CS-844 Assignment 1: Technical Report

Neural Autoregressive Density Estimation (NADE)

Student: Muhammad Soban Shaukat **Registration:** 538822

Instructor: Dr. Zulqarnain Qayyum Khan

Submission Date: February 19, 2026

---

1 Introduction

This report explains the mathematical foundation of Neural Autoregressive Density Estimation (NADE) and demonstrates how our PyTorch implementation directly corresponds to the theoretical framework. NADE models complex probability distributions over high-dimensional binary data by decomposing the joint distribution into tractable conditional probabilities using the chain rule of probability.

2 Mathematical Foundation

2.1 Autoregressive Factorization

For a binary image $\mathbf{x} = (x_1, x_2, \ldots, x_{784})$ where each pixel $x_i \in \{0, 1\}$, NADE uses the chain rule to factorize the joint probability:

$$p(\mathbf{x}) = \prod_{d=1}^{784} p(x_d \mid x_{<d}) \tag{1}$$

where $x_{<d} = (x_1, \ldots, x_{d-1})$ represents all pixels before position d in raster scan order.

Implementation: The `sample()` method generates pixels sequentially in a `for` loop over 784 dimensions, ensuring each x_d depends only on previously generated pixels $x_1, \ldots, x_{d-1}$.

2.2 Conditional Probability with Neural Networks

Each conditional distribution is modeled as a Bernoulli distribution with probability computed using a single-layer neural network:

$$p(x_d = 1 \mid x_{<d}) = \sigma\left(\mathbf{V}_d^T \mathbf{h}_d + b_d\right) \tag{2}$$

where $\sigma(z) = 1/(1 + e^{-z})$ is the sigmoid function, $\mathbf{V}_d \in \mathbb{R}^H$ is the output weight vector for dimension d, b_d is the output bias, and $\mathbf{h}_d \in \mathbb{R}^H$ is the hidden state encoding information from $x_{<d}$.

Implementation:

```
logit = torch.matmul(h, self.V[d]) + self.b[d]
prob_d = torch.sigmoid(logit)
```

2.3 Hidden State Computation

The hidden state is computed recursively to accumulate information from previous pixels:

$$\mathbf{a}_d = \mathbf{c} + \sum_{i=1}^{d-1} \mathbf{W}_{:,i} \cdot x_i, \quad \mathbf{h}_d = \tanh(\mathbf{a}_d) \tag{3}$$

where $\mathbf{W} \in \mathbb{R}^{H \times D}$ is the input-to-hidden weight matrix, $\mathbf{c} \in \mathbb{R}^H$ is the hidden bias, and $\mathbf{a}_d$ is the pre-activation accumulator. The incremental update is $\mathbf{a}_{d+1} = \mathbf{a}_d + \mathbf{W}_{:,d} \cdot x_d$.

Implementation:

```
a = self.c.unsqueeze(0).repeat(n_samples, 1)  # Initialize
for d in range(self.input_dim):
    h = torch.tanh(a)                          # Hidden state
    # ... compute probability and sample ...
    a = a + samples[:, d].unsqueeze(1) * self.W[:, d]  # Update
```

3 Weight Tying and Parameter Efficiency

NADE employs **weight tying** where the same weight matrix $\mathbf{W}$ is shared across all conditionals. This reduces parameters from quadratic $O(D^2)$ in FVSBN to linear $O(D \cdot H)$ in NADE:

Parameter	Shape	Count
$\mathbf{W}$ (input $\to$ hidden)	500×784	392,000
$\mathbf{V}$ (hidden $\to$ output)	784×500	392,000
$\mathbf{c}$ (hidden bias)	500	500
$\mathbf{b}$ (output bias)	784	784
Total		**785,284**

This weight sharing enables efficient $O(D)$ time complexity for probability evaluation and sampling.

4 Ancestral Sampling

To generate new samples, NADE uses **ancestral sampling**:

1. Initialize: $\mathbf{a}_1 = \mathbf{c}$, $\mathbf{h}_1 = \tanh(\mathbf{a}_1)$
2. For $d = 1, \ldots, 784$:
   - Compute $p_d = \sigma(\mathbf{V}_d^T \mathbf{h}_d + b_d)$
   - Sample $x_d \sim \text{Bernoulli}(p_d)$
   - Update $\mathbf{a}_{d+1} = \mathbf{a}_d + \mathbf{W}_{:,d} \cdot x_d$
3. Return $\mathbf{x} = (x_1, \ldots, x_{784})$

Implementation: The `sample()` method uses `torch.bernoulli(prob_d)` to stochastically sample each pixel given its conditional probability.

5 Log-Probability Computation

For density estimation, the log-probability of an observed sample $\mathbf{x}$ is computed as:

$$\log p(\mathbf{x}) = \sum_{d=1}^{784} [x_d \log p_d + (1 - x_d) \log(1 - p_d)] \tag{4}$$

where $p_d = p(x_d = 1 \mid x_{<d})$.

Implementation:

```
log_prob += x[:, d] * torch.log(prob_d + 1e-8) + \
            (1 - x[:, d]) * torch.log(1 - prob_d + 1e-8)
```

The small constant 10^{-8} prevents numerical instability from $\log(0)$.

6 Conclusion

Our implementation faithfully translates NADE's mathematical formulation into executable PyTorch code. The model uses autoregressive factorization with weight-tied neural networks to achieve parameter efficiency while maintaining the ability to represent complex distributions. The sequential nature of sampling (generating pixels one at a time) directly mirrors the conditional probability structure

$p(x_d \mid x_{<d})$. Without training, the model generates random binary patterns; with training on MNIST, it would learn to generate realistic digit images.