PyDAAL 超入門: パート 4 分散処理とオンライン処理

この記事は、インテル® デベロッパー・ゾーンに公開されている「Gentle Introduction to PyDAAL: Vol 4 Distributed and Online Processing」の日本語参考訳です。

概要

パート 3 では、バッチ処理環境でインテル® Distribution for Python* (IDP) とインテル® データ・アナリティクス・アクセラレーション・ライブラリー (インテル® DAAL) を使用した場合の、予測モデル・フィッティングとデプロイメント・プロセスのさまざまな段階を紹介しました。パート 4 では、インテル® DAAL のそれ以外の処理モードについて、訓練段階の高速化に注目して詳しく見ていきます。この記事では、インテル® DAAL と一般的なデータ・アナリティクス・ライブラリーとを区別する 2 つの計算モード、分散処理モードとオンライン処理モードについて説明します。

モデルの訓練プロセスを高速化するため、インテル® DAAL は大規模なデータセット向けに**分散処理モード**をサポートしており、マスタースレーブ方式の実装を容易にするプログラミング・モデルを提供しています。インテル® DAAL のシリアル化クラスと逆シリアル化クラスは並列計算中にノード間でデータのやり取りを可能にします。そのため、Mpi4py は PyDAAL (インテル® DAAL の Python* API) と簡単に連携することができます。

モデルの再訓練プロセスを高速化し、計算リソースの制限に対応するため、インテル® DAAL は**オンライン処理モード**を提供しています。

超入門シリーズ

- **パート 1: データ構造** - インテル® DAAL のデータ管理コンポーネントとカスタムデータ構造 (数値テーブルとデータ辞書) をサンプルコードを使用して紹介します。
- **パート 2: 数値テーブルの基本操作** - インテル® DAAL のカスタムデータ構造 (数値テーブルとデータ辞書) で実行可能な操作について、サンプルコードを使用して紹介します。
- **パート 3: 解析モデルの構築とデプロイメント** - バッチ処理でシリアル化されたデプロイメントを利用するインテル® DAAL の解析モデリングと評価プロセスを紹介します。
- **パート 4: 分散処理とオンライン処理** - 大規模なデータやストリーミング・データのデータ分析やモデル・フィッティングをサポートするインテル® DAAL の高度な処理モード (分散とオンライン) を紹介します。

インテル® Distribution for Python* とインテル® DAAL のインストール

この記事のデモには、インテル® Distribution for Python*、インテル® DAAL、mpi4py (Anaconda* Cloud から無料で入手可能) が必要です (インテル® Parallel Studio XE 2019 ではパッケージに同梱されています)。

1. 必要なパッケージをインストールするため、インテル® Distribution for Python* の完全な環境をインストールします。

   ```
   conda create -n IDP -c intel intelpython3_full python=3.6
   ```

2. インテル® Distribution for Python* 環境をアクティベートします。

   ```
   source activate IDP
   ```

   または

   ```
   activate IDP
   ```

詳細は、インストール・オプションとインテルのパッケージの一覧 (英語) を参照してください。

1. PyDAAL と MPI による分散学習

1.1 背景

近年、一般的なマシンラーニング・アルゴリズムは、マシンラーニング実務者の作業を容易にするため単純なツールキットにパッケージ化されています。これらのツールキットのほとんどのライブラリーは、バッチ処理と呼ばれる逐次アルゴリズム計算を実行します。このタイプの処理は、ビッグデータを扱う場合に問題となります。バッチ処理モードの計算がビッグデータで単一のモデル結果を生成するのに時間がかかる場合、パラメーター・チューニングはほぼ不可能です。この制限に対処するため、インテル® DAAL はデータ・サイエンス・コミュニティーの標準的なプラクティスに対応する「分散処理」を提供しています。

予測的解析では、PyDAAL と mpi4py を使用して、Single Program Multiple Data (SPMD) 手法により、多くのインテル® DAAL のアルゴリズム実装でモデルの訓練を素早く分散できます。ほかの Python* マシンラーニング・ライブラリーは、バッチ・パラメーター・チューニング・グリッド検索を簡単に適用できます。これは主に、驚異的な並列性 (Embarrassingly Parallel) を備えたプロセスであるためです。インテル® DAAL の特徴は、多くのモデル訓練クラスの IA 向けに最適化された分散バージョンが含まれており、それらが高速でスケーラブルな訓練結果をもたらして、大規模なデータセットのパラメーター・チューニングを高速化します。さらに、バッチ学習と同様の構文で単一のモデルの訓練を高速化できます。これらの実装においてインテル® DAAL エンジニ

アリング・チームは、スレーブが行でグループ化したデータチャンクの部分訓練結果を計算し、マスターが部分結果を最終モデル結果にリダクションするマスタースレーブ方式を提供しています。

1.1.1 シリアル化とメッセージパッシング

MPI4Py で渡されるメッセージは、シリアル化されたオブジェクトとして渡されます。MPI4Py は、この処理の実行中に内部で一般的な Python* オブジェクト・シリアル化ライブラリーの Pickle を使用しています。PyDAAL は、ラッパー・インターフェイスとして SWIG (Simplified Wrapper and Interface Generator) を使用しています。残念ながら、SWIG は Pickle と互換性がありませんが、幸いにも、インテル® DAAL には、ビルトインのシリアル化と逆シリアル化機能があります。詳細は、パート 3 の「訓練済みモデルの移植性」セクションを参照してください。次の表は、PyDAAL の共分散モデルメソッドの分散バージョンにおけるマスタースレーブ方式を示しています。

1.2 バッチ処理と分散処理の概要

バッチ訓練	分散訓練
1. 訓練オブジェクトを初期化 `algo= training.Batch()`	1. 訓練オブジェクトを初期化 `algo = training.Distributed()`
2. アルゴリズム入力パラメーターを設定 `algo.input.set(training.<InputID>, data)`	2. アルゴリズム入力パラメーターを設定 `algo.input.set(training.<InputID>, data)`

	3. すべてのスレーブノードの部分結果を計算
	`algo.compute ()`
	`partialResult = algo.getPartialResult ()`

3. 訓練結果を計算して取得
`algo.compute()`
`trainingResults=algo.getResult()`
	4. すべての部分結果を加えてマスターノードの最終的な結果を計算
	`algo.inpute.add(training.partialModel, partialResult)`
	`algo.finalizecompute()`
	`trainingResults=algo.getResult()`

1.3 インテル® DAAL で利用可能な分散処理アルゴリズム

教師ありアルゴリズム:

1. 線形回帰およびリッジ回帰
2. ナイーブベイズ分類器
3. 推薦システム

4. ニューラル・ネットワーク

教師なしアルゴリズム:

1. K 平均法
2. 主成分分析

その他の分析

1. モーメント、低次
2. 共分散行列
3. 特異値分解
4. QR 分解

今後のリリースでは、分散処理に対応したアルゴリズムがさらに増える予定です。

1.4 インテル® DAAL の詳細な分散処理ワークフロー

段階 1: スレーブノードの部分訓練

```
from daal.algorithms.covariance
import (
Distributed_Step1LocalFloat64DefaultDense,
data, partialResults)


# スレーブ計算ルーチンを定義
def computestep1Local(serialnT):


    # デフォルトメソッドを使用して分散処理の密分散共分散行列を
    計算するアルゴリズム・オブジェクトを作成
    model = Distributed_Step1Local()

    # ヘルパー関数を使用して逆シリアル化
    partialnT = deserialize(serialnT)


    # アルゴリズムに適した入力データを設定
    model.input.set(data, partialnT)


    # ローカルノードの部分推定結果を計算
    partialResult = model.compute()


    # 計算した部分推定結果を取得
    serialpartialResult = serialize(
       partialResult)


    return serialpartialResult
```

段階 2: マスターノードの処理

```
from daal.algorithms.covariance
import
Distributed_Step2MasterFloat64DefaultDense



# マスター計算ルーチンを定義
def computeOnMasterNode(
  serialPartialResult, size)


  # デフォルトメソッドを使用して分散処理の密分散共分散行列を
  計算するアルゴリズム・オブジェクトを作成
  model = Distributed_Step2Master()

  for i in range(size):
  # ヘルパー関数を使用して逆シリアル化
  partialResult = deserialize(
     serialPartialResult[i])

  # アルゴリズムに適した入力データを設定
  model.input.add(partialResults,
     partialResult)

  # ローカルノードの部分推定結果からマスターノードの部分
  推定結果を計算
  model.compute()

  # 分散処理モードの最終的な結果を計算
  finalResult = mode-l.finalizeCompute()

  # 計算した密分散共分散行列を取得

  return finalResult
```

注: serialize と deserialize ヘルパー関数は、パート 3 の「訓練済みモデルの移植性」にあります。または、daaltces の GitHub* リポジトリー (英語) から customUtils をインポートできます。

1.5 共分散行列の分散処理のデモ

このデモは、Linux* 向けに設計されています。

ヘルパー関数: 共分散行列

注: ここで使用するヘルパー関数には、「customUtils」モジュールが必要です。「customUtils」は daaltces の GitHub* リポジトリー (英語) から入手できます。

以下のコードをコピーしてユーザーのスクリプトに貼り付けたり、特定のユースケースに合わせて変更できます。またヘルパー関数ブロックは、共分散行列の計算の分散処理に使用できます。さらに、その他のタイプのモデルに合わせて変更することもできます。分散モデル・フィッティングに関する詳細は、開発者ドキュメントの「Computation Modes (計算モード)」(英語) セクションを参照してください。ヘルパー関数の後にコードの使用例が続きます。

ヘルパー関数:

```
'''
------------------------------------------------------------------------------
----
*************************HELPER FUCNTION STARTS
HERE****************************
------------------------------------------------------------------------------
----
'''
# Define slave compute routine

'''
Defined Slave and Master Routines as Python Functions
Returns serialized partial model result. Input is serialized partial numeric
table
'''
from customUtils import getBlockOfNumericTable, serialize, deserialize #
customUtils is available on daaltces GitHub page
https://github.com/daaltces/pydaal-getting-started/tree/master/3-custom-
modules/customUtils.
from daal.data_management import HomogenNumericTable
from daal.algorithms.covariance import (
    Distributed_Step1LocalFloat64DefaultDense, data, partialResults,
    Distributed_Step2MasterFloat64DefaultDense
)


def computestep1Local(serialnT):
   # Deseralize using Helper Function
   partialnT = deserialize(serialnT)
   # Create partial model object
```

```
    model = Distributed_Step1LocalFloat64DefaultDense()
    # Set input data for the model
    model.input.set(data, partialnT)
    # Get the computed partial estimate result
    partialResult = model.compute()
    # Seralize using Helper Function
    serialpartialResult = serialize(partialResult)

    return serialpartialResult

# Define master compute routine
'''
Imports global variable finalResult. Computes master version of the model and
sets full model result into finalResult. Inputs are array of serialized
partial results and MPI world size
'''
def computeOnMasterNode(serialPartialResult, size):
    global finalResult
    # Create master model object
    model = Distributed_Step2MasterFloat64DefaultDense()
    # Add partial results to the distributed master model
    for i in range(size):
        # Deseralize using Helper Function
        partialResult = deserialize(serialPartialResult[i])
        # Set input objects for the model
        model.input.add(partialResults, partialResult)
    # Recompute a partial estimate after combining partial results
    model.compute()
    # Finalize the result in the distributed processing mode
    finalResult = model.finalizeCompute()
'''
-------------------------------------------------------------------------------
----
************************HELPER FUCNTION ENDS
HERE*****************************
-------------------------------------------------------------------------------
----
'''
```

使用例: 共分散行列

次の例は、上記のヘルパー関数の計算ブロックを使用して、共分散行列を構築するため mpi4py で computestep1Local() 関数と computeOnMasterNode() 関数を実装します。

```
from mpi4py import MPI
from customUtils import getBlockOfNumericTable, serialize, deserialize
# customUtils is available on daaltces GitHub page
https://github.com/daaltces/pydaal-getting-started/tree/master/3-custom-
modules/customUtils.
from daal.data_management import HomogenNumericTable

'''
Begin MPI Initialization and Run Options
'''
# Get MPI vars
size = MPI.COMM_WORLD.Get_size()
```

```
rank = MPI.COMM_WORLD.Get_rank()
name = MPI.Get_processor_name()

# Initialize result vars to fill
serialPartialResults = [0] * size
finalResult = None

'''
Begin Data Set Creation

The below example variable values can be used:
numRows, numCols = 1000, 100

'''
# Create random array for demonstration
# numRows, numCols defined by user
seeded = np.random.RandomState(42)
fullArray = seeded.rand(numRows, numCols)

# Build seeded random data matrix, and slice into chunks
# rowStart and rowEnd determined by size of the chunks
sizeOfChunk = int(numRows/size)
rowStart = rank*sizeOfChunk
rowEnd = ((rank+1)*sizeOfChunk)-1
array = fullArray[rowStart:rowEnd, :]
partialnT = HomogenNumericTable(array)
serialnT = serialize(partialnT)


'''
Begin Distributed Execution
'''

if rank == 0:

   serialPartialResults[rank] = computestep1Local(serialnT)

   if size > 1:
      # Begin to receive slave partial results on the master
      for i in range(1, size):
         rank, size, name, serialPartialResults[rank] =
                              MPI.COMM_WORLD.recv(source=MPI.ANY_SOURCE,
tag=1)

   computeOnMasterNode(serialPartialResults,size)

else:
   serialPartialResult =  computestep1Local(serialnT)
   MPI.COMM_WORLD.send((rank, size, name, serialPartialResult), dest=0,
tag=1)


'''
----------------------------------------------------------------
LINUX shell commands to run the covariance matrix usage example
----------------------------------------------------------------
'''
```

```
# Source and activate Intel Distribution of Python (IDP) env
source ../anaconda3/bin/activate
source activate idp
# optionally set mpi environmental variable to shared memory mode
export I_MPI_SHM_LMT=shm

# Cd to script directory, and call Python interpreter inside mpirun command
cd ../script_directory
mpirun -n # python script.py
```

2. PyDAAL による漸次学習

2.1 背景

継続的なデータセットの更新やリソースの制限により、バッチモードで訓練できなくなったらどうなるのでしょうか?

漸次学習 (インテル® DAAL のオンライン処理) は、既存の訓練済みモデルを新しいデータ・インスタンスで拡張するプロセスです。次のようなシナリオで広く使用されています。

1. インメモリー・リソースの制限により大きなデータセットをロードできません。このような場合、データセットをブロックに分割してロードし、モデルを漸次的に訓練します。
2. 定期的に新しいデータストリームが供給され、以前に訓練したモデルを更新する必要があります (既存のデータ・インスタンスが関連性を保持している場合)。新しいデータ・インスタンスがロードされるたびにモデル全体を再訓練するのは大変です。漸次学習アルゴリズムは、既存の訓練済みモデルの詳細を保持し、新しいデータの場合のみモデルを更新します。

ロボット工学、自動運転、株式取引などの分野は予測的解析に大きく依存しており、新しい学習経験でモデルを更新することが求められます。バッチ処理ではこのような状況に対応することはできません。また、顧客との直接的なやり取りを伴うデータ解析アプリケーション (ソーシャルメディア、オンライン・ショッピングなど) は、顧客の体験に基づいた最新の訓練済みモデルを必要とします。漸次学習は、新しいデータを受け取るたびにモデルを再訓練する時間と労力を排除して、ソリューションを素早く提供します。また漸次学習は、リソースが不足していても、ビッグデータでモデルの訓練が可能です。

2.2 バッチ処理とオンライン処理の概要

バッチ訓練	オンライン訓練
1. 訓練オブジェクトを初期化 `algo= training.Batch()`	1. 訓練オブジェクトを初期化 `algo= training.Online()`
2. アルゴリズム入力パラメーターを設定 `algo.input.set(training.<InputID>, data)`	2. アルゴリズム入力パラメーターを設定 `algo.input.set(training.<InputID>, data)`

	3. インクリメンタル計算
	`algo.compute()`
	`partialResult = algo.getPartialResult ()`

3. 訓練結果を計算して取得
`algo.compute()`
`trainingResults=algo.getResult()`
	4. 最終的な訓練結果を計算して取得
	`algo.finalizecompute()`
	`trainingResults=algo.getResult()`

2.2 インテル® DAAL で利用可能な漸次学習アルゴリズム

教師あり学習アルゴリズム:

1. 線形回帰
2. リッジ回帰
3. ナイーブベイズ

教師なし学習アルゴリズム:

主成分分析

その他の分析

1. 特異値分解
2. モーメント、低次
3. 相関と共分散

今後のリリースでは、オンライン処理に対応したアルゴリズムがさらに増える予定です。

2.3 インテル® DAAL の詳細な漸次学習ワークフロー

シナリオ 1: 制限されたインメモリー領域での訓練	シナリオ 2: 新しいデータ・インスタンスでの訓練

from daal.algorithms.<algo> import training
from daal.algorithms.<algo>.training
import data, dependentVariables
 	
# オンライン・アルゴリズム・オブジェクトを作成 	
algorithm = training.Online()
 	
# データセットを 'n' ブロックに分割 	
all_data = ['data-block-1.csv','data-block-2.csv',……,'data-block-n.csv']
 	
# すべてのデータブロックを反復して結果を訓練/更新 	
for block in all_data:
  # "block" から "inpdata"、"labels" 数値テーブルを割り当ててロードする PyDAAL の FileDataSource オブジェクトを作成
  inpdata = getBlockOfNumericTable (nT, Columns=10)
  labels = getBlockOfNumericTable (nT, Columns=[10,])
  # アルゴリズム入力パラメーターを設定
  algorithm.input.set (data, inpdata)
  algorithm.input.set (dependentVariables, labels)
  # 訓練結果を計算して更新
  algorithm.compute()
 	
# 最終的な訓練結果を計算 	
trainingResult = algorithm.finalizeCompute ()
	from daal.algorithms.<algo> import training
	from daal.algorithms.<algo>.training
	import data, dependentVariables
	
	# オンライン・アルゴリズム・オブジェクトを作成
	algorithm = training.Online()
	
	# 以前取得した部分結果 "parTrainingResult" を partialResults に設定
	parTrainingResult.setInitFlag(True)
	algorithm.setPartialResult(parTrainingResult)
	
	# 新しいインスタンスでデータセットを訓練
	newData = 'newData.csv'
	
	# "newData" から "inpdata"、"labels" 数値テーブルを割り当ててロードする PyDAAL の FileDataSource オブジェクトを作成
	inpdata = getBlockOfNumericTable (nT, Columns=10)
	labels = getBlockOfNumericTable (nT, Columns=[10,])
	# アルゴリズム入力パラメーターを設定
	algorithm.input.set (data, inpdata)
	algorithm.input.set (dependentVariables, labels)
	# 訓練結果を計算して更新
	algorithm.compute()
	
	# 最終的な訓練結果を計算
	trainingResult = algorithm.finalizeCompute ()

2.4 線形回帰のオンライン処理のデモ

注: ここで紹介するデモには、「customUtils」モジュールが必要です。daaltces の GitHub* リポジトリー (英語) からインポートできます。

準備として、3 つのデータ・パーティションを作成し、ディスクに保存します。これらのデータ・パーティションを使用して、前述の 2 つのシナリオを説明します。

```
#Create 4 random data partitions

import numpy as np
all_data = ['data-block-1.csv', 'data-block-2.csv', 'data-block-3.csv','data-block-new.csv']
for f in all_data:
    data = np.random.rand (1000, 11)
```

```
    np.savetxt(f,data,delimiter=",")
```

2.4.1 シナリオ 1: 限られたインメモリー領域での訓練

2 つのデータ・パーティションで線形回帰モデルを漸次的に訓練します。

```
import sys, os
sys.path.append(os.path.join(os.path.dirname(sys.executable),'share','pydaal_
examples','examples','python','source'))
from daal.algorithms.linear_regression import training
from daal.algorithms.linear_regression.training import data,
dependentVariables
from customUtils import getNumericTableFromCSV, \
                        getBlockOfNumericTable, serialize, deserialize#
customUtils is available on daaltces GitHub page
https://github.com/daaltces/pydaal-getting-started/tree/master/3-custom-
modules/customUtils.

from utils import printNumericTable

# Create an online algorithm object
algorithm = training.Online ()

# Create list of data blocks
all_data = ['data-block-1.csv', 'data-block-2.csv','data-block-3.csv']

# Iterate through all data blocks and train/update results
for block in all_data:
     nT = getNumericTableFromCSV (block)
     # Split nT into predictors and labels
     inpdata = getBlockOfNumericTable (nT, Columns=10)
     labels = getBlockOfNumericTable (nT, Columns=[10, ])
     # Set the algorithm input parameters
     algorithm.input.set (data, inpdata)
     algorithm.input.set (dependentVariables, labels)
     # compute partial model results
     algorithm.compute ()

# Serialize and save the partial results to disk.
# This partial result will be later used in the next usage example,
# to re-train on new instances
par_trainingResult = algorithm.getPartialResult ()
serialize (par_trainingResult, fileName="par_trainingResult.npy")
# Compute final results
trainingResult = algorithm.finalizeCompute ()
printNumericTable (trainingResult.get (training.model).getBeta (), "Linear
Regression coefficients:")
```

2.4.2 シナリオ 2: 新しいデータ・インスタンスでの訓練

シナリオ 1 で取得したシリアル化された「parTrainingResult」を新しいデータ・パーティションで再訓練します。

```
import sys, os
```

```
sys.path.append(os.path.join(os.path.dirname(sys.executable),'share','pydaal_
examples','examples','python','source'))
from daal.algorithms.linear_regression import training
from  daal.algorithms.linear_regression.training import data,
dependentVariables
from customUtils import getNumericTableFromCSV, \
    getBlockOfNumericTable, serialize, deserialize
from utils import printNumericTable
# customUtils is available on daaltces GitHub page
https://github.com/daaltces/pydaal-getting-started/tree/master/3-custom-
modules/customUtils.

algorithm_new = training.Online ()
#Deserialize and set the partial training results
par_trainingResult = deserialize(fileName="par_trainingResult.npy")
par_trainingResult.setInitFlag(True)
algorithm_new.setPartialResult (par_trainingResult)
#Create a numeric table of new data data instances
new_nT = getNumericTableFromCSV ('data-block-new.csv')
#Split new_nT into predictors and labels
new_inpdata = getBlockOfNumericTable (new_nT, Columns=10)
new_labels = getBlockOfNumericTable (new_nT, Columns=[10, ])
# Set the algorithm_new input parameters
algorithm_new.input.set (training.data, new_inpdata)
algorithm_new.input.set (training.dependentVariables, new_labels)
#Compute partial model results
algorithm_new.compute ()
#Compute final results
trainingResult_new = algorithm_new.finalizeCompute ()
printNumericTable (trainingResult_new.get (training.model).getBeta (),
"Linear Regression coefficients:")
```

`-par_trainingResult.setInitFlag(True)` は、以前に訓練したモデル結果を含めるため明示的に訓練結果フラグを設定するのに必要です。

部分結果は予測の実行には使用できません。予測/評価にアルゴリズムを適用するには、最終結果を計算する必要があります。予測プロセスと評価プロセスについては、パート 3 で説明しています。

3. まとめ

インテル® DAAL の分散処理モードとオンライン処理モードは、ビッグデータとストリーミング・データによって課されるさまざまな課題に対応します。インテル® DAAL は、必要な処理に応じて柔軟な実装を提供します。急速に大量のデータを扱わなければならない今日、インテル® DAAL は高速で優れたソリューションを提供できます。

パート 4 では、インテル® DAAL のさまざまな処理モード、予測的解析におけるそれらの重要性、およびモデルを訓練するための実装を説明しました。また、使用例を用いてインテル® DAAL のこれらの計算モードがバッチ処理の拡張であることを示しました。

4. その他の関連リンク

- PyDAAL 超入門: パート 1 データ構造
- PyDAAL 超入門: パート 2 数値テーブルの基本操作
- PyDAAL 超入門: パート 3 解析モデルの構築とデプロイメント
- インテル® DAAL デベロッパー・ガイド (英語)
- PyDAAL GitHub* チュートリアル (英語)

コンパイラーの最適化に関する詳細は、最適化に関する注意事項を参照してください。