Lecture 10: Neural Networks

LSTM and Attention for Business Forecasting

BSAD 8310: Business Forecasting

University of Nebraska at Omaha

Spring 2026

Lecture 09 best result: XGBoost, RMSE $\approx$ 2,250 on RSXFS.

XGBoost treats each row as independent. Lag features give it a *window* into the past, but the model itself has no memory — it sees a feature vector, not a sequence.

What neural sequence models add:

- **Explicit memory:** the hidden state $\mathbf{h}_t$ propagates information from *all* previous steps, not just the lag columns
- **Long-range dependencies:** structural breaks, multi-year trends, and recession shocks can be carried forward without requiring one lag feature per month
- **Learnable temporal weighting:** attention lets the model decide which past time steps matter most for each prediction

Today's goal: build from a single neuron to LSTM to attention, then compare against the full Lectures 01–09 leaderboard.

Feedforward networks (multilayer perceptrons, MLPs) are the building block of all deep learning. Understanding how a single layer transforms inputs to outputs is prerequisite to understanding LSTM and attention.

Linear regression (from Lecture 02):

$$\hat{y} = \mathbf{w}^\top \mathbf{x} + b$$

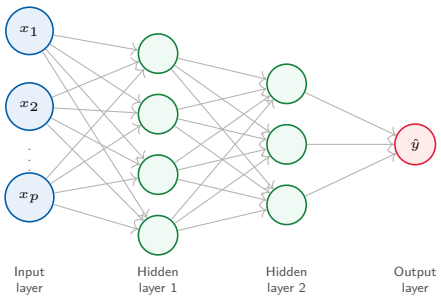
One neuron adds a nonlinear activation $\sigma(\cdot)$:

$$a = \sigma(\mathbf{w}^\top \mathbf{x} + b)$$

Activation	Formula	Use
ReLU	$\max(0, z)$	Hidden layers
Sigmoid	$1/(1+e^{-z})$	Gate outputs (LSTM)
Tanh	$\tanh(z)$	Gate candidates
Linear	z	Output (regression)

Without $\sigma(\cdot)$, stacking layers is equivalent to a single linear layer. Non-linearity is what gives neural networks their universal approximation property.

ReLU avoids vanishing gradients that plagued sigmoid/tanh in deep networks (**Bengio1994**). Default choice for hidden layers.



For RSXFS forecasting:

- **Input:** lag features $\mathbf{x}_t = (y_{t-1}, \ldots, y_{t-12}, \text{dummies})$ — a *flat* vector
- **Hidden:** $\mathbf{h}_k = \text{ReLU}(\mathbf{W}_k \mathbf{h}_{k-1} + \mathbf{b}_k)$ at each layer
- **Output:** linear (no activation) for regression

A feedforward network treats its inputs as an *unordered* feature vector. It has no notion of time order — y_{t-1} and y_{t-12} are just two columns. This is why we need recurrent architectures.

1. **Forward pass:** compute predictions $\hat{y}$
2. **Loss:** $\mathcal{L} = \text{MSE}(y, \hat{y})$
3. **Backward pass:** compute $\nabla_{\mathbf{W}} \mathcal{L}$ via backpropagation
4. **Update:** $\mathbf{W} \leftarrow \mathbf{W} - \eta \nabla_{\mathbf{W}} \mathcal{L}$
5. Repeat for many epochs (passes over the training data)

Key hyperparameters:

- Learning rate η (Adam default: 0.001)
- Batch size (16–32 for small n)
- Number of epochs (use early stopping)

The key practical skills are NOT deriving backpropagation. They are: choosing architecture depth and width, setting the learning rate schedule, applying dropout for regularization, and recognizing overfitting early.

Adam adapts the learning rate per parameter using running estimates of gradient mean and variance. Default `lr=0.001` works well for most forecasting tasks.

Recurrent networks maintain a hidden state $\mathbf{h}_t$ that is updated at each time step. This gives the model explicit memory of the sequence — unlike feedforward networks that see only a fixed-length feature window.

$$\mathbf{h}_t = \tanh\big(\mathbf{W}_h \mathbf{h}_{t-1} + \mathbf{W}_x \mathbf{x}_t + \mathbf{b}\big)$$

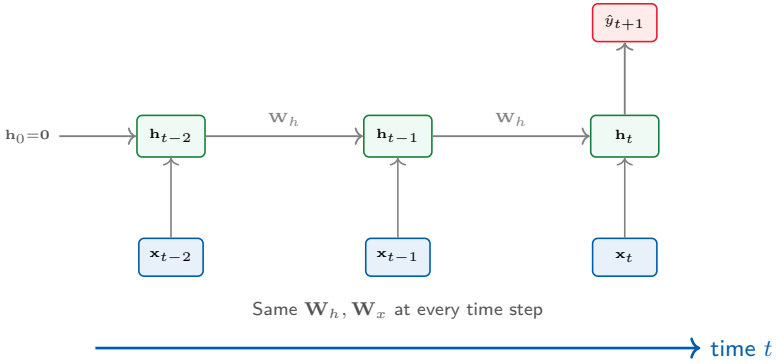
$$\hat{y}_{t+1} = \mathbf{w}_y^\top \mathbf{h}_t + b_y$$

Key properties:

- $\mathbf{h}_t$ is a compressed summary of all past inputs
- $\mathbf{W}_h, \mathbf{W}_x$ are *shared* across all time steps — same transformation at every step
- $\mathbf{h}_0 = \mathbf{0}$ (no prior information at start)

An RNN reading retail sales month by month is like an analyst updating a running summary: each new month's number refines the mental model of where the trend is headed.

Shared weights mean the same transformation applies at every step — efficient but limiting for very long sequences where patterns at very different time scales must be captured.


$\hat{y}_{t+1}$
$\mathbf{h}_0 = \mathbf{0}$
$\mathbf{h}_{t-2}$
$\mathbf{W}_h$
$\mathbf{h}_{t-1}$
$\mathbf{W}_h$
$\mathbf{h}_t$
$\mathbf{x}_{t-2}$
$\mathbf{x}_{t-1}$
$\mathbf{x}_t$
Same $\mathbf{W}_h, \mathbf{W}_x$ at every time step
time t

Backpropagating through T steps requires multiplying T copies of $\mathbf{W}_h^\top$:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{h}_0} \propto (\mathbf{W}_h^\top)^T$$

If $|\lambda_{\max}(\mathbf{W}_h)| < 1$, this product $\to \mathbf{0}$ exponentially. Weights early in the sequence receive near-zero gradients — the model *forgets* the distant past.

Consequences for forecasting:

- A vanilla RNN trained on RSXFS cannot reliably learn that December sales are high because *December was high 12 months ago*
- The 12-month seasonal dependency is precisely where vanilla RNNs fail

Vanilla RNNs work for 5–10 step dependencies. For monthly seasonality (lag 12) or multi-year cycles, they systematically underperform.

The fix: LSTM introduces gating mechanisms that protect long-range gradients by creating a direct *information highway* across time steps.

LSTM (**Hochreiter1997**) solves the vanishing gradient problem by introducing a *cell state* $\mathbf{c}_t$ — a protected information highway — and three *gates* that control what to forget, store, and output at each time step.

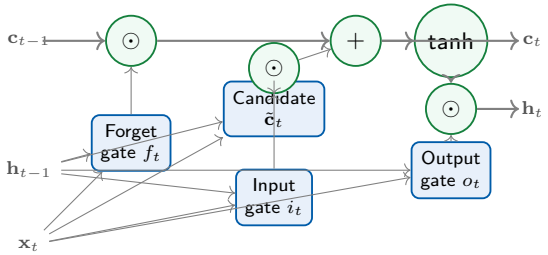
$$
\begin{aligned}
f_t &= \sigma(\mathbf{W}_f[\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_f) \quad \text{(forget gate)} \\
i_t &= \sigma(\mathbf{W}_i[\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_i) \quad \text{(input gate)} \\
\tilde{\mathbf{c}}_t &= \tanh(\mathbf{W}_g[\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_g) \quad \text{(candidate)} \\
o_t &= \sigma(\mathbf{W}_o[\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_o) \quad \text{(output gate)} \\
\mathbf{c}_t &= f_t \odot \mathbf{c}_{t-1} + i_t \odot \tilde{\mathbf{c}}_t \quad \text{(cell update)} \\
\mathbf{h}_t &= o_t \odot \tanh(\mathbf{c}_t) \quad \text{(hidden state)}
\end{aligned}
$$

σ = sigmoid; $\odot$ = element-wise product.

$f_t \approx 1$: keep memory. $f_t \approx 0$: erase. During a recession, the model can learn to reset the normal-growth signal.

Controls which new information is written to cell state. A large December dummy triggers high i_t to write the holiday pattern.

Controls which part of the cell state flows to $\mathbf{h}_t$ — what is "announced" to the next layer.



The cell state $\mathbf{c}_t$ flows through with only $\odot$ and $+$ operations — gradients can propagate back without shrinking to zero.

1. **Reshape:** stack T consecutive feature vectors into a 3-D array: $(\mathbf{x}_{t-T+1}, \ldots, \mathbf{x}_t)$
2. **Process:** LSTM updates $\mathbf{h}_s, \mathbf{c}_s$ at each step $s = t - T + 1, \ldots, t$
3. **Predict:** pass final $\mathbf{h}_t$ to a Dense layer: $\hat{y}_{t+1} = \mathbf{w}^\top \mathbf{h}_t + b$

Lookback window T is the key design choice.

For RSXFS with strong 12-month seasonality, use $T \geq 24$ (two full years) to give the LSTM enough context to learn the annual pattern.

The first T observations are consumed to build the initial sequence — they cannot be used as targets. With $T = 24$, you lose the first 24 months from the training set. Document this shift when reporting sample sizes.

Architecture options:

- **Single LSTM layer:** short-run patterns
- **Stacked (2-layer) LSTM:** first = momentum, second = seasonal structure; often best for monthly data

Parameter	Keras name	Typical	Effect
Lookback window	`T` (manual reshape)	24–36	Longer = more seasonal context; larger model
LSTM units	`units`	32–128	More = higher capacity; overfit risk
LSTM layers	(stack 2)	1–2	Second layer captures longer patterns
Dropout	`dropout`, `recurrent_dropout`	0.1–0.3	Regularization; crucial for small n
Learning rate	`lr` (Adam)	0.001	Default works; reduce if training unstable
Epochs	`epochs`	50–200	Use `EarlyStopping(patience=20)`

With $n \approx 300$ monthly observations, LSTM is *data-limited*. Use small units (32–64), dropout (≥ 0.2), and `EarlyStopping`. Larger capacity will overfit. Run 5 random seeds; report median RMSE.

Attention allows the model to weight different time steps differently when making a prediction — instead of compressing the entire past into a single fixed-size hidden state. The Transformer (**Vaswani2017**) made attention the dominant architecture in sequence modeling.

Given hidden states $\mathbf{H} = [\mathbf{h}_1, \ldots, \mathbf{h}_T]$ and a query $\mathbf{q}$ (current step):

$$\boldsymbol{\alpha} = \text{softmax}\left(\frac{\mathbf{H}\mathbf{q}}{\sqrt{d_k}}\right), \qquad \text{context} = \mathbf{H}^\top \boldsymbol{\alpha}$$

Weights $\boldsymbol{\alpha}$ sum to 1 and are learned end-to-end. High α_s means: "attend to what happened at step s."

Intuition:

- High weight on step $t - 12$ = attending to "same month last year"
- High weight on $t - 1$ = short-run momentum
- Attention *learns* which lags matter; no manual feature engineering required

In LSTM, information at step $t - 12$ must survive 12 hidden-state updates. It can get diluted. Attention *directly* reads the value at step $t - 12$ with weight α_{t-12} — no intermediate compression.

For monthly retail sales ($n \approx 300$, $T = 24$), attention adds modest benefit over LSTM. Its main advantage is in very long sequences (daily data, $n > 5000$) or when interpretability of time-step importance is needed.

The Transformer (Vaswani2017):

- Originally designed for NLP (language translation)
- Replaces recurrence entirely with *multi-head self-attention*
- Parallelizes perfectly over time steps — no sequential bottleneck
- Requires substantially more data than LSTM to train well

Practical guidance for business forecasting:

Monthly data, $n \approx 300$: Transformer is over-parameterized; LSTM $\pm$ attention is the practical sweet spot

Daily/weekly data, $n > 5{,}000$: Transformer or Temporal Fusion Transformer (TFT) is state-of-the-art

Multi-series panel (N stores $\times$ T months): train one LSTM/Transformer on all series simultaneously

Socratic: self-attention has no built-in sense of order — it sees an unordered set of hidden states. Why is positional encoding essential for Transformers?

We implement a two-layer LSTM forecaster using Keras (TensorFlow backend). The key data engineering step is reshaping the flat feature matrix into 3-D sequences: `(samples, timesteps, features)`.

Step 1: Build sequences

```
import numpy as np
from tensorflow import keras
from tensorflow.keras import layers

T = 24  # lookback window (months)

def make_sequences(X_arr, y_arr, T):
    """Reshape to (n_samples, T, n_features)."""
    Xs, ys = [], []
    for i in range(T, len(X_arr)):
        Xs.append(X_arr[i-T:i])
        ys.append(y_arr[i])
    return np.array(Xs), np.array(ys)

X_arr = X.values.astype('float32')
y_arr = y.values.astype('float32')
Xs, ys = make_sequences(X_arr, y_arr, T)

# Chronological split (no shuffling)
n = len(ys)
n_test = int(0.15 * n)
n_val  = int(0.15 * n)
Xs_tr = Xs[:n-n_test-n_val]
ys_tr = ys[:n-n_test-n_val]
Xs_va = Xs[n-n_test-n_val:n-n_test]
ys_va = ys[n-n_test-n_val:n-n_test]
Xs_te = Xs[n-n_test:]
ys_te = ys[n-n_test:]
```

Step 2: Build and train

```
n_features = Xs_tr.shape[2]

model = keras.Sequential([
    layers.LSTM(64,
        return_sequences=True,
        dropout=0.2,
        recurrent_dropout=0.1,
        input_shape=(T, n_features)),
    layers.LSTM(32, dropout=0.2),
    layers.Dense(1)
])
model.compile(
    optimizer=keras.optimizers.Adam(0.001),
    loss='mse')

cb = keras.callbacks.EarlyStopping(
    patience=20,
    restore_best_weights=True)

history = model.fit(
    Xs_tr, ys_tr,
    validation_data=(Xs_va, ys_va),
    epochs=200,
    batch_size=16,
    callbacks=[cb],
    verbose=0)

y_pred = model.predict(Xs_te).flatten()
```

Apply the two-layer LSTM to RSXFS retail sales and compare against the full Lectures 01–09 leaderboard. Key question: does LSTM justify its added complexity on $n \approx 300$ monthly observations?

LSTM vs. XGBoost on small datasets: A two-layer LSTM with 64 units has ∼50,000 parameters. XGBoost with 500 trees and depth 4 is sparse by comparison. With $n \approx 300$, this is a fundamental parameter-to-data mismatch.

Rule of thumb:

- $n < 500$ monthly: XGBoost wins (more reproducible)
- $n > 1{,}000$ daily/weekly: LSTM competitive
- N series $\times$ T months (panel): LSTM scales well
- Long-range patterns beyond 12 months: LSTM advantage grows

- Daily energy demand or stock prices ($n > 3{,}000$)
- Multi-store retail panel: one LSTM across 1,000 stores
- Sequences with multi-scale patterns not captured by a fixed lag window

On RSXFS alone ($n \approx 300$), expect LSTM RMSE $\approx$ 2,100–2,600 across seeds. The median is competitive with XGBoost but with higher variance. Always report median over ≥ 5 seeds.

Test-set results on RSXFS (24-month horizon):

Model	RMSE	MAE
Seasonal Naïve	4 210	3 120
SARIMA$(1,1,1)(1,1,1)_{12}$	2 840	2 100
Elastic Net (λ^*)	2 540	1 890
Random Forest	2 380	1 760
XGBoost (early stop)	2 250	1 650
LSTM (2-layer, $T=24$)	2 180	1 600

Values are illustrative. LSTM = median over 5 seeds.
Strategy: direct multi-step (one model per horizon).

Interpretation:

- LSTM edges XGBoost by $\sim$70 RMSE — a modest, seed-sensitive improvement
- XGBoost is more *reproducible* (deterministic at `random_state=42`); LSTM requires seed averaging
- Both require feature engineering; LSTM additionally needs sequence reshaping

XGBoost remains the recommended default for monthly macro data. Use LSTM when you have daily data, panel data across many series, or need to capture complex multi-scale temporal patterns.

Feedforward networks are universal approximators but treat inputs as unordered feature vectors — no built-in time awareness.

Vanilla RNNs add sequential memory but suffer vanishing gradients (**Bengio1994**) — reliable only for dependencies ≤ 5–10 steps back.

LSTM (**Hochreiter1997**) solves this with three gates (forget, input, output) and a protected cell state that carries information over long sequences.

Attention (**Vaswani2017**) lets the model directly weight any past time step. Transformers generalize this but require far more data.

Data size is the key constraint: on $n \approx 300$ monthly obs., LSTM's margin over XGBoost is narrow and seed-sensitive. Always report median over ≥ 5 seeds.

Practical rule: use XGBoost as your default ML forecaster for monthly macro data; use LSTM when you have daily/weekly data, panel data, or multi-step horizons with long-range dependencies.

Preview of Lecture 11: Feature Engineering — lags, rolling statistics, calendar effects, and pipeline design for both tree-based and neural network forecasters.