ChatGPT

統合MLトレーニングパイプラインの全体像

統合機械学習トレーニングパイプライン（`scripts/integrated_ml_training_pipeline.py`）は、株式リターン予測モデル ATFT-GAT-FAN の学習を一貫して実行するための包括的なスクリプトです [1]。このパイプラインは**SafeTrainingPipeline**によるデータ準備と品質検証のステップと、ATFT-GAT-FANモデルの学習・評価を行うステップに大別できます。全体のフローは次のようになります：

1. **MLデータセットの構築**: まず別途データ収集パイプラインを実行して、過去の株価・指標データから機械学習用データセット（特徴量と目的変数を含む）をParquetファイルに生成します（例：`output/ml_dataset_*.parquet`）。その後、安全学習パイプラインまたはデータビルダーを用いて特徴量拡張や品質検証を行った高品質なデータセットを準備します [2]。

2. **安全学習パイプライン　(SafeTrainingPipeline)**: 構築したMLデータセットに対し、安全性とデータリーク防止を重視した7つのステップからなる前処理パイプラインを実行します [3]。具体的には: **(1)データ読み込み、(2)高品質特徴量生成、(3)ウォークフォワード分割、(4)日次横断正規化、(5)ライトGBMによるベースライン学習、(6)株価相関グラフ構築、(7)性能レポート作成**　の順に処理されます [3] [4]。このパイプラインにより、データセット内の特徴量数や期間、銘柄数の報告、分割の検証、正規化統計、ベースラインモデルのIC・RankIC評価、簡易なグラフ構造（銘柄間の相関ネットワーク）などが出力され、データに問題がないか確認します [5] [6]。安全学習パイプラインの結果はJSONファイルとして保存されます [7]。

3. **ATFT完全学習パイプライン　(CompleteATFTTrainingPipeline)**: 検証済みデータセットを用いて、Adaptive Temporal Fusion Transformer + Graph Attention Network + Frequency Adaptive Normalization (ATFT-GAT-FAN) モデルの本格的な学習と評価を行います。統合パイプライン内部では非同期処理を用いながら次のステップを順次実行します [8] [9]:

4. **(1) 環境設定**: 再現実験に必要な各種環境変数（特殊損失やドロップアウト、AMP精度など）を設定し、ログディレクトリや出力ディレクトリを準備します [10] [11]。例えば`USE_AMP=1`や`DEGENERACY_GUARD=1`等、ATFTモデルの安定動作に関するパラメータがここで環境変数として適用されます [12] [13]。
5. **(2) MLデータセット読込と検証**: 指定されたパスまたは最新の出力フォルダから機械学習用データセット（Parquet形式）をPolarsで読み込みます [14] [15]。データが見つからない場合はテスト用にサンプルデータを生成します。読み込んだ`DataFrame`に対し、必要なカラム（例:`Code`, `Date`, `Close`など）が揃っているか、目的変数列（例えば`returns_1d`や`target`）が含まれるか、特徴量数が十分（50列以上）あるか、行数が0ではないか等を検証します [16] [17]。不足があればエラーとして中断し、問題なければデータ統計（行数・列数・期間・銘柄数など）をログ出力します [18] [19]。
6. **(3) 特徴量フォーマット変換**: Polarsのデータフレームを、ATFT-GAT-FANモデルが入力として要求する形式・特徴集合に変換します [20] [21]。具体的には、**UnifiedFeatureConverter**を使用して機械学習用データをモデル訓練用のデータセットに変換します [21]。このコンバータでは、Gogooku3で計算された豊富な特徴量群（約145次元）から、ATFTモデル用に必要な約13次元の特徴量に選択・集約します（例えば、1日・5日・20日リターンやRSI・MACD等のテクニカル指標、出来高比や価格位置など [22] [23]）。また、変換過程で**訓練データ・検証データ・テストデータ**に分割したParquetファイル群を出力ディレクトリ（`output/atft_data/train/`, `val/`, `test/`）に保存します [24] [25]。既に同じ出力ディレクトリに変換済みデータが存在する場合、環境変数`FORCE_CONVERT`が`1`でない限りそのデータを再利用します [26]。

7. **(4) 学習データ準備**: 変換されたデータのパス情報を集約し、訓練に必要なメタ情報を構造化します。例えば、訓練用Parquetファイルリストや検証用ファイルリスト、系列長や入力次元などを `training_data_info` にまとめます [27] [28]。訓練用ファイルが一つも無い場合はエラーとしますが、通常はconverterが出力したファイルパス群がここに格納されます [29]。**Directモード**（UnifiedFeatureConverterが未利用の場合）の場合は、データフレームを直接渡すフラグを立て、後段で直接データロードする準備をします [30] [31]。
8. **(5) ATFTモデル学習の実行**: PyTorch/Hydraベースの内部トレーナーをサブプロセスとして起動し、ATFT-GAT-FANモデルの学習を開始します [32] [33]。具体的には `python scripts/train_atft.py` を呼び出し、Hydra設定によって以下を上書き指定しています [34]:
    - データディレクトリ: `data.source.data_dir=<変換後データ>/train` [35]
    - ハイパーパラメータ: `train.batch.train_batch_size`（バッチサイズ）, `train.optimizer.lr`（学習率）, `train.trainer.max_epochs`（エポック数）, `train.trainer.precision`（精度モード）等 [36]
    - チェックポイント間隔やプログレスバー有効化など学習設定 [37]。
      このスクリプトはHydraの `config`（`configs/atft/config.yaml`）を基にモデル・データローダ・学習ループを構築します [38] [39]。データロード部分では**ProductionDataModuleV2**（またはLightningのDataModuleに相当するクラス）が用いられ、`output/atft_data/train` 内の複数Parquetファイルを**StreamingParquetDataset**として逐次読み込みます [40] [41]。このデータセットは大規模データでもメモリ常駐させず、バッチごとにストリーミングで読み出す仕組みで、日次横断的な**オンライン正規化**（各日のデータをその日の平均・標準偏差で標準化）も組み込まれています [42] [43]。モデル部分では移行済みの**ATFT_GAT_FAN**クラスを初期化し、設定に応じてTorchの `compile()`（モデルのJITコンパイル）を適用します [44]。学習は**RobustTrainer**（自前実装のトレーニングループ or PyTorch Lightningに類似）により実行され、最大エポック数や早期停止パラメータに従って訓練・検証を繰り返します [45]。訓練中、各エポックの損失やシャープレシオなど指標がログ出力され、検証用に最後のエポックで**予測値**も保存されます（`runs/last/predictions_val.parquet` に検証データに対する予測と実測を出力） [46] [47]。なお、学習時に環境変数 `CV_FOLDS` を指定すれば**Purged K-Fold**によるクロスバリデーションも可能で、スクリプト内で日付ベースのfold分割とエンバーゴ期間考慮を行い、所定のfoldで学習・評価する実装も含まれています [48] [49]（デフォルトでは単一の訓練・検証分割）。
9. **(6) 学習結果の検証**: サブプロセスの実行完了後、統合パイプラインはモデルの成果が期待値を満たしているか検証します。例えば、出力された最新チェックポイントファイル（`models/checkpoints/` 内の.ptファイル）を探し、モデルの**総パラメータ数**を算出して想定値（約560万）と比較します [50] [51]。また、学習ログや保存されたmetrics JSONから**Sharpe比**を抽出し、目標値0.849と照合します [52] [53]。チェックポイントが見つからない場合でも `runs/last/metrics_summary.json` 等からSharpe平均値を読み取り、結果をまとめます [54] [55]。この検証ステップにより、「モデルパラメータ数が期待通りか」「目標のSharpeに到達したか」が確認されます [56] [57]。
10. **(7) 結果の保存と報告**: 全ステップ完了後、パイプライン全体の結果を辞書オブジェクトにまとめてタイムスタンプ付きJSONファイルに保存します（`output/results/complete_training_result_<日時>.json`） [58] [59]。この結果には実験名、所要時間、使用した設定（fold数やエンバーゴ日数など）、各ステップの指標（例：特徴量拡張で増えた特徴数、正規化統計、ベースライン性能指標、グラフノード・エッジ数、訓練ログ抜粋、得られたSharpe値等）が含まれます [60] [61]。ログにはパイプライン全体の所要時間やメモリ使用量もまとめられ、コンソールにも成功完了のメッセージと達成したSharpe比が表示されます [62] [63]。
11. **(8) ポートフォリオ最適化（オプション）**: 学習モデルの検証結果として予測値が得られている場合、追加で**高度ポートフォリオ最適化**スクリプトを実行できます [64]。`runs/last/predictions_val.parquet` に保存された検証データの予測リターンを利用し、ロング・ショート戦略でのポートフォリオ構築シミュレーションを行います（例：上位20%銘柄をロング、下位20%をショート、手数料5bps考慮など） [65] [66]。`scripts/advanced_portfolio_optimization.py` をサブプロセスで呼び出し、その結果として `output/portfolio/report_*.json` にシャープレシオや最大ド

ローダウン等のポートフォリオ評価指標が保存されます [67] [68] 。統合パイプラインはこのJSONを読み込み、ポートフォリオのSharpe比等をログ出力します [69] 。

以上がフルのトレーニングフローです。要約すると、**安全学習パイプライン**でデータ品質と基本モデル性能をチェックしつつ特徴量を準備し、**統合ATFT学習パイプライン**でそのデータを圧縮・変換して最先端モデルの学習と評価を行い、成果を検証・保存する構造になっています。

パイプラインの依存モジュールとデータの流れ

上述の各ステップは、社内実装されたさまざまなモジュール・クラスによって支えられています。重要なコンポーネントとその役割、およびデータの流れについて以下に整理します。

- **データローダ & データセット**:
- ProductionDatasetOptimized / ProductionDatasetV2: 大規模な生データを高速に読み込むためのデータセットクラスです。Polarsを使った遅延評価や必要カラムのプロジェクション機能を備え、数百万行規模のデータでも効率的に読み出せます（ドキュメントでは `ProductionDatasetV3` とも記載） [70] 。SafeTrainingPipeline内部では簡易に `pl.read_parquet` で読み込んでいますが [71] 、将来的にはこの最適化版ローダに置き換えることで読み込み速度とメモリ効率がさらに向上する予定です。
- ProductionDataModuleV2: PyTorch Lightning風のデータモジュール実装であり、訓練・検証・テスト用のDataLoaderを一括管理します。 `train_atft.py` 内部でHydra設定 `final_config` を渡して初期化され、 `data_dir` 配下のParquet群から `setup()` でデータを準備します [72] [73] 。この際、環境変数で簡易モード（SMOKEテスト）として読み込むファイル数を制限することも可能です [74] 。ProductionDataModuleV2は内部で**StreamingParquetDataset**を使用しており、データをすべてメモリに載せずバッチ単位で逐次読み取ります [40] 。
- StreamingParquetDataset: 複数のParquetファイルをシーケンスデータセットとして扱うクラスです。ファイルリストと使用する特徴量カラム、目的変数カラム、シーケンス長などを指定して初期化します [40] [43] 。このクラスは `__getitem__` 呼び出し時に必要なファイルブロックを読み込み、オンザフライで**バッチ正規化**（日次のZスコア正規化）を適用しつつ、入力テンソルとターゲットを整形します。結果、巨大な時系列データも扱えるスケーラブルなデータ供給が実現されています。
- DayBatchSampler: DataLoaderに渡すサンプラで、日付単位でデータをまとめてバッチ化するカスタムサンプラです。ATFTモデルは日次横断での特徴量正規化を仮定するため、原則として各バッチが単一の営業日の全銘柄データになるようにSamplerで調整しています。環境変数 `USE_DAY_BATCH=0` を指定するとこれを無効化し、通常のランダムシャッフルバッチになる設定も可能です [75] 。小規模データ検証ではサンプラ無効化で動作確認し、大規模本番訓練では有効化して厳密な日次単位学習とする設計です。

- LightGBMFinancialBaseline: ライトGBMを用いた金融時系列用ベースラインモデルクラスです。マルチホライズン（複数期間先のリターン予測）に対応し、特徴量の正規化やエンバーゴ期間の考慮など金融タスク特有の設定を持っています [76] 。SafeTrainingPipelineのStep5でインスタンス化され、簡易なfitと性能評価（情報係数ICなど）が行われます [77] [78] 。デフォルトでは最初の1フォールド分のデータで訓練するため高速ですが、全データに対する厳密な評価にはなっていません [79] 。

- **特徴量エンジニアリング & 前処理**:

- QualityFinancialFeaturesGenerator: 高品質な財務特徴量を追加生成するクラスです。安全学習パイプラインのStep2で用いられ、クロスセクショナルな分位やボラティリティ閾値に基づき、元データに数種類の特徴量を付加します（例: シグマ2倍を超える異常値フラグや、銘柄横断でのランキング特徴など） [80] [81] 。これにより元の特徴量数139列から+6列程度増加し、計145列の拡張データを生成します [82] 。実装上、一旦Polars DataFrameをPandasに変換して処理しているため [83] 、この部分は将来的にPolarsネイティブ実装にすることでメモリ効率を改善できる余地があります。

- CrossSectionalNormalizer (V2): 日次横断（クロスセクション）方向の正規化を行うクラスです。各営業日について、その日に存在する全銘柄の特徴量分布を用いてZスコア正規化（平均0・標準偏差1化）し、時系列情報のリークなしにスケーリングを行います [84]。SafeTrainingPipelineのStep3で用いられ、まず訓練期間データで正規化パラメータ（各日の平均・標準偏差）を計算し、それを使って訓練・テストそれぞれに変換を適用します [85]。これによりデータ全体のばらつきを抑え、極端値の影響を軽減する効果があります [86] [87]。統合パイプラインでは明示的にこのNormalizerを呼んでいませんが、StreamingParquetDataset内で`online_normalization=True`により**各バッチ内で同等の処理**を実施しています [88]。2重適用を避けるため、安全学習パイプラインで正規化済みデータを使う場合は統合パイプライン側でonline_normalizationをOFFにする設計も考えられます。

- WalkForwardSplitterV2: 時系列データ専用のウォークフォワード交差検証用分割クラスです。指定した分割数n_splitsで、時系列を前方へずらしながら訓練期間とテスト期間のインデックスペアを生成します [89]。同時に、エンバーゴ期間（日数）を考慮して訓練とテストの間に隙間を空け、情報漏洩を防ぎます [90]。SafeTrainingPipelineのStep4で用いられ、生成した各フォールドのサイズやオーバーラップがないかを検証し、平均訓練サンプル数・テストサンプル数をログ出力します [91] [92]。統合パイプラインでも同様の目的で、Hydra設定もしくは環境変数`CV_FOLDS`によってPurged K-Foldを実行可能です [48] [93]。実装は異なりますが理念は同じで、複数期間にわたるモデル検証を可能にしています。

- **モデル関連**:

- ATFT_GAT_FANモデルクラス: Transformer系時系列モデル(ATFT)とGraph Attention Network、およびFAN正規化を組み合わせた社内独自モデルです [94]。全体で約560万パラメータを持ち、60営業日分のシーケンス長・145特徴量次元を入力として複数期間先のリターンを同時予測します [95]。PyTorch 2.0+で実装され、bfloat16の混合精度にも対応しています [96]。統合パイプラインではHydra設定`config.model`をもとにインスタンス化され [44]、Torch compileによる高速化やGPU張り付きの計算設定（TF32許可など）も行われます [97] [98]。このモデルは内部で**時系列注意機構**（Temporal Fusion Transformer部分）と**銘柄間関係モデリング**（GAT部分）を備えており、さらに**周波数適応型正規化**（FAN）で特徴スケールを動的調整します。これにより、4つの予測期間それぞれについてシャープレシオ0.849という高い性能が達成されています [99]。

- 学習スクリプト（train_atft.py）: ATFTモデル学習の中核を担うPythonスクリプトです。Hydraの`@hydra.main`デコレータで設定管理されており、`configs/atft/config.yaml`をデフォルト設定として読み込みます [100]。起動時には環境変数による設定上書きをまず適用し（例: `BATCH_SIZE`, `DEGENERACY_GUARD` 等） [101] [102]、設定の妥当性チェックを行った上で再現性のためのシード固定やGPU/AMP設定（A100向けTF32最適化など）を行います [103] [98]。その後、データ設定・モデル設定を統合して`final_config`を作成し、ProductionDataModuleV2からDataLoaderを取得、モデルを構築、学習ループを回します [104] [105]。学習中はEpochごとの損失や評価指標をログし、完了後にMLflowやTensorBoard/W&Bを使った実験管理ロガーが結果を保存します [106] [107]。train_atft.pyは7,000行以上に及ぶ大規模スクリプトで、様々な学習安定化の工夫（NaNクリップ関数や目標値クリップ [108] [109]、位相別損失スケジューリング、複数GPU対応のスカフォールドなど）が盛り込まれており、実運用で培われたノウハウが凝縮されています。

- **周辺ツール**:

- FinancialGraphBuilder: 銘柄間の相関グラフを構築するユーティリティです。SafeTrainingPipelineのStep6で使われ、過去60日間のリターン相関を計算してノード（銘柄）間エッジを張ります [110] [111]。負の相関も含め、各ノード当たり最大10エッジまで、相関係数閾値0.3以上を繋ぐといったロジックです [112]。本番データ全銘柄では完全グラフは巨大になるため、例として時価総額トップ50銘柄を対象に構築し、ノード数・エッジ数・平均次数などを出力しています [113] [114]。この結果はモデル解釈や特徴量追加に用いられます。一方、統合ATFT学習では**学習中に動的にグラフを構築**する仕組

みも用意されています。`integrated_ml_training_pipeline.py`で`--adv-graph-train`オプションを付けると、環境変数`USE_ADV_GRAPH_TRAIN=1`が設定され、Hydra設定内で高度なグラフ特徴抽出が有効化されます[115]。具体的には、EWM（指数移動平均）による動的相関計算やShrinkage推定のパラメータ（`GRAPH_CORR_METHOD`, `EWM_HALFLIFE`, `SHRINKAGE_GAMMA`など）が環境変数で訓練コードに渡され[116]、モデル内部またはデータロード時にこれらを用いてグラフ構造を計算・活用します。これにより、SafeTrainingPipelineで事前計算した固定グラフに比べ、学習中に最新のデータに基づく相関を反映したGAT層のトレーニングが可能です。

以上のモジュール群が連携し、**データ読み込み → 特徴量拡張 → 正規化・分割 → ベースライン検証 → 特徴量フォーマット変換 → モデル訓練 → 結果評価**という一連のデータフローを実現しています。生データからモデル出力までの流れをまとめると:

- 入力: 過去数年分の全銘柄株価データ（日次、数百万行規模, 140列超の特徴量）。
- 前処理 (SafeTrainingPipeline): データ読み込み・検証 → 高度特徴量追加（約+6列） → 銘柄横断正規化（各日Zスコア化） → 時系列分割とCV検証 → 簡易モデルでの性能チェック・基準策定 → 相関グラフ解析。
- 変換: 前処理済みデータセットをATFTモデル用に縮小・再構成（13特徴量×シーケンス長60に絞り込み、訓練/検証/テストファイル群に分割）。
- モデル学習 (CompleteATFTTrainingPipeline): GPU上で高次元時系列モデル訓練（Adaptive TFT + GAT）、バッチごとに内部正規化しつつ誤差逆伝播。期間中最良モデルをチェックポイント保存。
- 出力: 学習済みモデル（PyTorch state_dict, 約77MB）[117]、検証セットでの予測値一覧、評価指標（Sharpe, Lossなど）のログ、性能要件達成を示すレポートJSON、必要に応じポートフォリオシミュレーション結果。

この一連の流れによって、**データから最終成果までのトレーサビリティ**が保たれ、各段階での情報（例えば「特徴量何列増えたか」「正規化後の平均±偏差」「CV分割に重複はないか」「ベースラインICはどの程度か」「モデルのSharpeはいくらか」等）が詳細に記録されるようになっています。では、次に現在の実装で改善し得るポイントを機能・保守・性能の観点から考察します。

改善の提案: 機能面

現在のパイプラインは金融時系列特有の処理（ウォークフォワード検証や日次正規化、エンバーゴ期間考慮など）を網羅しており、機能的に充実していますが、いくつか**機能拡張や見直し**が考えられます。

- **ハイパーパラメータチューニングの統合**: 現状ではハイパーパラメータの最適化は別スクリプト（例:`scripts/hyperparameter_tuning.py`）で実行する想定ですが、統合パイプライン内でCVを活用したチューニングを行えると便利です。例えばOptuna等を組み込み、SafeTrainingPipelineで得られたベースラインを初期値としてATFTモデルの学習率や正則化パラメータを自動探索する機能を追加できます。これにより人手による試行錯誤を減らし、Sharpe向上につながる設定を発見しやすくなります。

- **ベースライン結果のフィードバック**: SafeTrainingPipelineで算出したLightGBMベースラインの性能指標（ICやRankIC）を、ATFTモデル学習時の比較対象や目的関数に組み込むことを検討します。例えば「モデルのSharpeがベースラインのSharpeを上回ること」を早期停止の一条件にする、あるいはベースライン予測を追加特徴量に加える（メタモデル化）といった拡張が考えられます。現状ベースラインは完全に独立しており、せっかく算出した指標が学習プロセスに活かされていません[79]。これを統合すれば、**「ベースラインを上回るモデル」を自動選別**でき、実運用上有益です。

- **特徴量生成の柔軟化**: QualityFinancialFeaturesGeneratorで追加している特徴量のセットを、設定ファイル経由で簡単にオン/off切り替えできるようにすると機能拡張性が上がります。例えば「シグ

マ閾値2による異常値指標」を無効化してみる、あるいは他の特徴量（例: **テキスト情報**や**ニュースイベントダミー**）を差し込めるようにする、といったニーズに対応できます。現在はジェネレータ内部にロジックが固定実装されていますが、YAML設定で「use_cross_sectional_quantiles: True/False」等切り替えられるよう拡張すれば、追加実験や将来の特徴量アップデートに柔軟に対応可能です [118]。

- **データ検証ルールの拡張**: 現在のデータ検証では主にカラムの存在や数をチェックしています [16]。今後、**欠損値の割合**や**目的変数の分布**も確認すると信頼性が増すでしょう。たとえば「欠損が一定以上ある特徴量は除外する」ポリシーや、「目的変数（リターン）の水準がおかしくないか」を事前に検知する仕組みです。特に金融データは外れ値や異常値が発生しやすいため、そうした検証をSafeTrainingPipelineに組み込んでおくと、学習前にデータ品質問題へ対処しやすくなります。

- **モデル評価指標の充実**: Sharpe比以外にも、情報係数(IC)の時系列推移やHit Ratio（予測の当たり率）、あるいは最大ドローダウンなど、モデルを多角的に評価する指標を算出・記録すると機能面で充実します。現在は最終的にSharpe平均値のみを重視していますが [53]、例えば**期間ごとのSharpe**や**年次ごとの性能**をSafeTrainingPipelineのレポートに載せたり、**検証データにおける累積リターン曲線**をプロット・保存したりすることで、モデル改善のヒントが得られます。統合パイプラインで生成した検証予測を使って、簡易なバックテスト結果を可視化する機能を追加するのも有用でしょう。

- **推論パイプラインとの連携**: 現在は訓練完了後に `scripts/models/atft_inference.py` を用いて推論を行う想定ですが [119]、統合パイプライン内で学習→推論まで一括実行し、例えば直近データでの予測結果をすぐ得られるようにすることも考えられます。これは運用フローの簡略化につながります。例えば `integrated_ml_training_pipeline.py` に `--run-inference` フラグを追加し、完了後に直近日データを投入して翌日リターン予測を出力する、といった拡張です。学習から実運用予測までをワンストップで実行できれば、人為ミスを減らし自動化が進みます。

- **直接学習モードの明確化**: 現行コードではUnifiedFeatureConverter未導入時に `train_files = ["direct_training"]` として直接学習モードをエミュレートしていますが [30]、これは裏舞台の実装が不明瞭です。もしDirectモードを正式にサポートするなら、`train_atft.py` 内で「データフレームを直接受け取って学習できる」インターフェースを用意し、統合パイプラインからそれを呼び出す形に改めると分かりやすいでしょう。現在はサブプロセスにDataFrameを受け渡せないため結局Parquet保存が必要ですが、この部分の設計を整理し「ファイル経由でなくメモリ上データで学習できる簡易モード」として実装すれば、開発・デバッグ時の迅速なトライアル（小規模データでの素早い学習確認）に役立つ機能となります。

改善の提案: 保守性

コードベースは全体的に機能豊富ですが、その分複雑で長大になっています。保守性（モジュール性・可読性・拡張容易性）を高めるために、以下のようなリファクタリングや構造見直しを提案します。

- **コードのモジュール分離と再利用**: 現在 `integrated_ml_training_pipeline.py` や `train_atft.py` には多数の機能が一箇所に詰め込まれています（特にtrain_atft.pyは7千行超 [120]）。これらを機能ごとに分割し、共通処理はユーティリティモジュール化することで可読性・再利用性を向上できます。例えば、「環境変数の設定適用」「シード固定処理」「学習結果解析」などは専用の関数もしくはクラスに分けるとスッキリします。実際、SafeTrainingPipelineでは各ステップをメソッド `_load_data`, `_generate_features`, `_normalize_features` … として分離しており [121] [122]、理解しやすく保守もしやすい構造です。同様に統合パイプライン側も、例えば「データセット検証」部分を関数 `_validate_dataset(df)` に切り出す [16]、`_convert_ml_to_atft_format` 内部で長い処理を更にサブ関数に分ける、といったリファクタリングが考えられます。

- **SafeTrainingPipelineとの統合または連携**: 現状ではSafeTrainingPipeline（安全学習）とCompleteATFTTrainingPipeline（統合学習）は別々のクラス・スクリプトとして実装されています。両者には重複する処理（データ読み込みや検証）があり、メンテ時に同期が必要です。例えばSafeTrainingPipeline内でもPolars→Pandas変換やウォークフォワード分割等を行っていますが [83] [89]、統合パイプライン側でも類似の処理を再度実施しています（CVの分割など） [48] [93]。これらを**一元化**するために、両パイプラインを統合した「FullTrainingPipeline」クラスを新設し、フラグによって「データ準備＋ベースラインのみ実行」か「フルにモデル学習まで実行」かを切り替える、という構造にすることができます。擬似コードで示すと以下のようなイメージです:

```
class FullTrainingPipeline:
    def __init__(self, ...):
        self.safe_pipeline = SafeTrainingPipeline(...)
        self.complete_pipeline = CompleteATFTTrainingPipeline(...)
    def run(self, run_full_model: bool = True):
        data_ok = self.safe_pipeline.run_pipeline(...)
        if not data_ok:
            return {"error": "Data preparation failed"}
        if run_full_model:
            result = self.complete_pipeline.run_complete_training_pipeline(...)
            return result
        else:
            return {"message": "Baseline training completed"}
```

こうすることで、データ整備とモデル訓練を統一的なインターフェースで扱え、例えばメインスクリプトから `FullTrainingPipeline.run(run_full_model=True)` を呼ぶだけで一連の処理を実行できるようになります。将来的にSafe部分の改良（例: 正規化方法の変更）があった場合でも、一箇所の修正で統合パイプラインにも反映されるため保守負荷が軽減します。

- **環境変数依存の削減と設定ファイルへの移行**: 本システムは各所で環境変数をフラグやパラメータとして使用しています（例: `FORCE_CONVERT`, `GRAPH_K`, `USE_AMP`, `BATCH_SIZE` など） [12] [75]。環境変数は手軽な反面、見落としや設定漏れが発生しやすく、またコード上でどの変数が有効なのか把握しづらいという問題があります。保守性向上のため、**Hydraの設定ファイルに極力集約**することを推奨します。例えば `configs/atft/config.yaml` に `convert.force: false` や `graph.builder: advanced` 等のオプションを設け、環境変数ではなくHydraオプション（コマンドライン引数やdefaults）で切り替えるようにします。これにより設定が明示的にドキュメント化され、チーム内で共有・追跡しやすくなります。また、環境依存を減らすことで別環境（例: ローカルPCやCI上）で動かす際の不具合も減るでしょう。

- **ログと例外ハンドリングの整理**: 現在、標準出力への `print` や `logger.info` が混在しており [123] [62]、例外発生時には `traceback.print_exc()` で詳細を出力しています [124]。ログ出力は基本的に `logging` モジュールに統一し、冗長なprintは避けるとよいでしょう。また、例外処理も現在は広域でキャッチして失敗を記録するのみですが [125]、可能であれば**段階別の例外**を定義して原因を特定しやすくすることが考えられます。例えばデータ欠損によるエラーと、モデル収束失敗によるエラーを区別するカスタム例外クラスを作り、呼び出し元で適切にハンドリングする、といった実装です。これにより、問題発生箇所の切り分けが保守担当者にとって容易になります。

- **コメントとドキュメントの充実**: コード内には日本語コメントやDocstringが多く書かれており理解の助けになっています [1] [126]。この方針を維持・強化し、特に複雑な箇所（例えばPurged K-Fold分割のロジック [48] [93] や学習時の損失スケジューリング処理）には詳細な説明コメントを追加すると、

将来新たな開発者が参加した際にもスムーズにコードを追えるでしょう。また現在README やdocs フォルダに豊富な資料がありますが、コードと同期していない記述も散見されます（たとえば SafeTrainingPipelineのステップ順がドキュメントと実装で異なる箇所など [4] [3] ）。ドキュメントは定期的にアップデートし、設計変更時にはREADMEやアーキテクチャ図も更新する運用を徹底することが望ましいです。

- **テストの整備**:　機能が増えるにつれ、単体テスト・集約テストの整備も保守性には不可欠です。現状 `tests/` 以下にどの程度テストコードがあるか不明ですが、例えば**データ変換ロジック**について、小規模サンプルで期待通り13列に削減されるかをチェックするテスト [127] [25] 、学習ループについてエポック数を1に設定して正常終了するかを見るスモークテストなどを用意すると安心です。特に Purged　K-Foldや高度ポートフォリオの部分はロジックが複雑なので、極力自動テストで退行がないことを保証できるようにすると良いでしょう。テストが充実すれば、今後リファクタリングを行う際も安心してコードを変更できます。

改善の提案: 性能面

大規模データとディープラーニングを扱う本パイプラインでは、速度・メモリ効率・ハードウェア活用も重要です。現在の実装にはPolarsやPyTorch高速化技法の活用など工夫が見られますが、さらなる性能向上のための提案を以下に示します。

- **Polarsの更なる活用と不要な変換削減**: SafeTrainingPipelineではPolarsで読み込んだ後、一部処理で Pandasに変換しており [83] 、この変換コストとメモリ使用量が懸念されます。可能であれば特徴量生成や日次正規化もPolars上で完結させ、Pandas変換を無くすことが望ましいです。Polarsはグループ集計や結合も効率的に行えるため、QualityFinancialFeaturesGeneratorの処理（例:　銘柄横断の分位計算）もPolarsの `groupby` や `quantile` を駆使して実装すれば、より大規模データにスケールしやすくなります。あるいはPandasに頼る場合でも、 `df.to_pandas()` ではなく必要な列だけ抽出して変換するなど、メモリ節約の工夫が考えられます。例えば「特徴量生成では価格・出来高関連の十数列しか使わない」のであれば、それら列だけPolars→NumPy配列にして計算し、結果をPolarsに戻すことでDataFrame全体のコピーを避けるといった方法です。

- **データロードの並列化と遅延評価**: ProductionDataModuleV2ではDataLoaderの `num_workers` を環境変数で指定でき、既定では16ワーカーで並列読み込みしています [128] 。これは良いですが、SafeTrainingPipeline側の最初のデータ読み込み（pl.read_parquet）はシングルスレッドです。Polarsは `use_pyarrow=True` や `parallel=True` オプションで高速化できますし、また複数ファイルを一度に読む際はPolarsのscan機能で遅延読込＋必要列プロジェクションすることで、不要なI/Oとデータを読み飛ばせます [70] 。例えば「目的変数のない列はロードしない」「期間フィルタを先に適用して必要範囲のみ読む」などの対応です。実運用では数百万行・150列近いデータになるため、**列・行のフィルタリング読み込み**はメモリと読み込み時間の削減に直結します。

- **GPUリソースの有効活用**: ATFTモデル学習ではA100 GPUを使っているとのことですが [129] 、その使用率を常に高く保つ工夫が考えられます。例えば現在の実装では1GPUで学習していますが [130] 、将来的にデータがさらに増えた場合やモデルをアンサンブルする場合、**マルチGPU並列**や**分散学習**に対応できると望ましいです。PyTorch　LightningのDDPモードやHorovodなどを導入し、Hydra設定で `train.trainer.devices=4` のように指定するだけで自動的に4GPU並列学習できる設計にしておくと拡張性があります（Lightningに近い実装なら比較的対応容易と思われます）。また、GPUの混合精度は既に有効化されていますが、Grad-CAMのような**省メモリアルゴリズム**（勾配チェックポイントなど）を組み込む余地もあります。特にシーケンス長60・バッチサイズ2048はメモリ負荷が大きいため、例えば「中間層アクティベーションを逐次再計算してメモリ節約する（checkpointing）」「不要な一時Tensorを明示的に `del` して `torch.cuda.empty_cache()` を呼ぶ」等でGPUメモリ使用ピークを抑える工夫も考えられます。

- **計算量のボトルネック解消**:　学習ループ内で計算負荷の高い箇所を特定し、最適化できるなら行います。例えばGraph Attention部分で全ノード間の注意係数を計算しているなら、スパース行列演算に置き換える・事前に隣接リストを絞り込んでおくなどです。すでにFinancialGraphBuilderでエッジ数を絞っていますが [112] [114]、学習時に動的グラフを使う場合は負荷増大が予想されます。そこで**グラフ計算の低頻度化**（例えば毎エポックではなく数エポックごとに更新）や、グラフ関係部分をカスタムCUDAカーネル化する、といった検討もあり得ます。ただしこれは高度な改善になるため、まずはPyTorch Profiler等でボトルネック解析を行い、「データロード待ちが発生していないか」「特定のレイヤーで計算時間の大半を費やしていないか」を確認することを推奨します。その上で、もしデータロードがボトルネックなら前述の並列度調整やprefetchを増やす [131]、計算がボトルネックなら該当部分の演算を見直す、といった対策を取ると良いでしょう。

- **メモリ使用量の監視と制御**: ドキュメントによればピーク時メモリ使用量は8〜16GB程度とのことですが [132]、PolarsデータフレームとPyTorchテンソルを同時に保持するとそれなりのメモリを消費します。SafeTrainingPipelineでは `memory_limit_gb` 引数を受け取っており、メモリ超過時に警告や対応をしているようですが [133] [134]、例えば**メモリ計測を各ステップ終了時にログ**したり、`df` を使い終えたら明示的に開放（PolarsのLazyFrameを使えば不要データの解放は自動）することも考えられます。また、不要になった中間結果（例えばbaselineモデル本体や生成済みグラフオブジェクト）は適宜破棄し、Pythonのガベージコレクタを手動起動する（`gc.collect()`）こともメモリ確保に有効です。現在の実装でも `import gc` はされていますが [135]、効果的に使われていないようです。特に学習前に巨大DataFrameを保持したままだとGPUメモリだけでなくCPUメモリも圧迫するため、**学習開始前に不要データをクリア**しておくと、OSによるスワップを防ぎ安定動作につながります。

- **I/Oボトルネックへの対処**: Parquetの読み書きが頻繁に発生する構造上、ストレージI/Oが遅延要因になる可能性があります。対策として、NVMeなど高速ストレージの活用や、可能なら**メモリ上の仮想ファイルシステム（RAMディスク）**に一時データを書き出すことも検討できます。また、Parquet圧縮を必要に応じて無効化または簡易な圧縮コーデックにする（圧縮はサイズ削減になる反面CPUを消費）など、I/OとCPU負荷のトレードオフを調整します。さらに、学習時のチェックポイント保存頻度も適切に設定するべきです。例えば各エポック終了毎に数百MBのモデルを保存するとディスク負荷が高いため、`save_top_k=3` のみ保存するよう既に設定されています [136] が、これを厳守すること、および不要な古いチェックポイントや一時ファイルを定期的にクリーンアップする運用が必要です。

以上の性能チューニング提案により、パイプライン全体の処理時間短縮や資源効率の向上が期待できます。実際、J-Quants　4年分データ取得〜学習完了まで現状でも数時間程度とのことですが [117]、最適化次第では更なる短縮も可能でしょう。特にボトルネックを慎重に測定し、**「待ち時間を隠す」**（例:　データロードとGPU計算の並行実行）や**「無駄な計算をしない」**（例: 不要特徴量のカット）といった原則に沿って改善することで、よりスケーラブルで高速なパイプラインに仕上げていけると考えられます。

アーキテクチャ上の不整合と設計上の懸念

最後に、現在のアーキテクチャにおける一貫性の欠如や将来的な開発で障害となり得る設計上の懸念点を指摘します。

- **パイプライン間の処理順序の差異**: SafeTrainingPipelineでは「特徴量生成→ウォークフォワード分割→正規化」の順序ですが [3]、文書上は正規化を分割前に行うようにも読め、実装とドキュメントに齟齬があります。また統合パイプラインでは正規化ステップを明示的に行わずモデル内に任せています。これらの**順序や責務の違い**は将来混乱を招きかねません。推奨される解決策は、**全パイプラインで共通のデータ前処理順序を定義し遵守する**ことです。例えば「正規化は常に学習データに対してfitし、その統計でテストデータも変換する」というルールを決め、SafeでもATFTでもそれに従う実装にします（ATFT側は現在オンライン正規化だが、将来的にオフライン統計を事前計算して与える方式に

変えるなど）。一貫性を持たせないと、場合によってはモデル性能の比較がフェアでなくなる可能性があります。

- **重複実装の存在**: 例えばウォークフォワードCVはSafeTrainingPipelineでは専用クラスを使い、ATFT側では手動実装しています [89] [48]。二重実装はバグ温床になりやすく、実際片方でエンバーゴの扱いミスが起きてももう片方には反映されない恐れがあります。これは先述の**モジュール統合/共通化**で解決できます。同じアルゴリズムは一箇所にまとめ、両方から呼び出すようにすべきです。

- **非同期処理とサブプロセス使用**: 統合パイプラインはasync/awaitを用いており [8]、さらに内部で複数のsubprocess呼び出し（train_atft.pyやoptimization.py）をしています [32] [65]。非同期とマルチプロセスが混在する構造は理解が難しく、デバッグもしにくいです。例えば例外が発生したときにどのタスクで起きたかトレースしづらい、並行実行が増えた場合のリソース競合が見えにくい等の問題があります。シンプルさと保守性のためには、**可能なら同期処理に戻しつつサブプロセス呼び出しを減らす**ことが検討できます。HydraはPython APIから呼び出すことも可能なので（`hydra.compose`で設定生成しモデルクラスを直接呼ぶ等）、train_atft.pyをサブプロセスで叩かず統合パイプラインの中でモデル訓練用クラス（IntegratedTrainerなど）を直接インスタンス化して実行する構造にすれば、プロセス間通信のオーバーヘッドやログの分断もなくなります。現状はラッパースクリプトを多用しているため、実行フローの追跡が難しいという欠点があります。

- **設定の散在と暗黙の依存**: 環境変数、Hydra設定、コード内部定数が混在しており、ある機能が有効かどうかの判断が難しくなっています。例えばグラフ関連機能は`USE_ADV_GRAPH_TRAIN`環境変数に依存していますが [115]、これはドキュメントや設定ファイル上には明示されていません。開発者はコードを熟読しないとこのスイッチに気づけないでしょう。このような**隠れたスイッチ**が他にもないか精査し、一元管理することが重要です。理想はHydraの設定ファイルに全てのフラグを盛り込み、ドキュメント化することです。「暗黙知」を排し、「明示的な設定」として扱うことで、将来新機能追加時にも影響範囲を把握しやすくなります。

- **巨大スクリプトの肥大化**: `train_atft.py`のように非常に長いスクリプトは、そのまま将来も増築していくと手に負えなくなる恐れがあります。読み手にとっても重要箇所の把握が難しく、バグが紛れ込んでも見落とすリスクがあります。適切に機能分割し、ファイル数が多少増えても論理的にまとまった構成にリファクタリングすることが求められます。現在すでに`src/`以下にモデルやデータローダ、トレーナー等が移行されていますが [137] [138]、scripts内にも残っているロジックを段階的にsrc側に移していくと良いでしょう。将来的には`pip install gogooku3`でパッケージとして導入できるように整理すれば、他プロジェクトからの再利用も容易になり、保守コストも下がるはずです。

以上、詳細に検討した結果、本システムの機能・保守性・性能はいずれも高水準にありますが、さらなる改善余地も確認できました。提案した対策を講じることで、より**柔軟で頑健**かつ**効率的**なトレーニングパイプラインになることが期待されます。そして設計の一貫性を保つことが、長期的な開発・運用における品質確保に繋がるでしょう。

References: SafeTrainingPipeline 実装 [3] [86]、統合パイプライン実装 [8] [24]、ATFT移行ドキュメント [38] [23] など（本文中に引用）。

---

[1] [8] [9] [10] [11] [14] [15] [16] [17] [20] [21] [24] [25] [26] [27] [28] [29] [30] [31] [32] [33] [34] [35] [36] [37] [46] [47] [50] [51] [52] [53] [54] [55] [56] [57] [58] [59] [62] [63] [64] [65] [66] [67] [68] [69] [75] [115] [116] [124] [127]

integrated_ml_training_pipeline.py

https://github.com/wer-inc/gogooku3/blob/eeb49b1bbe910e8a3ead56864736cd89bfa15a51/scripts/integrated_ml_training_pipeline.py

2 132 JQUANTS_FULL_SCALE_IMPLEMENTATION.md
https://github.com/wer-inc/gogooku3/blob/eeb49b1bbe910e8a3ead56864736cd89bfa15a51/docs/architecture/JQUANTS_FULL_SCALE_IMPLEMENTATION.md

3 5 6 7 18 19 60 61 71 77 78 79 81 83 86 87 89 90 91 92 112 114 118 121 122 123 125 126 133 134 safe_training_pipeline.py
https://github.com/wer-inc/gogooku3/blob/eeb49b1bbe910e8a3ead56864736cd89bfa15a51/src/gogooku3/training/safe_training_pipeline.py

4 70 76 80 82 84 85 94 95 96 110 111 113 model-training.md
https://github.com/wer-inc/gogooku3/blob/eeb49b1bbe910e8a3ead56864736cd89bfa15a51/docs/ml/model-training.md

12 13 128 130 136 train_atft_wrapper.py
https://github.com/wer-inc/gogooku3/blob/eeb49b1bbe910e8a3ead56864736cd89bfa15a51/scripts/_archive/train_atft_wrapper.py

22 23 ATFT_TRAINING_MIGRATION_PLAN.md
https://github.com/wer-inc/gogooku3/blob/eeb49b1bbe910e8a3ead56864736cd89bfa15a51/docs/ml/atft/ATFT_TRAINING_MIGRATION_PLAN.md

38 39 99 117 119 137 138 ATFT_MIGRATION_COMPLETE.md
https://github.com/wer-inc/gogooku3/blob/eeb49b1bbe910e8a3ead56864736cd89bfa15a51/docs/ml/atft/ATFT_MIGRATION_COMPLETE.md

40 41 42 43 44 45 88 106 107 131 integrated_trainer.py
https://github.com/wer-inc/gogooku3/blob/eeb49b1bbe910e8a3ead56864736cd89bfa15a51/src/training/integrated_trainer.py

48 49 72 73 74 93 97 98 100 101 102 103 104 105 108 109 120 129 135 train_atft.py
https://github.com/wer-inc/gogooku3/blob/eeb49b1bbe910e8a3ead56864736cd89bfa15a51/scripts/train_atft.py