PROBABILISTIC TIME SERIES FORECASTING WITH NUMPYRO: A PRACTICAL GUIDE

JUAN CAMILO ORDUZ

Abstract. This article presents a comprehensive guide to probabilistic time series forecasting using NumPyro, a lightweight probabilistic programming library built on JAX. We demonstrate a progression from simple exponential smoothing models to sophisticated hybrid deep state-space models, emphasizing practical implementation patterns and scalability considerations. Through concrete examples, we illustrate key concepts including the `scan` function for recursive relationships, hierarchical model structures for sharing information across time series, custom likelihoods for censored and intermittent demand data, and model calibration techniques using additional likelihood terms. Two showcase applications—availability-constrained TSB models and Hilbert Space Gaussian Process calibration—highlight the flexibility of probabilistic programming for encoding domain knowledge directly into forecasting models. The integration with the broader JAX ecosystem, including Optax for optimization and Flax for neural network components, enables scalable inference via stochastic variational inference (SVI) on thousands of time series.

1. Introduction

Forecasting is a critical component of business planning across industries—from retail inventory management to financial market analysis, from energy demand prediction to marketing budget allocation. Traditionally, businesses have relied on point forecasts that provide a single estimate of future values. While these approaches can work reasonably well under stable conditions, they often fail to capture the inherent uncertainty in real-world systems and can lead to suboptimal decisions when confronted with volatile or complex environments.

Consider a retailer deciding how much inventory to stock. A point forecast of 100 units tells them nothing about whether true demand might be 80 or 120. If they stock exactly 100 units and demand turns out to be 120, they lose 20 sales. If demand is only 80, they're stuck with 20 units of excess inventory. The *cost* of these errors is often asymmetric—lost sales may damage customer relationships permanently, while excess inventory merely ties up capital. Without understanding the full distribution of possible outcomes, the retailer cannot make an informed decision that accounts for these asymmetric risks.

The Paradigm Shift: From Point Estimates to Distributions. Probabilistic forecasting addresses these limitations by generating complete probability distributions over possible future outcomes rather than single-point predictions. This paradigm shift provides decision-makers with a more comprehensive view of potential scenarios, enabling robust planning that accounts for risk and uncertainty.

Date: January 17, 2026.

But probabilistic forecasting offers more than just uncertainty quantification. As we will demonstrate throughout this article, probabilistic programming frameworks enable us to directly incorporate business constraints, domain knowledge, and even causal relationships into our models. When sales data is censored by stockouts, we can model the censoring mechanism explicitly. When we have experimental results from lift tests, we can inject that knowledge as calibration constraints. When different product categories share common seasonality patterns, we can let them borrow statistical strength from each other through hierarchical structures.

This flexibility is what sets probabilistic forecasting apart from traditional methods. Classical approaches treat the model as a black box that maps inputs to outputs. Probabilistic programming inverts this relationship: we specify *how we believe the data was generated*, and the inference machinery works backward to learn the parameters consistent with our observations. This generative perspective makes it natural to encode domain expertise directly into the model structure.

NumPyro and the JAX Ecosystem. NumPyro [14] is a lightweight probabilistic programming library that provides a NumPy backend for Pyro [2]. It relies on JAX [4] for automatic differentiation and JIT compilation to GPU/CPU, enabling efficient inference on large-scale problems. This combination of expressiveness and performance makes NumPyro particularly well-suited for time series forecasting applications where we may need to fit models to thousands of related time series.

The integration with the broader JAX ecosystem—including Optax for gradient-based optimization and Flax for neural network components—enables sophisticated hybrid architectures that combine the interpretability of state-space models with the flexibility of deep learning. Modern inference algorithms like Stochastic Variational Inference (SVI) allow scaling to datasets that would be computationally infeasible with traditional Markov Chain Monte Carlo methods.

Article Overview. This article presents a progression from foundational concepts to advanced techniques, with emphasis on the *motivation* behind each approach and the business problems they solve. We begin with the `scan` function, the computational building block that enables efficient implementation of recursive time series models. We then develop exponential smoothing and ARIMA models, demonstrating the basic patterns for state-space modeling in NumPyro.

The middle sections address specialized challenges that arise in real-world forecasting: hierarchical models for sharing information across related time series, intermittent demand models for products with sporadic sales, and censored demand models for situations where stockouts mask true demand. Two showcase sections demonstrate the unique power of probabilistic programming: an availability-constrained TSB model that encodes business logic directly into the transition function, and a model calibration technique that injects domain expertise through additional likelihood terms.

We conclude with stochastic variational inference for scalable training and hybrid deep state-space models that integrate neural network components for learning complex patterns across many time series.

2. Literature Review and Related Work

Probabilistic forecasting has a rich history spanning classical statistical methods and modern deep learning approaches. This section surveys key developments and positions the NumPyro-based approach within this landscape.

2.1. **Classical Statistical Methods. State space models and ETS**: Hyndman et al. [9, 8] reformulated exponential smoothing as innovations state space models, enabling likelihood estimation and automatic model selection. The ETS (Error-Trend-Seasonality) taxonomy provides a systematic framework for exponential smoothing variants.

ARIMA family: The Box-Jenkins methodology established autoregressive integrated moving average models as a cornerstone of time series analysis. Extensions include seasonal ARIMA (SARIMA) and vector autoregressive (VAR) models for multivariate forecasting.

Structural time series: Harvey [7] and Durbin & Koopman [6] developed comprehensive frameworks for state space modeling with Kalman filtering, providing the theoretical foundation for many modern approaches.

2.2. **Deep Learning Approaches. DeepAR** [17]: A seminal work in probabilistic deep learning for forecasting, DeepAR trains a global autoregressive RNN model across many related time series. The key innovation is learning to share patterns across series while producing probabilistic forecasts through parametric output distributions.

Temporal Fusion Transformers [10]: Attention-based architectures for multi-horizon forecasting with interpretable attention weights and handling of static and dynamic covariates.

N-BEATS and N-HiTS [13]: Pure deep learning approaches without time series-specific components, using basis expansion for interpretability.

2.3. **Probabilistic Programming Frameworks. Pyro Forecasting Module**: Built on Pyro/PyTorch, this module provides hierarchical models with global-local structure and SVI for scalable inference. The Dynamic Linear Model tutorial [16] demonstrates powerful calibration techniques that we extend in this work.

Prophet [21]: Facebook's decomposable model combining trend, seasonality, and holiday effects, designed for business forecasting at scale with Stan backend for uncertainty quantification.

GluonTS [1]: A unified Python interface for probabilistic time series modeling, implementing DeepAR, Transformers, and many other methods.

2.4. **Recent Advances. LGT/SGT Models** [18]: Bayesian exponential smoothing with flexible trend components between linear and exponential, using Student-t errors for robustness.

ADAM [19]: The Augmented Dynamic Adaptive Model provides a comprehensive state space framework with extensive treatment of intermittent demand and censored observations.

2.5. **Gap Addressed by This Work.** Existing tools often lack easy customization of model components, direct integration of domain knowledge, and transparent model specification. NumPyro addresses these gaps by providing:

- Composable model building with `scan` for time series
- Custom likelihoods (censored, zero-inflated, availability-constrained)
- JAX ecosystem integration for GPU acceleration
- SVI for scalable inference on thousands of series
- Neural network integration via Flax NNX

3. Technical Foundation: The Scan Function

Time series models are fundamentally about *sequential dependence*—each observation depends on what came before it. The current sales level depends on yesterday's level. Today's forecast error depends on yesterday's forecast error. The seasonal pattern at time t depends on the seasonal pattern from one period ago. This recursive structure, where the present depends on the past, is what distinguishes time series from cross-sectional data.

In most programming languages, we would express this dependence with a simple `for` loop. But `for` loops are problematic in the context of automatic differentiation and GPU computation. They execute sequentially in Python, preventing parallelization, and frameworks like JAX cannot efficiently compute gradients through them. The `scan` function solves this problem by expressing sequential computation in a form that JAX can compile and differentiate efficiently.

The `scan` function takes a transition function that describes how to move from one time step to the next, an initial state, and a sequence of inputs. It returns the final state and all intermediate outputs—exactly what we need for time series forecasting where we want to track latent states (level, trend, seasonality) and generate predictions at each step.

Listing 1. Pure Python implementation of the scan function

```
def scan(f, init, xs):
    """Pure Python implementation of scan.

    Parameters
    ----------
    f : callable
        A function to be scanned: (carry, x) -> (carry, y)
    init : any
        Initial loop carry value
    xs : array
        Values over which to scan along leading axis
    """
    carry = init
    ys = []
    for x in xs:
        carry, y = f(carry, x)
        ys.append(y)
```

```
18     return carry, np.stack(ys)
```

The transition function `f` takes the current carry state and input, returning an updated carry and output. This pattern naturally maps to state-space model formulations where the carry represents latent states (level, trend, seasonality) and the output represents predictions.

Remark 1. In JAX, `jax.lax.scan` compiles the loop efficiently, avoiding Python overhead and enabling automatic differentiation through the entire sequence. NumPyro's `contrib.control_flow.scan` extends this with proper handling of probabilistic primitives.

4. Exponential Smoothing Models

Consider a retailer whose sales patterns shift gradually over time. Last month's average sales provide useful information about this month's expected level, but not perfect information—consumer preferences evolve, competitors enter the market, and economic conditions change. The retailer faces a fundamental tradeoff: rely too heavily on recent observations and the forecast becomes noisy, overreacting to random fluctuations; rely too heavily on distant history and the forecast becomes sluggish, slow to adapt to genuine changes in the underlying pattern.

Exponential smoothing elegantly resolves this tradeoff through a single parameter α that controls the balance between *memory* and *responsiveness*. When α is close to 1, the model places most weight on the most recent observation, adapting quickly to changes but potentially overreacting to noise. When α is close to 0, the model averages over a long history, providing stability but potentially missing genuine shifts in the pattern.

What makes exponential smoothing particularly appealing as a starting point is its interpretability. Unlike black-box machine learning models, exponential smoothing has a clear structure: it maintains an estimate of the current "level" of the series (and optionally trend and seasonality), updating this estimate as new observations arrive. This transparency makes it easier to diagnose problems and communicate results to stakeholders.

4.1. Simple Exponential Smoothing.

The level equations for simple exponential smoothing capture this intuition mathematically:

$$\begin{aligned} \hat{y}_{t+h|t} &= l_t \\ l_t &= \alpha y_t + (1-\alpha) l_{t-1} \end{aligned}$$

where y_t is the observed value, l_t is the level, and $\alpha \in (0, 1)$ is the smoothing parameter.

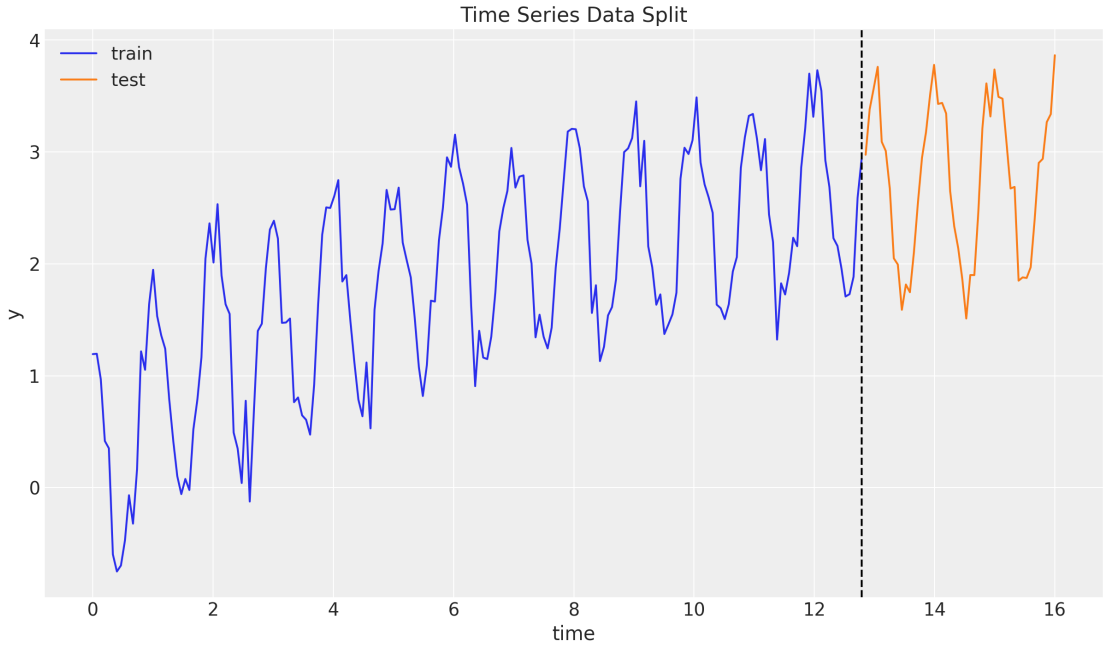

FIGURE 1. Example time series data for exponential smoothing. The data exhibits a clear trend component that simple exponential smoothing must capture through the level dynamics.

The NumPyro implementation uses the `scan` pattern:

LISTING 2. Transition function for simple exponential smoothing

```
def transition_fn(carry, t):
    previous_level = carry

    level = jnp.where(
        t < t_max,
        level_smoothing * y[t] + (1 - level_smoothing) * previous_level,
        previous_level, # Forecast: no update
    )

    mu = previous_level
    pred = numpyro.sample("pred", dist.Normal(loc=mu, scale=noise))

    return level, pred
```

The full model wraps this transition function with appropriate priors. We place a Beta$(1, 1)$ prior on the smoothing parameter α, allowing the data to determine how quickly the model should adapt to new observations. The initial level receives a weakly informative normal prior, and the observation noise receives a half-normal prior ensuring

positivity. NumPyro's `scan` function then iterates the transition function through time, with `handlers.condition` linking the model's predictions to the observed data.

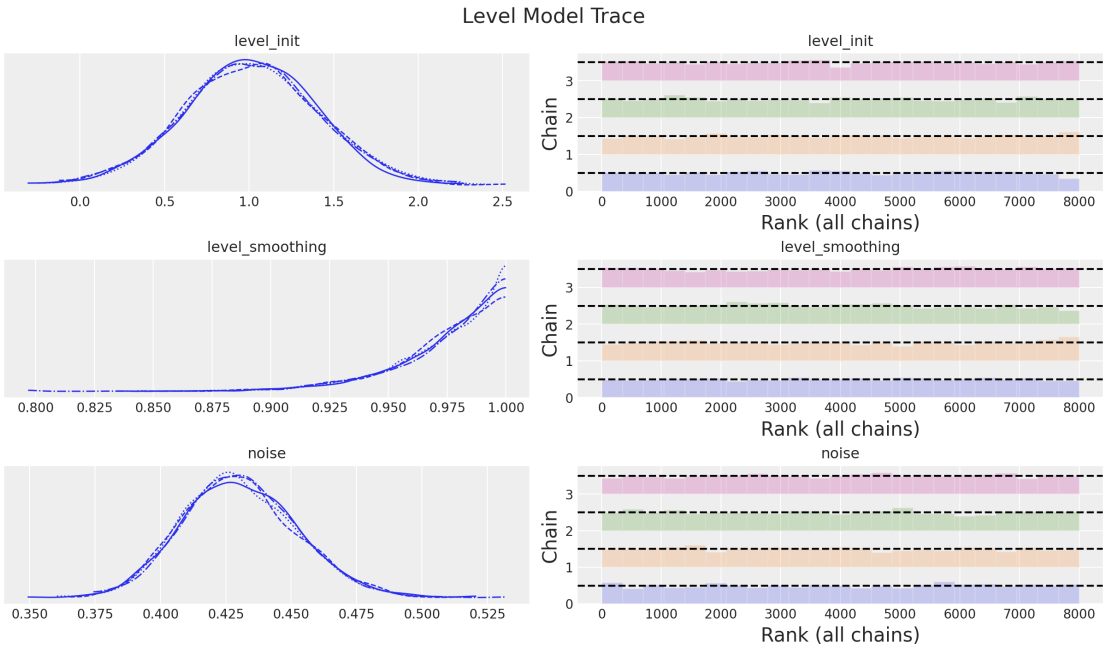

FIGURE 2. Posterior distributions of model parameters. The smoothing parameter α concentrates around 0.8, indicating relatively fast adaptation to recent observations.

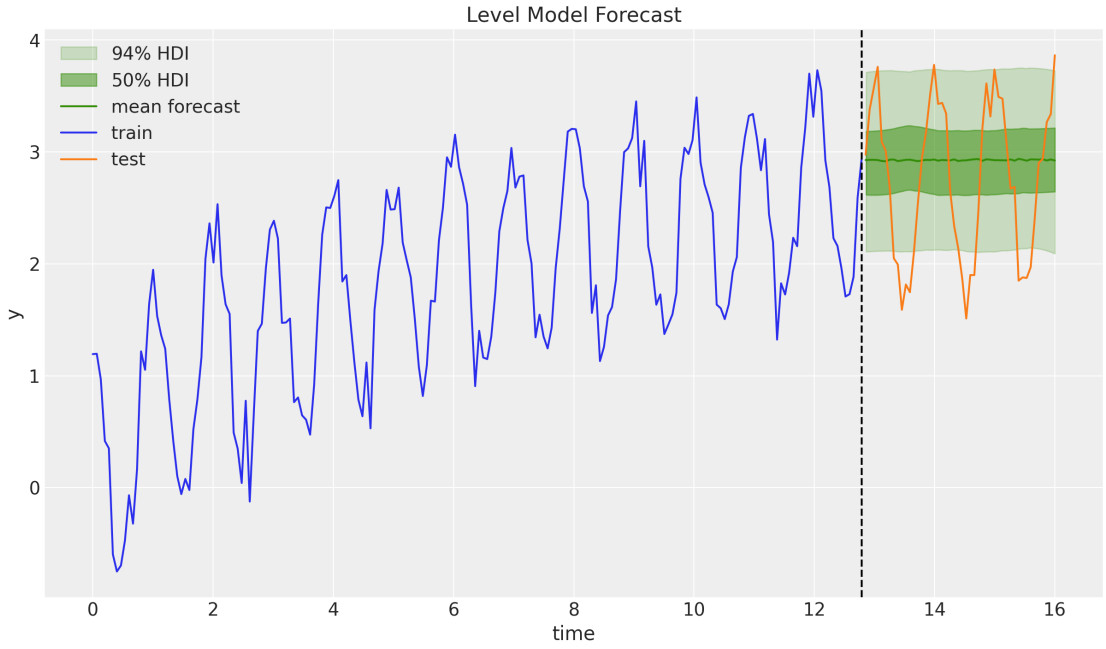

FIGURE 3. In-sample fit and forecast with 94% highest density intervals. The model captures the underlying trend while appropriately quantifying forecast uncertainty.

4.2. **Extensions: Trend, Seasonality, and Damping.** The exponential smoothing framework extends naturally to include trend and seasonal components. The Holt-Winters model with damped trend combines:

$$
\begin{aligned}
\hat{y}_{t+h|t} &= l_t + (\phi + \phi^2 + \cdots + \phi^h) b_t + s_{t+h-m(k+1)} \\
l_t &= \alpha(y_t - s_{t-m}) + (1-\alpha)(l_{t-1} + \phi b_{t-1}) \\
b_t &= \beta(l_t - l_{t-1}) + (1-\beta)\phi b_{t-1} \\
s_t &= \gamma(y_t - l_{t-1} - \phi b_{t-1}) + (1-\gamma) s_{t-m}
\end{aligned}
$$

where b_t is the trend, s_t is the seasonal component, ϕ is the damping factor, and m is the seasonal period.

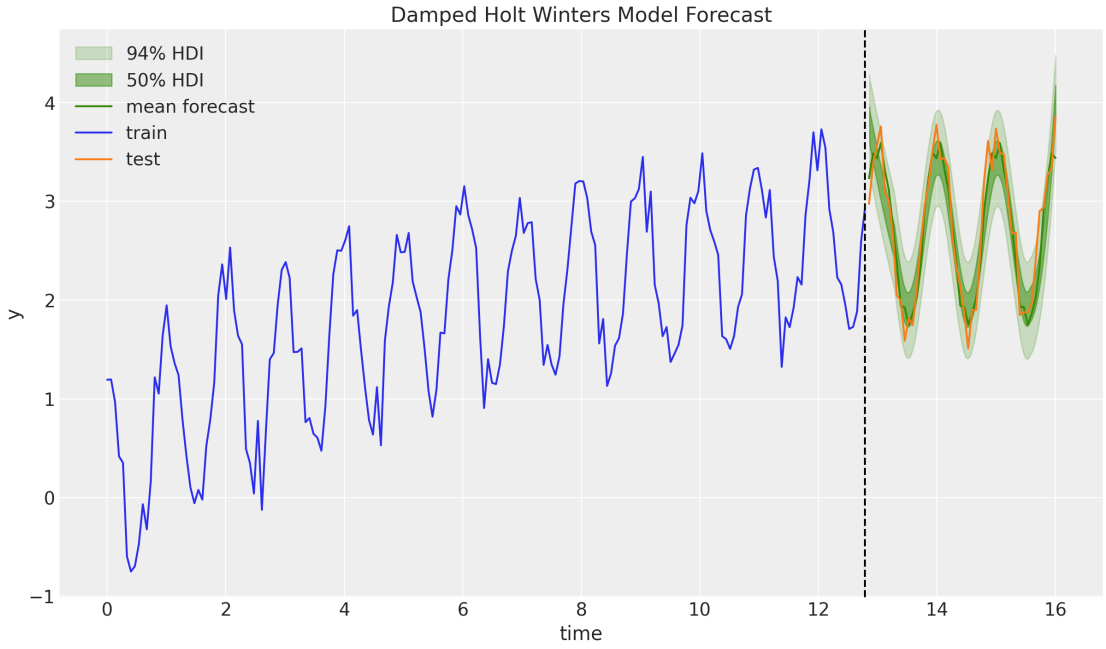

FIGURE 4. Damped trend seasonal model forecast. The model captures both the trend decay and seasonal patterns, with uncertainty bands widening appropriately into the forecast horizon.

5. ARIMA MODELS

While exponential smoothing models the *level* of a series directly, ARIMA models take a different perspective: they model the *autocorrelation structure* of the data. The key insight is that many time series exhibit predictable patterns in how current values relate to past values and past forecast errors.

Consider financial returns, where today's movement often partially reverses yesterday's movement (mean reversion), or manufacturing processes, where a machine running hot yesterday tends to run hot today (persistence). ARIMA models capture these patterns through two mechanisms: the *autoregressive* component models how the current value depends on past values directly, while the *moving average* component models how the current value depends on past forecast errors.

The distinction matters in practice. Autoregressive terms capture persistent deviations from the mean, while moving average terms capture transient shocks that fade quickly. A series dominated by AR terms will show long-memory behavior; a series dominated by MA terms will show short-memory behavior with quick reversion to the mean.

In the NumPyro framework, ARIMA models follow the same `scan` pattern as exponential smoothing, demonstrating the flexibility of this computational approach.

5.1. **ARMA(1,1) Model.**

The ARMA(1,1) model combines one autoregressive and one moving average term:

$$y_t = \mu + \phi y_{t-1} + \theta \varepsilon_{t-1} + \varepsilon_t$$

LISTING 3. Transition function for ARMA(1,1) model

```
def transition_fn(carry, t):
    y_prev, error_prev = carry
    ar_part = phi * y_prev
    ma_part = theta * error_prev
    pred = mu + ar_part + ma_part
    error = y[t] - pred
    return (y[t], error), error
```

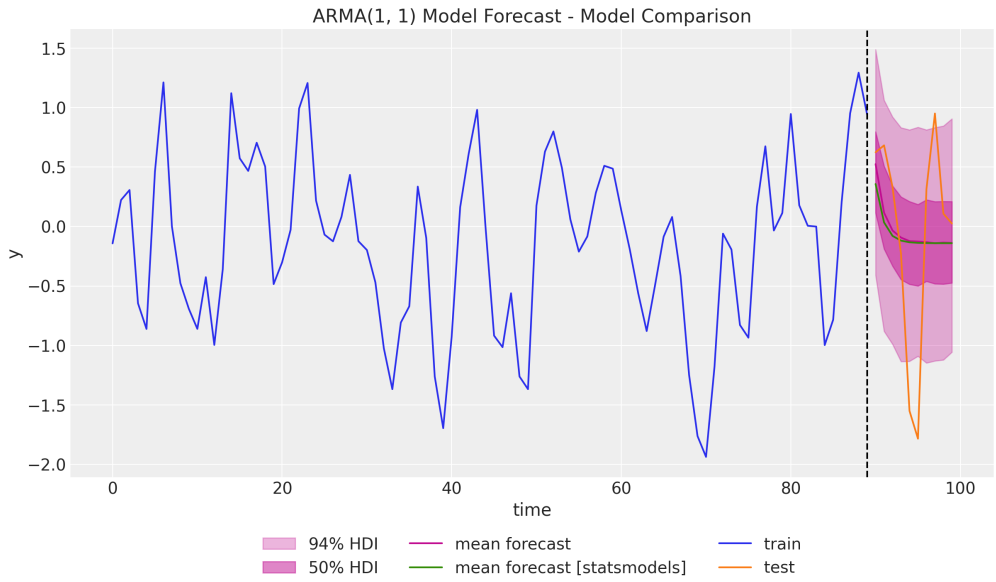

FIGURE 5. ARMA(1,1) model fit showing the in-sample predictions and residuals. The model effectively captures the short-term autocorrelation structure in the data.

5.2. **VAR Models and Impulse Response Functions.** Vector autoregressive (VAR) models extend ARMA to multivariate settings, enabling analysis of dynamic relationships between multiple time series. A key application is computing impulse response functions (IRFs) that trace the effect of a shock to one variable through the system.

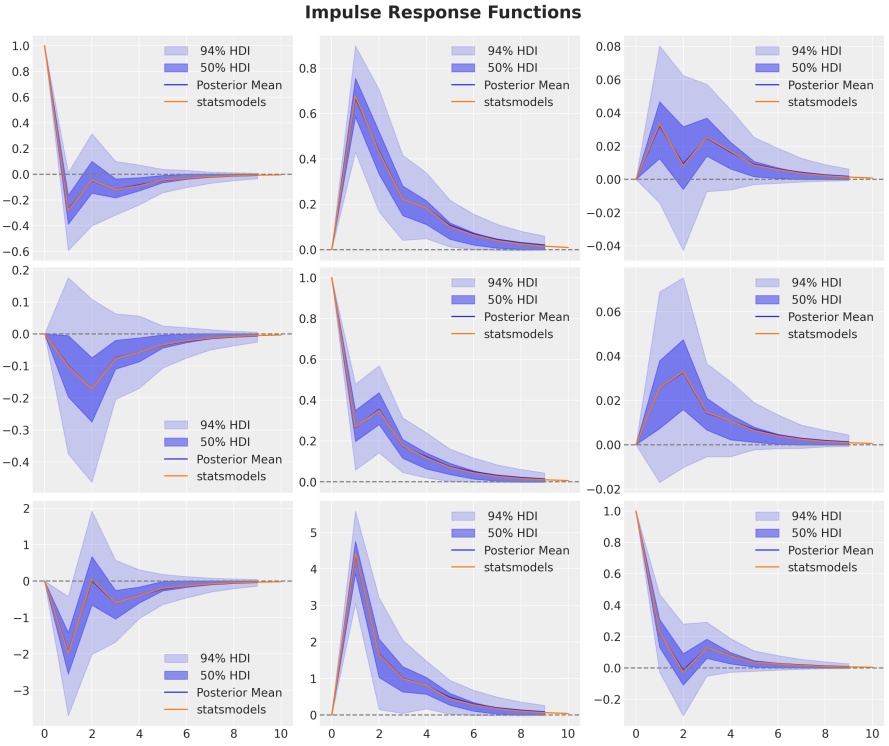

FIGURE 6. Impulse response functions from a Bayesian VAR model. The posterior distributions over IRFs quantify uncertainty in the dynamic relationships between variables.

6. HIERARCHICAL MODELS

Imagine you are forecasting tourism volumes for 308 combinations of Australian states, regions, and travel purposes. Some combinations—say, business travel to Sydney—have abundant historical data. Others—perhaps holiday travel to a small regional town—have only a handful of observations. How should you approach this problem?

One option is *complete pooling*: fit a single model to all data, assuming all regions behave identically. This approach gains statistical power but ignores meaningful regional differences. Another option is *no pooling*: fit a separate model to each region independently. This respects heterogeneity but suffers from data sparsity—how can you reliably estimate seasonality for a region with only 8 quarterly observations?

Hierarchical models offer a third way: *partial pooling*. The key insight is that related time series often share common structure while exhibiting individual variation. All Australian regions might share a similar seasonal pattern (more tourism in summer), but the amplitude and timing may differ. By modeling these shared patterns explicitly, hierarchical models allow data-rich series to "lend strength" to data-poor series, improving estimates across the board.

This is where Bayesian methods truly shine. The hierarchical structure is naturally expressed through the prior distribution: individual parameters are drawn from a population distribution whose parameters are themselves unknown and learned from data. Classical frequentist methods have no natural analog to this information-sharing mechanism.

6.1. **Hierarchical Exponential Smoothing.** We demonstrate this concept using the quarterly Australian tourism dataset, where visitor counts are organized by state, region, and purpose of travel.


NUTS Forecast
Northern Territory::Darwin::Holiday
Queensland::Darling Downs::Other
Queensland::Outback::Business
Northern Territory::Lasseter::Other
Northern Territory::East Coast::Other
Northern Territory::Hobart and the South::Holiday
South Australia::Adelaide Hills::Business
Victoria::Central Highlands::Visiting
Victoria::Central Murray::Other
Victoria::Ballarat::Holiday
trips
train
test
HoltWinters forecast
train-test split
94% HDI
50% HDI
posterior predictive mean

The hierarchical structure places priors on the smoothing parameters that allow information sharing:

$$\begin{aligned} \alpha_i &\sim \text{Beta}(\mu_\alpha \kappa, (1-\mu_\alpha)\kappa) \\ \mu_\alpha &\sim \text{Beta}(2, 2) \\ \kappa &\sim \text{HalfNormal}(10) \end{aligned}$$

where μ_α is the population mean smoothing parameter and κ controls the concentration around this mean.

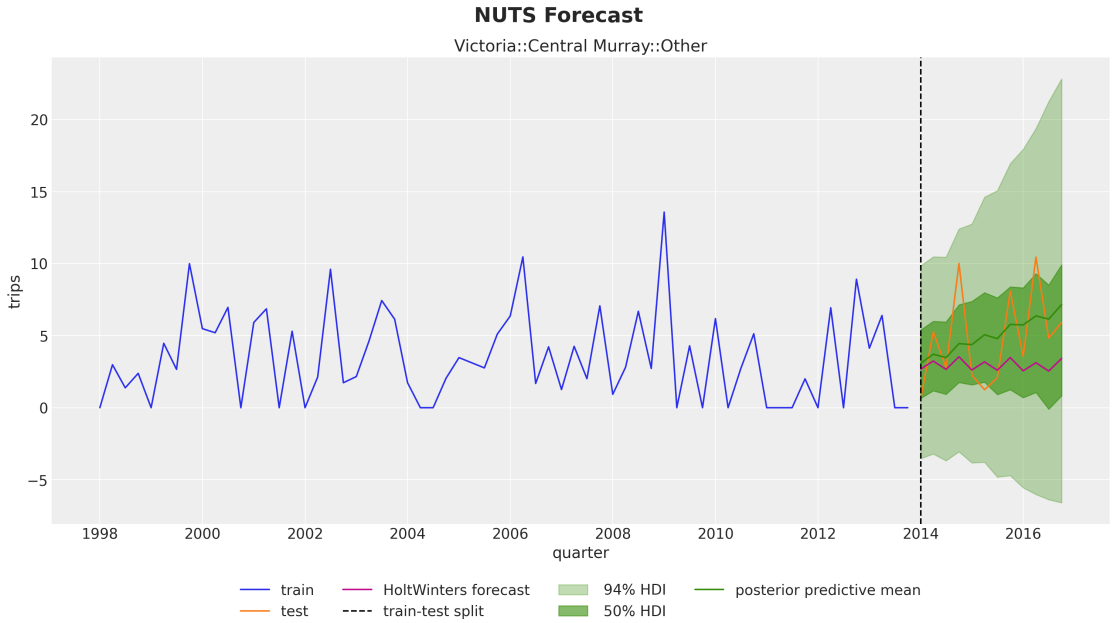

FIGURE 8. Posterior distributions of smoothing parameters across regions. The hierarchical structure shrinks extreme estimates toward the population mean while preserving meaningful heterogeneity.

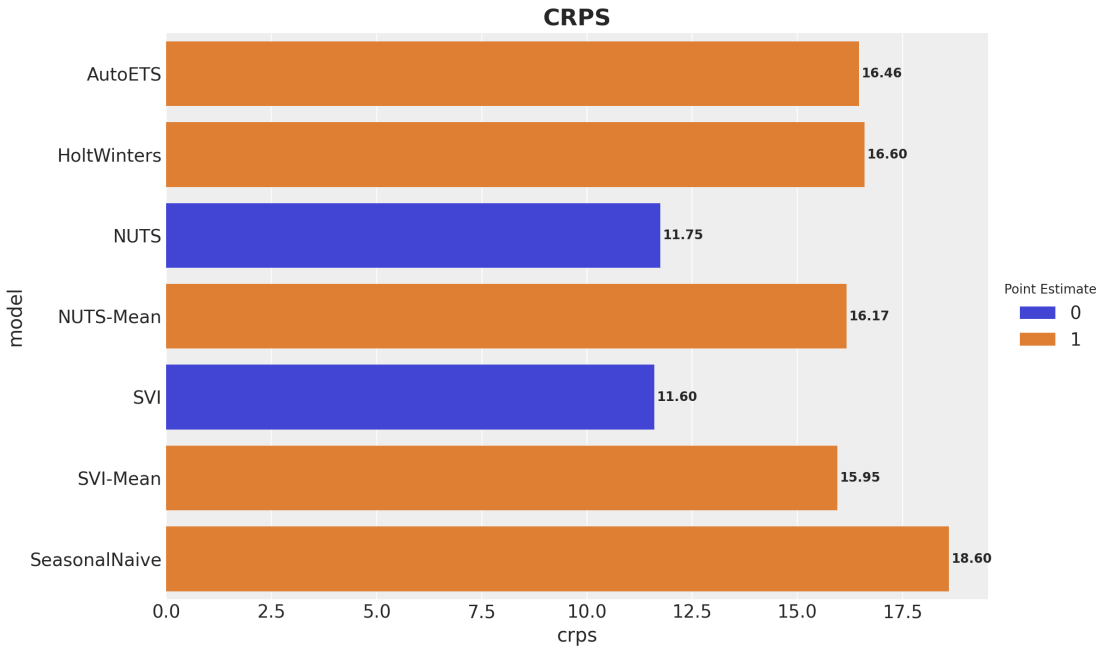

FIGURE 9. Hierarchical forecasts for selected regions with 94% HDIs. The model appropriately quantifies uncertainty, with wider intervals for regions with higher volatility.

6.2. **Baseline Production Model.** For large-scale applications, we developed a baseline model combining local level dynamics with Fourier seasonality and external covariates:

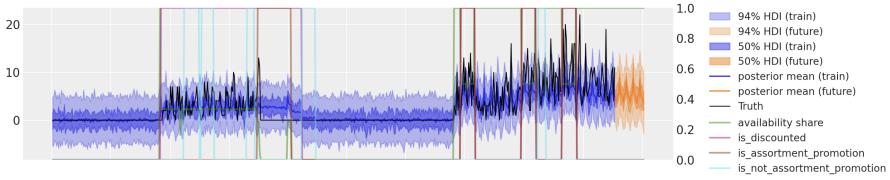

FIGURE 10. Production baseline model architecture: local level for trend, Fourier modes for seasonality, covariates for promotions/discounts, and a global availability factor. This model scales to approximately 40,000 time series in under 10 minutes on GPU.

7. INTERMITTENT DEMAND FORECASTING

In retail, intermittent time series are the norm rather than the exception. While top-selling products might show regular daily sales patterns, the vast majority of items in a typical retail catalog exhibit sporadic demand—many days showing zero sales followed by occasional purchases. This pattern is particularly common for niche products, seasonal items, spare parts, or products with long replacement cycles. A hardware store might

sell a specialized plumbing fitting once every two weeks; an auto parts retailer might see demand for a particular gasket only a few times per month.

Standard forecasting methods, designed for continuous demand streams, struggle with these patterns. Exponential smoothing applied directly to intermittent data will produce forecasts that are pulled toward zero by the many zero observations, systematically underestimating demand when it does occur. The fundamental problem is that these methods conflate two distinct phenomena: *whether* demand will occur and *how much* demand will occur given that it happens.

The insight behind specialized intermittent demand methods is to separate these two questions. Instead of modeling the raw demand series directly, we decompose it into a *demand size* component (how much is demanded when demand occurs) and a *demand occurrence* component (how often demand occurs). Each component can then be forecast using standard techniques, and the final forecast is their product.

7.1. **Croston's Method.** Croston's method [5] pioneered this decomposition approach. The idea is simple: extract the non-zero demand values into a "demand size" series z_t, and extract the intervals between demand occurrences into an "inter-arrival period" series p_t. Apply exponential smoothing to each series separately, then combine the forecasts as $\hat{y}_{t+h} = \hat{z}_{t+h}/\hat{p}_{t+h}$.

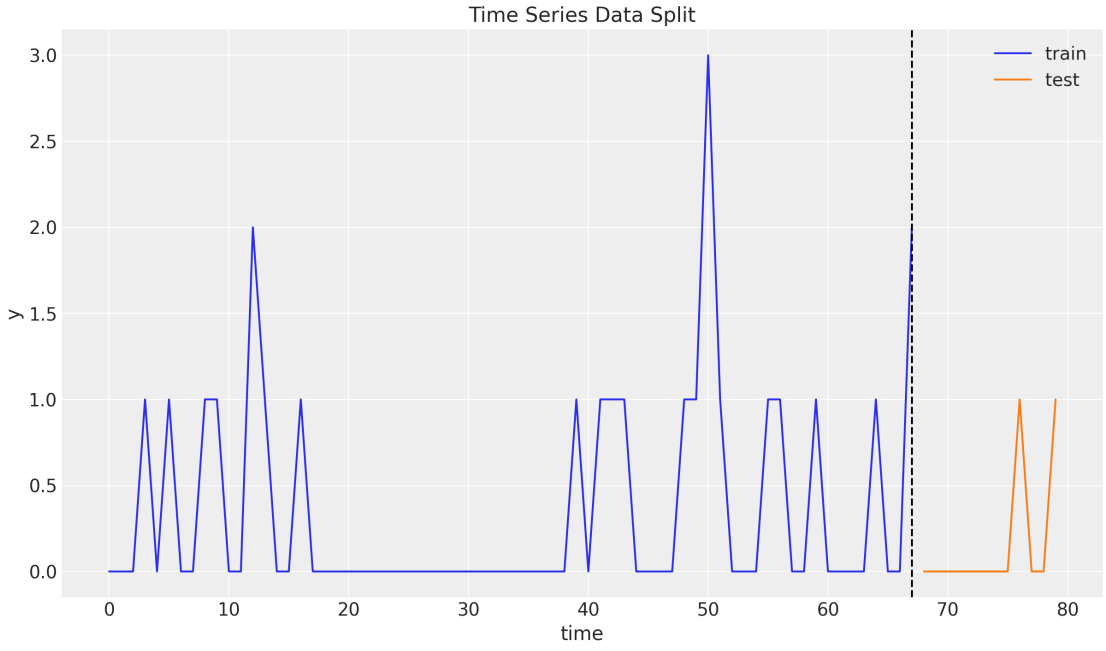

FIGURE 11. Example intermittent demand series showing sporadic non-zero values. The long stretches of zeros interspersed with variable demand sizes motivate the separation approach.

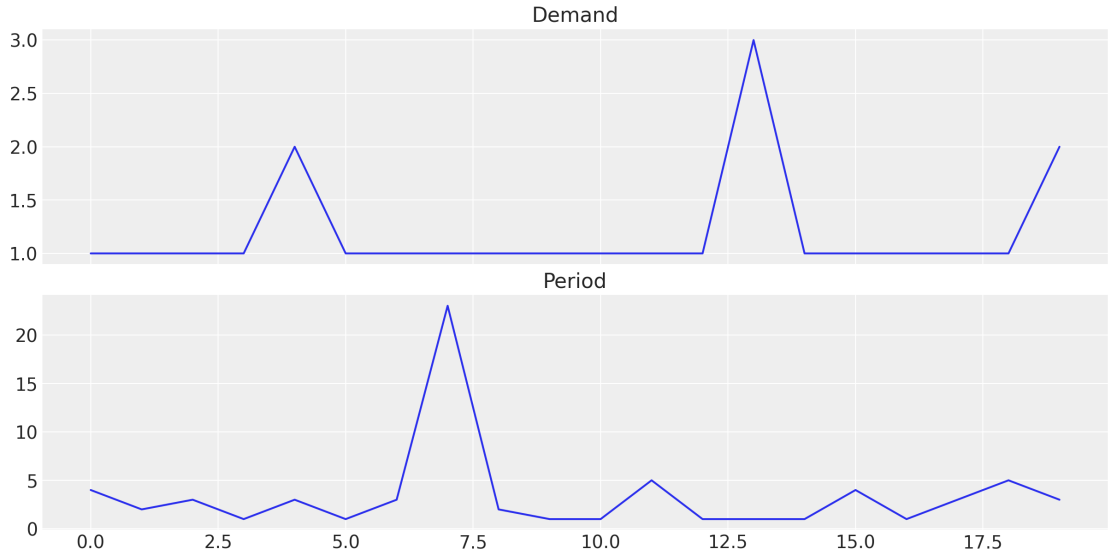

FIGURE 12. Decomposition into demand size (top) and inter-arrival time (bottom) series. Each component can now be modeled with standard exponential smoothing.

The NumPyro implementation uses `scope` to combine two exponential smoothing models:

LISTING 4. Croston's method using scoped models

```
def croston_model(z: ArrayLike, p_inv: ArrayLike, future: int = 0) -> None:
    z_forecast = scope(level_model, "demand")(z, future)
    p_inv_forecast = scope(level_model, "period_inv")(p_inv, future)

    if future > 0:
        numpyro.deterministic("z_forecast", z_forecast)
        numpyro.deterministic("p_inv_forecast", p_inv_forecast)
        numpyro.deterministic("forecast", z_forecast * p_inv_forecast)
```

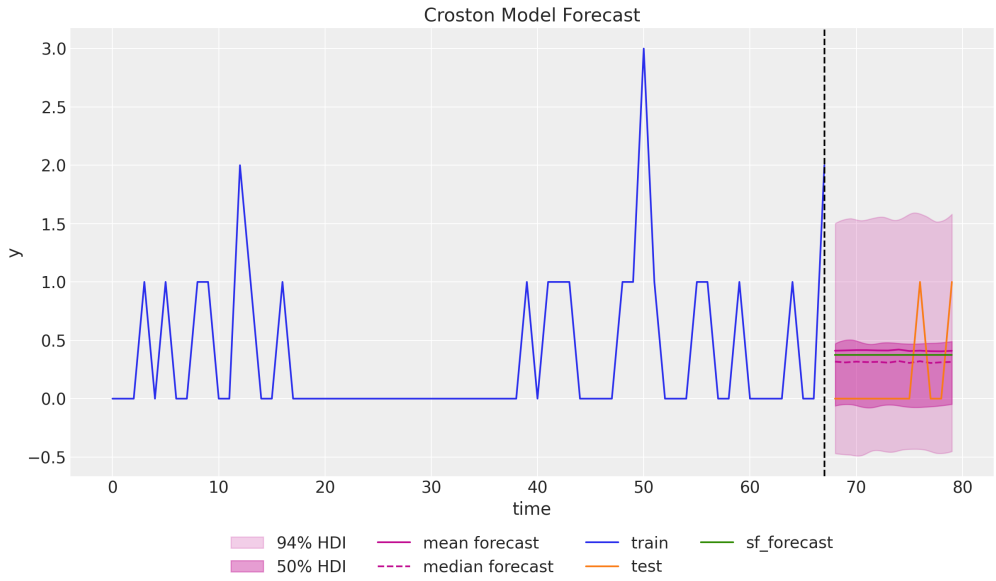

FIGURE 13. Croston's method forecast with uncertainty quantification. The probabilistic formulation provides credible intervals that appropriately reflect the high uncertainty inherent in intermittent demand forecasting.

7.2. **TSB Method.** The Teunter-Syntetos-Babai (TSB) method [22] takes a slightly different approach: instead of modeling inter-arrival periods, it directly models the *probability* that demand occurs in each period. The forecast becomes $\hat{y}_{t+h} = \hat{z}_{t+h} \cdot \hat{p}_{t+h}$, where p_t now represents the probability of non-zero demand rather than the inverse of the inter-arrival time. This formulation often provides more stable forecasts and forms the foundation for the availability-constrained model we develop in the next section.

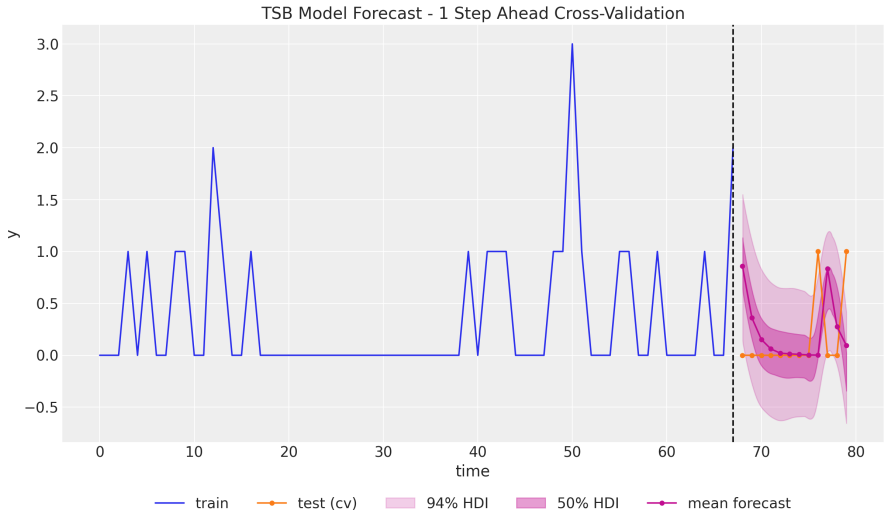

FIGURE 14. TSB method cross-validation results showing one-step-ahead forecasts. The probability formulation often provides more stable forecasts than the period-based Croston approach.

7.3. **Zero-Inflated TSB Model.** For count data with excess zeros, we can modify the TSB model to use a zero-inflated negative binomial distribution:

LISTING 5. Zero-inflated TSB transition function

```
def transition_fn(carry, t):
    z_prev, p_prev = carry
    z_next = ... # Demand size update
    p_next = ... # Probability update

    mu = z_next
    gate = 1 - p_next
    pred = numpyro.sample(
        "pred",
        dist.ZeroInflatedNegativeBinomial2(
            mean=mu, concentration=concentration, gate=gate
        ),
    )
    return (z_next, p_next), pred
```

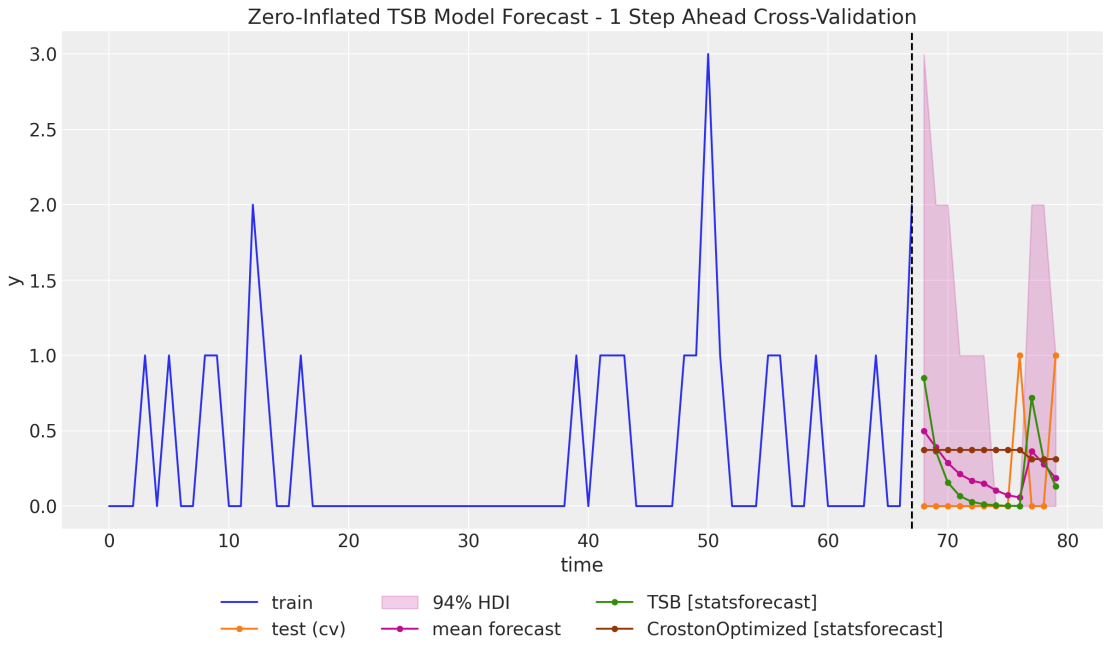

FIGURE 15. Zero-inflated TSB model cross-validation. The explicit zero-inflation component better captures the two-stage nature of intermittent demand: first whether demand occurs, then the demand size conditional on occurrence.

8. AVAILABILITY-CONSTRAINED TSB: A CUSTOM PROBABILISTIC MODEL

This section showcases one of the most powerful capabilities of probabilistic programming: the ability to encode domain knowledge directly into the model structure through a simple modification to the transition function. The solution we present here would be difficult or impossible to implement in a traditional forecasting package, yet in NumPyro it requires changing just a few lines of code.

8.1. **The Problem: Why Zeros Happen.** As discussed by Svetunkov [20], zeros in intermittent time series can arise from two fundamentally different causes. A zero might represent *true zero demand*—no customer wanted the product that day. Or it might represent an *availability constraint*—the product was out of stock, so no sales were recorded even though customers may have wanted to buy.

This distinction has profound implications for forecasting. If a product shows three consecutive zeros because nobody wanted it, the model should reasonably conclude that demand probability has declined—future forecasts should be lower. But if those three zeros occurred because the product was out of stock, the model should *not* update its demand probability at all. The zeros contain no information about underlying demand; they merely reflect a supply constraint.

Standard TSB models cannot make this distinction. They treat all zeros equally, causing the estimated demand probability to decay regardless of the cause. The business

consequence is severe: after a stockout, the model systematically underforecasts future demand. This creates a vicious cycle where low forecasts lead to low inventory orders, which lead to more stockouts, which further suppress forecasts. The retailer may conclude that a product is dying when in fact it remains popular—they just keep running out of it.

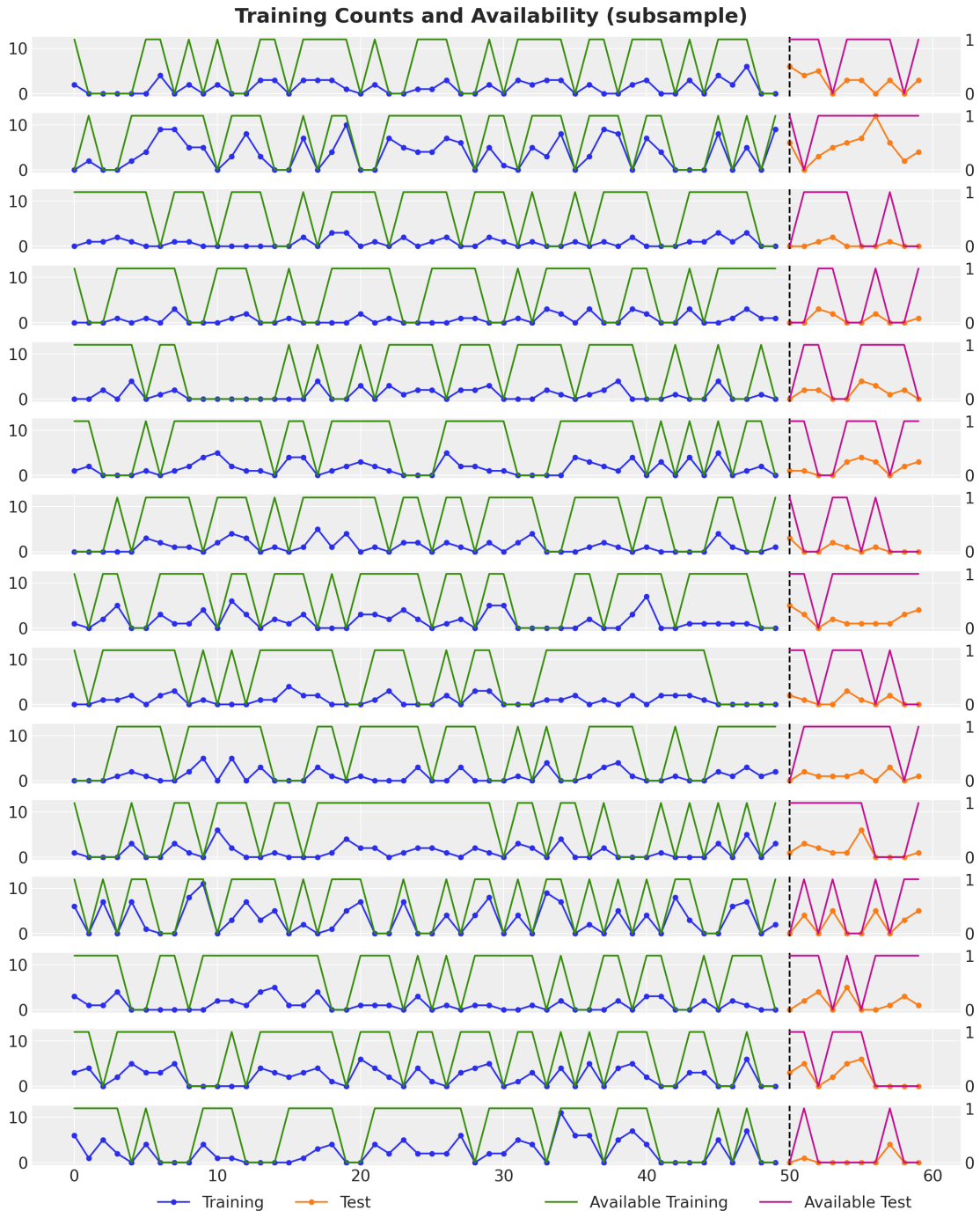

FIGURE 16. Simulated data with availability mask. The red regions indicate periods of unavailability where zeros are not informative about true demand.

8.2. **The Elegant Mathematical Modification.** The key insight is to modify the probability update equation to account for availability:

Standard TSB (when $y_t = 0$):

$$z_{t+1} = z_t$$
$$p_{t+1} = (1 - \beta) \cdot p_t$$

Availability-Constrained TSB (when $y_t = 0$):

$$z_{t+1} = z_t$$
$$p_{t+1} = (1 - a_t \cdot \beta) \cdot p_t$$

where $a_t \in \{0, 1\}$ indicates availability. When $a_t = 0$ (unavailable), the probability does not decay. Additionally, the forecast incorporates availability:

$$\hat{y}_{t+1} = a_t \cdot z_t \cdot p_t$$

Listing 6. Availability-constrained TSB transition function

```python
def transition_fn(carry, t):
    z_prev, p_prev = carry

    z_next = jnp.where(
        counts[t] > 0,
        z_smoothing * counts[t] + (1 - z_smoothing) * z_prev,
        z_prev,
    )

    p_next = jnp.where(
        counts[t] > 0,
        p_smoothing + (1 - p_smoothing) * p_prev,
        (1 - available[t] * p_smoothing) * p_prev, # Key modification
    )

    mu = z_next * p_next
    pred = numpyro.sample("pred", dist.Normal(loc=mu, scale=noise))
    pred = numpyro.deterministic("pred_obs", available[t] * pred)

    return (z_next, p_next), pred
```

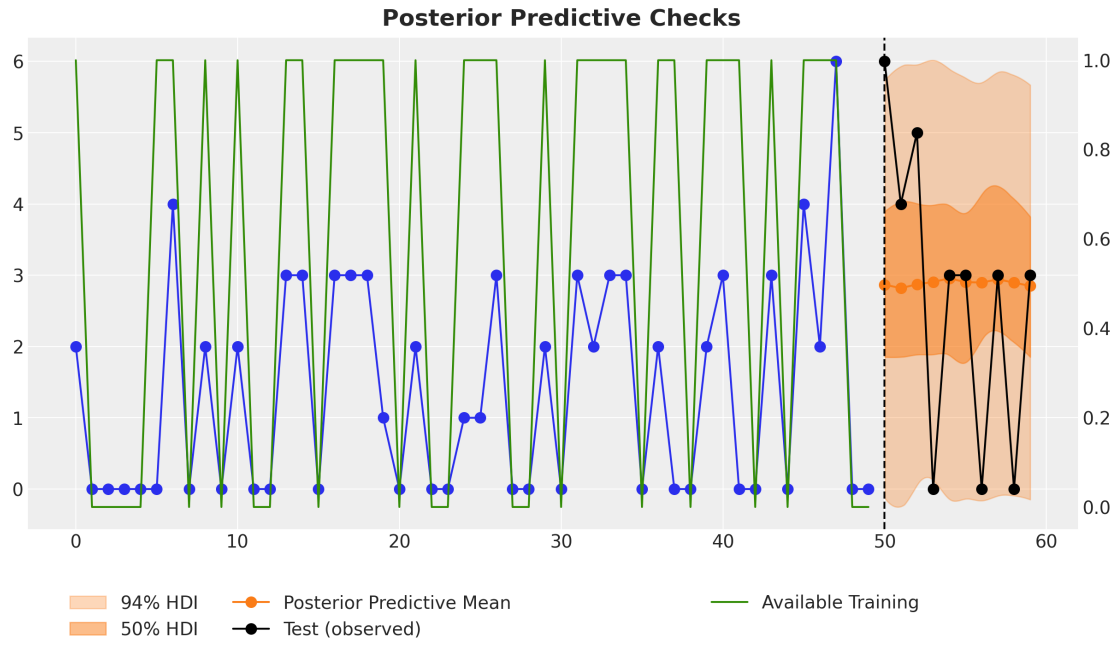

FIGURE 17. Key result: forecast comparison after zeros with availability=0. The standard TSB (left) incorrectly decays the demand probability, while the availability-constrained model (right) maintains the probability, correctly anticipating demand recovery.

8.3. **Why This Matters: The Power of Custom Probabilistic Models.** This example illustrates what makes probabilistic programming fundamentally different from using off-the-shelf forecasting packages. In a traditional tool, you are limited to the models the developers anticipated. If the standard TSB model doesn't handle availability constraints—and no standard implementation does—you are stuck. Your only options are to ignore the problem (and accept biased forecasts) or to abandon the tool entirely and code a custom solution from scratch.

In NumPyro, the modification required just a few lines of code. We changed how the probability updates when a zero is observed, conditioning on availability. The rest of the model—the priors, the likelihood, the inference procedure—remained unchanged. This is the essence of composable probabilistic programming: you specify the generative process, and the framework handles inference.

The business implications extend beyond forecast accuracy. Because the model explicitly separates demand from availability, we can now perform *scenario planning*. What would demand look like if we maintained full availability? Simply set the future availability mask to 1 and generate forecasts. This capability is invaluable for inventory planning and capacity decisions.

The approach also scales efficiently. Using stochastic variational inference, we can fit this model to 1,000 time series with 60 observations each in approximately 10 seconds—fast enough for operational use in a real retail environment.

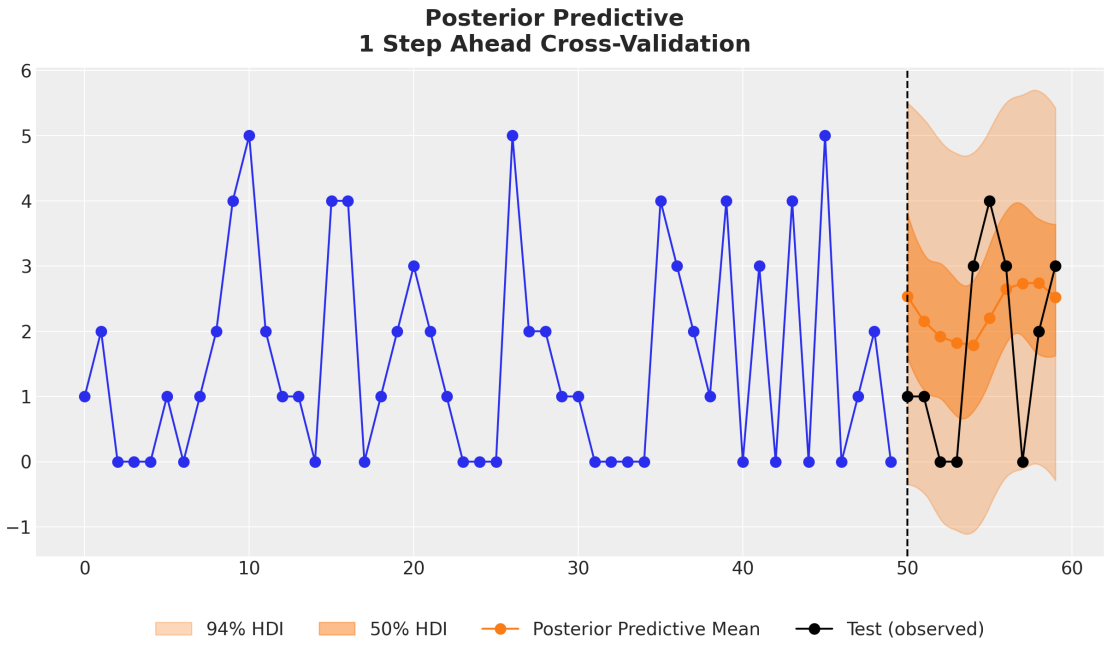

FIGURE 18. Cross-validation results demonstrating the model's ability to adapt forecasts based on availability information across multiple series.

9. CENSORED DEMAND FORECASTING

Sales data lies. When a product is out of stock, you observe sales of zero—but true demand may have been substantial. When inventory is limited to 100 units and you sell all 100, you observe 100 sales—but true demand may have been 150. In both cases, the observed data *systematically understates* actual demand. Classical forecasting methods, which take observed values at face value, will systematically underestimate future demand.

This problem is pervasive in retail and supply chain contexts. Stockouts are common, especially for popular products and during demand spikes. Inventory constraints limit what can be sold regardless of underlying demand. Promotional periods may exhaust stock quickly. In all these cases, the historical data record is *censored*—we observe a lower bound on demand, not demand itself.

Why Probabilistic Methods Excel Here. Classical forecasting methods have no mechanism for handling censoring. They treat 100 observed sales as evidence that demand was 100. A probabilistic framework, by contrast, can model the *data-generating process* explicitly. We don't model "sales" directly; we model "demand" and then account for the censoring mechanism that transforms demand into observed sales.

This is a profound conceptual shift. Instead of asking "what is my forecast given the observed data?" we ask "what demand process could have generated the observed data, accounting for the fact that demand above the inventory level gets recorded as exactly

the inventory level?" The inference machinery then finds demand parameters consistent with this generative story.

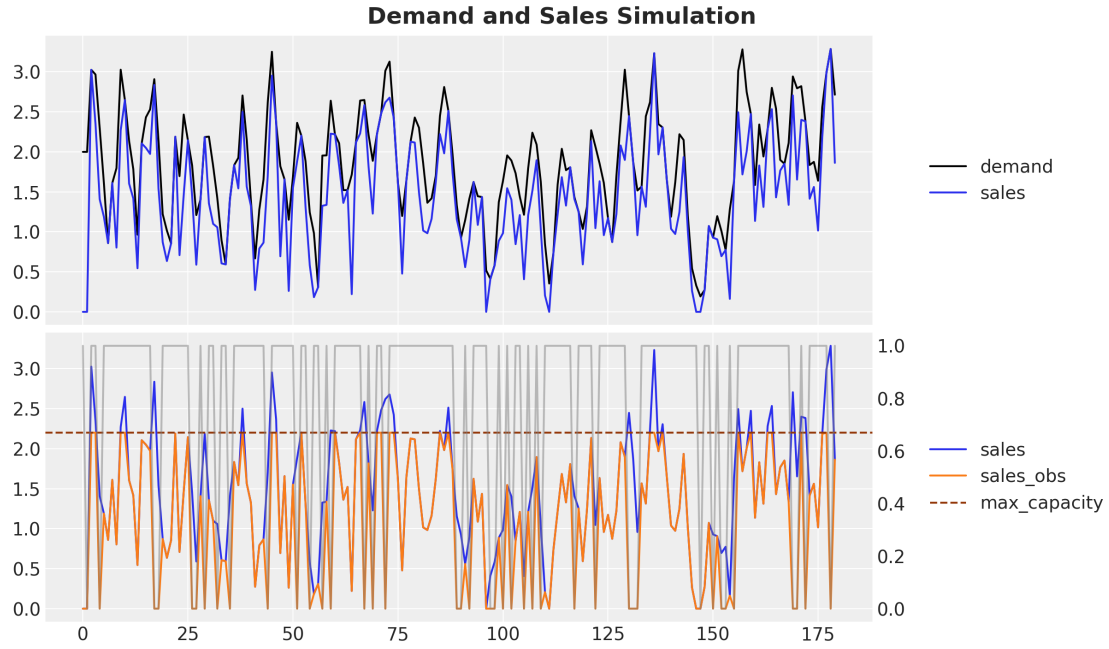

FIGURE 19. Demand data with stockout periods (shaded). During stockouts, observed sales represent a lower bound on true demand, not demand itself.

9.1. **Censored Likelihood.** The censored normal likelihood accounts for observations that are right-censored:

LISTING 7. Censored normal likelihood implementation

```
def censored_normal(loc, scale, y, censored):
    distribution = dist.Normal(loc=loc, scale=scale)
    ccdf = 1 - distribution.cdf(y)
    numpyro.sample(
        "censored_label",
        dist.Bernoulli(probs=ccdf).mask(censored == 1),
        obs=censored
    )
    return numpyro.sample("pred", distribution.mask(censored != 1))
```

For uncensored observations, we use the standard normal likelihood. For censored observations, we contribute only the survival function $P(Y > y)$ to the likelihood.

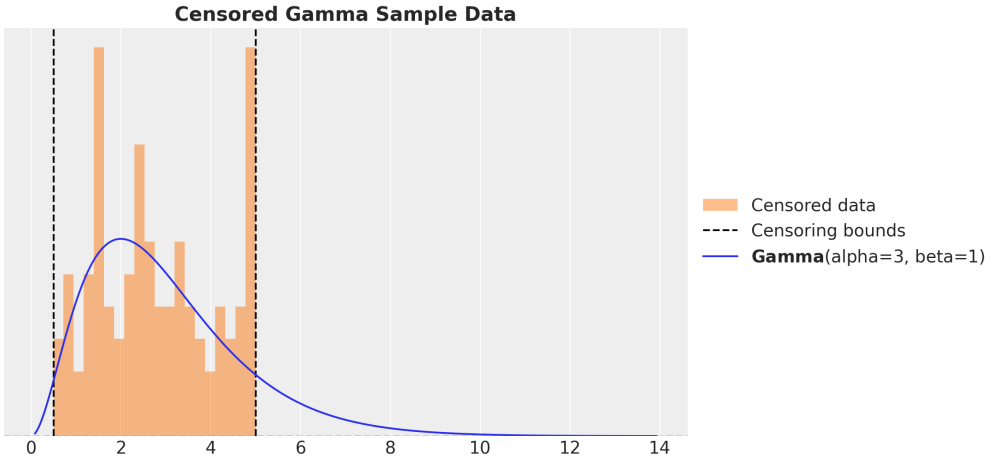

FIGURE 20. Illustration of censoring: when inventory is limited, we observe the inventory level but not the true demand that exceeded it.

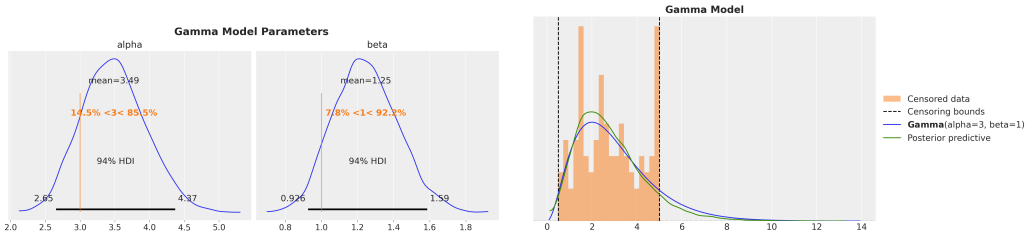

FIGURE 21. Left: Standard model ignoring censoring underestimates parameters. Right: Censored likelihood recovers true parameters, correctly accounting for truncated observations.

9.2. **Censored Time Series Model.** Integrating the censored likelihood into a time series model requires modifying the transition function:

LISTING 8. AR(2) transition with censored likelihood

```
def transition_fn(carry, t):
    y_prev_1, y_prev_2 = carry
    ar_part = phi_1 * y_prev_1 + phi_2 * y_prev_2
    pred_mean = mu + ar_part + seasonal[t]
    # Censored likelihood
    pred = censored_normal(pred_mean, sigma, y[t], censored[t])
    return (pred, y_prev_1), pred
```

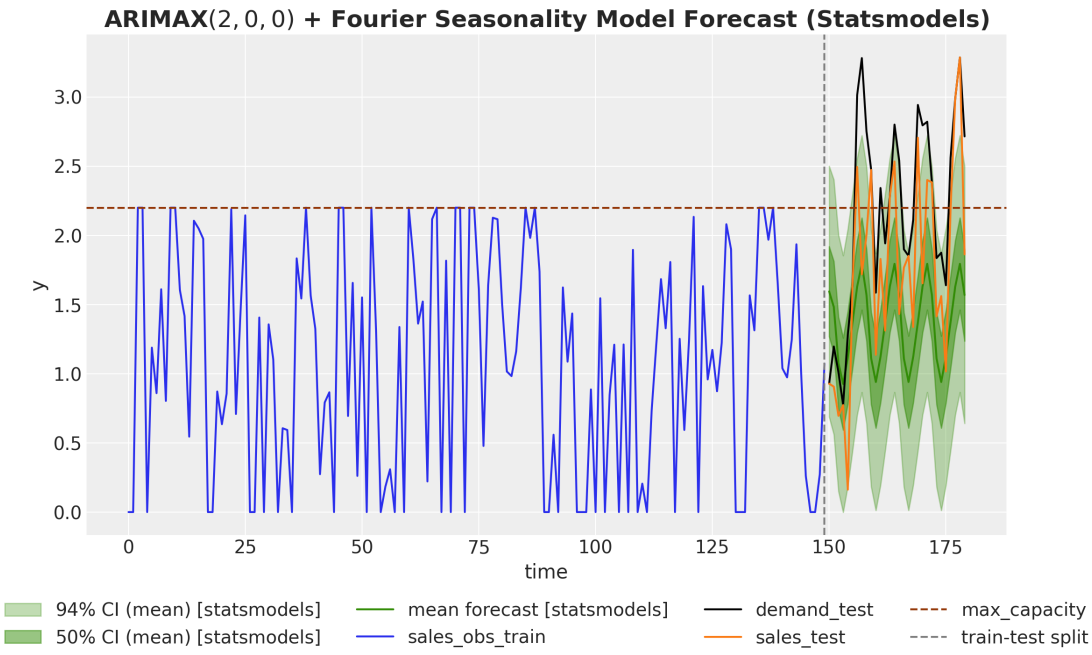

FIGURE 22. Standard ARIMA forecast ignoring censoring. The model underestimates demand during stockout periods and produces biased forecasts.

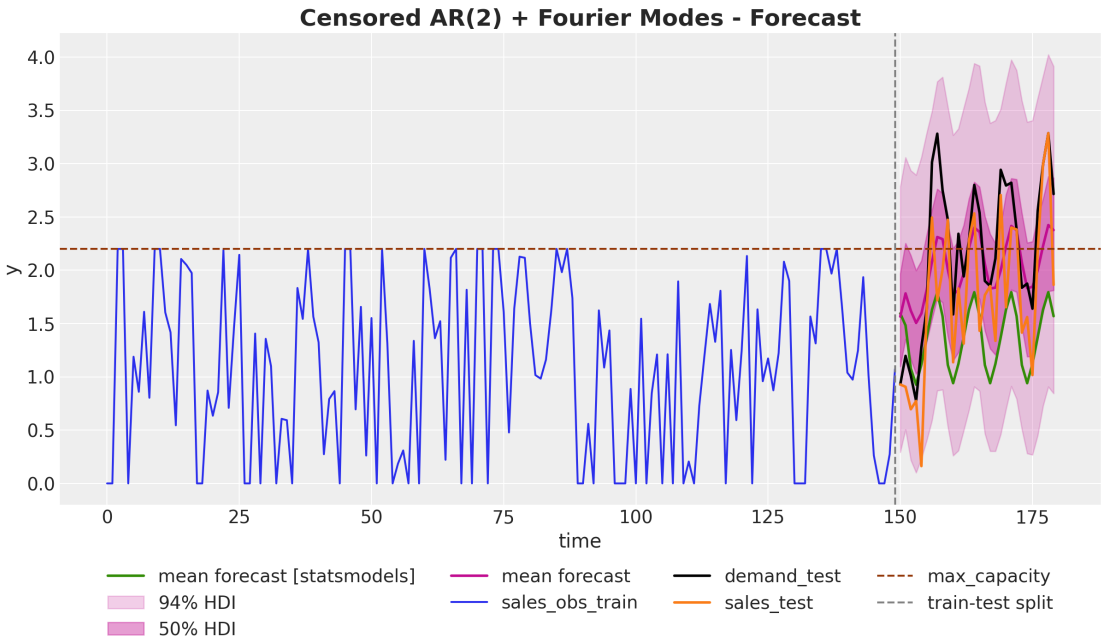

FIGURE 23. Censored ARIMA forecast correctly inferring latent demand. The model produces unbiased forecasts by properly accounting for the censoring mechanism.

10. Hierarchical Price Elasticity Models

Price elasticity estimation—measuring how demand responds to price changes—benefits from hierarchical modeling when data is sparse at the individual product level but abundant across products.

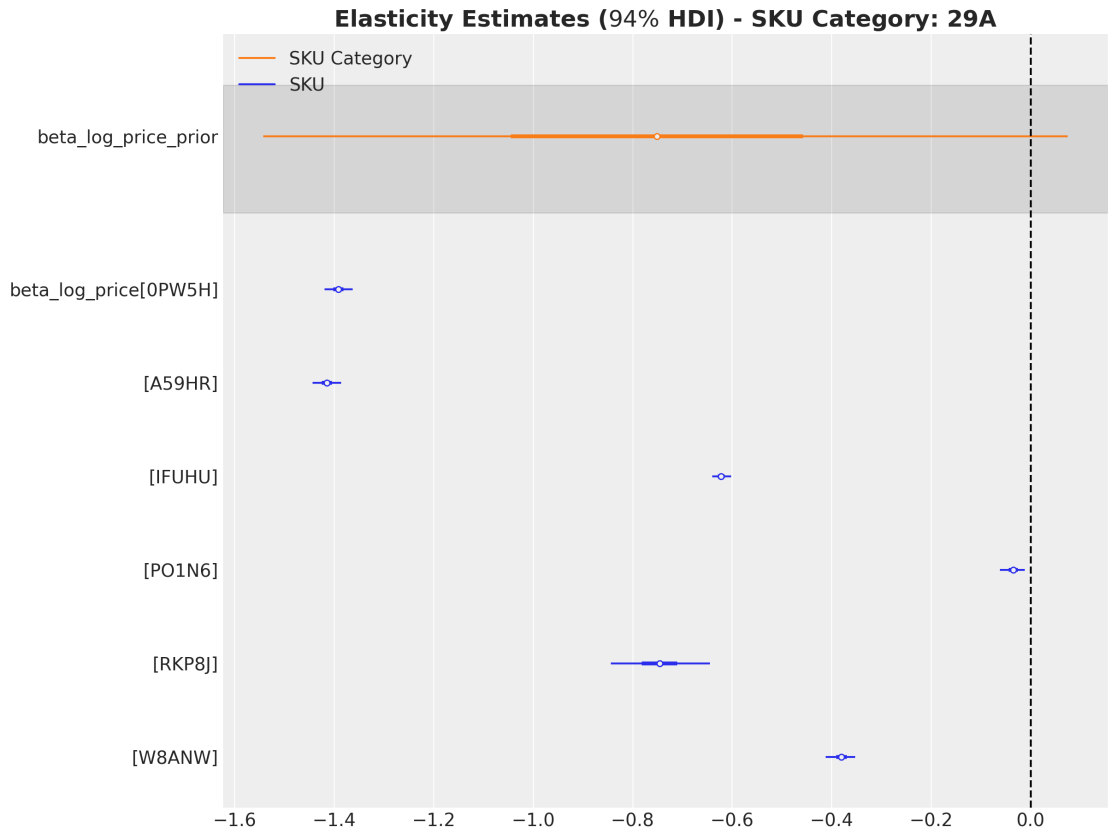

FIGURE 24. Posterior distributions of price elasticities across SKUs. The hierarchical structure shrinks extreme estimates toward the category mean while preserving meaningful product-level heterogeneity.

The hierarchical prior on elasticities takes the form:

$$\begin{aligned}\beta_{\text{price},i} &\sim \text{Normal}(\mu_\beta, \sigma_\beta)\\ \mu_\beta &\sim \text{Normal}(-1, 0.5)\\ \sigma_\beta &\sim \text{HalfNormal}(0.5)\end{aligned}$$

This structure regularizes elasticity estimates for products with limited price variation while allowing products with rich data to deviate from the category mean.

11. Model Calibration with Additional Likelihoods

What if you have domain expertise that the historical data doesn't fully capture? Perhaps you know from physics that a certain relationship should be approximately linear in a particular regime, even though your data is too sparse in that region to reveal the pattern. Perhaps you have experimental results—a randomized controlled trial, an A/B test, a lift test—that provide ground truth about a causal relationship that observational data alone cannot identify.

Traditional forecasting methods have no mechanism for incorporating such knowledge. You can adjust priors in a Bayesian model, but priors influence the entire parameter space uniformly. What if you need to constrain the model only in specific regimes, at specific time points, or for specific relationships?

This section showcases a powerful technique that addresses exactly this need: treating domain knowledge as additional likelihood terms. The insight, inspired by the Pyro DLM tutorial [16], is that in probabilistic programming, there is no fundamental distinction between "data we observed" and "constraints we want to impose." Both are expressed as likelihood contributions that pull the posterior toward values consistent with our information.

11.1. **The Core Insight: Priors as Likelihoods.** The technique is elegant in its simplicity. To constrain a latent variable to be approximately equal to some known value at specific points, we add an "observation" of that latent variable with appropriate uncertainty. The inference machinery then balances this constraint against the actual data, finding parameter values consistent with both.

This approach is remarkably flexible. We can calibrate coefficients at specific time points when we have experimental data from those periods. We can constrain model behavior in specific regimes where we have domain knowledge but sparse data. And as we will demonstrate, we can calibrate *any* latent component—including flexible nonparametric components like Gaussian Processes.

11.2. **Key Application: HSGP Calibration for Electricity Demand.** Consider the problem of forecasting electricity demand as a function of temperature [11]. The relationship is clearly non-linear: at mild temperatures, demand is low (no heating or cooling needed); as temperature rises, air conditioning loads increase demand; at very high temperatures, this effect may saturate as most air conditioners reach maximum capacity.

We model this non-linear relationship using a Hilbert Space Gaussian Process (HSGP)—an efficient approximation to a full Gaussian Process that scales well with data size. The model also includes hour-of-day and day-of-week seasonal effects, heteroscedastic noise (demand variance increases with temperature), and a Student-t likelihood for robustness to outliers.

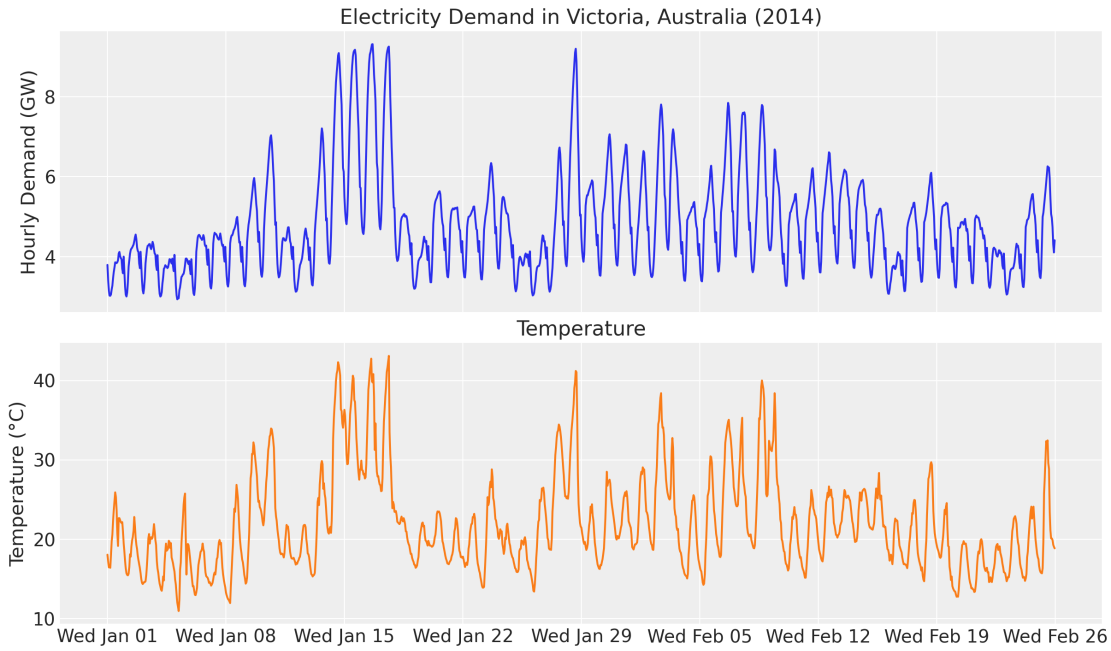

FIGURE 25. Electricity demand data showing the non-linear relationship with temperature. At extreme high temperatures, the relationship appears to flatten, but data is sparse in this regime, making the pattern difficult to estimate from observations alone.

The challenge arises at extreme temperatures above 32°C. Historical data in this regime is sparse—heat waves are rare events. Yet this is precisely where accurate forecasts matter most, because heat waves strain the electrical grid and accurate demand prediction is critical for preventing blackouts.

Domain knowledge suggests the temperature effect should stabilize around 0.13 in this high-temperature regime: once most air conditioners are running at full capacity, additional temperature increases have diminishing marginal effect on demand. But the uncalibrated model, lacking sufficient data, produces uncertain and oscillating estimates in this region.

The solution is to inject this domain knowledge as a calibration constraint:

LISTING 9. HSGP calibration for electricity demand

```
# Temperature effect as HSGP (Matern 5/2 approximation)
beta_temperature = numpyro.deterministic(
    "beta_temperature",
    hsgp_matern(x=temperature, nu=5/2, alpha=alpha,
                length=length_scale, ell=ell, m=m)
)

# Calibrate GP for high temperatures (>32C)
temperature_prior_idx = jnp.where(temperature > 32.0)[0]
```

```
if temperature_prior_idx is not None:
    numpyro.sample(
        "temperature_prior",
        dist.Normal(loc=0.13, scale=0.01), # Domain knowledge
        obs=beta_temperature[temperature_prior_idx]
    )
```

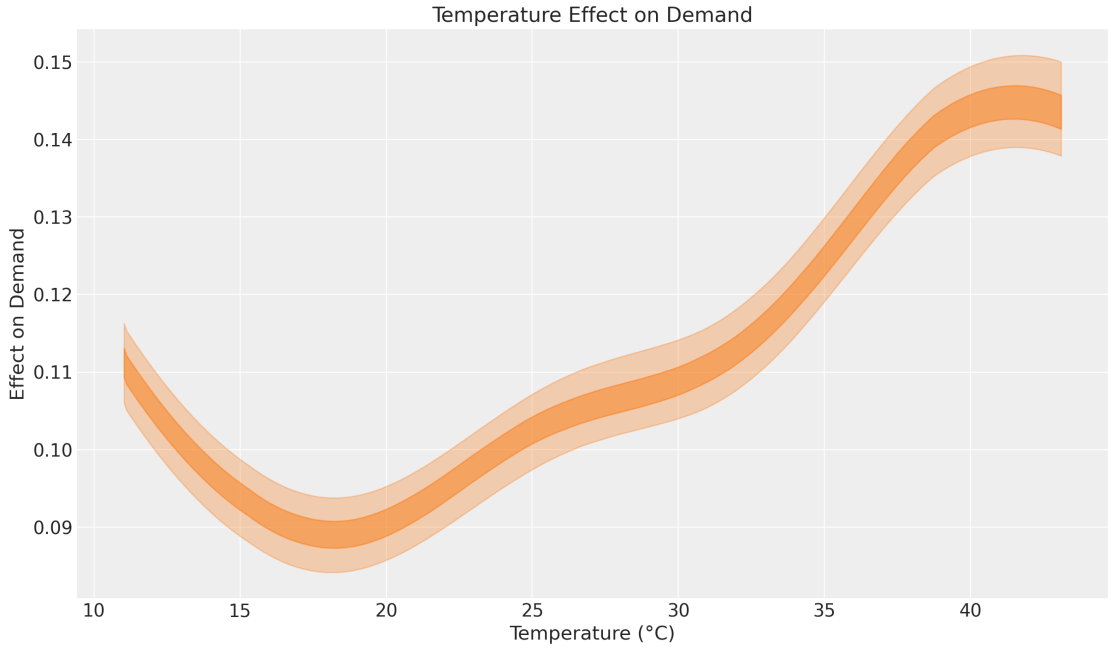

FIGURE 26. Uncalibrated temperature effect from the baseline model. The effect oscillates between 0.11 and 0.15 in the high-temperature regime due to data sparsity.

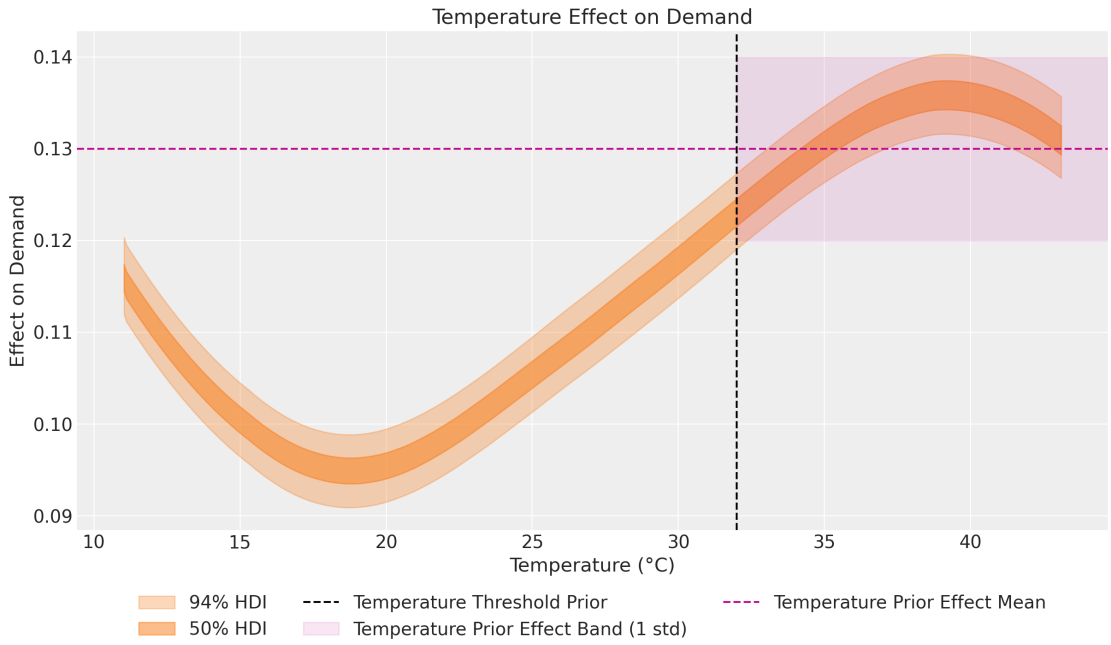

FIGURE 27. Calibrated temperature effect. The domain knowledge constraint stabilizes the effect around 0.13 for temperatures above 32°C while maintaining flexibility elsewhere. Forecast metrics (CRPS) remain essentially unchanged.

11.3. **Why HSGP Calibration is Novel.** Most calibration examples in the literature focus on simple linear coefficients. What makes this example particularly powerful is that it demonstrates calibration of a *flexible nonparametric component*. The Gaussian Process can learn arbitrary smooth functions of temperature, yet we can still inject domain knowledge at specific regimes where we have expertise but sparse data.

This combination—flexibility where data is abundant, constraint where data is sparse—is exactly what practitioners need. The model retains the ability to discover unexpected nonlinear relationships in temperature ranges with good data coverage. But in the extreme high-temperature regime, where data is scarce and forecasts are most critical, the domain knowledge provides stability.

Point estimation methods have no natural analog to this capability. You cannot "inject knowledge at specific regimes" into a maximum likelihood estimate or a gradient-boosted tree. The probabilistic framework's explicit representation of uncertainty and the generative modeling perspective make this calibration approach possible.

11.4. **Application: MMM Calibration with Lift Tests.** The same calibration principle applies to Marketing Mix Models (MMM), where the challenge is even more acute. MMM attempts to estimate the causal effect of advertising spend on sales from observational data. But observational data is confounded: companies spend more on advertising during periods when they expect high sales anyway. Lift tests—randomized experiments

that measure the true causal effect of advertising—provide ground truth that can break this confounding.

By incorporating lift test results as calibration likelihoods, we constrain the model's media effectiveness estimates to be consistent with experimental evidence [15]. The model learns from both the observational data (which provides information about trends, seasonality, and relative effects) and the experimental data (which provides unconfounded estimates of absolute effects):

FIGURE 28. MMM calibration with lift tests. Experimental results constrain media effectiveness estimates, addressing the unobserved confounders problem common in marketing attribution.

12. Stochastic Variational Inference

Markov Chain Monte Carlo (MCMC) can be computationally expensive for large-scale forecasting problems involving thousands of time series. Stochastic Variational Inference (SVI) [3] transforms posterior inference into an optimization problem, enabling scalable inference.

12.1. **Core Concepts. Variational family**: Approximate the complex posterior $p(\theta|y)$ with a simpler distribution $q_\phi(\theta)$ parameterized by ϕ.

ELBO: The Evidence Lower Bound serves as the optimization target:

$$\text{ELBO}(\phi) = \mathbb{E}_{q_\phi}[\log p(y|\theta)] - \text{KL}[q_\phi(\theta)\|p(\theta)]$$

The first term encourages the approximation to explain the data well; the second term regularizes toward the prior.

12.2. **NumPyro SVI Workflow.**

LISTING 10. SVI setup with Optax optimizer

```
from numpyro.infer import SVI, Trace_ELBO
from numpyro.infer.autoguide import AutoNormal
```

```
import optax

# Define guide (variational family)
guide = AutoNormal(model)

# Define optimizer with Optax
scheduler = optax.linear_onecycle_schedule(
    transition_steps=num_steps,
    peak_value=0.01,
    pct_start=0.1
)
optimizer = optax.chain(
    optax.clip_by_global_norm(1.0),
    optax.adam(learning_rate=scheduler)
)

# Initialize SVI
svi = SVI(model=model, guide=guide, optim=optimizer, loss=Trace_ELBO())

# Run optimization
svi_result = svi.run(rng_key, num_steps, y=data)
```

12.3. **JAX Ecosystem Integration.** NumPyro integrates seamlessly with the broader JAX ecosystem:

- **Optax**: Composable gradient transformations including Adam, learning rate schedulers, and gradient clipping.
- **Flax NNX**: Neural network modules that can be integrated into probabilistic models (see Section 13).
- **JAX JIT**: Automatic compilation for efficient execution on GPU/TPU.

12.4. **AutoGuides.** NumPyro provides automatic guide construction:

- `AutoNormal`: Mean-field approximation with independent Normal for each parameter
- `AutoMultivariateNormal`: Full covariance approximation
- `AutoLowRankMultivariateNormal`: Low-rank approximation for high-dimensional problems

The automatic handling of constraints and transformations makes SVI accessible without deep expertise in variational inference.

13. Hybrid Deep State-Space Models

The final advancement combines neural network components with probabilistic state-space models, enabling learning of complex patterns while maintaining interpretable structure [12].

13.1. **Neural Network Components.** We define embedding and transition networks using Flax NNX:

LISTING 11. Neural network components for hybrid model

```
class StationEmbedding(nnx.Module):
    def __init__(self, n_stations: int, embedding_dim: int, rngs: nnx.Rngs):
        self.embedding = nnx.Embed(
            num_embeddings=n_stations,
            features=embedding_dim,
            rngs=rngs
        )

    def __call__(self, station_idx):
        return self.embedding(station_idx)


class TransitionNetwork(nnx.Module):
    def __init__(self, embedding_dim: int, hidden_dim: int, rngs: nnx.Rngs):
        self.dense1 = nnx.Linear(embedding_dim + 1, hidden_dim, rngs=rngs)
        self.dense2 = nnx.Linear(hidden_dim, 1, rngs=rngs)

    def __call__(self, embedding, level):
        x = jnp.concatenate([embedding, level[..., None]], axis=-1)
        x = nnx.relu(self.dense1(x))
        return self.dense2(x).squeeze(-1)
```

13.2. **Hybrid Model Structure.** The hybrid model augments a local level state-space model with neural network corrections:

LISTING 12. Hybrid transition function with NN component

```
def transition_fn(carry, t):
    level_prev = carry

    # Station embedding
    embedding = station_embedding(station_idx)

    # NN correction to drift
    nn_correction = transition_network(embedding, level_prev)

    # Local level update with NN correction
    level = level_prev + drift + nn_correction
    level = jnp.where(
        t < t_max,
        level_smoothing * y[:, t] + (1 - level_smoothing) * level,
        level
    )

```

```
    mu = level
    pred = numpyro.sample("pred", dist.Normal(loc=mu, scale=noise))

    return level, pred
```

The neural network learns station-specific adjustments to the level dynamics that cannot be captured by the simple local level model alone.

13.3. **SVI Training.** Training hybrid models requires SVI due to the non-conjugate neural network parameters:

Listing 13. SVI training for hybrid model

```
# Initialize neural network
nn_params = nnx.state(transition_network)

# AutoNormal guide for probabilistic parameters
guide = AutoNormal(hybrid_model, init_loc_fn=init_to_mean)

# Optax optimizer with OneCycle schedule
scheduler = optax.linear_onecycle_schedule(
    transition_steps=20000,
    peak_value=0.005
)
optimizer = optax.adam(learning_rate=scheduler)

svi = SVI(hybrid_model, guide, optimizer, Trace_ELBO())
svi_result = svi.run(rng_key, 20000, y=data, station_idx=stations)
```

13.4. **When Neural Networks Help.** Neural network corrections are most valuable when:

- Multiple related time series share complex non-linear patterns
- Standard state-space models leave systematic residual structure
- Sufficient data exists to learn embeddings without overfitting

For simpler problems, pure state-space models often suffice and provide better interpretability.

14. Conclusion

14.1. **The Paradigm Shift.** Throughout this article, we have explored a fundamental shift in how we approach time series forecasting. Traditional methods ask: "Given historical data, what is my best guess for future values?" Probabilistic forecasting asks a different question: "What is the full distribution of possible future values, and how confident should I be in different outcomes?"

This shift from point estimates to distributions has profound implications for decision-making. A retailer deciding on inventory levels needs to know not just the expected demand, but the probability of demand exceeding various thresholds. An energy grid

operator planning for peak load needs to understand the range of possible demand scenarios, not just the central forecast. A marketing team allocating budget needs to quantify uncertainty in media effectiveness estimates, not just obtain a single "best" number.

But probabilistic forecasting offers more than uncertainty quantification. As we have demonstrated throughout this article, probabilistic programming enables us to directly incorporate business constraints, domain knowledge, and potentially causal relationships into our models. When sales data is censored by stockouts, we model the censoring mechanism. When different regions share common patterns, we let them borrow statistical strength through hierarchical structures. When we have experimental evidence about causal effects, we inject it as calibration constraints. This flexibility to encode domain expertise directly into model structure is what makes probabilistic forecasting truly transformative for business applications.

14.2. **Matching Problems to Solutions.** A key theme of this article has been matching the right technique to the right problem. Not every forecasting challenge requires sophisticated methods—sometimes a simple exponential smoothing model suffices. The value of understanding the full toolkit is knowing when each approach provides genuine advantage.

When data is sparse across related series, hierarchical models shine. The tourism forecasting example demonstrated how regions with limited history can borrow strength from data-rich regions. Classical methods, which treat each series independently, cannot achieve this information sharing.

When stockouts mask true demand, censored likelihoods are essential. Standard methods treat observed sales as ground truth, systematically underestimating demand. Only by modeling the data-generating process—including the censoring mechanism—can we recover unbiased demand estimates.

When zeros arise from availability constraints rather than lack of demand, the availability-constrained TSB model provides the solution. By modifying a single line in the transition function, we encode the distinction between true zero demand and supply-constrained zeros directly into the model structure.

When domain expertise exceeds what historical data can reveal, calibration via additional likelihoods enables injecting that knowledge. The HSGP calibration example showed how to constrain a flexible Gaussian Process in specific regimes where we have expertise but sparse data.

When scale matters—thousands of time series to forecast—stochastic variational inference with GPU acceleration makes probabilistic methods practical. What would take hours with MCMC can run in minutes with SVI, enabling rapid iteration and operational deployment.

14.3. **Limitations and When to Use Simpler Methods.** Intellectual honesty requires acknowledging when probabilistic methods are *not* the right choice. Strong baseline models are often hard to beat, and the additional complexity of probabilistic programming may not be justified for every forecasting task.

Computational cost: Probabilistic methods are more computationally expensive than simple exponential smoothing or ARIMA models. If you need to forecast millions of time series and point forecasts suffice, simpler methods may be more practical.

Expertise required: Specifying a probabilistic model requires understanding of statistical modeling, prior selection, and inference diagnostics. Teams without this expertise may be better served by automated tools like AutoETS or Prophet that make reasonable default choices.

When baselines suffice: For stable, well-behaved time series with abundant historical data and no special constraints, classical methods often perform comparably. The advantages of probabilistic forecasting emerge most clearly when data has quirks (censoring, intermittency, hierarchical structure) or when domain knowledge needs to be incorporated.

Model misspecification risk: A misspecified probabilistic model can produce confidently wrong forecasts with misleadingly narrow uncertainty bands. Proper model checking, posterior predictive checks, and out-of-sample validation are essential but require additional effort.

The pragmatic recommendation is to start simple, establish strong baselines, and add complexity only when it demonstrably improves performance on held-out data or enables incorporating domain knowledge that simpler methods cannot capture.

14.4. **Future Directions.** The field of probabilistic forecasting continues to evolve rapidly. Several directions hold particular promise:

State space models in PyMC-Extras: Recent development of structural time series components in PyMC provides composable building blocks (local linear trend, seasonal components, regression effects) with Kalman filter-based inference.

Foundation models for time series: Large pre-trained models like Chronos and TimeGPT offer zero-shot forecasting capabilities. How to combine these with domain-specific probabilistic structure remains an open question.

Causal extensions: The calibration technique we demonstrated with lift tests is a step toward causal forecasting. Methods like Causal DeepAR extend these ideas to deep learning architectures, enabling counterfactual prediction.

Integration with decision optimization: The natural next step after probabilistic forecasting is decision-making under uncertainty. Connecting forecast distributions to optimization objectives (inventory costs, energy dispatch, marketing ROI) closes the loop from prediction to action.

14.5. **Broader Impact.** The techniques presented in this article enable a different kind of relationship between forecasters and decision-makers. Instead of delivering a single number that stakeholders must trust or reject, probabilistic forecasting delivers a range of scenarios with associated probabilities. This transparency enables more informed decision-making and more honest communication about uncertainty.

Perhaps more importantly, the ability to encode domain knowledge directly into models changes the role of subject matter experts. In traditional forecasting, domain knowledge enters informally—an analyst might "adjust" a model's output based on intuition. In probabilistic programming, domain knowledge enters formally—as priors, constraints, or structural assumptions that become part of the model itself. This formalization makes assumptions explicit, auditable, and updatable as understanding evolves.

The combination of rigorous uncertainty quantification, flexible model specification, and the ability to incorporate domain expertise makes probabilistic forecasting a powerful tool for any organization serious about data-driven decision-making under uncertainty.

14.6. **Acknowledgments.** We gratefully acknowledge the NumPyro core developers—Du Phan, Neeraj Pradhan, and Martin Jankowiak—for creating and maintaining this excellent library. The Pyro team at Uber AI Labs, including Eli Bingham, Fritz Obermeyer, and Noah Goodman, laid the foundation with Pyro [2]. The JAX team at Google enables the performance that makes these methods practical.

We also thank the broader probabilistic programming community, whose contributions through blog posts, tutorials, and open-source code have made these techniques accessible to practitioners. Special thanks to the PyMC Labs team for their work on probabilistic forecasting for business applications, which inspired many of the examples in this article.

References

[1] Alexandrov, A., Benidis, K., Bohlke-Schneider, M., et al. GluonTS: Probabilistic and neural time series modeling in Python. *Journal of Machine Learning Research 21*, 116 (2020), 1–6.

[2] Bingham, E., Chen, J. P., Jankowiak, M., Obermeyer, F., Pradhan, N., Karaletsos, T., Singh, R., Szerlip, P. A., Horsfall, P., and Goodman, N. D. Pyro: Deep universal probabilistic programming. *J. Mach. Learn. Res. 20* (2019), 28:1–28:6.

[3] Blei, D. M., Kucukelbir, A., and McAuliffe, J. D. Variational inference: A review for statisticians. *Journal of the American Statistical Association 112*, 518 (2017), 859–877.

[4] Bradbury, J., Frostig, R., Hawkins, P., Johnson, M. J., Leary, C., Maclaurin, D., Necula, G., Paszke, A., VanderPlas, J., Wanderman-Milne, S., and Zhang, Q. JAX: composable transformations of Python+NumPy programs, 2018.

[5] Croston, J. D. Forecasting and stock control for intermittent demands. *Journal of the Operational Research Society 23*, 3 (1972), 289–303.

[6] Durbin, J., and Koopman, S. J. *Time Series Analysis by State Space Methods*, 2nd ed. Oxford University Press, 2012.

[7] Harvey, A. C. *Forecasting, Structural Time Series Models and the Kalman Filter*. Cambridge University Press, 1989.

[8] Hyndman, R. J., Koehler, A. B., Ord, J. K., and Snyder, R. D. *Forecasting with Exponential Smoothing: The State Space Approach*. Springer Science & Business Media, 2008.

[9] Hyndman, R. J., Koehler, A. B., Snyder, R. D., and Grose, S. A state space framework for automatic forecasting using exponential smoothing methods. *International Journal of Forecasting 18*, 3 (2002), 439–454.

[10] Lim, B., Arik, S. Ö., Loeff, N., and Pfister, T. Temporal fusion transformers for interpretable multi-horizon time series forecasting. *International Journal of Forecasting 37*, 4 (2021), 1748–1764.

[11] Orduz, J. Electricity Demand Forecast: Dynamic Time-Series Model with Prior Calibration. `https://juanitorduz.github.io/electricity_forecast_with_priors/`, 2024.

[12] Orduz, J. From Pyro to NumPyro: Forecasting Hierarchical Models - Part III. `https://juanitorduz.github.io/numpyro_hierarchical_forecasting_3/`, 2024.

[13] Oreshkin, B. N., Carpov, D., Chapados, N., and Bengio, Y. N-BEATS: Neural basis expansion analysis for interpretable time series forecasting. In *International Conference on Learning Representations* (2020).

[14] Phan, D., Pradhan, N., and Jankowiak, M. Composable Effects for Flexible and Accelerated Probabilistic Programming in NumPyro. *arXiv preprint arXiv:1912.11554* (2019).

[15] PyMC-Labs Development Team. PyMC-Marketing: Lift Test Calibration. `https://www.pymc-marketing.io/en/stable/notebooks/mmm/mmm_lift_test.html`, 2024.

[16] Pyro Development Team. Forecasting with Dynamic Linear Model (DLM). https://pyro.ai/examples/forecasting_dlm.html, 2020.
[17] Salinas, D., Flunkert, V., Gasthaus, J., and Januschowski, T. DeepAR: Probabilistic forecasting with autoregressive recurrent networks. *International Journal of Forecasting 36*, 3 (2020), 1181–1191.
[18] Smyl, S., Bergmeir, C., Ranganathan, S., et al. Local and global trend Bayesian exponential smoothing models. *International Journal of Forecasting* (2024).
[19] Svetunkov, I. *Forecasting and Analytics with the Augmented Dynamic Adaptive Model (ADAM)*. Chapman and Hall/CRC, 2023.
[20] Svetunkov, I. Why Zeroes Happen? https://openforecast.org/2024/11/18/why-zeroes-happen/, 2024.
[21] Taylor, S. J., and Letham, B. Forecasting at scale. *The American Statistician 72*, 1 (2018), 37–45.
[22] Teunter, R. H., Syntetos, A. A., and Babai, M. Z. Intermittent demand: Linking forecasting to inventory obsolescence. *European Journal of Operational Research 214*, 3 (2011), 606–615.

Email address: juanitorduz@gmail.com
URL: https://juanitorduz.github.io/

Berlin, Germany