Probabilistic Forecasts for Cryptocurrency

Jake Taylor
Department of Statistics
Stanford University
jakee417@stanford.edu

Samuel Wong
Department of Statistics
Stanford University
samwwong@stanford.edu

https://github.com/jakee417/ProbabilisticForecasting

1 Introduction

Forecasting is used in many industrial settings when future decision making is complicated by the inherent uncertainty of a process. Modern examples include forecasting future product demand for a web commerce retailer (1), efficient energy use as part of a smart grid (SG) system (2), or central processing unit (CPU) utilization across a network of virtual routers for a cloud based computer security company (3). Classical statistical analysis of time series data commonly starts with an Autoregressive-Moving-Average (ARMA) model (4) and builds upon this foundation with a variety of generalizations such as ARIMA, SARIMA, ARMAX, NARMAX, or VARMA (5). More recently, approaches involving Deep Neural Networks (DNNs) have been used to learn highly non-linear functions for large amounts of time series data while also incorporating a probabilistic component to measure the forecast's uncertainty (1). We would like to explore the application of these more recent approaches to analyze large amounts of cryptocurrency transaction data.

2 Dataset and Features

Our dataset comes from transaction data present in the blockchains of popular cryptocurrencies. Since these blockchains are all digital public ledgers on a permissionless system, the data is readily available for us to use to perform our forecasting using Bayesian techniques. From each block, we are able to obtain the timestamp (hour, day, month, year), as well as the number of transactions in the block. We convert the timestamp into time series features and use the number of transactions as our target.

Our goal is twofold: we want to detect anomalous transaction volume as well as forecast future transaction volume. We believe that accurate forecasting of these targets will create a strong tool for financial analysis. Our paper presents the models used for the largest cryptocurrency: Bitcoin.

With the growing popularity of dApps on the Ethereum blockchain built on smart contracts, particularly in DeFi with the rise of DEX (UniSwap, SushiSwap, Curve, etc.), there still remains an open question of scalability in this space. We believe our predictions of transaction volume can help guide decisions for new entrants as to whether to stay within the Ethereum blockchain framework and risk higher Gas fees and an overall longer block confirmation time, or to build their product on a separate chain (Solana, Cardano, Polkadot etc.).

3 Modeling Approaches

Consider a time series consisting of $X = \{(t_n, y_n)\}_{n=1}^{N}$ and $Y = \{y_{N+k}\}_{k=1}^{K}$ where X represents N tuples of timestamp t_n and observed value y_n. Analogously, Y represents the K future values we

Stat271: Applied Bayesian Analysis, Spring 2021, Stanford University, CA. (template credit NIPS 2017.)

are interested in forecasting. Our goal is to model future values conditioned on the past:

$$\mathbb{P}(Y|X) = \mathbb{P}(\{y_{N+k}\}_{k=1}^{K}|\{(t_n, y_n)\}_{n=1}^{N}) \tag{1}$$

$$= \mathbb{P}(\vec{y}_{N+1:N+K}|\vec{t}_{1:N}, \vec{y}_{1:N}) \tag{2}$$

Where we are using vector notation $\vec{a}_{b:c}$ to represent a random vector a starting at time index b and ending at time index c. We call (2) the **forecast distribution**.

3.1 Generative Learning

A generative modeling approach starts by using Bayes' Rule:

$$\mathbb{P}(\vec{y}_{N+1:N+K}|\vec{t}_{1:N}, \vec{y}_{1:N}) = \frac{\mathbb{P}(\vec{y}_{1:N+K}, \vec{t}_{1:N})}{\mathbb{P}(\vec{t}_{1:N}, \vec{y}_{1:N})}$$

$$= \frac{\mathbb{P}(\vec{y}_{1:N+K}, \vec{t}_{1:N})}{\int_{y_{N+1}} \dots \int_{y_{N+K}} \mathbb{P}(\vec{t}_{1:N}, \vec{y}_{1:N+K}) dy_{N+1} \dots dy_{N+K}}$$

Focusing on just the numerator, we can rewrite this by iterating the Chain Rule of Probability:

$$\mathbb{P}(\vec{y}_{1:N+K}, \vec{t}_{1:N}) = \mathbb{P}(y_{N+K}|\vec{y}_{1:N+K-1}, \vec{t}_{1:N})\mathbb{P}(\vec{y}_{1:N+K-1}, \vec{t}_{1:N})$$

$$\vdots$$

$$= \mathbb{P}(y_{N+K}|\vec{y}_{1:N+K-1}, \vec{t}_{1:N})\mathbb{P}(y_{N+K-1}|\vec{y}_{1:N+K-2}, \vec{t}_{1:N}) \dots \mathbb{P}(y_1|\vec{t}_{1:N})\mathbb{P}(\vec{t}_{1:N})$$

For ease of exposition, consider a simplified case when y_n can take on d discrete values at each timestep, $y_n \in \{1, \dots, d\}$ and y_n does not depend on the time covariate $\vec{t}_{1:N}$, i.e.:

$$\mathbb{P}(y_n|\vec{y}_{1:n-1}, \vec{t}_{1:N}) = \mathbb{P}(y_n|\vec{y}_{1:n-1})$$

Then, to fully parameterize this probability distribution with a set of parameters θ, we would still need $|\theta| = O(d^{N+K})$ parameters. For a sufficiently long time series, this is both intractable and imposes overly restrictive assumptions. We could further simplify this generative model with a p-order Markov assumption in which y_n only depends on the p previous values $\vec{y}_{n-p:n}$ or use an approximate sampling technique to estimate this distribution (HMC, Variational Inference, etc.), but we instead switch to a discriminative modeling approach to sidestep having to model the joint probability $\mathbb{P}(\vec{y}_{1:N+K}, \vec{t}_{1:N})$.

3.2 Discriminative Learning

Consider modeling $\mathbb{P}(\vec{y}_{N+1:N+K}|\vec{t}_{1:N}, \vec{y}_{1:N})$ directly in a manner similar to DeepAR found in (1):

$$\mathbb{P}(\vec{y}_{N+1:N+K}|\vec{t}_{1:N}, \vec{y}_{1:N}) \approx Q_{\boldsymbol{\theta}}(\vec{y}_{N+1:N+K}|\vec{y}_{N-L:N}, \vec{t}_{N-L:N}) \tag{3}$$

$$\stackrel{\perp}{=} \prod_{l=1}^{K} Q_{\boldsymbol{\theta}}(y_{N+l}|\vec{y}_{N-L:N}, \vec{t}_{N-L:N}) \tag{4}$$

$$\approx \mathcal{L}_{N+1}\big(y_{N+1}; g_{N+1}(\vec{y}_{N-L:N}, \vec{t}_{N-L:N}; \vec{\theta}_{N-L:N+1})\big) \tag{5}$$

$$\cdot \prod_{l=2}^{K} \mathcal{L}_{N+l}\big(y_{N+l}; g_{N+l}(g_{N+l-1}(\vec{\eta}_{N+l-2}; \vec{\theta}_{N+l-1}); \vec{\theta}_{N+l})\big) \tag{6}$$

Where in (3) we assume that the forecast distribution can be modeled by some approximation $Q_{\boldsymbol{\theta}}$ involving the last L values before time $N+1$, (4) we assume conditional independence of $\vec{y}_{N+1:N+K}$ given the L past values $\vec{y}_{N-L:N}, \vec{t}_{N-L:N}$, and (5,6) we assume that our approximation $Q_{\boldsymbol{\theta}}$ is a simple likelihood model $\mathcal{L}$ (i.e. $\mathcal{N}(\mu, \sigma^2)$ or a mixture of univariate distributions) which are parameterized by some function $g_{N+l}(\cdot)$ with parameters $\vec{\theta}_{N+l}$. The assumption in (4) that $\vec{y}_{N+1:N+K}$ is conditionally dependent given its past is clearly not realistic since $\vec{y}_{N+1:N+K}$ is time series data that is likely to have a rich set of time dependencies. To mitigate this unrealistic assumption, we nest the previous timestep's likelihood parameters $\eta_{N+l-1} = g_{N+l-1}(\vec{\eta}_{N+l-2}; \vec{\theta}_{N+l-1})$ for $\mathcal{L}_{N+l-1}(\eta_{N+l-1})$ into

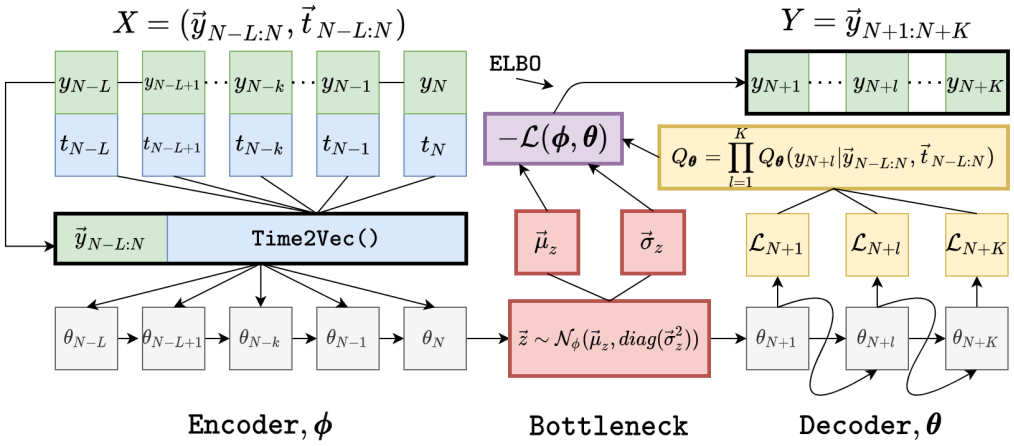

Figure 1: Model Architecture. An Encoder/Decoder with recurrent LSTM cells connects an observed time series to a forecast. `Time2Vec` and a Bottleneck Variational Layer are used to increase model capacity. A model of this form allows for direct sampling from an approximation of the forecast distribution.

the current parameters $\eta_{N+l} = g_{N+l}(g_{N+l-1}(\vec{\eta}_{N+l-2}; \theta_{N+l-1}); \vec{\theta}_{N+l})$ for $\mathcal{L}_{N+l}(\eta_{N+l})$. The aim is to implicitly add dependency across parameters to the otherwise conditionally independent $Q_{\boldsymbol{\theta}}$. We hypothesize that when $g_{N+l}(\cdot)$ is taken to be a sufficiently complex Neural Network (i.e. a recurrent set of Long Short Term Memory (LSTM) cells), $Q_{\boldsymbol{\theta}}$ is able to approximate the dependencies in $\mathbb{P}(\vec{y}_{N+1:N+K}|\vec{t}_{1:N}, \vec{y}_{1:N})$ in a realistic manner. The final architecture (with additions from the next section) is shown in Figure 1.

4 Further Modeling Considerations

We now discuss other modeling considerations that are accessory to our main method, but necessary additions for modeling complex cryptocurrency datasets.

4.1 Modeling $\vec{t}_{N-L:N}$'s Periodicity with `Time2Vec` Embeddings

Since our data may exhibit periodicity, we use `Time2Vec` embeddings from (6) to model any periodic nature in the time series. In order to facilitate approximating functions with non-periodic patterns and help with generalization, we include both a linear projection of time as well as a sinusodial function of that representation.

The `Time2Vec` at a particular point τ, which is a vector of size $k+1$, is defined as follows:

$$t2v(\tau)[i] = \begin{cases} w_i\tau + \varphi_i, & \text{if } i = 0 \\ \sin(w_i\tau + \varphi_i), & \text{if } 1 \leq i \leq k \end{cases} \tag{7}$$

We chose to use sine as the periodic activation function, although other periodic activation functions can also suffice. The benefit of incorporating this activation function is that we are able to capture periodic behaviors without the need for explicit feature engineering. In the model where τ represents one hour, the sine function where $w = \frac{2\pi}{24}$ repeats every 24 hours, and thus we are able to model daily patterns.

This desirable property of `Time2Vec` leads it to be invariant to time rescaling, thus applicable to any blockchain. For example, even though the Bitcoin network produces a block every 10 minutes on average while Ethereum produces a block every 15 seconds on average, the `Time2Vec` embedding can be learned for either blockchain.

4.2 Adding a Bottleneck Variational Layer

Given our encoder-decoder scheme, a natural extension is to include a variational layer to increase our model's capacity to learn. Consider a latent random vector $\vec{z} \in \mathbb{R}^P$:

$$\mathbb{P}(\vec{y}_{N+1:N+K}|\vec{t}_{1:N}, \vec{y}_{1:N}) = \int \mathbb{P}(\vec{y}_{N+1:N+K}|\vec{t}_{1:N}, \vec{y}_{1:N}, \vec{z})p(\vec{z})d\vec{z} \tag{8}$$

$$\approx \int \prod_{l=1}^{K} Q_{\vec{\theta}}(y_{N+l}|\vec{y}_{N-L:N}, \vec{t}_{N-L:N}, \vec{z})p(\vec{z})d\vec{z} \tag{9}$$

$$Q_{\boldsymbol{\theta}}(y_{N+1}|\vec{y}_{N-L:N}, \vec{t}_{N-L:N}, \vec{z}) = \mathcal{L}_{N+1}\big(y_{N+1}; g_{N+1}(\vec{y}_{N-L:N}, \vec{t}_{N-L:N}, \vec{z}; \vec{\theta}_{N-L:N+1})\big) \tag{10}$$

$$\mathbb{P}(\vec{z}) = \mathcal{N}(\vec{z}; \vec{0}, I) \tag{11}$$

And $Q_{\boldsymbol{\theta}}(y_{N+l}|\vec{y}_{N-L:N}, \vec{t}_{N-L:N}, \vec{z}), l > 1$ remains the same as in (6) but with the additional nested dependency on $\vec{z}$. We now turn to variational expectation-maximization (vEM) to learn our latent and model parameters given in (9). We first use an inference network:

$$g_z : \mathcal{D} = (\vec{y}_{1:N+K}, \vec{t}_{1:N}) \to \boldsymbol{\phi} = (\mu_z, \sigma_z^2)$$

To help amortize the cost of approximating the posterior of the latent vector $\mathbb{P}(\vec{z}|\vec{y}_{N-L:N}, \vec{t}_{N-L:N})$ in vEM's E-step:

$$\mathbb{P}(\vec{z}|\vec{y}_{N-L:N}, \vec{t}_{N-L:N}) \approx q(\vec{z}; \vec{\lambda}) = q(\vec{z}; g_z(\vec{y}_{N-L:N}, \vec{t}_{N-L:N}, \boldsymbol{\phi})) = \mathcal{N}\big(\vec{z}; \vec{\mu}_z, \mathrm{diag}(\vec{\sigma}_z^2)\big)$$

Where we have assumed an *approximate*, spherical, p-dimensional $\mathcal{N}(\cdot)$ posterior that is conjugate with the prior on $\vec{z}$ and absorbed $\vec{\theta}_{N-L:N}$ into $\boldsymbol{\phi}$. All of these modifications amount to performing stochastic gradient descent on the negative Evidence Lower Bound (ELBO) given by:

$$-\mathcal{L}(\boldsymbol{\phi}, \boldsymbol{\theta}|\mathcal{D}) \approx -\frac{\beta}{2}\sum_{p=1}^{P}\big(1 + \log(\sigma_p)^2 - (\mu_p)^2 - (\sigma_p)^2\big) \tag{12}$$

$$-\frac{1}{S}\sum_{s=1}^{S}\sum_{l=1}^{K}\log Q_{\boldsymbol{\theta}}(y_{N+l}|\vec{y}_{N-L:N}, \vec{t}_{N-L:N}, \vec{z}^{(s)}), \beta \geq 1 \tag{13}$$

$$\vec{z}^{(s)} = \vec{\mu}_z + \vec{\sigma}_z \odot \vec{\epsilon}^{(s)}, \vec{\epsilon}^{(s)} \sim \mathcal{N}(\vec{0}, I) \tag{14}$$

Using Monte Carlo estimates of the gradient in lines (12, 13), the reparameterization trick from (7) in line (14), and a β-hyperparameter for line (12) that encourages more disentangled latent representations of $\vec{z}$ motivated in (8). This is in contrast to minimizing just the original negative log-likelihood: $-\log\sum_{l=1}^{K}\mathcal{L}_{N+l}(\cdot)$ which results from (5, 6).

4.3 Approximating count data $\vec{y}_{N+1:N+K}$ with a Location-Scale Family likelihood $\mathcal{L}$

Since we are primarily interested in modeling the number of cryptocurrency transactions which is a non-negative, count-type data, a strong candidate for the likelihood model $\mathcal{L}$ would be the Poisson distribution or the more robust Negative Binomial distribution. One complication with using a distribution with strictly non-negative support over integer values is the question of standardizing for learning parameters inside a Neural Network framework. This is exacerbated by the typical range of our datasets where $y \in \{10^2, \ldots, 10^6\}$ if the original time series has been resampled using a `sum()` aggregation rule. One way to attain a standardized dataset $y' = \frac{y-\mu_y}{\sigma_y}$ and likelihood model $\mathcal{L}$ that are amenable to training with a Neural Network is finding a real-valued approximation to these count-type distributions. Using a well known fact from the Central Limit Theorem:

$$X \sim \text{Poisson}(\lambda) \implies X \approx \mathcal{N}(\mu = \lambda, \sigma = \sqrt{\lambda}) : \lambda > 100$$

Where the approximation improves as λ increases. This gives us a theoretical justification for letting the likelihood model $\mathcal{L} = \mathcal{N}(\mu, \sigma)$. We further relax this approximation by also considering other Location-Scale distributions such as the Student's t, Laplace, Uniform, and even mixture models all four of these distributions.

5 Comparison to DeepAR

5.1 Sampling from the Forecast Distribution $\mathbb{P}(\vec{y}_{N+1:N+K}|\vec{t}_{1:N}, \vec{y}_{1:N})$

Our architecture is similar to DeepAR as presented in (1) but with some important modifications. Firstly, DeepAR uses all N training points $\vec{y}_{1:N}$ as opposed to a lookback window $\vec{y}_{N-L,N}$ of size L, to forecast the K future values $\vec{y}_{N+1:N+K}$. Training on all $\vec{y}_{1:N}$ samples enables DeepAR to learn deep relationships in the training data which significantly reduces the complexity in making their forecasts. More concretely, once the DeepAR distribution is trained on all N time points:

$$\mathbb{P}(\vec{y}_{N+1:N+K}|\vec{t}_{1:N}, \vec{y}_{1:N}) \approx Q'_{\boldsymbol{\theta}}(\vec{y}_{N+1:N+K}|\vec{y}_{1:N}, \vec{t}_{1:N+K})$$

Forecasting starts with ancestral sampling from the learned likelihood model:

$$\tilde{y}_{N+1} \sim \mathcal{L}'_{N+1}\big(y_{N+1}; g_{N+1}(\vec{y}_{1:N}, \vec{t}_{1:N}; \vec{\theta}_{N+1})\big)$$

And then using $\tilde{y}_{N+1}$ to generate the likelihood $\mathcal{L}'_{N+2}$ at time $N+2$, where the process is iterated until K total samples have been drawn. Repeating this process (generating batches of K samples each time) represents an approximate sample from $\mathbb{P}(\vec{y}_{N+1:N+K}|\vec{t}_{1:N+K}, \vec{y}_{1:N})$. One shortcoming of this approach is that subsequent samples will inherit any error of their parent nodes. Thus, for sufficiently large values of K, it is common to see a compounding increase in sample path variability.

In contrast to this approach, we estimate $\mathbb{P}(\vec{y}_{N+1:N+K}|\vec{t}_{1:N}, \vec{y}_{1:N})$ in a more direct fashion. Let $X^{(i)} = \{(t_n, y_n)\}_{n=1}^{N}$ be a training example and $Y^{(i)} = \{y_{N+k}\}_{k=1}^{K}$ be the label where $X^{(i)}$ denotes batch i from our dataset $\mathcal{D}$. At forecast time, we sample directly from:

$$Q_{\boldsymbol{\theta}}(\vec{y}_{N+1:N+K}|\vec{y}_{N-L:N}, \vec{t}_{N-L:N}) \overset{\perp}{=} \prod_{l=1}^{K} Q_{\boldsymbol{\theta}}(y_{N+l}|\vec{y}_{N-L:N}, \vec{t}_{N-L:N})$$

As this has been explicitly trained on our $(X^{(i)}, Y^{(i)})$ training/label pairs. Due to the fact that we have assumed conditional independence given $\vec{y}_{N-L:N}, \vec{t}_{N-L:N}$, there is no need to perform ancestral sampling to get an approximate sample from $\mathbb{P}(\vec{y}_{N+1:N+K}|\vec{t}_{1:N}, \vec{y}_{1:N})$ which sidesteps any compounding error from performing such a procedure.

5.2 Time Covariates $\vec{t}_{1:N+K}$ vs $\vec{t}_{N-L:N}$

Another important difference is that DeepAR assumes that all time covariates, $\vec{t}_{1:N+K}$, are available at both train *and* forecast time, where we just depend on the L most recent covariate values $\vec{t}_{N-L:N}$ for *just* training. One example of such a time covariate could be timestamps i.e. 2021-05-25T04:31:45+00:00 which could possibly include several categories (Hour of Day, Day of Week, Month, etc) that are highly correlated with $\vec{y}_{1:N+K}$. We noted minimal impact to our forecasts just using the reduced set $\vec{t}_{N-L:N}$ *except* for the case when a Holiday or otherwise non-regular but cyclic event occurs inside the forecast window (i.e. Super Bowl, Thanksgiving, etc.). In a more general time series setting, it is very important to contextualize the model with the forecast window's absolute time to account for such non-seasonal and non-regular behavior not captured in the size L lookback window. Modifying our approach to incorporate the full set of time covariates $\vec{t}_{1:N+K}$ is left for further work since it has minimal connections with Bayesian Analysis.

6 Model Checking

6.1 Posterior Forecast Checks

Once $Q_{\boldsymbol{\theta}}(\vec{y}_{N+1:N+K}|\vec{y}_{N-L:N}, \vec{t}_{N-L:N})$ is learned, a single summary statistic describing how well $Q_{\boldsymbol{\theta}}$ fits (or overfits) the data $\mathcal{D}$ is needed to make comparisons across model hyperparameters. At a minimum, we hope that given a set of training examples and labels $\mathcal{D} = (X^{(i)}, Y^{(i)})$ the best approximation $Q^*_{\boldsymbol{\theta}}$ is able to faithfully produce $Y^{(i)}$ from $X^{(i)}$. One way to approximate the difference between $Q_{\boldsymbol{\theta}}$ and $Q^*_{\boldsymbol{\theta}}$ would be to:

- Generate S samples from our forecast: $\tilde{Y}^{(i)} = \{\tilde{Y}^{(i,s)} \sim Q_{\boldsymbol{\theta}}(Y^{(i)}|X^{(i)})\}_{s=1}^{S}$

- From $\tilde{Y}^{(i)}$, compute test quantities $T(\tilde{Y}^{(i)}, \mathcal{T})$ over a set of J thresholds $\mathcal{T} : |\mathcal{T}| = J$
- Compare $T(\tilde{Y}^{(i)}, \mathcal{T}_j)$ against $T(Y^{(i)}, \mathcal{T}_j)$ $\forall j \in J$. Store the J discrepancies for $Y^{(i)}$.
- Repeat $\forall i \in \mathcal{D}$ and aggregate the discrepancies into $\text{Err}_{\boldsymbol{\theta}} : |\text{Err}_{\boldsymbol{\theta}}| = J$.
- Compute $T(\text{Err}_{\boldsymbol{\theta}}, \text{Err}^*_{\boldsymbol{\theta}})$, where $\text{Err}^*_{\boldsymbol{\theta}}$ represents the J idealized discrepancies from $Q^*_{\boldsymbol{\theta}}$.

In practice, we compute this over batches of data using vectorized operations for better performance. This is very straightforward using Tensorflow Probability's batch semantics (9).

6.2 Posterior Forecast Check Example

Consider an example where $\mathcal{T} = \{10, 20, \ldots, 90\}$ representing credible intervals for our forecasts and $T(\tilde{Y}^{(i)}, \mathcal{T}_j)$ is the percentile of $\tilde{Y}^{(i)}$ that corresponds to a credible interval $\mathcal{T}_j$. We then compute $T(Y^{(i)}, \mathcal{T}_j)$, the number of times that $Y^{(i)}$ also fell outside the percentiles $T(\tilde{Y}^{(i)}, \mathcal{T}_j)$ produced at each of the $\mathcal{T}$ levels. We summarize this in Figure 2 which shows $\mathcal{T}$ on the x-axis and the comparison of $T(\tilde{Y}^{(i)}, \mathcal{T}_j)$ and $T(Y^{(i)}, \mathcal{T}_j)$ on the y-axis:

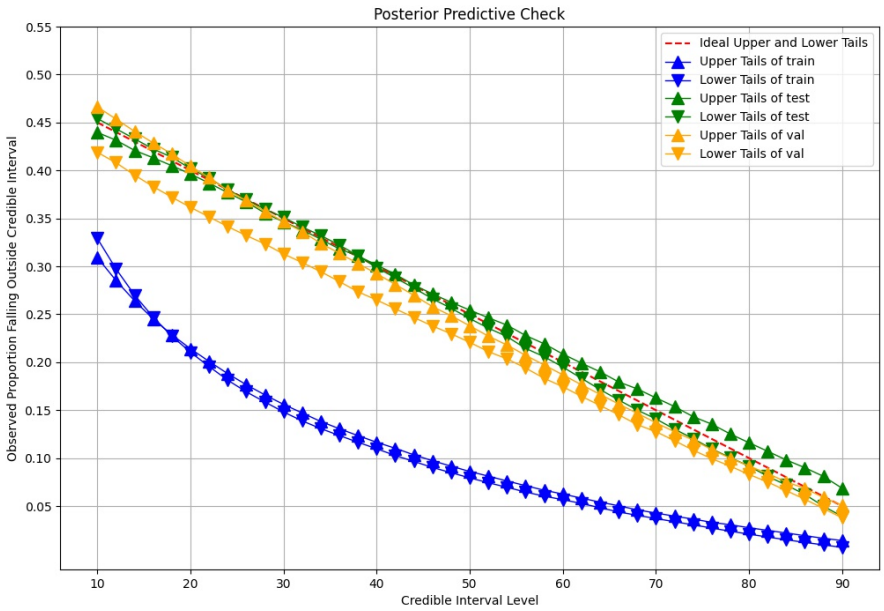

Figure 2: Upper tails (upwards triangles) and lower tails (downwards triangles) in a model check for a train (blue) validation(yellow) and test (green) set.

The dashed red line shows what would happen in an ideal setting under a perfect approximation $Q^*_{\boldsymbol{\theta}}$. For example, a perfect distribution would say that 25% of the observed $Y^{(i)}$'s fell above and below a credible interval containing 50% of the simulated forecasts, $\tilde{Y}^{(i)}$. For Figure 2, we tuned the parameters of $Q_{\boldsymbol{\theta}}$ to avoid overfitting to the train set for good generalization on the test set. This is evident in how the upper and lower tails do not tightly fit the training data for most credible interval levels, but the validation and test set are quite similar to what $Q^*_{\boldsymbol{\theta}}$ would produce at all credible interval levels. We finally condense this plot into a single summary statistic by computing:

$$T(\text{Err}_{\boldsymbol{\theta}}, \text{Err}^*_{\boldsymbol{\theta}}) = \|Err_{\boldsymbol{\theta}} - Err^*_{\boldsymbol{\theta}}\|_1 \tag{15}$$

Which has the interpretation of an approximation to the absolute area under the curve from $\text{Err}_{\boldsymbol{\theta}}$ to $\text{Err}^*_{\boldsymbol{\theta}}$ when $\mathcal{T}$ is a fine enough grid of credible interval levels. $T(\text{Err}_{\boldsymbol{\theta}}, \text{Err}^*_{\boldsymbol{\theta}})$ is then our single summary of how close $Q_{\boldsymbol{\theta}}$ is to $Q^*_{\boldsymbol{\theta}}$. The goal is to make this number as low as possible (≈ 0) on the held-out dataset. We take this approach later for our results in Section 7.

6.3 Forecast Likelihood

Similar to a predictive likelihood, we can also measure how likely the forecast labels are under our supposed model $Q_{\boldsymbol{\theta}}$. Since our model is already setup to compute this to train it's weights, we need only to compute one forward pass of $Q_{\boldsymbol{\theta}}$ after training has concluded for the train, validation, and test sets.

6.4 Forecasting *and* Anomaly Detection with $\mathbb{P}(\vec{y}_{N+1:N+K}|\vec{t}_{1:N}, \vec{y}_{1:N})$

The goal for this model is twofold: to *forecast* the number of transactions in the future, as well as *detect anomalies*. Our modeling strategy provides us both results simultaneously by predicting a distribution, rather than a point estimate. As previously described, the model will accept $(\vec{t}_{1:N}, \vec{y}_{1:N})$ as input and return an approximation to $\mathbb{P}(\vec{y}_{N+1:N+K}|\vec{t}_{1:N}, \vec{y}_{1:N})$. As described in the previous sections, obtaining Monte Carlo estimations of the form $T(\tilde{Y}^{(i)}, \mathcal{T}_j)$ allows us to approximate a wide variety of functionals of interest.

For forecasting, combining the approaches of Sections 5.1 and 6 can give realistic approximations to queries of interest about the forecast distribution. For anomaly detection, we shift our forecast indices, $N+1:N+K$ to overlap with our training indices $N-L:N$ as in a Variational Autoencoder (VAE) described in (10). In this setting, $\vec{y}_{N-L:N}$ have already occurred and we want to detect a subset of points that are anomalous rather than forecast new points. We accomplish this by seeing if any of the observed values lie within a certain credible interval generated from $T(\tilde{Y}^{(i)}, \alpha)$ supposing that $Q_{\boldsymbol{\theta}} \approx Q^*_{\boldsymbol{\theta}}$. For a computed value of α from $Q_{\boldsymbol{\theta}}$, define $T(Y^{(i)}, \alpha)$ as:

$$T(Y^{(i)}, \alpha) = \{y_{N-l} > \alpha\} \cup \{y_{N-l} < (1-\alpha)\}, 0 < l < L \tag{16}$$

then $T(Y^{(i)}, \alpha)$ is a summary of all anomalies that have occurred in our window of size L. We show an example of this in Figure 3 where the red dots are anomalies that are outside a 90% credible interval generated from our model.

7 Results for various Likelihood Models $\mathcal{L}$

We now conduct experiments to find the most appropriate likelihood model, $\mathcal{L}$. For the following experiments, we forecast 24 hours into the future ($K = 24$) using a lookback window of 96 hours ($L = 96$) and detect anomalies using $K = L = 24$ as described in Section 6. We fit our model to the `bitcoin` dataset which has a total of $72,894$ training points. We choose not to resample our data and train in batches of 256 datapoints. We use 70% of the dataset for training, 20% for validation, and 10% for test. We train up to 40 epochs with patience of 2 and Reduce-on-Plateau learning rate scheduling. Finally, we set the hidden units to 32, `Time2Vec` embedding to 8, Latent Dimension to 2, β to 1, kernel regularization strength λ to 0.01, and compare performance across likelihood models. We consider the following likelihood models:

- Poisson Approx. ($\mathcal{N}(\lambda, \sqrt{|\lambda|})$) Model: K learnable univariate Gaussian distributions.
- Gaussian ($\mathcal{N}(\mu, \sigma^2)$) Model: K learnable univariate Gaussian distributions.
- Student's t-distribution ($t(\mu, \tau^2, \nu)$) Model: K learnable univariate Student's t-distributions.
- Laplace distribution (`Laplace`(μ, b)) Model: K learnable univariate Laplace distributions.
- Location-Scale Mixture (`Mixture`) Model: Consists of a learnable mixture of K, learnable $\mathcal{N}(\mu, \sigma^2)$, $t(\mu, \sigma, \nu)$, and Laplace distributions.
- Hidden Markov (`HMM`) Model: K-dimensional joint distribution consisting of a learnable mixture of 50, learnable $\mathcal{N}(\mu, \sigma^2)$ emissions with half-normal priors that are time dependent.

We summarize the performance in Tables 1 & 2 measured with $T(\text{Err}_{\boldsymbol{\theta}}, \text{Err}^*_{\boldsymbol{\theta}})$ from Section 6.

8 Discussion

From Tables 1 & 2, we see that the learned distributions fit the held-out test set with minimal hyperparameter tuning. In particular, $\mathcal{N}(\mu, \sigma^2)$ appears to perform well in forecasting which is

Table 1: Performance in Forecast Task

$\mathcal{L}$	train	val	test
$\mathcal{N}(\lambda, \sqrt{\lvert\lambda\rvert})$	10.48	2.06	2.75
$\mathcal{N}(\mu, \sigma^2)$	3.05	3.34	2.32
$t(\mu, \tau^2, \nu)$	1.84	4.62	5.96
$\texttt{Laplace}(\mu, b)$	5.04	7.44	7.91
`Mixture`	5.51	4.42	3.23
`HMM`	13.53	10.39	7.98

Table 2: Performance in Anomaly Task

$\mathcal{L}$	train	val	test
$\mathcal{N}(\lambda, \sqrt{\lvert\lambda\rvert})$	11.8	1.52	0.55
$\mathcal{N}(\mu, \sigma^2)$	2.36	1.66	1.62
$t(\mu, \tau^2, \nu)$	0.63	3.79	4.42
$\texttt{Laplace}(\mu, b)$	4.33	3.56	3.40
`Mixture`	5.38	3.03	3.53
`HMM`	13.79	4.02	3.81

shown in Figure 3. In 2. of Figure 3, we see that μ even learns a periodicity that is consistent with Day-of-Week (DOW) behavior. Despite this, there are still some points that fall outside the 90% credible interval (more apparent in 1.) which we conclude to be anomalous activity. The most notable instances occur in mid-2015, late-2015, late-2017, and late-2018. Upon further investigation, we see that these were exactly the times at which the Bitcoin network was affected by Distributed Denial of Service (DDoS) attacks (by hackers demanding a ransom) shown by 3.. In addition, our model is robust towards concept drift by learning the σ^2 parameter. For example, in the period between December 2017 and February 2018, the Bitcoin price plummeted from almost $20,000 to $6,000. During this period there was a large number of panic sellers, thus increasing the trading volume. After this unstable period, many investors dropped out of the cryptocurrency market and thus trading volume decreased. As shown in Figure 3, the model is able to adjust to this new (lower volume) trend and classify data points in this range as not anomalous. For future considerations, we want to use this modeling framework on other blockchains as well. Given the success on the `bitcoin` dataset, we believe that this tool could help developers better manage these architectures in a production setting. From a modeling perspective, we would incorporate time features in the forecast window and conduct a more rigorous hyperparameter search to better tune our model.

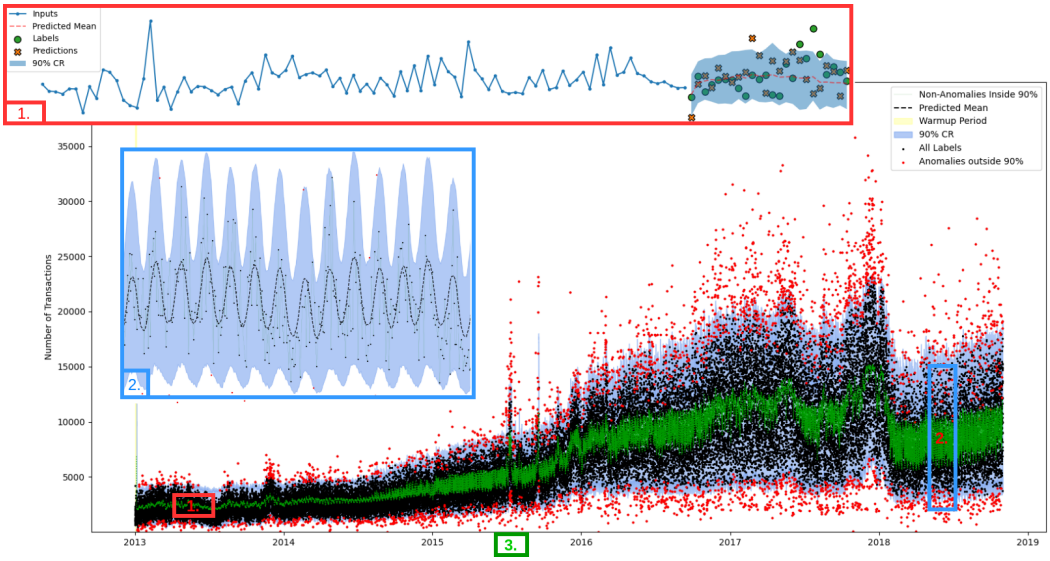

Figure 3: Forecast Example for $\mathcal{L} = \mathcal{N}(\mu, \sigma^2)$. The figure shows a rolling forecast over a 6-year span for Bitcoin. Black points are non-anomalous (inside 90%), red points anomalous (outside 90%), and the green line and blue shading shows the forecast's mean and 90% credible interval. 1. shows an example of one training batch ($L = 96$, $K = 24$) with green labels and orange X's for predictions. 2. shows learned periodicity over a two week period. 3. shows a detected DDoS attack as described above.

References

[1] D. Salinas, V. Flunkert, and J. Gasthaus, "Deepar: Probabilistic forecasting with autoregressive recurrent networks," 2019.

[2] D. Kaur, S. N. Islam, M. A. Mahmud, and Z. Dong, "Energy forecasting in smart grid systems: A review of the state-of-the-art techniques," *CoRR*, vol. abs/2011.12598, 2020.

[3] S. R. Karingula, N. Ramanan, R. Tahsambi, M. Amjadi, D. Jung, R. Si, C. Thimmisetty, and C. N. C. J. au2, "Boosted embeddings for time series forecasting," 2021.

[4] P. Whittle, "Hypothesis testing in time series analysis," 1951.

[5] R. H. Shumway and D. S. Stoffer, *Time Series Analysis and Its Applications With R Examples*. Springer, 2006.

[6] S. M. Kazemi, R. Goel, S. Eghbali, J. Ramanan, J. Sahota, S. Thakur, S. Wu, C. Smyth, P. Poupart, and M. Brubaker, "Time2vec: Learning a vector representation of time," 2019.

[7] D. P. Kingma and M. Welling, "Auto-encoding variational bayes," 2014.

[8] C. P. Burgess, I. Higgins, A. Pal, L. Matthey, N. Watters, G. Desjardins, and A. Lerchner, "Understanding disentangling in β-vae," 2018.

[9] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015. Software available from tensorflow.org.

[10] D. P. Kingma and M. Welling, "An introduction to variational autoencoders," *CoRR*, vol. abs/1906.02691, 2019.

[11] J. V. Dillon, I. Langmore, D. Tran, E. Brevdo, S. Vasudevan, D. Moore, B. Patton, A. Alemi, M. Hoffman, and R. A. Saurous, "Tensorflow distributions," 2017.

[12] P. Malhotra, A. Ramakrishnan, G. Anand, L. Vig, P. Agarwal, and G. Shroff, "Lstm-based encoder-decoder for multi-sensor anomaly detection," *CoRR*, vol. abs/1607.00148, 2016.