Model Architecture Analysis for BatteryML

Analysis Report

September 15, 2025

1 Model Overview

BatteryML implements a comprehensive suite of models for RUL (Remaining Useful Life) prediction, spanning both traditional machine learning and deep learning approaches. The models are organized into two main categories:

1. **Scikit-learn Models**: Traditional ML algorithms with statistical features
2. **Neural Network Models**: Deep learning architectures with matrix features

2 Base Model Architecture

2.1 BaseModel Interface

All models inherit from `BaseModel` which provides a scikit-learn-like interface:

Listing 1: BaseModel Interface

```
class BaseModel(abc.ABC):
    def fit(self, dataset: DataBundle, timestamp: str = None)
    def predict(self, dataset: DataBundle, data_type: str='test') -> torch.Tensor
    def dump_checkpoint(self, path: str)
    def load_checkpoint(self, path: str)
```

2.2 SklearnModel Base Class

Traditional ML models inherit from `SklearnModel`:

Listing 2: SklearnModel Implementation

```
class SklearnModel(BaseModel):
    def fit(self, dataset: DataBundle, timestamp: str = None):
        # Flatten features: (N, H, W) -> (N, H*W)
        feature = dataset.train_data.feature.view(len(feature), -1)
        self.model.fit(feature, dataset.train_data.label.to('cpu'))

    def predict(self, dataset: DataBundle, data_type: str='test') -> torch.Tensor:
        feature = dataset.test_data.feature.view(len(feature), -1)
        scores = self.model.predict(feature.numpy())
        return torch.from_numpy(scores).view(-1)
```

2.3 NNModel Base Class

Deep learning models inherit from `NNModel`:

Listing 3: NNModel Implementation

```
class NNModel(BaseModel, nn.Module):
    def __init__(self, batch_size=32, epochs=10000, lr=1e-3,
                 evaluate_freq=500, checkpoint_freq=1000):
        # Training configuration
        self.train_epochs = epochs
        self.lr = lr
        self.evaluate_freq = evaluate_freq
        self.checkpoint_freq = checkpoint_freq

    def fit(self, dataset: DataBundle, timestamp: str = None, seed: int = 0):
        # Adam optimizer with MSE loss
        optimizer = optim.Adam(self.parameters(), lr=self.lr)
        # Training loop with checkpointing and evaluation
```

3 Scikit-learn Models

3.1 Linear Regression

Architecture: Simple linear regression with no regularization.

Listing 4: Linear Regression

```
@MODELS.register()
class LinearRegressionRULPredictor(SklearnModel):
    def __init__(self, *args, workspace: str = None, **kwargs):
        self.model = LinearRegression(*args, **kwargs)
```

Hyperparameters: Uses default scikit-learn parameters. **Output:** Continuous RUL prediction (number of cycles).

3.2 Ridge Regression

Architecture: Linear regression with L2 regularization.

Listing 5: Ridge Regression

```
@MODELS.register()
class RidgeRULPredictor(SklearnModel):
    def __init__(self, *args, workspace: str = None, **kwargs):
        self.model = Ridge(**kwargs)
```

Hyperparameters:

- `alpha`: Regularization strength (default: 1.0)
- `fit_intercept`: Whether to fit intercept (default: True)
- `normalize`: Whether to normalize features (default: False)

Output: Continuous RUL prediction with regularization to prevent overfitting.

3.3 Elastic Net

Architecture: Linear regression with both L1 and L2 regularization.

Listing 6: Elastic Net

```
@MODELS.register()
class ElasticNetRULPredictor(SklearnModel):
    def __init__(self, *args, workspace: str = None, **kwargs):
        self.model = ElasticNetCV(**kwargs)
```

Hyperparameters:

- `l1_ratio`: Mixing parameter between L1 and L2 (0-1)
- `alpha`: Regularization strength
- `cv`: Cross-validation folds for parameter selection

Output: Continuous RUL prediction with automatic feature selection.

3.4 Random Forest

Architecture: Ensemble of decision trees with bootstrap aggregation.

Listing 7: Random Forest

```
@MODELS.register()
class RandomForestRULPredictor(SklearnModel):
    def __init__(self, *args, workspace: str = None, **kwargs):
        self.model = RandomForestRegressor(*args, **kwargs)
```

Hyperparameters:

- `n_estimators`: Number of trees (default: 100)
- `max_depth`: Maximum tree depth (default: None)
- `min_samples_split`: Minimum samples to split (default: 2)
- `min_samples_leaf`: Minimum samples per leaf (default: 1)
- `random_state`: Random seed for reproducibility

Output: Continuous RUL prediction with feature importance ranking.

3.5 XGBoost

Architecture: Gradient boosting with extreme gradient boosting optimization.

Listing 8: XGBoost

```
@MODELS.register()
class XGBoostRULPredictor(SklearnModel):
    def __init__(self, *args, workspace: str = None, **kwargs):
        self.model = XGBRegressor(*args, **kwargs)
```

Hyperparameters:

- `n_estimators`: Number of boosting rounds (default: 100)
- `max_depth`: Maximum tree depth (default: 6)
- `learning_rate`: Step size shrinkage (default: 0.1)
- `subsample`: Fraction of samples for each tree (default: 1.0)
- `colsample_bytree`: Fraction of features for each tree (default: 1.0)

Output: Continuous RUL prediction with high accuracy and feature importance.

3.6 Support Vector Regression (SVR)

Architecture: Support vector machine for regression with kernel trick.

Listing 9: Support Vector Regression

```
@MODELS.register()
class SVMRULPredictor(SklearnModel):
    def __init__(self, *args, workspace: str = None, **kwargs):
        self.model = SVR(*args, **kwargs)
```

Hyperparameters:

- `kernel`: Kernel type ('rbf', 'linear', 'poly', 'sigmoid')
- `C`: Regularization parameter (default: 1.0)
- `epsilon`: Epsilon-tube for SVR (default: 0.1)
- `gamma`: Kernel coefficient for 'rbf', 'poly', 'sigmoid'

Output: Continuous RUL prediction with non-linear decision boundaries.

3.7 Gaussian Process Regression

Architecture: Bayesian non-parametric regression with uncertainty quantification.

Listing 10: Gaussian Process Regression

```
@MODELS.register()
class GaussianProcessRULPredictor(SklearnModel):
    def __init__(self, *args, workspace: str = None, **kwargs):
        kernel = DotProduct() + RBF()
        self.model = GaussianProcessRegressor(kernel)
```

Hyperparameters:

- `kernel`: Covariance function (DotProduct + RBF)
- `alpha`: Added to diagonal of kernel matrix (default: 1e-10)
- `optimizer`: Kernel hyperparameter optimizer
- `n_restarts_optimizer`: Number of optimizer restarts (default: 0)

Output: Continuous RUL prediction with uncertainty estimates.

3.8 Principal Component Regression (PCR)

Architecture: Linear regression on principal components of features.

Listing 11: Principal Component Regression

```
@MODELS.register()
class PCRRULPredictor(SklearnModel):
    def __init__(self, *args, workspace: str = None, **kwargs):
        self.model = make_pipeline(PCA(*args, **kwargs), LinearRegression())
```

Hyperparameters:

- `n_components`: Number of principal components
- `whiten`: Whether to whiten components (default: False)
- `svd_solver`: SVD solver ('auto', 'full', 'arpack', 'randomized')

Output: Continuous RUL prediction with dimensionality reduction.

3.9 Partial Least Squares Regression (PLSR)

Architecture: Linear regression using partial least squares.

Listing 12: Partial Least Squares Regression

```
@MODELS.register()
class PLSRRULPredictor(SklearnModel):
    def __init__(self, *args, workspace: str = None, **kwargs):
        self.model = PLSRegression(*args, **kwargs)
```

Hyperparameters:

- `n_components`: Number of components to keep (default: 2)
- `scale`: Whether to scale features (default: True)
- `max_iter`: Maximum iterations (default: 500)

Output: Continuous RUL prediction with latent variable modeling.

3.10 Dummy Regressor

Architecture: Baseline model that predicts mean of training labels.

Listing 13: Dummy Regressor

```
@MODELS.register()
class DummyRULPredictor(SklearnModel):
    def __init__(self, *args, workspace: str = None, **kwargs):
        self.model = DummyRegressor(**kwargs)
```

Hyperparameters:

- `strategy`: Prediction strategy ('mean', 'median', 'quantile')
- `constant`: Constant value for 'constant' strategy
- `quantile`: Quantile for 'quantile' strategy

Output: Constant RUL prediction (baseline for comparison).

4 Neural Network Models

4.1 Multi-Layer Perceptron (MLP)

Architecture: Feedforward neural network with multiple hidden layers.

Listing 14: MLP Architecture

```
class MultiLayerPerceptronModule(nn.Module):
    def __init__(self, in_channels, channels, activation='tanh', dropout=0.05):
        self.layer = nn.Linear(in_channels, channels)
        self.activation = getattr(torch, activation.lower())
        self.dropout = nn.Dropout(dropout)

    def forward(self, x):
        return self.dropout(self.activation(self.layer(x)))

class MLPRULPredictor(NNModel):
    def __init__(self, in_channels, channels, input_height, input_width,
```

```
                dropout=0.05, **kwargs):
        self.proj1 = nn.Sequential(
            MultiLayerPerceptronModule(input_width, channels, dropout=dropout),
            MultiLayerPerceptronModule(channels, 1, dropout=dropout))
        self.proj2 = MultiLayerPerceptronModule(in_channels, 1)
        self.proj3 = nn.Sequential(
            MultiLayerPerceptronModule(input_height, channels),
            nn.Linear(channels, 1))
```

Architecture Details:

- **Input Processing**: 3D tensor (batch, channels, height, width)
- **Path 1**: Process width dimension with 2-layer MLP
- **Path 2**: Process channel dimension with 1-layer MLP
- **Path 3**: Process height dimension with 2-layer MLP
- **Output**: Final linear layer produces RUL prediction

Hyperparameters:

- `in_channels`: Input channel dimension (default: 1)
- `channels`: Hidden layer size (default: 32)
- `input_height`: Height dimension (default: 1)
- `input_width`: Width dimension (default: 1000)
- `dropout`: Dropout rate (default: 0.05)
- `epochs`: Training epochs (default: 5000)
- `batch_size`: Batch size (default: 128)
- `lr`: Learning rate (default: 1e-3)

Output: Continuous RUL prediction with non-linear feature learning.

4.2 Convolutional Neural Network (CNN)

Architecture: 2D convolutional neural network for matrix feature processing.

Listing 15: CNN Architecture

```
class ConvModule(nn.Module):
    def __init__(self, din, dout, kernel_size, act_fn='relu', dropout=0.1):
        self.conv1 = nn.Conv2d(din, dout, kernel_size)
        self.pool1 = nn.AvgPool2d(kernel_size)
        self.conv2 = nn.Conv2d(dout, dout, kernel_size)
        self.pool2 = nn.AvgPool2d(kernel_size)
        self.act_fn = getattr(torch, act_fn)
        self.dropout = nn.Dropout2d(dropout)

class CNNRULPredictor(NNModel):
    def __init__(self, in_channels, channels, input_height, input_width,
                 kernel_size=3, act_fn='relu', **kwargs):
        self.encoder = ConvModule(in_channels, channels, kernel_size, act_fn)
        H, W = self.encoder.output_shape(input_height, input_width)
        self.proj = nn.Conv2d(channels, channels, (H, W))
        self.fc = nn.Linear(channels, 1)
```

Architecture Details:

- **Conv Block 1**: Conv2D → ReLU → AvgPool2D
- **Conv Block 2**: Conv2D → ReLU → AvgPool2D
- **Global Pooling**: Conv2D with kernel size = feature map size
- **Output Layer**: Linear layer for RUL prediction

Hyperparameters:

- `in_channels`: Input channels (default: 1)
- `channels`: Number of filters (default: 16)
- `input_height`: Input height (default: 100)
- `input_width`: Input width (default: 1000)
- `kernel_size`: Convolution kernel size (default: 3)
- `act_fn`: Activation function (default: 'relu')
- `epochs`: Training epochs (default: 1000)
- `batch_size`: Batch size (default: 128)

Output: Continuous RUL prediction with spatial feature extraction.

4.3 Long Short-Term Memory (LSTM)

Architecture: Recurrent neural network for sequential data processing.

Listing 16: LSTM Architecture

```
class LSTMRULPredictor(NNModel):
    def __init__(self, in_channels, channels, input_height, input_width, **kwargs):
        self.lstm = nn.LSTM(
            in_channels * input_width, channels, 2, batch_first=True)
        self.fc = nn.Linear(channels, 1)

    def forward(self, feature, label, return_loss=False):
        B, _, H, _ = feature.size()
        x = feature.permute(0, 2, 1, 3).contiguous().view(B, H, -1)
        x, _ = self.lstm(x)
        x = x[:, -1].contiguous().view(B, -1)
        x = self.fc(x).view(-1)
        return x
```

Architecture Details:

- **Input Reshape**: (B, C, H, W) → (B, H, C*W)
- **LSTM Layer**: 2-layer LSTM with batch_first=True
- **Output Selection**: Use last time step output
- **Linear Layer**: Final prediction layer

Hyperparameters:

- `in_channels`: Input channels (default: 1)

- `channels`: LSTM hidden size (default: 32)
- `input_height`: Sequence length (default: 100)
- `input_width`: Feature dimension (default: 1000)
- `epochs`: Training epochs (default: 1000)
- `batch_size`: Batch size (default: 128)

Output: Continuous RUL prediction with temporal sequence modeling.

4.4 Transformer

Architecture: Attention-based neural network for sequence processing.

Listing 17: Transformer Architecture

```
class TransformerRULPredictor(NNModel):
    def __init__(self, in_channels, channels, input_height, input_width,
                 num_layers=2, nhead=4, **kwargs):
        self.proj = nn.Linear(in_channels * input_width, channels)
        decoder_layer = nn.TransformerDecoderLayer(
            d_model=channels, nhead=nhead, dim_feedforward=channels,
            batch_first=True)
        self.transformer = nn.TransformerDecoder(decoder_layer, num_layers)
        self.fc = nn.Linear(channels * input_height, 1)

    def forward(self, feature, label, return_loss=False):
        B, C, N, L = feature.shape
        feature = feature.transpose(1, 2).contiguous().view(B, N, -1)
        input_ = self.proj(feature)
        embedding = self.transformer(input_, input_).view(B, -1)
        pred = self.fc(embedding).view(-1)
        return pred
```

Architecture Details:

- **Input Projection**: Linear layer to project features to model dimension
- **Transformer Decoder**: Self-attention mechanism with multiple layers
- **Attention Heads**: Multi-head attention (default: 4 heads)
- **Feedforward**: Position-wise feedforward network
- **Output Layer**: Linear layer for RUL prediction

Hyperparameters:

- `in_channels`: Input channels (default: 1)
- `channels`: Model dimension (default: 16)
- `input_height`: Sequence length (default: 100)
- `input_width`: Feature dimension (default: 1000)
- `num_layers`: Number of transformer layers (default: 2)
- `nhead`: Number of attention heads (default: 4)
- `epochs`: Training epochs (default: 1000)
- `batch_size`: Batch size (default: 128)

Output: Continuous RUL prediction with attention-based sequence modeling.

5 Model Configuration Summary

5.1 Feature Extractor Mapping

Table 1: Model to Feature Extractor Mapping

Model Type	**Feature Extractor**
Linear Regression, Ridge, Elastic Net	Statistical features
Random Forest, XGBoost, SVR, GPR	Matrix features
PCR, PLSR	Matrix features
CNN, LSTM, MLP, Transformer	Matrix features

5.2 Hyperparameter Ranges

Table 2: Key Hyperparameters by Model Type

Model Type	**Key Hyperparameters**
Linear Models	alpha (regularization)
Tree-based	n_estimators, max_depth, learning_rate
SVM	C, kernel, epsilon, gamma
Neural Networks	channels, epochs, batch_size, lr, dropout

5.3 Training Configuration

Table 3: Training Configuration by Model Type

Model Type	**Optimizer**	**Loss Function**	**Evaluation**
Scikit-learn	N/A (closed-form)	MSE (implicit)	Cross-validation
Neural Networks	Adam	MSE	RMSE every 100 epochs

6 Model Outputs

6.1 Output Format

All models output:

- **Type**: `torch.Tensor` with shape `(N,)` where N is number of test samples
- **Values**: Continuous RUL predictions (number of cycles)
- **Device**: Moved to appropriate device (CPU/GPU)
- **Preprocessing**: Inverse transformed from log-scale and z-score normalization

6.2 Evaluation Metrics

Models are evaluated using:

- **RMSE**: Root Mean Square Error
- **MAE**: Mean Absolute Error
- **MAPE**: Mean Absolute Percentage Error

7 Summary

BatteryML provides a comprehensive suite of models for RUL prediction, ranging from simple linear models to sophisticated deep learning architectures. The models are designed to work with both statistical features (for traditional ML) and matrix features (for deep learning), providing flexibility for different use cases and data characteristics.

Key strengths:

- **Comprehensive coverage**: From simple baselines to state-of-the-art deep learning
- **Unified interface**: Consistent API across all model types
- **Flexible features**: Support for both statistical and matrix features
- **Reproducible**: Deterministic training with proper seeding
- **Extensible**: Easy to add new models following the base class pattern