第5章 マルコフ連鎖とRNN(Recurrennt Neural Network)

[はじめに]

第 4 章では、ニューラルネットワークの基礎として基本的な仕組みを学んだ．そこでは入力に重み付け和をとって活性化関数で出力を決める基本的なニューラルネットワークを学んだのに対し、そこからの発展として、画像をより良く扱うことのできる CNN（Convolutional Neural Network、畳み込みニューラルネットワーク）や、時系列データを扱う際に威力を発揮するRNN（Recurrennt Neural Network、回帰型ニューラルネットワーク）と呼ばれるものがある．これらはニューラルネットワークの可能性を大きく広げるものであり、ニューラルネットワークの応用そして実用化において、なくてはならないものとなっている．

本章ではRNNを取り上げるが、敢えて回り道をして、マルコフ連鎖をそのまま辞書にして文章を綴る、簡単なプログラムをつくるところからはじめる．そしてその辞書をニューラルネットワークに置き換えて、そこからRNNへとつなげていく．これを進めていく中で、時系列データとはどういうものなのか？ 文章を綴るというのはどういうことなのか？ を考えてほしい．回り道もきっと無駄ではないはずだ．

5.1 マルコフ連鎖

マルコフ連鎖については、ウィキペディアなどにも書かれているが、曰く、「重要な確率過程であって、未来の挙動が現在の値だけで決定され、過去の挙動と無関係である」とされる．そして一連の確率変数 $X_0, X_1, \cdots$ に対し、次式が成立するとされる．

$$Pr(X_{n+1} = x | X_n = x_n, \cdots, X_1 = x_1, X_0 = x_0) = Pr(X_{n+1} = x | X_n = x_n) \qquad (zzz.1.1)$$

ここでたとえば、時刻とともに変化する事象に当てはめるならば、左辺は過去から現在に至る一連の事象が起きた場合に、次の時刻で $X_{n+1} = x$ となる確率を表す．すなわち時刻 0 で、$X_0 = x_0$ 時刻 1 で $X_1 = x_1$ となって、それから現在の時刻 n で $X_n = x_n$ になるまで、各時刻で起きた事象の連なりを前提として、次の時刻に $X_{n+1} = x$ となる確率である．これに対し右辺は現在 $X_n = x_n$ であることだけを前提として、次の時刻に $X_{n+1} = x$ となる確率であって、過去の条件は含まれない．この両者が等しいということは、次の時刻に $X_{n+1} = x$ となる確率が、現在 $X_n = x_n$ であることだけに依存するということである．

5.1.1 文章をマルコフ連鎖で

文章は文字の並び、あるいは、語の並びであって、それらが確率的に次々と決まって綴られていくものと考えることができるから、マルコフ連鎖があてはまるものとして、文章を生成することができそうだ．これをもう少し具体的に考えてみよう．この文章のように横書きの文章は、左から右へと順に文字あるいは語が並ぶ．文章中のどれかある文字あるいは語に着目して、それを現在とするならば、その右隣の文字あるいは語が次に来る未来ということになる．すなわち着目している文字あるいは語だけに依存して、その右隣の文字あるいは語が確率的に決まるということだ．

これをプログラムで実現しようとするならば、python の辞書の機能を使うのが簡単だ．辞書は複数の要素の個々の値に対して一意なキーを与えるが、着目する文字あるいは語をキーとし、その右隣の文字あるいは語を、そのキーで索引される値とすれば良いだろう．ただし索引される値の方は、候補であって、当然ながら複数の候補があるのが普通だから、候補のリストとするのが適切だ．

むしろ例外的には「いろは歌」では 47 文字の仮名が重複なしに使われるから、一つのキーで一つの値を索引できれば良いし、たとえば、「い」の次は「ろ」、「ろ」の次は「は」というように、決まっているから、その確率は 1 だ．

とまれ候補が決まれば、あとは候補の中から、確率的にどれかを選んで、それを次の文字あるいは語とすれば良い．

それではさっそくプログラムを作ってみよう．まずはマルコフ連鎖の辞書を作るプログラムだ.

リスト 5.1.1：マルコフ連鎖の辞書

```
def make_markov(x_chain, verbose=False):
    """ マルコフ連鎖の辞書作成
    P(Xn+1 | Xn=An)
      x       key
    """
    markov = {}
    key = ""
    for x in x_chain:
        if key:
            # 該キーに対応するものがなければ空のリストで用意
            if key not in markov:
                markov[key] = []
                if verbose:
                    print('create key', key)
            # 該キーのリスト中に対象がなかったら加える
            if x not in markov[key]:
                markov[key].append(x)
                if verbose:
                    print('append key', key, ':', x)
        key = x # 現在の対象が次のキーになる
    return markov
```

リスト 5.1.1 に示すプログラムは、マルコフ連鎖の辞書を作る関数で、汎用性を意識した変数名を付けている．引数は文章などを与えるが、ここでは事象の連なりを意識して x_chain としいる．いずれにせよこの引数 x_chain には、文章であれば、それを文字あるいは語に分割して並べたリストを与えることを想定している．引数にはデフォルト引数 verbose もあるが、True を指定すれば、辞書を作成していく過程を、新たなキーを作った場合と、そのキーに対応するリストに値を加えた場合に print 文で表示して確認できるようにしている.

細かいことを先に説明したが、関数の中を順にみていく．はじめに、辞書は markov として空で用意し、キーも key = "" で用意しておく．そして x_chain から一つずつ要素を取出ながらループする．すなわち文章をリスト化したものならば、文字または語を左から一つずつ取り出すことになる．はじめは key = "" だから if key: は成立せずに、ループの中の最後の行にとび、key = x でコメントにあるように現在の対象が次のキーになる．ループの次に進むと 2 つ目の要素が x として取り出されて、今度は if key: は成立する．次の if 文に行くが、辞書 markov に変数 key に入れられた先ほどの x は登録されていないから、変数 key をキーとする新たなエントリを作る．このときそのキーに対する値としては空のリスト[]にしておく．そして次に変数 key をキーとする辞書 markov の値＝リストに、x があるかを調べる．はじめは前述の

ように空のリストを用意したのだから、当然 x はリスト中に無いので、これを
markov[key].append(x) で加える.
あとは今対象とした x を次のループのために key に入れ、また for 分に戻って次の x を取り出して、同様の動作を繰り返す. この間、verbose は前述のとおり. そして、x_chain の末尾まで処理すれば、マルコフ連鎖の辞書 markov が出来上がるから、これを返り値として関数を終わる. 繰り返しになるが、辞書の値をリストとして、その中に複数要素を持てるようにしていることに注意されたし. このプログラムは markov1.py としておこう.

さて、さっそくデバグを兼ねて動かしてみよう.

リスト 5.1.2：いろは歌

```
from markov1 import *

text = ('いろはにほへとちりぬるを'
        'わかよたれそつねならむ'
        'うゐのおくやまけふこえて'
        'あさきゆめみしゑひもせす')

charlist = list(text)
markov = make_markov(charlist, verbose=True)
```

リスト 5.1.2 では、先に例外的とした「いろは歌」を扱う. 変数 text は文字列だが、
list(text)とすれば、一文字ずつ分解されるから、これをリスト 5.1.1 で用意した関数
make_markov()に渡す. このとき verbose=True とすれば、以下のように出力される.

```
create key い
append key い : ろ
create key ろ
append key ろ : は
      :
    以下略
```

また出来た辞書は、以下のように各文字がキーとなって次の文字が索引できるようになっていることが確認できる.

```
>>> markov
{'い': ['ろ'], 'ろ': ['は'], 'は': ['に'], 'に': ['ほ'], 'ほ': ['へ'], 'へ': ['と'],
'と': ['ち'], 'ち': ['り'], 'り': ['ぬ'], 'ぬ': ['る'], 'る': ['を'], 'を': ['わ'], '
… 途中略 …
'ひ': ['も'], 'も': ['せ'], 'せ': ['す']}
```

もう一つ、ごく簡単な文例をやっておこう.

リスト 5.1.3：簡単な文例

```
from markov1 import *

text = '私はあなたが好きです。あなたは私が嫌いです。'

charlist = list(text)
markov = make_markov(charlist, verbose=True)
```

リスト 5.1.2 の text をリスト 5.1.3 のように変えて、マルコフ連鎖の辞書を作ると、辞書の内容は、次のようになっている.

{'私': ['は', 'が'], 'は': ['あ', '私'], 'あ': ['な'], 'な': ['た'], 'た': ['が', 'は'], 'が': ['好', '嫌'], '好': ['き'], 'き': ['で'], 'で': ['す'], 'す': ['。'], '。': ['あ'], '嫌': ['い'], 'い': ['で']}

今度はたとえば、キー'私'に対して['は', 'が']のように、2 つの候補が対応するように辞書が作られたことを確認できる.「私」に着目すると「私は」と「私が」は、この 2 つの文それぞれの主語と目的語で、「あなた」の「た」もキー'た'に対して値は['は', 'が']となっている. このほか関係を示すと次図のようになる.

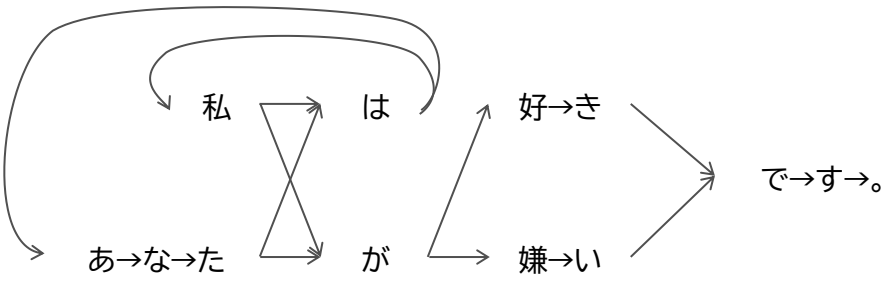

図 5.1.1 文字のつながり

たったこれだけでも十分に複雑だ. とまれ、たとえば「私」で始まれば次は「は」か「が」で、もし「は」ならば、次は「私」か「あ」というようになるから、「私は私が嫌いです。」とか「私が好きです。」とか、元の文になかった文も、この辞書から作ることができる. その際に選択肢が２つ以上ある場合に、確率的に選ぶ必要がある. リスト 5.1.1 により、辞書を索引したら

候補のリストが得られるように辞書を生成するから、そのリストからランダムに選択すれば良いだろう．ではこの文書を生成する操作をプログラミングしておこう．これをリスト 5.1.4 に示す．

リスト 5.1.4：マルコフ連鎖で文章生成

```
import random

def generate_text(markov, length=100, start=None, stop=None):
    """ マルコフ連鎖の辞書を使って自動生成 """
    x_chain = "" # 空の文字列
    if start is None:
        start = random.choice(list(markov))
    x = start
    while len(x_chain) < length:
        print(x, end='')
        x_chain += x
        if x==stop:
            break
        x = random.choice(markov[x]) # x をキーにして辞書を引き次の x を選ぶ
    return x_chain
```

先に述べたように辞書から得られた候補のリストから、ランダムに選択するためと、もうひとつ、出だしを何にするかを選ぶために、標準ライブラリの random モジュールを使うことにしてインポートする．そして文章生成は generate_text()という関数としてインプリメントする．マルコフ連鎖の辞書は引数で与えるが、生成する長さと、始まりと終わりも引数で指定できるようにしておこう．始まりが引数で指定された場合はそのまま使うが、指定されない場合には辞書のキーの中から、一つを関数 random.choice()でランダムに選ぶ．なお list(markov)で辞書 markov にあるすべてのキーのリストが得られる．そしてその始まりを最初の値 x として while のループに入る．while のループは、生成される文字列 x_chain が引数で指定される長さ length になるまで、あるいは、値 x が引数 stop に一致するまで繰り返される．

while のループの中では、はじめに値 x をプリントし、これを x_chain に加える．print 文の end=''はプリントの際に改行をさせないために必要だ．終わりのチェックの後、その値 x をキーに辞書を引き、得られるリストから、関数 random.choice()でランダムに一つ選んで、次の値 x とし、次のループになる．while のループを抜けると x_chain を返り値として、関数 generate_text()を終わる．

リスト 5.1.4 をリスト 5.1.1 と同じファイル markov1.py に書き加えよう．

それではさっそく、リスト 5.1.2 に文章生成を 1 行加えて実行してみよう.

リスト 5.1.5：いろは歌～その 2

```
from markov1 import *

text = ('いろはにほへとちりぬるを'
        'わかよたれそつねならむ'
        'うゐのおくやまけふこえて'
        'あさきゆめみしゑひもせす')

charlist = list(text)
markov = make_markov(charlist)
generate_text(markov, 47, 'い', 'す')
```

辞書を作る際のチェックは不要だから verbose は指定せずに、関数 make_markov()を実行し、できた辞書 markov を与えて関数 generate_text()を行う. 引数で文字数は 47、始まりは'い'、終わりは'す'を指定すれば、次の結果が得られる.

いろはにほへとちりぬるをわかよたれそつねならむうゐのおくやまけふこえてあさきゆめみしゑひもせす

いろは歌の場合には、各文字は 1 回だけ使われ、次の文字は一つに決まっているから、「い」で始めれば、そのまま元のいろは歌が再現できる.

ではリスト 5.1.3 の例ではどうなるだろうか？ これも verbose の指定を外し、文章生成のため 1 行を書き加えるだけだ.

リスト 5.1.6 簡単な文例～その 2

```
from markov1 import *

text = '私はあなたが好きです。あなたは私が嫌いです。'

charlist = list(text)
markov = make_markov(charlist)
generate_text(markov, 100)
```

次の文字を候補の中からランダムに選ぶから、実行するたびに結果が違うが、今度は元の文になかった文が生成される. 一例をあげると、

が好きです。あなたはあなたは私は私は私はあなたが好きです。あなたは私はあなたが好きです。あなたはあなたが好きです。あなたは私が好きです。あなたが好きです。あなたが好きです。あなたはあなたが嫌いです。あ

この場合、始まりも終わりも指定していないから語の途中の「が」から始まり、「あ」で終わっているが、ともかく、「あなたは私が好きです。」や「あなたはあなたが嫌いです。」といった元の分になかった文が生成されている．と同時に、図 5.1.1 からわかるように、「あなたはあなたは私は私は私はあなたが好きです。」のように、同じところでぐるぐる回って「私は」や「あなたは」を繰り返す文も作られている．この点を含めて課題はあるが、マルコフ連鎖によって文章が生成できることが確かめられた．

5.1.2 小説を扱ってみよう

5.1.1 では「いろは歌」や、ごく簡単な文章を使い、マルコフ連鎖の辞書を作って文章の生成を行った．そこで本格的な文章を扱いたいのだが、「青空文庫」というのがネット上に公開されていて容易に本格的な文章のテキストデータが入手できる．曰く、「青空文庫は、著作権が消滅した作品や著者が許諾した作品のテキストを公開しているインターネット上の電子図書館である。」URL は、 https://www.aozora.gr.jp/ だ．ここから好きな作家の好きな小説をテキスト形式で入手しよう．ここでは一例として、芥川龍之介の「羅生門」を入手する．
青空文庫のメニューで「公開中 作品別」から「ら」を選ぶと、50 件ずつ作品が出てきて、その中から「羅生門 新字新仮名 芥川 竜之介」を選ぶ(なぜか龍の字が竜になっているが...)．すると作品データや作家データなどに続き、ファイルのダウンロードという項目があって、その中のテキストファイルを選べば、ダウンロードされて、zip 形式を展開すれば、rashomon.txt が得られるから、それを適当なところに置けば完了だ．ここでは後で見やすいようにファイル名を 羅生門.txt と変名し、以下の例はそれを前提とした．ただしこれはファイル名を正しく指定すれば良いだけのことだから、そのままでも構わない．また、置き場所は、プログラムと同じフォルダか同列に aozorabunnko というのとフォルダを作って、そこに置けば、後から別の小説を加えたりするのに都合が良いし、ファイルを取り出すのに苦労しないが、これも各自好きなようにすれば良いと思う．とまれこれ以降は、aozorabunnko というフォルダをプログラムのフォルダと同列（共通の親のディレクトリの下）に置いた場合のプログラムを提示していくことにするが、そうでない場合は各自ファイルの操作の際に対応されたし．

では小説が用意できたので、これを使ってマルコフ連鎖の動作を見ていこう．リスト 5.1.7 にこのプログラムを示す．

リスト 5.1.7 羅生門でマルコフ連鎖

```
from markov1 import *

nobel = '羅生門'

path = '../aozorabunnko/'
file_name = nobel + '.txt'
with open(path + file_name, mode='r', encoding='utf-8') as f:
    text = f.read()

charlist = list(text)
markov = make_markov(charlist)
generate_text(markov, 100, start='あ')
```

ここではこのプログラムと同列の aozorabunnko というフォルダに 羅生門.txt を置く前提で、小説名とそのパスを指定している．path = '../aozorabunnko/' で '../' の部分は一つ上の階層を示すから、'../aozorabunnko/' は一つ上の階層の下の aozorabunnko ということ、つまり同列の aozorabunnko になる．またファイルを開く際に、encoding='utf-8' としているが、これは文字コードの指定で、これを指定しないとエラーする．とまれ text に内容が読込めたら、後はリスト 5.1.6 などと同じだ．「あ」から始めて見ると、一例として以下のような結果が得られる．

あす火事を増しら奪っといりすとい。羅生まいなわとの髪を憎む。洛中へ蹴倒しましたたっきった喉仏《な得意志に膿《くもよったこか云えじり見えても大き所へつけて—いろう方《ひわだとる火のほかざあ》だい出そう》

この場合に、「あす火事を」まではよかったのだが、その先は語としては「洛中」とか「蹴倒し」とか「喉仏」とか意味を成すものの、文章というには、あまりにもでたらめだ．

5.1.3 文字ではなく語を単位にしてみる

文章は文字の連なりには違いがないが、文字の連なりだけでマルコフ連鎖によって文章を生成しようとすると、うまくいかない．しょせん自由度の問題には違いないと思われるが、もう少し意味を成す単位を捉えて文章を生成すれば良いのではないかと考えられる．そこで、文章を文字ではなく、語の連なりと捉えたらどうだろうか？ しかしそもそも日本語は、英語などとは異なり、句読点で句切られる場合以外、語の境が簡単にはわからない．しかしこれを担っ

てくれる便利なツールがあるから、これを使ってやってみることにしよう．なかでも、とてもポピュラーなツールとして、「Janome」と「Mecab」があり、これらはいずれも本来は形態素解析、つまり、いわゆる自然言語、普通に人が書くような文章を、意味を持つ最小の単位(形態素)に細分化して、一つひとつの品詞・変化などを判別していくツールだ．この最小の単位をいささか厳密ではないが「語」ということにさせてもらい、細分化、つまり分割の機能を借用することにする．ここでは Janome をインストールして使うことにする．
インストールは windows 環境なら Windows PowerShell かコマンドプロンプトで、クロームブックなら terminal で、pip install janome として簡単にできる．以下は Janome が動作することを前提にする．そしてそれを使って文章を語に分割するプログラムをリスト 5.1.8 に示す．

リスト 5.1.8 Janome で語に分割

```
def split_by_janome(text):
    """ janome の形態素分析で文章を語に分割 """
    from janome.tokenizer import Tokenizer
    word_list = Tokenizer().tokenize(text, wakati=True)
    return list(word_list)
```

なお、Tokenizer().tokenize()からはジェネレータが渡されるので、リストに型変換して返り値としている．リスト 5.1.8 は後から様々な用途に使えそうだから、common_function.py に置いておこう．ではこれを使って、羅生門を語分割してみよう．

リスト 5.1.9 羅生門を語に分割

```
import common_function as cf

nobel = '羅生門'

path = '../aozorabunnko/'
file_name = nobel + '.txt'
with open(path + file_name, mode='r', encoding='utf-8') as f:
    text = f.read()

wordlist = cf.split_by_janome(text)
print(wordlist[:100])
```

リスト 5.1.9.では、common_function にリスト 5.1.8 の関数 split_by_janome()を置いたから、これをインポートする．aozorabunnko に置いた羅生門を読込んで、Janome で分割して最初の 100 語をプリントするだけだ．実行してちゃんとご分割されているのを確認されたし．次のようになっているはずだ．

['ある', '日', 'の', '暮方', 'の', '事', 'で', 'ある', '。', 'ー', '人', 'の', '下人', '《', 'げに', 'ん', '》', 'が', '、', '羅生門', '《', 'ら', 'しょ', 'う', 'もん', '》', 'の', '下', 'で', '雨', 'やみ', 'を', '待っ', 'て', 'い', 'た', '。', '¥n', '¥u3000', '広い', '門', 'の', '下', 'に', 'は', '、', 'この', '男', 'の', 'ほか', 'に', '誰', 'も', 'い', 'ない', '。', 'ただ', '、', '所々', '|', '丹塗', '《', 'に', 'ぬり', '》', 'の', '剥', '《', 'は', '》', 'げた', '、', '大きな', '円柱', '《', 'まる', 'ば', 'しら', '》', 'に', '、', '蟋蟀', '《', 'きりぎりす', '》', 'が', 'ー', '匹', 'とまっ', 'て', 'いる', '。', '羅生門', 'が', '、', '朱雀', '大路', '《', 'す', 'ざく']

さて語分割されることが確認出来たら、これをマルコフ連鎖で扱ってみよう．これをリスト5.1.10に示す．

リスト5.1.10 羅生門を語分割でマルコフ連鎖

```
from markov1 import *
import common_function as cf

nobel = '羅生門'

path = '../aozorabunnko/'
file_name = nobel + '.txt'
with open(path + file_name, mode='r', encoding='utf-8') as f:
    text = f.read()

wordlist = cf.split_by_janome(text)
markov = make_markov(wordlist)
generate_text(markov, 100, start='ある')
```

リスト5.1.9で語分割したのはじめの結果が'ある'なので、generate_text()の引数はstart='ある'とした．これを実行した結果の一例を示すと、

ある勇気でも「己に成就しび》たる夜をつかみながら映った臭気にかかえて来て、やっと急な梯子《おお》下《ひはぎ》だわ。
　作者はない」、崩れ目にとまって死なな梯子をともしてへん＋丑」ような侮蔑《こくとうとうしまいには死骸《くく》ると知れぬすざくお》む。丹塗《くも》が円満に蜘蛛《ざんじ》は次第に空を高く

おかしなところは多いが、リスト5.1.7の結果に比べて明らかに品質が向上し、少し読めるようになってきた．

5.1.4 N階マルコフ連鎖と文章の生成

文字ではなく語を最小の単位として、マルコフ連鎖により文章を生成することで、少しは読める文章が生成されることがわかったが、やはりそれでも意味不明な語の連なりであることは免れない. 過去に綴ってきたことをすべて無視して、たった一つの語から次の語を繰り出して、意味ある文章を作るのは、もとより無理があるだろう.

そこでマルコフ連鎖に再び目を向けると、これまで行ってきたように、「次の状態が現在の状態のみで決まる」というのではなく、「次の状態が現在を含めた過去N個の状態履歴に依存して決まる確率過程」もまた、マルコフ連鎖の範疇に含まれ、これを「N階マルコフ連鎖」とか「N次マルコフ連鎖」というようだ.

文章において、いくつかの文字ないしは語の連なりから、次の文字ないし語が確率的に決まるとすれば、より自然な文章を綴ることができるのではないだろうか？ そこでマルコフ連鎖の辞書を作るリスト5.1.1の関数、それから、この辞書を使って文章を生成するリスト5.1.4の関数を、複数の要素の並びを辞書のキーとするように拡張することにしよう. まずは辞書を作る方を、リスト5.1.11に示す.

リスト5.1.11：N階マルコフ連鎖の辞書

```
def make_markov(x_chain, n=2, verbose=False):
    """ マルコフ連鎖のテーブル作成(キーはn個の連続した値)
    P(Xt+1 | Xt=At, Xt-1=At-1, ── )
      x       key2   key1        ──
    """
    markov = {}
    key = []    # キーはリストで操作して辞書アクセスの際にタプルにする
    for x in x_chain:
        # ― 1つ前のxでキーの準備ができていたら実行 ―
        if len(key) == n:
            k = tuple(key)
            # 該キーに対応するものがなければ空のリストで用意
            if k not in markov:
                markov[k] = []
                if verbose:
                    print('create key', k)
            # 該キーのリスト中に対象がなかったら加える
            if x not in markov[k]:
                markov[k].append(x)
                if verbose:
                    print('append key', k, ':', x)
        # ― 次のxに向けてキーを整える ―
        key.append(x)
        if len(key) > n:
            key = key[1:] # 左端を捨てて左シフト
    return markov
```

リスト 5.1.11 はリスト 5.1.1 の素直な拡張だ．まず key の初期値は空白の文字のかわりに空のリストにする．キーが用意できたかどうかは if key のかわりに、if len(key)==n で所定の要素数が揃ったかどうかを見る．この所定の長さはデフォルト引数で与える．複数要素の並びを辞書の一つのキーとして扱うために、タプルに型変換する必要があるが、そのあと辞書を作る過程はリスト 5.1.1 と全く同じだ．そして最後に次に向けてキーを用意するところは、キーが要素の並びなので、キーの右端に現在の要素を加え、左端の要素を捨てる．つまり一つ左にシフトする．繰り返しになるが、複数要素の並んだリストを、そのままではキーにできないからタプルにしていることに注意されたし．
リスト 5.1.11 が出来たら、これを Markov.py として保存して、リスト 5.1.3 の markov1 を Markov に書き換えて実行して、意図したとおりに辞書ができることを確認しておこう．

さて次は辞書から文章を生成する方だ．これをリスト 5.1.12 に示す．リスト 5.1.11 の関数 make_markov()と併せて Markov.py に保存しよう．

リスト 5.1.12：N 階マルコフ連鎖で文章生成

```
import random

def generate_text(markov, length=100,
                  start=None, stop=None, print_text=True, end=''):
    """ マルコフ連鎖のテーブルを使って自動生成 """
    x_chain = ""
    key = []
    # ― start が無指定ならランダムに選ぶ ―
    if start is None:
        start = random.choice(list(markov))
    n = len(start) # キーの長さ

    for j in range(length):
        # ― 書出しはキーそのもの、その後はキーの指すものからランダムに選ぶ ―
        if j < n:
            x = start[j]
        else:
            x = random.choice(markov[tuple(key)])
        # ― 綴る ―
        x_chain += x
        if print_text:
            print(x, end=end)
        if x==stop or len(x_chain)>length:
            break
        # ― 次に備える ―
        key.append(x)
        key = key[-n:]

    return x_chain
```

リスト 5.1.12 で、関数名はリスト 5.1.4 と同じ generate_text()とするが、N 階マルコフ連鎖に対応する．加えて生成する文章を関数の中でプリント出力するかどうかを引数で指定できるようにし、併せてプリント出力の際に文字あるいは語の間を詰めるかどうかを引数 end で指定できるようにした．

関数の中でははじめに x_chain と key をそれぞれ空の文字列と空のリストで初期化する．それから start で出だしが指定されなかった場合にはランダムにマルコフ連鎖の辞書のキーから選んで start とする．そして start が用意できると、それがはじめの key なのだが、ここではその長さのみを n として求めておいて、後の for ループの中で再構成する．

以上で前処理は終わり、for ループで文章を綴っていく．まず if～else 節で x を一つ選ぶ．出だしは start から順に選び、その後はマルコフ連鎖の辞書を key で索引しながら選ぶ．辞書を引いてその候補からランダムに選ぶのはリスト 5.1.4 と同じだが、key は x そのものではなく、その並んだリストであり、リストのままでは辞書が引けないからタプルに型変換する．それならば key をはじめからタプルにしておけば良さそうだが、key はループのたびに加工が必要なため、リストでなければならない．

とまれ x が得られると x_chain に加えるとともに、引数 print_text が真ならば画面出力する．また x が stop に一致するか、x_chain の長さが length に達したら break でループを抜ける．x_chain の長さをチェックするのは、語で綴る文字列が length で指定した長さよりも長くなる場合があるからだ．

key はループの最後に作るが、先に述べたように、ここで start の範囲も含めて加工する．key.append(x)で x を key に組み込み、key = key[-n:]で、その key の右から n 個を切り出して、つまり start と同じ長さにして、新たな key とする．ループに入る前に key=[]としているから、はじめは for ループで繰り返すたびに、key に start から抜き出した x が加えられて一つずつ伸びていく．そして長さ n に至り、その後はマルコフ連鎖の辞書を引いて得られた新しい x が右に加えられて左端の一番古い x が捨てられていく．key は次のループの else 節で使われて、また新たな x を得ることを繰り返して文章が綴られていく．

これで Markov.py に N 階マルコフ連鎖の辞書を作る関数 make_markov()と、それを使って文章を生成する関数 generate_text()が用意できた．これを使ってさっそく芥川龍之介の羅生門を元に文章生成をやってみよう．

リスト 5.1.13：羅生門で N 階マルコフ連鎖

```
from Markov import *
import common_function as cf

nobel = '羅生門'
start = '一人の下人'

path = '../aozorabunnko/'
file_name = nobel + '.txt'
with open(path + file_name, mode='r', encoding='utf-8') as f:
    text = f.read()

wordlist = cf.split_by_janome(text)
start = cf.split_by_janome(start)
markov = make_markov(wordlist, len(start))
generate_text(markov, 100, start=start)
```

リスト 5.1.13 はリスト 5.1.10 とほとんど同じだから、違いだけに着目する．冒頭のインポートはもちろん markov1 を Markov に書き換える．そしてせっかく N 階マルコフ連鎖だから、文章の出だしは平文で与えるようにした．ここでは start = '一人の下人' だ．これは後から Janome で語分割して、その語数 len(start)を関数 make_markov()の引数とし、キーの要素数を指定する．そして関数 generate_text()では、start=start として、生成する文章の出だしとする．以下は、この実行結果の一例だ．

一人の下人《げにん》が、羅生門《らしょうもん》の下で雨やみを待っていた。旧記の記者の語を借りれば、「頭身《とうしん》の毛も太る」ように感じたのである。それから、皺で、ほとんど、鼻と一つになった唇を、何

どうだろうか？ 今度はしっかりと読める文章になっていると思う．ちなみにこの時、start = ['一', '人', 'の', '下人']となっているから、N 階の N は 4 で、4 階マルコフ連鎖で文章を生成したことになる．作られる辞書に含まれる値の自由度と、キーに含まれる要素の並びによる拘束のせめぎあいには違いないけれども、表現の幅が広いたくさんの文章を読込むと同時に、程よい長さの文字や語の並びをキーとして辞書を作れば、さらに大きな可能性が期待できるのではないだろうか？
ところで少し遡ると、文字で分割したリスト 5.1.7 ではほとんど文章になっていなかった．そ

こで Janome を導入して元の文章を語で分割し、マルコフ連鎖を適用して文章生成するようにしたのだった. そしてリスト 5.1.10 の結果はリスト 5.1.7 よりも良かった. リスト 5.1.13 は、この良かった方の語分割なのだが、ここで敢えて文字分割に戻してみたらどうなるだろうか？

リスト 5.1.14：羅生門で文字分割の N 階マルコフ連鎖

```
from Markov import *

nobel = '羅生門'
start = '一人の下人'

path = '../aozorabunnko/'
file_name = nobel + '.txt'
with open(path + file_name, mode='r', encoding='utf-8') as f:
    text = f.read()

charlist = list(text)
start = list(start)
markov = make_markov(charlist, len(start))
generate_text(markov, 100, start=start)
```

リスト 5.1.14 はリスト 5.1.13 とほとんど同じだが、分割の際に list(text)とか list(start)として、文字で分割している. さっそくこの結果の一例を示すと、

一人の下人《げにん》は、それらの死骸の腐爛《ふらん》した臭気に思わず、鼻を掩《おお》った。しかし勝敗は、はじめからわかっている。両手をわなわなふるわせて、肩で息を切りながら、さっきから朱雀大路にふる雨

このように語分割のリスト 5.1.13 の結果とほとんど品質の違いが分からないくらいの文章が生成された.　考えてみれば、語は文字の一つ以上の並びで構成されるから N 階マルコフ連鎖で、その文字の並びも含めて扱うことは、理にかなっているとも思われる. とくに日本語の場合には、語が比較的少数の文字で構成されるから、うまくいきやすいのではないだろうか？

5.2 ニューラルネットワークでマルコフ連鎖

前節で、マルコフ連鎖によって文章を生成するという試みは思いの外うまくいってしまった. この方法では、有限個の文字あるいは語の並びをキーとして、次の文字あるいは語の候補を得るマルコフ連鎖の辞書を、小説などの文章に含まれる文字あるいは語の並びから作った. そして一つの文字あるいは語から次の文字あるいは語の候補を得るのに対し、文字あるいは語の並びから次の文字あるいは語の候補を得るマルコフ連鎖の方法は、並びの数を N として N 階マルコフ連鎖であり、翻って一つの文字あるいは語のそれは単純マルコフ連鎖ということになる. リスト 5.1.13 やリスト 5.1.14 では N=4 とか N=5 であり、この数が大きくなればそれだけ、文章がうまく再現されるようになる一方、次の候補の数は少なくなり、文章が硬直化していくと考えられる. つまり元の文章に含まれる文章が再現されるだけになるだろうから、面白くない.

それでは何を求めているのかといえば、文章として読めるものでありながら、元の文章には含まれないものを生成したい、ということなのだ. ここまでのところでも文字あるいは語の並びを候補から選ぶ際にランダムに選ぶために、元の文章になかった並びとなるから、もう出来ているではないか、ということもできる. いっぽう逆に、文脈も何もなくランダムに選ぶだけだから、作られる文章に意味などあろうはずもなく、意味ある文章を作る機械など出来る筈はないと言われればその通りかもしれない.

しかし第 4 章の最後に敵対的生成ネットワーク GAN で生成した顔写真が、元の写真にある顔写真とは違っていて、それでも人の顔になっていたくらいのことならば、出来るのではないだろうか？そしてそこに何らかの意図をもって、作られるものを「いじる」ことができるならば、たとえば作家が従来のスタイルを踏襲しつつも、何らかの意図をもって新たなものを作るように、それは創造性以外のなにものでもないのではないだろうか？

そういう道筋を描いていくことを念頭に、ここでは単純な辞書からニューラルネットワークに置き換えていくことにしよう.

5.2.1 文字あるいは語の数値による表現

辞書の場合には、文字や語あるいはその並びがそのままキーとして使え、値としても文字や語がそのまま出力できた. しかしニューラルネットワークで同様のことを行おうとすると、それはできない. なぜならばニューラルネットワークは、入力の重み付け和をとり、それを活性化関数によって変換したものが出力だからだ. つまり数値計算される数値にしなければ扱えない. そこでまずは、文字あるいは語を id に変換し、文章を id の並びで表すプログラムを作ろう.

リスト 5.2.1：文章の文字あるいは語を id に変換

```
def preprocess(x_list, corpus=[], x_to_id={}, id_to_x={}):
    """ コーパスと変換辞書作成 """
    for x in x_list:
        if x not in x_to_id:
            new_id = len(x_to_id)
            x_to_id[x] = new_id
            id_to_x[new_id] = x
        corpus.append(x_to_id[x])
    return corpus, x_to_id, id_to_x
```

リスト 5.2.1 にそのプログラムを示すが、文章を生成する操作の前処理なので関数名を preprocess()とした．そして文字でも語でも使えるように引数は x_list とした．そしてこれ以外の引数は、デフォルト引数で corpus=[]、x_to_id={}、id_to_x={}とした．corpus には文章を id の並びに変換したものを入れる．x_to_id と id_to_x は、文字あるいは語から id への変換辞書、そしてその逆の辞書を入れる．ここでこれらの辞書は引数で予め何かを与えることもできるようにしているのは、すでに作られている変換辞書に、文章を追加で読込んでエントリーを付け加えることもできるようにするためだ．同様に corpus も予めリストを渡すことができるようにしている．とまれ通常はこれらはいずれも空からはじめ、for ループで x_list から一つずつ x を取り出して、すなわち文章から 1 文字あるいは 1 語ずつ取り出して、それが辞書に登録済みかどうかチェックし、なければ新規に id を割り当てて new_id とする．そして辞書 x_to_id にキーx、値 new_id を x_to_id[x] = new_id により登録する．いっぽう辞書 id_to_x には、値とキーを入れ替えてキーnew_id、値 x を id_to_x[new_id] = x により登録する．このとき new_id = len(x_to_id)としているが、これは id を一つずつ繰り上げながら割り当てていくためだ．

そして if 文の成否によらず、たった今あらたに辞書に登録された場合も含めて、corpus には辞書 x_to_id で id に変換した x の値が付け加えられる．

返り値は corpus と、x_to_id と、id_to_x だ．この関数は common_function.py に置いておこう．なおここでは、文章などを id の並びに変換したあとのものを corpus としている．一般には corpus とは自然言語の文章を大規模に集積したものそのものを指すようだ．また、文章に含まれる文字あるいは語のすべてを網羅したもの、すなわち総体を語彙ということにして、その種類の数、すなわち id の数あるいはカテゴリーの数を、それをここでは語彙数と呼ぶことにしよう．語ではなく文字の場合には言葉として適切ではないかもしれないが、便宜上そうしておこう．ともかくそうすると、語彙数は辞書 x_to_id あるいは id_to_x の要素数であり、len(x_to_id)や len(id_to_x)で求められる．そして、これらの辞書を作る際に id を 0 から順に繰り上げていくから、この値は同時に辞書に含まれる id の最大値となる．

5.2.2 ニューラルネットワークによる文章の生成にあたり

ニューラルネットワークで文章を生成するにあたり、入力はおいておいて、まず出力について考えてみよう．もちろん出力から次の文字あるいは語が知りたい．これは第 4 章で分類や画像認識の問題をニューラルネットワークで扱ったのと、少なくとも出力については、あい重なるのではないだろうか？ 分類や画像識別を行う場合には、ニューラルネットワークの出力層にすべてのカテゴリーに対応するニューロンを並べ、その出力の一つ一つの値の大小が、そのカテゴリーに属する可能性をあらわすようにした．これと同様にして、どの文字あるいは語かを分類する分類問題ととらえることができるのではないだろうか？ つまり出力層のニューロンを文字または語のそれぞれに対応するだけ、すなわち語彙数だけ並べ、その値の大小で、その文字あるいは語である可能性をあらわすのだ．そしてその出力は、one_hot 形式にした正解値と照合することによって、エラーを算出したり、勾配を求めて学習させたりすれば良い．

いっぽう入力はどうだろうか？ 画像識別では画像の画素のデータを、それ以外の分類問題でも何らかのデータを入力した．これに対しここでは元になるのは、文字、語、あるいはその並びを id に変換したものだ．しかしそのままではニューラルネットワークの入力に適さない．なぜなら、その id の値の大小には意味がない、あってはならないからだ．この事情は分類ないし識別問題の出力と同様だ．つまり文字あるいは語を示す id を one_hot に変換したものを入力とすれば良いだろう．そしてそれは語彙数の長さのベクトルになる．

これで基本的なアプローチは決まった．しかしもう少し考えることにして、マルコフ連鎖の辞書を作って文章を生成したことを振り返ってみよう．リスト 5.1.7 やリスト 5.1.10 の結果に見るように、単純マルコフ連鎖では意味の通じる文章を生成するのは難しかった．いっぽうリスト 5.1.13 やリスト 5.1.14 では丸暗記のきらいはあるものの、じゅうぶんに読める文章が生成されており、N 階マルコフ連鎖で複数個の文字あるいは語の並びから次を決めることで、文章の生成がうまくいくことをすでに見てきた．だから、マルコフ連鎖をニューラルネットワークで担うにあたって、はじめから N 階マルコフ連鎖に対応できるようにプログラムを作るべきだろう．

ではその場合にニューラルネットワークへの入力をどうすれば良いだろうか？ そして複数の文字あるいは語の並びから、ニューラルネットワークに入力できる形に変換するのに一番簡単なやり方は、複数の id の並びの各 id をそれぞれ one_hot に変換し、そのそれぞれが語彙数の長さのベクトルを一つに結合することだろう．つまりマルコフ連鎖の階数を N として、N 個の id の並びを語彙数×N の長さのベクトルに変換することだろう．ここで結合されるベクト

ルの並びは、その元となる文字あるいは語の並びとして、元の文章に無くても良いことは意味がある．いうまでもなく、辞書では登録されているものしか索引できない．しかしニューラルネットワークでは入力の幅さえ合っていれば、入力して出力を得ることができる．だから全体が語彙数×Nの長さのベクトルでありさえすれば良いのであって、結合する個々のベクトルがどう並んでいてもニューラルネットワークなら大丈夫なのだ．つまり元の文章に無い文字や語の並びをニューラルネットワークならば扱えるということである．新たな文章を生み出すということは、元の文章には無い文字や語の並びを作り出すことに他ならないから、これは重要なことなのだ．

5.2.3 ニューラルネットワークによる文章の生成

ようやくやり方が決まったが、マルコフ連鎖の辞書を作るリスト 5.1.1 やリスト 5.1.11 の関数 make_markov()に相当する操作は、ニューラルネットワークを学習させることに置き換えられるから、ここではちょっとおいておいて、先に文章の生成について考えていこう．

元になるのはN階マルコフ連鎖で文章を生成するリスト 5.1.12 だ．リスト 5.1.12 のマルコフ連鎖の辞書を使うやり方と大きく違うのは、いうまでもなくニューラルネットワークにしかるべき入力を用意して、順伝播を行い、その出力から結果を出す過程に他ならない．リスト 5.1.12 のマルコフ連鎖の辞書では、文字、語あるいはそれらの並びを直接キーにして辞書を引き、得た候補のリストからランダムに一つを選べばよかった．だが先に述べたようにニューラルネットワークでは、文字、語を、これを識別する id の並びとし、さらにこれを語彙数のベクトルに変換し、さらにこれを結合したものを入力として用意する必要がある．また出力も、ニューラルネットワークでは、すべての候補に数値を与えたものとなり、その数値は大きいほど、それである確率が高いということを示すものとなる．またそうなるように学習させるべきだ．そしてそこから何らかの方法で確率的に一つを選ばなければならない．

長々と述べたがこれを踏まえ、ニューラルネットワークで文章を生成するプログラム、そしてその中で確率的に選ぶ部分を、それぞれ関数 generate_text()、関数 select_category()として分けて作る．これらはリスト 5.2.2 とリスト 5.2.3 に示す．

リスト 5.2.2：文章の生成

```
def generate_text(func, x_to_id, id_to_x, n=1, length=100,
                  seed=None, stop=None, print_text=True, end='',
                  stocastic=True, beta=2):
    """ 文章生成：識別子 xid が n 個並んだ key を func に入力して文章を生成 """
    x_chain = ""
    key = []
    vocab_size = len(x_to_id)
    # ― seed 無指定なら辞書からランダムに選ぶ ―
    if seed is None:
        seed = random.choices(list(x_to_id), k=n)

    for j in range(length):
        # ― 書出しは seed から、その後は func 出力から ―
        if j < len(seed):      # seed の範囲
            x = seed[j]
            xid = x_to_id[x]
        else:                  # seed の範囲を超えた
            y = func(key)
            xid = select_category(y, stocastic, beta)
            x = id_to_x[xid]
        # ― 綴る ―
        x_chain += x           # x_chain は対象をつないだもの
        if print_text:
            print(x, end=end)
        if x==stop or len(x_chain)>length:
            break
        # ― 次に備える ―
        key.append(xid)        # キーは xid を並べたもの
        key = key[-n:]         # 左シフト

    return x_chain
```

リスト 5.1.12 ではマルコフ連鎖の辞書 markov を関数 generate_text()の引数として与えたが、リスト 5.2.2 では func、x_to_id、id_to_x、n=1 の 4 つの引数がそれに置き換わる．func にはニューラルネットワークの順伝播を与えることを想定するが、このとき入力を one_hot にするなど必要なことは func 内で完結するものとする．いっぽう x_to_id と id_to_x は、リスト 5.2.1 の関数 preprocess()で得られる文字あるいは語とそれに対応する識別子 id の相互の変換のための辞書であり、この関数内での変換に使う．また n=1 はマルコフ連鎖の階数で、マルコフ連鎖の辞書のキーに相当するニューラルネットワークの入力の id の並びの数を指定する．引数 seed=None、stop=None、print_text、end=''については基本的にリスト 5.1.12 と同じだが、出だしを与える引数は、自由度のある複数の並びなので start ではなく seed と名前を変えた．そして追加される stocastic=True と beta=2 の 2 つの引数は、ニューラルネットワークの出力のどれを選ぶかを決める際のパラメータでリスト 5.2.3 の関数 select_category()に渡す．
リスト 5.1.12 と同じところも多いので、x_chain など同じ働きの変数は同じにした．

はじめに x_chain=""、key=[] と初期化し、vocab_size = len(x_to_id)で語彙数を求める．少々厄介なのはニューラルネットワークの入力を key とすれば、それと文字あるいは語の識別子 id が必ずしも同じではなく、別扱いしなければならないことだ．
続けて引数で seed が指定されなかった場合には、辞書 x_to_id のキーから n 個をランダムに抽出して seed にする．リスト 5.1.12 でマルコフ連鎖の辞書 markov のキーなのに対し、ここではそれとは違って文字や語を識別子に変換する辞書 x_to_id のキーを使っている．
seed の手当が済むと、for ループに入り文章を綴っていく．
for ループの中で、出だしの seed の部分も含めて処理するのはリスト 5.1.12 と同じだが、seed の長さは key のそれとは違うこともある．とまれ出だしは、seed から 1 つずつ取り出して、それを x とし、辞書 x_to_id で変換した id を xid とする．この変換はリスト 5.1.12 にはなかった．
seed を使い終わると else 節が行われる．else 節ではニューラルネットワークに key を入力して y を得る．そして得られた y を後で説明する関数 select_category()で変換して xid を得る．そしてそれを今度は逆に辞書 id_to_x で変換して x を得る．key から次の x を得るということはリスト 5.1.12 と同じだが、やり方が違うのだ．
とまれこれら if～else 節のところで、文字あるいは語の x と、これに対応する識別子 xid を揃え、そのあと文章を綴っていくが、そこはリスト 5.1.12 と基本的に同じだ．そして key の加工を行うが、ここも x ではなくその識別子 xid ということ以外は同じで、ループの最後で作った key が次のループの else 節で使われる．
seed に十分な長さがあれば、if 節の seed の範囲の処理の間に key は長さ n に至る．だから、seed は n より短いのは困るが n より長くても問題ない．また語で綴る場合の seed は語の並ぶリストにする必要があるものの、文字で綴る場合は seed が一続きの文字列であっても、文字を区切って並べたリストであっても構わない．なぜなら seed[j]で 1 つが抜き出せれば良いだけだからだ．このプログラムはファイル markov_n.py に入れておこう．

それでは次に関数 select_category()を用意しよう．これをリスト 5.2.3 に示す．ニューラルネットワークの入出力のように、数値計算するのに都合が良いように変換したものをダミー変数、これに対し元の属性の id をカテゴリ変数という呼び方をすることがあるから、ここではそれに従うことにすると、ここでやりたいことはダミー変数からカテゴリ変数への変換ということになる．ちなみにニューラルネットワークに入力する際に one hot の形式にしたが、これはカテゴリ変数からダミー変数への変換ということになる．とまれリスト 5.2.3 はリスト 5.2.2 の generate_text()から直接呼べるように markov_n.py におくことにしよう．

リスト 5.2.3：ダミー変数をカテゴリ変数に変換

```
def select_category(x, stocastic=False, beta=2):
    """ ダミー変数をカテゴリ変数に変換 """
    x = x.reshape(-1)
    if stocastic: # 確率的に
        p = np.empty_like(x, dtype='f4') # 極小値多数でエラーしないため精度が必要
        p[...] = x ** beta
        p = p / np.sum(p)
        y = np.random.choice(len(p), size=1, p=p)
    else:         # 確定的に
        y = np.argmax(x)
    return int(y)
```

とまれリスト 5.2.3 の関数 select_category()でやりたいことは、ニューラルネットワークの出力を、それが示すカテゴリの id 番号に変換することだ．カテゴリ分類を行うニューラルネットワークの出力は、その各信号で、それに対応するカテゴリである可能性を数値の大小で示す．もちろんそれに応じてカテゴリ、すなわち id を決めるのだが、その決め方に 2 通りある．一つは単純に値の最も大きな信号がどれであるかによって、その位置とする方法であり、もう一つはその決める際に値の大小に応じた確率に従ってランダムにどれかを選び、その位置とする方法だ．

では関数 select_category()を見ていこう．引数 x はこの関数への入力だ．ニューラルネットワークの出力を想定するが、バッチ処理や時系列処理によらず、データは一つだけとする．そしてこれは関数内ですぐにベクトルに成形する．

いっぽう引数 stocastic は、先に述べた 2 通りの方法のいずれかを指定し、False ならば前者、True ならば後者の方法とする．もう一つの引数 beta は後者、つまり確率に従ってランダムに選ぶ場合の係数となる．

まず if が成立、つまりこの後者、確率的にランダムに選ぶ方法では、その確率を p = np.empty_like(x, dtype='f4') として精度のみ与え、p[...] = x ** beta で、その精度を保ちながら値を設定している．ここでニューラルネットワークの出力 y の beta 乗とすることで、その出力をどれだけシビアに見るかを調整する．また、わざわざ精度を予め設定しなくても多くの場合は大丈夫なのだが、ニューラルネットワークで 2 バイトの半精度の計算を行った場合などには、必要になるから入れておいた．さらに p = p / np.sum(p) で p の値が合計すると 1 になるようにしている．ともかくも確率をコントロールする p が決まると、numpy の random を用いて、y = np.random.choice(len(p), size=1, p=p) により確率的に結果が得られる．

後先になったが else 以下、つまり前者の方法の場合には、y = np.argmax(x) により、最も値が大きなものを指す id を得る．返り値は id なので int(y)により型を確定している．

5.2.4 ニューラルネットワークでマルコフ連鎖

それではあとさきになったが、ニューラルネットワークでマルコフ連鎖を担って文章を生成するプログラムの本体を作ろう.基本的な流れは第4章で画像認識を行ったのと同じだから、リスト4.7.6やリスト4.7.7をもとにすれば良いだろう. まずは最も簡単な現在だけに依存する単純マルコフ連鎖だ. 少し長いので2つに分けて説明する. NN.pyは第4章で作ったもの、またcommon_function.pyには第4章で作ったものと、第5章でここまでに付け加えた機能が含まれる前提だ.

リスト5.2.4：ニューラルネットワークでマルコフ連鎖(前半)

```
import numpy as np
import NN
import common_function as cf
import markov_n
import time

nobel = '羅生門'

# — テキストデータの取得 —
path = '../aozorabunnko/'
file_name = nobel + '.txt'
with open(path + file_name, mode='r', encoding='utf-8') as f:
    text = f.read()

wordlist = cf.split_by_janome(text)
seed = cf.split_by_janome('人間')

# corpusと辞書を作る
corpus, word_to_id, id_to_word = cf.preprocess(wordlist)
vocab_size = len(word_to_id)

# 1つずつずらして切出し、前を入力とし、後を正解値にする
X = corpus[:-1]
C = corpus[1: ]
print(X[:10])
print(C[:10])
X = np.eye(vocab_size)[X]
C = np.eye(vocab_size)[C]
print(X.shape, C.shape)
```

リスト5.2.4に示す前半部分は、まずは必要なモジュールをインポートし、続いて「羅生門」のwordlistを作る. この手順はリスト5.1.9などと同じだ. wordlistを作るついでに、学習後に生成する文章の出だしの部分も関数 split_by_janome()で語の並びのリストにしておく. wordlistを得たらリスト5.2.1の関数preprocess()で、idの並びであるcorpusに変換し、併せて変換辞書word_to_idとid_to_wordを得る. それからvocab_size = len(word_to_id) により語彙数を得る.

続いてニューラルネットワークの入力と正解値の準備だ．入力 X と正解値 C は、corpus の先頭から開始位置を一つずらして切出し、それぞれ numpy の eye()を使って one_hot に変換する．変換では語が語彙数の長さのベクトルになる．以上で前半は終わって後半だ．

リスト 5.2.5：ニューラルネットワークでマルコフ連鎖(後半)

```
# モデルを作る
model = NN.NN_m(vocab_size, ml_nn=100,
                ml_act='Sigmoid',
                ol_act='Softmax',
                loss='CrossEntropyError')
model.summary()

# 学習
epoch = 3001
batch_size = 100
start_time = time.time()
index_random = np.arange(len(X))
mesurement = cf.Mesurement(model)
for i in range(epoch):
    np.random.shuffle(index_random)
    for j in range(0, len(X), batch_size):
        idx = index_random[j:j+batch_size]
        x = X[idx]
        c = C[idx]
        y, l = model.forward(x, c)
        model.backward()
        model.update(eta=0.5)
    if i==0:
        print(' epoch elapse | error  accuracy')
    if i%100==0:
        l, acc = mesurement(X, C)
        print('{:6d} {:6.3f} | {:6.3f} {:6.3f}'￥
              .format(i, time.time()-start_time, float(l), acc))
        start_time = time.time()

# 学習後に経過表示と作文のテスト
errors, accuracy = mesurement.progress()
cf.graph_for_error(errors, accuracy, label=('error','accuracy'))
def func(x):
    x = np.eye(vocab_size)[x]
    y = model.forward(x)
    return y
x_chain = markov_n.generate_text(func, word_to_id, id_to_word,
                                 n=1, length=200, seed=seed)
```

後半はニューラルネットワークのモデルを作るところからはじめる．第 4 章で作ってきたのと同じで、出力数＝語彙数、中間層のニューロン数で基本的な大きさを指定し、中間層と出力層の活性化関数、損失関数を指定すれ良い．そして model_summary()で確認する．

続いてニューラルネットワークの学習だが、これも第 4 章でやってきたことと同様だ. リスト 4.7.6、4.7.7、4.8.5 あたりを参考にすれば良い. エポック数、バッチサイズを指定し、経過時間表示のために start_time に現在時刻を入れ、バッチ処理の際にデータをランダムに選ぶためのインデクスを用意し、common_function.py に作った Mesurement クラスをインスタンス化する. これで準備が整って for ループで、入力と正解のデータをバッチサイズ分だけ切り出してニューラルネットワークの順伝播、逆伝播、更新の手順を繰り返す. 経過表示のため for ループの最初に表示内容のタイトルを出力し、あとは 100 回に 1 回、経過時間とエラーと正解率を出力する.
for ループを抜けるとエラーと正解率をグラフ表示し、学習後のニューラルネットワークに出だしの語を与えて文章を生成する. 文章を生成する際には、markov_n.py に作ったリスト 5.2.2 の関数 generate_text()を使うが、そこで説明したように引数として与える func()の定義で少し細工が必要だ. ニューラルネットワークへの入力はキーに相当する id を one_hot に変換する必要があり、形状も細工が必要で reshape してベクトルにする.
とまれ、出だしの語は前半部分でリストにしてあるが、この出だしは「羅生門」に含まれる語ならば何でも良く、以下には、'人間'を指定して文章生成した一例を示す.

人間ばかりで、その太刀のように、ごろごろ床板の間に、その上であろうおおじ》の下である。そうして、楼のふる雨やみを、下人は、それが、その火のような、それが、何度も、老婆は、一晩楽に、右のである。その梯子の上で、老婆は、下人は、僅に、それを、この門の下である。
 下人は、何分か、饑死《まるば、この老婆は、この門ので、かすかに、それが、下人は、老婆に対する反感が、反対な声である。
 下人は、今、大きな面皰《して、第 3 水準 1-81]《こくげん》の下を、この男の上で、この男の上の楼のは、この門のほかには、勿論、それを、この男のであるばかりで、眼は

リスト 5.1.10 同様それなりの結果で、読める文章とはいいがたい. しかしともかく、動作は確認できた. そこで続いて N 階マルコフ連鎖に拡張しよう. と言っても変更はわずかだ. 前半部分に関しては、リスト 5.2.4 の下 8 行の入力と正解値を corpus から切出す部分をリスト 5.2.6 のように変える.

リスト 5.2.6：ニューラルネットワークで N 階マルコフ連鎖(前半の一部)

```
# 1つずつずらして切出し、つなげて入力とし、後1つを正解値にする
N = len(seed)
X = [corpus[i:i-N] for i in range(N)]
X = np.array(X)
X = X.T
C = corpus[N:]
print(X[:10])
print(C[:10])
X = np.eye(vocab_size)[X].reshape(-1, vocab_size*N)
C = np.eye(vocab_size)[C]
print(X.shape, C.shape)
```

N = len(seed) により、出だしの語の数に応じ、マルコフ連鎖の階数を決める．内包表記で、X = [corpus[i:i-N] for i in range(N)] とすることで、corpus から一つずつずらして N 通り切出したものをリストに並べる．たとえば N = 2 のとき X = [corpus[0:-2], corpus[1:-1]] だ．そしてこれを numpy の行列にしてから転置する．転置することでデータが末尾の軸で隣り合うようになる．いっぽう C は N 番目から始まること以外はリスト 5.2.4 と同様だ．あとはそれぞれ numpy の eye()を使って one_hot に変換する．変換では語が語彙数の長さのベクトルになるから、ニューラルネットワークへの入力データは、これをマルコフ連鎖の階数 N だけつないだものとする．すなわち X は vocab_size*N の長さのベクトルを並べたものにする．

後半部分も修正はわずかだ．修正する末尾部分だけ次に示す．

リスト 5.2.7：ニューラルネットワークでマルコフ連鎖(後半の一部)

```
# 学習後に経過表示と作文のテスト
errors, accuracy = mesurement.progress()
cf.graph_for_error(errors, accuracy, label=('error','accuracy'))
def func(x):
    x = np.eye(vocab_size)[x]
    y = model.forward(x.reshape(-1))
    return y
x_chain = markov_n.generate_text(func, word_to_id, id_to_word,
                                 n=N, length=200, start=seed)
```

変更するのは generate_text()に引数として渡す func()の定義の中で入力 x を複数のキーに対応して結合するために reshape(-1)することと、generate_text()の引数の n=N だけだ．ともかく出だしに指定するのは「羅生門」に含まれる語ならば何を組み合わせても良く、以下には、'人間の光'と指定して文章生成した一例を示す．

人間の光が、かすかに、その男の右の頬をぬらしている。ことに門の上の空が、夕焼けであかくなる時には、ただ、黒洞々《こくとうとう》たる夜があるばかりである。そうして、一足前へ出ると、不意に右の手でおさえながら、冷然として、この話を聞いている中に、赤く膿《うみ》を持って、その死骸は皆、それが胡麻《ごま》をまいたように、飛び上った。

下人《げにん》は、誰も知らない。だから「下人が雨やみをする市女笠《いちめがさ》や揉烏帽子《もみえぼし》が、先方へも通じたのである。

下人は、老婆の答が存外、平凡なのに失望した。しかし、その手は、次の瞬間には、噂に聞いた通り、幾つかの死骸《しがい

リスト 5.1.13 やリスト 5.1.14 の結果を見た後では、あまり驚かないとは思うが、それなりの文章が生成できている．品質はリスト 5.1.13 と比べて良いとは言い難いが、リスト 5.1.13 やリスト 5.1.14 で決して出来ない’人間の光’で始まる文章が生成されている．’人間の光’という語の並びは「羅生門」の元の文章には含まれず、それをキーとする辞書は作られないから、リスト 5.1.13 やリスト 5.1.14 では出来なかったことなのだ．その点を考慮すれば、優位性が理解できるだろう．しかし今のところ相変わらず、語数の制約から逃れられない．そして出だしの長さが変わると、N 階マルコフ連鎖の N を変えることとなって、はじめから学習し直さなければならず厄介だ．だから N 階マルコフ連鎖の階数にとらわれない汎用的な方法が望まれる．

5.3 ニューラルネットワーク全般にかかわるエンハンス

第 4 章ですでに基本的なニューラルネットワークの実装は済んでいるが、ここで少し寄り道をして、ニューラルネットワークをエンハンスすることにしよう．そしてそれは第 4 章でやってきたような画像認識などに効果が期待できるが、同時に後で説明する RNN の基礎ともなり、またその中に組み込んでいく際に使える機能でもある．

5.3.1 ニューラルネットワークの BaseLayer

リスト 4.1.1 で実装した BaseLayer クラスのコントラクタから始めよう．これを最適化関数が使えるようにエンハンスしよう．最適化関数そのものについては後述する．ここではその記述を書き加えるだけだから特に説明はいらないと思う．

リスト 5.3.1：BaseLayer のコントラクタ

```
class BaseLayer:

    def __init__(self, **kwargs):
        print('Initialize', self.__class__.__name__)
        self.width      = kwargs.pop('width',          None)
        activator_name  = kwargs.pop('activate', 'Identity')
        optimizer_name  = kwargs.pop('optimize',      'SGD')

        self.w = None; self.b = None
        self.activator = cf.eval_in_module(activator_name, Activators)
        self.optimizer_w = cf.eval_in_module(optimizer_name, Optimizers)
        self.optimizer_b = cf.eval_in_module(optimizer_name, Optimizers)
```

リスト 5.3.1 をリスト 4.1.1 と比較してもらえばよい．

続いて BaseLayer のパラメータ初期化をエンハンスしよう．

リスト 5.3.2：BaseLayer の init_parameter()メソッド

```
class BaseLayer:

    def init_parameter(self, m, n):
        if self.width is not None:
            width = self.width
        else:
            width = np.sqrt(1/m)  # Xavier の初期値
        self.w = (width * np.random.randn(m, n)).astype(Config.dtype)
        self.b = np.zeros(n, dtype=Config.dtype)
        print(self.__class__.__name__, 'init_parameters', m, n)
```

リスト 5.3.2 はリスト 4.1.7 とほとんど同じだが、Xavier の初期値を適用している．Xavier の初期値はリスト 5.3.2 の width = np.sqrt(1/m)だが、広がり係数を入力の幅に応じて調節するようになっている．これにより、リスト 4.1.7 で設定した 0.01 などの固定値よりも学習がうまく進みやすいメリットがある．
BaseLayer のエンハンスはもう一つ、コントラクタで指定した最適化関数を、重みとバイアスの更新で使うようにする．

リスト 5.3.3：重みとバイアスの更新

```
class BaseLayer:

    def update(self, **kwargs):
        self.w -= self.optimizer_w.update(self.grad_w, **kwargs)
        self.b -= self.optimizer_b.update(self.grad_b, **kwargs)
```

これで BaseLayer のエンハンスは完了だ．

5.3.2 最適化関数

最適化関数を実装しよう．後で説明するが、RNN はその RnnLayer の層一つだけを見ても、層が深くなった構造をしている．そのために学習をうまく進めるための仕組みがなくてはならない．ここでは勾配クリッピングという手法を組み込んだ形で、最適化関数を作っていこう．まずは最適化関数の基本部分を OptimizerBase クラスとして実装する．

リスト 5.3.4：最適化関数の基本部分

```
class OptimizerBase:
    ''' g_clip などの共通機能を各 optimizer に付与する '''
    def __init__(self, **kwargs):
        self.gradient_clipping = GradientClipping()

    def update(self, gradient, **kwargs):
        eta    = kwargs.pop('eta',    0.01)
        g_clip = kwargs.pop('g_clip', None)
        eta = self.gradient_clipping(gradient, eta, g_clip)
        self.eta = eta
        y = self.__call__(gradient, eta)
        return y
```

リスト 5.3.4 に示すように最適化関数の個別の機能は派生クラスに委ねることとして、基本部分では、勾配クリッピングをつなぎこんで自身の実行を行うだけだ．この自身の実行は、

self.__call__()だが、これは派生クラスにおいて定義するから、いずれもリスト 5.3.4 の範囲からはわからない．とまれ勾配クリッピングから示していこう．

リスト 5.3.5 勾配クリッピング

```
class GradientClipping:
    def __call__(self, gradient, eta, g_clip):
        if g_clip is None:
            return eta
        g_l2n = np.sqrt(np.sum(gradient ** 2)) # 勾配の L2 ノルム
        rate = g_clip / (g_l2n + 1e-6)
        eta_hat = eta * rate if rate < 1 else eta
        return  eta_hat
```

勾配クリッピングはリスト 5.3.5 に示すようにクラスの直下に特殊メソッド__call__()で実装する．勾配クリッピングはその名の通り、勾配がある値を超えないようにするのだが、勾配の値自体を抑えるかわりに、学習率 eta を小さくして同じ効果となるようにしている．
先に実装を示したが、勾配クリッピングそのものは「勾配のノルムが閾値以下となるようにクリッピングする」こととして説明される．ここでノルムは L2 ノルムであり、個々の勾配を g とて、これを数式 $\|\mathrm{g}\|$ で表わせば、

$$\|\mathrm{g}\| = \sqrt{\sum \mathrm{g}^2} \tag{5.3.1}$$

式 5.3.1 で明示しないが、右辺は関係するすべての勾配の二乗和の平方根であることに注意されたし．そして閾値を th 、クリッピング後の個々の勾配を $\hat{\mathrm{g}}$ として、

$$\hat{\mathrm{g}} = \begin{cases} \mathrm{g} & (\|\mathrm{g}\| < th) \\ \dfrac{th}{\|\mathrm{g}\|}\mathrm{g} & (\|\mathrm{g}\| \geqq th) \end{cases} \tag{5.3.2}$$

これによって、

$$\|\hat{\mathrm{g}}\| < th \tag{5.3.3}$$

となることを担保する．すなわち勾配のノルムを閾値以下とする．
リスト 5.3.5 のコードで勾配クリッピングがうまく働くことを念のためリスト 5.3.6 で確かめておこう．ここではリスト 5.3.5 を Optimizers.py に置いたものとする．

リスト 5.3.6 勾配クリッピングの動作確認

```
import numpy as np
from Optimizers import GradientClipping

g = np.random.rand(10) * 10
th = 5
eta = 1

eta_hat = GradientClipping()(g, eta, th)
g_hat = eta_hat * g

print(g)
print(g_hat)
l2n = (np.sum(g**2))**0.5
l2n_hat = (np.sum(g_hat**2))**0.5
print('L2norm of g     =', l2n)
print('L2norm of g_hat =', l2n_hat)
```

勾配クリッピングは、リスト 5.3.5 の実装では勾配そのもののかわりに学習率 eta の値を調節するようになってしている．そこで引数 eta = 1 として、返り値 eta_hat をもとの勾配 g にかけて勾配クリッピングにより調整された勾配を求める．とまれリスト 5.3.6 では、値の範囲を 0～10 に調整した乱数のベクトルを閾値 th = 5 で勾配クリッピングして、その前後のベクトルの値と L2 ノルムの値を表示する．勾配クリッピングによって、処理後の L2 ノルムが閾値より小さくなることが確かめられるはずだ．ベクトルの大きさやその値の範囲、そして閾値の値をいろいろ変えて確かめてほしい．

勾配クリッピングを先に説明してきたが、それでは最適化関数の本体部分を作っていこう．先に述べたように、派生クラスとして実装する．まずは第 4 章でも使ってきた勾配降下法の基本となる Stcastic Gradient Descent 略して SGD だ．

リスト 5.3.7 SGD

```
class SGD(OptimizerBase):
    def __init__(self, **kwargs):
        super().__init__(**kwargs)

    def __call__(self, gradient, eta):
        return eta * gradient
```

第 4 章のリスト 4.4.4 重みとバイアスの更新で示したように、SGD では学習率×勾配が重みやバイアスの更新値だ．したがって SGD クラスは勾配と学習率を引数として受け取り、学習率×勾配を返り値とするだけだ．

ここでリスト5.3.4のOptimizerBaseクラスとの関係を見ていこう.SGDクラスはOptimizerBaseクラスを基底クラスとする派生クラスだ. そしてそのコントラクタは、OptimizerBase クラスのコントラクタをそのまま呼出して、勾配クリッピングをインスタンス化してアトリビュートにつなぎこむ. 引数 kwargs のやり取りは将来の拡張のためであって、ここでは使わない.
重みやバイアスの更新を行う際には、リスト 5.3.3 に示すように、最適化関数の update()メソッドを呼出す.すなわちリスト5.3.4のupdate()メソッドだ.このとき派生クラスにはupdate()メソッドは定義されず、基底クラス OptimizerBase の update()メソッドが実行される. このupdate()メソッドでは、kwargs にキーワード'eta'で指定された学習率またはそのデフォルト値 0.01 を変数 eta に設定し、その eta は勾配クリッピングが適用される. そうして調整された eta を引数として勾配とともに派生クラスの__call__()メソッドに渡す. その派生クラスがSGDならばeta*gradientが返り値だ. その返り値をそのまま y として update()メソッドの返り値にする. すなわち勾配クリッピングによって調整した学習率で、派生クラスによって定義された方法による更新値が得られるのだ.

それではSGD以外の更新方法も実装していこう.

リスト 5.3.8 Momentum

```
class Momentum(OptimizerBase):
    def __init__(self, **kwargs):
        self.momentum = kwargs.pop('momentum', 0.9)
        super().__init__(**kwargs)
        self.vlcty = None

    def __call__(self, gradient, eta):
        if self.vlcty is None:
            self.vlcty = np.zeros_like(gradient)
        self.vlcty -= (1 - self.momentum) * (self.vlcty - gradient) # 移動平均
        return eta * self.vlcty
```

リスト 5.3.8 は Momentum だ. 一言でいうならば、これは勾配そのものにかえて、その移動平均を使って更新する方法だ.移動平均は時系列データを平滑化する手法として知られる.その一種の単純移動平均は第4章のリスト4.9.7のprogress()メソッドにも組み込んだが、Momentumでは指数移動平均を使う.
指数移動平均は指数平滑平均とも呼ばれるが、最近の値を重視し、過去になるほど減衰するように重み付けする. すなわち、ある時点 t での値を x_t その時点の移動平均を y_t とし、平滑化係数を α とすれば、$0 \leqq \alpha \leqq 1$ で次の漸化式であらわされる.

$$y_t = \alpha x_t + (1 - \alpha) y_{t-1} \tag{5.3.4}$$

これを変形すると、

$$y_t = y_{t-1} - \alpha(y_{t-1} - x_t) \tag{5.3.5}$$

ここで勾配の移動平均だが、値は gradient、移動平均は self.vlcty とし、平滑化係数は、その逆に過去の勢いをどれだけ重んじるかを self.momentum すなわち α = 1 - self.momentum とすれば、__call__()メソッド中の次式

self.vlcty -= (1 - self.momentum) * (self.vlcty - gradient)

となる．ここで右辺の self.vlcty は y_{t-1} に、左辺の self.vlcty は y_t に対応することに注意されたし．とまれあとは、コントラクタで self.momentum に kwargs からキーワード'momentum'でデフォルト 0.9 を与え、self.vlcty の初期値は None とし、__call__()メソッドが呼ばれたときに self.vlcty が None ではじめての時には、値 0 で初期化する．こうするのはコントラクタの段階では、まだ勾配の形状がわからないから__call__()メソッドで勾配が与えられてから、その形状に合わせて初期化するのが良いからだ．そして返り値は eta * gradient にかえて eta * self.vlcty だ．

これで Momentum が実装できたわけだが、あらためて考えると、勾配を移動平均で見ることにより、毎回勾配が小刻みに大きく変動する、いうならば振動するような場合に、その小刻みな変化を抑えて大きな方向性を捉えるという点に長じていると考えられる．

続いて RMSProp をリスト 5.3.9 に示す．

リスト 5.3.9 RMSProp

```
class RMSProp(OptimizerBase):
    def __init__(self, **kwargs):
        self.decayrate = kwargs.pop('decayrate', 0.9)
        super().__init__(**kwargs)
        self.hstry = None

    def __call__(self, gradient, eta):
        if self.hstry is None:
            self.hstry = np.ones_like(gradient)
        self.hstry -= (1 - self.decayrate) * (self.hstry - gradient ** 2) # 移動平均
        return eta * gradient / (np.sqrt(self.hstry) + 1e-7)
```

リスト 5.3.9 は、変数名が違う以外は、リスト 5.3.8 とよく似ている．移動平均を求めているのも同じだ．しかし移動平均は、勾配そのものではなく、その 2 乗に対して求めている．そして返り値を

eta * gradient / (np.sqrt(self.hstry) + 1e-7)

として、その移動平均の大きさに応じて調整するようにしている．これは、勾配そのものを平滑化するのではなくて、勾配の大きさの移動平均をとることにより、勾配の大きさを大局的に捉えて、それが大きければ小さく、小さければ大きくなるように更新量を調節する．勾配そのものを調節するかわりに学習率を調整している、という見方もあるようだが、いずれにせよ更新量に行きつくのであって、実際に勾配クリッピングのところで説明したように、どちらがどうというのでもない．ただ調節の効き方は大きく違うので、Momentum でうまくいくケースもあればRMSProp が良い場合もあって一概には言えない．むしろ肝要なのは、効き方の違うものを用意しておくことによって、様々な場合に対応できるようになるということなのだ．

ということで、また違う最適化関数として AdaGrad を実装しよう．

リスト 5.3.10 AdaGrad

```
class AdaGrad(OptimizerBase):
    def __init__(self, **kwargs):
        super().__init__(**kwargs)
        self.hstry = None

    def __call__(self, gradient, eta):
        if self.hstry is None:
            self.hstry = np.zeros_like(gradient)
        self.hstry += gradient ** 2
        return eta * gradient / (np.sqrt(self.hstry) + 1e-7)
```

リスト 5.3.10 に AdaGrad の実装を示すが、移動平均は使わず、むしろ単純な仕組みだ．変数 self.hstry は勾配の大きさの履歴を保持する変数だ．勾配の大きさは 2 乗で見ており、

self.hstry += gradient ** 2

で__call__()が呼ばれるたびに、すなわち、更新のたびに累積していく．そして返り値は、その累積値 self.hstry の平方根で割って調整する．だから更新量は次第に小さくなるように調整され、大きな勾配を経験するほど self.hstry が大きくなる．結果、はじめは大きかった更新量は次第に抑えられていき、いわば更新量の総和が何某かの値になるような調整となる．

ではもう一つ、Adam と呼ばれる最適化関数を実装しよう．これはリスト 5.3.8 の Momentum とリスト 5.3.9 の RMSProp を両方併せたような最適化関数だ．

リスト 5.3.11 Adam

```
class Adam(OptimizerBase):
    def __init__(self, **kwargs):
        self.momentum  = kwargs.pop('momentum',  0.9)
        self.decayrate = kwargs.pop('decayrate', 0.9)
        super().__init__(**kwargs)
        self.vlcty = None
        self.hstry = None

    def __call__(self, gradient, eta):
        if self.vlcty is None:
            self.vlcty = np.zeros_like(gradient)
        if self.hstry is None:
            self.hstry = np.ones_like(gradient)
        self.vlcty -= (1 - self.momentum)  * (self.vlcty - gradient)
        self.hstry -= (1 - self.decayrate) * (self.hstry - gradient ** 2)
        return eta * self.vlcty / (np.sqrt(self.hstry) + 1e-7)
```

リスト 5.3.11 に示すように、momentum と decayrate の 2 つのパラメタ、self.vlcty と self.hstry の 2 つの変数、それぞれ前者が Momentum、後者が RMSProp の同じ名前のパラメタと変数に対応する．そして返り値は、

eta * self.vlcty / (np.sqrt(self.hstry) + 1e-7)

となっているが、gradient の代わりに self.vlcty とするのが Momentum で、更新値を np.sqrt(self.hstry)で割るのが RMSProp だとみれば、両者を併せたものとなっていることがわかるだろう．そしてもちろん self.vlcty は勾配 gradient の移動平均、self.hstry は勾配 gradient の 2 乗の移動平均だ．

ここまで SGD、Momentum、RMSProp、AdaGrad、Adam と 5 種類の最適化関数を用意した．これだけあれば当面さまざまな場面に対応できるだろう．リスト 5.3.4～リスト 5.3.11 はファイル Optimizers.py としてまとめておこう．

説明は後先になったが、リスト 5.3.1 の BaseLayer のコントラクタで、

self.optimizer_w = cf.eval_in_module(optimizer_name, Optimizers)

self.optimizer_b = cf.eval_in_module(optimizer_name, Optimizers)

としているのは、この Optimizers.py の中に定義されている最適化関数の一つを optimizer_name で指定して、重みやバイアスの最適化関数としてインスタンス化しているのだ．そしてリスト 5.3.3 の update()メソッドでは、

self.w -= self.optimizer_w.update(self.grad_w, **kwargs)

self.b -= self.optimizer_b.update(self.grad_b, **kwargs)

とすることで、そのインスタンス化された最適化関数に従って重みやバイアスの更新が行われることになる.

これで最適化関数の実装ができたので、リスト 5.3.12 で簡単に動作を確認しておこう. これは第 4 章のリスト 4.6.13 とほとんど同じだ.

リスト 5.3.12 XOR の学習

```
import numpy as np
import NN
import common_function as cf

# 教師データの用意
x = np.array([[0, 0],[0, 1],[1, 0],[1, 1]])
t = np.array([[0],[1],[1],[0]])

# モデルを作る
model = NN.NN_m(1, ml_nn=10, ml_act='Sigmoid', optimize='Adam')
model.summary()

# 学習
n_learn = 1001
errors = []
for i in range(n_learn):
    y, l = model.forward(x, t)
    model.backward()
    model.update(eta=0.01, g_clip=1)
    errors.append(l)

    if i % 100==0:
        print('{:6d} {:6.3f}'.format(i, float(l)))

# 学習後に経過表示
cf.graph_for_error(errors, label='error', xlabel='n_learn')

# 学習済での順伝播と結果
print('input¥n', x)
y = model.forward(x)
z = y > 0.5
print('result¥n', z)
```

以前リスト 4.6.13 ではモデルを作る際に、重みの初期値の広がり係数 width=2.0 とし、これを指定しない場合には学習がうまくいかなかった. これに対しここでは、最適化関数を optimize='Adam' と指定すればうまく学習する. またこれ以外の最適化関数や、更新に際して勾配クリッピングを学習率と併せ指定するなどいろいろやってみて、動作を確認しておくのが良いだろう.

5.3.3 ニューラルネットワークの重みやバイアスの保存

これまでもニューラルネットワークの学習には時間がかかる場合が多々あった．しかしせっかく時間をかけて学習して良い重みやバイアスの値を獲得しても、これまではプログラムを終了するとそれらは消失し、消失するとまた乱数からはじめて学習しなおさなければならなかった．そこで、重みやバイアスをファイルに保存できるようにし、保存したものは、またニューラルネットワークに復元できるようにしておくことにしよう．

ところで、第4章の「4.6 多層化と活性化関数」で多層化したニューラルネットワークを損失関数も併せて一括管理する様にした．すなわち、リスト 4.6.1～リスト 4.6.6、リスト 4.6.11 で基底クラスとして NN_CNN_Base、派生クラスとして NN_0、NN_m を作った．そして今後 RNN も同様に一括管理していきたい．そこでこれらの一括管理の一環として、ニューラルネットワークの重みやバイアスを保存・復元を実装するのが良いだろう．

ではさっそく実装していこう．ニューラルネットワークから重みやバイアスを抽出する、それをファイルに記録する、ファイルに記録した重みやバイアスを取得する、それをニューラルネットワークに設定する、と 4 つの機能に分けて作っていこう．そしてこれらを NN_CNN_Base クラスのメソッドとして実装することにしよう．まずは重みやバイアスの抽出だが、RNN にも使えるように、はじめから戻りパスの重み v も含めておこう．

リスト 5.3.13 ニューラルネットワークから重みやバイアスを抽出

```
class NN_CNN_Base:

    def export_params(self):
        """ パラメタから辞書 """
        params = {}
        for i, layer in enumerate(self.layers):
            if hasattr(layer, 'w'):
                params['layer'+str(i)+'_w'] = np.array(layer.w)
            if hasattr(layer, 'v'):
                params['layer'+str(i)+'_v'] = np.array(layer.v) # RNN用
            if hasattr(layer, 'b'):
                params['layer'+str(i)+'_b'] = np.array(layer.b)
        return params
```

リスト 5.3.13 に export_params()メソッドを示す．抽出した重みやバイアスはパラメタとして辞書 params に入れて返り値とする．このためはじめに params = {}で空の辞書を用意し、for ループの中で、ニューラルネットワークを構成する層を self.layers から取り出して、その重みやバイアスを抽出する．このとき辞書のキーは、その層の名前、その層の番号、そして w か v か b かの区別をつなげたものとしている．いっぽう値の方は、np.array()で型指定している．またこのとき hasattr()でその層に w や v や b があるかを見ているが、こうしておけばニューロン層やRNN層に限らず、self.layersの要素としてさまざまな層を積み重ねておける．

次はリスト 5.3.13 とは逆に、ニューラルネットワークに辞書から値を設定する.

リスト 5.3.14 ニューラルネットワークに重みやバイアスを設定

```
class NN_CNN_Base:

    def import_params(self, params):
        """ 辞書からパラメタ """
        for i, layer in enumerate(self.layers):
            if hasattr(layer, 'w'):
                layer.w = np.array(params['layer'+str(i)+'_w'])
            if hasattr(layer, 'v'):
                layer.v = np.array(params['layer'+str(i)+'_v']) # RNN用
            if hasattr(layer, 'b'):
                layer.b = np.array(params['layer'+str(i)+'_b'])
```

リスト 5.3.14 に import_params()メソッドを示す. これはちょうどリスト 5.3.13 の逆の動作だ. 辞書は引数で受けとる. for ループで self.layers から layer を取り出して、その重みやバイアスに、リスト 5.3.13 で設定したキーで対応するものを引き出して値を設定する.

あとはファイルに記録するのと、ファイルに記録した重みやバイアスを取得する、2 つのメソッドを作ろう.

リスト 5.3.15 重みやバイアスをファイルに保存

```
import pickle

class NN_CNN_Base:

    def save_parameters(self, file_name):
        """ 学習結果の保存 """
        title = self.__class__.__name__
        params = self.export_params()
        params['title'] = title
        with open(file_name, 'wb') as f:
            pickle.dump(params, f)
        print(title, 'モデルのパラメータをファイルに記録しました=>', file_name)
```

リスト 5.3.15 に save_parameters()メソッドを示す. 格納先のファイルは引数で指定する. 格納は pickle を使うので、import pickle が必要で、どこかに記されていれば大丈夫だが、ファイル NN.py の冒頭にでも追記しておけばよいだろう. とまれ pickle を使えば、元の型や形状を保ったままでファイルとのやり取りができるから便利だ. この場合は辞書 params を pickle.dump()でファイルに格納する. その際に、辞書 params にニューラルネットワークのク

ラスの名前を付け加えて title としておく．self.__class__.__name__を使うが、これはこの NN_CNN_Base クラスを基底クラスとして継承する派生クラスのそれが得られる.

リスト 5.3.16 ファイルから重みやバイアスを取得

```
class NN_CNN_Base:

    def load_parameters(self, file_name):
        """ 学習結果の保存 """
        title = self.__class__.__name__
        with open(file_name, 'rb') as f:
            params = pickle.load(f)
        title_f = params.pop('title', None)
        print(title_f, 'モデルのパラメータをファイルから取得しました<=', file_name)
        if title_f == title:
            self.import_params(params)
            print('パラメータが継承されました')
        else:
            print('!!構成が一致しないためパラメータは継承されません!!')
        return params
```

リスト 5.3.16 は load_parameters()メソッドだ．リスト 5.3.15 と逆の動作で、pickle を使う．title に派生クラスの名前を得るのはリスト 5.3.15 と同じで、これはファイルから得られたそれと比較する．クラスの一致を確認するだけの粗いチェックだが、あったほうが良いだろう．返り値はファイルから得た辞書 params だが、title は pop()で取り出しているから、重みやバイアスの値のみが渡される．メッセージは適宜出力する.

ではさっそく重みやバイアスの保存と復元を使ってみよう．リスト 5.2.4、リスト 5.2.5 に分けて示したニューラルネットワークでマルコフ連鎖を例にしよう.
まずは重みやバイアスの保存を行うファイルを、たとえば次の 1 行で指定する．老婆心ながらこのとき、corpus_params というフォルダは予めこのプログラムを置く場所の 1 つ上の階層に作っておく必要がある.

リスト 5.3.17 重みやバイアスを保存するファイルの指定

```
file_name_m = '../corpus_params/params_' + nobel + '.pkl' # モデルのパラメータのファイル
```

この 1 行はファイルにアクセスする前ならばどこでも良いが、リスト 5.2.4 の冒頭の nobel = '羅生門'のすぐ後にでもしておくのが良いだろう．なおファイル名はここでは 1 例として上記のようにしたが適宜設定してもらえれば良い.

続いて重みやバイアスをファイルから復元する部分だ.

リスト 5.3.18 ファイルから重みやバイアスを取得

```
# モデルにパラメータを移植
if input('継承しますか？(y/n)=>') in ('y', 'Y'):
    model.load_parameters(file_name_m)
```

リスト 5.3.18 は、リスト 5.2.5 のモデルを作った後で学習の前に追記するのが良いだろう.

最後に、重みやバイアスをファイルに保存する部分だ.

リスト 5.3.19 ファイルに重みやバイアスを保存する

```
model.save_parameters(file_name_m)
```

リスト 5.3.19 の 1 行をリスト 5.2.5 の末尾に書き加えれば、プログラムを終わる際に重みやバイアスをファイルに保存する.
これに限らず、第 4 章でニューラルネットワークを使ったプログラムや、そしてこの先 RNN を使うプログラムにも適用可能なので、必要に応じて使っていけばよいだろう.

[補足 5.3.1]
リスト5.3.13では、= np.array(layer.w)というように、もともとnumpyの配列arrayとなっている重みやバイアスを、わざわざ型指定を行って代入しているように見える. これは冗長に見えるが、こうすることで元の配列がコピーされて別のものとなる. 逆にいうならば、numpyの配列をそのまま = で代入した場合には、その代入先の変数が元の配列を指すようになるだけ、つまり別名をつけるだけなのだ. たとえば、次の例でbはaがと同一であり、cはaの値がコピーされた別物だ.

```
>>> a = np.array([1,2,3])
>>> b = a
>>> c = np.array(a)
>>> id(a) == id(b)
True
>>> id(a) == id(c)
False
```

ここで続けて次のように辞書Pにaを入れても、その辞書内のaの変更が、元のaに反映されてしまう。

```
>>> P = {}
>>> P['a'] = a
>>> P['a'][0] = 0
>>> P['a']
array([0, 2, 3])
>>> a
array([0, 2, 3])
```

いっぽう、ここでnp.array()を使えば、辞書内のaの変更は、元のaに影響しなくなる.

```
>>> P['a'] = np.array(a)
>>> a
array([0, 2, 3])
>>> P['a'][1] = 0
>>> a
array([0, 2, 3])
```

これは不用意にコピーを作ってメモリの使用量を圧迫するようなことを避けるという点で効果的だが、プログラミングに際しては厄介なバグを作りこんでしまうことがあるので、意識しておく必要がある.

5.4 RNN

過去と現在から次に来るべきものを決めて文章を綴っていくことを、節 5.1 ではマルコフ連鎖の辞書を作って行い、節 5.2 では、その役割をニューラルネットワークが担うようにして行い、とやってきて、どちらのやり方でもそれなりの文書が生成できることを見てきた.
ここで辞書ということを少し離れて考えると、ニューラルネットワークで文章を綴っていく動作は、過去から現在に至るいくつかの状態を入力して、次の状態を出力しているにほかならない. 節 5.2 では、元の文章から一つずつ位置をずらしたデータを切り出して、これを入力とすることで過去から現在に至る状態をニューラルネットワークに与えた.
しかしいうまでもなく、現在は未来の過去なのだから、何らかの形で現在の状態を入力に戻せば、次の出力は現在の状態を含むものとなるだろう. そしてそれをまた入力に戻して、その状態を含めて次の次の出力とすれば、というように繰り返していけば、過去の状態の履歴を含んだ出力を得ることが出来るだろう. すなわち「次の状態が過去から現在に至る状態履歴に依存する」ということを、現在の状態を入力に戻すことによって、実現できるということなのだ.
そしてこのやり方の場合には、節 5.2 できちんと N 個の現在ならびに過去の状態を入力したたのに対し、いくつ過去に遡るのかは定かでない. しかし逆にとらえるなら、いかようにもできるということであり、「N 階マルコフ連鎖の階数にとらわれない汎用的な方法」に他ならない. そしてそれこそまさに、ニューラルネットワークの大きな一角を占め、時系列データ処理に欠くべからざる、RNN（Recurrent Neural Network、回帰型ニューラルネットワーク、リカレント・ニューラルネットワーク）そのものなのだ.

5.4.1 RNN をどう作るか？

それでは改めて、RNN をどう作るかを考えていくことにしよう. 図 5.4.1 に示すように、入力層、中間層、出力層からなる典型的なニューラルネットワークを例にしよう. 第 4 章で述べたように入力層は入力を次の中間層に配るだけなので実体はない. そして次の中間層はかくれ層とも呼ばれるが、ニューロンが配置されていて、入力に重みを掛けて、それらの和をバイアスとともにとる. すなわち重み付け和をとる. そしてそれを活性化関数で変換して中間層の出力とする. 出力層も同様に重み付け和、活性化関数で出力する.
ここで「現在の状態」とそれを入力に戻す戻りパスについて考える. 中間層であれ、出力層であれ、なんらかの現在の状態がある. ここで id やその one_hot に変換したものを、敢えて元の文字や語のどれに相当するかで表すことにして、具体例として「羅生門」の出だし「ある日の暮方の事である。」について考えてみよう.
まず出力層の出力を考えてみると、これは現在の出力＝次の値であって、これは文章を綴る場

合で言えば、次の文字や語そのものだ．ニューラルネットワークの入力と出力は、’ある’→’日’、その次に’日’→’の’というようにしたいのだが、この 2 つ目の’日’→’の’で考えると、2 つ目の入力は’日’だが、1 つ目の出力は’日’だから、’日’と’日’とを、入力と、1 つ目の出力からの戻りパスとして与えることとなって、同じ’日’を 2 つ与えても意味があるようには思われない．欲しいのは’日’の前が’ある’だったことなのだが、出力層の出力では「現在の状態」の情報がそぎ落とされてしまっている．逆に出力層の出力はあくまでも次に何が来るかを示すのであって、もはやそこでは「現在の状態」は余計な情報にすぎない．だからそこから「現在の状態」、この例では’ある’だったことを得ようというのは無理がある．ともかく出力層の出力を「現在の状態」として入力に戻すのはうまくないだろう．

それでは中間層の出力ならばどうだろうか？ 重み付け和をとった上に活性化関数によって変換されたものではあるけれども、まだこの段階ならば、その入力が何らかのかたちで情報として含まれる、あるいは含まれた状態を保つ余地がある、と言えるのではないだろうか？つまりあくまで模式的に言うならば次のように言えるだろう．すなわち’ある’が入力され、そしてニューラルネットワーク全体としては、そこから次の’日’を作ろうとする．そして現在が’ある’だということを中間層で把握するからこそ、次が’日’なのであって、その情報が出力層に渡されて、’日’を生成する、というふうに捉えれば分かり易いのではないだろうか？実際にはあくまでも、それぞれ重み付け和と活性化関数を備えたニューロン層を 2 つ重ねた途中の値に過ぎないから、ここで述べたように役割分担されるわけではないとは思うが …．

とまれこの見込みのある方法は、戻りパスがニューロン層の層内に閉じるから、実装しやすいというメリットもある．

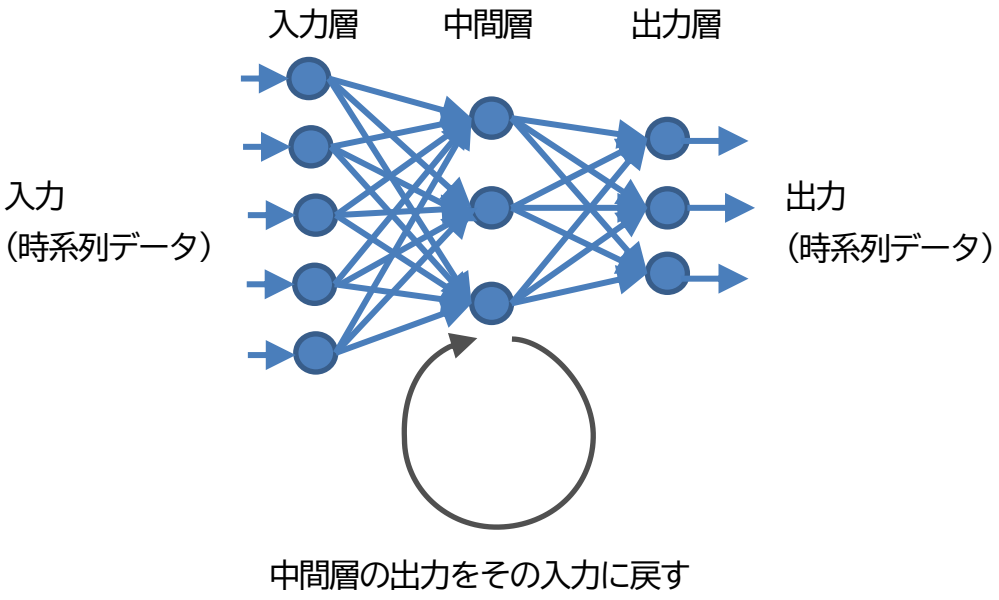

図 5.4.1 RNN

ともかくも、この戻りパスのある中間層は、上流の入力層からと、自身の出力からと、つごう2つの互いに独立な入力が必要だ．これを2つのニューロンを1組にして実現することもできる．しかしこの先ずっと使っていくことを考えて、これ専用にニューロン層を定義するところから始めていこう．

リスト5.4.1：戻りパスを有するニューロンユニット

```
class RNN_Unit:
    def forward(self, x, r, w, v, b):
        u = np.dot(x, w) + np.dot(r, v) + b
        y = np.tanh(u)
        self.state = x, r, y
        return y

    def backward(self, grad_y, w, v):
        x, r, y = self.state
        delta = grad_y * (1 - y**2)  # tanhの逆伝播
        grad_w = np.dot(x.T, delta)
        grad_v = np.dot(r.T, delta)
        grad_b = np.sum(delta, axis=0)
        grad_x = np.dot(delta, w.T)
        grad_r = np.dot(delta, v.T)
        return grad_x, grad_r, grad_w, grad_v, grad_b
```

リスト 5.4.1 に戻りパスを有するニューロンユニットを示す．ここではこれをクラスとしてRNN_Unitとした．ちなみにRNNは内部に戻りパスを持つニューラルネットワーク全体、たとえば図5.4.1の入力から出力に至る全体を指す．しかしここでは便宜上、その主役である戻りパスを持つニューロンユニットそのものをRNN層と呼ぶことにする．また見てもとおり、リスト5.4.1の範囲では単に互いに独立な入力が2つあるだけで、そこに何をつなぐかはわからない．しかし後でこのニューロンユニットを使う層を定義するところで 2 つの入力を使い分ける．とまれ上流からの入力は x とし、これと区別するために戻りパスからの入力は r とした．そして重みやバイアスは、引数として外から与えるようにした．そして上流からの入力に対応する重みとバイアスは、これまで同様に w、b とする一方、戻りパスに対する重みは v とした．順伝播は、第4章でさんざんやってきたことから容易に類推できると思う．はじめに行列積によって入力の重み付け和を得るのだが、このとき x の w による重み付け和と r の v による重み付け和を、それぞれ独立に取ってから、バイアス b を含めて足し合わせる．もちろんこれは x の影響は w で r の影響は v で、それぞれ独立に調整するためだ．
続いて活性化関数だが、ここではハイパボリックタンジェント関数を使う．これについては、第4章の4.9.2で説明した．これは出力に対する活性化関数であると同時に、値の範囲が-1～$+1$ になって、戻りパスとして r に入力される値の範囲を調整するように働く．

x 、 r 、そして y は逆伝播で必要なので、self.state として保存し、出力 y を返り値にして順伝播を完了する．逆伝播もまた、第 4 章からの類推で理解できると思う．

5.4.2 RNN 層の仕組み

それでは実際に内部に戻りパスを持つようにニューロン層を構成していこう．

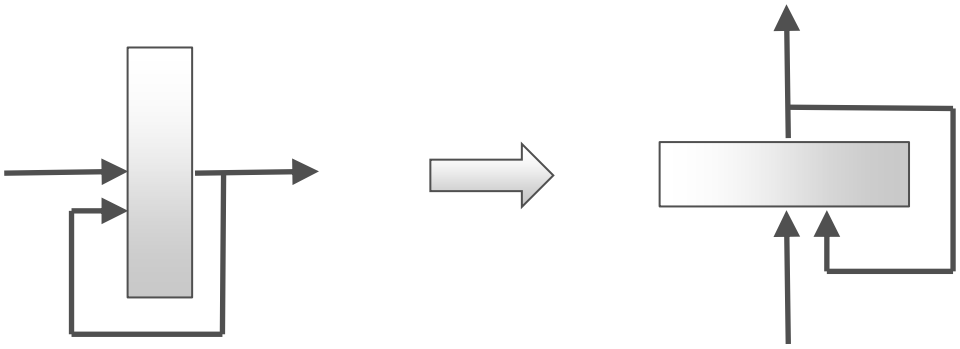

図 5.4.2 RNN

図 5.4.2 の左図に示すように、通常は信号の流れが右→左になるようにして、ニューロン層を表す．そしてこれまで述べてきたように、このニューロン層の出力を入力に戻すのだが、文章のような時系列データの流れを追うのに、これを 90° 回して同右図のように信号が下から上へと流れるように表すのが都合が良い．

そして時刻 $t = 0$ から順に $t = 1, \ldots, t = \mathrm{T} - 1$ に亘り、その入力を x 出力を y としてニューロン層で処理されるとすれば、その様子は図 5.4.3 のように表される．たとえば先の「羅生門」の出だしならば $x_{t=0}$ が ’ある’、$x_{t=1}$ が ’日’、… というように対応し、これに対してこの層の出力が $y_{t=0}$ ， $y_{t=0}$,…といった具合だ．繰り返しになるが、このとき出力の y はあくまでも途中結果だ．だから対応する何かを示すのは難しい．しかしともかくも、その時刻での入力の結果生じたその時刻での状態だ．そしてそれが次の時刻で入力 x とともにニューロン層に入り、次の時刻の出力を作る．

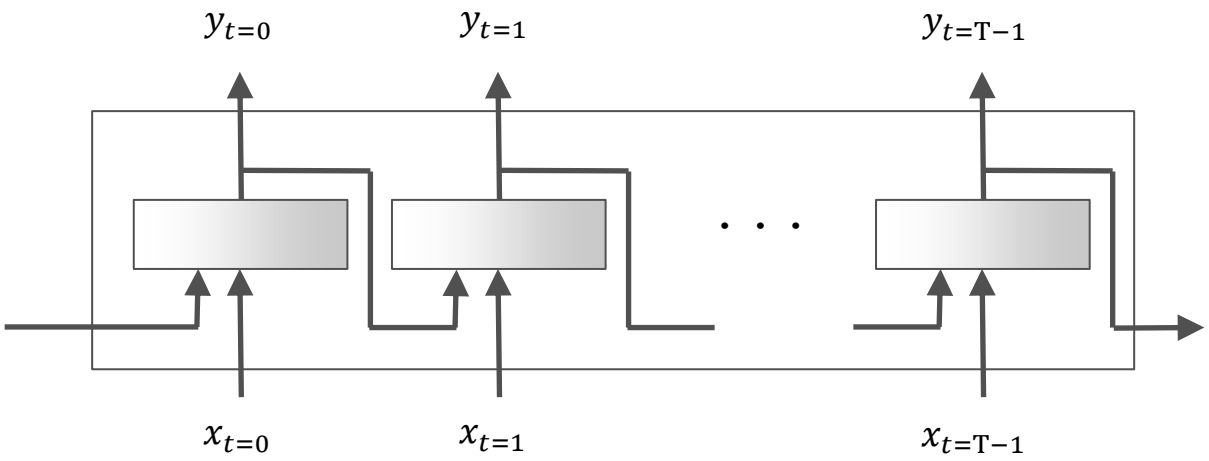

図 5.4.3 RNN層

図5.4.3の横に展開して示したニューロン層の小さな四角の中は、リスト5.4.1の戻りパスを有するニューロンユニットとすることが出来る．ここで「出来る」という言い方をするのは、図にあるような接続が出来さえすれば、内部の構造は必ずしもリスト5.4.1には限らないからだ．ともかくここではこの四角の中は、リスト5.4.1であるとしよう．リスト5.4.1では重みやバイアスは引数で外から与えるが、これはどうしたら良いだろうか？ 図5.4.3に示すように、入力、出力、そしてそれに応じた内部の状態は、時刻 t ごとに異なる．しかしこの四角の箱は同じニューロンユニットを時刻毎に展開したものだ．とするならば重みやバイアスは共通だ．逆に共通でなく、時刻毎に別の重みやバイアスを用意するとすれば、偶々ある時刻の範囲でうまく働くように調整した重みやバイアスが、別の時刻の範囲で入力が別の値で、その時の出力も別の値が欲しいとすれば、また別に重みやバイアスを用意するような具合になってきりがない．ならば時刻によらず共通としておいて、その重みやバイアスにあらゆる時刻での働きを期待するべきだろう．これは無理なことを言っているようにも聞こえる．しかし節5.2で、少なくとも小説に記述された範囲において、一つのニューラルネットワークが、あらゆる場面の文章を紡ぎ出していたことを思えば、これは根拠がないわけではない．ともかく重みやバイアスを時刻によらず共通のものとしよう．そしてそのかわり、ニューロンユニットを複数個並列に置いて、そのそれぞれに重みやバイアスを与えよう．節5.2のリスト5.2.5で中間層のニューロン数を100としたように．そしてこのようにニューロンをたくさん並べて、その重みやバイアスを学習の成り行きに任せて調整するのは、ニューラルネットワークでの常套手段だ．つまり図 5.4.3 の四角は、奥行き方向にたくさん並んでいると考えてもらえば良い．

5.4.3 RNN層の実装

RNNの仕組みがわかったので、これをプログラミングしていこう.

リスト5.4.2：RNN層の順伝播

```
class RnnLayer:

    def forward(self, x):
        B, T, m = x.shape
        _, n = self.config
        # 時系列の展開の準備をして、リカレントにr0をセット
        y = np.empty((B, T, n))
        self.layer = []
        rt = np.zeros((B, n)) if self.r0 is None else self.r0
        # unitを起こしながら、順伝播を繰返す
        for t in range(T):
            unit = RNN_Unit()
            xt = x[:, t, :]
            rt = unit.forward(xt, rt, self.w, self.v, self.b) # rt上書き
            y[:, t, :] = rt
            self.layer.append(unit)
        self.r0 = rt # last_y
        return y
```

図5.4.3に示すRNN層の順伝播をプログラムで表したものがリスト5.4.2だ. forward()メソッドのはじめに入力xの形状をB, T, m = x.shapeにより見ていて、BとTとmの3つの次元を持つことにしているが、これは次の理由だ. すなわち、図5.4.3で下方に示す入力xは時系列データで $x_{t=0},\cdots,x_{t=\mathrm{T}-1}$ と時刻の次元Tをもつ. RNN層での処理は、図5.4.3に示すように $t=1$ から $t=\mathrm{T}-1$ まで順を追って逐次的に行わなければならないから、処理時間が長くかかってしまう. そのためバッチ処理でデータをまとめて処理することは必須だ. したがって入力データはそのバッチ方向の次元Bもあり、さらにもともと一つのデータは大きさmを持つから、つごうB、T、mの3次元となる. ついでながら先に述べたように、図5.4.3の箱で示されるニューロンユニットは複数個なので、この個数は変数nで表している. self.configに入力の大きさとニューロン数を保持するのは、第4章でリスト4.1.4以来ずっとやってきたことだ. ここでは入力の大きさは、入力xからわかるから、_, n = self.configとしている.
時系列の展開をする前の準備として、出力yとself.layerと戻りパスself.r0を揃えておく. B,T,mの形状の入力xに対する出力yはB,T,nの形状になる. これは時刻方向の次元が増えていることを除けば、第4章でB,mの形状の入力に対する出力がB,nだったのと同様だ. とまれ出力yの値は展開の中で入れるようにして、その前にy = np.empty((B, T, n))で形状だけ決めておく. self.layerは空のリストを用意しておく. 戻りパスself.r0だが、図5.4.3の左端の時刻0のところでは、戻りパスに入れるべき値が場合によって異なる. 以前の状態がない

始まりのところでは、前の時刻の出力はまだ無い．その状態では self.r0 は None ということにしておこう．そういう場合には展開のはじめの時刻では戻りパスの値は 0 とする．リスト 5.4.1 からわかるように、それにより、ニューロンユニットの順伝播の計算で戻りパスの影響がなくなって、入力 x の重み付け和だけとなる．いっぽう前に行った順伝播に続いて forward() メソッドが呼ばれる場合を想定すると、その既に行った順伝播の最後の出力を戻りパスに与えるべきだろう．これはあたかも図 5.4.3 の右にまた図 5.4.3 が続くような場合だ．このとき、図 5.4.3 の右端の行先の示されない矢印を、この図の右におかれた図 5.4.3 の、その左端の出どころの示されない戻りパスにつなげば良い．この様子を図 5.4.4 に示す．このときは self.r0 に前回の最後の時刻の出力が保存されているものとして、それを展開の最初の戻りパスにセットする．

時系列の展開は for ループで入力 x の時系列の長さ T に亘り繰り返す．リスト 5.4.1 の RNN_Unit をインスタンス化して、その展開の unit とする．

入力 x の現在の時刻の値を xt = x[:, t, :]により取り出して、これを戻りパス rt とともに、rt = unit.forward(xt, rt, self.w, self.v, self.b)により、unit に与えて順伝播する．このとき unit の重みやバイアスは、このクラス RnnLayer でアトリビュートとして持っているものを使う．また出力は直接、戻りパスの変数 rt を上書きして、次のループに備える．そして先に用意しておいた y の現在の時刻の値として y[:, t, :] = rt でセットする．そしてループの最後に self.layer.append(unit)で self.layer に、たった今順伝播した unit を加える．これにより順伝播の結果の内部の状態を伴った unit が時系列の長さに亘って組み上げられていく．

時系列の展開を終えると、その最後の出力を self.r0 = rt で保存する．時系列の続きの処理では、この保存した self.r0 が使われる．

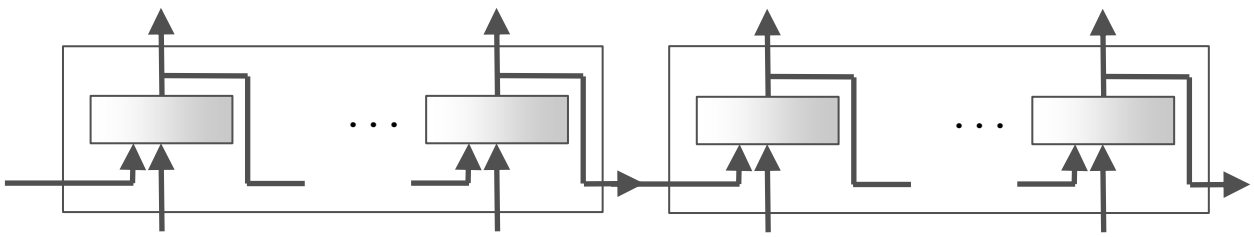

図 5.4.4 次の時刻への接続

さて図 5.4.3 の通りにリスト 5.4.2 に RNN 層の順伝播をプログラミングしたわけだが、この図 5.4.3 の $t = 0$ から $t = T$ にわたる時刻の範囲を一区切りとして時系列の処理をする．順伝播に応じて逆伝播でも、この一区切りの時刻の範囲を処理することになる．図 5.4.4 に示すよ

うに順伝播では、戻りパスの最初の値に、前の処理の最後の出力を与えることで、この一区切りで処理する時系列の長さを超えて時刻が続いていく．では逆伝播ではどうかというと、このクラス RnnLayer で保存しているのはあくまでも、この一区切りの時系列の中の状態だけだから、それを超えて逆伝播をデータの末尾からずっと続けていくのは無理がある．すなわち RnnLayer 内に self.layer としてその時刻の状態を含めて保存するばかりでなく、さらにその self.layer を時系列にわたり保存する RnnLayer を、その状態を含めて時系列のかたまりの順に全て積上げていく必要があって、長い小説を処理する場合などを想定すれば、とんでもない情報量となる上に、勾配を伝播していく経路の長さがとんでもなく長くなってしまって甚だ困難となると考えられ現実的でない．これに対し、逆伝播では時系列内についてだけ、勾配を伝播することにすれば、RnnLayer は一つだけで良いし、学習の際にも時系列で区切ったデータを用意して、順伝播→逆伝播→更新を繰り返せば良いことになる．このように逆伝播において、勾配の伝播を時系列長内に限定して行うやり方を、Truncated Backpropagation Throuh Time という．この場合にもリスト 5.4.2 で説明したように、順伝播では処理の最後の出力を保存しておいて、これに続く次の処理で、戻りパスの最初の値として与えることで、長い時系列のデータが処理できる．

説明が長くなったが逆伝播をプログラミングしよう．

リスト 5.4.3：RNN 層の逆伝播

```
class RnnLayer:

    def backward(self, grad_y):
        B, T, n = grad_y.shape
        m, _ = self.config
        self.grad_w = np.zeros_like(self.w)
        self.grad_v = np.zeros_like(self.v)
        self.grad_b = np.zeros_like(self.b)
        self.grad_x  = np.empty((B, T, m))
        grad_rt = 0
        for t in reversed(range(T)):
            unit = self.layer[t]
            grad_yt = grad_y[:, t, :] + grad_rt # 出力からとリカレントからの勾配を合算

            grad_xt, grad_rt, grad_wt, grad_vt, grad_bt = ¥
                        unit.backward(grad_yt, self.w, self.v) # grad_rt 上書き

            self.grad_w += grad_wt
            self.grad_v += grad_vt
            self.grad_b += grad_bt
            self.grad_x[:, t, :] = grad_xt
        self.grad_r0 = grad_rt
        return self.grad_x
```

順伝播で組み立てていったニューロンユニットを、逆伝播では時刻を逆に辿りながら解きほぐしていくように展開する．そして、時刻によらず共通の重みやバイアスでは、逆伝播の各時刻で得られる勾配を足し合わせたものが、その一度に処理する時系列の勾配ということになる．いっぽう順伝播の際の入力は、もともと時刻の次元を持つ時系列データだから、処理する時刻の勾配は、その時刻の入力に対する勾配となる．

このことを念頭において、リスト 5.4.4 の RnnLayer の逆伝播 backward()メソッドを見ていこう．下流からの勾配は引数 grad_y で受け取る．B, T, n = grad_y.shape でバッチサイズ B と時系列長 T とニューロン数 n を得る．この n は並列に動作するニューロンユニットの数であり、いっぽう T は、その n 個のニューロンユニットたちが時刻方向に逐次的に並ぶ数だ．入力の大きさは m, _ = self.config で得る．ちなみに、この '_' のところの値は n だが grad_y から得られるから省いている．

さて先に述べたように勾配は逆伝播の展開で積上げる w、v、b に関するものと、個別に設定する x に関するものと、それぞれ零および空で用意する．また、Truncated Backpropagation Throuh Time として説明したように、逆伝播では戻りパスは時系列長で区切るので、grad_rt = 0 として境界で勾配を 0 に初期化する．これは図 5.4.3 の外側の箱から右に出る矢印の逆伝播を行わないことになる．

ここからは時系列の展開だ．for ループで時系列長 T に亘り逆順に展開していく．

unit = self.layer[t]で順伝播の際に組み立てたものから unit を取り出す．コメントにあるとおり grad_yt = grad_y[:, t, :] + grad_rt で出力からとリカレントからの勾配を合算するが、これは順伝播の際に unit の出力を層の出力として外に出すのと、戻りパスとして自身の次の時刻に入力することに対応する．そして順伝播の最後の時刻 T-1 に対応する逆伝播の最初の処理では、ループに入る直前に初期化された 0 となって、層の出力からの勾配だけに依存する．とまれ unit で逆伝播すると各勾配が得られるから、w、v、b の勾配を積上げ、x の勾配はその時刻の勾配として設定する．このとき戻りパスの勾配 grad_rt は上書きして、次のループで層の外からの勾配と合算して出力の勾配として使う．

展開が終わると、最後に得られた戻りパスの勾配 grad_rt は self.grad_r0 として保存し、入力 x の勾配 grad_x を返り値として完了する．保存した self.grad_r0 の使い道は後に譲り、ここでは触れない．

説明してきたように、ここで実装したニューロン層 RnnLayer は、層内の構成をダイナミックに行う．すなわち、順伝播の際に RNN_Unit を内部の状態を伴って層を組み立てて、逆伝播ではそれを順伝播の際の状態とともに辿る．これによって、その処理するデータに合わせて時系列の長さを自由に設定できる．いっぽう重みやバイアスは、時刻によらず共通であって、並列に動作させるニューロンユニットの数によって決まる大きさを持ち、RnnLayer が管理する．逐次は可変で並列は不変の、この仕組みは第 4 章で扱ってきたニューロン層にはなかったもの

だ.
先に forward()メソッドと backward()メソッドを説明してきたが、コントラクタをはじめとして作っておかなけてはならないメソッドがあるので、あい前後するがこれらを説明していこう．まずはコントラクタだ.

リスト 5.4.4：RNN 層のコントラクタ

```
class RnnLayer:

    def __init__(self, *configuration, **kwargs):
        print('Initailize', self.__class__.__name__)
        self.width      = kwargs.pop('width',     None)
        optimizer_name  = kwargs.pop('optimize', 'SGD')

        self.w = None; self.v = None; self.b = None
        self.optimizer_w = cf.eval_in_module(optimizer_name, Optimizers)
        self.optimizer_v = cf.eval_in_module(optimizer_name, Optimizers)
        self.optimizer_b = cf.eval_in_module(optimizer_name, Optimizers)

        self.config = configuration
        m, n = self.config
        self.init_parameter(n, m, n)
        self.reset_state()
```

リスト 5.4.4 が RnnLayer クラスのコントラクタだが、活性化関数を指定しないかわりに重みやバイアスの最適化関数を指定する．最適化関数については後述するが、RNN ではこの指定が必要になる場合が多い．というのも図 5.4.3 の戻りパスの経路を逆伝播で辿ることを考えると、時系列長に亘りいくつものニューロンユニットを通ることになって、この一つのニューロン層だけでも、とても深い層の重なりに相当するからだ.
また、第 4 章で扱った NeuronLayer クラスでは、順伝播の際に、その入力の大きさで構成を確定できるようにしたが、ここではコントラクタの中で、その引数から構成を確定する．これは RnnLayer では入力の時系列長が可変なので、煩雑さを避けるためにこうした.
コントラクタの最後に reset_state()メソッドを呼んでいるが、これについては後述する.
これら以外の点では第 4 章の NeuronLayer と同様だ.
続いてパラメータの初期化をリスト 5.4.5 に示す．すでに述べたように、これはもっぱらコントラクタから呼ばれる.

リスト 5.4.5：RNN 層のパラメータの初期化

```
class RnnLayer:

    def init_parameter(self, l, m, n): # l:戻りパス、m:入力、n:ニューロン数
        width = self.width
        if width is not None:
            width_w = float(width)
            width_v = float(width)
        else:
            width_w = np.sqrt(1/m) # Xavier の初期値
            width_v = np.sqrt(1/n) # Xavier の初期値
        self.w = (width_w * np.random.randn(m, n)).astype(Config.dtype)
        self.v = (width_v * np.random.randn(l, n)).astype(Config.dtype)
        self.b = np.zeros(n, dtype=Config.dtype)
        print(self.__class__.__name__, 'init_parameters', l, m, n)
```

基本的なところは第 4 章のリスト 4.1.7 と同じだが以下の違いがある．すなわち、戻りパスの関係を追加、パラメータ初期値の広がりは Xavier、という違いがある．
戻りパスの関係では、引数 l で戻りパスの大きさを指定し、これに従って重み self.v を初期化する．引数 l は基本的には、自身のニューロンユニットからの出力のデータサイズで n を設定するから、不要に思われるかもしれない．しかし、リスト 5.4.1 の基本的な RNN_Unit ではその通りだが、進化したものではそうとは限らないから、このようにしている．
さらに細かい点だが、Xavier の初期値で広がりを決める際に、この l ではなくニューロン数 n を使っているが、これも同様の理由だ．細かい点だが、astype()で精度を決める前に width_w、width_v と乱数の行列のかけ算を済ませておかないと、self.w や self.v の精度が担保できないから注意が必要だ．とまれ Xavier の初期値についてここでは、こうしていくとうまくいきやすいと思っておいてくれれば良い．重みを初期化する際の広がり width の値が学習に大きく影響するということは、第 4 章の中で経験していると思うが、RNN は第 4 章で扱った基本的なニューラルネットワーク以上に学習がうまくいきやすいように配慮する必要があることは気に留めておいてほしい．

次はパラメータの更新だ．これをリスト 5.4.6 に示す．

リスト 5.4.6：RNN 層のパラメータの更新

```
class RnnLayer:

    def update(self, **kwargs):
        self.w -= self.optimizer_w.update(self.grad_w, **kwargs)
        self.v -= self.optimizer_v.update(self.grad_v, **kwargs)
        self.b -= self.optimizer_b.update(self.grad_b, **kwargs)
```

更新の際に最適化関数を呼び出す、戻りパスに関する重み self.v の更新も行う、という 2 点だけで、あとは説明は不要だろう.

続いて初期化をリスト 5.4.7 に示す.

リスト 5.4.7：RNN 層の初期化

```
class RnnLayer:

    def reset_state(self):
        self.r0 = None
```

初期化は戻りパスの時系列の最初のところにあたる self.r0 を None にする.
リスト 5.4.2 で示すように順伝播では、この self.r0 が None の場合には、時系列のはじまりとして、戻りパスの値を 0 で初期化する.
これはコントラクタから呼出すほかに、時系列の扱いに関して必要に応じて呼出す. これは RNN がそもそも前の状態を引き継いで、その後の状態を作るからにほかならない. だから、ある一連の時系列の処理の後に、別の時点の処理を行おうとする場合には、その前に行った処理の影響を初期化によって排除する必要がある. たとえば学習を進めたり、評価を行う際にその学習と評価の境目で、また生成を行う際の時系列の切れ目で初期化する.

5.5 層を重ねて一括して RNN として扱う

第 4 章で行ったと同様に層を重ねて多層化したものを RNN として一括して扱えるようにしていこう. なお前節では、ニューロン層とこれを多層化したニューラルネットワークの両方に機能を追加あるいは向上したが、いうまでもなく、これらは RNN においてそのまま役に立つ.

5.5.1 RNN の実装

RNN を実装するにあたり、第 4 章でニューラルネットワークを実装したのと同様に、基底クラスで RNN に共通の機能を実装し、個々の RNN はその派生クラスとして、これを継承するようにする. そしてその RNN の基底クラスは NN_CNN_Base を継承することで、第 4 章で実装した機能と前節で実装した機能を使えるようにする. つまり、基底クラスと派生クラスの関係を親子にたとえならば、個々の RNN は NN_CNN_Base の孫ということになる. とまれ RNN の基底クラスのコントラクタから作っていこう.

リスト 5.5.1 RNN の基底クラスのコントラクタ

```
import common_function as cf
import NN_CNN_Base

class RNN_Base(NN_CNN_Base):
    """ RNN 一般に共通の部分 """
    # 初期化以外は共通
    def __init__(self, *args, **kwargs):
        # RNN 専用"
        # args : l, m, n ないし v, l, m, n
        # v:語彙数、l:語ベクトル長、m:隠れ層ニューロン数、n:出力数
        self.config = args
        self.layers = []

        # 損失関数
        loss_function_name = kwargs.pop('loss', 'MeanSquaredError')
        self.loss_function = cf.eval_in_module(loss_function_name, LossFunctions)
```

リスト 5.5.1 に RNN_Base クラスのコントラクタを示す. 第 4 章のリスト 4.6.1 に示す NN_CNN_Base のコントラクタと同様に、コントラクタで損失関数の組み込みを行うが、少し違うところがある. RNN では構成に関連して指定が必要なものが違うため、これは引数 args からそのまま受け取って self.config としている. RNN では後述する Embedding 層を組み込むこともあってややこしくなるので、第 4 章までのニューラルネットワークとは異なり、入力の大きさも含めて大きさに関する指定はすべてコントラクタで行い、派生クラスに入出力の大きさを渡す必要もない. このためかえって簡単になっている.

リスト 5.5.2 RNN の初期化

```
class RNN_Base(NN_CNN_Base):

    def reset_state(self):
        for layer in self.layers:
            if hasattr(layer, 'reset_state'):
                layer.reset_state()
```

リスト 5.5.2 に RNN の初期化のための reset_state()メソッドを示す．初期化は各層に初期化のメソッドがあるものを順に実行するだけだから、for ループで self.layer から層をとり出して reset_state の有無を見て、あれば実行するだけだ．

これで一応 RNN の基底クラスの実装が完了する．そこでさっそく派生クラスとして、RNN 層に普通のニューロン層を重ねた基本的な RNN を作ろう．

リスト 5.5.3 RNN_rf

```
class RNN_rf(RNN_Base):
    def __init__(self, l, m, n, **kwargs):
        super().__init__(l, m, n, **kwargs)
        self.rnn_layer = Neuron.RnnLayer(l, m, **kwargs)
        self.neuron_layer = Neuron.NeuronLayer(m, n, **kwargs)
        self.layers.append(self.rnn_layer)
        self.layers.append(self.neuron_layer)
```

リスト 5.5.3 が最も基本的な RNN だ．ここでは普通のニューロン層を区別するため、全結合 full connection の意味を込めて f の文字をあてて、クラス名を RNN_rf とした．順・逆伝播をはじめとしてすべての機能は親ないし祖父母にあたる RNN_Base ないし NN_CNN_Base が担うので、コントラクタを記述すれば完了だ．このあたりの事情は第 4 章でリスト 4.6.6 で NN_0、リスト 4.6.12 で NN_m を作ったのと同様だ．簡単なので説明は要しないとは思うが、少し注意する点として、この派生クラスのコントラクタの引数 l, m, n で大きさに関して指定し、これが基底クラス RNN_base では引数 args となる．そして基底クラスのコントラクタのコメントにあるように、l:語ベクトル長、m:隠れ層ニューロン数、n:出力数だから、RNN 層は l, m、ニューロン層は m, n で、それぞれ大きさを指定してインスタンス化する．最後にインスタンス化した層を self.layers に要素として加えて完了だ．

リスト 5.5.1、リスト 5.5.2、リスト 5.5.3 は一緒にして RNN.py に入れておこう．

5.5.2　二進数の足し算

二進数の足し算を筆算ではどうやるだろう？ 一例として図 5.5.1 に 1001 + 0011 = 1100 を示す.

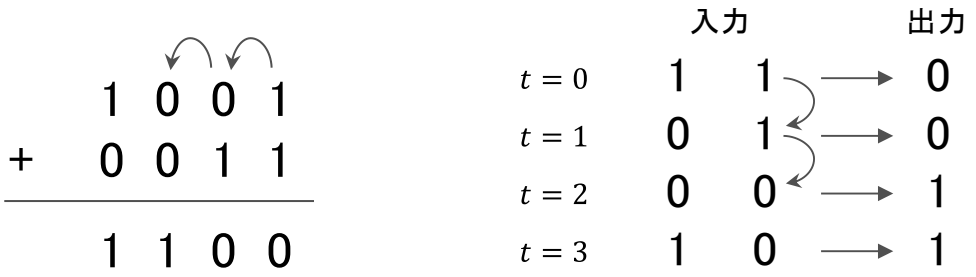

図 5.5.1 二進数の足し算の例

筆算の様子を図 5.5.1 の左図に示すが、手順は次のようになるだろう.
一番下の桁：1 ＋ 1 ＝ 0、繰り上がり 1
下から 2 番目の桁：0 ＋ 1 ＋ 繰り上がり 1 ＝ 0、繰り上がり 1
下から 3 番目の桁：0 ＋ 0 ＋ 繰り上がり 1 ＝ 1、繰り上がり無し
一番上の桁：1 ＋ 0 ＝ 1
この手順通りに左図を右図のように反時計回りに 90° 回して、小さい桁を上、大きい桁を下にすれば、時刻 $t=0$ から $t=3$ の時系列の処理として、2 つの二進数のそれぞれの桁を入力、答えを出力と捉えることができる. そうすると、繰り上がりがあるかどうかは、先に処理する時刻の小さい桁の入力の値に依存して、すなわち過去の履歴によって決まるから、これは戻りパスのある RNN で処理すれば、正しく答えが出せるはずだ.

それを実証していきたいが、下準備として十進数を二進数に変換するプログラムを作っておこう. 先に述べたように RNN での処理を考えると、小さい桁を先に大きい桁を後にする必要があるいっぽう、大きい桁を先に小さい桁を後にした方が分かり易い場合もある. 前者はリトルエンディアン Little Endian、後者はビッグエンディアン Big Endian と呼ばれるが、両方とも指定できるようにしておくと良いだろう. そして RNN の入力として使う際に都合が良いように二進数はリスト形式で返すようにしておく. また、二進数を十進数に戻すことも必要になると考えられるから、これも用意しよう. ということでリスト 5.5.4 に十進数と二進数の間の両方向の変換の関数を示す. これを base_conversion.py に置いておこう.

リスト 5.5.4 十進数と二進数の間の変換

```
def d2b(x, endian='big'):
    """ 十進数をリスト形式の 2 進数に変換 """
    y = format(x, 'b')
    if endian=='little':
        y = y[::-1]
    return list(y)

def b2d(x, endian='big'):
    """ リスト形式の 2 進数を十進数に変換 """
    y = "".join(map(str, x))  # リストの要素を結合して 1 つの文字列に
    if endian=='little':
        y = y[::-1]
    return int(y, 2)
```

リスト 5.5.4 の関数 d2b()は、十進数から二進数に変換する、そして関数 b2d()は、その反対に二進数から十進数に戻す逆変換だ．関数 d2b()で整数から二進数への変換は format()で'b'を指定するだけだ．あとはリトルエンディアンで桁を逆順にするために[::-1]とし、返り値はリスト形式にする．次はリスト 5.5.4 の関数 b2d()だが、引数 x は二進数の各桁が各要素として並んだベクトルなので、map(str, x)で要素を文字列に変換してから、"".join()で 1 つに結合する．後はリトルエンディアンなら逆順にし、int(y, 2)の 2 で二進数を指定して十進数に型変換する．このように python の標準機能を使って簡単にできるが、出力の書式設定で多用してきた format()とは使い方が違ったり、map()や int()での二進数文字列からの十進変換など、使い慣れない処理が多く、あまり嬉しくはない．おまけに二進数限定で、2 以外の基数の n 進数では全く役に立たず汎用性が乏しい．そこでこれはこれとして、別の一つのやり方、ついでに基数 n を指定できるようにして、n 進数を扱えるようにしたものを示しておこう．

リスト 5.5.5 十進数と n 進数の間の変換

```
def d2n(x, endian='big', n=2):
    """ n 進数に変換してリスト形式で返す """
    y = []
    q = x
    while q > 0:
        r = q %  n # remainder
        q = q // n # quotient
        y.append(r)
    return y if endian=='little' else y[::-1]

def n2d(x, endian='big', n=2):
    """ リスト形式の n 進数を 10 進数に変換 """
    y = 0
    itr = x if endian=='little' else reversed(x)
    for i, digit in enumerate(itr):
        y += int(digit) * n ** i
    return y
```

リスト 5.5.5 は関数 d2n()と関数 n2d()を示す．筆算で変換するのと同じ手順で、手数は少し多いが、使い慣れた処理だけでできる．関数 d2n()では十進数を n で割って、その商と余りを求め、余りをその桁の値とし、商を次の桁の計算に回す、という処理を、割られる数がある間繰り返す．n 進数はリスト形式で得られるから、ビッグエンディアンならばそのまま、リトルエンディアンならば桁を逆順にして返り値とする．関数 n2d()はこの逆の処理で、エンディアンを見て引数を itr にセットした後、そこから 1 桁取り出しては n の階乗しながら足し合わせていくだけだ．いずれもデフォルトで基数 n=2 としているから、これを指定しなければリスト 5.5.4 と同じ働きとなる．これもリスト 5.5.4 と併せ base_conversion.py に置いておこう．

さて下準備は出来たから、RNN の働きを確かめるべく進めていきたいが、その前にもうひと捻りしておこう．図 5.5.1 の 2 つの数は二進数としたが、もしこれらが仮に三進数だったらどうなるだろう？ 三進数だとしたら各桁の値は 0、1 だけでなく 2 もあり得るから 1001 + 0011 = 1012 となるのではないだろうか？そしてもとの 2 つの数の各桁の値が 0 か 1 しかないとすれば繰り上がりはない．繰り上がりがないとすれば、自身の桁だけで結果が決まる．そういう計算ならば、戻りパスのない普通のニューラルネットワークでも、うまく処理できるはずだ．

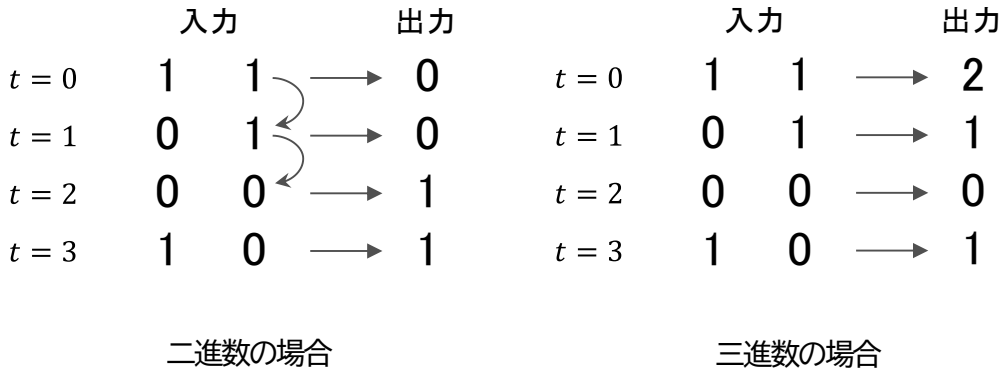

図 5.5.2 二進数と三進数の足し算の例

図 5.5.2 には二進数と三進数の足し算の例を示す．この左図は、図 5.5.1 の右図と同じだ．そして左右いずれも入力となる行列は同じ場合を示すが、左図は二進数、右図は三進数の場合だ．左の二進数の場合の答えは 0011、右の三進数の場合の答えは 2101 だ．

繰り上がりの有る場合と無い場合を両方とも用意したうえで、戻りパスのない普通のニューラルネットワークと戻りパスのある RNN とを両方とも用意すれば、ふつうのニューラルネットワークの出来ることと出来ないこと、そしてその限界を超える RNN の威力を確かめられるだろう．

そこで、これらのどの組み合わせがうまくいき、どれがうまくいかないかを確かめるべく、プログラムを作っていくことにしよう.

まずは戻りパスを持たない普通のニューラルネットワークから始めよう. 使うのは第4章での画像認識や「5.2 ニューラルネットワークでマルコフ連鎖」で実績のある入力層、中間層、出力層の3層のニューラルネットワークだ.

リスト5.5.6 n進数の足し算～その1、NN

```
import numpy as np
import NN
from base_conversion import *

# def show_result()は省略：リスト5.5.7(後述)による

# モデルを作る
model = NN.NN_m(2, 1, ml_nn=32)
model.summary()

# 学習
n_time = 8 # 時系列長==n進数の桁数
base = 3   # n進数の基数
for i in range(10001):
    # ― 乱数から0/1の並びのn_time行2列の行列を作る ―
    x  = (np.random.rand(n_time, 2) > 0.5).astype('int8')
    x[-1,:] = 0 # MSBは0にする

    # ― 行列の各列をリトルエンディアンのn進数として十進に戻し正解を用意 ―
    num1 = n2d(x[:,0], 'little', base)
    num2 = n2d(x[:,1], 'little', base)
    sumn = d2n(num1+num2, 'little', base)
    t = np.zeros((n_time, 1), dtype='int8')
    t[:len(sumn), 0] = sumn

    # ― モデルの順伝播、逆伝播、更新 ―
    y, error = model.forward(x, t)
    model.backward()
    model.update()

    # ― 経過の表示 ―
    if i%500==0:
        show_result()
```

リスト5.5.6はn進数の足し算の本体部分だ. 冒頭必要なモジュールのインポートし、後でリスト5.5.7で示す関数show_result()を置くが、紙面上は省略する. モデルを生成したら、すぐに学習を始める. そしてこのプログラムでは、学習を行うforループの中でデータを用意することとして、n_time = 8でn進数の桁数を、baseでその基数を指定する. n_timeは同時に

時系列長でもある．というのはその中は、下の桁から上の桁へと一つながりの処理だからだ．とはいえ時系列をちゃんと処理するためにはRNNが必要なはずだ．

forループの中で、まずニューラルネットワークへの入力を用意する．入力の行列xは、行数がn_time、列数が2、その値は0か1だ．これを作るのに、numpyのrandomモジュールでnp.random.rand(n_time, 2)として、値が0～1の範囲の乱数の行列を作り、それを > 0.5でTrueかFalseのブール値のランダムな並びに変え、さらにastype('int8')で各値を1か0の整数に変換している．さらにそのあとx[-1,:] = 0で末尾の行を0にしているのは、最上位の桁に1があると、2つの値を足したときに桁あふれが起きる可能性があるから、それを避けるためだ．とまれこれでニューラルネットワークへの入力は出来、その行列の列0と列1に足し合わせる2つの数が並んでいる．

正解値はここから2つの数を切出して求める．入力xは0か1が並んだ行列に他ならないから、何進数でもかまわないが、正解を求めるにあたっては、n進数の基数nが何なのかが問題となる．そこで入力の行列xの列0と列1を切り出して、それぞれをリスト5.5.5の関数n2d()で十進数に変換しnum1とnum2とする．変換の際には元の行列の並びがリトルエンディアンであり、n進数の基数nが指定するbaseの値であることとする．これは後先のようだけれども、もともと元になる行列は乱数であって0と1のランダムな並びでありさえすれば何でも良かったのだから、一向にかまわない．つまり後付けで、入力の行列に含まれる0/1の並びが、リトルエンディアンでn進数だったことにすれば良いのだ．

とまれこれで、十進数に変換された2つの数num1とnum2が得られるから、正解は十進数でnum1+num2であり、それを入力を変換したのと逆の変換でn進数にすれば良い．なお正解tを作るときに、一旦np.zeros()で大きさを決めてから、正解値を代入しているのは、tの形状をニューラルネットワークの出力と同じ行数n_timeとするためだ．

これでデータが準備できたので、あとはニューラルネットワークで、順伝播、逆伝播、更新で学習する．そしてループで500回に1回経過を表示する．この表示の関数は、説明が後回しになったが次に示す．

リスト5.5.7には二進数の足し算がうまくいったかどうかを表示する関数show_result()を示す．出力yはnp.round(y)で四捨五入してからastype('int8')で整数にする．正解値tと併せ表示のためreshape(-1)でベクトル形状にし、出力の十進での値もy10としてリスト5.5.5の関数n2d()で変換する．あとはこれらをprint()で出力する．表示のためにリトルエンディアンをビッグエンディアンにするのは[::-1]で簡単にできる．また判定は(y2 == t2).all()でy2とt2のすべての要素が一致するかを見ている．

リスト 5.5.7 足し算の結果の表示の関数

```
def show_result():
    y2 = np.round(y).astype('int8')
    y2 = y2.reshape(-1)
    t2 = t.reshape(-1)
    y10 = n2d(y2, 'little', base)
    print('n_learn:', i)
    print('error:', error)
    print('output :', y2[::-1]) # ビッグエンディアンにして表示
    print('correct:', t2[::-1]) # ビッグエンディアンにして表示
    c = '¥¥(^_^)/ :' if (y2 == t2).all() else 'orz :'
    print(c + str(num1) + ' + ' + str(num2) + ' = ' + str(y10))
    print('ー ー ー ー ー ー ー ー ー ー ー ー ー ー ー ー')
```

[補足 5.5.1]

リスト 5.5.7 の関数 show_result()は、関数内で使う変数のうち、y、t、error、num1、num2、base は外で定義されたものをそのまま使いたい．すなわちグローバル変数としたい．内部で更新してしまうと関数内だけのローカル変数になってしまうので、更新しないように注意が必要だ．

さて結果はどうなっただろう？学習の際に base に 3 以上の値を指定すれば、うまくいくのではないだろうか？ いうまでもないが、基数 n が 3 以上の n 進数では各桁の値が 0 か 1 の 2 つの数の足し算では繰り上がりは発生しないのだから．base = 3 とした場合の一例を以下に示す．

```
n_learn: 0
error: 0.8765836954116821
output : [0 0 0 0 0 0 0 0]
correct: [0 2 1 1 0 1 1 2]
orz : 742 + 1054 = 0
ー ー ー ー ー ー ー ー ー ー ー ー ー ー ー ー
n_learn: 500
error: 1.4888346413499676e-06
output : [0 1 0 1 1 0 1 1]
correct: [0 1 0 1 1 0 1 1]
¥(^_^)/ : 733 + 108 = 841
ー ー ー ー ー ー ー ー ー ー ー ー ー ー ー ー
```

以下は正解が続くので省略する．

このように n_learn: 500 でエラーは十分に小さくなり、その後は正解が続く．base の値をもっと大きな値に変えても同様でニューラルネットワークは少なくとも各桁の値が 0 か 1 であれば足し算は簡単にこなす．しかし base = 2 とした途端に次に示すように、時折まぐれで正解することがあるものの、まったく歯が立たなくなってしまう．

```
n_learn: 0
error: 0.373783677816391
output : [ 0 -1  0  0  0 -1  0  1]
correct: [0 1 0 1 0 1 0 1]
orz : 9 + 76 = -67
ー ー ー ー ー ー ー ー ー ー ー ー ー ー ー ー
n_learn: 500
error: 0.1370624601840973
output : [0 0 0 0 0 0 0 0]
correct: [0 1 1 1 0 1 0 1]
orz : 44 + 73 = 0
ー ー ー ー ー ー ー ー ー ー ー ー ー ー ー ー
n_learn: 1000
error: 0.11807379126548767
output : [1 1 1 1 1 1 1 1]
correct: [0 0 1 1 1 1 0 1]
orz : 45 + 16 = 255
ー ー ー ー ー ー ー ー ー ー ー ー ー ー ー ー
n_learn: 1500
error: 0.11630846560001373
output : [0 0 1 0 0 0 0 0]
correct: [0 0 1 1 0 0 1 0]
orz : 18 + 32 = 32
ー ー ー ー ー ー ー ー ー ー ー ー ー ー ー ー
```

以下いくら続けても、ちっとも正解しない．

さあいよいよRNNの出番だ．リスト5.5.8はリスト5.5.6のニューラルネットワークをRNNに置き換えたものだ．そしてリスト5.5.6が結果はさておき、ちゃんと動いていれば変更はわずかだ．まずは冒頭のモジュールのインポートでNNをRNNに変え、モデルを作るところで、NN.NN_m()をRNN.RNN_rf()にする．このとき中間層＝かくれ層の指定の順序がNN_mとRNN_rfで異なるから要注意だ．

続いてforループでの学習だが、ループのたびにreset_state()を実行して、過去の状態の履歴の影響を受けないようにする必要がある．また、ニューラルネットワークの入出力の形状が時系列の次元を持つので対応が必要だ．入力xや正解値tを作る際の行列は(1, n_time, 2)ないし(1, n_time, 1)の形状の配列で、リスト5.5.6で1つだったバッチと時系列の次元が分かれる．しかし変更はこれだけで、あとはリスト5.5.6と全く同じで良いはずだ．

リスト5.5.8 n進数の足し算～その2、RNN

```
import numpy as np
import RNN
from base_conversion import *

# def show_result()は省略：リスト5.5.7による

# モデルを作る
model = RNN.RNN_rf(2, 32, 1)
model.summary()

# 学習
n_time = 8 # 時系列長＝n進数の桁数
base = 2   # n進数の基数
for i in range(10001):
    model.reset_state()
    # ― 乱数から0/1の並びのn_time行2列の行列を作る ―
    x  = (np.random.rand(1,n_time,2) > 0.5).astype('int8')
    x[:,-1,:] = 0 # MSBは0にする

    # ― 行列の各列をリトルエンディアンのn進数として十進に戻し正解を用意 ―
    num1 = n2d(x[0,:,0], 'little', base)
    num2 = n2d(x[0,:,1], 'little', base)
    sumn = d2n(num1+num2, 'little', base)
    t = np.zeros((1,n_time,1), dtype='int8')
    t[0, :len(sumn), 0] = sumn

    # ― モデルの順伝播、逆伝播、更新 ―
    y, error = model.forward(x, t)
    model.backward()
    model.update(eta=0.1)

    # ― 経過の表示 ―
    if i%500==0:
        show_result()
```

リスト5.5.8のプログラムを走らせた場合の一例を以下に示す.

```
n_learn: 0
error: 0.23564758760525328
output : [0 0 1 0 0 0 0 1]
correct: [1 0 0 1 0 1 1 1]
orz : 46 + 105 = 33
ー ー ー ー ー ー ー ー ー ー ー ー ー ー ー ー
n_learn: 500
error: 0.018649621944716477
output : [1 0 1 1 1 0 1 0]
correct: [1 0 1 1 1 0 1 0]
¥(^_^)/ : 61 + 125 = 186
ー ー ー ー ー ー ー ー ー ー ー ー ー ー ー ー
n_learn: 1000
error: 0.002749014458411703
output : [0 1 1 1 0 1 1 0]
correct: [0 1 1 1 0 1 1 0]
¥(^_^)/ : 100 + 18 = 118
ー ー ー ー ー ー ー ー ー ー ー ー ー ー ー ー
n_learn: 1500
error: 0.0016137974467130222
output : [0 1 1 1 1 1 1 1]
correct: [0 1 1 1 1 1 1 1]
¥(^_^)/ : 79 + 48 = 127
ー ー ー ー ー ー ー ー ー ー ー ー ー ー ー ー
```

この後は正解が続くのでやはり省略する. 戻りパスを設け、時系列で展開して逐次的に処理するRNNの効果が確認できたのではないだろうか？ ひるがえって、繰り上がりがない、すなわち過去の履歴に依存しないような事象に対しては、戻りパスのない普通のニューラルネットワークは学習が早く、エラーも非常に小さくなり、RNNよりも優れているとさえいえる.
要は扱う問題の素性と、これに対するモデルが何なのかとの組み合わせが問題なのだ.

5.5.3 ずっと続くサインカーブ

過去の状態の履歴に従うとすれば、サインカーブを幾波か経験した後は、やはりサインカーブを続けるのではないだろうか？ 過去の成功体験から逃れられない人の性を見るようではあるけれど、どんな波もサインカーブやコサインカーブの組み合わせで表されることはフーリエ級数として知られており、ずっと続くサインカーブも、万物の根源的な素性の一つと言えるだろう.

これを試すプログラムを作っていこう．まずは助走を与えたら続きを生成する機能を関数 generate()として作る.

リスト 5.5.9 サインカーブ～その 1

```
def generate(func, seed, length=200):
    """ seed に続いて一つずつ生成していく """
    gen_data = np.array(seed)
    T, l = gen_data.shape
    for j in range(length-1):
        x = gen_data[j]
        y = func(x)
        if j+1 < T: # seed の範囲は書込まない
            continue
        next_one = y.reshape(1, l)
        gen_data = np.concatenate((gen_data, next_one))
    return gen_data
```

関数 generate()の引数は、その生成に使う関数 func、出だしを与える seed、生成する長さを指定する length だ．gen_data = np.array(seed)で初期値を seed そのものとし、形状からその時系列長 T と入力の大きさ l を得る.

for ループで 1 時刻ずつ seed の続きを生成する．RNN にはデータを 1 時刻ずつ入力していくが、seed の範囲は gen_data をさわらず、順伝播だけを行う．この順伝播で履歴を積み上げながら seed の末尾に至り、そこからは RNN の出力 y を numpy の concatenate()で gen_data に加えて次々と紡いでいく. seed と RNN の入出力の形状の違いは適宜 numpy の reshape()で吸収する. 引数 length で指定した長さになったら gen_data を返り値として終了する.

続いて結果を図示する関数 show_result()を作ろう.

リスト 5.5.10 サインカーブ～その 2

```
def show_result(seed, length):
    """ seed の続きを長さ length まで図示 """
    model.reset_state()
    forward = lambda x : model.forward(x.reshape(1, 1, 1))
    predicted = generate(forward, seed, length)
    plt.plot(data.tolist(), label="Correct")
    plt.plot(predicted.tolist(), label="Predicted", linestyle="dashed")
    plt.plot(seed, label='Seed')
    plt.legend()
    plt.show()
```

リスト 5.5.10 に関数 show_result()を示す．この関数はサインカーブのプログラムの中に記述する前提で model や data はグローバルな扱い、すなわちこの関数の外で定義したものが、そのまま使われる前提だ．引数 seed で出だしを指定し、引数 length で指定した長さまでリスト 5.5.9 の関数 generate()で生成し、それを matplotlib で図示する．このとき関数 generate()に渡す引数 func は、lambda 式の model.forwarda()で定義するが、その時 reshape(1, 1, 1)で RNN の入力に適合するように成形する．matplotlib で data と、predicted と seed を順に plot()していく．これにより学習したデータに対して、RNN が生成するものがどうなるのかを、出だしとして与えたデータがどうかと併せて視覚的に確認できるようにする．

最後に本体部分をリスト 5.5.11 に示す．冒頭で必要なモジュールをインポートし、続いてリスト 5.5.9 とリスト 5.5.10 に示した関数 generate()と show_result()を置くが、紙面上では省略する．

学習に使う訓練データは $-2\pi \sim 2\pi$ の範囲の 50 個の並びをサインカーブにしたものだ．

続いてモデルを生成するが、入出力ともサイズは 1 でその値が意味を持つ．いっぽう戻りパスのあるかくれ層はニューロン数を 20 としているが、この数はこの限りではない．学習は 2 重の for ループで行うが、内側のループで訓練データを使い終わって 1 エポック進んだら、それで外側で 1 回進む．すなわち外側のループがエポックだ．内側のループではデータを time_size で指定する長さずつ順に切り出して、その末尾まで学習していく．切り出すとき RNN の入力 x と正解 t は、もとの訓練データから、1 時刻ずらしている．これは言うまでもなく、RNN で学習したいのが現在に対する 1 時刻先の未来だからだ．x も t も RNN に適合するように形状を整えるのも忘れてはならない．順伝播、逆伝播、更新で学習するのはいつも通りだ．

ここで説明は後先になるが、内側のループの range()の増分は 10 としているが、これは切り出しの間隔となる．これは時系列長 time_size = 20 との組み合わせで、いろいろ試してみることをお勧めする．

最後の結果の図示では、seed の長さを変えながら結果を見るようにしている．

リスト 5.5.11 サインカーブ～その 3

```
import numpy as np
import matplotlib.pyplot as plt
import RNN
import common_function as cf

# def generate()は省略：リスト 5.5.9 による

# def show_result()は省略：リスト 5.5.10 による

# ― 訓練データ ―
data = np.linspace(-2*np.pi, 2*np.pi)    # 元ネタ
data = np.sin(data)                      # サインカーブ

# ― モデルの生成 ―
model = RNN.RNN_rf(1, 20, 1)
model.summary()

# ― 学習 ―
time_size = 20  # 時系列長
epochs = 6001
loss_record = []
for i in range(epochs):
    model.reset_state()
    for j in range(0, len(data)-time_size-1, 10):
        x = data[j:j+time_size].reshape(1, -1, 1)
        t = data[j+1:j+time_size+1].reshape(1, -1, 1)
        y, l = model.forward(x, t)
        model.backward()
        model.update(eta=0.05)

    loss_record.append(float(l))
    if i%100 == 0:
        print('Epoch:{:4d}/{:4d} Error:{:7.4f}'.format(i+1, epochs, float(l)))

# ― 結果の図示 ―
cf.graph_for_error(loss_record)
for t in range(10):
    seed = data[0:t+1].reshape(-1, 1)
    show_result(seed, len(data)*2)
```

それでは結果を見ていこう．エラーのグラフを図 5.5.3 に示すが、非常に素早くエラーが減少する．

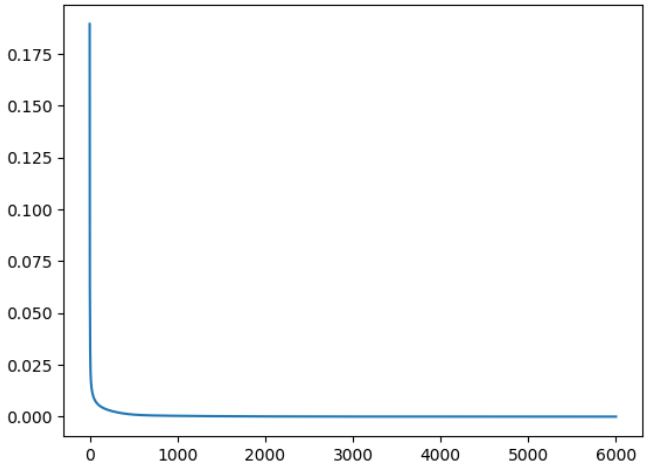

図 5.5.3 サインカーブの学習経過

図 5.5.4 には、関数 show_result()で seed を与えてその続きを描かせたものを、元のデータとともに示す。あまりにうまく重なっているため分かりずらいが、グラフ x 軸の真ん中あたりまでが元のデータで、右端まで至るのは関数 generate()で len(data)*2 として元のデータの 2 倍まで外挿した結果だ．これは*2 の 2 を大きくして実際にやってみてほしいが、どれだけ長くしても、サインカーブを描き続ける様子が確認できるはずだ．

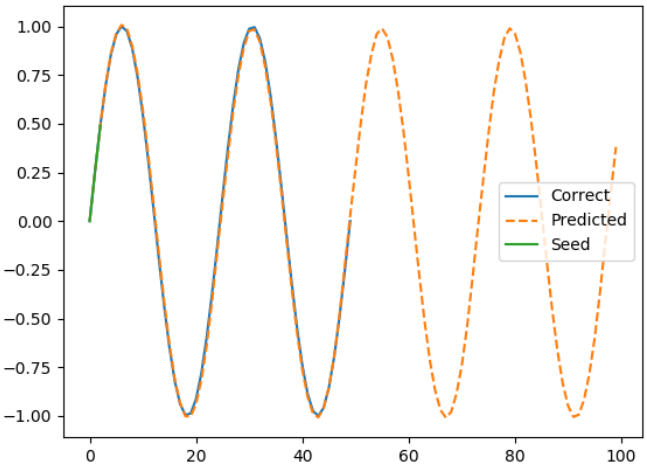

図 5.5.4 ずっと続くサインカーブ

学習のさせ方にもよるが、RNN が少なくとも単純な繰り返しに対して、非常に強力なことが確認できる．一方その学習のさせ方だが、先に述べたように、時系列長や増分を変えたり、あるいは学習率、そして実は学習データの起点や区間、などなど、多々ある条件によっては、うまくいったりいかなかったりする．

ここでちょっと視点を変えて「5.5.2 二進数の足し算」でやったように、戻りパスの効果を確認しよう．戻りパスの効果を確認するためには、リスト 5.5.6 でやったように、はじめから戻りパスのない普通のニューラルネットワークを使う方法もあるが、ちょっと違うやり方をしていこう．
まずリスト 5.5.11 を走らせて、サインカーブを RNN がうまく続けてくれることを確認しよう．
そしてその後に、
```
>>> model.rnn_layer
```
と入力してみよう．すると、
```
<Neuron.RnnLayer object at 0x00000216467AA848>
```
などと返ってくるはずだ．これはプログラムの中でインスタンス化した RNN の rnn_layer がオブジェクトとしてアクセスできることを示す．そこでさらに、
```
>>> model.rnn_layer.v.shape
```
とすれば、リスト 5.5.11 の通りにしていれば、
```
(20, 20)
```
と返ってくるはずで、model の rnn_layer の v が 20×20 の大きさの行列だと確認できる．リスト 5.4.1 の RNN_Unit の forward()メソッドに定義される通り、その v は戻りパスに対する重みだ．それを model.rnn_layer.v とすればアクセスできる．
そこでこの戻りパスの重み v の値を 0 に固定して、実質的に RNN_Unit の戻りパスからの影響をなくしてしまうことで、その戻りパスの効果を確認することにしよう．
そうと決まればやることは簡単だ．リスト 5.5.11 の model.update()のあとに、インデントに注意して次の 1 行を書き加えるだけだ．
```
model.rnn_layer.v[...] = 0
```
ここで[...]の 3 点ドットはリスト 5.2.2 にも出てきたが、Ellipsis と呼ばれ省略記号だ．これを使うと行列の要素を次元を省略して指定、つまり全要素を指定することになる．
それではこれを書き加えて、戻りパスの効果を無くした場合の実行結果を見てみよう．
図 5.5.5 に示すように学習経過をみると、エラーがある値に達した後はそれ以上小さくならない．また、図 5.5.6 に示すように seed を超えたところからすぐにサインカーブから外れてしまう．これは seed をもっと長くしても同じだ．

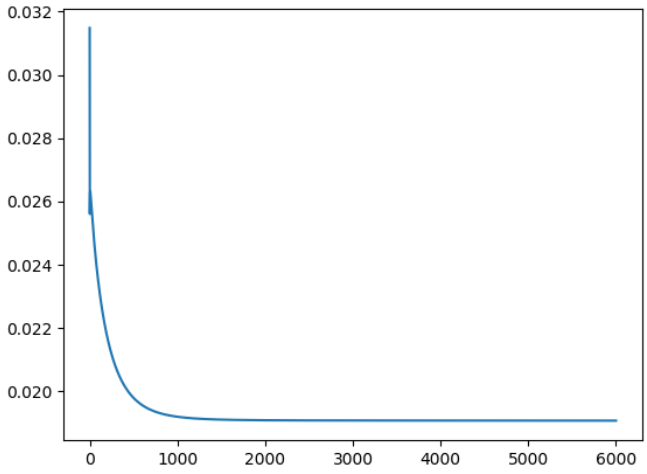

図 5.5.5 戻りパスを無くした場合の学習経過

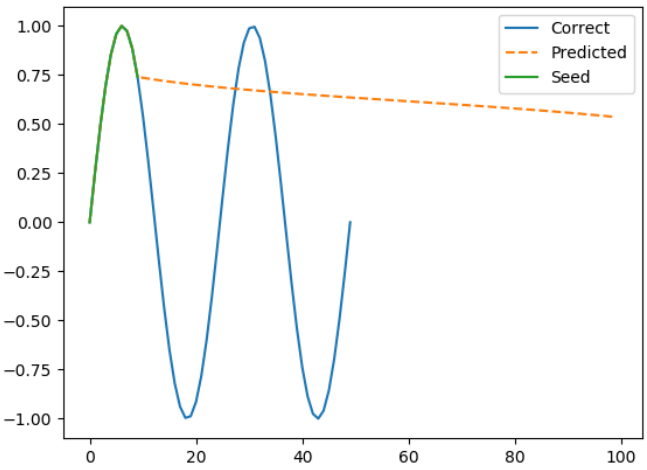

図 5.5.6 続かないサインカーブ

戻りパスがあってはじめてサインカーブを描き続けることができる、ということが確認できたのではないだろうか？

これはすなわち、過去の履歴の先に未来がある、ということに他ならず、それをモデルとして実現するのが RNN だということなのだ.

5.5.4 いろは歌

RNN を使って二進数に足し算、サインカーブとやってきたが、ここで本章ではじめに扱った「いろは歌」を取り上げよう．リスト 5.5.12 に「いろは歌」を示すが、文字列 text はリスト 5.1.2 と同じだ．

リスト 5.5.12 いろは歌～その 1

```
import numpy as np
import common_function as cf


text = ('いろはにほへとちりぬるを'
        'わかよたれそつねならむ'
        'うゐのおくやまけふこえて'
        'あさきゆめみしゑひもせす')

# corpus と辞書を作る
corpus, char_to_id, id_to_char = cf.preprocess(text)
vocab_size = len(char_to_id)
corpus = np.array(corpus)
# 1 つずつずらして切出し、前を入力とし、後を正解値にする
X = corpus[:-1]
C = corpus[1: ]
X = np.eye(vocab_size, dtype='bool')[X]
C = np.eye(vocab_size, dtype='bool')[C]
X = X.reshape(1, -1, vocab_size)
C = C.reshape(1, -1, vocab_size)
print(X.shape, C.shape)
```

この後の処理はリスト 5.2.4 と同様で、リスト 5.2.1 の関数 preprocess()で corpus、char_to_id、id_to_char を得、vocab_size を求めておいて、ニューラルネットワークへの入力と正解の準備をする．1 つずつずらすのも one hot にするのもリスト 5.2.4 同様だが、RNN への入力なので、時系列の次元に文字の並びを割り当てる．

次は結果を見るための関数だ．サインカーブの場合同様に、ここでも出だしを与えて続きを書かせてみたい．そのための関数を用意するのだが、「5.2.3 ニューラルネットワークによる文章の作成」で述べたように、確率的に次を選んで文章を綴っていくようにしよう．「いろは歌」では、現在の文字に対して次の文字が一意に決まるから、確率的である必要はない．しかしはじめからそうしておくことで、小説などの一般的な場合にも使えるようにしておこう．これをリスト 5.5.13 に関数 generate_text()として示す．なおニューラルネットワークの出力から、確率的に一つを選ぶのは、リスト 5.2.3 で関数 select_category()として定義したものを再利用する．

リスト 5.5.13 いろは歌～その 2 文字列生成

```
def generate_text(func, x_to_id, id_to_x, length=100,
                  seed=None, stop=None, print_text=True, end='',
                  stocastic=True, beta=2):
    """ 文字列の生成 x:文字または語 x_chain:その連なり key:次の索引キー """
    x_chain = ""
    vocab_size = len(x_to_id)
    # ― seed 無指定なら辞書からランダムに選ぶ ―
    if seed is None:
        seed = random.choices(list(x_to_id))

    for j in range(length):
        # ― 書出しは seed から、その後は func 出力から ―
        if j < len(seed): # seed の範囲
            x = seed[j]
            key = x_to_id[x]
        else:             # seed の範囲を超えた
            key = select_category(y, stocastic, beta=beta)
            x = id_to_x[key]
        # ― 綴る ―
        x_chain += x
        if print_text:
            print(x, end=end)
        if x==stop or len(x_chain)>length:
            break
        # ― 次に備える ―
        y = func(key)     # seed の範囲を含めて順伝播

    return x_chain
```

リスト 5.5.13 で関数 generate_text()は、リスト 5.2.2 のそれとほとんど同じだ．だから違う点をかいつまんで説明しよう．func の入力の key の長さは、func として RNN を想定するのだから自由にできる．そこで簡単に長さは 1 とする．それにより、if～else 節で文字または語の識別子を得たら、それをそのまま key とし、key の加工は不要となる．いっぽう y = func(key)によってニューラルネットワークの出力を得る操作は、RNN に seed から始まる履歴を与えるために seed の範囲を含めて毎回行う．そして for ループ内の最後に行った結果の y が次回の key と x のもととなる．この操作は、リスト 5.2.2 で for ループ内の最後に key を加工して次回に備えたことに対応するが、現在までの履歴から次を予測する RNN の動作そのものだ．このようにリスト 5.2.2 のいやらしい部分がなくなって、かえって分かり易いのではないだろうか？ なお関数 select_category()を呼出しているが、これはリスト 5.2.3 をそのまま使えば良い．

では「いろは歌」の本体部分だ．リスト 5.5.12 とリスト 5.2.3 とリスト 5.5.13 をコピーして貼り付ければ、データと必要な関数は用意できる．その部分を省略したものをリスト 5.5.14 に示す．

リスト 5.5.14 いろは歌～その 3

```
import numpy as np
import common_function as cf
import RNN

# text 以下は省略：リスト 5.5.12 による

# def select_category()は省略：リスト 5.2.3 による

# def generate_text()は省略：リスト 5.5.13 による

# ― モデルの生成 ―
model = RNN.RNN_rf(vocab_size, 32, vocab_size,
                   activate='Softmax',
                   loss='CrossEntropyError')
model.summary()

# ― 学習 ―
mesurement = cf.Mesurement(model)
print('学習を開始します')
for i in range(301):
    model.reset_state()
    model.forward(X, C)
    model.backward()
    model.update(eta=0.1)
    if i%10==0:
        model.reset_state()
        l, acc = mesurement(X, C)
        print('epoch{:6d} | error{:6.3f} accuracy{:6.3f}'¥
              .format(i, float(l), acc))

# グラフの描画と最終確認
errors, accuracy = mesurement.progress()
cf.graph_for_error(errors, accuracy, label=('error','accuracy'))
model.reset_state()
def func(x):
    x = np.eye(vocab_size)[x]
    y = model.forward(x.reshape(1, 1, -1))
    return y
generate_text(func, char_to_id, id_to_char, 47, 'いろは')
```

リスト 5.5.14 でやることは、モデルの生成からだが、入力、正解値とも one hot 形式で語彙数 vocab_size の幅を持ち、出力も同じ形状だから、これを入出力の大きさに指定し、戻りパスのあるかくれ層のニューロン数は任意だがここでは 32 とした．文字を選ぶカテゴリ分類問題なのだから、出力層の活性化関数はソフトマックス、損失関数はクロスエントロピーエラーを指定する．この事情はリスト 5.2.5 でモデル生成した際と同様だ．

これらの指定が反映される経路は少々ややこしいので確認しておく．まず損失関数だが、RNN_Base のコントラクタの引数 kwargs に loss='CrossEntropyError'が渡されて、リスト 5.5.1 にある通りに損失関数として設定される．いっぽう活性化関数は出力層に関するものだが、これはリスト 5.5.3 に示すように RNN_rf のコントラクタの引数 kwargs を経由して、NeuronLayer のコントラクタのコントラクタに渡る．そしてその基底クラスであるリスト 4.1.1 に示す BaseLayer のコントラクタの引数となって activate='Softmax'が渡されて、リスト 4.1.1 にある通りに、その層、ここでは出力層の活性化関数として設定される．なおリスト 5.5.3 に示すように RnnLayer にも同じものが引数として渡されるが、リスト 5.4.4 にあるように、'activate'というキーワードは使われず影響しない．
ともかく model.summary()は確認しておきたい．

つづいて学習にはいるが、入力と正解のデータは用意できており、これらの大きさも、さほどでもないから、ここはいっぺんに全部使うようにしてしまおう．学習の経過の記録を兼ねて、第 4 章のリスト 4.7.2 で用意した Mesurement クラスを使うが、これもリスト 5.2.5 と同じだ．経過表示はリスト 5.2.5 よりも少し簡単にした．ともかく、順伝播、逆伝播、更新を繰り返して学習する．RNN なので履歴を初期化する reset_state()は必要だが、それ以外とくに変わった事は必要ない．
学習が終わると、経過をグラフ表示し、最後に関数 generate_text()で出だしに'いろは'を指定して 47 文字を生成する．関数 generate_text()に引数 func として与える関数は、入力を one_hot にしたり形状を加工したりするように定義するが、これもリスト 5.2.5 同様だ．
これを実行した際の学習曲線の一例を図 5.5.7 に示そう．

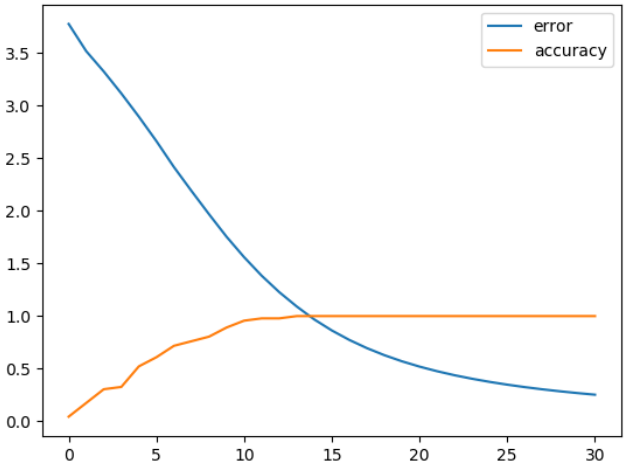

図 5.5.7 いろは歌の学習経過

きれいに学習して、最後に実行する関数 generate_text()によって「いろは歌」がそのまま出力されるはずだ.

5.5.5 少し整理しておこう

RNN まわりで生成に関して少し整理しておこう.

リスト 5.5.9 の関数 generate()は、出だしを与えて、その続きのサインカーブを描くために用意し、リスト 5.5.11 の中に取り込んだ. しかしこれはサインカーブに限らず、もっと広く一般に、何か出だしのデータを与えて、その続きのデータを生成する目的に使える. そこで、これは、RNN でデータからデータを生成するのに使うために、リスト 5.5.1、リスト 5.5.2、リスト 5.5.3 と一緒に RNN.py に入れておこう.

またリスト 5.5.13 の関数 generate_text()は、リスト 5.5.14 に取り込んで、いろは歌で'いろは'などの出だしを与えて、その続きを書かせるのに使った. これも広く一般に出だしを与えて、その続きの文章を綴るのに使えるから、RNN.py に入れておこう. そしてリスト 5.2.3 の関数 select_category()も、関数 generate_text()の中で使うから RNN.py にいれておこう.

これで RNN を使ってデータからデータを生成するには関数 generate()、RNN で文章を綴るには関数 generate_text()と使い分ければ良いだろう.

なお関数 generate()を使って文章が綴れないわけではないので、一応その例もリスト 5.5.15 に示しておく. なおこの例では関数 texts_from_texts()はプログラム本体中に置かれ、seed と length 以外は関数内からプログラム本体に定義された model、char_to_id、id_to_char、vocab_size がグローバル変数としてアクセスできるものとする. そのほか細かい説明は省略するがリスト 5.5.14 で、リスト 5.2.3 とリスト 5.5.13 のかわりに、リスト 5.5.9 の関数 generate()とリスト 5.5.15 の関数 texts_from_texts()を置く. そして、プログラムの末尾の部分、model.reset_state()以下を texts_from_texts('いろは', 47)などとすればもとのリスト 5.5.14 と同じように動くはずだ.

リスト 5.5.15 関数 generate()を使った文章生成

```
def texts_from_texts(seed, length=1):
    seedx = []
    for c in seed:
        s = char_to_id[c]
        seedx.append(s)
    seedx = np.eye(vocab_size)[seedx]
    forward = lambda x: model.forward(x.reshape(1, -1, vocab_size))
    model.reset_state()
    created_data = generate(forward, seedx, length)
    created_data = np.argmax(created_data, axis=1)
    for d in created_data:
        print(id_to_char[int(d)], end='')
    print()
```

5.6 Embedding

Embedding と呼ばれる操作をその入り口で行うようにすれば、識別子 id をそのまま入力できるようになり、ニューラルネットワークを上手に文章を扱う機械に進化させられる.

5.6.1 one hot の問題とその対応

前節で、いろは歌をニューラルネットワークで扱う際には one hot 形式に変換した. さかのぼって「5.2.5 ニューラルネットワークでマルコフ連鎖」でもやはり、小説を識別子 id の並びにした後に、その id を one hot 形式に変換してから、ニューラルネットワークに入力した.
このように文章は、その文字や語を識別子に変換し、その識別子をさらに one hot 形式に変換してから、ニューラルネットワークで扱う場合に入力する. 文字や語を識別子という数値に変換するのは数値計算の対象とするためだが、さらに one hot にするのは、値の大きさに意味のない識別子の値が、入力×重みの計算で意味を持ってしまわないようにするためだ.
「いろは歌」はさておき、「羅生門」では語彙数が 990 となっている. 語彙数はそのまま one hot のベクトル長となる. しかしこのベクトルは、まさしく 1 つの要素が 1 で他はすべて 0 のベクトルであって非常に簡単ではあるけれども無駄が多い. そこでその一つだけが 1 で他はすべて 0 ということをやめて、もっと上手に何らかの識別子に還元できるベクトルで表すことが考えられる.
ごくごく簡単な例で考えてみよう. 仮に語彙数が 2 つだけだったらどうだろう？そうするとこれを識別する識別子は 0 と 1 となる. これに対応する one hot 形式の 2 つの値は、ベクトル(1　0)とベクトル(0　1)だ. そしてこれを平面上であらわせば図 5.6.1 の左のようになって、いずれも長さが 1 だ.

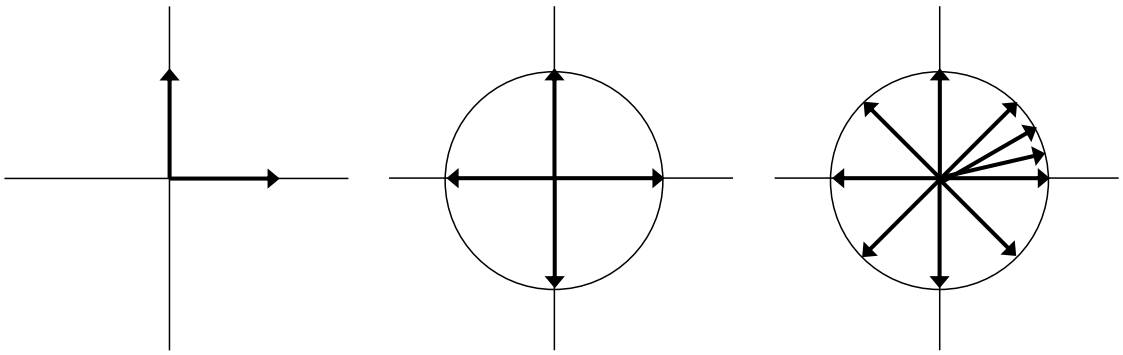

図 5.6.1

ここで仮に語彙数が 4 に増えたらどうだろうか？ 識別子の値 0、1、2、3 に対して one hot ならばベクトルの要素数を 4 に増やして、$(1 \quad 0 \quad 0 \quad 0)$、$(0 \quad 1 \quad 0 \quad 0)$、$(0 \quad 0 \quad 1 \quad 0)$、$(0 \quad 0 \quad 0 \quad 1)$となる．しかしもとの要素数 2 のベクトルで図 5.6.1 の中央の図のように、$(-1 \quad 0)$と$(0 \quad -1)$の 2 つを追加して、4 通りを表現することもできる．そして識別子 0、1、2、3 に対して、$(1 \quad 0)$、$(0 \quad 1)$、$(-1 \quad 0)$、$(0 \quad -1)$を対応させれば良さそうだ．ニューラルネットワークの入力の観点でも、入力は重みと掛け合わせるのだから、'-1'があるのは全く問題ない．

さて図 5.6.1 に戻って、これら 4 つのベクトルを眺めると、長さ 1 のベクトルを $\pi/2$ ずつ回転させて得られる 4 つのベクトルとなっている．識別子はカテゴリ変数、ベクトルはダミー変数に他ならないから、それぞれ $category$、$dummy$ とすれば、両者の関係は次式により表せる．

$$dummy = (\cos\theta \quad \sin\theta)$$
$$\theta = \frac{\pi}{2} * category \tag{5.6.1}$$

複素平面に置き換えれば、オイラーの公式により、

$$dummy = \cos\theta + i\sin\theta = e^{-i\theta} \tag{5.6.2}$$

オイラーの公式まで登場させて、ややこしいことを言っているようだが、それをうんぬんするのが趣旨ではない．そうではなくて、回転の角度に置き換えることでカテゴリ変数をダミー変数に変換しているのだということを、イメージして欲しい．

ともかく、平面上で右向き、上向き、左向き、下向きのあわせて 4 つの矢印であらわされる 4 つのベクトルで 4 つのカテゴリなのだ．

それではさらに語彙数が多かったらどうすれば良いだろうか？もうすでに気づいていることとは思うが、刻みの角度を $\pi/2$ ではなく、図 5.6.1 の右図に示すようにもっと細かくすれば、いくつでもあらわせる．実際に語彙数 $vocabulary$ に対し、

$$\theta = \frac{2\pi}{vocabulary} * category \tag{5.6.3}$$

とすれば、理論上は語彙数の制限なく対応できるはずだ．さっそく試してみよう．そこで次のプログラムを用意する．

リスト 5.6.1 回転角によるカテゴリ変数からダミー変数への変換

```
def category_to_dummy(X, vocab_size):
    notch = 2 * np.pi / vocab_size
    Y = []
    for x in X:
        y1 = np.cos(x*notch)
        y2 = np.sin(x*notch)
        Y.append([y1, y2])
    return np.array(Y)
```

リスト 5.6.1 で用意する関数 category_to_dummy()は、カテゴリ変数を引数 X で、また語彙数を引数 vocab_size で与える．引数 X はコーパスなどから切出したリストと想定し、返り値もそれに対応して行列とする．notch として求めるのは式 5.6.3 の右辺の第 1 項で、for ループの中で X から取り出した x をかけて式 5.6.3 のθとなる．後はこのサインとコサインを求めて、それを並べたベクトルを作り、返り値に積上げる．とまれ式 5.6.1～式 5.6.3 そのものだ．
さっそく、これを「いろは歌」で確かめよう．リスト 5.5.12～リスト 5.5.14 をもとにすればわずかな変更だ．

リスト 5.6.2 いろは歌で実験～その 1

```
import numpy as np
import common_function as cf
import RNN

text = ('いろはにほへとちりぬるを'
        'わかよたれそつねならむ'
        'うゐのおくやまけふこえて'
        'あさきゆめみしゑひもせす')

# def category_to_dummy()は省略：リスト 5.7.1 による

# corpus と辞書を作る
corpus, char_to_id, id_to_char = cf.preprocess(text)
vocab_size = len(char_to_id)
corpus = np.array(corpus)
# 1 つずつずらして切出し、前を入力とし、後を正解値にする
X = corpus[:-1]
C = corpus[1: ]
X = category_to_dummy(X, vocab_size)
C = np.eye(vocab_size, dtype='bool')[C]
X = X.reshape(1, *X.shape)
C = C.reshape(1, -1, vocab_size)
print(X.shape, C.shape)

# def select_category()は省略：リスト 5.2.3 による

# def generate_text()は省略：リスト 5.5.13 による
```

リスト 5.6.2 に示す前半では、リスト 5.6.1 の関数 category_to_dummy()を書き加えることと、入力 X を one hot にするかわりに、その関数 category_to_dummy()による変換に置き換えることだ.後者に関連して、データ 1 つあたりのベクトル長が vocab_size ではなくなるから、reshape()の際の指定を変える必要がある. (1, *X.shape)としているのがそれだが、もとの X の形状に対して一番上に 1 次元を加えて、RNN に入力する(バッチサイズ, 時系列, ベクトル長)の並びにする.
続いて後半だ.

リスト 5.6.3 いろは歌で実験～その 2

```
# ― モデルの生成 ―
model = RNN.RNN_rf(2, 32, vocab_size,
                   activate='Softmax',
                   loss='CrossEntropyError',
                   )
model.summary()

# ― 学習 ―
mesurement = cf.Mesurement(model)
print('学習を開始します')
for i in range(1001):
    model.reset_state()
    model.forward(X, C)
    model.backward()
    model.update(eta=0.1)# ,g_clip=1)
    if i%10==0:
        model.reset_state()
        l, acc = mesurement(X, C)
        print('epoch{:6d} | error{:6.3f} accuracy{:6.3f}'¥
              .format(i, float(l), acc))

# グラフの描画と最終確認
errors, accuracy = mesurement.progress()
cf.graph_for_error(errors, accuracy, label=('error','accuracy'))
model.reset_state()
def func(x):
    x = category_to_dummy([x], vocab_size)
    y = model.forward(x.reshape(1, 1, -1))
    return y
generate_text(func, char_to_id, id_to_char, 47, 'いろは')
```

リスト 5.6.3 に示す後半も変更箇所は少ない. まずはモデル生成時に入力の幅は vocab_size ではなく 2 だ. いっぽう学習回数は 1000 回程度に増やす必要があるようだ. あともう一つの変更は、関数 generate_text()で結果を確かめる際に、その引数として渡す関数 func()の定義の中で、x を関数 category_to_dummy()で変換することだ.

では結果を見てみよう.

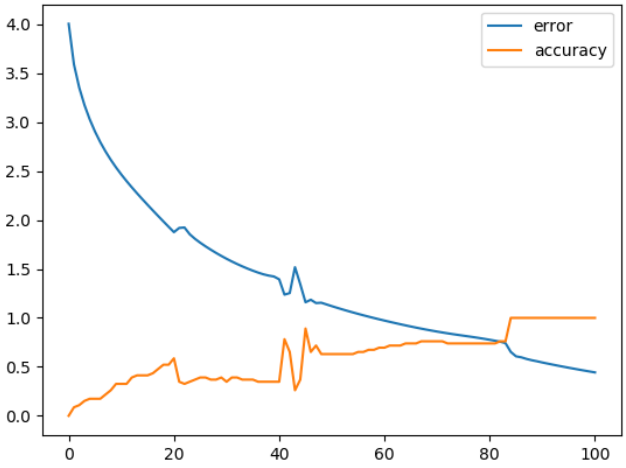

図 5.7.2 いろは歌の実験の学習過程

図 5.6.2 には学習曲線を示すが、見たとおり滑らかではない. そしてやるたびに違う. コメントアウトしてあるが、model.update()のところで eta と併せて g_clip = 1 などと、勾配クリッピングを指定すれば凹凸は少なくなるものの、必ずしも結果は伴わない. それでも勾配クリッピングはリスト 5.3.5 で実装したから、この効果を確かめるにはちょうど良い. とまれ学習がうまくいけば、いろは歌をそのまま出力して終わる.

このように 47 文字あって one hot では長いベクトルにしなければならなかったものが、わずか 2 元のベクトルに置き換えられることが実証できた.

いっぽう学習が難しくなっていて、いろは歌はさておき、羅生門の語彙数 990 や、もっと長い小説で数千におよぶ語彙数では、計算精度の問題もあって困難が予想される. そんなわけで、もっとうまい方法はないかを考えなければならないだろう.

5.6.2 Embeddingの考え方

前項リスト5.6.1～リスト5.6.3では回転角による変換を用いて「いろは歌」を取り上げたが、少し高いところから眺めて図にすると次のようになるのではないだろうか？

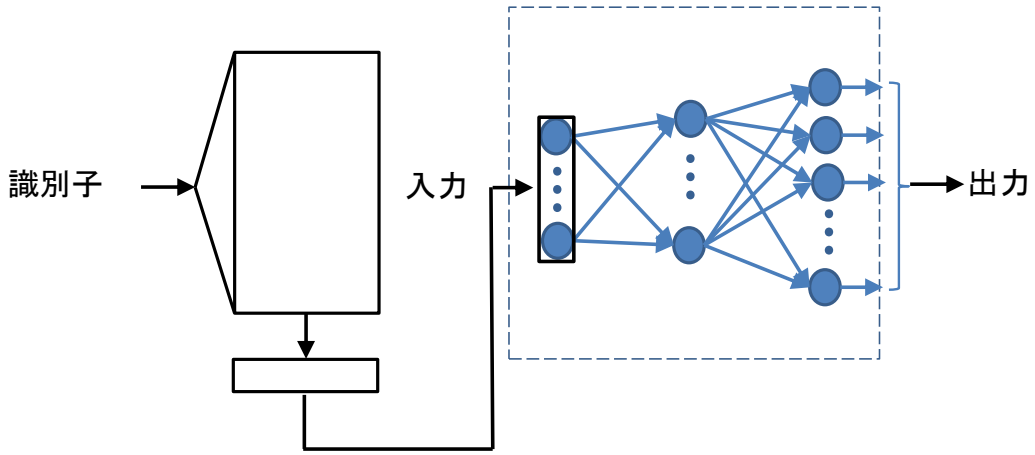

図 5.6.3 ニューラルネットワークとその入力

図5.6.3で左の四角は識別子のカテゴリ変数をダミー変数に変換するもの．そして右の点線で囲ったものはニューラルネットワークだ．

少し遡って同じく「いろは歌」を扱ったリスト5.5.12～リスト5.5.14では、カテゴリ変数からダミー変数への変換を事前にまとめて行っていて、変換の単位と時点はことなるものの、やはり系全体としては、この図のようになっていると捉えられるだろう．さらに小説「羅生門」を扱ったリスト5.2.4～リスト5.2.5においても、右の点線内のニューラルネットワークがマルコフ連鎖を担ったという点は違うが、系全体として、この図の範疇だろう．

いずれにせよ点線の中のニューラルネットワークは、図の 3 層の例でいうならば、左から入力層、隠れ層（中間層）、出力層であって、入力層は、入力がそのまま入るだけで実際にはニューロンは無く、この図では左の四角から得られたベクトルがそのまま入力層に入る様子を示している．隠れ層は図のように1層の場合もあれば、もっと層数が多い場合もあり、そしてそれぞれ戻りパスがある場合もない場合もある．出力層はその出力一つが一つのカテゴリに対応して、カテゴリの数だけ、すなわち語彙数だけある．

とまれ問題は左の四角だ．リスト5.2.4、リスト5.5.12では、corpusから切出したデータXに含まれる識別子xをnumpyのeye()を使ってone hot形式に変換する．このとき左の四角は、語彙数の大きさの単位行列だ．変換はx = np.eye(vocab_size)[x]により、その単位行列の行を識別子 x で選択することによって行っている．いずれにしても単位行列を使った変換だ．一応確認しておこう．リスト5.2.4～リスト5.2.5を走らせた後で、次のようにnp.eye()によって、

```
>>> np.eye(vocab_size)
array([[1., 0., 0., ..., 0., 0., 0.],
       [0., 1., 0., ..., 0., 0., 0.],
       [0., 0., 1., ..., 0., 0., 0.],
       ...,
       [0., 0., 0., ..., 1., 0., 0.],
       [0., 0., 0., ..., 0., 1., 0.],
       [0., 0., 0., ..., 0., 0., 1.]])
```

と単位行列が作られていることが確認できる. この単位行列の行を識別子で選んで one hot 形式に変換したのであり、図 5.6.3 の左の四角には、この単位行列が置かれていたことになる.

回転角によるカテゴリ変数からダミー変数への変換についても見ておこう. 変換はリスト 5.6.1 によって行った. これもリスト 5.6.1～リスト 5.6.3 を走らせてから、range(vocab_size) で 0 から vocab_size に至る識別子の並びを作って次のようにリスト 5.6.1 の関数 category_to_dummy()によって、

```
>>> category_to_dummy(range(vocab_size), vocab_size)
array([[ 1.        ,  0.        ],
       [ 0.99107749,  0.13328696],
       [ 0.96446918,  0.2641954 ],
       [ 0.92064989,  0.39038928],
       [ 0.86040158,  0.50961664],
       （途中略）
       [ 0.99107749, -0.13328696]])
```

このように 47 行 2 列の行列が作られる. リスト 5.6.1 では都度変換する前提なので、この 47 行 2 列の行列は実際には作られないが、この変換はこの 47 行 2 列の行列を作っておいて、そこから識別子の指す行を抜き出すのと等価だ. すなわち図 5.6.3 の左の四角に、この 47 行 2 列の行列を配するのと同じだ.

見てきたこれらの場合には、カテゴリ変数をダミー変数に変換するのに、単位行列を使ったり、平面上のベクトルの回転を使ったりしている. しかしともかく、識別子を何らかのベクト

ルに変換して、それがニューラルネットワークに入力するのに具合が良いものならば、それで良いということにほかならない. すなわち、そのベクトルをどう作るかは問題ではないはずなのだ. 平面のベクトルは要素が 2 つだが、それは作り方の問題であるにすぎないから、一般的には「適当」な長さのベクトルを用意するべきだろう. ここでいう「適当」は「いいかげん」ということではなくて、日本語に遊ばれているようだが、「良い」＋「加減」だ. つまり about ではあるけれども、ちょうど良い範囲であるべきなのだ. もっとも何がちょうど良いのかは、やってみないとわからない.
ともかくそういうベクトルで、入力がニューラルネットワークに入り、出力がうまい値となれば良いのだ. 逆に言うならば、うまい出力を得られるような入力となるベクトルが作れれば、良いのだ.

うまくいったと仮定しよう. そのときベクトルはどうなっているだろうか？識別子が違えば対応するベクトルは違っているはずだ. 同じだとすれば出力は同じになってしまうからだ. つまり識別子ごとに異なるベクトルとなっていて、それを入力したニューラルネットワークの出力は目的と一致する. すなわち出力のカテゴリごとの数値は、しかるべきターゲットを指し示すように並ぶ. これは分類や画像認識を行うニューラルネットワークの動作とあい重なる. そして識別子をベクトルに変換することも、ニューラルネットワークの機能の一部として整合しさえすれば良いはずだ. そうするとベクトルを外であらかじめ決めた値で作って与えなくても良さそうだ.
そういうことなのだ. ベクトルの長さはやってみないとわからないから適当に決める. そしてこれを、1 つのベクトルが 1 つの識別子に対応するように、語彙数の分だけ用意する. 各ベクトルの値は出力が目的に合致するようにニューラルネットワークの学習で獲得する. つまり識別子をダミー変数に変換するベクトルは、ニューラルネットワーク自身に任せてしまうのだ.
この学習について、もう少し具体的に考えてみよう. はじめは左の四角の中の行列は値を決めようがないから乱数で初期化されているものとしよう. そして簡単のためにデータを一つずつ処理するものとしよう. 順伝播では識別子が指すベクトルが行列の中から選ばれる. だから選ばれたベクトルは乱数の並んだものだ. これがニューラルネットワークに入り、出力が作られる. ニューラルネットワークの中の重みやバイアスも初期化したばかりならば、当然ながら出力はでたらめな値が並ぶ. しかしこの出力と正解値を比較する. その出力と正解値の隔たりを小さくするように、逆伝播によって勾配が下流側から上流側へと伝えられ、ニューラルネットワークの入力の勾配が得られる. これはニューラルネットワークの入力としたベクトルの勾配に他ならない. そしてそのベクトルは順伝播の際に入力したものであって、左の四角の中の行列から識別子が選んだ行を抜き出したものだ. だからニューラルネットワークから渡さ

れた勾配はその行列の行に対するものとすれば良い. そしてもちろん、勾配が得られるならば値を更新して、出力を正解に近づけていくことができる. これはこれまでニューラルネットワークを学習させる過程に他ならない. もはや図 5.6.3 の左の四角は、ニューラルネットワークの中に取り入れて、新たな層とするべきだろう. そしてこの変換の仕組みを Embedding、それを担う新たな層を Embedding Layer と呼ぶ. ここまでさんざん出てきた識別子を変換してベクトルにして表したものをベクトル表現という.

5.6.3 Embeddin 層の実装

ではさっそく、Embedding Layer を Neuron.py にクラス Embedding として実装していこう. 順伝播と逆伝播と更新は前項で述べてきたことを、そのままプログラミングすれば良い. そこでコントラクタは後回しにして、forward()メソッド、backward()メソッド、そして update()メソッドから作ろう.

リスト 5.6.4 Embedding Layer の実装～その 1

```
class Embedding:

    # def __init__():はリスト 5.6.5、def init_parameter(self, m, n):はリスト 5.6.6 による

    def forward(self, x, DO_rate=0.0):
        self.x = x                  # 入力 x は w のどの行を抽出するかを示す
        y = self.w[x]               # y は x の指す w の行を並べたもの
        return y                    # y の形状は(B, T, n)
                                    # 即ち長さ n のベクトルがバッチ数×時系列長だけ並ぶ
    def backward(self, dy):
        self.grad_w = np.zeros_like(self.w, dtype=Config.dtype)
        for i, idx in enumerate(self.x):
            self.grad_w[idx] += dy[i] # forward の際に抽出した行の勾配とする

    def update(self, **kwargs):
        self.w -= self.optimizer_w.update(self.grad_w, **kwargs)
```

リスト 5.6.4 に Embedding Layer の forward()、backward()、update()の 3 つのメソッドを示す. カテゴリ変数をダミー変数に変換するための行列 w は、すでに与えられているものとして、これらを見ていく.

順伝播の forward()メソッドでは、入力 x によって選ばれた行列 w の行が出力 y となる. ただし入出力の形状には注意が必要だ. すなわち、入力 x はバッチ数×時系列長の大きさの行列であり、出力 y はバッチ数×時系列長×ベクトル長の 3 次元配列だ.

いっぽう逆伝播の backward()メソッドで、下流側からの勾配は、順伝播の際に抽出した行列 w の行に対応させる．ただしこれも下流からの勾配の形状に注意が必要だ．順伝播で入力 x の指す w の行を x の並びに従って順に並べたのだから、下流からの勾配も、その並び、すなわち入力 x の並びに従って、順に取り出して、その x の値の指す w の行の勾配とする．そのために、

```
self.grad_w = np.zeros_like(self.w, dtype=Config.dtype)
```

で w の勾配 grad_w を値 0 で初期化しておいて、

```
for i, idx in enumerate(self.x):
    self.grad_w[idx] += dy[i]
```

によって、入力の際に保存した x から、順に値を取り出して、その指す grad_w に下流からの勾配を適用する．

更新の update()メソッドは、逆伝播で保存した grad_w により w の更新を行う．このやり方はリスト 5.3.3 と同じだ．

では後回しにしたコントラクタ周りを実装しよう．カテゴリ変数をダミー変数に変換する行列 w は、第 4 章のリスト 4.1.1～リスト 4.1.8 に示す BaseLayer、NeuronLayer における重み w の行列と同じように扱う．そしてその初期化は、コントラクタと init_parameter()メソッドに分けて実装する．

リスト 5.6.5 Embedding Layer の実装～その 2

```
class Embedding:
    def __init__(self, *configuration, **kwargs):
        if len(configuration) == 2:
            m, n = configuration
        if len(configuration) == 1:
            m = 10000; n, = configuration
        if len(configuration) == 0:
            m = 10000; n = 100
        self.config = m, n
        print('Initialize', self.__class__.__name__, self.config)
        self.width      = kwargs.pop('width',     None)
        optimizer_name  = kwargs.pop('optimize', 'SGD')

        self.w = None
        self.optimizer_w = cf.eval_in_module(optimizer_name, Optimizers)
        self.init_parameter(m, n)
```

リスト 5.6.5 に示す Embedding Layer のコントラクタで、行列 w の大きさは引数*configuration で指定する．その w の行数は語彙数であり、列数はダミー変数のベクトル長だが、ここではそれぞれ m、n としている．その両方を指定する場合は、引数で指定した値をそのまま設定する

が、ベクトル長のみを指定する場合には、語彙数 m = 10000 に、いずれも指定しない場合には、m = 10000、n = 100 にする. m, n はインスタンス変数 config に保存する.
あとは引数**kwargs に従い、行列 w の初期値の広がり係数 width と更新の際の最適化関数を指定する. なおこれはリスト 5.3.1 に示す BaseLayer クラスの最適化関数の指定と同じやり方だ.
第 4 章の NeuronLayer はリスト 4.1.5 に示すように forward()メソッドの最初の実行時に init_parameter()メソッドを呼んで重み w やバイアス b を初期化するが、Embedding の場合には行列 w の初期化はコントラクタから init_parameter()メソッドを呼んで行う.

リスト 5.6.6 に行列 w の初期化を行う init_parameter()メソッドを示す.

リスト 5.6.6 Embedding Layer の実装～その 3

```
class Embedding:

    def init_parameter(self, m, n):
        if self.width is not None:
            width = self.width
        else:
            width = np.sqrt(1/n)  # 語ベクトルの各要素の活性化のため
                                  # (通常の NeuronLayer とは違う)
        self.w = (width * np.random.randn(m, n)).astype(Config.dtype)
        print(self.__class__, 'init_parameters', m, n)
```

リスト 5.6.6 はリスト 5.3.2 の BaseLayer クラスの同名メソッドと基本的には同じだ. しかし、バイアス b が無いばかりでなく、初期値の与え方に若干違いがある. それはリスト 5.3.2 で Xavier の初期値を入力幅を示す m の平方根の逆数 np.sqrt(1/m)としているのに対し、ここではベクトル長 n の平方根の逆数 np.sqrt(1/n)としている点だ. これは一つのベクトルが一つの識別子に対応するからで、識別子の値が違えば、別のベクトルが選ばれる. いっぽう NeuronLayer の入力では、各入力を要素として、それを集めたベクトルに対して、その各要素の重み付け和を求める. したがって NeuronLayer では入力全体で一つであるのに対し、Embedding ではダミー変数のベクトル一つ一つが単位となる. だから前者では入力幅を示す m が、後者ではベクトル長を示す n が一つの情報の単位の大きさなのであり、これに対応して、その要素を活性化するには、その平方根の逆数 np.sqrt(1/m)ないしは np.sqrt(1/n)を広がり係数にするのが良いということになる.

5.6.4 Embeddin 層を RNN に組み込む

前項で Embedding Layer の実装が完了したので、これを RNN に組み込んでいこう. RNN はリスト 5.1.1～リスト 5.1.3 で作ってファイル RNN.py に置いたから、ここに書き加えよう.

リスト 5.6.7 Embedding Layer を組み込んだ RNN

```
class RNN_erf(RNN_Base):
    def __init__(self, v, l, m, n, **kwargs):
        super().__init__(v, l, m, n, **kwargs)
        self.embedding_layer = Neuron.Embedding(v, l, **kwargs)
        self.rnn_layer = Neuron.RnnLayer(l, m, **kwargs)
        self.neuron_layer = Neuron.NeuronLayer(m, n, **kwargs)
        self.layers.append(self.embedding_layer)
        self.layers.append(self.rnn_layer)
        self.layers.append(self.neuron_layer)
```

Embedding Layer を組み込んだ RNN は、リスト 5.6.7 に示すコントラクタを定義するだけだ. クラス名は層構成に従って RNN_erf とした. これはリスト 5.5.3 の RNN_rf に Embedding Layer を加えたものだ. コントラクタの引数は Embedding Layer のために語彙数 v を指定する必要があって一つ増えている. なお、リスト 5.5.1 に示す RNN_base クラスのコントラクタに、引数のコメントをつけておいた.

このクラスは RNN_Base クラスを親とする派生クラスであり、その RNN_Base クラスは NN_CNN_Base クラスを親とし、そのメソッドを継承する. だからたとえば順伝播は、大元になるリスト 4.6.2 の NN_CNN_Base クラスの forward()メソッドが呼ばれて、入力から self.layers に積み重ねた各層を順にたどって伝播される. そして Embedding Layer に関しては、リスト 5.6.4 の forward()メソッドが実行される. 逆伝播や更新なども同様だ.

とまれ、層を重ねて深いニューラルネットワークを作るのは、いとも簡単だ.

5.7 RNNで小説を扱う

これまでRNNで、二進数の足し算、ずっと続くサインカーブ、「いろは歌」を扱ってきて、これら性質の異なることをRNNが処理できることを確かめてきた.
あらためて眺めれば、RNNは過去から現在の履歴にもとづく未来予測に使えるのだ. 何かを作り出す創造性の本質の一つが、この未来予測にあるとするならば、RNNは創造性を獲得したニューラルネットワークということができるだろう.
前節ではRNNにEmbedding Layerを導入し、本格的な文章が扱える準備を整えた. さあここまでくれば RNN に小説を学習させてみたくなることだろう. しかし実際にやろうとすると相当に困難を伴うと思われる. そこで本節では RNN で小説を扱うにあたってやっておくべきことに取り組んでいこう. そしてRNNで小説を題材にして少し本格的な文章を綴ることにしよう.

5.7.1 時系列データの切り出し

RNNでまず何とかしなければならないのは時間だ. リスト5.5.14の中で用意したデータは、リスト5.5.12の「いろは歌」だった. これでは大幅に後退したようになるが、実のところプログラムのデバッグなどを行うのにも、あまりにも時間がかかると効率が悪いから「いろは歌」なのだ.
リスト5.2.4には小説「羅生門」から、ニューラルネットワークに入れるデータを作るところがある. そこでリスト5.5.14において、リスト5.5.12のかわりにリスト5.2.4を持ってきて、char → wordなどの変更を行えば、すぐに動く. そう動くには動くのだ. が、しかし、問題は、時間がかかることなのだ. そのうえ学習も遅い. リスト 5.2.4〜5.2.5 やリスト 5.2.6〜5.2.7で、ニューラルネットワークでマルコフ連鎖を行った際にも、時間がかかることは経験したと思うが、それよりも更に更に遅い. ちなみに「羅生門」から切出した入力データXと正解値Cの形状を見るともとのテキストの加工によって多少の違いはあるだろうけれど、手元のデータではone_hotに変換後で、いずれも(1, 4471, 990)となっている. これはバッチ処理のできない形であり、時系列長は4471とやたらと長く、一つのデータのベクトル長は990だ. これは「いろは歌」の(1, 46, 47)とは大違いだ.
図5.4.3を思い出してもらえばわかるように、時系列にかかわる処理は、t=0の結果が出ないとt=1が計算できず、t=1の結果が出ないとt=2が計算できず... という具合に逐次的にしか進まないし、その逆伝播ではまたこれを逆から辿ることになって、これが片方向で4471もあったら時間がかかって当然なのだ. おまけにそんなに長い間の蓄積から、いっぺんにエラーを減らす方向を見出すことは、想像することすらままならない.
だから断固として、時系列長を適当に短く切って、バッチ処理で同時に扱えるように加工すべ

きなのだ．そして時系列長を短くしても、必要に応じて、図5.4.4に示すように、一連の時系列に続いて、それに続く一連の時系列というように、順につないでいけば、長いシーケンス ― 文章ならば文脈とでもいうべきか ― を勘定に入れることもできるだろう．

さあ説明が長くなったから、このためのプログラムを用意しよう．これをリスト 5.7.1 に示す．やることは簡単だ．頭から尻尾まで一続きのデータから、それを時系列長の連続する部分をひとかたまりとして切り出して並べるだけだ．そして切り出す際に一時刻だけずらしたものを切り出せば入力と正解となる．そしてそのかたまりの中をきちんと逐次処理することにする．入力と正解の対応さえ合っていれば、別の塊の組をバッチの処理で同時に処理してしまえる．これで大幅に時間短縮できるはずだ．これは common_function.py に入れておこう．

リスト 5.7.1 時系列データの切り出し

```
def arrange_time_data(source, time_size, step=None, array=True):
    """ ひと続きの学習データを入力データと正解データとして切り出す """
    if step is None:
        step = time_size
    print('時系列長は',time_size, 'データの切出し間隔は', step, 'です')
    input_data, correct_data = [], []
    for i in range(0, len(source)-time_size, step):
        input_data.append(source[i:i+time_size])
        correct_data.append(source[i+1:i+1+time_size])
    if array:
        input_data   = np.array(input_data)
        correct_data = np.array(correct_data)
    return input_data, correct_data
```

リスト 5.7.1 の関数 arrange_time_data()は、引数 source に corpus などの一つながりの元データを与え、引数 time_size で切り出す際の時系列長を指定する．また引数 step は切り出しの間隔を指定する．たとえば、「いろはにほへとちりぬるをわかよたれそつねならむうゐのおくやまけふこえてあさきゆめみしゑひもせす」で time_size=4 ならば、最初に切り出されるのは「いろはに」で、次に切り出すのは step=1 ならば「ろはにほ」だし、step=2 ならば「はにほへ」という具合だ．引数にもう一つ用意した array は、返り値を numpy の array にするのか、それとも切り出して並べただけのリストにするのかを指示する．先に述べたように input_data に対して correct_data は一時刻だけ後ろにずらしたものだ．

ではさっそく動作を確かめてみよう．リスト 5.2.4 から借用してリスト 5.7.2 を作る．リスト 5.2.4 で 1 つずつずらして切出し、前を入力とし、後を正解値にしていたところをリスト 5.7.1 の関数 arrange_time_data()を呼ぶようにするだけだ．念のためデータの確認も加えた．

リスト 5.7.2 切り出し動作の確認

```
import numpy as np
import common_function as cf

nobel = '羅生門'

# — テキストデータの取得 —
path = '../aozorabunnko/'
file_name = nobel + '.txt'
with open(path + file_name, mode='r', encoding='utf-8') as f:
    text = f.read()

wordlist = cf.split_by_janome(text)

# corpus と辞書を作る
corpus, word_to_id, id_to_word = cf.preprocess(wordlist)
vocab_size = len(word_to_id)
corpus = np.array(corpus)

# 時系列長を指定して切出す
time_size = 30        # 時系列の数
X, C = cf.arrange_time_data(corpus, time_size, step=1)
print(X.shape, C.shape)

# データの確認
for x in X[0]:
    print(id_to_word[int(x)], end='')
print()
for c in C[0]:
    print(id_to_word[int(c)], end='')
```

リスト 5.7.2 を実行すると、入力と正解のはじめのデータが語に変換されて次の 2 行が出力される．

ある日の暮方の事である。一人の下人《げにん》が、羅生門《らしょうもん》の下で雨
日の暮方の事である。一人の下人《げにん》が、羅生門《らしょうもん》の下で雨やみ

このように入力と正解が 1 語ずれたものになっていることが確認できる．これでデータの準備はできた．

5.7.2 文章の生成のために

RNN に小説を学習させて、やりたいことは文章の生成だ．その文章の生成は、リスト 5.5.14 やリスト 5.2.5 では、学習が終わってから確認として 1 回だけ行っていた．しかし学習の途中でも確認して、RNN の成長を見ていきたい．そこで少しでも簡単に文章が生成できるようにしておこう．文章の生成のために RNN.py の中には関数 generate_text()を置いた．もちろんこれを呼出して使うが、それでもいくつか必要なことがあるから、これを関数内に閉じ込めてしまおう．

リスト 5.7.3 Embedding Layer がある場合の文章の生成

```
def generate_text(length, seed):
    seed = cf.split_by_janome(seed)
    model.reset_state()
    def func(x):
        x = np.array([x])
        y = model.forward(x.reshape(1, 1))
        return y
    RNN.generate_text(func, word_to_id, id_to_word, length, seed)
    print('¥n')
```

リスト 5.7.3 に示す関数 generate_text()は Embedding Layer がある場合に対応し、Embedding Layer がない場合には若干の変更が必要で、それをリスト 5.7.4 に示す．いずれにせよ本体のプログラム内に置く前提だ．

リスト 5.7.4 Embedding Layer がない場合の文章生成

```
def generate_text(length, seed):
    seed = cf.split_by_janome(seed)
    model.reset_state()
    def func(x):
        x = np.eye(vocab_size)[x]
        y = model.forward(x.reshape(1, 1, -1))
        return y
    RNN.generate_text(func, word_to_id, id_to_word, length, seed)
    print('¥n')
```

リスト 5.7.3 とリスト 5.7.4 はいずれも、RNN.py 内に置いた同名の関数を実行する関数だ．関数名は同じでも、これらと RNN.py に置いたそれは別物だから注意されたし．

引数で与えた seed を janome で語分割しているが、こうしておけば、この関数を呼び出す際に語の分割を気にせずに seed に文字列を与えられるから便利だ．そのほか生成前に reset_state()で初期化すること、引数として与える func の定義、その関数への入力の変換と

整形なども、この関数内で片付けてしまおう．Embedding Layer がある場合には、func への入力は、カテゴリ変数のままで numpy の１行１列の array にする．いっぽう Embedding layer がない場合には、カテゴリ変数を one hot 形式のダミー変数に変換し、形状もバッチサイズ、時系列長をいずれも１として 1×1×語彙数の配列にする．

5.7.3 RNN に小説を学習させる

準備に手数を要したが、ここからプログラムの本体を作ろう．いよいよ RNN に小説を学習させて、その学習した RNN で文章を生成することになる．

本体のプログラムをリスト 5.7.5 に示す．ここではまず Embedding Layer がある場合を示し、それがない場合については、必要な変更箇所を示すことにする．
前処理として、小説を読込んで語に分割し、id の並んだ corpus と、語と id の変換の辞書を作り、そして corpus から時系列長のデータを切り出す．これらの処理はリスト 5.7.2 をそのまま使う．次に文章生成の関数 generate_text()だが、これはリスト 5.7.3 で作った．これらはいずれも本体のプログラム内に置くが、紙面上は省略する．
これらの間に挟まれて見落としそうだが、RNN が学習したパラメータ（重みやバイアス）を保存するためのファイルは、
file_name_m = '../corpus_params/RNN_erf_params_' + nobel + '.pkl'
で指定する．これは一つ上の階層にある corpus_params というフォルダにそのファイルを置くように指定している．そしてこの場合に nobel は'羅生門'だから、ファイル名は、
RNN_erf_params_羅生門.pkl　となる．保存済みのパラメータを移植するかどうかは、モデルの生成後に問い合わせに応じて行い、ファイルへのパラメータの保存はプログラムの末尾で行う．
続いてモデルの生成だ．リスト 5.5.14 と同様だが、ここでは RNN.py の中にリスト 5.6.7 で作った RNN_erf を model としてインスタンス化する．インスタンス化の際に引数として、語彙数、語ベクトル長、隠れ層ニューロン数、出力数を指定するが、出力数は語彙数に等しい．またここでは語ベクトル長は 256 とし、隠れ層ニューロン数も 256 とした．そして出力層の活性化関数は Softmax、損失関数は CrossEntropyError だ．model.summary()で構成を確認しておこう．

リスト 5.7.5 Embedding Layer がある RNN で羅生門を学習する

```
import numpy as np
import common_function as cf
import RNN
import time

#nobel = '羅生門' 以下省略：リスト 5.7.2 による

file_name_m = '../corpus_params/RNN_erf_params_' + nobel + '.pkl'

#def generate_text()は省略：リスト 5.7.3 による

# ― モデルの生成 ―
model = RNN.RNN_erf(vocab_size, 256, 256, vocab_size,
                    activate='Softmax',
                    loss='CrossEntropyError')
model.summary()

# モデルにパラメータを移植
if input('継承しますか？(y/n)=>') in ('y', 'Y'):
    model.load_parameters(file_name_m)

# ― 学習 ―
epoch = 101
batch_size = 50
start = time.time()
index_random = np.arange(len(X))
mesurement = cf.Mesurement(model)
for i in range(epoch):
    np.random.shuffle(index_random)
    for j in range(0, len(X), batch_size):
        model.reset_state()
        idx = index_random[j:j+batch_size]
        x = X[idx]
        c = C[idx]
        c = np.eye(vocab_size, dtype='f4')[c]
        y, l = model.forward(x, c)
        model.backward()
        model.update(eta=0.6, g_clip=0.25)
    if i==0:
        print(' epoch elapse | error  accuracy')
    if i%10==0:
        model.reset_state()
        l, acc = mesurement(x, c)
        print('{:6d} {:6.2f} | {:6.3f} {:6.3f}'¥
              .format(i, time.time()-start, float(l), acc))
        start = time.time()
        generate_text(100, 'ある日の暮方の')

# グラフの描画と最終確認
errors, accuracy = mesurement.progress()
cf.graph_for_error(errors, accuracy, label=('error','accuracy'))
generate_text(400, 'ある日の暮方の')
model.save_parameters(file_name_m)
```

パラメタの移植はすでに述べたから、次は学習だ．学習はリスト 5.2.5 と同様だが、いくつか違う点がある．まず最内ループでバッチ処理するたびに reset_state()で初期化が必要だ．これはバッチのたびに時間的に連続しない部分を扱うからだ．reset_state()は mesurement()で測定を行う際にも同様に時間的に連続しないから必要だ．もうひとつ違う点として、リスト 5.2.5 では入力 X と正解 C を事前に one hot 形式にしているが、ここでそれはしていない．入力については Embedding Layer でダミー変数に変換するから不要だが、正解については one hot 形式に変換する必要があり、model.forward()の前で

```
    c = np.eye(vocab_size, dtype='f4')[c]
```

により行っている．
あとは、学習率 eta や g_clip の指定、表示のインターバルが違う．そして表示の際にはリスト 5.7.3 で定義した関数 generate_text()を実行して、「ある日の暮方の」から始まる文章を作成する．
100 エポックの学習が終わると学習曲線を表示し、最後に文章作成の確認を行って、学習したパラメータをファイルに保存する．

それではこれを実行した結果を示そう．まずは実行してすぐに表示されるモデルの構成だ．

```
~~ model summary of RNN_erf ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
layer 0 Embedding
 configuration = (990, 256)
------------------------------------------------------------
layer 1 RnnLayer
 configuration = (256, 256)
------------------------------------------------------------
layer 2 NeuronLayer
 configuration = (256, 990)
 activate = Softmax
------------------------------------------------------------
loss_function = CrossEntropyError
~~ end of summary ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

このように意図した通りのモデルとなっていることがわかる．

続いて学習が始まると、経過が以下のように表示される.

```
 epoch elapse | error  accuracy
     0  14.37 |  5.069  0.181
ある日の暮方のはをに《、あるにたた。には、あるのはを《で。。は、て下人てて《た。
が。のにのは、》の、にで》での。を、のににに
も、。は、下人のに、はての》に、老婆のに《に》》。にはの。を、ののての》に、老婆

    10 143.32 |  0.632  0.917
ある日の暮方のは、その男に裸の死骸の上から、梯子の口に、一死骸の頭から、ざるの修理
《さや》の暮しような声である。この時のこの男の心もちから云えば、饑死などと云う事
は、ほとんど、考える事さえ出来ないほど、

    20 145.74 |  0.261  0.956
ある日の暮方の事である。一人の下人《げにん》が、羅生門《らしょうもん》の下で雨やみ
を待っていた。
  広い門の下には、この男のほかに誰もいない。ただ、所々｜丹塗《にぬり》の剥《は》げ
た、大きな円柱《まるば

    30 145.07 |  0.207  0.959
ある日の暮方の事である。一人の下人《げにん》が、羅生門《らしょうもん》の下で雨やみ
を待っていた。
  広い門の下には、この男のほかに誰もいない。ただ、所々｜丹塗《にぬり》の剥《は》げ
た、大きな円柱《まるば
```

（以下省略）

以下は省略するが、手元の Intel の第 7 世代の CPU の PC で 10 エポック当たり 145 秒くらいかかっていて、20 エポックあたりからは、「羅生門」の元の文章の通りに文章が生成される.

次に学習曲線を図 5.7.1 に示す. 20 エポックあたりまで急速に学習が進み、そのあたりからはゆっくりと学習していく様子が伺える. これは 20 エポックから、元の文章を正しく綴っていることと符合する. 図の線がカクカクしているのは、測定のインターバルが 10 エポックごとだからだ. なお測定は全データでやれば精度が上がるが、時間がかかってしまう. あ

くまでも学習の進捗のチェックが目的だから、最後に切り出したバッチデータをそのまま使う、こういうやり方も許されるだろう.

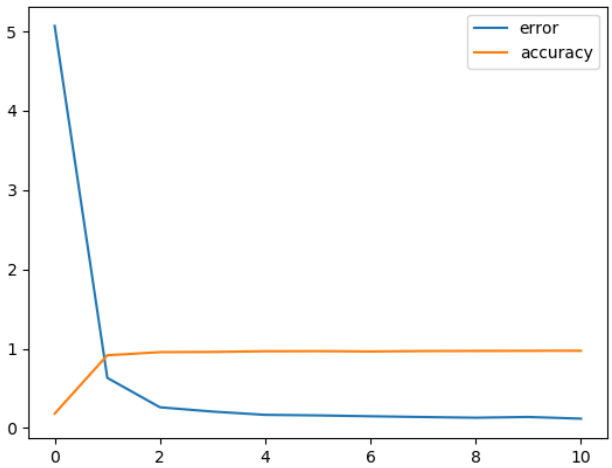

図 5.7.1 羅生門の学習経過

最後の確認では次のように「羅生門」が正しく作成される.

ある日の暮方の事である。一人の下人《げにん》が、羅生門《らしょうもん》の下で雨やみを待っていた。

広い門の下には、この男のほかに誰もいない。ただ、所々｜丹塗《にぬり》の剥《は》げた、大きな円柱《まるばしら》に、蟋蟀《きりぎりす》が一匹とまっている。羅生門が、朱雀大路《すざくおおじ》にある以上は、この男のほかにも、雨やみをする市女笠《いちめがさ》や揉烏帽子《もみえぼし》が、もう二三人はありそうなものである。それが、この男のほかには誰もいない。

何故かと云うと、この二三年、京都には、地震とか辻風《つじかぜ》とか火事とか饑饉とか云う災《わざわい》がつづいて起った。そこで洛中《らくちゅう》のさびれ方は一通りではない。旧記によると、仏像や仏具を打砕いて、その丹《に》がついたり、金銀の箔《はく》がついたりした木を、路ばたにつみ重ねて、薪《たきぎ》の料《しろ》に売っていたと云う事である。洛中がその始末

正しく元の文章が再現できたということが意味するのは、RNN がちゃんと学習したということの証だ.

では元の文章にはなかったものを与えて文章を生成させてみよう．ファイルに保存したパラメタを移植すれば、すぐさま文章の生成が行える．リスト 5.7.5 の学習に関する部分をすっぽり削除して、学習済みのパラメータを使って文章を作成するプログラムを作っておこう．それをリスト 5.7.6 に示す．リスト 5.7.5 から不要な部分を削除すれば良いだけだ．

リスト 5.7.6 羅生門を学習した RNN で文章を生成する

```
import numpy as np
import common_function as cf
import RNN
import time

#nobel = '羅生門' 以下省略：リスト 5.7.2 による

file_name_m = '../corpus_params/RNN_erf_params_' + nobel + '.pkl'

#def generate_text()は省略：リスト 5.7.3 による

# ― モデルの生成 ―
model = RNN.RNN_erf(vocab_size, 256, 256, vocab_size,
                    activate='Softmax',
                    loss='CrossEntropyError')
model.summary()

# モデルにパラメータを移植
model.load_parameters(file_name_m)

# ― 文章生成 ―
generate_text(400, '人間の光')
```

さっそくリスト 5.7.6 を実行して結果を見てみよう．「5.2 ニューラルネットワークでマルコフ連鎖」で試したのと同じ「人間の光」で始まる文章だ．

人間の光が、かすかに、まだ男の右の頬をぬらしている。短い鬚の中に、赤く膿《うみ》を持った面皰《にきび》のある頬である。下人は、始めから、この上にいる者は、死人ばかりだと高を括《くく》っていた。それが、梯子を二三段上って見ると、上では誰か火をとぼして、しかもその火をそこここと動かしているらしい。これは、その濁った、黄いろい光が、隅々に蜘蛛《くも》の巣をかけた天井裏に、揺れながら映ったので、すぐにそれと知れたのである。この雨の夜に、この羅生門の上で、火をともしているからは、どうせただの者ではない。

下人は、守宮《やもり》のように足音をぬすんで、やっと急な梯子を、一番上の段まで這うようにして上りつめた。そうして体を出来るだけ、平《たいら》にしながら、頸を出来るだけ、前へ出して、恐る恐る、楼の内を覗《のぞ》いて見た。

見ると、楼の内には、噂に聞いた通り、幾つかの死骸《しがい》が、無造作に棄ててある

はじめの文を除いて「羅生門」の途中からの文章がそのまま出てくるが、少なくとも出だしの「人間の光」から、それらしくつながっている．偶々いみじくも最初の一文は「5.2 ニューラルネットワークでマルコフ連鎖」で試した結果と全く同じとなった．これが同じかどうかはさておき「N階マルコフ連鎖の階数にとらわれない汎用的な方法」というものに辿り着くことが出来たと言えるのではないだろうか？
ここでは「羅生門」を学習させただけだから、そこに含まれる語のつながりで文章を作る限り、どうしても元の羅生門の文章を綴り続けてしまうのはやむを得ないだろう．しかしもっとたくさんの長い文章を学習させたならば、状況は変わってくることが期待される．問題は学習に時間がかかることと、その学習がうまく進むかどうか、そしてその際にそもそも、それを担えるRNNとなっているのかどうか、ということに尽きる．とまれ一つのゴールにたどり着いたことは確かだ．

さて蛇足ともなろうが敢えて少し戻ってみよう．ここではEmbedding Layerを備えたRNNで小説を扱ったが、Embedding Layerは文章を扱うのに必須ではない．そこでEmbedding Layerなしの RNNで「羅生門」を扱ってみよう．
リスト5.7.5からの修正はわずかだ．まずモデルはRNN_erfにかえてRNN_rfだ．RNN_rfはリスト5.5.14では「いろは歌」を学習させた．ともかく、
file_name_m = '../corpus_params/RNN_erf_params_' + nobel + '.pkl' を
file_name_m = '../corpus_params/RNN_rf_params_' + nobel + '.pkl' とし、

```
model = RNN.RNN_erf(vocab_size, 256, 256, vocab_size,
                    activate='Softmax',
                    loss='CrossEntropyError')
```

を以下のようにする．

```
model = RNN.RNN_rf(vocab_size, 256, vocab_size,
                    activate='Softmax',
                    loss='CrossEntropyError')
```

そして、リスト5.7.5では省略しているが、def generate_text()はリスト5.7.3にかえてリスト5.7.4を使う．
もう一つ学習のループの中で、正解をone hot形式にしているが、順伝播の際に入力も同様にone hot形式にする必要があるから、
c = np.eye(vocab_size, dtype='f4')[c] の前に
x = np.eye(vocab_size, dtype='f4')[x] を置く．
修正はわずかだが念のために、これらの修正を行ったものをリスト5.7.7に示す．

リスト 5.7.7 羅生門を Embedding Layer がない RNN で羅生門を学習する

```
import numpy as np
import common_function as cf
import RNN
import time

#nobel = '羅生門' 以下省略：リスト 5.7.2 による

file_name_m = '../corpus_params/RNN_rf_params_' + nobel + '.pkl'

#def generate_text()は省略：リスト 5.7.4 による

# ― モデルの生成 ―
model = RNN.RNN_rf(vocab_size, 256, vocab_size,
                   activate='Softmax',
                   loss='CrossEntropyError')
model.summary()

# モデルにパラメータを移植
if input('継承しますか？(y/n)=>') in ('y', 'Y'):
    model.load_parameters(file_name_m)

# ― 学習 ―
epoch = 101
batch_size = 50
start = time.time()
index_random = np.arange(len(X))
mesurement = cf.Mesurement(model)
for i in range(epoch):
    np.random.shuffle(index_random)
    for j in range(0, len(X), batch_size):
        model.reset_state()
        idx = index_random[j:j+batch_size]
        x = X[idx]
        c = C[idx]
        x = np.eye(vocab_size, dtype='f4')[x]
        c = np.eye(vocab_size, dtype='f4')[c]
        y, l = model.forward(x, c)
        model.backward()
        model.update(eta=0.6, g_clip=0.25)
    if i==0:
        print(' epoch elapse | error  accuracy')
    if i%10==0:
        model.reset_state()
        l, acc = mesurement(x, c)
        print('{:6d} {:6.2f} | {:6.3f} {:6.3f}'¥
              .format(i, time.time()-start, float(l), acc))
        start = time.time()
        generate_text(100, 'ある日の暮方の')

# グラフの描画と最終確認
errors, accuracy = mesurement.progress()
cf.graph_for_error(errors, accuracy, label=('error','accuracy'))
generate_text(400, 'ある日の暮方の')
model.save_parameters(file_name_m)
```

リスト 5.7.7 を走らせてみて、どうだろうか？ Embedding Layerr が無くても学習するし、それなりに文章も紡ぐだろう．ただその学習にかかる時間が長くなり、進捗も少し劣るのではないだろうか？ これは Embedding Layer の効果を逆から見ていることに他ならない．すなわち学習にかかる時間が短くなり、進捗も良くなるという効果が、Embedding Layer によってもたらされているのだ．

もう一つやっておこう．リスト 5.7.5 では文章を Janome を使って語に分割した．しかしそうせずに文字で分割する方法もある．節「5.1 マルコフ連鎖」でも、リスト 5.1.13 とリスト 5.1.14 の結果を比べて、違いが無いくらいの文章が綴られていた．そこで、リスト 5.7.5 を元にして文字分割で「羅生門」を扱ってみよう．
まずはリスト 5.7.5 で省略しているリスト 5.7.2 の切り出しで、
wordlist = cf.split_by_janome(text) は不要だ．そして、それ以下の部分で、
wordlist を text、word_to_id を char_to_id、id_to_word を id_to_char と変更する．
修正は多くないが、やはり念のために修正を行ったものをリスト 5.7.8 に示す．なおここでは冒頭の必要なモジュールの import は省略する．

リスト 5.7.8 文字分割での切り出し動作の確認

```
nobel = '羅生門'

# — テキストデータの取得 —
path = '../aozorabunnko/'
file_name = nobel + '.txt'
with open(path + file_name, mode='r', encoding='utf-8') as f:
    text = f.read()

# corpus と辞書を作る
corpus, char_to_id, id_to_char = cf.preprocess(text)
vocab_size = len(char_to_id)
corpus = np.array(corpus)

# 時系列長を指定して切出す
time_size = 30       # 時系列の数
X, C = cf.arrange_time_data(corpus, time_size, step=1)
print(X.shape, C.shape)

# データの確認
for x in X[0]:
    print(id_to_char[int(x)], end='')
print()
for c in C[0]:
    print(id_to_char[int(c)], end='')
```

またこれもリスト 5.7.5 で省略している部分で、リスト 5.7.3 の関数 generate_text()だが、同様に、冒頭の seed = cf.split_by_janome(seed)は不要で、word_to_id を char_to_id、id_to_word を id_to_char に変更する. これも修正は多くないがリスト 5.7.9 に示す.

リスト 5.7.9 文字分割で Embedding Layer がある場合の文章の生成

```
def generate_text(length, seed):
    model.reset_state()
    def func(x):
        x = np.array([x])
        y = model.forward(x.reshape(1, 1))
        return y
    RNN.generate_text(func, char_to_id, id_to_char, length, seed)
    print('¥n')
```

修正箇所を示してきたが、まとめると語分割を文字分割に変えるには、リスト 5.7.5 の中でリスト 5.7.2 で示す切り出しの部分をリスト 5.7.8 に置き換え、リスト 5.7.3 で示す関数 generate_text()をリスト 5.7.9 に置き換えれば良い.

さて文字分割の場合はどうだろうか？ 結果は省略するが、元の語分割の場合に比べて何ら遜色ないのではないだろうか？ 学習に要する時間もほとんど変わらず、学習の進捗も変わらないことだろう.

文章を文字か語に分割し、それを識別子 id に変換し、その変換の辞書が用意できれば、なんでも良いのだ. ここでは日本語を扱ってきたが、たとえば英語なども同じように扱えるだろう. むしろ英語は日本語と違って、語の分割が空白によって明示的なので、語分割は簡単にできる. また文字も、アルファベットと数字と、その他の記号なので、日本語に比べて圧倒的に種類が少なく扱いやすい. だから是非、英語の文章を扱ってみて欲しい. またその際に、文章を扱う RNN のモデルも Embedding Layer がある場合、無い場合、いずれも作ってきたから、これも必要に応じて選べば良いだろう.
そして更に、ドイツ語、フランス語、イタリア語、スペイン語、ラテン語、中国語、ハングル、アラビア語、などなど世界の様々な言語に取り組むのも面白そうだし、小説に限らず、学術論文とか、新聞とか、日常会話とか、様々な文章があるだろう. それらのどれもが、RNN で扱う題材となりうるだろうし、それを学習した RNN が紡ぎ出す文章がどうなるのか、興味が尽きない.

5.8 RNNの高度化

これまで小説をRNNで学習して、学習したRNNを使って文章を綴るということが出来た．綴られる文章は与える出だしに続いて読める文章となっている．しかしすぐに学習した小説の文章そのものに戻ってしまい、そこからはその続きを綴るだけとなってしまう．これは文字や語の並びの短い連なりから、次に来るべきものを選んで綴っているからにほかならず、そうなってしかるべきとも考えられる．人は文脈を捉えて文章を読み進めて意味を理解していくのだから、RNNごときに出来るはずもないと言ってしまえば、それまでだろう．しかしRNNの高度化によって、文脈あるいはそれに相当する何かを加味することは不可能ではない．
すなわち、これまで扱ってきた単純な RNN が短期的な依存関係のみによっているとすれば、RNNの高度化によって、長期的な依存関係をも担うようにすることができる．そしてそれによって、文脈を捉えることに近づける可能性があるだろう．
そこでここでは、ゲート付きRNNとも言われるLSTMとGRUの二つのメジャーなRNNの高度化した方式を導入する．なおこれらの詳しい説明は、他に譲ることにして割愛し、これまで作ってきたRNNにどう組み込むかを説明して、実際に動かしていくことにしよう．

5.8.1 RNNの作り

RNNを高度化していくにあたり、もう一度RNNのつくりをおさらいしておこう．節「5.4 RNN」では、戻りパスを有するニューロンユニットをリスト 5.4.1 で作った．これを改めて単純なRNN Unitとして図示すると、図5.8.1のようになる．

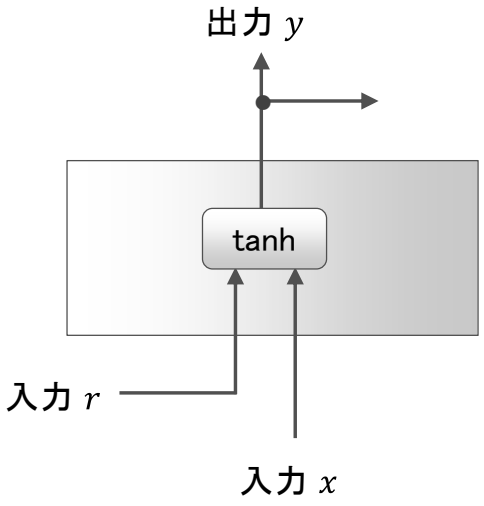

図 5.8.1 単純なRNN Unit

この図は図5.4.2と同じ部分を表すが、図5.4.2では出力が自身に戻るように表されている．しかし実際にはリスト5.4.1に示すようにその接続は外で行われ、ニューロンユニットとして

は単純に x と r の二つの入力を備えているだけなので、図 5.8.1 ではそれを明示した．また、入力がニューロンユニットに入る際に必ず行われる重み付け和の計算は省略しているが、その活性化関数を明示した．

とまれニューロン層としては、これをリスト 5.4.2～リスト 5.4.7 に示すように組み込む．これは図 5.8.1 で示すものを、図 5.4.3 ないし図 5.4.4 に示すように連ねることと対応する．そして、ニューロンユニットの x と r の二つの入力は、一方は外からの入力となり、そしてもう一方は一つ前の時点の出力が入れられる．注意しておきたいのは、出力が入力に戻されるのだが、その時刻が違うという点であり、一つ前の時刻の出力が外からの入力とともに与えられて現在の時点の計算が行われ、それがまた次の時刻の入力の一方に入る、これを繰り返して次々と時系列に連なるデータが処理されていくということだ．

5.8.2 GRU

GRU は Gated Recurrent Unit の略で LSTM よりも後から発表されたが、GRU から実装していくことにする．GRU を図 5.8.2 に示す．

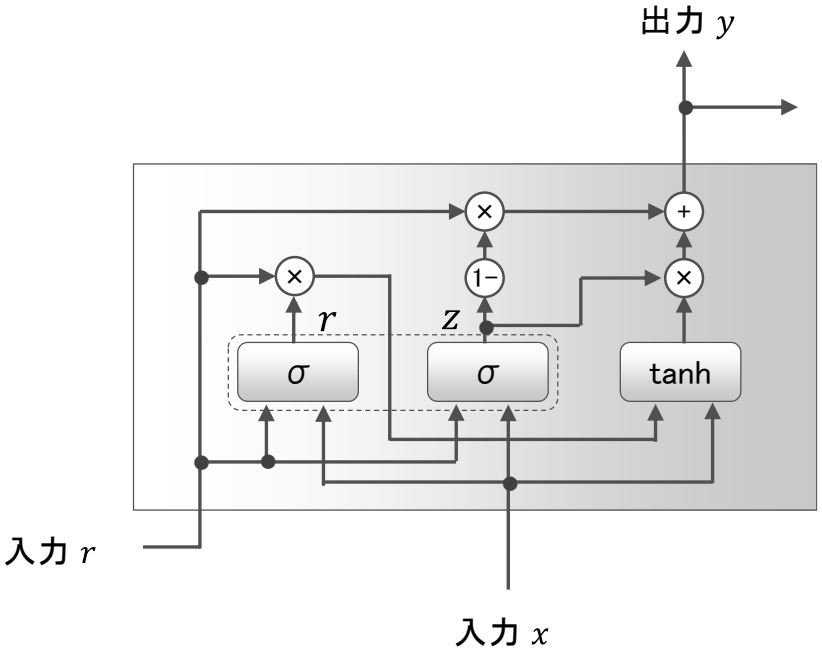

図 5.8.2 GRU

GRU は内部にゲートと呼ばれる調整機構を備え、図 5.8.1 と図 5.8.2 を比較してわかる通り、かなり複雑だ．しかし GRU の外部とのインターフェイス、すなわち入出力は図 5.8.1 に示す

単純なRNN Unitと同じだ．これはニューロンユニットを使って層を構成するのに、単純なRNNと同じようにすれば良いということだ．

とまれGRUを図5.8.2を見ながら、ごく簡単に説明しておこう．入力xから出力yに向かい図の右端のtanhを通る．これは入力の重み付け和をとり、活性化関数tanhで出力に至るから、図5.8.1の単純なRNN Unitの入力xから出力yに至るのと同様だ．ただし、出力の前にゲートがあって値が調節され、また戻りパスである入力rからの値を加えるやり方も違う．この両方が真ん中のσの出力zによって制御され、1-と二つの×と＋でつごう4つの〇で示されるが、やっていることは、zと1－zの割合で入力xから作られた値と、戻りパスrの値を混ぜているだけだ．

またいっぽう、戻りパスの入力rは左端のσの出力r(どちらも同じrだが別)により左端のゲートで調整されてから、入力xと活性化関数の前で加え合わされる．このように二つのσで示される左と中の四角の出力rと出力zは、いずれも×で示される調節用のゲートを制御しているが、これらはそれぞれリセットゲート、更新ゲートと呼ばれる．そしてそれらには入力xと入力rの両方がつながれ、また活性化関数はシグモイド関数だ．

ともかくゲートを設けて、新たな入力と戻りパスからの記憶とを調整しつつ伝えることで、単純なRNNでは成しえなかった長期・短期の両方にわたる依存関係を担うことが可能になっている．

では、GRUを順伝播から実装していこう．

リスト5.8.1 GRUのニューロンユニット～その1

```
class GRU_Unit:
    def forward(self, x, r, w, v, b):
        B, n = r.shape
        wg, wm = w[:, :2*n], w[:, 2*n:]
        vg, vm = v[:, :2*n], v[:, 2*n:]
        bg, bm = b[:2*n],    b[2*n:]

        # 更新ゲートとリセットゲート
        gu = np.dot(x, wg) + np.dot(r, vg) + bg
        g = 1 / (1 + np.exp(-gu))          # sigmoid
        gz, gr = g[:, :n], g[:, n:]        # 更新ゲートとリセットゲート
        # 新しい記憶
        u = np.dot(x, wm) + np.dot(gr * r, vm) + bm
        y = (1 - gz) * r + gz * np.tanh(u)
        self.state = x, r, y, g
        return y
```

リスト5.8.1にGRUのニューロンユニットのforward()メソッドを示す．リスト5.4.1の単純なRNN_Unitと比べると複雑だが変数名は対応する．図5.8.2に示すように二つの入力xとrはσにもtanhにも入るが、これらは個別に重み付け和をとったものが必要で、そのために重

みwとvとバイアスbは、出力の幅の3倍の列数を用意する必要がある．リスト5.8.1では図5.8.2で点線でくくっている二つのσの分をまとめてwg、vg、bgとし、tanhの分をwm、vm、bmとしている．
そして更新ゲートとリセットゲートの分は重み付け和をとってからシグモイド関数にいれる．そしてそこまで処理してから、更新ゲートgz、リセットゲートgrに分ける．このときもともと重み付け和をとる際に重みやバイアスを倍の幅にしていて、まとめて計算していても、両者は別々に計算しているのと同じことになる．すなわち重みやバイアスは更新ゲートとリセットゲートで独立だ．
いっぽう新しい記憶とあるコメントの下は、更新ゲートやリセットゲートとは、また別に重み付け和をとるが、このとき入力rはリセットゲートgrで調整する．そしてその重み付け和を活性化関数tanhにいれて、今度は更新ゲートgzで入力rとの割合を調整して加え合わせる．これらの処理は図5.8.2に示し先に説明した通りだ．
そして最後にself.state = x, r, y, gにより、順伝播の際の入出力とゲートの値を逆伝播に備えて保存する．

続いて逆伝播だ．これは順伝播がややこしい分だけ面倒だ．
しかしたとえば順伝播の最後のところの、
```
y = (1 - gz) * r + gz * np.tanh(u)
```
の逆伝播はtanh(u)をtanh_uとして、
```
grad_r  = grad_y * (1 - gz)
grad_gz = grad_y * (tanh_u - r)
```
というように、一つずつ順伝播の処理を忠実に逆に辿れば、求めることができる．
重み付け和のところも、リスト4.4.3で求めたやり方を振り返りつつ、リスト5.4.1を参考にすれば求められる．とまれここではリスト5.8.2にbackward()メソッドを示すだけとして、この説明は割愛させていただく．

リスト 5.8.2 GRUのニューロンユニット～その 2

```
class GRU_Unit:

    def backward(self, grad_y, w, v):
        x, r, y, g = self.state
        B, n = r.shape
        wg, wm = w[:, :2*n], w[:, 2*n:]
        vg, vm = v[:, :2*n], v[:, 2*n:]
        gz, gr = g[:, :n],   g[:, n:]

        # y 算出の逆伝播
        tanh_u  = (y - (1 - gz) * r) / gz
        grad_r  = grad_y * (1 - gz)
        grad_gz = grad_y * (tanh_u - r)

        # 新しい記憶
        delta_m = grad_y * gz * (1 - tanh_u ** 2)
        grad_wm = np.dot(x.T, delta_m)
        grad_vm = np.dot((gr * r).T, delta_m)
        grad_bm = np.sum(delta_m, axis=0)
        grad_x  = np.dot(delta_m, wm.T)
        grad_rm = np.dot(delta_m, vm.T)     # gr * r : リカレントな記憶の勾配

        # gr * r の逆伝播
        grad_r += gr * grad_rm
        grad_gr = grad_rm * r

        # 更新ゲートとリセットゲート
        delta_g = np.hstack((grad_gz, grad_gr)) * g * (1 - g) # sigmoidの微分
        grad_wg = np.dot(x.T, delta_g)
        grad_vg = np.dot(r.T, delta_g)
        grad_bg = np.sum(delta_g, axis=0)
        grad_w  = np.hstack((grad_wg, grad_wm))
        grad_v  = np.hstack((grad_vg, grad_vm))
        grad_b  = np.hstack((grad_bg, grad_bm))
        grad_x += np.dot(delta_g, wg.T)
        grad_r += np.dot(delta_g, vg.T)

        return grad_x, grad_r, grad_w, grad_v, grad_b
```

5.8.3 単純な RNN と GRU を備えた RNN

GRU のニューロンユニットがリスト 5.8.1、リスト 5.8.2 で出来たので、これを使った層を作っていこう．基本的には単純な RNN のそれがそのまま使えるのだが、ここでリスト 5.4.1 の RNN_Unit とリスト 5.8.1、リスト 5.8.2 の GRU の両方を共通に扱えるようにリスト 5.4.2～リスト 5.4.7 を細工する．

まずはこれまでリスト 5.4.2～リスト 5.4.7 は、RnnLayer という名前のクラスだったが、これを RnnBaseLayer とクラス名を変更する．
そしてコントラクタに self.unit = None の 1 行を書き加える．コントラクタでは、以下の 2 行は派生クラスに任せることにして削除する．

```
m, n = self.config
self.init_parameter(n, m, n)
```

さらに forward()メソッドの unit = RNN_Unit() を unit = self.unit() に書き換える．

リスト 5.8.3 RnnLayer を RnnBaseLayer に

```
class RnnBaseLayer: #RnnLayer の名前を変更
    def __init__(self, *configuration, **kwargs):
        self.unit = None
        print('Initailize', self.__class__.__name__)
        self.width      = kwargs.pop('width',     None)
        optimizer_name  = kwargs.pop('optimize', 'SGD')

        self.w = None; self.v = None; self.b = None
        self.optimizer_w = cf.eval_in_module(optimizer_name, Optimizers)
        self.optimizer_v = cf.eval_in_module(optimizer_name, Optimizers)
        self.optimizer_b = cf.eval_in_module(optimizer_name, Optimizers)

        self.config = configuration

    def forward(self, x):

        # 途中リスト 5.4.2 と同じ

        # unit を起こしながら、順伝播を繰返す
        for t in range(T):
            unit = self.unit()

            # 以下リスト 5.4.2 と同じ

    # 他のメソッドは変更しないので省略
```

リスト 5.8.3 には RnnLayer から RnnBaseLayer への変更箇所のみを示す．示した場所以外は、そのままなので、くれぐれも作りこわさないように気をつけるべし．

とまれこれで RnnLayer から RnnBaseLayer への変更は終わるが、そうすると RnnLayer がなくなってしまうが、心配には及ばない．まずは RnnLayer を復活させよう．

リスト 5.8.4 新たな RnnLayer

```
class RnnLayer(RnnBaseLayer):
    def __init__(self, *configuration, **kwargs):
        super().__init__(*configuration, **kwargs)
        m, n = self.config
        super().init_parameter(n, m, n)
        self.unit = RNN_Unit
```

リスト 5.8.4 に新たな RnnLayer を示す．見ての通り RnnBaseLayer を親として継承し、派生クラスとして定義する．そして RnnBaseLayer のコントラクタから削除した以下の 2 行はこちらに記述する．

```
        m, n = self.config
        self.init_parameter(n, m, n)
```

RnnBaseLayer では None とした self.unit は、

```
        self.unit = RNN_Unit
```

で RNN_Unit を指定する．

もともと RnnLayer は RNN_Unit を forward()メソッドを実行する際に呼び出して展開していた．しかし変更後は RnnBaseLayer を継承するクラスのコントラクタで self.unit を指定し、いっぽうその forward()メソッドは self.unit を展開する．新たな RnnLayer は、コントラクタで self.unit = RNN_Unit としているから、もとの RnnLayer と同じく forward()メソッドで展開されるのは RNN_Unit となる．

では続いて GRU を使ったニューロン層を作ろう．

リスト 5.8.5 GRU を使ったニューロン層

```
class GRU(RnnBaseLayer):
    def __init__(self, *configuration, **kwargs):
        super().__init__(*configuration, **kwargs)
        m, n = self.config
        super().init_parameter(n, m, n*3)
        self.unit = GRU_Unit
```

リスト 5.8.5 に示すとおり、リスト 5.8.4 からの変更はわずかだ．GRU_Unit を使うべく、self.unit = GRU_Unit と指定するのは言うまでもない．しかし注意すべきは init_parameter()メソッドに渡す構成を指定する引数の大きさだ．リスト 5.8.1 に GRU_Unit の forward()メソ

ッドを示したが、そこで説明したように、入力は用途ごとに3通りの重み付け和をとるから、それぞれに対応する3組の重みとバイアスが必要であり、それを結合した3倍の幅の重みとバイアスを用意する．そのため init_parameter(n, m, n*3)とする．

GRUを使ったニューロン層が出来たので、これをRNNに組み込もう．リスト5.5.3で基本的なRNNは作ったが、層の指定を変えるだけだ．

リスト5.8.6 GRUを使ったRNN

```
class RNN_gf(RNN_Base):
    def __init__(self, l, m, n, **kwargs):
        super().__init__(l, m, n, **kwargs)
        self.gru_layer = Neuron.GRU(l, m, **kwargs)
        self.neuron_layer = Neuron.NeuronLayer(m, n, **kwargs)
        self.layers.append(self.gru_layer)
        self.layers.append(self.neuron_layer)
```

Embedding Layerを組み込んだはリスト5.6.7で作った．これもGRUを使うようにしよう．

リスト5.8.7 GRUを使いEmbeddingを備えたRNN

```
class RNN_egf(RNN_Base):
    def __init__(self, v, l, m, n, **kwargs):
        super().__init__(v, l, m, n, **kwargs)
        self.embedding_layer = Neuron.Embedding(v, l, **kwargs)
        self.gru_layer = Neuron.GRU(l, m, **kwargs)
        self.neuron_layer = Neuron.NeuronLayer(m, n, **kwargs)
        self.layers.append(self.embedding_layer)
        self.layers.append(self.gru_layer)
        self.layers.append(self.neuron_layer)
```

いずれも簡単に作ることができる．これら以外にも層を多数積上げたRNNを作るのはたやすい．

さあそれではGRUを使ってみよう．リスト5.7.5、リスト5.7.6で、Embedding Layerを備えた基本的なRNNを使って、「羅生門」を学習して文章を生成させた．またリスト5.7.7ではEmbedding Layerがない場合を確かめた．これらにおいてモデルを置き換えて、GRUの動作を確かめよう．RNN_erfとなっているところをRNN_egfとするだけだからリストは省略するが、モデルの生成のほかにパラメータを格納するファイルの指定も忘れずに書き換えよう．
実行結果はどうだろうか？ すでに基本的なRNNで確認済みのことが同じようにうまくいくはずだ．それだけではGRUを導入した意味はないが、もっと長編小説や複数の小説を学習させ

るなどすれば違いが出てくるはずだ．だがそのためには、たとえばGPUを使って学習をより早く実行できる環境など、揃えなければならないことが多く荷が重い．そこでここでは、少しわき道にそれて、ちょっとした細工でより柔軟に文章を生成できるようにしておこう．

リスト5.5.13でRNN用に文字列生成の関数generate_text()を作って、ファイルRNN.pyに置いた．これは出だしを与えて文章を生成するのに使ってきたが、与える出だしの文字あるいは語が変換用の辞書にない場合にkeyエラーしていたことだろう．そこでこれをRNNの性質をうまく使って回避するようにしよう．

リスト5.8.8 keyエラーを回避して文章の生成

```
def generate_text(func, x_to_id, id_to_x, length=100,
                  seed=None, stop=None, print_text=True, end='',
                  stocastic=True, beta=2):
    """ 文字列の生成 x:文字または語 x_chain:その連なり key:次の索引キー """
    x_chain = ""
    vocab_size = len(x_to_id)
    # ― seed無指定なら辞書からランダムに選ぶ ―
    if seed is None:
        seed = random.choices(list(x_to_id))

    for j in range(length):
        # ― 書出しはseedから、その後はfunc出力から ―
        if j < len(seed): # seedの範囲
            x = seed[j]
            try:
                key = x_to_id[x]
            except:
                key = None
        else:             # seedの範囲を超えた
            key = select_category(y, stocastic, beta=beta)
            x = id_to_x[key]
        # ― 綴る ―
        x_chain += x
        if print_text:
            print(x, end=end)
        if x==stop or len(x_chain)>length:
            break
        # ― 次に備える ―
        if key is not None:
            y = func(key)      # seedの範囲を含めて順伝播

    return x_chain
```

リスト 5.8.8 はリスト 5.5.13 とほとんど変わらないが、seed からの書き出しで辞書を参照する際に、try → except で、辞書 x_to_id に x が無い場合には、key を None にする．そして、for ループの最後に次に備える際に、その key が None の場合には順伝播しない．これは何をやっているかというと、はじめ seed を辿って順伝播を行い、RNN に seed の状態を作っていくが、そこに辞書にない文字や語が出てきたら、それは順伝播せずに読み飛ばすのだ．
人が文章を読む場合にも、読めない漢字や意味の分からない語が出てきたときには、それを読み飛ばすのは普通にやっていることだろう．そうして他の部分で補って何とか分かろうとする．もちろん読んでいる文章に重要なキーワードが分からない場合には、文章全体の意味が分からなくなったり、まったく意味を取り違えてしまうこともあるが、それでもすべてを放棄するよりもずっと良い．
それと同じことを RNN にさせようというわけだ．どうだろうか？ これでうまくいくならば、文脈の理解とまでは言えないまでも、それに一歩近づくことになるのではないだろうか？

これを確かめるために、リスト 5.7.6 を細工しよう．リスト 5.8.9 をリスト 5.7.6 の末尾の # ― 文章生成 ― のところに置いて、対話式に出だしを入力して続きの文章を生成するようにしよう．

リスト 5.8.9 文章の生成の置き換え

```
# 省略：リスト 5.7.6 による

# ― 文章生成 ―
while True:
    seed = input('¥n 文章の出だしを入力してください=> ')
    if seed == '':
        print('― テスト終了 ―')
        break
    generate_text(100, seed)
```

ではリスト 5.7.5 で、model や file_name_m の指定を GRU を使うように修正して走らせて学習済みのパラメータをファイルに保存したうえで、上述のリスト 5.7.6 の修正も行って、これを走らせてみよう．
文章の出だしは何でも良いけれど、RNN は羅生門しか学習していないことも考えてやってみた結果のほんの一例を示そう．

文章の出だしを入力してください⇒ どんな人間も死んでしまえば皆
どんな人間も死んでしまえば皆、それが円満に成就した時の、安らかな得意と満足とがあるばかりである。そこで、下人は、老婆を見下しながら、少し声を柔らげてこう云った。
「己《おれ》は検非違使《けびいし》の庁の

GRU を導入した効果を、こんな形で感覚的に味わってみるのも悪くはないだろう．ついでに以下はどうだろう．

文章の出だしを入力してください⇒ こんな形で感覚的に味わってみるのも
こんな形で感覚的に味わってみるのも悪い事とは思わぬぞよ。これとてもやはりせねば、饑死をするじゃて、仕方がなくする事じゃわいの。じゃて、その仕方がない事を、よく知っていたこの女は、大方わしのする事も大目に

羅生門に限らず、多くの文章を学習した RNN がどんな文章を紡ぎ出すのか興味が尽きない．

5.8.4 LSTM

後回しにしたLSTMだが、これはLong Short Term Memoryの略であり、その名のごとく長・短さまざまな期間にわたる依存関係にうまく対応できるようにRNNが進化したものだ．そしてLSTMは、はじめに発表されてからも進化を続け、その間にさまざまな亜種が生まれたから、一通りではなく、GRUもまたその一つとみることもできる．それはさておき、ここではLSTMの代表的なものを実装していくことにしよう．それを図5.8.3に示す．

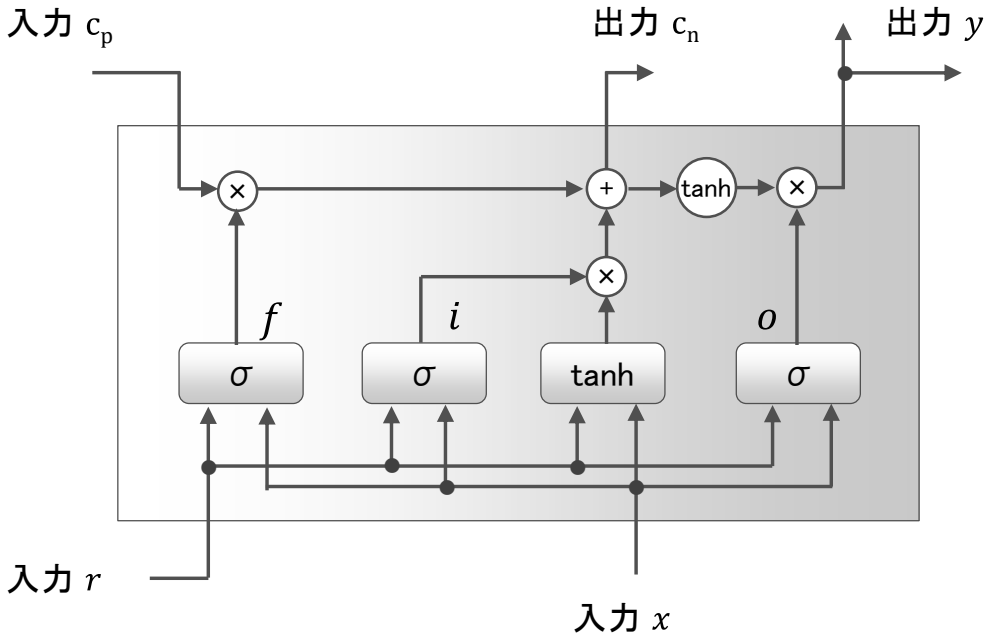

図 5.8.3 LSTM

図5.8.3に示すLSTMもGRU同様に内部にゲートと呼ばれる調整機構を備える．そしてGRUよりもさらに複雑だ．なにより単純なRNNやGRUとは外部のインターフェイスが違い、記憶セルとか単にセルと呼ばれ、cで示される信号が加えられている．こう説明すると、何か特別の記憶素子があるかのように思うがそうではない．このcは図に示すように、図の下方に示す入力xとrからtanhを経て上に向かう経路で求められた値だ．そしてこれは最後に枝分かれして、再びtanhを経て出力yに向かうから、出力に至る途中の値とみることもできる．そしてcは左から c_p が入って、右から c_n が出るようになっている．これはこのユニットの外で出力yから入力rへと戻りパスを繋ぐのと同様に、時系列に沿って順につながれる．そして順につながれることによって、ある時刻の値が、次の時刻へと渡され、それがまた次の時刻へと引き継がれて、いわゆる「記憶」として作用するのだ．これは単純なRNNにもある戻りパスが「記憶」として作用するのを補うように働く．そして入力 c_p から出力 c_n に至る経路が×と

＋で示される単純な処理だけしか通らないから、このユニットを多数つないで長い時系列のデータの処理においても、それに伴う問題が緩和される.

さてあとさきになるが、ざっくり入力から見ていこう．下側の入力 x と入力 r は 4 つの行先がある．もちろんこの 4 つはそれぞれ重み付け和をとったものだ.

tanh に入って上へとつながるのが入力から出力へ至る経路だ．これに対し、σで示されるのは Sigmoid 関数だが、これらは調節するための信号を作り、左から順に f は忘却ゲート、i は入力ゲート、o は出力ゲートと呼ばれる．忘却ゲート f は、左から入る c_p を右から出る c_n にどれだけ伝えるかを調節する．c は記憶セルとして作用するから、それを調節するのが忘却ゲートなのだろう．また入力ゲート i は、tanh からの出力すなわち入力からの情報をどれだけ伝えるかを調節する.

過去から引き継いだ記憶と、入力から作られた新しい記憶は、＋のところで合わされる．そして次に渡す記憶 c となり、さらに tanh を経て再び出力ゲートで調節されて出力 y となる.

ではこれを実装していこう.

リスト 5.8.10 LSTM のニューロンユニット～その 1

```
class LSTM_Unit:
    def forward(self, x, r, cp, w, v, b):  # 入力、前時刻状態
        B, n = r.shape
        u = np.dot(x, w) + np.dot(r, v) + b
        gz = 1 / (1 + np.exp(-u[:, :3*n])) # sigmoid 諸ゲート
        gm = np.tanh(u[:, 3*n:])           # tanh  新しい記憶
        gf = gz[:, :n]                     # 忘却ゲート
        gi = gz[:, n:2*n]                  # 入力ゲート
        go = gz[:, 2*n:]                   # 出力ゲート
        cn = cp * gf + gm * gi             # 旧記憶＊忘却ゲート＋新記憶＊入力ゲート
        y = np.tanh(cn) * go               # 記憶＊出力ゲート
        g = np.hstack((gz, gm))            # 内部状態
        self.state = x, r, cp, y, cn, g
        return y, cn
```

リスト 5.8.10 に LSTM のニューロンユニットの forward()メソッドを示す．先に図 5.8.3 で説明した通りに実装する．ただし GRU でもそうだったが一括してやっても結果が変わらないものは、まとめて処理する．このため重みやバイアス w、v、b は出力の幅の 4 倍の列数を用意している．そしてまとめて重み付け和をとった後で、先頭から 3n までの部分は諸ゲートとして、まとめてシグモイド関数で処理し、末尾の出力の幅 n の分は tanh 関数に入れて新しい記憶としている．諸ゲートは忘却ゲート、入力ゲート、出力ゲートの 3 つ分なので、それぞれ幅 n ずつ切り出して、gf、gi、go としている．あとはそれぞれ図 5.8.3 の通りに調節のための掛算を行って、必要な出力を得るとともに、逆伝播に備えて self.state に保存する.

では続いて逆伝播を実装しよう.

リスト 5.8.11 LSTM のニューロンユニット～その 2

```
class LSTM_Unit:

    def backward(self, grad_y, grad_cn, w, v):
        x, r, cp, y, cn, g = self.state
        B, n = r.shape
        gz = g[:,:3*n]                  # 忘却ゲート、入力ゲート、出力ゲート
        gf = g[:,:n]                    # 忘却ゲート
        gi = g[:,n:2*n]                 # 入力ゲート
        go = g[:,2*n:3*n]               # 出力ゲート
        gm = g[:,3*n:]                  # 新しい記憶
        tanh_c = np.tanh(cn)
        dcn = grad_cn + (grad_y * go) * (1 - tanh_c ** 2)
        dgm = dcn * gi                  # 新しい記憶の勾配

        # 諸ゲートの勾配： 忘却 dgf  入力 dgi  出力 dgo
        dgz = np.hstack((dcn * cp, dcn * gm, grad_y * tanh_c))

        # 諸ゲート sigmoid の微分 と 新しい記憶  tanh の微分
        delta = np.hstack((dgz * gz * (1 - gz), dgm * (1 - gm ** 2)))

        grad_cp = dcn * gf
        grad_w = np.dot(x.T, delta)
        grad_v = np.dot(r.T, delta)
        grad_b = np.sum(delta, axis=0)
        grad_x = np.dot(delta, w.T)
        grad_r = np.dot(delta, v.T)

        return grad_x, grad_r, grad_cp, grad_w, grad_v, grad_b
```

リスト 5.8.11 に LSTM のニューロンユニットの逆伝播のメソッド backward()を示す. 順伝播の forward()メソッドの処理を忠実に逆に辿るが説明は割愛する.

5.8.5 LSTM を組み込む

前項で LSTM のニューロンユニットができた．しかし RNN_Unit や GRU_UNIT には無かった記憶セル c のインターフェイスが付け加えられているから、リスト 5.8.3 で作った RnnBaseLayer の変更が必要だ．

まずは RnnBaseLayer の reset_state()メソッドだが、これはリスト 5.8.12 に示すように記憶セル c に関する初期化を加えるだけだ．

リスト 5.8.12 LSTM を組み込むための RnnBaseLayer の reset_state()メソッドの変更

```
class RnnBaseLayer:

    def reset_state(self):
        self.r0, self.c0 = None, None
```

続いて forward()メソッドだ．

リスト 5.8.13 LSTM を組み込むための RnnBaseLayer の forward()メソッドの変更

```
class RnnBaseLayer:

    def forward(self, x):
        B, T, m = x.shape
        _, n = self.config
        # 時系列の展開の準備をして、リカレントに r0 をセット
        y = np.empty((B, T, n))
        self.layer = []
        rt = np.zeros((B, n)) if self.r0 is None else self.r0
        ct = np.zeros((B, n)) if self.c0 is None else self.c0 # 追加
        # unit を起こしながら、順伝播を繰返す
        for t in range(T):
            unit = self.unit()
            xt = x[:, t, :]
            rt, ct = unit.forward(xt, rt, ct, self.w, self.v, self.b) # 修正
            y[:, t, :] = rt
            self.layer.append(unit)
        self.r0, self.c0 = rt, ct # 修正
        return y
```

リスト 5.8.13 に示すように、forward()メソッドは、記憶セル c に関して追加と修正を行う．戻りパスに関して rt を 0 で初期化する際に ct も同じく初期化する．また for ループで時系列を展開しながらニューロンユニットを順伝播する際に、LSTM で追加された記憶セル c のやり取りが必要だ．そして for ループを抜けた後に次の順伝播に備えて戻りパスの出だし self.r0 にループ最後の出力 rt を設定するが、その際に記憶セルも同様に行う．以上の 3 つだ．

次は backward()メソッドだ.

リスト 5.8.14 LSTM を組み込むための RnnBaseLayer の backward()メソッドの変更

```
class RnnBaseLayer:

    def backward(self, grad_y):
        B, T, n = grad_y.shape
        m, _ = self.config
        self.grad_w = np.zeros_like(self.w)
        self.grad_v = np.zeros_like(self.v)
        self.grad_b = np.zeros_like(self.b)
        self.grad_x  = np.empty((B, T, m))
        grad_rt, grad_ct = 0, 0 # 修正
        for t in reversed(range(T)):
            unit = self.layer[t]
            grad_yt = grad_y[:, t, :] + grad_rt # 出力からとリカレントからの勾配を合算

            grad_xt, grad_rt, grad_ct, grad_wt, grad_vt, grad_bt = ¥
                    unit.backward(grad_yt, grad_ct, self.w, self.v)      # 修正

            self.grad_w += grad_wt
            self.grad_v += grad_vt
            self.grad_b += grad_bt
            self.grad_x[:, t, :] = grad_xt
        self.grad_r0, self.grad_c0 = grad_rt, grad_ct # 修正
        return self.grad_x
```

backward()メソッドも forward()メソッド同様に、記憶セルに関する処理を加える. ループの前の初期化、ループ中にニューロンユニットの逆伝播でのやり取り、そしてループを抜けた後で次に備えて勾配を設定の 3 つの修正を行う.

LSTM を組み込むために RnnBaseLayer を修正したから、RNN_Unit や GRU も変更が必要になる.
しかし内部の動作が変わるわけではなく、引数と返り値のつじつま合わせをすれば良い.
まずはリスト 5.4.1 で作った RNN_unit の修正だ.

リスト 5.8.15 RNN_Unit の修正

```
class RNN_Unit:
    def forward(self, x, r, c, w, v, b):
        u = np.dot(x, w) + np.dot(r, v) + b
        y = np.tanh(u)
        self.state = x, r, y
        return y, c

    def backward(self, grad_y, grad_c, w, v):
        x, r, y = self.state
        delta = grad_y * (1 - y**2)  # tanh の逆伝播
        grad_w = np.dot(x.T, delta)
        grad_v = np.dot(r.T, delta)
        grad_b = np.sum(delta, axis=0)
        grad_x = np.dot(delta, w.T)
        grad_r = np.dot(delta, v.T)
        return grad_x, grad_r, grad_c, grad_w, grad_v, grad_b
```

リスト 5.8.15 に示すように forward()メソッドでは引数に c が加え、その加えた c をそのまま返り値に加える. つまり何も変わらないがインターフェイスだけを RnnBaseLayer の変更に合わせる. リスト 5.8.13 に示す RnnBaseLayer の forward()メソッドでは、ct は値 0 で初期化し、その後 for ループの中で unit.forward()で時系列を展開してニューロンユニットを順伝播する. このとき ct は引数で与えて、返り値で上書きする. だから、RNN_Unit の forward()メソッドを、リスト 5.8.15 のようにすれば、値 0 の ct が時系列の順繰りに渡っていくだけで動作に影響しない.
backward()メソッドも同様だ. 引数に grad_c を加え、それをそのまま返り値に加える. これも何も動作に影響しない.

続いて GRU_UNIT の修正だが、これも RNN_Unit と全く同様だ. 修正するのは forward()メソッドと backward()メソッドの引数と返り値だけなので、途中はすべて省略して、リスト 5.8.16 に示す.

リスト 5.8.16 GRU_UNIT の修正

```
class GRU_Unit:
    def forward(self, x, r, c, w, v, b):

        # 途中略：リスト 5.8.1 のまま

        return y, c

    def backward(self, grad_y, grad_c, w, v):

        # 途中略：リスト 5.8.2 のまま

        return grad_x, grad_r, grad_c, grad_w, grad_v, grad_b
```

もはや説明は不要だろう．しかし適宜リスト 5.7.5 やリスト 5.7.7 を走らせて、従来の機能が維持されていることを確かめておこう．

では準備ができたから、LSTM を使ったニューロン層を作ろう．リスト 5.8.4 の RnnLayer やリスト 5.8.5 の GRU とほとんど同じで、init_parameter()メソッドに渡す構成を重みやバイアスが 4 組分とすることだけに注意すれば良い．

リスト 5.8.17 LSTM を使ったニューロン層

```
class LSTM(RnnBaseLayer):
    def __init__(self, *configuration, **kwargs):
        super().__init__(*configuration, **kwargs)
        m, n = self.config
        super().init_parameter(n, m, n*4)
        self.unit = LSTM_Unit
```

ではこれを RNN に組み込もう．リスト 5.8.6 およびリスト 5.8.7 で GRU を使った RNN を作ったのと同様だ．

リスト 5.8.18 LSTM を使った RNN

```
class RNN_lf(RNN_Base):
    def __init__(self, l, m, n, **kwargs):
        super().__init__(l, m, n, **kwargs)
        self.lstm_layer = Neuron.LSTM(l, m, **kwargs)
        self.neuron_layer = Neuron.NeuronLayer(m, n, **kwargs)
        self.layers.append(self.lstm_layer)
        self.layers.append(self.neuron_layer)
```

リスト 5.8.19 LSTM を使い Embedding を備えた RNN

```
class RNN_elf(RNN_Base):
    def __init__(self, v, l, m, n, **kwargs):
        super().__init__(v, l, m, n, **kwargs)
        self.embedding_layer = Neuron.Embedding(v, l, **kwargs)
        self.lstm_layer = Neuron.LSTM(l, m, **kwargs)
        self.neuron_layer = Neuron.NeuronLayer(m, n, **kwargs)
        self.layers.append(self.embedding_layer)
        self.layers.append(self.lstm_layer)
        self.layers.append(self.neuron_layer)
```

リスト 5.8.18 とリスト 5.8.19 も説明は不要だろう.

LSTM も GRU 同様に動かしてみよう. さあ今度はなにを学習させて何を綴らせようかと、迷わずにはいられない. とりあえず LSTM を使い Embedding を備えた RNN で「羅生門」を学習させて、文章を綴ってみた一例を示しておく.

文章の出だしを入力してください=> さあ今度は何を綴らせようか
さあ今度は何を綴らせようか盗人になるかに、迷わなかったばかりではない。その時のこの男の心もちから云えば、饑死などと云う事は、ほとんど、考える事さえ出来ないほど、意識の外に追い出されていた。
「きっと、そう

もはや何をか言わん. 別の小説や文章を RNN に読ませたり、複数のそれを併せて読ませたりして、いろいろやってみてほしい.

[おわりに]

第5章も長い道のりとなってしまった．しかしマルコフ連鎖の簡単なプログラムからはじめて、RNNに至り、さらにGRUやLSTMも含めて実装するところまで辿ってきた．

文章を扱うことを軸にしてきたがRNNの利用はそれにとどまらない．たとえば音は波であり、音楽は連続する波が奏でるのだから、やはりRNNで扱うことが出来る．また株価の変動も同様に時系列データの作る波として見て、RNNで予測を行うことが出来る．またたとえば、一部が隠された画像の隠された部分を予測するのも、画像を時系列データとして扱うことで可能だ．さらにRNNを組み合わせてチャットボットを作ったり、CNNとRNNの組み合わせで画像にコメントをつけたり、その反対に文字で書かれたことを画像にしたり、といったこともされている．

これらを実現するには、まだまだ、やらなければならないことは多いが、そこに至る道筋の大きな一歩といえるだろう．予測というものが過去の蓄積の上に可能となる、それ故に、予測は外れるのが宿命であることを思い知らされざるを得ない一方で、未来をデジャブのように見る驚きに導かれることもないわけではない、そんなことを予感しながら取り組むのも良いだろう．

ファイル	クラス 関数	メソッド	リスト番号
Neuron.py			
	class Config		4.1.8
	class BaseLayer		
		def __init__()	4.1.1→5.3.1
		def init_parameter()	4.1.7→5.3.2
		def update()	4.4.4→5.3.3
	class NeuronLayer		
		def __init__()	4.1.4
		def fix_configuration()	4.1.6
		def forward()	4.1.5
		def backward()	4.4.3
	class RNN_Unit		
		def forward()	5.4.1→5.8.15
		def backward()	5.4.1→5.8.15
	class GRU_Unit		
		def forward()	5.8.1→5.8.16
		def backward()	5.8.2→5.8.16
	class LSTM_Unit		
		def forward()	5.8.10
		def backward()	5.8.11
	class RnnLayer→class RnnBaseLayer		→5.8.3
		def __init__()	5.4.4→5.8.3
		def init_parameter()	5.4.5
		def forward()	5.4.2→5.8.3→5.8.13
		def backward()	5.4.3 →5.8.14
		def update()	5.4.6
		def reset_state()	5.4.7 →5.8.12
	class RnnLayer		→5.8.4
		def __init__()	5.8.4
	class GRU		
		def __init__()	5.8.5
	class LSTM		
		def __init__()	5.8.17
	class Embedding		
		def __init__()	5.6.5
		def init_parameter()	5.6.6
		def forward()	5.6.4
		def backward()	5.6.4
		def update()	5.6.4

ファイル	クラス 関数	メソッド	リスト番号
NN.py			
	class NN_CNN_Base		
		def __init__()	4.6.1
		def forward()	4.6.2
		def backward()	4.6.3
		def summary()	4.6.5
		def update()	4.6.4
		def export_params()	5.3.13
		def import_params()	5.3.14
		def save_parameters()	5.3.15
		def load_parameters()	5.3.16
	class NN_0		
		def __init__()	4.6.6
	class NN_m		
		def __init__()	4.6.12
GAN0.py			
	class GAN_m_m		
		def __init__()	4.9.6
RNN.py			
	class RNN_Base		
		def __init__()	5.5.1
		def reset_state()	5.5.2
	class RNN_rf		
		def __init__()	5.5.3
	class RNN_gf		
		def __init__()	5.8.6
	class RNN_lf		
		def __init__()	5.8.18
	class RNN_erf		
		def __init__()	5.6.7
	class RNN_egf		
		def __init__()	5.8.7
	class RNN_elf		
		def __init__()	5.8.19
	def generate()		5.5.9
	def generate_text()		5.5.13→5.8.8

ファイル	クラス 関数	メソッド	リスト番号
Activators.py			
	class Identity		
		def forward()	4.1.2
		def backward()	4.4.2
	class Step		
		def __init__()	4.1.9
		def backward()	4.1.9
	class Sigmoid		
		def forward()	4.6.8
		def backward()	4.6.8
	class Softmax		
		def forward()	4.7.3
		def backward()	4.7.3
	class ReLU		
		def forward()	4.9.1
		def backward()	4.9.1
	class LReLU		
		def __init__()	4.9.2
		def forward()	4.9.2
		def backward()	4.9.2
	class Tanh		
		def forward()	4.9.4
		def backward()	4.9.4
LossFunctions.py			
	class MeanSquaredError		
		def forward()	4.3.1→4.4.1
		def backward()	4.4.1
	class CrossEntropyError		
		def forward()	4.7.4
		def backward()	4.7.4
	class CrossEntropyError2		
		def forward()	4.9.5
		def backward()	4.9.5

ファイル	クラス 関数	メソッド	リスト番号
Optimizer.py			
	class OptimizerBase		
		def __init__()	5.3.4
		def update()	5.3.4
	class GradientClipping		
		def __call__()	5.3.5
	class SGD		
		def __init__()	5.3.7
		def __call__()	5.3.7
	class Momentum		
		def __init__()	5.3.8
		def __call__()	5.3.8
	class RMSProp		
		def __init__()	5.3.9
		def __call__()	5.3.9
	class AdaGrad		
		def __init__()	5.3.10
		def __call__()	5.3.10
	class Adam		
		def __init__()	5.3.11
		def __call__()	5.3.11

ファイル	クラス 関数	メソッド	リスト番号
common_function.py			
	def eval_in_module()		4.1.3
	def get_accuracy()		4.4.5
	def graph_for_error()		4.4.6
	def test_sample()		4.5.3
	def split_train_test()		4.5.3
	def normalize()		4.7.5
	class Mesurement		
		def __init__()	4.7.2
		def __call__()	4.7.2
		def progress()	4.7.2
	def mesurement()		4.7.2
	class Mesurement_for_GAN		
		def __init__()	4.9.7
		def __call__()	4.9.7
		def progress()	4.9.7
	def generate_random_images()		4.9.10
	def split_by_janome()		5.1.8
	def preprocess()		5.2.1
	def arrange_time_data()		5.7.1

ファイル	クラス 関数	メソッド	リスト番号
Iris.py			
	def get_data()		4.2.1
	def label_list()		4.2.1
Digits.py			
	def get_data()		4.5.1
	def show_sample()		4.5.1
OlivettiFaces.py			
	def get_data()		4.5.2
	def label_list()		4.5.2
	def show_sample()		4.5.2
CIFER10.py			
	def unpickle()		4.8.1
	def read_batch()		4.8.1
	def load_data()		4.8.1
	def get_data()		4.8.4
	def label_list()		4.8.2
	def show_sample()		4.8.3
Markov.py			
	def make_markov()		5.1.11
	def generate_text()		5.1.12
markov_n.py			
	def select_category()		5.2.3
	def generate_text()		5.2.2