深度学习与机器学习方法在风电功率预测中的应用与比较研究

张晋铭

June 6, 2025

Abstract

风电功率预测是保障电网稳定运行和提高风能利用效率的关键技术。本研究构建了一套完整的风电功率预测系统，深入比较了三种主流深度学习模型（LSTM、PatchTST、iTransformer）和传统机器学习方法（CatBoost）在短期风电功率预测中的实际性能。基于真实风电场数据，针对4小时（16个时间点）的多步预测任务展开深入分析。研究设计了完整的特征工程方案，构建了滞后特征、趋势特征、爬坡特征和统计特征等综合特征体系。实验结果显示，LSTM模型在功率预测准确率（CR）指标上表现最优，达到62.56%；PatchTST在传统指标RMSE和MAE上表现最佳；而CatBoost方法在保持59.12% CR性能的同时，大幅降低了计算成本和特征工程复杂度。特征工程消融实验表明，统计特征对性能提升贡献最大（+3.19%）。而在对CatBoost的实验结果分析中可得到，通过SHAP分析的特征选择策略能够在维持预测精度的同时显著简化模型复杂度。本研究的所有代码和模型已开源，项目地址：https://github.com/Jinming00/EE。

关键词：风电功率预测、时间序列预测、特征工程、LSTM、Transformer、CatBoost

Contents

1 引言

1.1 研究背景

风力发电作为清洁可再生能源的重要组成部分，在全球能源转型中发挥着关键作用。然而，风电功率的间歇性和随机性给电网运行带来了重大挑战。准确的风电功率预测技术是解决这一问题的核心，不仅有助于优化电网调度，还能提高风能利用效率，降低系统运行成本。

传统风电功率预测方法主要依赖物理模型和统计模型。物理模型基于数值天气预报和风机特性曲线，虽然具备较强的物理解释性，但难以准确刻画复杂的大气边界层流动和地形影响；统计模型通过历史数据建立数学关系，但在处理非线性特征和长期依赖关系方面存在局限性。

近年来，深度学习技术在时间序列预测领域取得了重大突破，为风电功率预测提供了新的技术路径。深度学习模型能够自动学习数据中的复杂模式和非线性关系，在处理大规模、高维度时间序列数据方面展现出明显优势。

1.2 研究目标与贡献

本研究通过系统的对比分析，评估不同类型机器学习方法在风电功率预测中的适用性和有效性。具体实现并比较了以下四类方法：

深度学习方法：

1. **增强型LSTM（Attention-Enhanced LSTM）**：基于传统LSTM架构，融合多头注意力机制和残差连接，擅长处理时间序列数据中的长期依赖关系
2. **PatchTST（Patching Time Series Transformer）**：专为时间序列预测设计的轻量级Transformer，通过分块机制提高计算效率
3. **iTransformer（Inverted Transformer）**：创新性地颠倒了Transformer架构中时间和变量维度的处理方式，特别适合多变量预测

机器学习方法：

4. **CatBoost**：基于梯度提升决策树的集成学习方法，具有高效性和强解释性

2 相关工作

深度学习模型在时间序列预测领域已取得显著进展：

- **RNN及其变体**：LSTM [2]和GRU通过门控机制解决了传统RNN的梯度消失问题，能够有效捕捉长期依赖关系。
- **CNN在时间序列中的应用**：TCN（Temporal Convolutional Network） [3]通过因果卷积和扩张卷积实现了对长序列的高效处理。
- **Transformer模型**：自注意力机制使Transformer [4]在捕捉序列中的长距离依赖关系方面表现出色，但标准Transformer在处理长时间序列时计算成本较高。
- **改进的Transformer变体**：为解决标准Transformer的局限性，研究者提出了多种改进模型，如Informer [5]、Autoformer [6]、PatchTST [7]和iTransformer [8]等。

3 问题定义与数据集

3.1 问题定义

风电功率短期预测问题可形式化定义为：给定历史时间窗口内的风电功率观测序列 $\{P_{t-L+1}, P_{t-L+2}, \ldots,$ 及其衍生特征，预测未来时间窗口内的功率值序列 $\{\hat{P}_{t+1}, \hat{P}_{t+2}, \ldots, \hat{P}_{t+H}\}$。

本研究采用以下设置：

- **输入序列长度** $L = 96$：对应24小时的历史数据（每15分钟一个采样点）
- **预测序列长度** $H = 16$：对应未来4小时的预测窗口
- **采样频率**：15分钟

3.2 数据集

本研究使用某风电场的实际功率数据，数据已完成标准化处理，数据集的基本统计信息如表1所示：

Table 1: 数据集基本统计信息

数据集	样本数量	功率值范围	均值	标准差
训练集	7,000	[-0.0094, 1.0000]	0.257	0.249
测试集	3,000	[-0.0093, 0.9446]	0.266	0.253

数据预处理步骤：

1. **异常值处理**：将少量负功率值（物理上不合理）修正为0
2. **跳跃/滑动步长**：默认设置为1，以获得最大数据集
3. **数据分割**：训练集与验证集按8:2比例划分

4 方法与模型

4.1 特征工程设计

特征工程是提升风电功率预测精度的关键环节。基于风电功率时间序列的特性分析，设计了一套系统的特征工程方案，构建了四大类共57个特征，有效捕捉了风电功率数据的多维度特征。

4.1.1 滞后特征（Lag Features）

滞后特征通过引入不同时间延迟的历史功率值，帮助模型捕捉时间序列的自相关性和周期性模式。选择了8个关键滞后步长：[1, 2, 4, 8, 16, 24, 48, 96]，分别对应15分钟、30分钟、1小时、2小时、4小时、6小时、12小时和24小时的历史信息。

Listing 1: 滞后特征实现代码

```
# 滞后特征实现
if self.config.add_lag_features:
    for lag in self.config.lag_steps:
```

```
        if lag < len(self.data):
            lag_feature = np.roll(self.data, lag)
            lag_feature[:lag] = lag_feature[lag]
            features.append(lag_feature)
            feature_names.append(f'lag_{lag}')
```

设计理念：不同滞后步长捕捉不同时间尺度的依赖关系，短期滞后反映功率的连续性，长期滞后捕捉日内和日间的周期性模式。

4.1.2 趋势特征（Trend Features）

趋势特征通过一阶和二阶差分运算，量化功率的变化趋势和变化速率，帮助模型理解功率演变的动态特性。

Listing 2: 趋势特征实现代码

```
def add_trend_features(self, power_series):
    features = []
    feature_names = []

    # 一阶差分变化率() - 标准化
    diff_1 = power_series.diff(1).fillna(0)
    # 裁剪到合理范围并标准化到[-1, 1]
    diff_1_normalized = np.clip(diff_1.values, -0.5, 0.5)
    features.append(diff_1_normalized)
    feature_names.append('diff_1')

    # 二阶差分加速度() - 标准化
    diff_2 = power_series.diff(2).fillna(0)
    # 裁剪到合理范围并标准化到[-1, 1]
    diff_2_normalized = np.clip(diff_2.values, -0.3, 0.3)
    features.append(diff_2_normalized)
    feature_names.append('diff_2')

    return features, feature_names
```

设计理念：一阶差分反映功率变化的方向和幅度，二阶差分捕捉变化趋势的加速或减速特征，通过clip操作避免极值对模型的负面影响。

4.1.3 功率爬坡特征（Ramp Features）

风电功率的爬坡事件（快速上升或下降）是影响电网稳定性的重要因素。设计了10个爬坡相关特征，从多个角度刻画功率的快速变化特性。

Listing 3: 爬坡特征实现代码

```
if self.config.add_ramp_features:
    for window in self.config.ramp_windows:
        # 功率爬坡率：单位时间内功率变化幅度
        ramp_rate = power_series.diff(window).fillna(0).values
        features.append(ramp_rate)
        feature_names.append(f'ramp_rate_{window}')

        # 绝对爬坡率
        abs_ramp_rate = np.abs(ramp_rate)
        features.append(abs_ramp_rate)
        feature_names.append(f'abs_ramp_rate_{window}')

        # 功率爬坡强度（滚动窗口内最大绝对变化）
        ramp_intensity = power_series.diff().abs().rolling(
            window=window, min_periods=1).max().fillna(0).values
        features.append(ramp_intensity)
        feature_names.append(f'ramp_intensity_{window}')

        # 爬坡方向（上升，下降，稳定）=1=-1=0
        ramp_direction = np.sign(ramp_rate)
        features.append(ramp_direction)
        feature_names.append(f'ramp_direction_{window}')
```

设计理念：通过多个时间窗口的爬坡特征，综合刻画功率变化的强度、方向和持续性，为模型提供丰富的变化模式信息。

4.1.4 统计滚动特征（Statistical Rolling Features）

统计特征通过滚动窗口计算，捕捉局部时间段内功率分布的统计特性，包括中心趋势、离散程度和极值信息。

Listing 4: 统计特征实现代码

```
def add_statistical_features(self, power_series):
    features = []
    feature_names = []
    windows = self.config.statistical_windows

    for window in windows:
        # 滚动均值
        if self.config.enable_rolling_mean:
            rolling_mean = power_series.rolling(
                window=window, min_periods=1).mean().fillna(0).values
            features.append(rolling_mean)
            feature_names.append(f'rolling_mean_{window}')

        # 滚动标准差
        if self.config.enable_rolling_std:
            rolling_std = power_series.rolling(
                window=window, min_periods=1).std().fillna(0).values
            features.append(rolling_std)
            feature_names.append(f'rolling_std_{window}')

        # 滚动最大值和最小值
        if self.config.enable_rolling_max:
            rolling_max = power_series.rolling(
                window=window, min_periods=1).max().fillna(0).values
            features.append(rolling_max)
            feature_names.append(f'rolling_max_{window}')

        if self.config.enable_rolling_min:
            rolling_min = power_series.rolling(
                window=window, min_periods=1).min().fillna(0).values
            features.append(rolling_min)
            feature_names.append(f'rolling_min_{window}')

    return features, feature_names
```

4.1.5 异常检测特征（Anomaly Features）

识别并标记时间序列中的异常点，帮助模型对异常情况做出更鲁棒的预测。

Listing 5: 异常检测特征实现代码

```
def add_anomaly_features(self, power_series):
    features = []
    feature_names = []

    # 分数异常检测Z
    z_scores = np.abs((power_series - power_series.mean()) /
                      power_series.std())
    is_outlier = (z_scores > 2).astype(int).values
    features.append(is_outlier)
    feature_names.append('is_outlier')

    # 异常检测IQR
    Q1 = power_series.quantile(0.25)
    Q3 = power_series.quantile(0.75)
    IQR = Q3 - Q1
    is_iqr_outlier = ((power_series < (Q1 - 1.5 * IQR)) |
                      (power_series > (Q3 + 1.5 * IQR))).astype(int).values
    features.append(is_iqr_outlier)
    feature_names.append('is_iqr_outlier')
```

```
    # 局部异常因子
    windows = self.config.anomaly_windows
    for window in windows:
        local_mean = power_series.rolling(window=window, center=True).mean()
        local_std = power_series.rolling(window=window, center=True).std()
        local_anomaly = np.abs(power_series - local_mean) > (2 * local_std)
        features.append(local_anomaly.fillna(0).astype(int).values)
        feature_names.append(f'local_anomaly_{window}')

    return features, feature_names
```

4.2 模型架构

4.2.1 增强型LSTM模型

LSTM（Long Short-Term Memory）是一种特殊的RNN结构，通过引入输入门、遗忘门和输出门解决了传统RNN的梯度消失问题，能够有效捕捉长期依赖关系。本研究在传统LSTM基础上设计了增强型架构，融合了多头注意力机制和残差连接，显著提升了模型的表达能力和预测精度。

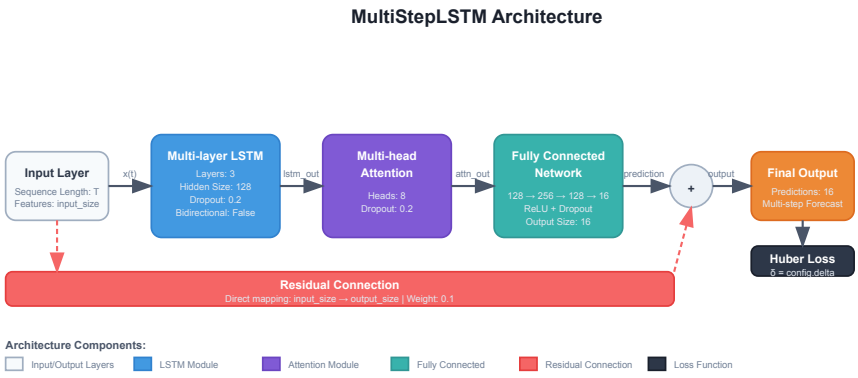


Figure 1: Attention-Enhanced LSTM Structure

模型配置：

Listing 6: Attention-Enhanced LSTM配置

```
class LSTMConfig:
    # 模型结构
    seq_length = 96        # 输入序列长度（小时）24
    pred_length = 16       # 预测序列长度（小时）4
    hidden_size = 128      # 隐藏层大小LSTM
    num_layers = 2         # 层数LSTM
    dropout = 0.3          # 率Dropout
    attention_heads = 8    # 注意力头数

    # 训练参数
    batch_size = 128
    learning_rate = 0.01
    num_epochs = 1000
    huber_delta = 1.0      # 损失参数Huber
```

模型架构：

Listing 7: Attention-Enhanced LSTM模型架构

```
class MultiStepLSTM(nn.Module):
    def __init__(self, input_size=1, hidden_size=128, num_layers=3,
                 output_size=16, dropout=0.2, attention_heads=8,
```

```
                 residual_weight=0.1, use_attention=True, use_residual=True):
        super(MultiStepLSTM, self).__init__()

        self.hidden_size = hidden_size
        self.num_layers = num_layers
        self.output_size = output_size
        self.residual_weight = residual_weight
        self.use_attention = use_attention
        self.use_residual = use_residual

        self.lstm = nn.LSTM(
            input_size=input_size,
            hidden_size=hidden_size,
            num_layers=num_layers,
            batch_first=True,
            dropout=dropout if num_layers > 1 else 0,
            bidirectional=False
        )

        if self.use_attention:
            self.attention = nn.MultiheadAttention(
                embed_dim=hidden_size,
                num_heads=attention_heads,
                dropout=dropout,
                batch_first=True
            )

        self.fc_layers = nn.Sequential(
            nn.Linear(hidden_size, hidden_size * 2),
            nn.ReLU(),
            nn.Dropout(dropout),
            nn.Linear(hidden_size * 2, hidden_size),
            nn.ReLU(),
            nn.Dropout(dropout),
            nn.Linear(hidden_size, output_size)
        )

        if self.use_residual:
            self.residual_fc = nn.Linear(input_size, output_size)

    def forward(self, x):
        batch_size = x.size(0)

        # 前向传播LSTM
        lstm_out, (hidden, cell) = self.lstm(x)

        if self.use_attention:
            attn_out, _ = self.attention(lstm_out, lstm_out, lstm_out)
            last_output = attn_out[:, -1, :]
        else:
            last_output = lstm_out[:, -1, :]

        # 全连接层预测
        prediction = self.fc_layers(last_output)

        if self.use_residual:
            residual = self.residual_fc(x[:, -1, :]).squeeze(-1)
            output = prediction + residual * self.residual_weight
        else:
            output = prediction

        return output

# 损失函数: Huber Loss
criterion = nn.HuberLoss(delta=config.huber_delta)
```

4.2.2 PatchTST模型

PatchTST（Patching Time Series Transformer）是一种为时间序列预测专门设计的轻量级Transformer变体。其核心思想是将时间序列划分为较短的"patches"，从而减少序列长度，提高计算效率。

Listing 8: PatchTST模型架构实现代码

```python
class PatchTSTConfig:
    # 数据参数
    seq_length = 96        # 输入序列长度
    pred_length = 16       # 预测序列长度
    patch_len = 8          # 每个的长度patch
    stride = 4             # 间步长patch

    # 模型参数
    d_model = 64           # 维度Transformer
    nhead = 8              # 注意力头数
    num_encoder_layers = 2 # 编码器层数
    dim_feedforward = 256 # 维度FFN
    dropout = 0.15

    # 训练参数
    batch_size = 64
    learning_rate = 0.01
    num_epochs = 500
    huber_delta = 1.0     # 损失参数Huber

class PatchTST(nn.Module):
    def __init__(self, config):
        super().__init__()
        self.patch_len = config.patch_len
        self.stride = config.stride
        self.patch_num = (config.seq_length - config.patch_len) // config.stride + 1

        # 嵌入Patch
        self.patch_embedding = nn.Linear(config.patch_len * input_size, config.d_model)

        # 位置编码
        self.pos_encoding = PositionalEncoding(config.d_model, config.dropout)

        # 编码器Transformer
        encoder_layer = nn.TransformerEncoderLayer(
            d_model=config.d_model, nhead=config.nhead,
            dim_feedforward=config.dim_feedforward, dropout=config.dropout,
            activation='gelu', batch_first=True, norm_first=True
        )
        self.transformer_encoder = nn.TransformerEncoder(encoder_layer, config.num_encoder_layers)

        # 输出投影
        self.output_projection = nn.Sequential(
            nn.Linear(config.d_model, config.d_model // 2),
            nn.GELU(), nn.Dropout(config.dropout),
            nn.Linear(config.d_model // 2, config.pred_length)
        )

    def create_patches(self, x):
        # x: [batch_size, seq_len, n_features]
        batch_size, seq_len, n_features = x.shape
        patches = []

        for i in range(0, seq_len - self.patch_len + 1, self.stride):
            patch = x[:, i:i+self.patch_len, :].reshape(batch_size, -1)
            patches.append(patch)

        return torch.stack(patches, dim=1) # [batch, patch_num, patch_len*features]

    def forward(self, x):
        # 创建并嵌入patches
        patches = self.create_patches(x) # [batch, patch_num, patch_len*features]
        patch_embeddings = self.patch_embedding(patches)

        # 添加位置编码
        patch_embeddings = self.pos_encoding(patch_embeddings)

        # 编码Transformer
        encoded = self.transformer_encoder(patch_embeddings)

        # 全局平均池化 + 输出投影
        pooled = encoded.mean(dim=1) # [batch, d_model]
        output = self.output_projection(pooled)
        return output
```

```

# 损失函数：Huber Loss
criterion = nn.HuberLoss(delta=config.huber_delta)
```

4.2.3 iTransformer模型

iTransformer（Inverted Transformer）是一种创新的Transformer变体，其核心思想是颠倒了标准Transformer中时间步和特征维度的处理方式。

Listing 9: iTransformer模型架构实现代码

```
class iTransformerConfig:
    # 数据参数
    seq_length = 96        # 输入序列长度
    pred_length = 16       # 预测序列长度

    # 模型参数
    d_model = 64          # 维度Transformer
    nhead = 8             # 注意力头数
    num_encoder_layers = 2 # 编码器层数
    dim_feedforward = 256 # 维度FFN
    dropout = 0.1

    # 训练参数
    batch_size = 128
    learning_rate = 0.0001
    num_epochs = 300
    huber_delta = 1.0     # 损失参数Huber

class iTransformer(nn.Module):
    def __init__(self, config):
        super().__init__()
        self.seq_len = config.seq_length
        self.pred_len = config.pred_length
        self.d_model = config.d_model

        # 变量嵌入：从序列长度投影到模型维度
        self.variable_embedding = nn.Linear(self.seq_len, self.d_model)

        # 变量位置编码
        self.variable_pos_encoding = VariablePositionalEncoding(self.d_model)

        # 编码器层Transformer
        self.encoder_layers = nn.ModuleList([
            iTransformerLayer(
                d_model=config.d_model, nhead=config.nhead,
                dim_feedforward=config.dim_feedforward,
                dropout=config.dropout, activation='gelu'
            )
            for _ in range(config.num_encoder_layers)
        ])

        # 输出投影
        self.output_projection = nn.Linear(self.d_model, self.pred_len)

    def forward(self, x):
        # x: [batch_size, seq_len, n_vars]
        # 转置并嵌入：[batch_size, n_vars, d_model]
        x = x.transpose(1, 2) # [batch, n_vars, seq_len]
        x = self.variable_embedding(x) # [batch, n_vars, d_model]

        # 添加变量位置编码
        x = self.variable_pos_encoding(x)

        # 通过编码器层
        for layer in self.encoder_layers:
            x = layer(x)

        # 输出投影
        output = self.output_projection(x) # [batch, n_vars, pred_len]

        # 只返回原始功率特征的预测（第一个变量）
```

```
        return output[:, 0, :] # [batch, pred_len]

# 损失函数：Huber Loss
criterion = nn.HuberLoss(delta=config.huber_delta)
```

4.3 训练细节

损失函数设计

选择Huber损失函数作为统一的损失函数，相比于MSE损失对异常值更加鲁棒：

$$L_{\text{Huber}}(y, \hat{y}) = \begin{cases} \frac{1}{2}(y - \hat{y})^2 & \text{if } |y - \hat{y}| \leq \delta \\ \delta(|y - \hat{y}| - \frac{1}{2}\delta) & \text{otherwise} \end{cases} \tag{1}$$

其中$\delta = 1.0$。Huber损失结合了MSE和MAE的优点，在小误差时表现为二次函数，在大误差时表现为线性函数。

Listing 10: 训练配置实现代码

```
# 损失函数
criterion = nn.HuberLoss(delta=1.0)

# 优化器配置
optimizer = optim.AdamW(model.parameters(),
                        lr=config.learning_rate,
                        weight_decay=1e-4)

# 自适应学习率调度
scheduler = optim.lr_scheduler.ReduceLROnPlateau(
    optimizer, mode='min', factor=0.5, patience=8, verbose=True
)

# 早停机制
early_stop_patience = 20
early_stop_min_delta = 0.0001
```

优化策略

- 优化器：AdamW，权重衰减：1×10^{-4}
- 学习率调度：ReduceLROnPlateau，因子：0.5，耐心值：8
- 早停机制：耐心值20，最小改善：1×10^{-4}
- 硬件环境：RTX A6000 GPU

5 实验结果与综合分析

5.1 评估指标体系

本研究采用三个核心评估指标，从不同角度全面评估模型性能：

5.1.1 传统回归指标

均方根误差（RMSE）

$$\text{RMSE} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2} \tag{2}$$

RMSE对大误差敏感，能够有效惩罚预测偏差较大的样本。

平均绝对误差（MAE）

$$\text{MAE} = \frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i| \tag{3}$$

MAE提供了预测误差的直观度量，对所有误差等权重处理，不易受极值影响。

5.1.2 风电行业专用指标

功率预测准确率（CR）

CR是风电行业的专用评估指标，能够更好地反映预测精度对实际应用的价值：

Listing 11: CR指标计算实现代码

```python
def calculate_CR(PM, PP):
    """
    PM: 实测功率值
    PP: 预测功率值
    """
    N = len(PM)
    Ri = np.zeros(N)

    for i in range(N):
        if PM[i] > 0.2:
            Ri[i] = (PM[i] - PP[i]) / PM[i]
        else:
            Ri[i] = (PM[i] - PP[i]) / 0.2

    rms_error = np.sqrt(np.mean(Ri**2))
    CR = (1 - rms_error) * 100
    return CR
```

数学表达式为：

$$R_i = \begin{cases} \frac{PM_i - PP_i}{PM_i} & \text{if } PM_i > 0.2 \\ \frac{PM_i - PP_i}{0.2} & \text{if } PM_i \leq 0.2 \end{cases} \tag{4}$$

$$\text{CR} = \left(1 - \sqrt{\frac{1}{N}\sum_{i=1}^{N} R_i^2}\right) \times 100\% \tag{5}$$

其中PM_i为实测功率值，PP_i为预测功率值。

5.2 深度学习模型整体性能分析

5.2.1 主要实验结果

基于统一的实验设置，三种深度学习模型在16步预测任务上的综合性能对比如表2所示：

Table 2: 深度学习模型性能对比

模型	RMSE	MAE	Average CR(%)	参数量
Attention-Enhanced LSTM	0.1283	0.0880	**62.56**	0.36M
PatchTST	**0.1262**	**0.0862**	58.91	0.13M
iTransformer	0.1354	0.0970	52.54	0.11M

5.2.2 关键发现分析

Attention-Enhanced LSTM模型：

- CR准确率表现最优：62.56%，符合风电行业核心评估标准
- 特征适应性出色：对57个工程特征表现出良好的适应性

PatchTST模型：

- 传统指标表现最佳：RMSE（0.1262）和MAE（0.0862）均为最优
- 参数效率显著：Patch机制有效降低了复杂度，仅0.13M参数即达到接近Attention-Enhanced LSTM的性能

iTransformer模型：

- 当前性能受限：CR指标52.54%，在现有设置下表现相对较低，变量注意力机制在多变量场景下优势未充分体现

5.2.3 可视化结果分析（以Attention-Enhanced LSTM模型为例）

图2展示了多步预测效果。蓝色曲线为真实值，红色曲线为预测值。可以清晰观察到：

- 短期预测（1、4步）与真实值高度吻合
- 中期预测（8步）保持较好的趋势跟踪能力
- 长期预测（16步）虽有偏差但仍能捕捉主要变化模式

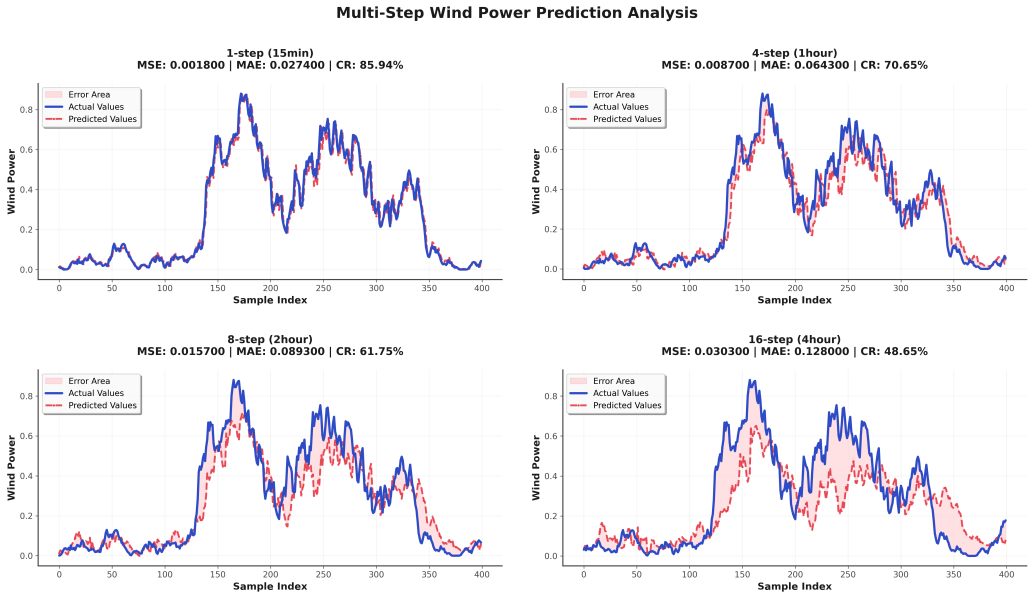


Figure 2: Attention-Enhanced LSTM多步预测效果展示

图3显示了RMSE、MAE、CR三个关键指标随预测步长的演变规律：

- CR指标从第1步的85.94%逐步下降到第16步的48.65%
- RMSE和MAE呈现稳定的递增趋势
- 所有指标都表现出预测精度随时间衰减的合理规律

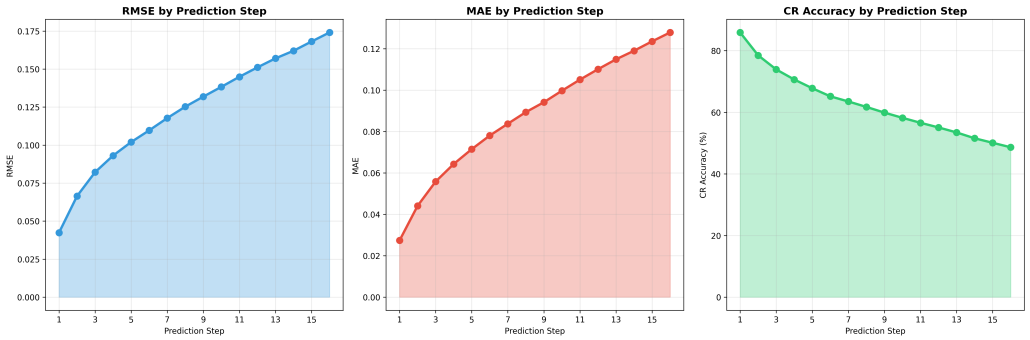


Figure 3: 预测指标随步长变化趋势

图4通过小提琴图展示了16个预测步长的误差分布特征：

- 短期预测误差分布集中，说明预测稳定性好
- 随着预测步长增加，误差分布逐渐扩散

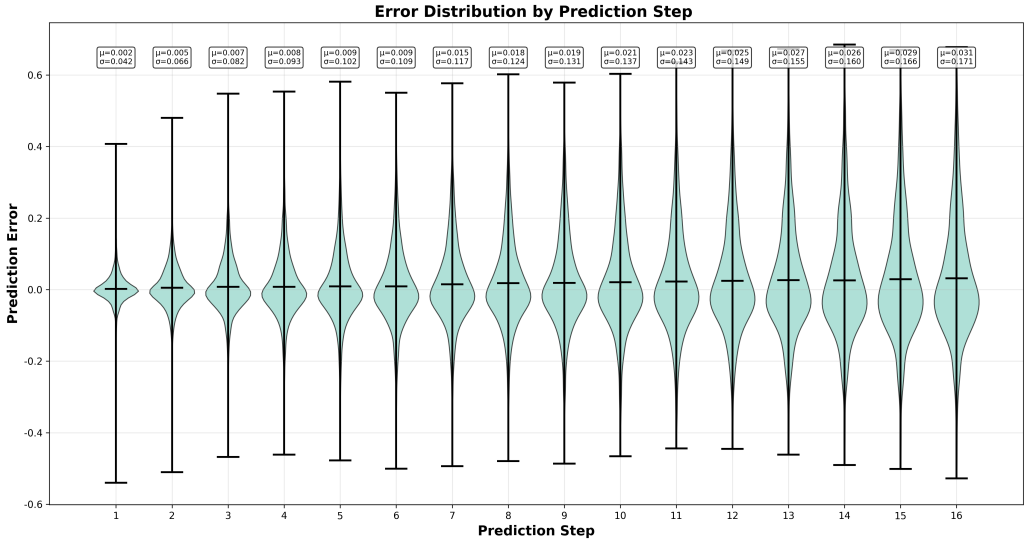


Figure 4: 预测误差分布特征分析

5.3 预测步长分析

表3展示了各模型在不同预测步长上的详细性能比较：

Table 3: 不同预测步长性能对比（选取关键步长）

步长	Attention-Enhanced LSTM			PatchTST			iTransformer		
	MSE	MAE	CR(%)	MSE	MAE	CR(%)	MSE	MAE	CR(%)
1	0.0018	0.0274	**85.94**	**0.0017**	**0.0263**	85.97	0.0042	0.0462	76.86
4	**0.0087**	0.0643	**70.65**	0.0089	**0.0637**	69.00	0.0111	0.0760	63.19
8	0.0157	0.0893	**61.75**	**0.0154**	**0.0885**	57.49	0.0180	0.0994	53.12
16	0.0303	0.1278	**48.65**	**0.0283**	**0.1244**	42.26	0.0305	0.1310	38.03

5.4 特征工程消融实验

基于Attention-Enhanced LSTM模型进行的特征消融实验结果如表4所示：

Table 4: 特征工程消融实验结果

特征组合	特征数量	CR(%)	性能提升
仅原始功率	1	57.07	baseline
+滞后特征	9	58.68	+1.61
+趋势特征	3	57.71	+0.64
+爬坡特征	10	52.23	-4.84
+异常特征	5	58.61	+1.54
+统计特征	31	60.26	+3.19
全部特征	57	62.56	+5.49

注意到：

1. **统计特征**对baseline带来显著提升（+3.19%），说明局部统计信息的重要性
2. **爬坡特征**在当前特征组合下对baseline产生负面影响（-4.84%）。但在加入其他所有特征后，其综合收益为正，推测是由于测试集样本不足，存在不稳定性或者模型本身表达能力的缺陷

6 基于CatBoost的机器学习方法

除了深度学习方法外，我们还实现了基于传统机器学习的CatBoost方法，并进行了系统的特征选择策略对比分析。

6.1 方法概述

CatBoost是基于梯度提升决策树的机器学习算法，在处理一维数据上有着优势。本研究采用独立多输出策略，即为每个预测步长训练独立的CatBoost模型。

任务定义：

- **输入**: 过去96个时间点的风电功率数据（24小时，15分钟间隔）
- **输出**: 未来16个时间点的风电功率预测（4小时，15分钟间隔）
- **方法**: 16个独立的CatBoost模型，每个模型预测一个特定的未来时间点

6.2 模型配置与特征选择策略

我们比较了四种不同的特征选择方法：

6.2.1 策略1：完整时间序列（96个特征）

使用全部96个历史时间点作为特征(24小时)。

6.2.2 策略2：基于SHAP的选择（8个特征）

采用SHAP（SHapley Additive exPlanations）分析，为每个预测时间点从原始96个时间点中识别8个最重要的时间点。

6.2.3 策略3：根据经验选择3个特征

仅使用三个手动选择的特征：

- **i-1_power**: 最近功率值（t-1）
- **min_power**: 历史窗口中的最小功率
- **max_power**: 历史窗口中的最大功率

6.2.4 策略4：单特征（1个特征）

仅使用最近的功率值：

- **i-1_power**: 最近功率值，即当前状态

6.3 实验结果

6.3.1 整体性能对比

基于16个独立CatBoost模型的综合实验结果如表5所示：

Table 5: CatBoost不同特征策略性能对比

方法	特征数量	Average RMSE	Average MAE	Average CR (%)	训练时间(s)
96点模型	96	0.1324	0.0988	57.61	2m34.1s
8特征模型(SHAP选择)	8	0.1232	0.0900	58.86	1m2.4s
3特征模型(统计特征)	3	0.1222	0.0883	**59.12**	11.0s
单特征模型(t-1)	1	**0.1221**	**0.0881**	58.51	4.0s

实验环境： AMD EPYC 7513 32-Core Processor，同时训练2个模型

6.3.2 分步性能对比

不同特征选择策略在各个预测步长上的CR指标表现如表6所示：

Table 6: CatBoost不同特征策略分步性能对比

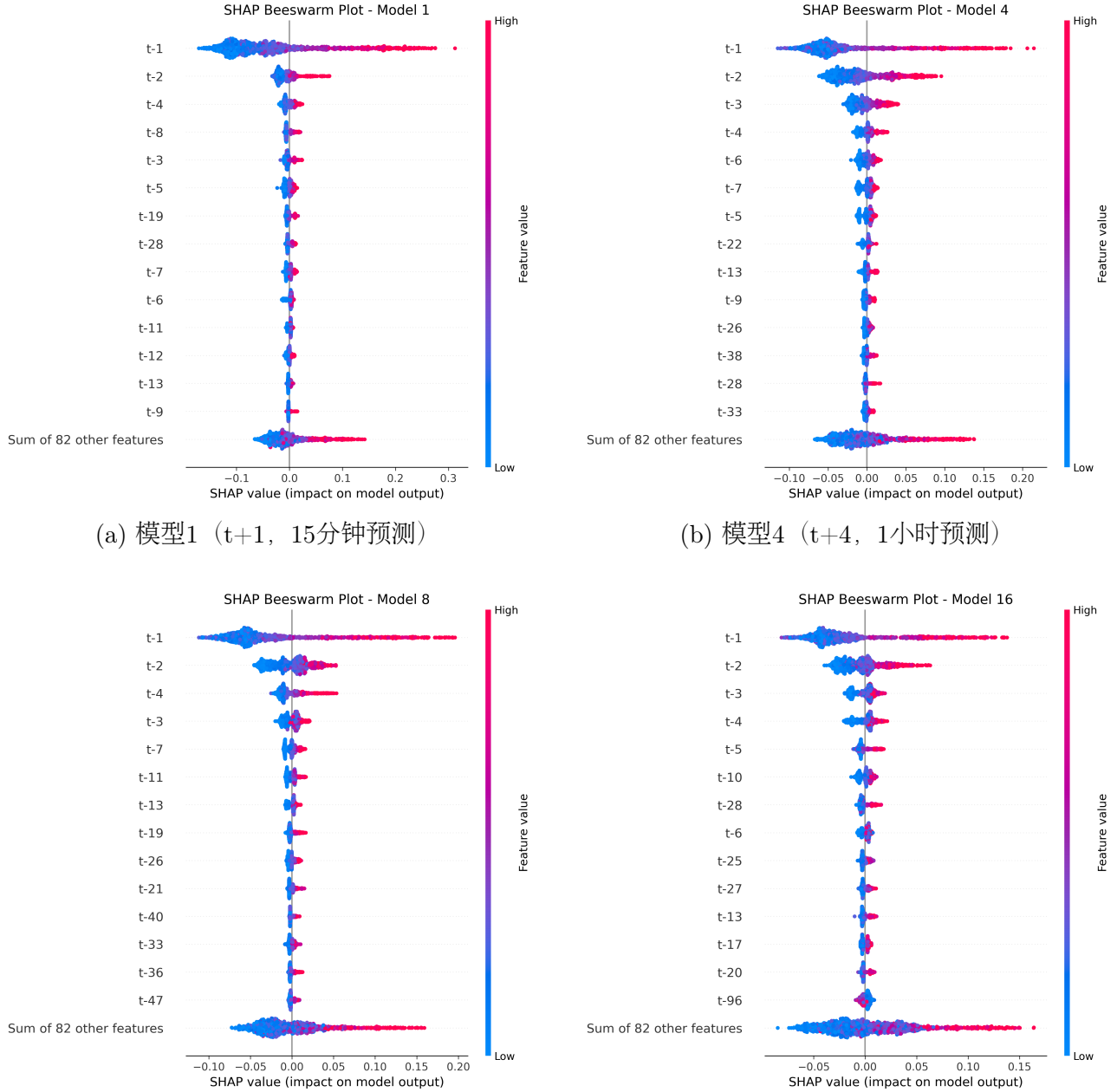
预测步长	96特征	8特征（基于SHAP）	3特征	1特征
t+1	83.73	**85.55**	85.44	85.51
t+4	67.34	68.61	**69.09**	68.58
t+8	57.27	**58.39**	58.12	57.74
t+16	40.65	41.26	**42.17**	41.29

性能总结：

特征数量：$96 \rightarrow 8 \rightarrow 3 \rightarrow 1$
CR性能：$57.61\% \rightarrow 58.86\% \rightarrow 59.12\% \rightarrow 58.51\%$
计算复杂度：高 $\rightarrow$ 中 $\rightarrow$ 低 $\rightarrow$ 极低

6.4 SHAP特征重要性分析

通过SHAP分析，我们识别了不同预测步长下的关键特征模式。图5展示了四个代表性模型的SHAP分析结果，揭示了特征重要性和影响模式。



(a) 模型1（t+1，15分钟预测）　(b) 模型4（t+4，1小时预测）

(c) 模型8（t+8，2小时预测）　(d) 模型16（t+16，4小时预测）

Figure 5: 不同预测步长模型的SHAP特征重要性分析

SHAP分析结果表明：(1) t-1特征在所有模型中均显示最强正向影响，证实了最近时刻功率值的重要性；(2) 较高功率值往往对预测产生正向贡献，体现功率连续性特征；(3) 不同预测步长的模型会自适应选择合适的历史时间点。

6.5 CatBoost方法的特点与优势

1. **计算效率高**：相比深度学习模型，在合适的特征数量下，训练时间显著缩短
2. **解释性强**：通过SHAP分析可以清晰理解特征重要性
3. **抗过拟合能力强**：内置正则化机制，在小数据集上表现稳定

6.6 与深度学习方法的对比

表7展示了不同方法类型的性能对比：

Table 7: 深度学习与机器学习方法对比

方法类型	代表模型	Average CR (%)
深度学习	Attention-Enhanced LSTM	62.56
深度学习	PatchTST	58.91
深度学习	iTransformer	52.54
机器学习	CatBoost (3特征)	59.12
机器学习	CatBoost (8特征)	58.86

CatBoost方法在保持较高预测精度的同时，具有训练快速、解释性强（结合SHAP)和无需复杂特征工程的优势

7 结论与未来工作

7.1 结论

本研究通过构建一套完整的风电功率预测系统，深入比较了深度学习与机器学习方法在短期风电功率预测中的性能表现，得出以下重要结论：

深度学习模型：

1. **Attention-Enhanced LSTM**：在功率预测准确率（CR）指标上达到62.56%，表现最优。其强大的序列建模能力和对特征工程的敏感性使其在风电功率预测任务中脱颖而出。
2. **PatchTST**：在传统评估指标RMSE（0.1262）和MAE（0.0862）上表现最佳，同时具有较低的参数量（0.13M），体现了良好的效率-性能平衡。
3. **iTransformer**：虽然在当前数据规模下CR指标为52.54%，但其变量注意力机制在更大规模数据集上可能展现更强优势。

机器学习方法：

4. **CatBoost**：仅使用3个简单特征即可达到59.12%的CR性能，训练时间仅需11秒，展现了传统机器学习方法的实用价值。

7.2 未来研究方向

本研究深度学习三个模型采用手动调参，未来可以进一步优化超参数和模型基本设置，比如采用网格搜索等方式实现自动调参。如果要应用到工业界，建议增加不确定性的评估，比如给模型的输出增加置信度标签，或者模型输出调整为一个区间而非定值。

References

[1] Giebel, G., et al. (2011). The state-of-the-art in short-term prediction of wind power: A literature overview.

[2] Hochreiter, S., & Schmidhuber, J. (1997). Long short-term memory. Neural computation, 9(8), 1735-1780.

[3] Bai, S., Kolter, J. Z., & Koltun, V. (2018). An empirical evaluation of generic convolutional and recurrent networks for sequence modeling. arXiv preprint arXiv:1803.01271.

[4] Vaswani, A., et al. (2017). Attention is all you need. In Advances in neural information processing systems.

[5] Zhou, H., et al. (2021). Informer: Beyond efficient transformer for long sequence time-series forecasting. In Proceedings of AAAI.

[6] Wu, H., et al. (2021). Autoformer: Decomposition transformers with auto-correlation for long-term series forecasting. Advances in Neural Information Processing Systems, 34.

[7] Nie, Y., et al. (2022). A time series is worth 64 words: Long-term forecasting with transformers. In International Conference on Learning Representations.

[8] Liu, T., et al. (2023). iTransformer: Inverted Transformers Are Effective for Time Series Forecasting. arXiv preprint arXiv:2310.06625.

[9] Prokhorenkova, L., et al. (2018). CatBoost: unbiased boosting with categorical features. In Advances in neural information processing systems.