Time Series Forecasting of Network Access to Educational Services

Abhimanyu Bangaru

April 2024

Contents

1 My Project Idea and Background Information

1.1 Goal

My primary interest is understanding how probabilistic graphical models can be applied to real-world data to model patterns, particularly in networks. The purpose of this project is to learn, and an outcome in which I can enhance my knowledge of the AI field, particularly models to represent time series data, and I can implement a model that predicts outcomes that would mark a satisfactory project. To motivate the goal of this project, I decided to center this project around access to educational services. So, I chose to test the network connection to Khan Academy, Masterclass, YouTube, and IXL.

1.2 Project Idea

I have explored the four following approaches to forecast network ping data: Markov Chains, Bayesian Autoregressive models, Transformers(Autoformer), and Long Short-Term Memory(LSTM). While I was not able to create a working implementation of an Autoformer for time series use, I was able to generate output for the other three models. I would like to note that due to the nature of the project, the information here was written before we talked about Autoregressive models in class, and dives a bit deeper into that area to apply it to network data forecasting.

1.3 Data used

In this exploration, all my training data for each model stemmed from network ping data I collected from 2024-3-19-00:00:03 to 2024-03-29 23:59:54. Due to the use of a transition matrix in my Markov Chain and the continuous ping (ms) values the round-trip-time can time, I discretized the ping times into different states, each of which accounted from 5-10 ms. I further discretized the time into buckets, each representing a minute interval. For the other models, I discretized the time into intervals of 30 seconds, and for each interval of 30 seconds, I rounded the average ping time to the nearest whole integer.

1.4 Background on Models Chosen

The first PGM I explored was possibly one of the simple ones: Markov Chain. The first-order MC operates under the principle of "memorylessness," formally known as the Markov property. This means the probability of transitioning to the next state depends only on the current state and not on the sequence of events that preceded it. However, we can extend the idea of the Markov Chain into multiple "orders", so if we have a 1000-order Markov Chain, each state relies on the 1000 states prior. For time series analysis, this model is particularly useful because it allows for the prediction of future states of a network (like traffic load or availability) based solely on a subset of the states, simplifying the analysis of complex temporal data.

Then, I chose to move more into the Bayesian approach. Bayesian AR models apply Bayesian inference to predict future values based on observations from the time series data. I chose to explore these next due to their relative ease and integration with class material. Unlike the standard Bayesian Networks which capture complex dependencies across variables, Bayesian AR models focus on relationships where data points occur during different times. This allows them to estimate how current and future values depend on previous time points. These models incorporate prior knowledge and quantify uncertainty in predictions, offering us a framework for updating beliefs as new data arrives. They are especially valuable in fields where accurate prediction of future trends is critical.

After experimenting with AR models, I sought an alternative which accounted for seasonality, thus I transitioned to transformers. Transformers are a bit more complicated and quite possibly a little outside the scope of the class. They consist of two main components: an encoder and a decoder, each comprising

multiple layers of self-attention and fully connected neural networks(which we have briefly covered in 391). The encoder processes the entire input sequence simultaneously and encodes it into a high-dimensional space, capturing the contextual relationships between all elements. The decoder then generates the output sequence, one element at a time, using the encoded information along with its self-attention mechanism to maintain coherence and relevance to the input. One of the resources I have identified covers its application to time-series analysis. Time series data is sequential and can benefit from models that understand the order and context of data points. Transformers treat time steps as sequence elements, similar to words in a sentence, allowing them to analyze the entire sequence of data points collectively rather than individually or in small, isolated chunks. This sequential form of analysis could be very helpful in the inference of network ping. I chose to explore Autoformer due to its promise of accurate forecasting.

Finally, I used an LSTM. Long Short-Term Memory (LSTM) networks are a form of Recurrent Neural Networks (RNNs). Their unique architecture which allows them to learn long-term dependencies and patterns in time-series data that simpler models might miss is what allows for LSTMs to be an interesting choice for sequence prediction problems. The LSTM's memory cell can maintain information in memory for long periods. I chose the explore LSTM networks after wanting an improvement over the Markov Chains. The information entering and leaving the cell is controlled by a set of gates, and the cell's state is updated and transferred from one-time step to the next. The gating mechanism plays a crucial role in enabling long-term dependencies to be captured by LSTMs, which makes them effective for a variety of intricate sequence tasks like language modeling, time-series forecasting, and speech recognition.

2 Forecasting Using Markov Chains

Throughout the course of this implementation, I solidfied my conceptual understanding gained during lecture.

2.1 Defining the State Spaces and Loading Data

The first step is to define the state spaces. An important choice I made for this project is to quantize the states. Network ping can be one of numerous specific times, and as such any project would require much data in order to come up with precise ping forecasts. Due to the relative data-sparsity of this project, For this project, to come up with the most interesting results – in other words, narrow enough results – without making it too complex, I have come up with the following states:

State 0: 0 - 10 ms, State 1: 11-15 ms, State 2: 16-20ms, State 3: 21-25ms, State 4: 26 - 30 ms, State 5: 31 - 40 ms, State 6: 41 - 50 ms, State 7: 51 - 60 ms, State 8: 61 - 70 ms, State 9: 71 - 80 ms, State 10: 81 - 90 ms, State 11: 91 - 100 ms, State 12: $>$ 100 ms

The next step in the implementation is to is the convert a sequence of historical data into a sequence of our defined states. This process involves reading from files corresponding to different services and mapping ping times to the defined state space. This is done using the following preprocessing method: "load_data_and_convert_states". This method prepares the dataset for subsequent analysis, ensuring data is in the appropriate format for calculating transition probabilities. I then created a function that maps continuous ping time measurements to discrete states, creating a finite state space necessary for Markov Chain modeling.

2.2 Calculating the Transition Matrix

In the process of forecasting network states using a Markov chain, we employ a probabilistic approach that hinges on determining the likelihood of transitioning to a subsequent state based on historical n-state sequences. This task is undertaken by the 'calculate_transition_matrix' method which calculates the transition matrix from a given array of states. This method analyzes historical data to compute transition probabilities, a critical step given the challenge of data sparsity in network forecasting. When encountering instances of insufficient data to reliably predict n-order states, the algorithm adaptively reduces n, descending

to a lower-order state with available valid probabilities to ensure continuity in forecasting without significant interruption.

The formulation of the state sequences culminates in the construction of the transition matrix, where each unique sequence's frequency of transitioning to a subsequent state is recorded. The matrix is dynamically updated to accommodate a new sequence construction, preventing any potential key errors. Afterwards, we normalize these frequencies to probabilities as it transforms the raw transition frequencies into a structured transition matrix, giving us the foundation of the forecasting model's probabilistic framework. We can expect each run to the difference, with the level of difference increasing as we increase the number of future steps forecasted. The figure below shows an example of how the forecast created by the Markov Chain can vary drastically,

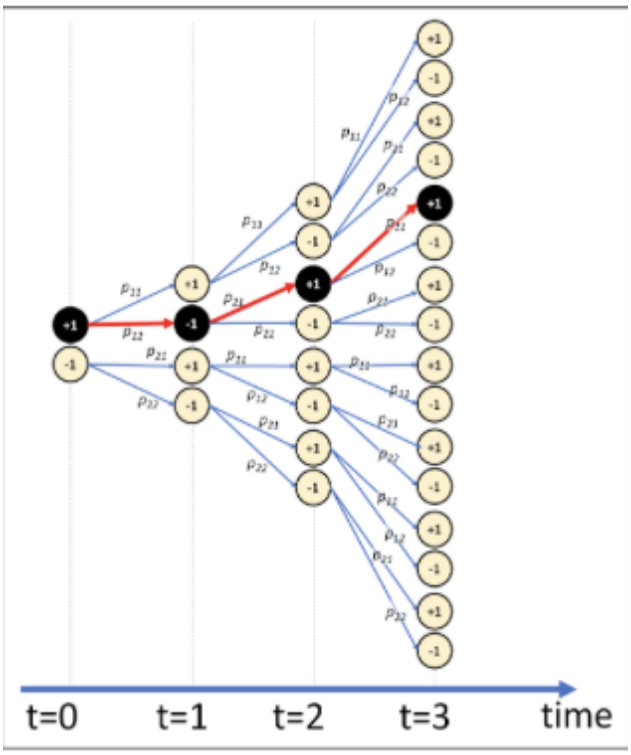

Figure 2.1: Adapted from timeseriesreasoning.com. This figure shows the memoryless nature of the First Order Markov Chain, as the state generated is solely dependent on the previous state. If we want to see a state "n" steps ahead, we must generate steps 1 to n - 1 first.

2.3 Forecast future states

Now that we have the ability to calculate the transition matrices, we can use these to infer the future states. This is done in `forecast_next_states(self, service_name, initial_sequence`.

As per project specifications, we must forecast for a user-inputted service. To meet this requirement, the program only uses the transition matrix made from data collected regarding that specific service. Due to the nature of the Markov chain, we must iterate the number of times requested by the user. For example, if inputted a sequence `[4,5,6]`, and the desired forecasted output is 5 out, we must calculate the first four in order the calculate the fifth one. For each of this steps, the program predicts the next state based on the current n-state sequence and updates this sequence with the predicted state for the following predictions. In order to counter data sparcity, my current implementation relies on falling back on lower orders of n. So, if

`[4,5,6]` is not in our transition matrix, then we try to find `[5,6]` and so on until we have exhausted all order options. When a "next state" is successfully predicted, it's appended to the initial sequence, which is then used for predicting the subsequent state. This process mimics the progression of time and the accumulation of new states. When a state has been found, the program must stochastically choose which of the states should be the next state. For example, if the next state of `[4,5,6]` gives us two options of 55% chance of 5 being next and 45% chance of 7 being next, the program must use these probabilities to choose the next state. So, different executions of the program can lead to different outputs as the Markov chain uses stochastic choosing. To meet these ends, I have made the following methods to divide up the responsibilities of forecasting the new states: "forecast_next_states" and"stochastic_next_state".

2.4 Outputs

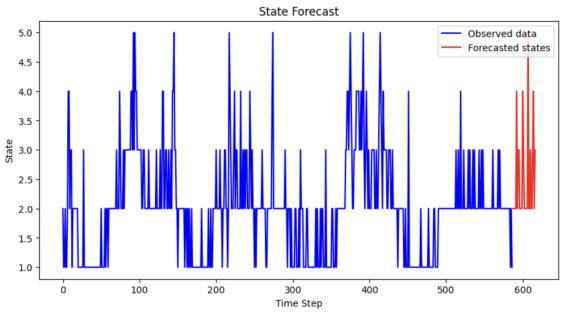

Figure 2.2: Next 30 steps ahead on one run

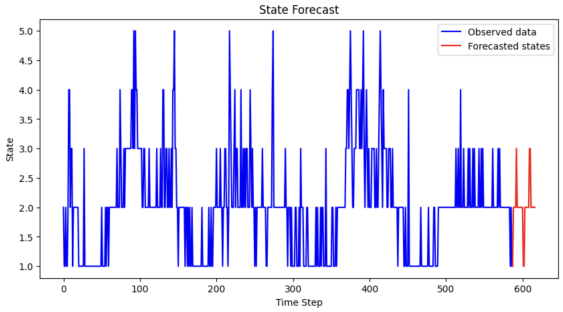

Figure 2.3: Next 30 steps ahead on another run

The images above are produced using a Third-Order MC. Due to the probabilistic nature of choosing the next state, the same input state could lead to different next states. Due to the discretization using the states, the graphs seem to be a bit jarring and demonstrate a lot of fluctuation. While network data fluctuates, due to our current network infrastructure it does follow a certain cyclical structure based on the time of day. Due to the relative simplicity of this model, I discretized the times in single minute intervals(as mentioned in the introduction chapter), thus each point forecasted essentially relies on the previous 30 minutes of data. My rationale for not choosing a smaller interval size was due to its effect on the complexity of the transition matrix, which would have to store a lot more unique sequences. Thus, the occurrence that we see where these two forecasts give us a somewhat similar output in terms of a smaller oscillation cycle to an extreme jump is relatively representative of the inputted data. An interesting observation to note is the similarities between a Markov Chain and Recurrent Neural Networks, which both process data sequentially. At this point during my exploration, using an RNN, in this case, an LSTM, was added to my goal list to reduce the negative impact caused by memorylessness. However, I had hoped to first incorporate more of our Bayesian understanding from class. Thus, I chose to use a Bayesian Autoregressive Model as my next method of forecasting, as the difference in orders of the AR(1) and AR(2) was a bit similar to the order of the "n"-Order Markov Chain.

3 Bayesian Forecasting with Autoregressive Models

3.1 Introduction to Bayesian Forecasting for Time Series Analysis

Bayesian inference offers a framework for model building and evaluation that is fundamentally different from traditional methods. At the heart of Bayesian methods is Bayes Theorem, which provides a way to update our beliefs about unknown parameters as new data becomes available. The main premise is to take what we already know and create a methodology for the addition of incorporating new experiences. This approach

is particularly powerful in the context of time series analysis, where the inherent uncertainties about future states can be systematically quantified and incorporated into the model.

Autoregressive models are a fundamental part of time series forecasting due to their simplicity and effectiveness. An AR model predicts future values based on a linear combination of past observations, making it well-suited for many practical forecasting tasks. The order of an AR model denoted as p, determines how many previous observations the model considers, with AR(1) considering one lag and AR(2) considering two lags, and so forth. Each lag is essentially a previous time step. This is a parallel to the "n-order Markov Chain" I have discussed in the previous chapter. For the sake of space I will omit more background on the way we define prior distributions, likelihood, and posterior as we have discussed these concepts in class.

3.2 Understanding Autoregressive Models

Autoregressive (AR) models are a class of linear models used for analyzing and forecasting time series data, where future values are assumed to be a linear combination of past values. They are a cornerstone in time series analysis due to their simplicity and ability to capture the essence of temporal dependencies.

An AR model uses a specified number of lagged (prior) observations in the time series as predictors for the current value. The model is defined as follows:

$$X_t = \alpha + \beta_1 X_{t-1} + \beta_2 X_{t-2} + ... + \beta_p X_{t-p} + \epsilon_t$$

where:

1. X_t is the current value of the time series at time t.
2. α is the intercept term of the model.
3. $\beta_1, \beta_2, ..., \beta_p$ are the coefficients that represent the impact of each lagged term on X_t.
4. p is the order of the AR model, determining how many lagged terms are included.
5. ϵ_t is the error term, representing random fluctuations that the model does not account for.

The order of an AR model, p, is a critical choice in modeling, as it determines the complexity and the capacity of the model to capture temporal patterns.

The AR(1) model is the simplest AR model, using only one lagged value: $X_t = \alpha + \beta_1 X_{t-1} + \epsilon_t$

The AR(2) model includes two lagged values: $X_t = \alpha + \beta_1 X_{t-1} + \beta_2 X_{t-2} + \epsilon_t$

Here, two coefficients, β_1 and β_2, capture the relationship of the current value with the two most recent past values. While the goal of AR(2) model is to describe more complex temporal structures, it should be noted that in large datasets the AR(2) model might not work well. The AR(2) model can describe more complex temporal structures than the AR(1) model, such as oscillations or cycles that cannot be captured with a single lag. Selecting the appropriate order of an AR model is crucial. A model that is too simple may not capture all the temporal dependencies, while a model that is too complex may fit noise in the data, leading to overfitting.

3.3 Bayesian Approach to AR Models

In a Bayesian framework, Autoregressive (AR) models are approached with a focus on probability distributions rather than point estimates. This allows us to express uncertainty around our model parameters and predictions, providing an alternative view of the future states of a time series as opposed to the previous Markov Chain approach I used. Bayesian inference revolves around updating our beliefs based on new data. When applied to AR models, this involves updating our beliefs about the model's parameters—typically the intercept and lag coefficients. Before observing any data, we have some beliefs—priors—about the likely values of our parameters. These priors can be informed by previous studies, domain knowledge, or they can be non-informative, expressing a state of relative ignorance. The likelihood function quantifies the plausibility

of our data given certain parameter values. For an AR model, this usually comes down to the likelihood of observing the time series given the autoregressive nature of the data. The posterior distribution is the goal of Bayesian inference, as it is essentially all that we know now after taking our new experiences into account. From this distribution, we can make predictions about future time series values. In other words, if we want to make future predictions to our future predictions, we would have to again apply the Bayesian rule to take into account the new states we generated.

In the Bayesian approach, we look to characterize the entire distribution of possible parameter values. I have characterized the distribution of possible parameter values using a Markov Chain Monte Carlo (MCMC) method - the Metropolis-Hastings algorithm, which provides us with a way to draw samples from the posterior distribution even when it cannot be described analytically. Forecasting in the Bayesian context means considering all possible futures weighted by their posterior probabilities.For instance, to generate forecasts using a Bayesian AR(1) model, we take the sampled posterior distributions of the parameters α and β_1 and use them to simulate many possible future paths of the time series. The mean of these paths can serve as our point forecast, while the variation among them provides a measure of forecast uncertainty. I chose the following priors for AR(1): $\alpha = 0, \beta = 0, \sigma = 1$. Due to the amount of data and number of iterations, regardless of the prior the MSE and estimates would converge to essentially the same numbers. I chose the following priors for AR(2): $\alpha = 0, \beta_1 = 0, \beta_2 = 0, \sigma = 1$.

3.4 Implementing Bayesian AR Models

While implementing these models, I learned how to use the Metropolis Hastings method, as well as gained a better intuition behind the Bayesian-way of updating a knowledge base. Due to the emphasis of the Bayesian Theorem in the course, going into this I already had a solid foundation; however, this allowed me the learn about the sampling method as well as apply the theoretical concepts to a practical foundation.

Implementing Bayesian AR models for time series forecasting involves multiple steps: defining the model, specifying priors, running the Metropolis-Hastings algorithm to sample from the posterior distribution, and validating the model's predictions.

The "metropolis_hastings_ar1" and "metropolis_hastings_ar2" functions are used for their corresponding model orders for parameter estimation. They perform MCMC sampling to approximate the posterior distributions of the AR model parameters given a set of ping time observations.

The "metropolis_hastings_ar1" function estimates parameters for a first-order autoregressive model, taking into account a single lagged observation.

Similarly, "metropolis_hastings_ar2" samples parameters for a second-order autoregressive model, considering two lagged observations. The function can be customized with different proposal standard deviations.

The "plot_forecasts" and "plot_forecasts_ar2" functions illustrate both the observed data and the predicted future values with the uncertainty represented by multiple forecast paths.

"plot_forecasts" uses the sampled parameters to simulate and plot possible future paths of ping times, providing a visual representation of the forecast uncertainty.

Similarly, "plot_forecasts_ar2" plots the forecasting results for the second-order AR model.

3.5 Outputs

As shown by the output, AR(1) was able to generate a more varied mean forecast however generated forecasts were very scattered as shown by the max height from 0. AR(2) was able to generate forecasts which were relatively more "contained", however the mean forecast beared a lot of resemblance to a zig-zag pattern. This is where its tendency to model cycles/ oscillations appears. Unfortunately, since network ping data cycles over a large time period, AR(2) seems to overly simplify the temporal relationship. Network ping data could resemble an oscillatory pattern; however, there is a lot of noise. It failed to account for essentially any variance. Additionally, due issues such as high traffic load, ping time does tend to drastically peak at times, which neither model was able to show. AR(1) didn't necessarily have the tightness of the pings the training data had and the AR(2) model didn't account for any variance. Now that we have the two models,

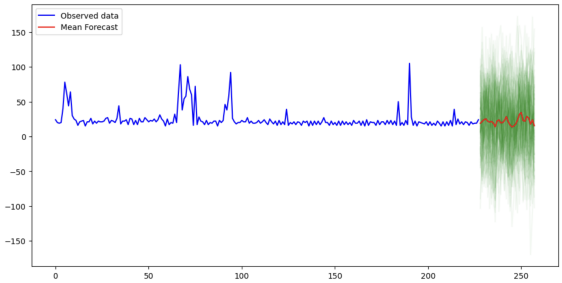

Figure 3.1: Next 30 steps ahead from the AR(1) model

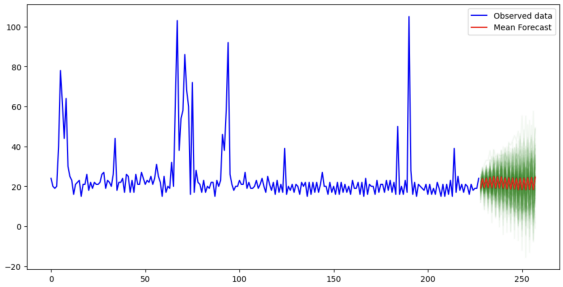

Figure 3.2: Next 30 steps ahead from the AR(2) model

we must validate them to see their overall performance. Using the following method, we can compare the two different order models to validate their usefulness. Given an 80-20 train test split, I have validated both of them using MSE to reach:

AR(1) Bayesian Model MSE: 298.5689674414406

AR(2) Bayesian Model MSE: 299.13034919428594

While this model might have been a bit more robust, I had hoped for a model which could incorporate more of the data, and realize the daily trend in the data as well as the overall trend in the data. Thus, I chose to try and implement an Autoformer to incorporate seasonality in my forecasting.

4 Understanding Transformers in Time Series Forecasting

4.1 Adapting Transformers for Time Series

Transformers, initially conceived for handling sequential linguistic data, have demonstrated remarkable versatility in time series forecasting due to the sequential and contextual nature common to both fields. Just as the meaning of a word like "bat" relies on its context within a sentence, the value of a time series data point is similarly influenced by its temporal neighbors. This underlying parallel has paved the way for adapting Transformer models, with their parallel processing of sequences and self-attention mechanisms, to forecast time series data effectively. These models dynamically adjust the importance of various input elements, providing a nuanced understanding of time-dependent relationships crucial for accurate predictions.

We can adapt transformers to address challenges posed by Traditional Time Series Forecasting by involving certain innovations made by researchers in this field. For example, we can encode temporal information explicitly, modify self-attention mechanisms to account for irregular intervals, or incorporate additional components to model seasonality and trends directly.

4.2 Autoformer: A Transformer for Time Series

The Autoformer represents a significant advance in time series forecasting, born out of the need to handle the challenges of sequential data in domains beyond the reaches of language. I learned about this advancement through the following **Hugging Face article**. The Autoformer brings a set of innovations designed specifically for time series analysis, responding to the unique challenges posed by this type of data.

Autoformer is tailored for time series forecasting. Its purpose is to understand and leverage the underlying patterns in time series data—particularly seasonality and trend—without being overwhelmed by noise and irregularities. The Autoformer accomplishes this by introducing two key architectural innovations:

Series Decomposition: At its heart, the Autoformer architecture features a series decomposition mechanism that separates the input time series into seasonal and trend-cyclical components. This separation allows the model to handle each component with the attention it requires, mitigating the challenge of conflating short-term fluctuations with long-term movements.

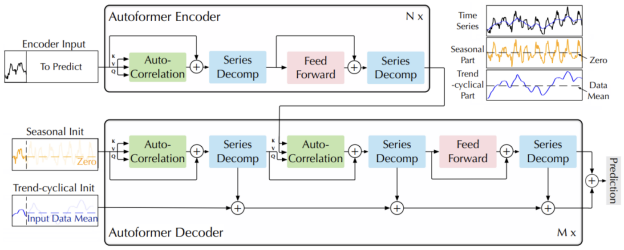

Figure 4.1: The Architecture of an Autoformer, adapted from Hugging Face

Auto-Correlation Attention: Building on the self-attention mechanism's strengths, the Autoformer integrates an auto-correlation module designed to capture the periodicity in the data. This module replaces the vanilla Transformer's self-attention with a mechanism that assesses the auto-correlation within the series, enabling the model to prioritize time steps that are most predictive of future values.

4.2.1 Integrating the Innovations

The Autoformer's encoder focuses on modeling the seasonal patterns, thus eliminating the long-term trend-cyclical component. This distinguishes between noise and seasonality - addressing one of the key limitations of conventional transformers in handling time series data. The decoder of the Autoformer works hand in hand with the encoder, utilizing the seasonal information processed by the encoder and combining it with the trend-cyclical information to produce accurate forecasts. This connection between the encoder and decoder, through a shared auto-correlation mechanism, ensures that the model's predictions are informed by the complete picture of the data's dynamics. The structure of the Autoformer is such that it processes the input multiple times (N times as denoted in the architecture), refining the decomposition and enhancing the forecast with each iteration. This iterative process allows for an increasingly refined understanding of the data's characteristics.

4.3 Mathematical Foundations

Autoformer model utilizes autocorrelation - a measure traditionally used in statistical analysis to quantify the linear relationship between observed values at different times - to incorporate a more complex understanding of time series data. Autocorrelation at a time lag τ is defined as the correlation of the time series with itself at that lag. For a discrete time series variable y, this is mathematically expressed as:

$$\text{Autocorrelation}(\tau) = \text{Corr}(y_t, y_{t-\tau})$$

This equation computes the Pearson correlation between values of y at the current time t and its past values at time $t - \tau$, providing a measure of how past values are associated with current values. The Autoformer leverages this concept by extracting frequency-based dependencies, avoiding the standard self-attention mechanism's dot-product for a methodology that better suits the temporal dynamics of time series data. In essence, it treats the autocorrelations $R_{Q,K}(\tau)$ as attention weights.

In practice, the autocorrelation of queries Q and keys K for all lags is computed efficiently using the Fast Fourier Transform (FFT). This method not only ensures computational efficiency but also aligns with the Wiener-Khinchin theorem, allowing for a time complexity of $O(L \log L)$, where L is the length of the input time series. The actual computation of the autocorrelation-based attention in the Autoformer can be described with the following steps: compute the autocorrelations $R_{Q,K}(\tau)$ for all lags using FFT, identify the top k significant autocorrelations using the arg Top-k operation, denoted as $\hat{R}_{Q,K}(\tau)$, and finally normalize these autocorrelation values using a softmax function to generate weights.

This process can be formulated as:

$$\hat{R}_{Q,K}(\tau_1), \hat{R}_{Q,K}(\tau_2), ..., \hat{R}_{Q,K}(\tau_k) = \arg \text{ Top-k}(R_{Q,K}(\tau))$$

$$\tilde{R}_{Q,K}(\tau_1), \tilde{R}_{Q,K}(\tau_2), ..., \tilde{R}_{Q,K}(\tau_k) = \text{Softmax}(\hat{R}_{Q,K}(\tau_1), \hat{R}_{Q,K}(\tau_2), ..., \hat{R}_{Q,K}(\tau_k))$$

Following this, the autocorrelation-attention for the time series can be obtained by:

$$\text{Autocorrelation-Attention} = \sum_{i=1}^{k} \text{Roll}(V, \tau_i) \cdot \tilde{R}_{Q,K}(\tau_i)$$

Here, $\text{Roll}(V, \tau_i)$ is the operation that aligns value vectors V with the time series' autocorrelations at lag τ_i, enabling element-wise multiplication that aligns the model's focus with the most relevant past values.

Incorporating autocorrelation into the attention mechanism allows the Autoformer to pay more attention to parts of the time series that are predictively relevant, providing a robust mechanism for forecasting future values in the presence of time-dependent patterns.

4.4 Challenges

Unfortunately due to the time constraint of the project I was unable to fully implement the Autoformer. While I wasn't able to implement it, I have a gained a much deeper understanding in this field which I lacked prior exposure to before. In order to validate another implementation technique worked, I moved onto the bit simpler Long Short Term Memory model.

5 Forecasting with LSTM: A Probabilistic Approach

LSTMS are a specialized type of RNN that are known for their ability to process sequences through their unique architecture. They capture long-term patterns in data. This capability is particularly advantageous in certain forecasting scenarios where understanding historical context is crucial. LSTMs are able to estimate entire distributions of future outcomes, providing a comprehensive view of possible future states. By designing LSTMs to output parameters such as the mean and variance of distributions—like the Gaussian distribution—these models can offer a more comprehensive understanding of uncertainty in predictions. This approach allows the model to handle the inherent variability and unpredictability of real-world network data effectively, making them very useful tools for tasks that require a better understanding of the way data changes over time.

5.1 Probabilistic Interpretation of LSTMs

In probabilistic forecasting with LSTMS to goal is to provide a distribution over possible outcomes rather than predicting a single point estimate for future values. This approach is valuable in expressing the uncertainty inherent in the predictions. LSTMs can be adapted for probabilistic forecasting by designing them to predict the parameters of a probability distribution at each time step.

Let's consider a Gaussian distribution, characterized by two parameters: mean (μ) and variance (σ^2). In a probabilistic interpretation, the LSTM is structured to predict both these parameters for each time step. This is accomplished by modifying the output layer to have two units per forecast dimension, one for the mean and one for the logarithm of the variance, which ensures that the variance remains positive. Given an LSTM outputting μ_t and $\log(\sigma_t^2)$ at time t, the predicted distribution of the target variable y_t is given by $y_t \sim \mathcal{N}(\mu_t, \sigma_t^2)$ The LSTM is trained to maximize the likelihood of the observed data under the predicted distribution. The loss function, therefore, is the negative log-likelihood (NLL) of the Gaussian distribution:

$$\text{NLL} = -\sum_{t=1}^{T} \log \mathcal{L}(y_t | \mu_t, \sigma_t^2)$$

where $\mathcal{L}$ denotes the likelihood function of the Gaussian distribution and T is the total number of time steps. The likelihood function of a Gaussian distribution is:

$$\mathcal{L}(y_t|\mu_t, \sigma_t^2) = \frac{1}{\sqrt{2\pi\sigma_t^2}} \exp\left(-\frac{(y_t - \mu_t)^2}{2\sigma_t^2}\right)$$

The logarithm of this likelihood, which we use in the NLL, simplifies to:

$$\log \mathcal{L}(y_t|\mu_t, \sigma_t^2) = -\frac{1}{2}\log(2\pi\sigma_t^2) - \frac{(y_t - \mu_t)^2}{2\sigma_t^2}$$

By dropping constant terms, which don't affect the optimization, the NLL simplifies to:

$$\text{NLL} = \frac{1}{2}\sum_{t=1}^{T}\left(\log(\sigma_t^2) + \frac{(y_t - \mu_t)^2}{\sigma_t^2}\right)$$

This goal of this portion of the exploration was to input my insight gained in class to a standard model - the LSTM.

5.2 Training and Validating the LSTM Model

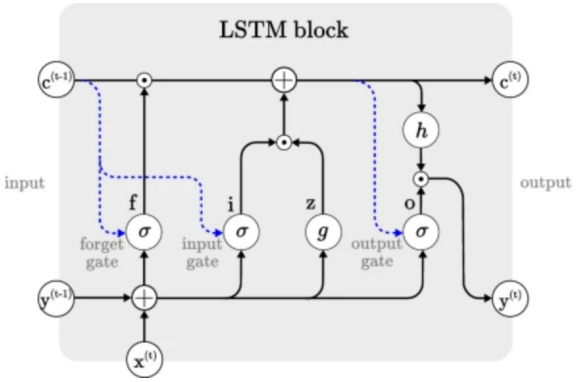

Figure 5.1: The Architecture of an LSTM Model

Here is an image of the typical architecture for an LSTM. Please note that while tanh is used as the gate g, upon further reading, I have found that ReLU is appropriate in some situations, and that was the case for this dataset. For my model, I used a sequence length of 10. What this essentially means is that each point is dependent on a sequence of 10 points. As mentioned before, each data point represents a 30-second interval, thus 10 points represent 300 seconds. The motive behind this decision is to rely on shallow domain knowledge. On the average occasion our network infrastucture should be able to recover from any anomolous behavior after a few minutes. Thus, by accounting for more than a couple minutes I hopes to capture more of the behavior of the data. Another important feature I added to my custom Gaussian-likelihood loss function(based on the math in section 5.1) is clipping to prevent exploding gradients. Overall, the training process remains relatively the same, however I did incorporate two different practices than normal. I implemented the negative log-likelihood in the function "gaussian_likelihood" and used that as the loss function to train my model on. I implemented the 'EarlyStopping' callback to halt the training when the validation improved. Evaluating LSTM forecasts, especially those with a probabilistic interpretation, requires a multistep approach. It's essential not only to assess the accuracy of the predictions but also to evaluate how well the model quantifies its uncertainty.

I used a similar Mean Squared Error as I did with the Bayesian AR models, to achieve a Mean Squared Error of. 0.0008467885675357501. I also examined the Mean Log Likelihood when comparing the validation predictions and the actual validation data to reach a MLL of 3.136842966079712. Just a note, I multiplied the NLL by -1 to reach this number purely for interpretability. As shown by the predicted means plot below, the LSTM was unable to capture the intricacies behind the spikes in the data, and during the regular ping times it wasn't able to best capture the standing variation.

5.3 Inference and Forecast Generation

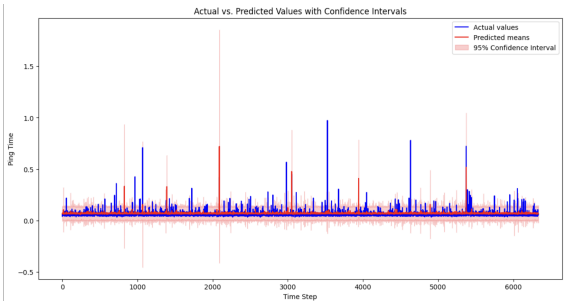

Figure 5.2: Actual vs Predicted Means

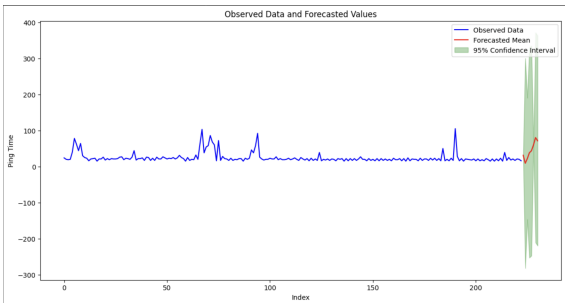

Figure 5.3: LSTM forecast next 8 steps

After training, the model can generate forecasts for new or unseen data. This process involves feeding input sequences into the model and interpreting its outputs as parameters of the predictive distribution. Using the prediction feature provided with the model, I predicted the next 8 steps. The tall confidence intervals indicate the model is less certain about its forecasts. The predictions could reasonably vary within a large range, suggesting that the underlying data might be volatile or the model has limited information about the data patterns at those points. Additionally, although I collected data for 11 days, there were only 31670 sequences. While network data does have a tendency to be volatile at times, as shown by the training data, most of the time it within a much tighter range. Based on the few next steps shown by the LSTM, the output seemed to have incorporated a bit much of the tendency to spike. The LSTM might not have been able to capture context behind when spikes occur compared to when the network is at its average speed.

6 End of Exploration

6.1 Challenges Due to the Nature of Networks Data

Network data is inherently complex and dynamic due to instead stochastic nature and its relation to multiple external factors. Unlike data from controlled experiments or systems with predictable behavior, network metrics such as latency or ping times are influenced by a wide array of factors, including network topology changes, routing protocols, load balancing, infrastructure scaling, maintenance activities, and user demand. Even if I were to ping certain IP addresses as opposed to services, due to Anycast(the methodology of giving several physical locations the same digital address) the same IP address could route us to either a server in Virginia, or Washington or even Toronto! These factors introduce a level of variability and non-stationarity that challenges traditional modeling approaches. Stochastic variability refers to the randomness in network behavior, particularly in response to varying load conditions and infrastructure changes, adding a layer of unpredictability not easily captured by deterministic models. Stationarity—a requirement for many time series models—refers to the statistical properties of the process generating the data do not change over time. Network ping data is unfortunately non-stationary. The performance metrics of networks are influenced by numerous unseen factors, creating complex dependencies that are difficult to model with simple probabilistic or linear assumptions.

6.2 Comparison of the Models and Final Thoughts

Time series forecasting with Markov chains presents a unique challenge, especially when considering first vs higher "n" order. With the first order, one of the primary issues is the reliance on the Markov property, which assumes that the future state depends only on the current state and not on the sequence of events that preceded it. This can be overly simplistic for many real-world time series data, where the underlying dynamics may be influenced by long-term dependencies and complex interactions not captured in the transition probabilities of a Markov chain. Additionally, the accurate estimation of transition probabilities necessitates a substantial amount of data, which might not be available or could be sparse, leading to a less reliable model. As the order increases, so does the complexity of the model. The amount of data needed to accurately estimate transition probabilities grows exponentially with the order of the chain due to the need to obtain enough historical data to cover all possible combinations of states. These limitations require us to use careful consideration and, pushed me to look into using RNNs for a more sophisticated approach.

RNNs maintain a form of memory by using their internal state (hidden layer outputs), which allows them to process sequences of data. RNNs, however, are subject to problems accompanied by the vanishing or exploding gradient, where the model loses its ability to learn long-term dependencies as the gradient of the loss function becomes too small or too large to handle effectively. LSTMs are essentially an evolution of RNNs designed specifically to overcome the problem of learning long-term dependencies. LSTMs can better regulate the flow of information by using a complex architecture with a system of gates (input, forget, and output gates) to decide what to keep or discard as the sequence is processed.

I chose the Bayesian AR Models for their probabilistic approach to forecasting, allowing for direct quantification of uncertainty—a critical aspect when dealing with stochastic network data. However, the assumption of stationary statistical properties poses a significant challenge, as network behavior is dynamic and influenced by factors such as load changes and infrastructure updates, leading to non-stationary data. Thus, a model such as the Autoformer might be more appropriate for this situation. MCMC algorithms such as the Metropolis-Hastings algorithm I used require a large number of iterations to adequately explore the parameter space and ensure convergence to the posterior distribution.

The Autoformer model was supposed to provide the best results as it's ability to better model seasonality(daily ping traffic) and general trends would allow for a much better prediction compared to the AR(2) model. One of the most difficult challenges encountered I faced while trying to implement the Autoformer was correctly implementing the matrix mathematics, particularly within the autocorrelation attention mechanism. Theoretical formulations can be expressed neatly in mathematical notation, but translating these into code that operates on tensors requires a careful handling of dimensions and a strong grasp understanding of the algebra libraries used. Debugging these calculations posed its own difficulties. Unlike scalar values, tensors can't be easily inspected or understood at a glance. Misalignments in matrix dimensions or incorrect applications of operations like transposing or reshaping can lead to bugs that I can't figure out.

I used LSTM for its advanced capability to learn long-term dependencies, addressing the limitations of basic RNNs and Markov Chains. The progression from simple Markov Chains to (RNNs) and eventually to LSTMs reflects an evolution in model capability to capture these longer-term dependencies, acknowledging that real-world phenomena often cannot be accurately represented under the assumption of memorylessness. The decision to use a sequence length of 10 was a balance between capturing sufficient context for prediction and mitigating the risk of exploding gradients— since other values I tested using would blow out the gradient past the number bit limit for Python.

Throughout the course of this project, I have gained a deeper understanding of the ways in which we can apply Probabilistic Graphical Models to time-series forecasting. I have solidified my understand of the Markov Chain Model, gained better intuition behind the Bayesian Approach, learned the theoretical background of the Autoformer model, as well as inputted my own understanding of probabilistic forecasting to provide a modified-application of the LSTM network.