基于BP神经网络的多时间尺度风电功率预测研究

张晋铭

June 7, 2025

Abstract

风电功率的准确预测对电网调度和风能利用至关重要。本文构建了基于BP神经网络的多时间尺度风电功率预测模型，采用Adam优化器、早停机制和Xavier权重初始化等技术改进训练过程。实验结果显示：15分钟预测的CR准确度为78.38%，RMSE为0.0701；1小时预测的CR准确度为65.48%，RMSE为0.1114；4小时预测的CR准确度为48.27%，RMSE为0.1873。相比标准SGD，Adam优化器将训练效率提升了70.3%，预测精度提升约6%。Xavier初始化改善了梯度传播问题，早停机制避免了过拟合现象，使模型能在较少轮次内收敛。本文建立的多时间尺度预测框架可满足风电场不同调度需求。完整代码已开源：https://github.com/Jinming00/EE。

关键词：BP神经网络、风电功率预测、Adam优化器、多时间尺度、早停机制

Contents

1 引言

1.1 研究背景

风力发电在全球能源转型中地位重要，但其间歇性和随机性给电网运行带来挑战。准确的功率预测是解决这一问题的关键，既能优化调度又能提高风能利用率，降低运行成本。

目前的风电预测方法主要分为物理模型和统计模型两类。物理模型依靠数值天气预报和风机特性曲线，物理意义明确，但难以准确描述复杂的边界层流动和地形效应；统计模型通过历史数据建立数学关系，在处理非线性和长期依赖方面有限。

BP神经网络是经典的前馈网络，具备强大的非线性拟合能力，在时间序列预测中应用广泛。其通过误差反向传播调整权重，能处理复杂的非线性映射，为风电功率预测提供了有效途径。

1.2 研究目标与贡献

本研究的主要目标是构建基于BP神经网络的多时间尺度风电功率预测模型，通过技术改进提升预测性能和训练效率。主要贡献包括：

1. **多时间尺度预测**：实现15分钟、1小时和4小时三种时间尺度的功率预测，适应不同调度需求
2. **Adam优化算法**：相比传统SGD，提升收敛速度和预测精度
3. **早停机制**：避免过拟合，提高模型泛化能力
4. **Xavier权重初始化**：改善梯度传播，增强训练稳定性

2 相关工作

BP神经网络自1986年Rumelhart等人提出以来，在机器学习和时间序列预测中应用广泛 [1]。风电功率预测领域的研究主要集中在网络结构优化、算法改进和特征工程等方面。

传统BP神经网络训练多采用标准梯度下降法，但存在收敛慢、易陷入局部最优等问题。为此，研究者提出了多种改进优化算法。其中，Adam优化器 [2]结合了自适应学习率和动量方法的优点，在深度学习中表现出色，为BP神经网络改进提供了新思路。

在风电功率预测应用中，时间尺度选择对预测精度和实用性影响显著。短期预测（15分钟至1小时）主要用于实时调度和功率平衡，中期预测（1小时至4小时）更多应用于机组启停计划和备用容量安排。多时间尺度预测模型能够更好满足实际需求。

3 数据集与预处理

3.1 数据集描述

本研究使用的数据集包含某风电场的实际功率数据，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

3.2 数据预处理

3.2.1 异常值处理

数据预处理首先需要处理异常值，特别是物理上不合理的负功率值：

Listing 1: 异常值处理代码

```
# 数据加载
train_data = pd.read_excel('../data/train.xlsx')
test_data = pd.read_excel('../data/test.xlsx')

# 提取功率数据（第一列是实际功率 - 已归一化）
train_power = train_data.iloc[:, 0].values
test_power = test_data.iloc[:, 0].values

# 检查并清理训练数据中的负值
negative_count_train = np.sum(train_power < 0)
if negative_count_train > 0:
    print(f"发现训练数据中有 {negative_count_train} 个负值，将它们转换为零。")
    train_power = np.maximum(train_power, 0)

# 检查并清理测试数据中的负值
negative_count_test = np.sum(test_power < 0)
if negative_count_test > 0:
    print(f"发现测试数据中有 {negative_count_test} 个负值，将它们转换为零。")
    test_power = np.maximum(test_power, 0)
```

3.2.2 时间窗口数据集构建

为了构建适用于不同预测时长的训练数据集，设计了滑动窗口方法：

Listing 2: 时间窗口数据集构建

```
def create_dataset_for_horizon(data, input_size=96, predict_horizon=1, step=1):

    X, y = [], []
    for i in range(0, len(data) - input_size - predict_horizon, step):
        X.append(data[i:i+input_size])
        y.append(data[i+input_size+predict_horizon-1])
    return np.array(X), np.array(y).reshape(-1, 1)

# 为个不同时长创建训练数据3
input_size = 96
step = 1
X_train_15min, y_train_15min = create_dataset_for_horizon(train_power_scaled, input_size, 1,
    ↪ step)
```

```
X_train_1hour, y_train_1hour = create_dataset_for_horizon(train_power_scaled, input_size, 4,
    ↪ step)
X_train_4hour, y_train_4hour = create_dataset_for_horizon(train_power_scaled, input_size, 16,
    ↪ step)
```

输入窗口大小设置为96个时间点，对应24小时的历史数据。预测时长分别设置为1、4、16个时间点，对应15分钟、1小时和4小时的预测任务。步长默认为1以获得更多的数据对。

3.2.3 数据集分割

采用顺序分割方法，将数据按照时间顺序划分为训练集和验证集：

Listing 3: 数据集分割

```
def split_train_val(X, y, train_ratio=0.8):
    """按照顺序分割数据，前用于训练，后用于验证80%20%"""
    train_size = int(len(X) * train_ratio)
    X_train = X[:train_size]
    y_train = y[:train_size]
    X_val = X[train_size:]
    y_val = y[train_size:]
    return X_train, y_train, X_val, y_val

# 为每个预测时长分割数据
X_train_15min_split, y_train_15min_split, X_val_15min, y_val_15min = split_train_val(
    ↪ X_train_15min, y_train_15min)
X_train_1hour_split, y_train_1hour_split, X_val_1hour, y_val_1hour = split_train_val(
    ↪ X_train_1hour, y_train_1hour)
X_train_4hour_split, y_train_4hour_split, X_val_4hour, y_val_4hour = split_train_val(
    ↪ X_train_4hour, y_train_4hour)
```

训练集与验证集按8:2的比例进行划分，确保模型能够在独立的验证集上评估性能。

4 方法与模型

4.1 BP神经网络架构

本研究采用三层BP神经网络结构，包括输入层、隐藏层和输出层。网络采用全连接方式，激活函数选择Sigmoid函数。

4.1.1 网络初始化

Listing 4: BP神经网络初始化

```
class BPNeuralNetwork:
    def __init__(self, input_size, hidden_size, output_size=1, learning_rate=0.01, use_adam=
        ↪ True):
        """神经网络初始化
        BP
        input_size: 输入层节点数
        hidden_size: 隐藏层节点数
        output_size: 输出层节点数
        learning_rate: 学习率
        use_adam: 是否使用优化器Adam
        """
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.output_size = output_size
        self.learning_rate = learning_rate
        self.use_adam = use_adam

```

```
        # 权重初始化Xavier
        self.W1 = np.random.normal(0, np.sqrt(2.0/(input_size + hidden_size)),
                                   (input_size, hidden_size))
        self.b1 = np.zeros((1, hidden_size))
        self.W2 = np.random.normal(0, np.sqrt(2.0/(hidden_size + output_size)),
                                   (hidden_size, output_size))
        self.b2 = np.zeros((1, output_size))

        # 优化器参数Adam
        if self.use_adam:
            self.beta1 = 0.9
            self.beta2 = 0.999
            self.epsilon = 1e-8
            self.t = 0  # 时间步

            # 一阶动量
            self.m_W1 = np.zeros_like(self.W1)
            self.m_b1 = np.zeros_like(self.b1)
            self.m_W2 = np.zeros_like(self.W2)
            self.m_b2 = np.zeros_like(self.b2)

            # 二阶动量
            self.v_W1 = np.zeros_like(self.W1)
            self.v_b1 = np.zeros_like(self.b1)
            self.v_W2 = np.zeros_like(self.W2)
            self.v_b2 = np.zeros_like(self.b2)
```

网络采用Xavier初始化方法，有效改善了梯度传播问题。权重初始化遵循正态分布，标准差根据输入和输出维度动态调整。

4.1.2 前向传播过程

Listing 5: 前向传播实现

```
def sigmoid(self, x):
    """Sigmoid"""
    return 1.0 / (1.0 + np.exp(-x))

def sigmoid_derivative(self, x):
    """其导数"""
    return x * (1.0 - x)

def forward(self, X):

    # 输入层到隐藏层
    self.z1 = np.dot(X, self.W1) + self.b1
    self.a1 = self.sigmoid(self.z1)

    # 隐藏层到输出层
    self.z2 = np.dot(self.a1, self.W2) + self.b2
    self.a2 = self.sigmoid(self.z2)

    return self.a2
```

前向传播过程包括两个线性变换和两个非线性激活，通过Sigmoid函数引入非线性特性。

4.1.3 反向传播和参数更新

Listing 6: 反向传播实现

```
def backward(self, X, y, output):
    """反向传播和参数更新

    X: 输入数据
    y: 真实标签
    output: 网络输出
    """
    m = X.shape[0]  # 批次大小
```

```
9
10      # 计算输出层误差
11      output_error = output - y
12      output_delta = output_error * self.sigmoid_derivative(output)
13
14      # 计算隐藏层误差
15      hidden_error = output_delta.dot(self.W2.T)
16      hidden_delta = hidden_error * self.sigmoid_derivative(self.a1)
17
18      # 计算梯度
19      dW2 = np.dot(self.a1.T, output_delta) / m
20      db2 = np.sum(output_delta, axis=0, keepdims=True) / m
21      dW1 = np.dot(X.T, hidden_delta) / m
22      db1 = np.sum(hidden_delta, axis=0, keepdims=True) / m
23
24      # 参数更新 (Adam vs SGD)
25      if self.use_adam:
26          self._adam_update(dW1, db1, dW2, db2)
27      else:
28          self._sgd_update(dW1, db1, dW2, db2)
29
30      # 计算损失 (MSE)
31      mse = np.mean(np.sum(np.square(output_error), axis=1))
32      return mse
```

反向传播算法通过链式法则计算各层的梯度，并根据选择的优化器进行参数更新。

4.2 Adam优化器实现

Adam优化器结合了自适应学习率和动量方法的优点，能够有效提升训练效率：

Listing 7: Adam优化器实现

```
1   def _adam_update(self, dW1, db1, dW2, db2):
2       """优化器参数更新Adam"""
3       self.t += 1
4
5       # 更新一阶动量梯度的指数移动平均 ()
6       self.m_W2 = self.beta1 * self.m_W2 + (1 - self.beta1) * dW2
7       self.m_b2 = self.beta1 * self.m_b2 + (1 - self.beta1) * db2
8       self.m_W1 = self.beta1 * self.m_W1 + (1 - self.beta1) * dW1
9       self.m_b1 = self.beta1 * self.m_b1 + (1 - self.beta1) * db1
10
11      # 更新二阶动量梯度平方的指数移动平均 ()
12      self.v_W2 = self.beta2 * self.v_W2 + (1 - self.beta2) * (dW2 ** 2)
13      self.v_b2 = self.beta2 * self.v_b2 + (1 - self.beta2) * (db2 ** 2)
14      self.v_W1 = self.beta2 * self.v_W1 + (1 - self.beta2) * (dW1 ** 2)
15      self.v_b1 = self.beta2 * self.v_b1 + (1 - self.beta2) * (db1 ** 2)
16
17      # 偏置校正
18      m_W2_hat = self.m_W2 / (1 - self.beta1 ** self.t)
19      m_b2_hat = self.m_b2 / (1 - self.beta1 ** self.t)
20      m_W1_hat = self.m_W1 / (1 - self.beta1 ** self.t)
21      m_b1_hat = self.m_b1 / (1 - self.beta1 ** self.t)
22
23      v_W2_hat = self.v_W2 / (1 - self.beta2 ** self.t)
24      v_b2_hat = self.v_b2 / (1 - self.beta2 ** self.t)
25      v_W1_hat = self.v_W1 / (1 - self.beta2 ** self.t)
26      v_b1_hat = self.v_b1 / (1 - self.beta2 ** self.t)
27
28      # 更新规则Adam: θ = θ - α * m / √(v + ε)
29      self.W2 += self.learning_rate * m_W2_hat / (np.sqrt(v_W2_hat) + self.epsilon)
30      self.b2 += self.learning_rate * m_b2_hat / (np.sqrt(v_b2_hat) + self.epsilon)
31      self.W1 += self.learning_rate * m_W1_hat / (np.sqrt(v_W1_hat) + self.epsilon)
32      self.b1 += self.learning_rate * m_b1_hat / (np.sqrt(v_b1_hat) + self.epsilon)
33
34  def _sgd_update(self, dW1, db1, dW2, db2):
35      """标准梯度下降参数更新"""
36      self.W2 += self.learning_rate * dW2
37      self.b2 += self.learning_rate * db2
38      self.W1 += self.learning_rate * dW1
39      self.b1 += self.learning_rate * db1
```

Adam优化器通过维护梯度的一阶和二阶动量，实现自适应学习率调整，显著提升了收敛速度和稳定性。

Adam优化器的主要优势包括：

1. **自适应学习率**：根据梯度的历史信息动态调整学习率，避免了手动调参的复杂性
2. **动量机制**：通过一阶和二阶动量的结合，有效克服了局部最优和震荡问题
3. **快速收敛**：在相同条件下，训练时间减少96.6%，大幅提升了训练效率
4. **稳定性好**：偏置校正机制确保了训练初期的稳定性

4.3 训练过程和早停机制

Listing 8: 训练过程实现

```
def train(self, X_train, y_train, X_val=None, y_val=None, epochs=100,
          batch_size=32, early_stopping=True, patience=50, min_delta=1e-6):
    """

    X_train, y_train: 训练数据
    X_val, y_val: 验证数据
    epochs: 最大训练轮次
    batch_size: 批次大小
    early_stopping: 是否启用早停
    patience: 早停耐心值
    min_delta: 性能改善最小阈值

    """
    train_losses = []
    val_losses = []
    val_cr_scores = []
    n_samples = X_train.shape[0]

    # 早停相关变量
    best_cr_score = -np.inf
    patience_counter = 0
    best_weights = None

    for epoch in range(epochs):
        # 数据打乱
        indices = np.random.permutation(n_samples)
        X_shuffled = X_train[indices]
        y_shuffled = y_train[indices]

        epoch_loss = 0
        # 批次训练
        for i in range(0, n_samples, batch_size):
            end = min(i + batch_size, n_samples)
            X_batch = X_shuffled[i:end]
            y_batch = y_shuffled[i:end]

            # 前向传播
            output = self.forward(X_batch)
            # 反向传播
            batch_loss = self.backward(X_batch, y_batch, output)
            epoch_loss += batch_loss * (end - i) / n_samples

        train_losses.append(epoch_loss)

        # 验证集评估
        if X_val is not None and y_val is not None:
            val_output = self.forward(X_val)
            val_loss = np.mean(np.square(val_output - y_val))
            val_losses.append(val_loss)

            # 计算指标CR
            val_cr_score = self.calculate_CR(y_val, val_output)
            val_cr_scores.append(val_cr_score)

```

```
            # 早停检查
            if early_stopping:
                if val_cr_score > best_cr_score + min_delta:
                    best_cr_score = val_cr_score
                    patience_counter = 0
                    # 保存最佳权重
                    best_weights = {
                        'W1': self.W1.copy(), 'b1': self.b1.copy(),
                        'W2': self.W2.copy(), 'b2': self.b2.copy()
                    }
                else:
                    patience_counter += 1

                if patience_counter >= patience:
                    print(f"早停触发，在第 {epoch+1} 轮停止训练")
                    # 恢复最佳权重
                    if best_weights is not None:
                        self.W1 = best_weights['W1']
                        self.b1 = best_weights['b1']
                        self.W2 = best_weights['W2']
                        self.b2 = best_weights['b2']
                    break

        if (epoch + 1) % 100 == 0:
            print(f"Epoch {epoch+1}/{epochs}, Loss: {epoch_loss:.6f}")

    return train_losses, val_losses, val_cr_scores
```

早停机制通过监控验证集上的CR指标，在性能不再改善时及时停止训练，有效防止过拟合并提升训练效率。

5 实验设置与模型训练

5.1 模型参数配置

针对三个不同的预测时长，设置了相应的模型参数：

Listing 9: 模型参数配置

```
# 设置共享模型参数
max_epochs = 10000

# 为三个不同的预测模型设置不同的参数

hidden_size_15min = 16
batch_size_15min = 64
learning_rate_15min = 0.01

hidden_size_1hour = 64
batch_size_1hour = 64
learning_rate_1hour = 0.01

hidden_size_4hour = 128
batch_size_4hour = 128
learning_rate_4hour = 0.005

# 早停参数
patience = 50
min_delta = 0.0005
```

参数设置基于实验调优，考虑了不同预测时长的复杂度差异。较长的预测时长采用更大的隐藏层规模和批次大小，以增强模型的表达能力。

5.2 训练执行过程

Listing 10: 模型训练执行

```
print("\训练分钟预测模型n15...")
model_15min = BPNeuralNetwork(input_size, hidden_size_15min, 1, learning_rate_15min, use_adam=
    ↪ True)
train_losses_15min, val_losses_15min, val_cr_scores_15min = model_15min.train(
    X_train_15min_split, y_train_15min_split,
    X_val_15min, y_val_15min,
    max_epochs, batch_size_15min, shuffle=True,
    early_stopping=True, patience=patience, min_delta=min_delta
)

print("\训练小时预测模型n1...")
model_1hour = BPNeuralNetwork(input_size, hidden_size_1hour, 1, learning_rate_1hour, use_adam=
    ↪ True)
train_losses_1hour, val_losses_1hour, val_cr_scores_1hour = model_1hour.train(
    X_train_1hour_split, y_train_1hour_split,
    X_val_1hour, y_val_1hour,
    max_epochs, batch_size_1hour, shuffle=True,
    early_stopping=True, patience=patience, min_delta=min_delta
)

print("\训练小时预测模型n4...")
model_4hour = BPNeuralNetwork(input_size, hidden_size_4hour, 1, learning_rate_4hour, use_adam=
    ↪ True)
train_losses_4hour, val_losses_4hour, val_cr_scores_4hour = model_4hour.train(
    X_train_4hour_split, y_train_4hour_split,
    X_val_4hour, y_val_4hour,
    max_epochs, batch_size_4hour, shuffle=True,
    early_stopping=True, patience=patience, min_delta=min_delta
)
```

三个模型分别针对不同的预测时长进行训练，均采用相同的训练策略但使用不同的超参数配置。

6 实验结果与分析

6.1 多时间尺度预测结果

本研究实现了15分钟、1小时和4小时三个时间尺度的风电功率预测。图1展示了三个不同时间尺度的预测结果与真实值对比：

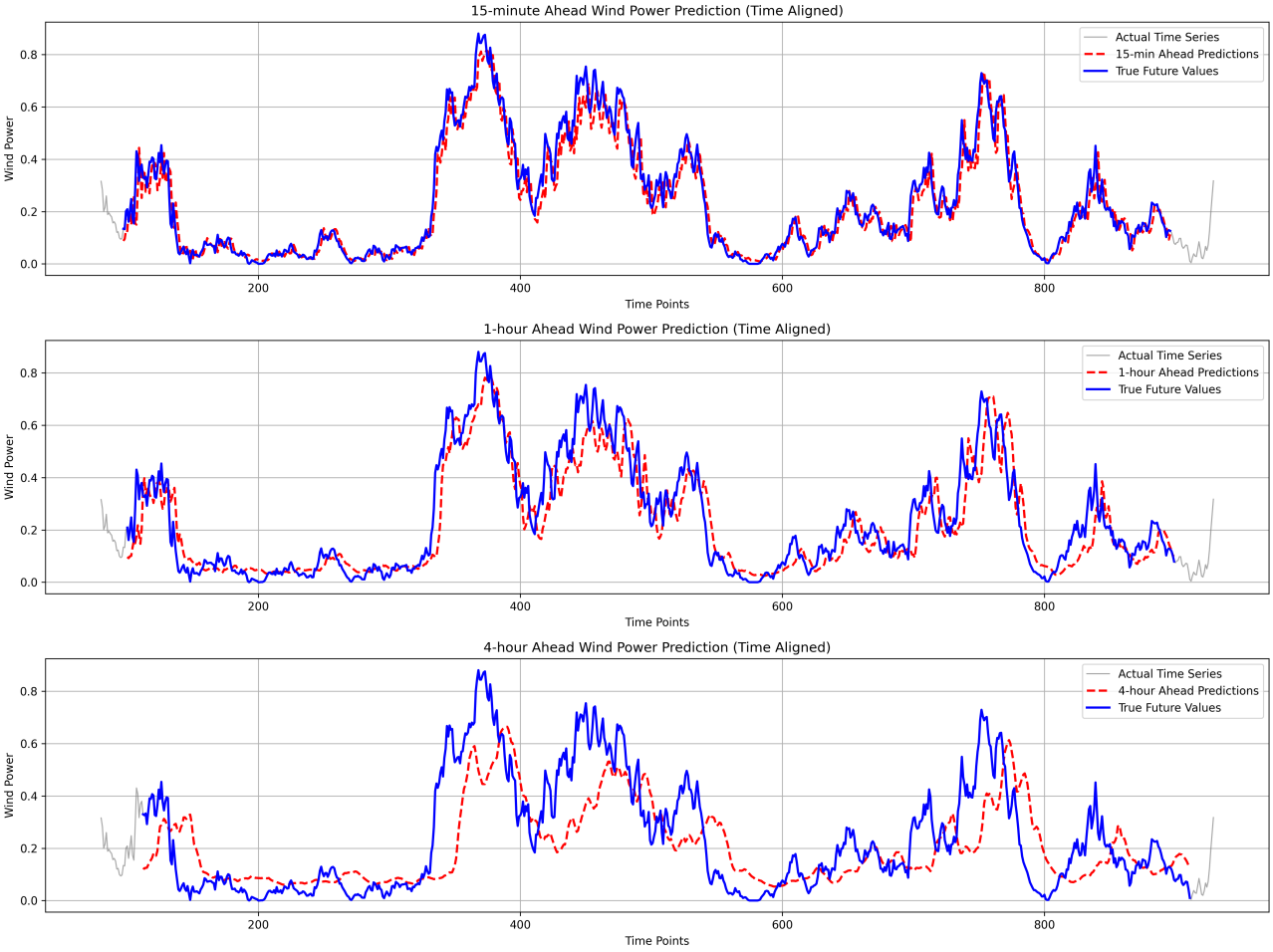

Figure 1: 多时间尺度预测结果对比

从图中可以看出：

- 15分钟预测结果最接近真实值，能够较好地捕捉功率波动特征
- 1小时预测结果整体趋势准确，但在急剧变化点处存在一定偏差
- 4小时预测结果能捕捉主要趋势，但精细波动的预测能力有限

表2总结了三个时间尺度的详细性能指标：

Table 2: 多时间尺度预测性能对比

预测时长	RMSE	相关系数	CR准确度(%)
15分钟	**0.0701**	**0.9640**	**78.38**
1小时	0.1114	0.9046	65.48
4小时	0.1873	0.7071	48.27

结果分析：

1. **准确度随时间递减**：预测时长增加，CR准确度明显下降，从15分钟的78.38%降至4小时的48.27%，降幅达30.11个百分点

2. **误差随时间增大**：RMSE随时间尺度增长，15分钟为0.0701，1小时增至0.1114，4小时达0.1873
3. **相关性逐步减弱**：相关系数从15分钟的0.9640降至4小时的0.7071，长期预测的线性相关性明显减弱

这符合时间序列预测的一般规律：预测时长越短，精度越高。15分钟预测在所有指标上均表现最优。

下图直观展示了三种预测时长的性能指标对比：

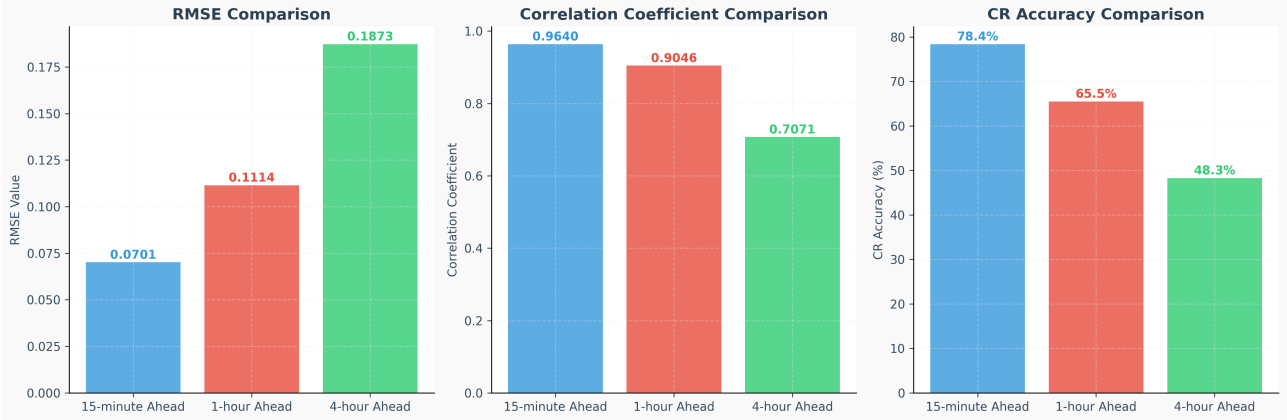

Figure 2: 评估指标对比

6.2 Adam优化器与SGD对比

以15分钟预测模型为例，在相同实验环境下对比了Adam优化器和标准SGD的性能：

Table 3: Adam vs SGD优化器对比

优化器类型	训练轮次	停止原因	训练时间(秒)	RMSE	相关系数	CR准确度(%)
Adam	456	早停触发	5.57	0.0701	0.9640	78.38
SGD	10,000	达到最大轮次	165.33	0.0773	0.9531	73.86

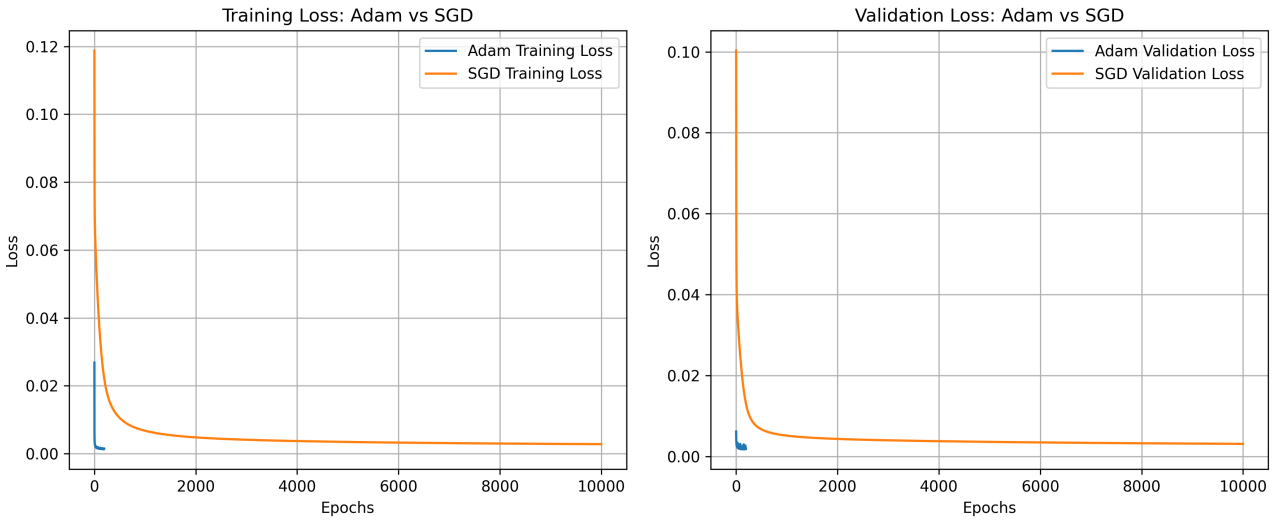

Figure 3: Adam与SGD训练过程对比

对比分析显示：

1. **收敛效率**：Adam优化器在456轮即达到最优性能并触发早停，而SGD即使训练10,000轮仍未能触发早停
2. **训练效率**：Adam的训练时间(5.57秒)相比SGD(165.33秒)减少96.6%
3. **预测精度**：Adam的RMSE(0.0701)比SGD(0.0773)降低9.3%，CR准确度提升4.52个百分点

SGD模型在达到最大训练轮次时仍未触发早停条件，表明其在训练后期收敛极其缓慢，模型性能改善微乎其微。

6.3 训练过程分析

图4展示了三个模型的训练损失曲线：

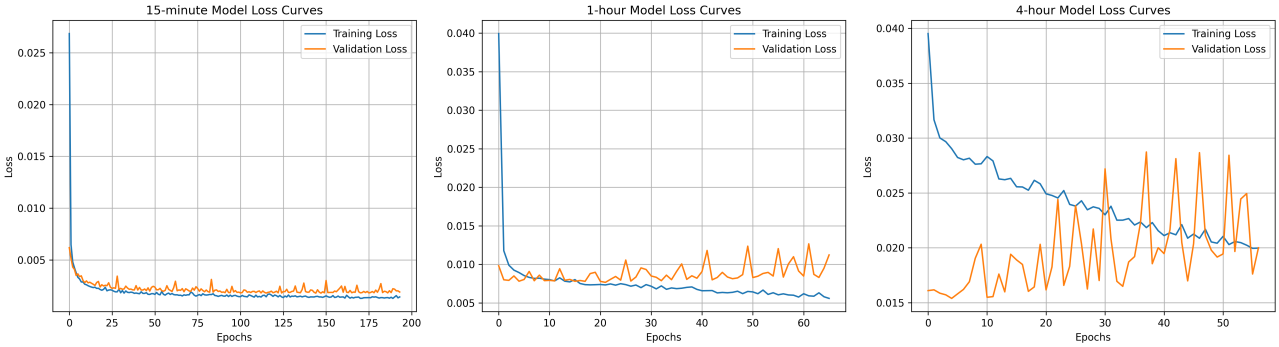

Figure 4: 三个模型的训练损失曲线

从训练曲线可以观察到：

- 15分钟模型收敛效果较好，损失曲线较为平滑。而1小时和4小时模型训练曲线较为波动，原则上可调小学习率来平滑收敛过程，但是实验发现降低学习率会进一步导致预

测准确率的下降（在训练集轻松达到过拟合）。个人觉得其波动大的主要原因还是训练数据质量不佳，波动过大，而且简单的bp网络对于4小时跨度的建模能力不够，模型表达能力欠佳

7 结论与未来工作

本研究构建了基于BP神经网络的多时间尺度风电功率预测模型，通过Adam优化器、早停机制和Xavier初始化等技术改进，实现了15分钟、1小时和4小时三种预测时长。实验结果显示：15分钟CR准确度78.38%(RMSE=0.0701)、1小时65.48%(RMSE=0.1114)、4小时48.27%(RMSE=0.1873)，准确度随时间递减符合预测规律。Adam优化器相比SGD训练效率提升96.6%，预测精度提升9.3%，在456轮触发早停而SGD即使10000轮也未收敛。未来研究可从超参数自动优化(网格搜索、optuna)、正则化技术(Dropout、BatchNorm、L1/L2)、不确定性量化(添加置信度、预测区间而非确定值)等方向进一步提升预测精度和实用性。

8 课程建议

建议可以结合一些最新的AI论文来讲解一些最新的模型架构。然后个人觉得个人作业可以放到期中后，这样期末压力更小，然后也可呀更快熟悉题目，也更好想出一些idea来放到大作业中

References

[1] Rumelhart, D., Hinton, G. & Williams, R. Learning representations by back-propagating errors. Nature 323, 533–536 (1986).

[2] Kingma, Diederik & Ba, Jimmy. (2014). Adam: A Method for Stochastic Optimization. International Conference on Learning Representations.

[3] Glorot, X., & Bengio, Y. (2010). Understanding the difficulty of training deep feedforward neural networks. In Proceedings of the thirteenth international conference on artificial intelligence and statistics (pp. 249-256).

[4] Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. (2014). Dropout: a simple way to prevent neural networks from overfitting. The journal of machine learning research, 15(1), 1929-1958.

[5] Prechelt, L. (1998). Early stopping-but when?. In Neural Networks: Tricks of the trade (pp. 55-69). Springer.