Sharif University of Technology

Masoud Tahmasbi Fard

| | | |

Student ID: 402200275

EE120: Deep Generative Models

Assignment #1

October 21, 2024

Table of Contents

1 Autoregressive Models of Order 1

1.1 Log-Likelihood Function

The AR(1) model is written as:

$$y_t = \phi y_{t-1} + \epsilon_t, \quad \epsilon_t \sim \mathcal{N}(0, \sigma^2)$$

Likelihood Function

Given the observations $y_1, y_2, \ldots, y_n$, the likelihood function expresses the joint probability of observing the data given the parameters ϕ and σ^2. The error term at time t can be written as:

$$\epsilon_t = y_t - \phi y_{t-1}$$

Each ϵ_t is independently and identically distributed according to a normal distribution, so the likelihood function $L(\phi, \sigma^2)$ is the product of the individual probability densities:

$$L(\phi, \sigma^2) = \prod_{t=1}^{n} \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(y_t - \phi y_{t-1})^2}{2\sigma^2}\right)$$

Log-Likelihood Function

Taking the natural logarithm of the likelihood function to derive the log-likelihood:

$$\log L(\phi, \sigma^2) = \sum_{t=1}^{n} \left(-\frac{1}{2}\log(2\pi\sigma^2) - \frac{(y_t - \phi y_{t-1})^2}{2\sigma^2}\right)$$

Simplifying the log-likelihood function:

$$\log L(\phi, \sigma^2) = -\frac{n}{2}\log(2\pi\sigma^2) - \frac{1}{2\sigma^2}\sum_{t=1}^{n}(y_t - \phi y_{t-1})^2$$

1.2 Maximum Likelihood Estimation

We will maximize the log-likelihood function to estimate the parameters ϕ and σ^2 for the AR(1) model.

The log-likelihood function for the AR(1) model is:

$$\log L(\phi, \sigma^2) = -\frac{n}{2}\log(2\pi\sigma^2) - \frac{1}{2\sigma^2}\sum_{t=1}^{n}(y_t - \phi y_{t-1})^2$$

Our goal is to find the parameter values ϕ and σ^2 that maximize this function. We will maximize it by taking the partial derivatives with respect to ϕ and σ^2 and setting them equal to zero.

Maximizing with respect to σ^2

Taking the partial derivative of $\log L(\phi, \sigma^2)$ with respect to σ^2:

$$\frac{\partial}{\partial\sigma^2}\log L(\phi, \sigma^2) = -\frac{n}{2\sigma^2} + \frac{1}{2(\sigma^2)^2}\sum_{t=1}^{n}(y_t - \phi y_{t-1})^2$$

Setting this equal to zero to find the critical point:

$$-\frac{n}{2\sigma^2} + \frac{1}{2(\sigma^2)^2}\sum_{t=1}^{n}(y_t - \phi y_{t-1})^2 = 0$$

Multiplying through by $2(\sigma^2)^2$ to clear the denominator:

$$-n\sigma^2 + \sum_{t=1}^{n}(y_t - \phi y_{t-1})^2 = 0$$

Solving for σ^2:

$$\sigma^2 = \frac{1}{n}\sum_{t=1}^{n}(y_t - \phi y_{t-1})^2$$

Thus, the estimate for σ^2 is the average of the squared residuals (i.e., the variance of the error term).

Maximizing with respect to ϕ

Taking the partial derivative of $\log L(\phi, \sigma^2)$ with respect to ϕ:

$$\frac{\partial}{\partial\phi}\log L(\phi, \sigma^2) = \frac{1}{\sigma^2}\sum_{t=1}^{n}(y_t - \phi y_{t-1})y_{t-1}$$

Setting this equal to zero:

$$\sum_{t=1}^{n}(y_t - \phi y_{t-1})y_{t-1} = 0$$

Expanding this expression:

$$\sum_{t=1}^{n} y_t y_{t-1} - \phi\sum_{t=1}^{n} y_{t-1}^2 = 0$$

Solving for ϕ:

$$\phi = \frac{\sum_{t=1}^{n} y_t y_{t-1}}{\sum_{t=1}^{n} y_{t-1}^2}$$

Final Estimates

- The estimate for ϕ is:

$$\hat{\phi} = \frac{\sum_{t=1}^{n} y_t y_{t-1}}{\sum_{t=1}^{n} y_{t-1}^2}$$

- The estimate for σ^2 is:

$$\hat{\sigma}^2 = \frac{1}{n}\sum_{t=1}^{n}(y_t - \hat{\phi} y_{t-1})^2$$

2 Autoregressive Models

2.1 Maximum Likelihood Estimation

In this problem, we're given an autoregressive (AR) model where the conditional distributions $p(x_t|x_1, x_2, \dots, x_{t-1})$ are modeled by a neural network f_θ with parameters θ. The conditional distribution follows a Gaussian distribution with a mean $f_\theta(x_1, x_2, \dots, x_{t-1})$ and a fixed variance σ^2, i.e.,

$$p(x_t|x_1, x_2, \dots, x_{t-1}) = \mathcal{N}(f_\theta(x_1, x_2, \dots, x_{t-1}), \sigma^2)$$

We need to derive the log-likelihood function for the joint distribution and compute gradients for training the neural network using backpropagation.

Joint Probability Distribution

The joint probability distribution of the sequence $\mathbf{x} = (x_1, x_2, \dots, x_T)$ is factorized as:

$$p(\mathbf{x}) = p(x_1)p(x_2|x_1)p(x_3|x_1, x_2)\cdots p(x_T|x_1, x_2, \dots, x_{T-1})$$

From the given assumption, each conditional probability follows a normal distribution. Thus, the joint probability becomes:

$$p(\mathbf{x}) = \prod_{t=1}^{T} \mathcal{N}(x_t|f_\theta(x_1, x_2, \dots, x_{t-1}), \sigma^2)$$

Log-Likelihood Function

The log-likelihood function $\log p(\mathbf{x})$ is the logarithm of the joint probability distribution:

$$\log p(\mathbf{x}) = \sum_{t=1}^{T} \log \mathcal{N}(x_t|f_\theta(x_1, x_2, \dots, x_{t-1}), \sigma^2)$$

For a Gaussian distribution $\mathcal{N}(x_t|\mu, \sigma^2)$, the log-probability density function is:

$$\log \mathcal{N}(x_t|\mu, \sigma^2) = -\frac{1}{2}\log(2\pi\sigma^2) - \frac{(x_t - \mu)^2}{2\sigma^2}$$

Substituting this into the log-likelihood function, we get:

$$\log p(\mathbf{x}) = \frac{T}{2}\log(2\pi\sigma^2) - \frac{1}{2\sigma^2}\sum_{t=1}^{T}(x_t - f_\theta(x_1, x_2, \dots, x_{t-1}))^2$$

Gradient Computation for Training

To train the neural network f_θ, we aim to maximize the log-likelihood with respect to the parameters θ. This is equivalent to minimizing the negative log-likelihood (which acts as a loss function). The negative log-likelihood is:

$$\mathcal{L}(\theta) = \frac{1}{2\sigma^2} \sum_{t=2}^{T} (x_t - f_\theta(x_1, x_2, \ldots, x_{t-1}))^2$$

This is the Mean Squared Error (MSE) between the true values x_t and the predictions $f_\theta(x_1, x_2, \ldots, x_{t-1})$. The gradients of this loss function with respect to the parameters θ can be computed using backpropagation.

The gradient of the loss with respect to θ is:

$$\frac{\partial \mathcal{L}(\theta)}{\partial \theta} = \frac{1}{\sigma^2} \sum_{t=2}^{T} (x_t - f_\theta(x_1, x_2, \ldots, x_{t-1})) \cdot \frac{\partial f_\theta(x_1, x_2, \ldots, x_{t-1})}{\partial \theta}$$

This gradient can be computed efficiently using backpropagation through the neural network f_θ, where the term $\frac{\partial f_\theta(x_1, x_2, \ldots, x_{t-1})}{\partial \theta}$ is obtained by differentiating the output of the network with respect to its parameters θ.

Thus, we use gradient-based optimization techniques (e.g., stochastic gradient descent) to update θ and minimize the loss function $\mathcal{L}(\theta)$.

2.2 Predictive Sampling

Naive Sampling Approach

In the naive sampling approach for generating a new sequence $\mathbf{x}' = (x'_1, x'_2, \ldots, x'_T)$ from the autoregressive model, we proceed sequentially. The key idea of the autoregressive model is that each value x'_t depends on the previous values $x'_1, x'_2, \ldots, x'_{t-1}$, and thus, the process of generating the sequence requires computing each value one step at a time.

Steps in the Naive Sampling Process:

1. **Initialization**: First, we compute the initial hidden representation h_1 based on an empty input (or some initial state). This is given by:

$$h_1 = f_\theta(\varnothing)$$

where f_θ is the neural network modeling the conditional distributions. The first value x'_1 is sampled from the conditional distribution:

$$x'_1 \sim \mathcal{N}(f_\theta(\varnothing), \sigma^2)$$

This samples x'_1 from the Gaussian distribution with mean given by $f_\theta(\varnothing)$ and variance σ^2.

2. **Sequential Sampling**: After sampling x'_1, the next hidden representation h_2 is computed based on x'_1, and the next value x'_2 is sampled:

$$h_2 = f_\theta(x'_1), \quad x'_2 \sim \mathcal{N}(f_\theta(x'_1), \sigma^2)$$

3. **Iteration**: This process repeats for each subsequent time step t, where we compute the hidden representation h_t based on the previously sampled values $x'_1, x'_2, \ldots, x'_{t-1}$ and sample x'_t from the corresponding conditional distribution:

$$h_t = f_\theta(x'_1, x'_2, \ldots, x'_{t-1}), \quad x'_t \sim \mathcal{N}(f_\theta(x'_1, x'_2, \ldots, x'_{t-1}), \sigma^2)$$

4. **Termination**: The process continues until the final value x'_T is sampled, completing the sequence.

The key limitation of the naive sampling approach is that it is inherently sequential: each new value depends on the previously sampled values, so we must generate the sequence step by step, and no parallelization is possible.

Predictive Sampling Approach [1]

The predictive sampling approach attempts to optimize the generation process by introducing the concept of *forecasts* for the next value in the sequence. If we have a high-confidence forecast $\tilde{x}_1$ that approximates the actual sampled value x'_1, we can potentially avoid redundant computations and speed up the process.

Steps in the Predictive Sampling Process:

1. **Forecasting**: Instead of sampling x'_1 directly, we first compute a forecast $\tilde{x}_1$, which is expected to be close to the true value x'_1 with high probability.

2. **Parallel Computation of Hidden Representations**: Using the forecast $\tilde{x}_1$, we can compute the next hidden representation h'_2 in parallel, even before the actual value x'_1 is sampled:

$$h'_2 = f_\theta(\tilde{x}_1)$$

   This pre-computed hidden representation is valid if the forecast $\tilde{x}_1$ turns out to be equal to the sampled value x'_1.

3. **Sampling**: We then sample x'_1 from the conditional distribution, as in the naive approach:

$$x'_1 \sim \mathcal{N}(f_\theta(\varnothing), \sigma^2)$$

   If $x'_1 = \tilde{x}_1$, the pre-computed hidden representation h'_2 is valid, and we can immediately sample x'_2 using h'_2 without recomputing:

$$x'_2 \sim \mathcal{N}(f_\theta(x'_1), \sigma^2)$$

   This saves one call to the neural network f_θ.

4. **Forecast Validity**: In the general case, for a sequence of n correct forecasts $\tilde{x}_1, \tilde{x}_2, \ldots, \tilde{x}_n$, the pre-computed hidden representations $h'_2, h'_3, \ldots, h'_{n+1}$ are valid if each forecast matches the sampled value. This allows us to sample multiple values in parallel without recomputation. When a forecast fails (i.e., $\tilde{x}_t \neq x'_t$), the subsequent hidden representations must be recomputed, and the process reverts to the sequential, naive sampling approach.

- **Naive Sampling**: Each value in the sequence is sampled one at a time, and each hidden representation is computed sequentially based on previously sampled values. This process is entirely sequential and cannot be parallelized.
- **Predictive Sampling**: By introducing forecasts, we can pre-compute hidden representations in parallel. If the forecasts are accurate, this approach saves computational effort by reducing redundant recomputations of the neural network's outputs.

2.3 KL Divergence between AR Models

The Kullback-Leibler (KL) divergence between two probability distributions $p_\theta(\mathbf{x})$ and $q_\phi(\mathbf{x})$, parameterized by f_θ and g_ϕ, respectively, is defined as:

$$\mathrm{KL}(p_\theta \| q_\phi) = \int p_\theta(\mathbf{x}) \log \frac{p_\theta(\mathbf{x})}{q_\phi(\mathbf{x})}\, d\mathbf{x}$$

In the case of autoregressive models, the joint probability distributions for the two models can be factorized as:

$$p_\theta(\mathbf{x}) = p_\theta(x_1) \prod_{t=2}^{T} p_\theta(x_t | x_1, x_2, \ldots, x_{t-1})$$

$$q_\phi(\mathbf{x}) = q_\phi(x_1) \prod_{t=2}^{T} q_\phi(x_t | x_1, x_2, \ldots, x_{t-1})$$

Thus, the KL divergence between the two autoregressive models becomes:

$$\mathrm{KL}(p_\theta \| q_\phi) = \int p_\theta(\mathbf{x}) \left(\log p_\theta(\mathbf{x}) - \log q_\phi(\mathbf{x})\right)\, d\mathbf{x}$$

Expanding this, we get:

$$\begin{aligned}\mathrm{KL}(p_\theta \| q_\phi) = \int p_\theta(x_1) \prod_{t=2}^{T} p_\theta(x_t | x_1, \ldots, x_{t-1}) \Bigg(& \log p_\theta(x_1) + \sum_{t=2}^{T} \log p_\theta(x_t | x_1, \ldots, x_{t-1}) \\ & - \log q_\phi(x_1) - \sum_{t=2}^{T} \log q_\phi(x_t | x_1, \ldots, x_{t-1}) \Bigg) d\mathbf{x}\end{aligned}$$

This expression highlights that evaluating the KL divergence involves comparing the log probabilities of each individual time step under the two models, p_θ and q_ϕ, across the entire sequence $\mathbf{x} = (x_1, x_2, \ldots, x_T)$.

Computational Challenges

Directly evaluating the KL divergence for high-dimensional sequences poses significant computational challenges:

1. **High-Dimensional Integrals:** The KL divergence involves an integral over the entire sequence space, which grows exponentially with the length of the sequence T. For large T, this integral becomes intractable to compute exactly.

2. **Autoregressive Dependence:** Both models are autoregressive, meaning that each term $p_\theta(x_t|x_1, \ldots, x_{t-1})$ and $q_\phi(x_t|x_1, \ldots, x_{t-1})$ depends on the entire history of previous values. This makes it difficult to evaluate the joint probability distributions efficiently, especially for long sequences.

Monte Carlo Approximation

A practical method for approximating the KL divergence in high-dimensional sequence models is to use *Monte Carlo* sampling. The idea is to approximate the expectation by drawing samples from the distribution $p_\theta(\mathbf{x})$ and then computing an empirical estimate of the KL divergence.

Steps for Monte Carlo Approximation:

1. **Sampling from p_θ:** Draw N independent samples $\{x^{(i)}\}_{i=1}^N$ from the autoregressive model $p_\theta(\mathbf{x})$.

$$x^{(i)} \sim p_\theta(\mathbf{x})$$

2. **Approximation of KL Divergence:** Using the samples, approximate the expectation in the KL divergence:

$$\mathrm{KL}(p_\theta \| q_\phi) = \mathbb{E}_{x \sim p_\theta}[\log p_\theta(x) - \log q_\phi(x)] \approx \frac{1}{N} \sum_{i=1}^{N} \left(\log p_\theta(x^{(i)}) - \log q_\phi(x^{(i)})\right)$$

Here, $\log p_\theta(x^{(i)})$ and $\log q_\phi(x^{(i)})$ are the log-likelihoods of the samples under the two models. Since both models are autoregressive, these log-likelihoods can be computed as the sum of the log conditional probabilities for each time step:

$$\log p_\theta(x^{(i)}) = \log p_\theta(x_1^{(i)}) + \sum_{t=2}^{T} \log p_\theta(x_t^{(i)}|x_1^{(i)}, \ldots, x_{t-1}^{(i)})$$

$$\log q_\phi(x^{(i)}) = \log q_\phi(x_1^{(i)}) + \sum_{t=2}^{T} \log q_\phi(x_t^{(i)}|x_1^{(i)}, \ldots, x_{t-1}^{(i)})$$

3. **Estimate of KL Divergence:** The final Monte Carlo estimate for the KL divergence is:

$$\hat{\mathrm{KL}}(p_\theta \| q_\phi) = \frac{1}{N} \sum_{i=1}^{N} \left(\log p_\theta(x^{(i)}) - \log q_\phi(x^{(i)})\right)$$

This approximation improves as N, the number of samples, increases.

The direct computation of KL divergence between two autoregressive models is computationally intractable due to high-dimensional integrals and the autoregressive structure. However, by using Monte Carlo sampling, we can approximate the KL divergence by drawing samples from p_θ and evaluating the log-likelihoods under both models. This approach provides a practical way to estimate the divergence, especially for high-dimensional sequence data.

2.4 Stationarity and Long-Term Dependencies

An autoregressive (AR) model of order p expresses the value at time t as a linear combination of the previous p values plus an error term. The AR(1) model, for example, is given by:

$$x_t = \phi x_{t-1} + \epsilon_t, \quad \epsilon_t \sim \mathcal{N}(0, \sigma^2)$$

Stationarity of AR Models

For an AR(1) model to be stationary, the coefficient ϕ must satisfy $|\phi| < 1$ [A]. When this condition holds, the effect of past observations decays geometrically as time progresses, which implies that the model forgets older inputs over time. The autocovariance function for the AR(1) model is [B]:

$$\gamma(h) = \mathbb{E}[x_t x_{t-h}] = \frac{\sigma^2}{1 - \phi^2} \phi^h$$

As h increases, $\gamma(h)$ decays exponentially. Therefore, the AR(1) model (and, more generally, AR models) has **short-term memory** and cannot capture **long-term dependencies** in the data.

For higher-order AR models (AR(p)), where:

$$x_t = \sum_{i=1}^{p} \phi_i x_{t-i} + \epsilon_t$$

The situation is similar: the dependence on past values still decays over time, making it challenging for the model to capture dependencies over long sequences. The AR model is inherently limited in capturing **long-term dependencies** due to the fixed number of lag terms and the rapid decay of influence as time progresses.

Mathematical Explanation of Limitation

The key limitation arises from the fact that the autoregressive model imposes a *geometric decay* of the influence of past observations on future values. Specifically, for the AR(1) model, the value x_t depends on x_{t-1} with coefficient ϕ, and the influence of x_{t-2} is ϕ^2, and so on. This means that:

$$\text{Influence of } x_{t-h} \text{ on } x_t \propto \phi^h$$

Thus, if $|\phi| < 1$, the influence of past observations decays exponentially with time, and the model cannot maintain memory of distant past values. This is problematic when trying to capture long-term dependencies in sequential data.

Modification: Using Recurrent Neural Networks (RNNs)

One way to address this limitation is to replace the linear autoregressive model with a **recurrent neural network (RNN)**, which can maintain a hidden state h_t that is updated at each time step based on the current input x_t and the previous hidden state h_{t-1}. The RNN model is given by:

$$h_t = \sigma(W_h h_{t-1} + W_x x_t + b_h)$$

$$x_t \sim \mathcal{N}(f_\theta(h_t), \sigma^2)$$

where $\sigma(\cdot)$ is a non-linear activation function, and W_h and W_x are weight matrices. The hidden state h_t serves as a summary of all past information up to time t, allowing the model to capture long-term dependencies.

Probabilistic Interpretation

In this modified RNN-based autoregressive model, the conditional distribution of x_t is now modeled as:

$$p(x_t|x_1, x_2, \ldots, x_{t-1}) = \mathcal{N}(f_\theta(h_t), \sigma^2)$$

where $f_\theta(h_t)$ is the output of the RNN, and h_t encodes information from all previous time steps. Unlike the linear AR model, the RNN can store and propagate information from distant past observations across time steps, thus capturing long-term dependencies.

Further Modification: Using Transformers

Another approach is to use a **Transformer** model, which is designed to capture both short-term and long-term dependencies via self-attention mechanisms. In the Transformer model, each time step attends to all previous time steps, which allows the model to capture dependencies across arbitrary distances.

For the autoregressive setting, the conditional distribution becomes:

$$p(x_t|x_1, x_2, \ldots, x_{t-1}) = \mathcal{N}(f_\theta(\text{Self-Attention}(x_1, x_2, \ldots, x_{t-1})), \sigma^2)$$

where the *self-attention* mechanism computes attention weights for each previous time step, allowing the model to learn dependencies between both nearby and distant time steps.

Probabilistic Interpretation of Transformer

In the Transformer-based autoregressive model, the self-attention mechanism computes a weighted combination of past inputs to produce the prediction for x_t. The weights are learned during training, and they allow the model to dynamically focus on the relevant past inputs, regardless of how far in the past they are.

Conclusion

The standard autoregressive model cannot capture long-term dependencies due to the exponential decay of past influences. To overcome this limitation, we can use models such as RNNs or Transformers, which can retain or attend to information from distant time steps, allowing them to capture long-term dependencies in the data.

2.5 Bonus

- **Hypothesis 1: Accumulation of Prediction Errors**

  In an autoregressive model, each predicted value $\hat{x}_t$ is based on the previously predicted value $\hat{x}_{t-1}$. As the sequence length increases, small prediction errors can accumulate, leading to larger overall deviations from the true values. This error accumulation grows as more time steps are predicted, degrading performance for longer sequences.

- **Solution 1: Incorporating Teacher Forcing During Training**

  To mitigate error accumulation, a common solution is to use *teacher forcing* during training, where the model is trained on the ground truth values rather than its own predictions. This ensures that the model does not propagate its own mistakes. During inference, a hybrid approach can be used, where the model alternates between using its predictions and ground truth values.

  In mathematical terms, teacher forcing introduces a modified training loss function where the loss at time t depends on the true value x_{t-1} rather than the predicted $\hat{x}_{t-1}$:

  $$\mathcal{L}_{\text{teacher forcing}} = \sum_{t=1}^{T} (\hat{x}_t - x_t)^2$$

  This helps reduce error propagation over long sequences.

- **Hypothesis 2: Vanishing Gradients in Long Sequences**

  As the length of the sequence increases, the model relies on long-range dependencies. In traditional autoregressive models, long-term dependencies can be difficult to capture effectively due to the vanishing gradient problem. Gradients can become very small when backpropagating through time, leading to ineffective parameter updates for early time steps in long sequences.

- **Solution 2: Using Gated Architectures or Attention Mechanisms**

  One solution is to use gated architectures, such as LSTM (Long Short-Term Memory) or GRU (Gated Recurrent Units), which are designed to handle long-range dependencies by using gates to control the flow of information. Alternatively, attention mechanisms, as used in Transformer models, allow the model to focus on specific time steps, thus addressing the limitations of purely autoregressive factorization by capturing long-term dependencies more effectively.

  For example, in the Transformer model, the attention score between time step t and t' is given by:

$$\text{Attention}(t, t') = \frac{\exp\left(\frac{q_t^T k_{t'}}{\sqrt{d_k}}\right)}{\sum_{t'} \exp\left(\frac{q_t^T k_{t'}}{\sqrt{d_k}}\right)}$$

  where q_t and $k_{t'}$ are the query and key vectors for time steps t and t', respectively. This mechanism allows the model to learn dependencies across all time steps, improving performance for longer sequences.

- **Hypothesis 3: Inability to Capture Non-Stationary Patterns**

  Another potential reason for performance degradation is that AR models are typically better suited for stationary time series, where the underlying distribution does not change over time. For non-stationary data (e.g., time series with trends or changing variance), the fixed model structure might fail to adapt to evolving patterns. In long sequences, if the data distribution shifts significantly over time, the model's fixed parameters θ may no longer capture the true dynamics of the data.

- **Solution 3: Adapt to the New Patterns**

  To handle non-stationarity, we can introduce models or techniques that can adapt to changing patterns over time:

  - **Dynamic AR Models:**

    Instead of using a fixed function f_θ to predict x_t, we can introduce time-varying parameters, allowing the model to adapt to evolving data distributions. One approach is to make the model parameters dependent on external signals or latent variables, which are learned jointly with the sequence. For example:

    $$x_t = f_{\theta(t)}(x_1, \ldots, x_{t-1}) + \epsilon_t,$$

    where $\theta(t)$ is a time-varying parameter that adapts over time.

  - **State-Space Models:**

    Another solution is to model the sequence using a state-space model, where a hidden state variable evolves according to a learned transition function. The hidden state captures the changing dynamics of the sequence, allowing the model to adapt to non-stationary patterns.

    $$h_t = g_\phi(h_{t-1}, x_t), \quad x_t \sim p_\theta(x_t|h_t).$$

    This approach can better capture non-stationary dynamics while maintaining the ability to make autoregressive predictions.

3 Real NADE Parameters

We are tasked with extending the Real NADE model to model the conditional distribution $p(x_i|x_{<i})$ as a *mixture of Gaussians*:

$$p(x_i|x_{<i}) = \sum_{c=1}^{C} \pi_i^c \mathcal{N}(x_i|\mu_i^c, (\sigma_i^c)^2)$$

where:

- π_i^c represents the mixture weights for the c-th component,
- μ_i^c is the mean of the c-th Gaussian component,
- $(\sigma_i^c)^2$ is the variance of the c-th Gaussian component.

Parameterization of π_i^c

The mixture weights π_i^c must satisfy the constraint that they are non-negative and sum to 1, i.e.,

$$\sum_{c=1}^{C} \pi_i^c = 1, \quad \pi_i^c \geq 0$$

A common approach to parameterizing π_i^c is to use a *softmax* function over the outputs of a neural network:

$$\pi_i^c = \frac{\exp(v_c^\pi \cdot h_i + b_c^\pi)}{\sum_{c'=1}^{C} \exp(v_{c'}^\pi \cdot h_i + b_{c'}^\pi)}$$

where:

- $h_i \in \mathbb{R}^H$ is the hidden representation at step i and H is the number of hidden units,
- $v_c^\pi \in \mathbb{R}^H$ is the weight vector for the c-th Gaussian component,
- $b_c^\pi \in \mathbb{R}$ is the bias term for the c-th Gaussian component.

Thus, we need $C \times H \times (D-1)$ parameters for the weight vectors and $C \times D$ parameters for the biases.

Parameterization of μ_i^c

The means μ_i^c of the Gaussian components are parameterized as a linear transformation of the hidden representation h_i:

$$\mu_i^c = v_c^\mu \cdot h_i + b_c^\mu$$

where:

- $v_c^\mu \in \mathbb{R}^H$ is the weight vector for the mean of the c-th Gaussian component,
- $b_c^\mu \in \mathbb{R}$ is the bias term for the mean of the c-th Gaussian component.

Thus, we need $C \times H \times (D-1)$ parameters for the weight vectors and $C \times D$ parameters for the biases.

Parameterization of σ_i^c

The variances $(\sigma_i^c)^2$ are positive, so we use an *exponential* transformation to ensure positivity. The logarithm of the standard deviation σ_i^c is parameterized as a linear transformation of the hidden representation h_i:

$$\log \sigma_i^c = v_c^\sigma \cdot h_i + b_c^\sigma$$

Thus:

$$\sigma_i^c = \exp(v_c^\sigma \cdot h_i + b_c^\sigma)$$

where:

- $v_c^\sigma \in \mathbb{R}^H$ is the weight vector for the log-standard deviation of the c-th Gaussian component,
- $b_c^\sigma \in \mathbb{R}$ is the bias term for the log-standard deviation of the c-th Gaussian component.

Thus, we need $C \times H \times (D-1)$ parameters for the weight vectors and $C \times D$ parameters for the biases.

Calculation of h_i

In the NADE framework, the hidden representations are obtained using the following formula:

$$\boldsymbol{h}_d = \operatorname{sigm}\left(\boldsymbol{W}_{:,o_{<d}} \boldsymbol{x}_{o_{<d}} + \boldsymbol{c}\right),$$

where $\text{sigm}(a) = 1/\left(1 + e^{-a}\right)$ is the logistic sigmoid, and with H as the number of hidden units, $\boldsymbol{W} \in \mathbb{R}^{H \times D}, \boldsymbol{c} \in \mathbb{R}^{H}$ are the parameters of the NADE model.

The hidden layer matrix $\boldsymbol{W}$ and bias $\boldsymbol{c}$ are **shared** by each hidden layer $\boldsymbol{h}_d$ (which are all of the same size). Thus, we need $H \times (D-1)$ parameters for hidden layer matrix $\boldsymbol{W}$and H parameters for the biases $\boldsymbol{c}$.

Total Number of Parameters

The total number of parameters required is the sum of the parameters for π_i^c, μ_i^c, σ_i^c, and h_i

- **Mixture weights** π_i^c: $C \times H \times (D-1)$ parameters for the weight vectors and $C \times D$ parameters for the biases,
- **Means** μ_i^c: $C \times H \times (D-1)$ parameters for the weight vectors and $C \times D$ parameters for the biases,
- **Log-standard deviations** σ_i^c: $C \times H \times (D-1)$ parameters for the weight vectors and $C \times D$ parameters for the biases,
- **Hidden Layers** h_i: $H \times (D-1)$ parameters for hidden layer matrix and H parameters for the biases.

Thus, the total number of parameters is:

$$\text{Total parameters} = 3 \times (C \times H \times (D-1) + C \times D) + H \times (D-1) + H$$

4 Monte Carlo Estimation

4.1 A Quick Warm Up

The expectation is given by:

$$\mathbb{E}_{x\sim N(0,2)}\left[x^2+x+1\right].$$

Using the linearity of expectation, we can break this into three terms:

$$\mathbb{E}\left[x^2+x+1\right]=\mathbb{E}[x^2]+\mathbb{E}[x]+\mathbb{E}[1].$$

Now, let's compute each term separately:

- $\mathbb{E}[x^2]$: For a normal distribution $N(0,\sigma^2)$, the expectation of x^2 is $\mathbb{E}[x^2]=\mathrm{Var}(x)+(\mathbb{E}[x])^2$. Since $\mathbb{E}[x]=0$ and $\mathrm{Var}(x)=2$, we have:

$$\mathbb{E}[x^2]=2.$$

- $\mathbb{E}[x]$: For a normal distribution with mean 0, $\mathbb{E}[x]=0$.

- $\mathbb{E}[1]$: This is just a constant, so $\mathbb{E}[1]=1$.

Putting it all together:

$$\mathbb{E}[x^2+x+1]=2+0+1=3.$$

Monte Carlo Estimation

To estimate this expectation using Monte Carlo estimation, we can follow these steps:

1. **Generate samples:** Draw n samples $\{x_1,x_2,\dots,x_n\}$ from the distribution $N(0,2)$.

2. **Evaluate the function:** For each sample x_i, compute $f(x_i)=x_i^2+x_i+1$.

3. **Compute the empirical average:** The Monte Carlo estimate of the expectation is given by the sample mean of the function evaluations:

$$\hat{\mathbb{E}}[f(x)]=\frac{1}{n}\sum_{i=1}^{n}f(x_i)=\frac{1}{n}\sum_{i=1}^{n}\left(x_i^2+x_i+1\right).$$

As n increases, the estimate converges to the true expectation $\mathbb{E}[f(x)]=3$, by the law of large numbers. results to estimate the expectation.

4.2 Variance of K-sample Estimator

Assume we want to estimate $\mathbb{E}[X]$ where X is a random variable. The K-sample Monte Carlo estimator is defined as:

$$\hat{\mu}_K = \frac{1}{K}\sum_{i=1}^{K} X_i$$

where $X_1, X_2, ..., X_K$ are independent and identically distributed (i.i.d.) samples from the distribution of X.

To find the variance of $\hat{\mu}_K$, we can use the following properties:

1. For i.i.d. random variables, the variance of their sum is the sum of their variances.
2. The variance of a constant times a random variable is the constant squared times the variance of the random variable.

Let's proceed:

$$\begin{aligned}
\mathrm{Var}(\hat{\mu}_K) &= \mathrm{Var}\left(\frac{1}{K}\sum_{i=1}^{K} X_i\right) \\
&= \frac{1}{K^2}\mathrm{Var}\left(\sum_{i=1}^{K} X_i\right) \\
&= \frac{1}{K^2}\sum_{i=1}^{K}\mathrm{Var}(X_i) \\
&= \frac{1}{K^2}\cdot K \cdot \mathrm{Var}(X) \\
&= \frac{\mathrm{Var}(X)}{K}
\end{aligned}$$

Therefore, the variance of the K-sample Monte Carlo estimator is:

$$\mathrm{Var}(\hat{\mu}_K) = \frac{\mathrm{Var}(X)}{K}$$

where $\mathrm{Var}(X)$ is the variance of the original random variable X.

This result shows that:

1. The variance of the estimator decreases as we increase the number of samples (K).

2. The rate of decrease is inversely proportional to K.

3. The variance of the estimator is always smaller than the variance of the original random variable (assuming $K > 1$).

This demonstrates why Monte Carlo methods become more accurate with more samples, and it quantifies exactly how the accuracy improves with sample size.

As an example, in the expectation of previous section, we have:

$$\hat{\mathbb{E}}_K = \frac{1}{K}\sum_{i=1}^{K} f(x_i)$$

where $x_i \sim N(0, 2)$ are independent and identically distributed (i.i.d.), and $f(x) = x^2 + x + 1$.

calculation of variance:

1. Since $x_i \sim N(0, 2)$, we first need to calculate the variance of the function $f(x) = x^2+x+1$. The variance of the function is given by:

$$\text{Var}(f(x)) = \mathbb{E}[f(x)^2] - (\mathbb{E}[f(x)])^2$$

We already computed $\mathbb{E}[f(x)] = 3$ in the previous section. Now we need to compute $\mathbb{E}[f(x)^2]$, which is:

$$\mathbb{E}[f(x)^2] = \mathbb{E}[(x^2 + x + 1)^2]$$

Expanding this expression:

$$(x^2 + x + 1)^2 = x^4 + 2x^3 + 3x^2 + 2x + 1$$

Thus, we need to compute the expectation of each term:

- $\mathbb{E}[x^4]$: Using the Moment Generating Function, $x \sim N(0, \sigma^2)$, $\mathbb{E}[x^4] = 3\sigma^4$. For $x \sim N(0, 2)$, $\sigma^2 = 2$, so:

$$\mathbb{E}[x^4] = 3 \cdot 2^2 = 12$$

- $\mathbb{E}[x^3]$: For a symmetric normal distribution (mean 0), all odd moments are 0, so:

$$\mathbb{E}[x^3] = 0$$

- $\mathbb{E}[x^2]$: This is just the variance of x, which is:

$$\mathbb{E}[x^2] = 2$$

- $\mathbb{E}[x]$: Since x is normally distributed with mean 0:

$$\mathbb{E}[x] = 0$$

- $\mathbb{E}[1]$: A constant, so:

$$\mathbb{E}[1] = 1$$

Therefore:

$$\mathbb{E}[(x^2 + x + 1)^2] = 12 + 0 + 3 \cdot 2 + 0 + 1 = 17$$

2. Now that we have $\mathbb{E}[f(x)^2] = 17$ and $\mathbb{E}[f(x)] = 3$, the variance of $f(x)$ is:

$$\text{Var}(f(x)) = 17 - 3^2 = 17 - 9 = 8$$

3. The Monte Carlo estimator is the average of K i.i.d. samples of $f(x)$. The variance of the average of K i.i.d. random variables is:

$$\text{Var}\left(\hat{\mathbb{E}}_K\right) = \frac{1}{K}\text{Var}(f(x))$$

Substituting $\text{Var}(f(x)) = 8$, we get:

$$\text{Var}\left(\hat{\mathbb{E}}_K\right) = \frac{8}{K}$$

4.3 Objective Minimization

To evaluate $F(\theta)$ and $\nabla F(\theta)$, we need to consider two parts: the sum and the regularization term.

For $F(\theta)$:

- We need to compute $f(\theta; n)$ for each n from 1 to N, and multiply by w_n.

- We then need to sum these N terms.
- Finally, we add $\lambda R(\theta)$.

Given that the function call of $f(\theta; n)$ is constant time, the complexity is $O(N)$ for the sum, plus the complexity of $R(\theta)$.

For $\nabla F(\theta)$:

- We need to compute $\nabla f(\theta; n)$ for each n from 1 to N, and multiply by w_n.
- We then need to sum these N terms.
- Finally, we add $\lambda \nabla R(\theta)$.

Again, given that the gradient evaluation of $f(\theta; n)$ is constant time, the complexity is $O(N)$ for the sum, plus the complexity of $\nabla R(\theta)$.

Therefore, the asymptotic complexity for both $F(\theta)$ and $\nabla F(\theta)$ is $O(N + C)$, where C is the complexity of evaluating $R(\theta)$ or $\nabla R(\theta)$. If $R(\theta)$ is a simple function (e.g., L2 regularization), C might be negligible compared to N, making the overall complexity effectively $O(N)$.

Monte Carlo Estimation

To use Monte Carlo estimation for an unbiased estimation of the objective and its gradient, we can follow these steps:

For the objective $F(\theta)$:

- Randomly sample M indices from $\{1, ..., N\}$ with replacement.
- Compute the estimate: $\hat{F}(\theta) = \frac{N}{M} \sum_{i=1}^{M} w_{n_i} f(\theta; n_i) + \lambda R(\theta)$

This is an unbiased estimator because:

$$\begin{aligned}
E[\hat{F}(\theta)] &= E\left[\frac{N}{M}\sum_{i=1}^{M} w_{n_i} f(\theta; n_i)\right] + \lambda R(\theta) \\
&= \frac{N}{M} \cdot M \cdot E[w_n f(\theta; n)] + \lambda R(\theta) \\
&= N \cdot E[w_n f(\theta; n)] + \lambda R(\theta) \\
&= \sum_{n=1}^{N} w_n f(\theta; n) + \lambda R(\theta) = F(\theta)
\end{aligned}$$

For the gradient $\nabla F(\theta)$:

- Use the same randomly sampled M indices.
- Compute the estimate: $\nabla \hat{F}(\theta) = \frac{N}{M}\sum_{i=1}^{M} w_{n_i} \nabla f(\theta; n_i) + \lambda \nabla R(\theta)$

This is also an unbiased estimator for similar reasons as above.

By using this Monte Carlo approach, we reduce the computational complexity from $O(N)$ to $O(M)$, where $M \ll N$. This can significantly speed up computations, especially for large datasets, at the cost of introducing some variance in the estimates.

The choice of M involves a trade-off between computation time and estimation accuracy. A larger M gives more accurate estimates but takes longer to compute.

A Appendix: Proof of Stationarity Condition for AR(1)

Consider the AR(1) process:

$$x_t = \phi x_{t-1} + \epsilon_t, \quad \epsilon_t \sim \mathcal{N}(0, \sigma^2)$$

where x_t is the value at time t, ϕ is the autoregressive coefficient, and ϵ_t is white noise.

Definition of Stationarity

A time series x_t is *stationary* if its mean, variance, and autocovariance are independent of time t. Specifically, we need:

- The mean $\mu = \mathbb{E}[x_t]$ is constant for all t,
- The variance $\gamma(0) = \mathrm{Var}(x_t) = \mathbb{E}[(x_t - \mu)^2]$ is finite and constant for all t,
- The autocovariance $\gamma(h) = \mathbb{E}[(x_t - \mu)(x_{t-h} - \mu)]$ depends only on the lag h and not on t.

Mean of the AR(1) Process

To find the mean of x_t, take the expectation of both sides of the AR(1) equation:

$$\mathbb{E}[x_t] = \phi \mathbb{E}[x_{t-1}] + \mathbb{E}[\epsilon_t]$$

Since $\mathbb{E}[\epsilon_t] = 0$, this simplifies to:

$$\mathbb{E}[x_t] = \phi \mathbb{E}[x_{t-1}]$$

For stationarity, the mean should be constant over time, so let $\mu = \mathbb{E}[x_t] = \mathbb{E}[x_{t-1}]$. Therefore, we have:

$$\mu = \phi \mu$$

This implies either $\mu = 0$ (for a zero-mean process), or $\phi = 1$, but the case $\phi = 1$ leads to non-stationarity, as we will show in the variance analysis.

Thus, the mean of the AR(1) process is $\mu = 0$, assuming the process is stationary.

Variance of the AR(1) Process

Now, let's compute the variance of x_t. Starting from the AR(1) equation:

$$x_t = \phi x_{t-1} + \epsilon_t$$

We square both sides to obtain:

$$x_t^2 = \phi^2 x_{t-1}^2 + 2\phi x_{t-1}\epsilon_t + \epsilon_t^2$$

Taking the expectation of both sides and using $\mathbb{E}[x_{t-1}\epsilon_t] = 0$ (since ϵ_t is white noise and independent of x_{t-1}):

$$\mathbb{E}[x_t^2] = \phi^2 \mathbb{E}[x_{t-1}^2] + \mathbb{E}[\epsilon_t^2]$$

Since $\mathbb{E}[\epsilon_t^2] = \sigma^2$, this becomes:

$$\text{Var}(x_t) = \phi^2 \text{Var}(x_{t-1}) + \sigma^2$$

For stationarity, the variance $\text{Var}(x_t)$ must be constant over time. Let $\gamma(0) = \text{Var}(x_t) = \text{Var}(x_{t-1})$, so we have:

$$\gamma(0) = \phi^2 \gamma(0) + \sigma^2$$

Solving for $\gamma(0)$:

$$\gamma(0)(1 - \phi^2) = \sigma^2$$

$$\Rightarrow \gamma(0) = \frac{\sigma^2}{1 - \phi^2}$$

For the variance $\gamma(0)$ to be finite and positive, we require:

$$1 - \phi^2 > 0 \quad \text{or} \quad |\phi| < 1$$

Thus, the condition $|\phi| < 1$ is necessary for the AR(1) process to be stationary. When $|\phi| \geq 1$, the variance $\gamma(0)$ either becomes infinite (for $|\phi| > 1$) or undefined (for $|\phi| = 1$), both of which lead to non-stationarity.

B Appendix: Derivation of the Autocovariance Function for an AR(1) Process

Consider an autoregressive model of order 1, denoted as AR(1):

$$x_t = \phi x_{t-1} + \epsilon_t, \quad \epsilon_t \sim \mathcal{N}(0, \sigma^2)$$

where:

- x_t is the value at time t,
- ϕ is the autoregressive coefficient,
- ϵ_t is white noise with zero mean and variance σ^2, and
- x_0 is a fixed known value.

We wish to derive the autocovariance function $\gamma(h) = \mathbb{E}[x_t x_{t-h}]$, which measures the covariance between x_t and x_{t-h}.

Stationary Condition for AR(1) Process

For the AR(1) process to be stationary, the parameter ϕ must satisfy $|\phi| < 1$. Under stationarity, the autocovariance $\gamma(h)$ depends only on the time lag h, and not on t itself.

Autocovariance at Lag 0

The variance of x_t, denoted as $\gamma(0) = \mathbb{E}[x_t^2]$, can be computed as follows. Using the AR(1) equation:

$$x_t = \phi x_{t-1} + \epsilon_t$$

Squaring both sides:

$$x_t^2 = \phi^2 x_{t-1}^2 + 2\phi x_{t-1}\epsilon_t + \epsilon_t^2$$

Taking the expectation of both sides, and noting that $\mathbb{E}[x_{t-1}\epsilon_t] = 0$ (since x_{t-1} and ϵ_t are independent):

$$\mathbb{E}[x_t^2] = \phi^2 \mathbb{E}[x_{t-1}^2] + \mathbb{E}[\epsilon_t^2]$$

Since $\mathbb{E}[\epsilon_t^2] = \sigma^2$, this becomes:

$$\gamma(0) = \phi^2\gamma(0) + \sigma^2$$

Solving for $\gamma(0)$:

$$\gamma(0) = \frac{\sigma^2}{1-\phi^2}$$

Autocovariance at Lag h

For $h \geq 1$, we use the fact that $x_t = \phi x_{t-1} + \epsilon_t$. The autocovariance at lag h is given by:

$$\gamma(h) = \mathbb{E}[x_t x_{t-h}]$$

Substitute the AR(1) process for x_t:

$$\gamma(h) = \mathbb{E}[(\phi x_{t-1} + \epsilon_t)x_{t-h}]$$

Since ϵ_t is independent of x_{t-h}, we have:

$$\gamma(h) = \phi\mathbb{E}[x_{t-1}x_{t-h}] + \mathbb{E}[\epsilon_t x_{t-h}]$$

Again, $\mathbb{E}[\epsilon_t x_{t-h}] = 0$, so this simplifies to:

$$\gamma(h) = \phi\mathbb{E}[x_{t-1}x_{t-h}] = \phi\gamma(h-1)$$

By recursively applying this formula, we get:

$$\gamma(h) = \phi^h\gamma(0)$$

Substitute $\gamma(0) = \frac{\sigma^2}{1-\phi^2}$ into the equation for $\gamma(h)$:

$$\gamma(h) = \phi^h\frac{\sigma^2}{1-\phi^2}$$

Thus, the autocovariance function for the AR(1) process is:

$$\gamma(h) = \frac{\sigma^2}{1-\phi^2}\phi^h$$

References

[1] A. Wiggers and E. Hoogeboom, "Predictive sampling with forecasting autoregressive models," in *International Conference on Machine Learning*, pp. 10260–10269, PMLR, 2020.