Robust High-Frequency Volatility Forecasting with a Three-Headed Transformer

Raghuvar Nadig

1 Abstract

We forecast **short-term volatility** for a large intraday equity dataset using a novel **Three-Headed Transformer (THT)** model [Vaswani et al., 2017] that outputs both **volatility predictions** and **confidence measures**. Experimental results demonstrate that our Transformer not only significantly reduces forecast errors (MAE, MSE) relative to a naive baseline and an XGBoost approach but also achieves higher $\mathbf{R}^2$ and **rank correlation**, indicating a tighter fit to actual market dynamics. Interestingly, while the naive method outperforms XGBoost on basic error metrics, the Transformer provides the best overall performance, largely due to its ability to capture intricate microstructure patterns. Beyond prediction accuracy, the volatility and return confidence heads produce sensible confidence estimates that can be leveraged for **dynamic position scaling**, ultimately enhancing risk management and boosting trading performance.

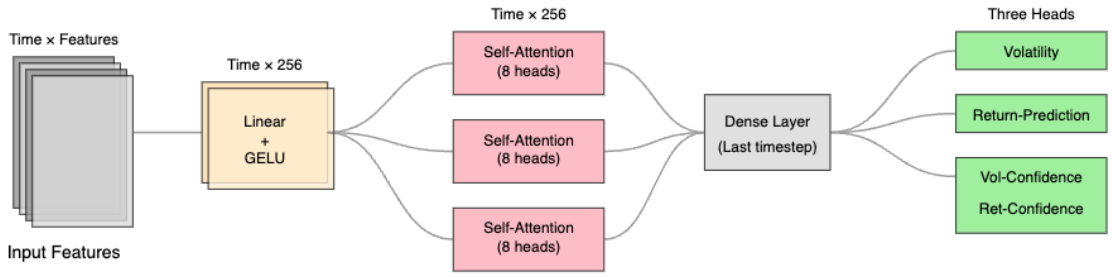
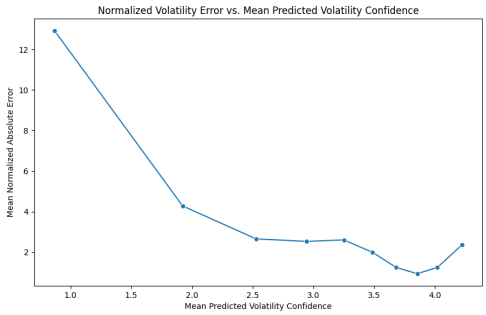


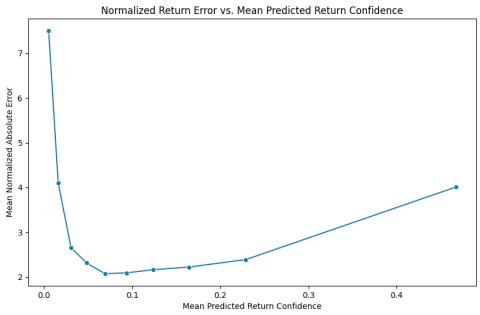
Figure 1: Three-headed Transformer architecture for price/volume feature processing and prediction. The network processes 30-minute sequences through an embedding layer and L-layer Transformer encoder before splitting into specialized prediction heads for volatility, uncertainty, and returns.

2 Key Results

- **Outperformance:** The Transformer model outperforms both a naive baseline and XGBoost in key performance metrics in out of sample data. It also outperforms a Multilayer Perceptron (MLP) network and a Convolutional Neural Network (CNN).
- **Sensible Confidence Calibration:** The model's calibrated confidence scores for volatility and return predictions are consistent with observed errors, suggesting their potential utility for scaling.
- **Scalability:** Leveraging self-attention, the model generates thousands of minute-level predictions concurrently.



(a) Normalized Volatility Error vs. Confidence



(b) Normalized Return Error vs. Confidence

Figure 2: Comparison of confidence calibration for volatility and return predictions.

2.1 Regime and Time-of-Day Analysis

Analysis across different volatility regimes and trading periods reveals key characteristics of our Transformer model:

- **Time-of-Day Dynamics:** Model performance is strongest in the morning (correlation 0.896 from 10:30am-12pm), with some degradation during mid-day trading. Late trading shows increased prediction error (correlation dropping to 0.784), though performance recovers somewhat into the close (0.875). This pattern suggests the model best captures morning volatility structures, when market activity is typically more directional.
- **Volatility Regime Behavior:** The model tends to compress predictions toward the mean:
  - Overestimates low volatility conditions (predicting 0.0295 bps vs actual 0.0176 bps in Very Low regime)
  - Shows high accuracy in normal conditions (0.0529 bps vs 0.0496 bps in Medium regime)
  - Underestimates high volatility events (0.158 bps vs 0.189 bps in Very High regime)
- **Confidence Measures:** Model confidence appropriately decreases as volatility increases, ranging from 2.17 in Very Low to 1.22 in Very High regimes, indicating well-calibrated uncertainty estimation.

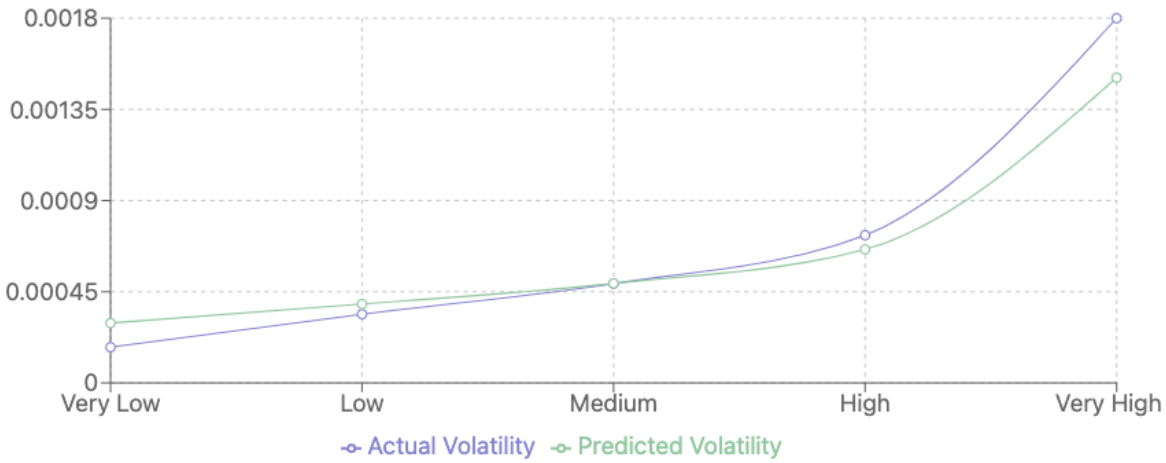


Figure 3: Predicted vs Actual Volatility across Different Trading Periods. Colors indicate volatility regimes.

This analysis suggests the model is most reliable during normal market conditions and early trading hours, with careful consideration needed when deploying it during extreme volatility events.

2.2 Market Open Considerations

Our Transformer's 30-minute sequence requirement creates a notable limitation during market open. The model cannot generate predictions until 10:00 AM ET, missing the first 30 minutes of trading when volatility is typically elevated. We propose either using simpler models (naive rolling volatility or XGBoost) during this period, or developing a specialized short-sequence model for market open. Given the distinct volatility patterns and heightened trading activity during this period, future work should focus on a hybrid approach that maintains prediction quality during these crucial early minutes.

3 Model Performance Comparison

3.1 Baseline Model

The persistent nature of intraday volatility [Andersen and Bollerslev, 1997] provides a strong naive baseline - using past 10-minute realized volatility to predict future volatility achieves correlations of 0.60-0.70, particularly during market open and high volatility periods. This sets a meaningful benchmark for more sophisticated approaches.

Our analysis spans one year of data from 2024-01-16 to 2025-01-14, divided into:

- Training: 2024-01-16 to 2024-06-30
- Validation: 2024-07-01 to 2024-08-31 (hyperparameter tuning)
- Test: 2024-09-01 to 2025-01-14 (reported results)

Our analysis focuses on a dynamic universe of the most actively traded 1000 stocks and ETFs, ranked daily by their rolling 10-day trading volume. This daily rebalancing approach ensures consistent liquidity and eliminates survivorship bias by allowing instruments to naturally enter and exit our universe based on their current trading activity. Performance metrics in Table 1 are derived exclusively from the test set, with model architecture and hyperparameters fixed after validation, maintaining strict separation between training and evaluation periods.

Model	MAE	MSE	RMSE	R^2	Avg Rank Corr
Naive	0.000329	7.29×10^{-7}	0.000854	0.4156	0.66417
XGBoost	0.000376	7.70×10^{-7}	0.000877	0.3834	0.63408
THT	**0.00023**	5.12×10^{-7}	**0.00072**	**0.4922**	**0.7618**

Table 1: Out-of-sample model performance metrics on the test set (2024-09-01 to 2025-01-14). Both XGBoost and the Transformer use the pruned feature set of 21 features.

4 Feature Importances

Using XGBoost, we ranked the pruned feature set for volatility prediction. The features that are most predictive of 10 minute volatility, sorted by their importance scores, are listed below with a one-line explanation for each:

- **roll_vol_5m (0.273):** Rolling volatility over the previous 5-minute window, capturing short-term market variability.
- **roll_vol_30m (0.186):** Rolling volatility over the previous 30-minute window, providing a broader perspective on market fluctuations.
- **vol_rank (0.072):** A normalized percentile ranking of each symbol's short-term average volatility.
- **log_ret_5m (0.065):** Logarithmic returns over the previous 5-minute interval, indicating short-term price movement.
- **vol_of_vol_5m (0.046):** The variability of volatility over a 5-minute window, quantifying volatility instability.

- **rel_vol (0.040):** The ratio of a symbol's short-term volatility to the market's short-term average.
- **log_turnover (0.039):** Log-transformed 10 day turnover, reflecting the intensity of trading activity.
- **log_ret_1m (0.036):** Logarithmic returns over a 1-minute interval, capturing very short-term price changes.
- **time_cos (0.036):** Cosine-transformed time feature, encoding cyclical intraday patterns.
- **mid (0.034):** Midprice (the average of bid and ask prices), a proxy for the fair market value.
- **imbalance (0.032):** Order book imbalance, indicating the relative pressure of buyers versus sellers.
- **spread_ma_10m (0.027):** A 10-minute moving average of the bid-ask spread.
- **spread (0.018):** Bid-ask spread.
- **normalized_time (0.015):** A normalized time variable to capture intraday temporal effects.
- **bid_sz_00 (0.014):** The size of the best bid, reflecting immediate buying interest.
- **time_sin (0.014):** Sine-transformed time, complementing the cosine feature for cyclical effects.
- **ask_sz_00 (0.014):** The size of the best ask, representing immediate selling pressure.
- **vol_regime (0.011):** A categorical indicator of volatility regimes, the ratio of the market short-window average to the market long-window average.
- **imbalance_ma_10m (0.011):** A 10-minute moving average of order imbalance.
- **avg_nn_log_ret_5m (0.008):** The average 5 minute log returns of the 5 nearest neighbors of the stock.
- **avg_nn_roll_vol_5m (0.006):** The average 5 minute volatility of the 5 nearest neighbors of the stock.

5 Methodology

We used one year of minutely-sampled NASDAQ ITCH data for approximately 1,000 equities [Zhang et al., 2019]. After extracting key microstructure features, we initially developed models with 55 features and subsequently pruned to 21. Both XGBoost and our three-headed Transformer utilized these features; additional experiments with Multi-Layer Perceptron (MLP) and Convolutional Neural Networks (CNN) did not outperform XGBoost in-sample. The key models are:

- **XGBoost Baseline:** A tree-based model that takes advantage of the pruned feature set to predict volatility.
- **Three-Headed Transformer (THT):** Processes a 30-minute input sequence via self-attention, simultaneously predicting 10-minute-ahead volatility, its associated confidence, and return forecasts with separate confidence scores.

To maintain data hygiene and prevent lookahead bias, missing quotes are forward-filled with explicit flags, and early market predictions use appropriately scaled shorter-window measurements. We implement a conservative evaluation lag of 1-minute for regression and XGBoost models and deep learning models, despite significantly shorter inference times (sub 1 second). Training and inference was performed on a modular, cloud-based, GPU-accelerated framework using AWS S3 storage and PyTorch with AMP.

6 Model Architecture and Training

6.1 Three-Headed Transformer Architecture

Our model employs a Transformer-based architecture with three specialized prediction heads, optimized to capture short-term market microstructure patterns while providing calibrated uncertainty estimates. The network processes 30-minute sequences of 21 features through several key components:

6.1.1 Input Processing and Embedding

The input features are first projected from their raw dimension ($d = 21$) to a higher-dimensional representation ($H = 256$) via a linear embedding layer. To preserve temporal information, we add sinusoidal positional encodings:

$$PE(t, 2i) = \sin\left(\frac{t}{10000^{\frac{2i}{H}}}\right), \quad PE(t, 2i+1) = \cos\left(\frac{t}{10000^{\frac{2i}{H}}}\right) \tag{1}$$

6.1.2 Transformer Encoder Configuration

The encoder leverages self-attention to dynamically weight the importance of different time steps and features, particularly valuable for capturing varying relevance of past volatility patterns under different market conditions. The architecture consists of:

Parameter	Value
Number of Layers (L)	4
Attention Heads (n)	8
Hidden Dimension (H)	256
Intermediate FF Dimension	512
Dropout Rate	0.15 (attention), 0.1 (FF)
Layer Normalization	Pre-norm configuration
Activation	GELU

Table 2: Transformer encoder hyperparameters

6.1.3 Prediction Heads

The final encoder representation feeds into three specialized prediction heads:

- **Volatility Head:** A two-layer MLP (256→128→1) predicting volatility (μ)
- **Log-Variance Head:** Parallel MLP predicting confidence ($\log \sigma^2$)
- **Returns Head:** Two-branch MLP outputting returns and confidence scores

6.2 Training Protocol

Training Parameter	Value/Configuration
Optimizer	AdamW ($\beta_1 = 0.9$, $\beta_2 = 0.999$)
Learning Rate	1e-4 with cosine decay
Batch Size	1024 sequences
Gradient Clipping	1.0
Weight Decay	0.01
Training Hardware	NVIDIA RTX 4090 (24GB)
Training Time	2-3 minutes per epoch

Table 3: Training hyperparameters and hardware details

6.3 Loss Function and Calibration

A central innovation in our approach is the custom loss function that minimizes prediction error while calibrating the network's confidence [Kendall and Gal, 2017]. The volatility loss is based on a heteroscedastic Gaussian negative log-likelihood:

$$\mathcal{L}_{\text{vol}}(\mu, \log \sigma^2; y) = \frac{1}{2} \log \sigma^2 + \frac{(y - \mu)^2}{2 \exp(\log \sigma^2)} + C, \tag{2}$$

where y is the true volatility, $\sigma^2 = \exp(\log \sigma^2)$, and $C = 0.5 \log(2\pi)$.

To encourage well-calibrated confidence estimates, we introduce a calibration penalty. We define a target confidence:

$$c_{\text{target}} = \frac{1}{1 + |y - \mu|}, \tag{3}$$

and penalize deviations between the predicted confidence, $\exp(-\log \sigma^2)$, and c_{target}:

$$\mathcal{L}_{\text{calib}} = \gamma \left(\exp(-\log \sigma^2) - c_{\text{target}}\right)^2. \tag{4}$$

The total volatility loss becomes:

$$\mathcal{L}_{\text{vol}}^{\text{total}} = \mathcal{L}_{\text{vol}} + \mathcal{L}_{\text{calib}}. \tag{5}$$

Intuition: This loss function not only penalizes large prediction errors but also adjusts the model's confidence based on the magnitude of the error. In essence, when the absolute error is low, the model is encouraged to be confident (and vice versa). In parallel, our return predictions are trained with a confidence-weighted MSE loss that includes an entropy penalty—ensuring the model avoids overconfidence.

For full implementation details, please refer to the accompanying PyTorch code listings of the various loss functions.

7 Conclusion and Future Work

Our multi-headed attention model jointly predicts returns and volatility—along with their confidence scores—providing a risk-aware forecast. The architecture not only outperforms both the naive and XGBoost baselines but also produces volatility and return confidence outputs that seem sensible for use in scaling trading positions.

Future work will extend this research by:

- Implementing rolling fits to adapt to evolving market dynamics.
- Incorporating more data, additional features, and multi-year datasets.
- Conducting broader hyperparameter searches and exploring deeper, alternative network architectures (e.g., temporal networks).

References

Torben G Andersen and Tim Bollerslev. Intraday periodicity and volatility persistence in financial markets. *Journal of Empirical Finance*, 4(2-3):115–158, 1997.

Alex Kendall and Yarin Gal. What uncertainties do we need in Bayesian deep learning for computer vision? *Advances in neural information processing systems*, 30, 2017.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

Michael Zhang, Bryan Li, and Ryan Smith. Deep learning for high-frequency trading. *Journal of Financial Data Science*, 1(4):100–120, 2019.

A Code Listings

Below are the cleaned up versions of the key classes used in our model implementation.

A.1 ThreeHeadedTransformer

```python
import math
import torch
import torch.nn as nn

class ThreeHeadedTransformer(nn.Module):
    def __init__(self,
                 input_dim: int,
                 hidden_dim: int = 256,
                 nhead: int = 8,
                 num_layers: int = 3,
                 dropout: float = 0.1,
                 max_seq_length: int = 100,
                 gamma: float = 0.1):
        super().__init__()
        self.embedding = nn.Linear(input_dim, hidden_dim)
        self.norm = nn.LayerNorm(hidden_dim)
        self.activation = nn.GELU()

        # Create positional encoding
        position = torch.arange(max_seq_length).unsqueeze(1)
        div_term = torch.exp(torch.arange(0, hidden_dim, 2) * (-math.log
            (10000.0) / hidden_dim))
        pe = torch.zeros(1, max_seq_length, hidden_dim)
        pe[0, :, 0::2] = torch.sin(position * div_term)
        pe[0, :, 1::2] = torch.cos(position * div_term)
        self.register_buffer('position_encoding', pe)

        encoder_layer = nn.TransformerEncoderLayer(
            d_model=hidden_dim,
            nhead=nhead,
            dropout=dropout,
            batch_first=True
        )
        self.transformer_encoder = nn.TransformerEncoder(encoder_layer,
            num_layers=num_layers)

        # Define three-headed outputs
        self.return_head = nn.Sequential(
            nn.Linear(hidden_dim, hidden_dim // 2),
            nn.GELU(),
            nn.Dropout(dropout),
            nn.Linear(hidden_dim // 2, 1)
        )
        self.volatility_head = nn.Sequential(
            nn.Linear(hidden_dim, hidden_dim // 2),
            nn.GELU(),
            nn.Dropout(dropout),
            nn.Linear(hidden_dim // 2, 1)
        )
        self.vol_logvar_head = nn.Sequential(
            nn.Linear(hidden_dim, hidden_dim // 2),
            nn.GELU(),
            nn.Dropout(dropout),
            nn.Linear(hidden_dim // 2, 1)
        )
        self.ret_confidence_head = nn.Sequential(
            nn.Linear(hidden_dim, hidden_dim // 2),
            nn.GELU(),
            nn.Dropout(dropout),
            nn.Linear(hidden_dim // 2, 1)
        )
```

```
        self._initialize_heads()

    def _initialize_heads(self):
        """
        Initialize final layers:
         - Logvar head: zero initialization for stable variance (predicts 1).
         - Return confidence head: zero initialization (predicts sigmoid(0)
            =0.5).
         - Other heads use default initialization.
        """
        nn.init.zeros_(self.vol_logvar_head[-1].weight)
        nn.init.zeros_(self.vol_logvar_head[-1].bias)
        nn.init.zeros_(self.ret_confidence_head[-1].weight)
        nn.init.zeros_(self.ret_confidence_head[-1].bias)

    def forward(self, x: torch.Tensor) -> dict:
        # x: (batch_size, seq_len, input_dim)
        x = self.embedding(x)
        x = self.activation(x)
        x = self.norm(x)
        x = x + self.position_encoding[:, :x.size(1), :]
        x = self.transformer_encoder(x)

        # Use the last time step for predictions
        x_last = x[:, -1, :]
        return {
            'returns': self.return_head(x_last),
            'volatility': self.volatility_head(x_last),
            'vol_logvar': self.vol_logvar_head(x_last),
            'ret_confidence': self.ret_confidence_head(x_last)
        }
```

A.2 StabilizedConfidenceReturnLoss

```
class StabilizedConfidenceReturnLoss(nn.Module):
    """
    Numerically stable confidence-weighted MSE loss for returns.
    """
    def __init__(self, alpha=0.1, eps=1e-6):
        super().__init__()
        self.alpha = alpha
        self.eps = eps

    def forward(self, pred: torch.Tensor, confidence: torch.Tensor, target:
        torch.Tensor) -> torch.Tensor:
        """
        Args:
            pred: Predicted returns [batch, 1]
            confidence: Confidence scores [batch, 1]
            target: Actual returns [batch, 1]
        """
        pred = torch.nan_to_num(pred, nan=0.0, posinf=1.0, neginf=-1.0)
        confidence = torch.nan_to_num(confidence, nan=0.5)
        target = torch.nan_to_num(target, nan=0.0, posinf=1.0, neginf=-1.0)

        confidence = torch.sigmoid(confidence)
        confidence = torch.clamp(confidence, min=self.eps, max=1 - self.eps)
        mse = torch.clamp((pred - target).pow(2), max=100.0)

        weighted_mse = (confidence * mse).mean()
        entropy_reg = -self.alpha * torch.log(confidence).mean()
```

```
        return weighted_mse + entropy_reg
```

A.3 StabilizedCombinedLoss

```
class StabilizedCombinedLoss(nn.Module):
    """
    Combines stabilized volatility and return confidence losses with robust
        weighting.
    """
    def __init__(self, vol_weight=1.0, ret_weight=1.0,
                 logvar_min=-5.0, logvar_max=5.0, gamma=0.1):
        super().__init__()
        self.vol_loss = CalibratedConfidenceVolatilityLoss(
            logvar_min=logvar_min,
            logvar_max=logvar_max,
            eps=1e-6,
            reg_lambda=0.01,
            gamma=gamma
        )
        self.ret_loss = StabilizedConfidenceReturnLoss(alpha=0.02)
        self.vol_weight = vol_weight
        self.ret_weight = ret_weight

    def check_tensor(self, tensor: torch.Tensor, name: str) -> bool:
        """Helper to check tensor validity and print diagnostics."""
        if tensor is None:
            print(f"Warning: {name} is None")
            return False
        if not isinstance(tensor, torch.Tensor):
            print(f"Warning: {name} is not a tensor, got {type(tensor)}")
            return False
        if tensor.nelement() == 0:
            print(f"Warning: {name} is empty")
            return False

        n_nan = torch.isnan(tensor).sum().item()
        n_inf = torch.isinf(tensor).sum().item()
        if n_nan > 0 or n_inf > 0:
            print(f"Warning: {name} contains {n_nan} NaNs and {n_inf} Infs")
        return True

    def forward(self, outputs: dict, targets: dict) -> tuple:
        """
        Args:
            outputs: Dictionary containing model outputs.
            targets: Dictionary containing true values.
        Returns:
            Tuple of (total_loss, dict of individual losses).
        """
        for key in ['volatility', 'vol_logvar', 'returns', 'ret_confidence']:
            if key not in outputs:
                raise KeyError(f"Missing required output: {key}")
        for key in ['volatility', 'returns']:
            if key not in targets:
                raise KeyError(f"Missing required target: {key}")

        vol_loss = self.vol_loss(
            outputs['volatility'],
            outputs['vol_logvar'],
            targets['volatility']
        )
```

```
        ret_loss = self.ret_loss(
            outputs['returns'],
            outputs['ret_confidence'],
            targets['returns']
        )
        total_loss = self.vol_weight * vol_loss + self.ret_weight * ret_loss
        loss_dict = {
            'vol_loss': vol_loss.item(),
            'ret_loss': ret_loss.item(),
            'total_loss': total_loss.item()
        }
        return total_loss, loss_dict
```

A.4 CalibratedConfidenceVolatilityLoss

```
class CalibratedConfidenceVolatilityLoss(nn.Module):
    """
    Numerically stabilized heteroscedastic Gaussian NLL for volatility prediction,
    with an extra calibration penalty that encourages the predicted confidence to
    reflect the actual error.
    """
    def __init__(self, logvar_min=-3.0, logvar_max=3.0, eps=1e-6,
                 reg_lambda=0.1, gamma=0.1):
        super().__init__()
        self.logvar_min = logvar_min
        self.logvar_max = logvar_max
        self.eps = eps
        self.reg_lambda = reg_lambda
        self.gamma = gamma
        self.constant = 0.5 * math.log(2 * math.pi)

    def forward(self, mu: torch.Tensor, logvar: torch.Tensor,
                target: torch.Tensor) -> torch.Tensor:
        """
        Args:
            mu: Predicted volatility [batch, 1].
            logvar: Predicted log variance (confidence) [batch, 1].
            target: Actual volatility [batch, ...].
        """
        mu = torch.nan_to_num(mu, nan=0.0, posinf=1.0, neginf=-1.0)
        logvar = torch.nan_to_num(logvar, nan=0.0, posinf=self.logvar_max,
            neginf=self.logvar_min)
        target = torch.nan_to_num(target, nan=0.0, posinf=1.0, neginf=-1.0)
        logvar = torch.clamp(logvar, min=self.logvar_min, max=self.logvar_max)

        sq_error = (target - mu).pow(2)
        weighted_sq_error = 0.5 * torch.exp(-logvar) * sq_error
        log_var_term = 0.5 * logvar
        reg_term = self.reg_lambda * logvar.pow(2)

        abs_error = torch.abs(target - mu)
        target_conf = 1.0 / (1.0 + abs_error + self.eps)
        pred_conf = torch.exp(-logvar)
        calib_term = self.gamma * torch.mean(torch.abs(pred_conf - target_conf)
            )

        nll = weighted_sq_error + log_var_term + self.constant + reg_term +
            calib_term
        return nll.mean()
```