Adaptive Deep Bayesian Neural Network (ADBNN)

Technical Documentation and Implementation Guide

Ninan Sajeeth Philip
Artificial Intelligence Research and Intelligent Systems
Thelliyoor 689544, India

February 13, 2025

Contents

1 Introduction

1.1 Overview

The Adaptive Deep Bayesian Neural Network (ADBNN) is a sophisticated machine learning framework that combines Bayesian probability theory with adaptive learning techniques. This document provides a comprehensive technical overview of the implementation, architecture, and usage patterns.

1.2 Key Features

- Three model types: Histogram-based, Gaussian, and Invertible DBNN
- Adaptive learning with cardinality-based sample selection
- Memory-efficient batch processing with GPU acceleration
- Automatic feature pair generation and optimization
- Built-in cross-validation and model evaluation

2 System Architecture

2.1 High-Level Design

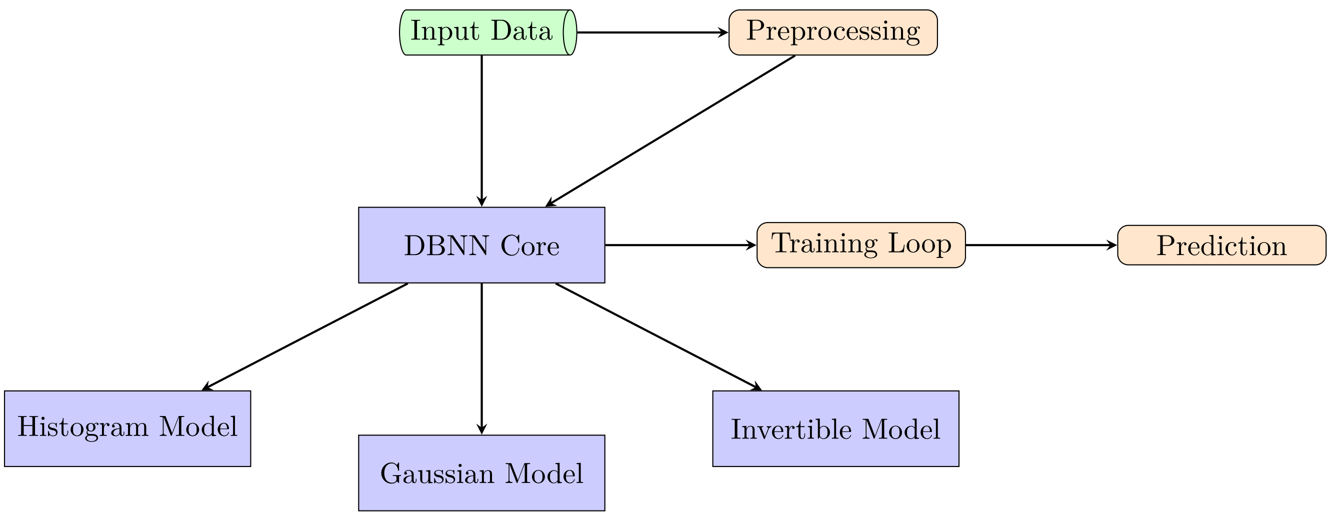

Figure 1: ADBNN System Architecture

2.2 Core Components

2.2.1 DBNN Base Class

The DBNN base class provides the foundation for all model implementations:

```python
class DBNN:
    def __init__(self, dataset_name: str, config: Optional[Union[GlobalConfig,
    Dict]] = None):
        self.dataset_name = dataset_name
        self.config = self._validate_config(config)
        self.device = self._setup_device_and_precision()
```

2.2.2 BinningHandler

The BinningHandler class manages data discretization and feature space binning:

```python
class BinningHandler:
    def __init__(self, n_bins_per_dim: int = 20,
                 padding_factor: float = 0.01,
                 device=None):
        self.n_bins = n_bins_per_dim
        self.padding_factor = padding_factor
        self.device = device or torch.device('cuda' if torch.cuda.is_available()
     else 'cpu')
```

3 Mathematical Foundation

3.1 Bayesian Framework

The ADBNN implements Bayesian inference through:

$$P(C|X) = \frac{P(X|C)P(C)}{P(X)} \tag{1}$$

where:

- $P(C|X)$ is the posterior probability
- $P(X|C)$ is the likelihood
- $P(C)$ is the prior probability
- $P(X)$ is the evidence

3.2 Feature Pair Likelihood

For feature pairs (i, j), the likelihood is computed as:

$$P(X_{i,j}|C) = \prod_{k=1}^{n} P(x_i^k, x_j^k|C) \tag{2}$$

4 Implementation Details

4.1 Memory Management

4.1.1 GPU Memory Optimization

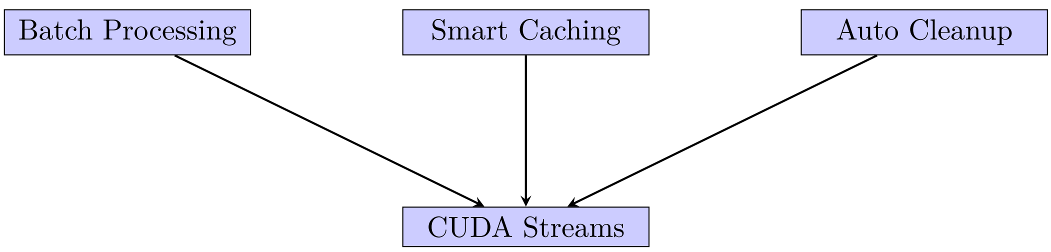

Figure 2: Memory Management Strategies

4.2 Adaptive Learning Process

The adaptive learning process follows:

Algorithm 1 Adaptive Learning

1: Initialize model with uniform priors
2: **while** not converged **do**
3: Select samples based on cardinality
4: Update feature pairs likelihood
5: Compute posterior probabilities
6: Update weights based on errors
7: Check convergence criteria
8: **end while**

5 Model Types

5.1 Histogram-based Model

The histogram-based model implements a non-parametric approach to probability density estimation through adaptive binning.

5.1.1 Bin Selection

The bin selection process is governed by:

$$n_{bins} = \min\left(\max\left(\sqrt{N}, 20\right), \frac{N}{10}\right) \tag{3}$$

where N is the number of samples in the dataset.

```python
def _compute_optimal_bins(self, n_samples: int) -> int:
    min_bins = 20  # Minimum number of bins
    max_bin_ratio = 10  # Maximum ratio of samples to bins

    optimal_bins = int(np.sqrt(n_samples))
    optimal_bins = max(optimal_bins, min_bins)
    optimal_bins = min(optimal_bins, n_samples // max_bin_ratio)

    return optimal_bins
```

Listing 1: Bin Selection Implementation

5.2 Gaussian Model

The Gaussian model uses multivariate normal distributions to model feature pair relationships:

$$P(X|C) = \frac{1}{(2\pi)^{d/2}|\Sigma|^{1/2}} \exp\left(-\frac{1}{2}(X-\mu)^T\Sigma^{-1}(X-\mu)\right) \tag{4}$$

```python
def _compute_gaussian_likelihood(self, features: torch.Tensor,
                                 mean: torch.Tensor,
                                 cov: torch.Tensor) -> torch.Tensor:
    dim = features.shape[1]
    centered = features - mean.unsqueeze(0)
    inv_cov = torch.inverse(cov)
    mahalanobis = torch.sum(
        torch.mm(centered, inv_cov) * centered,
        dim=1
    )
    det = torch.det(cov)
    norm_const = 1.0 / (torch.sqrt((2 * torch.pi) ** dim * det))
    return norm_const * torch.exp(-0.5 * mahalanobis)
```

Listing 2: Gaussian Model Implementation

5.3 Invertible DBNN

The invertible model adds bi-directional mapping capabilities:

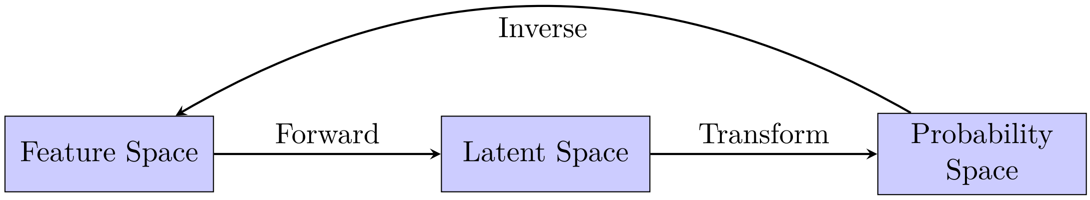

Figure 3: Invertible DBNN Architecture

6 Adaptive Learning

6.1 Sample Selection

The sample selection process uses cardinality-based criteria:

$$C(x) = \sum_{i,j} \min_{y \in S} d((x_i, x_j), (y_i, y_j)) \tag{5}$$

where:

- $C(x)$ is the cardinality score
- S is the set of selected samples
- $d(\cdot, \cdot)$ is the Euclidean distance

```python
def _compute_sample_divergence(self, sample_data: torch.Tensor,
                               feature_pairs: List[Tuple]) -> torch.Tensor:
    n_samples = sample_data.shape[0]
    pair_distances = torch.zeros(
        (len(feature_pairs), n_samples, n_samples),
        device=self.device
    )

    for i, pair in enumerate(feature_pairs):
        pair_data = sample_data[:, pair]
        diff = pair_data.unsqueeze(1) - pair_data.unsqueeze(0)
        pair_distances[i] = torch.norm(diff, dim=2)

    return torch.mean(pair_distances, dim=0)
```

Listing 3: Cardinality Computation

6.2 Weight Updates

The weight update process follows:

$$w_{t+1} = w_t + \eta \cdot \Delta w \tag{6}$$

where:

- w_t is the current weight
- η is the learning rate

- Δw is the weight update

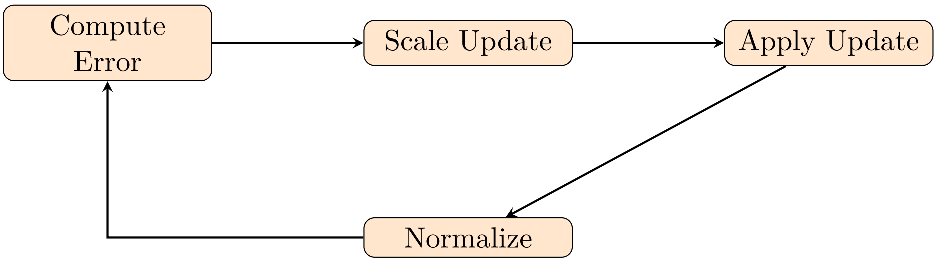

Figure 4: Weight Update Process

7 GPU Optimization

7.1 Batch Processing

The batch processing system uses dynamic batch sizing:

$$B_{opt} = \min\left(\frac{M_{avail}}{4 \cdot S_{sample}}, B_{max}\right) \tag{7}$$

where:

- B_{opt} is the optimal batch size
- M_{avail} is available GPU memory
- S_{sample} is sample size in bytes
- B_{max} is maximum allowed batch size

```python
def _calculate_optimal_batch_size(self, sample_tensor_size):
    if not torch.cuda.is_available():
        return 128  # Default for CPU

    total_memory = torch.cuda.get_device_properties(0).total_memory
    reserved_memory = torch.cuda.memory_reserved(0)
    allocated_memory = torch.cuda.memory_allocated(0)

    available_memory = (total_memory - reserved_memory -
                        allocated_memory) * 0.8

    memory_per_sample = sample_tensor_size * 4
    optimal_batch_size = int(available_memory / memory_per_sample)

    return max(32, min(optimal_batch_size, 512))
```

Listing 4: Dynamic Batch Sizing

8 Cache Management

8.1 Computation Cache

The computation cache implements an LRU strategy:

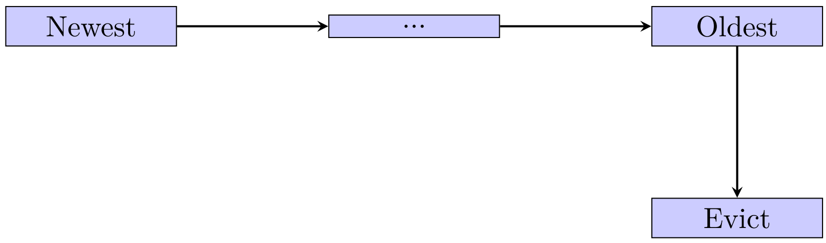

Figure 5: Cache Management Strategy

9 Parallel Processing Architecture

9.1 CUDA Implementation

The ADBNN framework implements sophisticated CUDA optimization strategies for parallel processing. The core parallel processing architecture is built around efficient tensor operations and memory management.

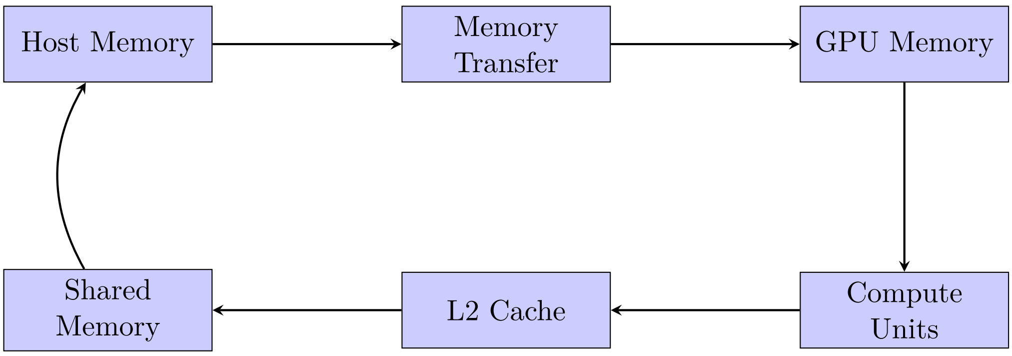

Figure 6: CUDA Memory Hierarchy and Data Flow

9.2 Memory Management Strategies

Memory management is crucial for performance optimization. The system implements several key strategies:

```
class MemoryManager:
    def __init__(self, device):
        self.device = device
        self.memory_pool = {}
        self.allocation_threshold = 1e9  # 1GB

    def allocate_tensor(self, shape, dtype):
        size = np.prod(shape) * dtype.itemsize
        if size > self.allocation_threshold:
            return self._allocate_chunked(shape, dtype)
        return torch.empty(shape, dtype=dtype,
                           device=self.device)

    def _allocate_chunked(self, shape, dtype):
        chunk_size = self.allocation_threshold // dtype.itemsize
        chunks = []
        for i in range(0, np.prod(shape), chunk_size):
            chunk_shape = self._calculate_chunk_shape(
                i, chunk_size, shape)
            chunk = torch.empty(chunk_shape, dtype=dtype,
                                device=self.device)
            chunks.append(chunk)
```

```
        return chunks
```

Listing 5: Memory Management Implementation

9.3 Batch Processing Optimization

Efficient batch processing is implemented through dynamic batch sizing and parallel execution:

$$B_{opt} = \min\left(\frac{M_{GPU}}{4 \cdot S_{sample} \cdot F_{safety}}, B_{max}\right) \tag{8}$$

where:

- M_{GPU} is total GPU memory
- S_{sample} is sample memory footprint
- F_{safety} is safety factor (typically 1.2)
- B_{max} is maximum allowed batch size

10 Numerical Stability

10.1 Log-Space Computations

To maintain numerical stability, probability computations are performed in log space:

$$\log P(C|X) = \log P(X|C) + \log P(C) - \log P(X) \tag{9}$$

Implementation details:

```
def _compute_log_posterior(self, features: torch.Tensor,
                           epsilon: float = 1e-10) -> torch.Tensor:
    # Compute log likelihoods
    log_likelihoods = self._compute_log_likelihood(features)

    # Add log priors
    log_priors = torch.log(self.priors + epsilon)
    log_unnormalized = log_likelihoods + log_priors

    # Log-sum-exp trick for numerical stability
    max_log = torch.max(log_unnormalized, dim=1,
                       keepdim=True)[0]
    log_sum = max_log + torch.log(torch.sum(
        torch.exp(log_unnormalized - max_log), dim=1,
        keepdim=True) + epsilon)

    return log_unnormalized - log_sum
```

Listing 6: Log-Space Computation

10.2 Gradient Scaling

For training stability, gradients are scaled using:

$$g_{scaled} = \text{clip}\left(\frac{g}{\|g\|_2 + \epsilon}, -\alpha, \alpha\right) \tag{10}$$

where α is the maximum gradient norm.

11 CUDA Stream Management

11.1 Asynchronous Execution

The system implements asynchronous execution through CUDA streams:

```python
class StreamManager:
    def __init__(self, n_streams=4):
        self.streams = [
            torch.cuda.Stream()
            for _ in range(n_streams)
        ]
        self.current_stream = 0

    def get_next_stream(self):
        stream = self.streams[self.current_stream]
        self.current_stream = ((self.current_stream + 1)
                                % len(self.streams))
        return stream

    @contextmanager
    def stream_context(self):
        stream = self.get_next_stream()
        with torch.cuda.stream(stream):
            yield stream
            torch.cuda.current_stream().wait_stream(stream)
```

Listing 7: CUDA Stream Management

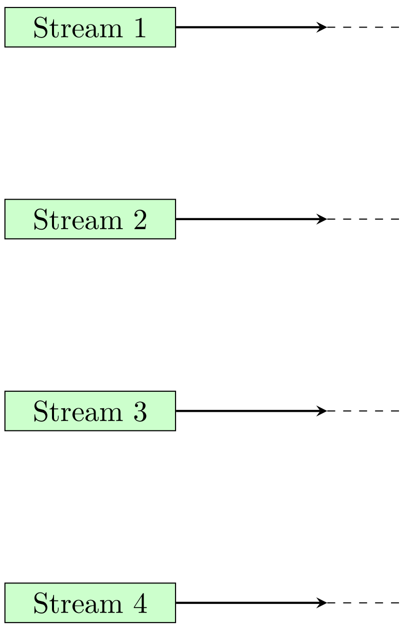

Figure 7: Asynchronous Stream Execution

11.2 Memory Transfers

Efficient memory transfers are crucial for performance:

```python
def _transfer_to_gpu(self, data: torch.Tensor,
                     stream: Optional[torch.cuda.Stream] = None):
    if not torch.cuda.is_available():
        return data

    with torch.cuda.stream(stream) if stream else nullcontext():
```

```
        if not data.is_cuda:
            data = data.cuda(non_blocking=True)

        if not data.is_contiguous():
            data = data.contiguous()

    return data
```

Listing 8: Optimized Memory Transfer

12 Feature Pair Generation

12.1 Pair Selection Algorithm

The feature pair selection process uses an information-theoretic approach:

$$I(X_i, X_j; C) = \sum_{c \in C} \sum_{x_i, x_j} P(x_i, x_j, c) \log \frac{P(x_i, x_j | c)}{P(x_i, x_j)} \tag{11}$$

Implementation:

```
def _select_feature_pairs(self, X: torch.Tensor,
                          y: torch.Tensor,
                          max_pairs: int) -> List[Tuple[int, int]]:
    n_features = X.shape[1]
    pairs_info = []

    # Compute mutual information for all pairs
    for i, j in combinations(range(n_features), 2):
        mi = self._compute_mutual_information(
            X[:, [i, j]], y)
        pairs_info.append((i, j, mi))

    # Sort by mutual information
    pairs_info.sort(key=lambda x: x[2], reverse=True)

    # Select top pairs
    selected_pairs = [
        (p[0], p[1])
        for p in pairs_info[:max_pairs]
    ]

    return selected_pairs
```

Listing 9: Feature Pair Selection

12.2 Feature Space Transformation

Features are transformed using adaptive binning:

$$b_{ij} = \left\lfloor \frac{x_{ij} - \min(x_j)}{\max(x_j) - \min(x_j)} \cdot n_{bins} \right\rfloor \tag{12}$$

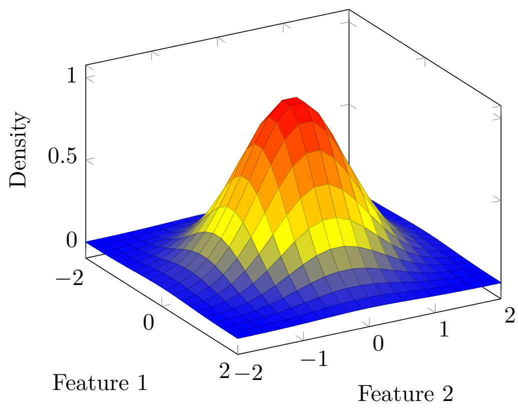

Figure 8: Feature Space Density Estimation

13 Adaptive Learning Implementation

13.1 Sample Selection Strategy

14 Advanced Training Strategies

14.1 Convergence Criteria

The training process employs multiple convergence criteria:

$$\Delta E = \frac{|E_t - E_{t-1}|}{E_{t-1}} < \epsilon_{conv} \tag{13}$$

where:

- E_t is the error at epoch t
- ϵ_{conv} is the convergence threshold

```python
def _check_convergence(self,
                       current_error: float,
                       error_history: List[float],
                       patience: int = 5,
                       min_improvement: float = 0.001) -> bool:
    if len(error_history) < patience:
        return False

    recent_errors = error_history[-patience:]
    improvements = [
        abs(recent_errors[i] - recent_errors[i-1]) / recent_errors[i-1]
        for i in range(1, len(recent_errors))
    ]

    return all(imp < min_improvement for imp in improvements)
```

Listing 10: Convergence Check Implementation

14.2 Adaptive Learning Rate

The learning rate is adjusted dynamically:

$$\eta_t = \eta_0 \cdot \frac{1}{1 + \alpha t} \cdot \sqrt{\frac{1 - \beta^t}{1 - \beta}} \tag{14}$$

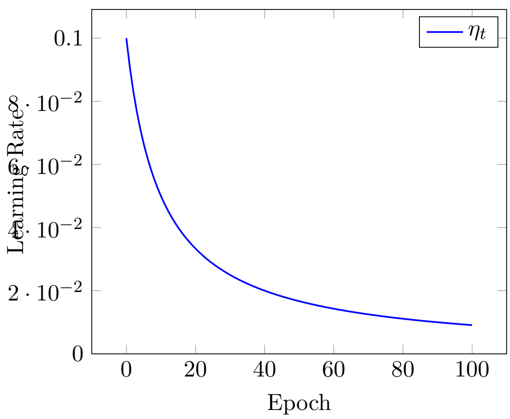

Figure 9: Adaptive Learning Rate Decay

15 Cross-Validation Strategy

15.1 K-Fold Implementation

The system implements stratified k-fold cross-validation:

```python
class StratifiedKFold:
    def __init__(self, n_splits: int = 5,
                 shuffle: bool = True,
                 random_state: Optional[int] = None):
        self.n_splits = n_splits
        self.shuffle = shuffle
        self.random_state = random_state

    def split(self, X: torch.Tensor,
              y: torch.Tensor) -> Iterator[Tuple[torch.Tensor,
                                                 torch.Tensor]]:
        # Get class distribution
        unique_classes = torch.unique(y)
        class_indices = {
            cls.item(): (y == cls).nonzero().view(-1)
            for cls in unique_classes
        }

        # Create stratified folds
        for fold in range(self.n_splits):
            train_indices = []
            val_indices = []

            for cls, indices in class_indices.items():
                n_samples = len(indices)
                n_val = n_samples // self.n_splits
                start_idx = fold * n_val
                end_idx = start_idx + n_val

                val_indices.extend(indices[start_idx:end_idx])
                train_indices.extend(
                    torch.cat([
                        indices[:start_idx],
                        indices[end_idx:]
                    ])
                )

            yield train_indices, val_indices
```

Listing 11: Cross-Validation Implementation

16 Weight Update Mechanisms

16.1 Momentum-based Updates

The weight update process incorporates momentum:

$$v_t = \gamma v_{t-1} + \eta \nabla w_t \tag{15}$$

$$w_{t+1} = w_t - v_t \tag{16}$$

Implementation details:

```python
def _update_weights_with_momentum(self,
                                gradients: Dict[str, torch.Tensor],
                                velocity: Dict[str, torch.Tensor],
                                learning_rate: float,
                                momentum: float = 0.9) -> Dict[str, torch.Tensor
    ]:
    new_velocity = {}
    updates = {}

    for param_name, grad in gradients.items():
        if param_name not in velocity:
            velocity[param_name] = torch.zeros_like(grad)

        # Update velocity
        new_velocity[param_name] = (
            momentum * velocity[param_name] +
            learning_rate * grad
        )

        # Compute update
        updates[param_name] = new_velocity[param_name]

    return updates, new_velocity
```

Listing 12: Momentum Update Implementation

17 Performance Metrics

17.1 Classification Metrics

The system computes comprehensive classification metrics:

$$\text{Balanced Accuracy} = \frac{1}{n_c} \sum_{i=1}^{n_c} \frac{TP_i}{TP_i + FN_i} \tag{17}$$

where: - n_c is the number of classes - TP_i is true positives for class i - FN_i is false negatives for class i

```python
def compute_metrics(self,
                   y_true: torch.Tensor,
                   y_pred: torch.Tensor,
                   probas: torch.Tensor) -> Dict[str, float]:
    metrics = {}

    # Compute confusion matrix
    cm = confusion_matrix(
        y_true.cpu(), y_pred.cpu()
    )

    # Class-wise metrics
```

```python
    n_classes = len(self.label_encoder.classes_)
    class_metrics = []

    for i in range(n_classes):
        tp = cm[i, i]
        fp = cm[:, i].sum() - tp
        fn = cm[i, :].sum() - tp

        precision = tp / (tp + fp) if (tp + fp) > 0 else 0
        recall = tp / (tp + fn) if (tp + fn) > 0 else 0
        f1 = 2 * (precision * recall) / (precision + recall) \
            if (precision + recall) > 0 else 0

        class_metrics.append({
            'precision': precision,
            'recall': recall,
            'f1': f1
        })

    # Overall metrics
    metrics['accuracy'] = accuracy_score(
        y_true.cpu(), y_pred.cpu()
    )
    metrics['balanced_accuracy'] = balanced_accuracy_score(
        y_true.cpu(), y_pred.cpu()
    )

    # ROC and AUC
    if probas is not None:
        metrics['roc_auc'] = roc_auc_score(
            y_true.cpu(),
            probas.cpu(),
            multi_class='ovr'
        )

    return metrics
```

Listing 13: Metrics Computation

17.2 Statistical Significance Testing

The system implements statistical significance tests:

```python
def perform_significance_test(self,
                              model1_preds: torch.Tensor,
                              model2_preds: torch.Tensor,
                              y_true: torch.Tensor,
                              alpha: float = 0.05) -> Dict[str, Any]:
    # McNemar's test for paired nominal data
    contingency_table = self._create_contingency_table(
        model1_preds, model2_preds, y_true
    )

    statistic, p_value = mcnemar(contingency_table,
                                  exact=True)

    return {
        'statistic': statistic,
        'p_value': p_value,
        'significant': p_value < alpha
    }
```

Listing 14: Statistical Testing

18 Error Analysis

18.1 Feature Importance Analysis

19 Custom Loss Functions

19.1 Weighted Cross-Entropy

20 Configuration Management

20.1 Dynamic Configuration

21 Common Issues and Solutions

21.1 Memory Management