4.1なぜシステムを作るのか

機械学習をプロダクトやサービスで有効活用するためにシステムに組み込む必要がある。他のソフトウェアと組み合わせて、モデルが他のソフトウェアから呼び出される仕組みを整える必要がある。

4.1.1　はじめに

機械学習のモデルをひとつ作るためには多種多様なデータ、アルゴリズム、パラメータを組み合わせる必要がある。

- 運用上考慮すべき点
    - データ品質管理
        - 定期的なデータの品質確認
    - 既存ソリューションの利用
        - 即時に利用可能なサービスや過去の案件で使用したソリューションを利用して、特定のユースケース以外に対する開発コストの削減を図る。
    - モデルのパフォーマンス指標の定義
        - パフォーマンス指標をビジネス目標に対応させる。
        - これらの指標を継続的に監視、モデルの再学習を行う体制を構築
    - バージョン管理
        - データセット、特徴量変換コード、ハイパーパラメータ、訓練したモデルのバージョン管理など
        - 開発自体のバージョン管理
    - 適切なサービス、ハードウェアの利用
        - 開発フェーズに応じた適切なリソース、ハードウェアの選択や管理
    - デプロイしたモデルの継続的な監視
        - データやモデルのドリフトを監視し、再訓練などの基準を設定
    - ワークフローの自動化
        - 一貫性のある自動化されたパイプラインの構築
        - ヒューマンエラーの防止やヒューマンリソースの効率化を目指す。
        - パイプラインには本番環境前にモデルを承認するフェーズを用意するなど、管理面での最適化を含む

本Chapterでは機械学習のモデルを本番システムに組み込む方法をパターン化し、それぞれの実装方法やメリット、デメリットを説明する。

4.1.2　機械学習を実用化する

- Webシングルパターン：
    - ひとつの小さなモデルをひとつの推論器で同期的に推論します。
- 同期推論パターン：
    - リクエストに対して同期的に推論します。
- 非同期推論パターン：
    - リクエストに対して非同期に推論します。
- バッチ推論パターン：
    - バッチ処理で推論を実行します。

- 前処理・推論パターン：
  - 前処理と推論でサーバを分割します。
- 直列マイクロサービスパターン：
  - 依存関係のある推論を順に実行します。
- 並列マイクロサービスパターン：
  - ひとつのリクエストに対して複数の推論器で推論します。
- 時間差推論パターン：
  - 同期推論と非同期推論を組み合わせて時間差で推論します。
- 推論キャッシュパターン：
  - 推論結果をキャッシュしパフォーマンスを改善します。
- データキャッシュパターン：
  - 推論前のデータをキャッシュしパフォーマンスを改善します。
- 推論器テンプレートパターン：
  - 推論パターンをテンプレート化し、開発効率の改善を図ります。
- EdgeAIパターン：
  - スマホや自動車等、クライアントデバイスで推論を実施します。

4.2 Webシングルパターン

WebAPIサービスに機械学習の推論モデルを組み込む。APIへデータとともにリクエストを送ることで、推論結果を入手できる仕組み。推論システムを作る際の最もシンプルで基礎的な構成。

- ユースケース
  - 最もシンプルな構成で推論器を素早くリリースし、モデルのパフォーマンスを検証したいとき。
- 解決したい課題
  - 直近の期間において、できる限り素早く学習した推論器をリリースし、高い正答率を持つモデルを本番環境で運用したい。
- 設計イメージ

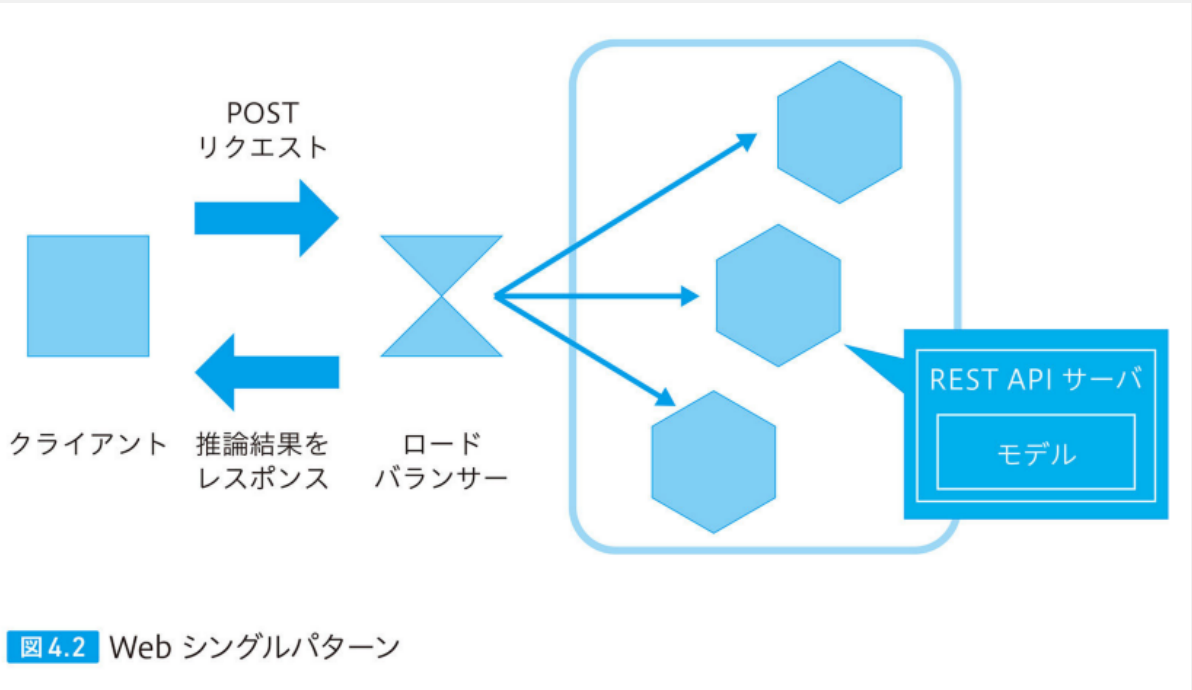

図4.2 Web シングルパターン

- 概要

    - 1サーバで最低限の機能のみを開発する。
    - Webアプリケーションサーバにモデルを同梱させ、同一サーバにFastapi1やFlaskなどのRESTインターフェイス、前処理と学習済みモデルをインストールすることで、シンプルな推論器を実装する。
    - DBやストレージ等のデータを永続的に保持するパーシステント層を用意することなく、Webサーバ1台で構成することが可能。
    - 可用性のために複数台のWebサーバで運用する場合はロードバランサーを導入して負荷分散する。

モデルを推論器に組み込む手法は3.4節「モデルインイメージパターン」や3.5節「モデルロードパターン」を参照。

- 利点
    - 推論器を素早く稼働させることができる。
    - 汎用的な仕組みなので、幅広い稼働させることのできる基盤の選択肢が豊富。
    - 開発、運用に加えて障害対応や復旧が簡単。
- デメリット
    - スケールアウトやログ設計が難しい
    - 複雑な処理やモデル、ワークフローに対応できないケースが多い。

外部クライアントからWed APIへの推論リクエストがあった場合、処理方法は大きく分けて、同期処理と非同期処理の2パターンある。

4.3 同期推論パターン

外部のクライアントから推論器に対してリクエストがあった場合、同期的に推論結果を返すようなシステムの設計にする場合。できる限りリクエストに対して推論の応答を返す必要がある。

- ユースケース
    - システムのワークフローとして、推論結果が出るまで次の段階に進めないような設計になっているケース
    - ワークフローが推論結果に依存しているケース
- 解決したい課題
    - クライアントのアプリケーションが推論結果に応じて後続の処理を分岐する場合。
    - 具体例：
        - 工場の生産ラインで製造物の異常品検知をするシステム

- 設計イメージ

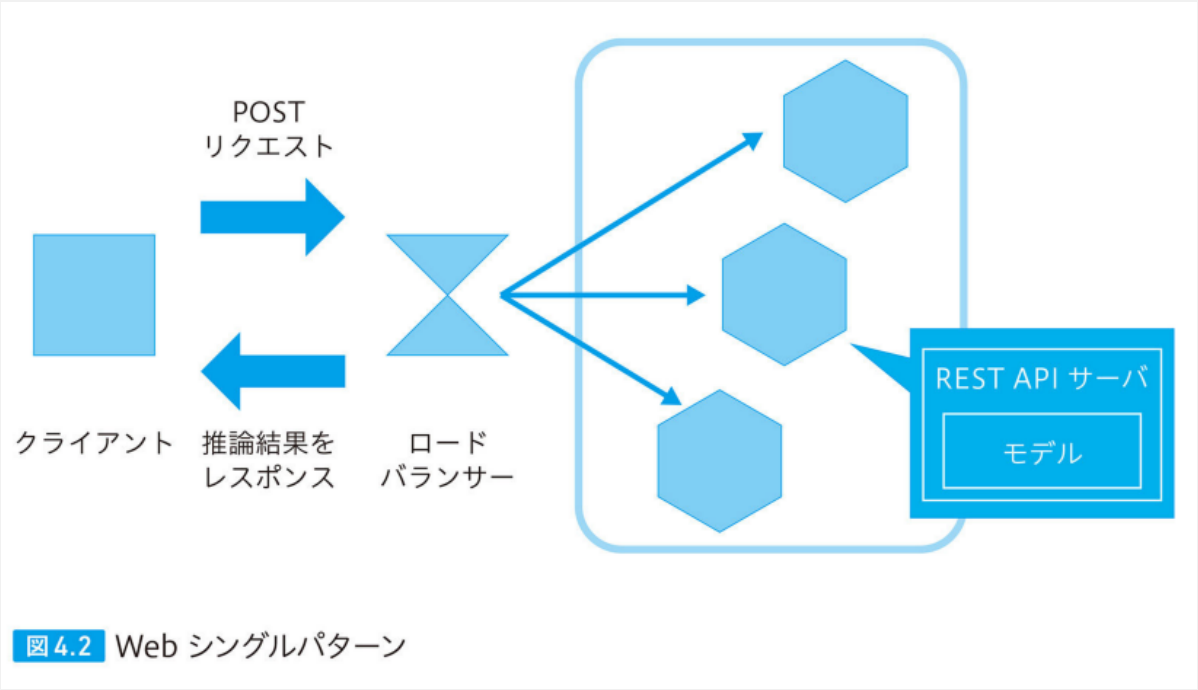

図4.2 Web シングルパターン

- 概要
  - クライアントは推論リクエストを送信後、レスポンスが得られるまで後続の処理を進まずに待機する。
  - 推論を含めたワークフローを逐次的に作ることができ、ワークフローそのものをシンプルにできる。
- 利点
  - シンプルで汎用的な仕組みなので、幅広い稼働させることのできる基盤の選択肢が豊富。
  - 開発、運用に加えて障害対応や復旧が簡単。
  - 順序だててワークフローを作成できる。
- デメリット
  - スケールアウトやログ設計が難しい
  - 推論プロセスの一部に遅い処理が入っている場合は、その部分を高速化するか、見直す必要がある。
  - システムの用途によっては、直接CXにつながる可能性がある。
  - 遅延や障害が発生する場合には、推論を待たずに次のプロセスに進ませるなどのサービス全体での工夫が必要になる可能性がある。

4.4 非同期推論パターン

外部のクライアントから推論器に対してリクエストがあった場合でも、推論結果をレスポンスしない場合。

- ユースケース

    - クライアントアプリケーションで推論リクエスト直後の処理が推論結果に依存しないワークフロー
    - 呼び出し元のクラインアントと推論結果の出力先を分岐させるとき
    - 推論に時間がかかり、クライアントの待機時間を作りたくないとき

- 解決したい課題

    - マルチモーダル機械学習(複数の推論器のアンサンブルなど)や、計算量の大きい深層学習モデルなど、推論に計算時間を要するようなモデルを活用する場合
    - 同期的に処理する必要のないワークフロー
    - 具体例：
        - インスタやFBへの投稿など

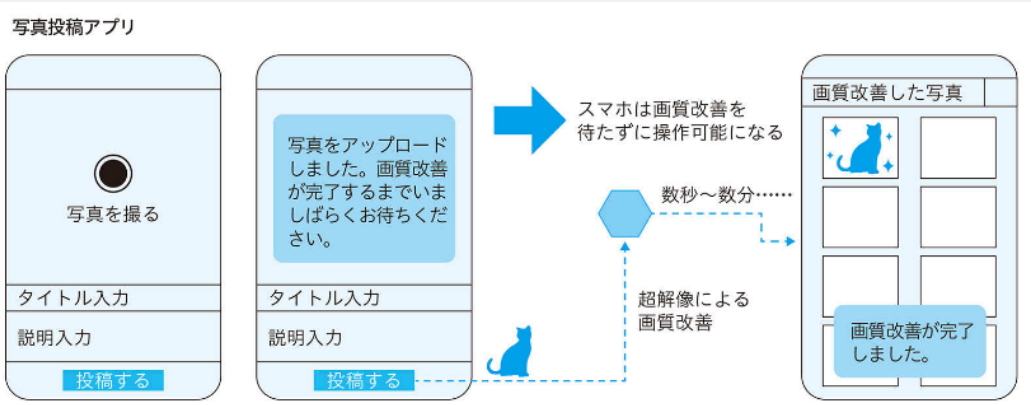

図4.7 ディープラーニングの超解像で画質改善してユーザに提供するスマホアプリ

多くのシステムは、工夫次第でワークフローを非同期に切り離すことができる。特に推論に時間を要するようなモデルを活用する場合は非同期なワークフローを組み込むことでシステム全体のパフォーマンスを維持することを推奨する。

- 設計イメージ

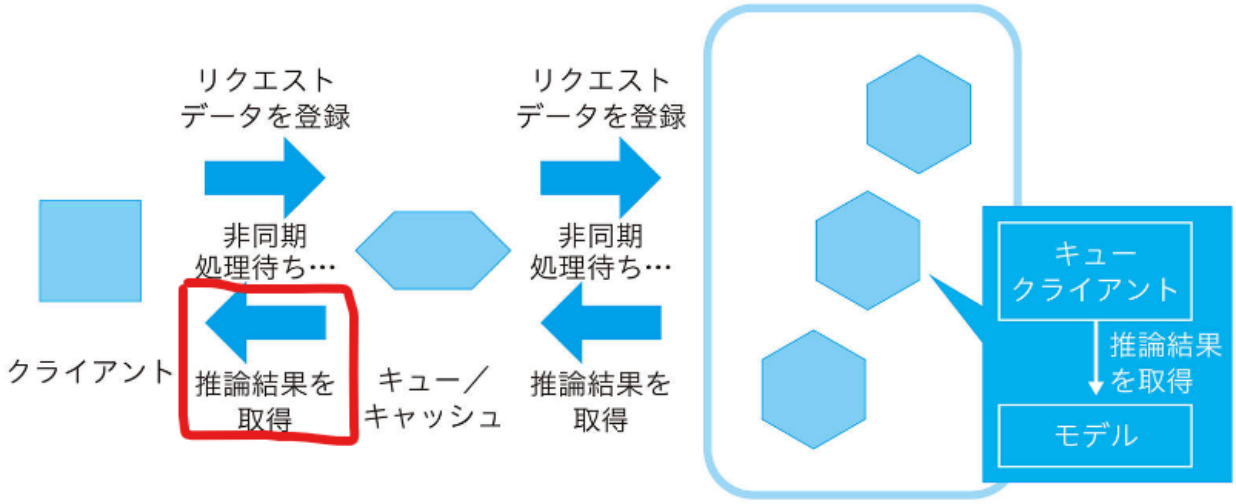

図4.8 非同期推論パターン①

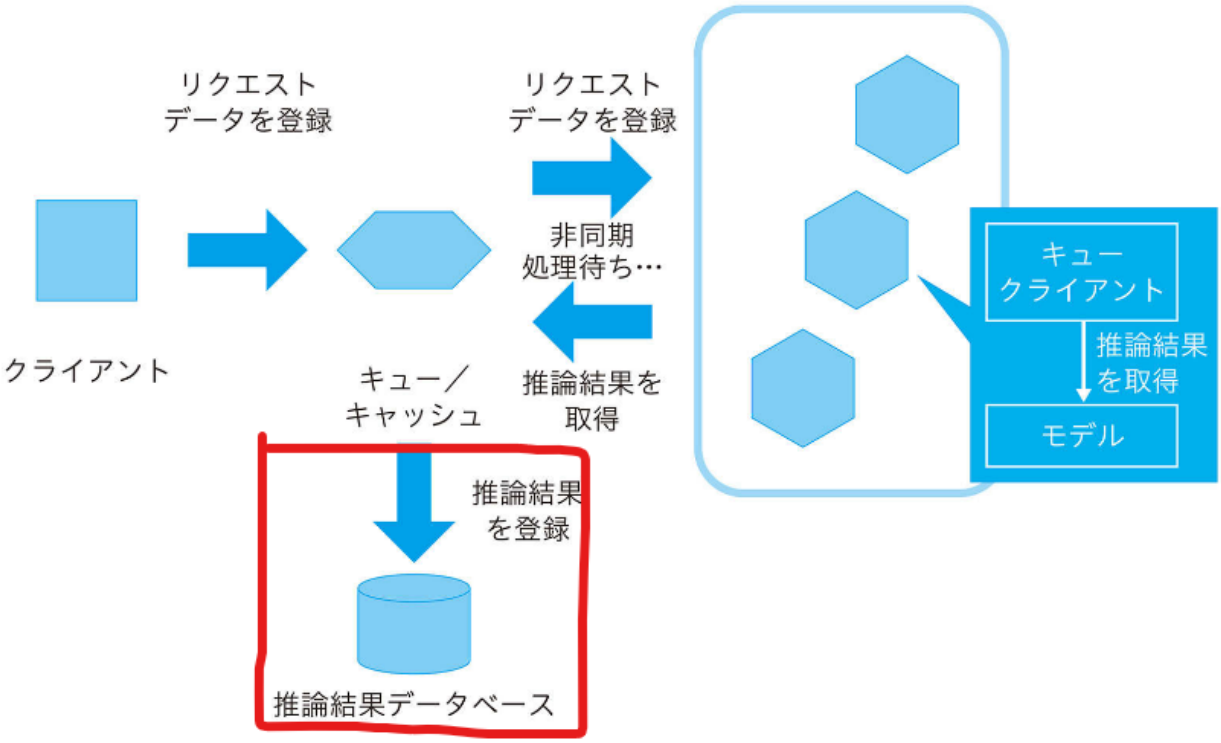

図4.9 非同期推論パターン②

- 概要
  - 中間にキューやキャッシュを配置して、推論リクエストと推論結果の取得を非同期にする。推論とリクエストを切り離すことで、クライアントは推論結果出力先に定期的にアクセスして習得すればよい。
  - 推論結果をクライアントに直接渡すようにすることもできるが、そのためのコネクションが必要になり、システムが複雑になるため推奨されない。
- 利点
  - クライアントのワークフローと推論を疎結合にすることが可能。
  - 推論の遅延が長い場合でも、クライアントへの悪影響を回避できる。
- 検討事項

    - 推論の順番を保つ必要がある場合(キューの利用を検討)
        - 推論器がサーバー障害等で推論のためのデータ習得に失敗したケースや推論自体に失敗したケースでは一度デキューした(キューより取り出した)リクエストを本来戻す必要がある。
        - 障害の原因によっては、キューを戻せなくなる場合もある。そのため、キュー方式では、すべてのデータについて推論できるとは限らないサービスとなってしまうケースがある。
    - 推論の順番にこだわらない場合(キャッシュの利用を検討)
        - 推論の完了したデータについては推論済みのデータとして登録すればよいので、キュー方式とは違い、リトライが可能になる。
    - データの状態が完全でないなど、推論が失敗する原因は多岐に渡り、一定時間以上処理に時間がかかる場合にはリクエストを破棄するようなシステム構築が必要。 基本的には時系列的なデータや推論の順場を守らなければならないようなケースには不向きなので、その場合は同期パターンを利用する方が良い。

4.5 バッチ推論パターン

ジョブとして前処理と推論を実行して、推論結果を保存する。

- ユースケース
    - リアルタイム又は準リアルタイムで推論する必要がないとき(準リアルタイムとは)
    - 過去のデータをまとめて推論したいとき
    - 定期的に蓄積されたデータを推論したいとき
- 解決したい課題
    - 具体例：
        - 過去3ヶ月のデータをもとに来月の人員配置を計画し、月末に処理したい。
- 設計イメージ
  ユースケースに沿って一定の時間や期間、条件で実行するようにバッチ・ジョブを起動するジョブ管理サーバーが必要になる。基本的にはクラウドサービスやKubernetesを利用し、コストの削減を行う。

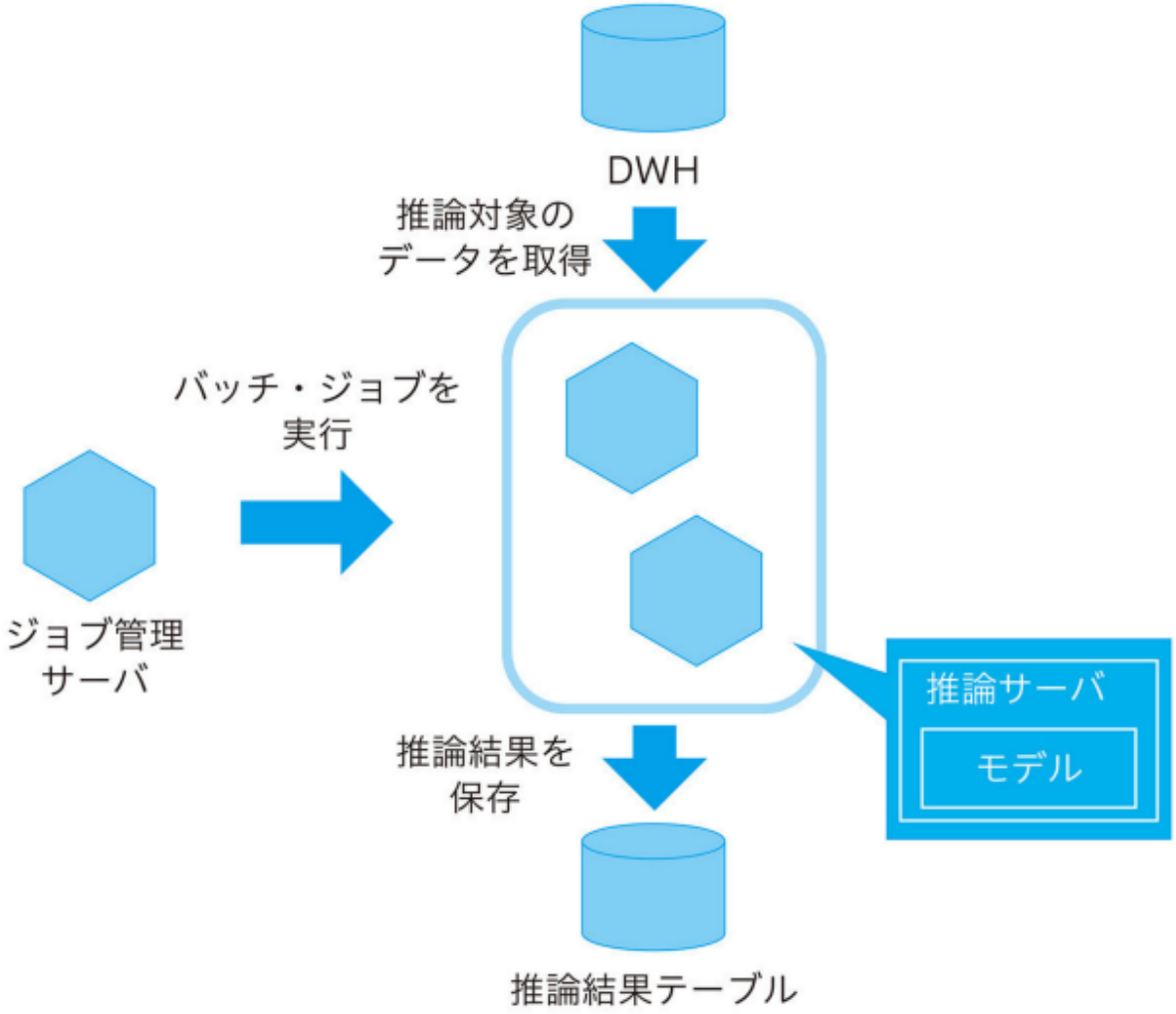

図4.11 バッチ推論パターン

- 概要
    - 中間にキューやキャッシュを配置して、推論リクエストと推論結果の取得を非同期にする。推論とリクエストを切り離すことで、クライアントは推論結果出力先に定期的にアクセスして習得すればよい。
    - 推論結果をクライアントに直接渡すようにすることもできるが、そのためのコネクションが必要になり、システムが複雑になるため推奨されない。
- 利点
    - サーバーのリソース管理を柔軟に行うことでコスト削減が可能
    - 時間に余裕をもってスケジュールすることができれば、サーバー障害等で失敗してもリトライが可能
- 検討事項
    - 1回あたりのバッチジョブで推論対象とするデータの範囲を定義する必要がある。データ量の多寡によって計算時間がしｈ額ので推論結果がスケジュール通りに完了できるように調整する必要がある。
    - バッチ処理が失敗した場合については以下の3つの方針に従って対策を行う。

        - 全件リトライ：
          失敗した場合は対象全件データに対してリトライする。データ間の相関関係が重要な場合、一部の推論の失敗が全体の推論結果に影響を与える場合に適用する。
        - 一部リトライ：
          失敗したデータのみ再度推論する
        - 放置：
          失敗してもリトライを起動しない。次回のバッチジョブでまとめて推論する。時間経過で失敗した推論が不要な場合はデータのみならずモデルも放置する。

4.6 前処理・推論パターン

モデルファイルの中には推論用モデルを保存し、前処理はプログラムのみ用意する。推論器では前処理と推論を異なるサーバーで実行することで各サーバのメンテナンス性を高める。

- ユースケース
    - 前処理と推論でライブラリやコードベース、ミドルウェア、必要リソースが大きく違う時
    - 前処理と推論をコンテナレベルで分離することで障害の切り分けや可用性、メンテナンス性が向上する時
- 解決したい課題
    - 具体例：
      前処理ではscikit-learnやOpenCVで実装する。
      モデル推論ではTensorFlowやPytorchで実装する。

特に、ディープラーニングの場合は、TensorFlow ServingやONNX Runtime Serverのようにディープラーニングのライブラリを単独の推論器として稼働させる手法が提供されている場合がある。

- 設計イメージ

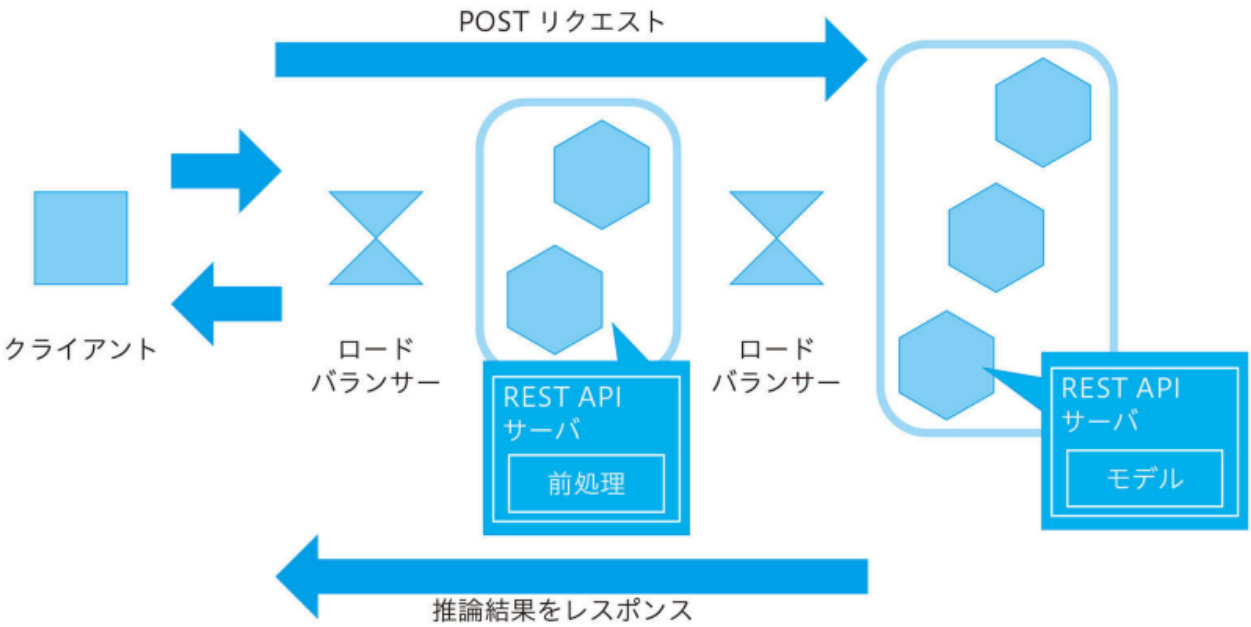

図 4.12 シンプルな前処理・推論パターン

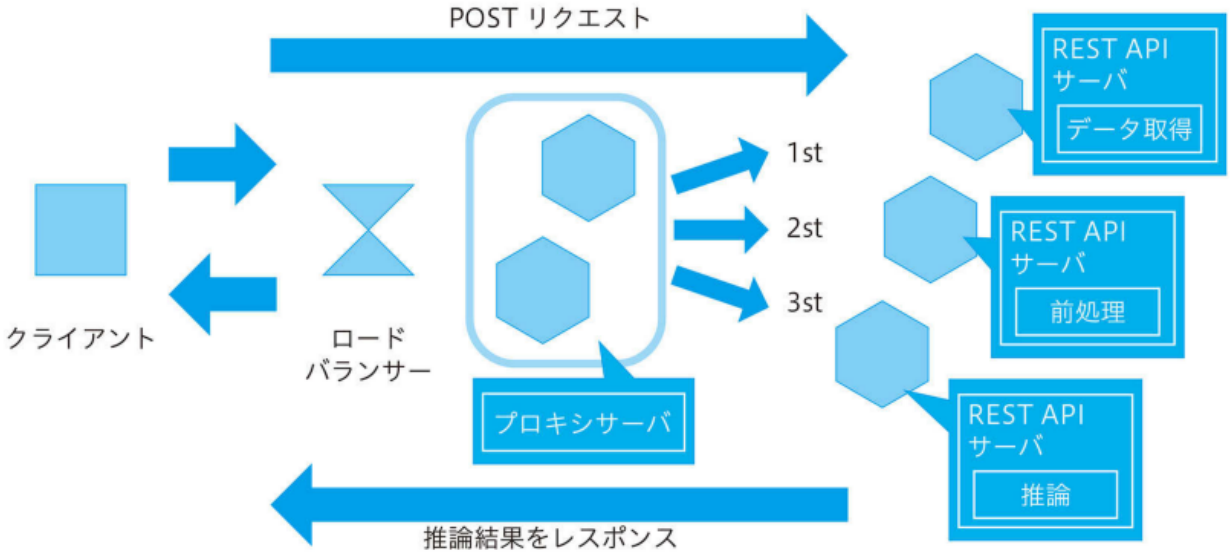

図 4.13 前段にプロキシを置く構成

- 概要
    - 前処理・推論パターンでは、クライアントから前処理サーバーにリクエストを送る。
    - 前処理サーバーではデータを変換した後、推論器をリクエストして推論結果を取得し、クライアントへレスポンスする。
    - 複数のサーバーを前処理側と推論側に置く場合には各サーバー間やクライアントとの間にロードバランサーを挟む。
    - 前段にプロキシを配置して前処理と推論をマイクロサービス化するパターンも可能。この構成ではデータの習得サーバ、前処理サーバ、推論サーバを独立したライブラリやコードベース、リソースで管理できる。ただし、コンポーネントが増えるので、コードベースやバージョン管理、障害対応が難しくなる。
- 利点

    - 前処理と推論器でサーバやコードベースを分割することで、リソースの効率化や障害時の切り分けが可能になる。
    - それぞれでリソースの増減を実装することができる。
    - 使用するライブラリのバージョンを前処理と推論で独立して選択することができる。

- 検討事項

    - 前処理と推論器を分離しても、前処理とモデルの学習時のものをセットで使用する必要がある。よって、リリースするときはバージョンの一致名でのの処理が必要になる。

4.7 直列マイクロサービスパターン

クラウド登場以降、サービスを構成する機能を小さな独立したサービスに分割するマイクロサービスアーキテクチャが普及している 。サービスの疎結合化、独立化を図り、個々のサービスが最適なライブラリやプログラミング言語で開発することが可能になっている。機械学習の推論においてもその方法論を取り込んだものがある。

- ユースケース

    - 複数の推論器で構成されるシステムで推論器官に依存関係がある時
    - 複数の推論器で構成されるシステムで推論の実行順が決まっている時

- 解決したい課題

    - 一つの入力データに対して複数の推論器を組み合わせて一つの推論とするワークフローなど
    - (佐藤コメント)具体例：
        - BERTで取り出した特徴量に対して決定木モデルを利用する場合など 複数の推論モデルが依存関係や順番に実行すｒ必要がある用途は多々あるが、推論器のサイズが大きくなりがちで、運用効率が下がる可能性がある。

- 設計イメージ

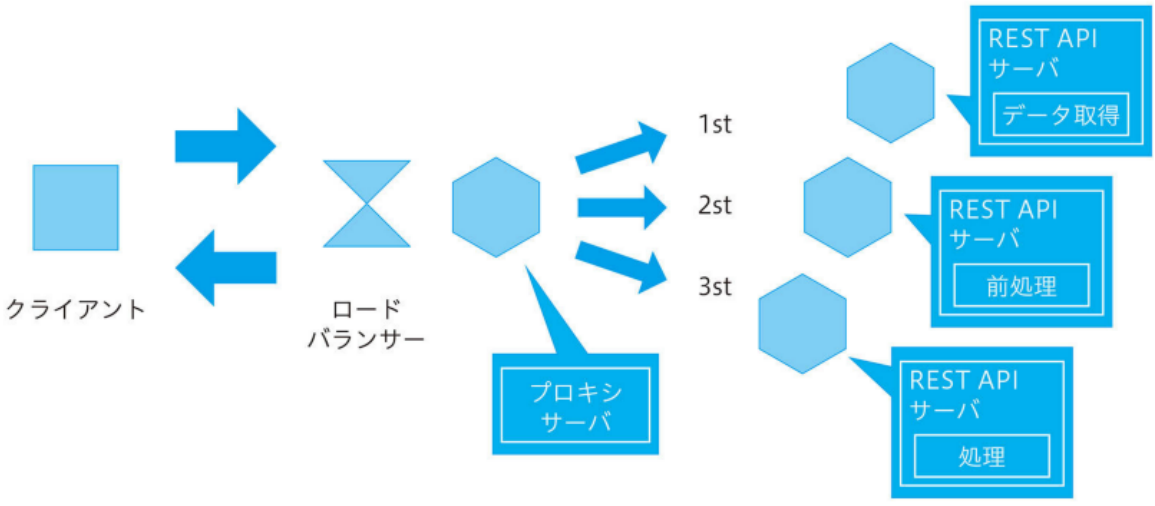

図4.15 直列マイクロサービスパターン

- 概要

    - 依存関係のある複数の推論モデルを個々の推論器として配置し、推論を数珠つなぎにしてワークフローを完成させる。
    - クライアントと推論器の間にはプロキシを仲介させて各推論器へのリクエストを行う。
    - 推論器はプロキシからリクエストを受けるマイクロサービスとして配置されるので、書く推論器の更新は柔軟にほかの推論器に影響を与えることなく実行可能。
    - ただし、前処理・推論パターン同様、推論モデルがほかの推論モデルに依存して学習している場合には、同時に更新が必要になる。
    - プロキシのリクエスト先になる推論器のエンドポイントは環境変数で設定できるのにしておくと運用上便利になる。
    - プロキシを配置することで全リクエストを共通にコントロールすることができるため、リクエスト経路の変更を柔軟に行うことができる。(特定の推論が失敗したケースではその推論を飛ばして次の推論ステップに自動で移行するなど)

- 利点

    - 各推論モデルを順番に実行することが重要
    - 前の推論モデルの結果次第で次のモデルへの推論リクエスト先を選択する構成も可能
    - 各推論でサーバのやコードベースを分割することでリソースの効率化や障害の切り分けが可能に
    - 各推論器毎のスケールアウトや更新が可能になる。

- 検討事項

    - 各推論器のｈ蘇陽時間の合計がクライアントへのレスポンスの所要時間になる。ここの推論器の推論が早くても、サービス全体として遅くなる可能性を考慮する必要がある。
    - データ通信速度についても検討が必要(スピード優先されるサービスには向いていない)
    - システムの構成が複雑になる
    - システムのエラーが発生したときにはソフトウェアエラー(サーバの障害)なのか、推論モデルのエラー(推定結果が間違っている)のか見極めが必要。

4.8 並列マイクロサービスパターン

個別の推論リクエストを送り、パターンの異なるモデルの推論結果を集約させる

- ユースケース

    - 依存関係のない推論器の推論を並列で実行するとき
    - 複数の推論結果を最後に集計するワークフローのとき
    - １データに対して複数の推論結果が必要なとき

- 解決したい課題

    - 一つのデータに対して分類と回帰で別々に推論結果を取得して異なる用途で使いたい場合(複数の分析を同時に行いたいという場合)
    - 同一の分析を別のモデルについて並列で実行してその結果を集約するなどしたい場合

- 設計イメージ

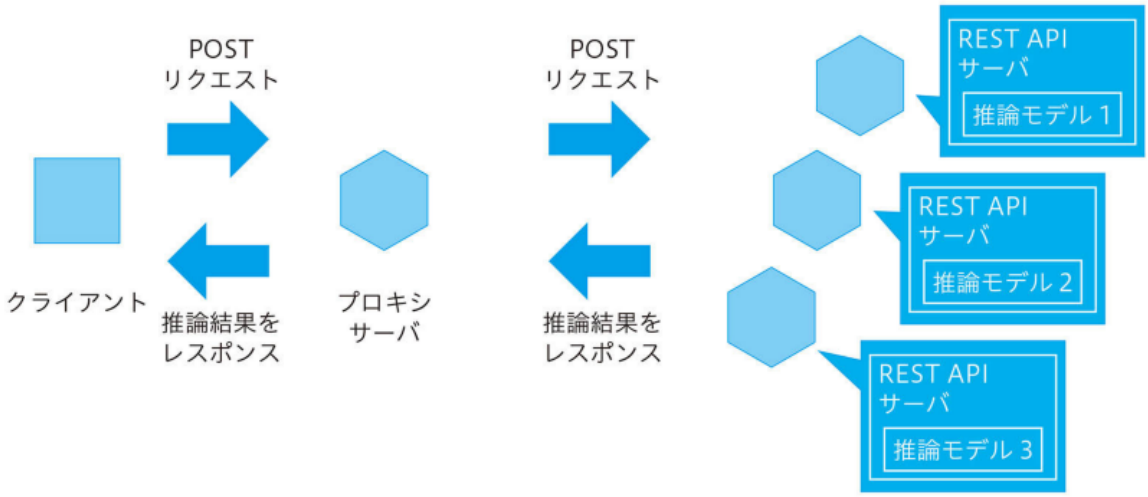

図4.16 並列マイクロサービスパターン

- 概要
  - 各推論器に同時に推論リクエストを送信し、複数の推論結果を取得する。
  - クライアントと推論器の間に仲介する独自のプロキシを置いてデータの取得や推論結果の集約タスクをクライアントから隔離する。
  - データの習得はプロキシで一括で収集すること、各推論サーバーで取得することの両方が考えられる。前者はオーバーヘッドの削減、後者は各モデルがデータの取得までやることで複雑なワークフローを実現することができる。

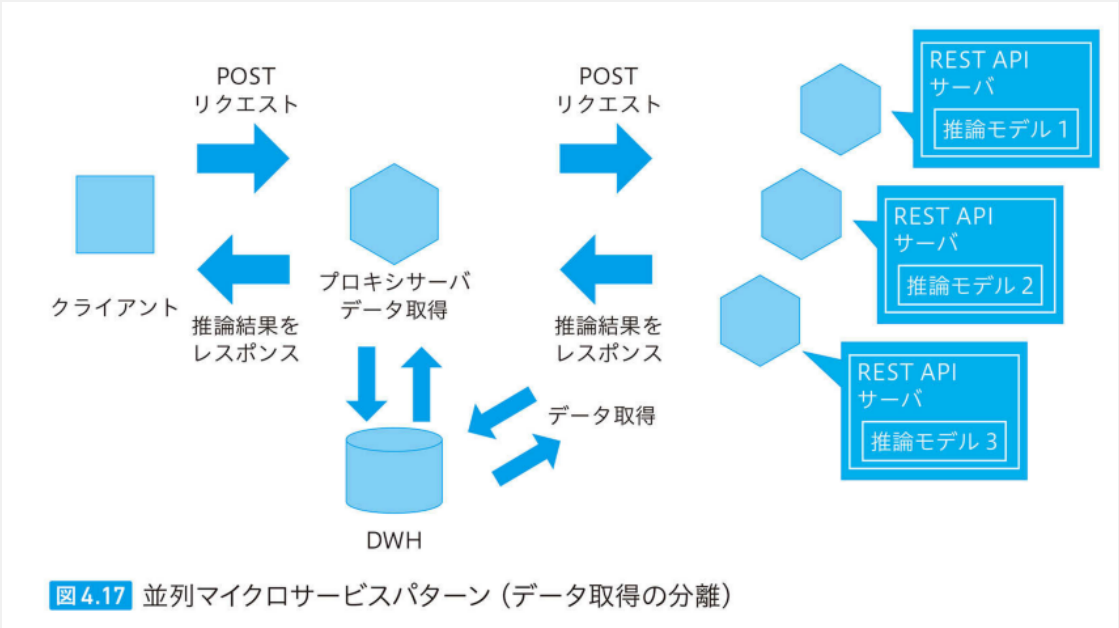

図4.17 並列マイクロサービスパターン（データ取得の分離）

- 利点
  - プロキシを仲介させるメリットは、機械学習の推論結果に応じてクライアントへのレスポンスを制御できることにある。
  - 推論器の追加、削除を柔軟に制御できるので、柔軟性や運用の容易性が格段に上がる。
  - 推論サーバーを分割することでリソースの調整や障害切り分けが可能。
  - 各推論サーバの更新等が容易

- 検討事項
    - 同期的な推論パターンにする場合、複数の推論器で推論をする性質上、計算速度の遅い推論器に引っ張られる形に全体の推論速度が決まる。
    - タイムアウトの設定戦略パターンを決める必要がある。推論器全体に制限時間をかけるパターンと、ここの推論器にタイムアウトを設定する方法の二つ

4.9 時間差推論パターン

レイテンシーに差がある推論器を有効活用する

- ユースケース
    - インタラクティブなアプリケーションに推論器を組み合わせたい
    - レスポンスが早い推論器と遅い推論器を組み合わせたワークフローを作成する場合
- 解決したい課題
    - 複数の推論器を使用する際に問題になる、レイテンシーの差に対応したい
    - 計算時間の短い推論器は同期的にレスポンスさせ、遅い推論器は非同期で推論させたい
- 設計イメージ

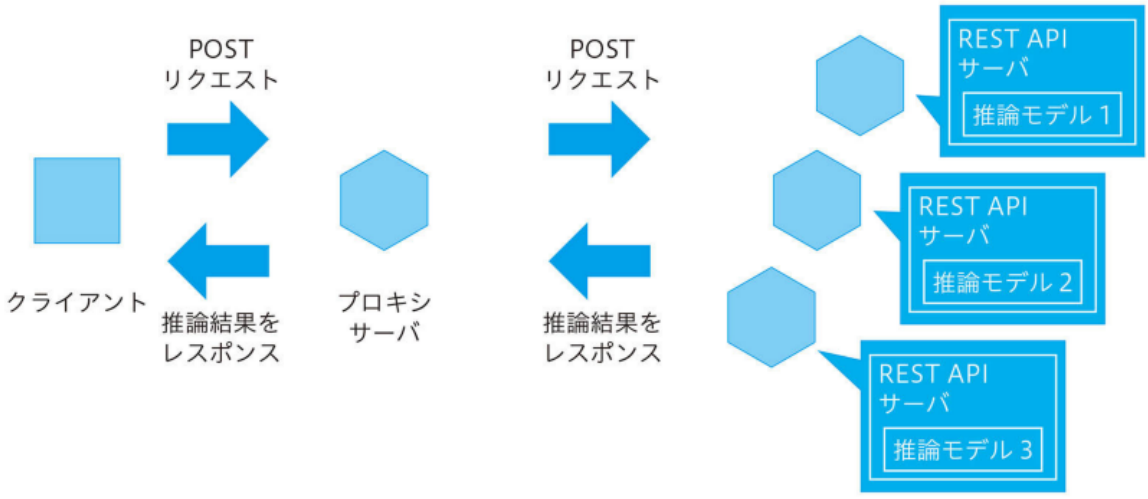

図4.16 並列マイクロサービスパターン

- 概要
    - 2種類の推論器を用意する
        - 素早く同期的に推論結果をレスポンスする推論器についてはREST APIやgRPC等をインターフェースする。
        - 非同期で推論については非同期な処理ができるメッセージングやキューを仲介する。

参考文献
fast api についての解説記事 https://qiita.com/bee2/items/75d9c0d7ba20e7a4a0e9

ONNX Runtimeについての解説記事 https://tech-blog.optim.co.jp/entry/2018/12/05/160831