** Lets the model learn which days matter inside the window instead of treating all days equally.
* **Is scaling on full data OK?** For pedagogy, yes; in practice, fit on train, apply to val/test.
* **Could we add exogenous data (news, macro)?** Yes. Extend features and retrain; the pipeline stays the same.

**Optional demo ideas**

* **Flip lookback** mid‑class (set to 1 vs 365) and discuss model.summary() differences.
* **Turn off attention** (remove the layer) and compare validation curves.
* **Swap GRU↔LSTM** for a quick architecture comparison.

**Appendix: What each saved artifact means**

* **\*\_prediction.png**: Overlay of actual vs predicted close.
* **\*\_loss.png**: Training vs validation loss per epoch.
* **\*.weights.h5**: Keras weights file for the trained architecture.
* **predictions.csv**: Date‑aligned table of actual and predicted closing prices.

**Final reminder**

Aside from the model build cell (Conv1D + deeper GRUs in 365D; shallower BiGRUs in 1D), the notebooks are the same. The lookback value typed at runtime controls the input shape and affects training speed, generalization, and how useful attention becomes.