型チェックのアノテーションによる保守・運用の改善

橋本 順之

動的型付け言語で行列やテンソルの次元やデータ型をコード中にアノテーションとして付与し検証をする方法を提案する．機械学習のソフトウェアの保守運用の効率化のため API や関数のインターフェースのチェックをし，レビューしやすいコードにすることが必要である．しかし，行列やテンソルの型や次元の検証は難しいという問題がある．このことについても論じる．本提案により関数やブロック単位で検証可能なインターフェースのコードを追加し，問題の改善を行う．

1 はじめに

1.1 目的

機械学習ソフトウェアの検証は難しく保守運用を困難にしている状況を考察し，問題を確認する．問題の改善のために API や関数のインターフェースのチェックをし，レビューしやすいコードにすることが必要である．しかし，行列やテンソルの型や次元の検証は難しいという問題がある．本提案は関数やブロック単位で行列やテンソルの型や次元について検証可能なコードをコメントとして記述し，それを検証することで問題の改善する行うことを目的としている．

1.2 機械学習ソフトウェアの特性と保守運用の問題確認

以下で機械学習ソフトウェアと保守運用で抱えている問題について確認をする．

機械学習ソフトウェアは研究目的やアルゴリズムの開発に重点を置いているため，開発速度を優先し動的型付け言語が好まれる傾向がある．特に深層学習ではその傾向は顕著である．静的型付け言語は動的型付け言語に比べコンパイルに時間がかかる傾向があり，コーディングと実行の繰り返しを頻繁に行い開発するケースでは動的型付け言語のほうが優位であろうと推測される．開発者が完全に利用しているライブラリやコードを理解している限りにおいてこの利点は強調される．

一方保守運用では状況は変わる．運用は日々同じソフトを動作させ，機械的な故障や平常時よりたまたま多いデータや不正なデータがくることにより，期待する動作をしない場合に，問題の原因を切り分け，一次対応のため問題を一時的に手作業で取り除きソフトを再実行をするなど，その場限りの対処を行い，そのしばらく後に恒久対応として不具合を修正する．

保守はソフトの機能の向上，セキュリティ上の要件，ハードウェアや OS の切り替えの必要性により，利用するライブラリや処理系のバージョンをあげたり，その結果本来の機能が動作しない場合の修正する作業がある．その作業の前後で同じ動作をすることが期待されるが，関数名の変更や仕様そのものの変更があり，実際にはうまくいかないので，不整合が起こる原因を特定する必要がでてくる．こちら [2] に Tensorflow のバージョンアップに必要な項目があるが関数の名前の変更や処理系により浮動小数点の挙動の違いもありバージョンアップは容易な作業ではない．しかし，開発時と同様のリソースをかけることが

Using type annotations to improve maintenance and operations.

Junji Hashimoto, グリー株式会社, Service Installation 2 Team, GREE, Inc..

できないので効率的な解決が求められる．

保守及び運用で問題の箇所の特定にはデバッグ手法が適用が有効である．デバッグは問題の種類の特定する．アルゴリズムのバグか，データのバグ（データに本来予期していないものがある場合），リソースのバグ（メモリリーク）か切り分ける．次に問題の場所を特定するために分割統治法で問題のコードの場所を特定し，原因を注意深く観察しコードを直して問題が改善するかチェックし，問題の解決を行う．

運用や保守では開発者ではない第 3 者がコードレビューを行うことが多く，変更の影響が把握できるレビューしやすいコードが重要である．静的型付け言語では入力と出力のデータが一目でわかるように記述可能であるが，動的型付け言語では関数やドキュメントを読むだけで動作を予測することが困難であり，どのような入出力を期待しているかはコードの中を読む他はなく、API やインターフェースの検証が困難である．

IT システムのソフトウェアではクラスをデータの単位とするため，Java や Python [5] がもつ型システムによってデータの不整合を検知しやすいが，機械学習ソフトウェアは型で簡単に解決できない．機械学習ソフトウェアは行列やテンソルの数値計算を主に行う傾向があるが，静的型付け言語であっても整数や浮動少数点の型のチェックは行えても，行列やテンソルの次元のチェックは行えないため，学習のデータやモデルの検証が困難という問題がある．

機械学習のソフトウェアは多数かつ高次元のデータ（ベクトルやテンソル）を扱う．IT システムのソフトウェアと違いフラットなデータ構造であり次元の異なるデータを扱うことが多く，クラスの中に小さいクラスを多数含むようなデータ構造を機械学習への入力のソースとして使うことはない．機械学習のソフトウェアはベクトルやテンソルを扱うため型システムで十分なチェックができない．例えば，Java などがもつ型システムはベクトルの長さをチェックしない．Python では動的型付け言語の利点を生かして，スカラの通しの演算，ベクトル通しの演算，スカラとベクトル混在の演算に対して，”＋”や”-”のようなシンプルな演算子を流用できる．開発時にはこの特徴は簡素にアルゴリズムを記述するのに便利なものの意図せぬ記述をした場合に脆弱である．例えば，ベクトルとベクトルを足すべき演算で，誤って片方のベクトルにスカラをいれてしまっても間違いに気づくことはできない．スカラやベクトルだけでなくテンソルといった高次元のデータでは問題は深刻なものになる．

保守運用だけでなく開発では後の行程ほど費用がかかる．学習のモデルの作成が数日でできたとしても, モデルのトレーニングには数週間かかることもあり，後の行程になるほど開発の手戻りの時間や費用が大きくなる．デバッグのために高速かつ多数のハードウェアを投入するのも限界がある．

API やインターフェースの検証が困難なことにより保守運用が困難という問題に対して，レビュー可能なように関数やインターフェースを記述し修正結果が容易に妥当であるかどうか容易にチェックできるようにすることが保守運用のために必要である．

1.3 改善のための問題確認

機械学習ソフトウェアの扱う行列やテンソル型の検証は困難でソフトウェアのインターフェースの検証が難しいという問題があるが，ライブラリと保守対象のコードついて以下の問題に絞り，改善手法を提案する．

- ライブラリ
  - API のバージョンアップによる非互換を機械的にチェックできない．
  - API のインターフェースの仕様が容易にわからない．ドキュメントやコードを精読する他にわからない．
  - ドキュメントとコードの動作の不一致があっても検出できない．
- 保守対象のコード
  - コードの関数・ブロック単位でのレビューが難しい．

これらの原因として考えているものは次の点である。

- 動的型付け言語を使用している．
- ブロックや関数の入出力の型が必ずしも記述されていない．
- 行列やテンソルの次元のチェックが難しい．

2 関連研究

2.1 依存型による改善

行列やテンソルの次元を検証する方法として，静的型付け言語を用いるケースで依存型を用いた型システム [3] でテンソルの次元を管理する手法 [4] がある．テンソルの次元（数字のリスト）を型として扱い入出力の次元を定義することができる．依存型を用いた型システムの例を Haskell の疑似コードでソースコード 1 に示す．入力する行列の型に具体的な数字を明記し，演算結果の型と不整合がないことをコンパイラが検証する．

ソースコード 1　依存型を用いた疑似コード

```
-- 7行列の型(iとoには数字が入る)
data Weights i o :: Nat -> Nat ->
     * where
    (:*) :: KnownNat h
         => !(Weights i h)
         -> !(Weights h j)
         -> !(Weights i j)
ih :: Weights 10 7
-- 7 x 4行列の型
hh :: Weights  7 4
-- 4 x 2行列の型
ho :: Weights  4 2

-- ih x hh x hoの行列の演算をし，
-- 10 x 2行列の型となる．
ih :* hh :* O ho :: Weights 10 2
```

コンパイル時に行列の次元も含めて静的に解析でき，ユーザーに明示的に入出力の仕様を記述させることを強要でき，型にインターフェースの情報が含まれているため，ドキュメントとコードの動作の不一致の問題が起こりにくい．しかし，この方法では機械学習で使われている Python 等の動的型付け言語の資産が利用できないという問題や，実際の計算アルゴリズムと型レベルの計算の両方を記述する必要があり手間が多い．

2.2 型のアノテーションによる改善

静的型付け言語である Java だけでなく，動的型付け言語の Python や Javascript において Mypy [5] や Typescript [6] といった型のアノテーションを付与しデータの型を検証する方法がある．これらはデータ型の不整合を検出するのに役に立ち IT システムのソフトウェアではクラスをデータの単位とするため有用である．

動的的型付け言語でも Python2 から Python3 へのアップデートの例 [1] のように API のインターフェースに型を利用することは有用である．

Mypy [5] で型を記述した例をソースコード 2 に示す．整数型 (int) やクラス型 (np.ndarray) は扱えるが型に数字の値を用いることができないため，テンソルの次元を扱うことはできない．そのため，機械学習ソフトウェアで扱う行列やテンソルに対して検証をすることはできない．行列やテンソルの次元を正しく扱うには次元を型として扱うことに加え，入力の次元に依存して出力の次元を決定する必要があるが，これらの型システムでは依存関係がある型が扱えない．

ソースコード 2　型のアノテーションの例

```
#整数型のアノテーション
def int_func(x: int) -> int :
  return x
#行列型のアノテーション
def ndarray_func(x: np.ndarray):
    np.ndarray
  return x
```

3 提案手法

本提案は型情報チェックするために関数やブロックの実行可能なコメントにアノテーションをつけ問題を改善する．コメントに実行可能なコードを埋め込みテストする手法として doctest [7] があり、これを利用する．doctest はプログラムの本体とは別に実行できるため高速にテストできる．>>> の後に検証するコードを記述し，続く行に期待値を記述するものである．その例をソースコード 3 に示す．

ソースコード 3　doctest の例

```
def 関数定義(関数の引数):
  """コメントの開始
  >>> 検証するコード(1)
  上記コードの期待値(1)
  別のコメントがあってもいい.
  >>> 検証するコード(2)
  上記コードの期待値(2)
  """コメントの終了
  関数本体が続く.
```

本提案手法では検証対象の関数のドキュメントに入出力のアノテーションをつける．実際の関数の処理とは別であるので，学習の計算に悪影響をあたえない．下記に手順を述べる．

(a) 検証対象の def で始まる関数のあとのコメントに次元付きの入力テンソルと関数の実行のコード (4) を書く．

(b) 関数の出力結果の期待値照合用のコードを書く．ここで出力テンソルのデータ型と次元を書く．

ソースコード 4　型チェックのアノテーションのフォーマット

```
def 関数名(引数):
  """関数の説明
  入力ベクトルの初期化
  >>> x = 入力ベクトル生成関数
  関数の実行
  >>> v = テストする関数(x)
  >>> v.shape
  (7, 10) #出力の次元
  >>> v.dtype
  tf.float32 #出力のデータ型
  """
  関数本体が続く.
```

実際のサンプルをソースコード (5) に示す．必要であれば使用しているライブラリの関数に対しても型チェックすると検証はより効果的なものになる．

ソースコード 5　型チェックのアノテーションの例

```
def cnn_model(features,name=None
    ):
  """Model function for CNN.
  >>> batch = 7
  >>> x = tf.zeros([batch,784],
      name="x")
  >>> v = cnn_model({'x':x},"cnnt
      ")
  >>> v.shape
  (7,10)
  >>> v.dtype
  tf.float32
  """
  ...
```

本手法の意義として，本体の処理と関係なく事前に実行されるために，処理時間が短く，動作に関して矛盾のない関数のドキュメントにもなる．入出力の次元を含めた型チェックが可能となる．仕様と実際の動作が乖離しやすく，ドキュメントは保守されない場合も多いが，本手法は実際のビヘイビアがドキュメントになり，不整合は発生しない．保守運用で不具合がある場合にデバッグでは分割統治法のようなものを使いバグのある場所を発見する必要があるが，本手法では処理単位で期待する動作をしているか検証のためのアノテーションを導入できるため，デバッグにも有用である．

4　従来手法と比較と考察

従来手法として依存型を用いた行列やテンソルの次元を明記しインターフェースを検証する方法と従来型の型のアノテーションを用いたインターフェースの検証方法を挙げ，提案手法として行列やテンソルの次元をコメント中に明記し検証する方法を述べた．それらの長所と短所を下記に列挙した．

依存型を用いた方法はすべてのシステムをそれで記述できればよいが，現実的には導入は難しい．Tensorflow の例をとるとフレームワークは様々な言語をサポートしているように見えるが，実際にサポートされているのはモデルのリードとライトとそれに

入出力を与えるまでにであり，C 言語の API の上に Python で様々な処理が記述されており，例えば数値最適化のアルゴリズムは Python で記述されている．実際にフレームワーク上でモデルを構築・最適化する上で Python 以外の選択肢はない．

提案手法は Python といった動的型付け言語上で行える現実的な手法である．しかし，検証のためとはいえ独自の記述を導入しているためコードレビューについて若干の問題を抱えている．

- 依存型を用いた方法
  - 手法.
    - 行列やテンソルの次元を型にいれる．また入出力の次元の関係も型で記述する．
  - Pros.
    - テンソルの次元の型が明記されている．
    - 関数やブロックによらずあらゆるレイヤーで型がチェックされる．
    - 網羅的に検証できる．
  - Cons.
    - Python などで書かれた従来の資産が利用できない．
- 従来型の型のアノテーションを用いた方法
  - 手法.
    - Java などの通常の型システムを用いる方法．行列やテンソルの型は用意するが次元について管理しない．
  - Pros.
    - Python などで書かれた従来の資産が利用できる．
  - Cons.
    - テンソルの次元の型が検証できない．
    - 網羅的な検証ができない．
- 提案手法の型のアノテーションを用いた方法
  - 手法.
    - 行列やテンソルの型はコメント部にてアノテーションとして記述しプログラミング本体とは別に検証する．
  - Pros.
    - Python などで書かれた従来の資産が利用できる．
    - テンソルの次元の型が検証できる．
  - Cons.
    - 網羅的な検証ができない．
    - テンソルの次元をチェックしレビュー可能なコードにするためとはいえ，一般的とはいえない本手法独自の記述がある．

5 まとめと今後の課題

機械学習のソフトウェアは保守運用において API や関数のインターフェースのチェックをし，レビューしやすいコードにすることが重要である．行列やテンソルの型や次元の検証は難しいことによるチェックが不十分である．本提案は関数やブロック単位で検証可能なインターフェースのコードを追加し，保守運用の改善を行った．コードの保守性を向上させるためとはいえ，一般的とはいえない新しい記述の導入をした．より汎用的な記述や広く認められる記述への検討や検証をしてない関数の検出（検証漏れのチェック）を行うなどの方法を検討することが今後の課題である．

謝辞 本論文の議論と掲載の機会を与えていただいた機械学習工学研究会（MLSE）に感謝する．

参 考 文 献

[1] Abbott, T.: Static types in Python, oh my(py)!, 10 2016.
[2] Google: Transitioning to TensorFlow 1.0, 06 2018.
[3] Gundry, A.: Type Inference, Haskell, and Dependent Types., 2013.
[4] Le, J.: Practical Dependent Types in Haskell: Type-Safe Neural Networks, 05 2016.
[5] Lehtosalo, J.: Mypy: Optional Static Typing for Python, 2012.
[6] Microsoft: Announcing TypeScript 1.0, 04 2014.
[7] Peters, T.: docstring-driven testing, 03 1990.