pyro-ppl/**numpyro**

Probabilistic programming with NumPy powered by JAX for autograd and JIT compilation to GPU/TPU/CPU.

Probabilistic Forecasting at Scale with Numpyro

Juan Orduz (PyMC Labs)

KEYWORDS — Probabilistic Models, Forecasting, JAX, NumPyro

I. JAX AND NUMPYRO

"JAX is a Python library for accelerator-oriented array computation and program transformation, designed for high-performance numerical computing and large-scale machine learning."

"NumPyro is a lightweight probabilistic programming library that provides a NumPy backend for Pyro. We rely on JAX for automatic differentiation and JIT compilation to GPU / CPU."

NumPyro integrates with the rich JAX ecosystem: BlackJAX (JAX samplers), Optax (optimization), flowMC (normalizing flows), etc.

II. CLASSICAL TIME SERIES MODELS

We provide implementation for the most common statistical time series models (exponential smoothing, ARIMAX, Croston's method, TSB and many more) and also state space models.

```python
# See https://juanitorduz.github.io/exponential_smoothing_numpyro/
def level_model(y: Array, future: int = 0) -> None:
    t_max = y.shape[0]
    # --- Priors ---
    ## Level
    level_smoothing = numpyro.sample(
        "level_smoothing", dist.Beta(concentration1=1, concentration0=1)
    )
    level_init = numpyro.sample("level_init", dist.Normal(loc=0, scale=1))
    ## Noise
    noise = numpyro.sample("noise", dist.HalfNormal(scale=1))
    # --- Transition Function ---
    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,
        )
        mu = previous_level
        pred = numpyro.sample("pred", dist.Normal(loc=mu, scale=noise))
        return level, pred
    # --- Run Scan ---
    with numpyro.handlers.condition(data={"pred": y}):
        _, preds = scan(transition_fn, level_init, jnp.arange(t_max + future))
    # --- Forecast ---
    if future > 0:
        numpyro.deterministic("y_forecast", preds[-future:])
```

We can use different inference methods (MCMC, SVI) and different samplers (NUTS, HMC, etc.). Leverage on ArviZ to analyze the posterior distributions and model diagnostics.

```python
# Example of SVI
guide = AutoNormal(level_model)
optimizer = numpyro.optim.Adam(step_size=0.05)
svi = SVI(level_model, guide, optimizer, loss=Trace_ELBO())
num_steps = 15_000

rng_key, rng_subkey = random.split(key=rng_key)
svi_result = svi.run(rng_subkey, num_steps, y_train)
```

III. HIERARCHICAL MODELS

- Vectorize the model and add hierarchies to the parameters.
- Total flexibility to write custom models.
- Automatic model reparametrization (via SVI).

Scaling Time Series Models with GPUs/TPUs

We can can scale this models to the order of hundreds of thousands or millions of time series on GPU.

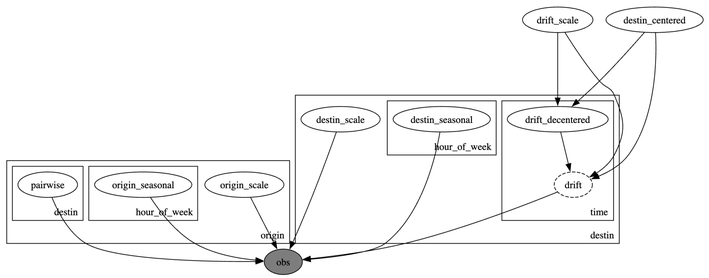

Figure 1: Hierarchical model graphical structure

IV. CENSORING LIKELIHOODS

In demand forecasting, we often face the challenge of censored data where observations are only partially known. A censored distribution occurs when values above (or below) a certain threshold are unobserved or replaced with the threshold value. This is particularly relevant in retail where sales data only captures observed demand when products are in stock, but the true demand remains unobserved during stockouts or when hitting inventory capacity limits. We can use censored likelihoods to model this data. This has proven to be very effective as a component of shelf-optimization.

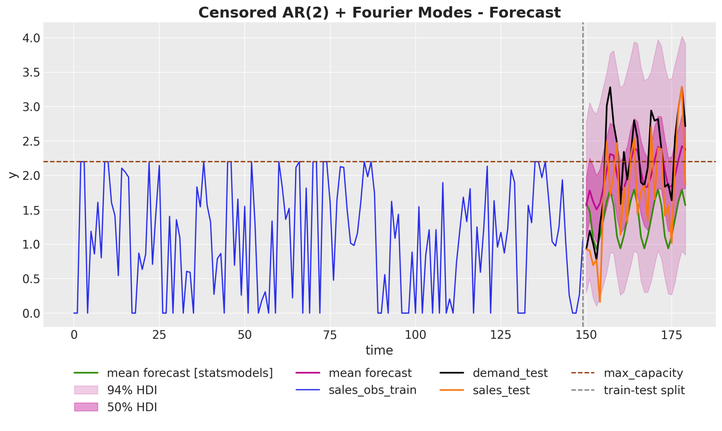

Figure 2: Censored Data

V. DYNAMIC MODELS & CALIBRATION

Probabilistic forecasting models can be further enhanced through parameter calibration using domain knowledge or experimental data. The case study "Electricity Demand Forecast with Prior Calibration" demonstrates how to incorporate external information to improve a dynamic time-series model for electricity demand forecasting. The main idea is to motivate electricity demand as a function of the temperature:

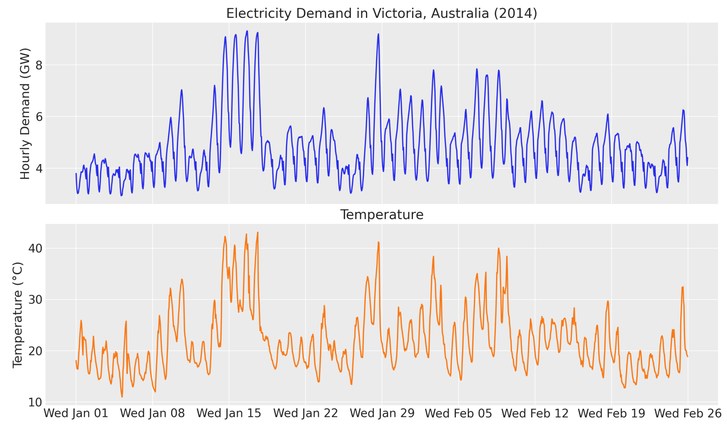

Figure 3: Electricity demand and temperature

The model uses a Hilbert Space Gaussian Process to capture the time-varying relationship between temperature and electricity demand. The key innovation is the use of a prior calibration process to constrain the temperature effect on demand, particularly for extreme temperatures with limited historical data. The calibration works by adding custom likelihoods that effectively "observe" what the parameter values should be in certain regimes. In this concrete example, we use a Gaussian process to model the temperature to electricity relationship while imposing a constrain in this ratio for large temperatures:

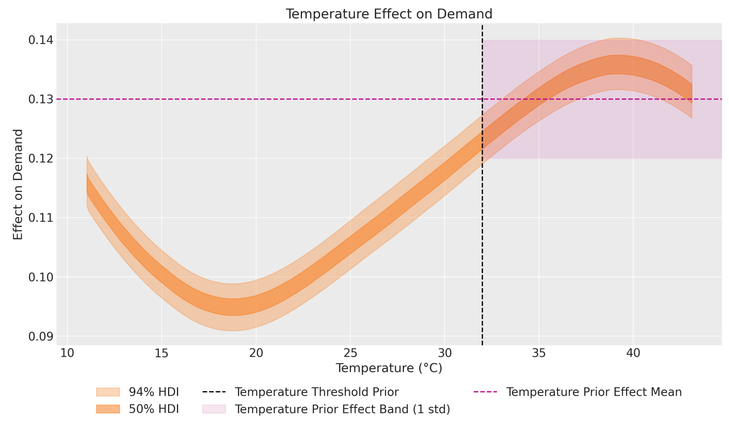

Figure 4: Calibrated Gaussian process dynamic latent variable