

ゼロから作る

Deep Learning 2

自然言語処理編



斎藤 康毅 著

Copyright ©2018 Koki Saitoh. All rights reserved.

本書で使用するシステム名、製品名は、いずれも各社の商標、または登録商標です。
なお、本文中では™、®、©マークは省略している場合もあります。

本書の内容について、株式会社オライリー・ジャパンは最大限の努力をもって正確を期していますが、
本書の内容に基づく運用結果については責任を負いかねますので、ご了承ください。

まえがき

作れないものは、理解できない
—— リチャード・ファインマン

ディープラーニングによって、世界は大きく変わろうとしています。スマートフォンの音声認識も、Web のリアルタイム翻訳も、為替の予測も、いまやディープラーニング抜きでは語れません。新薬の開発も、患者の診断も、車の自動運転も、ディープラーニングによって現実となりつつあります。この他にも、先進的な技術の裏側には、必ずと言ってよいほどディープラーニングが隠れています。そしてこれから先も、ディープラーニングによって世界はさらに前進するはずです。

本書は、『ゼロから作る Deep Learning』の続編です。前作から引き続き、ディープラーニングにまつわる技術を扱います。特に本作では、自然言語処理や時系列データ処理に焦点を当て、ディープラーニングを使ってさまざまな問題に挑みます。そして、前作同様「ゼロから作る」をコンセプトに、ディープラーニングに関する高度な技術をじっくりと堪能していきます。

本書のコンセプト

ディープラーニングを（もしくは、何らかの高度な技術を）深く理解するには、「ゼロから作る」という経験が重要だと筆者は考えます。ゼロから作るとは、自分の理解できる地点からスタートし、できるだけ外部の既製品は使わずに目的とする技術を完成させることです。そのような経験を通じて、表面的ではなく、しっかりとディープラーニングに精通すること——それが本書の目指すところです。

結局のところ、技術を深く理解するには、それを作れるだけの知識や技量が必要になります。本書では、ディープラーニングをゼロから作ります。そのためにさまざまなコードを書き、いろいろな実験を行います。それは時間のかかる作業であり、時に頭を悩ませることもあるでしょう。しかし、そのような時間のかかる作業には——むしろ、そのような作業にこそ——、技術を深く理解する上で重要なエッセンスが多く詰まっています。そのようにして得た知識は、既存のライブラリを使うにも、最先端の論文を読むにも、オリジナルのシステムを作るにも必ず役に立つはずです。そして何より、ディープラーニングの仕組みや原理をひとつずつ紐解きながら理解することは、純粋に楽しいものです。

自然言語処理の世界へ

本書の主なテーマは、ディープラーニングによる自然言語処理です。自然言語処理とは、簡単に言ってしまうと、私たちが普段話す言葉をコンピュータに理解させるための技術です。私たちの言葉をコンピュータに理解させることはとても難しい問題であり、そして同時に重要なテーマでもあります。実際に、この自然言語処理の技術によって、私たちの生活は大きく変わりました。Web検索や機械翻訳、音声アシスタントなど、世の中に大きな影響を与えた技術の根幹には、自然言語処理の技術が使われています。

このように私たちの生活に欠かせない自然言語処理の技術ですが、この分野においても、ディープラーニングはきわめて重要な位置を占めています。実際、ディープラーニングによって、これまでの自然言語処理の性能が大きく向上してきました。たとえば、Googleの機械翻訳ひとつをとっても、ディープラーニングベースの手法によって、大きな進化を遂げたのは記憶に新しいところです。

本書では、自然言語処理や時系列データ処理に焦点を当て、ディープラーニングにおける重要なテクニックを学びます。具体的には、word2vecやRNN、LSTMやGRU、seq2seqやAttentionといった技術です。本書ではこれらの技術ができるだけやさしい言葉で説明し、実際に作ることで理解を確かなものにします。また、実験を通して、それらの可能性を実際に感じてもらいたいと思います。

本書は、ディープラーニングを中心とした自然言語処理をめぐる冒險の書です。本書は全部で8章からなりますが、それらは一連の物語のように頭から順に読むように構成されています。問題が立ちはだかり、それを解決する新しい手法を考え、そしてさらに改良を加えます。そのような流れで、自然言語処理に関するさまざまな問題

を、ディープラーニングという武器を手にひとつずつ解決していきます。そしてその冒險を通じて、ディープラーニングにおける重要なテクニックを深いレベルで習得し、そのおもしろさを体感していただきたいと思っています。

誰のための本か

本書は前作『ゼロから作る Deep Learning』の続編です。そのため、前作で学んだ知識を前提としますが、本書の1章では、前作のダイジェスト版としてニューラルネットワークの復習を行います。よって、ニューラルネットワークやPythonの知識のある方は、前作の知識がなくても本書を読み進められるように配慮してあります。

ディープラーニングを深く理解するために、本書では前作に引き続き「作る」とこと、そして、「動かす」を中心には話を展開していきます。自分の分からないものは使わない。使うものは自分が理解しているものだけ——そのようなスタンスで、ディープラーニングの世界を、そして自然言語処理の世界を探索していきます。

ここでは、「誰のための本か」ということをより明確にするために、本書で行うこと、また本書の特徴を以下に列挙します。

- 外部のライブラリに頼らずに、ゼロからディープラーニングのプログラムを実装します。
- 前作『ゼロから作る Deep Learning』の続編として、自然言語処理や時系列データ処理に使われるディープラーニングの技術に焦点を当てます。
- 実際に動くPythonのソースコードとともに、読者の手元で実験できる学習環境を提供します。
- できるだけやさしい言葉で、明確な図を多用しながら説明します。
- 数式も使用しますが、それ以上にソースコードによる説明を重視します。
- 「なぜその手法が優れているのか?」、「なぜそれでうまくいくのか?」、「なぜそれが問題なのか?」——「なぜ」を大切にします。

続いて、本書で学ぶ技術を列挙します。

- Pythonによるテキスト処理
- ディープラーニング登場以前の「単語」の表現方法
- 単語ベクトルを得るためのword2vec(CBOWモデルとskip-gramモデル)

- 大規模データの学習を高速化する Negative Sampling
- 時系列データを処理する RNN、LSTM、GRU
- 時系列データの誤差逆伝播法である Backpropagation Through Time
- 文章生成を行うニューラルネットワーク
- 時系列データを別の時系列データに変換する seq2seq
- 重要な情報に注意を向ける Attention

本書では、これらの技術を丁寧に分かりやすく説明し、実装レベルでマスターできるように話を展開していきます。また、これらの技術の説明にあたって、単に事実を列挙するようなことは行わず、一連の物語のように（つながりのあるストーリーとして）話を展開していきます。

誰のための本ではないか

「誰のための本ではないか」を述べるのも大切なことだと思います。以下に、本書で行わないことを例挙します。

- ディープラーニングについて、最新の研究に関する詳しい解説・紹介は行いません。
- Caffe、TensorFlow、Chainer など、ディープラーニングのフレームワークに関する説明は行いません。
- ディープラーニングに関する詳しい理論的な解説は行いません。
- 本書は、主に自然言語処理をテーマとします。画像や音声、または強化学習などのテーマは扱いません。

このように、本書は最近の研究や理論的な詳細をカバーしていません。しかし、本書を読み終われば、その次のステップとして、最新の論文や自然言語処理に関する最先端の手法へと進むことができるでしょう。

実装環境

本書では、Python (3 系) のソースコードを提供します。そのソースコードを用いれば、読者の手元で実際に動かすことができます。コードを読みながら考え、自分で

思いついたことを新たに実装して試すことで、知識をより確かなものにできるでしょう。なお、本書で使用するソースコードは、以下の GitHub リポジトリからダウンロードすることができます。

<https://github.com/oreilly-japan/deep-learning-from-scratch-2>

さて、本書の目標とするところは、ゼロからディープラーニングを実装することです。そのため、外部のライブラリは極力使用しないというのが方針ですが、次の 2 つのライブラリは例外として用いることにします。ひとつは NumPy、もうひとつは Matplotlib というライブラリです。これら 2 つのライブラリを用いることで、ディープラーニングの実装を効率良く進めることができます。

NumPy は数値計算のためのライブラリです。このライブラリには、高度な数学アルゴリズムや配列（行列）を操作するための便利なメソッドが数多く用意されています。本書のディープラーニングの実装においては、それらの便利なメソッドを用いて効率的に実装を進めていきます。

Matplotlib はグラフ描画のためのライブラリです。Matplotlib を用いれば、実験の結果を可視化したり、ディープラーニングの学習過程を視覚的に確認したりといったことができます。本書ではこれらのライブラリを使用して、ディープラーニングのアルゴリズムを実装していきます。

なお本書のソースコードの多くは、一般的な PC でそれほど時間を要さずに処理できるように配慮しています。しかし、一部のコードでは——特に、大きなニューラルネットワークの学習では——多くの時間が必要になります。そこで本書では、時間のかかる処理を高速化するために、GPU 上で実行できるコード（仕組み）も提供します。これには CuPy と呼ばれるライブラリを使います（CuPy については「1 章 ニューラルネットワークの復習」で説明します）。NVIDIA の GPU が使えるマシンをお持ちの方は、CuPy をインストールすることで、本書の一部のコードを GPU 上で高速に処理することができます。



本書では下記のプログラミング言語とライブラリを使用します。

- Python 3 系
- NumPy
- Matplotlib
- CuPy (オプション)

再び作る旅へ

技術が進み、簡単にコピーができる時代になりました。写真も動画も、ソースコードもライブラリも、簡単にコピーができる便利な世の中です。しかし、いくら技術が発達し生活が便利になんてても、経験はコピーできません。手を動かして作り、時間をかけて考えたという経験は、決してコピーできるものではありません。そして、そのようなコピーできないものにこそ、いつの時代も変わらない価値があるはずです。

これで前置きは終わりになります。それでは、ディープラーニングを作る旅の再開です！

表記上のルール

本書では、次に示す表記上のルールに従います。

太字（Bold）

新しい用語、強調やキーワードフレーズを表します。

等幅（Constant Width）

プログラムのコード、コマンド、配列、要素、文、オプション、スイッチ、変数、属性、キー、関数、型、クラス、名前空間、メソッド、モジュール、プロパティ、パラメータ、値、オブジェクト、イベント、イベントハンドラ、XML タグ、HTML タグ、マクロ、ファイルの内容、コマンドからの出力を表します。その断片（変数、関数、キーワードなど）を本文中から参照する場合にも使われます。

等幅太字（Constant Width Bold）

ユーザーが入力するコマンドやテキストを表します。コードを強調する場合にも使われます。

等幅イタリック（Constant Width Italic）

ユーザーの環境などに応じて置き換えなければならない文字列を表します。



ソースコードの参照先（本書の GitHub リポジトリ）を表します。



ヒントや示唆、興味深い事柄に関する補足を表します。



ライブラリのバグやしばしば発生する問題などのような、注意あるいは警告を表します。

意見と質問

本書の内容については、最大限の努力をもって検証、確認していますが、誤りや不正確な点、誤解や混乱を招くような表現、単純な誤植などに気がつかれることもあるかもしれません。そうした場合、今後の版で改善できるようお知らせいただければ幸いです。将来の改訂に関する提案なども歓迎いたします。連絡先は次のとおりです。

株式会社オライリー・ジャパン
電子メール japan@oreilly.co.jp

本書の Web ページには次のアドレスでアクセスできます。

<https://www.oreilly.co.jp/books/9784873118369>
<https://github.com/oreilly-japan/deep-learning-from-scratch-2>

オライリーに関するその他の情報については、次のオライリーの Web サイトを参照してください。

<https://www.oreilly.co.jp/>
<https://www.oreilly.com/> (英語)

目次

| | |
|--------------------------------|----------|
| まえがき | iii |
| 1章 ニューラルネットワークの復習 | 1 |
| 1.1 数学と Python の復習 | 1 |
| 1.1.1 ベクトルと行列 | 2 |
| 1.1.2 行列の要素ごとの演算 | 4 |
| 1.1.3 ブロードキャスト | 4 |
| 1.1.4 ベクトルの内積と行列の積 | 6 |
| 1.1.5 行列の形状チェック | 7 |
| 1.2 ニューラルネットワークの推論 | 8 |
| 1.2.1 ニューラルネットワークの推論の全体図 | 8 |
| 1.2.2 レイヤとしてのクラス化と順伝播の実装 | 14 |
| 1.3 ニューラルネットワークの学習 | 18 |
| 1.3.1 損失関数 | 18 |
| 1.3.2 微分と勾配 | 21 |
| 1.3.3 チェインルール | 23 |
| 1.3.4 計算グラフ | 24 |
| 1.3.5 勾配の導出と逆伝播の実装 | 35 |
| 1.3.6 重みの更新 | 39 |
| 1.4 ニューラルネットワークで問題を解く | 41 |
| 1.4.1 スパイral・データセット | 41 |
| 1.4.2 ニューラルネットワークの実装 | 43 |

| | | |
|-----------|------------------------------|-----------|
| 1.4.3 | 学習用のソースコード | 45 |
| 1.4.4 | Trainer クラス | 48 |
| 1.5 | 計算の高速化 | 50 |
| 1.5.1 | ビット精度 | 50 |
| 1.5.2 | GPU (CuPy) | 52 |
| 1.6 | まとめ | 53 |
| 2章 | 自然言語と単語の分散表現 | 57 |
| 2.1 | 自然言語処理とは | 57 |
| 2.1.1 | 単語の意味 | 59 |
| 2.2 | シソーラス | 59 |
| 2.2.1 | WordNet | 61 |
| 2.2.2 | シソーラスの問題点 | 61 |
| 2.3 | カウントベースの手法 | 63 |
| 2.3.1 | Python によるコーパスの下準備 | 63 |
| 2.3.2 | 単語の分散表現 | 67 |
| 2.3.3 | 分布仮説 | 67 |
| 2.3.4 | 共起行列 | 69 |
| 2.3.5 | ベクトル間の類似度 | 72 |
| 2.3.6 | 類似単語のランキング表示 | 74 |
| 2.4 | カウントベースの手法の改善 | 77 |
| 2.4.1 | 相互情報量 | 77 |
| 2.4.2 | 次元削減 | 81 |
| 2.4.3 | SVD による次元削減 | 84 |
| 2.4.4 | PTB データセット | 86 |
| 2.4.5 | PTB データセットでの評価 | 88 |
| 2.5 | まとめ | 90 |
| 3章 | word2vec | 93 |
| 3.1 | 推論ベースの手法とニューラルネットワーク | 93 |
| 3.1.1 | カウントベースの手法の問題点 | 94 |
| 3.1.2 | 推論ベースの手法の概要 | 95 |
| 3.1.3 | ニューラルネットワークにおける単語の処理方法 | 96 |

| | | |
|------------|-----------------------------------|------------|
| 3.2 | シンプルな word2vec | 101 |
| 3.2.1 | CBOW モデルの推論処理 | 101 |
| 3.2.2 | CBOW モデルの学習 | 106 |
| 3.2.3 | word2vec の重みと分散表現 | 109 |
| 3.3 | 学習データの準備 | 111 |
| 3.3.1 | コンテキストとターゲット | 111 |
| 3.3.2 | one-hot 表現への変換 | 114 |
| 3.4 | CBOW モデルの実装 | 115 |
| 3.4.1 | 学習コードの実装 | 119 |
| 3.5 | word2vec に関する補足 | 121 |
| 3.5.1 | CBOW モデルと確率 | 121 |
| 3.5.2 | skip-gram モデル | 123 |
| 3.5.3 | カウントベース v.s. 推論ベース | 126 |
| 3.6 | まとめ | 127 |
| 4 章 | word2vec の高速化 | 131 |
| 4.1 | word2vec の改良① | 131 |
| 4.1.1 | Embedding レイヤ | 134 |
| 4.1.2 | Embedding レイヤの実装 | 135 |
| 4.2 | word2vec の改良② | 140 |
| 4.2.1 | 中間層以降の計算の問題点 | 140 |
| 4.2.2 | 多値分類から二値分類へ | 142 |
| 4.2.3 | シグモイド関数と交差エントロピー誤差 | 144 |
| 4.2.4 | 多値分類から二値分類へ（実装編） | 147 |
| 4.2.5 | Negative Sampling | 151 |
| 4.2.6 | Negative Sampling のサンプリング手法 | 154 |
| 4.2.7 | Negative Sampling の実装 | 158 |
| 4.3 | 改良版 word2vec の学習 | 160 |
| 4.3.1 | CBOW モデルの実装 | 160 |
| 4.3.2 | CBOW モデルの学習コード | 163 |
| 4.3.3 | CBOW モデルの評価 | 165 |
| 4.4 | word2vec に関する残りのテーマ | 168 |
| 4.4.1 | word2vec を使ったアプリケーションの例 | 169 |

| | | |
|-----------|-------------------------------------|------------|
| 4.4.2 | 単語ベクトルの評価方法 | 171 |
| 4.5 | まとめ | 173 |
| 5章 | リカレントニューラルネットワーク (RNN) | 175 |
| 5.1 | 確率と言語モデル | 175 |
| 5.1.1 | word2vec を確率の視点から眺める | 176 |
| 5.1.2 | 言語モデル | 178 |
| 5.1.3 | CBOW モデルを言語モデルに? | 180 |
| 5.2 | RNN とは | 183 |
| 5.2.1 | 循環するニューラルネットワーク | 183 |
| 5.2.2 | ループの展開 | 185 |
| 5.2.3 | Backpropagation Through Time | 187 |
| 5.2.4 | Truncated BPTT | 188 |
| 5.2.5 | Truncated BPTT のミニバッチ学習 | 193 |
| 5.3 | RNN の実装 | 195 |
| 5.3.1 | RNN レイヤの実装 | 196 |
| 5.3.2 | Time RNN レイヤの実装 | 200 |
| 5.4 | 時系列データを扱うレイヤの実装 | 205 |
| 5.4.1 | RNNLM の全体図 | 205 |
| 5.4.2 | Time レイヤの実装 | 208 |
| 5.5 | RNNLM の学習と評価 | 210 |
| 5.5.1 | RNNLM の実装 | 211 |
| 5.5.2 | 言語モデルの評価 | 214 |
| 5.5.3 | RNNLM の学習コード | 216 |
| 5.5.4 | RNNLM の Trainer クラス | 219 |
| 5.6 | まとめ | 220 |
| 6章 | ゲート付き RNN | 223 |
| 6.1 | RNN の問題点 | 223 |
| 6.1.1 | RNN の復習 | 224 |
| 6.1.2 | 勾配消失もしくは勾配爆発 | 225 |
| 6.1.3 | 勾配消失もしくは勾配爆発の原因 | 227 |
| 6.1.4 | 勾配爆発への対策 | 232 |

| | | |
|------------|---------------------------|------------|
| 6.2 | 勾配消失と LSTM | 233 |
| 6.2.1 | LSTM のインターフェース | 234 |
| 6.2.2 | LSTM レイヤの組み立て | 235 |
| 6.2.3 | output ゲート | 238 |
| 6.2.4 | forget ゲート | 240 |
| 6.2.5 | 新しい記憶セル | 241 |
| 6.2.6 | input ゲート | 242 |
| 6.2.7 | LSTM の勾配の流れ | 243 |
| 6.3 | LSTM の実装 | 244 |
| 6.3.1 | TimeLSTM の実装 | 249 |
| 6.4 | LSTM を使った言語モデル | 252 |
| 6.5 | RNNLM のさらなる改善 | 259 |
| 6.5.1 | LSTM レイヤの多層化 | 260 |
| 6.5.2 | Dropout による過学習の抑制 | 261 |
| 6.5.3 | 重み共有 | 266 |
| 6.5.4 | より良い RNNLM の実装 | 267 |
| 6.5.5 | 最先端の研究へ | 272 |
| 6.6 | まとめ | 274 |
| 7 章 | RNN による文章生成 | 277 |
| 7.1 | 言語モデルを使った文章生成 | 278 |
| 7.1.1 | RNN による文章生成の手順 | 278 |
| 7.1.2 | 文章生成の実装 | 282 |
| 7.1.3 | さらに良い文章へ | 285 |
| 7.2 | seq2seq | 287 |
| 7.2.1 | seq2seq の原理 | 287 |
| 7.2.2 | 時系列データ変換用のトイ・プロblem | 291 |
| 7.2.3 | 可変長の時系列データ | 292 |
| 7.2.4 | 足し算データセット | 294 |
| 7.3 | seq2seq の実装 | 296 |
| 7.3.1 | Encoder クラス | 296 |
| 7.3.2 | Decoder クラス | 299 |
| 7.3.3 | Seq2seq クラス | 305 |

| | | |
|-------|----------------------------|-----|
| 7.3.4 | seq2seq の評価 | 306 |
| 7.4 | seq2seq の改良 | 310 |
| 7.4.1 | 入力データの反転 (Reverse) | 310 |
| 7.4.2 | 覗き見 (Peeky) | 313 |
| 7.5 | seq2seq を用いたアプリケーション | 318 |
| 7.5.1 | チャットボット | 319 |
| 7.5.2 | アルゴリズムの学習 | 320 |
| 7.5.3 | イメージキャプション | 321 |
| 7.6 | まとめ | 323 |

8 章 Attention 325

| | | |
|-------|--|-----|
| 8.1 | Attention の仕組み | 325 |
| 8.1.1 | seq2seq の問題点 | 326 |
| 8.1.2 | Encoder の改良 | 327 |
| 8.1.3 | Decoder の改良① | 329 |
| 8.1.4 | Decoder の改良② | 337 |
| 8.1.5 | Decoder の改良③ | 342 |
| 8.2 | Attention 付き seq2seq の実装 | 348 |
| 8.2.1 | Encoder の実装 | 348 |
| 8.2.2 | Decoder の実装 | 349 |
| 8.2.3 | seq2seq の実装 | 351 |
| 8.3 | Attention の評価 | 351 |
| 8.3.1 | 日付フォーマットの変換問題 | 352 |
| 8.3.2 | Attention 付き seq2seq の学習 | 353 |
| 8.3.3 | Attention の可視化 | 357 |
| 8.4 | Attention に関する残りのテーマ | 360 |
| 8.4.1 | 双方向 RNN | 360 |
| 8.4.2 | Attention レイヤの使用方法 | 362 |
| 8.4.3 | seq2seq の深層化と skip コネクション | 365 |
| 8.5 | Attention の応用 | 367 |
| 8.5.1 | Google Neural Machine Translation (GNMT) | 367 |
| 8.5.2 | Transformer | 369 |
| 8.5.3 | Neural Turing Machine (NTM) | 374 |

| | |
|---|------------|
| 8.6 まとめ | 379 |
| 付録 A sigmoid 関数と tanh 関数の微分..... | 381 |
| A.1 sigmoid 関数..... | 381 |
| A.2 tanh 関数 | 384 |
| A.3 まとめ | 386 |
| 付録 B WordNet を動かす | 387 |
| B.1 NLTK のインストール | 387 |
| B.2 WordNet で同義語を得る | 388 |
| B.3 WordNet と単語ネットワーク | 390 |
| B.4 WordNet による意味の類似度 | 391 |
| 付録 C GRU..... | 393 |
| C.1 GRU のインターフェース | 393 |
| C.2 GRU の計算グラフ | 394 |
| おわりに | 397 |
| 参考文献 | 401 |
| 索引 | 407 |

1章 ニューラルネットワークの復習

ひとつ以上の方法を知るまでは、

ものごとを理解したことにはならない。

—— マービン・ミン斯基（コンピュータ科学者、認知科学者）

本書は、前作『ゼロから作る Deep Learning』の続編です。前作に引き続き、ディープラーニングの可能性をさらに深く探索していきます。もちろん、本書においても前作同様、ライブラリやフレームワークなどの既製品は使わずに、「ゼロから作る」ことを重視します。作ることを通して、ディープラーニングに関する技術のおもしろさや奥深さを探求していきたいと思います。

本章では、ニューラルネットワークの復習を行います。つまり、前作のダイジェスト版が本章に相当します。また本作では効率性を重視して、前作での実装ルールを一部変更した点があります（たとえば、メソッド名やパラメータの持ち方など）。その点も本章で確認ていきましょう。

1.1 数学と Python の復習

まず初めに数学の復習から始めます。具体的には、ニューラルネットワークの計算に必要な「ベクトル」や「行列」などをテーマに話を進めていきます。また、ニューラルネットワークの実装にスムーズに入りていけるように、Python によるコード——特に NumPy を使ったコード——も合わせて示していきます。

1.1.1 ベクトルと行列

ニューラルネットワークでは、「ベクトル」や「行列」（または「テンソル」）がいたるところで登場します。ここではそれらの用語について簡単に整理し、本書を読み進めるための準備を行います。

まずは「ベクトル」から始めます。ベクトルとは、大きさと向きを持つ量です。ベクトルは、数が一列に並んだ集まりとして表現でき、Pythonによる実装では1次元の配列として扱うことができます。それに対して「行列」は、2次元状（長方形状）に並んだ数の集まりです。ベクトルと行列の例を示すと図1-1 のようになります。

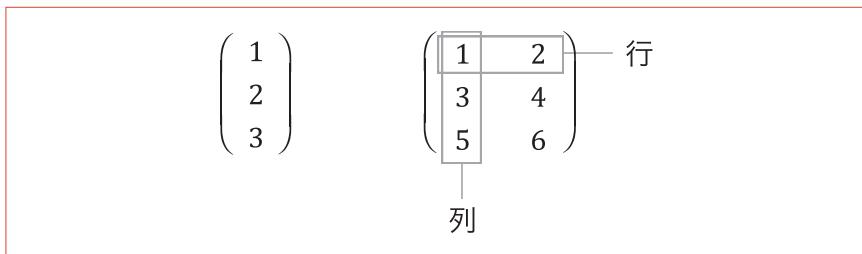


図1-1 ベクトルと行列の例

図1-1で示すように、ベクトルは1次元の配列、行列は2次元の配列で表現することができます。また行列では、横方向の並びを**行** (row) と言い、縦方向の並びを**列** (column) と言います。そのため、図1-1の行列は「3行2列の行列」と呼び、また「 3×2 の行列」と表記します。



ベクトルや行列を拡張させて、 N 次元の数の集まりを考えることもできます。これは一般的に**テンソル**と呼ばれます。

ベクトルは単純な概念ですが、ベクトルを表現するには2つの方法がある点に注意が必要です。図1-2に示すように、ひとつは縦方向の並びとして表す方法（=列ベクトル）、もうひとつは横方向の並びとして表す方法（=行ベクトル）です。

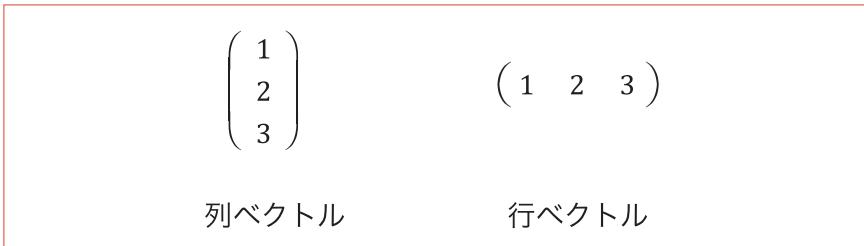


図 1-2 ベクトルの表し方

数学やディープラーニングなどの多くの分野では、ベクトルは「列ベクトル」として扱うのが一般的です。ただし、本書では実装面の親和性を考慮して、ベクトルは「行ベクトル」として扱います（その際は、行ベクトルであることを毎回明記します）。また、ベクトルや行列を数式で書く際には、**x** や **W** などの太字の記号で表すことにして、単一の要素（スカラ）とは区別して表記します（ソースコードでの変数は **W** や **x** などのフォントで記します）。



Python の実装においてベクトルを「行ベクトル」として扱う場合、ベクトルを横方向の「行列」に整形して扱うと明確になります。たとえば、要素数が N のベクトルであれば、 $1 \times N$ の形状の行列として扱うのです。具体的な例は、後ほど見ていきます。

それでは、Python の対話モードを使って、ベクトルや行列を生成してみましょう。もちろんここでは、行列を扱うときの定番ライブラリ NumPy を利用します。

```
>>> import numpy as np

>>> x = np.array([1, 2, 3])
>>> x.__class__           # クラス名を表示
<class 'numpy.ndarray'>
>>> x.shape
(3,)
>>> x.ndim
1

>>> W = np.array([[1, 2, 3], [4, 5, 6]])
>>> W.shape
(2, 3)
>>> W.ndim
2
```

ここで示したように、ベクトルや行列は `np.array()` メソッドで生成することができます。このメソッドによって、NumPy の多次元配列用のクラスである `np.ndarray` クラスが生成されます。`np.ndarray` クラスには便利なメソッドやインスタンス変数がたくさん用意されており、上の例では、インスタンス変数の `shape` と `ndim` を利用しています。`shape` は多次元配列の形状を、`ndim` は次元数を表します。上の結果を見ると、`x` は 1 次元の配列であり、それは要素数が 3 のベクトルであることが分かります。また、`W` は 2 次元の配列であり、 2×3 (2 行 3 列) の行列であることが分かります。

1.1.2 行列の要素ごとの演算

数の集まりを「ベクトル」や「行列」としてまとめることができました。それでは、それらを使って簡単な計算を行ってみましょう。ここでは初めに「要素ごとの演算」について見ていきます。ちなみに、「要素ごとの」とは英語で `element-wise` と言います。

```
>>> W = np.array([[1, 2, 3], [4, 5, 6]])
>>> X = np.array([[0, 1, 2], [3, 4, 5]])
>>> W + X
array([[ 1,  3,  5],
       [ 7,  9, 11]])
>>> W * X
array([[ 0,  2,  6],
       [12, 20, 30]])
```

ここでは NumPy の多次元配列に対して、`+` や `*` などの四則演算を行っています。このとき、多次元配列中の要素ごとで——各要素で独立して——演算が行われます。これが NumPy 配列の「要素ごとの演算」です。

1.1.3 ブロードキャスト

NumPy の多次元配列では、形状の異なる配列どうしの演算も可能です。たとえば、次のような計算です。

```
>>> A = np.array([[1, 2], [3, 4]])
>>> A * 10
array([[10, 20],
       [30, 40]])
```

この計算では、 2×2 の行列 `A` に対して、10 というスカラ値の乗算を行っています。

このとき図1-3で示すように、10というスカラ値が 2×2 の行列に拡張されて、その後で要素ごとの演算が行われます。この賢い機能は、**ブロードキャスト**（broadcast）と呼ばれます。

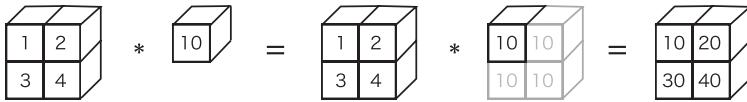


図1-3 ブロードキャストの例：スカラ値である「10」が 2×2 の行列として扱われる

ブロードキャストの別の例として、次の計算を見てみましょう。

```
>>> A = np.array([[1, 2], [3, 4]])
>>> b = np.array([10, 20])
>>> A * b
array([[10, 40],
       [30, 80]])
```

この計算では、図1-4のように、1次元配列である b が2次元配列 A と同じ形状になるように“賢く”拡張されます。

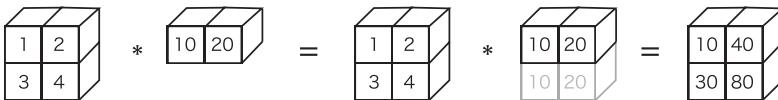


図1-4 ブロードキャストの例 2

このように、NumPy ではブロードキャストという機能があるため、形状の異なる配列どうしの演算をスマートに行うことができます。



NumPy のブロードキャストが有効に働くには、多次元配列の形状がいくつかのルールを満たす必要があります。ブロードキャストの詳しいルールについては、文献[1]を参照してください。

1.1.4 ベクトルの内積と行列の積

続いて、「ベクトルの内積」と「行列の積」について見ていきます。まずはベクトルの内積ですが、これは数式で次のように表されます。

$$\mathbf{x} \cdot \mathbf{y} = x_1y_1 + x_2y_2 + \cdots + x_ny_n \quad (1.1)$$

ここでは2つのベクトル $\mathbf{x} = (x_1, \dots, x_n)$ と $\mathbf{y} = (y_1, \dots, y_n)$ があると仮定します。このとき式(1.1)が示すように、ベクトルの内積は、2つのベクトル間の対応する要素の積を足し合わせたものになります。



ベクトルの内積は、直感的には「2つのベクトルがどれだけ同じ方向を向いているか」を表しています。ベクトルの長さが1の場合に限定すると、2つのベクトルがまったく同じ方向を向いていれば、そのベクトルの内積は1になります。一方、2つのベクトルが逆向きであれば-1になります。

続いて「行列の積」についてです。行列の積は、図1-5の手順に従って計算します。

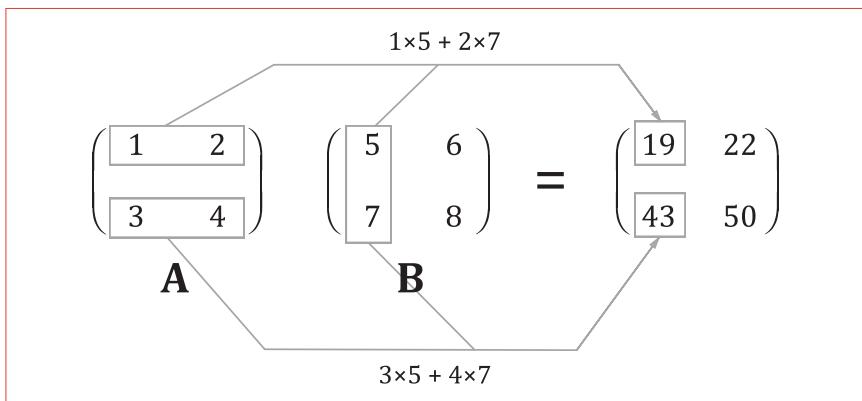


図1-5 行列の積の計算方法

図1-5のとおり、行列の積は、「左側の行列の行ベクトル（横方向）」と「右側の行列の列ベクトル（縦方向）」の内積（要素ごとの積と和）によって計算されます。このとき、その計算結果は新しい行列の対応する要素に格納されます。たとえば、**A**の1行目と**B**の1列目の結果は1行1列目の要素へ、**A**の2行目と**B**の1列目の結果

は 2 行 1 列目の要素へ……といったようになります。

それでは、ベクトルの内積と行列の積を Python で実装してみましょう。それに
は、`np.dot()` が利用できます。

```
# ベクトルの内積
>>> a = np.array([1, 2, 3])
>>> b = np.array([4, 5, 6])
>>> np.dot(a, b)
32

# 行列の積
>>> A = np.array([[1, 2], [3, 4]])
>>> B = np.array([[5, 6], [7, 8]])
>>> np.dot(A, B)
array([[19, 22],
       [43, 50]])
```

ここで示したように、ベクトルの内積と行列の積の計算には、両者とも `np.dot()`
が使えます。`np.dot(x, y)` の引数がともに 1 次元配列の場合はベクトルの内積を
計算し、引数が 2 次元配列の場合は行列の積を計算します。

ここで見た `np.dot()` メソッドの他にも、NumPy には行列計算を行う便利なメ
ソッドが数多く用意されています。それらを使いこなすことができれば、ニューラル
ネットワークの実装もスムーズに進めることができるでしょう。



習うより慣れろ

NumPy を身につけるには、実際に手を動かして練習するのが有効です。たと
えば、「100 numpy exercises」[2] には、NumPy の練習問題が 100 題用意
されています。NumPy の経験を積みたい方は、ぜひ挑戦してみてください。

1.1.5 行列の形状チェック

行列やベクトルを使った計算においては、その「形状」に注意を払うことが重要で
す。ここでは「行列の積」に関して、形状に注目して再度見ていきたいと思います。
さて、行列の積の計算手順はすでに説明しましたが、このとき図 1-6 で示す「形状
チェック」が重要になります。

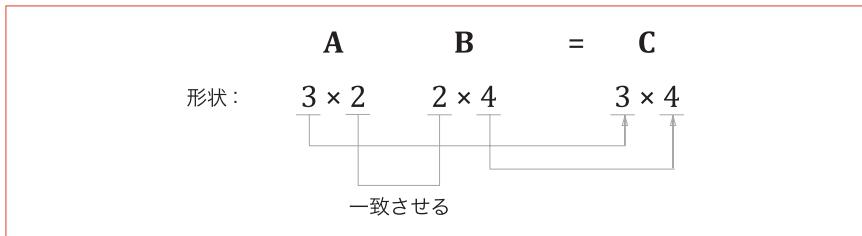


図1-6 形状チェック：行列の積では、対応する次元の要素数を一致させる

図1-6では、 3×2 の行列Aと 2×4 の行列Bの積によって、 3×4 の行列Cが生成される例が示されています。このとき、図が示すように、行列AとBの対応する次元の要素数を一致させる必要があります。そして、結果となる行列Cは、Aの行数とBの列数から構成されることになります。これが行列の「形状チェック」です。



行列の積などの計算では、形状チェック——行列の形状に注目し、その推移を見ていくこと——が重要です。それによって、ニューラルネットワークの実装をスムーズに進めることができます。

1.2 ニューラルネットワークの推論

それではニューラルネットワークの復習を始めます。ニューラルネットワークで行う処理は、2つのフェーズに分けられます。それは「学習」と「推論」です。ここではニューラルネットワークの「推論」だけにフォーカスして話を進めていきます。次節では、ニューラルネットワークの「学習」について見ていきます。

1.2.1 ニューラルネットワークの推論の全体図

ニューラルネットワークは、簡単に言ってしまえば、単なる「関数」です。関数とは、何かを入力したら何かが出力される変換器ですが、これと同じく、ニューラルネットワークも入力を出力へと変換します。

ここでは例として、2次元のデータを入力して、3次元のデータを出力する関数を考えます。これをニューラルネットワークで実現するには、入力層にニューロンを2つ、出力層にニューロンを3つそれぞれ用意します。そして隠れ層（中間層）にもいくつかニューロンを配置します。ここでは隠れ層に4つのニューロンを置くことにしましょう。そうすると、私たちのニューラルネットワークは図1-7のように書け

ます。

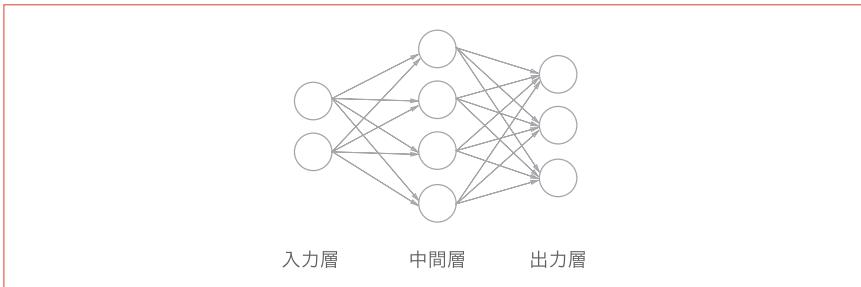


図1-7 ニューラルネットワークの例

図1-7では、ニューロンを○で表し、そのつながりを矢印で表しています。このとき、矢印に重みが存在し、その重みとニューロンの値がそれぞれ乗算され、その和が次のニューロンへの入力となります（正確には、それに活性化関数を適用した値が次のニューロンへの入力になります）。またこのとき、前層のニューロンの値には影響を受けない「定数」も加算されます。この定数はバイアスと呼ばれます。なお、図1-7のニューラルネットワークは、隣接するニューロン間のすべてに（矢印による）結びつきがあるため、これは全結合層と呼ばれます。



図1-7のネットワークは合計で3層から構成されますが、重みを持つ層は実質2層です。本書ではこのようなネットワークを「2層のニューラルネットワーク」と呼ぶことにします。文献によっては、図1-7のネットワークは3層から構成されるため、「3層のニューラルネットワーク」と呼ぶ場合があります。

それでは、図1-7のニューラルネットワークが行う計算を数式で表してみましょう。ここでは入力層のデータを (x_1, x_2) で表し、重みを w_{11} と w_{21} 、バイアスを b_1 のように表すとします。そうすると、図1-7で示す隠れ層の1番上のニューロンは、次のように計算されます。

$$h_1 = x_1 w_{11} + x_2 w_{21} + b_1 \quad (1.2)$$

式(1.2)のように、隠れ層のニューロンは重み付き和によって計算されます。後は、重みとバイアスの値を変えながら、式(1.2)の計算をニューロンの数だけ繰り返し行

います。そうすれば、隠れ層のすべてのニューロンの値を求めるすることができます。

重みとバイアスにはそれぞれ添字がありますが、この添字の規則——どのようなルールで添字に「11」や「12」などを設定するかなど——は重要ではありません。大切なのは、それが「重み付き和」で計算されるということ、そして、それは行列の積でまとめて計算できるということです。実際、全結合層による変換は、行列の積として、次のようにまとめて書くことができます。

$$(h_1, h_2, h_3, h_4) = (x_1, x_2) \begin{pmatrix} w_{11} & w_{12} & w_{13} & w_{14} \\ w_{21} & w_{22} & w_{23} & w_{24} \end{pmatrix} + (b_1, b_2, b_3, b_4) \quad (1.3)$$

ここで隠れ層のニューロンは (h_1, h_2, h_3, h_4) としてまとめおり、これは 1×4 の行列とみなすことができます（もしくは「行ベクトル」として扱います）。また入力は (x_1, x_2) であり、これは 1×2 の行列です。そして重みは 2×4 の行列、バイアスは 1×4 の行列に対応します。そうすると、式 (1.3) は簡略化して次のように書くことができます。

$$\mathbf{h} = \mathbf{x}\mathbf{W} + \mathbf{b} \quad (1.4)$$

ここで入力を \mathbf{x} 、隠れ層のニューロンを \mathbf{h} 、重みを \mathbf{W} 、バイアスを \mathbf{b} で表します。これらはすべて行列です。このとき、式 (1.4) の行列の形状に注目すると、次のように変換されることが分かります。

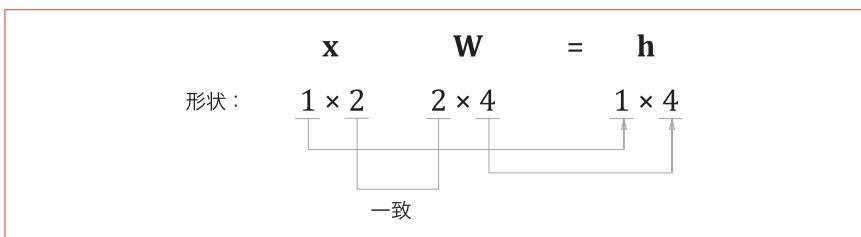


図1-8 形状チェック：対応する次元の要素数が一致することを確認（バイアスは省略）

図1-8に示すとおり、行列の積では、対応する次元の要素数を一致させます。このように行列の形状を見ることで、それが正しい変換であることが確認できます。



行列の積の計算では、行列の形状チェックが重要です。それによって、それが正しい計算かどうか——少なくとも、計算として成り立つかどうか——を確認することができます。

これで全結合層による変換を行列としてまとめて計算することができました。しかし、ここで行った変換はひとつのサンプルデータ（入力データ）だけを対象にしています。ニューラルネットワークの分野では、複数のサンプルデータ——これを「ミニバッチ」と呼びます——に対して、一斉に推論や学習を行います。それを行うには、行列 x の各行に個別のサンプルデータを格納します。たとえば N 個のサンプルデータをミニバッチとしてまとめて処理するとすれば、行列の形状に注目すると、次のように推移します。

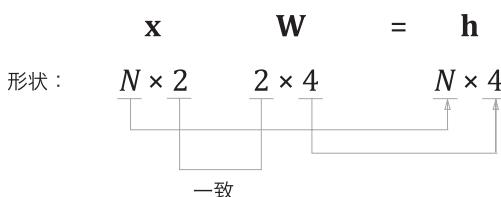


図1-9 形状チェック：ミニバッチ版の行列の積（バイアスは省略）

図1-9に示すように、形状チェックによって、各バッチが正しく変換されていることが分かります。このとき、 N 個のサンプルデータがまとめて全結合層によって変換されており、隠れ層には N 個分のニューロンがまとめて計算されています。それでは、ミニバッチ版の全結合層による変換を Python で書いてみましょう。

```
>>> import numpy as np
>>> W1 = np.random.randn(2, 4)      # 重み
>>> b1 = np.random.randn(4)        # バイアス
>>> x = np.random.randn(10, 2)     # 入力
>>> h = np.dot(x, W1) + b1
```

この例では 10 個のサンプルデータに対して、個別に全結合層による変換を行います。このとき、 x のひとつ目の次元が各サンプルデータに対応しています。たとえば、 $x[0]$ は 0 番目の入力データ、 $x[1]$ は 1 番目の入力データ……というようになります。これと同様に、 $h[0]$ は 0 番目のデータの隠れ層のニューロン、 $h[1]$ は 1

番目のデータの隠れ層のニューロン……と格納されます。



上のコードでは、バイアス $b1$ の足し算においてブロードキャストが働いています。 $b1$ の形状は $(4,)$ ですが、これが $(10, 4)$ の形状になるように自動で複製されます。

さて、全結合層による変換は「線形」な変換です。これに「非線形」な効果を与えるのが活性化関数です。より正確に言うと、非線形な活性化関数を用いることで、ニューラルネットワークの表現力を増すことができるのです。活性化関数にはさまざまなものがありますが、ここでは式 (1.5) で表されるシグモイド関数 (sigmoid function) を使うことにします。

$$\sigma(x) = \frac{1}{1 + \exp(-x)} \quad (1.5)$$

このシグモイド関数は、図 1-10 で示されるような S 字カーブの関数です。

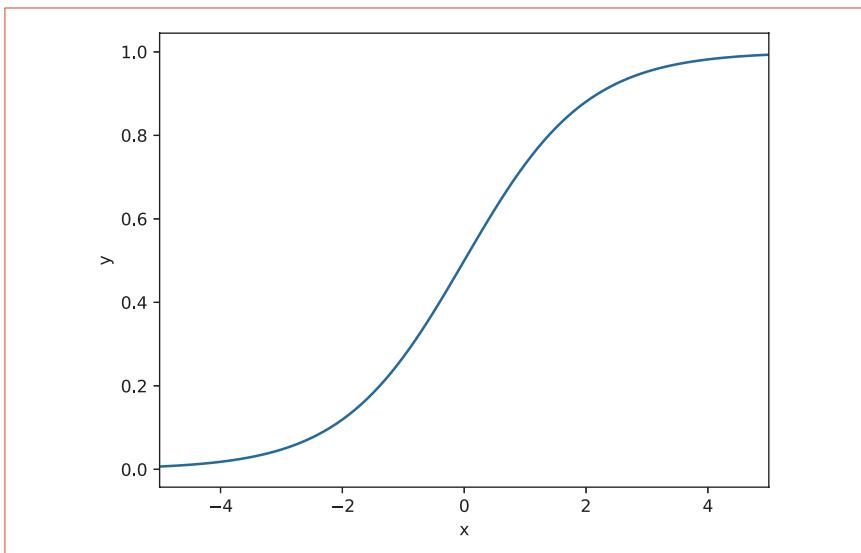


図 1-10 シグモイド関数のグラフ

シグモイド関数は入力として任意の実数を受け取り、0 から 1 の間の実数を出力し

ます。早速このシグモイド関数を Python で実装しましょう。

```
def sigmoid(x):
    return 1 / (1 + np.exp(-x))
```

これは式 (1.5) をそのまま実装したもので、特に難しいことはないでしょう。それでは、このシグモイド関数を使って、先ほどの隠れ層のニューロンを変換します。

```
>>> a = sigmoid(h)
```

これでシグモイド関数によって、非線形な変換がきました。続いて、この活性化関数の出力である a (これは「アクティベーション」と呼ばれます) を、また別の全結合層によって変換します。ここでは、隠れ層のニューロンは 4 つ、出力層のニューロンは 3 つなので、全結合層に使われる重みの行列は、 4×3 の形状に設定しなければなりません。これで出力層のニューロンを得ることができます。以上がニューラルネットワークの推論です。それでは、これまでの話をまとめて Python で書いてみます。

```
import numpy as np

def sigmoid(x):
    return 1 / (1 + np.exp(-x))

x = np.random.randn(10, 2)
W1 = np.random.randn(2, 4)
b1 = np.random.randn(4)
W2 = np.random.randn(4, 3)
b2 = np.random.randn(3)

h = np.dot(x, W1) + b1
a = sigmoid(h)
s = np.dot(a, W2) + b2
```

ここでは x の形状は $(10, 2)$ です。これは 2 次元のデータが 10 個、ミニバッチとしてまとめられていることを意味します。そして最終的に出力される s の形状は $(10, 3)$ になります。繰り返しになりますが、これは 10 個のデータがまとめて処理され、各データが 3 次元データに変換されたことを意味します。

さて、上のニューラルネットワークは 3 次元のデータを出力します。そのため、各次元の値を用いることで、3 クラス分類を行うことができます。その場合、出力された 3 次元ベクトルの各次元は各クラスの「スコア」に対応します（ひとつ目のニューロンがひとつ目のクラス、2 つ目のニューロンが 2 つ目のクラス……）。実際に分類

を行うには、出力層のニューロンの値が一番大きい値を探し、そのニューロンに対応するクラスを結果とします。



スコアとは「確率」になる前の値です。スコアの値が高ければ高いほど、そのニューロンに対応するクラスの確率も高くなります。なお、後ほど見ていきますが、スコアを Softmax 関数に入れると、確率が得られます。

以上が、ニューラルネットワークの推論処理の実装になります。続いて、ここで行った処理を「レイヤ」として Python のクラスを使って実装していきます。

1.2.2 レイヤとしてのクラス化と順伝播の実装

それでは、ニューラルネットワークで行う処理を「レイヤ」として実装しましょう。ここでは全結合層による変換を Affine レイヤとして、シグモイド関数による変換を Sigmoid レイヤとして実装します。なお、全結合層による変換は幾何学の分野におけるアフィン変換に相当するため、Affine レイヤと名付けることにします。また各レイヤは Python のクラスとして実装し、メインとなる変換を `forward()` というメソッド名で実装することにします。



ニューラルネットワークの推論で行う処理は、ニューラルネットワークの順伝播に相当します。順伝播とは、その言葉が示すように、入力層から出力層へ向けた伝播です。このとき、ニューラルネットワークを構成する各レイヤは入力から出力方向へ処理結果を順次伝播していきます。後ほどニューラルネットワークの学習を行いますが、そこでは順伝播とは逆方向にデータ（勾配）を伝播します。それは逆伝播と呼ばれます。

ニューラルネットワークには、さまざまなレイヤが登場します。私たちはそれを Python のクラスとして実装します。そのようなモジュール化を行うことで、レゴブロックを組み合わせるように、ネットワークを構築することができます。本書では、そのようなレイヤを実装するにあたり、次の「実装ルール」を設けることにします。

- すべてのレイヤは、メソッドとして `forward()` と `backward()` を持つ
- すべてのレイヤは、インスタンス変数として `params` と `grads` を持つ

この実装ルールについて簡単に説明します。まずは `forward()` メソッドと `backward()` メソッドですが、これはそれぞれが順伝播と逆伝播に対応します。また、`params` は重みやバイアスなどのパラメータをリストとして保持します（パラメータは複数ある可能性があるのでリストとします）。そして、`grads` は、`params` のパラメータに対応する形で、各パラメータの勾配をリストとして保持します（勾配については後述します）。これが本書の「実装ルール」です。



上記の「実装ルール」に従うことで、見通しの良い実装が行えます。なぜそのようなルールに従うのか、またその有効性については後ほど明らかになります。

ここでは順伝播のみの実装を考えているので、上記の「実装ルール」の次の点だけに焦点を当てて実装を行います。ひとつは「レイヤに `forward()` メソッドを実装すること」そして、「パラメータをインスタンス変数の `params` にまとめる」とことです。この実装ルールに従い、レイヤの実装を行います。それでは初めに、Sigmoid レイヤから実装します。これは次のように実装することができます（➡ ch01/forward_net.py）。

```
import numpy as np

class Sigmoid:
    def __init__(self):
        self.params = []

    def forward(self, x):
        return 1 / (1 + np.exp(-x))
```

上のように、シグモイド関数をクラスとして実装し、メインの変換処理を `forward(x)` メソッドとして実装します。ここで、Sigmoid レイヤには学習するパラメータは存在しないので、インスタンス変数の `params` は空のリストで初期化します。それでは続いて、全結合層である Affine レイヤの実装を示します（➡ ch01/forward_net.py）。

```
class Affine:
    def __init__(self, W, b):
        self.params = [W, b]

    def forward(self, x):
        W, b = self.params
```

```
out = np.dot(x, W) + b  
return out
```

Affine レイヤは、初期化時に重みとバイアスを受け取ります。このとき、Affine レイヤのパラメータは、重みとバイアスになります（その 2 つのパラメータが、ニューラルネットワークの学習で随時更新されていきます）。そこで、その 2 つをインスタンス変数の `params` にリストとして保持します。後は、`forward(x)` による順伝播の処理を実装します。



本書の「実装ルール」により、すべてのレイヤには、学習すべきパラメータがインスタンス変数の `params` に必ず存在することになります。そのため、ニューラルネットワークのすべてのパラメータを簡単にまとめることができ、これにより、パラメータの更新作業やパラメータのファイルへの保存が容易になります。

それでは、上で実装したレイヤを使って、ニューラルネットワークの推論処理を実装します。ここでは、図 1-11 に示すレイヤ構成のニューラルネットワークを実装します。

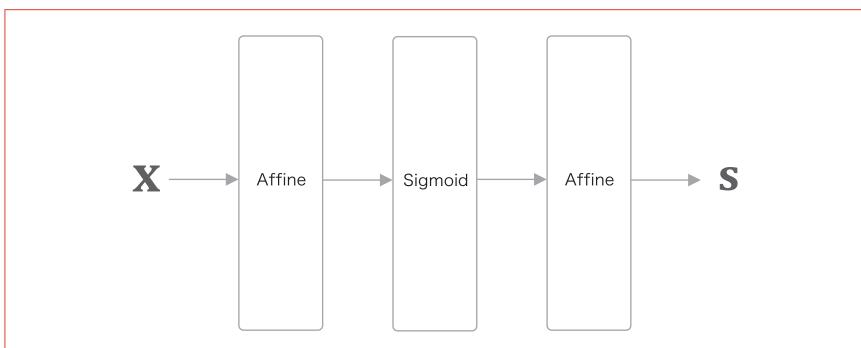


図 1-11 実装するニューラルネットワークのレイヤ構成

図 1-11 に示すとおり、ここでは入力 `x` が Affine レイヤ、Sigmoid レイヤ、Affine レイヤを経て、スコアである `s` が出力されます。私たちは、このニューラルネットワークを `TwoLayerNet` という名前のクラスとして、メインの推論処理を `predict(x)` というメソッドで実装することにします。



ニューラルネットワークを図示するにあたり、これまでには図1-7のように、“ニューロン視点”的図を用いてきました。それに対して図1-11では、“レイヤ視点”でネットワークを図示しました。

それでは、TwoLayerNet の実装を示します (☞ ch01/forward_net.py)。

```
class TwoLayerNet:
    def __init__(self, input_size, hidden_size, output_size):
        I, H, O = input_size, hidden_size, output_size

        # 重みとバイアスの初期化
        W1 = np.random.randn(I, H)
        b1 = np.random.randn(H)
        W2 = np.random.randn(H, O)
        b2 = np.random.randn(O)

        # レイヤの生成
        self.layers = [
            Affine(W1, b1),
            Sigmoid(),
            Affine(W2, b2)
        ]

        # すべての重みをリストにまとめる
        self.params = []
        for layer in self.layers:
            self.params += layer.params

    def predict(self, x):
        for layer in self.layers:
            x = layer.forward(x)
        return x
```

このクラスのイニシャライザでは、初めに重みを初期化し、3つのレイヤを生成します。また、学習すべき重みパラメータを `params` リストにまとめて保持します。ここでは、各レイヤのインスタンス変数の `params` に学習パラメータが保持されているので、それらを連結するだけで完了です。これで `TwoLayerNet` の `params` 変数には、すべての学習パラメータが存在することになりました。このように、パラメータをひとつのリストにまとめることで、「パラメータの更新」や「パラメータの保存」が簡単に行えるようになります。

なお、Python ではリストどうしの連結は + 演算子によって行えます。簡単な例をひとつ示すと、次のようになります。

```
>>> a = ['A', 'B']
>>> a += ['C', 'D']
>>> a
['A', 'B', 'C', 'D']
```

ここで示したように、リストどうしの足し算によって、リストが連結されます。上の TwoLayerNet の実装では、各レイヤの `params` リストを加算することで、すべての学習パラメータをひとつのリストにまとめ上げました。それでは、TwoLayerNet クラスを利用して、ニューラルネットワークの推論を行いましょう。

```
x = np.random.randn(10, 2)
model = TwoLayerNet(2, 4, 3)
s = model.predict(x)
```

これで入力データ `x` に対して、スコア (`s`) を求めることができました。このように、レイヤとしてクラス化することで、ニューラルネットワークが容易に実装できます。またこのとき、`model.params` には学習すべきパラメータがひとつのリストにまとめられているので、続いて行うニューラルネットワークの学習が行いやすくなります。

1.3 ニューラルネットワークの学習

ニューラルネットワークは学習を行わなければ、“良い推論”はできません。そのため、最初に学習を行い、その学習されたパラメータを利用して推論を行うという流れが一般的です。推論とは、前節で見たような、多クラス分類などの問題に答えを出す作業です。一方、ニューラルネットワークの学習は、最適なパラメータを見つける作業になります。本節では、ニューラルネットワークの学習について見てきます。

1.3.1 損失関数

ニューラルネットワークの学習には、学習がどれだけうまくしているかを知るための「指標」が必要になります。一般的にそれは**損失** (loss) と呼ばれます。損失とは、学習段階のある時点におけるニューラルネットワークの性能を示す指標です。損失は、教師データ——学習において与えられる正解データ——とニューラルネットワークの予測結果を元に、それがどれだけ悪いかをスカラ（单一の数値）として算出したものになります。

ニューラルネットワークの損失を求めるには**損失関数** (loss function) を使用しま

す。多クラス分類を行うニューラルネットワークの場合、損失関数として**交差エントロピー誤差**（Cross Entropy Error）を用いる場合が多くあります。このとき交差エントロピー誤差は、ニューラルネットワークが出力する各クラスの「確率」と「教師ラベル」から求められます。

それでは、私たちがこれまで扱ってきたニューラルネットワークに対して、損失を求めてみましょう。ここでは、前節までのネットワークに対して、Softmax レイヤと Cross Entropy Error レイヤを新たに追加します（Softmax レイヤはソフトマックス関数を、Cross Entropy Error レイヤは交差エントロピー誤差を求めるレイヤとします）。このときのネットワーク構成を“レイヤ視点”で描くと、図 1-12 のようになります。

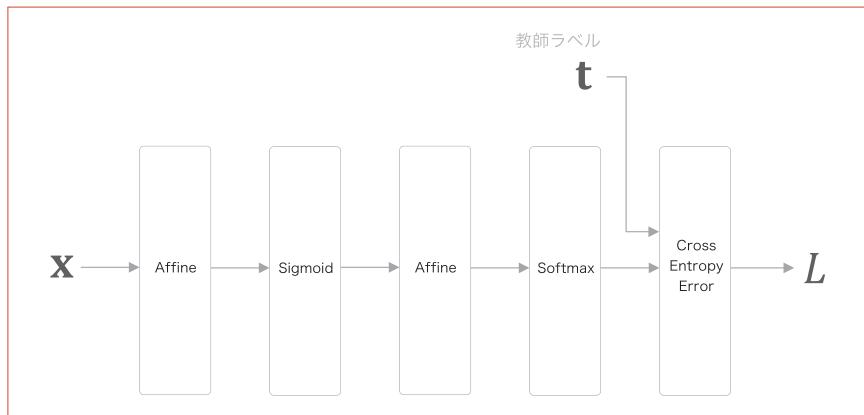


図 1-12 損失関数を適用したニューラルネットワークのレイヤ構成

図 1-12 の x は入力データ、 t は教師ラベル、 L は損失を表します。このとき、Softmax レイヤの出力は確率となり、Cross Entropy Error レイヤには確率と教師ラベルが入力されます。

それでは、Softmax 関数と交差エントロピー誤差について説明します。まずは Softmax 関数についてですが、これは次の式で表されます。

$$y_k = \frac{\exp(s_k)}{\sum_{i=1}^n \exp(s_i)} \quad (1.6)$$

ここでは出力が全部で n 個あるとして、 k 番目の出力 y_k を求める計算式を示しています。この y_k は、 k 番目のクラスに対応する Softmax 関数の出力です。式 (1.6) に示したように、Softmax 関数の分子はスコア s_k の指数関数、分母はすべての入力信号の指数関数の和から構成されます。

Softmax 関数の出力の各要素は 0.0 以上 1.0 以下の実数になります。そして、その要素をすべて足し合わせると 1.0 になります。そのため、ソフトマックスの出力は「確率」として解釈できます。このソフトマックスの出力である「確率」が、続いて交差エントロピー誤差へ入力されます。このとき交差エントロピー誤差は、次の式で表されます。

$$L = - \sum_k t_k \log y_k \quad (1.7)$$

ここで、 t_k は k 番目のクラスに対応する教師ラベルです。 \log は、ネイピア数 e を底とする対数を表します（これは正確には \log_e と表記します）。教師ラベルについては、 $\mathbf{t} = (0, 0, 1)$ のように、one-hot ベクトルで表記します。



one-hot ベクトルとは、ひとつの要素が 1 で、それ以外は 0 となるベクトルです。ここで 1 の要素を正解となるクラスに対応させます。そのため式 (1.7) は、実質的に正解ラベルが 1 の要素に対応する出力の自然対数 (\log) を計算するだけになります。

また、ミニバッチ処理を考慮したとき、交差エントロピー誤差は次の式で表されます。ここではデータが N 個あるとして、 t_{nk} は n 個目のデータの k 次元目の値を意味します。 y_{nk} はニューラルネットワークの出力、 t_{nk} は教師ラベルを表します。

$$L = -\frac{1}{N} \sum_n \sum_k t_{nk} \log y_{nk} \quad (1.8)$$

式 (1.8) は少し複雑に見えますが、これはひとつのデータに対する損失関数を表す式 (1.7) を、単に N 個分のデータに拡張しただけです。ただし、式 (1.8) では N で割ることによって、1 個あたりの「平均の損失関数」を求めます。そのように平均化することで、ミニバッチのサイズに関係なく、いつでも統一した指標が得られます。

本書では、Softmax 関数と交差エントロピー誤差を計算するレイヤを Softmax with Loss レイヤとして実装することにします（この 2 つのレイヤをまとめることで、逆伝播の計算が簡単になります）。そのため、私たちの（学習時における）ニュー

ラルネットワークは、図1-13のようなレイヤ構成になります。

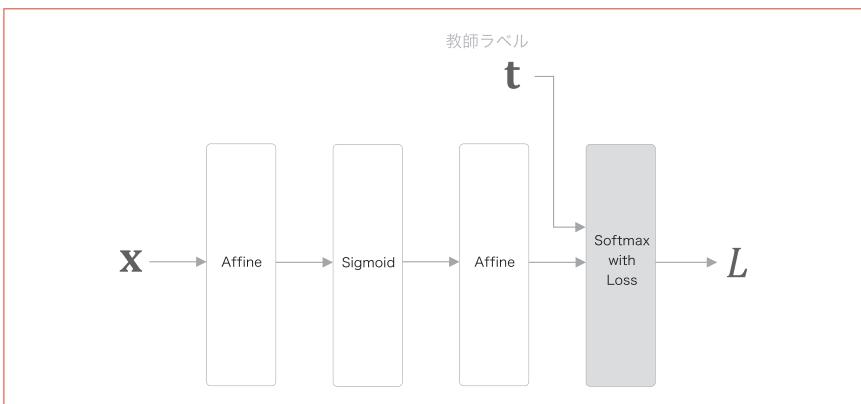


図1-13 Softmax with Loss レイヤを利用して、損失を出力する

図1-13のように、本書ではSoftmax with Loss レイヤを利用します。ここでは、その実装についての説明は省略します。実装ファイルはcommon/layers.pyにあるので、興味のある方は参照してください。また、前作『ゼロから作る Deep Learning』の「4.2 損失関数」において、Softmax with Loss レイヤについて詳しく説明しています。

1.3.2 微分と勾配

ニューラルネットワークの学習の目標は、損失ができるだけ小さくするパラメータを見つけることです。このとき重要なのが「微分」であり、「勾配」です。ここでは、微分と勾配について簡単に説明します。

さて、ある関数 $y = f(x)$ があるとしましょう。このとき、 x に関する y の微分は、 $\frac{dy}{dx}$ と書きます。この $\frac{dy}{dx}$ が意味することは、 x の値を“少しだけ”変化させたとき——より正確には、その「少しの変化」を極限までに小さくしたとき——に、 y の値がどれだけ変化するか、という「変化の度合い」です。

たとえばここに、 $y = x^2$ という関数があるとします。このとき、この関数の微分は解析的に求めることができます。それは $\frac{dy}{dx} = 2x$ となります。そしてこの微分の結果は、各 x における変化の度合いを表します。実際それは、図1-14に示すように関数の「傾き」に相当します。

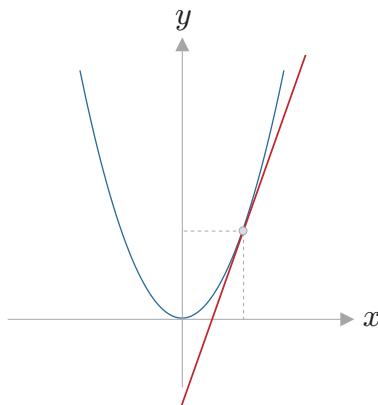
図 1-14 $y = x^2$ の微分は、各 x における傾きを表す

図 1-14 では x というひとつの変数について微分を求めましたが、複数の変数（多変数）についても同様に微分を求めることができます。たとえばここに、 L をスカラ、 \mathbf{x} をベクトルとして、 $L = f(\mathbf{x})$ という関数があるとします。このとき、 x_i (\mathbf{x} の i 番目の要素) に関する L の微分は、 $\frac{\partial L}{\partial x_i}$ と書けます。そして、ベクトルの他の要素の微分についても求めることができ、それは次のようにまとめることができます。

$$\frac{\partial L}{\partial \mathbf{x}} = \left(\frac{\partial L}{\partial x_1}, \frac{\partial L}{\partial x_2}, \dots, \frac{\partial L}{\partial x_n} \right) \quad (1.9)$$

このように、ベクトルの各要素に関する微分をまとめたものを **勾配** (gradient) と呼びます。

また行列についても、ベクトルの場合と同様に勾配を考えることができます。たとえば、 \mathbf{W} を $m \times n$ の行列とした場合、 $L = g(\mathbf{W})$ という関数の勾配は次のようにまとめて書くことができます。

$$\frac{\partial L}{\partial \mathbf{W}} = \begin{pmatrix} \frac{\partial L}{\partial W_{11}} & \cdots & \frac{\partial L}{\partial W_{1n}} \\ \vdots & \ddots & \\ \frac{\partial L}{\partial W_{m1}} & & \frac{\partial L}{\partial W_{mn}} \end{pmatrix} \quad (1.10)$$

式 (1.10) に示すように、 L の \mathbf{W} に関する勾配は行列としてまとめることができます

す（正確には、行列の勾配を上のように定義します）。ここで重要な点は、 \mathbf{W} と $\frac{\partial L}{\partial \mathbf{W}}$ の形状が同じだということです。この「行列とその勾配が同じ形状である」という性質を利用することで、パラメータの更新作業やチェインルールの実装が簡単に行えます（チェインルールについては、すぐ後に詳しく説明します）。



本書で使用する「勾配」という用語は、数学で使われる「勾配」とは厳密には異なります。数学で「勾配」と言うと、それはベクトルに対しての微分に限定されます。一方、ディープラーニングの分野では、行列やテンソルについても微分を定義し、それを「勾配」と呼ぶのが一般的です。

1.3.3 チェインルール

学習時におけるニューラルネットワークは、学習データを与えると損失を出力します。ここで私たちが得たいものは、各パラメータに関する損失の勾配です。その勾配が得られれば、それを使ってパラメータの更新を行うことができます。では、ニューラルネットワークの勾配はどのように求めることができるのでしょうか。ここで誤差逆伝播法（back-propagation）が登場します。

誤差逆伝播法を理解する上で、キーとなるのが**チェインルール（連鎖律）**です。チェインルールとは、合成関数に関する微分の法則です（合成関数とは、複数の関数によって構成される関数です）。

それでは、チェインルールについて学びましょう。ここでは例として、 $y = f(x)$ と $z = g(y)$ という 2 つの関数について考えます。そして $z = g(f(x))$ で表されるように、最終的な出力 z は 2 つの関数によって計算されるとします。このとき、この合成関数の微分—— x に関する z の微分——は次のように求めることができます。

$$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial y} \frac{\partial y}{\partial x} \quad (1.11)$$

式 (1.11) が示すように、 x に関する z の微分は、 $y = f(x)$ の微分と $z = g(y)$ の微分の積によって求められます。これがチェインルールです。このチェインルールが重要なのは、私たちの扱う関数がどれだけ複雑だとしても——どれだけ複数の関数が連結したとしても——、その微分は個別の関数の微分によって求めることができるからです。つまり、各関数の局所的な微分を計算できれば、その積によって最終的な全体の微分を求めることができます。



ニューラルネットワークは、複数の「関数」が連結されたものと考えることができます。誤差逆伝播法は、その複数の関数（ニューラルネットワーク）に対して、チェインルールを効率良く利用して勾配を求めます。

1.3.4 計算グラフ

これから誤差逆伝播法について見ていきますが、ここではその前準備として「計算グラフ」について説明します。計算グラフは、計算を視覚的に表すものです。早速ですが、計算グラフの例を図1-15に示します。

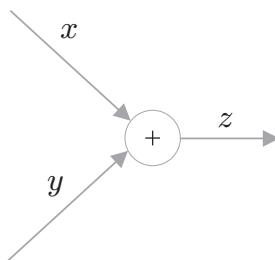


図1-15 $z = x + y$ を表す計算グラフ

図1-15で示すように、計算グラフはノードと矢印で図示します。このとき、加算を「+」ノードで表し、変数 x と y を各矢印の上に書くことにします。このように、計算グラフでは演算をノードで表し、その処理結果が順に——この例では左から右へと——流れます。これが計算グラフの「順伝播」です。

計算グラフを使えば、計算を視覚的に把握することができます。さらに、その勾配も直感的に求めることができます。ここで重要なのが、勾配は順伝播とは逆方向に伝播させることです。この逆方向の伝播が「逆伝播」です。

逆伝播の説明の前に、逆伝播が行われる全体像をより明確にしたいと思います。ここで私たちは $z = x + y$ という計算を扱っていますが、その計算の前後には「何らかの計算」があることを想定します（図1-16）。そして、最終的にスカラである L が出力されると仮定します（ニューラルネットワークの学習においては、計算グラフの最終的な出力は損失であり、これはスカラになります）。

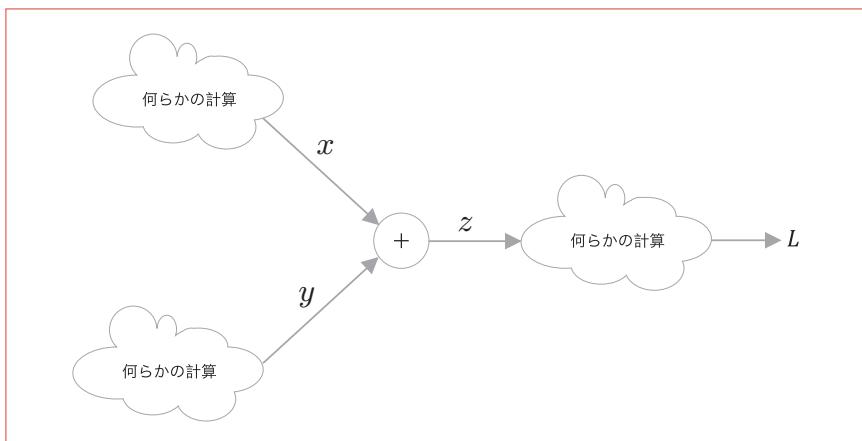


図 1-16 加算ノードは“複雑な計算”の一部を構成する

私たちの目標とすることは、 L の微分（勾配）を各変数について求めることです。そうすると、計算グラフの逆伝播は図 1-17 のように書くことができます。

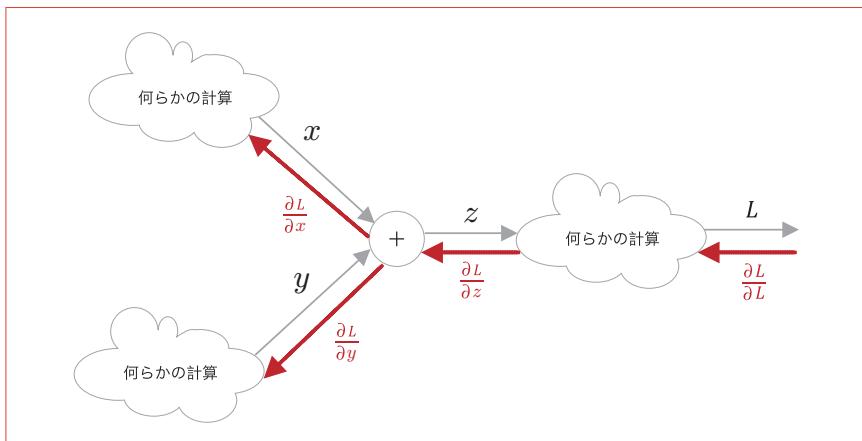


図 1-17 計算グラフの逆伝播

図 1-17 に示すとおり、逆伝播は赤い太線の矢印で描画し、矢印の下側に「伝播する値」を書くことにします。このとき「伝播する値」とは、最終的な出力 L の各変数に関する微分になります。この例では、 z に関する微分は $\frac{\partial L}{\partial z}$ であり、 x と y に関する

る微分は、それぞれ $\frac{\partial L}{\partial x}$ と $\frac{\partial L}{\partial y}$ になります。

そして、ここでチェインルールが登場します。先ほど復習したチェインルールに従えば、逆伝播で流れる微分の値は、上流から流れる微分と各演算ノードの局所的な微分との積によって計算されます（ここで「上流」は出力側を指します）。そのため、上の例では、 $\frac{\partial L}{\partial x} = \frac{\partial L}{\partial z} \frac{\partial z}{\partial x}$ であり、 $\frac{\partial L}{\partial y} = \frac{\partial L}{\partial z} \frac{\partial z}{\partial y}$ となります。

さて、ここで私たちちは、 $z = x + y$ という加算ノードによる演算を扱っています。このとき $\frac{\partial z}{\partial x} = 1$ 、 $\frac{\partial z}{\partial y} = 1$ とそれぞれ（解析的に）求められます。そのため加算ノードでは、図1-18 のように、上流から伝わる伝播に 1 を乗算して下流へと勾配を伝播します。つまり、上流からの勾配をそのまま流すだけになります。

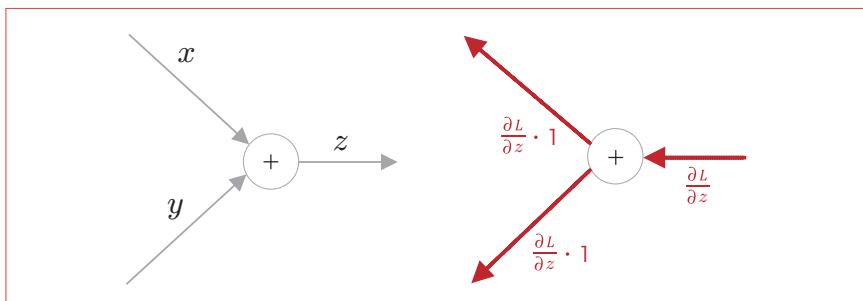


図1-18 加算ノードの順伝播（左図）と逆伝播（右図）

このように計算グラフは計算を視覚的に表します。そして、逆伝播による勾配の流れを見ることで、その導出の過程を理解するのにも役立てることができます。

計算グラフを構築する演算ノードには、ここで見てきた「加算ノード」の他にもさまざまな演算が考えられます。続いて、代表的な演算ノードをいくつか紹介します。

1.3.4.1 乗算ノード

乗算ノードは、 $z = x \times y$ という計算です。このとき微分は、 $\frac{\partial z}{\partial x} = y$ 、 $\frac{\partial z}{\partial y} = x$ とそれぞれ求めることができます。そのため、乗算ノードの逆伝播は図1-19 のように、「上流から伝わる勾配」に「順伝播時の入力を入れ替えた値」を乗算します。

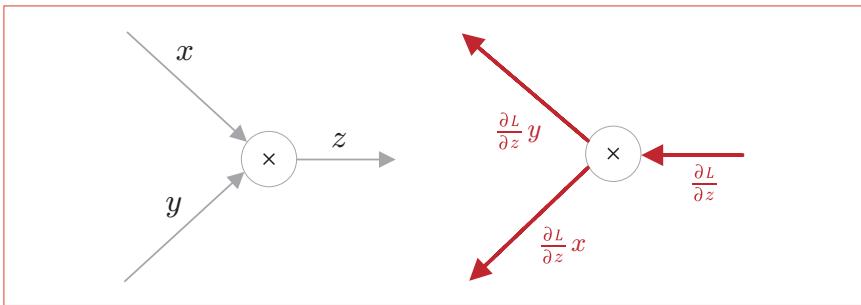


図 1-19 乗算ノードの順伝播（左図）と逆伝播（右図）

なお、これまで見てきた加算ノードと乗算ノードの説明では、ノードを流れるデータは「1変数」でした。しかし、流れるデータは1変数ではなく、多変数——ベクトルや行列、テンソル——であっても問題ありません。加算ノード（もしくは乗算ノード）を流れるデータがテンソルの場合は、テンソルの各要素を独立に計算するだけです。つまりその場合は、テンソルの他の要素とは独立に、「要素ごとの（element-wise）演算」を行います。

1.3.4.2 分岐ノード

分岐ノードは、図 1-20 に示すように、分岐するノードです。

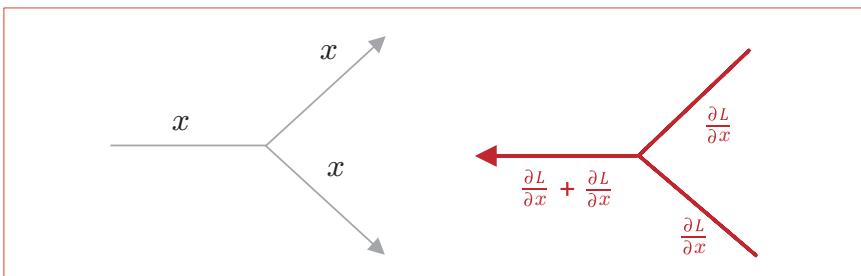


図 1-20 分岐ノードの順伝播（左図）と逆伝播（右図）

分岐ノードは、正確にはノードとしては描画せずに、単に線が2つに分かれるよう描画します。このとき、同じ値がコピーされて分岐します。そのため、分岐ノードは「コピーノード」と呼ぶこともできます。そしてその逆伝播は、図 1-20 に示すとおり、上流からの勾配の「和」になります。

1.3.4.3 Repeat ノード

分岐ノードは 2 つの分岐でしたが、これを一般化させると N 個の分岐（コピー）が考えられます。ここではそれを Repeat ノードと呼ぶことにします。それでは、Repeat ノードの例を計算グラフで書いてみましょう。

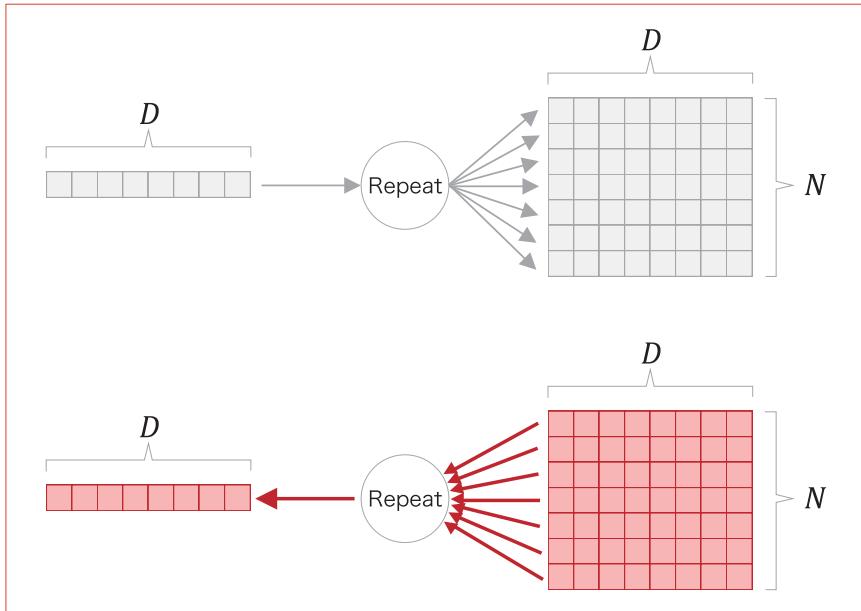


図 1-21 Repeat ノードの順伝播（上図）と逆伝播（下図）

図 1-21 に示すとおり、ここでは長さが D の配列を N 個だけ複製する例を示しています。この Repeat ノードは、 N 個の分岐ノードとみなすことができるため、その逆伝播は N 個の勾配の総和として求めることができます。実際に実装例を挙げると、次のように書くことができます。

```
>>> import numpy as np
>>> D, N = 8, 7
>>> x = np.random.randn(1, D) # 入力
>>> y = np.repeat(x, N, axis=0) # forward

>>> dy = np.random.randn(N, D) # 仮の勾配
>>> dx = np.sum(dy, axis=0, keepdims=True) # backward
```

ここでは、`np.repeat()` メソッドによって、要素の複製を行います。上の例では配列 `x` を `N` 回複製しますが、このとき `axis` を指定することで、どの軸方向に沿って複製するかを指定できます。また、逆伝播では総和を求めることになるので、NumPy の `sum()` メソッドを利用します。このときも、引数の `axis` を指定することで、どの軸方向に沿って和を求めるかを指定します。さらに引数で `keepdims=True` を指定することで、2 次元配列の次元数を維持します。上の例では、`keepdims=True` の場合、`np.sum()` の結果の形状は $(1, D)$ になり、`keepdims=False` の場合は $(D,)$ になります。



NumPy のブロードキャストは、配列の要素の複製を行います。これは、Repeat ノードを使って表すことができます。

1.3.4.4 Sum ノード

Sum ノード（総和ノード）は汎用的な加算ノードです。ここでは $N \times D$ の配列に対して、その総和を第 0 軸に対して求める計算を考えます。このとき、Sum ノードの順伝播と逆伝播は図 1-22 のようになります。

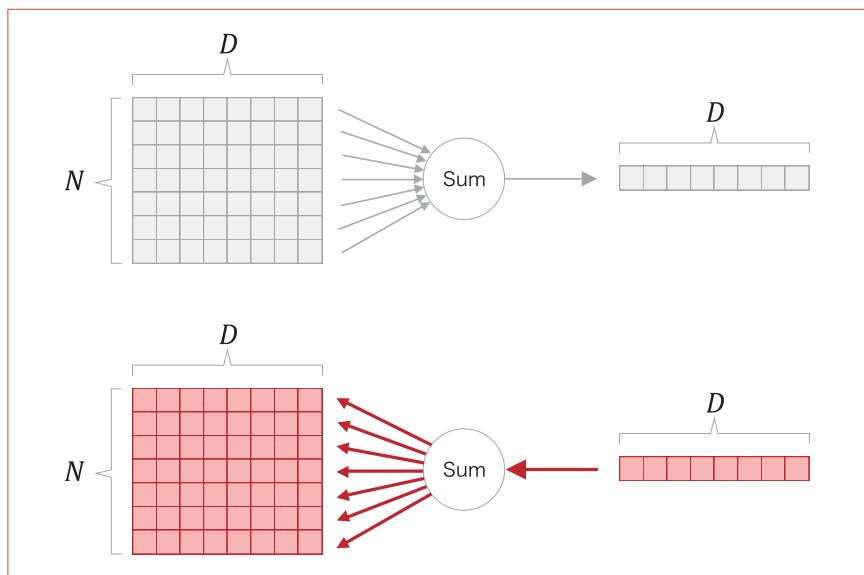


図 1-22 Sum ノードの順伝播（上図）と逆伝播（下図）

図1-22に示すとおり、Sumノードの逆伝播は、上流からの勾配をすべての矢印に分配します。これは加算ノードの逆伝播の自然な拡張です。それでは、Repeatノードと同様に、Sumノードの実装例も示しましょう。これは次のようになります。

```
>>> import numpy as np
>>> D, N = 8, 7
>>> x = np.random.randn(N, D)           # 入力
>>> y = np.sum(x, axis=0, keepdims=True) # forward

>>> dy = np.random.randn(1, D)          # 仮の勾配
>>> dx = np.repeat(dy, N, axis=0)       # backward
```

見てのとおり、Sumノードの順伝播は`np.sum()`メソッド、逆伝播は`np.repeat()`メソッドによって実装できます。ここでの興味深い点は、SumノードとRepeatノードは、それぞれ「逆の関係」にあるということです。逆の関係とは、Sumノードの順伝播がRepeatノードの逆伝播に相当し、Sumノードの逆伝播がRepeatノードの順伝播に相当するということです。

1.3.4.5 MatMulノード

本書では、行列の積をMatMulノード（「Matrix Multiply」の略）とします。MatMulノードの逆伝播はやや複雑になるため、ここでは一般的な説明を行った後で、直感的な理解が得られる説明を行います。

MatMulノードを説明するにあたり、 $y = xW$ という計算を考えます。ここで、 x 、 W 、 y の形状は、それぞれ $1 \times D$ 、 $D \times H$ 、 $1 \times H$ とします（図1-23）。

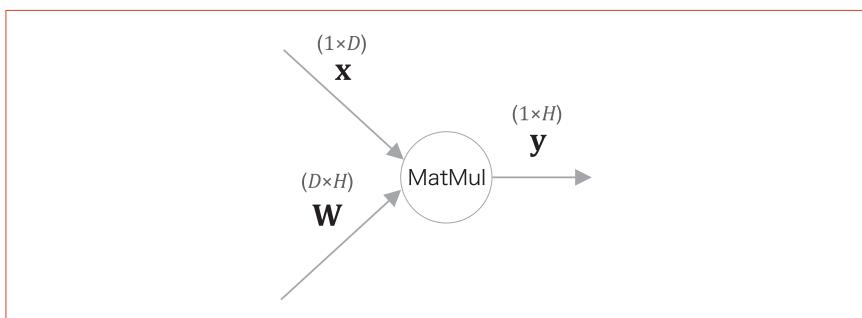


図1-23 MatMulノードの順伝播：各変数の上側に形狀を示す

このとき、 \mathbf{x} の i 番目の要素に関する微分 $\frac{\partial L}{\partial x_i}$ は、次のように求められます。

$$\frac{\partial L}{\partial x_i} = \sum_j \frac{\partial L}{\partial y_j} \frac{\partial y_j}{\partial x_i} \quad (1.12)$$

式 (1.12) の $\frac{\partial L}{\partial x_i}$ は、 x_i を（少しだけ）変化させたときに L がどれだけ変化するか、という「変化の度合い」を表します。ここで x_i を変化させたとき、ベクトル \mathbf{y} のすべての要素が変化します。そして、 \mathbf{y} の各要素の変化を通じて、最終的に L が変化することになります。そのため、 x_i から L にいたるチェインルールの経路は複数あり、その総和が $\frac{\partial L}{\partial x_i}$ となります。

さて式 (1.12) ですが、これはまだ簡単にすることができます。それには、 $\frac{\partial y_j}{\partial x_i} = W_{ij}$ が成り立つことを利用して、それを式 (1.12) に代入します。

$$\frac{\partial L}{\partial x_i} = \sum_j \frac{\partial L}{\partial y_j} \frac{\partial y_j}{\partial x_i} = \sum_j \frac{\partial L}{\partial y_j} W_{ij} \quad (1.13)$$

式 (1.13) から、 $\frac{\partial L}{\partial x_i}$ は、「ベクトル $\frac{\partial L}{\partial \mathbf{y}}$ 」と「 \mathbf{W} の i 行目のベクトル」の内積によって求められることが分かります。この関係から、次の式が導けます。

$$\frac{\partial L}{\partial \mathbf{x}} = \frac{\partial L}{\partial \mathbf{y}} \mathbf{W}^T \quad (1.14)$$

式 (1.14) が示すように、 $\frac{\partial L}{\partial \mathbf{x}}$ は行列の積によって一度に求められます。ここで、 \mathbf{W}^T の T は転置行列であることを表します。それでは式 (1.14) に対して「形状チェック」を行ってみましょう。その結果は図1-24 のようになります。

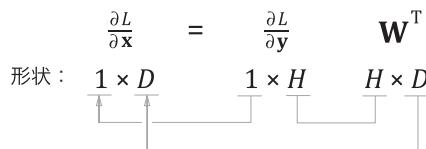


図1-24 行列の積の形状チェック

図1-24 より、行列の形状の推移は正しいことが分かります。これによって、式 (1.14) が正しい計算であることが確認できるのです。また、それを逆手に取って——

つまり、その整合性が成り立つようになります——、逆伝播の式（実装）を導くこともできます。その方法を説明するにあたって、再度 $\mathbf{y} = \mathbf{x}\mathbf{W}$ という行列の積の計算を考えます。ただし今回はミニバッチ処理を考慮して、 \mathbf{x} には N 個のデータが格納されているものとします。このとき、 \mathbf{x} 、 \mathbf{W} 、 \mathbf{y} の形状はそれぞれ $N \times D$ 、 $D \times H$ 、 $N \times H$ となり、逆伝播の計算グラフは図1-25 のようになります。

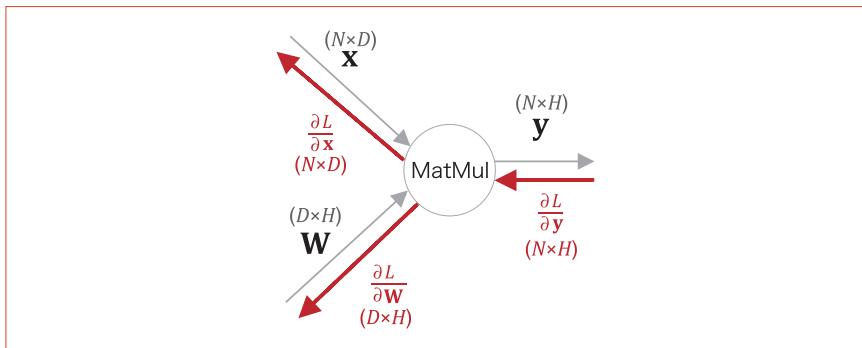


図1-25 MatMul ノードの逆伝播

それでは $\frac{\partial L}{\partial \mathbf{x}}$ は、どのように計算できるか考えましょう。このとき、 $\frac{\partial L}{\partial \mathbf{x}}$ に関係する変数（行列）は、上流からの勾配 $\frac{\partial L}{\partial \mathbf{y}}$ と \mathbf{W} です。ここでなぜ \mathbf{W} が関係するかというと、これは乗算の逆伝播を考えると分かりやすいでしょう。というのも、乗算の逆伝播では「順伝播時の入力を入れ替えた値」を使用しました。それと同じく、行列の積の逆伝播でも「順伝播時の入力を入れ替えた行列」を使用するのです。後は、それぞれの行列の形状に注目し、その整合性が保たれるように行列の積を組み立てることにします。そうすると、図1-26 のように行列の積の逆伝播が導けます。

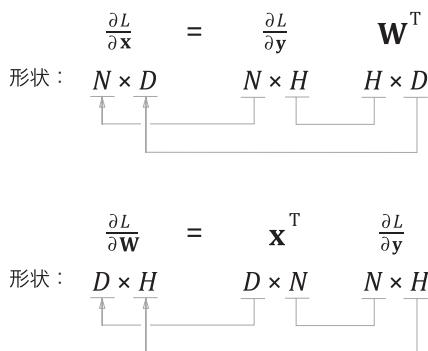


図 1-26 行列の形状を確認することで、逆伝播の式を導く

図 1-26 に示すとおり、行列の形状を確認することで、行列の積の逆伝播の式を組み立てることができました。これで、MatMul ノードの逆伝播が導けました！ それでは、MatMul ノードをレイヤとして実装しましょう。これは次のように実装できます (☞ common/layers.py)。

```
class MatMul:
    def __init__(self, W):
        self.params = [W]
        self.grads = [np.zeros_like(W)]
        self.x = None

    def forward(self, x):
        W, = self.params
        out = np.dot(x, W)
        self.x = x
        return out

    def backward(self, dout):
        W, = self.params
        dx = np.dot(dout, W.T)
        dW = np.dot(self.x.T, dout)
        self.grads[0][...] = dW
        return dx
```

MatMul レイヤは `params` に学習するパラメータを保持します。そしてそれに対応させる形で、勾配を `grads` に保持させることにします。逆伝播では `dx` と `dW` を求めて、重みの勾配をインスタンス変数の `grads` に設定します。

なお、勾配の値を設定する際に、`grads[0][...] = dW` というように「3点リーダー」を使っています。この「3点リーダー」を用いることで、NumPy 配列のメモリ位置を固定した上で、NumPy 配列の要素を上書きします。



「3点リーダー」と同じようなことは、`grads[0] = dW` の「代入」によっても行えます。一方、「3点リーダー」の場合は、NumPy 配列の「上書き」が行われます。これは「浅いコピー(shallow copy)」か「深いコピー(deep copy)」かという違いです。`grads[0] = dW` の代入は「浅いコピー」、`grads[0][...] = dW` の上書きは「深いコピー」に相当します。

「3点リーダー」については少し話が複雑になってきたので、具体例を出して説明します。ここに 2つの NumPy 配列 `a` と `b` があるとします。

```
>>> a = np.array([1, 2, 3])
>>> b = np.array([4, 5, 6])
```

ここで `a = b` と `a[...] = b` の場合では、`a` にはともに `[4, 5, 6]` が代入されます。しかし、このとき `a` が指すメモリの位置は異なります。実際に（単純化した）メモリを可視化すると、図 1-27 のようになります。

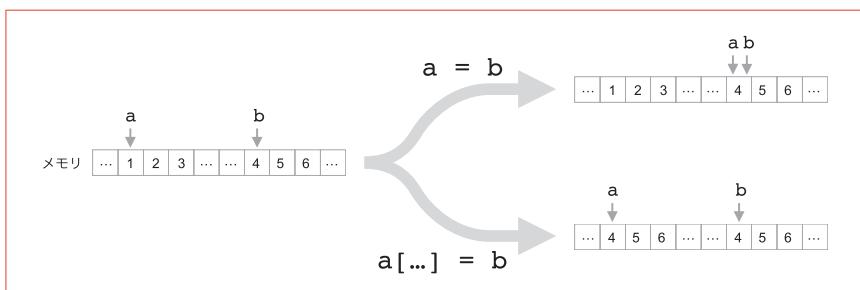


図 1-27 `a=b` と `a[...]=b` の違い：「3点リーダー」はデータが上書きされるため、変数が指すメモリ位置は変わらない

図 1-27 に示すように、`a = b` では、`a` が指すメモリ位置が `b` のそれと同じになります。実際のデータ (4, 5, 6) は複製されないため、これは「浅いコピー」と言えます。一方、`a[...] = b` のときは、`a` のメモリ位置は固定のまま、`a` の指すメモリ上に `b` の要素がコピーされます。これは実際のデータが複製されるため、「深いコピー」

と言えます。

以上より「3点リーダー」を用いることで、変数のメモリアドレスを固定できることが分かりました（上の例では `a` のアドレスは固定されています）。私たちの場合、このメモリアドレスを固定することによって、インスタンス変数の `grads` の扱いがよりシンプルになります。



`grads` リストには各パラメータの勾配を格納します。このとき、`grads` リストの各要素は NumPy 配列として、レイヤの生成時に一度だけ生成します。後は、「3点リーダー」を用いて、その NumPy 配列のメモリ上の場所を動かすことなく上書きします。そうすることで、勾配をひとまとめにする作業は、最初に一度だけ行えばよいことになります。

以上が MatMul レイヤの実装です。なお、この実装ファイルは、`common/layers.py` にあります。

1.3.5 勾配の導出と逆伝播の実装

計算グラフの説明が終わったので、続いて実用的なレイヤを実装しましょう。ここでは、Sigmoid レイヤ、全結合層の Affine レイヤ、Softmax with Loss レイヤを実装します。

1.3.5.1 Sigmoid レイヤ

シグモイド関数は、 $y = \frac{1}{1+\exp(-x)}$ という式で表されます。そして、シグモイド関数の微分は、次の式で表されます。

$$\frac{\partial y}{\partial x} = y(1 - y) \quad (1.15)$$

式 (1.15) より、Sigmoid レイヤの計算グラフは図 1-28 のように書くことができます。ここでは、出力側のレイヤから伝わってきた勾配 ($\frac{\partial L}{\partial y}$) に Sigmoid 関数の微分 ($\frac{\partial y}{\partial x}$) をかけ合わせて、それを入力側のレイヤに伝播します。

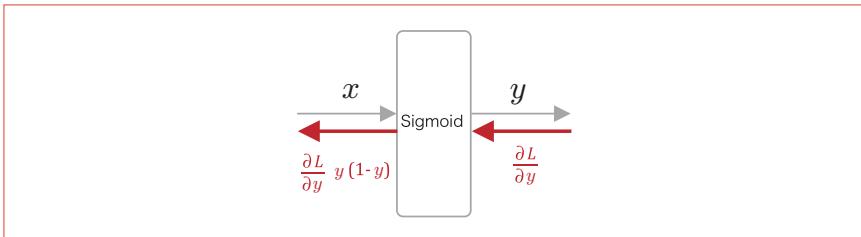


図 1-28 Sigmoid レイヤの計算グラフ



ここでは、シグモイド関数の微分の導出過程は省略します。「付録 A sigmoid 関数と tanh 関数の微分」では、計算グラフを使ってシグモイド関数の微分を求める過程を説明しています。興味のある方は参考にしてください。

それでは、Sigmoid レイヤを Python で実装します。[図 1-28](#) を参考にすれば、次のように実装することができます ([common/layers.py](#))。

```
class Sigmoid:
    def __init__(self):
        self.params, self.grads = [], []
        self.out = None

    def forward(self, x):
        out = 1 / (1 + np.exp(-x))
        self.out = out
        return out

    def backward(self, dout):
        dx = dout * (1.0 - self.out) * self.out
        return dx
```

ここでは、順伝播の出力をインスタンス変数の `out` に保持しておきます。そして、逆伝播において、その `out` 変数を使って計算を行います。

1.3.5.2 Affine レイヤ

前に示したとおり、Affine レイヤの順伝播は、 $y = \text{np.dot}(x, W) + b$ で実装できました。このときバイアスの加算では、NumPy のブロードキャストが使われています。その点を明示的に表すと、Affine レイヤの計算グラフは[図 1-29](#) のように書けます。

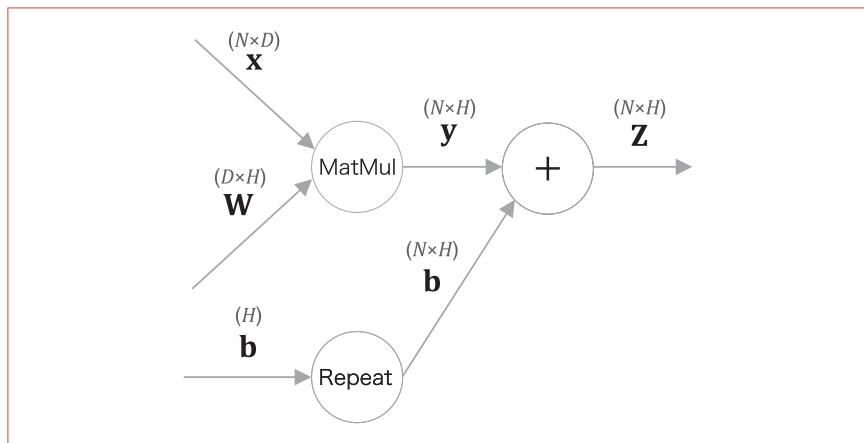


図 1-29 Affine レイヤの計算グラフ

図 1-29 に示すとおり、MatMul ノードで行列の積の計算を行います。そして、バイアスが Repeat ノードで複製され、その後で加算が行われます（NumPy のプロトキャストの機能は、裏側で Repeat ノードの計算が行われていると考えることができます）。それでは、Affine レイヤの実装を示します（[common/layers.py](#)）。

```

class Affine:
    def __init__(self, W, b):
        self.params = [W, b]
        self.grads = [np.zeros_like(W), np.zeros_like(b)]
        self.x = None

    def forward(self, x):
        W, b = self.params
        out = np.dot(x, W) + b
        self.x = x
        return out

    def backward(self, dout):
        W, b = self.params
        dx = np.dot(dout, W.T)
        dW = np.dot(self.x.T, dout)
        db = np.sum(dout, axis=0)

        self.grads[0][...] = dW
        self.grads[1][...] = db
        return dx
  
```

本書の実装ルールに従い、インスタンス変数の `params` にはパラメータを、`grads` には勾配を保持します。逆伝播の実装は、`MatMul` ノードと `Repeat` ノードの逆伝播を行えば求められます。`Repeat` ノードの逆伝播は `np.sum()` によって計算できますが、このとき行列の形状に注目することで、どの軸 (axis) で和を求めるべきかがはっきりします。最後に、重みパラメータの勾配をインスタンス変数の `grads` に設定します。以上が、`Affine` レイヤの実装です。



`Affine` レイヤは、すでに実装した `MatMul` レイヤを用いると、より簡単に実装できます。ここでは復習もかねて、`MatMul` レイヤを用いずに、NumPy のメソッドによる実装を行いました。

1.3.5.3 Softmax with Loss レイヤ

私たちは、Softmax 関数と交差エントロピー誤差を合わせて Softmax with Loss レイヤとして実装することにします。このとき、計算グラフは図1-30 のようになります。

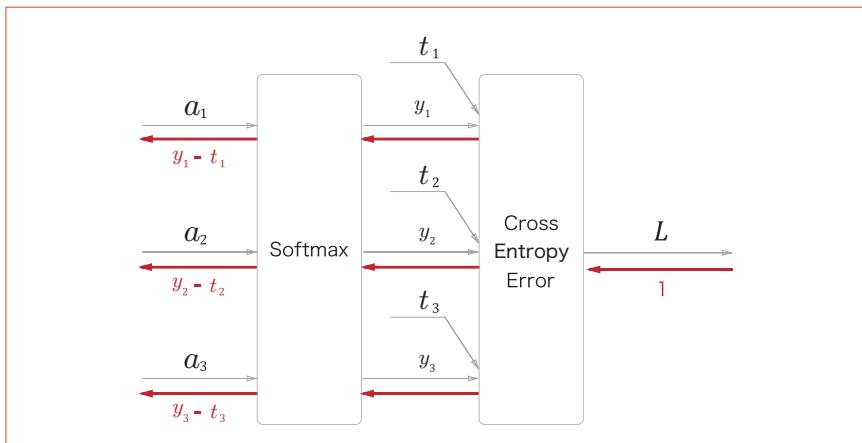


図1-30 Softmax with Loss レイヤの計算グラフ

図1-30 の計算グラフでは、Softmax 関数は Softmax レイヤとして、交差エントロピー誤差は Cross Entropy Error レイヤとして表記します。ここでは、3 クラス分類を行う場合を想定し、前レイヤ (=入力層に近い側のレイヤ) から 3 つの入力を

受け取ることを仮定します。

図 1-30 に示すように、Softmax レイヤは、入力である (a_1, a_2, a_3) を正規化して、 (y_1, y_2, y_3) を出力します。Cross Entropy Error レイヤは、Softmax の出力 (y_1, y_2, y_3) と教師ラベルの (t_1, t_2, t_3) を受け取り、それらのデータから損失 L を出力します。



図 1-30 で注目すべきは、逆伝播の結果です。Softmax レイヤからの逆伝播は、 $(y_1 - t_1, y_2 - t_2, y_3 - t_3)$ という“キレイ”な結果になっています。 (y_1, y_2, y_3) は Softmax レイヤの出力、 (t_1, t_2, t_3) は教師データなので、 $(y_1 - t_1, y_2 - t_2, y_3 - t_3)$ は Softmax レイヤの出力と教師ラベルの差分になります。ニューラルネットワークの逆伝播では、この差分（誤差）が前レイヤへ伝わっていくのです。これはニューラルネットワークの学習における重要な性質です。

ここでは、Softmax with Loss レイヤの実装については説明を省略します。なお、実装は `common/layers.py` にあります。また、Softmax with Loss レイヤの逆伝播の導出過程は、前作『ゼロから作る Deep Learning』の「付録 A Softmax-with-Loss レイヤの計算グラフ」に詳しい説明があります。興味のある方は参照してください。

1.3.6 重みの更新

誤差逆伝播法によって勾配を求めることができたら、その勾配を使ってニューラルネットワークのパラメータを更新します。このとき、ニューラルネットワークの学習は次の手順で行います。

- **Step-1** (ミニバッチ)
訓練データの中からランダムに複数のデータを選び出す
- **Step-2** (勾配の算出)
誤差逆伝播法により、各重みパラメータに関する損失関数の勾配を求める
- **Step-3** (パラメータの更新)
勾配を使って重みパラメータを更新する
- **Step-4** (繰り返す)
Step-1、Step-2、Step-3 を必要な回数だけ繰り返す

上記の手順に従って、ニューラルネットワークの学習が行われます。まずはミニバッチでデータを選び、続いて誤差逆伝播法によって重みの勾配を得ます。この勾配は、現時点での重みパラメータにおいて、損失を最も増やす方向を指します。そのため、パラメータをその勾配の逆方向に更新することで、損失を下げることができます。これが**勾配降下法** (Gradient Descent) です。後は、その作業を必要な回数だけ繰り返し行います。

さて、私たちは上記の Step-3 において重みを更新しますが、この重みの更新方法としてはさまざまな手法が提案されています。ここでは、の中でも最も単純な **SGD** (Stochastic Gradient Descent: 確率的勾配降下法) という手法を実装します。なお、「Stochastic (確率的)」は、ランダムに選ばれたデータ（ミニバッチ）に対する勾配を用いることを意味します。

SGD は単純な方法です。これは、(現状の) 重みを勾配方向へある一定の距離だけ更新します。数式で表すと、式 (1.16) のようになります。

$$\mathbf{W} \leftarrow \mathbf{W} - \eta \frac{\partial L}{\partial \mathbf{W}} \quad (1.16)$$

ここで、更新する重みパラメータを \mathbf{W} とし、 \mathbf{W} に関する損失関数の勾配を $\frac{\partial L}{\partial \mathbf{W}}$ とします。 η は学習係数を表し、実際には 0.01 や 0.001 といった値をあらかじめ決めて使います。

それでは、SGD の実装に移りましょう。ここではモジュール化を考慮して、パラメータの更新を行うクラスを `common/optimizer.py` に実装することにします（このファイルでは、SGD の他にも AdaGrad や Adam などの実装も行います）。

そして、パラメータの更新を行うクラスは `update(params, grads)` という共通のメソッドを持つように実装します。ここで引数の `params` にはニューラルネットワークの重みが、`grads` には勾配がそれぞれリストとして格納されているものとします。そして、`params` と `grads` の同じインデックスには、対応するパラメータと勾配がそれぞれ格納されていることを想定します。そうすると、SGD は次のように実装できます (☞ `common/optimizer.py`)。

```
class SGD:
    def __init__(self, lr=0.01):
        self.lr = lr

    def update(self, params, grads):
        for i in range(len(params)):
            params[i] -= self.lr * grads[i]
```

初期化の引数である `lr` は learning rate (学習係数) を表します。ここでは、この学習係数をインスタンス変数として保持します。そして、`update(params, grads)` というメソッドの中で、パラメータの更新処理を実装します。

この `SGD` というクラスを使えば、ニューラルネットワークのパラメータの更新は、次のように行なうことができます（次に示すコードは、実際には動作しない擬似コードです）。

```
model = TwoLayerNet(...)
optimizer = SGD()

for i in range(10000):
    ...
    x_batch, t_batch = get_mini_batch(...) # ミニバッチの取得
    loss = model.forward(x_batch, t_batch)
    model.backward()
    optimizer.update(model.params, model.grads)
    ...
```

このように、最適化を行うクラスを分離して実装することで、機能のモジュール化が容易になります。本書では、SGD の他にも、Momentum や AdaGrad、Adam などの手法を実装しています。それらの実装は、`common/optimizer.py` にあります。ここではそれらの最適化手法についての説明は省略します。詳しくは前作『ゼロから作る Deep Learning』の「6.1 パラメータの更新」を参照してください。

1.4 ニューラルネットワークで問題を解く

以上で準備が整いました。これから簡単なデータセットに対して、ニューラルネットワークの学習を行います。

1.4.1 スpiral・データセット

本書ではデータセットのための便利なクラスを `dataset` ディレクトリにいくつか提供しています。本節ではその中から `dataset/spiral.py` というファイルを利用します。このファイルは「スパイラル（渦巻き）データ」を読み込むクラスが実装されており、次のように利用します（☞ `ch01/show_spiral_dataset.py`）。

```
import sys
sys.path.append('..') # 親ディレクトリのファイルをインポートするための設定
from dataset import spiral
```

```
import matplotlib.pyplot as plt
```

```
x, t = spiral.load_data()  
print('x', x.shape) # (300, 2)  
print('t', t.shape) # (300, 3)
```

上の例では、ch01 ディレクトリから dataset ディレクトリにある `spiral.py` をインポートして利用します。そのため、上記ソースコード中の `sys.path.append('..')` によって、親ディレクトリをインポートの検索パスに追加します。

そして、`spiral.load_data()` によってデータの読み込みを行います。このとき、`x` が入力データで `t` が教師ラベルです。`x` と `t` の形状を見ると、それぞれ 300 個のサンプルデータがあり、`x` は 2 次元データ、`t` は 3 次元データであることが分かります。なお、`t` は one-hot ベクトルであり、対応するクラスが 1 に、それ以外は 0 にラベル付けされています。それでは、このデータをグラフ上にプロットしてみましょう。その結果は図 1-31 のようになります。

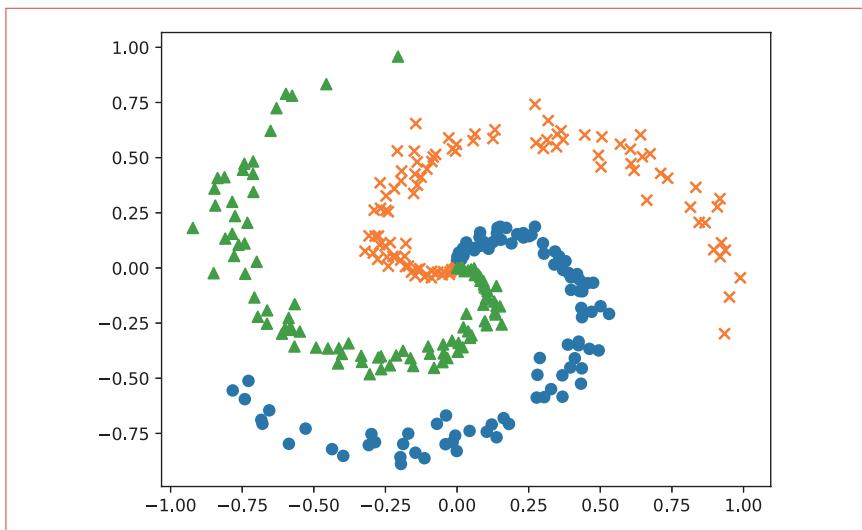


図 1-31 学習に使用するスパイラル・データセット ($\times\blacktriangle\bullet$ で、3 つのクラスを表す)

図 1-31 に示すように、入力は 2 次元のデータで、分類するクラス数は 3 つあります。このデータセットを見ると、これは直線によって分離できないことが分かり

ます。そのため、非線形な分離線を学習する必要があるということです。私たちのニューラルネットワーク——活性化関数に非線形な Sigmoid 関数を使用した隠れ層のあるニューラルネットワーク——は、その非線形なパターンを正しく学習できるのでしょうか？早速、実験してみましょう。



ここでは簡単な実験のため、データセットを訓練データや検証データ、テストデータへ分離することは行いません。実際の問題では、データセットを訓練用とテスト用（さらに検証用）のデータに分離し、学習と評価を行います。

1.4.2 ニューラルネットワークの実装

それでは、ニューラルネットワークの実装を行います。ここでは隠れ層がひとつの中のニューラルネットワークを実装します。まずは、インポート文とイニシャライザの`__init__()`を示します ([ch01/two_layer_net.py](#))。

```
import sys
sys.path.append('..')
import numpy as np
from common.layers import Affine, Sigmoid, SoftmaxWithLoss

class TwoLayerNet:
    def __init__(self, input_size, hidden_size, output_size):
        I, H, O = input_size, hidden_size, output_size

        # 重みとバイアスの初期化
        W1 = 0.01 * np.random.randn(I, H)
        b1 = np.zeros(H)
        W2 = 0.01 * np.random.randn(H, O)
        b2 = np.zeros(O)

        # レイヤの生成
        self.layers = [
            Affine(W1, b1),
            Sigmoid(),
            Affine(W2, b2)
        ]
        self.loss_layer = SoftmaxWithLoss()

    # すべての重みと勾配をリストにまとめる
    self.params, self.grads = [], []
    for layer in self.layers:
        self.params += layer.params
        self.grads += layer.grads
```

イニシャライザは3つの引数を受け取ります。`input_size`は入力層のニューロンの数、`hidden_size`は中間層のニューロンの数、そして`output_size`は出力層のニューロンの数を表します。中身の実装では、まずはバイアスを0（ゼロ）ベクトル(`np.zeros()`)で初期化します。そして、重みを小さなランダム値(`0.01 * np.random.randn()`)で初期化します。なお、重みを小さなランダム値にすることで、学習がうまく進みやすくなります。続いて、必要なレイヤを生成し、それをインスタンス変数の`layers`リストにまとめます。最後に、このモデルで使用するパラメータと勾配をひとつにまとめます。



Softmax with Loss レイヤは、他のレイヤとは扱いが異なるため、`layers`リストには入れず、インスタンス変数の`loss_layer`に別に保持することにします。

続いて、`TwoLayerNet`に3つのメソッドを実装します。ここでは、推論を行う`predict()`メソッド、順伝播の`forward()`メソッド、逆伝播の`backward()`メソッドを実装します(☞ ch01/two_layer_net.py)。

```
def predict(self, x):
    for layer in self.layers:
        x = layer.forward(x)
    return x

def forward(self, x, t):
    score = self.predict(x)
    loss = self.loss_layer.forward(score, t)
    return loss

def backward(self, dout=1):
    dout = self.loss_layer.backward(dout)
    for layer in reversed(self.layers):
        dout = layer.backward(dout)
    return dout
```

見てのとおり、ここでの実装はスッキリとしたものになりました！私たちはすでにニューラルネットワークで使用する処理ブロックを「レイヤ」として実装してきたので、ここでは、それらのレイヤの`forward()`と`backward()`を適切な順番で呼ぶだけなのです。

1.4.3 学習用のソースコード

続いて、学習を行うコードを示します。ここでは、学習データを読み込み、ニューラルネットワーク（モデル）とオプティマイザを生成します。そして、先ほど示した学習の4ステップの手順に従って学習を行います。なお、機械学習の分野では、問題のために設計した手法（ニューラルネットワークやSVMなど）を指して、「モデル」と呼ぶことが一般的です。それでは、学習用のコードを次に示します（[ch01/train_custom_loop.py](#)）。

```
import sys
sys.path.append('..')
import numpy as np
from common.optimizer import SGD
from dataset import spiral
import matplotlib.pyplot as plt
from two_layer_net import TwoLayerNet

# ①ハイパーパラメータの設定
max_epoch = 300
batch_size = 30
hidden_size = 10
learning_rate = 1.0

# ②データの読み込み、モデルとオプティマイザの生成
x, t = spiral.load_data()
model = TwoLayerNet(input_size=2, hidden_size=hidden_size, output_size=3)
optimizer = SGD(lr=learning_rate)

# 学習で使用する変数
data_size = len(x)
max_iters = data_size // batch_size
total_loss = 0
loss_count = 0
loss_list = []

for epoch in range(max_epoch):
    # ③データのシャッフル
    idx = np.random.permutation(data_size)
    x = x[idx]
    t = t[idx]

    for iters in range(max_iters):
        batch_x = x[iters*batch_size:(iters+1)*batch_size]
        batch_t = t[iters*batch_size:(iters+1)*batch_size]
```

```

# ④勾配を求め、パラメータを更新
loss = model.forward(batch_x, batch_t)
model.backward()
optimizer.update(model.params, model.grads)

total_loss += loss
loss_count += 1

# ⑤定期的に学習経過を出力
if (iters+1) % 10 == 0:
    avg_loss = total_loss / loss_count
    print('| epoch %d | iter %d / %d | loss %.2f'
          % (epoch + 1, iters + 1, max_iters, avg_loss))
    loss_list.append(avg_loss)
    total_loss, loss_count = 0, 0

```

まずはコードの①の場所で、ハイパーパラメータを設定します。具体的には、学習するエポック数、バッチサイズや隠れ層のニューロン数、学習係数を設定します。続いて②の場所で、データの読み込みを行い、ニューラルネットワーク（モデル）とオプティマイザを生成します。私たちはすでに2層のニューラルネットワークをTwoLayerNetクラスとして、またオプティマイザをSGDクラスとして実装しました。ここではそれらを利用します。



エポック (epoch) は学習の単位を表します。1エポックは、学習データをすべて“見た”とき——データセットを1周したとき——に相当します。ここでは300エポックの学習を行います。

学習を行う際には、ミニバッチとしてランダムにデータを選びます。ここでは、エポック単位でデータのシャッフルを行い、シャッフルを行ったデータに対して、頭から順にデータを抜き出します。データのシャッフルには——正確には、データの「インデックス」のシャッフルには——、np.random.permutation()メソッドを使います。これは引数に N を与えると、0から $N - 1$ までのランダムな並びを作成して返します。実際の使用例は次のようになります。

```

>>> import numpy as np
>>> np.random.permutation(10)
array([7, 6, 8, 3, 5, 0, 4, 1, 9, 2])

>>> np.random.permutation(10)
array([1, 5, 7, 3, 9, 2, 8, 6, 0, 4])

```

このように、`np.random.permutation()` を呼ぶと、データのインデックスを無作為にシャッフルすることができます。

続いてコードの④で勾配を求め、パラメータを更新します。最後に⑤で、定期的に学習の結果を出力します。ここでは、10 イテレーションごとに損失の平均を求め、それを変数の `loss_list` に追加します。以上が学習を行うためのソースコードです。



ここで実装したニューラルネットワークの学習用のコードは、本書の他の場所でも使用します。そのため本書では、このコードを `Trainer` クラスとして提供します。`Trainer` クラスを使うことで、ニューラルネットワークの学習の詳細を `Trainer` クラスの内部に押し込めることができます。詳しい使い方は「1.4.4 Trainer クラス」で説明します。

それでは、上のコード (`ch01/train_custom_loop.py`) を実行してみましょう。そうすると、ターミナルに出力される損失の値が順調に下がっていくことが分かります。そして、その結果をプロットすると、次のようになります。

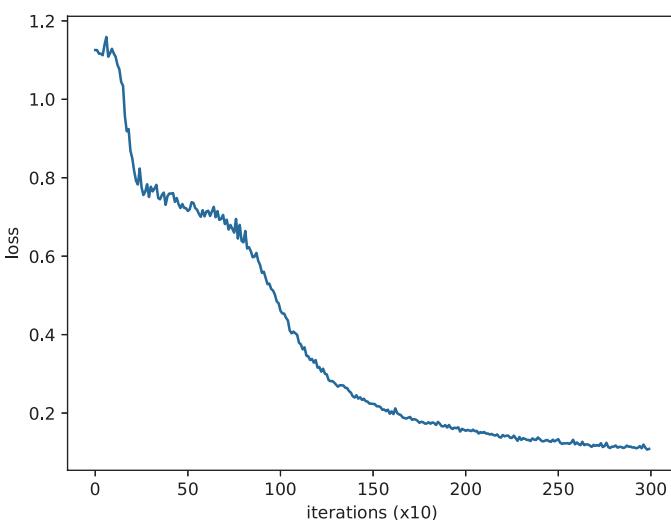


図1-32 損失のグラフ：横軸は学習のイテレーション（目盛りの値の 10 倍）、縦軸は学習 10 イテレーションごとの損失の平均

図1-32 のとおり、学習を進めるに従って、損失が減っていることが分かります。私たちのニューラルネットワークは、正しい方向に学習しているようです！ それでは、学習後のニューラルネットワークがどのような分離領域——これは**決定境界** (decision boundary) と呼びます——を作っているのかを可視化してみましょう。その結果は、図1-33 のようになります。

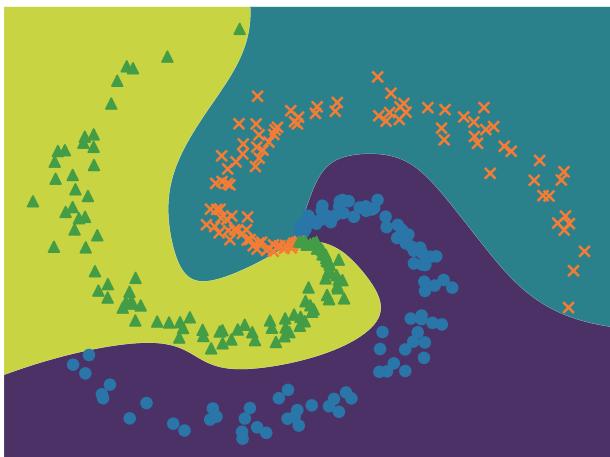


図1-33 学習後のニューラルネットワークの決定境界（ニューラルネットワークが識別するクラスごとの領域を色を分けて描画）

図1-33 に示すように、学習後のニューラルネットワークは、「渦巻き」のパターンを正しく捉えていることが分かります。つまり、非線形な分離領域を学習することができたのです！ このように、ニューラルネットワークは隠れ層を持つことで複雑な表現が可能になります。さらに層を重ねることで、その表現力がより豊かになるのがディープラーニングの特徴です。

1.4.4 Trainer クラス

前に述べたとおり、本書ではニューラルネットワークの学習を実行する機会が多くあります。そこでは、先ほど行ったような学習用のコードを書く必要がありますが、毎回同じようなコードを書くのは退屈でしょう。そこで本書では、学習を行うクラスを **Trainer** クラスとして提供します。中身の実装は、先ほどのソースコードとほと

んど同じです。一部、新しい機能を追加していますが、詳しい使い方は必要になった際に説明します。

`Trainer` クラスは、`common/trainer.py` にあります。このクラスのイニシャライザは、ニューラルネットワーク（モデル）とオプティマイザを受け取ります。具体的には、次のように使います。

```
model = TwoLayerNet(...)
optimizer = SGD(lr=1.0)
trainer = Trainer(model, optimizer)
```

そして、`fit()` メソッドを呼んで学習を開始します。この `fit()` メソッドは表1-1 に示す引数を持ちます。

表1-1 `Trainer` クラスの `fit()` メソッドの引数。表中の「(= XX)」はデフォルトの引数を表す

| 引数 | 説明 |
|-----------------------------------|--|
| <code>x</code> | 入力データ |
| <code>t</code> | 教師ラベル |
| <code>max_epoch (= 10)</code> | 学習を行うエポック数 |
| <code>batch_size (= 32)</code> | ミニバッチのサイズ |
| <code>eval_interval (= 20)</code> | 結果（平均損失など）を表示するインターバル たとえば <code>eval_interval=20</code> と設定すると、20 イテレーションごとに損失の平均を求める、その結果を画面に出力する |
| <code>max_grad (= None)</code> | 勾配の最大ノルム 勾配のノルムがこの値を超えた場合、勾配を小さくする（勾配クリッピング）。詳細は「5 章 リカレントニューラルネットワーク（RNN）」参照 |

また、`Trainer` クラスには `plot()` というメソッドがあります。これは `fit()` メソッドで記録した損失——正確には `eval_interval` のタイミングで評価された平均損失——をプロットします。それでは、`Trainer` クラスを使って学習を行うコードを次に示します ([ch01/train.py](#))。

```
import sys
sys.path.append('..')
from common.optimizer import SGD
from common.trainer import Trainer
from dataset import spiral
from two_layer_net import TwoLayerNet

max_epoch = 300
```

```
batch_size = 30
hidden_size = 10
learning_rate = 1.0

x, t = spiral.load_data()
model = TwoLayerNet(input_size=2, hidden_size=hidden_size, output_size=3)
optimizer = SGD(lr=learning_rate)

trainer = Trainer(model, optimizer)
trainer.fit(x, t, max_epoch, batch_size, eval_interval=10)
trainer.plot()
```

このコードを実行すると、前回同様にニューラルネットワークの学習が行われます。以前に示した学習用のコードを `Trainer` クラスに担わせることで、コードがスッキリしました。本書ではこれ以降、`Trainer` クラスを使って学習を行います。

1.5 計算の高速化

ニューラルネットワークの学習や推論では、多くの計算が必要になります。そのため、ニューラルネットワークをいかに高速に計算するかということは重要なテーマです。ここでは、ニューラルネットワークの高速化に有効な「ビット精度」と「GPU」について簡単に説明します。



本書では、計算の高速化よりも実装の分かりやすさを優先しています。ただし、計算の高速化という観点から、これ以降、データのビット精度を意識した実装を行います。また、計算に多くの時間がかかる箇所では、(オプションとして) GPU で実行できるような工夫がされています。

1.5.1 ビット精度

NumPy の浮動小数点数は、標準で 64 ビットのデータ型が使用されます（64 ビットかどうかは、読者の環境——OS や Python/NumPy のバージョンなど——によって変わる可能性があります）。実際に 64 ビットの浮動小数点数が使われることは、次のコードで確かめられます。

```
>>> import numpy as np
>>> a = np.random.randn(3)
>>> a.dtype
dtype('float64')
```

NumPy 配列のインスタンス変数 `dtype` によって、データの型を見るることができます。上の結果は `float64` と表示されていますが、これは 64 ビットの浮動小数点数を表します。

このように NumPy では標準で 64 ビットの浮動小数点数が使われます。しかし、ニューラルネットワークの推論および学習は、32 ビットの浮動小数点数で問題なく——認識精度をほとんど落とすことなく——行えることが知られています。メモリの観点では、32 ビットは 64 ビットの半分になるため、常に 32 ビットが好ましいと言えます。また、ニューラルネットワークの計算では、データを転送する「バス帯域」がボトルネックになる場合があります。その場合も、もちろんデータ型は小さいほうが望ましいです。そして、計算速度の点においても、32 ビット浮動小数点数のほうが高速に計算できます（浮動小数点数の計算速度は、CPU や GPU のアーキテクチャに依存します）。

そのため、本書では 32 ビット浮動小数点数を優先して使用することにします。NumPy で 32 ビット浮動小数点数を使うには、次のようにデータ型を `np.float32` や '`f`' と指定します。

```
>>> b = np.random.randn(3).astype(np.float32)
>>> b.dtype
dtype('float32')

>>> c = np.random.randn(3).astype('f')
>>> c.dtype
dtype('float32')
```

また、ニューラルネットワークの推論に限定すれば、16 ビット浮動小数点数で、認識精度をほとんど劣化させずに処理できることが分かっています [6]。ただし、NumPy には 16 ビットの浮動小数点数が用意されていますが、一般的な CPU や GPU では演算自体は 32 ビットで行われます。そのため、16 ビットの浮動小数点数に変換したところで、計算自体は 32 ビットの浮動小数点数で行われてしまい、処理速度の点では恩恵を受けられません。

しかし、学習した重みを（外部ファイルに）保存するようなケースでは、16 ビットの浮動小数点数は有効です。具体的には、重みデータを 16 ビット精度で保存する場合、32 ビットのときの半分の容量で保存することができます。そのため本書では、学習した重みを保存するときに限り、16 ビットの浮動小数点数に変換することにします。



ディープラーニングが注目されるに伴って、最近の GPU では 16 ビットの半精度浮動小数点数が「ストレージ」と「演算」の両方でサポートされるようになってきました。また、Google の TPU と呼ばれる独自チップは、8 ビットで計算できるような工夫がされています [7]。

1.5.2 GPU (CuPy)

ディープラーニングの計算は、大量の積和演算によって構成されます。この大量の積和演算の多くは並列計算が可能であり、それは CPU よりも GPU が得意とするところです。そこで、一般的なディープラーニングのフレームワークでは、CPU に加えて GPU でも実行できるように設計されています。

本書では、CuPy [3] と呼ばれる Python ライブライアリをオプションとして使用します。CuPy は、GPU による並列計算を行うためのライブラリです。CuPy を利用するには、NVIDIA 製の GPU を備えたマシンが必要になります。また、CUDA と呼ばれる GPU 向けの汎用並列コンピューティング・プラットフォームをインストールする必要があります。詳しいインストール方法は、CuPy 公式のインストール・ドキュメント [4] を参照してください。

この CuPy を使えば、NVIDIA 製の GPU を使って簡単に並列計算を行うことができます。さらに重要なことは、CuPy は NumPy と共に API を持つことです。簡単な使用例を示すと、次のようになります。

```
>>> import cupy as cp
>>> x = cp.arange(6).reshape(2, 3).astype('f')
>>> x
array([[ 0.,  1.,  2.],
       [ 3.,  4.,  5.]], dtype=float32)
>>> x.sum(axis=1)
array([ 3.,  12.], dtype=float32)
```

ここで示したように、CuPy は基本的に NumPy と同じ使い方ができます。そして、その裏側では GPU を使って計算が行われます。これが意味することは、NumPy で書かれたコードがあれば、それを“GPU 版”に簡単に変更できるということです。なぜなら、そのとき私たちが行うことは、（基本的には）`numpy` を `cupy` に置き換えるだけだからです！



2018年6月現在、CuPyはNumPyのメソッドのすべてをカバーしていません。CuPyはNumPyとの完全な互換性はありませんが、共通のAPIを多く持ちます。

繰り返しになりますが、本書では実装の分かりやすさを優先し、CPUによる実装を基本とします。ただし計算に多くの時間がかかるコードに関しては、オプションとしてCuPyを使った実装を提供します。そしてCuPyを使う場合においても、読者がCuPyを使うことを意識する必要はないように配慮しています。

本書でGPU実行可能なコードが最初に登場するのは4章の`ch04/train.py`です。この`ch04/train.py`は、次のインポート文で始まります。

```
import sys
sys.path.append('..')
import numpy as np
from common import config
# GPUで実行する場合は、下記のコメントアウトを消去（要cupy）
# =====
# config.GPU = True
# =====
...
```

このコードの実行にはCPUで数時間かかりますが、GPUを使えば數十分程度で完了します。そして、本書が提供するコードでは、上のソースコードを1行修正するだけでGPUモードで実行できます。具体的には「`# config.GPU = True`」のコメントアウトを外せば、NumPyの代わりにCuPyが使われるようになります。それによって、GPU上で実行され、高速に学習が行われます。GPUをお持ちの方は、ぜひ利用してみましょう。



NumPyがCuPyに切り替わる仕組みはとても簡単です。興味のある方は、`common/config.py`や`common/np.py`、`common/layers.py`のインポート文を参照してください。

1.6 まとめ

本章では、ニューラルネットワークの基本を復習しました。まずはベクトルや行列などの数学の復習からスタートし、Python（特にNumPy）の基本的な使い方の確認を行いました。そして、ニューラルネットワークの仕組みを見てきました。特に、

計算グラフの基本パーツ（加算ノードや乗算ノードなど）をいくつか取り上げ、その順伝播と逆伝播を説明しました。

また、ニューラルネットワークの実装も行いました。私たちはモジュール性を考慮し、ニューラルネットワークの基本パーツをレイヤとして実装しました。レイヤの実装では、クラスのメソッドとして `forward()` と `backward()` を持つこと、インスタンス変数として `params` と `grads` を持つことを本書の「実装ルール」としました。これによって、ニューラルネットワークの実装が見通しの良いものとなりました。

最後に本章では、人工的な「渦巻きデータセット」に対して、隠れ層がひとつあるニューラルネットワークで学習を行いました。そして、そのモデルが正しく学習できることを確認しました。これでニューラルネットワークの復習は終わりです。これからニューラルネットワークという頼もしい武器を手に、自然言語処理の世界へと乗り込みます。それでは先に進みましょう！

本章で学んだこと

- ニューラルネットワークは、入力層、隠れ層、出力層を持つ
- 全結合層によって線形な変換が行われ、活性化関数によって非線形な変換が行われる
- 全結合層やミニバッチ処理は、行列としてまとめて計算することができる
- 誤差逆伝播法を使って、効率的にニューラルネットワークの損失に関する勾配を求めることができる
- ニューラルネットワークで行う処理は、計算グラフによって可視化することができます、順伝播や逆伝播の理解に役立つ
- ニューラルネットワークの実装では、構成要素を「レイヤ」としてモジュール化することで、組み立てが容易になる
- ニューラルネットワークの高速化において、データのビット精度と GPU による並列計算が重要である

2章 自然言語と単語の分散表現

マーティ：「これはヘビーだ」

ドク：「未来ではそんなに物が重いのか？」

——映画『バック・トゥ・ザ・フューチャー』

これからいよいよ自然言語処理の世界へ足を踏み入れます。自然言語処理が扱う分野は多岐にわたりますが、その本質的問題は、コンピュータに私たちの言葉を理解させることにあります。本章では、コンピュータに言葉を理解させるはどういうことか、そしてどのようなアプローチが存在するのか、ということを中心に考察を進めます。特に古典的な手法——ディープラーニング登場以前の手法——について詳しく見ていきます。そして次章から、ディープラーニング（正確にはニューラルネットワーク）をベースとした手法へと進みます。

また、本章ではテキストを Python で扱う練習も行います。テキストを単語に分割する処理や、単語を単語 ID に変換する処理などを実装します。そして本章で実装する関数は、次章以降でも利用します。つまり本章は、これから先のテキスト処理の下準備も兼ねています。それでは、自然言語処理の世界へと進みましょう！

2.1 自然言語処理とは

日本語や英語など、私たちが普段使っている言葉を **自然言語** (Natural Language) と言います。**自然言語処理** (Natural Language Processing : NLP) とは、文字どおり解釈すれば、「自然言語を処理する分野」ということになります。これは分かりやすく言うと、「私たちの言葉をコンピュータに理解させるための技術（分野）」とい

うことです。つまり、自然言語処理の目標とは、人の話す言葉をコンピュータに理解させ、私たちにとって役に立つことをコンピュータに行わせることにあるのです。

ところで、コンピュータが理解できる言語というと、「プログラミング言語」や「マークアップ言語」のようなものを思い浮かべるかもしれません。それらの言語は、コードの意味が一意に解釈できるように文法が定義されており、コンピュータは決まったルールに従ってコードを解析します。

皆さんもご存知のとおり、一般的なプログラミング言語は機械的で無機的な言語です。これは言ってみれば、“固い言語”と言えます。一方、英語や日本語などの自然言語は、“柔らかい言語”です。「柔らかい」というのは、同じ意味の文章でもさまざまな表現が可能であったり、文章に曖昧さがあつたりと、柔軟に意味や形が変わることを意味します。また、時代とともに新しい言葉や新しい意味が生まれます（または、廃れます）。これも自然言語の柔らかさの表れです。

このように自然言語は生きた言語であり、そこには“柔らかさ”があります。そのため、頭の固いコンピュータに自然言語を理解させるということは、一筋縄ではいかない難しい問題なのです。ですが、その難問をクリアできれば——コンピュータに自然言語を理解させることができれば——、人にとって役に立つことをコンピュータに行わせることができます。実際、そのような例はたくさん見ることができます。たとえば、検索エンジンや機械翻訳などは分かりやすい例でしょう。その他にも、質問応答システムやIME（かな漢字変換）、文章の自動要約や感情分析など、私たちの身の回りではすでに自然言語処理の技術が数多く使われています。



自然言語処理のアプリケーションのひとつに「質問応答システム」があります。その代表例として、IBM のWatson^{ワトソン}が有名です。Watsonの名を広く世に知らしめたのは、2011年のアメリカのクイズ番組『ジェバディ！』でした。その番組において、Watsonは誰よりもクイズに的確に答え、歴代のチャンピオンに勝利したのです（ただし、Watsonには問題文がテキストで与えられるというアドバンテージがありました）。この“事件”は世間から多くの注目を集め、この時期あたりから、人工知能に対する世間の期待と不安が高まってきたように感じます。なお、IBMはWatsonを「意思決定支援システム」とも呼んでおり、最近では、過去の膨大な医療データを活用して、難病患者の正しい治療法を提案し、患者の命を救った事例が報告されています。

2.1.1 単語の意味

私たちの言葉は「文字」によって構成されます。そして、私たちの言葉の意味は「単語」によって構成されます。単語は、言ってみれば、意味の最小単位です。そのため、自然言語をコンピュータに理解させるためには、他でもなく「単語の意味」を理解させることが重要であると言えそうです。

本章のテーマは、コンピュータに「単語の意味」を理解させることです。より正確に言うと、「単語の意味」をうまく捉えた表現方法について考えていきます。具体的には、本章と次章で次の3つの手法を見ていきます。

- シソーラスによる手法 本章
- カウントベースの手法 本章
- 推論ベースの手法 (word2vec) 次章

最初に、人の手によって作られたシソーラス（類語辞書）を利用する方法について簡単に見ていきます。続いて、統計情報から単語を表現する手法——ここでは、「カウントベースの手法」と呼ぶことにします——について説明します。ここまでが本章で学ぶ内容です。そして次章では、ニューラルネットワークによる「推論ベース」の手法（具体的には、word2vecと呼ばれる手法）を扱います。なお、本章の構成は、スタンフォード大学の授業「CS224d: Deep Learning for Natural Language Processing」[10]を参考にしています。

2.2 シソーラス

「単語の意味」を表すためには、人の手によって単語の意味を定義することが考えられます。そのひとつ的方法としては、『広辞苑』などの辞書のように、一つひとつの単語に対してその単語の意味を説明していくことが考えられるでしょう。たとえば、「自動車」という単語を辞書で引くと、「車輪を取り付けてそれによって進むようになっている乗り物や運搬具……」のような説明が続きます。そのように単語を定義すれば、コンピュータも単語の意味を理解できるかもしれません。

自然言語処理の歴史を振り返ってみると、単語の意味を人手で定義するという試みは数多く行われてきました。ただし、『広辞苑』のように人が使う一般的な辞書ではなく、**シソーラス** (thesaurus) と呼ばれるタイプの辞書が多く使われてきました。

シソーラスとは（基本的には）類語辞書であり、「同じ意味の単語（同義語）」や「意味の似た単語（類義語）」が同じグループに分類されています。たとえば、car の同義語には、automobile や motorcar などが存在することが、シソーラスを使えば分かります（図2-1 参照）。

| | | | | | |
|-----|---|------|------------|---------|----------|
| car | = | auto | automobile | machine | motorcar |
|-----|---|------|------------|---------|----------|

図2-1 同義語の例：car、auto、automobile などは、「自動車」を表す同義語

また、自然言語処理において利用されるシソーラスでは、単語の間で、「上位と下位」、「全体と部分」などの、より細かい関連性が定義されている場合があります。具体的には、図2-2 のように、グラフ構造によって各単語の関係性が定義されているケースです。

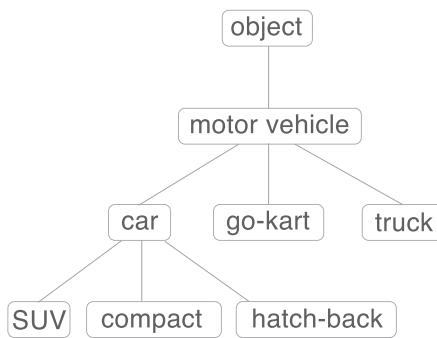


図2-2 各単語の意味に対して、上位・下位の関係性に基づき、グラフを形成する（図は文献[14]を参考に作成）

図2-2 では、「car」という単語の上位概念として、「motor vehicle（動力車）」という単語が位置しています。また、「car」の下位の概念に、「SUV」や「compact」、「hatch-back」などの、より具体的な車種があることが示されています。

このようにすべての単語に対して、類義語の集合を作り、また、それぞれの単語の関係をグラフで表現することで、単語間のつながりを定義できます。私たちはこの“単語ネットワーク”を利用することで、コンピュータに単語間の関連性を教えるこ

とができます。これは、コンピュータに単語の意味を（間接的にであれ）授けることができたと言えそうです。そして、その知識を利用すれば、私たちにとって有用なことをコンピュータに行わせることができるでしょう。



シソーラスをどのように利用するかは、自然言語処理のアプリケーションによって変わってきます。たとえば、情報検索において利用するシーンでは、automobile と car が類義語であることを知つていれば、automobile の検索結果を car の検索結果に含めることができます。

2.2.1 WordNet

自然言語処理の分野において、最も有名なシソーラスは [WordNet](#)[17]です。WordNet はプリンストン大学で 1985 年に開発がスタートした伝統あるシソーラスで、これまでに多くの研究で利用されてきました。また、さまざまな自然言語処理のアプリケーションにおいても大いに活躍しています。

WordNet を使えば、類義語を取得したり、「単語ネットワーク」を利用したりすることができます。また、単語ネットワークを使って単語間の類似度を算出することも可能です。ここでは、WordNet について詳細な説明は行いません。WordNet を使った Python の実装について興味のある方は、「付録 B WordNet を動かす」を参照してください。付録 B では、WordNet をインストールして（正確には NLTK というモジュールをインストールして）、いくつかの簡単な実験を行います。



「付録 B WordNet を動かす」では、実際に WordNet を使って、単語の類似度を求める実験を行います。そこでは、人の手によって定義された“単語ネットワーク”を元に、単語間の類似度を算出できる事例を見ていきます。単語の類似度を（ある程度正しく）求めることができるならば、それは「単語の意味」の理解への第一歩と言えるでしょう。

2.2.2 シソーラスの問題点

WordNet のようなシソーラスでは、多くの単語に対して同義語や階層構造などの関係性が定義されています。そして、それらの知識を利用すれば、「単語の意味」というものを（間接的にであれ）コンピュータに授けることができそうです。しかし、それらの人の手によるラベル付けには大きな欠点が存在します。以下に、シソーラスによる手法の問題点を列挙し、それぞれに簡単な説明を加えます。

時代の変化に対応するのが困難

私たちの使う言葉は生きています。時とともに新しい言葉が生まれ、ホコリをかぶった古い言葉はいつの日か忘れ去られます。たとえば、「クラウド・ファンディング (crowdfunding)」という言葉は、最近使われ始めた新しい造語です。

また、時代によって言葉の意味が変化するケースもあります。たとえば、英語の「heavy (ヘビー)」という単語には、「(ものごとが) 深刻である」という意味があります（これは主に俗語として用いられます）。しかし、昔はそのような使い方はされませんでした。映画『バック・トゥ・ザ・フューチャー』では、1985年 の未来から来たマーティと 1955 年に住むドクとの間で「ヘビー」の意味が通じないシーンがあります。そのような単語の変化に対応するのであれば、シソーラスを人手によって絶えず更新しなければなりません。

人の作業コストが高い

シソーラスを作るには、大変な人的コストが発生します。英語を例に挙げると、現存する英単語の総数は 1,000 万語を超えると言われています。そのため、理想的にはそのような膨大な単語に対して、単語の関連付けを行う必要があるのです。ちなみに WordNet では、20 万語を超える単語が登録されているそうです。

単語の細かなニュアンスを表現できない

シソーラスでは、類義語として似たような単語をグループ化します。しかし実際には、似たような単語であっても、それぞれにニュアンスは異なるものです。たとえば、「ヴィンテージ」という単語と「レトロ」という単語は同じような意味を表しますが、その使われ方は異なります。シソーラスでは、このような微妙なニュアンスの差異を表すことができません（もし人手で表すとすれば、それは相当に困難なものになるでしょう）。

このように、シソーラスを使う手法——言い換えると、単語の意味を人手によって定義する手法——には大きな問題があります。このような問題を避けるために、続けて「カウントベースの手法」、そして、ニューラルネットワークを使った「推論ベースの手法」へと進みます。それら 2 つの手法では、大量のテキストデータから自動的に「単語の意味」を抽出します。そのため、人の手作業によって単語を関連付けるというような重労働から解放されるのです！



自然言語処理に限らず、画像認識の分野においても、人の手によって特徴量を設計するという作業が長年行われてきました。それがディープラーニングによって、生の画像から目的の結果を直接得ることができるようになり、人が介入する必要性が格段に減りました。自然言語処理の分野においても、同じような現象が起こっています。つまり、人手によるシソーラスや素性（特徴量）の設計から、人は極力介入せずにテキストデータだけから目的の結果を得るパラダイムへと移行しているのです。

2.3 カウントベースの手法

これからカウントベースの手法に進むにあたって、私たちはコーパス (corpus) を利用します。コーパスとは、簡単に言ってしまえば、大量のテキストデータです。ただし、やみくもに集められたテキストデータではなく、自然言語処理の研究やアプリケーションのために目的をもって収集されたテキストデータを、一般的には「コーパス」と呼びます。

結局のところ、コーパスとはテキストデータにすぎません。しかし、そこに含まれる文章は人の手によって書かれたものです。これは別の見方をすれば、コーパスには自然言語に対する人の“知識”がふんだんに含まれているということです。文章の書き方、単語の選び方、そして、単語の意味——そのような人の自然言語に対する知識が、コーパスには含まれています。カウントベースの手法の目標は、そのような人の知識が詰まったコーパスから、自動的に、そして効率良く、そのエッセンスを抽出することにあります。



自然言語処理の分野で使われるコーパスは、テキストデータに対して、さらに追加の情報が与えられている場合があります。たとえば、テキストデータの個々の単語に対して「品詞」がラベル付けされているようなケースなどです。その場合、コンピュータが扱いやすいように、コーパスは構造化されて（たとえば、木構造などのデータ形式で）与えられるのが一般的です。今回私たちが使用するコーパスは、そのような追加のラベルは用いず、単なるテキストデータとして——ひとつの大きなテキストファイルとして——与えられることを想定します。

2.3.1 Python によるコーパスの下準備

自然言語処理の分野で用いられるコーパスには、さまざまなもののが存在します。有

名なもので言えば、Wikipedia や Google News などのテキストデータが挙げられます。また、シェイクスピアや夏目漱石など、偉大な作家の作品群もコーパスとして利用されます。本章では、まず初めに 1 文からなる単純なテキストをコーパスとして利用します。その後で、より実用的なコーパスを扱います。

それでは Python の対話モードを使って、とても小さなテキストデータ（コーパス）に前処理を行います。ここで言う前処理とは、テキストデータを単語に分割し、その分割した単語を単語 ID のリストへと変換することです。

では、ひとつずつ確認しながら、ステップ・バイ・ステップで組み立てていきましょう。まずは、今回コーパスとして利用するサンプルの文章からです。

```
>>> text = 'You say goodbye and I say hello.'
```

ここではひとつの文からなるテキストをコーパスとして利用します。本来であれば、このテキスト (`text`) には何千、何万を超える文が（連結されて）含まれるでしょう。しかし、ここでは簡易性を優先して、この小さなテキストデータに対して前処理を行っていきます。それでは、上の `text` を単語単位に分割しましょう。

```
>>> text = text.lower()
>>> text = text.replace('. ', '.')
>>> text
'you say goodbye and i say hello .'

>>> words = text.split(' ')
>>> words
['you', 'say', 'goodbye', 'and', 'i', 'say', 'hello', '.']
```

ここでは最初に、すべての文字を小文字に変換するために `lower()` メソッドを使います。これは、文頭の単語でも共通の単語として扱えるようにするための処置です。そして、`split(')` によって、スペースを「区切り文字」として分割します。ただしここでは文末のピリオド (.) を考慮して、ピリオドの前にスペースを挿入して——正しくは「.」を「.」に置換して——から分割を行っています。



ここで単語の分割を行うにあたって、ピリオドの前にスペースを入れるような「その場しのぎの実装」を行いました。これには、もっと賢く汎用性の高いやり方があります。それは「正規表現」を利用する方法です。ひとつの方法として、正規表現の `re` モジュールをインポートして、`re.split('(\W+)?', text)` とすることで、単語単位の分割ができます。正規表現の詳細については書籍『詳説 正規表現 第3版』[15]などを参考にしてください。

これで、元の文章を単語のリストとして利用できるようになりました。単語が分割されたことで扱いやすくなりましたが、単語をテキストのまま操作するのは何かと不便です。そこで、単語に ID を振って、ID のリストとして利用できるように、さらに手を加えます。その下準備として、単語の ID と単語の対応表を Python のディクショナリで作成します。

```
>>> word_to_id = {}
>>> id_to_word = {}
>>>
>>> for word in words:
...     if word not in word_to_id:
...         new_id = len(word_to_id)
...         word_to_id[word] = new_id
...         id_to_word[new_id] = word
```

単語 ID から単語への変換を `id_to_word` という変数が担当し（キーが単語 ID で、値が単語）、単語から単語 ID への変換を `word_to_id` が担当します。ここでは単語単位に分割された `words` の各要素を先頭からひとつずつ見ていき、単語が `word_to_id` に存在しない場合は、`word_to_id` と `id_to_word` に新しい ID と単語をそれぞれ追加します。なお、ディクショナリの長さを新しい単語 ID として設定するため、単語 ID は、0、1、2……と増えていくことになります。

これで、単語 ID と単語の対応表を作ることができました。それでは、実際に中身を見てみましょう。

```
>>> id_to_word
{0: 'you', 1: 'say', 2: 'goodbye', 3: 'and', 4: 'i', 5: 'hello', 6: '.'}
>>> word_to_id
{'you': 0, 'say': 1, 'goodbye': 2, 'and': 3, 'i': 4, 'hello': 5, '.': 6}
```

これらのディクショナリを使えば、単語から単語 ID を検索したり、逆に単語 ID から単語を検索したりすることができます。実際にやってみると、次のようになります。

```
>>> id_to_word[1]
'say'
>>> word_to_id['hello']
5
```

それでは最後に、「単語のリスト」を「単語 ID のリスト」に変換しましょう。ここでは、Python の内包表記を使って、単語リストから単語 ID リストへ変換します。

そして、それを NumPy 配列に変換します。

```
>>> import numpy as np
>>> corpus = [word_to_id[w] for w in words]
>>> corpus = np.array(corpus)
>>> corpus
array([0, 1, 2, 3, 4, 1, 5, 6])
```



内包表記とは、リストやディクショナリなどのループ処理をシンプルに書くための記法です。たとえば、`xs = [1,2,3,4]` というリストの各要素を 2 乗して新しいリストを作るとすれば、`[x**2 for x in xs]` と書けます。

これで、コーパスを利用する下準備が整いました。それでは以上の処理を `preprocess()` という名前の関数として、まとめて実装することにします (➡ `common/util.py`)。

```
def preprocess(text):
    text = text.lower()
    text = text.replace('。', '。')
    words = text.split('。')

    word_to_id = {}
    id_to_word = {}
    for word in words:
        if word not in word_to_id:
            new_id = len(word_to_id)
            word_to_id[word] = new_id
            id_to_word[new_id] = word

    corpus = np.array([word_to_id[w] for w in words])

    return corpus, word_to_id, id_to_word
```

この関数を使えば、コーパスの前処理は次のように行うことができます。

```
>>> text = 'You say goodbye and I say hello.'
>>> corpus, word_to_id, id_to_word = preprocess(text)
```

これでコーパスの前処理は終わりです。ここで準備した `corpus`、`word_to_id`、`id_to_word` という 3 つの変数名は、これから先も本書の多くの場所で登場します。`corpus` は単語 ID のリスト、`word_to_id` は単語から単語 ID へのディクショナリ、`id_to_word` は単語 ID から単語へのディクショナリを表します。

以上でコーパスを扱う準備が整いました。私たちのこれから の目標は、コーパスを

使って「単語の意味」を抽出することです。そのために本節では「カウントベースの手法」を見ていきます。この手法によって、私たちは単語をベクトルで表すことができるようになります。

2.3.2 単語の分散表現

突然ですが、世の中にはさまざまな「色」が存在します。たとえば、「コバルトブルー」や「シンクレッド」といったように、色には固有の名前が付けられています。その一方で、色は RGB (Red/Green/Blue) の 3 成分がどれだけ存在するかといった方法でも表すことができます。前者が色の数だけ異なる名前を命名する一方で、後者は、色を 3 次元のベクトルとして表現します。

ここで注目したい点は、RGB のようなベクトル表現のほうが、正確に色を指定できるということです。さらに、3つの成分というコンパクトな表現が可能であり、(多くの場合) 色のイメージがつきやすいという利点もあります。たとえば、「こきあけ深緋」という色がどのような色か分からなくても、 $(R, G, B) = (201, 23, 30)$ と言われれば、それは赤系の色だということが分かります。また、色どうしの関連性——似た色かどうかなど——もベクトル表現のほうが容易に判断しやすく、定量化も簡単に行えます。

それでは、ここで話した「色」のベクトル表現のようなことを、「単語」でも行えないでしょうか？ より正確に言うならば、コンパクトで理にかなったベクトル表現を、「単語」というフィールドにおいても構築できないでしょうか？ これから私たちが目指すべき場所は、「単語の意味」を的確に捉えたベクトル表現です。これは自然言語処理の分野では、単語の分散表現と呼ばれます。



単語の分散表現は、単語を固定長のベクトルで表現します。そして、そのベクトルは密なベクトルで表されるのが特徴です。密なベクトルというのは、ベクトルの各要素（多く）が 0 ではない実数値として表されることを意味します。たとえば、3 次元の分散表現は、 $[0.21, -0.45, 0.83]$ のようになります。このような単語の分散表現をどのように構築するかということが、これから重要なテーマです。

2.3.3 分布仮説

自然言語処理の歴史において、単語をベクトルで表す研究は数多く行われてきました。そのような研究を見ていくと、重要な手法のはほとんどすべてが、あるひとつのシンプルなアイデアに基づいていることが分かります。そのアイデアとは、「単

語の意味は、周囲の単語によって形成される」というものです。これは、**分布仮説** (distributional hypothesis) と呼ばれるもので、単語をベクトルで表す最近の研究の多くが、この仮説に基づいています。

単語の意味は、周囲の単語によって形成される——分布仮説が言わんとすることは、とてもシンプルです。単語自体には意味がなく、その単語の「コンテキスト（文脈）」によって、単語の意味が形成されると言うのです。確かに、意味的に同じ単語は、同じような文脈で多く出現します。たとえば、「I drink beer.」「We drink wine.」のように drink の近くには飲み物が表れやすいでしょう。また、「I guzzle beer.」「We guzzle wine.」のような文章があれば、guzzle という単語が drink と同じような文脈で使われるということが分かります。そして、guzzle と drink が近い意味の単語だということも導けます（ちなみに、guzzle とは「がぶがぶ飲む」という意味の単語です）。

さて、これから先、「コンテキスト」という言葉を多く使います。本章で「コンテキスト」と言うとき、それは（注目する単語に対して）その周囲に存在する単語を指します。たとえば、図2-3の例では、左右の2単語が「コンテキスト」に相当します。



図2-3 ウィンドウサイズが2の「コンテキスト」の例。「goodbye」という単語に注目したとき、その左右の2単語をコンテキストとして利用する

図2-3に示すとおり、「コンテキスト」とは、ある中央の単語に対して、その周囲にある単語を指します。またここでは、コンテキストのサイズ——つまり、周囲の単語をどれだけ含めるかということ——を、「ウィンドウサイズ (window size)」という言葉で表すことになります。ウィンドウサイズが1の場合は左右の1単語、ウィンドウサイズが2の場合は左右の2単語、といったようになります。



ここでは、コンテキストとして左右の単語を均等に含めました。しかし状況によっては、左の単語だけや右の単語だけをコンテキストとして使用する場合も考えられます。また、文の区切りを考慮したコンテキストを考えることもできます。本書では単純さを優先して、文の区切りを考慮せず、左右均等のコンテキストのみを扱います。

2.3.4 共起行列

それでは、分布仮説に基づいて、単語をベクトルで表す方法を考えましょう。そのための素直な方法は、周囲の単語を“カウント”することです。具体的に言うと、ある単語に着目した場合、その周囲にどのような単語がどれだけ現れるのかをカウントし、それを集計するのです。ここではこれを「カウントベースの手法」と呼ぶことにします。なお、これは文献によっては「統計的手法」と呼ばれる場合もあります。

それでは、カウントベースの手法を見ていきましょう。ここでは、「2.3.1 Pythonによるコーパスの下準備」のコーパスと `preprocess()` 関数を使用して、再度下準備を行うところからスタートします。

```
import sys
sys.path.append('..')
import numpy as np
from common.util import preprocess

text = 'You say goodbye and I say hello.'
corpus, word_to_id, id_to_word = preprocess(text)

print(corpus)
# [0 1 2 3 4 1 5 6]

print(id_to_word)
# {0: 'you', 1: 'say', 2: 'goodbye', 3: 'and', 4: 'i', 5: 'hello', 6:
→ '.'}
```

上の結果より、ここでは語彙数が全部で 7 個あることが分かります。それでは、それぞれの単語について、そのコンテキストに含まれる単語の頻度を数えていきましょう。ここでは、ウィンドウサイズを 1 として、まずは単語 ID が 0 の「you」からスタートします。

you say goodbye and i say hello .

図2-4 単語「you」のコンテキストをカウントする

図2-4を見れば一目瞭然ですが、単語「you」のコンテキストには、「say」という単語がひとつだけ存在することが分かります。そのため、これをテーブルで表せば、

図2-5 のようになります。

| | | | | | | | |
|-----|-----|-----|---------|-----|---|-------|---|
| | you | say | goodbye | and | i | hello | . |
| you | 0 | 1 | 0 | 0 | 0 | 0 | 0 |

図2-5 単語「you」のコンテキストに含まれる単語の頻度を、テーブルで表す

図2-5 は、単語「you」に対して、コンテキストとして共起する単語の頻度を表したものです。そしてこれは同時に、「you」という単語が $[0, 1, 0, 0, 0, 0, 0]$ というベクトルで表現できるということも意味します。

それでは、続いて単語 ID が 1 の「say」について、同じ作業を行います。その結果は、図2-6 のようになります。

you say goodbye and i say hello .

| | | | | | | | |
|-----|-----|-----|---------|-----|---|-------|---|
| | you | say | goodbye | and | i | hello | . |
| say | 1 | 0 | 1 | 0 | 1 | 1 | 0 |

図2-6 単語「say」のコンテキストに含まれる単語の頻度を、テーブルで表す

上の結果から、「say」という単語は、 $[1, 0, 1, 0, 1, 1, 0]$ というベクトルで表現できることが分かります。以上の作業を、すべての単語について——ここでは 7 つの単語について——行います。その結果は、図2-7 のようになります。

| | you | say | goodbye | and | i | hello | . |
|---------|-----|-----|---------|-----|---|-------|---|
| you | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| say | 1 | 0 | 1 | 0 | 1 | 1 | 0 |
| goodbye | 0 | 1 | 0 | 1 | 0 | 0 | 0 |
| and | 0 | 0 | 1 | 0 | 1 | 0 | 0 |
| i | 0 | 1 | 0 | 1 | 0 | 0 | 0 |
| hello | 0 | 1 | 0 | 0 | 0 | 0 | 1 |
| . | 0 | 0 | 0 | 0 | 0 | 1 | 0 |

図2-7 各単語について、そのコンテキストに含まれる単語の頻度をカウントし、テーブルにまとめる

図2-7は、すべての単語に対して、共起する単語をテーブルにまとめたものです。このテーブルの各行が、該当する単語のベクトルに対応します。なお、図2-7のテーブルは行列の形をしているため、**共起行列**（co-occurrence matrix）と呼ばれます。

それでは、上の共起行列を実際に作ってみましょう。ここでは、図2-7の結果をそのまま手で打ち込むことにします。

```
C = np.array([
    [0, 1, 0, 0, 0, 0, 0],
    [1, 0, 1, 0, 1, 1, 0],
    [0, 1, 0, 1, 0, 0, 0],
    [0, 0, 1, 0, 1, 0, 0],
    [0, 1, 0, 1, 0, 0, 0],
    [0, 1, 0, 0, 0, 0, 1],
    [0, 0, 0, 0, 0, 1, 0],
    [0, 0, 0, 0, 0, 0, 0]
], dtype=np.int32)
```

これで共起行列ができました。この共起行列を使えば、各単語のベクトルは次のようにして得ることができます。

```
print(C[0]) # 単語IDが0のベクトル
# [0 1 0 0 0 0 0]

print(C[4]) # 単語IDが4のベクトル
# [0 1 0 1 0 0 0]
```

```
print(C[word_to_id['goodbye']]) # 「goodbye」のベクトル
# [0 1 0 1 0 0]
```

ここで示したように、私たちは共起行列によって単語をベクトルで表すことができました。ここでは手動で共起行列を作りましたが、もちろんこれは自動化できます。それでは、コーパスから共起行列を作る関数を実装しましょう。ここでは、`create_co_matrix(corpus, vocab_size, window_size=1)` という関数名で実装します。引数の `corpus` は単語 ID のリスト、`vocab_size` は語彙数、`window_size` はウィンドウサイズを表します (☞ `common/util.py`)。

```
def create_co_matrix(corpus, vocab_size, window_size=1):
    corpus_size = len(corpus)
    co_matrix = np.zeros((vocab_size, vocab_size), dtype=np.int32)

    for idx, word_id in enumerate(corpus):
        for i in range(1, window_size + 1):
            left_idx = idx - i
            right_idx = idx + i

            if left_idx >= 0:
                left_word_id = corpus[left_idx]
                co_matrix[word_id, left_word_id] += 1

            if right_idx < corpus_size:
                right_word_id = corpus[right_idx]
                co_matrix[word_id, right_word_id] += 1

    return co_matrix
```

ここでは、まず初めに `co_matrix` を要素が 0 の 2 次元配列で初期化します。後は、コーパス中の各単語すべてに対して、そのウィンドウに含まれる単語をカウントしていきます。このとき、コーパスの左端と右端において、はみ出しているかのチェックを行います。

これでコーパスがどれだけ大きくなったとしても、自動で共起行列を作ることができます。これ以降、私たちはこの関数を使ってコーパスの共起行列を作成します。

2.3.5 ベクトル間の類似度

私たちは、共起行列によって単語をベクトルで表すことができました。それでは続いて、ベクトル間の類似度を計測する方法を見ていきます。

ベクトル間の類似度を計測するには、さまざまな方法が考えられます。たとえ

ば、ベクトルの内積やユークリッド距離などが代表例として挙げられます。その他にもさまざまな方法がありますが、単語のベクトル表現の類似度に関しては、**コサイン類似度** (cosine similarity) がよく用いられます。コサイン類似度は、 $\mathbf{x} = (x_1, x_2, x_3, \dots, x_n)$ と $\mathbf{y} = (y_1, y_2, y_3, \dots, y_n)$ の 2 つのベクトルがあるとき、次の式で定義されます。

$$\text{similarity}(\mathbf{x}, \mathbf{y}) = \frac{\mathbf{x} \cdot \mathbf{y}}{\|\mathbf{x}\| \|\mathbf{y}\|} = \frac{x_1 y_1 + \dots + x_n y_n}{\sqrt{x_1^2 + \dots + x_n^2} \sqrt{y_1^2 + \dots + y_n^2}} \quad (2.1)$$

式 (2.1) では、分子にベクトルの内積、分母に各ベクトルの「ノルム」があります。ノルムはベクトルの大きさを表したもので、ここでは「L2 ノルム」を計算します (L2 ノルムは、ベクトルの各要素の 2 乗和の平方根によって計算されます)。式 (2.1) のポイントは、ベクトルを正規化してから内積を取ることです。



コサイン類似度は、直感的には「2 つのベクトルがどれだけ同じ方向を向いているか」を表します。2 つのベクトルが完全に同じ方向を向いているときコサイン類似度は 1 になり、完全に逆向きだと -1 になります。

それでは、コサイン類似度を実装してみましょう。式 (2.1) に従って実装すると、次のように書けます。

```
def cos_similarity(x, y):
    nx = x / np.sqrt(np.sum(x**2)) # xの正規化
    ny = y / np.sqrt(np.sum(y**2)) # yの正規化
    return np.dot(nx, ny)
```

ここでは、引数の x と y は NumPy 配列であることを想定します。中身の実装は、初めにベクトルの正規化を行い、その後に、2 つのベクトルの内積を求めます。これでコサイン類似度の実装は終わりですが、この実装にはひとつ問題があります。それは、ゼロベクトル（ベクトルの要素がすべて 0 のベクトル）が引数に入ると、“0 除算” が発生してしまうことです。

このような問題に対しての常套手段は、除算を行う際に小さな値を加算することです。ここでは、小さな値として eps を引数で指定するようにして、何も指定しない場合は $\text{eps}=1e-8$ ($= 0.00000001$) が設定されるように修正します (eps は「イプシロン」の略です)。それでは、改善版の実装を次に示します ([common/util.py](#))。

```
def cos_similarity(x, y, eps=1e-8):
    nx = x / (np.sqrt(np.sum(x ** 2)) + eps)
    ny = y / (np.sqrt(np.sum(y ** 2)) + eps)
    return np.dot(nx, ny)
```



ここで小さな値として `1e-8` を用いましたが、これぐらい小さな値であれば、通常は浮動小数点数の「丸め誤差」により、他の値に“吸収”されます。上の実装においては、その小さな値はベクトルのノルムに“吸収”されることになるため、ほとんどの場合、`eps` の加算は最終的な計算結果に影響を与えません。その一方で、ベクトルのノルムが 0 のときは、その小さな値がそのまま残り、“0 除算”を防いでくれます。

この関数を用いると、単語ベクトルの類似度は次のように求めることができます。ここでは、「you」と「i (= I)」の類似度を求めてみます ([ch02/similarity.py](#))。

```
import sys
sys.path.append('..')
from common.util import preprocess, create_co_matrix, cos_similarity

text = 'You say goodbye and I say hello.'
corpus, word_to_id, id_to_word = preprocess(text)
vocab_size = len(word_to_id)
C = create_co_matrix(corpus, vocab_size)

c0 = C[word_to_id['you']] # 「you」の単語ベクトル
c1 = C[word_to_id['i']] # 「i」の単語ベクトル
print(cos_similarity(c0, c1))
# 0.7071067691154799
```

上記の結果から、「you」と「i」のコサイン類似度は、0.70... という結果になりました。コサイン類似度は 1 から -1 までを取るので、この値は、比較的高い値（類似性がある）と言えそうです。

2.3.6 類似単語のランキング表示

コサイン類似度の実装が完了したので、その関数を使って別の便利な関数を実装したいと思います。それは、ある単語がクエリとして与えられたときに、そのクエリに対して類似した単語を上位から順に表示する関数です。ここでは、その関数名を `most_similar()` と呼び、次のような引数で実装することにします ([表2-1](#))。

```
most_similar(query, word_to_id, id_to_word, word_matrix, top=5)
```

表2-1 most_similar() 関数の引数

| 引数名 | 説明 |
|-------------|---|
| query | クエリ（単語） |
| word_to_id | 単語から単語 ID へのディクショナリ |
| id_to_word | 単語 ID から単語へのディクショナリ |
| word_matrix | 単語ベクトルをまとめた行列。各行に対応する単語のベクトルが格納されていることを想定する |
| top | 上位何位まで表示するか |

早速、most_similar() 関数の実装を次に示します（[common/util.py](#)）。

```
def most_similar(query, word_to_id, id_to_word, word_matrix, top=5):
    # ①クエリを取り出す
    if query not in word_to_id:
        print('%s is not found' % query)
        return

    print('\n[query] ' + query)
    query_id = word_to_id[query]
    query_vec = word_matrix[query_id]

    # ②コサイン類似度の算出
    vocab_size = len(id_to_word)
    similarity = np.zeros(vocab_size)
    for i in range(vocab_size):
        similarity[i] = cos_similarity(word_matrix[i], query_vec)

    # ③コサイン類似度の結果から、その値を高い順に出力
    count = 0
    for i in (-1 * similarity).argsort():
        if id_to_word[i] == query:
            continue
        print(' %s: %s' % (id_to_word[i], similarity[i]))

        count += 1
        if count >= top:
            return
```

上の実装は、次の手順によって行います。

- ① クエリの単語ベクトルを取り出す
- ② クエリの単語ベクトルと、他のすべての単語ベクトルとのコサイン類似度をそれぞれ求める
- ③ コサイン類似度の結果に対して、その値が高い順に表示する

ここでは、③のコードについてだけ補足します。③では、`similarity` 配列中の要素のインデックスを高い順に並び替え、その上位を出力します。このとき配列のインデックスの並び替えには、`argsort()` というメソッドを使います。この `argsort()` メソッドは、NumPy 配列の要素を小さい順にソートします（ただし、返り値は配列のインデックス）。ひとつ例を示すと、次のようにになります。

```
>>> x = np.array([100, -20, 2])
>>> x.argsort()
array([1, 2, 0])
```

ここでは `[100, -20, 2]` という NumPy 配列に対して、各要素の値を小さい順にソートします。このとき返り値の配列の各要素は、配列のインデックスに対応します。上の結果は、「1番目の要素（-20）」、「2番目の要素（2）」、「0番目の要素（100）」の順になります。ここで私たちが行いたいことは、単語の類似度を“大きい”順にソートすることです。そのため、NumPy 配列の各要素にマイナスをかけた後で、`argsort()` メソッドを利用します。上の例に續いて示すと、次のようにになります。

```
>>> (-x).argsort()
array([0, 2, 1])
```

この `argsort()` を使うことで、単語の類似度の高い順に出力することができます。以上が、`most_similar()` 関数の実装です。それでは、`most_similar()` 関数を使ってみましょう。ここでは「you」という単語をクエリとして、類似する単語を表示させてみます。コードで示すと、次のようにになります（☞ ch02/most_similar.py）。

```
import sys
sys.path.append('..')
from common.util import preprocess, create_co_matrix, most_similar

text = 'You say goodbye and I say hello.'
corpus, word_to_id, id_to_word = preprocess(text)
vocab_size = len(word_to_id)
C = create_co_matrix(corpus, vocab_size)

most_similar('you', word_to_id, id_to_word, C, top=5)
```

これを実行すると、次の結果が得られます。

```
[query] you
goodbye: 0.7071067691154799
```

```
i: 0.7071067691154799
hello: 0.7071067691154799
say: 0.0
and: 0.0
```

この結果は、「you」という単語をクエリに対して類似単語を上位から順に5つだけ表示したものです。このとき各単語の隣にある値はコサイン類似度です。上の結果を見ると、「you」に一番近い単語は3つあり、それは「goodbye」、「i (= I)」、「hello」となっています。確かに、「I」と「you」はともに人称代名詞であるため、その結果には納得できるでしょう。しかし、「goodbye」や「hello」のコサイン類似度の値が高いというのは、私たちの感覚と大きなズレがあります。もちろんこれは、コーパスのサイズが極端に小さいことに原因があります。後ほど、より大きなコーパスを使って同じ実験を行います。

さて、これまで見てきたように、私たちは、単語を共起行列によってベクトルで表すことができました。これでカウントベースの手法の「基本」は終わりです。「基本」というだけあって、まだまだ話すことはいくつも残されています。次節では、現状の手法をさらに改善するためのアイデアを説明し、実際にその改善案を実装していきます。

2.4 カウントベースの手法の改善

前節で私たちは、単語の共起行列を作りました。それによって単語をベクトルで表すことには成功しました。しかし、その共起行列にはまだまだ改善すべきところがあります。本節ではその改善に取り組みます。そして、その改善が済んだところで、より実用的なコーパスを使って“本物”的な単語の分散表現を手にしたいと思います。

2.4.1 相互情報量

前節の共起行列の要素は、2つの単語が共起した回数を表しています。しかし、この“生”の回数というのはあまり良い性質を持ちません。その理由は、高頻度単語（多く出現する単語）に目を向けるとはっきりします。

たとえば、あるコーパスにおいて「the」と「car」の共起を考えてみましょう。その場合、「... the car ...」というフレーズは多く見られるでしょう。そのため、その共起する回数は大きな値になります。一方、「car」と「drive」という単語には明らかに強い関連性があります。しかし、単に出現回数だけを見てしまうと、「car」は

「drive」よりも「the」のほうに強い関連性を持つてしまうでしょう。つまりこれは、「the」という単語が高頻度な単語であるがゆえに、「car」と強い関連性を持つように評価されてしまうということです。

そのような問題を解決するために、**相互情報量** (Pointwise Mutual Information) [19] と呼ばれる指標が使われます（以降、PMI と略記）。これは x と y という確率変数に対して次の式で定義されます（確率については「3.5.1 CBOW モデルと確率」で詳しく説明します）。

$$\text{PMI}(x, y) = \log_2 \frac{P(x, y)}{P(x)P(y)} \quad (2.2)$$

式 (2.2) が PMI の定義式です。ここで $P(x)$ は x が起こる確率、 $P(y)$ は y が起こる確率を表します。そして、 $P(x, y)$ は x と y が同時に起こる確率を表します。この PMI は、その値が高いほど関連性が高いことを示します。

これを私たちの自然言語の例に当てはめると、 $P(x)$ というのは x という単語がコーパスに現れる確率を指します。たとえば、10,000 個の単語からなるコーパスで「the」という単語が 100 回出現したとしましょう。そうすると、 $P("the") = \frac{100}{10000} = 0.01$ となります。また、 $P(x, y)$ は単語 x と y が共起する確率を表します。これも例を出すると、たとえば「the」と「car」が 10 回共起した場合、 $P("the", "car") = \frac{10}{10000} = 0.001$ となります。

それでは共起行列（各要素は共起した単語の回数）を使って、式 (2.2) を書き換えてみましょう。ここでは共起行列を C として、単語 x と y の共起する回数を $C(x, y)$ で表します。また、単語 x 、 y の出現する回数はそれぞれ $C(x)$ 、 $C(y)$ で表します。このとき、コーパスに含まれる単語数を N とすると、式 (2.2) は次のように書き換えられます。

$$\text{PMI}(x, y) = \log_2 \frac{P(x, y)}{P(x)P(y)} = \log_2 \frac{\frac{C(x, y)}{N}}{\frac{C(x)}{N} \frac{C(y)}{N}} = \log_2 \frac{C(x, y) \cdot N}{C(x)C(y)} \quad (2.3)$$

式 (2.3) によって、共起行列から PMI を求めることができます。それでは式 (2.3) を使って、具体的に計算してみましょう。ここでは、コーパス中の単語数 (N) を 10,000 として、「the」が 1,000 回、「car」が 20 回、「drive」が 10 回出現したとします。そして、「the」と「car」の共起が 10 回、「car」と「drive」の共起が 5 回だと仮定しましょう。このとき、共起する回数の視点では、「car」は、「drive」よりも「the」のほうに関連性が強いことになります。では、PMI の視点ではどうでしょう

か。これは次のように計算できます。

$$\text{PMI}(\text{"the"}, \text{"car"}) = \log_2 \frac{10 \cdot 10000}{1000 \cdot 20} \approx 2.32 \quad (2.4)$$

$$\text{PMI}(\text{"car"}, \text{"drive"}) = \log_2 \frac{5 \cdot 10000}{20 \cdot 10} \approx 7.97 \quad (2.5)$$

この結果が示すとおり、PMI を用いることで、「car」は、「the」よりも「drive」のほうに関連性を持つようになりました。これは私たちの要望どおりの結果です。このような結果になったのは、単語単独の出現回数が考慮されたことによります。この例では、「the」が多く出現していることから、PMI のスコアが低減されたのです。なお、式中の \approx は「ニアリー・イコール」という記号で、近似的に等しいことを意味します。

これで PMI という良い指標を手にできましたが、この PMI にはひとつ問題があります。それは 2 つの単語で共起する回数が 0 の場合、 $\log_2 0 = -\infty$ となってしまう点です。それに対応するため、実践上では次の正の相互情報量 (Positive PMI) が使われます（以降、PPMI と略記）。

$$\text{PPMI}(x, y) = \max(0, \text{PMI}(x, y)) \quad (2.6)$$

式 (2.6) により、PMI がマイナスのときは、それを 0 として扱います。これで、単語間の関連度を 0 以上の実数によって表すことができます。それでは、共起行列を PPMI 行列に変換する関数を実装しましょう。私たちはそれを `ppmi(C, verbose=False, eps=1e-8)` という名前で実装します ([common/util.py](#))。

```
def ppmi(C, verbose=False, eps=1e-8):
    M = np.zeros_like(C, dtype=np.float32)
    N = np.sum(C)
    S = np.sum(C, axis=0)
    total = C.shape[0] * C.shape[1]
    cnt = 0

    for i in range(C.shape[0]):
        for j in range(C.shape[1]):
            pmi = np.log2(C[i, j] * N / (S[j]*S[i]) + eps)
            M[i, j] = max(0, pmi)

    if verbose:
        cnt += 1
```

```

if cnt % (total//100 + 1) == 0:
    print('%.1f%% done' % (100*cnt/total))
return M

```

ここで引数の C は共起行列、verbose は進行状況を出力するかどうかを決めるフラグを表します。この verbose は大きなコーパスを扱う際に、verbose=True とすることで、進行状況を確認する用途に利用します。このコードでは、共起行列だけから PPMI 行列を求められるようにするために簡易的な実装を行っています。具体的には、単語 x と y の共起する回数を $C(x, y)$ としたとき、 $C(x) = \sum_i C(i, x)$ 、 $C(y) = \sum_i C(i, y)$ 、 $N = \sum_i \sum_j C(i, j)$ であるものとして——そのような近似を行い——実装しています。また、上のコードでは $\text{np.log2}(0)=-\infty$ を避けるために eps という微小な値を使用しています。



「2.3.5 ベクトル間の類似度」では、「0 除算」を防止するために微小な値を分母に追加しました。ここでも同様に、 $\text{np.log}(x)$ という計算を $\text{np.log}(x + \text{eps})$ とすることで、対数計算の無限小への発散を防止します。

それでは、共起行列を PPMI 行列に変換してみましょう。これは次のように実装できます (☞ ch02/ppmi.py)。

```

import sys
sys.path.append('..')
import numpy as np
from common.util import preprocess, create_co_matrix, cos_similarity,
→ ppmi

text = 'You say goodbye and I say hello.'
corpus, word_to_id, id_to_word = preprocess(text)
vocab_size = len(word_to_id)
C = create_co_matrix(corpus, vocab_size)
W = ppmi(C)

np.set_printoptions(precision=3) # 有効桁3桁で表示
print('covariance matrix')
print(C)
print(' -'*50)
print('PPMI')
print(W)

```

このファイルを実行すると、次の結果が得られます。

```
covariance matrix
[[0 1 0 0 0 0]
 [1 0 1 0 1 1 0]
 [0 1 0 1 0 0 0]
 [0 0 1 0 1 0 0]
 [0 1 0 1 0 0 0]
 [0 1 0 0 0 0 1]
 [0 0 0 0 0 1 0]]

-----
PPMI
[[ 0.      1.807   0.      0.      0.      0.      0.      ]
 [ 1.807   0.      0.807   0.      0.807   0.807   0.      ]
 [ 0.      0.807   0.      1.807   0.      0.      0.      ]
 [ 0.      0.      1.807   0.      1.807   0.      0.      ]
 [ 0.      0.807   0.      1.807   0.      0.      0.      ]
 [ 0.      0.807   0.      0.      0.      0.      2.807]
 [ 0.      0.      0.      0.      2.807   0.      0.      ]]
```

これで、共起行列を PPMI 行列に変換することができました。このとき、PPMI 行列の各要素は 0 以上の実数値です。これで私たちはより良い指標からなる行列——より良い単語ベクトル——を手にできたのです。

しかし、この PPMI 行列にも、まだ大きな問題があります。それは、コーパスの語彙数が増えるにつれて、各単語のベクトルの次元数も増えていくという問題です。たとえば、コーパスに含まれる語彙数が 10 万に達すれば、そのベクトルの次元数も同様に 10 万になります。実際のところ、10 万次元のベクトルを扱うというのはあまり現実的ではありません。

また、この行列の中身を見てみると、その要素の多くが 0 であることが分かります。これは、ベクトルのほとんどの要素が重要ではない——つまり、各要素の持つ“重要度”は低いということを意味します。そして、そのようなベクトルは、ノイズに弱く、頑健性に乏しいという欠点があります。こういった問題に対してよく行われるのが、ベクトルの次元削減です。

2.4.2 次元削減

次元削減 (dimensionality reduction) は、文字どおり、ベクトルの次元を削減する手法を指します。ただし、単に削減するのではなく、“重要な情報”をできるだけ残した上で削減するというところがポイントです。直感的なイメージとしては、図 2-8 に示すように、データの分布を見て重要な“軸”を見つけることを行います。

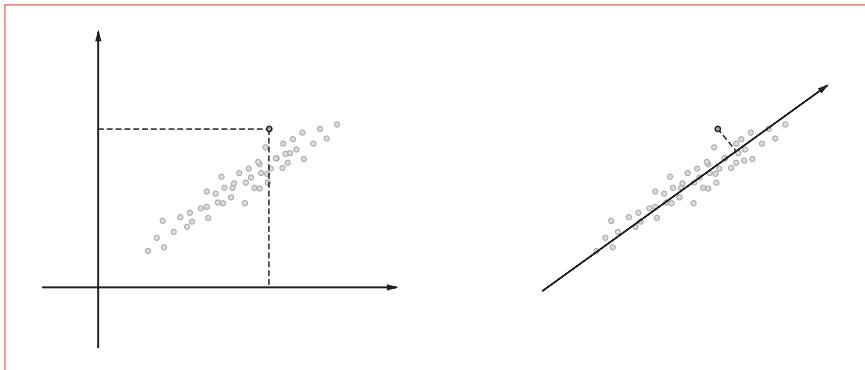


図2-8 次元削減のイメージ：2次元データを、1次元で表現するために、重要な軸——データの広がりの大きな軸——を見つける

図2-8では、元々は2次元の座標で表されていたデータ点を、データの広がりを考慮して、ひとつの座標軸で表すように新たな軸を導入します。このとき各データ点の値は、新しい軸への射影された値によって表されます。ここで大切な点は、データの広がりを考慮した軸をとることで、1次元の値だけでもデータの本質的な差異を捉えられるということです。これと同じようなことは、さらに多次元のデータでも行えます。



ベクトル中のほとんどの要素が0である行列（またはベクトル）を、「疎な行列」（または「疎なベクトル」）と言います。ここでのポイントは、疎なベクトルから重要な軸を見つけて、より少ない次元で表現し直すことです。その結果として、「疎なベクトル」は、ほとんどの要素が0でない「密なベクトル」へと変換されます。この密なベクトルこそが、私たちの求める単語の分散表現です。

次元削減を行う方法はいくつかあります。ここでは**特異値分解**（Singular Value Decomposition : SVD）を使った次元削減を行います。SVDは、任意の行列を3つの行列の積へと分解します。数式で書くと、次のように表されます。

$$\mathbf{X} = \mathbf{U} \mathbf{S} \mathbf{V}^T \quad (2.7)$$

式(2.7)が示すように、SVDは、任意の行列 \mathbf{X} を、 \mathbf{U} 、 \mathbf{S} 、 \mathbf{V} の3つの行列の積に分解します。ここで \mathbf{U} と \mathbf{V} は直交行列であり、その列ベクトルは互いに直交します。また、 \mathbf{S} は対角行列であり、これは対角成分以外はすべて0の行列です。このと

き、これらの行列を視覚的に表すと、図2-9 のようになります。

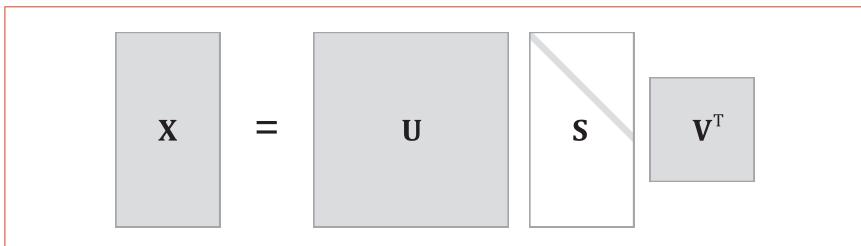


図2-9 SVDによる行列の変換（行列の“白い部分”は0の要素を表す）

さて、式(2.7)において \mathbf{U} は直交行列です。そして、この直交行列は何らかの空間の軸（基底）を形成しています。私たちの文脈においては、この \mathbf{U} という行列を「単語空間」として扱うことができるのです。また、 \mathbf{S} は対角行列で、この対角成分には、「特異値」というものが大きい順に並んでいます。特異値とは、簡単に言えば、「対応する軸」の重要度とみなすことができます。そこで、図2-10 のように、重要な要素を削ることができます。

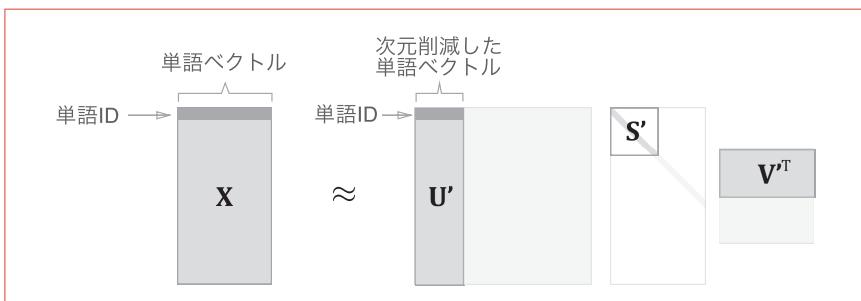


図2-10 SVDによる次元削減のイメージ

図2-10に示すように、行列 \mathbf{S} の特異値が小さいものは重要度が低いので、行列 \mathbf{U} から余分な列ベクトルを削ることで、元の行列を近似することができます。これを私たちが扱っている「単語のPPMI行列」に当てはめると、行列 \mathbf{X} の各行には対応する単語IDの単語ベクトルが格納されており、それらの単語ベクトルが行列 \mathbf{U}' として次元削減されたベクトルで表現されることになります。



単語の共起行列は正方行列ですが、図 2-10 では、前の図と統一するため、長方形のまま図で示しています。また、ここでは SVD について、直感的で概要的な説明にとどめました。数学的に詳しく述べたい方は、文献 [20]などを参考にしてください。

2.4.3 SVDによる次元削減

Python で実際に SVD を行ってみましょう。この SVD は NumPy の `linalg` モジュールにある `svd` というメソッドで実行できます。ちなみに、`linalg` とは linear algebra（線形代数）の略称です。それでは、共起行列を作り、PPMI 行列に変換し、それに対して SVD を適用します（☞ ch02/count_method_small.py）。

```
import sys
sys.path.append('..')
import numpy as np
import matplotlib.pyplot as plt
from common.util import preprocess, create_co_matrix, ppmi

text = 'You say goodbye and I say hello.'
corpus, word_to_id, id_to_word = preprocess(text)
vocab_size = len(id_to_word)
C = create_co_matrix(corpus, vocab_size, window_size=1)
W = ppmi(C)

# SVD
U, S, V = np.linalg.svd(W)
```

これで、SVD を実行できました。上の `U` という変数に、SVD によって変換された密なベクトル表現が格納されています。それでは、実際に中身を見てみましょう。ここでは、単語 ID が 0 の単語ベクトルを見てみます。

```
print(C[0]) # 共起行列
# [0 1 0 0 0 0]

print(W[0]) # PPMI行列
# [ 0.      1.807  0.      0.      0.      0.      0.      ]

print(U[0]) # SVD
# [ 3.409e-01 -1.110e-16 -1.205e-01 -4.441e-16  0.000e+00 -9.323e-01
#   2.226e-16]
```

上の結果が示すとおり、元は疎なベクトルであった `W[0]` が、SVD によって密な

ベクトル $U[0]$ へと変換されています。そして、この密なベクトルを次元削減するには、たとえば、それを 2 次元のベクトルに削減するのであれば、単に先頭の 2 つの要素を取り出します。

```
print(U[0, :2])
# [ 3.409e-01 -1.110e-16]
```

これで、次元削減は終わりです。それでは、各単語を 2 次元のベクトルで表し、それをグラフにプロットしてみましょう。それには、次のように書きます。

```
for word, word_id in word_to_id.items():
    plt.annotate(word, (U[word_id, 0], U[word_id, 1]))
plt.scatter(U[:,0], U[:,1], alpha=0.5)
plt.show()
```

`plt.annotate(word, x, y)` という関数は、2 次元のグラフ上の座標が (x, y) の地点に、`word` というテキストを描画します。それでは、上のコードを実行してみます。その結果は、図2-11 のようになります^{†1}。

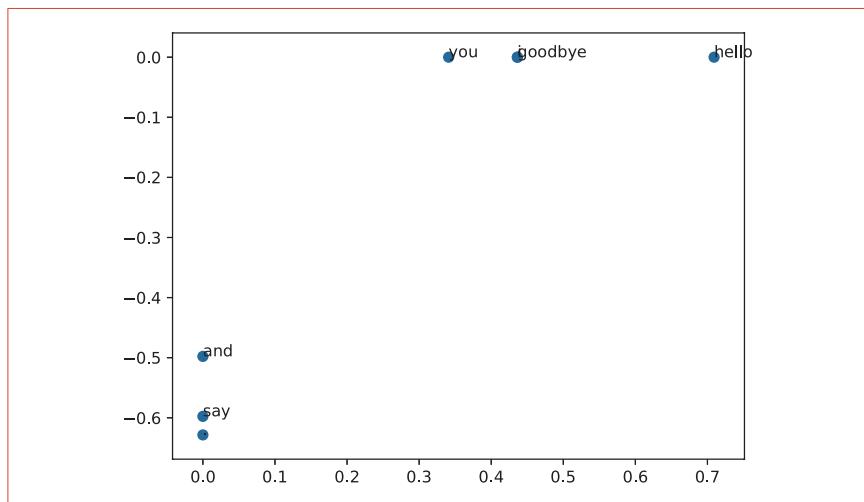


図2-11 共起行列に対して SVD を行い、各単語を 2 次元ベクトルにしてグラフにプロットする（※「i」と「goodbye」が重なっている）

^{†1} 出力されるグラフは、OS の種類や Matplotlib のバージョンに応じて、図2-11 とは見た目が異なる可能性があります。

このプロット図を見ると、「goodbye」と「hello」、「you」と「i」が近い場所に位置することが分かります。これは私たちの直感と比較的近いものでしょう。しかし、ここでは小さなコーパスを使っている関係で、この結果は正直微妙なものです。それでは続いて PTB データセットという、より大きなコーパスを使って同じ実験を行ってみます。まずは PTB データセットについて簡単に説明します。



行列のサイズを N とした場合、SVD の計算は $\mathcal{O}(N^3)$ のオーダーになります。これは N の 3 乗に比例した計算量が必要になるということです。現実には、そのような計算は手に負えなくなるため、Truncated SVD [21]などのより高速な手法が用いられます。Truncated SVD は特異値の小さなものは切り捨てる (= Truncated) ことで高速化を図ります。次節では、オプションとして sklearn ライブラリの Truncated SVD を用います。

2.4.4 PTB データセット

これまで私たちは、とても小さなテキストデータをコーパスとして使用してきました。ここでは、“本格的”なコーパス——それでいて大きすぎない手ごろなコーパス——を利用したいと思います。それは Penn Treebank (ペン・ツリー・バンク) と呼ばれるコーパスです（以降、PTB と略して表記します）。



PTB コーパスは、提案手法の品質を測定するためのベンチマークとしてよく利用されます。本書でも PTB コーパスを利用して、さまざまな実験を行います。

私たちが利用する PTB コーパスは、word2vec の発明者である Tomas Mikolov 氏の Web ページで用意されているものです。この PTB コーパスはテキストファイルで提供されており、元となる PTB の文章に対して、いくつかの前処理が施されています。どのような前処理かというと、たとえば、レアな単語を<unk>という特殊文字で置き換えたり（unk は「unknown」の略）、具体的な数字を「N」で置き換えたりといったことが行われています。そのような前処理を行った後のテキストデータを、私たちは PTB コーパスとして利用します。参考までに、図2-12 に PTB コーパスの中身を示します。

```

1 consumers may want to move their telephones a little closer to the tv set
2 <unk> <unk> watching abc 's monday night football can now vote during <unk> for the greatest play in N years from
   among four or five <unk> <unk>
3 two weeks ago viewers of several nbc <unk> consumer segments started calling a N number for advice on various
   - <unk> issues
4 and the new syndicated reality show hard copy records viewers ' opinions for possible airing on the next day 's show
5 interactive telephone technology has taken a new leap in <unk> and television programmers are racing to exploit the
   - possibilities
6 eventually viewers may grow <unk> with the technology and <unk> the cost

```

図2-12 PTB コーパス（テキストファイル）の例

図2-12 で示すように、PTB コーパスではひとつの文が 1 行ごとに保存されています。本書では、各文を連結したものを「ひとつの大きな時系列データ」として扱うことになります。またこのとき、各文の終わりに<eos>という特殊文字を挿入します（eos は「end of sentence」の略）。



本書では、文の区切りを考慮せずに、複数の文を連結したものを「ひとつの大きな時系列データ」とみなします。もちろん、文単位で処理すること——たとえば、文単位で単語の頻度をカウントすること——も可能です。しかし本書では単純さを優先して、文ごとの処理は行いません。

本書では、Penn Treebank のデータセットを簡単に利用できるように、専用の Python コードを準備しています。このファイルは、dataset/ptb.py にあり、「チャプターのディレクトリ（ch01, ch02, ...）」から使うことを想定しています。たとえば、ch02 ディレクトリに移動してから、そのディレクトリ内において、「python show_ptb.py」のように利用します。それでは、ptb.py を使う例を次に示します（☞ ch02/show_ptb.py）。

```

import sys
sys.path.append('..')
from dataset import ptb

corpus, word_to_id, id_to_word = ptb.load_data('train')

print('corpus size:', len(corpus))
print('corpus[:30]:', corpus[:30])
print()
print('id_to_word[0]:', id_to_word[0])
print('id_to_word[1]:', id_to_word[1])

```

```

print('id_to_word[2]:', id_to_word[2])
print()
print("word_to_id['car']:", word_to_id['car'])
print("word_to_id['happy']:", word_to_id['happy'])
print("word_to_id['lexus']:", word_to_id['lexus'])

```

このコードの説明は後にするとして、これを実行した結果は次のようになります。

```

corpus size: 929589
corpus[:30]: [ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18
19 20 21 22 23
24 25 26 27 28 29]

id_to_word[0]: aer
id_to_word[1]: banknote
id_to_word[2]: berlitz

word_to_id['car']: 3856
word_to_id['happy']: 4428
word_to_id['lexus']: 7426

```

コーパスの扱い方については、これまでと同じです。`corpus` には単語 ID のリストが格納されます。`id_to_word` は単語 ID から単語への変換を行うディクショナリ、`word_to_id` は単語から単語 ID への変換を行うディクショナリを表します。

先のソースコードに示すように、`ptb.load_data()` でデータをロードします。このとき引数には、「train」、「test」、「valid」のいずれかを指定します。これはそれぞれ、「訓練用 / テスト用 / 検証用」のデータのいずれかに該当します。以上が `ptb` の使い方の説明です。

2.4.5 PTB データセットでの評価

PTB データセットに対してカウントベースの手法を適用してみましょう。ここでは、大きな行列に SVD を行うため、より高速な SVD を利用することを推奨します。それには、`sklearn` モジュールをインストールする必要があります。もちろん、シンプルな SVD (`np.linalg.svd()`) も使えますが、多くの時間とメモリが必要になります。それでは、ソースコードをまとめて示します([ch02/count_method_big.py](#))。

```

import sys
sys.path.append('..')
import numpy as np
from common.util import most_similar, create_co_matrix, ppmi
from dataset import ptb

```

```

window_size = 2
wordvec_size = 100

corpus, word_to_id, id_to_word = ptb.load_data('train')
vocab_size = len(word_to_id)
print('counting co-occurrence ...')
C = create_co_matrix(corpus, vocab_size, window_size)
print('calculating PPMI ...')
W = ppmi(C, verbose=True)

print('calculating SVD ...')
try:
    # truncated SVD (fast!)
    from sklearn.utils.extmath import randomized_svd
    U, S, V = randomized_svd(W, n_components=wordvec_size, n_iter=5,
                             random_state=None)
except ImportError:
    # SVD (slow)
    U, S, V = np.linalg.svd(W)

word_vecs = U[:, :wordvec_size]

querys = ['you', 'year', 'car', 'toyota']
for query in querys:
    most_similar(query, word_to_id, id_to_word, word_vecs, top=5)

```

ここでは SVD を行うために、sklearn の `randomized_svd()` というメソッドを利用します。これは、乱数を使った Truncated SVD で、特異値の大きいものだけに限定して計算することで、通常の SVD よりも高速な計算が行えます。残りのコードは、前の小さなコーパスのときのコードとほとんど同じです。それでは、上のコードを実行してみましょう。そうすると、次の結果が得られます（Truncated SVD の場合は、乱数を使う関係で下記の結果は毎回異なります）。

```
[query] you
i: 0.702039909619
we: 0.699448543998
've: 0.554828709147
do: 0.534370693098
else: 0.512044146526
```

```
[query] year
month: 0.731561990308
quarter: 0.658233992457
last: 0.622425716735
earlier: 0.607752074689
```

```
next: 0.601592506413
```

```
[query] car
luxury: 0.620933665528
auto: 0.615559874277
cars: 0.569818364381
vehicle: 0.498166879744
corsica: 0.472616831915
```

```
[query] toyota
motor: 0.738666107068
nissan: 0.677577542584
motors: 0.647163210589
honda: 0.628862370943
lexus: 0.604740429865
```

結果を見ていくと、まずは「you」というクエリに対して、「i」や「we」の人称代名詞が上位を占めていることが分かります。これは文法的な使い方の点で共通の単語です。また、「year」がクエリの場合は「month」や「quarter」、「car」がクエリの場合は「auto」や「vehicle」など類義語が得られています。さらに、「toyota」をクエリとした場合は、「nissan」や「honda」、「lexus」など自動車のメーカーやブランドが来ることも確認できます。このように、単語の意味的な点と文法的な点において、似た単語どうしが近いベクトルで表されました。これは私たちの感覚に近いものと言えそうです。

おめでとうございます！ 私たちはついに「単語の意味」をベクトルにうまくエンコードすることに成功しました！ コーパスを使い、コンテキストの単語をカウントし、そしてそれを PPMI 行列に変換して、SVD による次元削減を行うことでより良い単語ベクトルを得られたのです。これこそが単語の分散表現であり、各単語は固定長のベクトルとして、そして密なベクトルとして表現されました。

本章の実験では、一部の単語だけに対して類似単語を見たにすぎません。しかし、他の多くの単語でもそのような性質を確認できるでしょう。さらに大規模なコーパスを使用することで、より優れた単語の分散表現になることが期待されます。

2.5 まとめ

本章では、自然言語を対象として、特に「単語の意味」をコンピュータに理解させることをテーマに話を進めてきました。そのような目的を達成するために、シソーラスを用いた手法を説明し、続いてカウントベースの手法を見てきました。

シソーラスを用いる手法では、人の手によってひとつずつ単語の関連性を定義します。しかし、そのような作業はとても大変であり、また表現力の点で限界があります（細かなニュアンスを表せない、など）。一方、カウントベースの手法は、コーパスから自動的に単語の意味を抽出し、それをベクトルで表します。具体的には、単語の共起行列を作り、PPMI 行列に変換し、ロバスト性を高めるために SVD による次元削減を行い、各単語の分散表現を得ました。そして、その分散表現は、意味的に（また文法的な使い方の点においても）似た単語がベクトル空間上で互いに近い場所にいることが確認できました。

また本章では、コーパスのテキストデータを扱いやすくするための下準備の関数をいくつか実装しました。具体的には、ベクトル間の類似度を計測するための関数 (`cos_similarity()`) や類似単語のランキング表示する関数 (`most_similar()`) を実装しました。これらの関数は、次章以降でも使用します。

本章で学んだこと

- WordNetなどのシソーラスを利用して、類義語の取得や単語間の類似度の計測など有用なタスクを行うことができる
- シソーラスを用いる手法には、シソーラスを作成する人の作業量や新しい単語への対応などの問題がある
- 現在では、コーパスを利用して単語をベクトル化するアプローチが主流である
- 近年の単語ベクトル化の手法では、「単語の意味は、周囲の単語によって形成される」という分布仮説に基づくものがほとんどである
- カウントベースの手法は、コーパス中の各単語に対して、その単語の周囲の単語の頻度をカウントし集計する（=共起行列）
- 共起行列を PPMI 行列に変換し、それを次元削減することで、巨大な「疎なベクトル」を小さな「密なベクトル」へと変換することができる
- 単語のベクトル空間では、意味的に近い単語はその距離が近くなることが期待される

3章 word2vec

判断材料がないのに、推論するのは禁物だ。

—— コナン・ドイル 『シャーロック・ホームズの冒険（ボヘミアの醜聞）』

前章から引き続き、本章のテーマも単語の分散表現です。前章では、「カウントベースの手法」によって単語の分散表現を得ました。本章では、「カウントベースの手法」に代わる強力な手法として「推論ベースの手法」を見ていきます。

「推論ベースの手法」は、その名前が示すとおり、推論をする手法です。もちろん、その推論にはニューラルネットワークが使えます。そして、ここで有名な word2vec が登場します。本章では、word2vec の仕組みをじっくりと時間をかけて見ていき、それを実装することで理解を確かなものとします。

本章の目標は、“シンプル”な word2vec を実装することです。このシンプルな word2vec では、処理効率は犠牲にして、分かりやすさを優先しています。そのため、大きなデータセットは扱えませんが、小さなデータセットであれば問題なく処理できます。次章では、本章のシンプルな word2vec にいくつかの改良を加え、“本物”的な word2vec を完成させます。それでは、推論ベースの手法へ、word2vec の世界へ進みましょう！

3.1 推論ベースの手法とニューラルネットワーク

単語をベクトルで表す研究は、これまで盛んに行われてきました。その中でも成功を収めた手法を見ていくと、それらは大きく 2 つに分けられます。ひとつは「カウントベースの手法」、もうひとつは「推論ベースの手法」です。単語の意味を獲得する

ためのアプローチは両者で大きく異なりますが、その背景には両者ともに分布仮説があります。

ここでは、カウントベースの手法の問題点を指摘し、それに代わる推論ベースの手法の利点を大きな視点で説明します。そして、word2vecへの下準備を行うために、ニューラルネットワークで「単語」を処理する例を見ていきます。

3.1.1 カウントベースの手法の問題点

これまで見てきたように、カウントベースの手法では、周囲の単語の頻度によって単語を表現しました。具体的には、単語の共起行列を作り、その行列に対して SVD を適用することで、密なベクトル——単語の分散表現——を獲得したのです。しかし、カウントベースの手法には問題があります。その問題は、大規模なコーパスを扱う場合に発生します。

現実的には、コーパスで扱う語彙数は非常に巨大になります。たとえば、英語の語彙数は 100 万をゆうに超えると言われています。語彙数を 100 万とした場合、カウントベースの手法では、 $100\text{ 万} \times 100\text{ 万}$ の巨大な行列を作ることになります。しかし、そのような巨大な行列に対して SVD を行うこととは、現実的ではありません。



SVD は、 $n \times n$ の行列に対して、 $O(n^3)$ の計算コストがかかります。 $O(n^3)$ とは、 n の大きさの 3 乗に比例して計算時間が増えていくということです。そのような計算コストでは、スーパーコンピュータをもってしても太刀打ちできません。実際には、近似的な手法や疎な行列の性質などの利用によって、処理速度を向上させることができます。しかし、その場合も多くの計算リソースと計算時間が必要になります。

カウントベースの手法は、コーパス全体の統計データ（共起行列や PPMI など）を利用して、1 回の処理（SVD など）で単語の分散表現を獲得します。一方、推論ベースの手法では、たとえばニューラルネットワークを用いる場合は、ミニバッチで学習するのが一般的です。ミニバッチで学習するとは、ニューラルネットワークは一度に少量（ミニバッチ）の学習サンプルを見ながら、重みを繰り返し更新するということです。この学習における枠組みの違いは、図で表すと図 3-1 のようになります。

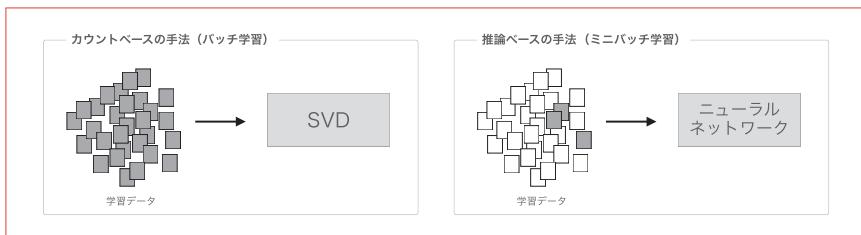


図3-1 カウントベースの手法と推論ベースの手法の比較

図3-1に示すように、カウントベースの手法は、学習データを一度にまとめて処理します。それに対して、推論ベースの手法では、学習データの一部を使って逐次的に学習します。これが意味することは、語彙数が大きいコーパスにおいて SVD などの計算量が膨大で処理が難しい場合でも、ニューラルネットワークではデータを小分けにして学習できるということです。さらに、ニューラルネットワークの学習は複数マシン/複数 GPU の利用による並列計算も可能であり、全体の学習も高速化できます。この点において、推論ベースの手法に分があります。

この他にも、推論ベースの手法がカウントベースの手法よりも魅力的な点があります。その点については、推論ベースの手法について（特に word2vec について）詳しく説明した後に、「3.5.3 カウントベース v.s. 推論ベース」で、もう一度議論したいと思います。

3.1.2 推論ベースの手法の概要

推論ベースの手法では、「推論」することが主な作業になります。これは、図3-2で示すように、周囲の単語（コンテキスト）が与えられたときに、「？」にどのような単語が出現するのかを推測する作業です。

you ? goodbye and I say hello.

図3-2 両隣の単語をコンテキストとして、「？」にどのような単語が出現するのかを推測する

図3-2のような推論問題を解くこと、そして、学習することが「推論ベースの手法」の扱う問題です。このような推論問題を繰り返し解くことで、単語の出現パターンを

学習します。このとき“モデル視点”に立つと、この推論問題は図3-3 のように見えます。

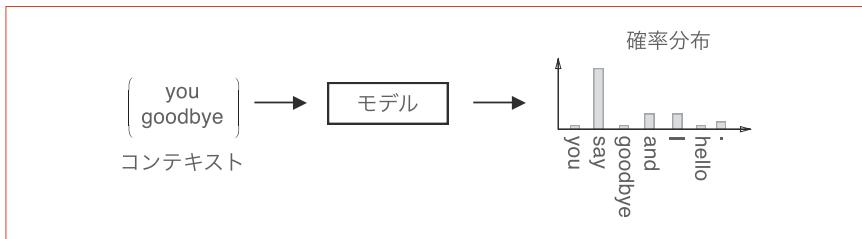


図3-3 推論ベースの手法：コンテキストを入力として与えれば、モデルは各単語の出現確率を出力する

図3-3に示すように、推論ベースの手法では、何らかのモデルが登場します。私たちは、そのモデルにニューラルネットワークを使います。モデルはコンテキスト情報を入力として受け取り、(出現しうるであろう) 各単語の出現する確率を出力します。そのような枠組みの中で、正しい推測ができるように、コーパスを使ってモデルの学習を行います。そして、その学習の結果として、単語の分散表現を得られるというものが推論ベースの手法の全体図になります。



推論ベースの手法も、カウントベースの手法と同じく分布仮説に基づきます。分布仮説とは「単語の意味は、周囲の単語によって形成される」というものでしたが、推論ベースの手法では、これを上のような推測問題へと帰着させたのです。このように、どちらの手法も、分布仮説に基づく「単語の共起性」をいかにモデル化するかという点が重要な研究テーマとなります。

3.1.3 ニューラルネットワークにおける単語の処理方法

私たちはこれからニューラルネットワークを使って「単語」を処理します。しかし、ニューラルネットワークは“you”や“say”などの単語をそのままでは処理できません。ニューラルネットワークで単語を処理するには、それを「固定長のベクトル」に変換する必要があります。そのための方法のひとつは、単語を **one-hot 表現**（または**one-hot ベクトル**）へと変換することです。one-hot 表現とは、ベクトルの要素の中でひとつだけが 1 で、残りはすべて 0 であるようなベクトルを言います。

one-hot 表現について具体的に見てきましょう。ここでは前章と同じく、「You

say goodbye and I say hello.」という 1 文をコーパスとして扱うものとして話を進めます。このコーパスでは、語彙が全部で 7 個存在します (“you”, “say”, “goodbye”, “and”, “I”, “hello”, “?”)。このとき各単語は図3-4 のように one-hot 表現へ変換することができます。

| 単語 | 単語ID | one-hot表現 |
|--|--|--|
| $\begin{pmatrix} \text{you} \\ \text{goodbye} \end{pmatrix}$ | $\begin{pmatrix} 0 \\ 2 \end{pmatrix}$ | $\begin{pmatrix} (1, 0, 0, 0, 0, 0, 0, 0) \\ (0, 0, 1, 0, 0, 0, 0, 0) \end{pmatrix}$ |

図3-4 単語と単語 ID、そして one-hot 表現

図3-4 に示すように単語は、テキスト、単語 ID、そして one-hot 表現でそれぞれ表現できます。このとき単語を one-hot 表現に変換するには、語彙数分の要素を持つベクトルを用意して、単語 ID の該当する箇所を 1 に、残りはすべて 0 に設定します。このように単語を固定長のベクトルに変換してしまえば、私たちのニューラルネットワークの入力層は、図3-5 のようにニューロンの数を“固定”することができます。

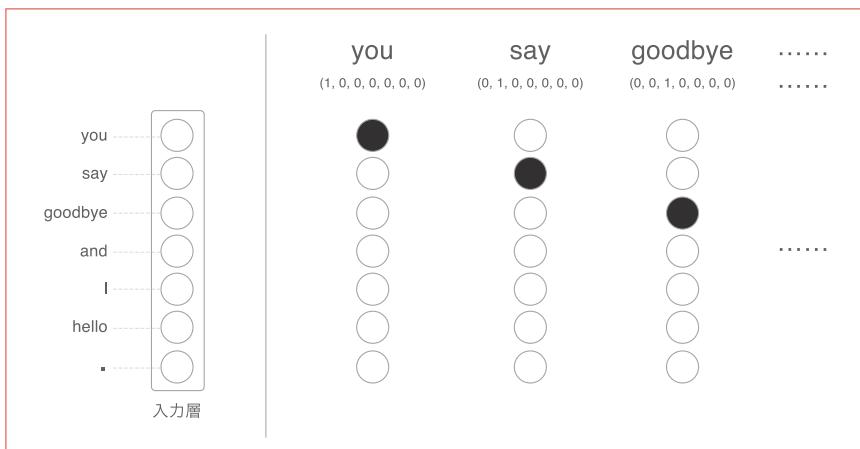


図3-5 入力層のニューロン：各ニューロンが各単語に対応する。図では、ニューロンが 1 の場所は黒で描画し、0 の場所は白で描画する

図3-5 に示すように、入力層は 7 つのニューロンによって表されます。このとき、7

つのニューロンはそれぞれ 7 つの単語に対応します（ひとつ目のニューロンは「you」に、2 つ目のニューロンは「say」といったように）。

これで話は単純になりました。なぜなら単語をベクトルで表すことができれば、そのベクトルはニューラルネットワークを構成するさまざまな「レイヤ」によって処理することができるからです。たとえば、one-hot 表現で表されたひとつの単語に対して、全結合層で変換する場合は図3-6 のように書くことができます。

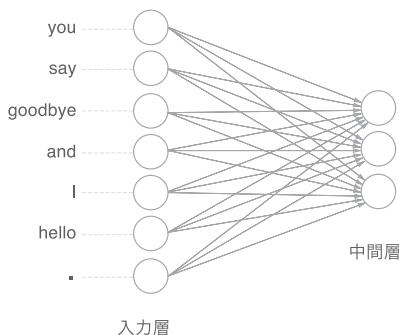


図3-6 ニューラルネットワークの全結合層による変換：入力層の各ニューロンは 7 つの単語に対応する（ひとまずここでは、中間層に 3 つのニューロンを用意する）

全結合層ということで、図3-6 に示すように、すべてのノードには矢印によるつながりがあります。この矢印には重み（パラメータ）が存在し、入力層のニューロンとの重み付き和が中間層のニューロンとなります。なお、本章で使用する全結合層ではバイアスは省略することにします（これは次の word2vec の説明を見越してのことです）。



バイアスを用いない全結合層は、「行列の積」の計算に相当します。多くのディープラーニングのフレームワークでは、全結合層の生成時にバイアスを用いないような選択ができます。本書の場合、バイアスを用いない全結合層は MatMul レイヤに相当します（MatMul レイヤは 1 章で実装しました）。

さて、図3-6 ではニューロン間の結びつきを矢印によって図示しましたが、これ以降は、重みを明確に示すため図3-7 のような図法を用いることにします。

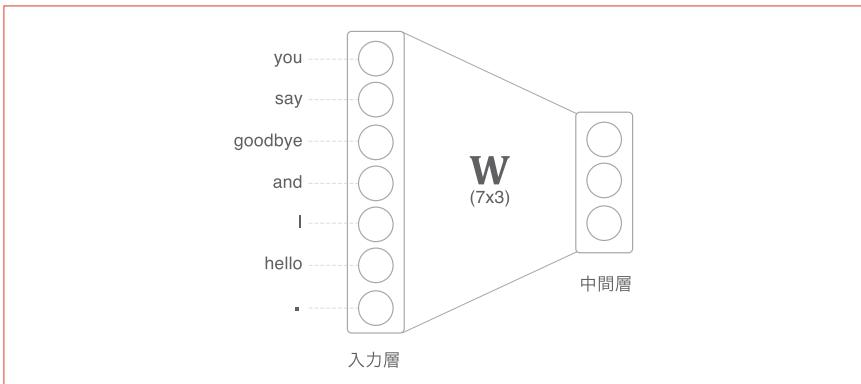


図 3-7 全結合層による変換を簡略化して図示する：ここでは全結合層の重みを、 7×3 の形状の \mathbf{W} という行列で表す

それでは、ここまで話題をコードベースで見ていきましょう。早速ですが、ここで全結合層による変換は、Python で次のように書くことができます。

```
import numpy as np

c = np.array([[1, 0, 0, 0, 0, 0, 0]]) # 入力
W = np.random.randn(7, 3) # 重み
h = np.dot(c, W) # 中間ノード
print(h)
# [[-0.70012195  0.25204755 -0.79774592]]
```

このコード例では、単語 ID が 0 の単語を one-hot 表現で表し、それを全結合層によって変換する例を示しています。復習になりますが、全結合層の計算は行列の積によって行うことができました。そしてそれは NumPy の `np.dot()` で実装できます（バイアスは省略）。



ここで入力データ（コードでは `c`）の次元数 (`ndim`) は 2 です。これはミニバッチ処理を考慮したもので、最初の次元 (0 次元目) に各データを格納します。

さて、上のコードで注目してほしいのは、`c` と `W` の行列の積の箇所です。ここで、`c` は one-hot 表現であるため、単語 ID に対応する要素が 1 で、それ以外は 0 であるベクトルになります。そのため、上のコードの `c` と `W` の行列の積で行っていること

は、図3-8に示すように、重みの行ベクトルを“抜き出す”ことに相当します。

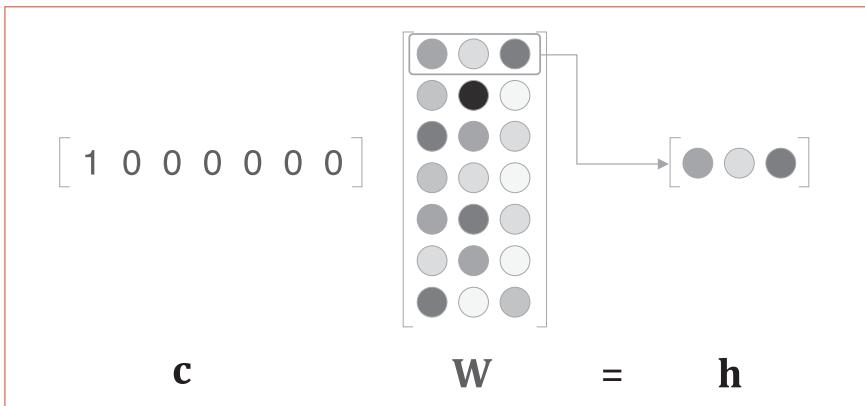


図3-8 コンテキスト c と重み W の乗算では、該当する場所の行ベクトルが抜き出される（重みの各要素の大きさを白黒の濃淡で示す）

ここで、重みから行ベクトルを抜き出すためだけに行列の積を計算するのは非効率に感じられるかもしれません。その点については、「4.1 word2vec の改良①」で改良を行う予定です。なお、上のコードで行ったことは、(1章で実装済みの) MatMul レイヤによっても行うことができます。これは次のようなコードになります。

```
import sys
sys.path.append('..')
import numpy as np
from common.layers import MatMul

c = np.array([[1, 0, 0, 0, 0, 0, 0]])
W = np.random.randn(7, 3)
layer = MatMul(W)
h = layer.forward(c)
print(h)
# [[-0.70012195  0.25204755 -0.79774592]]
```

ここでは、common ディレクトリにある MatMul レイヤをインポートして利用します。後は、MatMul レイヤに重み W を設定し、`forward()` メソッドによって順伝播の処理を行います。

3.2 シンプルな word2vec

前節では推論ベースの手法を学び、ニューラルネットワークにおける単語の処理方法についてコードベースで見てきました。これで準備が整いました。それではいよいよ、word2vec の実装に取りかかります。

これから行なうことは、図3-3で示される「モデル」にニューラルネットワークを組み込むことです。ここでは、そのニューラルネットワークに、word2vecで提案されている **continuous bag-of-words**（以降、**CBOW**と略記）と呼ばれるモデルを使います。



word2vec という用語は、本来はプログラムやツール類を指して用いられます。ただし word2vec という言葉はポピュラーになり、文脈によっては、その言葉がニューラルネットワークのモデルを指す場合も多く見受けられます。正しくは、CBOW モデルと skip-gram モデルという 2 つのモデルが word2vec で使用されるニューラルネットワークです。ここでは CBOW モデルを対象に話を進めていきます。この 2 つのモデルの違いなどについては「3.5.2 skip-gram モデル」で詳しく説明します。

3.2.1 CBOW モデルの推論処理

CBOW モデルは、コンテキストからターゲットを推測することを目的としたニューラルネットワークです（「ターゲット」は中央の単語、その周囲の単語が「コンテキスト」）。この CBOW モデルができるだけ正確な推測ができるように訓練することで、私たちは単語の分散表現を獲得することができます。

CBOW モデルへの入力はコンテキストです。このコンテキストは、['you', 'goodbye'] のような単語のリストで表されます。私たちはそれを one-hot 表現に変換することで、CBOW モデルが処理できるように調整します。それを踏まえると、CBOW モデルのネットワークは図3-9のように書けます。

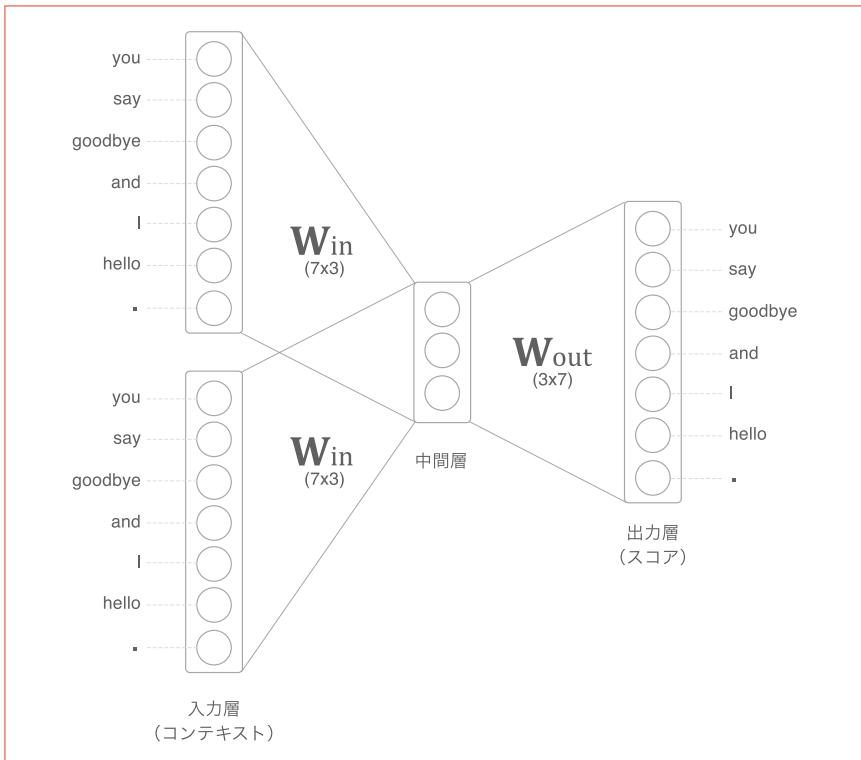


図3-9 CBOW モデルのネットワーク構造

図3-9 が、CBOW モデルのネットワークです。入力層が 2 つあり、中間層を経て、出力層へとたどり着きます。ここで、入力層から中間層への変換は、同じ全結合層（重みは W_{in} ）によって行われます。そして、中間層から出力層のニューロンへの変換は、別の全結合層（重みは W_{out} ）によって行われます。



ここではコンテキストとして 2 つの単語を考えているため、入力層が 2 つ存在します。もしコンテキストとして N 個の単語を扱うのであれば、入力層は N 個存在することになります。

それでは図3-9 の中間層に注目しましょう。このとき、中間層にあるニューロンは、各入力層の全結合による変換後の値が「平均」されたものになります。上の例で言えば、全結合によってひとつ目の入力層が h_1 に、2 つ目の入力層が h_2 に変換さ

れたとすると、中間層のニューロンは $\frac{1}{2}(\mathbf{h}_1 + \mathbf{h}_2)$ になります。

最後に図3-9の出力層についてです。この出力層には7個のニューロンがありますが、ここで重要なのが、それらのニューロンは各単語に対応しているということです。そして出力層のニューロンは各単語の「スコア」であり、その値が高ければ高いほど、それに対応する単語の出現確率も高くなります。なお、スコアとは確率として解釈される前の値であり、このスコアにSoftmax関数を適用することで「確率」が得られます。



スコアをSoftmaxレイヤに通した後のニューロンを「出力層」と呼ぶ場合もあります。ここではスコアを出力するノードを「出力層」と呼ぶことにします。

図3-9のとおり、入力層から中間層への変換は、全結合層（重みは \mathbf{W}_{in} ）によって行われます。このとき、全結合層の重み \mathbf{W}_{in} は 7×3 の形状の行列ですが、先にタネを明かすと、この重みこそが単語の分散表現の正体になります。これは図で表すと、図3-10のようになります。

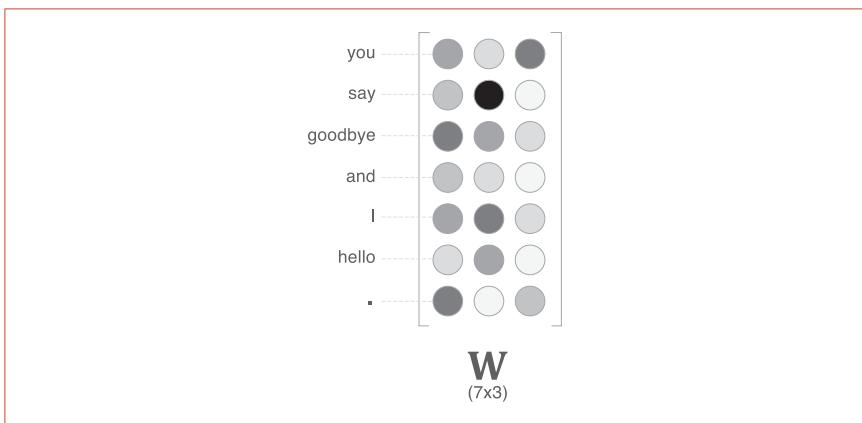


図3-10 重みの各行が、各単語の分散表現に対応する

図3-10に示すように、重み \mathbf{W}_{in} の各行にはそれぞれの単語の分散表現が格納されていると考えます。そして学習を重ねることで、コンテキストから出現する単語をうまく推測できるように各単語の分散表現が更新されていくのです。そして驚くべき

ことには、そのようにして得られたベクトルには「単語の意味」もうまくエンコードされているのです！これがword2vecの全体像になります。



中間層のニューロンの数を入力層のそれよりも減らすことが重要なポイントです。それによって中間層には、単語を予測するために必要な情報を“コンパクト”に収める必要があり、結果として密なベクトル表現が得られます。このとき、その中間層の情報は、私たち人間には理解できない“コード”で書かれています。これは、「エンコード」という作業に相当します。一方、中間層の情報から目的的結果を得る作業は、「デコード」と言います。これはエンコードされた情報を、私たち人間が理解できる表現へと復元する作業です。

ところで、私たちはこれまでのところ、CBOWモデルを「ニューロン視点」で図示してきました。ここでは「レイヤ視点」でCBOWモデルを図示してみたいと思います。そうすると、そのネットワーク構成は図3-11のようになります。

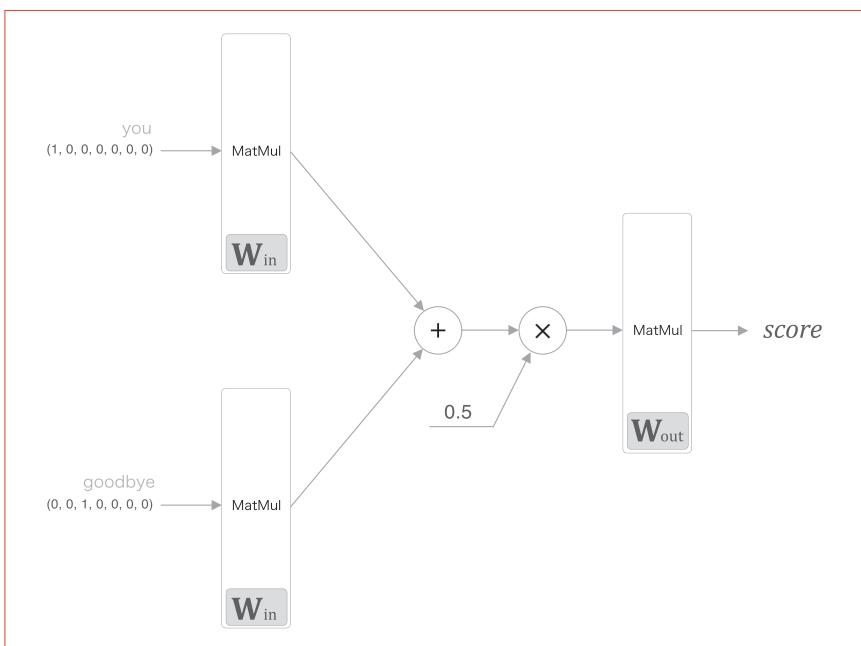


図3-11 レイヤ視点のCBOWモデルのネットワーク構成：MatMulレイヤ内で使用する重み（ \mathbf{W}_{in} , \mathbf{W}_{out} ）は、それぞれのレイヤ内に描画する

図3-11が示すとおり、CBOWモデルは最初に2つのMatMulレイヤがあり、その2つの出力が互いに加算されます。そして、その加算された値に0.5を乗算することで「平均」が求められ、それが中間層のニューロンとなります。最後に、その中間層のニューロンに対して、別のMatMulレイヤが適用され「スコア」が出力されます。



バイアスを用いない全結合層の処理はMatMulレイヤの順伝播によって行います。このレイヤは、内部で行列の積を計算します。

それでは図3-11を参考に、CBOWモデルの推論処理をPythonで実装してみましょう（推論処理とは「スコア」を求める処理を指します）。これは次のように実装できます（☞ ch03/cbow_predict.py）。

```
import sys
sys.path.append('..')
import numpy as np
from common.layers import MatMul

# サンプルのコンテキストデータ
c0 = np.array([[1, 0, 0, 0, 0, 0, 0]])
c1 = np.array([[0, 0, 1, 0, 0, 0, 0]])

# 重みの初期化
W_in = np.random.randn(7, 3)
W_out = np.random.randn(3, 7)

# レイヤの生成
in_layer0 = MatMul(W_in)
in_layer1 = MatMul(W_in)
out_layer = MatMul(W_out)

# 順伝播
h0 = in_layer0.forward(c0)
h1 = in_layer1.forward(c1)
h = 0.5 * (h0 + h1)
s = out_layer.forward(h)

print(s)
# [[ 0.30916255  0.45060817 -0.77308656  0.22054131  0.15037278
#   -0.93659277 -0.59612048]]
```

ここでは最初に、必要な重み (`W_in` と `W_out`) を初期化します。そして、入力層を処理する `MatMul` レイヤをコンテキストの数だけ（ここでは 2 つ）生成し、出力層側の `MatMul` レイヤはひとつだけ生成します。このとき、入力層側の `MatMul` レイヤは重み `W_in` を共有する点に注意が必要です。

後は、入力層側の `MatMul` レイヤ (`in_layer0` と `in_layer1`) の `forward()` メソッドを呼び、中間データを計算し、出力層側の `MatMul` レイヤ (`out_layer`) によって各単語のスコアを求めます。

以上が CBOW モデルの推論処理です。ここで見てきたように CBOW モデルは、活性化関数を使わないシンプルなネットワーク構成です。入力層が複数あり、そこでの重みを共有することを除けば、残りは難しい点はないでしょう。それでは続いて、CBOW モデルの学習について見ていきます。

3.2.2 CBOW モデルの学習

ここまで説明した CBOW モデルは、出力層において各単語のスコアを出力しました。このスコアに対して `Softmax` 関数を適用することで、「確率」を得ることができます（図 3-12）。この確率は、コンテキスト（前後の単語）が与えられたときに、その中央にどの単語が出現するのかを表します。

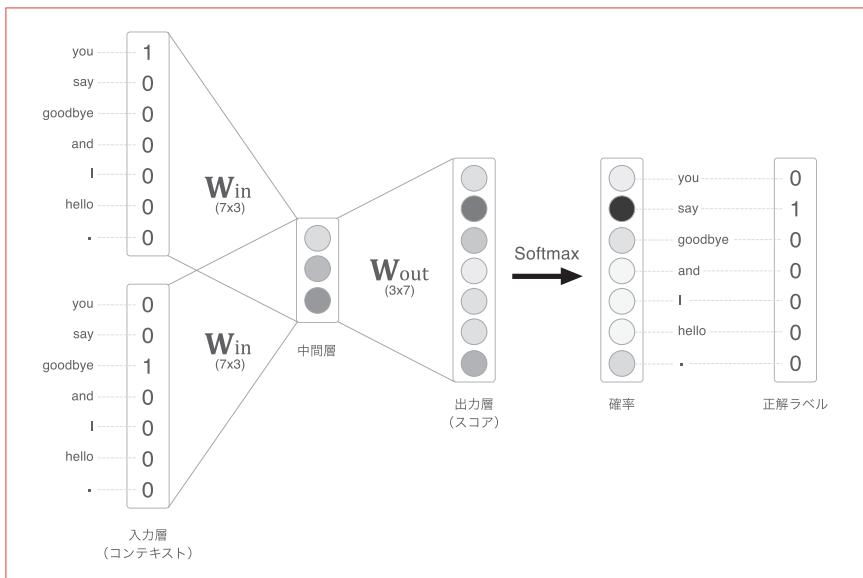


図3-12 CBOW モデルの具体例（ノードの値を白黒の濃淡で示す）

図3-12 で示す例では、コンテキストは「you」と「goodbye」、正解ラベルは——ニューラルネットワークが予測すべき単語は——「say」である例です。このとき“良い重み”的なネットワークがあれば、「確率」を表すニューロンにおいて、正解に対応するニューロンが高くなっていることが期待できます。

CBOW モデルの学習で行なうことは、正しい予測ができるように重みを調整することです。その結果として、重みの W_{in} に——正確には W_{in} と W_{out} の両方に——、単語の出現パターンを捉えたベクトルが学習されます。そしてこれまでの実験によって、CBOW モデル（と skip-gram モデル）で得られる単語の分散表現は——特に、Wikipediaなどの大規模コーパスを使って得られる単語の分散表現は——、単語の意味的な点や文法的な点において、私たちの直感と合致するケースが多く見られるのです。



CBOW モデルは、コーパスにおける単語の出現パターンを学ぶだけです。そのため、コーパスが違えば、学習で得られる単語の分散表現も異なります。たとえば、コーパスとして「スポーツ」の記事だけを使う場合と「音楽」の記事だけを使う場合とでは、得られる単語の分散表現は大きく異なるでしょう。

それでは、上のニューラルネットワークの学習について考えてみましょう。といつても、これからのは簡単です。ここで私たちが扱うモデルは多クラス分類を行うニューラルネットワークです。そのため、それを学習するには、Softmax と交差エントロピー誤差を用いるだけです。ここでは、スコアを Softmax で確率に変換し、その確率と教師ラベルから交差エントロピー誤差を求め、それを損失として学習を行います。これは図で表すと、図3-13 のようになります。

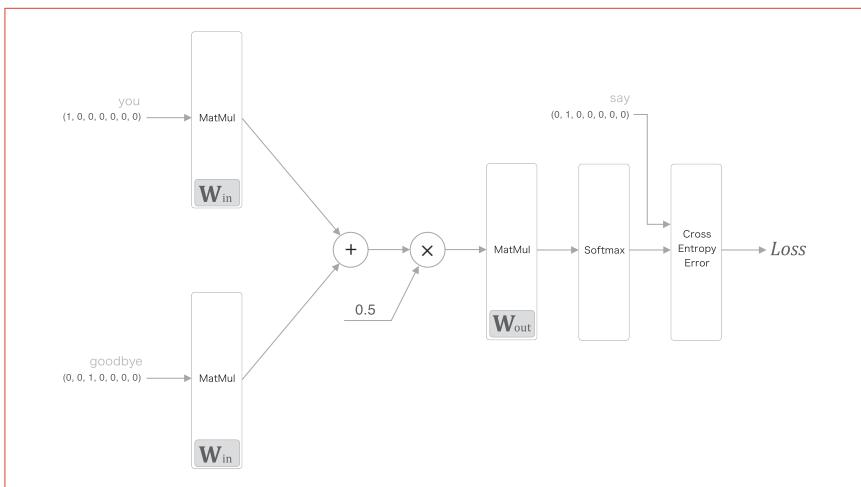


図3-13 学習時における CBOW モデルのネットワーク構成

図3-13 のとおり、ここでは前節で示した推論処理を行う CBOW モデルに対して、Softmax レイヤと Cross Entropy Error レイヤを追加するだけです。これで損失を得ることができます。以上が、CBOW モデルの損失を求める計算の流れです。これがニューラルネットワークの順方向の伝播になります。

なお、図3-13 では Softmax レイヤと Cross Entropy Error レイヤを用いましたが、私たちはその 2 つのレイヤを Softmax with Loss レイヤというひとつのレイヤで実装しています。そのため、私たちがこれから実装するネットワークは、正確には図3-14 のように書けます。

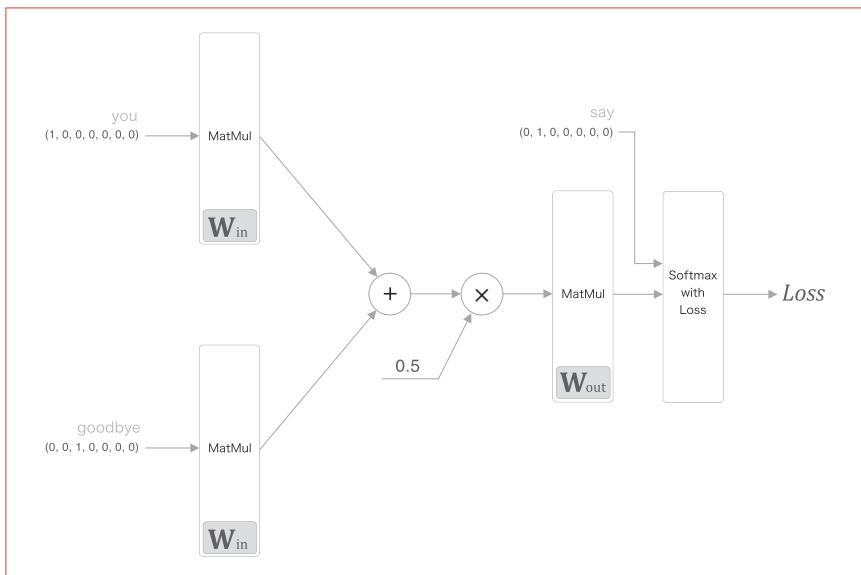


図3-14 Softmax レイヤと Cross Entropy Error レイヤを Softmax with Loss レイヤにまとめて図示する

3.2.3 word2vecの重みと分散表現

これまで説明してきたように、word2vecで使用されるネットワークには2つの重みがあります。それは、入力側の全結合層の重み（ \mathbf{W}_{in} ）と、出力側の全結合層の重み（ \mathbf{W}_{out} ）です。そして、入力側の重み \mathbf{W}_{in} の各行が、各単語の分散表現に対応します。さらに、出力側の重み \mathbf{W}_{out} についても、単語の意味がエンコードされたベクトルが格納されていると考えられます。ただし、出力側の重みは、図3-15に示すように列方向（縦方向の並び）に各単語の分散表現が格納されています。

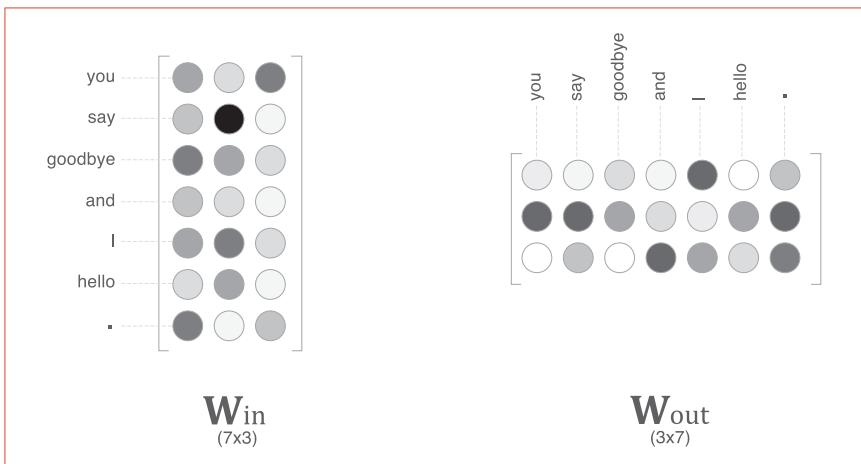


図3-15 各単語の分散表現は、入力側と出力側の両方に見られる

それでは、最終的に利用する単語の分散表現は、どちらの重みを使えばよいでしょうか？選択肢は3つあります。

- A. 入力側の重みだけを利用する
- B. 出力側の重みだけを利用する
- C. 2つの重みの両方を利用する

AとBの案は、どちらか一方の重みだけを利用するというものです。最後の案Cについては、どのように2つの重みを組み合わせるかということで、いくつか方策が考えられます。ひとつ的方法としては、2つの重みを単に足し合わせることが考えられます。

word2vec（特にskip-gramモデル）に関して言えば、Aの「入力側の重みだけを利用する」というのが最もポピュラーな選択肢です。多くの研究では、出力側の重みは利用せずに入力側の重み \mathbf{W}_{in} だけを最終的な単語の分散表現として利用します。私たちもそれにならい、 \mathbf{W}_{in} を単語の分散表現として利用します。



文献[38]では、word2vecのskip-gramモデルにおける \mathbf{W}_{in} の有効性が、実験によって示されています。また、GloVe[27]と呼ばれるword2vecと近しい別の手法では、2つの重みを足し合わせることで良い結果が得られたことが報告されています。

3.3 学習データの準備

これから word2vec の学習を行うにあたって、まずは学習データの準備を行います。ここでは簡単な例として、これまでと同じく「You say goodbye and I say hello.」という 1 文をコーパスとして利用します。

3.3.1 コンテキストとターゲット

word2vec で用いるニューラルネットワークの入力は、「コンテキスト」です。そしてその正解ラベルは、コンテキストに囲まれた中央の単語——ここではこれを「ターゲット」と呼びます——になります。つまり、私たちが行うべきことは、ニューラルネットワークに「コンテキスト」を入力したときに、「ターゲット」が出現する確率を高くすることです（そうなるように学習を行うのです）。

それでは、図 3-16 のように、コーパスから「コンテキスト」と「ターゲット」を作ることを考えましょう。

| corpus | contexts | target |
|--|--------------|---------|
| you <u>say</u> goodbye and I say hello . | you, goodbye | say |
| you say <u>goodbye</u> and I say hello . | say, and | goodbye |
| you say goodbye <u>and</u> I say hello . | goodbye, I | and |
| you say goodbye and <u>I</u> say hello . | and, say | I |
| you say goodbye and I <u>say</u> hello . | I, hello | say |
| you say goodbye and I say <u>hello</u> . | say, . | hello |

図 3-16 コーパスからコンテキストとターゲットを作成する例

図 3-16 では、コーパス中から対象とする単語を「ターゲット」として、その周囲の単語を「コンテキスト」として抜き出します。その作業を、コーパス中のすべての単語（ただし、両端の単語は除く）に対して行います。このようにして作られたのが、図 3-16 の右側にある contexts (コンテキスト) と target (ターゲット) です。この contexts の各行がニューラルネットワークの入力になり、target の各行が正解ラベル（予測すべき単語）となります。なお、各サンプルデータにおいて、コンテキストは複数あり（この例では 2 個）、ターゲットはひとつだけです。そのため、コンテキストだけ contexts と複数形にしています。

これから、コーパスからコンテキストとターゲットを作成する関数を実装します。その前にここでは、前章の復習から始めます。まずは、コーパスのテキストを単語IDに変換するところからです。これには、2章で実装した `preprocess()` 関数を使います。

```
import sys
sys.path.append('..')
from common.util import preprocess

text = 'You say goodbye and I say hello.'
corpus, word_to_id, id_to_word = preprocess(text)
print(corpus)
# [0 1 2 3 4 1 5 6]

print(id_to_word)
# {0: 'you', 1: 'say', 2: 'goodbye', 3: 'and', 4: 'i', 5: 'hello', 6:
→  '}'}
```

それでは、この単語IDの配列である `corpus` から、`contexts` と `target` を作ります。具体的には、図3-17に示すように、`corpus` を与えると、`contexts` と `target` を返す関数を実装します。

| corpus | contexts | target |
|-------------------|----------|------------|
| [0 1 2 3 4 1 5 6] | [[0 2] | [1 |
| | [1 3] | 2 |
| | [2 4] | 3 |
| | [3 1] | 4 |
| | [4 5] | 1 |
| | [1 6]] | 5] |
| → 形状： (8,) | | 形状： (6, 2) |
| | | 形状： (6,) |

図3-17 単語IDの配列である `corpus` から、`contexts` と `target` を作成する例（ウィンドウサイズが1のコンテキストの場合）

図3-17に示すように、`contexts` は2次元の配列です。このとき、`contexts` の0次元目には各コンテキストデータが格納されます。具体的に言うと、`contexts[0]` は0番目のコンテキスト、`contexts[1]` は1番目のコンテキスト……となります。

同様に、ターゲットについても、`target[0]` は 0 番目のターゲット、`target[1]` は 1 番目のターゲット……と格納されます。

それでは、このコンテキストとターゲットを作成する関数を実装します。ここでは `create_contexts_target(corpus, window_size)` という名前で次のように実装します（➡ `common/util.py`）。

```
def create_contexts_target(corpus, window_size=1):
    target = corpus[window_size:-window_size]
    contexts = []

    for idx in range(window_size, len(corpus)-window_size):
        cs = []
        for t in range(-window_size, window_size + 1):
            if t == 0:
                continue
            cs.append(corpus[idx + t])
        contexts.append(cs)

    return np.array(contexts), np.array(target)
```

この関数は引数を 2 つ取ります。ひとつは単語 ID の配列 (`corpus`)、もうひとつはコンテキストのウィンドウサイズ (`window_size`) です。そして、コンテキストとターゲットをそれぞれ NumPy の多次元配列として返します。それでは、この関数を実際に使ってみます。先ほどの実装に続けて書くと次のようになります。

```
contexts, target = create_contexts_target(corpus, window_size=1)

print(contexts)
# [[0 2]
# [1 3]
# [2 4]
# [3 1]
# [4 5]
# [1 6]]

print(target)
# [1 2 3 4 1 5]
```

これで、コーパスからコンテキストとターゲットを作ることができました。後は、これを CBOW モデルに与えるだけです。ただし、このコンテキストとターゲットの各要素は単語 ID のままなので、続いてこれを one-hot 表現に変換しましょう。

3.3.2 one-hot表現への変換

続いて、コンテキストとターゲットを one-hot 表現に変換します。このとき行う変換は、図 3-18 のようになります。

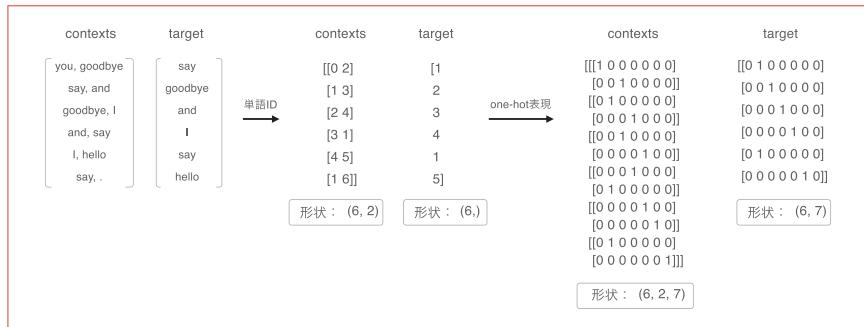


図3-18 「コンテキスト」と「ターゲット」を one-hot 表現へと変換する例

図3-18 のとおり、コンテキストとターゲットを単語 ID から one-hot 表現へと変換します。ここでは、それぞれの多次元配列の形状に注目しましょう。たとえば上の例で、単語 ID を利用したときの contexts の形状は (6, 2) ですが、それを one-hot 表現に変換すると (6, 2, 7) になります。

one-hot 表現への変換は、本書が提供する `convert_one_hot()` 関数を使います。この関数の実装の説明は省略しますが、実装は `common/util.py` にあり、中身はとても簡単です。この関数は引数に、「単語 ID のリスト」と「語彙数」を与えます。それでは、これまでのデータの準備の処理をまとめて書いてみます。結果は、次のようになります。

```

import sys
sys.path.append('..')
from common.util import preprocess, create_contexts_target,
→ convert_one_hot

text = 'You say goodbye and I say hello.'
corpus, word_to_id, id_to_word = preprocess(text)

contexts, target = create_contexts_target(corpus, window_size=1)

vocab_size = len(word_to_id)
target = convert_one_hot(target, vocab_size)

```

```
contexts = convert_one_hot(contexts, vocab_size)
```

これで学習データの準備が終わりました。続いて、本題となる CBOW モデルの実装へと進みます。

3.4 CBOW モデルの実装

それでは、CBOW モデルの実装を行います。繰り返しの掲載になりますが、ここで私たちが実装するニューラルネットワークは、図3-19 のようになります。

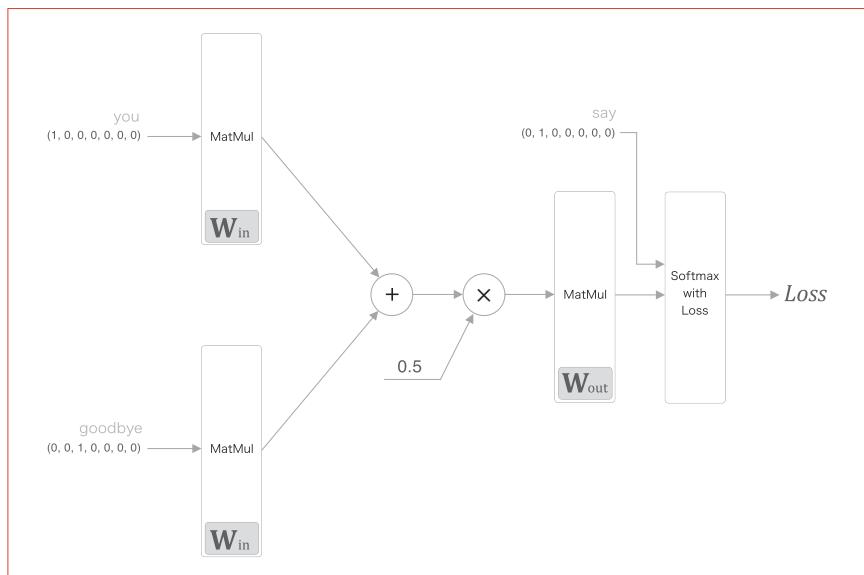


図3-19 CBOW モデルのネットワーク構成

私たちは図3-19 のニューラルネットワークを SimpleCBOW という名前で実装することにします（次章では、これを改良した CBOW クラスを実装します）。それでは初めに、SimpleCBOW クラスのイニシャライザを示します（☞ ch03/simple_cbow.py）。

```
import sys
sys.path.append('..')
import numpy as np
from common.layers import MatMul, SoftmaxWithLoss
```

```

class SimpleCBOW:
    def __init__(self, vocab_size, hidden_size):
        V, H = vocab_size, hidden_size

        # 重みの初期化
        W_in = 0.01 * np.random.randn(V, H).astype('f')
        W_out = 0.01 * np.random.randn(H, V).astype('f')

        # レイヤの生成
        self.in_layer0 = MatMul(W_in)
        self.in_layer1 = MatMul(W_in)
        self.out_layer = MatMul(W_out)
        self.loss_layer = SoftmaxWithLoss()

        # すべての重みと勾配をリストにまとめる
        layers = [self.in_layer0, self.in_layer1, self.out_layer]
        self.params, self.grads = [], []
        for layer in layers:
            self.params += layer.params
            self.grads += layer.grads

        # メンバ変数に単語の分散表現を設定
        self.word_vecs = W_in

```

ここではイニシャライザの引数として、語彙数の `vocab_size` と中間層のニューロン数の `hidden_size` を取ります。まずは重みの初期化についてですが、ここでは重みを 2つ生成します (`W_in` と `W_out`)。この 2つの重みは、それぞれ小さなランダム値によって初期化します。またこのとき NumPy 配列のデータ型を `astype('f')` で指定します。これによって、32 ビットの浮動小数点数で初期化されます。

続いて、必要なレイヤを作成します。まずは入力側の `MatMul` レイヤを 2つ、出力側の `MatMul` レイヤをひとつ、そして `Softmax with Loss` レイヤをひとつ生成します。ここで入力側のコンテキストを処理する `MatMul` レイヤは、コンテキストで使用する単語の数だけ作ります（この例では 2つ）。そして、同じ重みを利用するよう `MatMul` レイヤを初期化します。

最後に、このニューラルネットワークで使われるパラメータと勾配をメンバ変数の `params` と `grads` に配列としてそれぞれまとめます。



ここでは、同じ重みを複数のレイヤで共有しています。そのため、`params` 配列には、同じ重みが複数存在することになります。しかし `params` 配列に同じ重みが存在してしまうと、Adam や Momentum などのオプティマイザの処理が本来の挙動と異なってしまいます（少なくとも、私たちの実装に関しては）。そこで、`Trainer` クラスの内部では、パラメータの更新時にパラメータの重複を取り除く簡単な作業を行っています。ここではその説明は省略します。興味のある方は、`common/trainer.py` の `remove_duplicate(params, grads)` 関数を参照してください。

続いて、ニューラルネットワークの順伝播である `forward()` メソッドを実装します。このメソッドは、引数として `contexts` と `target` の 2 つを取り、損失 (`loss`) を返します。

```
def forward(self, contexts, target):
    h0 = self.in_layer0.forward(contexts[:, 0])
    h1 = self.in_layer1.forward(contexts[:, 1])
    h = (h0 + h1) * 0.5
    score = self.out_layer.forward(h)
    loss = self.loss_layer.forward(score, target)
    return loss
```

ここで引数の `contexts` は 3 次元の NumPy 配列であることを想定します。これは、前節の図 3-18 の例では $(6, 2, 7)$ の形状になります。その 0 番目の次元の要素数はミニバッチの数だけあり、1 番目の次元の要素数はコンテキストのウィンドウサイズ分だけあります。そして、2 番目の次元は one-hot ベクトルで表されます。また、`target` の形状は 2 次元であり、これはたとえば、 $(6, 7)$ のような形状になります。

最後に、逆伝播の `backward()` を実装します。この逆伝播の計算グラフは図 3-20 のようになります。

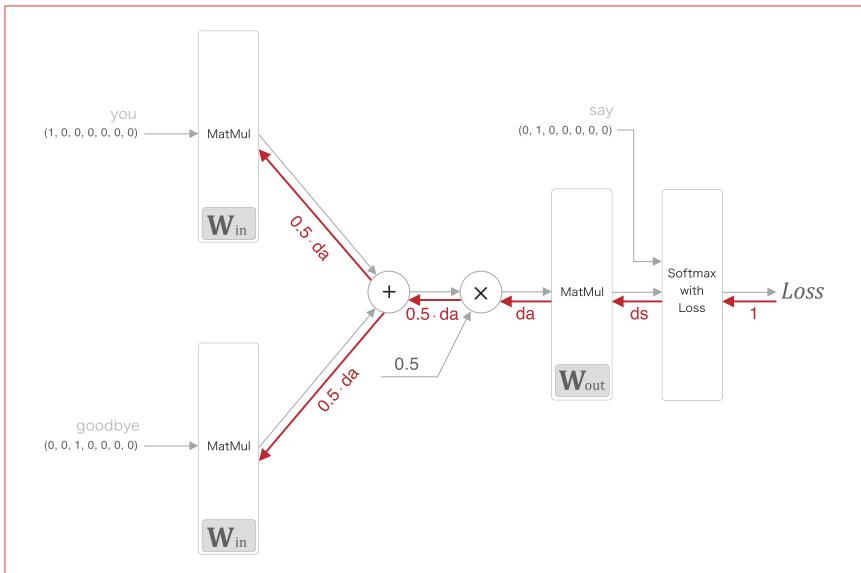


図3-20 CBOW モデルの逆伝播：逆伝播の流れは赤い太線で示す

ニューラルネットワークの逆伝播は、勾配を順伝播とは逆方向に伝播します。この逆伝播は「1」からスタートし、それを Softmax with Loss レイヤへ入力します。そして、Softmax with Loss レイヤの逆伝播の出力を ds として、その ds を出力側の MatMul レイヤへと入力します。

後は「+」と「×」の演算の逆伝播になります。「×」の逆伝播は、順伝播時の入力値を“入れ替えて”勾配に乗算します。「+」の逆伝播は、勾配を“そのまま通す”だけです。それでは、図3-20 に従って、逆伝播の実装を行います。

```
def backward(self, dout=1):
    ds = self.loss_layer.backward(dout)
    da = self.out_layer.backward(ds)
    da *= 0.5
    self.in_layer1.backward(da)
    self.in_layer0.backward(da)
    return None
```

これで逆伝播の実装は終わりです。私たちはすでに、各パラメータの勾配をメンバ変数の `grads` にまとめています。そのため、`forward()` メソッドを呼び、続けて `backward()` メソッドを呼ぶことで、`grads` 配列にある勾配が更新されます。それ

では続いて SimpleCBOW クラスの学習へと進みましょう。

3.4.1 学習コードの実装

CBOW モデルの学習は、通常のニューラルネットワークの学習とまったく同じです。まずは学習データを準備して、ニューラルネットワークに与えます。そして勾配を求めて、重みパラメータを逐一アップデートしていきます。ここでは、その学習プロセスを 1 章で説明した Trainer クラスに行わせます。それでは学習のためのソースコードを示します（☞ ch03/train.py）。

```
import sys
sys.path.append('..')
from common.trainer import Trainer
from common.optimizer import Adam
from simple_cbow import SimpleCBOW
from common.util import preprocess, create_contexts_target,
→ convert_one_hot

window_size = 1
hidden_size = 5
batch_size = 3
max_epoch = 1000

text = 'You say goodbye and I say hello.'
corpus, word_to_id, id_to_word = preprocess(text)

vocab_size = len(word_to_id)
contexts, target = create_contexts_target(corpus, window_size)
target = convert_one_hot(target, vocab_size)
contexts = convert_one_hot(contexts, vocab_size)

model = SimpleCBOW(vocab_size, hidden_size)
optimizer = Adam()
trainer = Trainer(model, optimizer)

trainer.fit(contexts, target, max_epoch, batch_size)
trainer.plot()
```

パラメータの更新を行う手法は、common/optimizer.py の中に、SGD や AdaGrad など有名な手法がいくつか実装されています。ここでは Adam というアルゴリズムを選びました。また、1 章で説明したとおり、Trainer クラスは、ニューラルネットワークの学習を行ってくれます。これは学習データからミニバッチを選び出し、それをニューラルネットワークに与えて勾配を求め、その勾配を

Optimizer に与えてパラメータの更新をするという一連の作業を行います。



これ以降も、ニューラルネットワークの学習では `Trainer` クラスを利用します。`Trainer` クラスを利用することで、複雑になりがちな学習のコードをスッキリさせることができます。

それでは、上のコードを実行してみましょう。その結果は図3-21 のようになります。

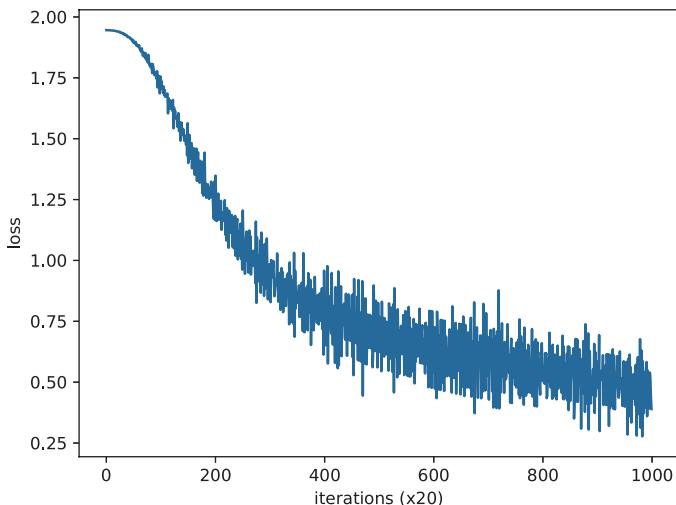


図3-21 学習の経過をグラフで表示（横軸は学習の回数、縦軸は損失）

図3-21 のとおり、学習回数を重ねるごとに、損失が減少していることが分かります。うまく学習できているようです。それでは、学習が終わった後の重みパラメータを見てみましょう。ここでは、入力側の MatMul レイヤの重みを取り出し、実際に中身を確認してみることにします。なお、入力側の MatMul レイヤの重みはメンバ変数の `word_vecs` に設定されています。先ほどのコードに引き続き、次のコードを追加します。

```
word_vecs = model.word_vecs
for word_id, word in id_to_word.items():
```

```
print(word, word_vecs[word_id])
```

ここでは `word_vecs` という名前で重みを取り出します。この `word_vecs` の各行には、対応する単語 ID の分散表現が格納されています。実際に上のコードを実行すると、次の結果が得られます。

```
you [-0.9031807 -1.0374491 -1.4682057 -1.3216232  0.93127245]
say [ 1.2172916  1.2620505 -0.07845993  0.07709391 -1.2389531 ]
goodbye [-1.0834033 -0.8826921 -0.33428606 -0.5720131  1.0488235 ]
and [ 1.0244362  1.0160093 -1.6284224 -1.6400533 -1.0564581]
i [-1.0642933 -0.9162385 -0.31357735 -0.5730831  1.041875 ]
hello [-0.9018145 -1.035476 -1.4629668 -1.3058501  0.9280102]
. [ 1.0985303  1.1642815  1.4365371  1.3974973 -1.0714306]
```

ついに私たちは、単語を密なベクトルで表すことができました！これが単語の分散表現です。この分散表現は、「単語の意味」をうまく捉えたベクトル表現になっていることが期待できます。

しかし残念ながら、ここで扱った小さなコーパスでは良い結果は得られません。もちろんその理由は、コーパスのサイズがあまりにも小さいからです。実際、コーパスを大きく実用的なものに変更すれば、良い結果が得られるでしょう。しかし、その場合は処理速度の点で問題が発生します。というのも、現時点での CBOW モデルの実装は処理効率の点でいくつか問題を抱えているのです。次章では現状の“シンプル”な CBOW モデルに対して改良を加え、“本物”的な CBOW モデルを実装します。

3.5 word2vec に関する補足

これまで私たちは word2vec の CBOW モデルについて詳しく見てきました。ここでは、これまで話せなかった word2vec に関して重要なテーマをいくつか補足したいと思います。まずは CBOW モデルを「確率」の視点からもう一度見ていきます。

3.5.1 CBOW モデルと確率

まずは「確率」の表記について簡単に説明します。本書では確率を $P(\cdot)$ のように表します。たとえば、 A という事象が起こる確率は $P(A)$ と書きます。また、**同時確率**は $P(A, B)$ のように表します。同時確率とは「 A と B が同時に起こる確率」を意味します。

事後確率は $P(A|B)$ のように書きます。これは文字どおり、「**事が起こった後の確**

率」です。これは別の見方をすると、「 B （という情報）が与えられたときに、 A が起こる確率」という解釈もできます。

それでは、CBOW モデルを確率の表記によって記述してみましょう。CBOW モデルが行なうこととは、コンテキストを与えるとターゲットとなる単語の確率を出力することでした。ここでは、 w_1, w_2, \dots, w_T という単語の列で表されるコーパスを扱うことになります。そして、図 3-22 のように、 t 番目の単語に対して、ウィンドウサイズが 1 のコンテキストを考えます。

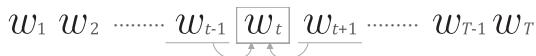


図 3-22 word2vec の CBOW モデル：コンテキストの単語からターゲットとなる単語を推測する

では、コンテキストとして w_{t-1} と w_{t+1} が与えられたときに、ターゲットが w_t となる確率を式で表してみましょう。それには事後確率を使って式 (3.1) のように書くことができます。

$$P(w_t | w_{t-1}, w_{t+1}) \quad (3.1)$$

式 (3.1) は、「 w_{t-1} と w_{t+1} が起きた後に w_t が起こる確率」を表します。そしてそれは、「 w_{t-1} と w_{t+1} が与えられたときに w_t が起こる確率」と解釈できます。つまり、CBOW は式 (3.1) をモデル化しているということです。

ここで式 (3.1) を用いれば、CBOW モデルの損失関数も簡潔に表すことができます。それには、1 章で説明した交差エントロピー誤差（式 (1.7)）を当てはめます。式 (1.7) は、 $L = -\sum_k t_k \log y_k$ であり、 y_k は「 k 番目に対応する事象が起こる確率」を表します。そして、 t_k は教師ラベルであり、これは one-hot ベクトルの要素になります。ここでの問題では、「 w_t が起こる事象」が正解であるため、それに対応する one-hot ベクトルの要素が 1 で、それ以外は 0 になります（つまり、 w_t 以外が起こる場合については、それらに対応する one-hot ラベルの要素は 0 になります）。その点を考慮すると、次の式が導けます。

$$L = -\log P(w_t | w_{t-1}, w_{t+1}) \quad (3.2)$$

CBOW モデルの損失関数は、単に式 (3.1) の確率に対して \log を取り、マイナス

を付けたものになります。ちなみにこれは、**負の対数尤度** (negative log likelihood) と呼ばれます。なお式 (3.2) は、ひとつのサンプルデータに関する損失関数です。これをコーパス全体に拡張すると、損失関数は次のように書けます。

$$L = -\frac{1}{T} \sum_{t=1}^T \log P(w_t | w_{t-1}, w_{t+1}) \quad (3.3)$$

CBOW モデルの学習で行なうことは、式 (3.3) で表される損失関数をできる限り小さくすることです。そして、そのときの重みパラメータが私たちの目的とする単語の分散表現になります。ここではウィンドウサイズが 1 の場合に限定して考えてきましたが、他のウィンドウサイズの場合も（また m などの汎用的なウィンドウサイズの場合も）、簡単に数式で表すことができるでしょう。

3.5.2 skip-gram モデル

前にも述べたとおり、word2vec では 2 つのモデルが提案されています。ひとつはこれまで見てきた CBOW モデル、そしてもうひとつが skip-gram と呼ばれるモデルです。skip-gram は、CBOW で扱うコンテキストとターゲットを逆転させたモデルです。具体例を出すと、それらの 2 つのモデルが解く問題は図 3-23 のようになります。

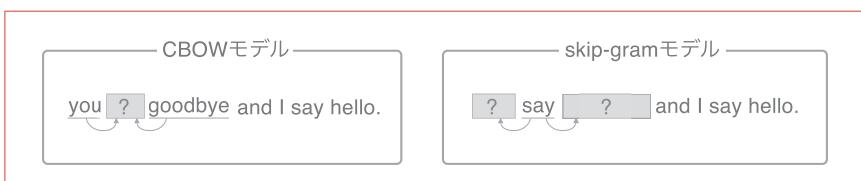


図 3-23 CBOW モデルと skip-gram モデルが扱う問題

図 3-23 に示すように、CBOW モデルはコンテキストが複数あり、その複数のコンテキストから中央の単語（ターゲット）を推測します。一方、skip-gram モデルでは、中央の単語（ターゲット）から、周囲の複数ある単語（コンテキスト）を推測します。このとき、skip-gram モデルのネットワーク構成は図 3-24 のようになります。

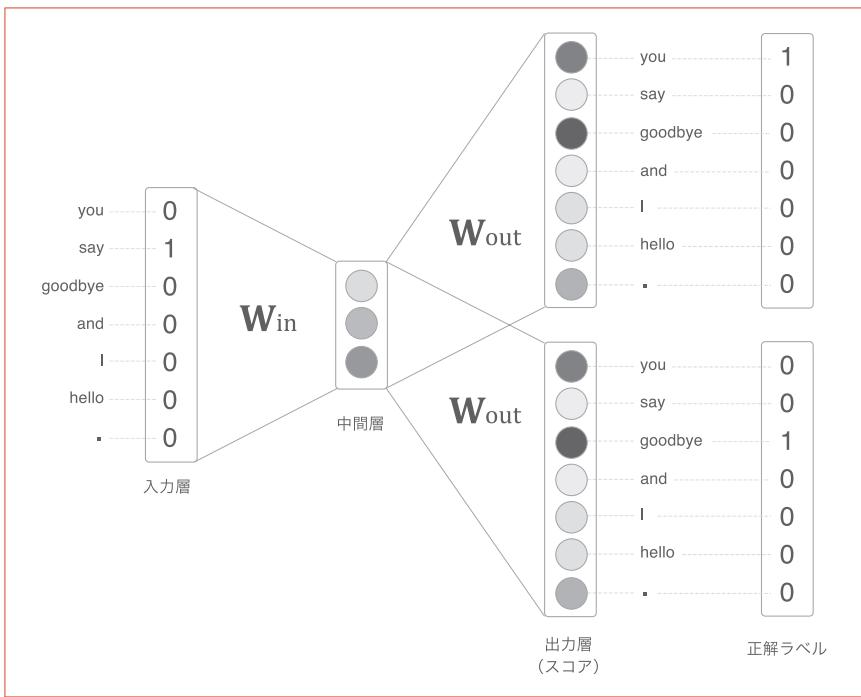


図3-24 skip-gram モデルの例

図3-24に示すように、skip-gram モデルの入力層はひとつです。そして、出力層はコンテキストの数だけ存在します。そのため、それぞれの出力層では個別に損失を求める（Softmax with Loss レイヤなどによって）、それらを足し合わせたものを最終的な損失とします。

それでは、確率の表記を使って、skip-gram モデルを表してみたいと思います。ここで中央の単語（ターゲット）を w_t として、そこからコンテキストの w_{t-1} と w_{t+1} を推測する場合を考えます。このとき、skip-gram は次の式 (3.4) をモデル化します。

$$P(w_{t-1}, w_{t+1} | w_t) \quad (3.4)$$

式 (3.4) は「 w_t が与えられたときに、 w_{t-1} と w_{t+1} が同時に起こる確率」を表します。ここで skip-gram モデルでは、コンテキストの単語の間に関連性がないと仮定し、次のように分解します（これは正しくは「条件付き独立」を仮定しています）。

$$P(w_{t-1}, w_{t+1} | w_t) = P(w_{t-1} | w_t)P(w_{t+1} | w_t) \quad (3.5)$$

そして、式 (3.5) を交差エントロピー誤差に適用することで、skip-gram モデルの損失関数が導けます。

$$\begin{aligned} L &= -\log P(w_{t-1}, w_{t+1} | w_t) \\ &= -\log P(w_{t-1} | w_t)P(w_{t+1} | w_t) \\ &= -(\log P(w_{t-1} | w_t) + \log P(w_{t+1} | w_t)) \end{aligned} \quad (3.6)$$

ここでは、対数の $\log xy = \log x + \log y$ という関係を利用しています。式 (3.6) が示すように、skip-gram モデルの損失関数は、コンテキスト分の損失をそれぞれ求めて、それらを足し合わせたものになります。なお、式 (3.6) はサンプルデータひとつに対する skip-gram の損失関数です。これをコーパス全体に拡張すると、skip-gram モデルの損失関数は次の式 (3.7) で表されます。

$$L = -\frac{1}{T} \sum_{t=1}^T (\log P(w_{t-1} | w_t) + \log P(w_{t+1} | w_t)) \quad (3.7)$$

式 (3.7) を CBOW モデルの式 (3.3) と比較すると、その違いが鮮明になるでしょう。skip-gram モデルはコンテキストの数だけ推測をするため、その損失関数は各コンテキストで求めた損失の総和を求める必要があります。一方、CBOW モデルはひとつつのターゲットの損失を求めます。以上が skip-gram モデルの説明です。

それでは、CBOW モデルと skip-gram モデルのどちらを使うべきでしょうか？その答えは、「skip-gram モデル」と言えるでしょう。これは単語の分散表現の精度の点において、多くの場合、skip-gram モデルのほうが良い結果が得られるからです。特に、コーパスが大規模になるにつれて、低頻出の単語や類推問題の性能の点において、skip-gram モデルのほうが優れた結果が得られる傾向にあります（単語の分散表現の評価方法については「4.4.2 単語ベクトルの評価方法」で説明します）。なお、学習速度の点では、CBOW モデルのほうが skip-gram モデルよりも高速です。これは skip-gram モデルの場合は、コンテキストの数だけ損失を求めるため、その計算コストが大きくなることに原因があります。



skip-gram モデルでは、ひとつの単語からその周囲の単語を予測します。これは、なかなか難しい問題と言えそうです。たとえば、図3-23 の問題を私たちが解く場合を考えてみましょう。このとき、CBOW モデルの問題については「say」と容易に答えが出せるでしょう。しかし、skip-gram モデルの問題については、さまざまな候補が考えられそうです。そういった点で、skip-gram モデルのほうが“タフ”な問題に取り組んでいると言えます。そして、そのタフな問題に鍛えられることで、skip-gram モデルのほうがより優れた単語の分散表現を得られるかもしれません。

さて、skip-gram モデルの実装については、CBOW モデルの実装が理解できていれば、特に難しいことはありません。そのため、ここでは skip-gram の実装の説明は行いません。skip-gram の実装は ch03/simple_skip_gram.py にあるので、興味のある方は参考にしてください。

3.5.3 カウントベース v.s. 推論ベース

これまで私たちはカウントベースの手法と推論ベースの手法（特に word2vec）を見てきました。学習の枠組みにおいては、2つの手法に大きな違いがありました。カウントベースの手法は、コーパスの全体の統計データから1回の学習で単語の分散表現を得ました。一方、推論ベースでは、コーパスの一部を何度も見ながら学習しました（ミニバッチ学習）。ここでは、その他の点において、その2つの手法を比較しながら考察したいと思います。

まず初めに、語彙に新しい単語を追加するケースで、単語の分散表現の更新作業が発生した場面を考えます。このとき、カウントベースの手法ではゼロから計算を行う必要があります。仮に単語の分散表現を少しだけ修正したいとしても、再度、共起行列を作り直し、SVD を行うといった一連の作業が必要になります。それに対して、推論ベースの手法（word2vec）は、パラメータの再学習が行えます。具体的には、これまでに学習した重みを初期値として再学習することで、その学習した経験を損なわずに、単語の分散表現の更新が効率的に行えるのです。その点において、推論ベースの手法（word2vec）のほうが優勢です。

それでは、2つの手法で得られる単語の分散表現の性質や精度についてはどうでしょうか。分散表現の性質について言えば、カウントベースの手法では主に単語の類似性がエンコードされることが分かっています。一方 word2vec（特に skip-gram モデル）では、単語の類似性に加えて、さらに複雑な単語間のパターンも捉えられるこ

とが分かっています。これは、word2vec の「king – man + woman = queen」のような類推問題を解ける話で有名です（類推問題については次章の 4.4.2 節で説明します）。

ここでよくある誤解としては、推論ベースの手法がカウントベースの手法よりも精度的に優れているというものです。実際のところ、単語の類似性に関する定量評価に関して言えば、推論ベースとカウントベースの手法には優劣がつけられないことが報告されています [25]。



「Don't count, predict! (カウントするな、推測せよ！)」というタイトルで始まる論文 [24] が 2014 年に発表されました。この論文では、カウントベースの手法と推論ベースの手法を体系的に比較した結果、推論ベースの手法が常に精度的に上回ったと報告されました。しかしその後、別の論文 [25] によって、単語の類似性に関するタスクにおいては、ハイパーパラメータの依存度が大きく、カウントベースと推論ベースには明確な優劣はつけられないことが報告されています。

また、重要な事実として、推論ベースの手法とカウントベースの手法には、関連性があることが分かっています。具体的には、skip-gram と（次章で扱う）Negative Sampling を利用したモデルは、コーパス全体の共起行列（実際にはそれに少し手を加えた行列）に対して特殊な行列分解をしているのと同じであることが示されたのです [26]。つまり、2つの世界は、（ある条件において）“つながっていた”的な関連性があります。

さらに word2vec 以降、推論ベースとカウントベースの手法を融合させたような GloVe [27] という手法も提案されています。その手法のアイデアは、コーパス全体の統計データの情報を損失関数に取り入れミニバッチ学習をすることにあります（詳細は論文 [27] を参照）。それによって、2つの世界を明示的に融合させることに成功しました。

3.6 まとめ

word2vec は、Tomas Mikolov 氏による一連の論文 ([22] と [23]) によって提案されました。それが発表されてからというもの、word2vec は多くの注目を集めてきました。そして、その有用性は多くの自然言語処理のタスクにおいて示されてきたのです。次章では、word2vec の重要性について——特に word2vec の転移学習の有用性

について——具体例を交えて説明します。

本章では、word2vec の CBOW モデルと呼ばれるニューラルネットワークについて詳しく説明し、その実装を行いました。CBOW モデルは基本的には 2 層のニューラルネットワークで、とてもシンプルな構成です。私たちは、MatMul レイヤと Softmax with Loss レイヤを使って CBOW モデルを構築し、小さなコーパスで学習できることを確認しました。残念ながら、現状の CBOW モデルは、処理効率の点でいくつか問題を抱えています。しかし、本章の CBOW モデルを理解できたならば、本物の word2vec まであと一歩です。それでは続いて、CBOW モデルの改良へと進みましょう！

本章で学んだこと

- 推論ベースの手法は、推測することを目標として、その副産物として単語の分散表現を得られる
- word2vec は推論ベースの手法であり、シンプルな 2 層のニューラルネットワークで構成される
- word2vec には、skip-gram モデルと CBOW モデルがある
- CBOW モデルは複数の単語（コンテキスト）からひとつの単語（ターゲット）を推測する
- skip-gram モデルは逆に、ひとつの単語（ターゲット）から複数の単語（コンテキスト）を推測する
- word2vec は重みの再学習ができるため、単語の分散表現の更新や追加が効率的に行える

4章 word2vecの高速化

何もかも知ろうとして苦労はするな。
さもないと、何ひとつ覚えられない。
—— デモクリトス（古代ギリシアの哲学者）

私たちは前章で word2vec の仕組みを学び、CBOW モデルを実装しました。CBOW モデルはシンプルな 2 層のニューラルネットワークであったため、簡単に実装することができました。しかしその実装にはいくつか問題があります。大きな問題は、コーパスで扱う語彙数が増えるに従って、計算量が増加する点にあります。実際、ある程度の語彙数に到達すると、前章の CBOW モデルでは計算にあまりにも多くの時間がかかるてしまいます。

そこで本章では、word2vec の高速化に主眼を置き、word2vec の改善に取り組みます。具体的には、前章のシンプルな word2vec に対して 2 つの改良を加えます。ひとつ目の改良は Embedding レイヤという新しいレイヤを導入することです。そして、もうひとつの改良として、Negative Sampling という新しい損失関数を導入します。これによって私たちは、“本物” の word2vec を完成させることができます。本物の word2vec が完成した暁には、PTB データセット（実用的なサイズのコーパス）を対象に学習を行います。そして、その単語の分散表現の良さを実際に評価したいと思います。

4.1 word2vec の改良①

それでは前章の復習から始めましょう。私たちは前章で、図 4-1 で表される CBOW

モデルを実装しました。

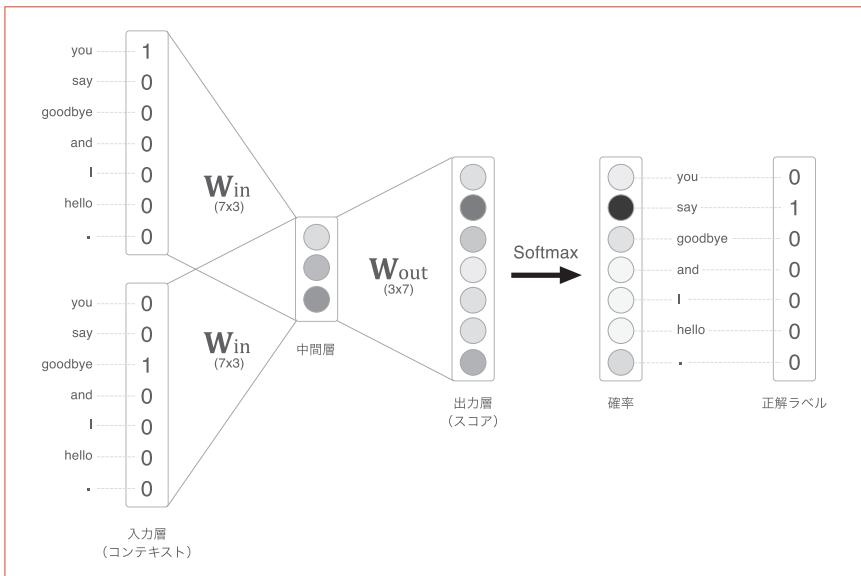


図 4-1 前章で実装した CBOW モデル

図 4-1 のとおり、前章の CBOW モデルは 2 つの単語をコンテキストとして処理し、それを元にひとつの単語（ターゲット）を推測します。このとき、入力側の重み (W_{in}) との行列の積によって中間層が計算され、出力側の重み (W_{out}) との行列の積によって各単語のスコアが求められるでした。そして、そのスコアが Softmax 関数を経て各単語の出現確率が得られ、それを正解ラベルと比較することで——正確には、交差エントロピー誤差を適用することで——損失が求められたのです。



前章では、コンテキストのウインドウサイズを 1 に限定しました。これはつまり、コンテキストとしてターゲットの前後の単語だけを使用したことになります。本章では後ほど、任意のサイズのコンテキストを扱えるように機能追加を行います。

図 4-1 の CBOW モデルは、小さなコーパスを扱う分には特に問題はありません。実際、図 4-1 で扱う語彙数は全部で 7 個でしたが、その規模ではまったく問題なく処理できます。

理できます。しかし、巨大なコーパスを扱う場合、いくつか問題が発生します。その問題点を指摘するにあたり、ここではひとつの例として、語彙数が 100 万、中間層のニューロン数が 100 の場合における CBOW モデルを考えたいと思います。このとき word2vec で行う処理は、次のようにになります。

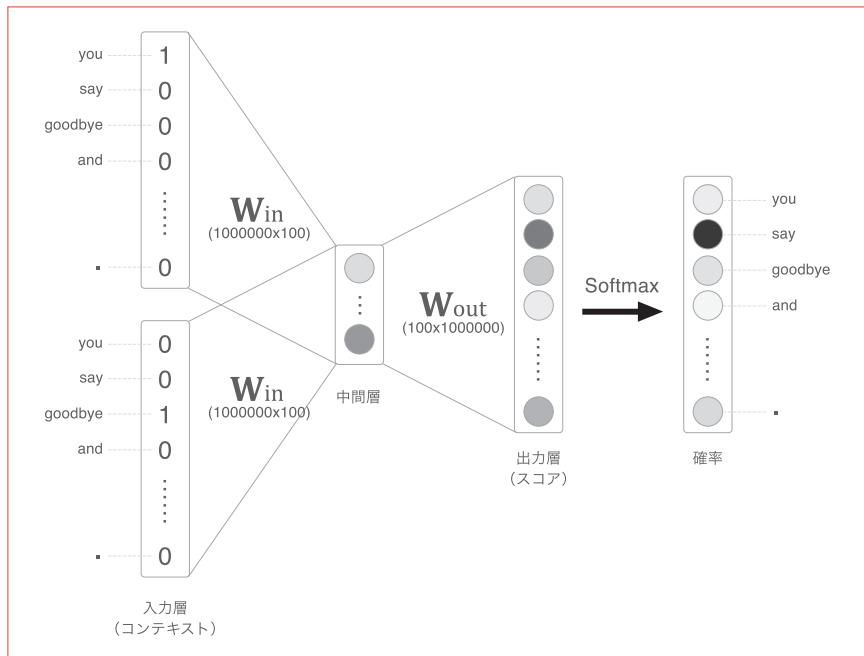


図 4-2 語彙数が 100 万のときを想定した CBOW モデル

図 4-2 が示すとおり、入力層と出力層には 100 万個のニューロンが存在します。この巨大なニューロンによって、途中の計算では多くの時間を要します。具体的には、このとき次の 2箇所の計算がボトルネックとなります。

- 入力層の one-hot 表現と重み行列 W_{in} の積による計算（4.1 節で解決）
- 中間層と重み行列 W_{out} の積および Softmax レイヤの計算（4.2 節で解決）

ひとつ目の問題は、入力層の one-hot 表現に関する箇所です。これは、単語を one-hot 表現で扱っているため、語彙数が増えるにつれ one-hot 表現のベクトルのサ

イズも増えることに起因します。たとえば、語彙数が 100 万の場合、その one-hot 表現だけでも 100 万の要素を占めるメモリサイズが必要になります。さらに、その one-hot ベクトルと重み行列 \mathbf{W}_{in} の積の計算を行う必要があり、それだけで多くの計算リソースが必要になってしまいます。この問題は、4.1 節で Embedding レイヤを新たに導入することで解決します。

2 つ目の問題は、中間層以降の計算です。まずは中間層と重み行列 \mathbf{W}_{out} の積で多くの計算が必要になります。そして、Softmax レイヤに関わる箇所でも、扱う語彙数が増えるにつれて計算量が増加することが問題になります。この問題については、4.2 節で Negative Sampling という新しい損失関数を導入することで解決します。それでは、その 2 つのボトルネックを解消すべく、それぞれ改良を行っていきましょう。



改良前のファイル（前章の word2vec の実装）は、ch03 ディレクトリの `simple_cbow.py`（もしくは `simple_skip_gram.py`）にあります。改良後の word2vec のファイルは、ch04 ディレクトリの `cbow.py`（もしくは `skip_gram.py`）にあります。

4.1.1 Embedding レイヤ

前章の word2vec の実装では、単語を one-hot 表現に変換しました。そしてそれを MatMul レイヤに入力し、MatMul レイヤ内で one-hot ベクトルと重み行列の積を計算したのです。ここで単語の語彙数が 100 万の場合を想像してみましょう。このとき、中間層のニューロンの数を 100 とすると、MatMul レイヤにおける行列の積は図4-3 のように書けます。

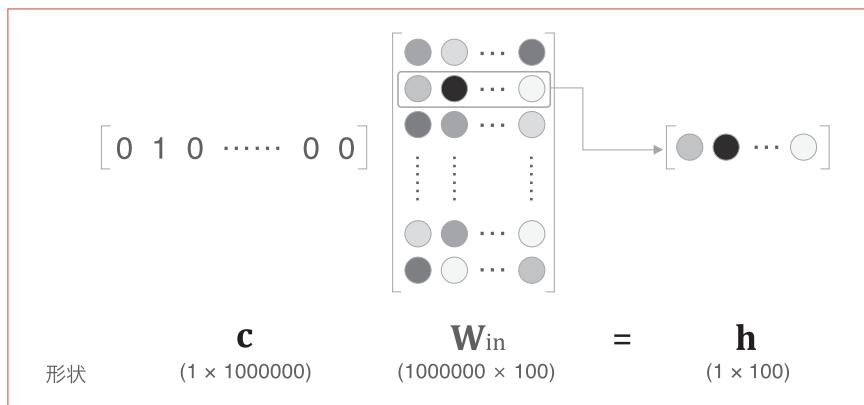


図4-3 コンテキスト (one-hot 表現) と MatMul レイヤの重みの積による計算

図4-3で示すように、もし100万語の語彙数からなるコーパスがあるとしたら、単語のone-hot表現の次元数も100万になります。そして、そのような巨大なベクトルと重み行列の積を計算する必要があるのです。しかし、図4-3で行っていることは、單行列の特定の行を抜き出すことだけです。そのため素直に考えれば、one-hot表現への変換とMatMulレイヤでの行列の乗算は必要なさそうです。

それでは、重みパラメータから「単語IDに該当する行（ベクトル）」を抜き出すためのレイヤを作りましょう。ここでは、そのレイヤをEmbeddingレイヤと呼ぶことにします。ちなみに、Embeddingとは、単語の埋め込み（word embedding）という用語に由来します。つまり、このEmbeddingレイヤに単語の埋め込み（分散表現）が格納されるのです。



自然言語処理の分野では、単語の密なベクトル表現は、**単語の埋め込み**（word embedding）や、単語の**分散表現**（distributed representation）と呼ばれます。また細かな違いとして、カウントベースの手法によって得られた単語ベクトルはdistributional representation、ニューラルネットワークを使った推論ベースの手法で得られた単語ベクトルはdistributed representationと呼ばれてきた過去があります。ただし、日本語ではともに「分散表現」と訳されます。

4.1.2 Embeddingレイヤの実装

行列から行を抜き出す処理は簡単に行えます。たとえばここで、重み W が NumPy

の 2 次元配列であるとします。このとき、この重みから特定の行を抜き出すには、単に $W[2]$ や $W[5]$ のように書くだけです。実際に Python で書くと次のようにになります。

```
>>> import numpy as np
>>> W = np.arange(21).reshape(7, 3)
>>> W
array([[ 0,  1,  2],
       [ 3,  4,  5],
       [ 6,  7,  8],
       [ 9, 10, 11],
       [12, 13, 14],
       [15, 16, 17],
       [18, 19, 20]])
>>> W[2]
array([6, 7, 8])
>>> W[5]
array([15, 16, 17])
```

また、重み W から複数の行をまとめて抽出することも簡単に行えます。それには、配列によって行の番号を指定するだけです。実際にやってみると、次のようになります。

```
>>> idx = np.array([1, 0, 3, 0])
>>> W[idx]
array([[ 3,  4,  5],
       [ 0,  1,  2],
       [ 9, 10, 11],
       [ 0,  1,  2]])
```

この例では、4 つのインデックス（1、0、3、0 番目）をまとめて抽出しています。引数に配列を与えることで、複数の行をまとめて抽出できます。ちなみにこれは、ミニバッチ処理を想定した場合の実装です。

それでは、Embedding レイヤの `forward()` メソッドを実装しましょう。これまでの例を考えれば、次のような実装になります（[common/layers.py](#)）。

```
class Embedding:
    def __init__(self, W):
        self.params = [W]
        self.grads = [np.zeros_like(W)]
        self.idx = None

    def forward(self, idx):
        W, = self.params
```

```
self.idx = idx
out = W[idx]
return out
```

本書の実装ルールに従い、メンバ変数として `params` と `grads` を使用します。また、メンバ変数の `idx` には、抽出する行のインデックス（単語 ID）を配列として格納します。

続いて、逆伝播 (backward) について考えます。Embedding レイヤの順伝播は、重み W の特定の行を抜き出すだけでした。これは単に、重みの特定の行のニューロンだけを——何の手も加えず——次の層へと流したことになります。そのため逆伝播では、前の層（出力側の層）から伝わってきた勾配を次の層（入力側の層）へそのまま流すだけになります。ただし、前層から伝わる勾配を、重みの勾配 dW の特定の行 (`idx`) に設定するようにします。これは図で表すと、図 4-4 のようになります。

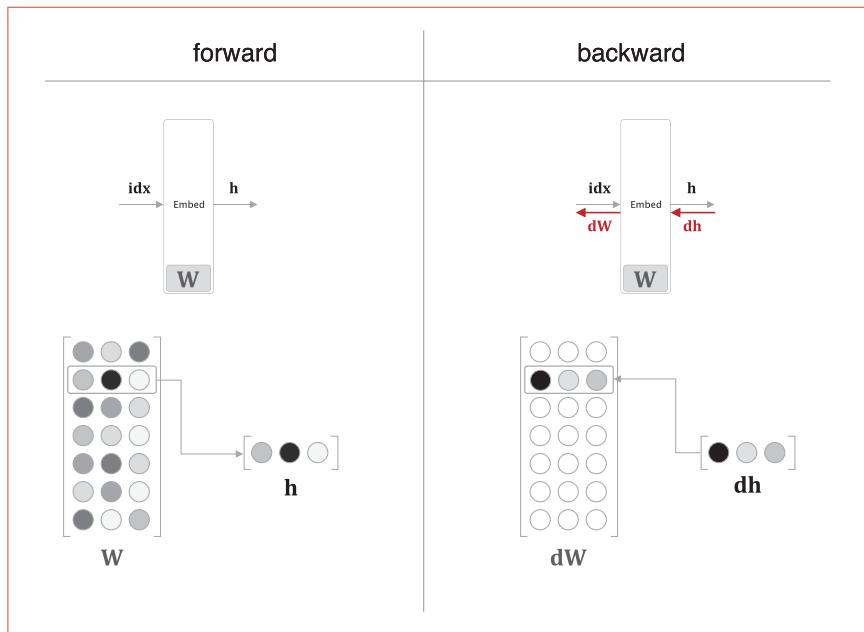


図 4-4 Embedding レイヤの forward と backward 处理の概要 (Embedding レイヤは Embed と表記)

以上を踏まえて、`backward()` の実装を行いましょう。これは次のように書くことができそうです。

```
def backward(self, dout):
    dW, = self.grads
    dW[...] = 0
    dW[self.idx] = dout # 実は悪い例
    return None
```

ここでは、重みの勾配を dW として取り出し、 $dW[...] = \theta$ で dW の要素を 0 で上書きします (dW を 0 にするのではなく、 dW の形状を保ったまま、その要素を 0 にします)。そして、前層から伝わってくる勾配 $dout$ を、 idx で指定された行に代入します。



ここでは、重み W と同じ大きさの行列 dW を作成し、その dW の該当する行に勾配を代入しました。しかし、最終的に行いたいことは、重み W の更新であるため、わざわざ dW のような (W と同じ大きさの) 行列を作る必要はありません。そうではなく、更新したい行番号 (idx) とその勾配 ($dout$) を保持しておけば、その情報から重み (W) の特定の行だけを更新することができます。ただしここでは、すでに実装している更新用のクラス (`Optimizer`) と組み合わせて使用することを考えて、このような実装にしています。

さて、先の `backward()` の実装には、実はひとつ問題があります。その問題は、 idx の要素が重複するときに発生します。たとえば、 idx が $[0, 2, 0, 4]$ のような場合を考えてみましょう。このとき、図 4-5 の問題が発生します。

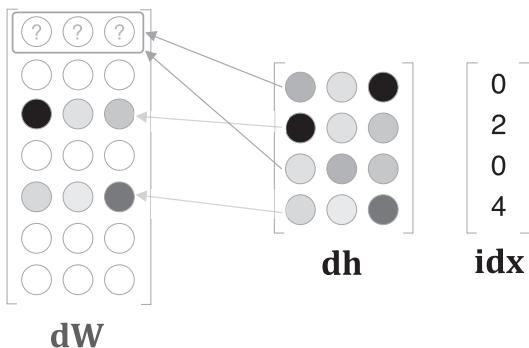


図 4-5 idx 配列の要素に同じ行番号がある場合、 dh の行を代入するだけでは問題が起こる

図4-5 に示すとおり、`dh` の各行の値を `idx` で指定された場所に代入してみます。そうするとこの場合、`dW` の 0 番目の行に 2 つの値が代入されることになります。それでは、どちらかの値が上書きされてしまいます。

この重複問題に対応するためには、「代入」ではなく「加算」を行う必要があります（なぜ加算を行うかは、各自で考えてみましょう）。つまり、`dh` の各行の値を、`dW` の対応する行に加算するのです。それでは、正しい逆伝播の実装を次に示します。

```
def backward(self, dout):
    dW, = self.grads
    dW[...] = 0

    for i, word_id in enumerate(self.idx):
        dW[word_id] += dout[i]
    # もしくは
    # np.add.at(dW, self.idx, dout)

    return None
```

ここでは、`for` 文を使って該当するインデックスに勾配を加算します。これで、`idx` に重複するインデックスがあったとしても正しく処理されます。なお、ここでの `for` 文を使った実装は、NumPy の `np.add.at()` でも行えます。`np.add.at(A, idx, B)` は、`B` を `A` に加算しますが、その際、加算を行う `A` の行を `idx` によって指定します。



一般的に、Python で `for` 文を使って処理するよりも、NumPy の組み込みメソッドを使ったほうが高速に処理できます。これは NumPy のメソッドが、低レイヤにおいて高速化や処理効率向上のためのチューニングが行われているからです。そのため、上のソースコードでは、`for` 文を使った実装よりも、`np.add.at()` を使ったほうが処理効率は断然良くなります。

以上で Embedding レイヤの実装は終わりです。これで word2vec (CBOW モデル) の実装では、入力側の MatMul レイヤを Embedding レイヤに切り替えることができます。それによって、メモリの使用量を減らし、さらに無駄な計算を省くことができるようになりました！

4.2 word2vec の改良②

続いて、word2vec の 2 つ目の改良に取り組みます。前にも述べたとおり、残るボトルネックは中間層以降の処理——行列の積と Softmax レイヤの計算——です。このボトルネックの解消が、本節の目標です。ここではその解決策として、**Negative Sampling**（負例サンプリング）と呼ばれる手法を用います。Softmax の代わりに Negative sampling を用いることで、語彙数がどれだけ多くなったとしても、計算量を少なく一定に抑えることができるのです。

本節の話は、いくぶん複雑になります。特に実装面がやや込み入ったものになります。そのためここでは、ひとつずつ確認しながらステップ・バイ・ステップで進んでいきたいと思います。

4.2.1 中間層以降の計算の問題点

中間層以降の計算の問題点を指摘するにあたり、前節と同様に語彙数が 100 万、中間層のニューロン数が 100 のときの word2vec (CBOW モデル) を考えることにします。このとき word2vec で行う処理は、次のようにになります。

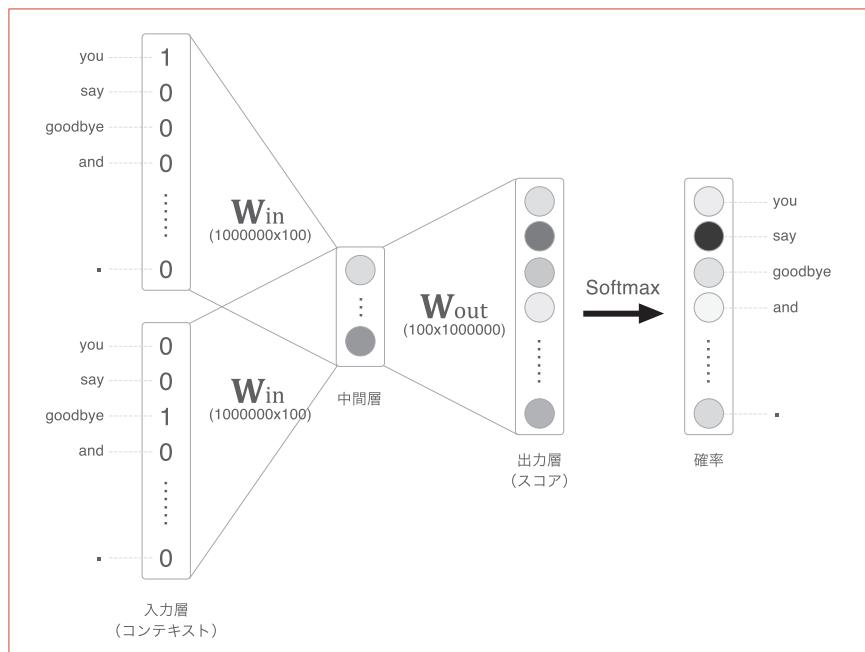


図4-6 語彙数が100万のときを想定したword2vec:「you」と「goodbye」をコンテキストとして、「say」をターゲット（予測すべき単語）とする

図4-6が示すとおり、入力層と出力層には100万個のニューロンが存在します。前節ではEmbeddingレイヤを導入することで、入力層の計算については無駄を省くことができました。残る問題は、中間層以降の処理です。このとき、次の2つの場所において多くの計算時間が必要になります。

- 中間層のニューロンと重み行列 (W_{out}) の積
- Softmax レイヤの計算

ひとつ目の問題は巨大な行列の積についてです。上の例では、中間層のベクトルのサイズが100、重み行列のサイズが 100×100 万になりますが、そのような巨大な行列の積の計算は多くの時間を要します（そして、多くのメモリを必要とします）。さらに、逆伝播のときにも同様の計算を行う必要があるため、その行列の積の計算を“軽く”することが求められます。

2つ目の Softmax についても、同じ問題が発生します。つまり、語彙数が増えるに従い、Softmax も計算量が増加するのです。これは、次の Softmax の式を見ると、より明確になるでしょう。

$$y_k = \frac{\exp(s_k)}{\sum_{i=1}^{1000000} \exp(s_i)} \quad (4.1)$$

式 (4.1) は、 k 番目の要素（単語）を対象としたときの Softmax の計算式です（スコアの各要素は、 s_1, s_2, \dots ）。ここでは、語彙数が 100 万であることを想定しているため、式 (4.1) の分母の計算については、 \exp の計算を 100 万回も行う必要があるのです。この計算も語彙数に比例して増加することになるため、Softmax に代わる“軽い”計算が求められます。

4.2.2 多値分類から二値分類へ

これから Negative sampling という手法について説明していきます。早速ですが、この手法のキーとなるアイデアは「二値分類」にあります。より正確に言うと、「多値分類」を「二値分類」で近似すること——これが、Negative sampling を理解する上で重要なポイントになります。

さて、これまで私たちは「多値分類」の問題を扱ってきました。先ほどの例について言えば、それは、100 万個の単語の中から正しい単語をひとつ選ぶ問題として考えてきたのです。それでは、そのような問題を「二値分類」の問題として扱えないでしょうか？より正確に言うと、「多値分類」の問題を「二値分類」の問題で近似できないでしょうか？



二値分類は、「Yes/No」で答える問題を扱います。たとえば「この数字は 7 ですか？」、「これは猫ですか？」、「ターゲットになる単語は『say』ですか？」といった質問が、二値分類の扱う問題です。このような問題にはすべて「Yes/No」で答えることができます。

これまで私たちが行ってきたことは、コンテキストを与えたとき、正解となる単語を高い確率で推測できるようにすることでした。たとえば、コンテキストとして「you」と「goodbye」を与えたときに、正解である「say」という単語の確率が高くなるように、ニューラルネットワークの学習を行ったのです。そして、うまく学習を

行えば、そのニューラルネットワークは正しい推測を行えるようになります。つまりこのとき、「コンテキストが『you』と『goodbye』のとき、ターゲットとなる単語は何ですか?」という質問に、ニューラルネットワークが正しい答えを出すことができます。

ここで私たちが考えるべきは、「多値分類」の問題を「二値分類」として解決することです。そのためには、「Yes/No」で答えを出せるような質問を考えます。たとえば、「コンテキストが『you』と『goodbye』のとき、ターゲットとなる単語は『say』ですか?」という質問に答えるニューラルネットワークを考えるのです。そうした場合、出力層にはニューロンをひとつだけ用意すれば事足ります。その出力層のニューロンが、「say」であるスコアを出力すると考えることができます。

それでは、このとき CBOW モデルはどのような処理を行うのでしょうか? 実際に図で表すと、それは図 4-7 のように書けます。

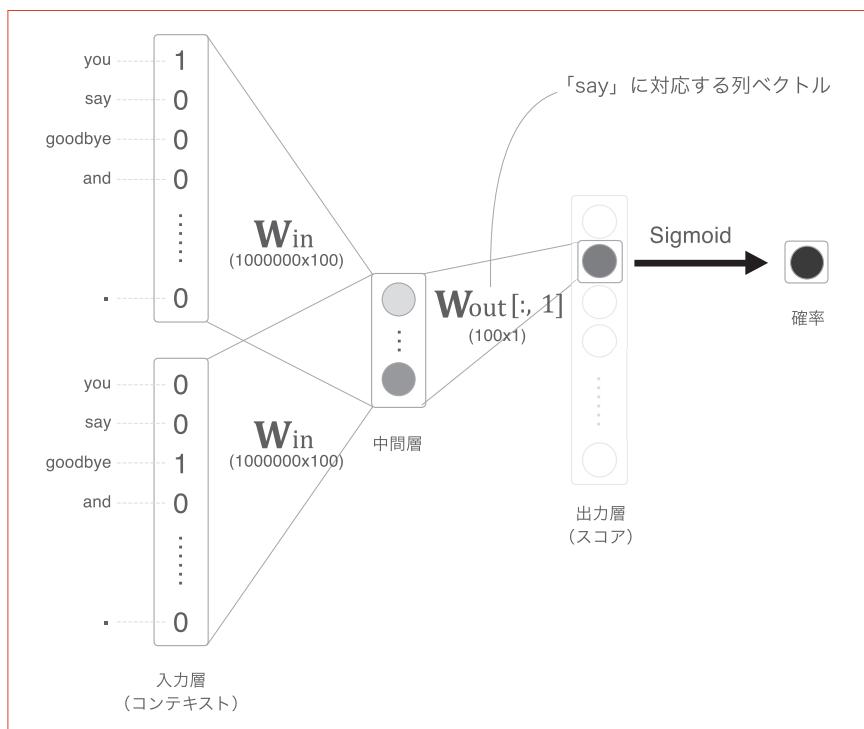


図 4-7 ターゲットとなる単語だけのスコアを求めるニューラルネットワーク

図4-7に示すように、出力層のニューロンはひとつだけです。そのため、中間層と出力側の重み行列の積は、「say」に対応する列（単語ベクトル）だけを抽出し、その抽出したベクトルと中間層のニューロンとの内積を計算すればよいことになります。この計算は、詳しく書くと図4-8のようになります。

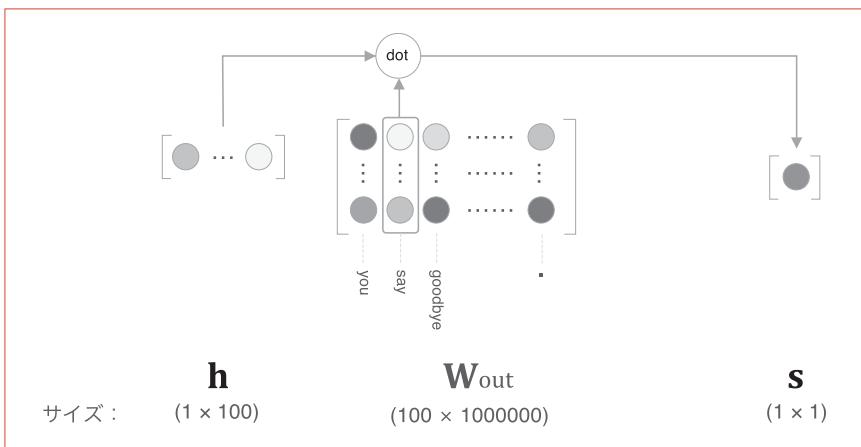


図4-8 「say」に対応する列ベクトルと中間層の内積を計算する（図中の「dot」は内積の計算）

図4-8に示すように、出力側の重み W_{out} では、各単語 ID の単語ベクトルが各列に格納されています。ここでは、「say」という単語ベクトルを抽出します。そして、そのベクトルと中間層のニューロンとの内積を求めます。これが最終的なスコアになります。



これまでの出力層では、すべての単語を対象に計算を行いました。ここでは「say」という単語だけに着目し、そのスコアだけを計算します。そして、そのスコアを確率に変換するために、シグモイド関数を適用します。

4.2.3 シグモイド関数と交差エントロピー誤差

二値分類の問題をニューラルネットワークで解くには、スコアにシグモイド関数を適用し、確率を得ます。そして損失を求めるために、損失関数として「交差エントロピー誤差」を使用します。これは二値分類におけるニューラルネットワークの常套手段です。



多値分類の場合、出力層には（スコアを確率へ変換するには）「ソフトマックス関数」、損失関数には「交差エントロピー誤差」を用います。二値分類の場合、出力層には「シグモイド関数」、損失関数には「交差エントロピー誤差」を用います。

ここではシグモイド関数について復習したいと思います。早速ですが、シグモイド関数は、式 (4.2) のように書くことができました。

$$y = \frac{1}{1 + \exp(-x)} \quad (4.2)$$

式 (4.2) はグラフで書くと、図 4-9 の右図のようになります。図を見て分かるとおり、グラフの形状は S 字カーブをしており、入力された値 (x) は 0 から 1 の間の実数へと変換されます。ここでポイントは、シグモイド関数の出力 (y) が「確率」として解釈できるということです。

なお、シグモイド関数については、Sigmoid レイヤとしてすでに実装済みです。これは図 4-9 の左図のように書けます。

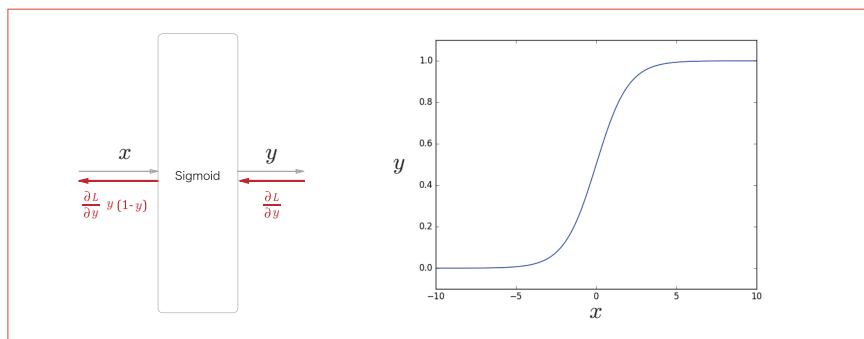


図 4-9 左図：Sigmoid レイヤ。右図：シグモイド関数のグラフ

シグモイド関数によって確率 y を得たら、この確率 y から損失を求めます。シグモイド関数に対して使用される損失関数は、多値分類のときと同じく「交差エントロピー誤差」です。この交差エントロピー誤差は、次のように書くことができます。

$$L = -(t \log y + (1 - t) \log (1 - y)) \quad (4.3)$$

ここで y はシグモイド関数の出力、 t は正解ラベルとします。この正解ラベル t は、0 か 1 のどちらかの値を取ります。 t が 1 のときは正解が「Yes」、 t が 0 のときは正解が「No」です。そのため、 t が 1 のときは $-\log y$ が出力され、逆に t が 0 のときは $-\log(1 - y)$ が出力されます。



二値分類と多値分類では両者ともに、損失関数には「交差エントロピー誤差」を使います。それぞれ式で表すと式 (4.3) と式 (1.7) になりますが、それらは式の書き方が違うだけで、同じことを表しています。正確に言うと、多値分類のときで出力層に 2 つのニューロンを利用した場合は、二値分類の式 (4.3) と完全に一致します。そのため、Sigmoid with Loss レイヤの実装は、Softmax with Loss レイヤの実装に、少しだけ手を加えれば完成します。

続いて、Sigmoid レイヤと Cross Entropy Error レイヤを図で表します。これらは図で書くと、図 4-10 のようになります。

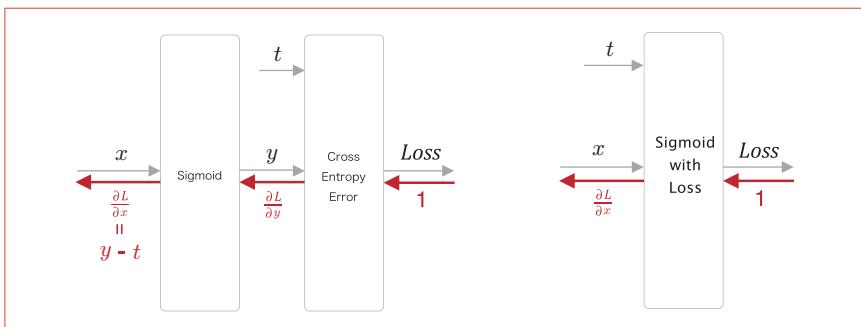


図 4-10 Sigmoid レイヤと Cross Entropy Error レイヤの計算グラフ。右図は Sigmoid with Loss レイヤとして、まとめて記載する

図 4-10 に関して注目すべき点は、逆伝播の $y - t$ という値です。ここで y はニューラルネットワークが出力した確率で、 t は正解ラベルを表します。そして、 $y - t$ はちょうどその 2 つの値の差になっています。これはたとえば、正解ラベルが 1 のとき、 y ができるだけ 1 (100%) に近づくと、その誤差は小さくなるということを意味します。逆に y が 1 から遠ざかると、その誤差は大きくなります。そして、その誤差が前レイヤに流れていくことで、大きい誤差の場合は“大きく”学習し、小さい誤差の場合は“小さく”学習するようになるのです。



「シグモイド関数」と「交差エントロピー誤差」の組み合わせによって、逆伝播の値が $y - t$ という“キレイな結果”になりました。同様に、「ソフトマックス関数」と「交差エントロピー誤差」の組み合わせ、また、「恒等関数」と「2乗和誤差」の組み合わせも、逆伝播時には $y - t$ の値が伝播します。

4.2.4 多値分類から二値分類へ（実装編）

ここまで話を、実装を行う視点で整理したいと思います。私たちはこれまで、多値分類の問題を扱ってきました。そこでは、出力層に語彙数の分だけニューロンを用意し、それを Softmax レイヤに通したのです。このとき用いられるニューラルネットワークを、「レイヤ」と「演算」に着目して図示すると、図 4-11 のように書くことができます。

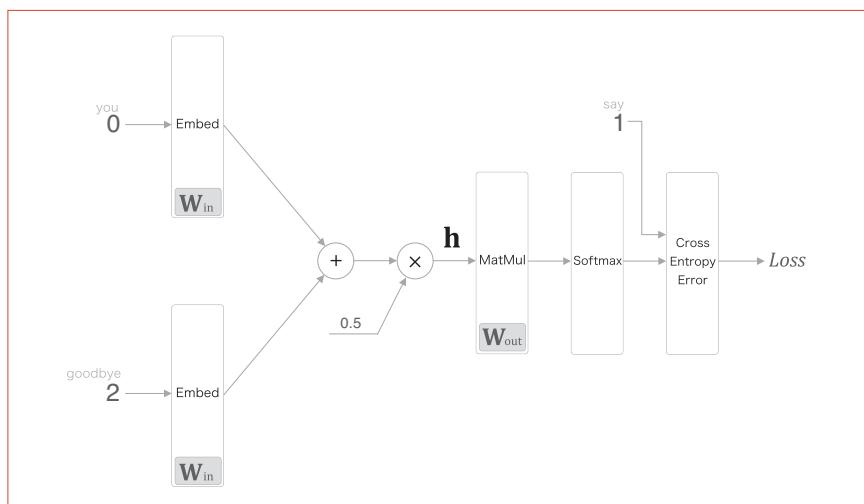


図 4-11 多値分類を行う CBOW モデルの全体図（Embedding レイヤは Embed と表記）

図 4-11 では、コンテキストが「you」と「goodbye」、正解となるターゲット（予測すべき単語）が「say」の場合の例を示しています（単語 ID については、「you」が 0、「say」が 1、「goodbye」が 2 だと仮定しています）。また入力層では、それぞれに対応する単語 ID の分散表現を抜き出すために、Embedding レイヤを用いています。



前節では Embedding レイヤを実装しました。このレイヤは対象とする単語 ID の分散表現（単語ベクトル）を抜き出します。以前は、これを MatMul レイヤで代替していました。

それでは、図4-11 のニューラルネットワークを二値分類を行うネットワークに変換します。早速そのネットワーク構成を示すと、それは図4-12 のようになります。

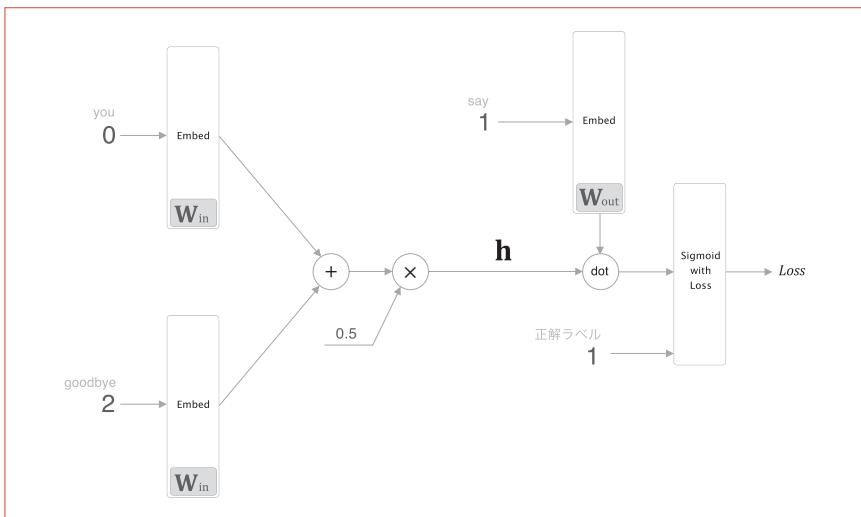


図4-12 二値分類を行う word2vec (CBOW モデル) の全体図

ここでは中間層のニューロンを \mathbf{h} として、出力側の重み \mathbf{W}_{out} の単語「say」に対応する単語ベクトルとの内積を計算しています。そして、その出力を Sigmoid with Loss レイヤに入れることで、最終的な損失を得ます。



図4-12 では、Sigmoid with Loss レイヤに正解ラベルとして「1」を入力しています。これは、今扱っている問題の答えが「Yes」であることを意味します。答えが「No」の場合は、Sigmoid with Loss レイヤに正解ラベルとして「0」を入力します。

さて、これから先の話の見通しを良くするため、ここでは図4-12 の後半部分をよりシンプルにしたいと思います。そのため、Embedding Dot レイヤを導入しま

す。このレイヤは、図4-12のEmbeddingレイヤと「dot演算（内積）」の2つの処理を合わせたレイヤです。このレイヤを使うことで、図4-12の後半部分は次のように書くことができます。

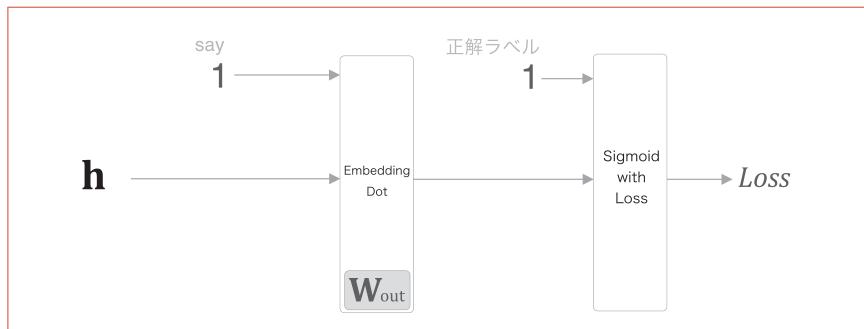


図4-13 図4-12の中間層以降の処理だけにフォーカスして描画する。Embedding Dot レイヤを使って、Embedding レイヤと内積の計算をまとめて行う

中間層のニューロン h は、Embedding Dot レイヤを経て、Sigmoid with Loss レイヤを通ります。見てのとおり、Embedding Dot レイヤを使うことで、中間層以降の処理をシンプルに書くことができました。

それでは、Embedding Dot レイヤの実装について簡単に見ていきましょう。ここでは、このレイヤを `EmbeddingDot` クラスとして次のように実装します ([ch04/negative_sampling_layer.py](#))。

```

class EmbeddingDot:
    def __init__(self, W):
        self.embed = Embedding(W)
        self.params = self.embed.params
        self.grads = self.embed.grads
        self.cache = None

    def forward(self, h, idx):
        target_W = self.embed.forward(idx)
        out = np.sum(target_W * h, axis=1)

        self.cache = (h, target_W)
        return out

    def backward(self, dout):
        h, target_W = self.cache

```

```
dout = dout.reshape(dout.shape[0], 1)

dtarget_W = dout * h
self.embed.backward(dtarget_W)
dh = dout * target_W
return dh
```

`EmbeddingDot` クラスには、全部で 4 つのメンバ変数——`embed`、`params`、`grads`、`cache`——があります。本書の実装ルールどおり、`params` にはパラメータを、`grads` には勾配をそれぞれ格納します。また、`embed` は Embedding レイヤを、`cache` は順伝播の際に計算した結果を一時的に保持するための変数として使用します。

順伝播の `forward(h, idx)` メソッドでは、引数に中間層のニューロン (`h`) と単語 ID の NumPy 配列 (`idx`) を受け取ります。ここで `idx` は単語 ID の配列になっていますが、これはデータをまとめて処理する「ミニバッチ処理」を想定しているためです。

上のコードの `forward()` メソッドでは、まず Embedding レイヤの `forward(idx)` を呼び、続いて内積の計算を行います。内積の計算は、`np.sum(self.target_W * h, axis=1)` の 1 行によって行われます。この実装を理解するには、具体的な値を見るのが手っ取り早いでしょう。そこで、図 4-14 に具体例を示します。

| W | idx | target_W | h | target_W * h | out |
|-------------|---------|-------------|----------|--------------|-------------|
| [[0 1 2] | [0 3 1] | [[0 1 2] | [[0 1 2] | [[0 1 4] | [5 122 86] |
| [3 4 5] | | [9 10 11] | [3 4 5] | [27 40 55] | |
| [6 7 8] | | [3 4 5]] | [6 7 8]] | [18 28 40]] | |
| [9 10 11] | | | | | |
| [12 13 14] | | | | | |
| [15 16 17] | | | | | |
| [18 19 20]] | | | | | |

図 4-14 Embedding Dot レイヤ内の各変数の具体的な値

図 4-14 に示すとおり、適当な `W` と `h`、そして `idx` を用意します。ここで `idx` は `[0, 3, 1]` ですが、これは 3 つのデータをミニバッチとしてまとめて処理する例を表しています。`idx` は `[0, 3, 1]` なので、`target_W` は、`W` の 0 番、3 番、1 番目の

行を抜き出した結果になります。そして、`target_W * h` では、要素ごとの積が計算されます（NumPy の「`*`」は要素ごとの積）。そして、その結果を行ごとに（`axis=1` によって）総和を求めて、最終的な結果 `out` を得ます。

Embedding Dot レイヤの順伝播の説明は以上になります。逆伝播については、順伝播とは逆順に勾配を伝達していくことで求められます。ここでは、その実装の説明については省略します（難しい問題ではないので、各自で考えてみましょう）。

4.2.5 Negative Sampling

ここまで議論によって、解くべき問題を「多値分類」から「二値分類」へと変換することができました。しかし、これで問題が解決できたかというと、残念ながらそうではありません。というのも、今のままでは、正例（正しい答え）についてだけ学習を行ったに過ぎないからです。そのため、負例（誤った答え）については、どのような結果になるのか定かではありません。

それでは前の例について、もう一度考えましょう。前の例とは、コンテキストが「you」と「goodbye」で、正解となるターゲットが「say」の場合です。私たちはこれまでのところ、正例の「say」だけを対象に二値分類を行ってきました。もしここで、“良い重み”があれば、Sigmoid レイヤの出力（確率）は 1 に近づくことになるでしょう。このときの処理を計算グラフで表せば、図 4-15 のようになります。

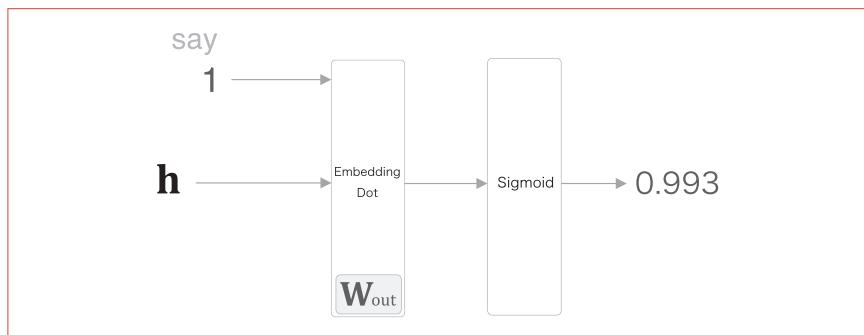


図 4-15 CBOW モデルの中間層以降の処理の例：ここでは、コンテキストが「you」と「goodbye」とする。このとき、ターゲットが「say」である確率は 0.993 (99.3%)

現状のニューラルネットワークでは、正例の「say」についてだけ学習を行うことになります。しかし、負例——「say」以外の単語——については、何の知識も身に

ついていません。ここで私たちが本当に行いたいことは何でしょうか。それは、正例（「say」）については Sigmoid レイヤの出力を 1 に近づけ、負例（「say」以外の単語）については、Sigmoid レイヤの出力を 0 に近づけることなのです。これは図で表すと、図 4-16 のように書けます。

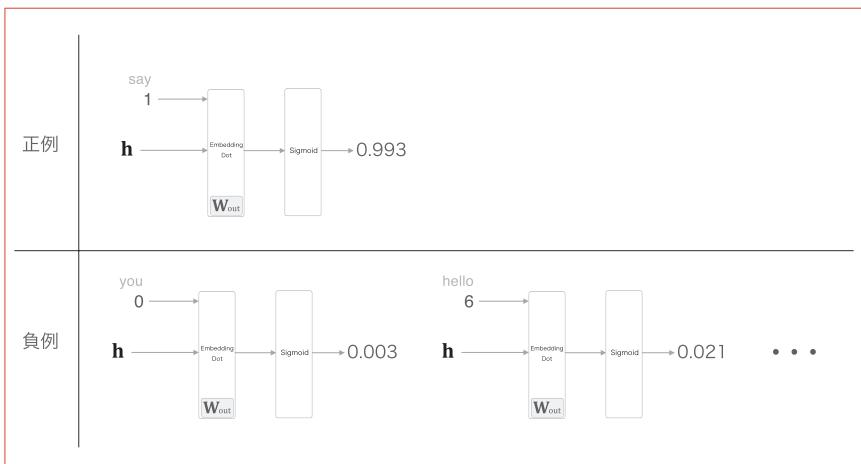


図 4-16 正例（正しい答え）を「say」とすると、「say」を入力したときの Sigmoid レイヤの出力は 1 に近く、「say」以外の単語を入力したときその出力は 0 に近くなる。そのような重みが求められる

たとえば、コンテキストが「you」と「goodbye」のとき、ターゲットが「hello」である確率——間違った単語の場合の確率——は低い値であることが望されます。図 4-16 では、ターゲットが「hello」のときの確率は 0.021 (2.1%) になっていますが、そのような 0 に近い値を出力するような重みが要求されるのです。



多値分類の問題を二値分類として扱うためには、「正しい答え（正例）」と「間違った答え（負例）」のそれぞれに対して、正しく（二値）分類できる必要があります。そのため、正例と負例の両者を対象として問題を考えなければなりません。

それでは、すべての負例を対象にして、二値分類の学習を行うのでしょうか？ 答えはもちろん「ノー」です。すべての負例を対象にしたのでは、語彙数が増えると手に負えなくなります（そもそも語彙数の増加に対応することが本章の目的でした）。そ

こで、近似解として負例をいくつか——5個とか、10個とか——ピックアップします（どのように選ぶかについては後述します）。つまり、ネガティブな例（負例）を少数サンプリングして用いるのです。これが「Negative sampling」という手法の意味するところです。

以上をまとめると、Negative samplingという手法は、正例をターゲットとした場合の損失を求めます。それと同時に、負例をいくつかサンプリングし（選び出し）、その負例に対しても同様に損失を求めます。そして、それぞれのデータ（正例とサンプリングされた負例）における損失を足し合わせ、その結果を最終的な損失とします。

ここまで話を、具体的な例を出して説明しましょう。ここでは、これまでと同じ例（正例のターゲットは「say」）を扱います。そして、負例のターゲットを2つサンプリングしたと仮定して、このときサンプリングされた単語が「hello」と「I」だとします。そうした場合、CBOW モデルの中間層以降だけに注目すると、Negative Sampling の計算グラフは図4-17のように書けます。

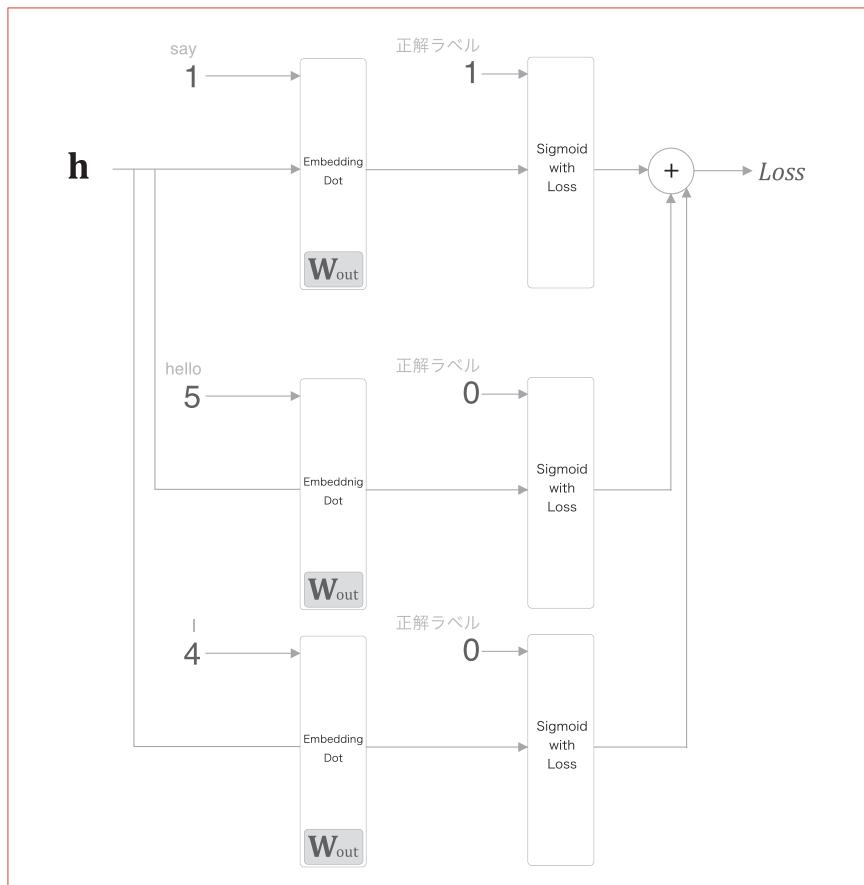


図 4-17 Negative sampling の例（中間層以降の処理に注目し、レイヤによる計算グラフを図示する）

図 4-17 で注意すべき点は、正例と負例の扱いについてです。正例（「say」）については、これまでどおり Sigmoid with Loss レイヤへ正解ラベルとして「1」を入力します。一方、負例（「hello」と「I」）については、それが間違った例であるため、Sigmoid with Loss レイヤへは正解ラベルとして「0」を入力します。後は、それぞれのデータにおける損失を足し合わせて最終的な損失を出力します。

4.2.6 Negative Sampling のサンプリング手法

Negative Sampling について説明すべきことは、残すところあとひとつになります。

した。それは、負例をどのようにサンプリングするかということです。これには、ランダムにサンプリングするよりも良い方法が知られています。それは、コーパスの統計データに基づいて、サンプリングを行うのです。具体的に言うと、コーパス中でよく使われる単語は抽出されやすくし、コーパス中であまり使われない単語は抽出されにくくします。

コーパス中の単語の使用頻度に基づいてサンプリングするには、コーパスから各単語の出現した回数を求め、これを「確率分布」で表します。そして、その確率分布から単語をサンプリングするのです（図4-18）。

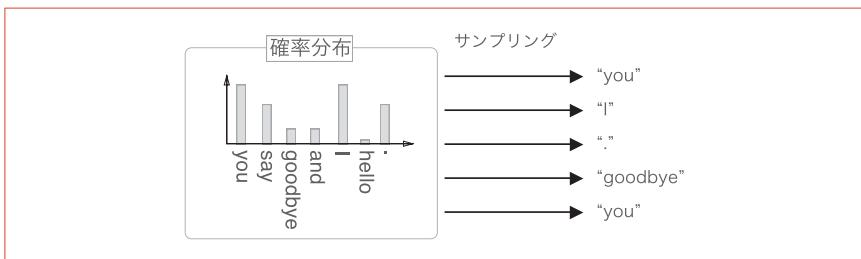


図4-18 確率分布に従ってサンプリングを複数回行う例

コーパス中の各単語の出現回数を元に確率分布を求めれば、後はその確率分布に従ってサンプリングを行なうだけです。確率分布に基づいてサンプリングすることで、コーパス中で多く登場した単語は抽出されやすくなります。一方、“レアな単語”は、抽出されにくくなるのです。



Negative Sampling では、できることならば、負例として多くの単語をバーすることが望されます。しかし計算量の問題から、負例を少数（5個や10個など）に限定する必要があります。ここで負例として、“レアな単語”ばかりが選ばれたとしたらどうでしょうか？ それでは悪い結果になるでしょう。なぜなら、現実的な問題においても、レアな単語はほとんど出現しないからです。つまり、レアな単語を相手にする重要性は低いということです。それよりも、高頻出な単語に対応できたほうが、良い結果につながるでしょう。

それでは、確率分布に従ってサンプリングする例を Python を使って説明します。この用途には、NumPy の `np.random.choice()` メソッドが使えます。ここでは、そのメソッドの使い方を示すため、いくつか使用例を示します。

```

>>> import numpy as np

# 0 から 9 の数字の中からひとつの数字をランダムにサンプリング
>>> np.random.choice(10)
7
>>> np.random.choice(10)
2

# words からひとつだけランダムにサンプリング
>>> words = ['you', 'say', 'goodbye', 'I', 'hello', '.']
>>> np.random.choice(words)
'goodbye'

# 5 つだけランダムサンプリング（重複あり）
>>> np.random.choice(words, size=5)
array(['goodbye', '.', 'hello', 'goodbye', 'say'],
      dtype='<U7')

# 5 つだけランダムサンプリング（重複なし）
>>> np.random.choice(words, size=5, replace=False)
array(['hello', '.', 'goodbye', 'I', 'you'],
      dtype='<U7')

# 確率分布に従ってサンプリング
>>> p = [0.5, 0.1, 0.05, 0.2, 0.05, 0.1]
>>> np.random.choice(words, p=p)
'you'

```

ここで示すように、`np.random.choice()` はランダムにサンプリングする用途に利用できます。このとき、引数に `size` を指定すれば、複数回まとめてサンプリングを行います。また、引数に `replace=False` を指定すれば重複をなくしてサンプリングします。そして、引数の `p` に確率分布を表すリストを指定することで、確率分布に従ってサンプリングが行われます。後は、この関数を使って負例をサンプリングするだけです。

さて、word2vec で提案された Negative sampling では、先の確率分布に対して、一手間加えることが提案されています。それは、式 (4.4) で示すように、元となる確率分布に対して 0.75 を累乗するのです。

$$P'(w_i) = \frac{P(w_i)^{0.75}}{\sum_j P(w_j)^{0.75}} \quad (4.4)$$

ここで $P(w_i)$ は、 i 番目の単語の確率を表します。式 (4.4) は単に、元の確率分布

の各要素を「0.75 乗」するだけです。ただし、変換後も確率の総和が 1 になるように、分母には「変換後の確率分布の総和」が必要になります。

それでは、なぜ式 (4.4) のような変換を行うのでしょうか？ それは、出現確率の低い単語を“見捨てない”ようにするためです。より正確に言うと、「0.75 乗」することによって、確率の低い単語に対してその確率を少しだけ高くすることができます。実際に例を示すと、次のようにになります。

```
>>> p = [0.7, 0.29, 0.01]
>>> new_p = np.power(p, 0.75)
>>> new_p /= np.sum(new_p)
>>> print(new_p)
[ 0.64196878  0.33150408  0.02652714]
```

この例が示すところによれば、変換前は 0.01 (1%) の確率だった要素が、変換後は 0.026... (2.6...%) になっています。このように、低確率の単語が（少しだけ）サンプリングされやすくなるための救済処置として、「0.75 乗」を行います。なお、0.75 という数値については理論的な意味ではなく、0.75 以外の値に設定することも可能です。

以上で見てきたように、Negative sampling はコーパスから単語の確率分布を作成して、それを「0.75 乗」し、先ほどの `np.random.choice()` を使って負例をサンプリングします。本書では、そのような処理を行うクラスを `UnigramSampler` という名前のクラスで提供します。ここでは、`UnigramSampler` の使い方だけを簡単に説明します。`UnigramSampler` クラスは `ch04/negative_sampling_layer.py` にあるので、実装に興味のある方は参照してください。



ユニグラム (Unigram) とは、「ひとつの（連続した）単語」を意味します。これと同じ要領で、バイグラムは「2 つの連続した単語」、トリグラムは「3 つの連続した単語」を意味します。ここで `UnigramSampler` という名前のクラスにしたのは、ひとつの単語を対象に確率分布を作るからです。もしこれが「バイグラム」となれば、('you', 'say')、('you', 'goodbye') ……のような 2 つの単語の組み合わせを対象にした確率分布を作ることになります。

`UnigramSampler` クラスは、初期化の際に 3 つの引数を取ります。それは、単語 ID のリストの `corpus`、確率分布に対する「累乗」の `power` (デフォルトは 0.75)、負例サンプリングを行う個数の `sample_size` です。また、`UnigramSampler` クラスは `get_negative_sample(target)` というメソッドを持ちます。これは、引数の `target` で指定したものを正例として、それ以外の単語 ID のサンプリングを行い

ます。それでは、`UnigramSampler` クラスを使う例を抜粋して次に示します。

```
corpus = np.array([0, 1, 2, 3, 4, 1, 2, 3])
power = 0.75
sample_size = 2

sampler = UnigramSampler(corpus, power, sample_size)
target = np.array([1, 3, 0])
negative_sample = sampler.get_negative_sample(target)
print(negative_sample)
# [[0 3]
# [1 2]
# [2 3]]
```

ここでは正例として `[1, 3, 0]` の 3 つのデータをミニバッチとして考えています。このとき、それぞれのデータに対して負例として 2 つサンプリングを行います。上の例では、ひとつ目のデータに対する負例は `[0 3]`、2 つ目は `[1 2]`、3 つ目は `[2 3]` となっていることが分かります。これで、負例のサンプリングが行えました。

4.2.7 Negative Sampling の実装

それでは最後に Negative Sampling の実装を行います。ここでは `NegativeSamplingLoss` というクラスで実装することにします。まずはイニシャライザから示します (☞ ch04/negative_sampling_layer.py)。

```
class NegativeSamplingLoss:
    def __init__(self, W, corpus, power=0.75, sample_size=5):
        self.sample_size = sample_size
        self.sampler = UnigramSampler(corpus, power, sample_size)
        self.loss_layers = [SigmoidWithLoss() for _ in range(sample_size
        ↵ + 1)]
        self.embed_dot_layers = [EmbeddingDot(W) for _ in
        ↵ range(sample_size + 1)]

        self.params, self.grads = [], []
        for layer in self.embed_dot_layers:
            self.params += layer.params
            self.grads += layer.grads
```

初期化の引数は、出力側の重みを表す `W`、コーパス（単語 ID のリスト）の `corpus`、確率分布の累乗の値 `power`、そして、負例のサンプリング数の `sample_size` を取ります。ここでは、前節で説明した `UnigramSampler` クラスを生成し、それをメンバ変数の `sampler` で保持します。また、負例のサンプリング数をメンバ変数の

`sample_size` に設定します。

メンバ変数の `loss_layers` と `embed_dot_layers` には、必要なレイヤをリストで保持します。このとき、その 2 つのリストには、`sample_size + 1` 個のレイヤを生成しますが、これは正例用のレイヤをひとつ、負例用のレイヤを `sample_size` 個だけ生成するためです。ここでは、リストの最初のレイヤが正例を扱うものとします。つまり、`loss_layers[0]` と `embed_dot_layers[0]` が、正例を扱うレイヤです。後は、このレイヤで使用するパラメータと勾配をそれぞれ配列にまとめます。続いて、順伝播の実装を示します（☞ ch04/negative_sampling_layer.py）。

```
def forward(self, h, target):
    batch_size = target.shape[0]
    negative_sample = self.sampler.get_negative_sample(target)

    # 正例のフォワード
    score = self.embed_dot_layers[0].forward(h, target)
    correct_label = np.ones(batch_size, dtype=np.int32)
    loss = self.loss_layers[0].forward(score, correct_label)

    # 負例のフォワード
    negative_label = np.zeros(batch_size, dtype=np.int32)
    for i in range(self.sample_size):
        negative_target = negative_sample[:, i]
        score = self.embed_dot_layers[1 + i].forward(h, negative_target)
        loss += self.loss_layers[1 + i].forward(score, negative_label)

    return loss
```

`forward(h, target)` メソッドが受け取る引数は、中間層のニューロン `h` と、正例のターゲットを表す `target` です。ここで行う処理は、まず初めに `self.sampler` で負例のサンプリングを行い、これを `negative_sample` とします。後は、正例と負例のそれぞれのデータに対して順伝播を行い、損失を加算していきます。具体的には、Embedding Dot レイヤの `forward` でスコアを出力し、続いて、そのスコアとラベルを Sigmoid with Loss レイヤに入れて損失を求めます。ここで、正例の場合は「1」を、負例の場合は「0」を正解ラベルとすることに注意しましょう。

それでは最後に、逆伝播の実装を示します。

```
def backward(self, dout=1):
    dh = 0
    for l0, l1 in zip(self.loss_layers, self.embed_dot_layers):
        dscore = l0.backward(dout)
        dh += l1.backward(dscore)
```

```
    return dh
```

逆伝播の実装は簡単です。これは、順伝播のときとは逆順に各レイヤの backward() を呼ぶだけになります。中間層のニューロンは順伝播の際に複数にコピーされました。これは「1.3.4.3 Repeat ノード」で説明した Repeat ノードに相当します。そのため、その逆伝播では、複数の勾配を加算することになります。以上が Negative Sampling の実装の説明です。

4.3 改良版 word2vec の学習

これまで word2vec の改良を行ってきました。最初に Embedding レイヤについて説明し、続いて Negative Sampling という手法について説明しました。そして、実際にその 2 つを実装してきました。それでは、それらの改良点を取り入れたニューラルネットワークを実装しましょう。そして、PTB データセットを使って学習し、より実用的な単語の分散表現を獲得したいと思います。

4.3.1 CBOW モデルの実装

ここでは、CBOW モデルを実装するにあたり、前章の単純な SimpleCBOW クラスを改良することにします。改良点は、Embedding レイヤと Negative Sampling Loss レイヤを使うことです。さらに、コンテキストとして、任意のウインドウサイズが扱えるように拡張します。

改良版である CBOW クラスの実装を次にまとめて示します。まずは、イニシャライザを示します (☞ ch04/cbow.py)。

```
import sys
sys.path.append('..')
import numpy as np
from common.layers import Embedding
from ch04.negative_sampling_layer import NegativeSamplingLoss

class CBOW:
    def __init__(self, vocab_size, hidden_size, window_size, corpus):
        V, H = vocab_size, hidden_size

        # 重みの初期化
        W_in = 0.01 * np.random.randn(V, H).astype('f')
```

```

W_out = 0.01 * np.random.randn(V, H).astype('f')

# レイヤの生成
self.in_layers = []
for i in range(2 * window_size):
    layer = Embedding(W_in) # Embeddingレイヤを使用
    self.in_layers.append(layer)
self.ns_loss = NegativeSamplingLoss(W_out, corpus, power=0.75,
→ sample_size=5)

# すべての重みと勾配を配列にまとめる
layers = self.in_layers + [self.ns_loss]
self.params, self.grads = [], []
for layer in layers:
    self.params += layer.params
    self.grads += layer.grads

# メンバ変数に単語の分散表現を設定
self.word_vecs = W_in

```

このイニシャライザでは 4 つの引数を受け取ります。`vocab_size` は語彙数、`hidden_size` は中間層のニューロン数、`corpus` は単語 ID のリストを表します。そして、コンテキストのサイズ——周囲の単語をどれだけコンテキストとして含めるか——を `window_size` で指定します。たとえば、`window_size` が 2 であれば、対象とする単語の左右の 2 単語（計 4 単語）がコンテキストになります。



`SimpleCBOW` クラス（改良前の実装）では、入力側の重みと出力側の重みの形状は異なっており、出力側の重みでは、列方向に単語ベクトルが配置されるようになっていました。一方 `CBOW` クラスの出力側の重みは、入力側の重みと同じ形状で、行方向に単語ベクトルが配置されています。これは、`NegativeSamplingLoss` クラス内で `Embedding` レイヤを使用しているためです。

重みの初期化が終わったら、続いてレイヤを作成します。ここでは、`Embedding` レイヤを `2 * window_size` 個作成し、それをメンバ変数の `in_layers` に配列で保持します。そして、`Negative Sampling Loss` レイヤを作成します。

レイヤの作成が終わったら、このニューラルネットワークで使用するすべてのパラメータと勾配をメンバ変数の `params` と `grads` にまとめます。また、後ほど単語の分散表現にアクセスできるように、メンバ変数の `word_vecs` に重み `W_in` を設定します。それでは続いて、順伝播の `forward()` メソッドと逆伝播の `backward()` メ

ソッドを示します (☞ ch04/cbow.py)。

```
def forward(self, contexts, target):
    h = 0
    for i, layer in enumerate(self.in_layers):
        h += layer.forward(contexts[:, i])
    h *= 1 / len(self.in_layers)
    loss = self.ns_loss.forward(h, target)
    return loss

def backward(self, dout=1):
    dout = self.ns_loss.backward(dout)
    dout *= 1 / len(self.in_layers)
    for layer in self.in_layers:
        layer.backward(dout)
    return None
```

ここでの実装は、各レイヤの順伝播（もしくは逆伝播）を適切な順番で呼ぶだけです。これは、前章の SimpleCBOW クラスの自然な拡張です。ただし、`forward(contexts, target)` メソッドでは、引数にコンテキストとターゲットを取りますが、それらは単語 ID から構成されます（前章では、それが単語 ID ではなく、one-hot ベクトルでした）。これは具体例を示すと、図4-19 のようになります。

| contexts | target | | contexts | target |
|--------------|---------|---|----------|--------|
| you, goodbye | say | | [0 2] | [1 |
| say, and | goodbye | | [1 3] | 2 |
| goodbye, I | and | → | [2 4] | 3 |
| and, say | I | | [3 1] | 4 |
| I, hello | say | | [4 5] | 1 |
| say, . | hello | | [1 6]] | 5] |

図4-19 コンテキストとターゲットを単語 ID で表した例：ここではコンテキストとしてウィンドウサイズが 1 の場合を示す

図4-19 の右側に示す単語 ID の配列が、`contexts` と `target` の例です。見て分かるとおり、`contexts` は 2 次元配列、`target` は 1 次元配列になります。そのようなデータが、`forward(contexts, target)` に入力されます。以上が、CBOW クラスの説明です。

4.3.2 CBOW モデルの学習コード

それでは最後に、CBOW モデルの学習を実装します。ここでは単に、ニューラルネットワークの学習を行うだけになります。早速コードを次に示します（[ch04/train.py](#)）。

```
import sys
sys.path.append('..')
import numpy as np
from common import config
# GPUで実行する場合は、下記のコメントアウトを消去（要cupy）
# =====
# config.GPU = True
# =====
import pickle
from common.trainer import Trainer
from common.optimizer import Adam
from cbow import CBOW
from common.util import create_contexts_target, to_cpu, to_gpu
from dataset import ptb

# ハイパーパラメータの設定
window_size = 5
hidden_size = 100
batch_size = 100
max_epoch = 10

# データの読み込み
corpus, word_to_id, id_to_word = ptb.load_data('train')
vocab_size = len(word_to_id)

contexts, target = create_contexts_target(corpus, window_size)
if config.GPU:
    contexts, target = to_gpu(contexts), to_gpu(target)

# モデルなどの生成
model = CBOW(vocab_size, hidden_size, window_size, corpus)
optimizer = Adam()
trainer = Trainer(model, optimizer)

# 学習開始
trainer.fit(contexts, target, max_epoch, batch_size)
trainer.plot()

# 後ほど利用できるように、必要なデータを保存
word_vecs = model.word_vecs
```

```

if config.GPU:
    word_vecs = to_cpu(word_vecs)
params = {}
params['word_vecs'] = word_vecs.astype(np.float16)
params['word_to_id'] = word_to_id
params['id_to_word'] = id_to_word
pkl_file = 'cbow_params.pkl'
with open(pkl_file, 'wb') as f:
    pickle.dump(params, f, -1)

```

今回のCBOWモデルは、ウィンドウサイズを5、隠れ層のニューロンの数を100に設定しています。対象とするコーパスにもよりますが、ウィンドウサイズは2～10、中間層のニューロン数（単語の分散表現の次元数）は50～500ぐらいが良い結果になるようです。このようなハイパーパラメータに対する議論は後ほど行います。

さて、今回扱うPTBコーパスは、これまでよりもサイズが格段に大きくなっています。そのため、学習には多くの時間（半日程度）が必要になります。ここではオプションとして、GPUを使って実行できるようなモードを用意しています。GPUで実行するには、ファイルの先頭にある「# config.GPU = True」を有効にします。ただし、GPUで実行するには、NVIDIAのGPUを備えていてCuPyがインストールされているマシンが必要です。

学習が終われば、重みを取り出し（ここでは入力側の重みだけ）、後ほど利用できるようにファイルに保存します（単語と単語IDの変換用のディクショナリも一緒にファイルへと保存します）。ここではファイルに保存するために、Pythonの「ピクル（pickle）」という機能を利用します。ピクルは、Pythonコード中のオブジェクトをファイルへ保存（もしくはファイルから読み込み）するために利用できます。



学習済みのパラメータは、ch04/cbow_params.pklに用意してあります。学習が終わるのを待てない場合は、本書提供の学習済みパラメータを利用してください。なお、学習した重みデータは、各自の学習した環境によって異なります。これは重みの初期化に用いるランダムな初期値、またミニバッチを無作為に選ぶこと、そしてNegative samplingにおけるサンプリングのランダム性が原因です。それらのランダム性により、最終的に得られる重みは各自の環境で異なりますが、大きな視点で見ると、同じような結果（傾向）が得られるでしょう。

4.3.3 CBOW モデルの評価

それでは、前節で学習した単語の分散表現を評価したいと思います。ここでは、2章で実装した `most_similar()` メソッドを利用し、いくつかの単語に対して最も距離の近い単語を表示してみたいと思います (☞ `ch04/eval.py`)。

```
import sys
sys.path.append('..')
from common.util import most_similar
import pickle

pkl_file = 'cbow_params.pkl'

with open(pkl_file, 'rb') as f:
    params = pickle.load(f)
    word_vecs = params['word_vecs']
    word_to_id = params['word_to_id']
    id_to_word = params['id_to_word']

querys = ['you', 'year', 'car', 'toyota']
for query in querys:
    most_similar(query, word_to_id, id_to_word, word_vecs, top=5)
```

上のコードを実行すると、次の結果が得られます（ここで得られた結果は、各自の学習した環境によって多少異なります）。

```
[query] you
we: 0.610597074032
someone: 0.591710150242
i: 0.554366409779
something: 0.490028560162
anyone: 0.473472118378

[query] year
month: 0.718261063099
week: 0.652263045311
spring: 0.62699586153
summer: 0.625829637051
decade: 0.603022158146

[query] car
luxury: 0.497202396393
arabia: 0.478033810854
auto: 0.471043765545
disk-drive: 0.450782179832
travel: 0.40902107954
```

```
[query] toyota
ford: 0.550541639328
instrumentation: 0.510020911694
mazda: 0.49361255765
bethlehem: 0.474817842245
nissan: 0.474622786045
```

結果を見てみましょう。まずは「you」をクエリとした場合、類似単語には人称代名詞の「i (= I)」や「we」などが来ています。続く「year」をクエリとした場合は、「month」や「week」などの期間を表す同じ性質の単語が見られます。そして「toyota」をクエリとした場合は、「ford」や「mazda」、「nissan」といった自動車メーカーが得られています。このような結果を見ると、CBOW モデルで獲得された単語の分散表現は、良い性質を持っていると言えそうです。

さらに word2vec で得られた単語の分散表現は、類似単語を近くに集めるだけではなく、より複雑なパターンを捉えることが分かっています。その代表例が、「king – man + woman = queen」で有名な類推問題（アナロジー問題）です。これはより正確に言うと、word2vec の単語の分散表現を使えば、類推問題をベクトルの加算と減算で解くことができるということです。

実際に類推問題を解くには、図 4-20 に示すように、単語ベクトルの空間上において「man → woman」ベクトルと「king → ?」ベクトルができるだけ近くなるような単語を探します。

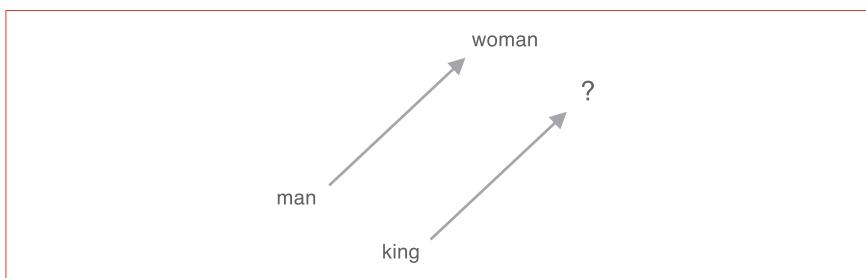


図 4-20 「man : woman = king : ?」という類推問題を解くにあたって、単語ベクトルの空間上での各単語の関係性を示す

ここで、単語「man」の分散表現(単語ベクトル)を「 $\text{vec}(\text{'man'})$ 」で表すとします。そうすると、図 4-20 で求めたい関係性を数式で表せば「 $\text{vec}(\text{'woman'}) - \text{vec}(\text{'man'}) = \text{vec}(?) - \text{vec}(\text{'king'})$ 」となり、これを変形すると「 $\text{vec}(\text{'king'}) + \text{vec}(\text{'woman'}) -$

`vec('man') = vec(?)`となります。つまり私たちが解くべき問題は、「`vec('king') + vec('woman') - vec('man')`」というベクトルに一番距離の近い単語ベクトルを求めることがあります。本書では、そのようなロジックを実装した関数を `common/util.py` に `analogy()` として提供します。この関数を使えば、先のような類推問題は `analogy('man', 'king', 'woman', word_to_id, id_to_word, word_vecs, top=5)` のように 1 行で答えさせることができます。そしてそのとき、次のような結果がターミナルに出力されます。

```
[analogy] man:king = woman:?
word1: 5.003233
word2: 4.400302
word3: 4.22342
word4: 4.003234
word5: 3.934550
```

上のように、1 行目に問題文が表示され、スコアが高い順に 5 つの単語が出力されます。このとき、各単語の隣にはスコアが表示されます。それでは、実際に類推問題をいくつか解かせてみましょう。ここでは次の 4 つの問題を出題してみます（☞ `ch04/eval.py`）。

```
analogy('king', 'man', 'queen', word_to_id, id_to_word, word_vecs)
analogy('take', 'took', 'go', word_to_id, id_to_word, word_vecs)
analogy('car', 'cars', 'child', word_to_id, id_to_word, word_vecs)
analogy('good', 'better', 'bad', word_to_id, id_to_word, word_vecs)
```

そして、これを実行すると、次のような結果が得られます。

```
[analogy] king:man = queen:?
woman: 5.161407947540283
veto: 4.928170680999756
ounce: 4.689689636230469
earthquake: 4.633471488952637
successor: 4.6089653968811035

[analogy] take:took = go:?
went: 4.548568248748779
points: 4.248863220214844
began: 4.090967178344727
comes: 3.9805688858032227
oct.: 3.9044761657714844

[analogy] car:cars = child:?
children: 5.217921257019043
```

```

average: 4.725458145141602
yield: 4.208011627197266
cattle: 4.18687629699707
priced: 4.178797245025635

[analogy] good:better = bad:?
more: 6.647829532623291
less: 6.063825607299805
rather: 5.220577716827393
slower: 4.733833312988281
greater: 4.672840118408203

```

結果は私たちの期待どおりです！ひとつ目の問題は「king : man = queen : ?」ですが、これには正しく「woman」と答えています。続いて2題目は「take : took = go : ?」という問題で、これにも期待どおりに「went」と答えています。これは現在形と過去形のパターンを捉えている証拠で、時制に関する情報が単語の分散表現にエンコードされていると解釈できます。そして、3題目では単語の单数形と複数形の組み合わせを正しく捉えていることが分かります。残念ながら、4題目の「good : better = bad : ?」には「worse」と答えられませんでした。しかし、「more」や「less」などの比較級の単語を答えていることを見ると、そのような性質も単語の分散表現にエンコードされていることがうかがえます。

このように、word2vec で得た単語の分散表現を使えば、ベクトルの加減算で類推問題が解けます。さらに単語の意味だけではなく、文法的な情報についても、そのパターンを捉えているのです。この他にも、「good」と「best」の間には「better」が存在するといったような関係性など、word2vec の単語の分散表現には興味深い結果がいくつも見つかっています。



ここでの類推問題の結果はとても良いものに感じられるでしょう。しかし残念ながら、これはうまく解ける問題を筆者のほうで選別したにすぎません。実際には、多くの問題において期待する結果が得られないでしょう。これは PTB データセットが小規模であることが主な原因です。大きなコーパスを対象として学習すれば、より精度が高く、よりロバストな単語の分散表現を得られ、類推問題の正解率も大きく向上するでしょう。

4.4 word2vec に関する残りのテーマ

word2vec の仕組みや実装については、ほとんど説明は終わりました。本節では

word2vecについて、これまで扱いきれなかったテーマを紹介したいと思います。

4.4.1 word2vecを使ったアプリケーションの例

word2vecで得られた単語の分散表現は、類似単語を求める用途に利用できます。しかし、単語の分散表現の利点はそれだけではありません。自然言語処理の分野において、単語の分散表現が重要な理由は**転移学習**（transfer learning）にあります。これは、ある分野で学んだ知識を別の分野にも適用できるということです。

自然言語のタスクを解く場合、word2vecによる単語の分散表現をゼロから学習するようなことはほとんど行いません。そうではなく、先に大きなコーパス（WikipediaやGoogle Newsのテキストデータなど）で学習を行い、その学習済みの分散表現を個別のタスクで利用するのです。たとえば、テキスト分類や文書クラスタリング、品詞タグ付け、感情分析などの自然言語のタスクにおいて、単語をベクトルに変換する最初のステップでは、学習済みの単語の分散表現を利用することができます。そして、その多種多様な自然言語処理のタスクのほとんどすべてにおいて、単語の分散表現は素晴らしい結果をもたらすのです！

また、単語の分散表現の利点は、単語を固定長のベクトルに変換できることにあります。さらに、文章（単語の並び）に対しても、単語の分散表現を使って、固定長のベクトルに変換することができます。どのように文章を固定長のベクトルに変換するかは、盛んに研究されています。最も単純な方法としては、文章の各単語を分散表現に変換し、それらの総和を求めることが考えられます。これは bag-of-words と呼ばれる単語の順序を考慮しないモデル（考え方）です。また、5章で説明するリカレンティニューラルネットワーク（RNN）を使えば、さらに洗練された方法で、word2vecの単語の分散表現を利用しながら、文章を固定長のベクトルに変換することができます。

単語や文章を固定長のベクトルに変換できることはとても重要です。なぜなら、自然言語をベクトルに変換できれば、一般的な機械学習の手法（ニューラルネットワークやSVMなど）が適用できるからです。これは図で表すと、**図4-21** のようになります。

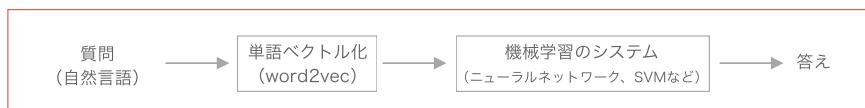


図4-21 単語の分散表現を用いたシステムの処理フロー

図4-21 に示すように、自然言語で書かれた質問を固定長のベクトルに変換することができれば、そのベクトルを別の機械学習システムの入力とすることができます。自然言語をベクトルに変換することによって、一般的な機械学習システムの枠組みで目的の答えを出力すること（そして、学習すること）が可能になるのです。



図4-21 のようなパイプラインにおいては、単語の分散表現の学習と機械学習システムの学習は別のデータセットを使い、個別に学習を行うのが一般的です。たとえば、単語の分散表現については、Wikipediaのような汎用的なコーパスを使って学習を先に済ませておきます。そして、現在直面している問題に対して集められたデータを対象に、機械学習システム（SVMなど）の学習を行います。ただし、現在直面する問題の学習データが大量に存在するのであれば、単語の分散表現と機械学習システムの学習を同時にゼロから行うことも考えられます。

それでは具体的な例を出しながら、単語の分散表現の使い方について見てきましょう。今あなたは、利用者が1億人を超えるスマートフォンアプリの開発・運営をしているとします。あなたの会社には、手に負えないほどのメールがユーザーから毎日届きます（または、Twitterなどで多くのつぶやきを見つけることができます）。好意的な意見がある一方で、中には、不満を持つユーザーの声も存在するでしょう。

そこであなたは、送られてくるメール（そして“つぶやき”など）を自動で分類できるシステムを作れないかと考えます。たとえば、**図4-22** のように、メールの内容からユーザーの感情を3段階に分類することはできないかと考えます。ユーザーの感情を正しく分類することができれば、不満を持つユーザーのメールから順に目を通すことができます。そうすれば、アプリの致命的な問題を発見し、早期に手を打つことができるかもしれません。それによって、ユーザーの満足度を上げることができるでしょう。



図4-22 メールの自動分類システムの例（感情分析）

メールの自動分類システムを作成するには、まずはデータ（メール）を収集するこ

とから始めます。ここでの例ではユーザーから送られてきたメールを集め、そのメールに対して人手でラベル付けを行います。たとえば、3段階の感情を表すラベル——positive / neutral / negative——を付与します。そのラベル付けの作業が終わったら、学習済みの word2vec を用いてメールをベクトルへ変換します。後は、感情分析を行う何らかの分類システム（SVM やニューラルネットワークなど）に対して、ベクトル化されたメールと感情ラベルを与えて学習させるのです。

この例のように、自然言語を扱う問題は、単語の分散表現によってベクトルに変換することができます。それによって、通常の機械学習の手法で解くことができるようになります。さらに、word2vec の転移学習の恩恵が得られます。つまり、自然言語の多くのタスクでは、word2vec の単語の分散表現を利用することで精度の向上が期待できるのです。

4.4.2 単語ベクトルの評価方法

word2vec によって単語の分散表現を得ることができました。それでは、その分散表現の良さはどのように評価すべきでしょうか？ここでは、単語の分散表現の評価方法について簡単に説明します。

単語の分散表現は、先ほどの感情分析の例のように、現実的には何らかのアプリケーションで使われることがほとんどです。その場合、私たちが最終的に望むものは、精度の良いシステムです。ここで考えなければならないのが、そのシステム（たとえば感情分析を行うシステム）は、複数のシステムから構成されるということです。複数のシステムとは、先の例で言うと、単語の分散表現を作るシステム（word2vec）と、特定の問題に対して分類を行うシステム（たとえば感情を分類する SVM など）です。

単語の分散表現の学習と分類を行うシステムの学習は別に行うかもしれません。その場合、たとえば単語の分散表現の次元数が最終的な精度にどのように影響するかを調べるには、まず単語の分散表現の学習を行い、そしてその分散表現を使って、もうひとつの機械学習システムの学習を行わなければなりません。つまり、2段階の学習を行った上で評価する必要があるのです。さらにその場合、2つのシステムにおいて最適なハイパーパラメータのためのチューニングも必要になり、多くの時間がかかりてしまいます。

そこで、単語の分散表現の良さを評価するにあたり、現実的なアプリケーションとは切り離して評価を行うというのが一般的によく行われます。その際によく用いられる評価指標が単語の「類似性」や「類推問題」による評価です。

単語の類似性の評価では、人間が作成した単語類似度の評価セットを使って評価する事が多く行われます。たとえば、0 から 10 の間でスコア化するとして、「cat」と「animal」の類似度は 8、「cat」と「car」の類似度は 2 といったように、人が単語間の類似性を採点します。そして、人が出したスコアと word2vec によるコサイン類似度のスコアを比較して、その相関性を見るのです。

類推問題による評価は、「king : queen = man : ?」のような類推問題を出題し、その正解率でもって単語の分散表現の良さを測ります。たとえば、論文[27]では、類推問題による評価結果が掲載されています。ここでは、その結果から一部を抜粋して、図 4-23 に示します。

| モデル | 次元数 | コーパスのサイズ | Semantics | Syntax | Total |
|-----------|------|----------|-----------|--------|-------|
| CBOW | 300 | 16億 | 16.1 | 52.6 | 36.1 |
| skip-gram | 300 | 10億 | 61 | 61 | 61 |
| CBOW | 300 | 60億 | 63.6 | 67.4 | 65.7 |
| skip-gram | 300 | 60億 | 73.0 | 66.0 | 69.1 |
| CBOW | 1000 | 60億 | 57.3 | 68.9 | 63.7 |
| skip-gram | 1000 | 60億 | 66.1 | 65.1 | 65.6 |

図 4-23 類推問題による単語ベクトルの評価結果（表は論文[27]より一部抜粋して作成）

図 4-23 では、word2vec のモデル、単語の分散表現の次元数、そしてコーパスのサイズをパラメータとして、比較実験を行っています。それぞれの結果は、右側の 3 列です。図 4-23 の「Semantics」の列は、単語の意味を類推する類推問題への正解率を示します。これはたとえば、「king : queen = actor : actress」のような単語の意味を問う問題です。一方「Syntax」は、単語の形態情報を問う問題で、たとえば「bad : worst = good : best」のような問題に相当します。



図 4-23 の結果から、次のことが分かります。

- モデルによって精度が異なる（コーパスに応じて最適なモデルを選ぶ）
- コーパスが大きいほど良い結果になる（ビッグデータは常に望まれる）
- 単語ベクトルの次元数は適度な大きさが必要（大きすぎても精度が悪くなる）

類推問題によって、「単語の意味や文法的な問題を正しく理解しているか」ということを（ある程度）計測することができます。そのため、類推問題を精度良く解くことのできる単語の分散表現であれば、自然言語を扱うアプリケーションにおいても良い結果が期待できるでしょう。ただし、単語の分散表現の良さが、目的とするアプリケーションにどれだけ貢献するのか（もしくは貢献しないのか）ということは、アプリケーションの種類やコーパスの内容など、取り扱う問題の状況に応じて変化します。つまり、類推問題による評価が高いからといって、目的とするアプリケーションでも必ず良い結果になるということは保証できないのです。その点には注意が必要です。

4.5 まとめ

本章では、word2vec の高速化をテーマに、前章の CBOW モデルに対して改良を加えました。具体的には、Embedding レイヤを実装し Negative Sampling という新しい手法を導入しました。この背景には、コーパスの語彙数が増えるのに比例し、計算量が増加することが挙げられます。

本章での重要なテーマは、「すべて」ではなく「一部」を処理することです。結局のところ、人がすべてを知ることができないように、コンピュータも（現在の性能では）すべてのデータを処理することは現実的ではありません。それよりも、自分にとって大切な少数のことに限定して取り組むほうが実りが多いのです。本章ではその考えに基づく手法——Negative sampling——を詳しく見てきました。Negative sampling は、「すべて」の単語ではなく、「一部」の単語だけを対象にすることで、計算の効率化を達成しました。

前章と本章にて、word2vec をテーマとする一連の話は終わりです。word2vec は自然言語の分野に多大な影響を与えてきました。そこで得られた単語の分散表現は、さまざまな自然言語処理のタスクに利用されています。さらに word2vec の思想は、自然言語だけではなく他の分野——音声、画像、動画など——にも応用されています。本章で word2vec をしっかりと理解したのであれば、その知識はさまざまな分野で役に立つはずです。

本章で学んだこと

- Embedding レイヤは単語の分散表現を格納し、順伝播において該当する単語 ID のベクトルを抽出する
- word2vec では語彙数の増加に比例して計算量が増加するので、近似計算を行う高速な手法を使うとよい
- Negative Sampling は負例をいくつかサンプリングする手法であり、これを利用すれば多値分類を二値分類として扱うことができる
- word2vec によって得られた単語の分散表現は、単語の意味が埋め込まれたものであり、似たコンテキストで使われる単語は単語ベクトルの空間上で近い場所に位置するようになる
- word2vec の単語の分散表現は、類推問題をベクトルの加算と減算によって解ける性質を持つ
- word2vec は転移学習の点で特に重要であり、その単語の分散表現はさまざまな自然言語処理のタスクに利用できる

5章 リカレントニューラル ネットワーク (RNN)

何でも薄暗いじめじめした所で
ニヤーニヤー泣いていた事だけは記憶している。
——夏目漱石『吾輩は猫である』

これまで私たちが見てきたニューラルネットワークは、**フィードフォワード**と呼ばれるタイプのネットワークです。フィードフォワードとは流れが一方向のネットワークのことです。これは具体的に言うと、入力信号が次の層（隠れ層）へ信号を伝達し、信号を受け取った層はその次の層へ伝達し、そしてまた次の層へ……といったように一方向だけの信号伝達を行います。

フィードフォワード・ネットワークは単純な構成で、仕組みも理解しやすく、それでいて多くの問題に応用できます。しかし、このタイプのネットワークには大きな問題があります。その問題とは、時系列データをうまく扱えないことです。これはより正確に言うと、単純なフィードフォワード・ネットワークでは、時系列データの性質（パターン）を十分に学習することができないのです。そこで、**リカレントニューラルネットワーク** (Recurrent Neural Network)、略して **RNN** の出番です。

本章では、フィードフォワード・ネットワークの問題点を指摘し、RNN がその問題を見事に解決できることを説明します。また、RNN の構造をじっくりと時間をかけて解説しながら、その処理を Python で実装していきます。

5.1 確率と言語モデル

ここでは RNN の話を始める前の準備として、前章の word2vec の復習から始めま

す。また、自然言語に関する現象を「確率」を使って記述し、最後に、言語を確率として扱う「言語モデル」について説明します。

5.1.1 word2vec を確率の視点から眺める

それでは、前章の word2vec の CBOW モデルの復習から始めましょう。ここでは、 w_1, w_2, \dots, w_T という単語の列で表されるコーパスを考えます。そして、 t 番目の単語を「ターゲット」として、その前後の単語 ($t - 1$ 番目と $t + 1$ 番目) を「コンテキスト」として扱います。



本書では、ターゲットは「中央の単語」を、コンテキストはターゲットの「周囲の単語」を指す言葉として用います。

このとき CBOW モデルが行なうこととは、図 5-1 に示すように、コンテキストの w_{t-1} と w_{t+1} から、ターゲットの w_t を推測することです。

$$w_1 \ w_2 \ \cdots \ \underline{w_{t-1} \boxed{w_t} \ w_{t+1}} \ \cdots \ w_{T-1} \ w_T$$

図 5-1 word2vec の CBOW モデル：コンテキストの単語からターゲットとなる単語を推測する

それでは、 w_{t-1} と w_{t+1} が与えられたとき、ターゲットが w_t となる確率を数式で表してみましょう。これは式 (5.1) のように書くことができます。

$$P(w_t | w_{t-1}, w_{t+1}) \quad (5.1)$$

CBOW モデルは、式 (5.1) の事後確率をモデル化します。この事後確率は、「 w_{t-1} と w_{t+1} が与えられたとき、 w_t が起こる確率」を表します。これがウインドウサイズが 1 のときの CBOW モデルです。

ところで、私たちはこれまでコンテキストとして左右対称のウインドウを考えてきました。ここではコンテキストとして、左のウインドウだけに限定してみたいと思います。たとえば、図 5-2 のようなケースを考えてみます。

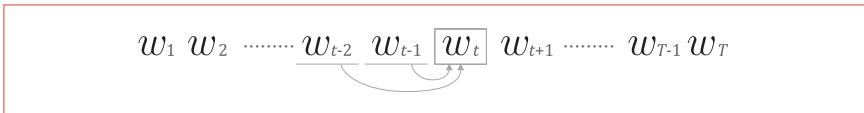


図5-2 コンテキストとして左側のウィンドウだけを対象とする

図5-2 のように、ここで私たちは左側の 2 つの単語だけをコンテキストとして考えることにします。そうすると、CBOW モデルが output する確率は式 (5.2) のようになります。

$$P(w_t|w_{t-2}, w_{t-1}) \quad (5.2)$$



word2vec のコンテキストのウィンドウサイズは、ハイパーパラメータであり、任意の値に設定できます。ここでは、ウィンドウサイズを「左を 2 単語、右を 0 単語」として左右非対称で設定しました。このように設定した理由は、後ほど説明する「言語モデル」を見越してのことです。

さて、式 (5.2) の表記を用いれば、CBOW モデルが扱う損失関数は、式 (5.3) のように書くことができます。ちなみに式 (5.3) は、交差エントロピー誤差により導かれた結果です（詳細は「1.3.1 損失関数」を参照）。

$$L = -\log P(w_t|w_{t-2}, w_{t-1}) \quad (5.3)$$

CBOW モデルの学習で行うこととは、式 (5.3) で表される損失関数——正確には、コーパス全体の損失関数の総和——をできる限り小さくする重みパラメータを見つけることです。もしそのような重みパラメータが見つかれば、CBOW モデルはコンテキストからターゲットをより正しく推測できるようになります。

このように CBOW モデルの学習の本来の目的は、コンテキストからターゲットを正しく推測できるようになることです。この目的を達成するために学習を進めると、（その副産物として）単語の意味がエンコードされた「単語の分散表現」が得られました。

それでは、CBOW モデルの本来の目的であった「コンテキストからターゲットを推測すること」は、何かに利用できるのでしょうか？ 式 (5.2) で表される確率 $P(w_t|w_{t-2}, w_{t-1})$ は、実用的な用途で利用できるのでしょうか？ そこに「言語モデル」が関係します。

5.1.2 言語モデル

言語モデル (Language Model) は、単語の並びに対して確率を与えます。単語の並びに対して、それがどれだけ起こりえるのか——それがどれだけ自然な単語の並びであるのか——ということを確率で評価するのです。たとえば、「you say goodbye」という単語の並びには高い確率（たとえば 0.092）を出力し、「you say good die」という単語の並びには低い確率（たとえば 0.000000000032）を出力する、というようなことを言語モデルは行います。

この言語モデルは、さまざまなアプリケーションで利用することができます。機械翻訳や音声認識などが、その代表例です。たとえば、ある音声認識システムの場合、人の発話からいくつかの文章を候補として生成するでしょう。その場合、言語モデルを使えば、候補となる文章が「文章として自然であるかどうか」という基準でランク付けすることができます。

また、言語モデルは新しい文章を生成する用途にも利用できます。というのも、言語モデルは単語列の自然さを確率的に評価できるため、その確率分布に従って単語を“紡ぎ出す”（サンプリングする）ことができるのです。なお、言語モデルを使った文章生成は「7章 RNNによる文章生成」で取り上げます。

それでは、数式を使って言語モデルを記述しましょう。ここでは、 w_1, \dots, w_m という m 個の単語からなる文章について考えます。このとき、 w_1, \dots, w_m という順序で単語が出現する確率は、 $P(w_1, \dots, w_m)$ で表されます。この確率は複数の事象が同時に起こる確率であるため、同時確率と呼ばれます。

この同時確率の $P(w_1, \dots, w_m)$ ですが、これは事後確率を使って次のように分解して書くことができます。

$$\begin{aligned} P(w_1, \dots, w_m) &= P(w_m|w_1, \dots, w_{m-1})P(w_{m-1}|w_1, \dots, w_{m-2}) \\ &\quad \dots P(w_3|w_1, w_2)P(w_2|w_1)P(w_1) \\ &= \prod_{t=1}^m P(w_t|w_1, \dots, w_{t-1})^{\dagger 1} \end{aligned} \tag{5.4}$$

式 (5.4) の \prod (パイ) という記号は、すべての要素をかけ合わせる「総乗」を表します（総和を表すのが \sum (シグマ) であるのに対して、総乗の記号は \prod で表されます）。式 (5.4) で示すとおり、同時確率は事後確率の総乗で表すことができます。

†1 数式の簡略化のため、ここでは $P(w_1|w_0)$ を $P(w_1)$ として扱います。

式 (5.4) の結果は、確率の**乗法定理**から導くことができます。ここでは、少し時間をとって乗法定理について説明し、式 (5.4) の導出する過程を見ていきたいと思います。早速、確率の乗法定理についてですが、これは次の式で表されます。

$$P(A, B) = P(A|B)P(B) \quad (5.5)$$

この式 (5.5) で表される乗法定理が、確率論において最も重要な定理です。この定理が意味することは、「 A と B の両方が起こる確率 $P(A, B)$ 」は、「 B が起こる確率 $P(B)$ 」と「 B が起きた後に A が起こる確率 $P(A|B)$ 」をかけ合わせたものになるということです（この解釈は自然なものに感じられるでしょう）。



$P(A, B)$ という確率は、 $P(A, B) = P(B|A)P(A)$ のように分解することも可能です。つまり、 A と B のどちらを事後確率の条件にするかで、 $P(A, B) = P(B|A)P(A)$ と $P(A, B) = P(A|B)P(B)$ の 2 つの表し方が存在します。

この乗法定理を使えば、 m 個の単語の同時確率 $P(w_1, \dots, w_m)$ を事後確率で表すことができます。このとき行う式変形を分かりやすく示すと、次のようにになります。

$$P(\underbrace{w_1, \dots, w_{m-1}}_A, w_m) = P(A, w_m) = P(w_m|A)P(A) \quad (5.6)$$

ここでは、 w_1, \dots, w_{m-1} をまとめて A という記号で表します。そうすると、乗法定理に従うことで式 (5.6) が導かれます。続いて、この A (w_1, \dots, w_{m-1}) に対して、再び同じ式変形を行います。

$$P(A) = P(\underbrace{w_1, \dots, w_{m-2}}_{A'}, w_{m-1}) = P(A', w_{m-1}) = P(w_{m-1}|A')P(A') \quad (5.7)$$

このように単語の並びをひとつずつ小さくしながら、そのつど事後確率に分解していきます。後は同様の手順を繰り返すことで、式 (5.4) を導くことができるのです。

さて、式 (5.4) が示すように、目的とする同時確率の $P(w_1, \dots, w_m)$ は、事後確率の総乗である $\prod P(w_t|w_1, \dots, w_{t-1})$ によって表すことができます。ここで注目すべきは、その事後確率は、対象の単語より左側のすべての単語をコンテキスト（条件）としたときの確率であるということです。これは図で表すと、図 5-3 のようにな

ります。

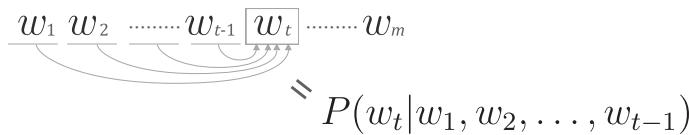


図5-3 言語モデルが扱う事後確率：\$t\$ 番目の単語をターゲットとして、\$t\$ 番目より左側の単語すべてをコンテキスト（条件）とする

ここまで話をまとめると、私たちの目標は $P(w_t|w_1, \dots, w_{t-1})$ という確率を得ることです。その確率が計算できれば、言語モデルの同時確率 $P(w_1, \dots, w_m)$ を求めることができます。



$P(w_t|w_1, \dots, w_{t-1})$ を表すモデルは、**条件付き言語モデル** (Conditional Language Model) と呼ばれます。また、 $P(w_t|w_1, \dots, w_{t-1})$ を表すモデルを指して、それを「言語モデル」と呼ぶ場合も多く見られます。

5.1.3 CBOW モデルを言語モデルに？

それでは、word2vec の CBOW モデルを、（無理やり）言語モデルに適用するにはどうしたらよいでしょうか？ それには、コンテキストのサイズをある値に限定することで、近似的に表すことができます。これは数式で表すと次のようになります。

$$P(w_1, \dots, w_m) = \prod_{t=1}^m P(w_t|w_1, \dots, w_{t-1}) \approx \prod_{t=1}^m P(w_t|w_{t-2}, w_{t-1}) \quad (5.8)$$

ここでは、コンテキストを左側の 2 つの単語に限定することにします。そうすれば、CBOW モデルによって——CBOW モデルの事後確率によって——、近似的に表すことができます。



機械学習や統計学の分野では「マルコフ性」（または「マルコフモデル」や「マルコフ連鎖」など）という言葉をよく耳にします。「マルコフ性」とは、未来の状態が現在の状態だけに依存して決まることを言います。また、ある事象の確率がその直前の N 個の事象だけに依存するとき、これを「N 階マルコフ連鎖」と言います。ここで示したのは直前の 2 つの単語だけに依存して次の単語が決まるモデルであるため、「2 階マルコフ連鎖」と呼ぶことができます。

式 (5.8) では、コンテキストとして 2 つの単語を用いる例を示しましたが、このコンテキストのサイズは任意の長さに設定できます（たとえば、5 や 10 など）。しかし、任意の長さに設定できるとはいえ、それはある長さに“固定”しなければなりません。たとえば、左側の 10 個の単語をコンテキストとして CBOW モデルを作るにしても、そのコンテキストよりもさらに左側にある単語の情報は無視されてしまします。これが問題になるのは、たとえば図5-4 のような例です。

Tom was watching TV in his room. Mary came into the room. Mary said hi to ?

図5-4 「？」に入る単語は何？：長いコンテキストが必要な問題の例

図5-4 の問題では、「Tom が部屋でテレビを見ていて、Mary がその部屋に入ってきた」とあります。その文脈（コンテキスト）を踏まえると、Mary が「Tom」（もしくは「彼」）にあいさつをするというのが正解になります。ここで正しい答えを得るには、例文の「？」から 18 個も前に登場する Tom を記憶しておく必要があるのです。もし CBOW モデルのコンテキストが 10 個までだったとしたら、この問題を正しく答えることはできないでしょう。

それでは、CBOW モデルのコンテキストのサイズを 20 や 30 といったように大きくすれば問題は解決するのでしょうか。確かに、CBOW モデルのコンテキストのサイズはいくらでも大きくすることはできます。しかし、CBOW モデルではコンテキスト内の単語の並びが無視されるという問題があります。



CBOW とは continuous bag-of-words の略です。bag-of-words とは「袋の中にある単語」を意味し、これは袋の中の単語の「並び」が無視されることを表しています。

コンテキストの単語の並びが無視される問題について、ひとつ具体例を出して説明しましょう。たとえば、コンテキストとして2個の単語を扱う場合、CBOWモデルではその2個の単語ベクトルの「和」が中間層に来ます。図で表せば、図5-5の左図のようになります。

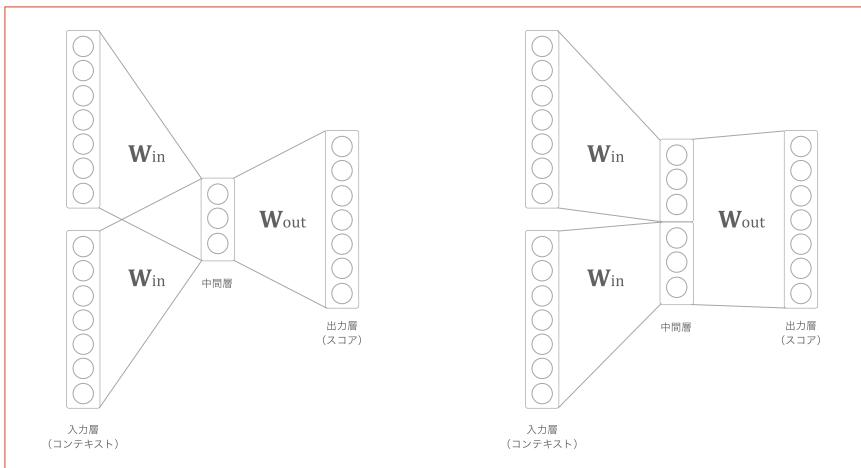


図5-5 左図が通常のCBOWモデル。右図は中間層において、各コンテキストの単語ベクトルを「連結」したモデルを示す（この図においては、入力層はone-hotベクトルであるものとして記載する）

図5-5の左図に示すように、CBOWモデルの中間層では単語ベクトルの和が求められるため、コンテキストの単語の並び方は無視されます。たとえば(you, say)と(say, you)というコンテキストが同じものとして扱われます。

本来であれば、コンテキストの単語の並びも考慮したモデルが望まれることでしょう。これを達成するためには、図5-5の右図に示すように、コンテキストの単語ベクトルを中間層において「連結(concatenate)」することが考えられます。実際、Neural Probabilistic Language Model[28]で提案されたモデルは、そのようなアプローチをとっています（モデルの詳細は論文[28]を参照）。しかし、連結するアプローチをとったとしたら、コンテキストのサイズに比例して重みパラメータが増えることになります。もちろん、そのようなパラメータの増加は歓迎されません。

それでは、ここで指摘した問題を解決するにはどうしたらよいでしょうか？お察しのとおり、ここで登場するのがリカレントニューラルネットワーク、略してRNN

です。RNN は、コンテキストがどれだけ長くても、そのコンテキストの情報を記憶するメカニズムを持ちます。そのため RNN を使えば、どんなに長い時系列データにも立ち向かうことができるのです！ それでは続いて、この素晴らしい RNN の世界をじっくり堪能していきたいと思います。



word2vec は、単語の分散表現を獲得することを目的に考案された手法です。そのため、それが言語モデルとして使用されることはありません。ここでは RNN の魅力を引き出すために、word2vec の CBOW モデルを無理やり言語モデルに適用するような話の展開にしました。ちなみに、word2vec は 2013 年、この後で見ていく RNN による言語モデルは 2010 年に Tomas Mikolov 氏らのチームによって、それぞれ提案されました。RNN による言語モデルでも単語の分散表現を獲得できたのですが、語彙数増加への対応や単語の分散表現の「質」の向上のために、word2vec が提案されたというのが、これらの研究の背景にあります。

5.2 RNN とは

RNN (Recurrent Neural Network) にある Recurrent とはラテン語から来た言葉です。それは「何度も繰り返し起こること」を意味します。日本語では、「再発する」「周期的に起こる」「循環する」といったように訳されます。そのため、RNN を直訳すると「再発するニューラルネットワーク」や「循環するニューラルネットワーク」ということになります。ここではまず初めに、「循環する」という言葉について少し考えてみたいと思います。



Recurrent Neural Network は、日本語では「再帰ニューラルネットワーク」や「循環ニューラルネットワーク」と訳されます（前者のほうが一般的です）。また、Recursive Neural Network（リカーシブニューラルネットワーク）という種類のネットワークもあります。これは主に木構造のデータを処理するためのネットワークで、リカレントニューラルネットワークとは別物です。

5.2.1 循環するニューラルネットワーク

早速ですが、「循環する」とはどういう意味でしょうか。もちろんそれは「繰り返し回り続ける」ことを意味します。ある地点をスタートしたものが、時間を経て再び元の場所へと戻ってくること、そして、それを繰り返すこと。それが「循環する」とい

う言葉の意味です。ここでひとつ注目したいのは、循環するためには「閉じた経路」が必要だということです。

「閉じた経路」もしくは「ループする経路」——そのような経路が存在することで初めて、媒体（もしくはデータ）は同じ場所を繰り返し行き来することができます。そして、データがループすることで、情報は絶えず更新されることになります。



血液は、私たちの体内を循環します。今流れている血液は、昨日の血液から継続して流れ続けてきました。そして、それは 1 週間前から、1 ヶ月前から、1 年前から、そして命を得たときから継続して流れてきた歴史を持ちます。血液は体内を循環することによって、過去から現在へと絶え間なく“更新”され続けるのです。

RNN の特徴は、ループする経路（閉じた経路）を持つことです。このループする経路によって、データは絶えず循環することができます。そしてデータが循環することにより、過去の情報を記憶しながら、最新のデータへと更新されます。

それでは、RNN について具体的に見ていきましょう。ここでは、RNN で用いられるレイヤを「RNN レイヤ」という名前で呼ぶことにします。この RNN レイヤは、図で表すと図 5-6 のように書くことができます。

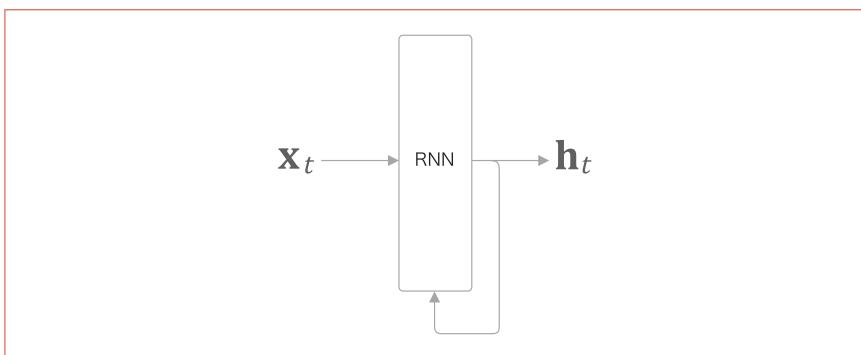


図 5-6 ループする経路を持つ RNN レイヤ

図 5-6 に示すとおり、RNN レイヤはループする経路を持ちます。このループする経路によって、データがレイヤ内を循環することができるのです。なお図 5-6 では、時刻を t として、 x_t を入力としています。これは時系列データとして、

$(\mathbf{x}_0, \mathbf{x}_1, \dots, \mathbf{x}_t, \dots)$ というデータがレイヤへ入力されることを暗に示しています。そして、その入力に対応する形で、 $(\mathbf{h}_0, \mathbf{h}_1, \dots, \mathbf{h}_t, \dots)$ が出力されます。

またここで、RNN レイヤへは各時刻に \mathbf{x}_t が入力されますが、 \mathbf{x}_t は何らかのベクトルを想定します。たとえば、文章（単語の並び）を扱う場合、各単語の分散表現（単語ベクトル）を \mathbf{x}_t として、それを RNN レイヤへ入力するようなケースがひとつ の例です。



図5-6 をよく見ると、その出力は 2 つに分岐していることが分かります。ここで言う「分岐」とは、同じものがコピーされて分岐することを意味します。そして、その分岐した出力のひとつが自分自身への入力となります。

続いて、図5-6 のループ構造について詳しく見ていきます。その前にここでは RNN レイヤの描画方法について、次のように変更を加えたいと思います。

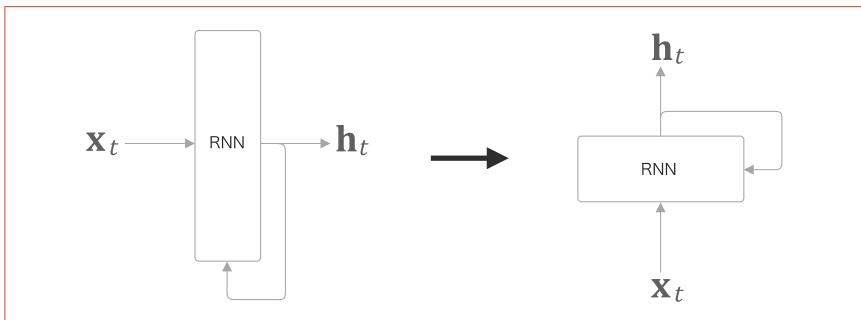


図5-7 レイヤを 90 度回転させて描画する

図5-7 に示すとおり、これまでデータが左から右へ流れることを想定してレイヤを描画してきました。しかしこれからは、紙面の都合上、データは下から上へ流れることを想定してレイヤの描画を行います（これは、この後で行うループを展開したときに、左右方向にレイヤを展開して描画するためです）。

5.2.2 ループの展開

これで準備は整いました。それでは、RNN レイヤのループ構造について詳しく見ていきましょう。RNN のループ構造は、今までのニューラルネットワークには存在しなかった構造です。しかし、このループを展開することによって、なじみのある

ニューラルネットワークへと“変身”させることができます。百聞は一見にしかず。実際にループを展開してみます（図5-8）。

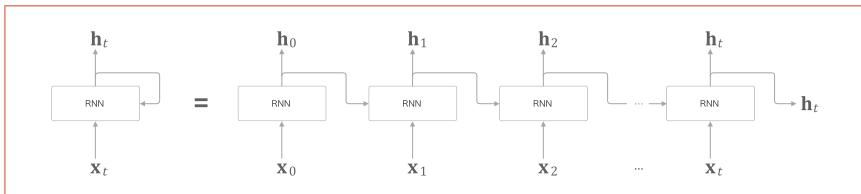


図5-8 RNN レイヤのループの展開

図5-8に示すように、RNN レイヤのループを展開することで、右方向に伸びる長いニューラルネットワークへと変身させることができました。これは、今まで見てきたフィードフォワード型ニューラルネットワークと同じ構造です（フィードフォワードとは、データが一方向だけに進みます）。ただし、図5-8にある複数のRNN レイヤは、すべて「同じレイヤ」であることが、これまでのニューラルネットワークとは異なります。



時系列データは、時間方向にデータが並びます。そのため、時系列データのインデックスを指すために「時刻」という言葉を使います（たとえば、時刻 t の入力データ x_t といったように）。自然言語の場合においても、「 t 番目の単語」や「 t 番目の RNN レイヤ」という表現を使う一方で、「時刻 t の単語」や「時刻 t の RNN レイヤ」のように表現する場合があります。

図5-8を見て分かるとおり、各時刻のRNN レイヤは、そのレイヤへの入力とひとつ前のRNN レイヤからの出力を受け取ります。そして、その2つの情報を元に、その時刻の出力が計算されます。このとき行う計算は、数式で次のように表されます。

$$\mathbf{h}_t = \tanh(\mathbf{h}_{t-1} \mathbf{W}_h + \mathbf{x}_t \mathbf{W}_x + \mathbf{b}) \quad (5.9)$$

まずは式(5.9)の記号について説明します。RNNでは重みが2つあります。ひとつは、入力 \mathbf{x} を出力 \mathbf{h} に変換するための重み \mathbf{W}_x 、もうひとつは、ひとつ前のRNNの出力を次時刻の出力に変換するための重み \mathbf{W}_h です。また、バイアスとして \mathbf{b} があります。なお、ここでは \mathbf{h}_{t-1} と \mathbf{x}_t は行ベクトルとします。

式(5.9)では、行列の積による計算を行い、それらの和を tanh 関数（双曲線正接

関数) によって変換します。その結果が時刻 t の出力 \mathbf{h}_t となります。この \mathbf{h}_t は、別のレイヤへ向けて上方へ出力されると同時に、次時刻の RNN レイヤ（自分自身）へ向けて右方向へも出力されます。

ところで式 (5.9) を見ると、現在の出力 (\mathbf{h}_t) は、ひとつ前の出力 (\mathbf{h}_{t-1}) によって計算されることが分かります。これは別の見方をすると、RNN は \mathbf{h} という「状態」を持っており、式 (5.9) の形で更新されると解釈できます。これが、RNN レイヤは「状態を持つレイヤ」や「メモリ（記憶力）を持つレイヤ」であると言われる所以です。



RNN の \mathbf{h} は「状態」を記憶し、時間が 1 ステップ（1 単位）進むに従い、式 (5.9) の形で更新されます。多くの文献では、RNN の出力 \mathbf{h}_t は、**隠れ状態** (hidden state) や**隠れ状態ベクトル** (hidden state vector) と呼ばれます。本書でも同様に、RNN の出力 \mathbf{h}_t を「隠れ状態」や「隠れ状態ベクトル」と呼ぶことにします。

なお、多くの文献では、展開後の RNN レイヤは図 5-9 の左図のように描かれます。

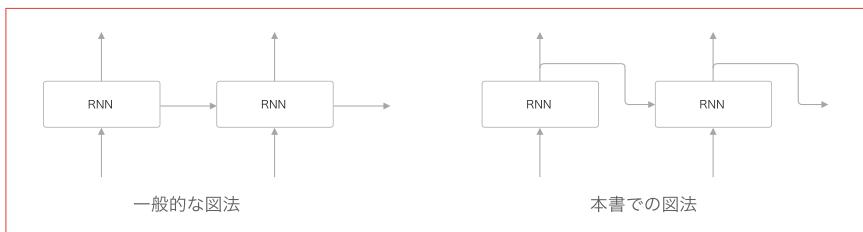


図 5-9 展開後の RNN レイヤの描画方法の比較

図 5-9 の左図では、RNN レイヤから 2 本の矢印が出力されていますが、その矢印は同じデータであること——正確には、同じデータがコピーして分岐されたものであることに注意が必要です。本書では（これまでどおり）図 5-9 の右図のように、出力が分岐していることを明示的に表すことにします。

5.2.3 Backpropagation Through Time

RNN レイヤを展開すると、横方向に伸びたニューラルネットワークとみなすことができました。そのため RNN の学習も、通常のニューラルネットワークと同じ手

順で学習することができます。これは図で表すと、図5-10のように書くことができます。

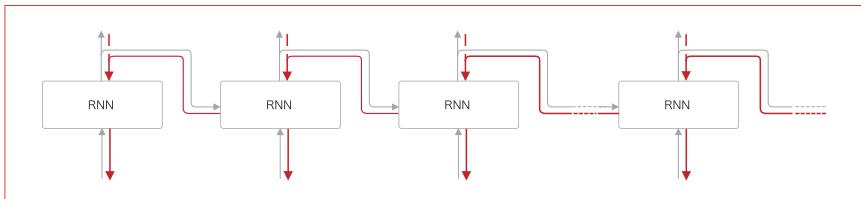


図5-10 ループ展開後の RNN レイヤに対する誤差逆伝播法

図5-10に示すとおり、ループを展開した後のRNNは（通常の）誤差逆伝播法を使うことができます。つまり、最初に順伝播を行い、続いて逆伝播を行うことで、目的とする勾配を求めるることができます。ここで誤差逆伝播法は、「時間方向に展開したニューラルネットワークの誤差逆伝播法」ということで、Backpropagation Through Timeと呼ばれます。もしくは、それを略してBPTTと呼ばれます。

このBPTTによって、RNNの学習は行えそうに見えます。しかし、その前にひとつ解決しなければならない問題があります。それは、長い時系列データを学習する場合の問題です。なぜそれが問題になるかというと、時系列データの時間サイズが大きくなるに比例して、BPTTで消費するコンピュータの計算リソースも増加することになるからです。また、時間サイズが長くなると、逆伝播時の勾配が不安定になることもあります。



BPTTによって勾配を求めるには、各時刻のRNNレイヤの中間データをメモリに保持しておかなければなりません（RNNレイヤの逆伝播は、後ほど説明します）。そのため時系列データが長くなるに従って、（計算量だけではなく）コンピュータのメモリ使用量も増加することになります。

5.2.4 Truncated BPTT

大きな時系列データを扱うときに通常よく行われるのが、ネットワークのつながりを適当な長さで“断ち切る”ことです。これは、時間軸方向に長くなりすぎたネットワークを適当な場所で切り取ることで、小さなネットワークを（複数）作るというアイデアです。そして、その切り取った小さなネットワークに対して、誤差逆伝播法を

行うのです。これが **Truncated BPTT** と呼ばれる手法です。



Truncated とは「切り取られた」という意味の単語です。Truncated BPTT は、適当な長さで「切り取った」誤差逆伝播法ということになります。

Truncated BPTT では、ネットワークのつながりを断ち切りますが、これは正しくは、ネットワークの「逆伝播」のつながりだけを断ち切ります。重要な点は、順伝播のつながりは維持されたままということです。つまり、順伝播の流れは途切れることなく伝播します。一方、逆伝播のつながりは適当な長さで切り取り、その切り取られたネットワーク単位で学習を行います。

それでは Truncated BPTT について、具体例を出して見ていきましょう。たとえばここに、1,000 個の長さのある時系列データがあるとします。自然言語の例で言うと、1,000 個の単語が並んだコーパスに相当します。ちなみに、私たちがこれまで扱ってきた PTB データセットでは、複数の文を連結したものをひとつの大きな時系列データとして扱ってきました。ここでも同様に、複数の文を連結したものをひとつの時系列データとして扱うことにします。

さて、1,000 個の長さのある時系列データを扱うとき、RNN レイヤを展開すると、それは横方向に 1,000 個のレイヤが並んだネットワークになります。もちろん、どれだけレイヤが並んだとしても、誤差逆伝播法によって勾配を計算することは可能です。しかしそれがあまりにも長いと、計算量やメモリの使用量などの点で問題になります。またレイヤが長くなるに従い勾配が徐々に小さくなることがあり、勾配が前時刻へと届かなくなります。そこで、図 5-11 で示すように、横方向に長く伸びたネットワークの逆伝播のつながりを適当な長さで断ち切ることを考えます。

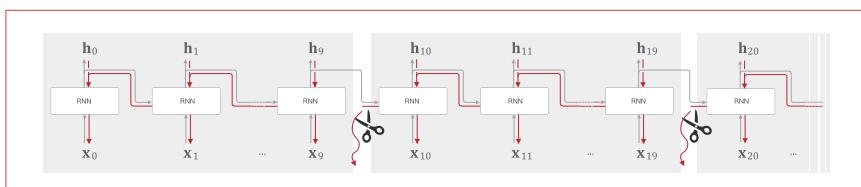


図 5-11 逆伝播のつながりを適当な場所で切断する。ここでは、逆伝播のつながりのある一連の RNN レイヤを「ブロック」と呼ぶことにし、ブロックの背景をグレーで描画する

図5-11では、RNNレイヤの長さが10個単位で学習できるように、逆伝播のつながりを切断しています。このように逆伝播のつながりを切ってしまえば、それより未来のデータについて考える必要がなくなります。そのため、各ブロック単位で——未来のブロックとは独立して——誤差逆伝播法を完結させることができます。

ここで注目してほしい点は、逆伝播のつながりは切断しますが、順伝播のつながりは切断しないということです。そのため、RNNの学習を行う際には、順伝播のつながりがあることを考慮しなければなりません。それが意味することは、データを順番に（“シーケンシャル”に）与える必要があるということです。データをシーケンシャルに与えるはどういうことか、この点に関して続けて具体的に説明します。



これまで私たちが見てきたニューラルネットワークでは、ミニバッチ学習を行うとき、データはランダムに選んで与えました。しかし、RNNにおいてTruncated BPTTを行う場合、データは“シーケンシャル”に与える必要があります。

それでは、Truncated BPTTによってRNNを学習させることを考えましょう。私たちがまず初めに行なうことは、ひとつ目のブロックの入力データ (x_0, \dots, x_9) をRNNレイヤに与えることです。そうすると、ここで行なう処理は図5-12のようになります。

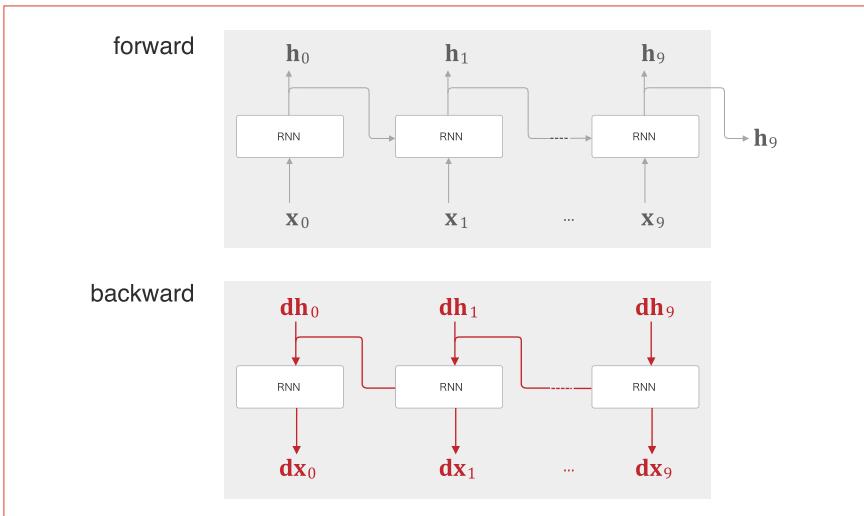


図5-12 ひとつ目のブロックの順伝播と逆伝播：これより先の時刻からの勾配は断ち切られているため、このブロック内だけで誤差逆伝播法が完結する

図5-12に示すように、初めに順伝播を行い、続けて逆伝播を行います。これにより、目的とする勾配を得ることができます。続いて、次のブロックの入力データ—— x_{10} から x_{19} ——を対象に誤差逆伝播法を行います。これは図で表すと図5-13のようになります。

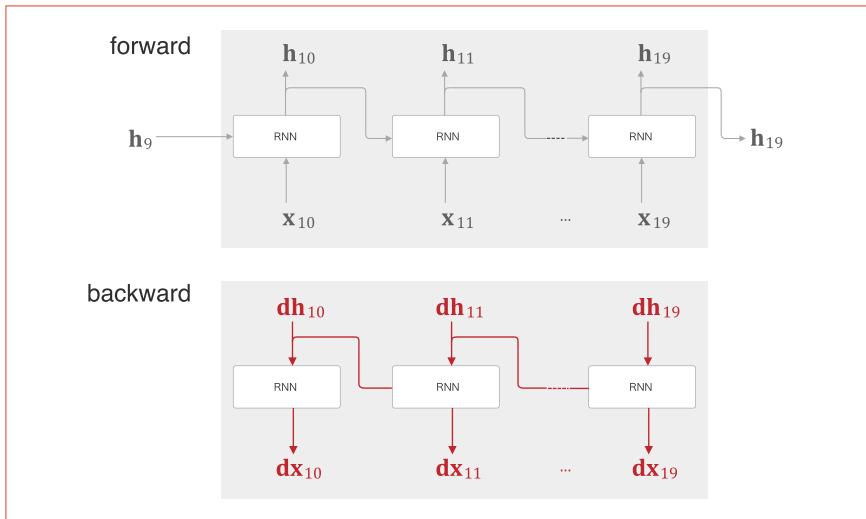


図5-13 2つ目のブロックの順伝播と逆伝播

ここでもひとつ目のブロックと同じように、順伝播を行い、続けて逆伝播を行います。そしてここで重要な点が、この順伝播の計算には前ブロックの最後の隠れ状態である h_9 が必要であるということです。これによって、順伝播のつながりは維持することができます。

同じ要領で、続いて3つ目のブロックを対象に学習を行います。このときも、2つ目のブロックの最後の隠れ状態 (h_{19}) を利用します。このように、RNN の学習ではデータをシーケンシャルに与えることで、隠れ状態を引き継ぎながら学習を行います。ここまで議論から、RNN の学習の流れは図5-14 のようになることが分かります。

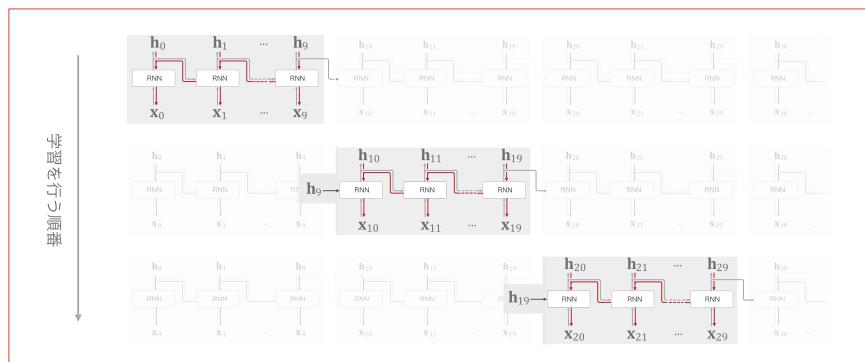


図 5-14 Truncated BPTT におけるデータの処理順

図 5-14 に示すように、Truncated BPTT ではデータをシーケンシャルに与えて学習を行います。これにより、順伝播のつながりを維持させながら、ブロック単位で誤差逆伝播法を適用できるのです。

5.2.5 Truncated BPTT のミニバッチ学習

ここまで Truncated BPTT に関する話は、ミニバッチ学習の「バッチ」については考えていませんでした。強いて言うならば、これまでの話はバッチ数が 1 のときに相当します。私たちはミニバッチ学習を行うため、本来ならばバッチを考慮して、図 5-14 のようにシーケンシャルにデータを与える必要があります。そのためには、データを与える開始位置を各バッチで“ズラす”必要があります。

“ズラす”という点を説明するため、ここでも前と同じく、1,000 個の長さの時系列データに対して、時間の長さを 10 個単位で切る Truncated BPTT で学習する場合を例にして説明します。それではこのとき、ミニバッチのバッチ数を 2 として学習するにはどうしたらよいでしょうか？その場合、RNN レイヤの入力データとして、ひとつ目のバッチ（サンプルデータ）には、先頭から順にデータを与えていきます。そして、2 つ目のバッチには、500 番目のデータを開始位置として、そこから順にデータを与えていくのです——つまり、開始位置を 500 だけ“ズラす”的な操作になります。これは図で表すと図 5-15 のようになります。

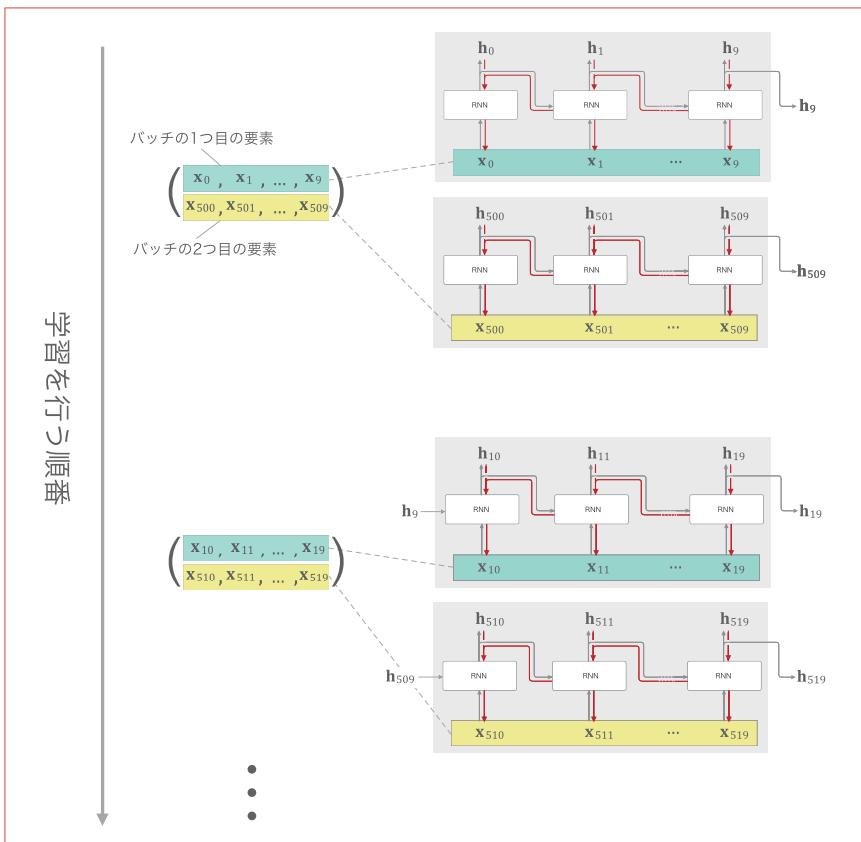


図5-15 ミニバッチ学習を行うとき、各バッチ（各サンプル）でデータを与える開始位置をズラす

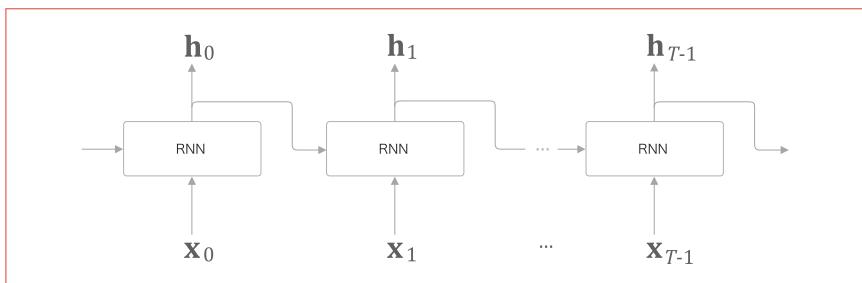
図5-15で示すように、ひとつ目のバッチの要素を x_0, \dots, x_9 として、2つ目のバッチの要素を x_{500}, \dots, x_{509} とします。そして、このミニバッチデータを RNN の入力データとして学習を行います。続いて与えるデータは、シーケンシャルに進めていくので、時系列データの 10~19 番目のデータと 510~519 番目のデータになります。このようにミニバッチ学習を行う場合は、各バッチの開始位置をオフセットとしてズラして、シーケンシャルに与えていきます。なお、シーケンシャルにデータを与えていく途中で終端に達した場合は、先頭に戻すような対応が必要になります。

以上見てきたように Truncated BPTT の原理は単純ですが、その際の「データの与え方」にはいくつか注意が必要です。具体的な注意点は「データをシーケンシャル

に与えること」、そして「各バッチでデータを与える開始位置をズラすこと」です。この辺の話はいくらか複雑なので、現時点では腑に落ちないかもしれません。ですが、後ほど実際のソースコードを見て動かすことで納得できると思います。

5.3 RNN の実装

ここまで話から、RNN の全貌が見えてきました。私たちがこれから実装すべきは、つまるところ、横方向に伸びたニューラルネットワークです。そしてそれは、Truncated BPTT による学習を考慮すると、横方向に固定サイズ分の一連のネットワークを作ればよいことになります。実際にこれは図で示すと、図5-16 のようになります。



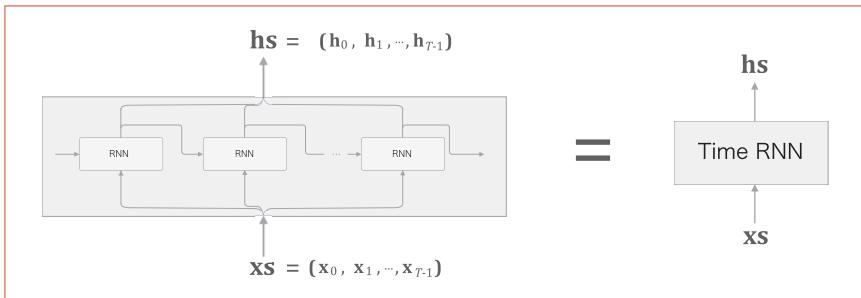


図5-17 Time RNN レイヤ：ループ展開後のレイヤをひとつのレイヤとみなす

図5-17に示すように、上下方向の入力と出力をそれぞれひとつに束ねれば、横並びのレイヤをひとつのレイヤとみなすことができます。つまり、 $(x_0, x_1, \dots, x_{T-1})$ を束ねた xs を入力すると、 $(h_0, h_1, \dots, h_{T-1})$ を束ねた hs を出力するレイヤであるとみなすことができるのです。ここで私たちは、Time RNN レイヤ内の1ステップの処理を行うレイヤを「RNN レイヤ」と呼び、 T ステップ分の処理をまとめて行うレイヤを「Time RNN レイヤ」と呼ぶことにします。



Time RNN のように、時系列データをまとめて処理するレイヤには、「Time」という単語を頭に付けることにします。これは本書独自の命名規則です。後ほど Time Affine レイヤや Time Embedding レイヤも実装しますが、それらも時系列データをまとめて処理します。

これから行う実装の流れは、まず初めに RNN の1ステップの処理を行うクラスを RNN クラスとして実装します。そして、その RNN クラスを利用して、 T ステップの処理をまとめて行うレイヤを TimeRNN クラスとして完成させます。

5.3.1 RNN レイヤの実装

それでは、RNN の1ステップの処理を行う RNN クラスの実装を行いましょう。復習になりますが、RNN の順伝播は、次の式 (5.10) で表されます（再掲）。

$$\mathbf{h}_t = \tanh(\mathbf{h}_{t-1} \mathbf{W}_h + \mathbf{x}_t \mathbf{W}_x + \mathbf{b}) \quad (5.10)$$

ここで私たちはミニバッチとしてデータをまとめて処理します。そのため、 \mathbf{x}_t （と \mathbf{h}_t ）には各サンプルデータを行方向に格納します。なお、行列の計算においては行列

の「形状チェック」が重要です。ここで計算では、バッチサイズが N で入力ベクトルの次元数が D 、隠れ状態ベクトルの次元数が H の場合、形状チェックは次のように書くことができます。

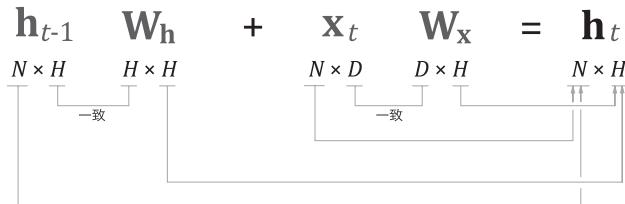


図5-18 形状チェック：行列の積では、対応する次元の要素数を一致させる（バイアスは省略）

図5-18に示すように、行列の形状チェックを行うことで、それが実装として正しいか——少なくとも計算として成り立っているか——を確認できます。それでは以上を踏まえて、RNN クラスの初期化と順伝播の `forward()` メソッドを示します（[common/time_layers.py](#)）。

```
class RNN:
    def __init__(self, Wx, Wh, b):
        self.params = [Wx, Wh, b]
        self.grads = [np.zeros_like(Wx), np.zeros_like(Wh),
        ← np.zeros_like(b)]
        self.cache = None

    def forward(self, x, h_prev):
        Wx, Wh, b = self.params
        t = np.dot(h_prev, Wh) + np.dot(x, Wx) + b
        h_next = np.tanh(t)

        self.cache = (x, h_prev, h_next)
        return h_next
```

RNN の初期化では、引数として重みを 2 つとバイアスをひとつ受け取ります。ここでは、引数で渡されたパラメータをメンバ変数の `params` にリストとして設定します。そして、各パラメータに対応する形で勾配を初期化し、それらを `grads` に格納します。最後に、逆伝播の計算時に使用する中間データを `cache` として、これを `None` で初期化します。

順伝播の `forward(x, h_prev)` メソッドでは、引数を 2 つ——下からの入力 x と、左からの入力 h_{prev} ——を受け取ります。後は、式 (5.10) に従って実装するだけです。ちなみにここでは、ひとつ前の RNN レイヤから受け取る入力を h_{prev} とし、現時刻での RNN レイヤの出力 (= 次時刻のレイヤへの入力) を h_{next} としています。

続いて、RNN の逆伝播の実装に進みましょう。その前に、RNN の順伝播を図 5-19 の計算グラフでもう一度確認したいと思います。

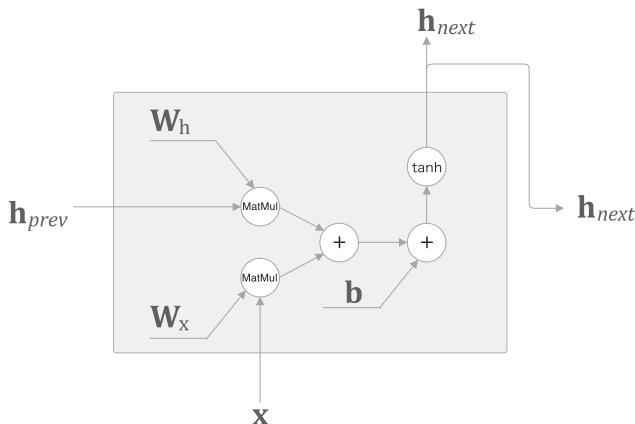


図 5-19 RNN レイヤの計算グラフ（`MatMul` ノードは行列の積を表す）

RNN レイヤの順伝播は、図 5-19 の計算グラフで表すことができます。ここで行う計算は、行列の積の「`MatMul`」と、足し算の「`+`」、そして「`tanh`」の 3 つの演算から構成されます。なお、バイアス b の足し算ではブロードキャストが発生するため、正確には `Repeat` ノードを用いますが、ここでは簡易性を優先して、その記載は省略することにします（「1.3.4.3 Repeat ノード」参照）。

では、図 5-19 の計算グラフの逆伝播はどのようになるでしょうか。その答えは簡単です。なぜなら、その 3 つの演算の逆伝播について、私たちはすでに習得しています（逆伝播のロジックについては 1.3 節を振り返ってください）。後は図 5-20 に従って、順伝播とは逆方向に各演算子の逆伝播を行うだけです。

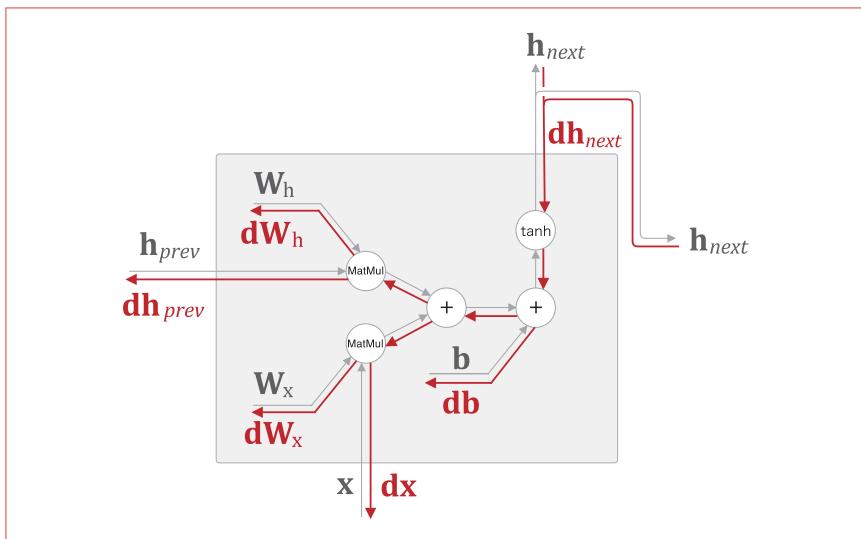


図5-20 RNN レイヤの計算グラフによる逆伝播

それでは、RNN レイヤの `backward()` の実装です。これは図5-20 を参考にすれば、次のように実装できます。

```
def backward(self, dh_next):
    Wx, Wh, b = self.params
    x, h_prev, h_next = self.cache

    dt = dh_next * (1 - h_next ** 2)
    db = np.sum(dt, axis=0)
    dWh = np.dot(h_prev.T, dt)
    dh_prev = np.dot(dt, Wh.T)
    dWx = np.dot(x.T, dt)
    dx = np.dot(dt, Wx.T)

    self.grads[0][...] = dWx
    self.grads[1][...] = dWh
    self.grads[2][...] = db

    return dx, dh_prev
```

以上が、RNN レイヤの逆伝播の実装です。それでは、続いて Time RNN レイヤの実装へと進みましょう。

5.3.2 Time RNN レイヤの実装

Time RNN レイヤは、 T 個の RNN レイヤから構成されます（この T は任意の数に設定できます）。繰り返しになりますが、これは図で表すと図5-21 のようになります。

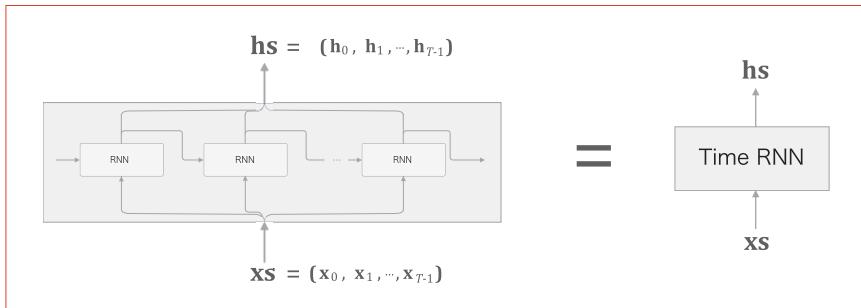


図5-21 Time RNN レイヤと RNN レイヤ

図5-21 に示すとおり、Time RNN レイヤは T 個の RNN レイヤを連結したネットワークです。このネットワークを、私たちは Time RNN レイヤとして実装します。そしてここでは、RNN レイヤの隠れ状態 h をメンバ変数に保持することにします。これは、図5-22 のように、隠れ状態の“引き継ぎ”を行う際に利用します。

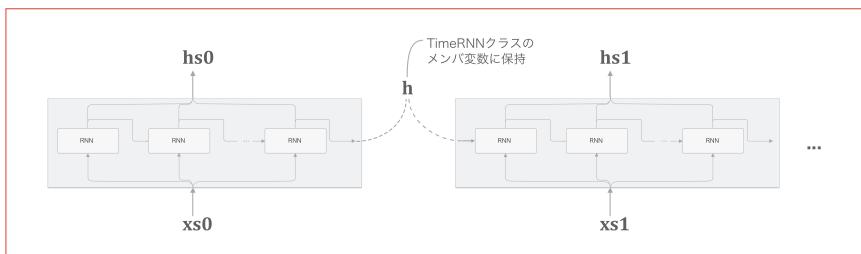


図5-22 Time RNN レイヤは、隠れ状態をメンバ変数 h として保持する。そうすることで、ブロック間での隠れ状態の引き継ぎが可能になる

図5-22 のように、私たちは RNN レイヤの隠れ状態を Time RNN レイヤに管理させることにします。そうすることで、Time RNN を使う人は、RNN レイヤの隠れ状態の“引き継ぎ作業”を考えなくてよくなります。そして私たちはこの機能を

——隠れ状態を引き継ぐかどうかを—— `stateful` という引数で調整できるようにします。

それでは、Time RNN レイヤの実装を示します。まずはイニシャライザとメソッドを 2 つ実装します (☞ `common/time_layers.py`)。

```
class TimeRNN:
    def __init__(self, Wx, Wh, b, stateful=False):
        self.params = [Wx, Wh, b]
        self.grads = [np.zeros_like(Wx), np.zeros_like(Wh),
        ← np.zeros_like(b)]
        self.layers = None

        self.h, self.dh = None, None
        self.stateful = stateful

    def set_state(self, h):
        self.h = h

    def reset_state(self):
        self.h = None
```

イニシャライザでは引数として、重みとバイアス、そして `stateful` というブーリアン (`True/False`) を受け取ります。メンバ変数については、まずは `layers` があります。これは複数の RNN レイヤをリストで保持するために使用します。また、メンバ変数の `h` は `forward()` メソッドを呼んだときの最後の RNN レイヤの隠れ状態を保持します。そして、メンバ変数の `dh` は、`backward()` を呼んだときに、ひとつの前ブロックへの隠れ状態の勾配を保持します（この `dh` については、逆伝播の実装で説明します）。



`TimeRNN` クラスは拡張性を考えて、Time RNN レイヤの隠れ状態を設定するメソッドを `set_state(h)` として実装します。また、隠れ状態をリセットするメソッドを `reset_state()` として実装します。

上の引数にある `stateful` とは「状態を持つ」という意味の単語です。本書の実装では、`stateful` が `True` のとき、Time RNN レイヤは「状態を持つ」ようにします。ここで言う「状態を持つ」とは、Time RNN レイヤの隠れ状態を維持することを意味します。つまり、どんなに長い時系列データであっても、Time RNN レイヤの順伝播を断ち切ることなく伝播させるということです。一方、`stateful` が `False` のときは、Time RNN レイヤの `forward()` が呼ばれるたびに最初の RNN レイヤの

隠れ状態を“ゼロ行列”（要素がすべて 0 の行列）で初期化します。これは状態は持たないモードであり、「ステートレス」と呼ばれます。



長い時系列データを処理するとき、RNN の隠れ状態を維持する必要があります。このような隠れ状態を維持する機能は「stateful」という言葉でよく表されます。多くのディープラーニングのフレームワークでは、RNN レイヤの引数に `stateful` と言うものがあり、前時刻の隠れ状態を保持するかどうか指定することができます。

それでは続いて、順伝播の実装です。

```
def forward(self, xs):
    Wx, Wh, b = self.params
    N, T, D = xs.shape
    D, H = Wx.shape

    self.layers = []
    hs = np.empty((N, T, H), dtype='f')

    if not self.stateful or self.h is None:
        self.h = np.zeros((N, H), dtype='f')

    for t in range(T):
        layer = RNN(*self.params)
        self.h = layer.forward(xs[:, t, :], self.h)
        hs[:, t, :] = self.h
        self.layers.append(layer)

    return hs
```

順伝播の `forward(xs)` メソッドでは、下側からの入力 `xs` を受け取ります。この `xs` は、`T` 個分の時系列データをひとつにまとめたものです。そのため、バッチサイズを `N` 個、入力ベクトルの次元数を `D` とすれば、`xs` の形状は (N, T, D) になります。

RNN レイヤの隠れ状態 `h` は、初回呼び出し時 (`self.h` が `None` のとき) には、要素がすべて 0 の行列で初期化します。また、メンバ変数の `stateful` が `False` の場合は常に `h` をゼロ行列でリセットします。

メインの実装では、最初に `hs = np.empty((N, T, H), dtype='f')` によって、出力用の“容器”を用意します。続いて `T` 回の繰り返し処理を行う `for` 文の中で RNN レイヤを生成し、それをメンバ変数の `layers` に追加します。そこでは RNN レイヤが各時刻の隠れ状態を計算し、それを `hs` の該当するインデックス（時刻）に

設定します。



Time RNN レイヤの `forward()` メソッドが呼ばれると、メンバ変数の `h` には最後の RNN レイヤの隠れ状態が設定されます。`stateful` が `True` の場合は、次に `forward()` メソッドが呼ばれるとき、そのメンバ変数の `h` が継続して利用されます。一方、`stateful` が `False` の場合は、メンバ変数の `h` がゼロ行列でリセットされます。

続いて、Time RNN レイヤの逆伝播の実装です。この逆伝播は、計算グラフで書くと図 5-23 のようになります。

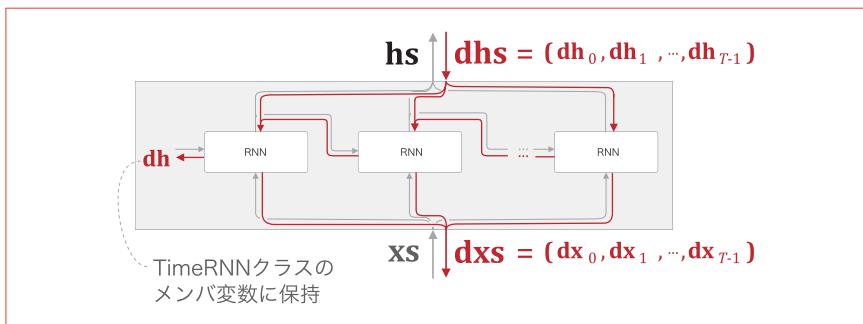
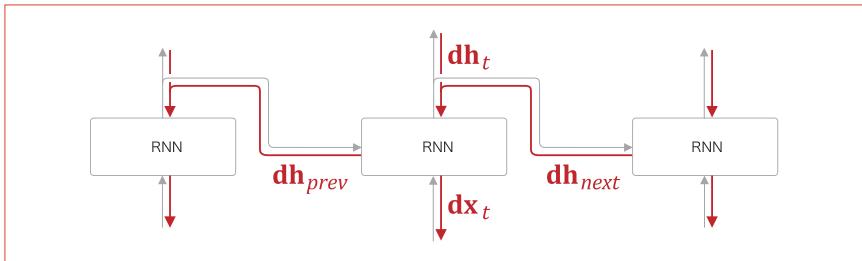


図 5-23 Time RNN レイヤの逆伝播

図 5-23 のとおり、ここでは上流（出力側の層）から伝わる勾配を `dhs` で表し、下流への勾配を `dxs` で表します。ここで私たちは Truncated BPTT を行うため、このブロックの前時刻の逆伝播は必要ありません。ただし、前時刻への隠れ状態の勾配についてはメンバ変数の `dh` に保持することにします。これは、7 章で扱う seq2seq（シーケンス・ツー・シーケンス）で必要になるためです（詳しくはその際に説明します）。

以上が Time RNN レイヤの逆伝播の全体図になります。このとき、 t 番目の RNN レイヤに着目すると、その逆伝播は図 5-24 のように書けます。

図5-24 t 番目の RNN レイヤの逆伝播

t 番目の RNN レイヤでは、上からの勾配 \mathbf{dh}_t と「ひとつ未来のレイヤ」からの勾配 \mathbf{dh}_{next} が伝わります。ここでの注意点は、RNN レイヤの順伝播では出力が 2 つに分岐しているということです。順伝播で分岐した場合、逆伝播では各勾配が合算されて伝わります。そのため、逆伝播時に RNN レイヤへは、合算された勾配—— $\mathbf{dh}_t + \mathbf{dh}_{next}$ ——が入力されます。以上の点に注意すれば、逆伝播の実装は次のようにになります。

```

def backward(self, dhs):
    Wx, Wh, b = self.params
    N, T, H = dhs.shape
    D, H = Wx.shape

    dxs = np.empty((N, T, D), dtype='f')
    dh = 0
    grads = [0, 0, 0]
    for t in reversed(range(T)):
        layer = self.layers[t]
        dx, dh = layer.backward(dhs[:, t, :] + dh) # 合算した勾配
        dxs[:, t, :] = dx

        for i, grad in enumerate(layer.grads):
            grads[i] += grad

    for i, grad in enumerate(grads):
        self.grads[i][...] = grad
    self.dh = dh

    return dxs

```

ここでも初めに、下流へ流す勾配の“容器”(`dxs`)を作ります。そして、順伝播とは逆の順番に RNN レイヤの `backward()` メソッドを呼び出し、各時刻の勾配 `dx` を求め、`dxs` の該当するインデックスに代入します。また重みパラメータについて

も、各 RNN レイヤでの重みの勾配を加算していき、最終的な結果をメンバ変数の `self.grads` に「... (3点ドット)」を使って上書きします。



Time RNN レイヤの中には、RNN レイヤが複数あります。そして、それらの RNN レイヤでは、同じ重みを使用しています。そのため、Time RNN レイヤの（最終的な）重みの勾配は、各 RNN レイヤの重みの勾配を足し合わせたものになります。

以上が Time RNN レイヤの実装の説明です。

5.4 時系列データを扱うレイヤの実装

本章での私たちの目標は、RNN を使って「言語モデル」を実装することです。私たちはこれまでに、RNN レイヤ——そして、時系列データをまとめて処理する Time RNN レイヤ——を実装してきました。ここでは、時系列データを扱うレイヤを新たにいくつか作りたいと思います。なお RNN による言語モデルは、RNN Language Model であることから **RNNLM** と呼ぶことにします。それでは、RNNLM の完成を目指して先へ進みましょう。

5.4.1 RNNLM の全体図

まず初めに RNNLM で使われるネットワークを実際に見てていきます。ここでは、最もシンプルな RNNLM のネットワーク図を図 5-25 に示します。図 5-25 の左図には RNNLM のレイヤ構成を示し、その右図に時間軸に展開したネットワークを示します。

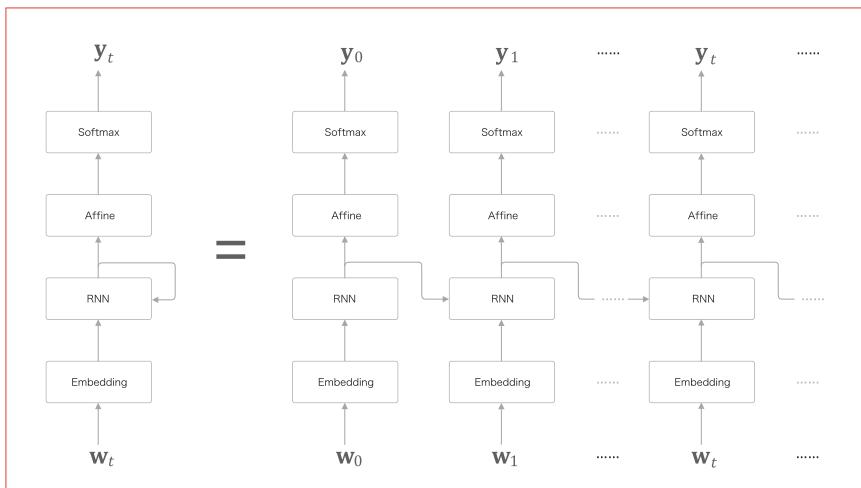


図 5-25 RNNLM のネットワーク図（左図が展開前、右図が展開後）

図 5-25 の最初の層は Embedding レイヤです。このレイヤは、単語 ID を単語の分散表現（単語ベクトル）へと変換します。そしてその分散表現が、RNN レイヤへと入力されます。RNN レイヤは隠れ状態を次の層へ（上方向に）出力すると同時に、次時刻の RNN レイヤへ（右方向に）出力します。そして RNN レイヤが上方向に出力した隠れ状態は、Affine レイヤを経て Softmax レイヤへと伝わっていきます。

それでは図 5-25 のニューラルネットワークに対して、順伝播だけを考えて、具体的なデータを流してみることにします。そしてそのときの出力結果を観察してみましょう。なおここで扱う文章は、おなじみの「You say goodbye and I say hello.」とします。このとき、RNNLM が行う処理は図 5-26 のようになります。

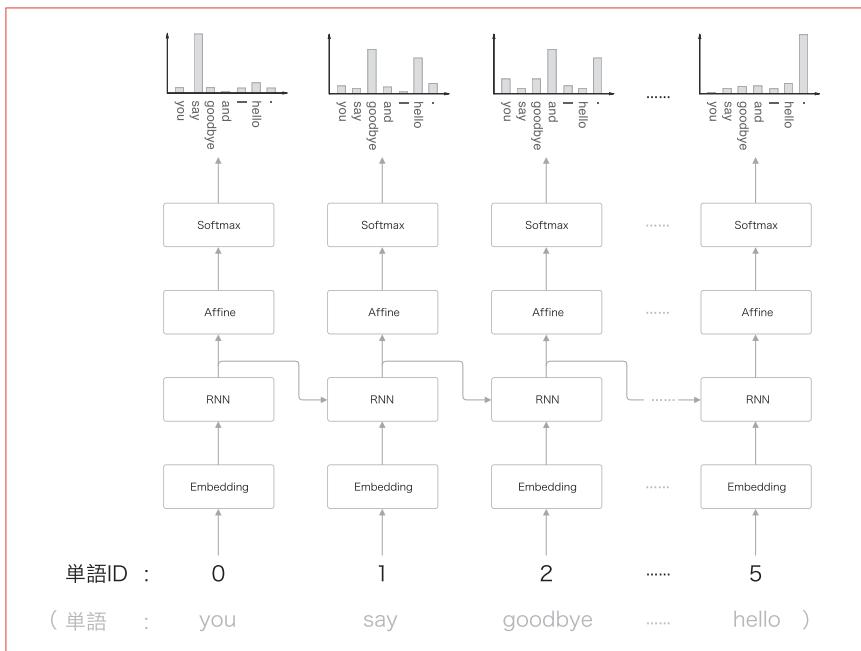


図5-26 サンプルのコーパスとして「you say goodbye and I say hello .」を処理する RNNLM の例

図5-26に示すとおり、入力に用いられるデータは単語IDの配列です。まずは最初の時刻に注目してみましょう。ここでは先頭の単語として、単語IDが0の「you」が入力されます。このときSoftmaxレイヤがoutputする確率分布を見ると、「say」で最も高くなっていることが分かります。これは、「you」の次に出現する単語が「say」であることを正しく予測しているのです。当然ですが、このような正しい予測は「良い重み（うまく学習した重み）」があって初めて可能になります。

続いて、2つ目の単語である「say」を入力する箇所に注目してみましょう。このときSoftmaxレイヤの出力は「goodbye」と「hello」の2箇所で高くなっています。確かに、「you say goodbye」と「you say hello」はどちらも自然な文章です（ちなみに、ここで正解は「goodbye」です）。ここで注目すべきは、RNNレイヤは「you say」という文脈を“記憶”しているということです。これはより正確に言うと、「you say」という過去の情報をコンパクトな隠れ状態ベクトルとしてRNNが保持しているのです。そのような情報を上層のAffineレイヤへ、そして次時刻のRNNレイヤへと伝達するのがRNNレイヤの行う仕事です。

このように、RNNLM はこれまで入力された単語を“記憶”し、それを元に次に出現する単語を予測します。これを可能にしているのが RNN レイヤの存在です。RNN レイヤが過去から現在へとデータを継続して流すことによって、過去の情報をエンコードして記憶することを可能にしているのです。

5.4.2 Time レイヤの実装

これまでに私たちは、時系列データをまとめて処理するレイヤを Time RNN レイヤとして実装してきました。ここでも同様に、時系列データをまとめて処理するレイヤを Time Embedding レイヤや Time Affine レイヤなどといった名前で実装したいと思います。この Time ○○レイヤができると、私たちの目的とするニューラルネットワークは、図 5-27 のように実装することができます。

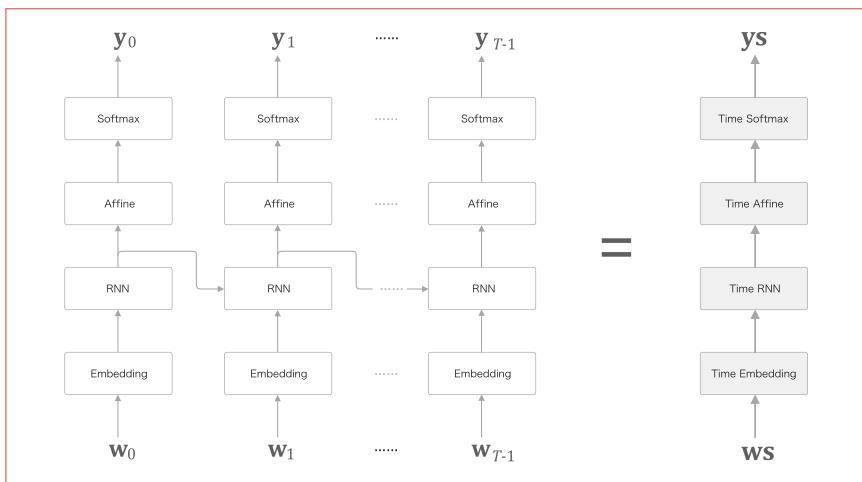


図 5-27 時系列データをまとめて処理するレイヤを Time ○○レイヤとして実装する



T 個分の時系列データをまとめて処理するレイヤを、私たちは「Time ○○レイヤ」と呼ぶことにします。そのようなレイヤが実装できれば、そのレイヤをレゴブロックのように組み合わせることで、時系列データを扱うネットワークを完成させることができます。

Time レイヤの実装は簡単です。たとえば、Time Affine レイヤの場合は、図 5-28 のように Affine レイヤを T 個だけ用意して、各時刻のデータを個別に処理するだけ

です。

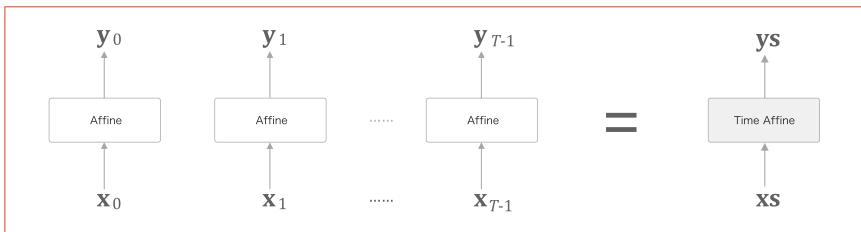


図5-28 Time Affine レイヤは、 T 個の Affine レイヤの集合として実装する

Time Embedding レイヤも同様に、順伝播時に T 個の Embedding レイヤを用意し、そして、各 Embedding レイヤが各時刻のデータを処理します。

Time Affine レイヤと Time Embedding レイヤは特に難しいことはないので、その説明は省略します。なお、Time Affine レイヤについては、単に T 個の Affine レイヤを利用するような実装は行わずに、行列計算としてまとめて処理する効率の良い実装を行っています。興味のある方はソースコード (`common/time_layers.py` の `TimeAffine` クラス) を参考にしてください。続いて時系列版の Softmax についてです。

Softmax については、損失誤差である Cross Entropy Error レイヤも合わせて実装します。ここでは、Time Softmax with Loss レイヤとして、図5-29 のようなネットワーク構成で実装します。

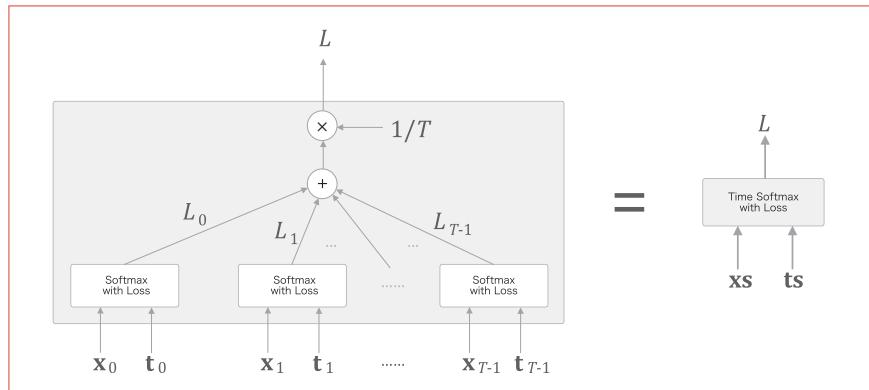


図 5-29 Time Softmax with Loss レイヤの全体図

図 5-29 にある x_0 や x_1 などのデータは、下の層から伝わる「スコア」を表します（スコアとは、確率へと正規化される前の値です）。また、 t_0 や t_1 などのデータは、正解ラベルを表します。図に示すとおり、 T 個の Softmax with Loss レイヤは、それぞれ損失を算出します。そしてそれらを合算し、その平均を取ったものを最終的な損失とします。なお、このとき行う計算は、数式で表すと次のようにになります。

$$L = \frac{1}{T}(L_0 + L_1 + \dots + L_{T-1}) \quad (5.11)$$

ところで本書の Softmax with Loss レイヤは、ミニバッチに関してその損失の平均を求めました。これは具体的に言うと、ミニバッチに N 個のデータがあれば、 N 個の損失の和を求めてそれを N で割ることで、データひとつあたりの平均の損失を求めました。ここでも同様に、時系列に関する平均を取ることで、最終的な出力としてデータひとつあたりの平均の損失を求めます。

以上が Time レイヤについての説明です。ここでは概要だけ簡単に説明しました。実際の実装については、`common/time_layers.py` にあります。興味のある方は参考にしてください。

5.5 RNNLM の学習と評価

RNNLM の実装に必要なレイヤは、すべて出揃いました。それでは RNNLM を実装し、実際に学習させてみましょう。そして、その出来栄えを評価したいと思います。

5.5.1 RNNLM の実装

ここでは、RNNLM で使用するネットワークを `SimpleRnnlm` というクラス名で実装したいと思います。この `SimpleRnnlm` のレイヤ構成は図 5-30 のようになります。

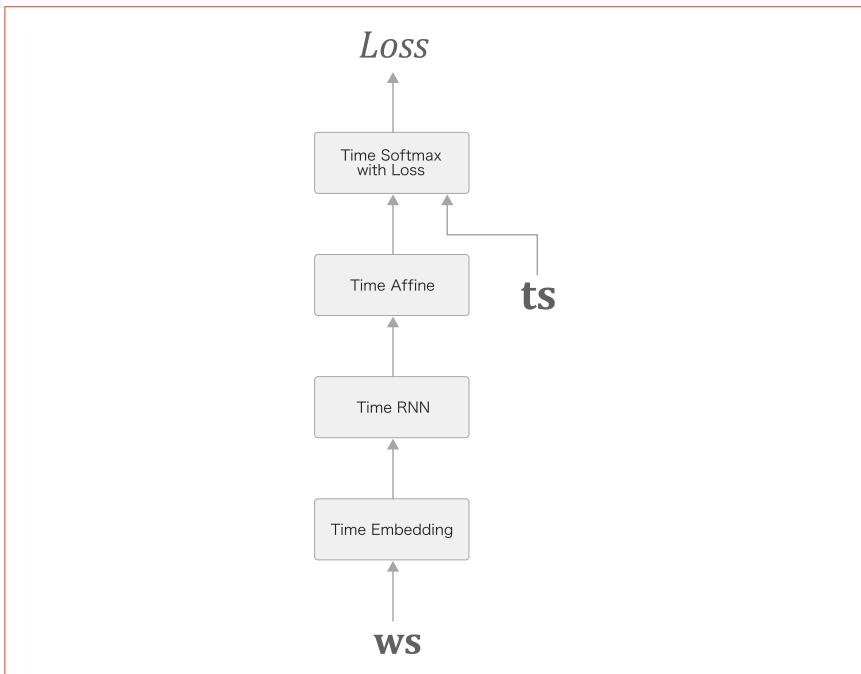


図 5-30 SimpleRnnlm のレイヤ構成：RNN レイヤの状態は、クラス内部で管理する

図 5-30 のとおり、`SimpleRnnlm` クラスは、4 つの Time レイヤを重ねたニューラルネットワークです。それでは最初に初期化のコードを示します ([☞ ch05/simple_rnnlm.py](#))。

```

import sys
sys.path.append('..')
import numpy as np
from common.time_layers import *

class SimpleRnnlm:
    def __init__(self, vocab_size, wordvec_size, hidden_size):
        self.layers = [TimeEmbedding(vocab_size, wordvec_size),
                      TimeRNN(wordvec_size, hidden_size),
                      TimeAffine(hidden_size, wordvec_size),
                      TimeSoftmaxWithLoss()]
        self.params, self.grads = [], []
        for layer in self.layers:
            self.params += layer.params
            self.grads += layer.grads
        self.wordvec_size = wordvec_size
        self.hidden_size = hidden_size
        self.vocab_size = vocab_size
        self.wordvec = np.random.randn(vocab_size, wordvec_size)
        self.h = None
        self.loss_layer = TimeSoftmaxWithLoss()
  
```

```

V, D, H = vocab_size, wordvec_size, hidden_size
rn = np.random.randn

# 重みの初期化
embed_W = (rn(V, D) / 100).astype('f')
rnn_Wx = (rn(D, H) / np.sqrt(D)).astype('f')
rnn_Wh = (rn(H, H) / np.sqrt(H)).astype('f')
rnn_b = np.zeros(H).astype('f')
affine_W = (rn(H, V) / np.sqrt(H)).astype('f')
affine_b = np.zeros(V).astype('f')

# レイヤの生成
self.layers = [
    TimeEmbedding(embed_W),
    TimeRNN(rnn_Wx, rnn_Wh, rnn_b, stateful=True),
    TimeAffine(affine_W, affine_b)
]
self.loss_layer = TimeSoftmaxWithLoss()
self.rnn_layer = self.layers[1]

# すべての重みと勾配をリストにまとめる
self.params, self.grads = [], []
for layer in self.layers:
    self.params += layer.params
    self.grads += layer.grads

```

ここでは、各レイヤで使用するパラメータ（重みとバイアス）を初期化し、必要なレイヤを生成します。また、Truncated BPTT で学習を行うことを想定して、Time RNN レイヤは `stateful` を `True` に設定しています。それによって、Time RNN レイヤは前時刻の隠れ状態を引き継ぐことができるようになります。

なお上の初期化のコードでは、RNN レイヤと Affine レイヤにおいて、「Xavier の初期値」を利用しています。Xavier の初期値では、前層のノードの個数を n とした場合、図 5-31 のように、 $\frac{1}{\sqrt{n}}$ の標準偏差を持つ分布を使います^{†2}。ちなみに標準偏差は、直感的には、データのばらつきを表す指標と解釈できます。

^{†2} ただし、これは簡略版の実装であり、オリジナルの論文では次層のノード数も考慮して重みの初期値が提案されています。

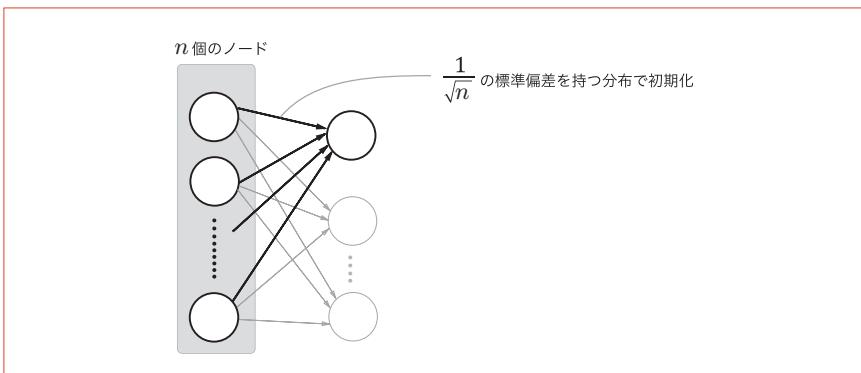


図 5-31 Xavier の初期値：前層から n 個のノードの接続がある場合、 $\frac{1}{\sqrt{n}}$ の標準偏差を持つ分布を初期値として使う



ディープラーニングでは重みの初期値が重要です。この点については、前作『ゼロから作る Deep Learning』の「6.2 重みの初期値」で詳しく議論しました。同様に RNN においても重みの初期値はとても重要です。良い初期値を設定することで、学習の進み方や最終的な精度が大きく変わります。本書ではこれ以降も、重みの初期値として「Xavier の初期値」を用います。なお、言語モデルを扱った研究では、`0.01 * np.random.uniform(...)` のようなスケール変換した一様分布を利用するケースが多く見受けられます。

それでは続いて、`forward()` メソッド、`backward()` メソッド、`reset_state()` メソッドの実装を示します。

```

def forward(self, xs, ts):
    for layer in self.layers:
        xs = layer.forward(xs)
    loss = self.loss_layer.forward(xs, ts)
    return loss

def backward(self, dout=1):
    dout = self.loss_layer.backward(dout)
    for layer in reversed(self.layers):
        dout = layer.backward(dout)
    return dout

def reset_state(self):
    self.rnn_layer.reset_state()

```

見てのとおり、この実装はとても簡単です。各レイヤには順伝播と逆伝播の実装

が適切に行われています。そのためここでは、各レイヤの `forward()` (もしくは `backward()`) を適切な順番で呼ぶだけになります。またここでは利便性を考え、ネットワークの状態をリセットするメソッドを `reset_state()` として実装します。以上が `SimpleRnnlm` クラスの説明です。

5.5.2 言語モデルの評価

`SimpleRnnlm` の実装が終わりました。後は、そのネットワークにデータを与えて学習を行うだけです。これから学習のためのコードを実装しますが、その前に言語モデルの「評価方法」について話をします。

言語モデルでは、過去の与えられた単語（情報）から次に出現する単語の確率分布を出力します。このとき、言語モデルの予測性能の良さを評価する指標として、**パープレキシティ** (perplexity) がよく用いられます。

パープレキシティは、簡単に言うと「確率の逆数」を表します（この解釈はデータ数がひとつのときに厳密に一致します）。「確率の逆数」という考え方を説明するために、ここでは「you say goodbye and I say hello .」というコーパスを考えます。このとき、「モデル 1」の言語モデルに「you」という単語を与えると、図 5-32 の左図に示すような確率分布を出力したとします。ここで次に出現する単語は「say」が正解だとすると、その確率は 0.8 です。これはなかなか良い予測と言えます。このときのパープレキシティは、その確率の逆数を取って $\frac{1}{0.8} = 1.25$ と計算できます。

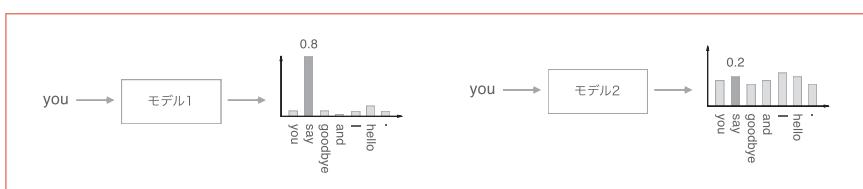


図 5-32 「you」という単語を入力し、次に出現する単語の確率分布を出力するモデルの例

一方図 5-32 の右図のモデル（「モデル 2」）は、正解となる単語について 0.2 と予測したとします。これは明らかに悪い予測と言えるでしょう。このときのパープレキシティは、 $\frac{1}{0.2} = 5$ となります。

ここまで話をまとめると、「モデル 1」は精度良く予測することができ、そのパープレキシティは 1.25 でした。一方、「モデル 2」は当てずっぽうな予測であり、このときのパープレキシティは 5.0 でした。この例から、パープレキシティは小さければ

小さいほど良いということが分かります。

それでは、1.25 や 5.0 という値は、直感的にはどのように解釈できるでしょうか？これは「分岐数」と解釈することができます。分岐数とは、次に取りうる選択肢の数——具体的に言うと、次に出現しうる単語の候補の数——ということです。先の例だと、良いモデルを予測した「分岐数」が 1.25 というのは、次に出現する候補が 1 個程度に絞れたということを意味します。一方、悪いモデルでは、次の候補がまだ 5 個もある状態を示しています。



上の例が示すように、パープレキシティによってモデルの予測性能を評価することができます。良いモデルは、正解となる単語を高い確率で予測できます。そのため、パープレキシティの値は小さくなります（パープレキシティの最小値は 1.0）。一方、悪いモデルは、正解となる単語を低い確率で予測します。そのため、パープレキシティの値は大きくなります。

ここまで話は、入力データが一つの場合のパープレキシティについてです。それでは、入力データが複数の場合はどうなるでしょうか？その場合は、次の式に従つて計算します。

$$L = -\frac{1}{N} \sum_n \sum_k t_{nk} \log y_{nk} \quad (5.12)$$

$$\text{perplexity} = e^L \quad (5.13)$$

ここではデータが N 個あるとします。また、 t_n は one-hot ベクトルの正解ラベルであり、 t_{nk} は n 個目のデータの k 番目の値を意味します。そして、 y_{nk} は確率分布を表します（ニューラルネットワークでは Softmax の出力）。ちなみに、この L はニューラルネットワークの損失であり、(1.8) とまったく同じ式です。この L を使って、 e^L を計算したものが、パープレキシティになります。

上の式 (5.12) はいくらか複雑になっていますが、直感的な理解はデータがひとつのときに説明した「確率の逆数」であり、「分岐数」や「選択肢の数」という理解がそのまま通用します。つまり、パープレキシティが小さくなるほど、分岐数が減り良いモデルであることが示せるのです。



情報理論の分野では、パープレキシティは「平均分岐数」とも呼ばれます。これは、データが1個のときに説明した「分岐数」を N 個の場合で平均したものという解釈です。

5.5.3 RNNLMの学習コード

PTBデータセットを利用してRNNLMの学習を行いましょう。ただし、ここではPTBデータセット（訓練データ）の先頭の1,000個の単語だけを利用することにします。というのも、ここで実装したRNNLMでは、すべての訓練データを対象にすると全然良い結果を出せないので。この改良は、次章で行います。それでは、学習のための実装を次に示します（☞ ch05/train_custom_loop.py）。

```

import sys
sys.path.append('..')
import matplotlib.pyplot as plt
import numpy as np
from common.optimizer import SGD
from dataset import ptb
from simple_rnnlm import SimpleRnnlm

# ハイパーパラメータの設定
batch_size = 10
wordvec_size = 100
hidden_size = 100 # RNNの隠れ状態ベクトルの要素数
time_size = 5 # Truncated BPTTの展開する時間サイズ
lr = 0.1
max_epoch = 100

# 学習データの読み込み（データセットを小さくする）
corpus, word_to_id, id_to_word = ptb.load_data('train')
corpus_size = 1000
corpus = corpus[:corpus_size]
vocab_size = int(max(corpus)) + 1

xs = corpus[:-1] # 入力
ts = corpus[1:] # 出力（教師ラベル）
data_size = len(xs)
print('corpus size: %d, vocabulary size: %d' % (corpus_size, vocab_size))

# 学習時に使用する変数
max_iters = data_size // (batch_size * time_size)
time_idx = 0
total_loss = 0
loss_count = 0

```

```

ppl_list = []

# モデルの生成
model = SimpleRnnlm(vocab_size, wordvec_size, hidden_size)
optimizer = SGD(lr)

# ❶ ミニバッチの各サンプルの読み込み開始位置を計算
jump = (corpus_size - 1) // batch_size
offsets = [i * jump for i in range(batch_size)]

for epoch in range(max_epoch):
    for iter in range(max_iters):
        # ❷ミニバッチの取得
        batch_x = np.empty((batch_size, time_size), dtype='i')
        batch_t = np.empty((batch_size, time_size), dtype='i')
        for t in range(time_size):
            for i, offset in enumerate(offsets):
                batch_x[i, t] = xs[(offset + time_idx) % data_size]
                batch_t[i, t] = ts[(offset + time_idx) % data_size]
        time_idx += 1

        # 勾配を求め、パラメータを更新
        loss = model.forward(batch_x, batch_t)
        model.backward()
        optimizer.update(model.params, model.grads)
        total_loss += loss
        loss_count += 1

# ❸ エポックごとにパープレキシティの評価
ppl = np.exp(total_loss / loss_count)
print('| epoch %d | perplexity %.2f' %
      (epoch+1, ppl))
ppl_list.append(float(ppl))
total_loss, loss_count = 0, 0

```

これが学習のためのコードです。これは、基本的にこれまで見てきたニューラルネットワークの学習とほとんど同じです。ただし、大きな視点では、2点だけこれまでの学習を行うコードとは異なります。それは「データの与え方」と「パープレキシティの計算」の点です。ここでは、その2点に絞って上のコードの説明を行います。

まずは「データの与え方」についてです。ここで私たちは Truncated BPTT によって学習を行います。そのため、データはシーケンシャルに与え、そしてミニバッチの各バッチでデータを与える開始位置をズラして読み込む必要があります。ソースコードの❶では、各バッチでデータを読み込む開始位置を `offsets` として計算します。この `offsets` の各要素には、データの読み込む開始位置（“ズレ”）が格納されています。

続いて、ソースコードの❷でデータをシーケンシャルに読み込みます。ここで

は、まず初めに“容器”として `batch_x` と `batch_t` を用意します。そして、変数 `time_idx` を順に（シケンシャルに）増やしながら、`time_idx` の場所のデータをコーパスから取得します。ここで❶で計算した `offsets` を利用して、各バッチでオフセットを加えます。また、コーパスを読み込む場所がコーパスサイズを超えた場合に、コーパスの先頭に戻るようにするため、コーパスのサイズで割った余りをインデックスとして使用します。

最後に、パープレキシティの計算は式 (5.12) によって行います。これはコードの❸で行っています。ここではエポックごとのパープレキシティを求めるため、エポックごとに損失の平均を求め、それを使ってパープレキシティを求めます。

以上がコードの説明です。それでは学習の結果を見てみましょう。上のコードでは、エポックごとのパープレキシティの結果が `perplexity_list` に格納されているので、それをプロットしてみることにします。結果は図5-33 のようになります。

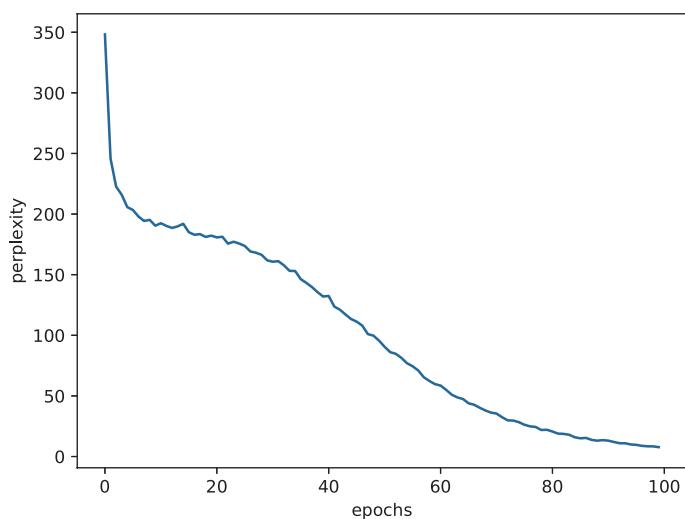


図5-33 パープレキシティの推移

図5-33を見ると、学習を進めるにつれてパープレキシティが順調に下がっていることが分かります。最初は300を超えていたパープレキシティが、最後のほうでは（最小値の）1に近づいています。ただし、ここで行った実験はコーパスサイズの小ささ

なデータセットを対象としたものです。実際のところ、コーパスのサイズが大きくなると、現状のモデルではまったく太刀打ちできません。次章では、現状の RNNLM の問題点を指摘し、その改良を行っていきます。

5.5.4 RNNLM の Trainer クラス

本書では、RNNLM を学習するために `RnnlmTrainer` クラスを提供します。このクラスは、先ほど行った RNNLM の学習を内部に隠蔽して行います。先ほどの学習コードを、`RnnlmTrainer` クラスを使って書き直すと、次のようにになります。ここではソースコードの一部を抜粋して示しています（実際のソースコードは `ch05/train.py` にあります）。

```
...
from common.trainer import RnnlmTrainer

...
model = SimpleRnnlm(vocab_size, wordvec_size, hidden_size)
optimizer = SGD(lr)
trainer = RnnlmTrainer(model, optimizer)

trainer.fit(xs, ts, max_epoch, batch_size, time_size)
```

上のように、初めに `RnnlmTrainer` クラスに `model` と `optimizer` を与えて初期化します。後は、`fit()` メソッドを呼ぶことで、学習が行われます。このとき、その内部では前節で行ったような一連の作業が行われます。その詳細を記せば、

- ミニバッチを“シーケンシャル”に作り、
- モデルの順伝播と逆伝播を呼び、
- オプティマイザで重みを更新し、
- パーペレキシティを評価する

という一連の処理になります。



`RnnlmTrainer` クラスは、「1.4.4 Trainer クラス」で説明した `Trainer` クラスと同じ API を持ちます。ニューラルネットワークの通常の学習は `Trainer` クラスを使い、RNNLM の学習には `RnnlmTrainer` クラスを使います。

`RnnlmTrainer` クラスを使うことで、私たちは毎回同じようなコードを書かずに

済みます。本書ではこれ以降、RNNLM の学習には `RnnlmTrainer` クラスを使うことにします。

5.6 まとめ

本章のテーマは RNN でした。RNN はデータを循環させることで、過去から現在、そして未来へとデータを継続して流します。それによって、RNN レイヤの内部に「隠れ状態」を記憶する能力を得ます。本章では、RNN レイヤの仕組みを時間をかけて説明し、実際に RNN レイヤ（そして Time RNN レイヤ）を実装しました。

また本章では、RNN を利用して言語モデルを作成しました。言語モデルは単語の羅列に対して確率を与えます。特に、条件付き言語モデルは、これまでの単語の羅列から次に出現する単語の確率を算出します。ここで RNN を利用したニューラルネットワークを構成することで、理論的には、どれだけ長い時系列データであっても重要な情報を RNN の隠れ状態に記憶することが可能になります。しかし実際の問題では、うまく学習できないケースが多くあります。次章では RNN の問題点を指摘し、RNN に代わる新しいレイヤ——LSTM レイヤや GRU レイヤ——を見ていきます。それらのレイヤは、時系列データ処理において最も重要なレイヤのひとつです。実際、最先端の研究においても多く利用されています。

本章で学んだこと

- RNN はループする経路があり、それによって「隠れ状態」を内部に記憶することができる
- RNN のループ経路を展開することで、複数の RNN レイヤがつながったネットワークと解釈することができ、通常の誤差逆伝播法によって学習することができる (= BPTT)
- 長い時系列データを学習する場合は、適当な長さでデータのまとまりを作り——これを「ブロック」と言う——、ブロック単位で BPTT による学習を行う (= Truncated BPTT)
- Truncated BPTT では逆伝播のつながりのみを切断する
- Truncated BPTT では順伝播のつながりは維持するため、データは“シーケンシャル”に与える必要がある
- 言語モデルは、単語の羅列を確率として解釈する
- RNN レイヤを利用した条件付き言語モデルは、(理論的には) それまで登場した単語の情報を記憶することができる

6章 ゲート付きRNN

忘却はよりよき前進を生む。

——ニーチェ

前章で見てきた RNN はループする経路を持ち、過去の情報を記憶することができました。さらに、その構造は単純であり、簡単に実装することができました。しかし残念なことに、前章の RNN はあまり性能が良くありません。その原因は、(多くの場合) 時系列データの長期の依存関係をうまく学習することができない点にあります。

現在では、前章の単純な RNN に代わって、LSTM や GRU と呼ばれるレイヤがよく使われます。実際のところ、「RNN」と言ったとき、それが指すレイヤは前章の RNN ではなく、LSTM であることが多く見受けられます。ちなみに、前章の RNN を明示的に指すときは「シンプルな RNN」や「エルマン (Elman)」などと呼びます。

LSTM や GRU には「ゲート」と呼ばれる仕組みが加わっています。そのゲートによって、時系列データの長期的な依存関係を学習することができるようになります。ここでは前章の RNN の問題点を指摘し、それに代わるレイヤとして LSTM や GRU のような「ゲート付き RNN」を紹介します。特に、LSTM の仕組みをじっくりと時間をかけて見ていき、それが“長い記憶”を可能にしているメカニズムを解き明かします。また、LSTM を使って言語モデルを作り、実際のデータでうまく学習できることを示します。

6.1 RNN の問題点

前章で説明した RNN は、時系列データの長期の依存関係を学習することが苦手で

す。その理由は、BPTT（Backpropagation Through Time）において勾配消失もしくは勾配爆発が起こることに原因があります。ここでは、前章で学んだ RNN レイヤの復習から始め、続いて、RNN レイヤが長期記憶を苦手とする理由を実例を使って説明していきます。

6.1.1 RNN の復習

RNN レイヤはループする経路を持ちます。そしてそのループを展開すると、それは横に長く伸びたネットワークとなりました。これは図で書くと図6-1 のようになります。

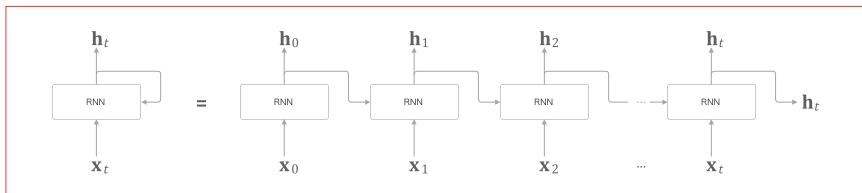


図6-1 RNN レイヤ：ループの展開前と展開後

図6-1 のように、RNN レイヤは時系列データである x_t を入力すると、 h_t を出力します。この h_t は、RNN レイヤの **隠れ状態** (hidden state) とも呼ばれ、その状態に過去からの情報が記憶されます。

RNN の特徴は、ひとつ前の時刻の隠れ状態を利用することです。それにより、過去の情報を引き継ぐことができるました。ちなみに、このとき RNN レイヤが行う処理を計算グラフで表すと図6-2 のようになります。

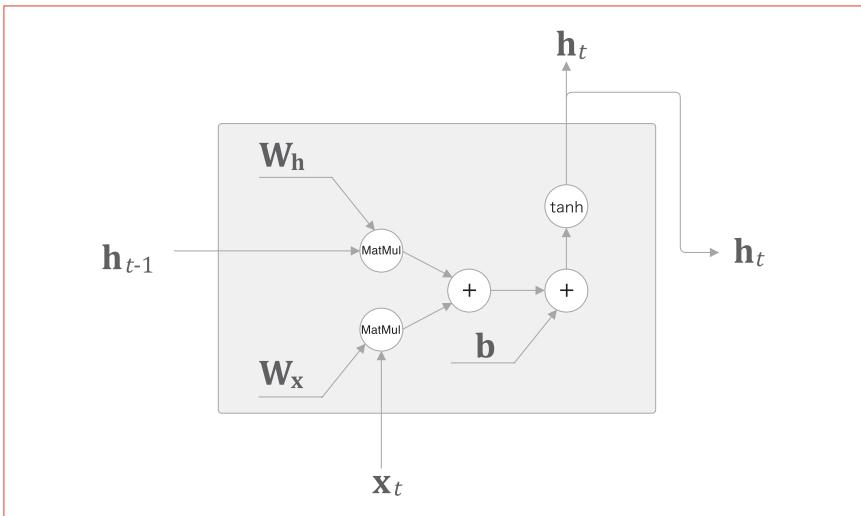


図6-2 RNN レイヤの計算グラフ（MatMul ノードは行列の積を表す）

図6-2で示すように、RNN レイヤの順伝播で行う計算は、行列の積と和、そして活性化関数である tanh 関数による変換から構成されます。以上が、前章で見てきた RNN レイヤです。続いて、この RNN レイヤが抱える問題——長期的な記憶に関する問題——を見ていきます。

6.1.2 勾配消失もしくは勾配爆発

言語モデルが行うこととは、これまでに与えられた単語から次に出現する単語を予測することです。前章では、RNN を使って言語モデルを実装しました（私たちはそれを RNNLM と呼びました）。ここでは RNNLM の問題点を指摘するにあたり、次の図6-3で示すタスクをもう一度考えてみたいと思います。

Tom was watching TV in his room. Mary came into the room. Mary said hi to ?

図6-3 「？」に入る単語は？：（ある程度の）“長い記憶”が必要な問題例

前にも述べたように、「？」に入る単語は「Tom」になります。これを RNNLM が正しく答えるには、現在の文脈として「Tom が部屋でテレビを見ていること」そして

「その部屋に Mary が入ってきたこと」を記憶しておく必要があります。そのような情報を RNN レイヤの隠れ状態にエンコードして保持しておかなければなりません。

それでは、上の問題例を RNNLM が学習する立場に立って考えてみたいと思います。ここでは、正解ラベルとして「Tom」という単語が与えられたとき、RNNLM の中で勾配がどのように伝播するかを見ていきます。もちろん、私たちはここで BPTT で学習を行います。そのため正解ラベルが「Tom」だと与えられた場所から、過去の方向に向かって勾配を伝えることになります。これは図で書くと図 6-4 のようになります。

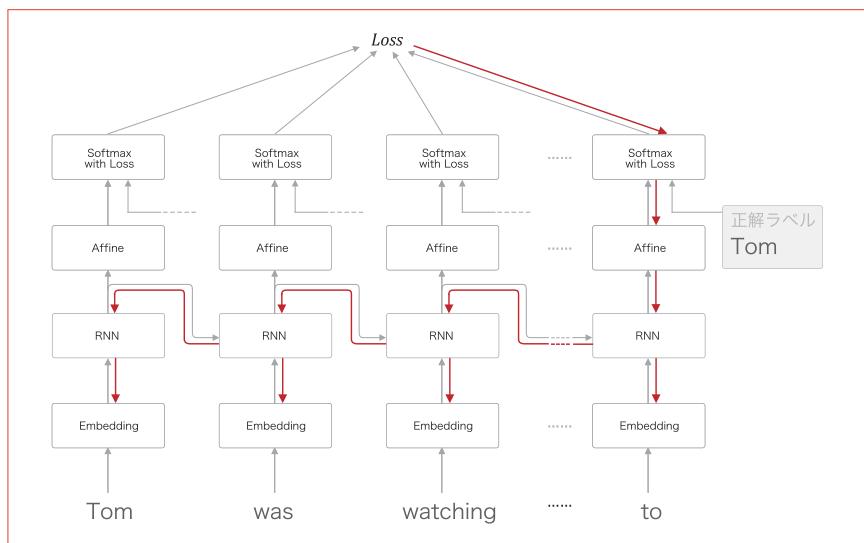


図 6-4 正解ラベルが「Tom」であることを学習するときの勾配の流れ

図 6-4 で示すように、正解ラベルが「Tom」であることを学習する際に重要なのが、RNN レイヤの存在です。RNN レイヤが過去方向に「意味のある勾配」を伝達することによって、時間方向の依存関係を学習することができます。このとき勾配には（本来であれば）学習すべき意味のある情報が入っており、それを過去に向かって伝えることで長期の依存関係を学習します。しかし、もしこの勾配が途中で弱まつたら——ほとんど何も情報を持たなくなってしまったら——、重みパラメータは更新されなくなります。つまり長期の依存関係を学習することができなくなってしまうのです。残念ながら、現在のシンプルな RNN レイヤでは、時間をさかのぼるに従って

勾配が小さくなる（勾配消失）もしくは大きくなる（勾配爆発）のどちらかの運命をたどってしまうことがほとんどなのです。

6.1.3 勾配消失もしくは勾配爆発の原因

それでは、RNN レイヤにおいて勾配消失（もしくは勾配爆発）が起こる原因を掘り下げて見ていきましょう。そのためにここでは図 6-5 に示すように、RNN レイヤの時間方向だけの勾配の伝播だけに着目します。

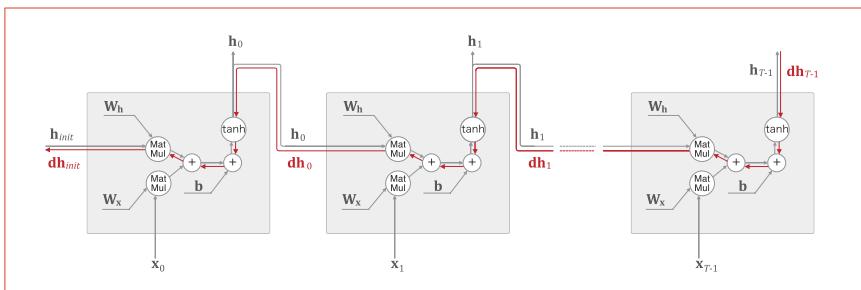


図 6-5 RNN レイヤの時間方向の勾配の伝播

図 6-5 に示すように、ここでは T 個の長さの時系列データを考え、 T 番目の正解ラベルから伝わる勾配がどのように変化するかに注視します。これは前の問題で言うと、 T 番目の正解ラベルが「Tom」である場合に相当します。このとき時間方向の勾配に着目すると、逆伝播によって伝わる勾配は「tanh」と「+」と「MatMul（行列の積）」の演算を通過することが分かります。

「+」の逆伝播は、上流から伝わる勾配をそのまま下流へ流すだけです。そのため、勾配の値は変わりません。それでは、残りの 2 つの演算「tanh」と「MatMul」ではどのように変化するでしょうか？まずは「tanh」について見てきましょう。

「付録 A sigmoid 関数と tanh 関数の微分」で詳しく説明しますが、 $y = \tanh(x)$ のとき、その微分は $\frac{\partial y}{\partial x} = 1 - y^2$ となります。このとき、 $y = \tanh(x)$ の値とその微分の値をそれぞれグラフにプロットすると、図 6-6 のようになります。

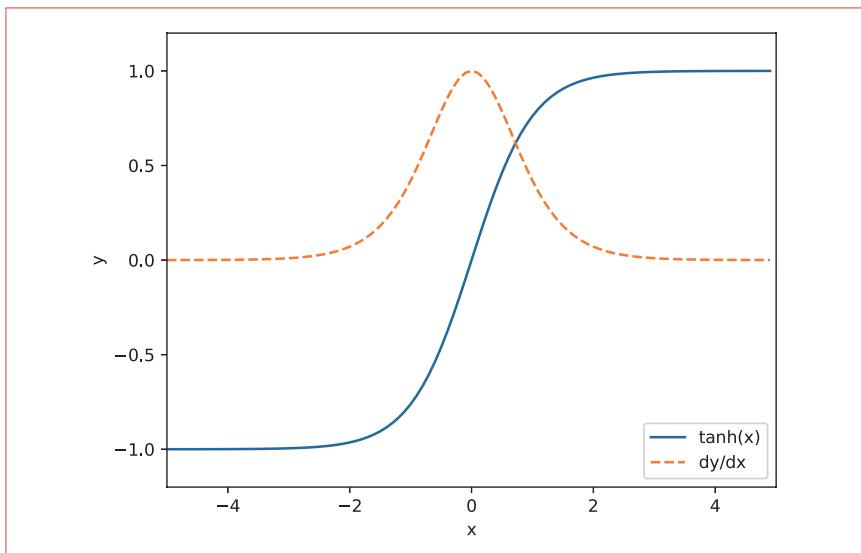
図6-6 $y = \tanh(x)$ のグラフ（破線は微分）

図6-6 の破線が $y = \tanh(x)$ の微分です。見てのとおり、その値は 1.0 以下となり、 x が 0 から遠ざかるにつれてその値は小さくなります。これが意味することは、逆伝播において勾配が tanh ノードを通るたびに、その値はどんどん小さくなっていくということです。そのため、tanh 関数を T 回通過すれば、勾配は T 回も繰り返し弱められることになります。



RNN レイヤの活性化関数では一般的に tanh 関数が使われますが、これを ReLU に変えることで勾配消失を抑えることが期待できます（ReLU への入力を x としたとき、その出力は $\max(0, x)$ となります）。なぜなら、ReLU の場合は入力 x が 0 以上であれば、逆伝播では上流の勾配をそのまま下流に流すことになり、勾配の“劣化”が起こらないからです。実際に、『Improving performance of recurrent neural network with relu nonlinearity』というタイトルの論文 [29] では、ReLU を用いて性能向上を達成しています。

続いて、図6-5 の MatMul（行列の積）ノードに注目しましょう。ここでは話を単純にするため、図6-5 の tanh ノードを無視することにします。そうすると、RNN レイヤの逆伝播の勾配は、図6-7 のように「MatMul」の演算によってのみ変化する

ことになります。

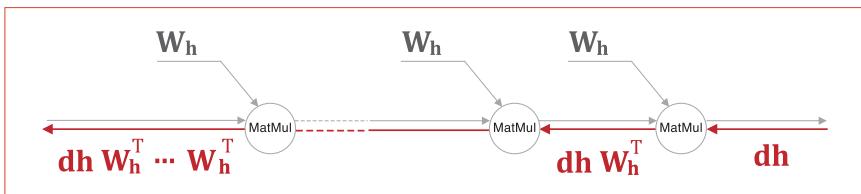


図 6-7 RNN レイヤの行列の積だけに注目したときの逆伝播の勾配

図 6-7 では、 dh という勾配が上流から伝わってくることを想定します。このとき MatMul ノードでの逆伝播は、 $\text{dh} \mathbf{W}_h^T$ による行列の積によって勾配が計算されます。後はこれを時系列データの時間サイズ分だけ繰り返します。ここで注目すべき点は、この行列の積の計算では、毎回同じ重みである \mathbf{W}_h が使われるということです。

それでは、逆伝播の際の勾配の値は、 MatMul ノードを通るに従ってどのように変化していくのでしょうか？何事も疑問に思ったら実験です！ここでは次のコードによって、勾配の大きさの変化を観察してみます([ch06/rnn_gradient_graph.py](#))。

```
import numpy as np
import matplotlib.pyplot as plt

N = 2 # ミニバッチサイズ
H = 3 # 隠れ状態ベクトルの次元数
T = 20 # 時系列データの長さ

dh = np.ones((N, H))
np.random.seed(3) # 再現性のため乱数のシードを固定
Wh = np.random.randn(H, H)

norm_list = []
for t in range(T):
    dh = np.dot(dh, Wh.T)
    norm = np.sqrt(np.sum(dh**2)) / N
    norm_list.append(norm)
```

ここでは dh を `np.ones()` によって初期化します (`np.ones()` は、すべての要素が 1 の行列)。そして、逆伝播の MatMul ノードの数だけ dh を更新し、各ステップでの dh の大きさ（ノルム）を配列 `norm_list` に追加します。なおここでは、 dh の大きさとしてミニバッチ（ N 個）における平均の「L2 ノルム」を求めていきます。L2 ノルムとは、各要素の 2 乗の総和に対して平方根を適用したものです。

それでは、上のコードを実行した結果（norm_list）をグラフにプロットしてみましょう。結果は次のようにになります。

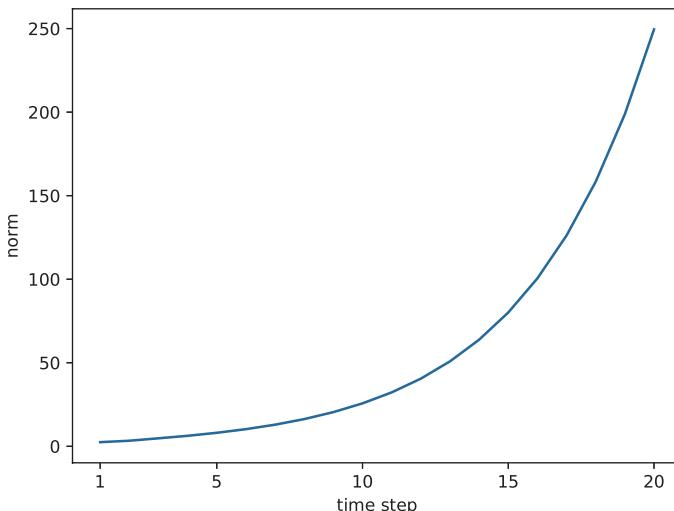


図6-8 勾配 dh の大きさは、時間サイズに比例して指数的に増加

図6-8のとおり、勾配の大きさは時間とともに指数的に増加していることが分かります。これが**勾配爆発**（exploding gradients）です。このような勾配爆発が起こってしまうと、最終的にはオーバーフローを起こして NaN（Not a Number）のような値が発生します。そのため、ニューラルネットワークの学習が正しく行えなくなります。

それでは2つ目の実験として、 Wh の初期値を以下のように変更したいと思います。

```
# Wh = np.random.randn(H, H)      # before
Wh = np.random.randn(H, H) * 0.5  # after
```

そして、この初期値を使って前と同じ実験を行ってみます。このときの結果は次のようになります。

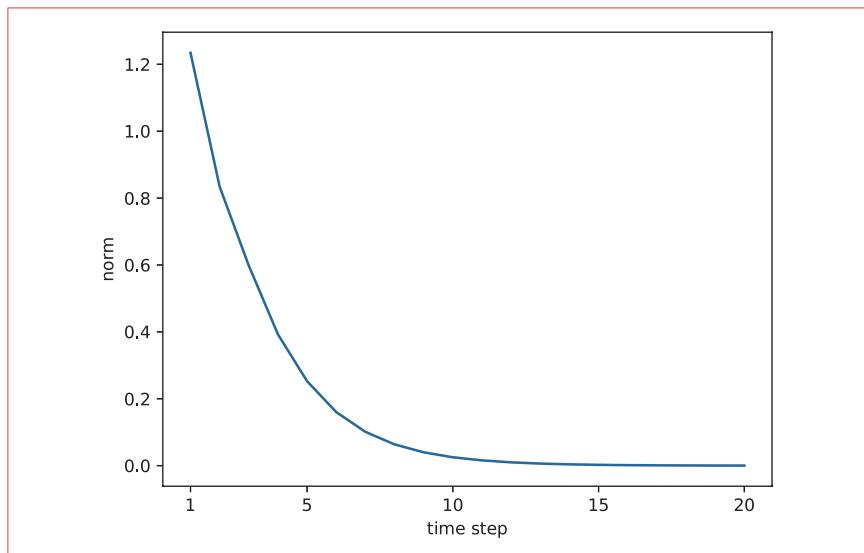


図6-9 勾配 dh の大きさは、時間サイズに比例して指数的に減少

図6-9を見て分かることおり、今度は指数的に減少する結果となりました。これが**勾配消失** (vanishing gradients) です。勾配消失が起こると、勾配の大きさが急速に小さくなります。もちろん勾配が小さくなると、重みパラメータが更新されなくなるため、長期的な依存関係は学習できなくなります。

ここで行った実験では、勾配の大きさは指数的に増加もしくは減少しました。なぜそのような指数的な変化が起こるかというと、もちろんそれは行列 Wh を T 回繰り返して乗算しているからです。もしここで Wh がスカラであれば話は単純です。その場合、 Wh が 1 より大きいときには指数的に増加し、1 より小さいときには指数的に減少します。

それでは Wh がスカラではなく行列の場合はどうでしょうか？ その場合、行列の「特異値」が指標になります。行列の特異値は、簡単に言えば、データにどれだけ広がりがあるかを表します。この特異値の値が——より正確には、複数ある特異値の中でその最大値が——1 より大きいかどうかで、勾配の大きさの変化を予測することができます。



特異値の最大値が 1 より大きい場合は指数的に増加する可能性が高いと予測できます。一方、特異値が 1 より小さい場合には指数的に減少すると判断できます。ここで、特異値が 1 より大きいときは必ず勾配爆発になるとは限りません。つまりこれは必要条件であって十分条件とはいえないのです。RNN の勾配消失／勾配爆発の詳細については、文献 [30] に詳しい議論があります。興味のある方は参照してください。

6.1.4 勾配爆発への対策

ここまで RNN の問題点——勾配爆発と勾配消失——について話してきました。それでは続いて、その回避策について見ていきましょう。ここでは初めに勾配爆発から見ていきます。

勾配爆発への対策は、定番の手法があります。それは **勾配クリッピング** (gradients clipping) と呼ばれる手法です。これはとても単純な手法であり、そのアルゴリズムは擬似コードで書くと次のようになります。

```
if || $\hat{g}$ ||  $\geq$  threshold :
     $\hat{g}$  =  $\frac{\text{threshold}}{||\hat{g}||} \hat{g}$ 
```

ここでは、ニューラルネットワークで使われるすべてのパラメータに対する勾配をひとつにまとめていることを想定して、これを \hat{g} という記号で表します。そして、*threshold* をしきい値として設定します。このとき、勾配の L2 ノルム（数式では $||\hat{g}||$ ）がしきい値を超えた場合、上のように勾配を修正します。これが勾配クリッピングです。見てのとおり単純なアルゴリズムですが、多くの場合うまくいきます。



\hat{g} は、ニューラルネットワークで使われるすべてのパラメータの勾配をひとつにまとめたものです。たとえば、あるモデルには重み $W1$ と $W2$ の 2 つがパラメータとしてあったとき、その 2 つのパラメータに対する勾配 $dW1$ と $dW2$ を結合したものを \hat{g} とします。

それでは、勾配クリッピングを Python で実装しましょう。ここでは、勾配クリッピングを `clip_grads(grads, max_norm)` という関数で実装することにします。引数の `grads` は勾配のリスト、`max_norm` はしきい値とします。このとき、勾配クリッピングは次のように実装できます (\Rightarrow `ch06/clip_grads.py`)。

```

import numpy as np

dW1 = np.random.rand(3, 3) * 10
dW2 = np.random.rand(3, 3) * 10
grads = [dW1, dW2]
max_norm = 5.0

def clip_grads(grads, max_norm):
    total_norm = 0
    for grad in grads:
        total_norm += np.sum(grad ** 2)
    total_norm = np.sqrt(total_norm)

    rate = max_norm / (total_norm + 1e-6)
    if rate < 1:
        for grad in grads:
            grad *= rate

clip_grads(grads, max_norm)

```

これが勾配クリッピングの実装です。見てのとおり、特に難しい点はないでしょう。この `clip_grads(grads, max_norm)` は、これから先も利用するので、`common/util.py` にも同じ実装を用意しておきます。



本書では、RNNLM を学習させるクラスを `RnnlmTrainer` クラス (`common/trainer.py`) として提供しています。その内部では、上の勾配クリッピングの関数を利用して勾配爆発を防止します。`RnnlmTrainer` クラスで勾配クリッピングを行うことについては「6.4 LSTM を使った言語モデル」で再度説明します。

以上が勾配クリッピングの説明です。続いて、勾配消失の対策について見ていきます。

6.2 勾配消失と LSTM

RNN の学習においては勾配消失も大きな問題です。そして、それを解決するには RNN レイヤのアーキテクチャを根本から変える必要があります。ここで登場するのが、本章のメインテーマである「ゲート付き RNN」です。このゲート付き RNN には

多くのアーキテクチャ（ネットワーク構成）が提案されており、その代表格に LSTM と GRU があります。本節では LSTM にフォーカスして、その仕組みをじっくりと見ていきます。そして、それが勾配消失を起こさない（もしくは起こしにくい）ことを明らかにします。なお、GRU については、「付録 C GRU」にて説明を行います。

6.2.1 LSTM のインターフェース

これから LSTM レイヤについて詳しく見ていきます。その前にここでは、今後のことを考えて、計算グラフ上での「シンプルな図法」を導入します。その図法は、図 6-10 のように、行列計算などをひとつの長方形ノードとしてまとめて表します。

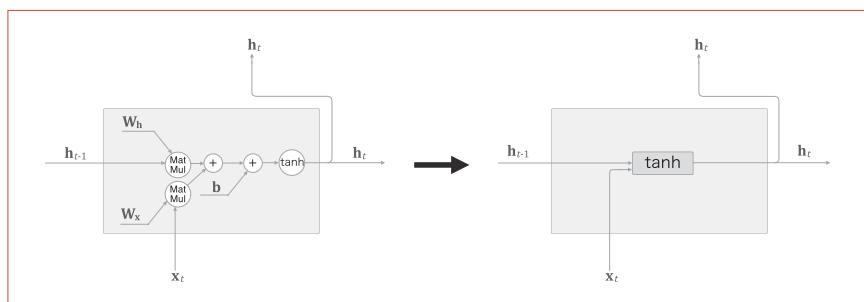


図 6-10 シンプルな図法を適用した RNN レイヤ：本節では見やすさを考慮して、シンプルな図法を使用する

図 6-10 に示すように、ここでは $\tanh(h_{t-1}W_h + x_tW_x + b)$ という計算を、ひとつの長方形ノードの「tanh」で表すことにします (h_{t-1} と x_t は行ベクトル)。この長方形ノードの中に行列の積やバイアスの和、そして tanh 関数による変換が含まれていると考えます。

これで準備が整いました。まずは LSTM のインターフェース（入出力）について、RNN と比較するところからスタートしましょう（図 6-11）。

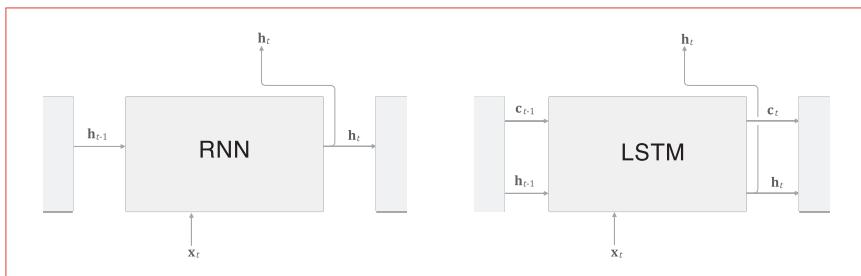


図6-11 RNN レイヤと LSTM レイヤの比較

図6-11に示すように、RNNとLSTMレイヤのインターフェースの違いは、LSTMには c という経路があることです。この c は記憶セル（もしくは単に「セル」と呼ばれ、LSTM専用の記憶部に相当します。

記憶セルの特徴は、それが自分自身だけで（LSTMレイヤ内だけで）データの受け渡しをするということです。つまり、LSTMレイヤ内だけで完結し、他のレイヤへは出力しません。一方、LSTMの隠れ状態 h は、RNNレイヤと同じく、（上方向へと）別のレイヤへと出力されます。



LSTMの出力を受け取る側から見ると、LSTMの出力は隠れ状態ベクトルの h だけになります。記憶セルの c は外部には見えません。もしくは、その存在を考える必要はありません。

6.2.2 LSTM レイヤの組み立て

それでは、LSTMレイヤの中身を見ていきましょう。ここではLSTMのパツをひとつずつ組み立てながら、その仕組みをじっくり見ていきたいと思います。なおここで行う説明は、LSTMについての素晴らしい解説記事「colah's blog : Understanding LSTM Networks」[31]を参考にしています。

さて前に述べたとおり、LSTMには記憶セル c_t があります。この c_t には、時刻 t におけるLSTMの記憶が格納されており、これに過去から時刻 t までにおいて必要な情報がすべて格納されていると考えられます（もしくは、そうなるように学習を行います）。そして、その必要な情報が詰まった記憶を元に、外部のレイヤへ（そして、次時刻のLSTMへ）隠れ状態 h_t を出力します。このとき行う計算は、図6-12に示すように、記憶セルを tanh 関数によって変換したものを出力します。

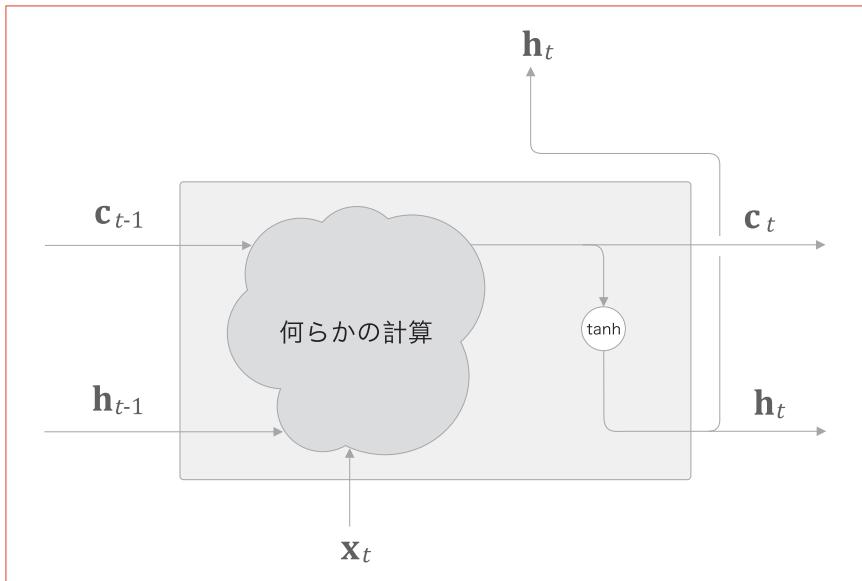
図6-12 記憶セル c_t を元に隠れ状態 h_t を計算する LSTM レイヤ

図6-12で示すように、現在の記憶セル c_t は、3つの入力 (c_{t-1} 、 h_{t-1} 、 x_t) から「何らかの計算」によって求められます（「何らかの計算」の詳細はすぐに説明します）。ここで大切なポイントは、更新された c_t を使って、隠れ状態の h_t が計算されるということです。なお、この計算は $h_t = \tanh(c_t)$ によって行われますが、これは c_t の各要素に対して \tanh 関数を適用することを意味します。



記憶セル c_t と隠れ状態 h_t の関係は、(ここまでのこと) 要素ごとに \tanh 関数を適用しただけです。これが意味することは、記憶セル c_t と隠れ状態 h_t の要素数は同じであるということです。たとえば、記憶セル c_t の要素数が 100 であれば、状態 h_t の要素数も 100 になります。

それでは次に進みましょう。その前にここでは「ゲート」という機能について簡単に説明します。ゲートとは日本語では「門」を意味する単語です。門が「開く/閉じる」のように、ゲートはデータの流れをコントロールします。イメージとしては、図6-13 のように、水の流れを止めたり出したりするのがゲートの役割です

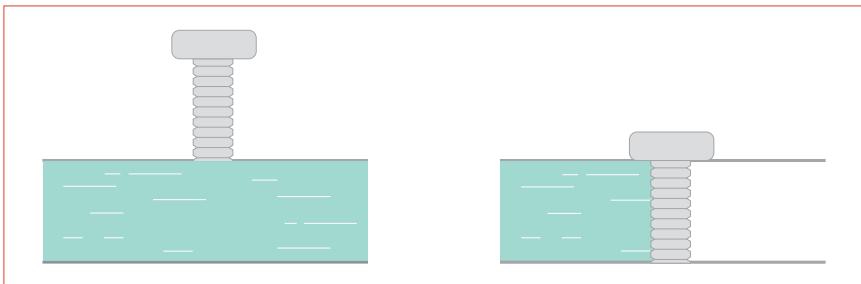


図6-13 ゲートのたとえ：水の流れをコントロールする

また、LSTMで使用するゲートは「開く/閉じる」の二択ではなく、どれだけゲートを開くか、そしてそれによって、どのくらいの量の水を次へ流すかということをコントロールします。これは図6-14のように、0.7(70%)や0.2(20%)のような「開き具合」もコントロールするということです。

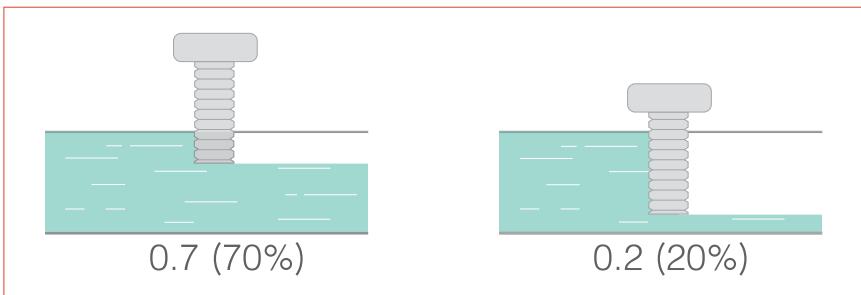


図6-14 水の流れる量を0.0~1.0の範囲でコントロールする

図6-14で示すように、ゲートの開き具合は0.0~1.0までの実数で表されます(1.0は全開)。そしてその数値によって、次へ流す水の量をコントロールするのです。ここで大切なのは、「どれだけゲートを開くか」ということも、データから(自動的に)学ばせることです。



ゲートでは、ゲートの開き具合をコントロールするために、それ専用の重みパラメータが用いられます。この重みパラメータは学習データによって更新されます。また、ゲートの開き具合を求めるにあたっては、sigmoid関数を使用します(sigmoid関数の出力は0.0~1.0の間の実数)。

6.2.3 output ゲート

それでは、LSTM に話を戻します。先ほどの説明では、隠れ状態 \mathbf{h}_t は記憶セル \mathbf{c}_t に対して、単に tanh 関数を適用しただけでした。ここで、 $\tanh(\mathbf{c}_t)$ に対してゲートを適用することを考えます。つまり、 $\tanh(\mathbf{c}_t)$ の各要素に対して、「それらが次時刻の隠れ状態としてどれだけ重要か」ということを調整するのです。なお、このゲートは、次の隠れ状態 \mathbf{h}_t の出力を司るゲートであることから **output ゲート**（出力ゲート）と呼ばれます。

output ゲートの開き具合——次へ何 % だけ通すか——は、入力 \mathbf{x}_t と前の状態 \mathbf{h}_{t-1} から求めます。このとき行う計算は次のようになります。なお、ここで使用する重みパラメータやバイアスの上添字には、output の頭文字である \mathbf{o} を追加しています。以降でも同様に上添字でゲートを示します。また sigmoid 関数は $\sigma()$ で表します。

$$\mathbf{o} = \sigma(\mathbf{x}_t \mathbf{W}_x^{(o)} + \mathbf{h}_{t-1} \mathbf{W}_h^{(o)} + \mathbf{b}^{(o)}) \quad (6.1)$$

式 (6.1) で示すように、入力 \mathbf{x}_t に対する重み $\mathbf{W}_x^{(o)}$ があり、前時刻の状態 \mathbf{h}_{t-1} に対する重み $\mathbf{W}_h^{(o)}$ があります (\mathbf{x}_t と \mathbf{h}_{t-1} は行ベクトルとします)。そして、これらの行列の積とバイアス $\mathbf{b}^{(o)}$ を足し合わせたものを sigmoid 関数を通して、それを出力ゲートの出力 \mathbf{o} とします。最後に、この \mathbf{o} と $\tanh(\mathbf{c}_t)$ の要素ごとの積を \mathbf{h}_t として出力します。それでは、ここで行う計算を計算グラフで書いてみましょう。結果は図6-15 のようになります。

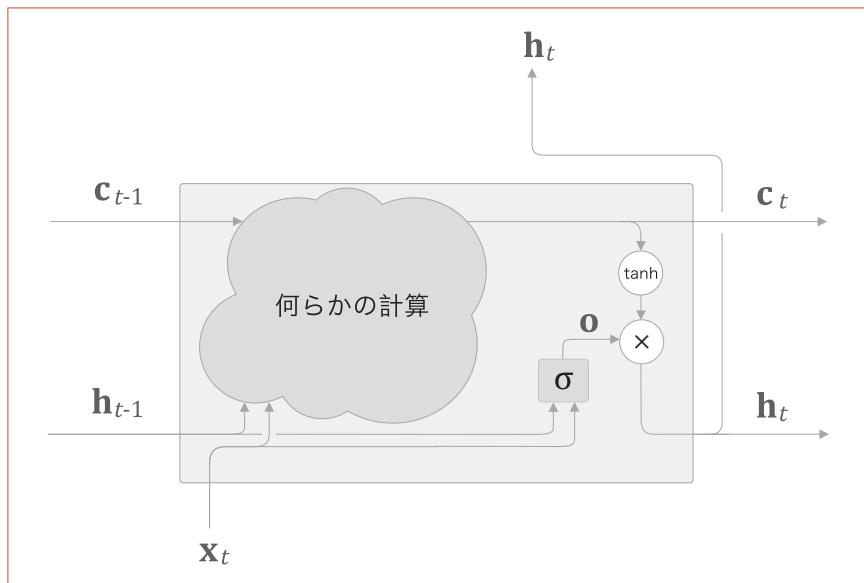


図 6-15 output ゲートを追加

図 6-15 に示すように、output ゲートで行う式 (6.1) の計算を「 σ 」で表すことにします。そして、その出力を \mathbf{o} とすると、 \mathbf{h}_t は \mathbf{o} と $\tanh(\mathbf{c}_t)$ の積によって計算されます。なお、ここで言う「積」とは要素ごとの積であり、これはアダマール積とも呼ばれます。アダマール積を \odot で表すとすると、ここで行う計算は次のようにになります。

$$\mathbf{h}_t = \mathbf{o} \odot \tanh(\mathbf{c}_t) \quad (6.2)$$

以上が LSTM の output ゲートです。これで LSTM の出力部分は完成です。続いて、記憶セルの更新部分を見ていきます。



\tanh の出力は $-1.0 \sim 1.0$ の実数です。この $-1.0 \sim 1.0$ の数値には、何らかのエンコードされた「情報」に対する強弱（度合い）が表されていると解釈できます。一方、sigmoid 関数の出力は $0.0 \sim 1.0$ の実数です。これは、データをどれだけ通すかという割合を表します。そのため、(多くの場合) ゲートでは sigmoid 関数、実質的な「情報」を持つデータには \tanh 関数が活性化関数として使われます。

6.2.4 forget ゲート

忘却はよりよき前進を生みます。私たちが次にやるべきことは、記憶セルに対して「何を忘れるか」を明示的に指示することです。それを実現するのには、もちろんゲートを使います。

それでは c_{t-1} の記憶から、不要な記憶を忘れるためのゲートを追加します。ここでは、それを **forget ゲート**（忘却ゲート）と呼びます。早速、forget ゲートを LSTM レイヤに追加してみると、計算グラフは図 6-16 のようになります。

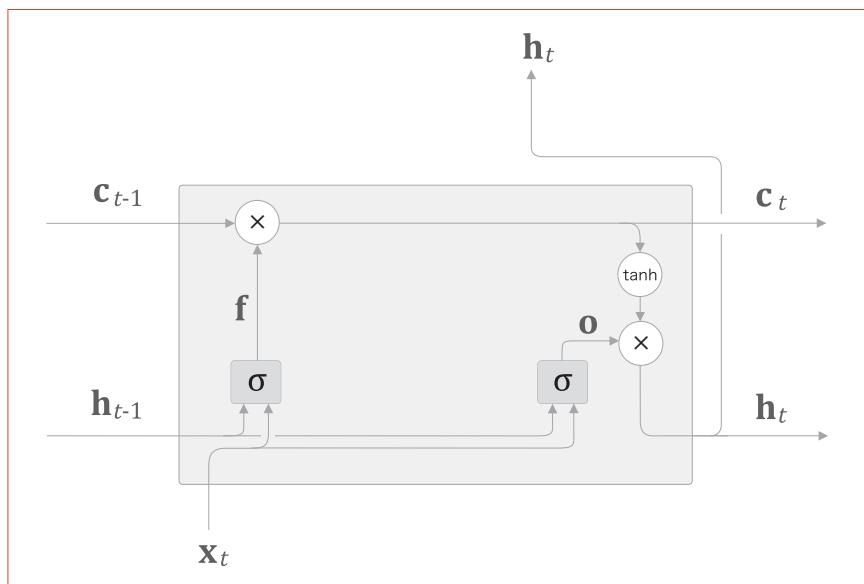


図 6-16 forget ゲートを追加

図 6-16 では forget ゲートで行う一連の計算を「 σ 」で表すことにします。この「 σ 」の中に forget ゲート専用の重みパラメータがあります。そして、このとき行う計算は次の式で表されます。

$$f = \sigma(x_t W_x^{(f)} + h_{t-1} W_h^{(f)} + b^{(f)}) \quad (6.3)$$

式 (6.3) によって、forget ゲートの出力 f が求められます。そして、この f と前の記憶セルである c_{t-1} との要素ごとの積によって——つまり、 $c_t = f \odot c_{t-1}$ によっ

て——、 c_t が求められます。

6.2.5 新しい記憶セル

forget ゲートによって、前時刻の記憶セルから忘れるべきものが削除されました。しかし、このままでは記憶セルは忘れることしかできません。そこで、新しく覚えるべき情報を記憶セルに追加したいと思います。そのためには図 6-17 のように tanh ノードを新たに追加します。

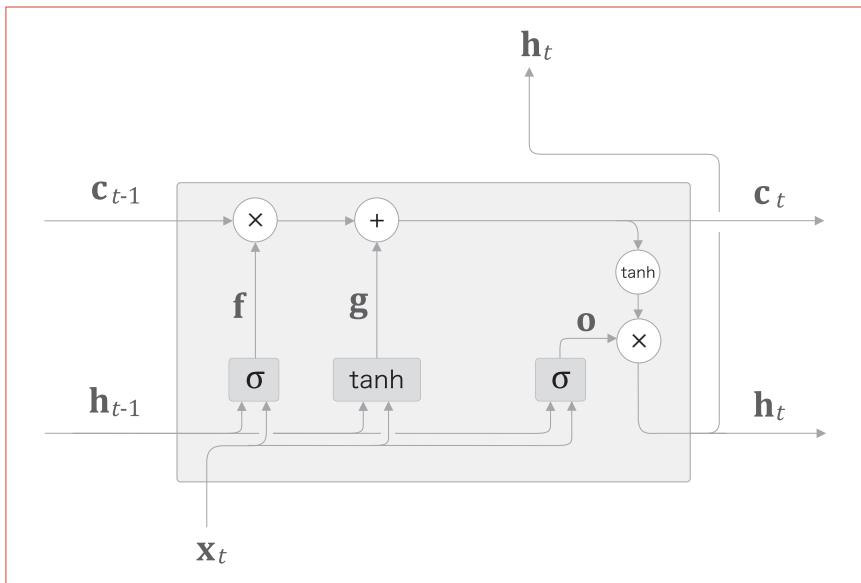


図 6-17 新しい記憶セルに必要な情報を追加

図 6-17 に示すように、tanh ノードによって計算された結果が前時刻の記憶セル c_{t-1} に加算されます。それによって、新しい「情報」が記憶セルに追加されます。この tanh ノードは「ゲート」ではなく、新しい「情報」を記憶セルに追加することを目的とします。そのため、活性化関数には sigmoid 関数ではなく tanh 関数が使われます。それでは、tanh ノードで行う計算を次に示します。

$$g = \tanh(x_t W_x^{(g)} + h_{t-1} W_h^{(g)} + b^{(g)}) \quad (6.4)$$

ここでは記憶セルに追加する新しい記憶を g で表すことにします。この g が前時刻の c_{t-1} に加算されることで、新しい記憶が生まれます。

6.2.6 input ゲート

最後に、図6-17の g に対してゲートを加えることを考えます。ここでは、新たに追加するゲートを **input ゲート**（入力ゲート）と呼びます。この input ゲートを追加すると、計算グラフは次のようになります。

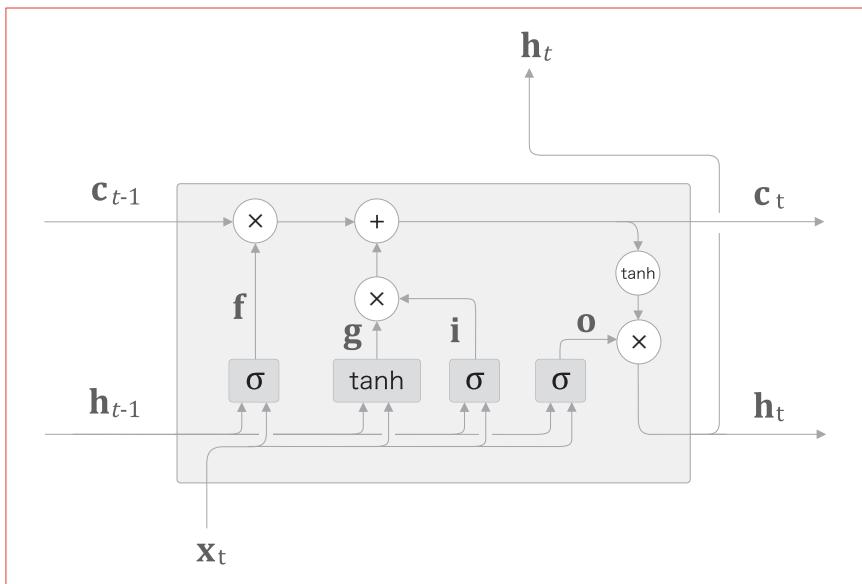


図6-18 input ゲートを追加

input ゲートは、 g の各要素が新たに追加する情報としてどれだけ価値があるかを判断します。この input ゲートによって、何も考えずに新しい情報を追加するのではなく、追加する情報の取捨選択を行います。これは別の見方をすると、input ゲートによって重み付けされた情報が新たに追加されることになります。

図6-18では、input ゲートを「 σ 」で表し、その出力を i とします。このとき行う計算は次のようになります。

$$i = \sigma(x_t W_x^{(i)} + h_{t-1} W_h^{(i)} + b^{(i)}) \quad (6.5)$$

後は、 i と g の要素ごとの積の結果を記憶セルに追加します。以上が、LSTM の内部で行われる処理の説明です。



LSTM には、いくつかの“亜種”があります。ここで説明した LSTM は代表的な LSTM ですが、この他にも、ゲートのつなぎ方の点で（若干）異なるレイヤも見受けられます。

6.2.7 LSTM の勾配の流れ

LSTM の仕組みは説明しましたが、これがなぜ勾配消失を起こさないのでしょうか？その理由は、記憶セル c の逆伝播に注目すると見えてきます。

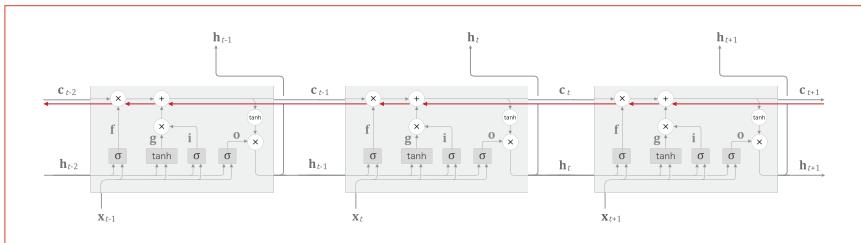


図 6-19 記憶セルの逆伝播

図 6-19 では、記憶セルだけにフォーカスして、その逆伝播の流れを描画しています。このとき記憶セルの逆伝播では「+」と「×」ノードだけを通ることになります。「+」ノードは、上流から伝わる勾配をそのまま流すだけです。そのため、勾配の変化（劣化）は起きません。

残るは「×」ノードの計算についてですが、これは「行列の積」ではなく、「要素ごとの積（アダマール積）」です。ちなみに、前に見た RNN の逆伝播では、同じ重みの行列を使って「行列の積」を繰り返し行っていました。そのため、それによって勾配消失（または勾配爆発）が起きたのです。一方、今回の LSTM の逆伝播では、「行列の積」の計算ではなく「要素ごとの積」です。そして、毎時刻、異なるゲート値によって要素ごとの積の計算が行われます。ここに勾配消失を起さない（もしくは起こしにくい）理由があります。

図 6-19 の「×」ノードの計算は forget ゲートによってコントロールされています（そしてそれは、毎時刻、異なるゲート値を出力します）。ここで forget ゲートが「忘

れるべき」と判断した記憶セルの要素に対しては、その勾配の要素は小さくなります。その一方で、forget ゲートが「忘れてはいけない」と導いた要素に対しては、その勾配の要素は劣化することなく過去方向へ伝わります。そのため、記憶セルの勾配は、(長期にわたって覚えておくべき情報に対しては) 勾配消失は起こさずに伝播することが期待できるのです。

以上の議論から、LSTM の記憶セルでは勾配消失が起きない（もしくは、起きにくい）ことが分かります。そのため、記憶セルには長期的な依存関係を保持すること（学習すること）が期待できるのです。



LSTM とは、Long short-term memory の頭文字から来ています。この言葉は、**短期記憶** (short-term memory) を**長い** (long) 時間継続できることを意味します。

6.3 LSTM の実装

それでは、LSTM の実装を行います。ここでは最初に 1 ステップの処理を LSTM クラスとして実装します。そして、T ステップ分をまとめて処理するクラスを TimeLSTM として実装します。初めに LSTM で行う計算をまとめて示します。

$$\begin{aligned} \mathbf{f} &= \sigma(\mathbf{x}_t \mathbf{W}_{\mathbf{x}}^{(f)} + \mathbf{h}_{t-1} \mathbf{W}_{\mathbf{h}}^{(f)} + \mathbf{b}^{(f)}) \\ \mathbf{g} &= \tanh(\mathbf{x}_t \mathbf{W}_{\mathbf{x}}^{(g)} + \mathbf{h}_{t-1} \mathbf{W}_{\mathbf{h}}^{(g)} + \mathbf{b}^{(g)}) \\ \mathbf{i} &= \sigma(\mathbf{x}_t \mathbf{W}_{\mathbf{x}}^{(i)} + \mathbf{h}_{t-1} \mathbf{W}_{\mathbf{h}}^{(i)} + \mathbf{b}^{(i)}) \\ \mathbf{o} &= \sigma(\mathbf{x}_t \mathbf{W}_{\mathbf{x}}^{(o)} + \mathbf{h}_{t-1} \mathbf{W}_{\mathbf{h}}^{(o)} + \mathbf{b}^{(o)}) \end{aligned} \quad (6.6)$$

$$\mathbf{c}_t = \mathbf{f} \odot \mathbf{c}_{t-1} + \mathbf{g} \odot \mathbf{i} \quad (6.7)$$

$$\mathbf{h}_t = \mathbf{o} \odot \tanh(\mathbf{c}_t) \quad (6.8)$$

上式が LSTM で行う計算です。ここで注目したいのは、式 (6.6) の 4 つのアフィン変換です（ここでの「アフィン変換」とは、 $\mathbf{x}\mathbf{W}_{\mathbf{x}} + \mathbf{h}\mathbf{W}_{\mathbf{h}} + \mathbf{b}$ のような式を指します）。式 (6.6) では 4 つの式によって個別にアフィン変換を行っていますが、これはひとつの式でまとめて計算することができるのです。その方法を図示すると、図 6-20 のようになります。

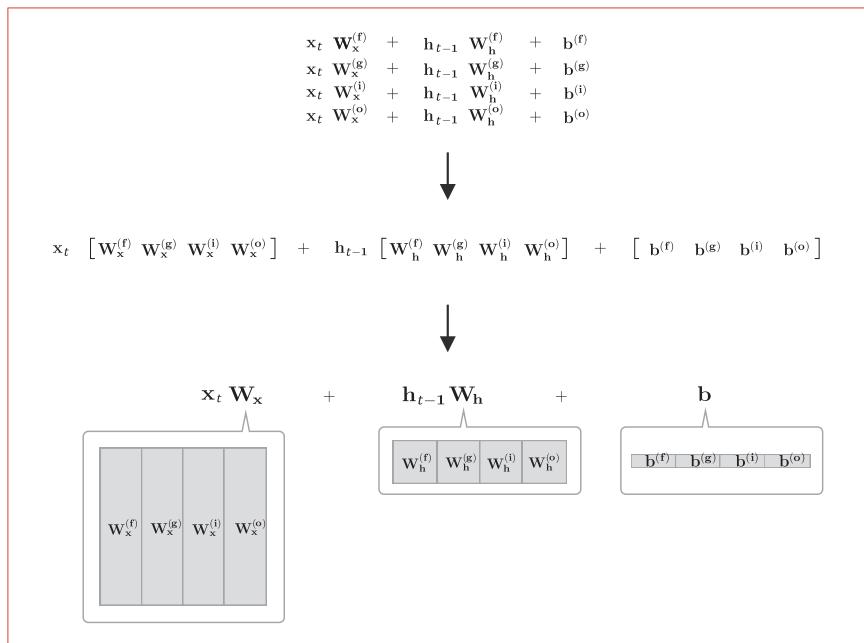


図6-20 4つの重みをまとめることで、1回のアフィン変換で4つの計算を行う

図6-20に示すように、4つの重み（またはバイアス）をひとつにまとめることができます。そうすれば、本来であれば4回個別に行ってアフィン変換の計算を1回の計算で済ませることができます。これによって、計算を高速化できます。というのも、行列ライブラリは「大きな行列」としてまとめて計算したほうが多くの場合速くなるからです。さらに重みをまとめて管理することでソースコードが簡潔になります。

それでは \mathbf{W}_x 、 \mathbf{W}_h 、 \mathbf{b} にそれぞれ 4 つ分の重み（もしくはバイアス）が含まれていると仮定して、このときの LSTM の計算グラフを図示します。そうすると、図6-21 のように書くことができます。

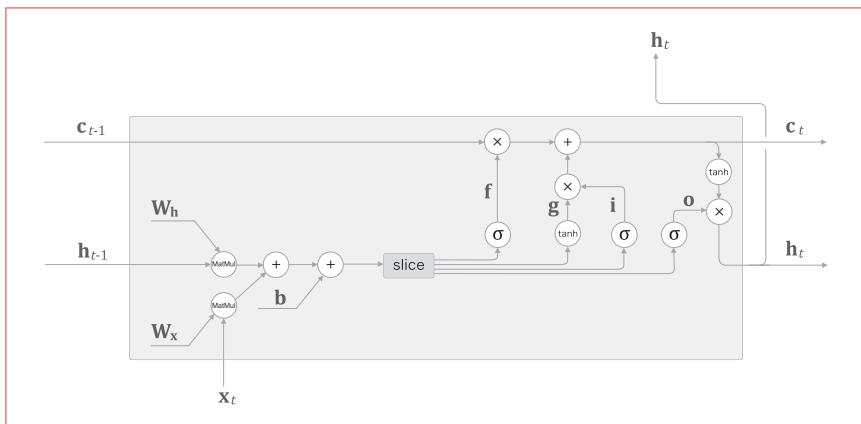


図6-21 4つ分の重みをまとめてアフィン変換を行う LSTM の計算グラフ

図6-21に示すように、ここでは初めに4つ分のアフィン変換をまとめて行います。そして、sliceノードによって、その4つの結果を取り出します。このsliceノードは、アフィン変換の結果（行列）を均等に4分割して取り出すだけの単純なノードです。sliceノードの後は、活性化関数（sigmoid関数またはtanh関数）を通り、前節で説明した計算を行います。

それでは図6-21を参考にして、LSTMクラスの実装を行いましょう。まずは、LSTMクラスの初期化のコードを示します（[common/time_layers.py](#)）。

```
class LSTM:
    def __init__(self, Wx, Wh, b):
        self.params = [Wx, Wh, b]
        self.grads = [np.zeros_like(Wx), np.zeros_like(Wh),
        → np.zeros_like(b)]
        self.cache = None
```

ここで初期化の引数は、重みパラメータの W_x と W_h 、そしてバイアスの b です。前にも述べたとおり、それらの重み（とバイアス）には4つ分の重みがまとめられています。この引数で与えられたパラメータはメンバ変数の `params` に設定し、それに対応する形で勾配も初期化します。またメンバ変数の `cache` は、順伝播での中間結果を保持するために使用し、それらは逆伝播の計算で使用します。

それでは続いて、順伝播の実装です。順伝播は `forward(x, h_prev, c_prev)` メソッドとします。引数は、現時刻の入力 x 、前時刻の隠れ状態 h_{prev} 、そして前時刻の記憶セル c_{prev} を受け取ることになります（[common/time_layers.py](#)）。

```
def forward(self, x, h_prev, c_prev):
    Wx, Wh, b = self.params
    N, H = h_prev.shape

    A = np.dot(x, Wx) + np.dot(h_prev, Wh) + b

    # slice
    f = A[:, :H]
    g = A[:, H:2*H]
    i = A[:, 2*H:3*H]
    o = A[:, 3*H:]

    f = sigmoid(f)
    g = np.tanh(g)
    i = sigmoid(i)
    o = sigmoid(o)

    c_next = f * c_prev + g * i
    h_next = o * np.tanh(c_next)

    self.cache = (x, h_prev, c_prev, i, f, g, o, c_next)
    return h_next, c_next
```

ここではまず初めに、アフィン変換が行われます。繰り返しになりますが、このときメンバ変数の Wx 、 Wh 、 b には、それぞれ 4 つ分のパラメータが格納されており、行列の形状に注目すると **図 6-22** のように推移します。

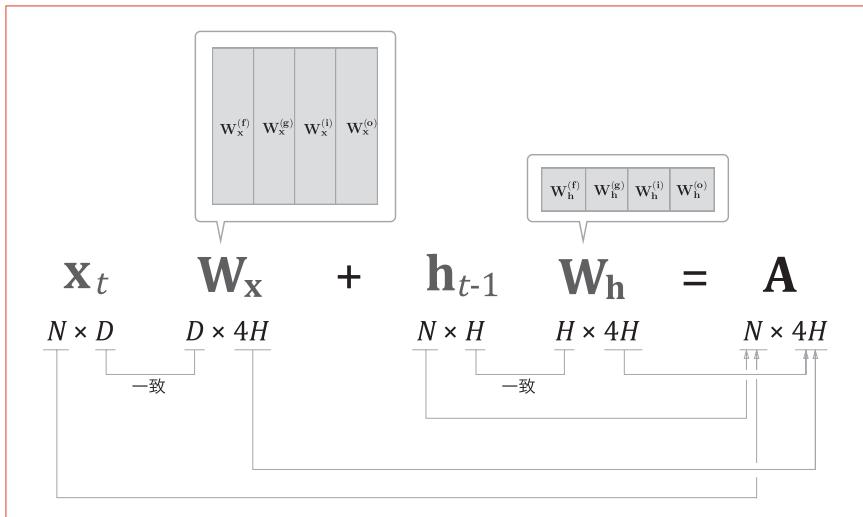


図 6-22 アフィン変換の形状の推移（バイアスは省略）

図 6-22 では、バッチ数を N 、入力データの次元数を D 、記憶セルと隠れ状態の次元数を両者とも H とします。そして、計算結果の A には 4 つ分のアフィン変換の結果が格納されています。そのため、そこから $A[:, :H]$ や $A[:, H:2*H]$ のようにスライスして取り出すことで、それ以降の演算ノードへ分配します。残りの実装については、LSTM の数式および計算グラフを参考にすれば、特に難しいことはないでしょう。



LSTM レイヤでは、4 つ分の重みをまとめて保持します。これにより、LSTM レイヤでは、 W_x と W_h そして b の 3 つのパラメータだけを管理すればよいことになります。ちなみに、RNN レイヤにおいても W_x と W_h そして b の 3 つのパラメータを保持しました。そのため、LSTM レイヤと RNN レイヤのパラメータの変数の数は同じですが、それらの形状が異なることになります。

LSTM の逆伝播は、図 6-21 の計算グラフを逆向きに伝播して求めることができます。これまでの知識を使えば、それは難しくはないはずです。ただし、slice ノードについては初見のため、その逆伝播についてだけ簡単に説明します。

slice ノードでは、行列を 4 分割して分配しました。そのためその逆伝播では、4 つの勾配を結合する必要があります。これは図で書くと図 6-23 のようになります。

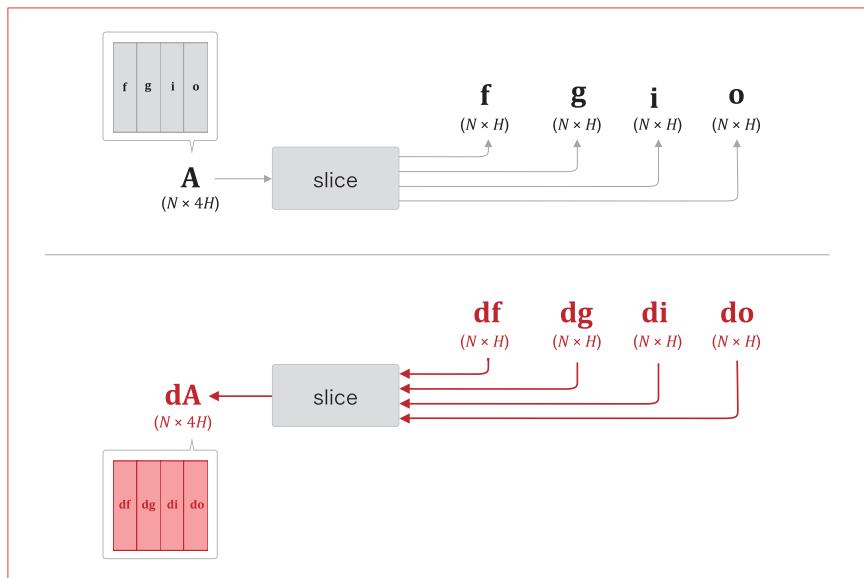


図6-23 slice ノードの順伝播（上）と逆伝播（下）

図6-23に示すように、sliceノードの逆伝播では4つの行列を連結します。図では、4つの勾配(df, dg, di, do)があり、それらを連結して dA を作ります。これをNumPyで行うには、`np.hstack()`が使えます。`np.hstack()`は、引数に与えられた配列を横方向に連結します（縦方向に連結するのは`np.vstack()`）。そのため、上の処理を行うには、次の1行で完成します。

```
dA = np.hstack((df, dg, di, do))
```

以上がsliceノードの逆伝播の説明です。

6.3.1 TimeLSTM の実装

それでは続いてTimeLSTMの実装に移ります。TimeLSTMは、 T 個分の時系列データをまとめて処理するレイヤです。その全体図は図6-24に示すように、 T 個のLSTMレイヤによって構成されます。

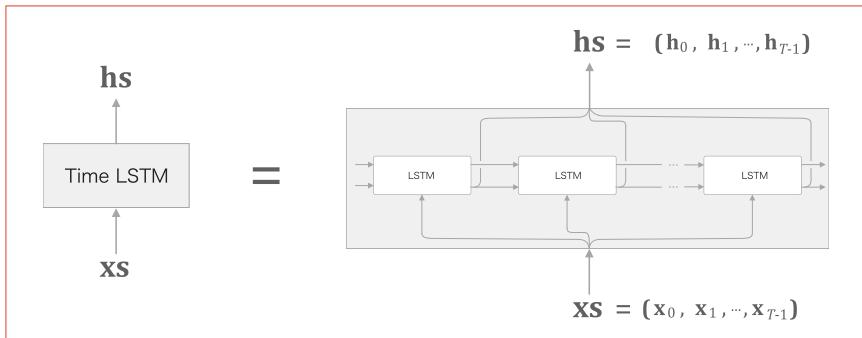


図6-24 Time LSTM の入出力

ところで、前にも述べたように、RNNで学習を行う際には Truncated BPTT を行います。Truncated BPTT は、逆伝播のつながりを適当な長さで断ち切りますが、順伝播の流れは維持する必要があります。そこで図6-25 のように、隠れ状態と記憶セルをメンバ変数に保持させるようにします。そうすることで、次の `forward()` が呼ばれたときに、前時刻の隠れ状態（と記憶セル）から開始させることができます。

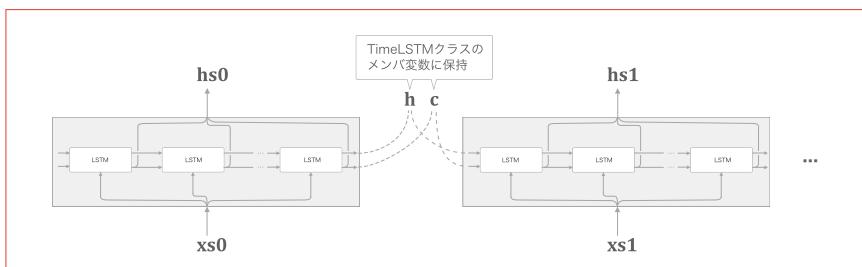


図6-25 Time LSTM の逆伝播の入出力

私たちはすでに Time RNN レイヤを実装してきました。ここでも同じ要領で Time LSTM レイヤを実装します。この `TimeLSTM` は次のように実装できます ([common/time_layers.py](#))。

```
class TimeLSTM:
    def __init__(self, Wx, Wh, b, stateful=False):
        self.params = [Wx, Wh, b]
```

```

    self.grads = [np.zeros_like(Wx), np.zeros_like(Wh),
    ↵ np.zeros_like(b)]
    self.layers = None

    self.h, self.c = None, None
    self.dh = None
    self.stateful = stateful

def forward(self, xs):
    Wx, Wh, b = self.params
    N, T, D = xs.shape
    H = Wh.shape[0]

    self.layers = []
    hs = np.empty((N, T, H), dtype='f')

    if not self.stateful or self.h is None:
        self.h = np.zeros((N, H), dtype='f')
    if not self.stateful or self.c is None:
        self.c = np.zeros((N, H), dtype='f')

    for t in range(T):
        layer = LSTM(*self.params)
        self.h, self.c = layer.forward(xs[:, t, :], self.h, self.c)
        hs[:, t, :] = self.h

    self.layers.append(layer)

return hs

def backward(self, dhs):
    Wx, Wh, b = self.params
    N, T, H = dhs.shape
    D = Wx.shape[0]

    dxs = np.empty((N, T, D), dtype='f')
    dh, dc = 0, 0

    grads = [0, 0, 0]
    for t in reversed(range(T)):
        layer = self.layers[t]
        dx, dh, dc = layer.backward(dhs[:, t, :] + dh, dc)
        dxs[:, t, :] = dx
        for i, grad in enumerate(layer.grads):
            grads[i] += grad

    for i, grad in enumerate(grads):
        self.grads[i][...] = grad

```

```
    self.dh = dh
    return dxs

def set_state(self, h, c=None):
    self.h, self.c = h, c

def reset_state(self):
    self.h, self.c = None, None
```

LSTM では隠れ状態の h に加えて記憶セル c も用いますが、TimeLSTM クラスの実装は TimeRNN クラスの場合とほとんど同じです。ここでも引数の `stateful` によって状態を維持するかどうかを指定します。それでは続いて、この TimeLSTM を使って言語モデルを作ります。

6.4 LSTM を使った言語モデル

Time LSTM レイヤの実装が終わったので、本題の「言語モデル」を実装します。ここで実装する言語モデルは、前章で実装した言語モデルとほとんど同じです。唯一異なる点は、図6-26 で示すように、前章では Time RNN レイヤだった場所に、今回は Time LSTM レイヤを使うことです。

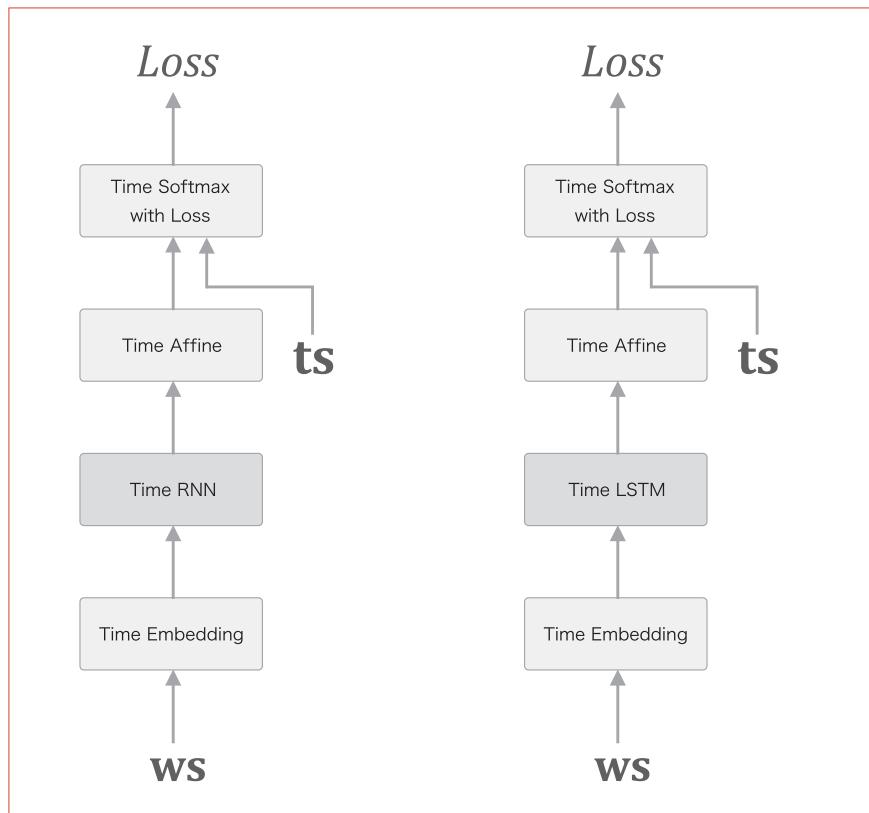


図6-26 言語モデルのネットワーク構成。左図は前章で作成した Time RNN を用いたモデル、右図は本章で作成する Time LSTM を用いたモデル

図6-26に示すとおり、前章で実装した言語モデルとの違いは LSTM を使用している点です。私たちは、図6-26の右図のニューラルネットワークを `Rnnlm` というクラス名で実装することにします。`Rnnlm` クラスは、前章で説明した `SimpleRnnlm` クラスとほとんど同じですが、新しいメソッドをいくつか追加します。それでは、LSTM レイヤを使った `Rnnlm` クラスの実装を次に示します^{†1}。

```
import sys
sys.path.append('..')
```

^{†1} ここで示す実装コードは `ch06/rnnlm.py` に対応しますが、`ch06/rnnlm.py` では `BaseModel` というクラスを継承することで、より簡略化された実装になっています。

```

from common.time_layers import *
import pickle

class Rnnlm:
    def __init__(self, vocab_size=10000, wordvec_size=100,
                 hidden_size=100):
        V, D, H = vocab_size, wordvec_size, hidden_size
        rn = np.random.randn

        # 重みの初期化
        embed_W = (rn(V, D) / 100).astype('f')
        lstm_Wx = (rn(D, 4 * H) / np.sqrt(D)).astype('f')
        lstm_Wh = (rn(H, 4 * H) / np.sqrt(H)).astype('f')
        lstm_b = np.zeros(4 * H).astype('f')
        affine_W = (rn(H, V) / np.sqrt(H)).astype('f')
        affine_b = np.zeros(V).astype('f')

        # レイヤの生成
        self.layers = [
            TimeEmbedding(embed_W),
            TimeLSTM(lstm_Wx, lstm_Wh, lstm_b, stateful=True),
            TimeAffine(affine_W, affine_b)
        ]
        self.loss_layer = TimeSoftmaxWithLoss()
        self.lstm_layer = self.layers[1]

        # すべての重みと勾配をリストにまとめる
        self.params, self.grads = [], []
        for layer in self.layers:
            self.params += layer.params
            self.grads += layer.grads

    def predict(self, xs):
        for layer in self.layers:
            xs = layer.forward(xs)
        return xs

    def forward(self, xs, ts):
        score = self.predict(xs)
        loss = self.loss_layer.forward(score, ts)
        return loss

    def backward(self, dout=1):
        dout = self.loss_layer.backward(dout)
        for layer in reversed(self.layers):
            dout = layer.backward(dout)
        return dout

```

```

def reset_state(self):
    self.lstm_layer.reset_state()

def save_params(self, file_name='Rnnlm.pkl'):
    with open(file_name, 'wb') as f:
        pickle.dump(self.params, f)

def load_params(self, file_name='Rnnlm.pkl'):
    with open(file_name, 'rb') as f:
        self.params = pickle.load(f)

```

`Rnnlm` クラスは、Softmax レイヤの直前までを処理する `predict()` メソッドを実装します。このメソッドは、7章で行う文章生成で使用します。また、パラメータの書き込み/読み込みのためのメソッド——`save_params()` と `load_params()` メソッド——も追加で実装します。残りは、前章の `SimpleRnnlm` クラスと同じ実装です。



`common/base_model.py` には `BaseModel` というクラスがあります。このクラスには、`save_params()` と `load_params()` メソッドが実装されています。そのため、`BaseModel` クラスを継承することでも、パラメータの読み書きの機能を得ることができます。さらに、`BaseModel` クラスにおける実装では、GPU 対応やビット削減（16 ビットの浮動小数点数で保存）などの最適化が行われています。

それでは、このネットワークを使って、PTB データセットの学習を行いましょう。今回は、PTB データセットの訓練データすべてを使って学習を行います（前章では、PTB データセットの一部を利用しました）。学習のためのコードを次に示します（[ch06/train_rnnlm.py](#)）。

```

import sys
sys.path.append('..')
from common.optimizer import SGD
from common.trainer import RnnlmTrainer
from common.util import eval_perplexity
from dataset import ptb
from rnnlm import Rnnlm

# ハイパーパラメータの設定
batch_size = 20

```

```

wordvec_size = 100
hidden_size = 100 # RNNの隠れ状態ベクトルの要素数
time_size = 35 # RNNを展開するサイズ
lr = 20.0
max_epoch = 4
max_grad = 0.25

# 学習データの読み込み
corpus, word_to_id, id_to_word = ptb.load_data('train')
corpus_test, _, _ = ptb.load_data('test')
vocab_size = len(word_to_id)
xs = corpus[:-1]
ts = corpus[1:]

# モデルの生成
model = Rnnlm(vocab_size, wordvec_size, hidden_size)
optimizer = SGD(lr)
trainer = RnnlmTrainer(model, optimizer)

# ❶ 勾配クリッピングを適用して学習
trainer.fit(xs, ts, max_epoch, batch_size, time_size, max_grad,
            eval_interval=20)
trainer.plot(ylim=(0, 500))

# ❷ テストデータで評価
model.reset_state()
ppl_test = eval_perplexity(model, corpus_test)
print('test perplexity: ', ppl_test)

# ❸ パラメータの保存
model.save_params()

```

ここで示すソースコードは、前章のコード（ch05/train.py）と多くが共通しています。そのため、ここでは前章から異なる点を中心に解説します。まずはコードの❶において、RnnlmTrainer クラスを使ってモデルの学習を行います。RnnlmTrainer クラスの fit() メソッドは、モデルの勾配を求め、モデルのパラメータを更新します。このとき、引数の max_grad を指定することで、勾配クリッピングが適用されます。ちなみに fit() メソッドの内部では、次のような実装が行われています（ここでは擬似コードを示します）。

```

# 勾配を求める
model.forward(...)
model.backward(...)
params, grads = model.params, model.grads
# 勾配クリッピング

```

```
if max_grad is not None:  
    clip_grads(grads, max_grad)  
# パラメータの更新  
optimizer.update(params, grads)
```

私たちは勾配クリッピングを `clip_grads(grads, max_grad)` として「6.1.4 勾配爆発への対策」で実装しました。ここでは、そのメソッドを利用して、勾配クリッピングを行います。

また、❶の `fit()` メソッドの `eval_interval=20` という引数によって、20 イテレーションおきにパープレキシティを評価します。今回はデータサイズが大きいのでエポックごとの評価ではなく、20 イテレーションごとに評価し、後ほどその結果を `plot()` メソッドでグラフとして描画します。

コードの❷の箇所では、学習が終わった後にテストデータを使用してパープレキシティの評価を行います。このとき、モデルの状態（LSTM の隠れ状態と記憶セル）をリセットして評価を行う点に注意しましょう。なお、パープレキシティを評価する関数は `common/util.py` に `eval_perplexity()` として実装しているので、そちらを利用します（その実装はとても単純です）。

最後に❸の箇所で、学習後のパラメータを外部ファイルに保存します。これは次章で、学習後の重みパラメータを使って文章生成をする際に使用します。以上が RNNLM の学習のコードです。それでは、このコードを実行してみましょう。そうすると、ターミナルに次の結果が出力されます。

```
$python train_rnnlm.py
| epoch 1 | iter 1 / 1327 | time 0[s] | perplexity 10000.84
| epoch 1 | iter 21 / 1327 | time 4[s] | perplexity 3065.17
| epoch 1 | iter 41 / 1327 | time 9[s] | perplexity 1255.96
| epoch 1 | iter 61 / 1327 | time 14[s] | perplexity 956.13
| epoch 1 | iter 81 / 1327 | time 18[s] | perplexity 806.56
| epoch 1 | iter 101 / 1327 | time 23[s] | perplexity 658.86
| epoch 1 | iter 121 / 1327 | time 27[s] | perplexity 636.88
| epoch 1 | iter 141 / 1327 | time 31[s] | perplexity 601.75
| epoch 1 | iter 161 / 1327 | time 35[s] | perplexity 575.78
| epoch 1 | iter 181 / 1327 | time 40[s] | perplexity 590.01
| epoch 1 | iter 201 / 1327 | time 44[s] | perplexity 479.95
| epoch 1 | iter 221 / 1327 | time 48[s] | perplexity 488.23
| epoch 1 | iter 241 / 1327 | time 53[s] | perplexity 443.62
| epoch 1 | iter 261 / 1327 | time 57[s] | perplexity 468.75
| epoch 1 | iter 281 / 1327 | time 61[s] | perplexity 447.81
| epoch 1 | iter 301 / 1327 | time 66[s] | perplexity 398.51
| epoch 1 | iter 321 / 1327 | time 70[s] | perplexity 350.89
| epoch 1 | iter 341 / 1327 | time 74[s] | perplexity 406.82
| epoch 1 | iter 361 / 1327 | time 79[s] | perplexity 409.33
| epoch 1 | iter 381 / 1327 | time 83[s] | perplexity 337.04
```

図6-27 ターミナルの出力結果

図6-27では、20イテレーションおきのパープレキシティの値が表示されています。その結果を見ると、先頭のパープレキシティは10000.84となっています。これは、(直感的には)次に来る単語が10,000個程度に絞れたことを意味します。今回のデータセットは語彙数が10,000個なので、これはまだ何も学習していない状態であり、当たずっぽうな予測ということになります。ですが学習を続けるに従い、期待どおりにパープレキシティが良くなっています。実際にはイテレーションが300回を超えたあたりからパープレキシティが400を下回っています。それでは、パープレキシティの推移をグラフで見てみましょう。結果は次のようになります。

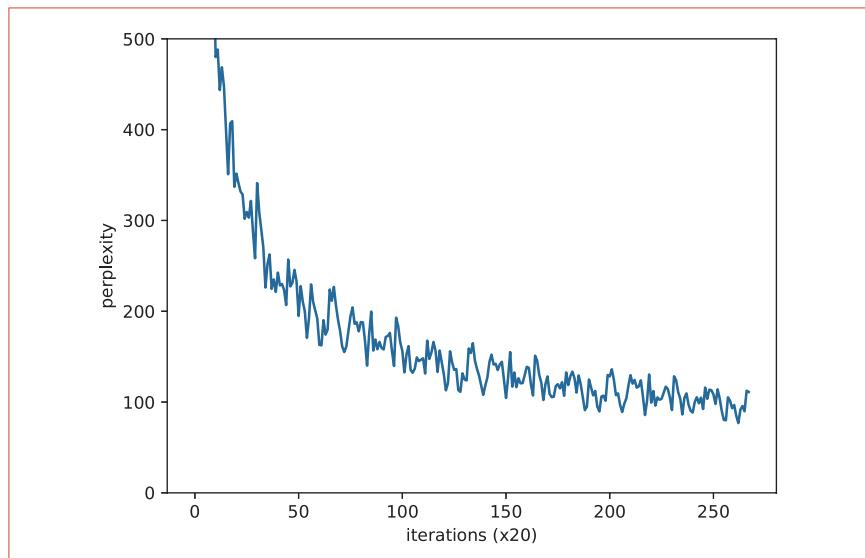


図6-28 パープレキシティの推移（訓練データを対象に 20 イテレーションごとに評価した結果）

今回の実験では、全部で 4 エポック（イテレーション換算では $1327 * 4$ 回）の学習を行いました。図6-28 が示すとおり、パープレキシティは順調に下がり、最終的には 100 程度になりました。そして最終的なテストデータでの評価（ソースコードの②の箇所）は 136.07... となります。この結果は、実行ごとに毎回異なりますが、およそ 135 前後になるでしょう。つまり私たちのモデルは、次に来る単語をおよそ（10,000 個の中から）136 個程度に絞れるまでに成長したということです。

それでは、136 というパープレキシティの値は、実際のところどうなのでしょうか？ 実を言うと、これはあまり良い結果ではありません。というのも、2017 年における最先端の研究では、PTB データセットのパープレキシティは 60 を下回っているのです [34]。私たちのモデルには、まだまだ改善の余地が大いにあります。続いて、現在の RNNLM をさらに改善していきます。

6.5 RNNLM のさらなる改善

本章では、現状の RNNLM に対して改善すべきポイントを 3 つ説明します。そして、それらの改善点を実装して、最終的にどれくらい精度が向上するのかを評価した

いと思います。

6.5.1 LSTM レイヤの多層化

RNNLM で高精度なモデルを作ろうとした場合、LSTM レイヤをディープにすること——レイヤを何層も重ねること——は、多くの場合有効です。これまで私たちは LSTM レイヤを 1 層だけ用いましたが、これを 2 層、3 層といったように複数重ねることで言語モデルの精度向上が期待できます。たとえば、LSTM を 2 層使って RNNLM を作るとすれば、図 6-29 のようになります。

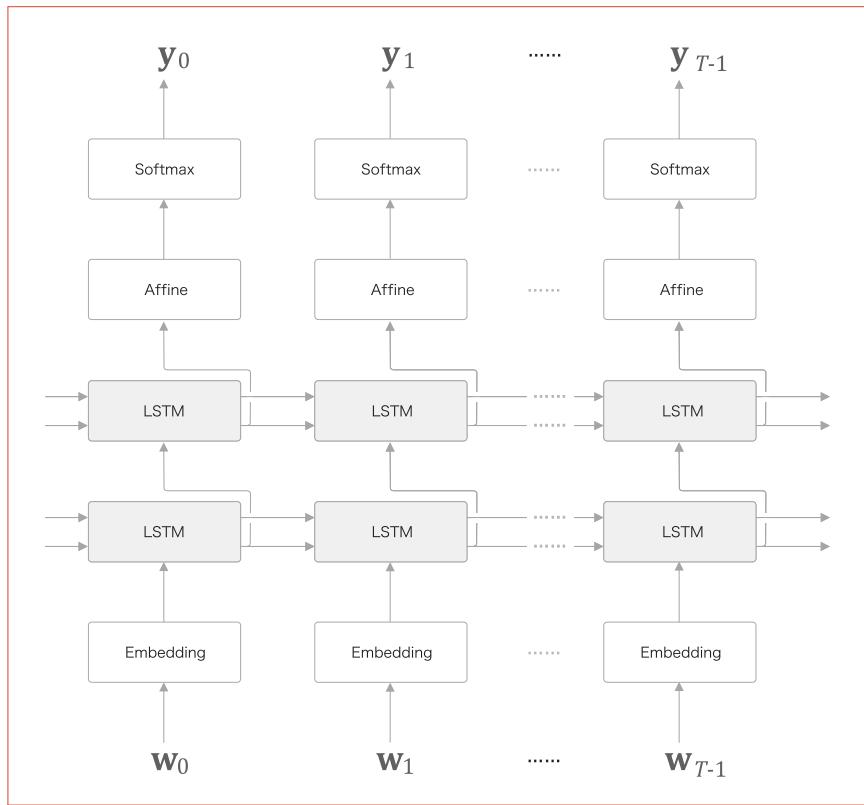


図 6-29 LSTM レイヤを 2 層重ねたときの RNNLM

図 6-29 は、LSTM レイヤを 2 層重ねた例を示しています。このとき、ひとつ目の

LSTM レイヤの隠れ状態が 2 つ目の LSTM レイヤの入力になっています。これと同じ要領で、LSTM レイヤを何層にも重ねることができます、それによって、より複雑なパターンを学習することが可能になります。これは、フィードフォワード型ニューラルネットワークのときのディープ化の話と同じです。前作『ゼロから作る Deep Learning』においては、Affine レイヤや Convolution レイヤなどを何層も重ねることで、より表現力の高いモデルを作成しました。

それでは、どれだけ層を重ねるべきでしょうか？もちろんそれはハイパーパラメータに関する問題です。重なる層数はハイパーパラメータであるため、取り組む問題の複雑さや用意された学習データの量に応じて適宜決める必要があります。ちなみに、PTB データセットの言語モデルの場合は、LSTM の層数は 2~4 程度が良い結果が得られるようです。



Google 翻訳で使われる GNMT [50] と呼ばれるモデルは、LSTM を 8 層も重ねたネットワークであることが報告されています。その例が示すように、取り組む問題が複雑で、学習データが大量に用意できるのであれば、LSTM レイヤを“ディープ”にすることによって、精度向上が期待できます。

6.5.2 Dropout による過学習の抑制

LSTM レイヤを多層化することで、時系列データの複雑な依存関係を学習することができます。これは言い方をえれば、層を深くすることで表現力が豊かなモデルを作れるということです。しかし、そのようなモデルは往々にして過学習 (overfitting、過剰適合とも言う) を起こします。さらに悪いニュースとして、RNN は通常のフィードフォワードなネットワークよりも簡単に過学習を起こします。そのため、RNN の過学習対策は重要であり、現在でも活発に研究が行われているテーマなのです。



過学習とは、訓練データだけに対して学習しすぎてしまう状態を指します。つまり、汎化能力が欠如した状態が過学習です。私たちが望むのは、汎化能力の高いモデルです。そのためには、訓練データの評価と検証データの評価の差から、過学習を起こしていないかを判断して、モデルの設計を行う必要があります。

過学習を抑制するには、定番の方法があります。まずは「訓練データを増やす」と。そして、「モデルの複雑さを減らす」こと。この 2 つが真っ先に考えられます。

その他には、モデルの複雑さにペナルティを与える正則化も有効です。たとえば、L2正則化では、重みが大きくなりすぎることにペナルティを課します。

また、Dropout [9] のように、訓練時にレイヤ内のニューロンのいくつか（たとえば 50% など）をランダムに無視して学習を行うのも、一種の正則化と考えられます（図 6-30）。ここでは Dropout について詳しく見ていき、それを RNN に適用したいと思います。

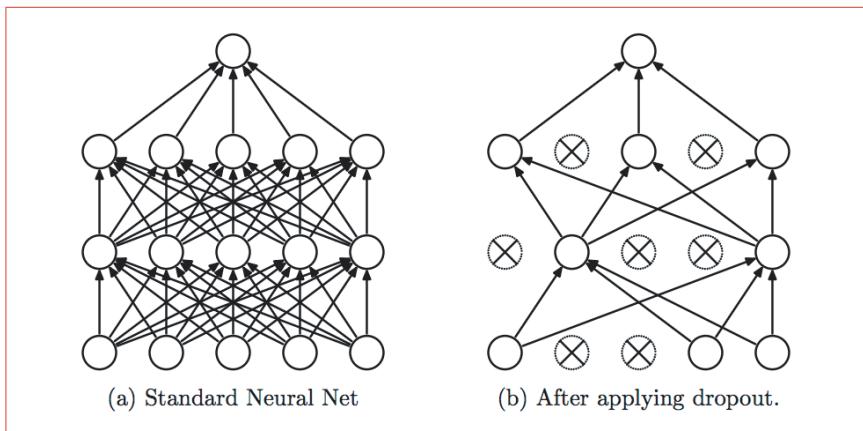


図 6-30 Dropout の概念図（文献[9]より引用）：左が通常のニューラルネットワーク、右が Dropout を適用したネットワーク

図 6-30 のとおり、Dropout はランダムにニューロンを選び、そのニューロンを無視することで、その先の信号の伝達をストップします。この“ランダムな無視”が制約となり、ニューラルネットワークの汎化性能を向上させることができます。前作『ゼロから作る Deep Learning』では、Dropout レイヤを実装しました。そこでは、図 6-31 のように Dropout レイヤを活性化関数の後に挿入する例を示し、それが過学習抑制に貢献することを示しました。

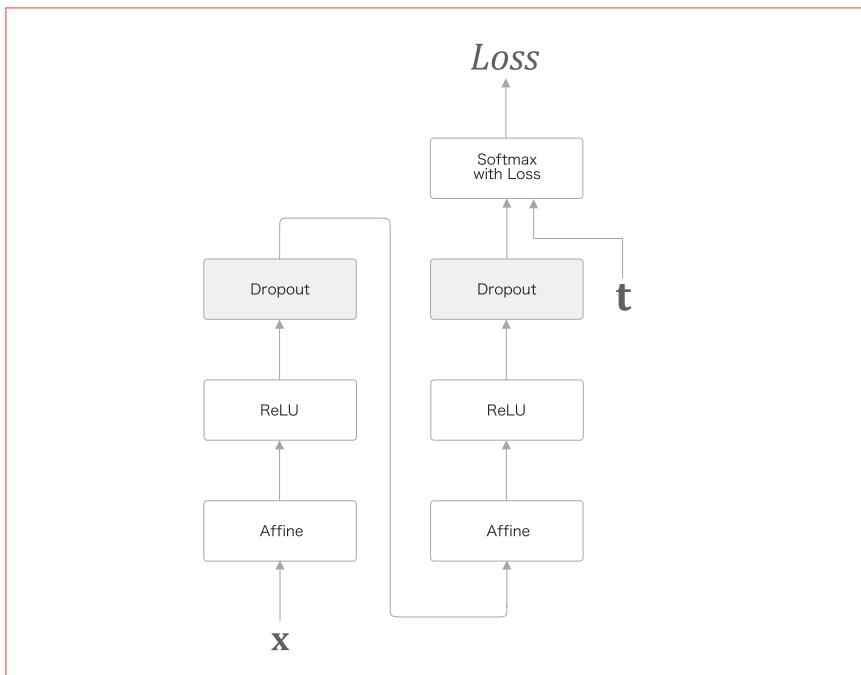


図6-31 フィードフォワード型ニューラルネットワークに Dropout レイヤを適用する例

それでは RNN を使ったモデルでは、どこに Dropout レイヤを挿入すべきでしょうか？ひとつ目に考えられるのは、図6-32 のように LSTM レイヤの時系列方向です。しかし先に答えを言ってしまうと、これは良い使い方ではありません。

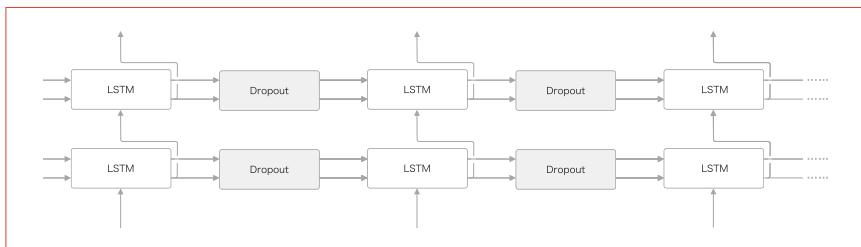


図6-32 悪い例：Dropout レイヤを時系列方向に挿入

RNN で時系列方向に Dropout を入れてしまうと、(学習時には) 時間が進むにつ

れて情報が失われることになります。つまり、時間が進むのに比例して Dropout によるノイズが蓄積することになるのです。ノイズの蓄積を考慮すると、時間軸方向への Dropout はやめたほうがよさそうです。そこで、図6-33 のように深さ方向（上下方向）への Dropout レイヤの挿入を考えます。

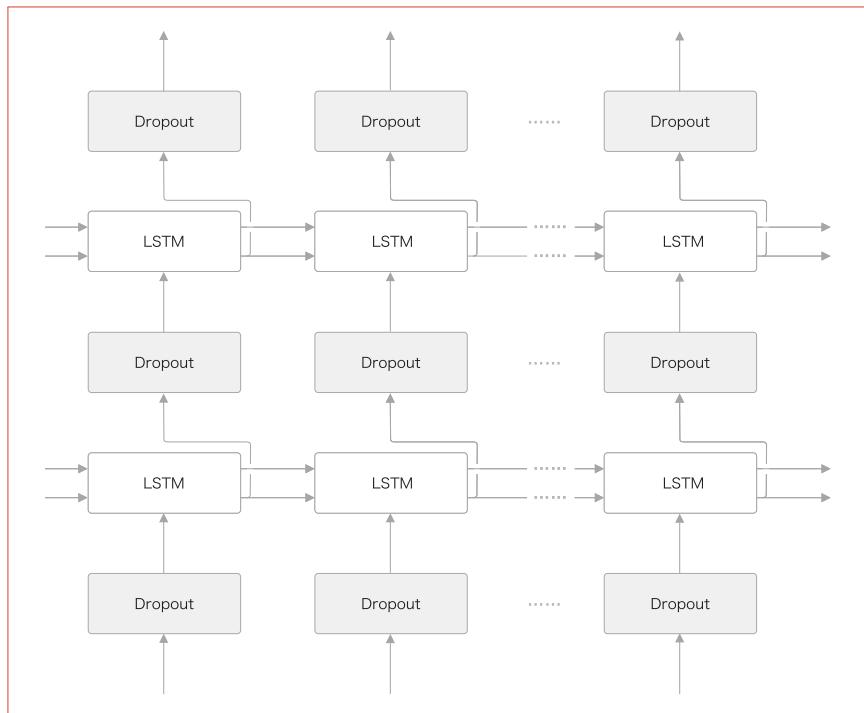


図6-33 良い例：Dropout レイヤを深さ方向（上下方向）に挿入

こちらのケースは、時間方向（左右方向）にどれだけ進んでも情報は失われません。Dropout は時間軸とは独立して、深さ方向（上下方向）にだけ有効に働きます。



図6-31 と図6-33 を見比べてみましょう。図6-31 の例では、フィードフォワード型ニューラルネットワークに Dropout を使った場合を示しましたが、これは深さ方向に Dropout を適用しています。これと同じ要領で、図6-33 でも深さ方向にだけ Dropout を適用することで、フィードフォワードのときと同じく、過学習の抑制を期待できます。

ここまで話してきたとおり、「通常の Dropout」は時間方向には適していません。しかし最近の研究では RNN の時間方向の正則化を目的として、さまざまな手法が提案されています。たとえば、文献[36]では「変分 Dropout (variational dropout)」が提案されており、時間方向への適用に成功しています。

変分 Dropout は深さ方向に加えて時間方向にも利用でき、言語モデルの精度向上をさらに押し上げます。その仕組みは、図6-34 に示すように、同じ階層にある Dropout では共通のマスクを利用するというものです。ここで言う「マスク」とは、データを「通す/通さない」を決める二値のランダムパターンを指します。

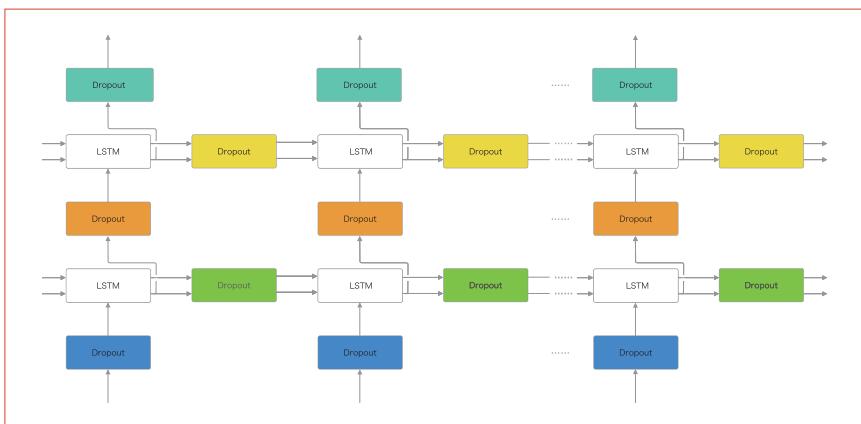


図6-34 変分 Dropout の例：同じ図柄の Dropout は共通のマスクを利用する。このように同じ階層にある Dropout では共通のマスクを利用することで、時間方向への Dropout も有効に働く

図6-34 に示すように、同じ階層にある Dropout ではマスクを共有することで、マスクは“固定”されることになります。それによって、情報の失われ方も“固定された”ものになるので、通常の Dropout のときのような指数的な情報の損失を避けることができます。



変分 Dropout は、通常の Dropout よりも良い結果になることが報告されています。ただし本章では変分 Dropout は用いず、通常の Dropout を用います。変分 Dropout のアイデアはとてもシンプルなので、興味のある方は自分で実装してみましょう！

6.5.3 重み共有

言語モデルを改良するとしても簡単なトリックに**重み共有**（weight tying）があります[37][38]。weight tyingは直訳すると「重みを結びつける」ということになりますが、これは図6-35のように重みを共有することを意味します。

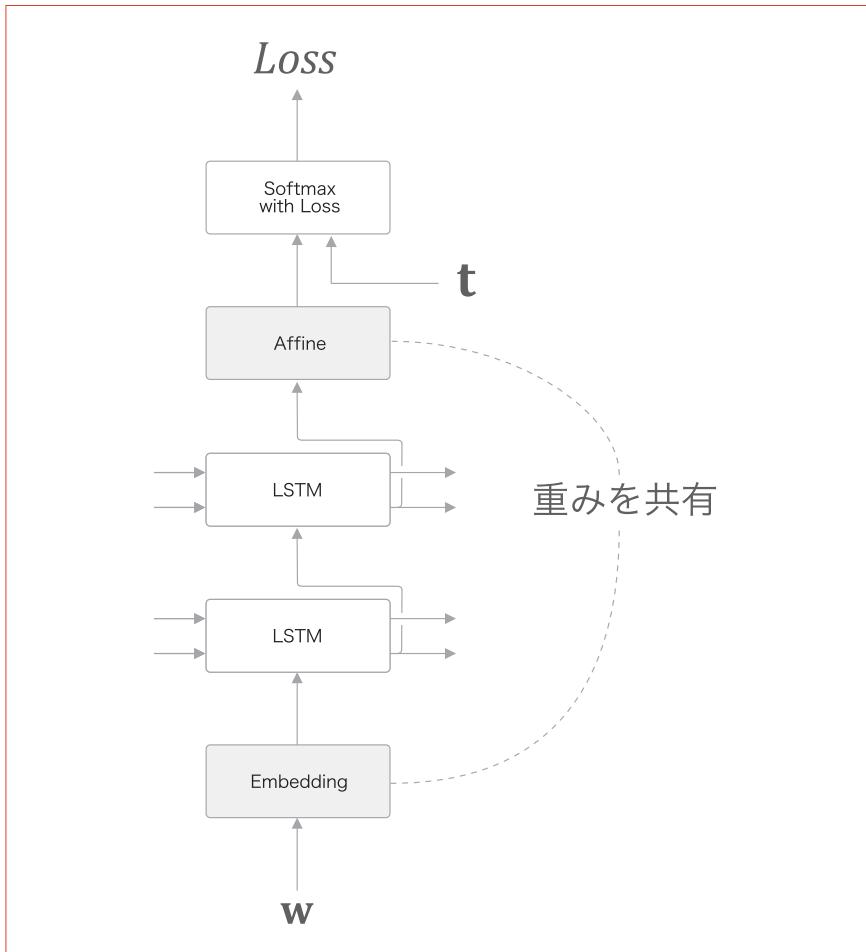


図6-35 言語モデルにおける重み共有の例：Embedding レイヤと Softmax の前の Affine レイヤで重みを共有する

図6-35 で示すように、Embedding レイヤの重みと Affine レイヤの重みを結びつける（共有する）テクニックが重み共有です。2つのレイヤで重みを共有することで、学習するパラメータを大きく減らすことができます。それでいて、精度を向上できるという、まさに一石二鳥のテクニックです！

それでは、重み共有を実装する視点で考えてみましょう。ここで語彙数を V として、LSTM の隠れ状態の次元数を H とします。そうすると、Embedding レイヤの重みの形状は $(V \times H)$ であり、Affine レイヤの重みの形状は $(H \times V)$ になります。このとき重み共有を適用するには、Embedding レイヤの重みの転置を Affine レイヤの重みに設定するだけになります。そして、このとても簡単なトリックによって、素晴らしい結果がもたらされます！



なぜ重み共有は有効なのでしょうか？直感的には、重みを共有することで、学習すべきパラメータ数を減らすことができ、それによって学習を容易にできるからと考えられます。さらに、パラメータ数が減ることで、過学習を抑制できる恩恵も得られます。なお、重み共有の有用性の根拠については、論文 [38] で理論的に述べられています。興味のある方は参考にしてください。

6.5.4 より良い RNNLM の実装

これまで RNNLM の改善点を 3 つ説明しました。それでは、それらのテクニックがどの程度効果があるのかを見てみたいと思います。ここでは `BetterRnnlm` クラスとして、**図6-36** のレイヤ構成で実装します。

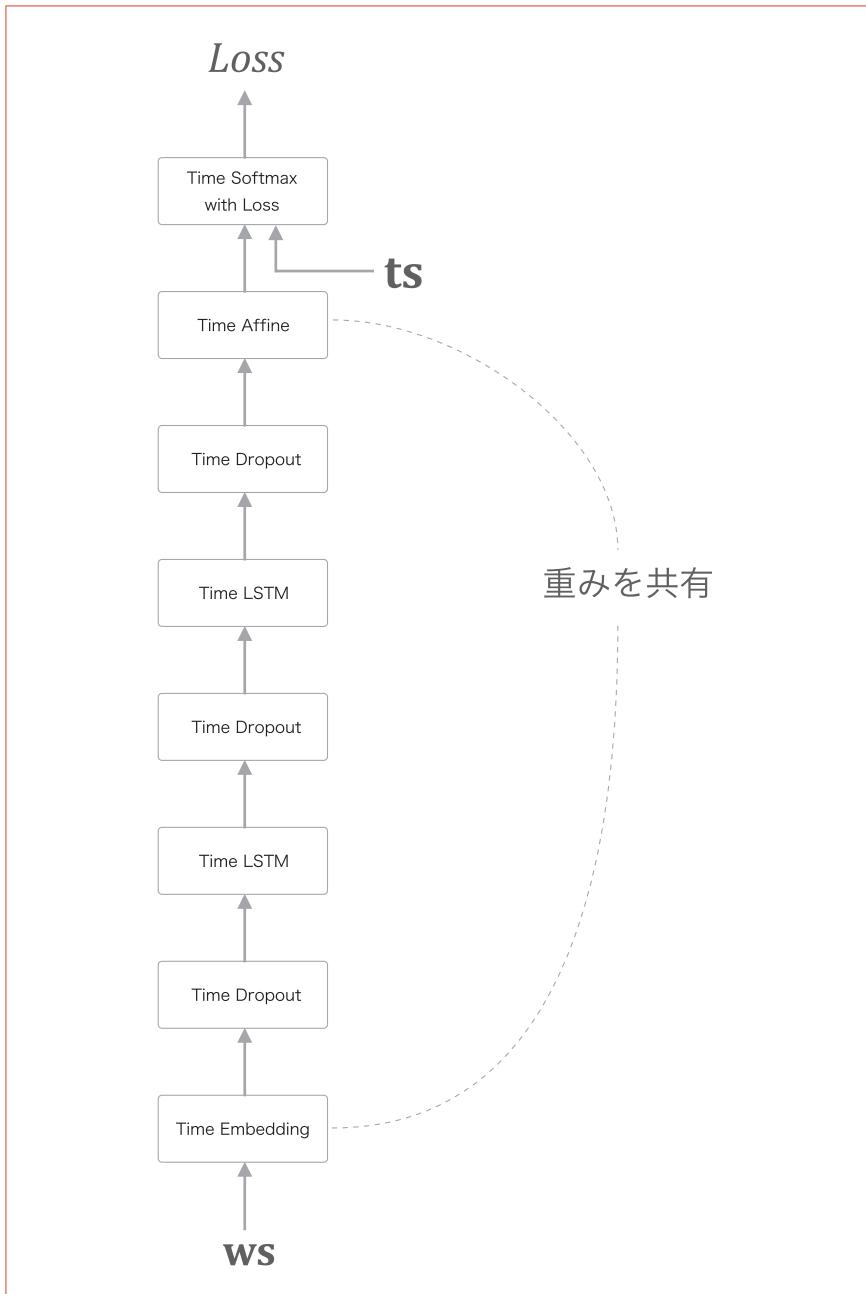


図6-36 BetterRnnlm クラスのネットワーク構成

図6-36 のとおり、ここで改善ポイントは次の3つです。

- LSTM レイヤの多層化（ここでは2層）
- Dropout を使用（深さ方向にのみ適用）
- 重み共有（Embedding レイヤと Affine レイヤで重み共有）

それでは、この3つの改善点を取り入れた `BetterRnnlm` クラスを実装します。これは次のように実装できます（☞ ch06/better_rnnlm.py）。

```
import sys
sys.path.append('..')
from common.time_layers import *
from common.np import *
from common.base_model import BaseModel

class BetterRnnlm(BaseModel):
    def __init__(self, vocab_size=10000, wordvec_size=650,
                 hidden_size=650, dropout_ratio=0.5):
        V, D, H = vocab_size, wordvec_size, hidden_size
        rn = np.random.randn

        embed_W = (rn(V, D) / 100).astype('f')
        lstm_Wx1 = (rn(D, 4 * H) / np.sqrt(D)).astype('f')
        lstm_Wh1 = (rn(H, 4 * H) / np.sqrt(H)).astype('f')
        lstm_b1 = np.zeros(4 * H).astype('f')
        lstm_Wx2 = (rn(H, 4 * H) / np.sqrt(H)).astype('f')
        lstm_Wh2 = (rn(H, 4 * H) / np.sqrt(H)).astype('f')
        lstm_b2 = np.zeros(4 * H).astype('f')
        affine_b = np.zeros(V).astype('f')

        # 3つの改善！
        self.layers = [
            TimeEmbedding(embed_W),
            TimeDropout(dropout_ratio),
            TimeLSTM(lstm_Wx1, lstm_Wh1, lstm_b1, stateful=True),
            TimeDropout(dropout_ratio),
            TimeLSTM(lstm_Wx2, lstm_Wh2, lstm_b2, stateful=True),
            TimeDropout(dropout_ratio),
            TimeAffine(embed_W.T, affine_b) # 重み共有!!
        ]
        self.loss_layer = TimeSoftmaxWithLoss()
        self.lstm_layers = [self.layers[2], self.layers[4]]
        self.drop_layers = [self.layers[1], self.layers[3]],
        ↵ self.layers[5]
```

```

    self.params, self.grads = [], []
    for layer in self.layers:
        self.params += layer.params
        self.grads += layer.grads

def predict(self, xs, train_flg=False):
    for layer in self.drop_layers:
        layer.train_flg = train_flg
    for layer in self.layers:
        xs = layer.forward(xs)
    return xs

def forward(self, xs, ts, train_flg=True):
    score = self.predict(xs, train_flg)
    loss = self.loss_layer.forward(score, ts)
    return loss

def backward(self, dout=1):
    dout = self.loss_layer.backward(dout)
    for layer in reversed(self.layers):
        dout = layer.backward(dout)
    return dout

def reset_state(self):
    for layer in self.lstm_layers:
        layer.reset_state()

```

ここではコードの背景がグレーの箇所において、先に述べた 3 つの改善が行われています。具体的には、Time LSTM レイヤを 2 つ重ね、Time Dropout レイヤを用います。そして、Time Embedding レイヤと Time Affine レイヤで重みを共有します。

それでは、この改善された `BetterRnnlm` クラスを学習させましょう。その前に、これから行う学習コードにおいてもひとつだけ工夫を加えることにします。その工夫とは、エポックごとに検証データでパープレキシティを評価し、その値が悪くなつた場合にのみ学習係数を下げるというものです。これは実践的によく用いられるテクニックで、多くの場合良い結果が得られます。なお、ここでの実装は PyTorch の言語モデルの実装例[39]を参考にしています。学習用のコードを次に示します ([ch06/train_better_rnnlm.py](#))。

```

import sys
sys.path.append('..')
from common import config
# GPUで実行する場合は下記のコメントアウトを消去（要cupy）

```

```
# =====
# config.GPU = True
# =====
from common.optimizer import SGD
from common.trainer import RnnlmTrainer
from common.util import eval_perplexity
from dataset import ptb
from better_rnnlm import BetterRnnlm

# ハイパーパラメータの設定
batch_size = 20
wordvec_size = 650
hidden_size = 650
time_size = 35
lr = 20.0
max_epoch = 40
max_grad = 0.25
dropout = 0.5

# 学習データの読み込み
corpus, word_to_id, id_to_word = ptb.load_data('train')
corpus_val, _, _ = ptb.load_data('val')
corpus_test, _, _ = ptb.load_data('test')

vocab_size = len(word_to_id)
xs = corpus[:-1]
ts = corpus[1:]

model = BetterRnnlm(vocab_size, wordvec_size, hidden_size, dropout)
optimizer = SGD(lr)
trainer = RnnlmTrainer(model, optimizer)

best_ppl = float('inf')
for epoch in range(max_epoch):
    trainer.fit(xs, ts, max_epoch=1, batch_size=batch_size,
                time_size=time_size, max_grad=max_grad)

    model.reset_state()
    ppl = eval_perplexity(model, corpus_val)
    print('valid perplexity: ', ppl)

    if best_ppl > ppl:
        best_ppl = ppl
        model.save_params()
    else:
        lr /= 4.0
        optimizer.lr = lr
```

```
model.reset_state()
print('-' * 50)
```

ここでは学習のエポックごとに検証データでパープレキシティを評価し、それがこれまでのパープレキシティ（`best_ppl`）を下回った場合、学習係数を $1/4$ にします。それを行うために、ここでは 1 エポック分の学習を `RnnlmTrainer` クラスの `fit()` メソッドで行い、その後で検証データでのパープレキシティの評価を行うという処理を `for` 文で繰り返し行います。それではこの学習用のコードを実行してみましょう。



この学習にはかなりの時間を要します。CPU で実行した場合は 2 日程度の時間が必要でしょう。一方 GPU を使えば 5 時間ほどで完了します（GPU で実行する場合は、ファイル先頭のインポート文にある「`# config.GPU = True`」のコメントアウトを外します）。なお、学習済みの重みは下記 URL から入手可能です。

<https://www.oreilly.co.jp/pub/9784873118369/BetterRnnlm.pkl>

上のコードを実行すると、順調にパープレキシティが下がっていきます。そして、最終的にテストデータでのパープレキシティは 75.76 という結果が得られます（この結果は実行ごとに異なります）。改善前の RNNLM のパープレキシティがおよそ 136 だったことを考えると、これはかなりの精度向上と言えるでしょう。LSTM の多層化で表現力を高め、Dropout で汎用性を向上させ、重み共有で重みを効率的に利用したことでのこのような大幅な精度向上を達成できたのです！

6.5.5 最先端の研究へ

これで私たちの RNNLM の改良は終わりです。私たちは、RNNLM に対していくつかの改善を行い、かなりの精度向上を達成できました。PTB データセットのテストデータのパープレキシティが 75 程度というのは、まずはこの結果と言えるでしょう。しかし、最先端の研究はさらに先へ進んでいます。ここでは最新の研究の結果だけを簡単に紹介したいと思います。それでは図 6-37 を見てみましょう。

| Model | Parameters | Validation | Test |
|---|-----------------|----------------|----------------|
| Mikolov & Zweig (2012) - KN-5 | 2M [†] | — | 141.2 |
| Mikolov & Zweig (2012) - KN5 + cache | 2M [†] | — | 125.7 |
| Mikolov & Zweig (2012) - RNN | 6M [†] | — | 124.7 |
| Mikolov & Zweig (2012) - RNN-LDA | 7M [†] | — | 113.7 |
| Mikolov & Zweig (2012) - RNN-LDA + KN-5 + cache | 9M [†] | — | 92.0 |
| Zaremba et al. (2014) - LSTM (medium) | 20M | 86.2 | 82.7 |
| Zaremba et al. (2014) - LSTM (large) | 66M | 82.2 | 78.4 |
| Gal & Ghahramani (2016) - Variational LSTM (medium) | 20M | 81.9 ± 0.2 | 79.7 ± 0.1 |
| Gal & Ghahramani (2016) - Variational LSTM (medium, MC) | 20M | — | 78.6 ± 0.1 |
| Gal & Ghahramani (2016) - Variational LSTM (large) | 66M | 77.9 ± 0.3 | 75.2 ± 0.2 |
| Gal & Ghahramani (2016) - Variational LSTM (large, MC) | 66M | — | 73.4 ± 0.0 |
| Kim et al. (2016) - CharCNN | 19M | — | 78.9 |
| Merity et al. (2016) - Pointer Sentinel-LSTM | 21M | 72.4 | 70.9 |
| Grave et al. (2016) - LSTM | — | — | 82.3 |
| Grave et al. (2016) - LSTM + continuous cache pointer | — | — | 72.1 |
| Inan et al. (2016) - Variational LSTM (tied) + augmented loss | 24M | 75.7 | 73.2 |
| Inan et al. (2016) - Variational LSTM (tied) + augmented loss | 51M | 71.1 | 68.5 |
| Zilly et al. (2016) - Variational RHN (tied) | 23M | 67.9 | 65.4 |
| Zoph & Le (2016) - NAS Cell (tied) | 25M | — | 64.0 |
| Zoph & Le (2016) - NAS Cell (tied) | 54M | — | 62.4 |
| Melis et al. (2017) - 4-layer skip connection LSTM (tied) | 24M | 60.9 | 58.3 |
| AWD-LSTM - 3-layer LSTM (tied) | 24M | 60.0 | 57.3 |
| AWD-LSTM - 3-layer LSTM (tied) + continuous cache pointer | 24M | 53.9 | 52.8 |

図6-37 PTB データセットに対する各モデルのパープレキシティの結果（文献[34]より抜粋）。表の「Parameters」はパラメータの総数、「Validation」は検証データに対するパープレキシティ、「Test」はテストデータに対するパープレキシティを表す

図6-37 は文献[34]から抜粋したものです。この表では、歴代ベストな言語モデルのPTB データセットに対するパープレキシティの結果がまとめられています。表の「Test」の列に注目すると、新しい手法が提案されるに従い、パープレキシティが下がっていることが分かります。そして、最後の行では 52.8 という結果になっています。実際のところ、この 52.8 という数字はとても素晴らしい結果であり、PTB データセットのパープレキシティが 50 に近づくということは、数年前までは考えられませんでした。

ここでは最先端の研究の結果だけを示しました。もちろん、私たちのモデルとはかなりの開きがあります。しかし大切なことは、図6-37 の最先端のモデルと私たちのモデルには、多くの共通点があるということです。たとえば、最先端のモデルにおいても、多層の LSTM を使用したモデルが使われています。そして、Dropout ベースの正則化を行い——実際には、変分 Dropout と DropConnect^{†2}——、重み共有を行っています。その上で、さらに最適化や正則化に関するいくつかのテクニックが使

†2 DropConnect とは重み自体をランダムに無視する手法です。

われ、また、ハイパーパラメータのチューニングもかなり厳密に行ってています。それによって、52.8 という驚異的な数値を達成しています。



図6-37 のモデル名には「AWD-LSTM 3-layer LSTM (tied) + continuous cache pointer」とあります。この continuous cache pointer という技術は、8 章で詳しく学ぶ Attention をベースとしたものです。Attention はとても重要な技術であり、さまざまな用途で使われます。言語モデルというタスクにおいても精度向上に大きく貢献しています。少し先になりますが、8 章の Attention を楽しみに待ちましょう！

6.6 まとめ

本章では、ゲート付き RNN を見てきました。前章の単純な RNN では勾配消失（または勾配爆発）が問題になることを指摘し、それに代わるレイヤとして、ゲート付きの RNN——具体的には、LSTM や GRU など——が有効であることを説明しました。それらのレイヤにはゲートという仕組みが使われており、それがデータと勾配の流れをうまくコントロールすることを可能にしました。

また、本章では LSTM レイヤを使用し、言語モデルを作成しました。そして、PTB データセットを対象に学習を行い、パープレキシティの評価を行いました。さらに、LSTM の多層化や Dropout、重み共有などのテクニックを利用して、大幅な精度向上にも成功しました。これらのテクニックは、2017 年の最先端の研究でも実際に使われています。

次章では、言語モデルを使って文章生成を行います。そして、機械翻訳のように、ある言語を別の言語へと変換するモデルを詳しく見ていきます。

本章で学んだこと

- 単純な RNN の学習では、勾配消失・勾配爆発が問題になる
- 勾配爆発には勾配クリッピング、勾配消失には LSTM や GRU などのゲート付き RNN が有効である
- LSTM には、input ゲート、forget ゲート、output ゲートの 3 つのゲートがある
- ゲートには専用の重みがあり、sigmoid 関数を使って 0.0 から 1.0 までの実数を出力する
- 言語モデルの改善には、LSTM レイヤの多層化、Dropout、重み共有などのテクニックが有効である
- RNN の正則化は重要なテーマであり、Dropout ベースのさまざまな手法が提案されている

7章 RNNによる文章生成

完璧な文章などといったものは存在しない。

完璧な絶望が存在しないようにね。

——村上春樹『風の歌を聴け』

5章と6章では、RNNとLSTMについて、その仕組みや実装について詳しく見てきました。いまや私たちは、それらを実装レベルで理解しています。本章では、これまでの成果——RNNやLSTM——が花開きます。ここでは、LSTMを利用して楽しいアプリケーションをいくつか実装します。

本章ではまず初めに、言語モデルを使って「文章生成」を行います。具体的には、コーパスを使って学習した言語モデルを用いて、新しい文章を作成します。さらに、改良した言語モデルを用いることで、より自然な文章が生成できることを見ていきましょう。ここでの作業を通じて、「AIに文章を書かせる」ということを（簡単にでも）実感できるでしょう。

また、本章ではseq2seqという新しい構造のニューラルネットワークを扱います。seq2seqとは「(from) sequence to sequence (時系列から時系列へ)」を意味する言葉で、ある時系列データを別の時系列データへと変換するものです。本章では、2つのRNNを組み合わせることで、いとも簡単にseq2seqが実装できることを見ていきます。このseq2seqは、機械翻訳やチャットボット、メールの自動返信など、さまざまなアプリケーションで利用することができます。このシンプルでありながら賢くパワフルなseq2seqを理解することで、ディープラーニングの可能性はさらに広がるはずです！

7.1 言語モデルを使った文章生成

これまで数章にわたって言語モデルを扱ってきました。前にも述べたとおり、言語モデルはさまざまなアプリケーションで利用できます。代表例としては機械翻訳や音声認識、そして文章生成などが挙げられます。ここでは、言語モデルに文章生成を行わせます。

7.1.1 RNNによる文章生成の手順

前章では、LSTM レイヤを利用して言語モデルを実装しました。そこで実装した言語モデルは図 7-1 に示すようなネットワーク構成でした。ちなみに、私たちは時系列データを (T 個分だけ) まとめて処理するレイヤを Time LSTM や Time Affine レイヤとして実装してきました。

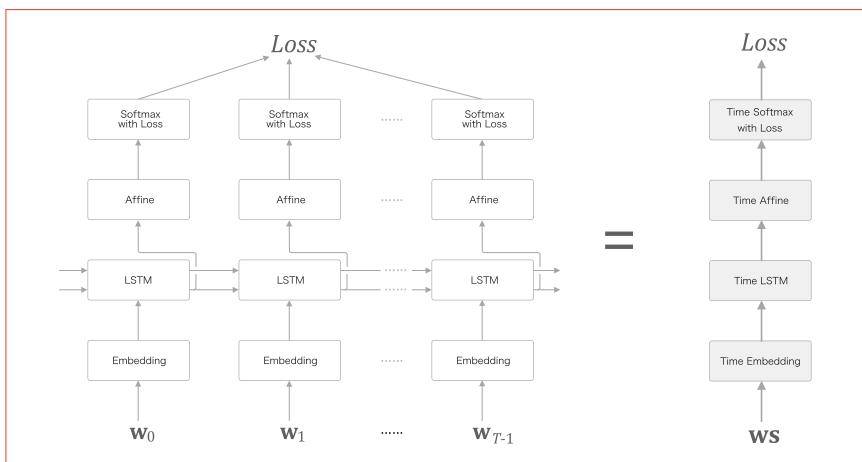


図 7-1 前章で実装した言語モデル：右図は時系列データをまとめて処理する Time レイヤを使用、左図はその展開後のレイヤ構成

それでは、言語モデルに文章を生成させる手順について説明します。ここでは説明のための例として「you say goodbye and I say hello .」というコーパスで学習を行った言語モデルを考えます。そして、この学習済みの言語モデルに対して、「I」という単語を与える場合を考えます。このとき、その言語モデルは図 7-2 のような確率分布を出力するとします。

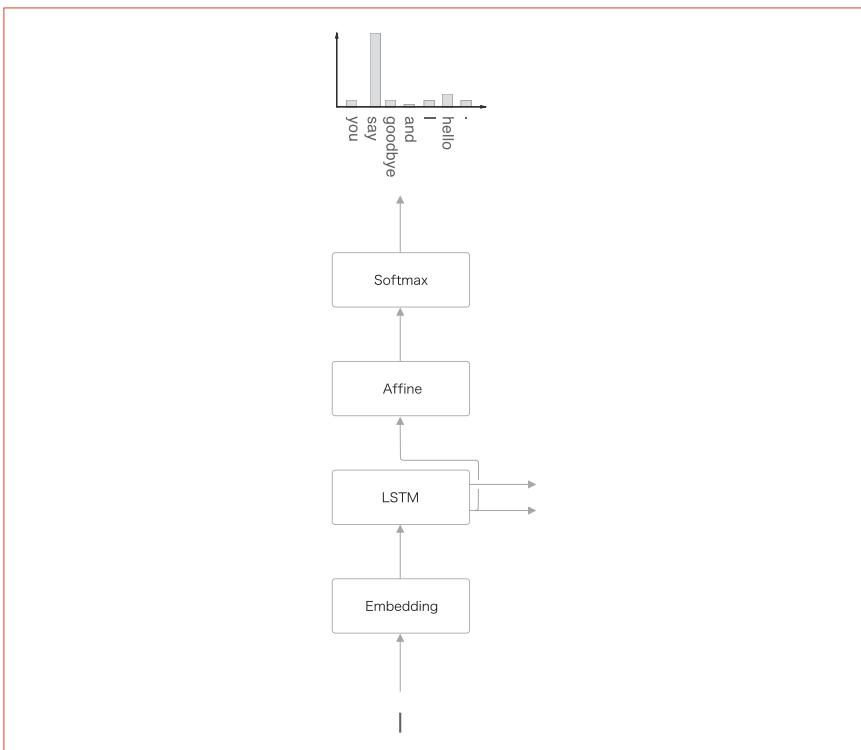


図 7-2 言語モデルは、次に出現する単語の確率分布を出力する

言語モデルは、これまでに与えられた単語から、次に出現する単語の確率分布を出力します。図 7-2 の例では、「I」という単語を与えたときに、次に出現する単語の確率分布を出力しています。それでは次の単語を新たに生成するにはどうしたらよいでしょうか？

考えられる方法のひとつは、最も確率の高い単語を選ぶことです。その場合、最も確率の高い単語を選ぶだけなので、結果は一意に決まります。つまり、これは「決定的」な方法です。また別の方法としては「確率的」に選ぶことが考えられます。確率分布に従って選ぶことで、確率の高い単語は選ばれやすく、確率の低い単語は選ばれにくくなります。このとき、選ばれる単語（サンプリングされる単語）は毎回異なります。

ここで私たちは、生成される文章が毎回変動するようにしたいと思います。そのほうが、生成される文章にバリエーションが出ておもしろいでしょう。そこで、後者の

方法——確率的に選ぶ方法——で単語を選ぶことにします。私たちの例に話を戻すと、このとき図7-3のように「say」という単語が（確率的に）選ばれたとします。

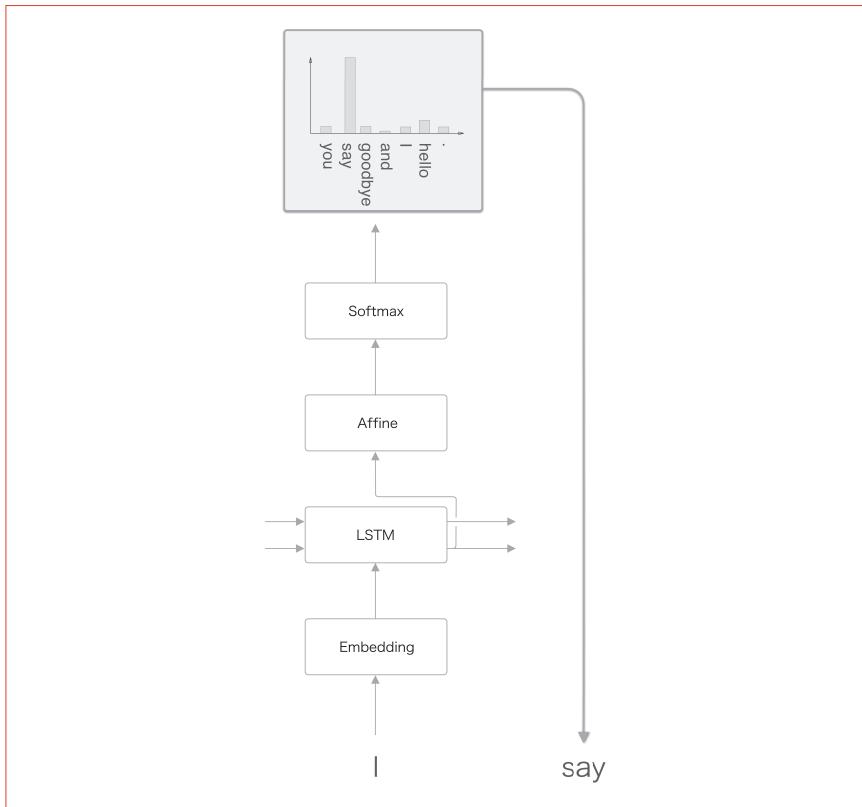


図7-3 確率分布から単語をひとつサンプリングする

図7-3では、確率分布からサンプリングを行い、その結果が「say」になっている例が示されています。というのも、図7-3の確率分布では「say」の確率が一番高いので、それがサンプリングされる確率が最も高いのです。ただし、ここで「say」が選ばれることは必然ではなく（「決定的」ではなく）、「確率的」に決められることに注意しましょう。そのため、「say」以外の単語も、各単語の出現確率に応じてサンプリングされることになります。



決定的とは（アルゴリズムの）結果が一意に決まること、結果が予測可能なことを言います。たとえば、上の例において最も確率の高い単語を選ぶようにすれば、それは「決定的」なアルゴリズムになります。一方、「確率的」なアルゴリズムでは、確率的に結果が決まります。そのため、選ばれる単語は試行のたびに変わります（もしくは変わる可能性があります）。

それでは続いて、2つ目の単語のサンプリングを行います。これは、先ほど行った手順を繰り返すだけです。つまり、先ほど生成した「say」という単語を言語モデルに入力し、単語の確率分布を得ます。そして、その確率分布から次に出現する単語をサンプリングするのです。この作業を図で表すと図7-4 のようになります。

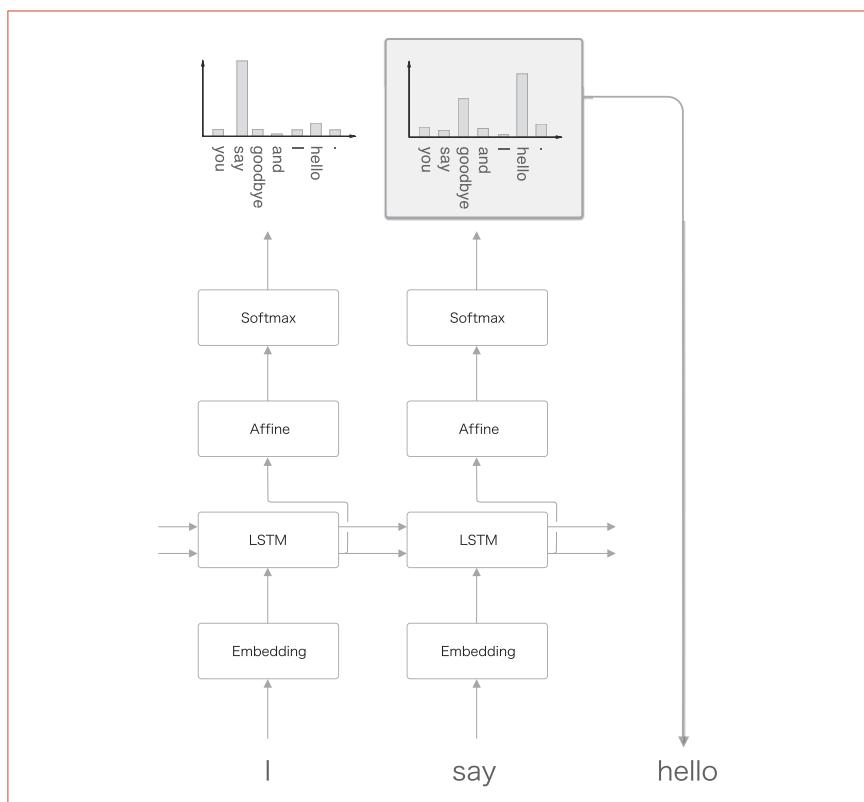


図 7-4 確率分布の出力およびサンプリングを繰り返す

後は、この作業を望む回数だけ繰り返します（もしくは、<eos>のような文末記号が出現するまで繰り返します）。そうすれば、新しい文章を生成することができます。

ここで注目すべきは、上のようにして新たに生成した文章は、訓練データには存在しない、新しく生成された文章であるということです。なぜなら、言語モデルは訓練データを丸暗記したのではなく、そこで使われる単語の並び方のパターンを学習しているからです。もし言語モデルがコーパスを使って単語の出現パターンを正しく学習できているのであれば、言語モデルが新たに生成する文章は、私たち人間にとっても自然な文章であり、意味の通る文章が期待できます。

7.1.2 文章生成の実装

それでは、文章生成を行うための実装を行いましょう。ここでは、前章で実装した Rnnlm クラス ([ch06/rnnlm.py](#)) をベースに、それを継承して RnnlmGen クラスを作成します。このクラスに文章生成を行うメソッドを追加します。



クラスの継承とは、既存のクラスを引き継ぎ、新しいクラスを作ることをいいます。Python でクラスの継承を行うには、`class New(Base):` のように書きます。これによって、Base クラスを継承して新たに New クラスを実装することができます。

それでは、RnnlmGen クラスの実装を次に示します ([ch07/rnnlm_gen.py](#))。

```
import sys
sys.path.append('..')
import numpy as np
from common.functions import softmax
from ch06.rnnlm import Rnnlm
from ch06.better_rnnlm import BetterRnnlm

class RnnlmGen(Rnnlm):
    def generate(self, start_id, skip_ids=None, sample_size=100):
        word_ids = [start_id]

        x = start_id
        while len(word_ids) < sample_size:
            x = np.array(x).reshape(1, 1)
            score = self.predict(x)
            p = softmax(score.flatten())

            sampled = np.random.choice(len(p), size=1, p=p)
            word_ids.append(sampled)

        return word_ids
```

```

if (skip_ids is None) or (sampled not in skip_ids):
    x = sampled
    word_ids.append(int(x))

return word_ids

```

このクラスで文章の生成を行うのが、`generate(start_id, skip_ids, sample_size)` です。ここで引数の `start_id` は最初に与える単語 ID、`sample_size` はサンプリングする単語の数を表します。また、引数の `skip_ids` は単語 ID のリストを想定し（たとえば、`[12, 20]`）、そこで指定された単語 ID がサンプリングされないようにします。これは PTB データセットにある `<unk>` や `N` などの前処理された単語をサンプリングしないようにする用途で使います。



PTB データセットでは、元の文章に対して前処理が行われており、レアな単語は `<unk>` に、数字は `N` に置き換えられています。また、私たちの場合、文章の区切りは `<eos>` という文字列を使用しています。

`generate()` のメソッドの中では、まず初めに `model.predict(x)` によって各単語のスコアを出力します（スコアとは正規化される前の値）。そして、`p = softmax(score)` によって、そのスコアを Softmax 関数で正規化します。これで目的とする確率分布を得ることができました。後は、その確率分布 `p` から次の単語をサンプリングします。なお、確率分布からのサンプリングには `np.random.choice()` を使います。この関数の使い方は、「4.2.6 Negative Sampling のサンプリング手法」で説明しました。



`model` の `predict()` メソッドはミニバッチ処理を行うので、入力 `x` は 2 次元配列にする必要があります。そのため、単語 ID をひとつだけ入力する場合でも、バッチサイズを 1 と考えて、 1×1 の NumPy 配列に整形します。

それでは、この `RnnlmGen` クラスを使って、文章生成を行わせましょう。ここでは何も学習を行っていない状態で——つまり重みパラメータはランダムな初期値の状態で——、文章生成を行わせます。文章生成のためのコードを次に示します（[ch07/generate_text.py](#)）。

```

import sys
sys.path.append('..')

```

```

from rnnlm_gen import RnnlmGen
from dataset import ptb

corpus, word_to_id, id_to_word = ptb.load_data('train')
vocab_size = len(word_to_id)
corpus_size = len(corpus)

model = RnnlmGen()
# model.load_params('../ch06/Rnnlm.pkl')

# start文字とskip文字の設定
start_word = 'you'
start_id = word_to_id[start_word]
skip_words = ['N', '<unk>', '$']
skip_ids = [word_to_id[w] for w in skip_words]

# 文章生成
word_ids = model.generate(start_id, skip_ids)
txt = ' '.join([id_to_word[i] for i in word_ids])
txt = txt.replace('<eos>', '\n')
print(txt)

```

ここでは先頭の単語を `you` とし、その単語 ID を `start_id` にした上で文章生成を行います。また、サンプリングしない単語として `['N', '<unk>', '$']` を指定します。なお、文章生成を行う `generate()` メソッドは、単語 ID を配列として返します。そのため、その単語 ID の配列を文章に変換する必要がありますが、ここでは `txt = ' '.join([id_to_word[i] for i in word_ids])` という 1 文で、その変換を行います。`join()` メソッドは、「'区切り文字'.join(リスト)」のように書くことで、単語を連結します。具体例を示すと次のようになります。

```
>>> ' '.join(['you', 'say', 'goodbye'])
'you say goodbye'
```

それでは、上のコードを実行してみましょう。結果は次のようになります。

```

you setback best raised fill steelworkers montgomery kohlberg told beam
worthy allied ban swedish aichi mather promptly ramada explicit leslie
bets discovery considering campaigns bottom petrie warm large-scale
frequent temple grumman bennett ...

```

見てのとおり、出力される文章はでたらめな単語の羅列です。当然ながら、ここでモデルの重みはランダムな初期値のため、意味の通じない文章が出力されます。それでは学習を行った言語モデルではどうでしょうか？ ここでは、前章

で学習済みの重みを利用して、文章の生成を行わせます。そのためには、`model.load_params('..../ch06/Rnnlm.pkl')`によって、前章で学習した重みパラメータを読み込んでから文章生成を行います。生成される文章を見てみましょう（結果は毎回異なります）。

```
you 'll include one of them a good problems.  
moreover so if not gene 's corr experience with the heat of bridges a  
new deficits model is non-violent what it 's a rule must exploit it.  
there 's no tires industry could occur.  
beyond my hours where he is n't going home says and japanese letter.  
knight transplants d.c. turmoil with one-third of voters.  
the justice department is ...
```

上の結果を見ると、文法的におかしい文章や意味の通じない文章がいくつか見られます、それらしい文章に読める箇所もあります。詳しく見ていくと「you 'll include ...」や「there 's no tires ...」、「knight transplants ...」など、主語と動詞の組み合わせを正しく生成しています。また、「good problems」や「japanese letter」のように、形容詞と名詞の使い方もある程度理解しているようです。また、最初の「you 'll include one of them a good problems.（そのうちのひとつに良い問題を含めるでしょう。）」は、意味的に正しい文章です。

このように、上の試行で生成された文章は、ある程度は正しい文章と言えそうです。しかし、不自然な文章も散見され、まだまだ改善の余地はあります。“完璧な文章”は存在しませんが、より自然な文章を私たちは求めるべきです。そのためにはどうすべきでしょうか？ それには、より良い言語モデルを使います！

7.1.3 さらに良い文章へ

良い言語モデルがあれば、良い文章が期待できます。前章では、単純な RNNLM を改良し、「より良い RNNLM」を実装しました。そこで私たちは、パープレキシティがおよそ 136 だったモデルを 75 までに改良したのです！ それでは、「より良い RNNLM」の文章生成の実力を見てみましょう。



前章では `BetterRnnlm` クラスの学習を行い、学習後の重みをファイルとして保存しました。ここでの実験は、その学習後の重みファイルが必要です。なお学習済みの重みファイルは、下記 URL からも取得することができます。その重みファイルを本書のソースコードの `ch06` ディレクトリに配置することで、ここでの実験コード `ch07/generate_better_text.py` を実行できます。

<https://www.oreilly.co.jp/pub/9784873118369/BetterRnnlm.pkl>

前章では、より良い言語モデルを `BetterRnnlm` クラスとして実装しました。ここでは、先ほど行ったように、このクラスを継承して文章を生成させる機能を持たせます。その実装は、先ほど `RnnlmGen` クラスで行った実装とまったく同じであるため、説明は省略します。

それでは、より良い言語モデルに文章生成を行わせてみましょう。前と同じように先頭文字に `you` を与えてみます。すると、次のような文章が生成されます（☞ `ch07/generate_better_text.py`）。

```
you 've seen two families and the women and two other women of students.
the principles of investors that prompted a bipartisan rule of which
had a withdrawn target of black men or legislators interfere with the
number of plants can do to carry it together.
the appeal was to deny steady increases in the operation of dna and
educational damage in the 1950s.
...
...
```

この結果を見ると、（やや主観的にはなりますが）前回よりも自然な文章を生成しているようです。最初の文章では「`you 've seen two families and the women ...`」とあり、主語、動詞、目的語を正しく使っています。さらに、`and` の使い方（「2つの家族とその女性」）もうまく学習しています。他の文章を読んでも、全体的にまずまずの結果と言えるかもしれません。

ここで生成された文章にはまだいくつか問題がありますが——特に文章の意味的な点ではまだまだですが——、私たちの「より良い言語モデル」は、（ある程度）自然な文章を自由に生成しているようです。きっと、このモデルにさらに改良を加え、さらに大きなコーパスを使うことで、より自然な文章を紡ぎ出してくれることでしょう。

最後に、私たちの「より良い言語モデル」に「`the meaning of life is`」という文字を与えてみたいと思います。そして、それに続く言葉を生成してもらいます（これは論文[35]で行われた実験です）。この実験を行うには、私たちのモデルに `['the', 'meaning', 'of', 'life']` という単語を順次与えて順伝播を行います。

このとき出力される結果は何も使用しませんが、LSTM レイヤにはその単語列の情報が保持されることになります。そして、`is` を先頭文字として文章生成を開始させることで、「the meaning of life is」に続く文章を生成させることができます。

この実験は `ch07/generate_better_text.py` で実行できます。実行ごとに生成される文章は異なりますが、ここではその中からおもしろい（興味深い）結果をひとつだけ紹介したいと思います。

```
the meaning of life is not a good version of paintings.
```

上のように、私たちの言語モデルは「人生の意味は、良い状態の絵画ではない」と言っています。「良い状態の絵画」……それが何を指すのか分かりませんが、これはこれで何か深い意味があるのかもしれません。

7.2 seq2seq

世の中には時系列データが溢れています。言語データや音声データ、そして動画像データはすべて時系列データです。そして、それらの時系列データを別の時系列データに変換するような問題も数多く考えられます。たとえば、機械翻訳や音声認識といったものが挙げられます。その他にも、チャットボットのように対話をを行うアプリケーションや、コンパイラのようにソースコードを機械語に変換するようなタスクも考えられます。

このように、入力と出力が時系列データであるような問題は、世の中に数多く存在します。これから私たちは、時系列データを別の時系列データに変換するモデルを考えます。そのための手法として、ここでは 2 つの RNN を利用する **seq2seq** (sequence to sequence) という手法を見ていきます。

7.2.1 seq2seq の原理

seq2seq は Encoder-Decoder モデルとも呼ばれます。その名前が示すとおり、そこには 2 つのモジュール——Encoder と Decoder——が登場します。文字どおり、Encoder は入力データをエンコードし、Decoder はエンコードされたデータをデコードします。



エンコード（符号化）とは、情報をある規則に基づいて変換することです。たとえば文字コードを例に出すと、「A」という文字を「1000001」（2進数）に変換することがエンコードの例として挙げられます。一方、デコード（復号）とは、エンコードされた情報を元の情報に戻すことです。文字コードの例で言えば、「1000001」というビットパターンを「A」という文字に変換することに相当します。

それでは、seq2seq の仕組みについて具体例を出しながら説明していきます。ここでは日本語から英語に翻訳する例を考えます。サンプルとして、「吾輩は猫である」という文章を「I am a cat」に翻訳するケースを扱います。このとき、seq2seq は図 7-5 のようになり、Encoder と Decoder によって時系列データの変換が行われます。

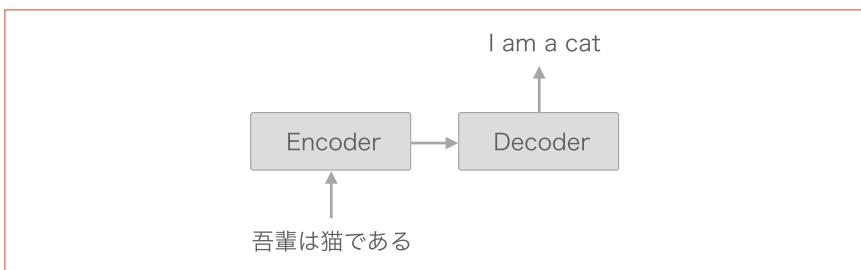


図 7-5 Encoder と Decoder によって翻訳を行う例

図 7-5 に示すように、まずは Encoder が「吾輩は猫である」という文章をエンコードします。そして、そのエンコードした情報を Decoder に渡し、目的とする文章を Decoder が生成します。このとき、Encoder がエンコードした情報には、翻訳するために必要な情報がコンパクトにまとめられています。Decoder はそのコンパクトにまとめられた情報を元に、目的とする文章を生成します。

これが seq2seq の全体図になります。Encoder と Decoder が協力して、時系列データを別の時系列データへと変換します。そして、これらの Encoder と Decoder には RNN を使うことができるのです。それでは、その詳細を見ていきましょう。まずは Encoder の処理にフォーカスします。早速、Encoder のレイヤ構成を図 7-6 に示します。

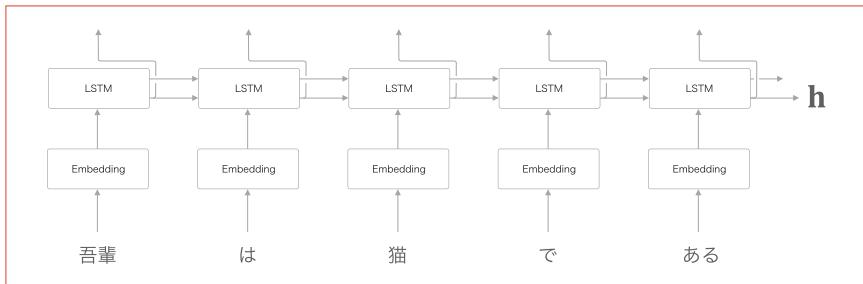


図7-6 Encoderを構成するレイヤ

図7-6に示すように、EncoderはRNNを利用して、時系列データを**h**という隠れ状態ベクトルに変換します。ここではRNNとしてLSTMを利用していますが、これは「シンプルなRNN」やGRUなどを用いることも、もちろん可能です。また、ここでは日本語の文章が単語単位に分割されて入力される場合を考えます。

さて、図7-6のEncoderが出力するベクトル**h**ですが、これはLSTMレイヤの最後の隠れ状態です。この最後の隠れ状態**h**に、入力文章を翻訳するために必要な情報がエンコードされます。そしてここで重要な点は、LSTMの隠れ状態**h**は固定長のベクトルであるということです。エンコードすると、つまるところ、任意の長さの文章を固定長のベクトルに変換することなのです（図7-7）。

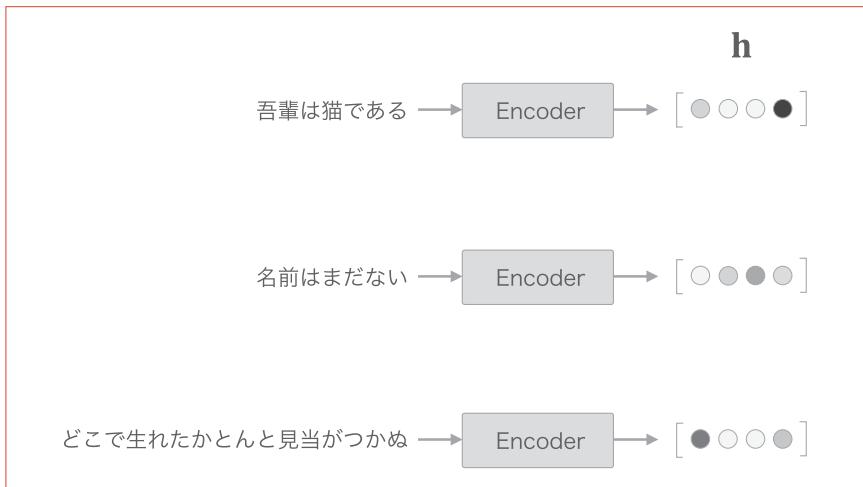


図7-7 Encoderは文章を固定長のベクトルにエンコードする

図7-7に示すように、Encoderは文章を固定長のベクトルに変換します。それでは、そのエンコードされたベクトルを、Decoderはどのように“料理”して、目的とする文章を生成するのでしょうか？その答えを私たちはすでに知っています。なぜなら、前節で扱った文章生成を行うモデルがそのまま利用できるからです。これは図で表すと、図7-8のようになります。

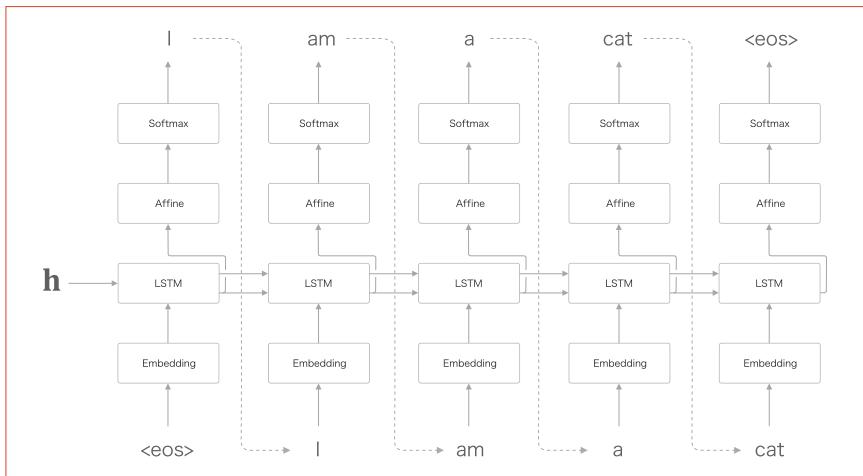


図7-8 Decoderを構成するレイヤ

図7-8のように、Decoderは前節のニューラルネットワークとまったく同じ構成です。ただし、前節で見たモデルとは異なる点がひとつだけあります。その違いは、LSTMレイヤがベクトル h を受け取るということです。ちなみに前節の言語モデルでは、LSTMレイヤは何も受け取りませんでした（強いて言うならば、LSTMの隠れ状態は“0ベクトル”を受け取ったとも言えます）。この唯一の、そして小さな違いが、一般的な言語モデルを翻訳をもこなせるDecoderへと進化させるのです！



図7-8では、<eos>という区切り文字（特殊文字）が利用されています。これは「区切り文字」であり、Decoderに文章生成の開始を知らせる合図として利用します。また、Decoderが<eos>を出力するまで単語のサンプリングを行うようにするため、これは終了の合図でもあります。つまり、<eos>はDecoderにとっての「開始/終了」を知らせる区切り文字として利用できるのです。他の文献では、この区切り文字として、<go>や<start>、または「_（アンダースコア）」などを用いている例もあります。

それでは、Decoder と Encoder をつなぎ合わせて、そのレイヤ構成を示します。そうすると、図7-9 のようになります。

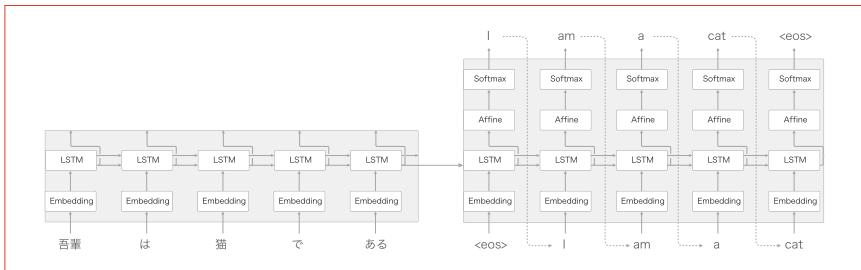


図7-9 seq2seq 全体のレイヤ構成

図7-9 に示すとおり、seq2seq は 2 つの LSTM——Encoder の LSTM と Decoder の LSTM——によって構成されます。このとき、LSTM レイヤの隠れ状態が Encoder と Decoder の「架け橋」となります。順伝播では、Encoder から Decoder へエンコードされた情報が LSTM レイヤの隠れ状態を通して伝わります。そして、seq2seq の逆伝播では、その「架け橋」を通って、勾配が Decoder から Encoder へ伝わります。

7.2.2 時系列データ変換用のトイ・プロblem

それでは seq2seq を実際に実装していきます。その前に、これから私たちが扱う問題について説明します。ここで私たちは、時系列変換の問題として、「足し算」を扱います。具体的には、図7-10 のように、「57+5」のような文字列を seq2seq に与え、「62」と正しく答えられるように学習するのです。ちなみに、このような機械学習を評価するために作られた簡単な問題は「トイ・プロblem (toy problem)」と呼ばれます。

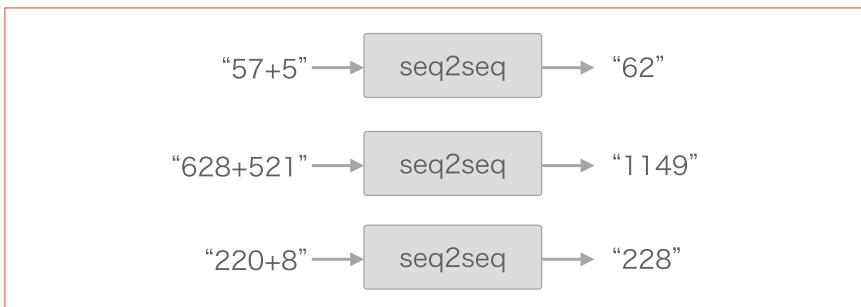


図 7-10 seq2seq に足し算を行う例を学習させる

ここで扱う足し算は、私たち人間にとっては簡単な問題です。しかし、seq2seq は足し算について——より正確に言うと、足し算のロジックについて——まったく何も知りません。seq2seq は、足し算の「例（サンプル）」から、そこで使われる文字のパターンを学習します。果たしてそれで、足し算のルールを正しく学習することができるのでしょうか？ それが今回の問題の見どころです。

ところで、私たちはこれまで word2vec や言語モデルなどにおいて、文章を「単語」という単位で分割してきました。しかし、必ずしも文章を単語に分割しなければならないということはありません。今回の問題に関しては、文を「単語」ではなく「文字」単位で分割したいと思います。文字単位の分割とは、たとえば「57+5」のような入力文の場合、['5', '7', '+', '5'] のリストとして扱います。

7.2.3 可変長の時系列データ

私たちは「足し算」を文字（数字）のリストとして扱うことになります。ここで注意が必要なのは、足し算の問題文（「57+5」や「628+521」など）やその答え（「62」や「1149」など）は、問題ごとに文字数が異なるということです。たとえば、「57+5」は全部で 4 文字ですが、「628+521」の場合は 7 文字です。

このように、今回の「足し算」問題では、サンプルごとにデータの時間方向のサイズが異なります。つまり、足し算問題では「可変長の時系列データ」を扱うことになるのです。そのため、ニューラルネットワークの学習において「ミニバッチ処理」を行うには、何らかの工夫が必要になります。



ミニバッチで学習を行うとき、複数のサンプルをまとめて処理します。このとき、（私たちの実装では）バッチ内の各サンプルのデータ形状をすべて同じにする必要があります。

可変長の時系列データをミニバッチ学習するための最も単純な方法は**パディング**を使うことです。パディングとは、本来のデータを（意味のない）無効なデータで埋め、データの長さを均一に揃えるテクニックです。今回の足し算の例で言えば、図7-11のように、すべての入力データの長さを統一し、余った場所には無効な文字——ここでは「空白文字」——を挿入します。

| 入力 | 出力 |
|---------------|-----------|
| 5 7 + 5 | _ 6 2 |
| 6 2 8 + 5 2 1 | _ 1 1 4 9 |
| 2 2 0 + 8 | _ 2 2 8 |

図7-11 ミニバッチ学習を行うため、「空白文字」でパディングを行い、入力・出力のデータサイズを揃える

今回の問題では、0～999の2つの数の足し算を扱うことにします。そのため、「+」を含めると、入力の最大文字数は7文字になります。また、足し算の結果は最大で4文字です（最大で「999 + 999 = 1998」）。そこで教師データも同様にパディングを行い、すべてのサンプルデータの長さを揃えます。さらに今回の問題では、区切り文字として「_（アンダースコア）」を出力の先頭に付けることにします。よって、出力データは文字数を5文字で揃えます。なお、この区切り文字は、Decoderに文字列生成を知らせる合図として使われます。



Decoderの出力に関しては、教師ラベルとして文字出力の終了を知らせる区切り文字を入れることができます（たとえば、「_62_」や「_1149_」）。しかし、ここでは話を単純にするため、文字列生成の終了を表す区切り文字は与えないことにします。つまり、Decoderは文字列生成を行うときは、常に決まった文字数だけ（ここでは「_」を含めて5文字）出力させるようにします。

このように、パディングを用いてデータサイズを揃えることで、可変長の時系列

データを扱うことができます。しかしパディングを用いることで、本来存在しなかったパディング用の文字まで seq2seq に処理させることになります。そのため、正確を期すならば、(パディングを用いるときには) パディング専用の処理を seq2seq に加える必要があります。たとえば、Decoder でパディングが入力されたときには、損失の結果に計上しないようにします(これは Softmax with Loss レイヤに「マスク」という機能を追加することで対処できます)。また、Encoder ではパディングが入力されたとき、LSTM レイヤは前時刻の入力をそのまま出力するようにします。これによって、LSTM レイヤはパディングが存在しなかったものとして、入力データをエンコードできます。

この辺の話は、やや複雑なため理解できなくても問題ありません。本章では、分かりやすさを優先して、パディング用の文字(空白文字)に対して特別な処理を行わず、通常のデータとして処理することにします。

7.2.4 足し算データセット

ここで説明した足し算の学習データ例は、dataset/addition.txt にあらかじめ収納されています。このテキストファイルには図7-12に示すように、足し算の例が50,000個あります。なお、この学習データは Keras の seq2seq の実装例[40]を参考に作成しました。

| | | |
|----|---------|-------|
| 1 | 16+75 | _91 |
| 2 | 52+607 | _659 |
| 3 | 75+22 | _97 |
| 4 | 63+22 | _85 |
| 5 | 795+3 | _798 |
| 6 | 706+796 | _1502 |
| 7 | 8+4 | _12 |
| 8 | 84+317 | _401 |
| 9 | 9+3 | _12 |
| 10 | 6+2 | _8 |
| 11 | 18+8 | _26 |
| 12 | 85+52 | _137 |
| 13 | 9+1 | _10 |
| 14 | 8+20 | _28 |
| 15 | 5+3 | _8 |

Lines: 50,000 Chars: 650,000 650 KB

図7-12 「足し算」の学習データ：空白文字(スペース)はグレーのドットで表示

また本書では、上のような seq2seq 用の学習データ（テキストファイル）を Python から簡単に扱えるように専用のモジュール (`dataset/sequence.py`) を提供します。このモジュールは `load_data()` と `get_vocab()` の 2 つのメソッドを持ちます。

`load_data(file_name, seed)` は、`file_name` で指定されたテキストファイルを読み込み、テキストを文字 ID に変換し、それを訓練データとテストデータに分けて返します。このとき内部では乱数のシードを `seed` に設定し、データをシャッフルしてから訓練データとテストデータに分離しています。また、`get_vocab()` メソッドは、文字と ID の対応関係を表すディクショナリを返します（実際には、`char_to_id` と `id_to_char` の 2 つのディクショナリを返します）。それでは、実際に使用例を見てみましょう（☞ `ch07/show_addition_dataset.py`）。

```
import sys
sys.path.append('..')
from dataset import sequence

(x_train, t_train), (x_test, t_test) = \
    sequence.load_data('addition.txt', seed=1984)
char_to_id, id_to_char = sequence.get_vocab()

print(x_train.shape, t_train.shape)
print(x_test.shape, t_test.shape)
# (45000, 7) (45000, 5)
# (5000, 7) (5000, 5)

print(x_train[0])
print(t_train[0])
# [ 3  0  2  0  0 11  5]
# [ 6  0 11  7  5]

print(''.join([id_to_char[c] for c in x_train[0]]))
print(''.join([id_to_char[c] for c in t_train[0]]))
# 71+118
# _189
```

このように、`sequence` モジュールを利用することで、seq2seq 用のデータを簡単に読み込むことができます。ここで、`x_train` や `t_train` には「文字 ID」が格納されています。また、文字 ID と文字の対応関係は、`char_to_id` もしくは `id_to_char` を使って変換することができます。



データセットは本来であれば、訓練用、検証用、テスト用の 3 つに分けて使用するべきです。訓練用データで学習を行い、検証用データでハイパーパラメータのチューニングを行います。そして最後に、テスト用データでモデルの実力を評価します。ただしここでは単純さを考慮して、訓練用とテスト用の 2 つだけに分離して、それを使ってモデルの訓練と評価を行います。

7.3 seq2seq の実装

seq2seq は、2 つの RNN を組み合わせたニューラルネットワークです。ここでは最初に、その 2 つの RNN を `Encoder` クラスと `Decoder` クラスとしてそれぞれ実装することにします。そして、その 2 つのクラスを組み合わせて、`Seq2seq` クラスを実装する流れで進めます。まずは `Encoder` クラスから始めます。

7.3.1 Encoder クラス

`Encoder` クラスは、図 7-13 に示すように、文字列を受け取り、それをベクトル \mathbf{h} に変換します。

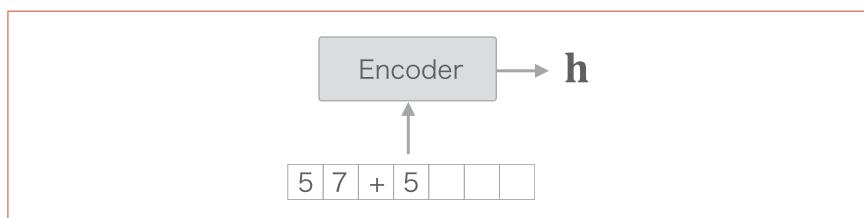


図 7-13 Encoder の入出力

前に説明したとおり、私たちは RNN を用いて Encoder を実現します。ここでは LSTM レイヤを用いて、図 7-14 のレイヤ構成で実装します。

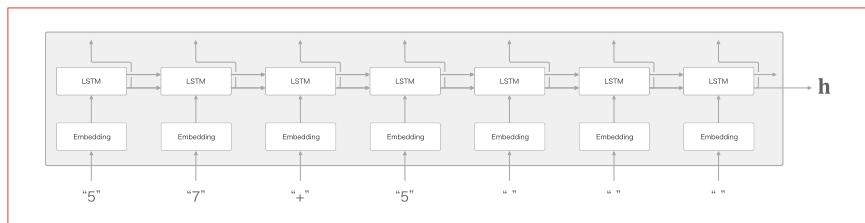


図 7-14 Encoder のレイヤ構成

図 7-14 に示すとおり、Encoder クラスは、Embedding レイヤと LSTM レイヤによって構成されます。Embedding レイヤでは、文字（正確には文字 ID）を文字ベクトルに変換します。そして、その文字ベクトルが LSTM レイヤへと入力されます。

LSTM レイヤは、右方向（時間方向）には隠れ状態とセルを出力し、上方向には隠れ状態だけを出力します。ここでは上方向にレイヤは存在しないため、LSTM レイヤの上方向の出力は破棄されることになります。図 7-14 に示すとおり、Encoder では最後の文字を処理した後、LSTM レイヤの隠れ状態 **h** を出力します。そして、この隠れ状態 **h** が Decoder に渡されることになります。



Encoder では、LSTM の隠れ状態だけを Decoder に渡します。LSTM のセルも Decoder に渡すことは可能ですが、LSTM のセルを他のレイヤに与えることは一般的にはあまり行われません。これは、LSTM のセルが自分自身だけで利用することを前提として設計されているためです。

ところで私たちは、時間方向をまとめて処理するレイヤを、Time LSTM レイヤや Time Embedding レイヤとして実装してきました。それらの Time レイヤを利用すると、私たちの Encoder は図 7-15 のようになります。

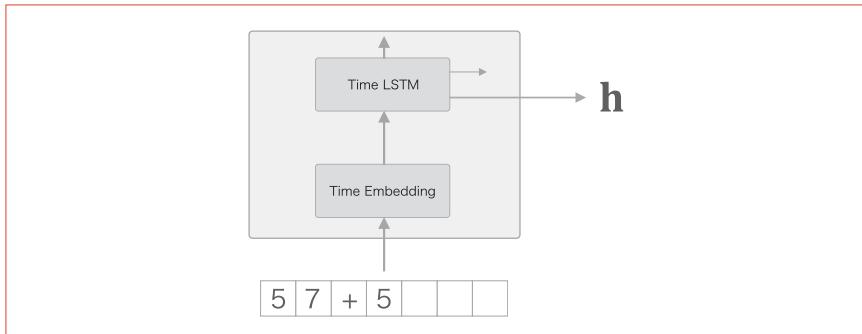


図 7-15 Encoder を Time レイヤで実装する

それでは、Encoder クラスを次に示します。この Encoder クラスは、初期化と順伝播と逆伝播——`__init__()`、`forward()`、`backward()`——の 3 つのメソッドを持ちます。まずは初期化のメソッドから示します ([ch07/seq2seq.py](#))。

```
class Encoder:
    def __init__(self, vocab_size, wordvec_size, hidden_size):
        V, D, H = vocab_size, wordvec_size, hidden_size
        rn = np.random.randn

        embed_W = (rn(V, D) / 100).astype('f')
        lstm_Wx = (rn(D, 4 * H) / np.sqrt(D)).astype('f')
        lstm_Wh = (rn(H, 4 * H) / np.sqrt(H)).astype('f')
        lstm_b = np.zeros(4 * H).astype('f')

        self.embed = TimeEmbedding(embed_W)
        self.lstm = TimeLSTM(lstm_Wx, lstm_Wh, lstm_b, stateful=False)

        self.params = self.embed.params + self.lstm.params
        self.grads = self.embed.grads + self.lstm.grads
        self.hs = None
```

初期化のメソッドでは、引数に `vocab_size`、`wordvec_size`、`hidden_size` の 3 つを受け取ります。`vocab_size` は語彙数であり、これは文字の種類に相当します。ちなみに今回は、0～9の数字と「+」と「」（空白文字）と「_」の計 13 文字が語彙数になります。また、`wordvec_size` は文字ベクトルの次元数、`hidden_size` は LSTM レイヤの隠れ状態ベクトルの次元数に対応します。

この初期化メソッドでは、重みパラメータの初期化とレイヤの生成を行います。最後に、重みパラメータと勾配をメンバ変数の `params` と `grads` のリストにそ

れぞれ集約します。また今回は、Time LSTM レイヤは状態を維持しないため、`stateful=False` に設定します。



5 章と 6 章の言語モデルは、「長い時系列データ」がひとつだけ存在する問題として扱いました。そこでは、Time LSTM レイヤの引数で `stateful=True` と設定し、それによって隠れ状態を維持したまま「長い時系列データ」を処理したのです。一方、今回は「短い時系列データ」が複数存在する問題です。そのため、問題ごとに LSTM の隠れ状態をリセットした状態（“0 ベクトル”）に設定します。

それでは続いて、`forward()` と `backward()` メソッドを次に示します（☞ ch07/seq2seq.py）。

```
def forward(self, xs):
    xs = self.embed.forward(xs)
    hs = self.lstm.forward(xs)
    self.hs = hs
    return hs[:, -1, :]

def backward(self, dh):
    dhs = np.zeros_like(self.hs)
    dhs[:, -1, :] = dh

    dout = self.lstm.backward(dhs)
    dout = self.embed.backward(dout)
    return dout
```

Encoder の順伝播では、Time Embedding レイヤと Time LSTM レイヤの `forward()` メソッドを呼びます。そして、Time LSTM レイヤの最後の時刻の隠れ状態だけを取り出し、それを Encoder の `forward()` メソッドの出力とします。

Encoder の逆伝播では、LSTM レイヤの最後の隠れ状態に対する勾配が `dh` として伝わります。この `dh` は Decoder から伝わる勾配です。逆伝播の実装では、要素が 0 のテンソル `dhs` を生成し、`dh` を `dhs` の該当する箇所に設定します。後は、Time Embedding レイヤと Time LSTM レイヤの `backward()` メソッドを呼ぶだけです。以上が Encoder クラスの実装です。

7.3.2 Decoder クラス

続いて Decoder クラスの実装へと移ります。Decoder クラスは、図 7-16 に示す

ように、Encoder クラスが output した \mathbf{h} を受け取り、目的とする別の文字列を出力します。

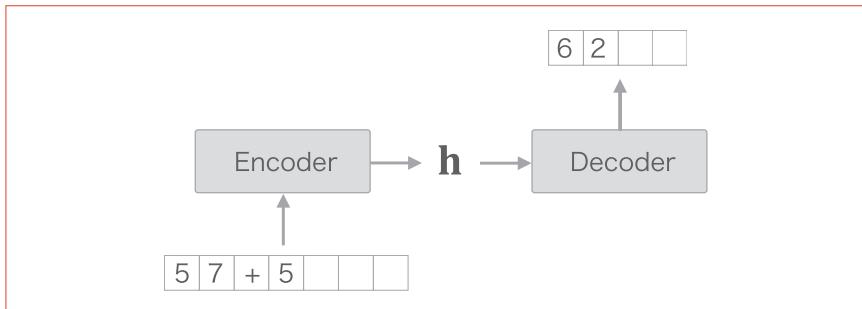


図 7-16 Encoder と Decoder

前節で説明したとおり、Decoder は RNN で実現することができます。ここでも Encoder と同様に、LSTM レイヤを使います。このとき、Decoder のレイヤ構成は図 7-17 のようになります。

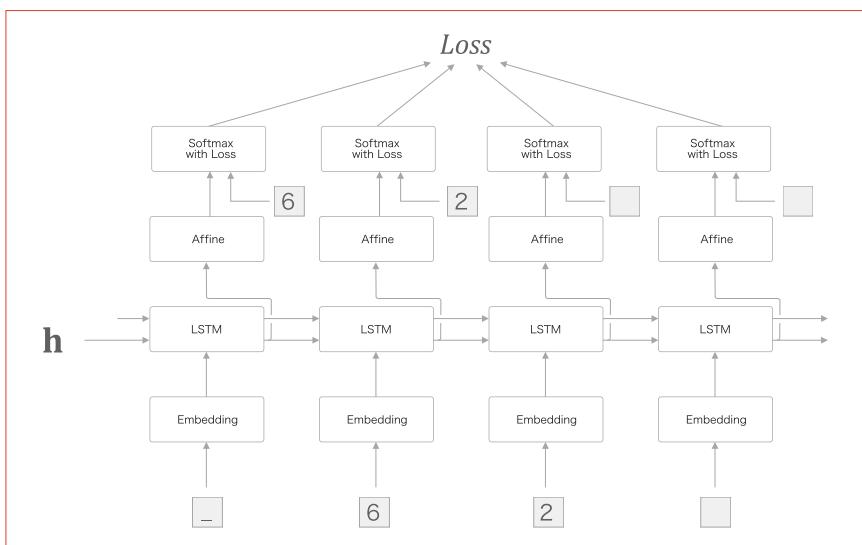


図 7-17 Decoder のレイヤ構成（学習時）

図7-17は、Decoderの学習時におけるレイヤ構成を示しています。ここでは「_62」という教師データを使いますが、このとき入力データは[「_」, '6', '2', ' ']として与え、それに対応する出力が['6', '2', ' ', ' ']となるように学習を行わせます。



RNNで文章生成を行う場合、学習時と生成時ではデータの与え方が異なります。学習時は正解が分かっているため、時系列方向のデータをまとめて与えることができます。一方、推論時には——新しい文字列を生成するときには——、最初に開始を知らせる区切り文字（今回の例では「_」）をひとつだけ与えます。そして、そのときの出力から文字をひとつサンプリングし、そのサンプリングした文字を次の入力にする、というような一連の処理を繰り返し行います。

ところで、7.1節で文章生成を行うときに、私たちはSoftmax関数の確率分布を元にサンプリングを行いました。そのため、生成される文章は確率的に変動しました。今回の問題は「足し算」ということで、そのような確率的な“揺れ”を排除して「決定的」に答えを生成したいと思います。そこで今回は、最も高いスコアを持つ文字をひとつ選ぶようにします。つまり、「確率的」ではなく「決定的」に選ぶのです。それでは、Decoderに文字列を生成させる流れを図7-18に示します。

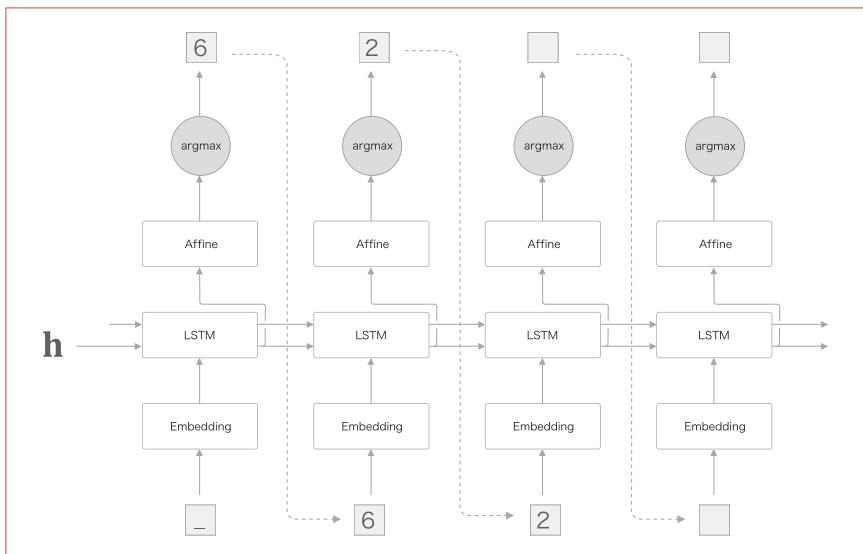


図7-18 Decoderの文字列生成の手順：Affineレイヤの出力から最大値を取りインデックス（文字ID）をargmaxノードで選ぶ

図7-18に示すとおり、ここでは新しく「argmax」というノードが登場します。これは最大値を取りインデックス（今回の例では文字ID）を選ぶノードです。図7-18の構成は、前節で示した文章生成のときの構成と同じです。ただし今回は、Softmaxレイヤを使わず、Affineレイヤの出力するスコアを対象に、その中から最大の値を持つ文字IDを選びます。



Softmaxレイヤは入力されたベクトルを正規化します。このとき、ベクトルの各要素の値は変換されますが、その大小関係は変わりません。そのため、図7-18のケースではSoftmaxレイヤを省略することができます。

ここで説明したように、Decoderでは学習時と生成時でSoftmaxレイヤの扱いが異なります。そこで、Softmax with Lossレイヤは、この後に実装するSeq2seqクラスに面倒を見てくれることにしましょう。そのため、Decoderクラスは、図7-19のようにTime Softmax with Lossレイヤの前までを担当させることにします。

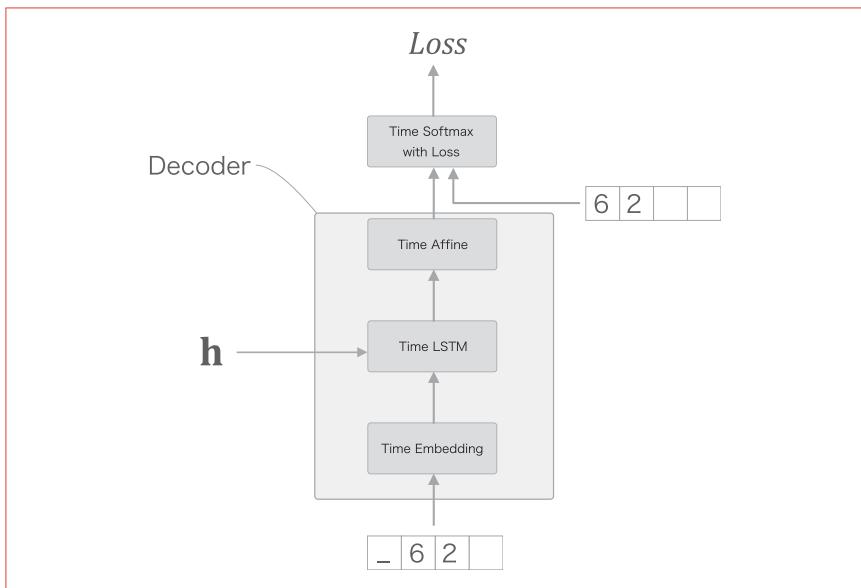


図 7-19 Decoder クラスの構成

図 7-19 に示すとおり、Decoder クラスは、Time Embedding、Time LSTM、Time Affine の 3 つのレイヤから構成されます。それでは、Decoder クラスの実装です。ここでは初期化の `__init__()`、順伝播の `forward()`、逆伝播の `backward()` をまとめて示します ([ch07/seq2seq.py](#))。

```
class Decoder:
    def __init__(self, vocab_size, wordvec_size, hidden_size):
        V, D, H = vocab_size, wordvec_size, hidden_size
        rn = np.random.randn

        embed_W = (rn(V, D) / 100).astype('f')
        lstm_Wx = (rn(D, 4 * H) / np.sqrt(D)).astype('f')
        lstm_Wh = (rn(H, 4 * H) / np.sqrt(H)).astype('f')
        lstm_b = np.zeros(4 * H).astype('f')
        affine_W = (rn(H, V) / np.sqrt(H)).astype('f')
        affine_b = np.zeros(V).astype('f')

        self.embed = TimeEmbedding(embed_W)
        self.lstm = TimeLSTM(lstm_Wx, lstm_Wh, lstm_b, stateful=True)
        self.affine = TimeAffine(affine_W, affine_b)

        self.params, self.grads = [], []
        self.cache = None
```

```

for layer in (self.embed, self.lstm, self.affine):
    self.params += layer.params
    self.grads += layer.grads

def forward(self, xs, h):
    self.lstm.set_state(h)

    out = self.embed.forward(xs)
    out = self.lstm.forward(out)
    score = self.affine.forward(out)
    return score

def backward(self, dscore):
    dout = self.affine.backward(dscore)
    dout = self.lstm.backward(dout)
    dout = self.embed.backward(dout)
    dh = self.lstm.dh
    return dh

```

ここでは逆伝播についてだけ簡単に補足します。`backward()` の実装では、上方向にある Softmax with Loss レイヤから勾配 `dscore` を受け取り、Time Affine レイヤ、Time LSTM レイヤ、Time Embedding レイヤの順に勾配を伝播させます。ここで、Time LSTM レイヤの時間方向への勾配は、`TimeLSTM` クラスのメンバ変数 `dh` に保持されています（詳細は「6.3 LSTM の実装」を参照）。そこで、その時間方向の勾配 `dh` を取り出し、それを `Decoder` クラスの `backward()` の出力とします。

前に述べたとおり、`Decoder` クラスは学習時と文章生成時で挙動が異なります。上記の `forward()` メソッドは学習時に使用されることを想定しています。`Decoder` クラスに文章生成を行うメソッドを `generate()` として実装します。

```

def generate(self, h, start_id, sample_size):
    sampled = []
    sample_id = start_id
    self.lstm.set_state(h)

    for _ in range(sample_size):
        x = np.array(sample_id).reshape((1, 1))
        out = self.embed.forward(x)
        out = self.lstm.forward(out)
        score = self.affine.forward(out)

        sample_id = np.argmax(score.flatten())
        sampled.append(int(sample_id))

    return sampled

```

この `generate()` メソッドは引数を 3 つ取ります。それは、Encoder から受け取る隠れ状態の `h`、最初に与える文字 ID の `start_id`、そして、生成する文字数の `sample_size` です。ここでは文字をひとつずつ与え、Affine レイヤが output するスコアから最大値を持つ文字 ID を選ぶ作業を繰り返し行います。以上が Decoder クラスの実装です。



今回の問題では、Encoder の出力 `h` を Decoder の Time LSTM レイヤに設定します。このとき、Time LSTM レイヤをステートフル (stateful) にしたことで、その隠れ状態はリセットされずに、Encoder の `h` を維持したまま順伝播が行われます。

7.3.3 Seq2seq クラス

最後に Seq2seq クラスの実装です。といっても、ここで行うことは、Encoder クラスと Decoder クラスをつなぎ合わせ、そして Time Softmax with Loss レイヤを使って損失を計算するだけです。それでは、Seq2seq クラスを次に示します ([ch07/seq2seq.py](#))。

```
class Seq2seq(BaseModel):
    def __init__(self, vocab_size, wordvec_size, hidden_size):
        V, D, H = vocab_size, wordvec_size, hidden_size
        self.encoder = Encoder(V, D, H)
        self.decoder = Decoder(V, D, H)
        self.softmax = TimeSoftmaxWithLoss()

        self.params = self.encoder.params + self.decoder.params
        self.grads = self.encoder.grads + self.decoder.grads

    def forward(self, xs, ts):
        decoder_xs, decoder_ts = ts[:, :-1], ts[:, 1:]

        h = self.encoder.forward(xs)
        score = self.decoder.forward(decoder_xs, h)
        loss = self.softmax.forward(score, decoder_ts)
        return loss

    def backward(self, dout=1):
        dout = self.softmax.backward(dout)
        dh = self.decoder.backward(dout)
        dout = self.encoder.backward(dh)
        return dout
```

```
def generate(self, xs, start_id, sample_size):
    h = self.encoder.forward(xs)
    sampled = self.decoder.generate(h, start_id, sample_size)
    return sampled
```

Encoder と Decoder の各クラスで、メインとなる処理はすでに実装されています。そのため、ここではそれらをつなぎ合わせるだけになります。以上が Seq2seq クラスです。それでは、この Seq2seq クラスを使って、「足し算」問題に挑みたいと思います。

7.3.4 seq2seq の評価

seq2seq の学習は、基本的なニューラルネットワークの学習と同じ流れで行われます。基本的なニューラルネットワークの学習とは、

1. 学習データからミニバッチを選び、
2. ミニバッチから勾配を計算し、
3. 勾配を使ってパラメータを更新する

というお決まりの流れです。ここでは、「1.4.4 Trainer クラス」で説明した Trainer クラスを使って、上の作業を行わせます。また、ここではエポックごとに seq2seq にテストデータを解かせ（文字列生成を行わせ）、その正解率を計測したいと思います。それでは早速、seq2seq の学習のコードを次に示します（☞ ch07/train_seq2seq.py）。

```
import sys
sys.path.append('..')
import numpy as np
import matplotlib.pyplot as plt
from dataset import sequence
from common.optimizer import Adam
from common.trainer import Trainer
from common.util import eval_seq2seq
from seq2seq import Seq2seq
from peeky_seq2seq import PeekySeq2seq

# データセットの読み込み
(x_train, t_train), (x_test, t_test) = sequence.load_data('addition.txt')
char_to_id, id_to_char = sequence.get_vocab()

# ハイパーパラメータの設定
```

```

vocab_size = len(char_to_id)
wordvec_size = 16
hidden_size = 128
batch_size = 128
max_epoch = 25
max_grad = 5.0

# モデル / オプティマイザ / トレーナーの生成
model = Seq2seq(vocab_size, wordvec_size, hidden_size)
optimizer = Adam()
trainer = Trainer(model, optimizer)

acc_list = []
for epoch in range(max_epoch):
    trainer.fit(x_train, t_train, max_epoch=1,
                batch_size=batch_size, max_grad=max_grad)

    correct_num = 0
    for i in range(len(x_test)):
        question, correct = x_test[[i]], t_test[[i]]
        verbose = i < 10
        correct_num += eval_seq2seq(model, question, correct,
                                    id_to_char, verbose)
    acc = float(correct_num) / len(x_test)
    acc_list.append(acc)
    print('val acc %.3f%%' % (acc * 100))

```

ここで示したコードは、基本的なニューラルネットワークの学習用のコードと同じです。ただしここでは評価指標として正解率——いくつの問題に正解できたか——を採用します。具体的には、エポックごとにテストデータにある問題の中でいくつの問題に正しく答えられたかを採点するのです。

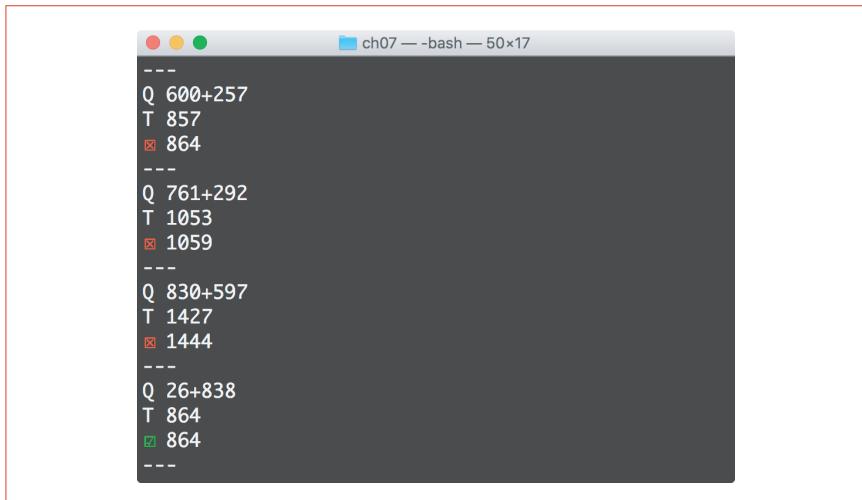
なお、上の実装での正解率を計測するために、common/util.py にある eval_seq2seq(model, question, correct, id_to_char, verbose, is_reverse) というメソッドを利用しています。このメソッドは、問題をモデルに与えて文字列生成を行わせ、それが答えと合っているかどうかを判定します。モデルの出す答えが合っていれば 1 を返し、間違っていれば 0 を返します。



eval_seq2seq(model, question, correct, id_to_char, verbose, is_reverse) メソッドの引数は 6 個あります。まずはモデルを表す model、問題文（文字 ID の配列）の question、正解（文字 ID の配列）の correct です。そして、文字 ID と文字の変換を行うディクショナリの id_to_char、結果を表示するかどうかを指定する verbose、入力文を反転したかどうかの is_reverse の計 6 個の引数です。verbose = True に

設定すると、結果がターミナルに表示されます。今回の実験では、テストデータの最初の 10 個分だけ表示することにしています。また、引数の `is_reverse` については後ほど明らかになります。

それでは、上のコードを実行してみましょう。このとき、次のような結果がターミナル（コンソール）に表示されます^{†1}。



```
ch07 — -bash — 50x17
---
Q 600+257
T 857
☒ 864
---
Q 761+292
T 1053
☒ 1059
---
Q 830+597
T 1427
☒ 1444
---
Q 26+838
T 864
☒ 864
---
```

図 7-20 ターミナルでの結果の表示例

図 7-20 に示すように、ターミナルにはエポックごとに結果が表示されます。各行にある「Q 600 + 257」というのが問題文で、その下の「T 857」が正解になります。そして「☒ 864」というのが、私たちのモデルの出した答えです。もし私たちのモデルが正解となる答えを出していれば、「☑ 864」のように表示されます。

それでは、学習が進むにつれて、上の結果はどのように変化していくのかを見てていきましょう。ここではその一例を、図 7-21 に示します。

^{†1} Windows 環境では、☒ の代わりに「O」が、☒ の代わりに「X」が表示されます。



図 7-21 ターミナルに表示される結果の推移

図 7-21 では、学習が進むに従って表示される結果をいくつか選んで示しています。この結果を見ると分かりますが、seq2seq は最初うまく答えを出せずにいます。しかし、学習を重ねるにつれて徐々に正しい答えに近づき、そして、いくつかの問題には正しく答えることができるようになります。それでは、エポックごとに正解率をプロットしてみましょう。その結果は図 7-22 のようになります。

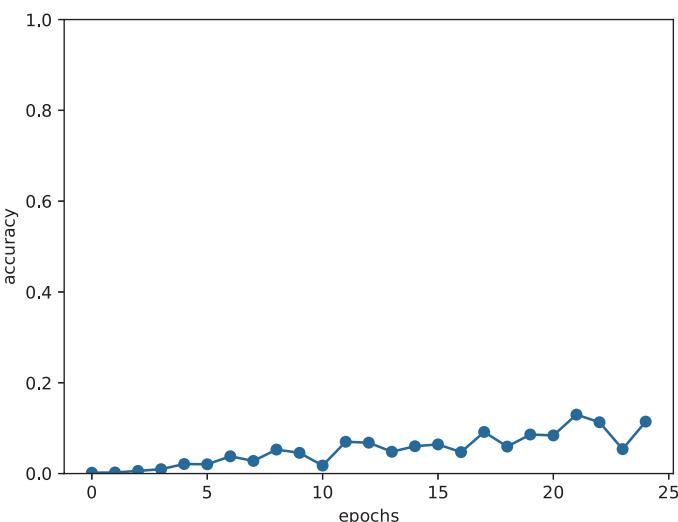


図 7-22 正解率の推移

図 7-22 のとおり、エポックを重ねるにつれて正解率が着実に上昇していることが分かります。今回の実験では 25 エポックで打ち切りましたが、その時点での正解率は 10% 程度です。グラフの伸びを見るに、さらに学習を重ねれば、まだまだ正解率は上昇していくことでしょう。しかし、この学習は一旦終わりにして、同じ問題（足し算問題）をより良く学習できように、seq2seq の改良を行いたいと思います。

7.4 seq2seq の改良

ここでは前節の seq2seq に対して改良を行い、学習の“進み”を改善したいと思います。そのためのテクニックはいくつか有望なものがあります。本節では 2 つの改善案を示し、実験によってその効果を確かめます。

7.4.1 入力データの反転 (Reverse)

ひとつ目の改善案は、とても簡単なトリックです。これは図 7-23 で示すように、入力データの順序を反転させるというものです。

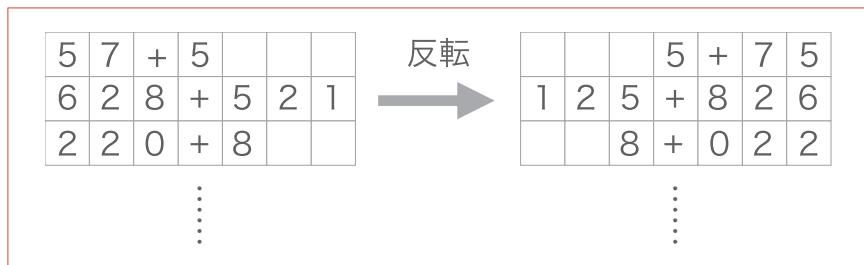


図 7-23 入力データを反転させる例

この入力データを反転させるというトリックは、文献[41]で提案されました。これを使うと、多くの場合は学習の進みが速くなり、最終的な精度も良くなることが報告されています。それでは実際に実験してみましょう。

入力データを反転させるためには、前節の学習用のコード(`ch07/train_seq2seq.py`)に対して、データセットを読み込んだ後に次のコードを追加します。

```
# データセットの読み込み
(x_train, t_train), (x_test, t_test) = sequence.load_data('addition.txt')
...
x_train, x_test = x_train[:, ::-1], x_test[:, ::-1]
...
```

ここで示すように、配列の並びを反転させるには、`x_train[:, ::-1]` という記法が使えます。それでは、入力データを反転することによってどれだけ正解率が上がるのかを見てみましょう。結果は図7-24 のようになります。

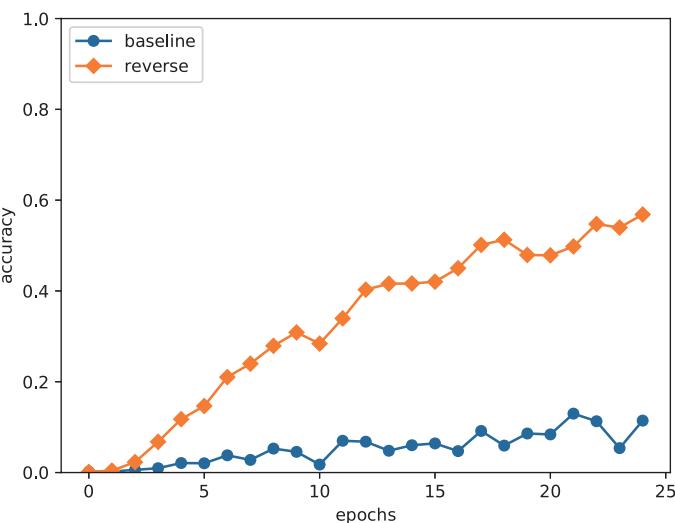


図 7-24 seq2seq の正解率の推移：baseline は前節の結果、reverse は入力データを反転させた場合の結果

図 7-24 を見て分かるとおり、入力データを反転させただけで、学習の進みが改善しました！ 25 エポックの時点で正解率は 50% 程度です。繰り返しになりますが、前回（グラフでは baseline）との違いは、入力データを反転させただけです。それだけで、これほど差がつくというのは驚きです。もちろん、データを反転させる効果は問題に応じて異なりますが、多くの場合、良い結果につながります。

それでは、なぜ入力データを反転させるだけで学習の進みが早くなり、精度が向上するのでしょうか？（理論的なことは分かっていませんが）直感的には勾配の伝播がスムーズになるからだと考えられます。たとえば、「吾輩は猫である」を「I am a cat」に翻訳する問題を考えたとき、「吾輩」という単語は「I」という単語へと変換される関係にあります。このとき、「吾輩」から「I」までの道のりは、「は」「猫」「で」「ある」の 4 単語分の LSTM レイヤを経由しなければなりません。そのため逆伝播を考えたとき、「I」からの伝わる勾配は「吾輩」にたどり着くまでに、その離れた距離分だけ影響を受けることになります。

そこで、入力文を反転させると、つまり「あるで猫は吾輩」にするとどうなるでしょうか。このとき、「吾輩」と「I」は隣どうしになるため、勾配はダイレクトに伝わります。このように、入力文の先頭部分では、反転することによって対応関係に

ある変換後の単語との距離が近くなるため（そうなるケースが多くなるため）、勾配が伝わりやすく、効率の良い学習ができると考えられます。ただし、入力データを反転しても、“平均的”な単語間の距離は変わりません。

7.4.2 観き見 (Peeky)

続いて、seq2seq の 2 つ目の改善に進みます。ここでは本題に入る前に、seq2seq の Encoder の働きにもう一度目を向けましょう。前にも説明したとおり、Encoder は入力文（問題文）を固定長のベクトル \mathbf{h} に変換します。このとき、その \mathbf{h} の中には、Decoder にとって必要な情報がすべて詰まっています。つまりそれが、Decoder にとっての唯一の情報源になるのです。しかし、現状の seq2seq は図 7-25 に示すとおり、最初の時刻の LSTM レイヤのみがベクトル \mathbf{h} を利用しています。この重要な情報である \mathbf{h} を、もっと活用できないでしょうか？

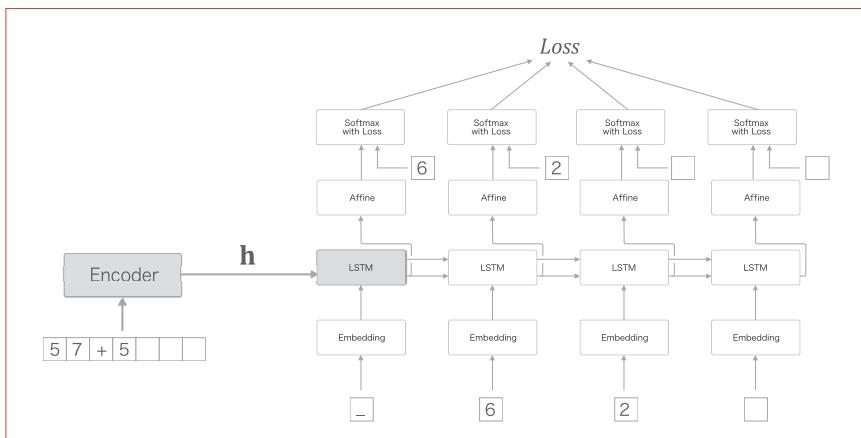


図 7-25 改善前：Encoder の出力 \mathbf{h} は、最初の LSTM レイヤだけが受け取る

そこで seq2seq の 2 つ目の改善案です。それは重要な情報が詰まった Encoder の出力 \mathbf{h} を、Decoder の他のレイヤにも与えるのです。私たちの Decoder の場合、図 7-26 のようなネットワーク構成が考えられます。

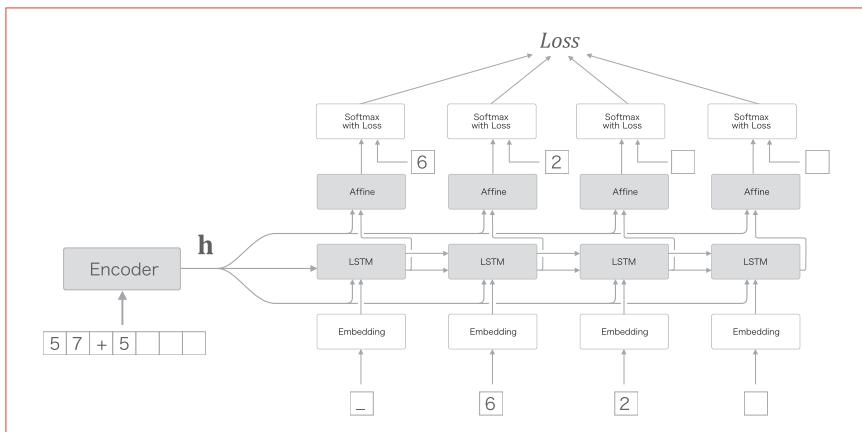


図 7-26 改善後：Encoder の出力 h を、すべての時刻の LSTM レイヤと Affine レイヤに与える

図 7-26 に示すとおり、すべての時刻の Affine レイヤと LSTM レイヤに Encoder の出力 h を与えます。図 7-26 と図 7-25 を見比べると、これまでひとつの LSTM レイヤだけが専有していた重要な情報 h を、複数のレイヤ（この例では 8 つのレイヤ）で共有できていることが分かります。重要な情報は一人が専有するのではなく、多くの人と共有する——これによって、より正しい決断が期待できます。



ここでの改善案は、エンコードされた情報を Decoder の別のレイヤにも与えるというものです。これは、別のレイヤにもエンコード情報を“覗かせる”と解釈できます。「覗く」とは、英語で peek と言うので、この改良を加えた Decoder を「Peeky Decoder」と呼ぶことにします。同様に、Peeky Decoder を利用する seq2seq を「Peeky seq2seq」と呼ぶことにします。なお、このアイデア（のベース）は文献 [42] に基づいています。

さて、図 7-26において LSTM レイヤと Affine レイヤには、2 本のベクトルが入力されています。これは実際には、2 つのベクトルが結合（concatenate）されたものを意味します。そのため先の図は、2 つのベクトルを結合させる concat ノードを用いて、図 7-27 のように書くのが正確な計算グラフです。

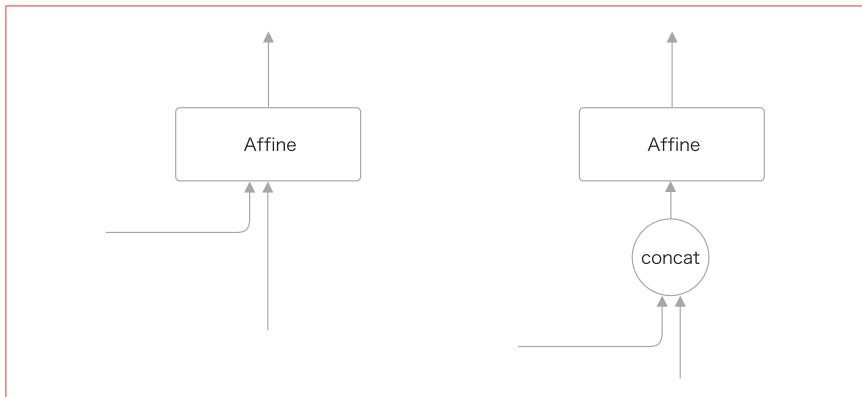


図 7-27 Affine レイヤへの入力が 2 本ある場合（左図）、正確に図示すると、その 2 つを連結したベクトルが Affine レイヤの入力となる（右図）

それでは、PeekyDecoder クラスの実装を次に示します。ここでは、初期化の`__init__()`と順伝播の`forward()`だけを示します。逆伝播の`backward()`と文章生成の`generate()`は、特に難しい点はないので省略します（☞ ch07/peeky_seq2seq.py）。

```
class PeekyDecoder:
    def __init__(self, vocab_size, wordvec_size, hidden_size):
        V, D, H = vocab_size, wordvec_size, hidden_size
        rn = np.random.randn

        embed_W = (rn(V, D) / 100).astype('f')
        lstm_Wx = (rn(H + D, 4 * H) / np.sqrt(H + D)).astype('f')
        lstm_Wh = (rn(H, 4 * H) / np.sqrt(H)).astype('f')
        lstm_b = np.zeros(4 * H).astype('f')
        affine_W = (rn(H + H, V) / np.sqrt(H + H)).astype('f')
        affine_b = np.zeros(V).astype('f')

        self.embed = TimeEmbedding(embed_W)
        self.lstm = TimeLSTM(lstm_Wx, lstm_Wh, lstm_b, stateful=True)
        self.affine = TimeAffine(affine_W, affine_b)

        self.params, self.grads = [], []
        for layer in (self.embed, self.lstm, self.affine):
            self.params += layer.params
            self.grads += layer.grads
        self.cache = None

    def forward(self, xs, h):
```

```

N, T = xs.shape
N, H = h.shape

self.lstm.set_state(h)

out = self.embed.forward(xs)
hs = np.repeat(h, T, axis=0).reshape(N, T, H)
out = np.concatenate((hs, out), axis=2)

out = self.lstm.forward(out)
out = np.concatenate((hs, out), axis=2)

score = self.affine.forward(out)
self.cache = H
return score

```

`PeekyDecoder` の初期化は、前節の `Decoder` とほとんど同じです。異なる点は、LSTM レイヤの重みと Affine レイヤの重みの形状だけです。今回の実装では、Encoder がエンコードしたベクトルも入力されるため、重みパラメータの形状がその分大きくなります。

続いて `forward()` の実装です。ここではまず初めに `h` を `np.repeat()` で時系列分だけ複製し、それを `hs` とします。後は、その `hs` を Embedding レイヤの出力と `np.concatenate()` で連結し、これを LSTM レイヤの入力とします。同様に、Affine レイヤでも、`hs` と LSTM レイヤの出力を連結したものに入力とします。



Encoder については、前節から変更はありません。そのため、前節の `Encoder` クラスをそのまま利用します。

最後に、`PeekySeq2seq` を実装します。といっても、これは前節の `Seq2seq` クラスとほとんど同じです。唯一異なる点は、Decoder レイヤです。前節の `Seq2seq` クラスが `Decoder` クラスを使用していたのに対して、今回は `PeekyDecoder` を用います。後のロジックはまったく同じです。そこで、`PeekySeq2seq` クラスの実装は、前章の `Seq2seq` クラスを継承して、初期化部分だけを変更することにします（☞ ch07/peeky_seq2seq.py）。

```

from seq2seq import Seq2seq, Encoder

class PeekySeq2seq(Seq2seq):

```

```

def __init__(self, vocab_size, wordvec_size, hidden_size):
    V, D, H = vocab_size, wordvec_size, hidden_size
    self.encoder = Encoder(V, D, H)
    self.decoder = PeekyDecoder(V, D, H)
    self.softmax = TimeSoftmaxWithLoss()

    self.params = self.encoder.params + self.decoder.params
    self.grads = self.encoder.grads + self.decoder.grads

```

これで準備は整いました。この PeekySeq2seq クラスを使って、足し算問題に再度挑みたいと思います。学習用のコードは前節で示したコードで、Seq2seq クラスを PeekySeq2seq クラスに変更するだけです。

```

# model = Seq2seq(vocab_size, wordvec_size, hidden_size)
model = PeekySeq2seq(vocab_size, wordvec_size, hidden_size)

```

また、ここではひとつ目の改善「Reverse（入力の反転）」も行った上で実験を行います。そうすると結果は図7-28 のようになります。

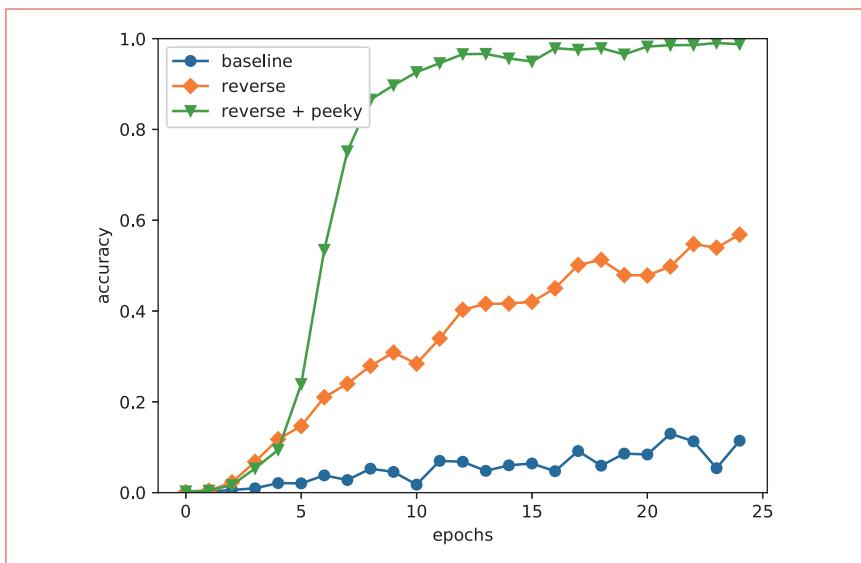


図7-28 「reverse + peeky」が本節の2点の改良を加えた結果

図7-28 に示すとおり、Peeky を加えた seq2seq は格段に良い結果となりました。10 エポックを過ぎたあたりですでに正解率が 90% を超えて、最終的には正解率が

100%に近づきます！

以上の実験結果から、Reverse と Peeky がともに効果的に働いていることが分かります。入力文を反転させる Reverse、そして Encoder の情報を広く行き渡らせる Peeky——この 2 つのテクニックによって、私たちは満足できる結果を手にしました！

ただし、ここでの実験には注意が必要です。というのも、Peeky を用いることによって、私たちのネットワークは重みパラメータが余計に増え、計算量も増加しているのです。そのため、ここでの実験結果は、増加したパラメータ分の“ハンデ”を考慮する必要があります。また、seq2seq の精度は、ハイパーパラメータの調整で大きく変わります。ここでの結果は頗もしいものですが、実際の問題においては、その効果は変わってくるでしょう。

これで seq2seq の改良はひとまず終わりにします。ですが、まだまだ話は続きます。実は、ここで行った改良は“小さな改良”と言えます。次章では、seq2seq に“大きな改良”を加えます。それは Attention と呼ばれるテクニックで、seq2seq を劇的に進化させることができます！

7.5 seq2seq を用いたアプリケーション

seq2seq は、「ある時系列データ」を「別の時系列データ」に変換します。この時系列データを変換するという枠組みはさまざまな問題に適用できます。具体例としては、次のようなタスクが挙げられます。

- 機械翻訳：「ある言語の文章」を「別の言語の文章」に変換する
- 自動要約：「ある長い文章」を「短い要約された文章」に変換する
- 質疑応答：「質問」を「答え」に変換する
- メールの自動返信：「受け取ったメールの文章」を「返信文」に変換する

このように、seq2seq は 2 つの対になった時系列データを扱う問題に利用できます。また、自然言語の他にも、音声や動画像などで利用できます。さらに、一見 seq2seq に当てはめられそうにない問題であっても、入力・出力データの前処理によって seq2seq に適用できる場合があります。ここでは、seq2seq を使ったアプリケーションをいくつか紹介します。それによって、seq2seq の可能性やおもしろさを感じてもらえば幸いです。

7.5.1 チャットボット

チャットボットとは、人とコンピュータがテキストで会話をするプログラムです。すでに Facebook や Twitter など、さまざまなサービスにおいてチャットボットが活用されています。

もちろん、このチャットボットにも seq2seq を使うことができます。というのも、対話というのは「相手の発言」と「自分の発言」によって構成されているため、「相手の発言」を「自分の発言」へと変換する問題として捉えることができるのです。つまり、対話のテキストデータがあれば、それを seq2seq に学習させることができるのです。

さらにチャットボットは、実用的なシーンで利用することができます。文献[43]では、IT のヘルプデスクを対象に、seq2seq ベースのチャットボットを使った実験が行われています。その結果のひとつとして、VPN の接続に関する問題を解決した例が報告されています。ここでは、そこでやり取りされた会話の一部を日本語に翻訳して掲載します（原文は英語での会話のやり取りですが、ここではその会話を筆者が翻訳したものを見します）。

```

機械：こんにちは
人：こんにちは
機械：お使いのOSを教えて頂けますか？
人：linuxを使っています。
.....
機械：実行したときに発生するエラーは何でしょうか？
人：VPNに関するエラーができます。
機械：VPNでネットワークに繋いだ時、エラーメッセージは何と表示されますか？
人：接続ができないか、もしくはそれに似た状態...
....
機械：今は自宅ですか、それとも会社でしょうか？
人：今、自宅にいます。
機械：それでは、<URL>に行って、あなたの名前とパスワードを入力してみて下さい。
人：やりました。
機械：<NAME>でログインして、うまくいか確認して下さい。
人：お、うまくいきました！
機械：良かったです。他に問題はありますか？
人：いいえ、とても助かりました。

```

図 7-29 seq2seq ベースのチャットボットによる会話例（文献[43]より一部抜粋し、英語を日本語に翻訳して掲載）

図7-29の会話をみると、機械（チャットボット）は見事に問題を解決していることが分かります。VPNの接続に困った人を、それを解決できるURLのリンクへと導いたのです。もちろん、これはITのヘルプデスクに関する問題だけを対象としているので、汎用的に使うことはできないでしょう。しかし、会話ベースで答えやヒントを得ることができるというのは、実用性が高く、さまざまな応用が効きそうです。実際に、このようなサービス（の簡易版）は、すでにいくつかのWebサイトに見ることができます。

7.5.2 アルゴリズムの学習

本章で行った実験は、「足し算」のような簡単な問題でした。しかし、原理的には、より高度な問題も扱うことができます。たとえば図7-30で示すようなPythonで書かれたコードを扱うことも可能です。

Input:

```
j=8584
for x in range(8):
    j+=920
    b=(1500+j)
    print((b+7567))
Target: 25011.
```

Input:

```
i=8827
c=(i-5347)
print((c+8704) if 2641<8500 else 5308)
Target: 12184.
```

図7-30 Pythonで書かれたコードの例：図中のInputは入力、Targetは出力（文献[44]より抜粋）

ソースコードも文字で書かれた時系列データです。何行にもわたるコードであっても、それはひとつの文として処理することができます（改行は改行コードとして扱えます）。そのため、ソースコードをそのままseq2seqに入力することができ、目的とする答えを対にして学習させることができます。

上で示したようなfor文やif文が含まれる問題は、一筋縄では解けないでしょう。しかし、そのような問題であってもseq2seqの枠組みで扱うことができるのです。そして、seq2seqの構造を工夫することで、そのような問題も解けるようになることが期待できます。



次章では RNN を拡張した NTM (Neural Turing Machine) というモデルを紹介します。そこでは、コンピュータ（チューリングマシン）がメモリの読み書きの手順を学習し、アルゴリズムを再現します。

7.5.3 イメージキャプション

私たちはこれまでのところ、seq2seq のアプリケーションとしてテキストを扱う例だけを見てきました。しかし、seq2seq はテキストの他にも、画像や音声などさまざまなデータを扱うことができます。ここでは、画像を文章に変換する **イメージキャプション** (Image Captioning) を紹介します [45] [46]。

イメージキャプションは、「画像」を「文章」へと変換します。そしてこれも、図 7-31 に示すように、seq2seq の枠組みで解くことができるのです。

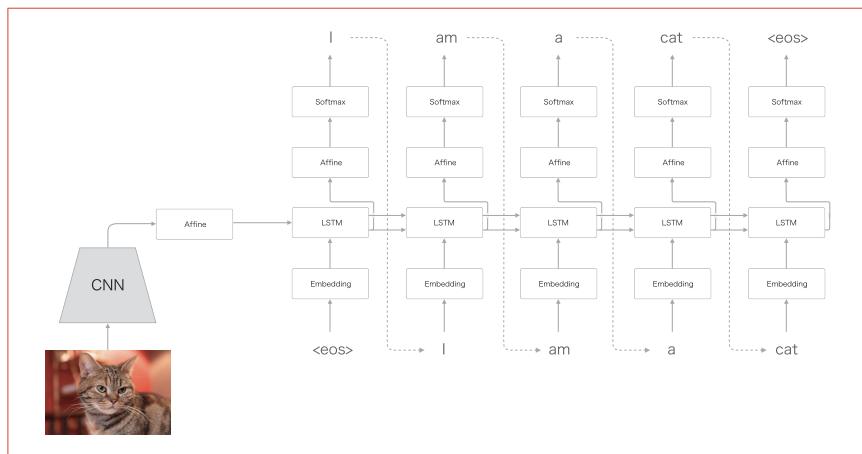


図 7-31 イメージキャプションを行う seq2seq のネットワーク構成例

図 7-31 は、私たちになじみのあるネットワーク構成です。実際、これまでと異なる点は、Encoder が LSTM から CNN (Convolutional Neural Network) に置き換わったことだけです。Decoder はこれまでと同じネットワークが用いられています。たったそれだけの変更で——LSTM から CNN への置き換えで——、seq2seq は画像も扱えるようになります。

図 7-31 の CNN について簡単に補足します。ここでは画像のエンコードを CNN が行います。このとき CNN の最終的な出力は特徴マップです。特徴マップは 3 次元

のボリューム（高さ・幅・チャンネル）であるため、それを Decoder の LSTM が処理できるように、さらに手を加える必要があります。そこで、CNN の特徴マップを平ら（1 次元）にして、全結合の Affine レイヤによって変換します。後は、その変換後のデータを Decoder に渡すことで、これまでどおりの文章生成を行わせることができます。



図 7-31 の CNN には、VGG や ResNet などの実績のあるネットワークを用い、さらにそこで使用する重みは、すでに別の画像データセット（ImageNet など）で学習済みのものを利用します。そうすることで、良いエンコードが得られ、良い文章が生成できるようになります。

それでは、seq2seq によるイメージキャプションを行った例をいくつか見てみましょう。ここでは、im2txt [47] と呼ばれる TensorFlow の実装コードで生成された例を示します。ここで使用されたネットワークは、**図 7-31** をベースとしたものですが、それにいくつかの改良が加えられています。

A person on a beach flying a kite.



A black and white photo of a train on a train track.



A person skiing down a snow covered slope.



A group of giraffe standing next to each other.



図 7-32 イメージキャプションの例：画像をテキストに変換する（画像は文献 [47] より抜粋）

図7-32を見てみると、「A person on a beach flying a kite (ビーチで凧をあげる人)」や「A black and white photo of a train on train track (線路上の電車の白黒写真)」など、とても素晴らしい結果と言えるでしょう。もちろん、これを可能にしているのは、画像と説明文の大量の学習データです（さらにImageNetなどの大量の画像データです）。そして、その学習データを効率良く学ぶことができる seq2seq の存在によって、図7-32のような素晴らしい結果が得られています。

7.6 まとめ

本章では、RNNによる文章生成をテーマに話を進めてきました。実際には、前章までに見てきたRNNを使った言語モデルに少しだけ手を加え、文章生成を行う機能を追加したのです。本章の後半では seq2seq を扱い、簡単な足し算問題を学習させることに成功しました。seq2seq は、Encoder と Decoder を連結したモデルで、それは 2 つの RNN を組み合わせた単純な構造です。しかしその単純さにもかかわらず、seq2seq は非常に大きな可能性を秘めており、さまざまなアプリケーションで利用できます。

また本章では、seq2seq を改良するアイデアを 2 つ紹介し—— Reverse と Peeky——、それらを実装し評価しました。そして、その効果を確認しました。次章では seq2seq をさらに改良していきます。そこに登場するのが、Attention という、ディープラーニングにおいて最も重要なテクニックのひとつです。次章では Attention というメカニズムを説明し、それを実装してさらに強力な seq2seq を実装します。

本章で学んだこと

- RNNを用いた言語モデルは新しい文章を生成することができる
- 文章生成を行う際には、ひとつの単語（もしくは文字）を与え、モデルの出力（確率分布）からサンプリングするという手順を繰り返し行う
- RNNを2つ組み合わせることで、時系列データを別の時系列データに変換することができる（seq2seq）
- seq2seqは、Encoderが入力文をエンコードし、そのエンコード情報をDecoderが受け取り、デコードして目的の出力文を得る
- 入力文を反転させること（Reverse）、またエンコード情報をDecoderの複数のレイヤに与えること（Peeky）は、seq2seqの精度向上に有効である
- 機械翻訳やチャットボット、イメージキャプションなど、seq2seqはさまざまなアプリケーションに利用できる

8章 Attention

注意こそはすべて
——バズワニらの論文タイトル[52]

前章では、RNN を使った文章生成を行いました。そして、2つの RNN を連結することで、時系列データを別の時系列データに変換させることができました。私たちはそれを seq2seq と呼び、足し算のような簡単な問題を解かせることに成功しました。またいくつかの改良を seq2seq に施し、簡単な足し算であれば、ほとんど完璧に解けることを確認しました。

本章では、seq2seq の可能性——そして RNN の可能性——をさらに探索していきます。そしてここに、Attention という強力で美しい技術が登場します。この Attention という技術は、近年のディープラーニングの分野において、間違いなく重要なテクニックのひとつです。私たちが本章で目指すことは Attention の仕組みをコードレベルで理解することです。そして、実際の問題に適用し、その素晴らしい効果を体験したいと思います。それでは最終章へと進みましょう！

8.1 Attention の仕組み

前章で見てきたように、seq2seq は非常に強力な枠組みであり、それにはさまざまな応用が考えられます。ここでは、これまで見てきた seq2seq をさらに強力にする**注意機構** (attention mechanism) というアイデアを紹介します (以降、Attention と略記)。この Attention というメカニズムによって、seq2seq は私たち人間と同じように、必要な情報だけに「注意」を向けさせることができます。さらに、この Attention

を利用することで、これまでの seq2seq が抱えていた問題を解決することができるのです。

本節では手始めに、現在の seq2seq が抱える問題を指摘します。その後に Attention の仕組みを説明しながら、並行してその実装を行いたいと思います。それではもう一度、seq2seq が行う処理に目を向けることにしましょう。



前章でも seq2seq の改良を行いましたが、それはどちらかと言うと“小さな改良”でした。これから説明する Attention という技術は、これまでの seq2seq が抱える根本的な問題を解決する“大きな改良”です。

8.1.1 seq2seq の問題点

seq2seq では、Encoder が時系列データをエンコードします。そして、そのエンコードした情報を Decoder に渡します。このとき、Encoder の出力は「固定長のベクトル」でした。実は、この「固定長」というところに大きな問題が潜んでいます。なぜなら固定長のベクトルというのは、入力文の長さにかかわらず——たとえ、それがどれだけ長くても——、常に同じ長さのベクトルに変換しなければならないからです。前章の翻訳の例を出して説明すれば、図 8-1 のように、どのような文章が入力されたとしても、それを固定長のベクトルに押し込まなければなりません。

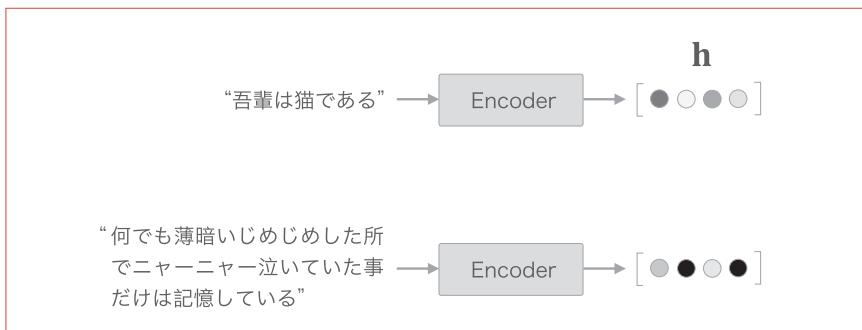


図 8-1 入力文の長さにかかわらず、Encoder は固定長のベクトルに押し込む

現在の Encoder では、どんなに長い文章であっても、それを固定長のベクトルへと変換します。まるでたくさんの洋服をタンスに詰め込むように、Encoder は無理やり固定長のベクトルへと押し込むのです。しかし、それではすぐに限界が訪れます。

結局のところ、洋服がタンスから溢れ出るように、必要な情報はベクトルからはみ出してしまいます。

それでは、seq2seq の改良に取り組みましょう。まずは Encoder の改良を行い、続いて Decoder の改良を行います。

8.1.2 Encoder の改良

これまで私たちは、LSTM レイヤの最後の隠れ状態だけを Decoder に渡しました。しかし、Encoder の出力は、入力される文章の長さに応じて、その長さを変えるべきでしょう。そこが Encoder の改良ポイントです。具体的には図8-2 に示すように、各時刻の LSTM レイヤの隠れ状態ベクトルをすべて利用するのです。

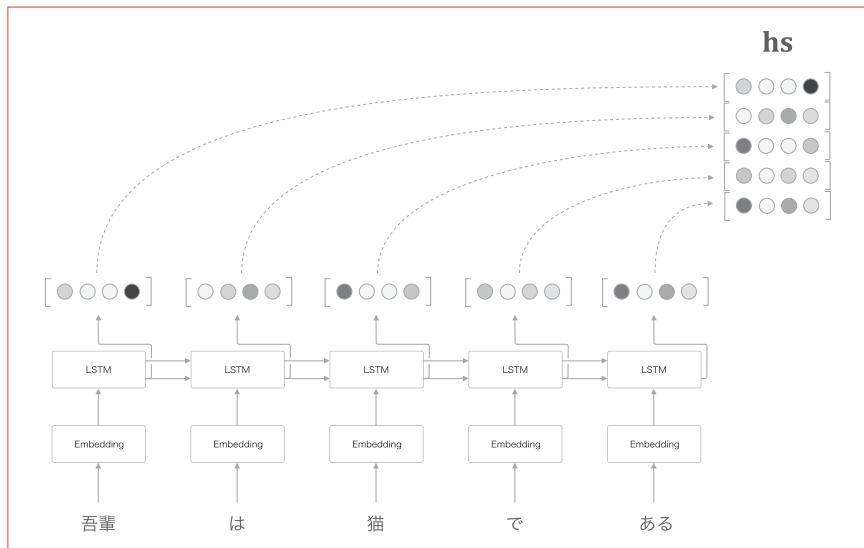


図8-2 Encoder の各時刻（各単語）の LSTM レイヤの隠れ状態をすべて利用する（ここではそれを hs で表す）

図8-2 で示すように、各時刻（各単語）の隠れ状態ベクトルをすべて利用すれば、入力された単語列と同じ数のベクトルを得ることができます。図8-2 の例では、5つの単語が入力されていますが、このとき Encoder は 5 つのベクトルを出力します。これで Encoder は「ひとつの固定長ベクトル」という制約から解放されることになります。



ディープラーニングの多くのフレームワークでは、RNN レイヤ（もしくは LSTM レイヤや GRU レイヤなど）の初期化の際に、「すべての時刻の隠れ状態ベクトルを返す」か、それとも「最後の隠れ状態ベクトルだけを返す」かのどちらかを選択できます。たとえば Keras の場合、RNN レイヤの初期化時に、`return_sequences` という引数に `True` か `False` を与えて、その設定を行います。

さて、図8-2で注目したいのは、LSTM レイヤの隠れ状態の「中身」についてです。このとき、各時刻の LSTM レイヤの隠れ状態にはどのような情報が詰まっているのでしょうか？ひとつ言えることは、各時刻の隠れ状態には、直前に入力された単語の情報が多く含まれているということです。図8-2の例で言えば、たとえば「猫」という単語を入力したときの LSTM レイヤの出力（隠れ状態）は、そのとき入力した「猫」という単語の影響を最も大きく受けることになります。そのため、その隠れ状態ベクトルは“「猫」成分”が多く入ったベクトルになっていると考えられます。そのように考えると、Encoder が output する `hs` という行列は、図8-3のように、各単語に対応したベクトルの集合とみなすことができそうです。

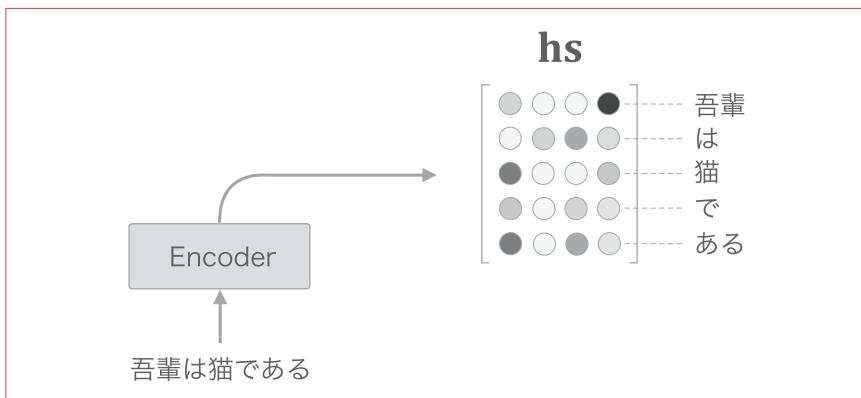


図8-3 Encoder の出力 `hs` は単語の数だけベクトルがあり、それぞれのベクトルは各単語に対応した情報を多く含む



Encoder は左から右方向に処理するため、先ほどの「猫」ベクトルには、正確に言うと「吾輩」「は」「猫」の 3 つの単語の情報が含まれることになります。ここで全体のバランスを考えれば、「猫」という単語の“周囲”的な情報をバランス

ス良く含ませたいと思うかもしれません。そういう場合、時系列データを双方から処理する**双方向 RNN**（もしくは**双方向 LSTM**）が有効です。双方向 RNN については、後ほど説明を行います。ここでは、これまでどおり单方向の LSTM を利用します。

以上が Encoder の改良です。ここで私たちが行った改良は、単に Encoder の隠れ状態をすべての時刻分だけ取り出しました。しかしその小さな修正によって、Encoder は入力文の長さに比例した情報をエンコードできるようになりました。それでは、その Encoder の出力を Decoder はどのように料理すべきでしょうか？ 続いて、Decoder の改良へと進みましょう。なお、Decoder の改良については話す事柄が多いため、3つの節に分けて説明を行います。

8.1.3 Decoder の改良①

Encoder は、各単語に対応する LSTM レイヤの隠れ状態ベクトルを **hs** としてまとめて出力します。そしてこの **hs** が Decoder に渡され、時系列変換が行われます（図8-4）。

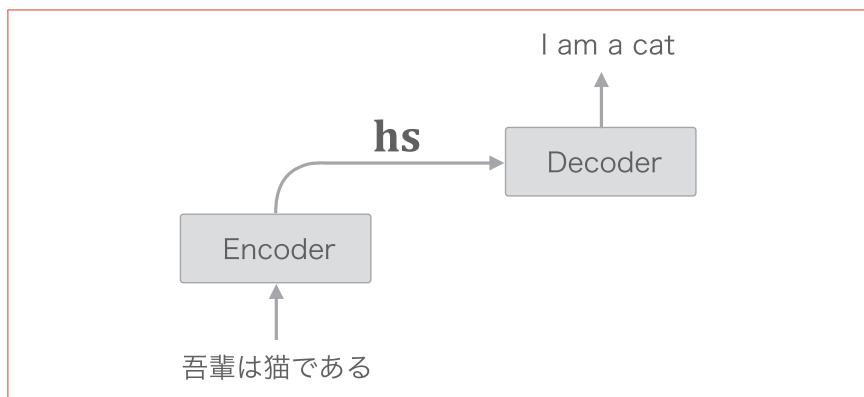


図8-4 Encoder と Decoder の関係

ちなみに、前章で示した最もシンプルな seq2seq では、Encoder の最後の隠れ状態ベクトルだけを Decoder に渡しました。これはより正確に言うと、Encoder の LSTM レイヤの「最後」の隠れ状態を、Decoder の LSTM レイヤの「最初」の隠れ状態に代入したのです。実際に図で表すと、Decoder のレイヤ構成は図8-5 のよう

に書けます。

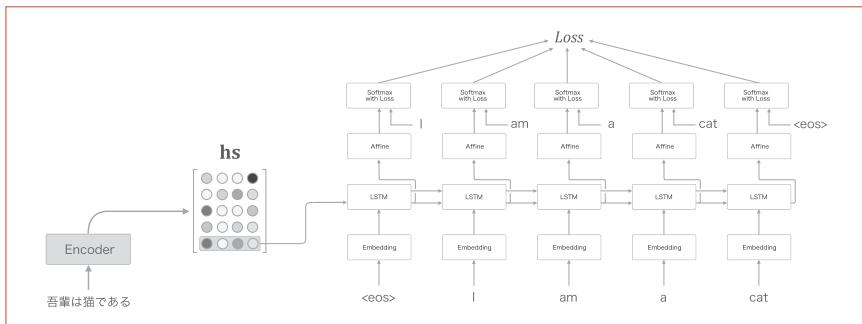


図8-5 前章のDecoderのレイヤ構成（学習時）

図8-5に示すとおり、前章のDecoderは、EncoderのLSTMレイヤにある最後の隠れ状態だけを利用しました。記号 **hs** を使えば、その最後の行だけ抜き出して、それをDecoderに渡すことになります。それでは、この **hs** すべてを活用できるように、Decoderの改良へと進みましょう。

さて、私たちが翻訳を行うとき、頭の中でどのようなことを行っているでしょうか。たとえば、「吾輩は猫である」という文を英語に翻訳するときはどうでしょう。きっと、「吾輩 = I」や「猫 = cat」というような知識を利用していいるはずです。つまり、私たちは「ある単語（もしくは単語の集まり）」に注目して、その単語の変換を隨時行っていると考えられます。それではこれと同じようなことを、私たちの seq2seq で再現できないでしょうか？より正確に言えば、「入力と出力でどの単語が関連しているのか」という対応関係を、seq2seq に学習させることはできないでしょうか？



機械翻訳の歴史において、「猫 = cat」のような単語の対応関係の知識を利用する研究は多く行われてきました。そのような単語（もしくはフレーズ）の対応関係を表す情報は、**アライメント**（alignment）と呼ばれます。これまでアライメントは主に人手によって作られてきました。しかし、これから説明する Attention という技術は、アライメントのアイデアを seq2seq に自動で取り入れることに成功しました。ここでも、「人手による作業」から「機械による自動化」の流れが進められています。

これから私たちの目指すべきところは、「翻訳先の単語」と対応関係にある「翻訳

元の単語」の情報を選び出すこと、そして、その情報を利用して翻訳を行うことです。つまり、必要な情報だけに注意を向けさせ、その情報から時系列変換を行うことを目指します。この仕組みは Attention と呼ばれ、これが本章のメインテーマとなります。

ここでは Attention の詳細に入る前に、まずは全体の枠組みを示したいと思います。これから私たちが実現したいネットワークは、レイヤ構成で示すと図8-6 のようになります。

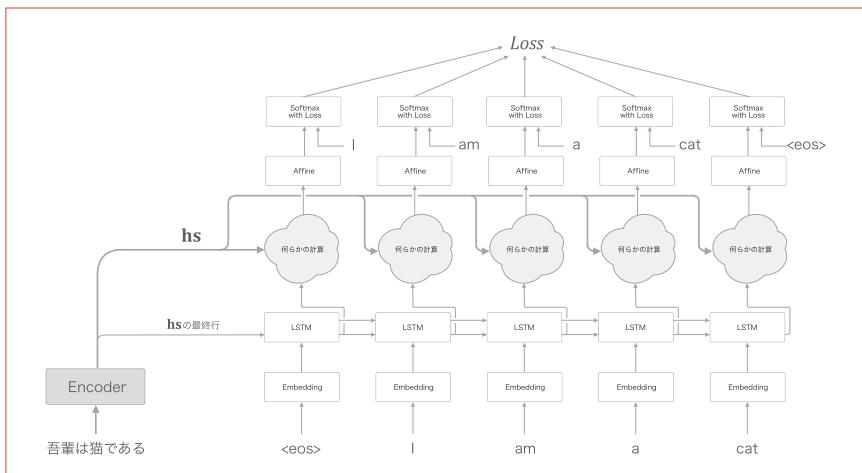


図8-6 改良後の Decoder のレイヤ構成

図8-6 に示すように、ここでは新たに「何らかの計算」を行うレイヤを追加することを考えます。この「何らかの計算」は、各時刻において LSTM レイヤの隠れ状態と Encoder からの hs を受け取ります。そしてそこから必要な情報だけを選び出し、それをその先の Affine レイヤへと出力します。なお、これまでどおり Encoder の最後の隠れ状態ベクトルは、Decoder の最初の LSTM レイヤに渡すことになります。

さて、図8-6 のネットワークで行いたいことは、単語のアライメント抽出です。これは具体的に言うと、各時刻において Decoder への入力単語と対応関係にある単語のベクトルを hs から選び出すことです。たとえば、図8-6 の Decoder が「I」を出力するとき、hs の「吾輩」に対応するベクトルを選び出すといったようなことを行わせたいのです。つまり、そのような選び出すという操作を「何らかの計算」で実現

したいことになります。しかしここで問題が発生します。その問題とは、選び出すという操作——複数のものから（いくつかを）選ぶという操作——は、微分ができないということです。



ニューラルネットワークの学習は、（一般的に）誤差逆伝播法によって行います。そのため、微分可能な演算によってニューラルネットワークを構築すれば、誤差逆伝播法の枠組みの中で学習を行うことができます。しかし、微分可能な演算を用いなければ、（基本的には）誤差逆伝播法を使うことはできません。

「選ぶ」という操作を微分可能な演算に置き換えることはできないでしょうか。実は、これを解決するアイデアはとても単純です（しかし、コロンブスの卵のように最初に思いつくのは困難でしょう）。そのアイデアとは、「ひとつを選ぶ」のではなく、「すべてを選ぶ」ようにするのです。そしてこのとき、図8-7のように、各単語の重要度（貢献度）を表す「重み」を別途計算するようにします。

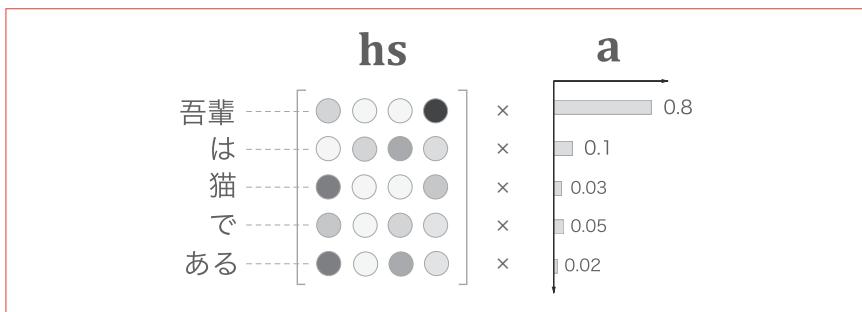


図8-7 各単語に対して、それがどれだけ重要であるかを表す「重み」を求める（どのように求めるかは後述）

図8-7に示すように、ここでは各単語の重要度を表す「重み」（記号 a で表す）を利用します。このとき、 a は確率分布と同じように、各要素は 0.0~1.0 のスカラ（单一の要素）であり、その総和は 1 になります。そして、この各単語の重要度を表す重み a と各単語のベクトル hs から、その重み付き和を求ることで、目的とするベクトルを得ます。この一連の計算を図で表すと図8-8のようになります。

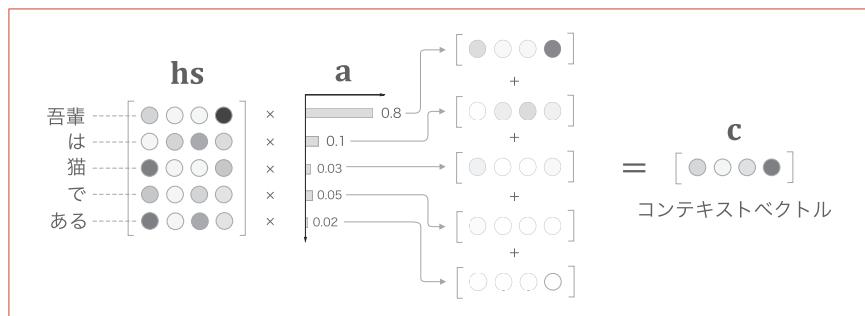


図8-8 重み付き和の計算によって「コンテキストベクトル」を算出する

図8-8に示すように、単語ベクトルの重み付き和を計算します。ここでは、その結果を「コンテキストベクトル」と呼び、 c という記号で表すことにします。ところで、図8-8をよく見ると、「吾輩」に対応する重みが0.8になっています。これが意味することは、コンテキストベクトルの c には「吾輩」ベクトルの成分が多く含まれているということです。つまり、「吾輩」ベクトルを“選ぶ”ような操作を、この重み付き和で代替していると言えます。仮に「吾輩」に対応する重みが1でそれ以外が0であれば、これは「吾輩」ベクトルを“選ぶ”ことに相当します。



コンテキストベクトル c には、現時刻の変換（翻訳）を行うために必要な情報が含まれています。より正確に言うならば、そうなるようにデータから学習するのです。

それでは、ここまで話をコードベースで見てていきましょう。ここでは、Encoderが output する hs と各単語の重み a を適当に作成し、その重み付き和を求める実装を示します。多次元配列の形状に注目しながら見ていくと次のようになります。

```
import numpy as np

T, H = 5, 4
hs = np.random.randn(T, H)
a = np.array([0.8, 0.1, 0.03, 0.05, 0.02])

ar = a.reshape(5, 1).repeat(4, axis=1)
print(ar.shape)
# (5, 4)

t = hs * ar
```

```
print(t.shape)
# (5, 4)

c = np.sum(t, axis=0)
print(c.shape)
# (4,)
```

ここでは、時系列の長さを $T = 5$ 、隠れ状態ベクトルの要素数を $H = 4$ として、重み付き和を求める過程を示しています。まずは上のコードの `ar = a.reshape(5, 1).repeat(4, axis=1)` という箇所に注目しましょう。このコードは、図8-9のように `a` を `ar` へと変換します。

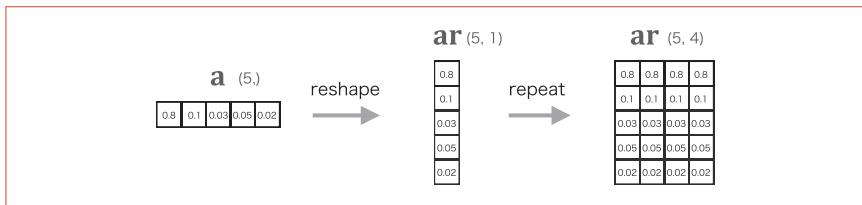


図8-9 `reshape()` と `repeat()` メソッドによって、`a` から `ar` を生成（変数の右隣にその形状を示す）

図8-9 のとおり、ここで私たちがやりたいことは形狀が $(5,)$ の `a` をコピーして、 $(5, 4)$ の配列を作ることです。そのために `a` の形狀が $(5,)$ であるところを、`a.reshape(5, 1)` によって $(5, 1)$ の形狀へと変形します。そして、この整形した配列の 1 軸目を 4 回繰り返すことで、形狀が $(5, 4)$ の配列を生成します。



`repeat()` メソッドは、多次元配列の要素をコピーして新しい多次元配列を生成します。これは、`x` という NumPy の多次元配列があったとき、`x.repeat(rep, axis)` のように使うことができます。ここで、引数の `rep` はコピーを繰り返す回数、`axis` は繰り返しを行う軸（次元）を指定します。たとえば、`x` の形狀が (X, Y, Z) のとき `x.repeat(3, axis=1)` とすると、`x` の 1 軸方向（1 次元方向）へのコピーが行われ、形狀が $(X, 3*Y, Z)$ の多次元配列が生成されます。

なお、ここでは `repeat()` メソッドは使わずに、NumPy のプロードキャストを使うことも可能です。その場合、`ar = a.reshape(5, 1)` として、`hs * ar` の計算を行います。このとき図8-10 のように、`ar` が `hs` の形狀と合致するように自動で拡張

されます。

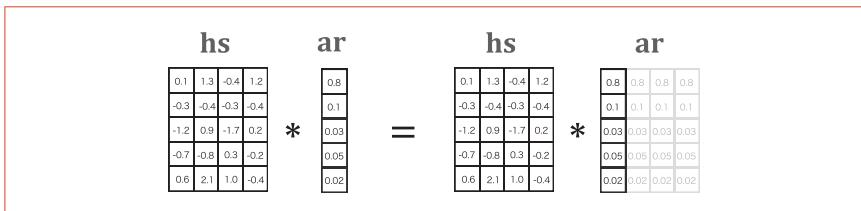


図8-10 NumPy のブロードキャスト

実装の効率を考えると、ここでは `repeat()` メソッドは使用せずに、NumPy のブロードキャストを利用すべきでしょう。ただしその場合、私たちの見えないところで多次元配列の要素がコピー（リピート）されているということに気をとめましょう。「1.3.4.3 Repeat ノード」によれば、これは計算グラフで Repeat ノードに相当します。そのため、逆伝播を求めるときには Repeat ノードの逆伝播を行う必要があります。

図8-10 のように要素ごとの積を計算したら、後は `c = np.sum(hs*ar, axis=0)` によって和を求めます。ここで引数に `axis` とありますが、この引数 `axis` によって、どの軸（次元）方向に和を取るかを指定できます。ここでも配列の形状に注目することで、`axis` の使い方が明瞭になります。たとえば、`x` の形状が (X, Y, Z) のとき、`np.sum(x, axis=1)` とすると、その出力（和）の形状は (X, Z) となります。このとき重要な点は、1 軸目が“消える”ように和が求められるということです。上の例では `hs*ar` の形状は $(5, 4)$ であり、0 軸目を“消す”ことで、 $(4,)$ の形状の行列（ベクトル）が求められます。



重み付き和の計算は「行列の積」を使うのが最も簡単で効率の良いやり方です。上の例で言えば、`np.dot(a, hs)` の 1 行だけで目的の結果を得ることができます。ただし、それはひとつのデータ（サンプル）に対しての処理になります。残念ながら、それをバッチ処理へと拡張することは容易ではありません。もしやろうと思えば「テンソル積」の計算を行う必要があり、話が少し複雑になります（その場合は、`np.tensordot()` や `np.einsum()` メソッドを使います）。ここでは行列の積を用いずに、分かりやすさを優先して、`repeat()` と `sum()` メソッドによって重み付き和を求める実装を行います。

続いて、バッチ処理版の重み付き和の実装を行います。これは次のように実装することができます（ここでは、`hs` と `a` をランダムに生成することにします）。

```
N, T, H = 10, 5, 4
hs = np.random.randn(N, T, H)
a = np.random.randn(N, T)
ar = a.reshape(N, T, 1).repeat(H, axis=2)
# ar = a.reshape(N, T, 1) # ブロードキャストを使う場合

t = hs * ar
print(t.shape)
# (10, 5, 4)

c = np.sum(t, axis=1)
print(c.shape)
# (10, 4)
```

ここでのバッチ処理は、前の実装とほとんど同じです。配列の形状に注意すれば、`repeat()` と `sum()` でどの次元（軸）を指定するかはすぐに分かるでしょう。それでは、ここまでまとめも兼ねて、重み付き和の計算を「計算グラフ」で表すことにします。

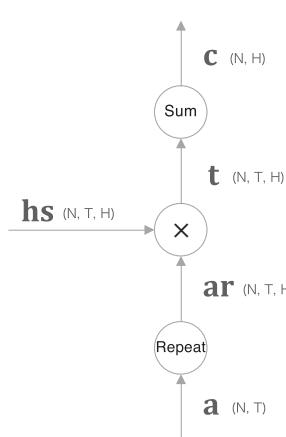


図8-11 重み付き和の計算グラフ

図8-11に示すとおり、ここでは `Repeat` ノードを用い、`a` のコピーを行います。後は「`×`」ノードで要素ごとの積を計算し、`Sum` ノードで和を求めます。それでは、

この計算グラフの逆伝播について考えましょう。といっても、すでに必要な知識はすべて出揃っています。Repeat ノード、Sum ノードの逆伝播については、第 1 章で説明しました。要点だけを述べれば、「Repeat の逆伝播は Sum」、「Sum の逆伝播は Repeat」になります。テンソルの形状に注意すれば、どの軸に対しての Sum か、どの軸に対しての Repeat かということもすぐに分かるでしょう。

それでは、図 8-11 の計算グラフをレイヤとして実装することにします。ここではそれを Weight Sum レイヤと呼ぶことにして、その実装を次に示します (☞ ch08/attention_layer.py)。

```
class WeightSum:
    def __init__(self):
        self.params, self.grads = [], []
        self.cache = None

    def forward(self, hs, a):
        N, T, H = hs.shape

        ar = a.reshape(N, T, 1).repeat(H, axis=2)
        t = hs * ar
        c = np.sum(t, axis=1)

        self.cache = (hs, ar)
        return c

    def backward(self, dc):
        hs, ar = self.cache
        N, T, H = hs.shape

        dt = dc.reshape(N, 1, H).repeat(T, axis=1)  # sum の逆伝播
        dar = dt * hs
        dhs = dt * ar
        da = np.sum(dar, axis=2)  # repeat の逆伝播

        return dhs, da
```

これがコンテキストベクトルを求める Weight Sum レイヤの実装です。このレイヤは学習するパラメータは持たないので、本書の実装ルールに従い `self.params = []` とします。後は特に難しい点はないはずです。それでは次へと進みましょう。

8.1.4 Decoder の改良②

各単語の重要度を表す重み a があれば、重み付き和によって「コンテキストベクトル」を得ることができます。では、この a はどのように求めればよいでしょうか？

もちろん、人の手によって指定するようなことはしません。データから自動で学習できるように、手はずを整えるのです。

それでは、各単語の重み \mathbf{a} を求める方法を見ていきましょう。その説明にあたって、まずは Decoder の最初のステップ（時刻）で LSTM レイヤが隠れ状態ベクトルを出力するまでの処理を図 8-12 に示します。

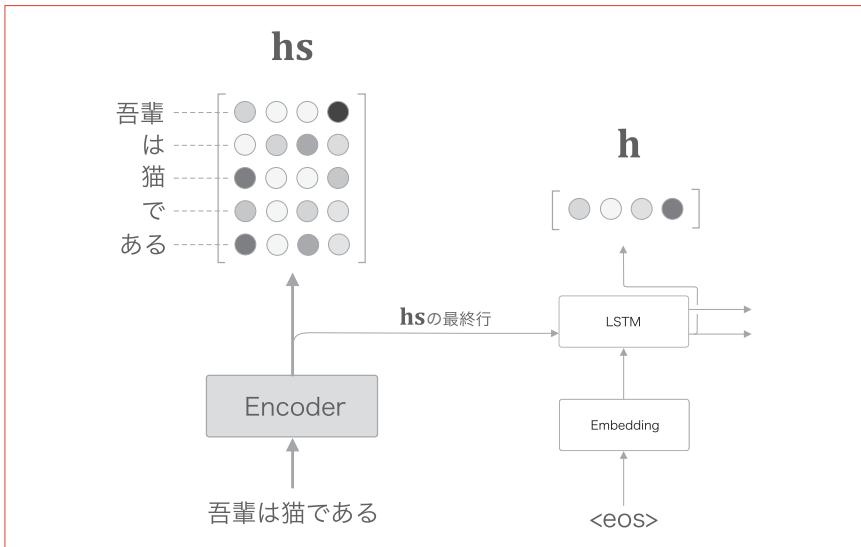


図 8-12 Decoder の最初の LSTM レイヤの隠れ状態ベクトル

図 8-12 では、Decoder の LSTM レイヤの隠れ状態ベクトルを \mathbf{h} で表しています。このとき私たちの目指すところは、この \mathbf{h} が \mathbf{hs} の各単語ベクトルとどれだけ“似ているか”を数値で表すことです。そのための方法はいくつか考えられますが、ここでは最も単純な方法である、ベクトルの「内積」を利用したいと思います。ちなみに、内積は、 $\mathbf{a} = (a_1, a_2, \dots, a_n)$ と $\mathbf{b} = (b_1, b_2, \dots, b_n)$ のベクトルに対して、次のように表されます。

$$\mathbf{a} \cdot \mathbf{b} = a_1 b_1 + a_2 b_2 + \cdots + a_n b_n \quad (8.1)$$

内積は式 (8.1) で表されますが、その直感的な意味は、2つのベクトルがどれだけ同じ方向を向いているかということです。そのため、2つのベクトルの「類似度」と

して、内積を用いることは自然な選択と言えます。



ベクトルの類似度を計算する方法は、内積以外にもいくつか考えられます。内積の他に、スコアを出力するために小さなニューラルネットワークを使うケースも見受けられます。文献[49]では、スコアを出力するための方法がいくつか提案されています。

それでは、内積によってベクトル間の類似度を算出するまでの処理を図で表しましょう（図8-13）。

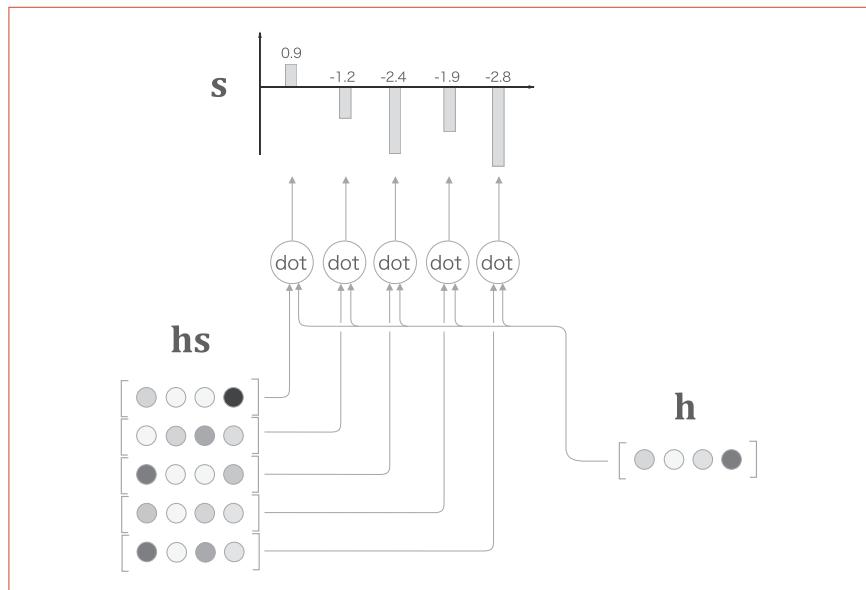


図8-13 内積により、 hs の各行と h の類似度を算出する（内積は dot ノードで図示する）

図8-13 で示すように、ここではベクトルの内積によって h と hs の各単語ベクトルとの類似度を算出します。そして、その結果を s で表すことになります。なお、この s は正規化前の値であり、「スコア」と呼ぶこともあります。続いて、 s を正規化するために、お決まりの Softmax 関数を適用します（図8-14）。

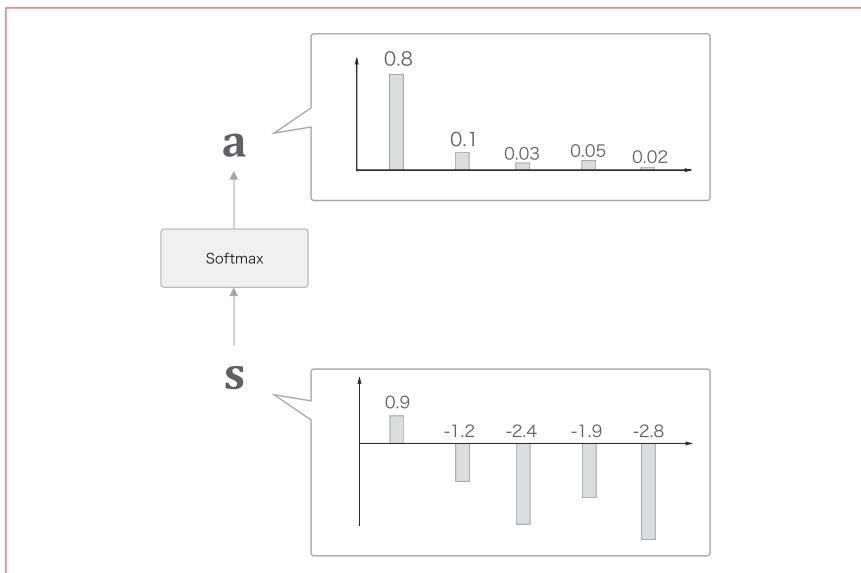


図 8-14 Softmax による正規化

Softmax 関数を用いることで、その出力である a の各要素は $0.0 \sim 1.0$ の値になり、その総和は 1 になります。以上で、各単語の重みを表す a を求めることができます。それでは、ここまで処理をコードベースで見ていきましょう。

```
import sys
sys.path.append('..')
from common.layers import Softmax
import numpy as np

N, T, H = 10, 5, 4
hs = np.random.randn(N, T, H)
h = np.random.randn(N, H)
hr = h.reshape(N, 1, H).repeat(T, axis=1)
# hr = h.reshape(N, 1, H) # ブロードキャストの場合

t = hs * hr
print(t.shape)
# (10, 5, 4)

s = np.sum(t, axis=2)
print(s.shape)
# (10, 5)
```

```
softmax = Softmax()
a = softmax.forward(s)
print(a.shape)
# (10, 5)
```

ここではミニバッチ処理を行う際のコードを示しています。前に説明したとおり、ここでも、`reshape()` と `repeat()` メソッドによって適切な形状の `hr` を生成します。なお、NumPy のブロードキャストを使う場合は、`repeat()` は必要ありません。それでは、このときの計算グラフを図8-15に示します。

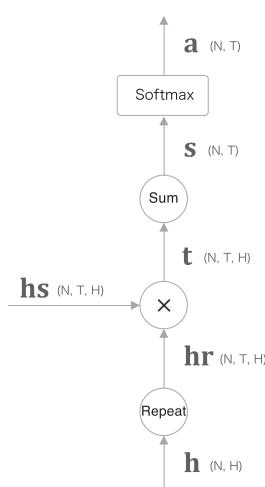


図8-15 各単語の重みを求める計算グラフ

図8-15のとおり、ここで計算グラフは、Repeat ノード、要素ごとの積の「×」ノード、Sum ノード、そして、Softmax レイヤによって構成されます。私たちは、この計算グラフで表される処理を `AttentionWeight` クラスとして実装することにします（☞ `ch08/attention_layer.py`）。

```
import sys
sys.path.append('..')
from common.np import * # import numpy as np
from common.layers import Softmax

class AttentionWeight:
```

```

def __init__(self):
    self.params, self.grads = [], []
    self.softmax = Softmax()
    self.cache = None

def forward(self, hs, h):
    N, T, H = hs.shape

    hr = h.reshape(N, 1, H).repeat(T, axis=1)
    t = hs * hr
    s = np.sum(t, axis=2)
    a = self.softmax.forward(s)

    self.cache = (hs, hr)
    return a

def backward(self, da):
    hs, hr = self.cache
    N, T, H = hs.shape

    ds = self.softmax.backward(da)
    dt = ds.reshape(N, T, 1).repeat(H, axis=2)
    dhs = dt * hr
    dhr = dt * hs
    dh = np.sum(dhr, axis=1)

    return dhs, dh

```

この実装においても、前回の Weight Sum レイヤの実装と同様に、Repeat と Sum の演算が登場します。逆伝播についてはその 2 つの演算に注意すれば、他は難しい点はないでしょう。それでは、Decoder の改善の最後のステップへと進みます。

8.1.5 Decoder の改良③

ここまで Decoder の改善策を 2 つの節に分けて説明してきました。8.1.4 節では Attention Weight レイヤを、8.1.3 節では Weight Sum レイヤをそれぞれ実装しました。それでは、その 2 つのレイヤを組み合わせてみましょう。その結果は図 8-16 のようになります。

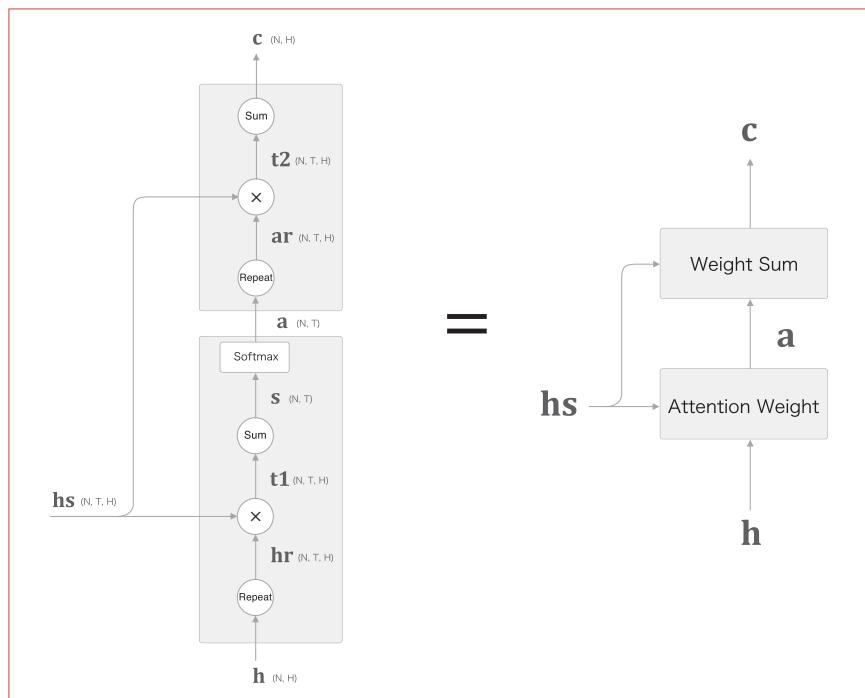


図8-16 コンテキストベクトルの算出を行う計算グラフ

図8-16がコンテキストベクトル c を求める計算グラフの全貌です。私たちはその計算を Weight Sum レイヤと Attention Weight レイヤの2つに分けて実装しました。繰り返しになりますが、ここで行う計算では、Encoder が出力する各単語のベクトル hs に対して Attention Weight レイヤが注意を払い、各単語の重み a を求めます。それに続き、Weight Sum レイヤが a と hs の重み付き和を求め、それをコンテキストベクトル c として出力します。私たちは、この一連の計算を行うレイヤを Attention レイヤと呼ぶことにします（図8-17）。

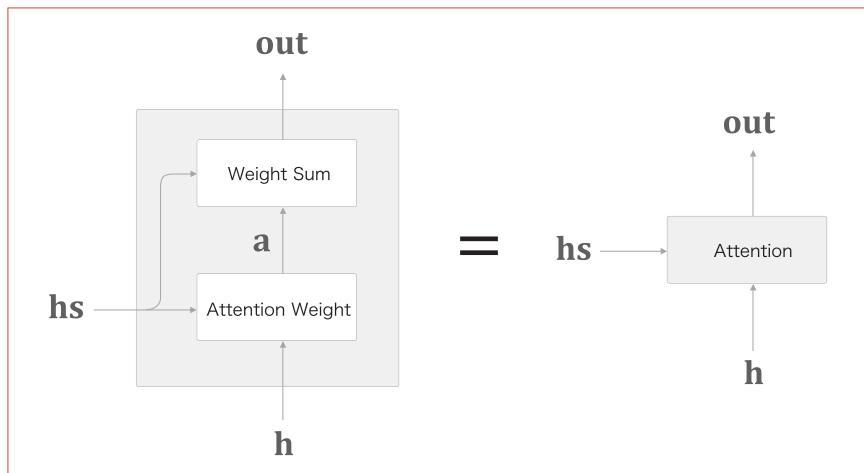


図 8-17 左の計算グラフを Attention レイヤとしてまとめる

以上が Attention という技術の核心です。Encoder が渡す情報 hs の重要な要素に注意を払い、それを元にコンテキストベクトルを算出し、それを上層へと伝播します（私たちの場合、上層には Affine レイヤが待ち受けています）。それでは Attention レイヤの実装を次に示します（☞ ch08/attention_layer.py）。

```
class Attention:
    def __init__(self):
        self.params, self.grads = [], []
        self.attention_weight_layer = AttentionWeight()
        self.weight_sum_layer = WeightSum()
        self.attention_weight = None

    def forward(self, hs, h):
        a = self.attention_weight_layer.forward(hs, h)
        out = self.weight_sum_layer.forward(hs, a)
        self.attention_weight = a
        return out

    def backward(self, dout):
        dhs0, da = self.weight_sum_layer.backward(dout)
        dhs1, dh = self.attention_weight_layer.backward(da)
        dhs = dhs0 + dhs1
        return dhs, dh
```

ここでは、2つのレイヤ——Weight Sum レイヤと Attention Weight レイヤ——

による順伝播と逆伝播を行うだけです。このとき、各単語の重みを後ほど参照できるようにするため、`attention_weight` というメンバ変数を設定します。以上で Attention レイヤの実装は終わりです。この Attention レイヤを、私たちは LSTM レイヤと Affine レイヤの間に挿入します。これは図で表すと、図 8-18 のようになります。

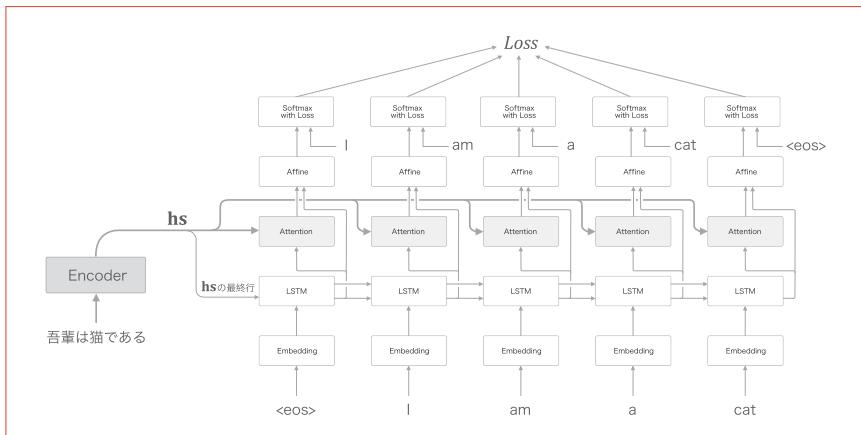


図 8-18 Attention レイヤを備えた Decoder のレイヤ構成

図 8-18 に示すとおり、各時刻の Attention レイヤには、Encoder の出力である `hs` が入力されます。またここでは、LSTM レイヤの隠れ状態ベクトルを Affine レイヤへ入力します。これは、前章の Decoder からの改良を考えると自然な拡張と言えるでしょう。というのも、図 8-19 で示すように、前章の Decoder に対して、Attention の情報を“追加”することになるからです。

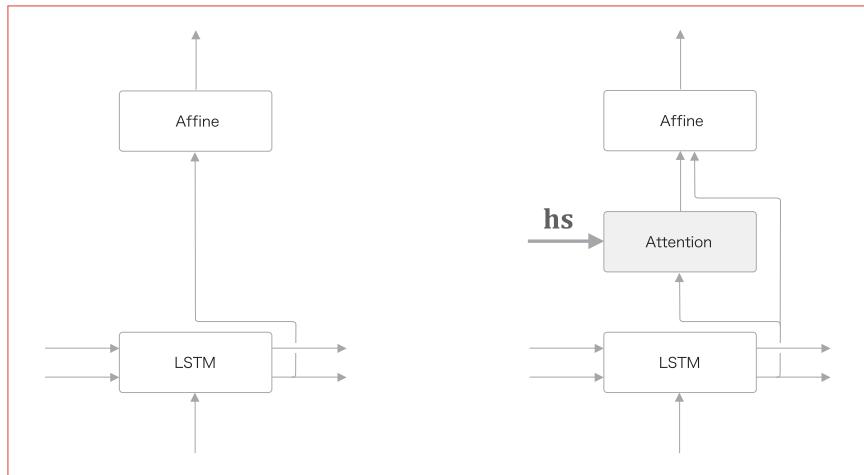


図8-19 前章のDecoder（左図）とAttention付きのDecoder（右図）の比較：LSTMレイヤからAffineレイヤまでの部分をピックアップして描画

図8-19に示すように、私たちは前章のDecoderに対して、Attentionレイヤによるコンテキストベクトルの情報を“追加”することを考えます。そのため、Affineレイヤへは、これまでどおりのLSTMレイヤの隠れ状態ベクトルを与え、それに追加してAttentionレイヤのコンテキストベクトルも入力することにします。



図8-19ではAffineレイヤへ「コンテキストベクトル」と「隠れ状態ベクトル」の2つのベクトルが入力されています。これは前にも述べたとおり、その2つのベクトルを連結し、その連結後のベクトルがAffineレイヤへと入力されることを意味します。

最後に、図8-18の時系列方向に広がった複数のAttentionレイヤをTime Attentionレイヤとしてまとめて実装することにします。これは図で表すと図8-20のようになります。

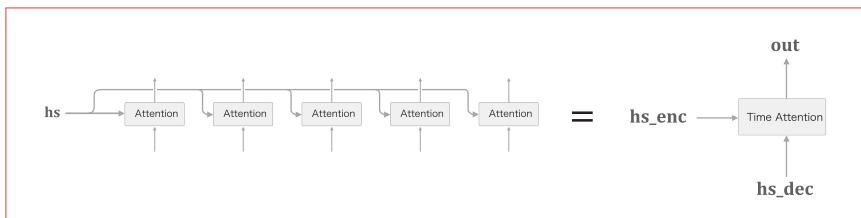


図8-20 複数のAttentionレイヤをTime Attentionレイヤとしてまとめて実装する

図8-20に示すとおり、Time Attention レイヤは、複数のAttention レイヤをまとめただけです。それでは、Time Attention レイヤの実装を次に示します（[ch08/attention_layer.py](#)）。

```

class TimeAttention:
    def __init__(self):
        self.params, self.grads = [], []
        self.layers = None
        self.attention_weights = None

    def forward(self, hs_enc, hs_dec):
        N, T, H = hs_dec.shape
        out = np.empty_like(hs_dec)
        self.layers = []
        self.attention_weights = []

        for t in range(T):
            layer = Attention()
            out[:, t, :] = layer.forward(hs_enc, hs_dec[:, t, :])
            self.layers.append(layer)
            self.attention_weights.append(layer.attention_weight)

        return out

    def backward(self, dout):
        N, T, H = dout.shape
        dhs_enc = 0
        dhs_dec = np.empty_like(dout)

        for t in range(T):
            layer = self.layers[t]
            dhs, dh = layer.backward(dout[:, t, :])
            dhs_enc += dhs
            dhs_dec[:, t, :] = dh

        return dhs_enc, dhs_dec

```

ここでは、必要な数だけ Attention レイヤを作成し（コードでは T 個）、それぞれが順伝播と逆伝播を行います。また、各 Attention レイヤの各単語への重みを `attention_weights` としてリストで持つようにします。

以上で Attention という仕組みの説明、およびその実装は終わりです。続いてこの Attention を使って、seq2seq の実装を行います。そして、現実的な問題に挑み、Attention の効果を確かめたいと思います。

8.2 Attention 付き seq2seq の実装

前節で Attention レイヤ（および Time Attention レイヤ）の実装が終わりました。ここでは、そのレイヤを使って「Attention 付きの seq2seq」の実装を行います。実際には、前章と同じく 3 つのクラス——Encoder と Decoder、そして seq2seq——を実装しますが、その 3 つのクラスは `AttentionEncoder`、`AttentionDecoder`、`AttentionSeq2seq` クラスとしてそれぞれ実装します。

8.2.1 Encoder の実装

まずは `AttentionEncoder` クラスの実装から始めます。このクラスは前章で実装した `Encoder` クラスとほとんど同じです。前章の `Encoder` クラスの `forward()` メソッドは、LSTM レイヤの最後の隠れ状態ベクトルだけを返しました。それに対して、今回はすべての隠れ状態を返します。そこが唯一の異なる点です。そのため、ここでは前章の `Encoder` を継承して実装することにします。それでは、`AttentionEncoder` クラスの実装を次に示します（➡ ch08/attention_seq2seq.py）。

```
import sys
sys.path.append('..')
from common.time_layers import *
from ch07.seq2seq import Encoder, Seq2seq
from ch08.attention_layer import TimeAttention

class AttentionEncoder(Encoder):
    def forward(self, xs):
        xs = self.embed.forward(xs)
        hs = self.lstm.forward(xs)
        return hs

    def backward(self, dhs):
```

```
dout = self.lstm.backward(dhs)
dout = self.embed.backward(dout)
return dout
```

8.2.2 Decoder の実装

続いて、Attention レイヤを利用した Decoder の実装です。Attention を用いた Decoder のレイヤ構成は、図8-21 のようになります。

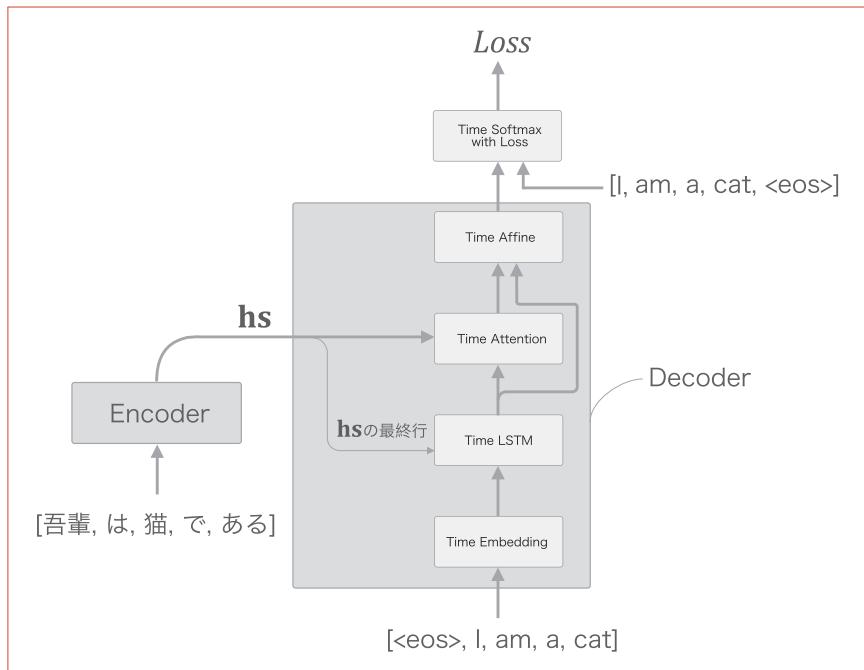


図8-21 Decoder のレイヤ構成

図8-21 のとおり、前章の実装と同じく Softmax レイヤ（正確には Time Softmax with Loss レイヤ）の手前までを Decoder として実装することにします。また前章同様、順伝播の `forward()` と逆伝播の `backward()` メソッドに加えて、新しい単語列（もしくは文字列）を生成するための `generate()` メソッドも実装します。ここでは Attention Decoder レイヤのイニシャライザと `forward()` メソッドの実装のみを次に示します（☞ ch08/attention_seq2seq.py）。

```

class AttentionDecoder:
    def __init__(self, vocab_size, wordvec_size, hidden_size):
        V, D, H = vocab_size, wordvec_size, hidden_size
        rn = np.random.randn

        embed_W = (rn(V, D) / 100).astype('f')
        lstm_Wx = (rn(D, 4 * H) / np.sqrt(D)).astype('f')
        lstm_Wh = (rn(H, 4 * H) / np.sqrt(H)).astype('f')
        lstm_b = np.zeros(4 * H).astype('f')
        affine_W = (rn(2 * H, V) / np.sqrt(2 * H)).astype('f')
        affine_b = np.zeros(V).astype('f')

        self.embed = TimeEmbedding(embed_W)
        self.lstm = TimeLSTM(lstm_Wx, lstm_Wh, lstm_b, stateful=True)
        self.attention = TimeAttention()
        self.affine = TimeAffine(affine_W, affine_b)
        layers = [self.embed, self.lstm, self.attention, self.affine]

        self.params, self.grads = [], []
        for layer in layers:
            self.params += layer.params
            self.grads += layer.grads

    def forward(self, xs, enc_hs):
        h = enc_hs[:, -1]
        self.lstm.set_state(h)

        out = self.embed.forward(xs)
        dec_hs = self.lstm.forward(out)
        c = self.attention.forward(enc_hs, dec_hs)
        out = np.concatenate((c, dec_hs), axis=2)
        score = self.affine.forward(out)

    return score

    def backward(self, dscore):
        # ソースコードを参照

    def generate(self, enc_hs, start_id, sample_size):
        # ソースコードを参照

```

ここでの実装は、Time Attention レイヤが新たに使われていることを除けば、前章の Decoder クラスとは大きくは変わりません。注意点としては、forward() メソッドにおいて、Time Attention レイヤの出力と LSTM レイヤの出力を結合していることです。上のコードでは、その結合に np.concatenate() メソッドを使用しています。

AttentionDecoder クラスの backward() と generate() メソッドの実装に

については、説明は省略します。それでは最後に、AttentionEncoder クラスと AttentionDecoder クラスを使用して AttentionSeq2seq クラスを実装します。

8.2.3 seq2seq の実装

AttentionSeq2seq クラスの実装も前章で実装した seq2seq とほとんど同じです。異なる点は、Encoder に AttentionEncoder クラスを、Decoder に Attention Decoder クラスを使うことだけです。そのため、前章の Seq2seq クラスを継承して、イニシャライザを変更するだけで AttentionSeq2seq クラスが実装できます（☞ ch08/attention_seq2seq.py）。

```
from ch07.seq2seq import Encoder, Seq2seq

class AttentionSeq2seq(Seq2seq):
    def __init__(self, vocab_size, wordvec_size, hidden_size):
        args = vocab_size, wordvec_size, hidden_size
        self.encoder = AttentionEncoder(*args)
        self.decoder = AttentionDecoder(*args)
        self.softmax = TimeSoftmaxWithLoss()

        self.params = self.encoder.params + self.decoder.params
        self.grads = self.encoder.grads + self.decoder.grads
```

以上で、Attention 付きの seq2seq の実装は終わりです。

8.3 Attention の評価

それでは、前節で実装した AttentionSeq2seq クラスを使って、実践的な問題に挑戦したいと思います。ここで本来であれば、翻訳のような問題に取り組み、Attention の効果を確認したいところです。しかし残念ながら、翻訳用のデータセットには手ごろなサイズのものがなかなか見当たりません。そこで本書では、「日付フォーマット」を変更する問題——データのサイズが小さく、どちらかと言うと人工的な問題——を取り組み、Attention 付き seq2seq の効果を確かめたいと思います。



翻訳用のデータセットでは、「WMT」と呼ばれる翻訳データセットが有名です。このデータセットには、英語とフランス語（または英語とドイツ語）の学習データが対になって用意されています。WMT データセットは多くの研究でベンチマークとして利用されており、seq2seq の性能を評価するためによく用いられます。ただしそのサイズは大きく（20GB 以上）、気軽に利用できません。

8.3.1 日付フォーマットの変換問題

ここで私たちが取り組む問題は「日付のフォーマット変換」です。これは、英語圏で利用されるさまざまな日付の表現を標準的なフォーマットに変換することを目標とするタスクです。例を挙げれば、図8-22に示すように、人が書いた「september 27, 1994」などの日付データを「1994-09-27」のような標準的なフォーマットへと変換します。

| | | |
|--------------------|---|------------|
| september 27, 1994 | → | 1994-09-27 |
| JUN 17, 2013 | → | 2013-06-17 |
| 2/10/93 | → | 1993-02-10 |

図8-22 日付フォーマット変換の例

ここで「日付のフォーマット変換問題」を採用したのには、2つの理由があります。ひとつは、この問題は見た目ほど簡単ではないということです。なぜなら、入力される日付データにはさまざまなバリエーションが存在するため、その変換ルールはそれなりに複雑になるからです。もしその変換ルールを人の手によってすべて書き出そうとすれば、それはとても面倒で骨の折れる作業になるでしょう。

また、この問題を採用したもうひとつの理由は、問題の入力（質問）と出力（回答）に分かりやすい対応関係があるということです。具体的に言うと、年・月・日の対応関係が存在します。そのため、Attention がそれぞれの要素に正しく注意を払っているかどうかを確認することができます。

ここで私たちが取り組む日付変換のデータは、dataset/date.txt にあらかじめ用意してあります。このテキストファイルには図8-23に示すように、日付変換の学習データが 50,000 個だけ格納されています。

| | | |
|----|-----------------------------|-------------|
| 1 | september 27, 1994 | _1994-09-27 |
| 2 | August 19, 2003 | _2003-08-19 |
| 3 | 2/10/93 | _1993-02-10 |
| 4 | 10/31/90 | _1990-10-31 |
| 5 | TUESDAY, SEPTEMBER 25, 1984 | _1984-09-25 |
| 6 | JUN 17, 2013 | _2013-06-17 |
| 7 | april 3, 1996 | _1996-04-03 |
| 8 | October 24, 1974 | _1974-10-24 |
| 9 | AUGUST 11, 1986 | _1986-08-11 |
| 10 | February 16, 2015 | _2015-02-16 |
| 11 | October 12, 1988 | _1988-10-12 |
| 12 | 6/3/73 | _1973-06-03 |
| 13 | Sep 30, 1981 | _1981-09-30 |
| 14 | June 19, 1977 | _1977-06-19 |
| 15 | OCTOBER 22, 2005 | _2005-10-22 |

Lines: 50,000 Chars: 2,050,000 2.05 MB

図 8-23 日付フォーマット変換のための学習データ：空白文字（スペース）はグレーのドットで表示

本書が用意する日付のデータセットは、入力文の長さを揃えるために、パディングとして空白文字を付けています。また、入力と出力の区切り文字として「_（アンダースコア）」を配置しています。なお、この問題では出力の文字数は一定であるため、出力の終了を知らせるための区切り文字は使用しません。



前章でも説明したように、本書では上のような seq2seq 用の学習データを Python から簡単に扱うモジュールを提供します。そのモジュールは、`dataset/sequence.py` にあります。

8.3.2 Attention 付き seq2seq の学習

それでは、日付変換用のデータセットを対象に、AttentionSeq2seq の学習を行わせましょう。以下に学習のコードを示します ([ch08/train.py](#))。

```
import sys
sys.path.append('..')
import numpy as np
from dataset import sequence
from common.optimizer import Adam
from common.trainer import Trainer
from common.util import eval_seq2seq
from attention_seq2seq import AttentionSeq2seq
from ch07.seq2seq import Seq2seq
from ch07.peeky_seq2seq import PeekySeq2seq
```

```
# データの読み込み
(x_train, t_train), (x_test, t_test) = sequence.load_data('date.txt')
char_to_id, id_to_char = sequence.get_vocab()

# 入力文を反転
x_train, x_test = x_train[:, ::-1], x_test[:, ::-1]

# ハイパーパラメータの設定
vocab_size = len(char_to_id)
wordvec_size = 16
hidden_size = 256
batch_size = 128
max_epoch = 10
max_grad = 5.0

model = AttentionSeq2seq(vocab_size, wordvec_size, hidden_size)
optimizer = Adam()
trainer = Trainer(model, optimizer)

acc_list = []
for epoch in range(max_epoch):
    trainer.fit(x_train, t_train, max_epoch=1,
                batch_size=batch_size, max_grad=max_grad)

    correct_num = 0
    for i in range(len(x_test)):
        question, correct = x_test[[i]], t_test[[i]]
        verbose = i < 10
        correct_num += eval_seq2seq(model, question, correct,
                                    id_to_char, verbose, is_reverse=True)

    acc = float(correct_num) / len(x_test)
    acc_list.append(acc)
    print('val acc %.3f%%' % (acc * 100))

model.save_params()
```

ここで示したコードは、前章の「足し算」の学習用コードとほとんど同じです。異なる点は、学習データとして日付データを読み込むこと、そしてモデルにAttentionSeq2seq を使うことです。またここでは、入力文を反転するテクニック(Reverse)も使用しています。後は学習を行いながら、エポックごとにテストデータを使用して正解率を計測します。このとき、最初の 10 題の問題については、どのような結果を出力したのか質問文と正解を含めてターミナルに出力しています。

それでは、上のコードを実行してみましょう。そうすると、学習が進むにつれて図8-24のような結果が表示されます。

The figure displays three terminal windows side-by-side, each showing the output of a different seq2seq model training script. A large arrow points from left to right, indicating the progression of training.

- Left Window:** Shows the output of `ch08 -- python train_seq2seq.py -- 64x2`. It contains two Q-T pairs: "Q Mar 25, 2003 T 2003-03-25" and "Q 1999-05-11 T 1970-07-18".
- Middle Window:** Shows the output of `ch08 -- python train_seq2seq.py -- 64x21`. It shows improved results: "Q Mar 25, 2003 T 2003-03-25", "Q Tuesday, November 22, 2016 T 2016-11-22", and "Q 1999-05-11 T 1970-07-18".
- Right Window:** Shows the output of `ch08 -- python train_seq2seq.py -- 64x21`. It shows the final state of the model: "Q Saturday, July 18, 1970 T 1970-07-18", "Q Tuesday, November 22, 2016 T 2016-11-22", and "Q 1999-05-11 T 1970-07-18".

図8-24 ターミナルに表示される結果の推移

図8-24を見て分かるように、Attention付きのseq2seqは、学習を重ねるにつれて賢くなります。実際、すぐにほとんどの問題に対して正解を出せるようになります。このとき、テストデータの正解率（上のコードの`acc_list`）をプロットすると、図8-25のようになります。

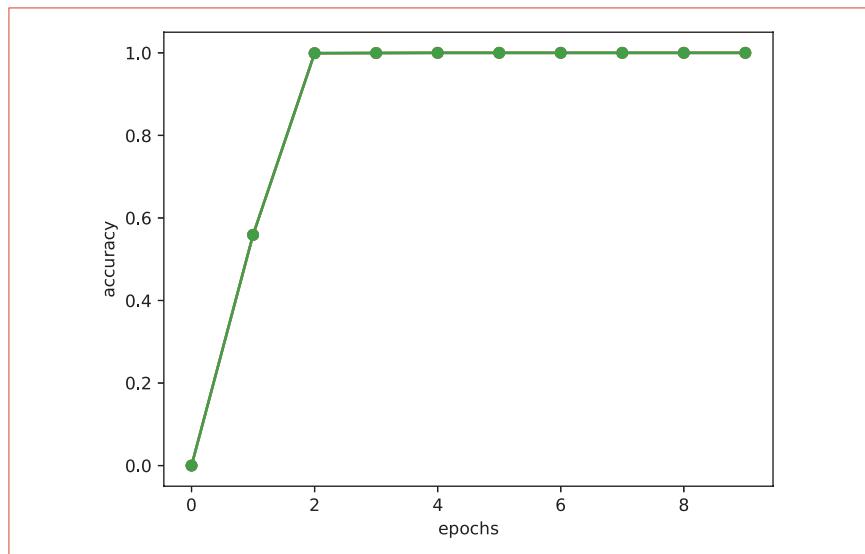


図 8-25 正解率の推移

図 8-25 のとおり、1 エポック目から急速に正解率を高め、2 エポック目ですでに、ほとんどすべての問題に正解します。これはとても良い結果と言えそうです。それではこの結果を、前章のモデルと比較してみましょう。結果は図 8-26 のようになります。

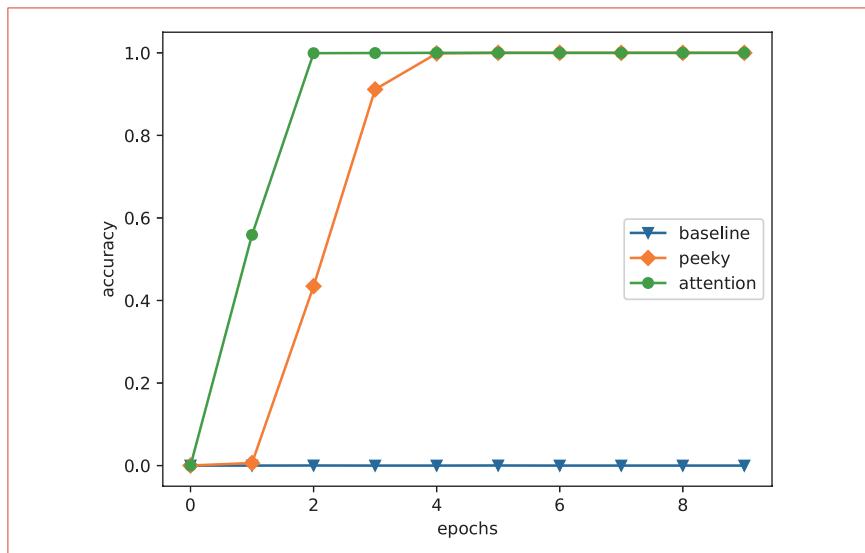


図8-26 他モデルとの比較：グラフの「baseline」は前章のシンプルな seq2seq、「peeky」は“覗き見”で改良した seq2seq（すべてのモデルにおいて入力文は反転して使用）

図8-26の結果を見ると、単純な seq2seq（グラフの baseline）はまったく使い物にならないことが分かります。10エポックを経過しても、ほとんどの問題で不正解です。一方、“覗き見”を使う Peeky は、良い結果を示しています。このモデルは3エポック目から正解率を上げていき、4エポック目で正解率が 100% に到達しています。しかしこれも学習速度の点で、Attention のほうがやや優勢です。

今回の実験では最終的な精度という点では、Attention と Peeky はともに同じ程度の結果になりました。しかし現実には、時系列データが長く複雑になるに従って、学習の速さだけでなく精度の点においても Attention が有利になるでしょう。

8.3.3 Attention の可視化

続いて、Attention の可視化を行いたいと思います。これは Attention が時系列変換を行うときに、どの要素に注意を払っているのかを実際に見てみようという試みです。というのも、Attention レイヤでは、各時刻の Attention の重みをメンバ変数として保持しているので、それを可視化することは簡単に行えるのです。

私たちの実装では、Time Attention レイヤにあるメンバ変数の `attention_weights` に各時刻の Attention の重みが保持されています。これを使えば、入力文

と出力文の各単語の対応関係を2次元のマップとして描画できます。ここでは、学習済みのAttentionSeq2seqに対して、日付変換を行ったときのAttentionの重みを可視化したいと思います。ここではコードの掲載は省略して、結果だけを図8-27に示します(☞ ch08/visualize_attention.py)。

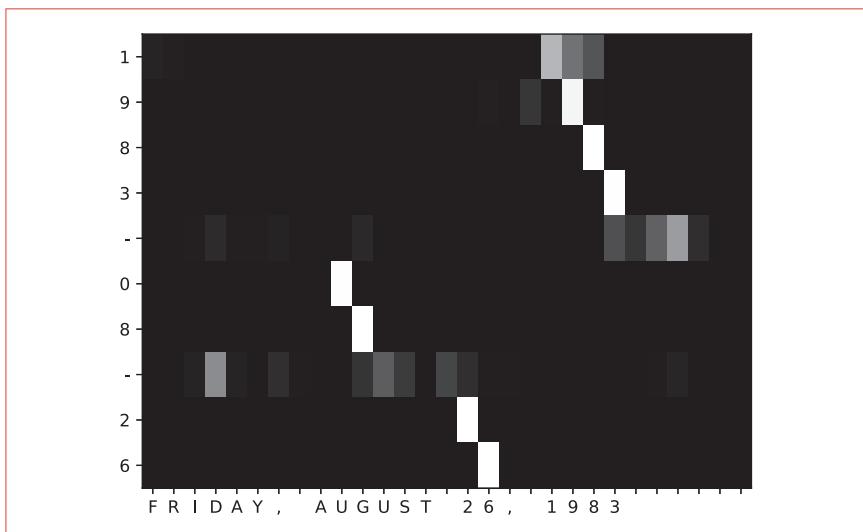


図8-27 学習済みモデルを使用して、時系列変換を行ったときのAttentionの重みを可視化する。横軸はモデルへの入力文、縦軸はモデルの出力文。マップの各要素は、白に近いほど値が大きい(1.0に近い)ことを表す。

図8-27は、seq2seqが時系列変換を行うときのAttentionの重みを可視化した結果です。たとえば、seq2seqが最初の「1」を出力するときには、入力文の「1」に注意が向けられていることが分かります。そして、ここで注目すべき点は、年・月・日の対応関係です。上の結果を詳しく見ると、縦軸(出力)の「1983」や「26」は、横軸(入力)の「1983」や「26」と見事に対応しています。さらに、月を表す「08」に入力文の「AUGUST」が対応している点は驚くべきことでしょう。seq2seqは、「August」が「8月」に対応するということをデータだけから学習したのです！

それでは、他の例をいくつか図8-28に示します。ここでも、年・月・日の対応関係がはっきりと見て取れます。

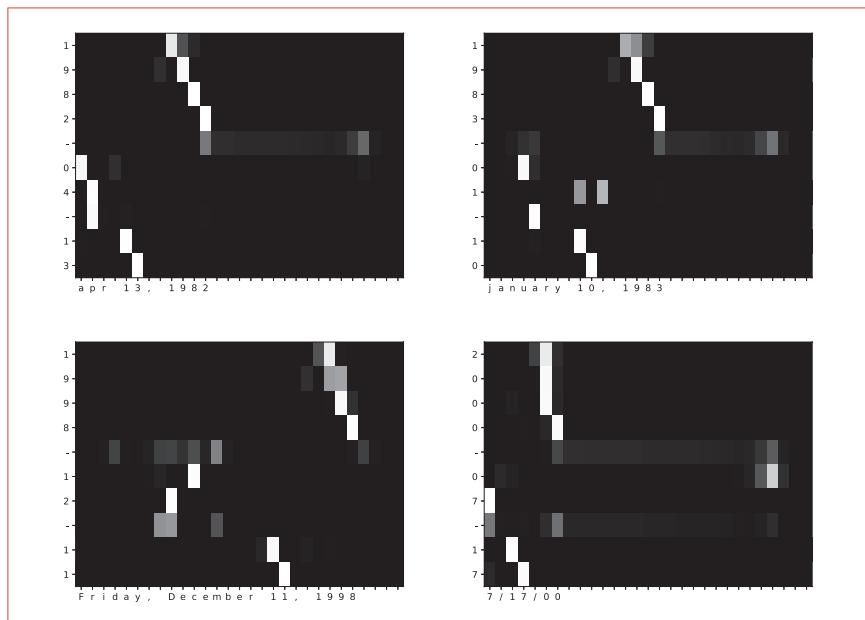


図 8-28 Attention の重みの例

このように Attention を用いることで、seq2seq は私たち人間と同じように、必要な情報に注意を向けることができました。これは見方を変えると、「Attention によって、私たち人間がモデルの行う処理を理解できるようになった」とも言えます。



ニューラルネットワークが内部でどのような処理を行っているのか——どのようなロジックで処理しているのか——ということを人間は理解できません。それに対して、Attention は「人間が理解可能な構造や意味」をモデルに与えます。上の例では、Attention を用いることで単語と単語の関連性を見ることができました。それによって、モデルの処理ロジックが人間のそれに従っているかどうかを判断することができます。

以上で Attention の評価は終わりです。ここでの実験を通して、Attention の素晴らしい効果を実感してもらえたことでしょう。これで Attention の中心的な話は終わりになりますが、まだまだ Attention については話すべきことが多く残っています。次節以降も Attention を中心に、より高度なテクニックを紹介したいと思います。

ます。

8.4 Attentionに関する残りのテーマ

これまで私たちは、Attentionについて——正確には、Attentionを備えたseq2seqについて——見てきました。ここではAttentionに関して、これまで扱えなかったトピックをいくつか紹介したいと思います。

8.4.1 双方向RNN

ここでは、seq2seqのEncoderに焦点を当てます。早速復習になりますが、前節までのEncoderは、図8-29のように表されます。

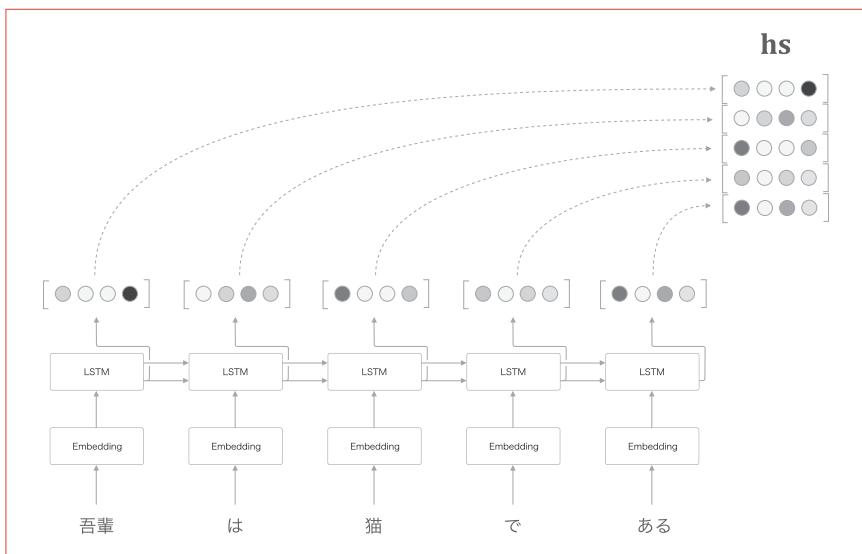


図8-29 LSTMレイヤにより hs を出力

図8-29に示すとおり、LSTMの各時刻の隠れ状態ベクトルはhsとしてまとめられます。ここでEncoderが出力するhsの各行には、それに対応する単語の成分が多く含まれます。

さて、ここで注目してほしいのは、私たちは文を左から右に読んでいるということです。そのため、たとえば図8-29の「猫」という単語に対応するベクトルには、「吾

輩」「は」「猫」の3つの単語の情報がエンコードされることになります。ここで全体のバランスを考えれば、「猫」という単語の“周囲”的な情報をバランス良く含ませたいと思うでしょう。



今回の翻訳の問題では、私たちは時系列データ（翻訳すべき文章）をすべて与えられています。そのため、文章は左から右方向だけではなく、右から左方向へも読む（処理する）ことができます。

そこで、LSTMを双方向から処理させることが考えられます。これが**双方向 LSTM**（**双方向 RNN**）と呼ばれるテクニックです。双方向 LSTMは図で表すと、図8-30のようになります。

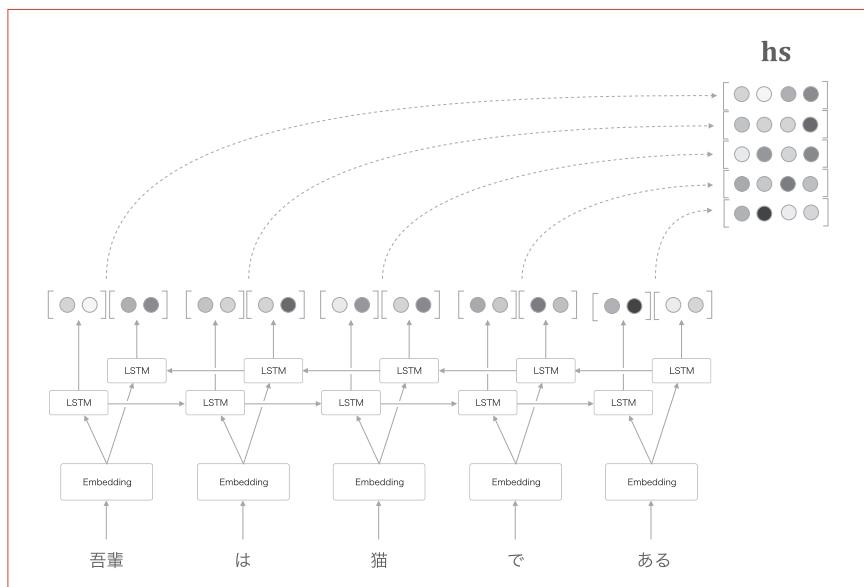


図8-30 双方向 LSTMによりエンコードを行う例（ここでは LSTM レイヤを簡略化して図示する）

図8-30に示すとおり、双方向 LSTMではこれまでのLSTMレイヤに加え、逆方向に処理するLSTMレイヤも追加します。そして、各時刻において2つのLSTMレイヤの隠れ状態を連結し、それを最終的な隠れ状態ベクトルとするのです（「連結」以外にも「和」や「平均」などが考えられます）。

このように双方向から処理することで、各単語に対応する隠れ状態ベクトルは、左と右の両方向からの情報を集約することができます。それによって、バランスのとれた情報がエンコードされるのです。

双方向 LSTM の実装は簡単です。その実装方法（のひとつ）は、2つの LSTM レイヤ（私たちの場合は Time LSTM レイヤ）を使って、それぞれのレイヤに与える単語の並びを調整します。具体的に言うと、ひとつの LSTM レイヤにはこれまでどおりの入力文を与えます。これは、入力文を「左から右方向」に処理する一般的な LSTM レイヤです。一方、もうひとつの LSTM レイヤには入力文の単語を右から左方向の並びとして与えます。たとえば、元の文章が「A B C D」だったとしたら「D C B A」の並びに変えます。この並びを変えた入力文を与えることで、もうひとつの LSTM レイヤは入力文を「右から左方向」に処理することになります。後は、その2つの LSTM レイヤの出力を連結するだけで、双方向 LSTM レイヤができます。



本章では、分かりやすさを優先して、Encoder には単方向の LSTM を利用しました。ですが、ここで説明した双方向 LSTM を Encoder に利用することも、もちろん可能です。興味のある方は、双方向 LSTM を使った Attention 付き seq2seq の実装に挑戦してみてください。なお、双方向 LSTM の実装は common/time_layers.py の TimeBiLSTM クラスにあります。興味のある方は参考にしてください。

8.4.2 Attention レイヤの使用方法

続いて、Attention レイヤの使い方について考えたいと思います。復習になりますが、これまで私たちは、図8-31 のレイヤ構成で Attention レイヤを用いてきました。

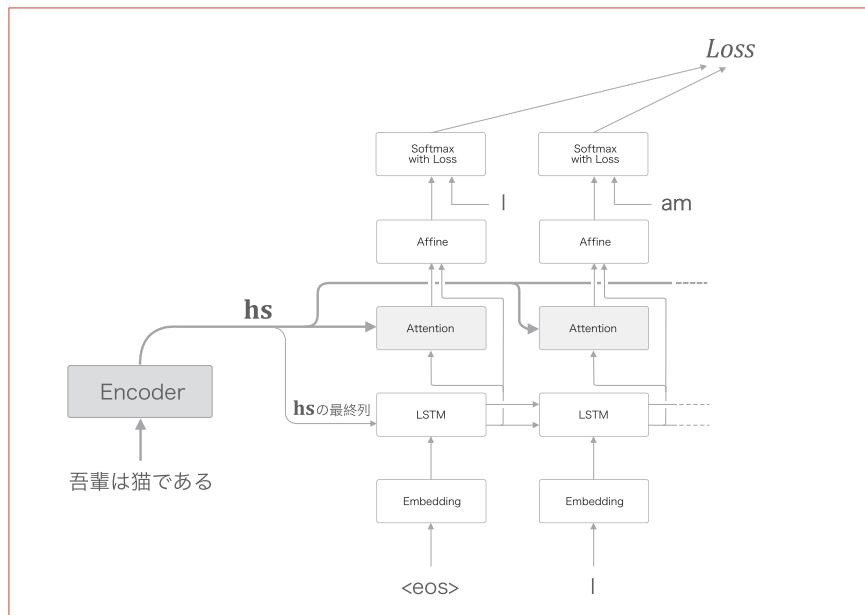


図8-31 前節までに使用してきた Attention 付き seq2seq のレイヤ構成

図8-31で示すように、私たちはAttentionレイヤをLSTMレイヤとAffineレイヤの間に挿入しました。しかし、Attentionレイヤを利用する場所は、必ずしも図8-31のとおりとは限りません。実際、Attentionを使用するモデルでは、上記以外にもいくつか候補が考えられます。たとえば、文献[48]では、図8-32のような構成でAttentionが使用されています。

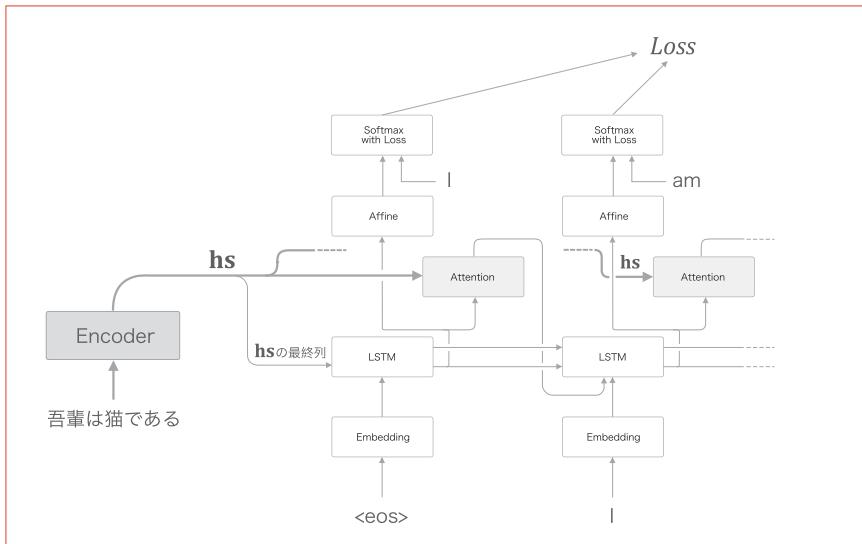


図8-32 Attention レイヤの別の使用例（ここでは文献[48]を参考に、簡略化したネットワーク構成を図示する）

図8-32では、Attention レイヤの出力——コンテキストベクトル——が、次時刻のLSTM レイヤへの入力へと接続されています。そのような構成にすることで、LSTM レイヤがコンテキストベクトルの情報を利用することができます。一方、私たちが実装したモデルはAffine レイヤがコンテキストベクトルを利用しました。

それでは、Attention レイヤの場所の違いによって、最終的な精度にどのような影響があるでしょうか？その答えは、やってみないと分かりません。現実は、実際のデータを使って検証するしかないのでです。ただし、上の2つのモデルにおいてはどちらの場合も、コンテキストベクトルはうまく活用される構成になっています。そのため、その2つのモデルに大きな精度の違いは見られないかもしれません。

なお、実装の観点からは、前者の構成——LSTM レイヤとAffine レイヤの間にAttention レイヤを挿入する構成——のほうが簡単に行えます。というのも、前者の構成ではDecoderにおけるデータの流れが下から上の一方向であるため、Attention レイヤのモジュール化が容易に行えるのです。実際、私たちはTime Attention レイヤとして簡単にモジュール化できました。

8.4.3 seq2seq の深層化と skip コネクション

翻訳などの現実的なアプリケーションにおいては、解くべき問題はより複雑になります。その場合、Attention 付き seq2seq には、より高い表現力が望まれるでしょう。その際まず考えるべきことは、RNN レイヤ (LSTM レイヤ) を深く重ねることです。層を深くすることで、私たちは表現力の高いモデルを作ることができます。これは Attention 付き seq2seq でも同じ話です。それでは、Attention 付き seq2seq を深くすると、どうなるでしょうか。ここではそのひとつの例を図 8-33 に示します。

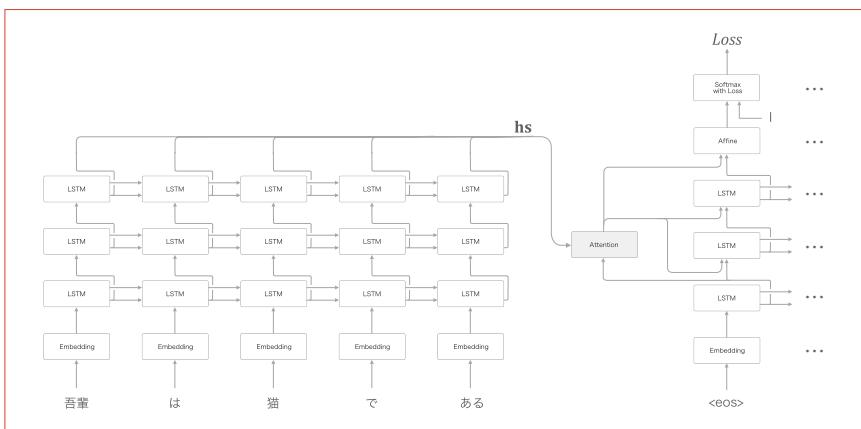


図 8-33 3 層の LSTM レイヤを使用した Attention 付きの seq2seq

図 8-33 のモデルでは、Encoder と Decoder で 3 層の LSTM レイヤを使用しています。この例のように、Encoder と Decoder では、同じ層数の LSTM レイヤを用いるのが一般的です。また、Attention レイヤの使い方はいろいろなバリエーションが考えられます。ここでは Decoder の LSTM レイヤの隠れ状態を Attention レイヤへ入力します。そして Attention レイヤの出力であるコンテキストベクトルを、Decoder の複数のレイヤ (LSTM レイヤや Affine レイヤ) へと伝播させます。



ここで示した図 8-33 のモデルは、ひとつの例にすぎません。この例の他にも、複数の Attention レイヤを使用したり、Attention の出力を次時刻の LSTM レイヤへ入力したりと、さまざまなバリエーションが考えられます。また前章で説明したように、レイヤを深くするときに汎化性能を落とさないことが重要になります。その場合、Dropout や重み共有などの技術が有効です。

また、層を深くするときに使われる重要なテクニックのひとつに **skip コネクション**があります（「residual コネクション」や「ショートカット」とも呼ばれます）。これは図8-34に示すように、層をまたいで“線をつなぐ”だけのシンプルなテクニックです。

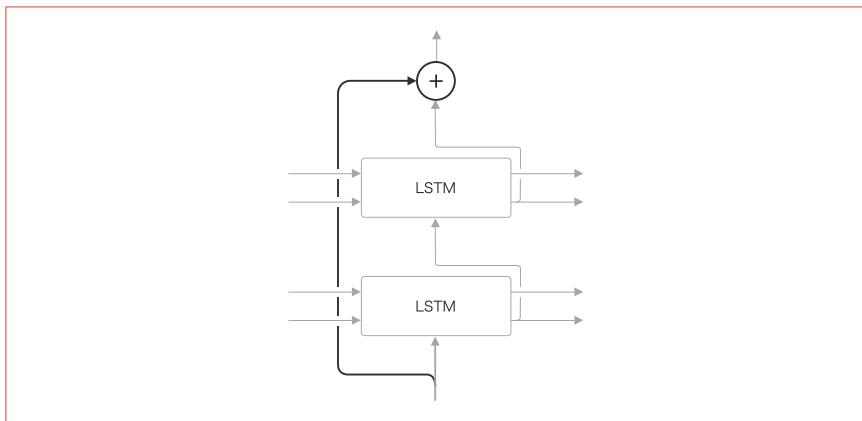


図8-34 LSTM レイヤにおける skip コネクションの例

図8-34のとおり、skip コネクションは「層をまたぐ接続」です。このとき、skip コネクションの接続部においては2つの出力が「加算」されます。この加算（正確には要素ごとの加算）というのが重要なポイントです。なぜなら、加算は、逆伝播時には勾配を「そのまま流す」だけなので、skip コネクションにおける勾配は何の影響も受けずに前層へと流れることになるからです。それによって、層を深くしても勾配が消失（もしくは爆発）することなく伝播し、良い学習が期待できます。



RNN レイヤの逆伝播では、時間方向において勾配の消失もしくは爆発が起きます。勾配消失に対しては、LSTM や GRU などの「ゲート付き RNN」によって対応し、勾配爆発に対しては、「勾配クリッピング」によって対応できます。一方、RNN の深さ方向に対する勾配消失は、ここで述べた skip コネクションが有効です。

8.5 Attention の応用

これまで私たちは Attention を seq2seq だけに適用してきました。しかし Attention というアイデア自体は汎用的であり、さらに多くの可能性を秘めています。実際のところ、近年のディープラーニングの研究においては、Attention が重要なテクニックとしてさまざまな場面で登場します。本節では、Attention の重要性と可能性を感じてもらうことを目的として、Attention を使った最先端の研究例を 3 つ紹介します。

8.5.1 Google Neural Machine Translation (GNMT)

機械翻訳の歴史を眺めると、主要な手法は時代とともに推移してきたことがうかがえます。その流れは、「ルールベース翻訳」から「用例ベース翻訳」へ、そして「統計ベース翻訳」へと移り変わってきました。そして現在では、これら従来の技術に代わり、**ニューラル翻訳** (Neural Machine Translation) が多くの注目を集めています。



ニューラル翻訳という用語は、これまでの統計翻訳と対比する形で用いられてきました。しかし最近では、seq2seq を使った機械翻訳の総称として用いられています。

Google 翻訳では、実際のサービスとして 2016 年からニューラル翻訳が使われています。この機械翻訳システムは、**GNMT** (Google Neural Machine Translation) と呼ばれており、文献 [50] にその技術の詳細が記されています。ここでは、GNMT のアーキテクチャについて、そのレイヤ構成を中心に見ていきたいと思います。それでは、GNMT のレイヤ構成を図 8-35 に示します。

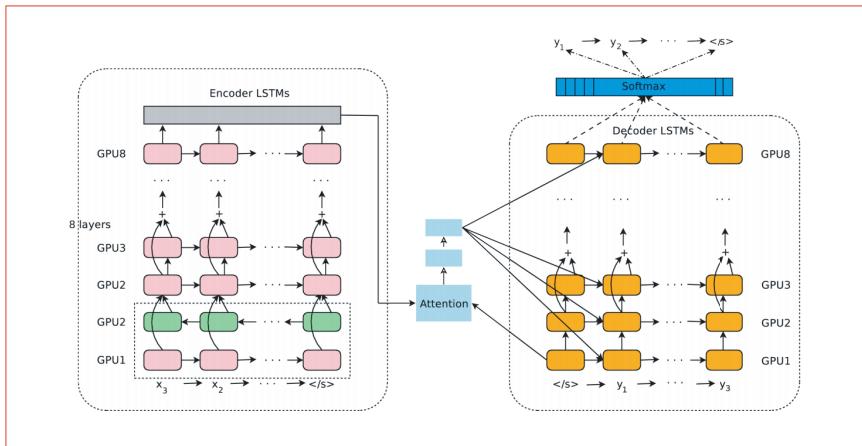


図 8-35 GNMT のレイヤ構成（図は文献[50]より抜粋）

GNMT は本章で実装した Attention 付き seq2seq と同じく、Encoder と Decoder そして Attention から構成されています。ただし、私たちのシンプルなモデルとは異なり、翻訳精度向上のために多くの改良が見受けられます。たとえば、LSTM レイヤの多層化や双方向 LSTM (Encoder の 1 層のみ)、そして skip コネクションなどが見られます。また、学習を高速に行うために、複数 GPU での分散学習を行っています。

GNMT では、以上のアーキテクチャ上の工夫の他にも、低頻出単語の対応や、推論時の高速化のための量子化などさまざまな工夫が行われています。そのようなテクニックを利用して、GNMT は非常に優れた結果を達成しました。実際に報告された結果は、図 8-36 のとおりです。

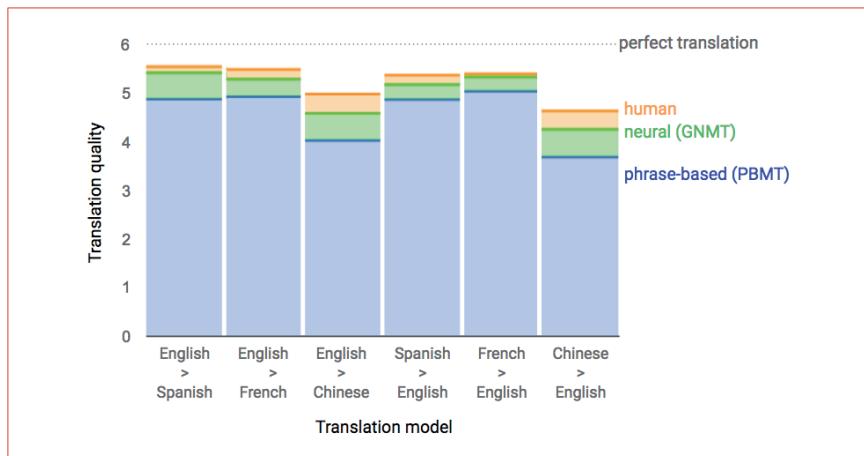


図8-36 GNMTの精度評価：縦軸は翻訳の良さを、人が0点から6点の間で評価したもの（図は文献[51]より抜粋）

図8-36のとおり、GNMTは従来の手法——統計ベース機械翻訳のひとつである「フレーズベースの機械翻訳」——と比較して、翻訳精度を向上させることに成功しました。そして、その精度は「人（による翻訳）」の精度に近づきつつあるということです。このように、GNMTは素晴らしい結果を示し、ニューラル翻訳の実用性と可能性を大いに見せつけました。ただし、Google翻訳を使ってみれば分かりますが、まだまだ不自然な翻訳や、人であれば絶対に犯さないようなミスも散見されます。これから先も、機械翻訳の研究は進められていくでしょう。実際、GNMTは始まりにすぎず、現在でもニューラル翻訳を中心に活発に研究が行われているのです。



GNMTを実現するには、大量のデータと大量の計算リソースが必要になります。文献[50]によると、大量の学習データを使用し、（ひとつのモデルに）100枚近くのGPUを使って、6日間の学習を行っているそうです。さらに8つのモデルを並行して学習するアンサンブル学習や、強化学習によってさらなる精度向上を成し遂げるといった工夫もされています。そのようなことは、一個人が実現できるものではありません。しかし、そこで使われる技術のコアとなる部分の知識を、私たちはすでに習得しているのです！

8.5.2 Transformer

私たちはこれまで、RNN（LSTM）をいたるところで使用してきました。言語モ

ルに始まり、文章生成、seq2seq、そして Attention 付き seq2seq と、その構成要素には必ず RNN が登場しました。そして、この RNN によって、可変長の時系列データはうまく処理され、(多くの場合) 良い結果を得ることができます。しかし、RNN にも欠点があります。その欠点のひとつに、並列化処理が挙げられます。

RNN は、前時刻に計算した結果を用いて逐次的に計算を行います。そのため、RNN の計算を、時間方向で並列的に計算することは（基本的には）できません。この点は、ディープラーニングの学習が GPU を使った並列計算の環境で行われることを想定すると、大きなボトルネックになります。そこで、RNN を避けたいというモチベーションが生まれます。

そのような背景から、現在では RNN を取り除く研究（もしくは並列計算可能な RNN の研究）が活発に行われています。その中で有名なものに、Transformer [52] と呼ばれるモデルがあります。これは『Attention is all you need』というタイトルの論文で提案された手法です。そのタイトルが示すとおり、RNN ではなく Attention を使って処理します。ここでは、この Transformer について簡単に見ていきます。



Transformer 以外にも、RNN を取り除こうという研究はいくつか行われています。その成果のひとつに、RNN の代わりに畳み込み層（Convolution レイヤ）を利用する研究 [54] があります。その研究の詳細はここでは触れませんが、基本的には、RNN の代わりに畳み込み層を用いて seq2seq を構成します。それにより、計算の並列化を実現できます。

Transformer は Attention によって構成されますが、その中でも **Self-Attention** というテクニックが利用される点が重要なポイントです。この Self-Attention は直訳すれば「自分自身に対しての Attention」ということになります。つまりこれは、ひとつの時系列データを対象とした Attention であり、ひとつの時系列データ内において各要素が他の要素に対してどのような関連性があるのかを見ていこうというものです。私たちの Time Attention レイヤを使って説明すると、Self-Attention は図 8-37 のように書けます。

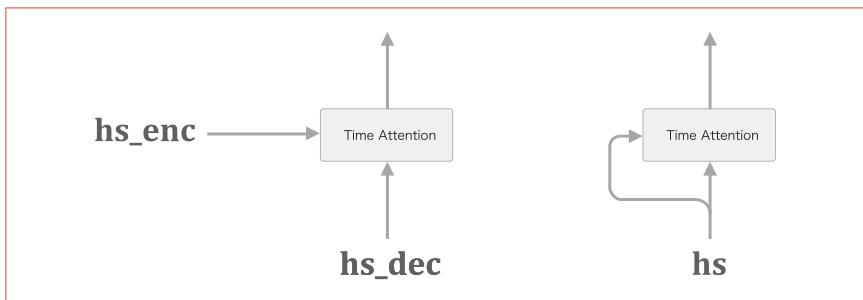


図 8-37 左図が通常の Attention、右図が Self-Attention

これまで私たちは「翻訳」のような 2 つの時系列データ間の対応関係を Attention で求めてきました。このとき Time Attention レイヤへの 2 本の入力には、図 8-37 の左図で示すように、異なる 2 つの時系列データが入力されます。これに対して Self-Attention は、図 8-37 の右図で示すように、2 本の入力線にひとつの時系列データが入力されます。そうすることで、ひとつの時系列データ内において各要素間の対応関係が求められます。

Self-Attention の説明が済んだので、続いて Transformer のレイヤ構成を見ていくましょう。Transformer の構成は、図 8-38 のようになります。

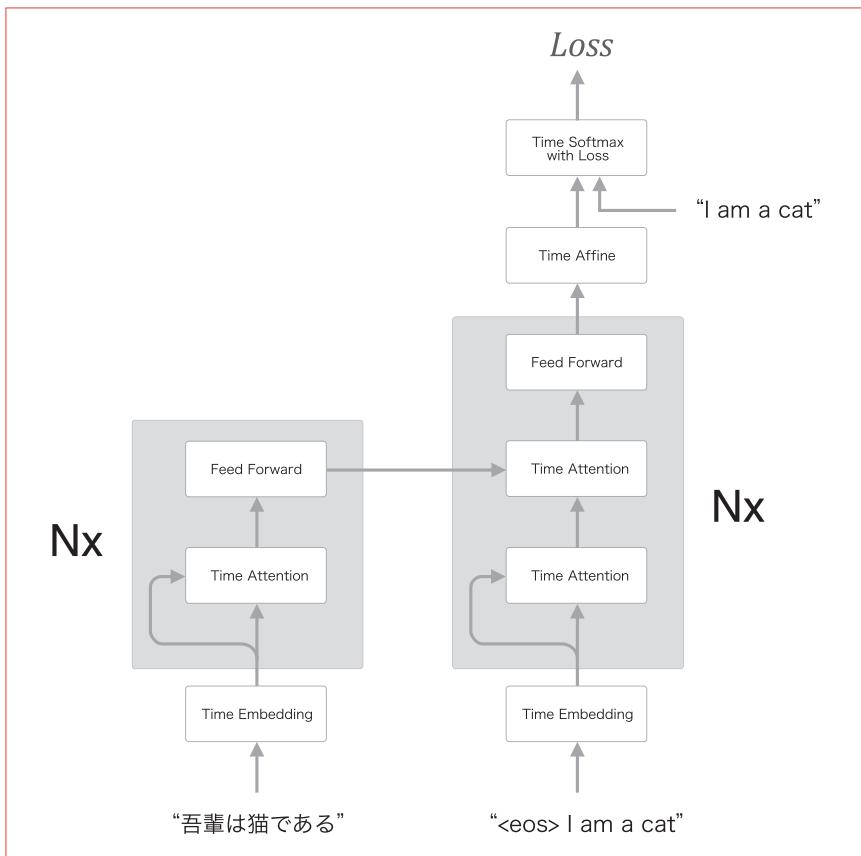


図 8-38 Transformer のレイヤ構成（文献 [52] を参考に、簡略化したモデルを図示する）

Transformer では、RNN の代わりに Attention が使われます。実際に図 8-38 を見ると、Encoder と Decoder の両者で Self-Attention が使われていることが分かります。なお、図 8-38 の Feed Forward レイヤは、フィードフォワードのネットワーク（時間方向に独立して処理するネットワーク）を表します。具体的には、隠れ層が 1 層で活性化関数に ReLU を用いた全結合のニューラルネットワークが用いられています。また、図中に「Nx」とありますが、これは背景がグレーで囲まれた要素を N 回積み重ねることを意味します。



図 8-38 は、Transformer を簡略化して示しています。実際には、ここで示したアーキテクチャに加えて skip コネクションや Layer Normalization [8]などが利用されています。また、複数個の Attention を（並行的に）用いたり、Positional Encoding と呼ばれる時系列データの位置情報をエンコードしたりといった工夫が見られます。

この Transformer を用いることで、計算量を抑え、GPU による並列計算の恩恵をより多く受けることができます。その成果として、Transformer は GNMT に比べて学習時間を大幅に減らすことに成功しました。さらに翻訳精度の点でも、**図 8-39** が示すように、精度向上を実現しました。

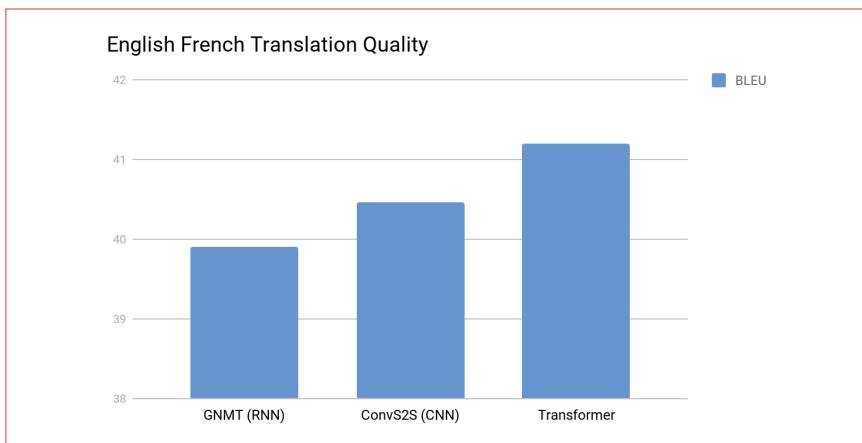


図 8-39 WMT というベンチマーク用の翻訳データを使って、英語とフランス語の翻訳の精度を評価した結果。縦軸は、翻訳精度の指標である BLEU スコアであり、これは数値が高いほど良いことを示す（図は文献[53]より抜粋）

図 8-39 では、3 つの手法が比較されています。その結果は、GNMT よりも「畳み込み層を用いた seq2seq」（図中では ConvS2S で表記）が精度が高く、さらに Transformer はそれをも上回っています。このように、Attention は、計算量だけでなく、精度の観点からも有望な技術であることが分かります。

私たちは、これまで Attention を RNN と組み合わせて利用してきました。しかし、ここでの研究が示唆するように、Attention は RNN を置き換えるモジュールとしても利用できるのです。これによって、さらに Attention の利用機会が増えていく

かもしれません。

8.5.3 Neural Turing Machine (NTM)

私たち人間は複雑な問題を解くとき、「紙」と「ペン」を使うことがあります。これは見方を変えれば、紙とペンという外部の「記憶装置」によって、私たちの能力が拡張されているとも解釈できます。これと同じように、ニューラルネットワークにも「外部メモリ」を利用させることで、さらなる力を付加することができます。ここで取り上げるテーマは、「外部メモリによる拡張」です。



RNN や LSTM は、内部状態を使うことで、時系列データを記憶することができました。しかし、その内部状態は固定長であり、そこに詰め込める情報量には制限があります。そこで、RNN の外側に記憶装置（メモリ）を配置し、必要な情報をそこに適宜記録するというようなアプローチが考えられます。

さて、Attention 付き seq2seq では、Encoder は入力文をエンコードします。そして、そのエンコードされた情報を、Attention を介して Decoder が利用しました。ここで注目したいのは、(やはり) Attention の存在です。この Attention によって、Encoder と Decoder は、コンピュータでいうところの「メモリ操作」のようなことを実現しているのです。つまり、Encoder が必要な情報をメモリに書き込み、Decoder は、そのメモリにある情報から必要な情報を読み込んでいると解釈ができます。

そのように考えると、コンピュータのメモリ操作をニューラルネットワークによって再現できそうなことが分かります。すぐに考えられるのは、RNN の外側に情報を記憶しておくためのメモリ機能を配置し、Attention を使って、そのメモリから必要な情報を読み書きさせる方法です。実際、そのような研究はいくつか行われています。その中でも有名な研究として、NTM (Neural Turing Machine) [55] があります。



NTM は、DeepMind のチームによって行われた研究です。それは後に、DNC (Differentiable Neural Computers) [56] と呼ばれる手法へと改良され、その技術論文は科学雑誌『Nature』へと掲載されました。DNC は、NTM のメモリ操作をよりパワーアップさせたものと考えられますが、その技術の本質は共通しています。

NTM の中身を説明する前に、まずは NTM の大枠を見ていきたいと思います。そ

の用途にうってつけなのが、図 8-40 の魅力的な絵です。これは NTM が行う処理をビジュアルで示したもので、NTM のエッセンスがうまくまとめられています（正確には、NTM を発展させた DNC の解説記事 [57] で用いられた図です）。

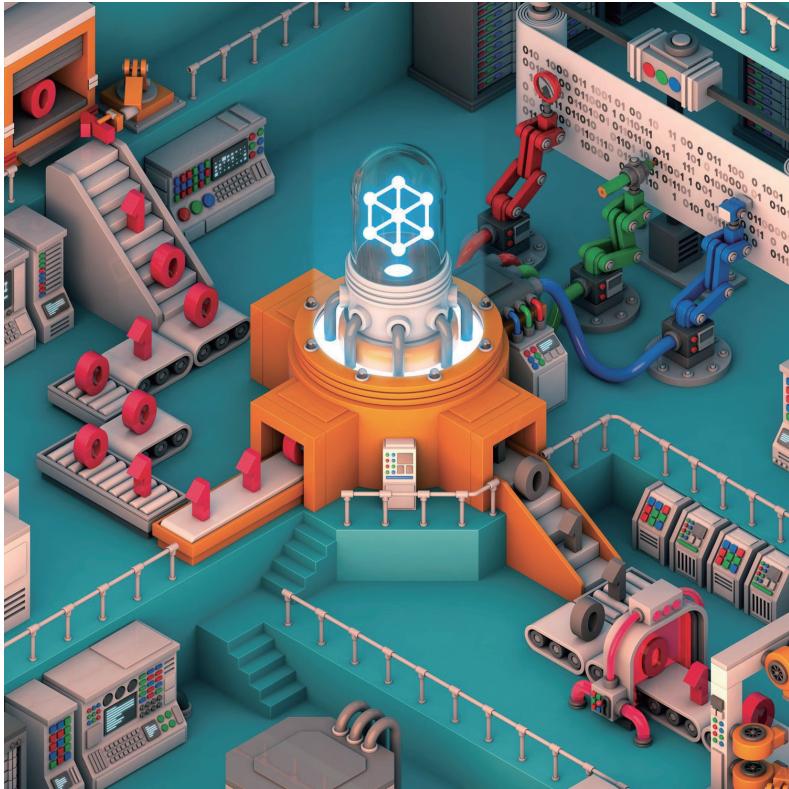


図 8-40 NTM のビジュアル図（図は文献 [57] より抜粋）

それでは、図 8-40 に目を向けましょう。ここで注目したいのは、図の真ん中にある「コントローラ」と呼ばれるモジュールです。これは情報を処理するモジュールで、具体的にはニューラルネットワーク（もしくは RNN）を利用する事が想定されます。図を見ると、このコントローラには次々と「0」や「1」のデータが流れてきており、それを処理して新たなデータを出力していることが分かります。

ここで重要なのが、このコントローラの外側にある「大きな紙（＝メモリ）」の存在です。このメモリによって、コントローラはコンピュータ（もしくはチューーリングマシン）の能力を手にします。その能力とは、具体的には、「大きな紙」に対して、必要な情報を書き、不要な情報を消去し、そして、必要な情報を読み込むという能力です。ちなみに、図8-40の「大きな紙」はロール式になっているので、各ノードは必要な場所のデータを読み書きできます。つまり、目的の場所に移動できるということです。

このように、NTMは外部メモリに対して読み書きを行いながら、時系列データを処理します。そしてNTMのおもしろいところは、そのようなメモリ操作を「微分可能」な計算で構築しているという点です。そのため、メモリ操作の手順をデータから学習することが可能になります。



コンピュータは、人の書いたプログラムによって動作します。これに対してNTMは、データからプログラムを学習します。つまりこれは、「アルゴリズムの入力と出力」から「アルゴリズム自体（ロジック）」を学習できるということを意味します。

NTMは外部メモリに対してコンピュータのように読み書きを行います。このときNTMのレイヤ構成は、簡略化して表すと図8-41のように書けます。

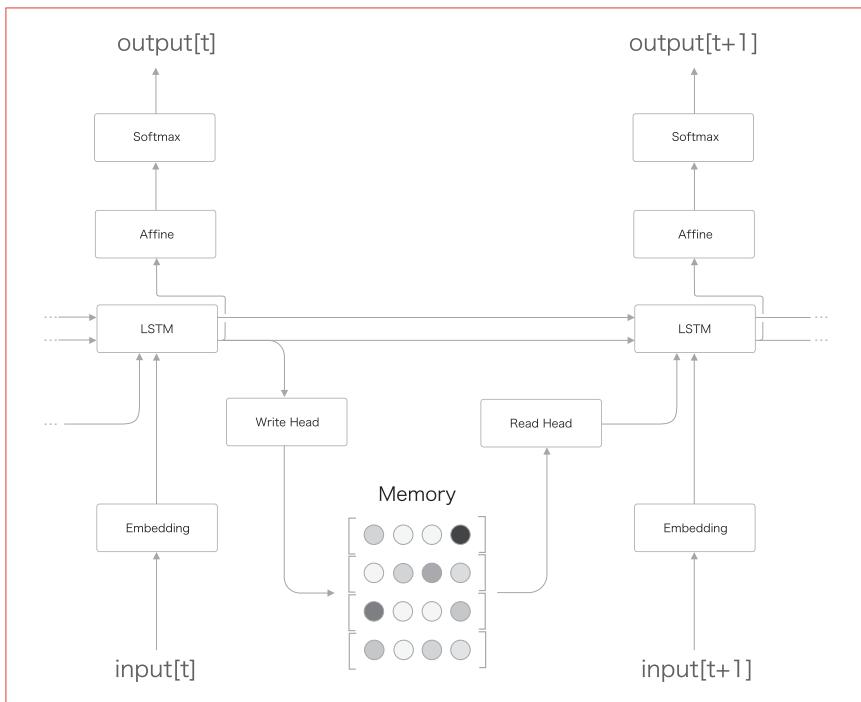


図8-41 NTMのレイヤ構成：新たに Write Head レイヤと Read Head レイヤが登場し、それらがメモリの書き込みと読み込みを行う

図8-41は、簡略化したNTMのレイヤ構成です。ここで、LSTMレイヤが「コントローラ」として、NTMのメインとなる処理を行います。そして、各時刻においてLSTMレイヤの隠れ状態をWrite Headレイヤが受け取り、必要な情報をメモリに書き込みます。さらにRead Headレイヤがメモリから重要な情報を読み込み、それを次時刻のLSTMレイヤへと渡すのです。

それでは、図8-41のWrite HeadレイヤとRead Headレイヤはどのようにメモリ操作を行うのでしょうか。もちろん、ここでもAttentionが使われます。



繰り返しになりますが、メモリに対して、ある番地に存在するデータを読み込む（もしくは書き込む）というとき、私たちはデータを「選ぶ」必要があります。この選ぶという操作自体は微分ができません。そこで、Attention を使い、すべての番地にあるデータを選ぶようにして、それぞれのデータに対する貢献度を表す「重み」を利用するのです。そうすることで、「選ぶ」という操作を微分可能な計算で代替することができます。

NTM が行うメモリ操作は、コンピュータのメモリ操作を模倣するために、2つの Attention を利用します。それは「コンテンツベースの Attention」と「位置ベースの Attention」です。コンテンツベースの Attention は、これまで私たちが見てきた Attention と同じで、あるベクトル（クエリベクトル）に対して、それに似たベクトルをメモリから見つける用途で利用します。

一方、位置ベースの Attention は、前時刻に注目したメモリの位置（=メモリの各位置に対する重み）に対して、その前後に移動（シフト）するような用途で使われます。この技術の詳細は省略しますが、これは1次元の畳み込み演算によって実現されます。このメモリ位置をシフトする機能により、メモリ位置をひとつずつ進めながら読んでいくというコンピュータ特有の動きを再現しやすくなります。



NTM のメモリ操作はいくらか複雑です。上記で説明した操作以外にも、Attention の重みをシャープにする処理や、前時刻の Attention の重みを足し合わせる処理なども含まれます。

このように、外部メモリを自由に利用することで、NTM は大きな力を手にします。実際、seq2seq が解けなかったような複雑な問題に対しても、NTM は驚くべき成果を上げています。具体的には、長い時系列を記憶する問題やソート（=数を大きい順に並べる）などの問題を NTM は見事に解くことに成功したのです。

このように、NTM は外部メモリを使うことで、アルゴリズムを学習する能力を手にします。そして、そこでは Attention が重要なテクニックとして利用されます。外部メモリによる拡張、そして Attention——これらは、今後ますます重要なテクニックとして、さまざまな場所で利用されていくことでしょう。

8.6 まとめ

本章では、Attention の仕組みを学び、Attention レイヤを実装しました。そして、Attention を用いた seq2seq を実装し、簡単な実験を通じて、Attention の素晴らしい効果を確認しました。さらに、モデルの推論時における Attention の重み（確率）を可視化しました。その結果から、Attention を備えたモデルは、私たち人間と同じように必要な情報に注意を向けていることが分かりました。

また本章では、Attention を中心として最先端の研究の動向も紹介しました。その例を見ると、Attention によってディープラーニングの持つ可能性がさらに広がったことが分かります。このように、Attention は応用の効くテクニックであり、多くの可能性を秘めています。ディープラーニングの分野では、これから先も Attention 自体が多くの“注意”を集めるでしょう！

本章で学んだこと

- 翻訳や音声認識など、ある時系列データを別の時系列データに変換するタスクでは、時系列データ間に対応関係が存在することが多くある
- Attention は 2 つの時系列データ間の対応関係をデータから学習する
- Attention では、(ひとつの方法として) ベクトルの内積を使ってベクトル間の類似度を算出し、その類似度を用いた重み付き和ベクトルが Attention の出力となる
- Attention で使用する演算は微分可能であるため、誤差逆伝播法によって学習ができる
- Attention が算出する重み（確率）を可視化することで、入出力の対応関係を見ることができる
- 外部メモリによるニューラルネットワークの拡張の研究例では、メモリの読み書きに Attention が用いられる

付録A sigmoid関数とtanh関数の 微分

ニューラルネットワークでは活性化関数にさまざまな関数が使われます。ここでは、その代表格である sigmoid 関数と tanh 関数を取り上げ、その 2 つの関数の微分を求める過程を異なる 2 つのアプローチで説明します。具体的には、sigmoid 関数の微分は計算グラフを使い、tanh の微分は式を展開して求めます。それぞれのアプローチを理解することで、微分の計算により親しくなるでしょう。

A.1 sigmoid 関数

sigmoid 関数は次の式で表されます。

$$y = \frac{1}{1 + \exp(-x)} \quad (\text{A.1})$$

このとき、式 (A.1) の計算グラフは図 A-1 のようになります。

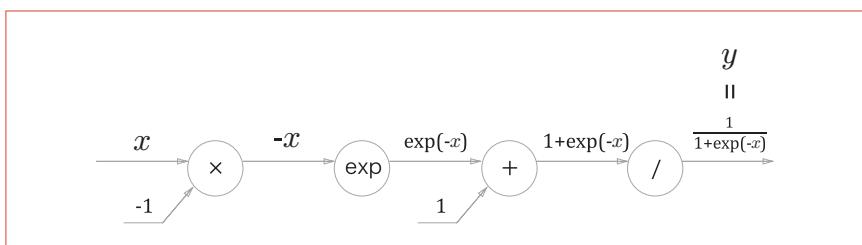


図 A-1 sigmoid レイヤの計算グラフ

図 A-1 では「×」と「+」ノードの他に、「exp」と「/」ノードがあります。exp

ノードは $y = \exp(x)$ の計算を行い、「/」ノードは、 $y = \frac{1}{x}$ の計算を行います。それでは計算グラフを使って、ひとつずつ確認しながら逆伝播を行ってみます。

Step.1

「/」ノードは $y = \frac{1}{x}$ を表しますが、この微分は解析的に次の式によって表されます。

$$\frac{\partial y}{\partial x} = -\frac{1}{x^2} = -\left(\frac{1}{x}\right)^2 = -y^2 \quad (\text{A.2})$$

式 (A.2) より、逆伝播のときは、上流の勾配に対して、 $-y^2$ (順伝播の出力の 2 乗にマイナスを付けた値) を乗算して下流へ流します。計算グラフで書くと図 A-2 のようになります。

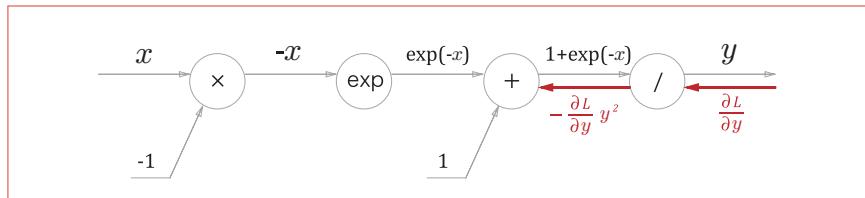


図 A-2 逆伝播ステップ 1

Step.2

「+」ノードは、上流の値を下流にそのまま流すだけです。計算グラフでは図 A-3 のようになります。

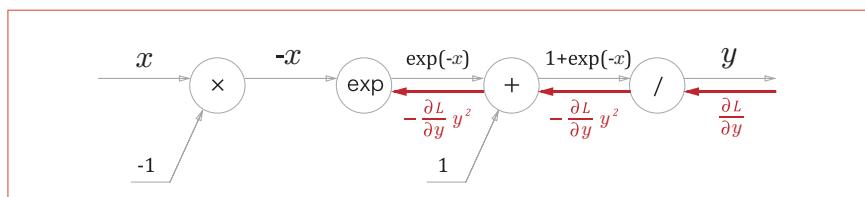


図 A-3 逆伝播ステップ 2

Step.3

\exp ノードは $y = \exp(x)$ を表し、その微分は次の式で表されます。

$$\frac{\partial y}{\partial x} = \exp(x) \quad (\text{A.3})$$

計算グラフでは、上流の勾配に対して、順伝播時の出力——この例では $\exp(-x)$ ——を乗算して下流へ流します（図 A-4）。

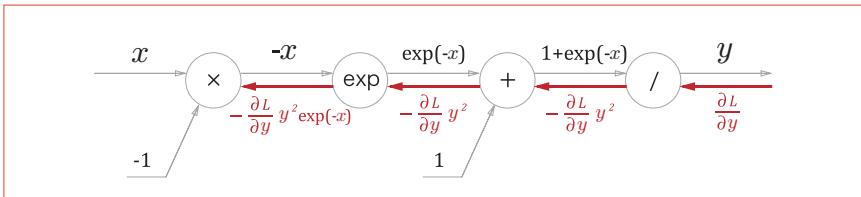


図 A-4 逆伝播ステップ 3

Step.4

「×」ノードは、順伝播時の入力を入れ替えた値を乗算します。そのため、ここで -1 を乗算します。

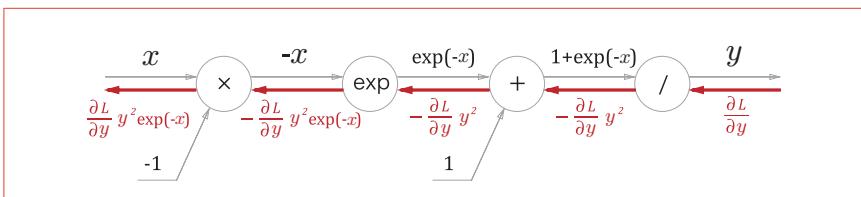


図 A-5 sigmoid 関数の計算グラフ（逆伝播）

以上より、図 A-5 の計算グラフとして sigmoid レイヤの逆伝播を行うことができました。図 A-5 の結果から逆伝播の出力は $\frac{\partial L}{\partial y} y^2 \exp(-x)$ となり、この値が下流にあるノードに伝播しています。さらに、この $\frac{\partial L}{\partial y} y^2 \exp(-x)$ は、次のように整理して書くことができます。

$$\begin{aligned} \frac{\partial L}{\partial y} y^2 \exp(-x) &= \frac{\partial L}{\partial y} \frac{1}{(1 + \exp(-x))^2} \exp(-x) \\ &= \frac{\partial L}{\partial y} \frac{1}{1 + \exp(-x)} \frac{\exp(-x)}{1 + \exp(-x)} \\ &= \frac{\partial L}{\partial y} y(1 - y) \end{aligned} \quad (\text{A.4})$$

この式展開より、sigmoid 関数の逆伝播は、順伝播時の出力だけから計算できることが分かります。以上をまとめると、sigmoid 関数の計算グラフは図 A-6 のように書くことができます。

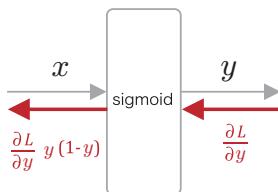


図 A-6 sigmoid 関数の計算グラフ

以上が sigmoid 関数の微分です。ここでは、計算グラフを使って sigmoid 関数の微分を求めました。続いて、tanh 関数の微分を解析的に求める過程を示します。

A.2 tanh 関数

tanh 関数は双曲線正接関数 やハイパボリック・タンジェント (hyperbolic tangent) と呼ばれます。この tanh 関数は式 (A.5) で表されます。

$$y = \tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \quad (\text{A.5})$$

ここで私たちの目指すことは、式 (A.5) に対して $\frac{\partial y}{\partial x}$ を求めることです。そのために、ここでは次の微分の公式を利用します。

$$\left\{ \frac{f(x)}{g(x)} \right\}' = \frac{f'(x)g(x) - f(x)g'(x)}{g(x)^2} \quad (\text{A.6})$$

式 (A.6) は分数関数の微分の公式です。ここでは見やすさを考慮して、 $\frac{f(x)}{g(x)}$ の x に関する微分を $\left\{ \frac{f(x)}{g(x)} \right\}'$ で表します。同様に $f(x)$ の x に関する微分を $f'(x)$ で表記します。

また、ネイピア数 (e) について、次の微分が解析的に導けます。

$$\frac{\partial e^x}{\partial x} = e^x \quad (\text{A.7})$$

$$\frac{\partial e^{-x}}{\partial x} = -e^{-x} \quad (\text{A.8})$$

以上の式 (A.6)、(A.7)、(A.8) を利用することで、tanh 関数の微分は次のように求めることができます。

$$\begin{aligned} \frac{\partial \tanh(x)}{\partial x} &= \frac{(e^x + e^{-x})(e^x + e^{-x}) - (e^x - e^{-x})(e^x - e^{-x})}{(e^x + e^{-x})^2} \\ &= 1 - \frac{(e^x - e^{-x})(e^x - e^{-x})}{(e^x + e^{-x})^2} \\ &= 1 - \left\{ \frac{(e^x - e^{-x})}{(e^x + e^{-x})} \right\}^2 \\ &= 1 - \tanh(x)^2 \\ &= 1 - y^2 \end{aligned} \quad (\text{A.9})$$

式 (A.9) のように、tanh 関数の微分は「分数関数の微分の公式」を用い、簡単な式の整形によって求められます。そしてその結果は、 $1 - y^2$ となりました。この結果から、tanh 関数の計算グラフは図 A-7 のように書くことができます。

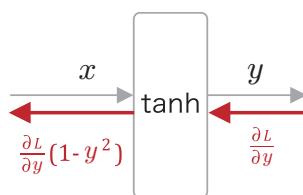


図 A-7 tanh 関数の計算グラフ

以上が tanh 関数の微分の導出です。数式の展開によって、簡潔に、そして明確に微分を求めることができました。

A.3 まとめ

ここで私たちは、微分を計算グラフと解析的な方法の 2 つの方法で解きました。この 2 つの方法は、ともに同じゴールにたどり着きます。そのため、問題に応じて、適宜どちらかを利用すればよいでしょう。おそらく慣れてくると数式のほうが都合が良いと思います。しかし、最初は——特に疑問に思った場合は——、計算グラフによる視覚的な方法が有効かもしれません。また、問題を複数の方法で解決できるというのは、時にとても重要です。ここで学んだように、あるときは数式で、あるときは計算グラフで問題を眺めることで、理解がより深まるでしょう。

付録B WordNetを動かす

本付録では、WordNet を実際に動かしてみます。Python から実際に動かすことで、WordNet にどのような“知識”が存在するのかを見てみます。なお、これから行う実装は、シソーラスの雰囲気を感じてもらうことを目的として、簡単な実験を行うにとどめます。



ここでは、WordNet や NLTK について簡単な紹介にとどめます。NLTK についての詳しい解説は、『入門 自然言語処理』[14] にあります。興味のある方は、参照してください。

B.1 NLTKのインストール

WordNet を Python から利用するには、**NLTK** (Natural Language Toolkit) というライブラリを使用します。NLTK とは、自然言語処理のための Python ライブラリであり、自然言語処理に関する便利な機能が多く収納されています。たとえば、品詞タグ付けや構文解析、情報抽出や意味解析などの機能があります。

それでは早速、NLTK をインストールしましょう。インストールについてはいくつか方法がありますが、ここでは pip によるインストールについて説明します（その他のインストール方法については、読者の環境に応じて適宜行ってください）。

NLTK をインストールするには、ターミナルから次の 1 行を入力します。

```
$ pip install nltk
```

これで NLTK のインストールは終わりです（インストールには、しばらく時間がかかります）。インストールが終わったら、正しくインストールできたことを確認す

るために、NLTK のインポートを行います。

```
>>> import nltk
>>>
```

ここでは、Python インタープリタを起動し、NLTK をインポートしました。NLTK が正しくインストールされていれば、上のようにエラーは何も表示されません。

B.2 WordNet で同義語を得る

それでは実際に WordNet を実際に使ってみましょう。それには、`nltk.corpus` から `wordnet` モジュールをインポートします。

```
>>> from nltk.corpus import wordnet
```

これで準備は整いました。それでは試しに「car」という単語について、その同義語を見てみましょう。まずは、「car」という単語にどれだけ異なる意味が存在するのかを確認します。それには `wordnet.synsets()` というメソッドを使います。



WordNet では、各単語が `synset` と呼ばれる同義語のグループに分類されています。「car」の同義語グループを得るには、`wordnet.synset()` というメソッドを呼ぶだけですが、ひとつだけ注意する点があります。それは「car」という単語には（多くの単語と同じように）複数の意味が存在するということです。具体的には、「自動車」という意味の他に、「（鉄道の）車両」や「ゴンドラ」のような意味も存在します。そこで、同義語を取得するにあたっては、（複数の意味の中から）どの意味に該当するものかを指定する必要があります。

それでは、「car」の同義語を WordNet で取得してみましょう。

```
>>> wordnet.synsets('car')
[Synset('car.n.01'),
 Synset('car.n.02'),
 Synset('car.n.03'),
 Synset('car.n.04'),
 Synset('cable_car.n.01')]
```

ここでは、5つの要素からなるリストが出力されました。これが意味することは、「car」という単語には 5 つの異なる意味——正確には、5 つの異なる同義語グループ

——が定義されているということです。

また、上のリストの要素には、car の「見出し語」が表示される点も大切なポイントです。WordNet で用いられる「見出し語」は、図 B-1 のように、ドットで区切られた 3 つの要素で指定されます。たとえば、「car.n.01」という見出し語は、「car」という名詞の 1 番目」の意味（グループ）であることを表します。

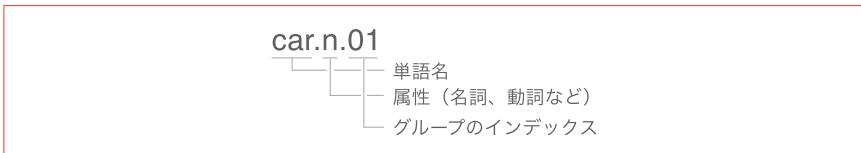


図 B-1 WordNet の「見出し語」の読み方 : n は、noun (名詞) の頭文字



多くの単語には複数の意味があります。WordNet では、ある単語に対して、複数の意味の中から特定の意味の単語を指定するために「見出し語」を使います。そのため、WordNet のメソッドで引数に単語名を指定する場合、「car」と指定するのではなく、「car.n.01」や「car.n.02」のように見出し語を指定して利用するケースが多くあります。

それでは、「car.n.01」という見出し語で指定される同義語について、その意味を確認してみましょう。それには、`wordnet.synset()` メソッドを使って、「car.n.01」の同義語グループを取得します。そして、その同義語グループに対して、`definition()` メソッドを呼びます。

```

>>> car = wordnet.synset('car.n.01') # 同義語グループ
>>> car.definition()
'a motor vehicle with four wheels; usually propelled by an internal
combustion engine'
  
```

上の結果を直訳すると、「4 つの車輪を有する自動車。通常は内燃機関によって進む。」ということになります。これが「car.n.01」という同義語グループの持つ意味です。なお、ここで利用した `definition()` メソッドは、主に（コンピュータではなく）人がその単語を理解するために利用されます。

それでは、「car.n.01」という見出し語の同義語グループには、どのような単語が存在するのかを実際に見てみましょう。それには、`lemma_names()` というメソッドを使います。

```
>>> car.lemma_names()
['car', 'auto', 'automobile', 'machine', 'motorcar']
```

このように `lemma_names()` というメソッドを使えば、同義語グループに存在する単語名を得ることができます。上の結果を見ると、「car」という単語——正確には、「car.n.01」という見出し語——には、auto、automobile、machine、motorcar の 4 つの単語が同義語として定義されていることが分かります。

B.3 WordNet と単語ネットワーク

続いて、「car」の単語ネットワークを使って、他の単語との意味的な上位・下位の関係性について見てみます。それには、`hypernym_paths()` というメソッドを用います。なお、`hypernym` とは主に言語学で用いられる単語で、「上位語」という意味があります。

```
>>> car.hypernym_paths()[0]
[Synset('entity.n.01'), Synset('physical_entity.n.01'),
Synset('object.n.01'), Synset('whole.n.02'), Synset('artifact.n.01'),
Synset('instrumentality.n.03'), Synset('container.n.01'),
Synset('wheeled_vehicle.n.01'), Synset('self-propelled_vehicle.n.01'),
Synset('motor_vehicle.n.01'), Synset('car.n.01')]
```

上の結果を見ると、「car」という単語は「entity」という単語からスタートして、「entity → physical_entity → object → … → motor_vehicle → car」という経路をたどることができます（ここでは、「見出し語」の表記は省略して記します）。具体的に各単語を見ていくと、「car」の上には、「motor vehicle (自動車)」という単語が位置し、その上には「self-propelled vehicle (自走車)」という単語が来ています。そして、さらに上に行くと、「object」や「entity」といった抽象的な単語が来ます。このように、WordNet を構成する単語ネットワークでは、上に行くに従ってより抽象的に、下に行くに従ってより具体的になるように、各単語が配置されています。



上の例において、`car.hypernym_paths()` はリストを返します。そのリストの要素に、具体的な経路の情報が格納されています。なぜリストを返すかというと、それは単語間の経路が複数存在するからです。上の例では、出発点の「entity」という単語から終着点の「car」まで複数の経路が存在します（単語によっては経路がひとつの場合もあります）。

B.4 WordNetによる意味の類似度

これまで説明してきたように、WordNetでは、多くの単語が同義語（類義語）によってグループ分けされています。また、単語間で意味的なネットワークが構築されています。そのような単語間のつながりの知識は、さまざまな問題で活用できます。ここでは、そのひとつの例として、単語間の類似度を算出する例を見てみましょう。

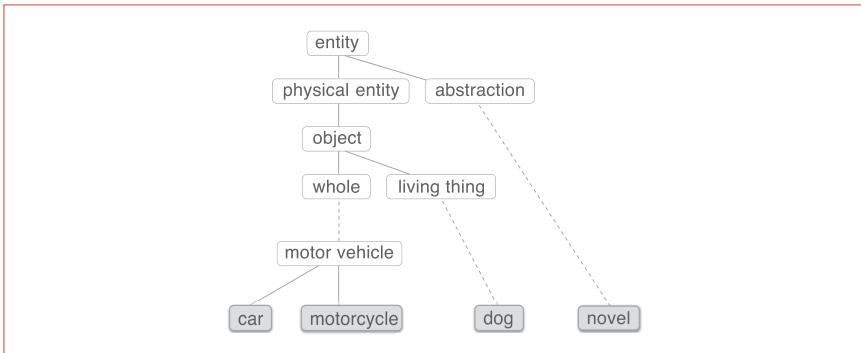
単語間の類似度を求めるには、`path_similarity()`というメソッドを用います。このメソッドは単語間の類似度を返しますが、その返す値は0から1の範囲の実数になります（数値が高いほど類似している）。それでは、単語間の類似度を実際に求めてみましょう。ここでは、「car（自動車）」という単語に対して、「novel（小説）」、「dog（犬）」、「motorcycle（オートバイ）」の3つの単語の類似度をそれぞれ求めてみます。

```
>>> car = wordnet.synset('car.n.01')
>>> novel = wordnet.synset('novel.n.01')
>>> dog = wordnet.synset('dog.n.01')
>>> motorcycle = wordnet.synset('motorcycle.n.01')

>>> car.path_similarity(novel)
0.0555555555555555
>>> car.path_similarity(dog)
0.07692307692307693
>>> car.path_similarity(motorcycle)
0.3333333333333333
```

ここで得られた結果を見てみると、「car」という単語に対しては、「motorcycle」という単語の類似度が最も高く、続いて「dog」、そして、最も似ていない単語が「novel」ということになりました。また、類似度の値を見てみると、「car」と「motorcycle」の類似度が大きく、その値は他の2つの単語に比べて何倍も大きな値になっています。このような結果を見ると、確かに私たちの感覚に近いものであると言えそうです。

さて、上の例で使用した`path_similarity()`というメソッドですが、このメソッドの裏側では、図B-2のような単語ネットワークの共通する経路を元に、単語間の類似度が計算されています。



図B-2 単語の意味的近さを、単語ネットワークの共通する経路を元に算出する（破線は、途中に複数の単語が存在することを示す）

図B-2は、WordNetの単語ネットワークを抜粋して——途中の単語を省略して——示したものです。この図から、「car」と「motorcycle」は、多くの経路が共通していることが分かります。実際、下から2番目の「motor vehicle」という単語までの経路が同じです。一方、「car」と「dog」を見比べると、その経路は「object」という単語で枝分かれしています。さらに、「car」と「novel」を比較すると、一番上の「entity」という単語すでに枝分かれています。`path_similarity()`というメソッドは、このような情報を元に単語間の類似度を算出しており、(今回の例では)その結果は私たちの感覚に近いものになりました。

このように、単語ネットワークを使えば、2つの単語間の類似度を算出することができます。単語間の類似度を求められるということは、単語と単語の意味的な近さを計測できるということです。そのような作業は、「単語の意味」を理解していないと正しく行えません。その点において、シソーラスはコンピュータに「単語の意味」を(間接的に)与えていると解釈できるでしょう。



WordNetには、`path_similarity()`というメソッドの他にも類似度を計測するための手法がいくつか用意されています(Leacock-Chodorow類似度やWu-Palmer類似度など)。興味のある方はWordNetのWebドキュメント[18]を参照してください。

付録C GRU

6章では「ゲート付きRNN」としてLSTMを詳しく説明しました。LSTMは大変良いレイヤですが、パラメータが多く、計算には時間がかかります。そこで最近では、LSTMに代わる「ゲート付きRNN」が数多く提案されています。ここでは、**GRU**(Gated Recurrent Unit) [42]と呼ばれる有名で実績のあるゲート付きRNNを紹介します。GRUは、LSTMのゲートを使用するというコンセプトはそのまままで、パラメータを削減し、計算時間を短縮します。それでは、GRUの中身を見ていきましょう。

C.1 GRUのインターフェース

LSTMの重要な点は、ゲートを使用することです。それによって、学習時における勾配はスムーズに流れるため、勾配消失を緩和できます。この思想をGRUも受け継いでいます。ただし、LSTMとGRUはいくつか相違点が見られます。大きな違いは、図C-1のように、レイヤのインターフェースに見られます。

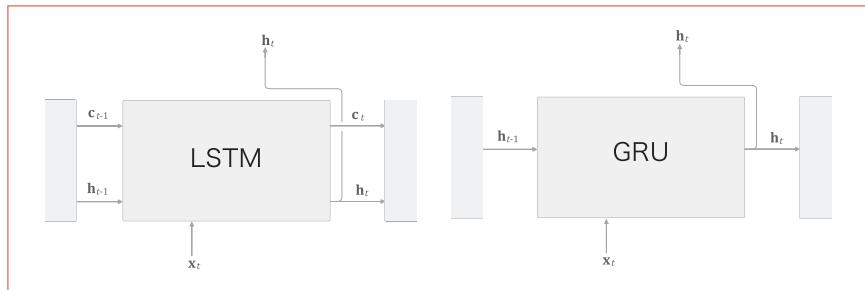


図 C-1 LSTM と GRU の比較

図 C-1 に示すように、LSTM が隠れ状態と記憶セルの 2 つのラインを使用するのに対して、GRU は隠れ状態だけを使用します。ちなみに、これは 5 章で扱った「シンプルな RNN」と同じインターフェースです。



LSTM の記憶セルは、他のレイヤに見せることのない、プライベートな記憶部です。LSTM は記憶セルに必要な情報を記録し、記憶セルの情報を元に隠れ状態を求めました。それに対して、GRU は記憶セルのような追加の記憶部を必要としません。

C.2 GRU の計算グラフ

それでは、GRU の内部で行う計算を見ていきましょう。ここでは、GRU で行う計算を数式で示し、それに対応させる形で計算グラフを示します。なお、計算グラフは、6 章の LSTM の計算グラフで用いた「 σ 」や「 \tanh 」などの簡略化したノードを用います。

$$\mathbf{z} = \sigma(\mathbf{x}_t \mathbf{W}_x^{(z)} + \mathbf{h}_{t-1} \mathbf{W}_h^{(z)} + \mathbf{b}^{(z)}) \quad (\text{C.1})$$

$$\mathbf{r} = \sigma(\mathbf{x}_t \mathbf{W}_x^{(r)} + \mathbf{h}_{t-1} \mathbf{W}_h^{(r)} + \mathbf{b}^{(r)}) \quad (\text{C.2})$$

$$\tilde{\mathbf{h}} = \tanh(\mathbf{x}_t \mathbf{W}_x + (\mathbf{r} \odot \mathbf{h}_{t-1}) \mathbf{W}_h + \mathbf{b}) \quad (\text{C.3})$$

$$\mathbf{h}_t = (1 - \mathbf{z}) \odot \mathbf{h}_{t-1} + \mathbf{z} \odot \tilde{\mathbf{h}} \quad (\text{C.4})$$

GRU で行う計算は、上の 4 つの式で表されます（ここで \mathbf{x}_t と \mathbf{h}_{t-1} は行ベクトルとします）。そして、この計算グラフは図 C-2 のようになります。

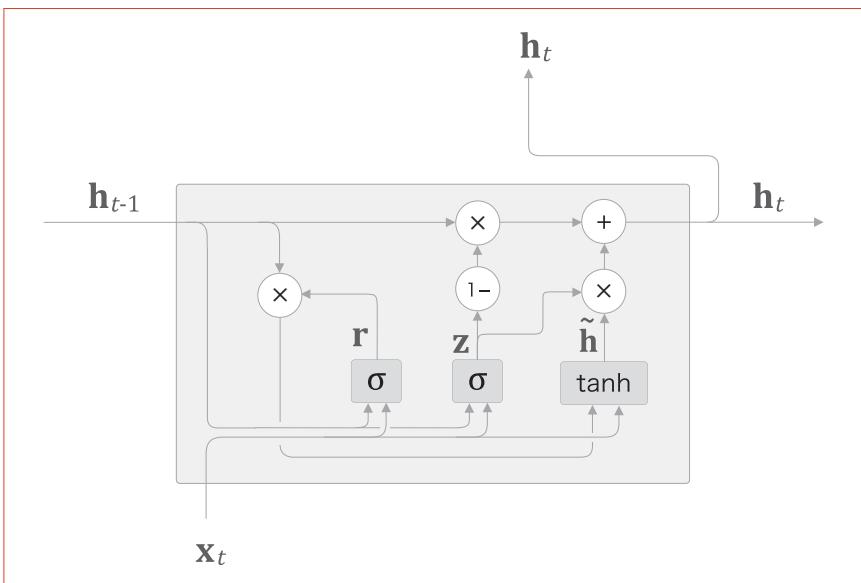


図 C-2 GRU の計算グラフ：「 σ 」と \tanh ノードには専用の重みがあり、ノード内部でアフィン変換を行う（「 $1 -$ 」ノードは、 x を入力すると $1 - x$ を出力するノード）

図 C-2 のとおり、GRU には記憶セルはなく、時間方向への伝播は、隠れ状態の \mathbf{h} ひとつだけです。そして、そこで使われるゲートは、 \mathbf{r} と \mathbf{z} の 2 つのゲートになります（ちなみに、LSTM は 3 つのゲートを使用しました）。ここで、 \mathbf{r} は reset ゲート、 \mathbf{z} は update ゲートと呼ばれます。

reset ゲートの \mathbf{r} は、過去の隠れ状態をどれだけ“無視”するのかを決定します。もし \mathbf{r} が 0 であれば、式 (C.3) より、新しい隠れ状態 $\tilde{\mathbf{h}}$ は、入力 \mathbf{x}_t のみから決定されます。つまり、このとき過去の隠れ状態は完全に無視されることになります。

一方、update ゲートは、隠れ状態を更新するゲートです。これは LSTM の forget ゲートと input ゲートの 2 つの役割を担います。forget ゲートとして機能するのは、式 (C.4) の $(1 - \mathbf{z}) \odot \mathbf{h}_{t-1}$ の箇所です。この計算によって、過去の隠れ状態から忘れるべき情報を消去します。そして input ゲートとして機能するのは、 $\mathbf{z} \odot \tilde{\mathbf{h}}$ の箇所です。これによって、新しく追加する情報に対して input ゲートの重み付けが行われ

ます。

このように、GRU は、LSTM を“よりシンプル”にしたアーキテクチャです。そのため、LSTM に比べて計算コストを抑え、パラメータも少なくすることができます。なお、GRU レイヤの実装はここでは行いませんが、それは `common/time_layers.py` にあります。興味のある方は参照してください。



LSTM と GRU のどちらを使うべきでしょうか？文献[32]や[33]によると、タスクやハイパーパラメータの調整によって、勝者は変動するようです。最近の研究では、LSTM（や LSTM の亜種）が多く使われています。一方、GRU も着々と人気を集めています。GRU はパラメータが少なく計算量も小さいため、データセットのサイズが小さい場合や、モデル設計で繰り返しの試行が必要な場合に特に適していると言えるでしょう。

おわりに

険しい丘に登るためには、最初にゆっくりと歩くことが重要である

—— シェイクスピア

ここまで私たちは、自然言語処理をテーマにディープラーニングの世界をゆっくりと歩いてきました。実際に、自然言語処理に関するさまざまなコードを実装し、多くの実験を通していくつの重要な技術を学んできました。読者の皆さんにおかれでは、そのような経験を通して、いくらかの学びがあったことを願っています。そして、そこにおもしろさや奥深さを感じてもらえたとしたら、著者として、それに勝る喜びはありません。

現在、ディープラーニングの分野は加速度的に進歩しています。大量の論文が毎日のように公開され、新たなアイデアがひっきりなしに提案されています。残念ながら、それらすべてに目を通すことはすでに不可能でしょう。この先ディープラーニングがどのような道を進むのか——それを正しく明確に予測することは誰にもできないはずです。

しかし、その一方で、ディープラーニングにおいて重要な技術は（ある程度）固まっています。本書で学んだディープラーニングに関する技術は、これから先も重要であり続けると筆者は考えています。本書で学んだ知識を足がかりにすれば、さらに広大な現在進行形のディープラーニングの世界へ、確かな足取りで進めるものと期待します。

これまで本書も終わりです。ここまで本書を読み進めていただき、本当にありがとうございました。歴史的に見れば、私たちはディープラーニングが世の中に浸透し、それが世界を変える過程を目のあたりにしている世代です。筆者はたまたまそのような

時代に生まれ、たまたまそのようなテーマの本を書いたにすぎません。しかし、そのような偶然の結びつきによって、こうやって、あなたと本書を通じて交流できたことを嬉しく思います。ありがとうございました。

謝辞

本書が存在するのは、これまでにディープラーニングや人工知能、さらには自然科学を推し進めてこられた偉大な先人たちがあつてのことです。まずは、そのような方々に感謝申し上げます。そして、僕の周りの人たちの支えや協力に心より感謝します。それらは直接的に、また間接的に僕を励まし支えてくれました。ありがとうございました。

また、本書の執筆では、新しい試みとして「公開レビュー」という校正方法を採用しました。この公開レビューでは、本書の原稿をWeb上に公開し、誰でも閲覧とコメントができるようにしました。その結果、わずか1ヶ月程度のレビュー期間ながら、1,500件を超える有用なコメントをいただきました。レビューに参加していただいた方には、心よりお礼申し上げます。本書をより一層磨くことができたのは、まぎれもなく、そのようなレビュアの方々のお力添えによるものです。本当に、ありがとうございました。もちろん、本書に不備や誤りがある場合、それは著者の責任であり、レビュアの方には一切の責任はありません。

最後に、本書が存在するのは、世界中の人々のおかげです。名前を知らない多くの人から、僕は日々影響を受けています。その誰か一人が欠けても、このような本は（少なくとも今の状態では）存在しなかったと思うのです。それと同じことは、僕の周りの自然にも言えます。河や木に、土や空に、僕の日常があります。名もなき人に、名もなき自然に心より感謝申し上げます。

2018年6月1日

斎藤 康毅

査読

| | | | | |
|----------------------|--------|---------|--------|-------|
| 寄田 明宏 | 玉川 晃之 | 吉永 彰成 | 高山 篤史 | 佐藤 太樹 |
| 小林 大将 | 高屋 英知 | 江崎 大嗣 | 兵頭 沖 | 鈴木 駿 |
| 藤岡 秀明 | 田島 英朗 | 原田 秀隆 | 河村 英幸 | 谷岡 広樹 |
| 荒井 浩 | 布施 快 | 測上 紘行 | 瀬戸口 久雄 | 藤田 勇 |
| 石川 敬規 | 上久保 景幸 | 神戸 宏之 | 金沢 隆史 | 永田 員広 |
| 石津 和紀 | 古賀 一徳 | 佐々木 知哉 | 大滝 啓介 | 山内 健太 |
| 小沢 学 | 寺田 学 | 長谷川 正彦 | 山本 正喜 | 藤井 昌紀 |
| 水谷 穂 | 三浦 康幸 | 野口 宗之 | 内藤 亮介 | 米澤 直記 |
| 佐藤 尚至 | 野口 聰明 | 伊藤 宣博 | 山下 修 | 佐藤 隆佑 |
| 佐藤 温 | 佐藤 潔 | 新谷 正嶺 | 小林 茂 | 清水 豊 |
| 永田 晋介 | 杉田 臣輔 | 和田 信也 | 白木 宏朋 | 中村 翔 |
| 千田 翔太 | 藤原 秀平 | 鈴木 英太 | 阿部 考志 | 宮阪 健夫 |
| 若杉 武史 | 鈴木 琢也 | 新村 拓也 | 田中 智史 | 蔡天星 |
| 石川 哲朗 | 鷹城 徹 | 森原 利之 | 大野 翼 | 土井 泰法 |
| 西田 泰士 | 高野 泰朋 | 長谷部 陽一郎 | 荻原 義寛 | 林 悠大 |
| 増宮 雄一 | 小池 祐二 | 飛永 由夏 | 宵 勇樹 | 藤本 裕介 |
| 古川 悠介 | 西口 祐介 | 吉成 祐人 | 水原 悠 | |
| TAICHI KAWABATA | | | | |
| AIMY —山形県人工知能コミュニティ— | | | | |

制作

武藤 健志
増子 萌

編集

宮川 直樹
岩佐 未央
小柳 彩良

参考文献

Python 関連

- [1] Broadcasting (<https://docs.scipy.org/doc/numpy-1.13.0/user/basics.broadcasting.html>)
- [2] 100 numpy exercises (<https://github.com/rougier/numpy-100>)
- [3] CuPy web page (<https://cupy.chainer.org/>)
- [4] CuPy install page (<https://docs-cupy.chainer.org/en/stable/install.html>)

ディープラーニングの基本事項

- [5] 斎藤康毅：『ゼロから作る Deep Learning —— Python で学ぶディープラーニングの理論と実装』（オライリー・ジャパン）
- [6] Gupta, Suyog, et al : "Deep learning with limited numerical precision." *Proceedings of the 32nd International Conference on Machine Learning (ICML-15)*. 2015.
- [7] Jouppi, Norman P., et al : "In-datacenter performance analysis of a tensor processing unit." *Proceedings of the 44th Annual International Symposium on Computer Architecture*. ACM, 2017.
- [8] Ba, Jimmy Lei, Jamie Ryan Kiros, and Geoffrey E. Hinton : "Layer normalization." arXiv preprint arXiv:1607.06450 (2016).
- [9] Srivastava, Nitish, et al : "Dropout: a simple way to prevent neural networks from overfitting." *Journal of machine learning research* 15.1 (2014): 1929-1958.

ディープラーニングによる自然言語処理

- [10] Stanford University CS224d: Deep Learning for Natural Language Processing (<http://cs224d.stanford.edu/>)
- [11] Oxford Deep NLP 2017 course (<https://github.com/oxford-cs-deepnlp-2017/lectures>)
- [12] Young, D. Hazarika, S. Poria, and E. Cambria : "Recent trends in deep learning based natural language processing," in arXiv preprint arXiv:1708.02709, 2017.
- [13] 坪井祐太、海野裕也、鈴木潤：『深層学習による自然言語処理（機械学習プロフェッショナルシリーズ）』（講談社）

ディープラーニング登場以前の自然言語処理

- [14] Steven Bird, Ewane Klein, Edward Loper :『入門 自然言語処理』萩原正人、中山敬広、水野貴明 訳（オライリー・ジャパン）
- [15] Jeffrey E.F. Friedl :『詳説 正規表現 第3版』株式会社ロングテール／長尾高弘 訳（オライリー・ジャパン）
- [16] Christopher D. Manning、Hinrich Schütze :『統計的自然言語処理の基礎』加藤恒昭、菊井玄一郎、林良彦、森辰則 訳（共立出版）
- [17] Miller, George A : "WordNet: a lexical database for English." *Communications of the ACM* 38.11 (1995): 39-41.
- [18] WordNet Interface (<http://www.nltk.org/howto/wordnet.html>)

カウントベース手法による単語ベクトル

- [19] Church, Kenneth Ward, and Patrick Hanks : "Word association norms, mutual information, and lexicography." *Computational linguistics* 16.1 (1990): 22-29.
- [20] Deerwester, Scott, et al : "Indexing by latent semantic analysis." *Journal of the American society for information science* 41.6 (1990): 391.
- [21] TruncatedSVD(<http://scikit-learn.org/stable/modules/generated/sklearn.decomposition.TruncatedSVD.html>)

word2vec 関連

- [22] Mikolov, Tomas, et al : "Efficient estimation of word representations in vector space." arXiv preprint arXiv:1301.3781 (2013).
- [23] Mikolov, Tomas, et al : "Distributed representations of words and phrases and their compositionality." *Advances in neural information processing systems*. 2013.
- [24] Baroni, Marco, Georgiana Dinu, and Germán Kruszewski : "Don't count, predict! A systematic comparison of context-counting vs. context-predicting semantic vectors." *ACL* (1). 2014.
- [25] Levy, Omer, Yoav Goldberg, and Ido Dagan : "Improving distributional similarity with lessons learned from word embeddings." *Transactions of the Association for Computational Linguistics* 3 (2015): 211-225.
- [26] Levy, Omer, and Yoav Goldberg : "Neural word embedding as implicit matrix factorization." *Advances in neural information processing systems*. 2014.
- [27] Pennington, Jeffrey, Richard Socher, and Christopher D. Manning : "Glove: Global Vectors for Word Representation." *EMNLP*. Vol.14. 2014.
- [28] Bengio, Yoshua, et al. "A neural probabilistic language model." *Journal of machine learning research* 3.Feb (2003): 1137-1155.

RNN 関連

- [29] Talathi, Sachin S., and Aniket Vartak : "Improving performance of recurrent neural network with relu nonlinearity." arXiv preprint arXiv:1511.03771 (2015).
- [30] Pascanu, Razvan, Tomas Mikolov, and Yoshua Bengio : "On the difficulty of training recurrent neural networks." *International Conference on Machine Learning*. 2013.
- [31] colah's blog : "Understanding LSTM Networks" (<http://colah.github.io/posts/2015-08-Understanding-LSTMs/>)
- [32] Chung, Junyoung, et al : "Empirical evaluation of gated recurrent neural networks on sequence modeling." arXiv preprint arXiv:1412.3555 (2014).

- [33] Jozefowicz, Rafal, Wojciech Zaremba, and Ilya Sutskever : "An empirical exploration of recurrent network architectures." *International Conference on Machine Learning*. 2015.

RNNによる言語モデル

- [34] Merity, Stephen, Nitish Shirish Keskar, and Richard Socher : "Regularizing and optimizing LSTM language models." arXiv preprint arXiv:1708.02182 (2017).
- [35] Zaremba, Wojciech, Ilya Sutskever, and Oriol Vinyals : "Recurrent neural network regularization." arXiv preprint arXiv:1409.2329 (2014).
- [36] Gal, Yarin, and Zoubin Ghahramani : "A theoretically grounded application of dropout in recurrent neural networks." *Advances in neural information processing systems*. 2016.
- [37] Press, Ofir, and Lior Wolf : "Using the output embedding to improve language models." arXiv preprint arXiv:1608.05859 (2016).
- [38] Inan, Hakan, Khashayar Khosravi, and Richard Socher : "Tying Word Vectors and Word Classifiers: A Loss Framework for Language Modeling." arXiv preprint arXiv:1611.01462 (2016).
- [39] PyTorch examples "Word-level language modeling RNN" (https://github.com/pytorch/examples/tree/0.3/word_language_model)

seq2seq 関連

- [40] Keras examples "Implementation of sequence to sequence learning for performing addition of two numbers (as strings)" (https://github.com/keras-team/keras/blob/2.0.0/examples/addition_rnn.py)
- [41] Sutskever, Ilya, Oriol Vinyals, and Quoc V. Le : "Sequence to sequence learning with neural networks." *Advances in neural information processing systems*. 2014.
- [42] Cho, Kyunghyun, et al : "Learning phrase representations using RNN encoder-decoder for statistical machine translation." arXiv preprint arXiv:1406.1078 (2014).
- [43] Vinyals, Oriol, and Quoc Le : "A neural conversational model." arXiv preprint arXiv:1506.05869 (2015).

- [44] Zaremba, Wojciech, and Ilya Sutskever : "Learning to execute." arXiv preprint arXiv:1410.4615 (2014).
- [45] Vinyals, Oriol, et al : "Show and tell: A neural image caption generator." *Computer Vision and Pattern Recognition (CVPR)*, 2015 IEEE Conference on. IEEE, 2015.
- [46] Karpathy, Andrej, and Li Fei-Fei : "Deep visual-semantic alignments for generating image descriptions." *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2015.
- [47] Show and Tell: A Neural Image Caption Generator (<https://github.com/tensorflow/models/tree/master/research/im2txt>)

Attention 関連

- [48] Bahdanau, Dzmitry, Kyunghyun Cho, and Yoshua Bengio : "Neural machine translation by jointly learning to align and translate." arXiv preprint arXiv:1409.0473 (2014).
- [49] Luong, Minh-Thang, Hieu Pham, and Christopher D. Manning: "Effective approaches to attention-based neural machine translation." arXiv preprint arXiv:1508.04025 (2015).
- [50] Wu, Yonghui, et al : "Google's neural machine translation system: Bridging the gap between human and machine translation." arXiv preprint arXiv:1609.08144 (2016).
- [51] Google Research Blog. (<https://research.googleblog.com/2016/09/a-neural-network-for-machine.html>)
- [52] Vaswani, Ashish, et al : "Attention Is All You Need." arXiv preprint arXiv:1706.03762 (2017).
- [53] Google Research Blog. (<https://research.googleblog.com/2017/08/transformer-novel-neural-network.html>)
- [54] Gehring, Jonas, et al : "Convolutional Sequence to Sequence Learning." arXiv preprint arXiv:1705.03122 (2017).

外部メモリ付き RNN

- [55] Graves, Alex, Greg Wayne, and Ivo Danihelka : "Neural turing machines." arXiv preprint arXiv:1410.5401 (2014).

- [56] Graves, Alex, et al : "Hybrid computing using a neural network with dynamic external memory." *Nature* 538.7626 (2016): 471.
- [57] DeepMind Blog : "Differentiable neural computers" (<https://deepmind.com/blog/differentiable-neural-computers/>)

索引

記号・数字

| | |
|--------|----|
| <eos> | 87 |
| <unk> | 86 |
| 3点リーダー | 34 |

A

| | |
|----------------------|-----|
| AdaGrad | 41 |
| Adam | 41 |
| Affine レイヤ | 36 |
| alignment | 330 |
| argsort() | 76 |
| Attention | 325 |
| attention mechanism | 325 |
| AttentionDecoder クラス | 349 |
| AttentionEncoder クラス | 348 |
| AttentionSeq2seq クラス | 351 |
| AttentionWeight クラス | 341 |
| Attention クラス | 344 |
| Attention レイヤ | 343 |
| axis | 29 |

B

| | |
|------------------------------|-----|
| back-propagation | 23 |
| Backpropagation Through Time | 188 |
| BaseModel クラス | 255 |

| | |
|-----------------|-----|
| BetterRnnlm クラス | 269 |
| BLEU スコア | 373 |
| BPTT | 188 |
| broadcast | 5 |

C

| | |
|------------------------------|-----|
| CBOW | 101 |
| CBOW クラス | 161 |
| clip_grads() | 233 |
| CNN | 321 |
| co-occurrence matrix | 71 |
| column | 2 |
| continuous bag-of-words | 101 |
| Convolutional Neural Network | 321 |
| corpus | 63 |
| cosine similarity | 73 |
| Cross Entropy Error | 19 |
| Cross Entropy Error レイヤ | 19 |
| CuPy | 52 |

D

| | |
|---------------------------------|-----|
| decision boundary | 48 |
| Decoder | 287 |
| Decoder クラス | 303 |
| deep copy | 34 |
| Differentiable Neural Computers | 374 |

| | |
|---------------------------------|-----|
| dimensionality reduction | 81 |
| distributed representation..... | 135 |
| distributional hypothesis | 68 |
| DNC | 374 |
| DropConnect | 273 |
| Dropout | 262 |
| dtype | 51 |

E

| | |
|---------------------------|-----|
| element-wise | 4 |
| Elman | 223 |
| Embedding Dot レイヤ | 148 |
| EmbeddingDot クラス | 149 |
| Embedding レイヤ | 135 |
| Encoder | 287 |
| Encoder-Decoder モデル | 287 |
| Encoder クラス | 298 |
| exploding gradients | 230 |

F

| | |
|------------------|-----|
| forget ゲート | 240 |
|------------------|-----|

G

| | |
|--|-----|
| Gated Recurrent Unit | 393 |
| GloVe | 127 |
| GNMT | 367 |
| Google Neural Machine Translation | 367 |
| GPU | 52 |
| Gradient Descent | 40 |
| gradients clipping | 232 |
| GRU | 393 |

H

| | |
|--------------------------|----------|
| hidden state | 187, 224 |
| hyperbolic tangent | 384 |
| hypernym_paths() | 390 |

I

| | |
|------------------|-----|
| id_to_word | 65 |
| ImageNet | 322 |
| input ゲート | 242 |

J

| | |
|--------------|-----|
| join() | 284 |
|--------------|-----|

K

| | |
|----------------|----|
| keepdims | 29 |
|----------------|----|

L

| | |
|---------------------|-----|
| L2 ノルム | 73 |
| learning rate | 41 |
| loss | 18 |
| loss function | 18 |
| lower() | 64 |
| LSTM | 234 |
| LSTM クラス | 246 |

M

| | |
|------------------|----|
| MatMul ノード | 30 |
| Momentum | 41 |

N

| | |
|----------------------------------|-----|
| NaN | 230 |
| ndim | 4 |
| negative log likelihood | 123 |
| Negative Sampling | 140 |
| Neural Machine Translation | 367 |
| Neural Turing Machine | 374 |
| NLTK | 387 |
| np.add.at() | 139 |
| np.array() | 4 |
| np.dot() | 7 |
| np.float32 | 51 |
| np.hstack() | 249 |
| np.ndarray クラス | 4 |

| | |
|-------------------------|----------|
| np.random.choice() | 155 |
| np.random.permutation() | 46 |
| np.random.randn() | 44 |
| np.repeat() | 29 |
| np.sum() | 29 |
| np.vstack() | 249 |
| np.zeros() | 44 |
| NTM | 374 |
| RNN | 175, 183 |
| RNN Language Model | 205 |
| RNNLM | 205 |
| RnnlmGen クラス | 282 |
| RnnlmTrainer クラス | 219, 233 |
| Rnnlm クラス | 253 |
| RNN クラス | 197 |
| RNN レイヤ | 184, 196 |
| row | 2 |

O

| | |
|--------------|--------|
| one-hot 表現 | 96 |
| one-hot ベクトル | 20, 96 |
| output ゲート | 238 |
| overfitting | 261 |

P

| | |
|------------------------------|-----|
| path_similarity() | 391 |
| peek | 314 |
| Peeky | 313 |
| PeekyDecoder クラス | 315 |
| PeekySeq2seq クラス | 316 |
| Penn Treebank | 86 |
| perplexity | 214 |
| pickle | 164 |
| plt.annotate() | 85 |
| PMI | 78 |
| Pointwise Mutual Information | 78 |
| Positive PMI | 79 |
| PPMI | 79 |
| PTB コーパス | 86 |

R

| | |
|--------------------------|----------|
| Recurrent Neural Network | 175, 183 |
| Recursive Neural Network | 183 |
| ReLU | 228 |
| repeat() | 334 |
| Repeat ノード | 28 |
| replace() | 64 |
| reset ゲート | 395 |
| residual コネクション | 366 |
| ResNet | 322 |

S

| | |
|------------------------------|--------|
| Self-Attention | 370 |
| seq2seq | 287 |
| Seq2seq クラス | 305 |
| SGD | 40 |
| shallow copy | 34 |
| shape | 4 |
| short-term memory | 244 |
| Sigmoid with Loss レイヤ | 146 |
| sigmoid 関数 | 381 |
| Sigmoid レイヤ | 35 |
| SimpleRnnlm クラス | 211 |
| Singular Value Decomposition | 82 |
| skip-gram | 123 |
| skip コネクション | 366 |
| sklearn | 88 |
| slice ノード | 246 |
| Softmax with Loss レイヤ | 20, 38 |
| Softmax 関数 | 19 |
| Softmax レイヤ | 19 |
| split() | 64 |
| stateful | 202 |
| Stochastic Gradient Descent | 40 |
| Sum ノード | 29 |
| SVD | 82 |
| sys.path.append('..') | 42 |

T

| | |
|--------------------|-----|
| tanh 関数 | 384 |
| thesaurus | 59 |
| Time Affine レイヤ | 208 |
| Time Attention レイヤ | 346 |

| | | | |
|----------------------------|---------------|------------|-----|
| Time Embedding レイヤ | 208 | アダマール積 | 239 |
| Time RNN レイヤ | 196, 200, 250 | アナロジー問題 | 166 |
| Time Softmax with Loss レイヤ | 209 | アフィン変換 | 14 |
| TimeAttention クラス | 347 | アライメント | 330 |
| TimeLSTM クラス | 250 | イメージキャプション | 321 |
| TimeRNN クラス | 201 | ウインドウサイズ | 68 |
| Trainer クラス | 48 | エポック | 46 |
| transfer learning | 169 | エルマン | 223 |
| Transformer | 370 | エンコード | 104 |
| Truncated BPTT | 189 | 重み | 9 |
| Truncated SVD | 86 | 重み共有 | 266 |

U

| | |
|--------------------|-----|
| UnigramSampler クラス | 157 |
| update ゲート | 395 |

V

| | |
|---------------------|-----|
| vanishing gradients | 231 |
| variational dropout | 265 |
| VGG | 322 |

W

| | |
|-------------------|-----|
| Watson | 58 |
| weight tying | 266 |
| WeightSum クラス | 337 |
| WMT | 351 |
| word embedding | 135 |
| word_to_id | 65 |
| word2vec | 101 |
| WordNet | 61 |
| wordnet.synset() | 389 |
| wordnet.synsets() | 388 |

X

| | |
|-------------|-----|
| Xavier の初期値 | 212 |
|-------------|-----|

あ行

| | |
|-----------|----|
| アクティベーション | 13 |
| 浅いコピー | 34 |

か行

| | |
|------------|----------|
| カウントベースの手法 | 69 |
| 過学習 | 261 |
| 学習係数 | 41 |
| 確率的 | 281 |
| 確率的勾配降下法 | 40 |
| 隠れ状態 | 187, 224 |
| 隠れ状態ベクトル | 187 |
| 隠れ層 | 8 |
| 加算ノード | 26 |
| 過剰適合 | 261 |
| 活性化関数 | 12 |
| 可変長の時系列データ | 293 |
| 感情分析 | 170 |
| 記憶セル | 235 |
| 逆伝播 | 14 |
| 行 | 2 |
| 共起行列 | 71 |
| 行ベクトル | 2 |
| 行列の積 | 6 |
| クラスの継承 | 282 |
| 訓練データ | 43 |
| 計算グラフ | 24 |
| 形状チェック | 7 |
| ゲート | 236 |
| 決定境界 | 48 |
| 決定的 | 281 |
| 言語モデル | 178 |
| 検索パス | 42 |
| 検証データ | 43 |
| 交差エントロピー誤差 | 19, 20 |

| | |
|----------|-----|
| 勾配 | 22 |
| 勾配クリッピング | 232 |
| 勾配降下法 | 40 |
| 勾配消失 | 231 |
| 勾配爆発 | 230 |
| コーパス | 63 |
| コサイン類似度 | 73 |
| 誤差逆伝播法 | 23 |
| コンテキスト | 68 |

さ行

| | |
|---------------|----------|
| 再帰ニューラルネットワーク | 183 |
| シグモイド関数 | 12 |
| 次元削減 | 81 |
| 事後確率 | 121 |
| 自然言語 | 57 |
| 自然言語処理 | 57 |
| シソーラス | 59 |
| 出力ゲート | 238 |
| 循環ニューラルネットワーク | 183 |
| 順伝播 | 14 |
| 条件付き言語モデル | 180 |
| 乗算ノード | 26 |
| 乗法定理 | 179 |
| ショートカット | 366 |
| 推論 | 8 |
| 推論ベースの手法 | 95 |
| スカラ | 3 |
| スコア | 103 |
| 正規表現 | 64 |
| 正則化 | 262 |
| 正の相互情報量 | 79 |
| セル | 235 |
| 全結合層 | 9 |
| 双曲線正接関数 | 384 |
| 相互情報量 | 78 |
| 双方向 LSTM | 329 |
| 双方向 RNN | 329, 360 |
| 総和ノード | 29 |
| 損失 | 18 |
| 損失関数 | 18 |

た行

| | |
|-----------|-----|
| ターゲット | 101 |
| 多値分類 | 142 |
| 短期記憶 | 244 |
| 単語の埋め込み | 135 |
| チェインルール | 23 |
| チャットボット | 319 |
| 注意機構 | 325 |
| 中間層 | 8 |
| デコード | 104 |
| テストデータ | 43 |
| 転移学習 | 169 |
| テンソル | 2 |
| トイ・プロblem | 291 |
| 同義語 | 60 |
| 統計的手法 | 69 |
| 同時確率 | 121 |
| 特異値分解 | 82 |

な行

| | |
|---------|-----|
| 内包表記 | 66 |
| 二値分類 | 142 |
| ニューラル翻訳 | 367 |
| 入力ゲート | 242 |
| 覗き見 | 313 |

は行

| | |
|----------------|-----|
| パープレキシティ | 214 |
| バイアス | 9 |
| ハイパボリック・タンジェント | 384 |
| パディング | 293 |
| ピクル | 164 |
| 非線形 | 12 |
| ビット精度 | 50 |
| 微分 | 21 |
| フィードフォワード | 175 |
| 深いコピー | 34 |
| 浮動小数点数 | 50 |
| 負の対数尤度 | 123 |
| 負例サンプリング | 140 |
| ブロードキャスト | 5 |

| | |
|------------------------|-----|
| 分岐ノード | 27 |
| 分散表現 | 67 |
| 文章生成 | 278 |
| 分布仮説 | 68 |
| 平均分岐数 | 216 |
| ベクトル | 1 |
| ベクトルの内積 | 6 |
| ペン・ツリー・バンク | 86 |
| 変分 Dropout | 265 |
| 忘却ゲート | 240 |
| ま行 | |
| マルコフ性 | 181 |
| マルコフ連鎖 | 181 |
| 丸め誤差 | 74 |
| ミニバッチ | 11 |
| モデル | 45 |
| や行 | |
| 要素ごと | 4 |
| ら行 | |
| リカレントニューラルネットワーク | 175 |
| 類義語 | 60 |
| 類推問題 | 166 |
| 列 | 2 |
| 列ベクトル | 2 |
| 連鎖律 | 23 |

●著者紹介

斎藤 康毅（さいとう こうき）

1984年長崎県対馬生まれ。東京工業大学工学部卒、東京大学大学院学際情報学府修士課程修了。現在、企業にて人工知能に関する研究開発に従事。著書に『ゼロから作る Deep Learning』、翻訳書に『実践 Python 3』『コンピュータシステムの理論と実装』『実践 機械学習システム』（以上、オライリー・ジャパン）などがある。

ゼロから作るDeep Learning ②

—自然言語処理編

2018年7月24日 初版第1刷発行

2020年6月3日 初版第2刷発行

| | |
|-------|---|
| 著 者 | 斎藤 康毅 (さいとう こうき) |
| 発 行 人 | ティム・オライリー |
| 制 作 | 株式会社トップスタジオ |
| 印刷・製本 | 日経印刷株式会社 |
| 発 行 所 | 株式会社オライリー・ジャパン 〒160-0002 東京都新宿区四谷坂町12番22号 Tel (03)3356-5227 Fax (03)3356-5263 電子メール japan@oreilly.co.jp |
| 発 売 元 | 株式会社オーム社 〒101-8460 東京都千代田区神田錦町3-1 Tel (03)3233-0641 (代表) Fax (03)3233-3440 |

Printed in Japan (ISBN978-4-87311-836-9)

乱丁本、落丁本はお取り替え致します。

本書は著作権上の保護を受けています。本書の一部あるいは全部について、株式会社オライリー・ジャパンから文書による許諾を得ずに、いかなる方法においても無断で複写、複製することは禁じられています。