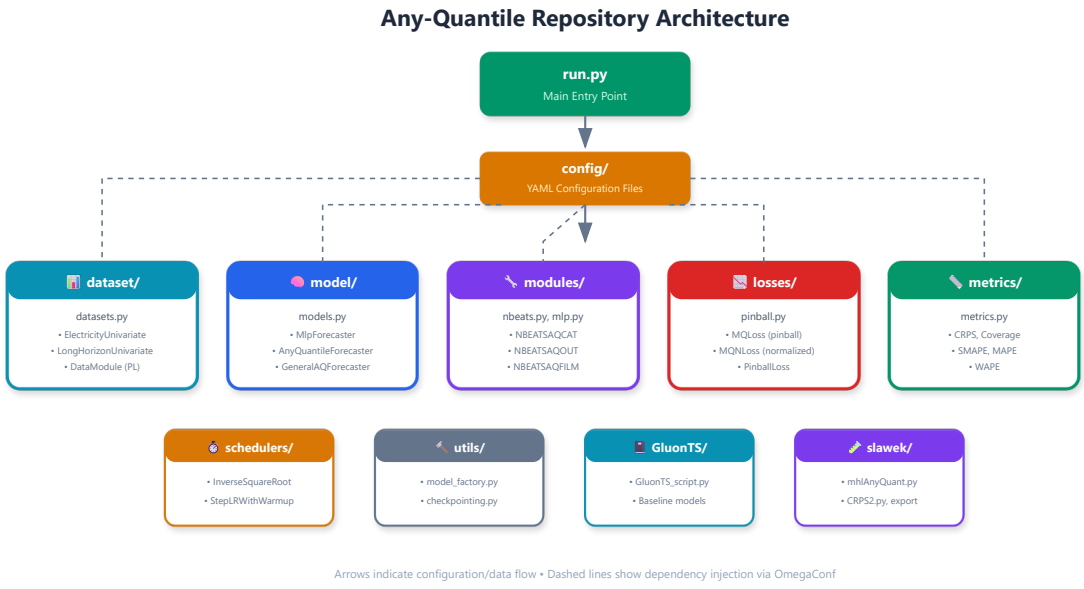
Code Architecture Guide & Improvement Roadmap

PyTorch Lightning Python 3.8+

Project Structure Overview

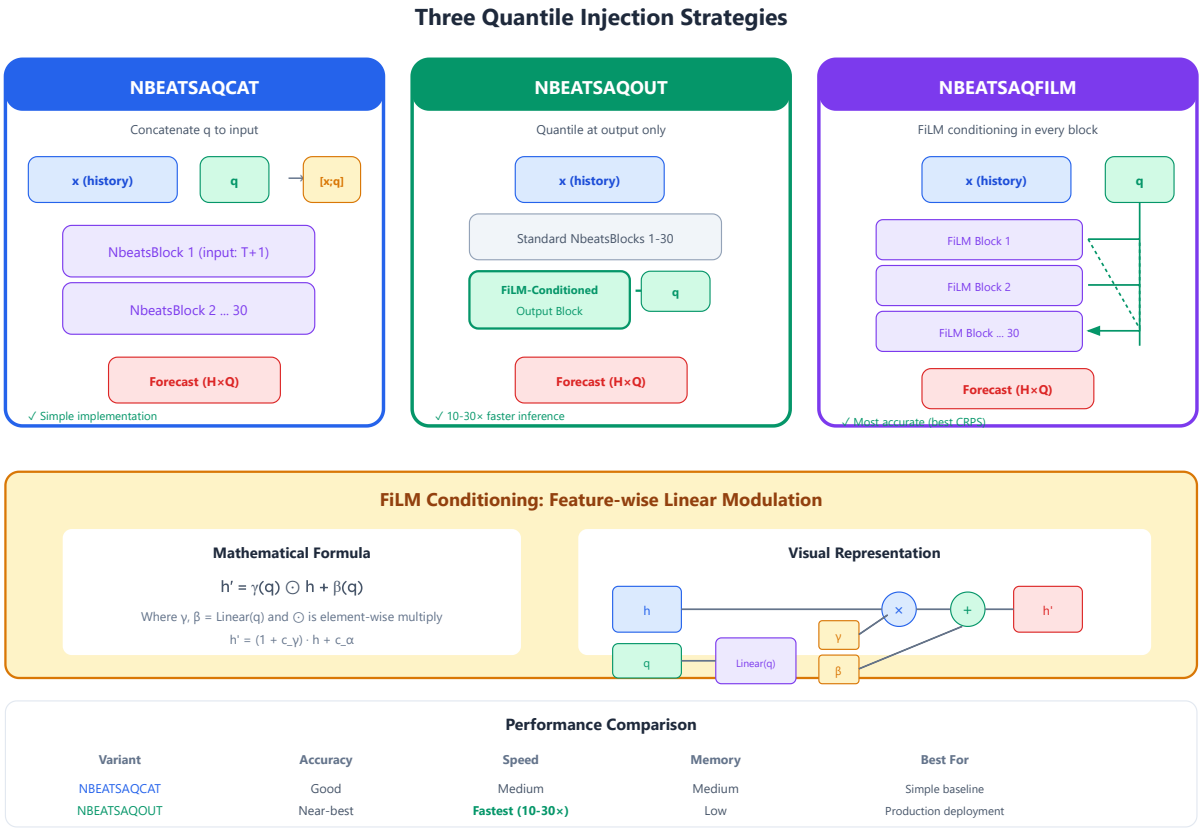



```
any-quantile-main/
├── run.py                  # Main training entry point
├── config/
│   ├── nbeatsaq-mhlv.yaml  # N-BEATS Any-Quantile config
│   └── snaive-mhlv.yaml    # Seasonal Naive baseline
├── dataset/
│   └── datasets.py         # Dataset classes & DataModules
├── model/
│   └── models.py           # PyTorch Lightning models
├── modules/
```

```
│   ├── nbeats.py          # N-BEATS architectures (CAT, OUT, FiLM)
│   ├── mlp.py             # MLP backbone
│   └── snaive.py          # Seasonal Naive baseline
├── losses/
│   └── pinball.py         # Quantile loss functions
├── metrics/
│   └── metrics.py         # CRPS, SMAPE, MAPE, Coverage
├── schedulers/
│   └── inverse_square_root.py # LR schedulers
└── utils/
    ├── model_factory.py     # Dynamic instantiation
    └── checkpointing.py     # Checkpoint management
```

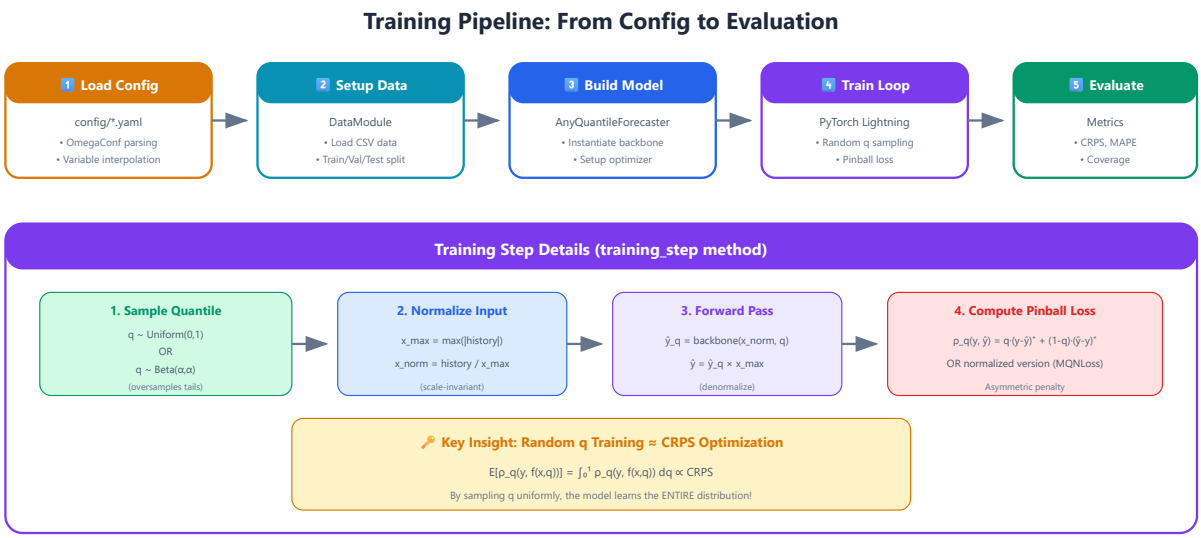
N-BEATS Architecture Variants

The codebase implements three variants of Any-Quantile N-BEATS, each with a different strategy for injecting the quantile level q into the network:



```
# modules/nbeats.py - FiLM Conditioning Implementation
class NbeatsBlockConditioned(NbeatsBlock):
    """Block with FiLM (Feature-wise Linear Modulation) conditioning"""
    def __init__(self, num_layers, layer_width, size_in, size_out):
        super().__init__(num_layers, layer_width, size_in, size_out)
        # FiLM: produces scale (γ) and shift (β) from condition
        self.condition_film = torch.nn.Linear(self.layer_width, 2*self.layer_width)
    def forward(self, x, condition):
        h = x
        for i, layer in enumerate(self.fc_layers):
            h = F.relu(layer(h))
            if i == 0:  # Apply FiLM after first layer
                condition_film = self.condition_film(condition)
                condition_offset = condition_film[..., :self.layer_width]  # β
                condition_delta = condition_film[..., self.layer_width:]  # γ
                h = h * (1 + condition_delta) + condition_offset  # h' = (1+γ)h + β
        return self.forward_projection(h), self.backward_projection(h)
```

Training Pipeline Flow




```
# model/models.py - AnyQuantileForecaster.training_step
def training_step(self, batch, batch_idx):
    batch_size = batch['history'].shape[0]
    # Step 1: Sample random quantiles for this batch
    if self.cfg.model.q_distribution == 'uniform':
        batch['quantiles'] = torch.rand(batch_size, 1).to(batch['history'])
    elif self.cfg.model.q_distribution == 'beta':
        batch['quantiles'] = torch.Tensor( np.random.beta(self.cfg.model.q_parameter, self.cfg.model.q_parameter, size=(batch_size, 1)) ).to(batch['history'])
    # Step 2-3: Forward pass (includes normalization)
    net_output = self.shared_forward(batch)
    y_hat =
```

```
net_output['forecast'] # B × H × Q quantiles = net_output['quantiles'][:, None] # B × 1 × Q #
Step 4: Compute pinball loss loss = self.loss(y_hat, batch['target'], q=quantiles) return loss
```

🔍 Key Components Deep Dive

📉 Losses Module

losses/pinball.py

→ **MQLoss**: Multi-quantile pinball loss

→ **MQNLoss**: Normalized version (divides by target)

→ **PinballLoss**: Fixed quantile version

→ **PinballMape**: MAPE-style pinball loss

📏 Metrics Module

metrics/metrics.py

→ **CRPS**: Continuous Ranked Probability Score

→ **Coverage**: Prediction interval coverage

→ **SMAPE**: Symmetric MAPE

→ **MAPE**: Mean Absolute Percentage Error

→ **WAPE**: Weighted APE

📊 Dataset Module

dataset/datasets.py

→ **ElectricityUnivariateDataset**: Main dataset class

→ **ElectricityUnivariateDataModule**: PL DataModule

→ **LongHorizonUnivariateDataset**: Alternative dataset

→ Handles train/val/test splits with timestamps

🔨 Utils Module

utils/

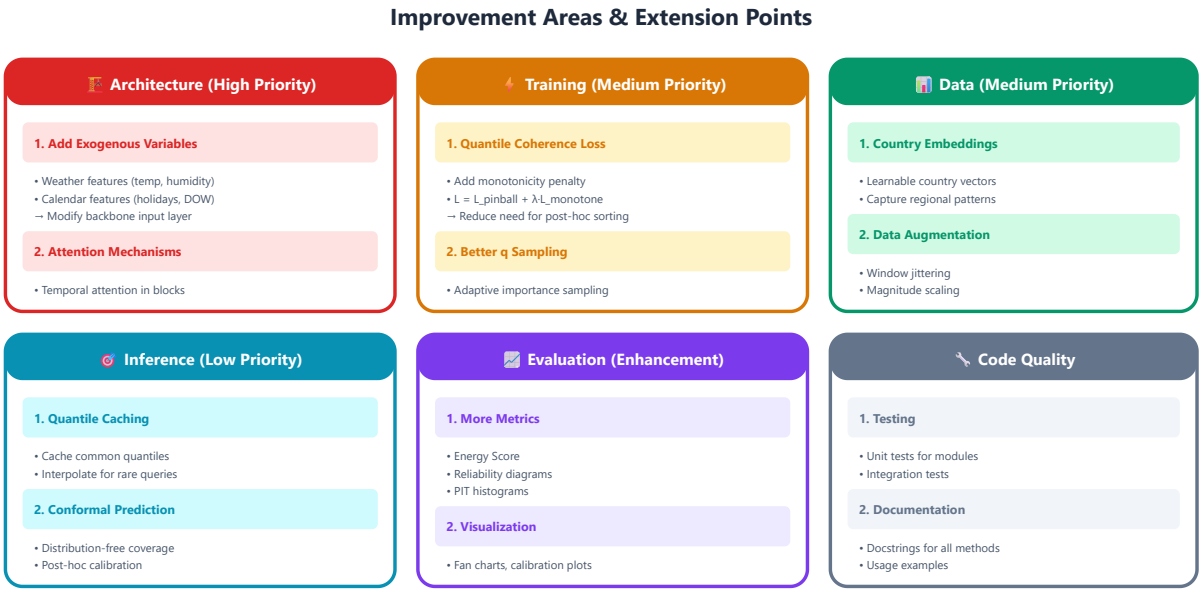
→ **model_factory.py**: Dynamic class instantiation

→ **checkpointing.py**: Checkpoint path resolution

→ Enables config-driven object creation

🚀 Improvement Roadmap

Based on analyzing the codebase and the paper, here are concrete improvement opportunities organized by priority:



🎯 Top Priority Improvements with Code Examples

```
self.model(x, q) # Expand by calibrated amount return pred - self.q_hat, pred + self.q_hat
```

🏃 Quick Start Guide

1. **Clone & Setup Environment**
   git clone <repo> && pip install -r requirements.txt
2. **Prepare Data**
   Place electricity CSV data in ./data/electricity/datasets/MHLV/M/df_y.csv
3. **Configure Experiment**
   Edit config/nbeatsaq-mhlv.yaml - adjust batch size, epochs, architecture
4. **Run Training**
   python run.py --config=config/nbeatsaq-mhlv.yaml
5. **Monitor & Evaluate**
   Check TensorBoard at ./lightning_logs/, results logged automatically

💡 Key Configuration Parameters

Architecture:

- num_blocks: 30
- num_layers: 3
- layer_width: 1024

Training:

- batch_size: 1024
- epochs: 15
- lr: 0.0005

Data:

- history_length: 168 (1 week)
- horizon_length: 48 (2 days)

Extension Points Summary

✅ Design Philosophy

The codebase uses **dependency injection via OmegaConf**. To add new components:

1. Create your class in the appropriate module (e.g., `modules/my_backbone.py`)
2. Add it to `__init__.py` exports
3. Reference it in config: `model.nn.backbone._target_: modules.MyBackbone`

No changes to `run.py` needed - the factory pattern handles instantiation automatically!

🚀 High Priority Improvements

These improvements offer the highest potential impact on model performance based on the paper's discussion of limitations and our analysis of the codebase.

HIGH PRIORITY

1. Add Exogenous Variables Support (Weather, Calendar)

The current implementation only uses historical demand values. Adding weather features (temperature, humidity, wind) and calendar features (holidays, day-of-week, hour-of-day) can significantly improve accuracy, especially for peak demand prediction.

📈 Expected Impact

- 10-20% CRPS improvement on weather-sensitive regions
- Better extreme quantile prediction (Q95, Q99)
- More robust forecasts during holidays/special events

Implementation Steps

1. **Modify Dataset Class** - Add exogenous feature loading and alignment with time series
2. **Create Feature Encoder** - Build embeddings for categorical features and normalization for continuous
3. **Modify Backbone** - Add exogenous input pathway to N-BEATS blocks
4. **Update Config** - Add exogenous feature configuration options

dataset/datasets.py — PYTHON

```python
class ElectricityWithExogDataset(ElectricityUnivariateDataset):
    """Extended dataset with exogenous features support"""

    def __init__(self,
                 name: str,
                 split: str,
                 exog_features: List[str] = ['temperature', 'humidity'],
                 calendar_features: bool = True,
                 **kwargs):
        super().__init__(name, split, **kwargs)

        self.exog_features = exog_features
        self.calendar_features = calendar_features

        # Load exogenous data (aligned with time series)
        if exog_features:
            exog_path = f'./data/exogenous/{name.lower()}_weather.csv'
            self.exog_df = pd.read_csv(exog_path, parse_dates=['ds'])
            self.exog_df = self.exog_df.set_index('ds')

    def _get_calendar_features(self, timestamps):
        """Generate calendar embeddings"""
        hour = timestamps.hour / 24.0                  # [0, 1]
        dow = timestamps.dayofweek / 7.0               # [0, 1]
        month = (timestamps.month - 1) / 12.0          # [0, 1]
        is_weekend = (timestamps.dayofweek >= 5).astype(float)
        return np.stack([hour, dow, month, is_weekend], axis=-1)

    def __getitem__(self, index):
        item = super().__getitem__(index)

        # Get timestamps for this window
        window_times = self.series_timestamps[window_idx:window_idx + self.tot_window_len]

        # Add exogenous features
        if self.exog_features:
            exog = self.exog_df.loc[window_times][self.exog_features].values
            item['exog_history'] = exog[:self.history_length].astype(np.float32)
            item['exog_future'] = exog[self.history_length:].astype(np.float32)

        if self.calendar_features:
            item['calendar'] = self._get_calendar_features(window_times)
```

```
        return item
```

modules/nbeats_exog.py — PYTHON

```python
import torch
import torch.nn as nn
import torch.nn.functional as F

class ExogenousEncoder(nn.Module):
    """Encode exogenous features into embedding space"""

    def __init__(self,
                 num_continuous: int = 4,      # temp, humidity, etc.
                 num_calendar: int = 4,        # hour, dow, month, weekend
                 embed_dim: int = 64):
        super().__init__()

        # Continuous feature normalization + projection
        self.continuous_proj = nn.Sequential(
            nn.Linear(num_continuous, embed_dim),
            nn.ReLU(),
            nn.Linear(embed_dim, embed_dim)
        )

        # Calendar feature embeddings
        self.hour_embed = nn.Embedding(24, embed_dim // 4)
        self.dow_embed = nn.Embedding(7, embed_dim // 4)
        self.month_embed = nn.Embedding(12, embed_dim // 4)
        self.weekend_embed = nn.Embedding(2, embed_dim // 4)

    def forward(self, continuous, calendar):
        """
        Args:
            continuous: [B, T, num_continuous] - weather features
            calendar: [B, T, 4] - (hour, dow, month, is_weekend) as indices
        Returns:
            [B, T, embed_dim] - combined embedding
        """
        # Project continuous features
        cont_embed = self.continuous_proj(continuous)

        # Get calendar embeddings
        cal_embed = torch.cat([
            self.hour_embed(calendar[..., 0].long()),
            self.dow_embed(calendar[..., 1].long()),
            self.month_embed(calendar[..., 2].long()),
            self.weekend_embed(calendar[..., 3].long())
        ], dim=-1)

        return cont_embed + cal_embed
```

modules/nbeats_exog.py (continued) — PYTHON

```python
class NbeatsBlockWithExog(NbeatsBlockConditioned):
    """N-BEATS block with exogenous feature conditioning"""

    def __init__(self,
                 num_layers: int,
                 layer_width: int,
                 size_in: int,
                 size_out: int,
                 exog_dim: int = 64):
        super().__init__(num_layers, layer_width, size_in, size_out)

        # Project exogenous embeddings to layer width
        self.exog_projection = nn.Linear(exog_dim, layer_width)

        # Gating mechanism to control exog influence
        self.exog_gate = nn.Sequential(
            nn.Linear(layer_width * 2, layer_width),
            nn.Sigmoid()
        )

    def forward(self, x, condition, exog_embed):
        """
        Args:
```

```
            x: [B, Q, T] - input time series (already replicated for Q quantiles)
            condition: [B, Q, layer_width] - quantile conditioning
            exog_embed: [B, T, exog_dim] - exogenous feature embeddings
        """
        h = x

        # Pool exogenous features over time dimension
        exog_pooled = self.exog_projection(exog_embed.mean(dim=1))  # [B, layer_width]
        exog_pooled = exog_pooled.unsqueeze(1).expand(-1, x.size(1), -1)  # [B, Q, layer_width]

        for i, layer in enumerate(self.fc_layers):
            h = F.relu(layer(h))
            if i == 0:
                # FiLM conditioning from quantile
                condition_film = self.condition_film(condition)
                offset, delta = condition_film[..., :self.layer_width], condition_film[..., self.layer_width:]
                h = h * (1 + delta) + offset

                # Gated addition of exogenous features
                gate = self.exog_gate(torch.cat([h, exog_pooled], dim=-1))
                h = h + gate * exog_pooled

        return self.forward_projection(h), self.backward_projection(h)
```

HIGH PRIORITY

2. Add Quantile Monotonicity Loss

Currently, quantile crossing (where Q(0.3) > Q(0.7)) is handled by post-hoc sorting. This is suboptimal because it doesn't provide gradient signal during training. Adding a monotonicity penalty during training can improve CRPS by ~34% (as shown in the paper's ablation).

Expected Impact

- Eliminate quantile crossing during inference
- Improve calibration quality
- More coherent uncertainty estimates

Implementation Steps

1. **Create MonotonicityLoss class** - Penalize when adjacent quantiles cross
2. **Modify training_step** - Sample multiple quantiles per batch, compute joint loss
3. **Add loss weighting** - Balance pinball loss with monotonicity penalty

losses/monotone.py — PYTHON

```
import torch
import torch.nn as nn
import torch.nn.functional as F

class MonotonicityLoss(nn.Module):
    """
    Penalize quantile crossing during training.

    When predicting multiple quantiles simultaneously, higher quantile levels
    should predict higher values. This loss penalizes violations of this constraint.
    """

    def __init__(self, margin: float = 0.0, reduction: str = 'mean'):
        """
        Args:
            margin: Minimum gap between adjacent quantile predictions
            reduction: 'mean', 'sum', or 'none'
        """
        super().__init__()
        self.margin = margin
        self.reduction = reduction

    def forward(self, predictions: torch.Tensor, quantiles: torch.Tensor):
        """
        Args:
            predictions: [B, H, Q] - predicted values for Q quantiles
            quantiles: [B, 1, Q] or [B, Q] - quantile levels (should be sorted)
```

```
        Returns:
            Scalar loss penalizing monotonicity violations
        """
        # Ensure quantiles are 3D
        if quantiles.dim() == 2:
            quantiles = quantiles.unsqueeze(1)

        # Sort quantiles and get sorting indices
        q_sorted, sort_idx = quantiles.sort(dim=-1)

        # Apply same sorting to predictions
        sort_idx_expanded = sort_idx.expand_as(predictions)
        pred_sorted = predictions.gather(-1, sort_idx_expanded)

        # Compute differences between adjacent quantile predictions
        # For proper ordering: pred[q_i+1] >= pred[q_i] + margin
        diffs = pred_sorted[..., 1:] - pred_sorted[..., :-1]  # [B, H, Q-1]

        # Penalize negative differences (crossings)
        # Using soft penalty: ReLU(-diff + margin)
        violations = F.relu(-diffs + self.margin)

        # Apply reduction
        if self.reduction == 'mean':
            return violations.mean()
        elif self.reduction == 'sum':
            return violations.sum()
        else:
            return violations
```

model/models.py - Modified training_step PYTHON

```python
class AnyQuantileForecasterWithMonotonicity(AnyQuantileForecaster):
    """Extended forecaster with monotonicity loss"""

    def __init__(self, cfg):
        super().__init__(cfg)
        self.monotonicity_loss = MonotonicityLoss(margin=cfg.model.monotone_margin)
        self.monotone_weight = cfg.model.monotone_weight  # e.g., 0.1

    def training_step(self, batch, batch_idx):
        batch_size = batch['history'].shape[0]

        # Sample MULTIPLE quantiles per sample for monotonicity training
        num_quantiles = self.cfg.model.num_train_quantiles  # e.g., 9

        if self.cfg.model.q_distribution == 'uniform':
            # Sample sorted quantiles
            q = torch.rand(batch_size, num_quantiles).to(batch['history'])
            q, _ = q.sort(dim=-1)  # Ensure sorted for monotonicity
        elif self.cfg.model.q_distribution == 'fixed':
            # Use fixed quantile grid
            q = torch.linspace(0.1, 0.9, num_quantiles)
            q = q.unsqueeze(0).expand(batch_size, -1).to(batch['history'])

        batch['quantiles'] = q

        # Forward pass - now predicts multiple quantiles
        net_output = self.shared_forward(batch)
        y_hat = net_output['forecast']       # [B, H, Q]
        quantiles = net_output['quantiles']  # [B, Q]

        # Pinball loss (main training objective)
        pinball_loss = self.loss(y_hat, batch['target'], q=quantiles[:, None, :])

        # Monotonicity loss (regularization)
        monotone_loss = self.monotonicity_loss(y_hat, quantiles)

        # Combined loss
        total_loss = pinball_loss + self.monotone_weight * monotone_loss

        # Logging
        self.log("train/pinball_loss", pinball_loss)
        self.log("train/monotone_loss", monotone_loss)
        self.log("train/total_loss", total_loss)

        return total_loss
```

HIGH PRIORITY

3. Add Learnable Country/Series Embeddings

The current cross-learning approach treats all 35 countries identically in terms of model parameters. Adding learnable embeddings for each country allows the model to capture country-specific patterns (e.g., industrial vs residential, climate differences) while still benefiting from shared learning.

Expected Impact

- Better capture of regional electricity patterns
- Improved forecasts for smaller countries with less data
- Enables analysis of learned country similarities

model/models.py - With Series Embeddings — PYTHON

```python
class AnyQuantileWithSeriesEmbedding(AnyQuantileForecaster):
    """Forecaster with learnable per-series (country) embeddings"""

    def __init__(self, cfg):
        super().__init__(cfg)

        # Learnable embeddings for each time series (country)
        self.num_series = cfg.model.num_series  # 35 for European countries
        self.embed_dim = cfg.model.series_embed_dim  # e.g., 32

        self.series_embedding = nn.Embedding(
            num_embeddings=self.num_series,
            embedding_dim=self.embed_dim
        )
```

```
        # Project to layer width for FiLM-style conditioning
        layer_width = cfg.model.nn.backbone.layer_width
        self.series_film = nn.Linear(self.embed_dim, 2 * layer_width)

        # Alternative: add to input
        self.series_proj = nn.Linear(self.embed_dim, cfg.model.input_horizon_len)

    def shared_forward(self, x):
        history = x['history'][:, -self.cfg.model.input_horizon_len:]
        series_id = x['series_id'].squeeze(-1)  # [B]
        q = x['quantiles']

        # Get series embedding
        series_embed = self.series_embedding(series_id)  # [B, embed_dim]

        # Option 1: Add to input (simple)
        series_bias = self.series_proj(series_embed)  # [B, T]
        history = history + series_bias

        # Normalize
        x_max = torch.abs(history).max(dim=-1, keepdims=True)[0]
        x_max[x_max == 0] = 1
        history_norm = history / x_max

        # Forward through backbone with series embedding
        # Option 2: Pass series_embed to backbone for FiLM conditioning
        forecast = self.backbone(
            history_norm,
            q,
            series_embed=series_embed  # New argument
        )

        return {
            'forecast': forecast * x_max[..., None],
            'quantiles': q,
            'series_embed': series_embed  # For analysis
        }
```

HIGH PRIORITY

4. Add Temporal Attention with Quantile Conditioning

The current N-BEATS uses fully-connected layers with fixed receptive fields. Adding attention allows the model to dynamically focus on relevant historical patterns for each quantile prediction. Different quantiles may need to attend to different parts of history (e.g., extreme quantiles to peak events).

Expected Impact

- Better handling of long-range dependencies
- Improved extreme quantile predictions
- Interpretable attention patterns

modules/attention.py — PYTHON

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import math

class QuantileConditionedAttention(nn.Module):
    """
    Multi-head attention where queries are conditioned on quantile level.

    This allows different quantiles to attend to different parts of history:
    - Extreme quantiles (0.01, 0.99) may focus on peak/trough events
    - Central quantiles (0.5) may attend more uniformly
    """

    def __init__(self,
                 d_model: int = 256,
                 n_heads: int = 8,
                 dropout: float = 0.1):
        super().__init__()

        assert d_model % n_heads == 0, "d_model must be divisible by n_heads"

```

```
        self.d_model = d_model
        self.n_heads = n_heads
        self.d_k = d_model // n_heads

        # Standard Q, K, V projections
        self.W_q = nn.Linear(d_model, d_model)
        self.W_k = nn.Linear(d_model, d_model)
        self.W_v = nn.Linear(d_model, d_model)
        self.W_o = nn.Linear(d_model, d_model)

        # Quantile conditioning for queries
        self.quantile_proj = nn.Sequential(
            nn.Linear(1, d_model),
            nn.ReLU(),
            nn.Linear(d_model, d_model)
        )

        self.dropout = nn.Dropout(dropout)
        self.layer_norm = nn.LayerNorm(d_model)

    def forward(self, x, quantile):
        """
        Args:
            x: [B, T, d_model] - input sequence
            quantile: [B, 1] or [B, Q] - quantile levels
        Returns:
            [B, T, d_model] - attended output
        """
        B, T, _ = x.shape

        # Get quantile conditioning
        q_embed = self.quantile_proj(quantile.unsqueeze(-1))  # [B, Q, d_model]

        # Compute Q, K, V
        Q = self.W_q(x) + q_embed  # Condition queries on quantile
        K = self.W_k(x)
        V = self.W_v(x)

        # Reshape for multi-head attention
        Q = Q.view(B, T, self.n_heads, self.d_k).transpose(1, 2)
        K = K.view(B, T, self.n_heads, self.d_k).transpose(1, 2)
        V = V.view(B, T, self.n_heads, self.d_k).transpose(1, 2)

        # Scaled dot-product attention
        scores = torch.matmul(Q, K.transpose(-2, -1)) / math.sqrt(self.d_k)
        attn = F.softmax(scores, dim=-1)
        attn = self.dropout(attn)

        # Apply attention and reshape
        out = torch.matmul(attn, V)
        out = out.transpose(1, 2).contiguous().view(B, T, self.d_model)

        # Output projection + residual + layer norm
        return self.layer_norm(x + self.W_o(out))
```

HIGH PRIORITY

5. Hierarchical Quantile Architecture

Instead of predicting all quantiles independently, use a two-stage approach: first predict the median and IQR, then condition extreme quantile predictions on these. This captures the natural hierarchical structure of quantiles and improves extreme quantile accuracy.

Expected Impact

- More coherent quantile predictions
- Better extreme quantile (Q95, Q99) accuracy
- Reduced quantile crossing

modules/hierarchical.py — PYTHON

```
class HierarchicalQuantilePredictor(nn.Module):
    """
    Two-stage hierarchical quantile prediction:
```

```
    Stage 1: Predict location (median) and scale (IQR or std)
    Stage 2: Predict quantile offsets conditioned on location/scale

    This ensures extreme quantiles are coherent with central tendency.
    """

    def __init__(self, backbone, size_in, size_out, layer_width):
        super().__init__()

        # Stage 1: Predict median and scale
        self.location_head = nn.Sequential(
            nn.Linear(layer_width, layer_width // 2),
            nn.ReLU(),
            nn.Linear(layer_width // 2, size_out)  # Median prediction
        )

        self.scale_head = nn.Sequential(
            nn.Linear(layer_width, layer_width // 2),
            nn.ReLU(),
            nn.Linear(layer_width // 2, size_out),
            nn.Softplus()  # Scale must be positive
        )

        # Stage 2: Quantile offset predictor
        self.offset_net = nn.Sequential(
            nn.Linear(layer_width + 1, layer_width),  # +1 for quantile
            nn.ReLU(),
            nn.Linear(layer_width, size_out)
        )

        self.backbone = backbone

    def forward(self, x, q):
        """
        Args:
            x: [B, T] - input history
            q: [B, Q] - quantile levels to predict
        Returns:
            [B, H, Q] - quantile predictions
        """
        # Get backbone features
        features = self.backbone.encode(x)  # [B, layer_width]

        # Stage 1: Location and scale
        median = self.location_head(features)  # [B, H]
        scale = self.scale_head(features)       # [B, H]

        # Stage 2: Quantile-specific offsets
        Q = q.shape[-1]
        features_expanded = features.unsqueeze(1).expand(-1, Q, -1)  # [B, Q, W]
        q_expanded = q.unsqueeze(-1)  # [B, Q, 1]

        offset_input = torch.cat([features_expanded, q_expanded], dim=-1)
        offsets = self.offset_net(offset_input)  # [B, Q, H]
        offsets = offsets.transpose(1, 2)  # [B, H, Q]

        # Final prediction: location + scale * offset
        # Offset is scaled by quantile distance from median
        q_centered = (q - 0.5).unsqueeze(1)  # [B, 1, Q]
        predictions = median.unsqueeze(-1) + scale.unsqueeze(-1) * offsets * q_centered.sign()

        return predictions
```

HIGH PRIORITY

6. Adaptive Importance Sampling for Quantiles

The current Beta(0.3, 0.3) distribution oversamples extreme quantiles. An adaptive approach can focus sampling on quantiles where the model has highest error, leading to more efficient training.

utils/adaptive_sampling.py — PYTHON

```
class AdaptiveQuantileSampler:
    """
    Sample quantiles with probability proportional to their training loss.

```

```
    Maintains a running estimate of per-quantile loss and samples more
    frequently from quantiles where the model struggles.
    """

    def __init__(self,
                 num_bins: int = 100,
                 momentum: float = 0.99,
                 temperature: float = 1.0,
                 min_prob: float = 0.001):
        self.num_bins = num_bins
        self.momentum = momentum
        self.temperature = temperature
        self.min_prob = min_prob

        # Running loss estimates per quantile bin
        self.loss_estimates = np.ones(num_bins)
        self.bin_edges = np.linspace(0, 1, num_bins + 1)

    def update(self, quantiles, losses):
        """Update loss estimates with new observations"""
        quantiles_np = quantiles.detach().cpu().numpy().flatten()
        losses_np = losses.detach().cpu().numpy().flatten()

        # Bin the quantiles
        bin_indices = np.digitize(quantiles_np, self.bin_edges) - 1
        bin_indices = np.clip(bin_indices, 0, self.num_bins - 1)

        # Update running estimates
        for bin_idx, loss in zip(bin_indices, losses_np):
            self.loss_estimates[bin_idx] = (
                self.momentum * self.loss_estimates[bin_idx] +
                (1 - self.momentum) * loss
            )

    def sample(self, batch_size):
        """Sample quantiles with probability proportional to loss"""
        # Softmax with temperature
        probs = np.exp(self.loss_estimates / self.temperature)
        probs = np.maximum(probs, self.min_prob)
        probs = probs / probs.sum()

        # Sample bins
        bins = np.random.choice(self.num_bins, size=batch_size, p=probs)

        # Sample uniformly within each bin
        quantiles = (self.bin_edges[bins] +
                     np.random.rand(batch_size) * (self.bin_edges[bins + 1] - self.bin_edges[bins]))

        return torch.tensor(quantiles, dtype=torch.float32)
```

⚙️ Configuration for Improvements

Add these parameters to your YAML config to enable the improvements:

config/nbeatsaq-improved.yaml YAML

```yaml
# === IMPROVEMENT 1: Exogenous Features ===
dataset.exog_features: ['temperature', 'humidity', 'wind_speed']
dataset.calendar_features: True
model.nn.exog_encoder._target_: modules.ExogenousEncoder
model.nn.exog_encoder.embed_dim: 64

# === IMPROVEMENT 2: Monotonicity Loss ===
model.use_monotonicity_loss: True
model.monotone_weight: 0.1
model.monotone_margin: 0.0
model.num_train_quantiles: 9  # Sample multiple q per batch

# === IMPROVEMENT 3: Series Embeddings ===
model.use_series_embedding: True
```

```
model.num_series: 35  # Number of countries
model.series_embed_dim: 32

# === IMPROVEMENT 4: Attention ===
model.nn.use_attention: True
model.nn.attention._target_: modules.QuantileConditionedAttention
model.nn.attention.d_model: 256
model.nn.attention.n_heads: 8
model.nn.attention.dropout: 0.1

# === IMPROVEMENT 5: Hierarchical Architecture ===
model.nn.backbone._target_: modules.HierarchicalQuantilePredictor
model.nn.backbone.use_hierarchical: True

# === IMPROVEMENT 6: Adaptive Sampling ===
model.q_sampling: 'adaptive'
model.adaptive_sampler.num_bins: 100
model.adaptive_sampler.momentum: 0.99
model.adaptive_sampler.temperature: 1.0

# === Standard Config (unchanged) ===
model._target_: model.AnyQuantileForecasterImproved
model.nn.backbone.layer_width: 1024
model.nn.backbone.num_blocks: 30
model.nn.backbone.num_layers: 3
trainer.max_epochs: 20  # More epochs for new features
```