感謝のお言葉と、とても詳細で専門的なフィードバックをいただきました。論文「Multi-Transformer: A New Neural Network-Based Architecture for Forecasting S&P Volatility」に関する理解が深く、コード実装の細部まで分析いただいています。

フィードバックのポイントを整理しながら、最終的な改良版コードを考えていきましょう。

1. Positional Encodingの追加
   - Transformerモデルには必須の要素
   - 論文のFigure 3、Figure 5にも明示的に記載されている
   - 現在の実装には欠けている

2. MultiTransformerLayerのランダムサンプリング戦略
   - 現在の実装: 特徴量（列）のランダムサンプリング（ランダムサブスペース法に近い）
   - 論文の示唆: 観測データ（行/サンプル）のランダムサンプリング（バギングに近い）
   - データの90%をランダムサンプリングする点を再現する

3. MultiTransformerLayerのアーキテクチャ構造
   - 現在: T個の完全なTransformerブロック（MHA+FFN）を独立して実行し、最終出力を平均
   - 論文: T個のMHAの出力を平均化した後、単一のFFNに渡す構造（Figure 5）
   - どちらのアプローチがボラティリティ予測/日中リターン予測に有効か検証

4. GARCHモデルとの統合
   - 現在: GARCHモデルとMultiTransformerが独立して動作
   - 論文: GARCHの出力がニューラルネットワークへの特徴量として供給される
   - ハイブリッドモデルとしての統合方法を検討

これらの点を考慮して、論文により忠実でありながら、J-Quants APIデータへの適用に最適化された改良版コードを提案します。

まず、Positional Encodingレイヤーを実装します:

```python
class PositionalEncodingLayer(tf.keras.layers.Layer):
    def __init__(self, sequence_length, d_model):
        super(PositionalEncodingLayer, self).__init__()
        self.sequence_length = sequence_length
        self.d_model = d_model

    def build(self, input_shape):
        # 正弦波と余弦波を使用したPositional Encoding
        position = np.arange(self.sequence_length)[:, np.newaxis]
        div_term = np.exp(np.arange(0, self.d_model, 2) * -(np.log(10000.0) / self.d_model))

        pe = np.zeros((self.sequence_length, self.d_model))
        pe[:, 0::2] = np.sin(position * div_term)
        pe[:, 1::2] = np.cos(position * div_term)

        # 0次元目(バッチ)を拡張するためのブロードキャスト用次元追加
```

```
        self.pe = tf.convert_to_tensor(pe[np.newaxis, ...], dtype=tf.float32)

    def call(self, inputs):
        # 入力にPositional Encodingを加算
        return inputs + self.pe
```

次に、論文の図5に基づいた改良版MultiTransformerレイヤーを実装します（MHA出力の平均化の後に共通FFNを使用）:

```python
class MultiTransformerLayerV2(tf.keras.layers.Layer):
    def __init__(self, d_model, num_heads, ff_dim, dropout_rate=0.1, num_transformers=3):
        super(MultiTransformerLayerV2, self).__init__()
        self.num_transformers = num_transformers
        self.d_model = d_model

        # 複数のMulti-Head Attentionを作成
        self.attention_layers = [
            tf.keras.layers.MultiHeadAttention(
                num_heads=num_heads, key_dim=d_model//num_heads, dropout=dropout_rate)
            for _ in range(num_transformers)
        ]

        # 共通の層
        self.attention_dropout = tf.keras.layers.Dropout(dropout_rate)
        self.attention_norm = tf.keras.layers.LayerNormalization(epsilon=1e-6)

        # 共通のFFN
        self.ffn1 = tf.keras.layers.Dense(ff_dim, activation='relu')
        self.ffn2 = tf.keras.layers.Dense(d_model)
        self.ffn_dropout = tf.keras.layers.Dropout(dropout_rate)
        self.ffn_norm = tf.keras.layers.LayerNormalization(epsilon=1e-6)

    def call(self, inputs, training=False, mask=None):
        batch_size = tf.shape(inputs)[0]
        seq_len = tf.shape(inputs)[1]

        # 各Multi-Head Attentionの出力を格納するリスト
        attention_outputs = []

        for i in range(self.num_transformers):
            # 論文セクション2.5に基づくバギングアプローチ
            # 観測データ（行/サンプル）の90%をランダムに選択
            if training:
                # 各バッチ内の各シーケンスに対して、90%のタイムステップをランダムに選択
                sample_rate = 0.9  # 論文に基づく90%
                random_indices = tf.random.shuffle(tf.range(seq_len))
```

```
            num_samples = tf.cast(tf.math.ceil(sample_rate * tf.cast(seq_len, tf.float32)),
tf.int32)
            selected_indices = random_indices[:num_samples]

            # 選択したタイムステップのみのデータを抽出
            input_sample = tf.gather(inputs, selected_indices, axis=1)

            # 選択していないタイムステップのマスク（オプション）
            # ここではSimple Attentionなのでマスクは省略

            # Multi-Head Attention
            attn_output = self.attention_layers[i](input_sample, input_sample)

            # 元のシーケンス長に復元（0パディング）
            # これは平均化のために形状を揃える必要がある
            padded_attn = tf.zeros_like(inputs)
            # 選択したインデックスに出力を配置
            for b in range(batch_size):
                padded_attn = tf.tensor_scatter_nd_update(
                    padded_attn,
                    tf.stack([
                        tf.ones(num_samples, dtype=tf.int32) * b,
                        selected_indices
                    ], axis=1),
                    attn_output[b]
                )

            attention_outputs.append(padded_attn)
        else:
            # 推論時は全データを使用
            attn_output = self.attention_layers[i](inputs, inputs)
            attention_outputs.append(attn_output)

    # T個のアテンション出力の平均（Equation 20: AMH）
    avg_attention = tf.reduce_mean(attention_outputs, axis=0)

    # 残りは通常のTransformerブロックの処理
    avg_attention = self.attention_dropout(avg_attention, training=training)
    out1 = self.attention_norm(inputs + avg_attention)  # 残差接続 + 正規化

    # 共通のFFN
    ffn_output = self.ffn1(out1)
    ffn_output = self.ffn2(ffn_output)
    ffn_output = self.ffn_dropout(ffn_output, training=training)
    out2 = self.ffn_norm(out1 + ffn_output)  # 残差接続 + 正規化

    return out2
```

GARCHモデルとMultiTransformerの統合を改善します:

```python
def prepare_hybrid_data(X_train_seq, X_test_seq, garch_forecasts, df_train, df_test):
    """GARCHモデルの予測値をTransformerの入力特徴量として統合する"""
    # ここでは簡略化のため、各銘柄の全期間に同じGARCH予測値を使用
    # 実際のアプリケーションでは、各時点ごとに異なるGARCH予測値を使用すべき

    # 訓練データとテストデータの銘柄情報を抽出
    train_codes = df_train['Code'].unique()
    test_codes = df_test['Code'].unique()

    # GARCH予測値を特徴量として追加
    X_train_with_garch = []
    for code in train_codes:
        if code in garch_forecasts:
            # この銘柄のGARCH予測値を取得
            garch_forecast = garch_forecasts[code]

            # この銘柄のシーケンスデータに対応するインデックスを見つける
            # (実際の実装では、銘柄と日付から適切なインデックスマッピングが必要)
            # ここでは、各銘柄ごとにシーケンスが連続して格納されていると仮定

            # GARCHの予測値を新たな特徴量として追加
            # (この部分は実際のデータ構造に合わせて調整が必要)
            # ここでは単純化のため、全シーケンスに同じGARCH予測値を追加

            # TODO: 実際の実装では、各シーケンスの対応する銘柄を特定し、
            # その銘柄のGARCH予測値を追加する必要がある

    # 同様にテストデータにもGARCH予測値を追加
    X_test_with_garch = []

    return X_train_with_garch, X_test_with_garch
```

最後に、これらの改良を取り入れた最終的なモデル構築関数:

```python
def build_improved_mt_garch_model(seq_length, num_features, prediction_type='volatility',
use_garch=True):
    """論文に忠実なMultiTransformer-GARCHモデルの構築(改良版)"""
    # 入力層
    inputs = tf.keras.layers.Input(shape=(seq_length, num_features))

    # Positional Encoding(論文Figure 3, 5準拠)
    x = PositionalEncodingLayer(seq_length, num_features)(inputs)
```

```
# MultiTransformerレイヤー（論文Figure 5準拠）
x = MultiTransformerLayerV2(
    d_model=num_features,
    num_heads=4,
    ff_dim=num_features*4,
    dropout_rate=0.1,
    num_transformers=3
)(x)

# GARCHモデルからの入力（オプション）
if use_garch:
    # GARCHモデルの予測値を入力として追加
    garch_inputs = tf.keras.layers.Input(shape=(1,))

    # Transformerの出力を平坦化
    x = tf.keras.layers.GlobalAveragePooling1D()(x)

    # Transformerの出力とGARCH予測を結合
    x = tf.keras.layers.Concatenate()([x, garch_inputs])

    # 残りは同じ
    x = tf.keras.layers.Dense(64, activation='relu')(x)
    x = tf.keras.layers.Dropout(0.2)(x)
    x = tf.keras.layers.Dense(32, activation='relu')(x)

    # 出力層（予測タイプに応じて調整）
    if prediction_type == 'volatility':
        # ボラティリティ予測の場合、非負の値を出力
        outputs = tf.keras.layers.Dense(1, activation='softplus')(x)
    else:  # 'intraday_return'
        # 収益率予測の場合、任意の値を出力
        outputs = tf.keras.layers.Dense(1)(x)

    model = tf.keras.models.Model(inputs=[inputs, garch_inputs], outputs=outputs)
else:
    # GARCHを使用しない場合はシンプルなモデル
    x = tf.keras.layers.GlobalAveragePooling1D()(x)
    x = tf.keras.layers.Dense(64, activation='relu')(x)
    x = tf.keras.layers.Dropout(0.2)(x)
    x = tf.keras.layers.Dense(32, activation='relu')(x)

    if prediction_type == 'volatility':
        outputs = tf.keras.layers.Dense(1, activation='softplus')(x)
    else:
        outputs = tf.keras.layers.Dense(1)(x)

    model = tf.keras.models.Model(inputs=inputs, outputs=outputs)
```

```
    model.compile(
        optimizer=tf.keras.optimizers.Adam(learning_rate=0.001),
        loss='mse',
        metrics=['mae']
    )

    return model
```

完全なパイプライン実装も修正します:

```python
def train_and_evaluate_improved_model(prediction_type='intraday_return',
use_garch=True):
    """改良されたMultiTransformerモデルの学習と評価"""
    # データ準備
    X_train_seq, y_train, X_test_seq, y_test, df_train, df_test, garch_forecasts =
prepare_mt_data_pipeline(prediction_type)

    # GARCHモデルを使用する場合、データを統合
    if use_garch:
        X_train_with_garch, X_test_with_garch = prepare_hybrid_data(
            X_train_seq, X_test_seq, garch_forecasts, df_train, df_test
        )

        # モデル構築
        model = build_improved_mt_garch_model(
            seq_length=X_train_seq.shape[1],
            num_features=X_train_seq.shape[2],
            prediction_type=prediction_type,
            use_garch=True
        )

        # モデル学習
        history = model.fit(
            [X_train_seq, X_train_with_garch], y_train,
            epochs=100,
            batch_size=32,
            validation_split=0.2,
            callbacks=[tf.keras.callbacks.EarlyStopping(
                monitor='val_loss', patience=10, restore_best_weights=True
            )]
        )

        # テストデータでの予測
        y_pred = model.predict([X_test_seq, X_test_with_garch])
    else:
```

```
    # GARCHを使用しない場合
    model = build_improved_mt_garch_model(
        seq_length=X_train_seq.shape[1],
        num_features=X_train_seq.shape[2],
        prediction_type=prediction_type,
        use_garch=False
    )

    history = model.fit(
        X_train_seq, y_train,
        epochs=100,
        batch_size=32,
        validation_split=0.2,
        callbacks=[tf.keras.callbacks.EarlyStopping(
            monitor='val_loss', patience=10, restore_best_weights=True
        )]
    )

    y_pred = model.predict(X_test_seq)

# 以下は評価部分は前回と同様
# ...
```

しかしながら、実際の`prepare_hybrid_data`関数の実装は、銘柄・日付ごとにGARCH予測値を紐付ける部分など、データ構造に強く依存するため、実装例の中では部分的に空欄や仮の実装としています。実際に使用する場合は、データセットの構造に合わせてこの部分を適切に実装する必要があります。

これらの改良点をまとめると:

1. Positional Encodingの追加
2. 論文Figure 5に忠実なMultiTransformerレイヤーの実装（MHAの出力を平均化した後に共通FFN）
3. 観測データ（行/サンプル）のランダムサンプリング（論文セクション2.5準拠）
4. GARCHモデルとの統合の改善（特徴量としての入力）

最後に、これらの改良と違いについてわかりやすく説明し、ユーザーが自分のニーズに合わせて選択できるようにします。また、J-Quants APIのデータ構造に合わせた具体的な使用法についても触れたいと思います。

論文に忠実なMultiTransformerモデルの最終改良版

いただいたフィードバックを元に、論文「Multi-Transformer: A New Neural Network-Based Architecture for Forecasting S&P Volatility」により忠実なモデル実装を提案します。特に、Positional Encoding、ランダムサンプリング戦略、アーキテクチャ構造、GARCHモデルとの統合について改良しました。

1. Positional Encodingの実装

```python
class PositionalEncodingLayer(tf.keras.layers.Layer):
    """Transformerに必須のPositional Encoding（論文Figure 3, 5に準拠）"""
    def __init__(self, sequence_length, d_model):
        super(PositionalEncodingLayer, self).__init__()
        self.sequence_length = sequence_length
        self.d_model = d_model

    def build(self, input_shape):
        # 正弦波と余弦波を使用したPositional Encoding
        position = np.arange(self.sequence_length)[:, np.newaxis]
        div_term = np.exp(np.arange(0, self.d_model, 2) * -(np.log(10000.0) / self.d_model))

        pe = np.zeros((self.sequence_length, self.d_model))
        pe[:, 0::2] = np.sin(position * div_term)
        pe[:, 1::2] = np.cos(position * div_term)

        self.pe = tf.convert_to_tensor(pe[np.newaxis, ...], dtype=tf.float32)

    def call(self, inputs):
        # 入力にPositional Encodingを加算
        return inputs + self.pe
```

2. 論文準拠のMultiTransformerレイヤー実装

```python
class MultiTransformerLayer(tf.keras.layers.Layer):
    """論文Figure 5に忠実なMultiTransformerレイヤー"""
    def __init__(self, d_model, num_heads, ff_dim, dropout_rate=0.1, num_transformers=3):
        super(MultiTransformerLayer, self).__init__()
        self.num_transformers = num_transformers
        self.d_model = d_model

        # 複数のMulti-Head Attentionを作成
        self.attention_layers = [
            tf.keras.layers.MultiHeadAttention(
                num_heads=num_heads, key_dim=d_model//num_heads, dropout=dropout_rate)
            for _ in range(num_transformers)
        ]

        # 共通の層（論文Figure 5に準拠）
        self.attention_dropout = tf.keras.layers.Dropout(dropout_rate)
        self.attention_norm = tf.keras.layers.LayerNormalization(epsilon=1e-6)

        # 共通FFN部分（Attentionの出力平均後に適用）
```

```
        self.ffn1 = tf.keras.layers.Dense(ff_dim, activation='relu')
        self.ffn2 = tf.keras.layers.Dense(d_model)
        self.ffn_dropout = tf.keras.layers.Dropout(dropout_rate)
        self.ffn_norm = tf.keras.layers.LayerNormalization(epsilon=1e-6)

    def call(self, inputs, training=False, mask=None):
        batch_size = tf.shape(inputs)[0]
        seq_len = tf.shape(inputs)[1]

        # 各Multi-Head Attentionの出力を格納するリスト
        attention_outputs = []

        for i in range(self.num_transformers):
            # 論文セクション2.5に基づくバギングアプローチ
            # 観測データ（行/サンプル）の90%をランダムに選択
            if training:
                # 各時系列の90%のタイムステップをランダムに選択（論文準拠）
                sample_rate = 0.9
                random_indices = tf.random.shuffle(tf.range(seq_len))
                num_samples = tf.cast(tf.math.ceil(sample_rate * tf.cast(seq_len, tf.float32)),
tf.int32)
                selected_indices = random_indices[:num_samples]

                # 選択したタイムステップのみのデータを抽出
                input_sample = tf.gather(inputs, selected_indices, axis=1)

                # Multi-Head Attention
                attn_output = self.attention_layers[i](input_sample, input_sample)

                # 元のシーケンス長に戻す（padded_attnの作成）
                padded_attn = tf.zeros_like(inputs)

                for b in range(batch_size):
                    # 各バッチに対して選択したインデックスに出力を配置
                    idx_tensor = tf.stack([
                        tf.ones(num_samples, dtype=tf.int32) * b,
                        selected_indices
                    ], axis=1)
                    padded_attn = tf.tensor_scatter_nd_update(
                        padded_attn, idx_tensor, attn_output[b]
                    )

                attention_outputs.append(padded_attn)
            else:
                # 推論時は全データを使用
                attn_output = self.attention_layers[i](inputs, inputs)
                attention_outputs.append(attn_output)
```

```
        # T個のアテンション出力の平均（論文Equation 20: AMHIに準拠）
        avg_attention = tf.reduce_mean(attention_outputs, axis=0)

        # 残りは通常のTransformerブロックの処理
        avg_attention = self.attention_dropout(avg_attention, training=training)
        out1 = self.attention_norm(inputs + avg_attention)  # 残差接続 + 正規化

        # 共通のFFN（平均化されたアテンション出力に対して適用）
        ffn_output = self.ffn1(out1)
        ffn_output = self.ffn2(ffn_output)
        ffn_output = self.ffn_dropout(ffn_output, training=training)
        out2 = self.ffn_norm(out1 + ffn_output)  # 残差接続 + 正規化

        return out2
```

3. 特徴量と目的変数の準備（論文準拠の設計）

```python
def calc_features_and_targets(df_ohlcv, prediction_type='volatility'):
    """論文セクション2.1に準拠した特徴量と目的変数の計算"""
    df_feats = df_ohlcv[['Date','Code','Open','Close','Volume']].copy()

    # 対数収益率（論文式(1)）
    df_feats['log_return'] = np.log(df_ohlcv['Close'] / df_ohlcv['Close'].shift(1))

    # 特徴量：過去の対数収益率とそのラグ（論文式(2)）
    for i in range(1, 11):  # ラグ1から10まで
        df_feats[f'log_return_lag{i}'] = df_feats['log_return'].shift(i)

    # 特徴量：過去5日間の対数収益率の標準偏差とそのラグ（論文式(3)）
    df_feats['volatility_5d'] = df_feats['log_return'].rolling(5).std()
    for i in range(1, 11):  # ラグ1から10まで
        df_feats[f'volatility_5d_lag{i}'] = df_feats['volatility_5d'].shift(i)

    # 目的変数の計算
    if prediction_type == 'volatility':
        # 将来5日間の実現ボラティリティ（論文式(4)）
        df_feats['target_vol'] = df_feats['log_return'].rolling(5).std().shift(-5)
    else:  # 'intraday_return'
        # 翌日の日中収益率（寄り引け戦略用）
        df_feats['target_return'] = (df_ohlcv['Close'] / df_ohlcv['Open'] - 1.0).shift(-1)

    return df_feats
```

4. GARCHモデルとの統合（ハイブリッドモデル）

```python
def fit_garch_models_by_code(df_train):
    """各銘柄ごとのGARCH(1,1)モデルを作成し予測値を返す"""
    from arch import arch_model

    garch_results = {}

    for code in tqdm(df_train['Code'].unique()):
        # 銘柄ごとのデータ抽出
        train_data = df_train[df_train['Code'] == code]['log_return'].dropna()

        if len(train_data) > 30:  # 十分なデータがある場合のみモデル作成
            try:
                # GARCH(1,1)モデルの作成と学習
                garch_model = arch_model(train_data, vol='Garch', p=1, q=1)
                garch_result = garch_model.fit(disp='off')

                # データと同じインデックスでボラティリティ予測値を返す
                forecast = garch_result.conditional_volatility

                # 銘柄コードとインデックスのマッピングを保存
                garch_results[code] = pd.Series(
                    forecast,
                    index=train_data.index
                )
            except:
                # エラーが発生した場合はスキップ
                print(f"Error fitting GARCH model for code {code}")

    return garch_results

def prepare_garch_features(df, garch_results):
    """GARCHモデルの予測値を特徴量として追加"""
    df_with_garch = df.copy()
    df_with_garch['garch_vol'] = np.nan

    for code, garch_series in garch_results.items():
        # この銘柄のインデックスを特定
        code_idx = df_with_garch['Code'] == code

        # インデックスが一致する行にGARCH予測値を代入
        for idx, value in garch_series.items():
            date_idx = df_with_garch['Date'] == idx
            df_with_garch.loc[code_idx & date_idx, 'garch_vol'] = value

    # 欠損値を前方補完
    df_with_garch['garch_vol'] =
df_with_garch.groupby('Code')['garch_vol'].fillna(method='ffill')
```

```
    # それでも残る欠損値を0で埋める
    df_with_garch['garch_vol'] = df_with_garch['garch_vol'].fillna(0)

    return df_with_garch
```

5. 論文準拠のハイブリッドモデル構築

```python
def build_mt_garch_model(seq_length, num_features, prediction_type='volatility',
include_garch=True):
    """論文Figure 5, 6に準拠したMultiTransformer-GARCHモデル"""
    # 通常の特徴量入力
    inputs = tf.keras.layers.Input(shape=(seq_length, num_features))

    # Positional Encoding（論文Figure 3, 5に必須）
    x = PositionalEncodingLayer(seq_length, num_features)(inputs)

    # MultiTransformerレイヤー（論文Figure 5に準拠）
    x = MultiTransformerLayer(
        d_model=num_features,
        num_heads=4,
        ff_dim=num_features*4,
        dropout_rate=0.1,
        num_transformers=3
    )(x, training=True)

    # シーケンス次元の集約
    x = tf.keras.layers.GlobalAveragePooling1D()(x)

    # GARCHモデルからの特徴量入力（論文Figure 6のハイブリッドモデル）
    if include_garch:
        garch_input = tf.keras.layers.Input(shape=(1,))

        # Transformerの出力とGARCH予測を結合
        x = tf.keras.layers.Concatenate()([x, garch_input])

        inputs_list = [inputs, garch_input]
    else:
        inputs_list = inputs

    # 最終的な予測層（論文Figure 5, 6準拠）
    x = tf.keras.layers.Dense(64, activation='relu')(x)
    x = tf.keras.layers.Dropout(0.2)(x)
    x = tf.keras.layers.Dense(32, activation='relu')(x)

# 出力層（予測タイプに応じて活性化関数を調整）
```

```
    if prediction_type == 'volatility':
        outputs = tf.keras.layers.Dense(1, activation='softplus')(x)  # 非負制約
    else:  # 'intraday_return'
        outputs = tf.keras.layers.Dense(1)(x)  # 制約なし

    model = tf.keras.models.Model(inputs=inputs_list, outputs=outputs)
    model.compile(
        optimizer=tf.keras.optimizers.Adam(learning_rate=0.001),
        loss='mse',
        metrics=['mae']
    )

    return model
```

6. J-Quants APIデータを使った完全なパイプライン

```python
def train_mt_model_with_jquants(prediction_type='volatility', include_garch=True):
    """J-Quants APIデータを使用したMultiTransformerモデルのトレーニング"""
    # J-Quants APIクライアント設定
    cli = jquantsapi.Client()

    # データ期間の設定
    HISTORICAL_DATA_YEARS = 5
    end_dt = datetime.now()
    start_dt = end_dt - timedelta(days=365*HISTORICAL_DATA_YEARS)

    print(f"データ取得期間: {start_dt.strftime('%Y-%m-%d')} から
{end_dt.strftime('%Y-%m-%d')}")

    # 上場銘柄情報の取得
    stock_list = cli.get_listed_info()

    # TOPIX500銘柄の抽出
    categories = ['TOPIX Mid400', 'TOPIX Large70', 'TOPIX Core30']
    tickers = stock_list[stock_list['ScaleCategory'].isin(categories)]['Code'].unique()
    tickers = tickers.astype(str)

    print(f"分析対象銘柄数: {len(tickers)}")

    # 株価データの取得
    stock_price = cli.get_price_range(start_dt=start_dt, end_dt=end_dt)

    # 銘柄コードの標準化
    stock_price["Code"] = stock_price["Code"].astype(str)
    stock_price.loc[(stock_price["Code"].str.len() == 5) &
                    (stock_price["Code"].str[-1] == "0"), "Code"] = \
```

```
        stock_price.loc[(stock_price["Code"].str.len() == 5) &
                        (stock_price["Code"].str[-1] == "0"), "Code"].str[:-1]

    # TOPIX500銘柄のみに絞る
    df_ohlcv = stock_price[stock_price['Code'].isin(tickers)]

    # 特徴量と目的変数の計算（論文準拠）
    print("特徴量と目的変数の計算中...")
    df_feats = calc_features_and_targets(df_ohlcv, prediction_type=prediction_type)

    # データの分割（訓練:テスト = 3:2）
    split_date = end_dt - timedelta(days=365*2)  # 最後の2年間をテストデータに
    df_train = df_feats[df_feats['Date'] < split_date]
    df_test = df_feats[df_feats['Date'] >= split_date]

    print(f"訓練データ期間: {df_train['Date'].min()} から {df_train['Date'].max()}")
    print(f"テストデータ期間: {df_test['Date'].min()} から {df_test['Date'].max()}")

    # 特徴量とターゲットの定義
    if prediction_type == 'volatility':
        feature_cols = [col for col in df_feats.columns if 'log_return_lag' in col or 'volatility_5d' in
col]
        target_col = 'target_vol'
    else:  # 'intraday_return'
        feature_cols = [col for col in df_feats.columns if 'log_return_lag' in col or 'volatility_5d' in
col]
        target_col = 'target_return'

    # GARCHモデルの適用（オプション）
    if include_garch:
        print("GARCHモデルの学習中...")
        garch_results = fit_garch_models_by_code(df_train)

        # GARCH予測値を特徴量として追加
        print("GARCH予測値を特徴量として追加中...")
        df_train = prepare_garch_features(df_train, garch_results)

        # テストデータ用のGARCH予測値も準備
        garch_test_results = {}
        for code in df_test['Code'].unique():
            if code in garch_results:
                # 訓練データから得たGARCHモデルを使用（単純化）
                # 実際の運用ではテスト期間に対する時系列予測が必要
                mean_vol = garch_results[code].mean()
                garch_test_results[code] = mean_vol

        df_test['garch_vol'] = df_test['Code'].map(garch_test_results).fillna(0)
```

```
    # シーケンスデータの準備
    print("シーケンスデータの準備中...")
    seq_length = 20  # 過去20日分のデータを使用

    # シーケンスの作成関数
    def prepare_sequences(df, features, target, seq_length):
        sequences = []
        targets = []
        garch_values = []

        for code, group in tqdm(df.groupby('Code')):
            group = group.sort_values('Date')

            feature_data = group[features].values
            target_data = group[target].values

            if include_garch:
                garch_data = group['garch_vol'].values

            for i in range(len(group) - seq_length):
                if not np.isnan(target_data[i+seq_length]):
                    seq = feature_data[i:i+seq_length]
                    tar = target_data[i+seq_length]

                    if not np.isnan(seq).any():
                        sequences.append(seq)
                        targets.append(tar)

                        if include_garch:
                            garch_values.append(garch_data[i+seq_length])

        if include_garch:
            return np.array(sequences), np.array(targets), np.array(garch_values).reshape(-1, 1)
        else:
            return np.array(sequences), np.array(targets)

    # 訓練データとテストデータのシーケンス作成
    if include_garch:
        X_train_seq, y_train, garch_train = prepare_sequences(df_train, feature_cols,
target_col, seq_length)
        X_test_seq, y_test, garch_test = prepare_sequences(df_test, feature_cols, target_col,
seq_length)
    else:
        X_train_seq, y_train = prepare_sequences(df_train, feature_cols, target_col,
seq_length)
        X_test_seq, y_test = prepare_sequences(df_test, feature_cols, target_col, seq_length)

    print(f"訓練データシーケンス数: {len(X_train_seq)}")
```

```
print(f"テストデータシーケンス数: {len(X_test_seq)}")

# モデル構築
print("モデル構築中...")
model = build_mt_garch_model(
    seq_length=X_train_seq.shape[1],
    num_features=X_train_seq.shape[2],
    prediction_type=prediction_type,
    include_garch=include_garch
)

# 早期停止のコールバック
early_stopping = tf.keras.callbacks.EarlyStopping(
    monitor='val_loss', patience=10, restore_best_weights=True
)

# モデル学習
print("モデル学習中...")
if include_garch:
    history = model.fit(
        [X_train_seq, garch_train], y_train,
        epochs=100,
        batch_size=32,
        validation_split=0.2,
        callbacks=[early_stopping]
    )
else:
    history = model.fit(
        X_train_seq, y_train,
        epochs=100,
        batch_size=32,
        validation_split=0.2,
        callbacks=[early_stopping]
    )

# テストデータでの予測
print("テストデータでの予測中...")
if include_garch:
    y_pred = model.predict([X_test_seq, garch_test])
else:
    y_pred = model.predict(X_test_seq)

# 評価
if prediction_type == 'volatility':
    # ボラティリティ予測の評価
    mse = np.mean((y_test - y_pred.flatten())**2)
    mae = np.mean(np.abs(y_test - y_pred.flatten()))
```

```
        print(f"テストデータMSE: {mse:.6f}")
        print(f"テストデータMAE: {mae:.6f}")

        # 散布図による可視化
        plt.figure(figsize=(10, 6))
        plt.scatter(y_test, y_pred, alpha=0.5)
        plt.plot([min(y_test), max(y_test)], [min(y_test), max(y_test)], 'r--')
        plt.xlabel('実際のボラティリティ')
        plt.ylabel('予測ボラティリティ')
        plt.title('ボラティリティ予測結果')
        plt.grid(True)
        plt.show()

    else:  # 'intraday_return'
        # 寄り引け戦略の評価
        # 予測値を基にしたトレーディングシグナル（上位20%をロング、下位20%をショート）
        percentile_80 = np.percentile(y_pred, 80)
        percentile_20 = np.percentile(y_pred, 20)

        signals = np.zeros_like(y_pred)
        signals[y_pred > percentile_80] = 1  # ロング
        signals[y_pred < percentile_20] = -1  # ショート

        daily_returns = signals.flatten() * y_test  # シグナル×実際のリターン

        # 勝率の計算
        win_rate = np.mean((daily_returns > 0).astype(float))

        # シャープレシオの計算（年率換算）
        sharpe_ratio = daily_returns.mean() / daily_returns.std() * np.sqrt(252)

        print(f"勝率: {win_rate:.4f}")
        print(f"シャープレシオ: {sharpe_ratio:.4f}")

        # 累積リターンの計算と可視化
        cumulative_return = np.cumprod(1 + daily_returns) - 1

        plt.figure(figsize=(12, 6))
        plt.plot(cumulative_return * 100)
        plt.title('寄り引けロングショート戦略の累積リターン (%)')
        plt.xlabel('取引回数')
        plt.ylabel('リターン (%)')
        plt.grid(True)
        plt.show()

    return model, history, y_pred, y_test
```

7. 論文とコードの主な違いと改良点

1. **Positional Encoding**:
   - 追加したPositionalEncodingLayerにより、Transformerが時系列の順序情報を学習できるようになりました
   - 論文のFigure 3、Figure 5に明示的に示されているこの重要な要素が実装できました

2. **ランダムサンプリング戦略の改良**:
   - 改良前: 特徴量（列）のランダムサンプリング
   - 改良後: 論文セクション2.5に基づき、観測データ（タイムステップ）のランダムサンプリング（90%）
   - これによりバギングの効果が本来の意図通りに機能します

3. **アーキテクチャ構造の忠実な実装**:
   - 改良前: 複数の独立したTransformerブロック（MHA+FFN）を並列実行し、出力を平均化
   - 改良後: 論文Figure 5に忠実に、複数のMHAの出力を平均化した後、共通のFFNに渡す構造
   - これにより論文のEquation 20のAMH（Average Multi-Head）が正確に実装されました

4. **GARCHモデルとの統合改善**:
   - GARCHモデルの予測値を特徴量として明示的に統合
   - 論文のハイブリッドモデル（Figure 6）のアーキテクチャを忠実に再現

5. **特徴量設計の論文準拠**:
   - 過去の対数収益率とそのラグ（論文式(2)）
   - 過去の実現ボラティリティとそのラグ（論文式(3)）
   - これらを論文に準拠した形で実装

これらの改良により、J-Quants APIから取得した日本株データに対して、論文により忠実なMultiTransformerモデルを適用することが可能になりました。特にバギングアプローチによる予測の安定化という論文の核心部分が正確に再現されています。

使用方法の例

ボラティリティ予測を行う場合:
```python
model, history, y_pred, y_test = train_mt_model_with_jquants(
    prediction_type='volatility',
    include_garch=True
)
```

寄り引けロングショート戦略用の日中収益率予測を行う場合:
```python
model, history, y_pred, y_test = train_mt_model_with_jquants(
    prediction_type='intraday_return',
    include_garch=True  # GARCHとの統合が不要な場合はFalseに設定
)
```

```
```