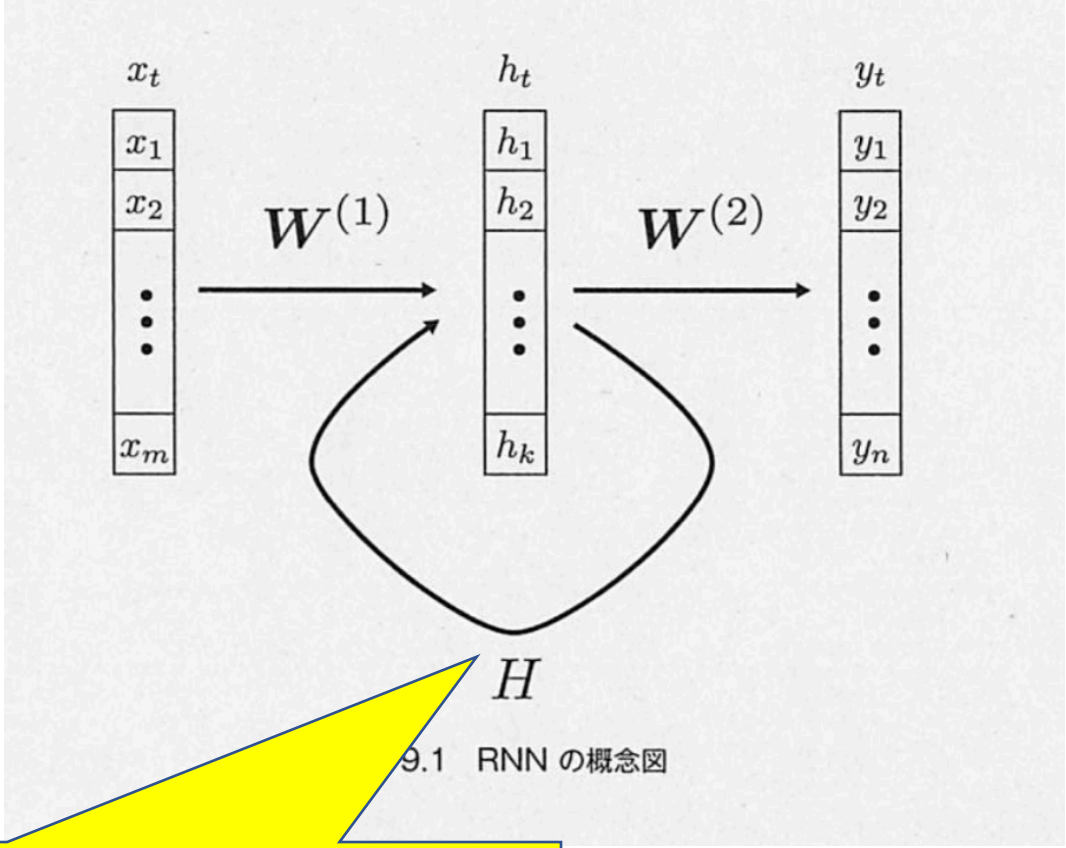
第9章

Recurrent Neural Network

CAO RUI

9.1 時系列データに対するRNN

- 入力：時刻$t=1$から$t=T$
- までの時系列データx_1
- ，x_2，…，x_T
- 出力：各時刻tに対す
- る出力y_tの列y_1，y_2，
- …，y_T



9.1 RNN の概念図

中間層のHの部分が再帰的になっていることが特徴

9.1 時系列データに対するRNN

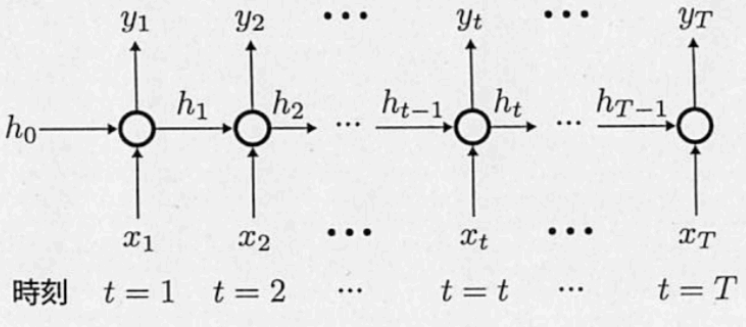


図 9.2　時間で展開した RNN のネットワーク

入力：一度にx_1，x_2，…，x_Tが入力ではなく，x_iは時刻tに沿って順番に入力されます，時刻tにおけるネットワークへの入力は，入力データx_tと時刻tにおける中間層への入力h_{t-1}である

出力：通常のy_tと時刻$t+1$における中間層への入力となるh_tである

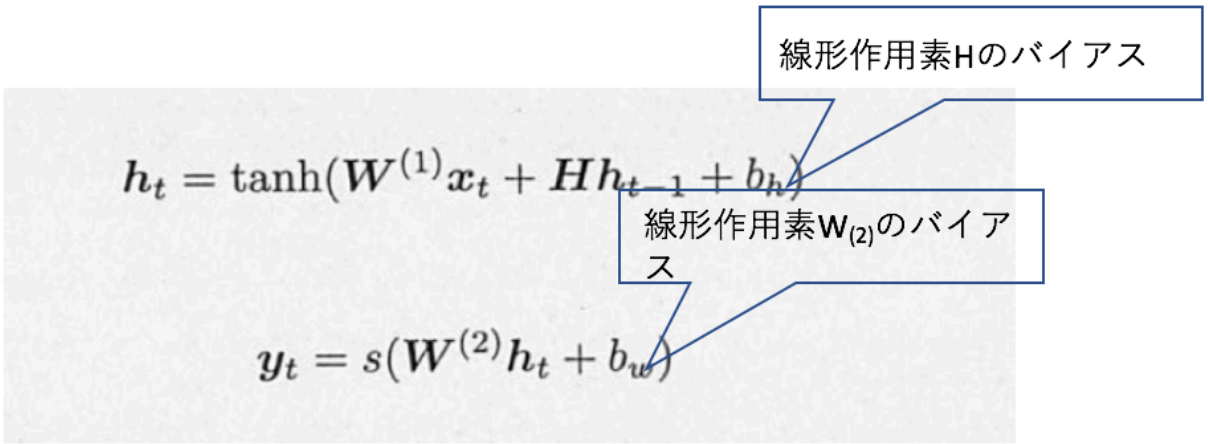
$$h_t = \tanh(W^{(1)}x_t + Hh_{t-1} + b_h)$$

$$y_t = s(W^{(2)}h_t + b_y)$$

線形作用素Hのバイアス

線形作用素$W_{(2)}$のバイアス

9.1 時系列データに対するRNN



この式によってx_1に対してy_1とh_1を得られます，次にx_2に対して，同じことをします，これを繰り返していくと，$y_1, y_2, \cdots, y_T$が得られます

次に，Hを利用して，以前の時系列のデータの情報を圧縮した形で次の時刻に引き継いで行く

学習では誤差の累積から誤差逆伝播を用います．まず，各x_iに対する教師信号t_iとしておきます-> t_iとy_iから誤差を求めます．これらの誤差を累積を求めて，最後に誤差逆伝播法を用いて，$W^{(1)}, W^{(2)}$およびHを求めて行く

9.2 言語モデル

- 言語モデルとは，文 s が現れる確率 $P(s)$ を与える確率モデルである

$$
\begin{aligned}
P(s) &= P(w_1, w_2 \cdots w_N) \\
&= P(w_1)P(w_2|w_1)P(w_3|w_1w_2)\cdots P(w_N|w_1w_2\cdots w_{N-1}) \\
&= P(w_1)\prod_{t=2}^{N} P(w_t|w_1w_2\cdots w_{t-1})
\end{aligned}
$$

ある単語列が与えられる時に次に現れる単語を予測する

9.2 言語モデル

- 言語モデルが得られて何か良いことがあるのか

$s =$ “生命ほけんを解約する。”

P(“生命保険を解約する。”)

P(“生命保健を解約する”)

確率を比較して高いほうを選ぶこと

9.3 RNNLMのネットワーク

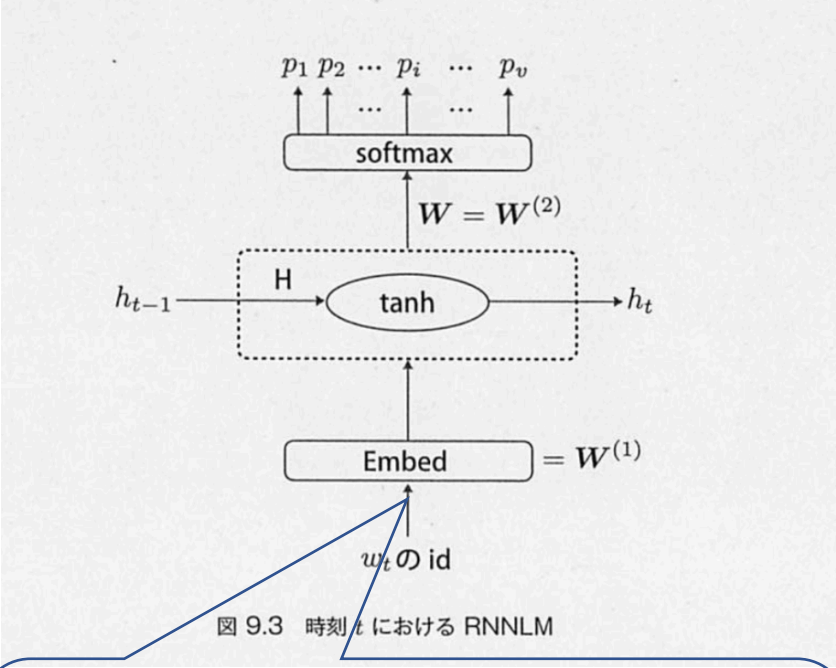


図 9.3　時刻 t における RNNLM

第 1 層目の線形作用素 $W^{(1)}$ に

Embedを使っているのがポイント

単語の種類数をvとすると，出力y_tは以下のv次元のベクトルです：

$y_t = (y_1, y_2, \cdots y_i \cdots, y_v)$

y_iは単語id i の単語が出現する確率 -> 出力層からの出力にはsoftmax関数をかぶせること

9.4 Chainerによる RNNLMの実装

- 素のRNNのプログラムの作り方：
- 必要なパッケージをインポートする：
- import numpy as np
- import chainer
- from chainer import cuda, Function, Variable, optimiziers, serializers, utils
- from chainer import Link, Chain, ChainList
- improt chainer.function as F
- import chainer.links as L

9.4 Chainerによる RNNLMの実装

- コーパスを読み込んで単語にidを付けるなどの事前処理:

```
>>> vocab = {}
>>> def load_data(filename):
>>>     global vocab
>>>     words = open(filename).read()\
...                 .replace('\n', '<eos>').strip().split()
>>>     dataset = np.ndarray((len(words),), dtype=np.int32)
>>>     for i, word in enumerate(words):
```

```
>>>         if word not in vocab:
>>>             vocab[word] = len(vocab)
>>>         dataset[i] = vocab[word]
>>>     return dataset
>>>
>>> train_data = load_data('ptb.train.txt')
>>> eos_id = vocab['<eos>']
```

9.4 Chainerによる RNNLMの実装

- モデルの部分：

```
class MyRNN(chainer.Chain):
    def __init__(self, v, k):
        super(MyRNN, self).__init__(
            embed = L.EmbedID(v, k),
            H  = L.Linear(k, k),
            W = L.Linear(k, v),
        )
    def __call__(self, s):
        accum_loss = None
        v, k = self.embed.W.data.shape
        h = Variable(np.zeros((1,k), dtype=np.float32))
        for i in range(len(s)):
            next_w_id = \
                eos_id if (i == len(s) - 1) else s[i+1]
            tx = Variable(np.array([next_w_id], \
                dtype=np.int32))
            x_k = self.embed(Variable(np.array([s[i]], \
                dtype=np.int32)))
```

```
        h = F.tanh(x_k + self.H(h))
        loss = F.softmax_cross_entropy(self.W(h),tx)
        accum_loss = loss if accum_loss is None \
                          else accum_loss + loss
    return accum_loss
```

- 最適化アルゴリズムの設定：

```
demb = 100
model = MyRNN(len(vocab), demb)
optimizer = optimizers.Adam()
optimizer.setup(model)
```

- 学習する部分：

```
for epoch in range(5):
    s = []
    for pos in range(len(train_data)):
        id = train_data[pos]
        s.append(id)
        if (id == eos_id):
            model.cleargrads()
            loss = model(s)
            loss.backward()
            optimizer.update()
            s = []
    outfile = "myrnn-" + str(epoch) + ".model"
    serializers.save_npz(outfile, model)
```

9.5 言語モデルの評価

- 言語モデルの評価には，一般にパープレキシティが使われます．モデルMのエントロピーがHである時，パープレキシティは2^Hとなる，Hの定義は以下になる：

$$H = \frac{1}{|D|} \sum_{i=1}^{|D|} -\log_2(P(w_i|M))$$

- 上記の関数を使って，パープレキシティの計算を行います．

9.5 言語モデルの評価

```
>>> test_data = load_data('ptb.test.txt')
>>> test_data = test_data[0:1000]
>>> model = MyRNN(len(vocab), demb)
>>> serializers.load_npz('myrnn.model', model)
>>> sum = 0.0
>>> wnum = 0
>>> s = []
>>> unk_word = 0
>>> for pos in range(len(test_data)):
>>>     id = test_data[pos]
>>>     s.append(id)
>>>     if (id > max_id):
>>>         unk_word = 1
>>>     if (id == eos_id):
>>>         if (unk_word != 1):
>>>             ps = cal_ps(model, s)
>>>             sum += ps
>>>             wnum += len(s) - 1
>>>         else:
>>>             unk_word = 0
>>>         s = []
>>> print math.pow(2, sum / wnum)
```

- 結果:

```
> python eval-rnn.py myrnn-0.model
208.103092604
> python eval-rnn.py myrnn-1.model
174.181462185
> python eval-rnn.py myrnn-2.model
164.917536858
> python eval-rnn.py myrnn-3.model
160.432661616
> python eval-rnn.py myrnn-4.model
157.165286105
```

- 徐々にパープレキシティが減少するのは確認できました

9.6 LSTM

- 素のRNNでは系列が長くなり深いネットワークになると誤差逆伝播のアルゴリズムでは勾配が消失したり発散したりする問題が発生します，その結果，長期依存をうまく扱えません，この点を改良したのがLSTMです

- LSTMブロックの中身を隠蔽してしまうと，LSTMは素のRNNと同じ形になります

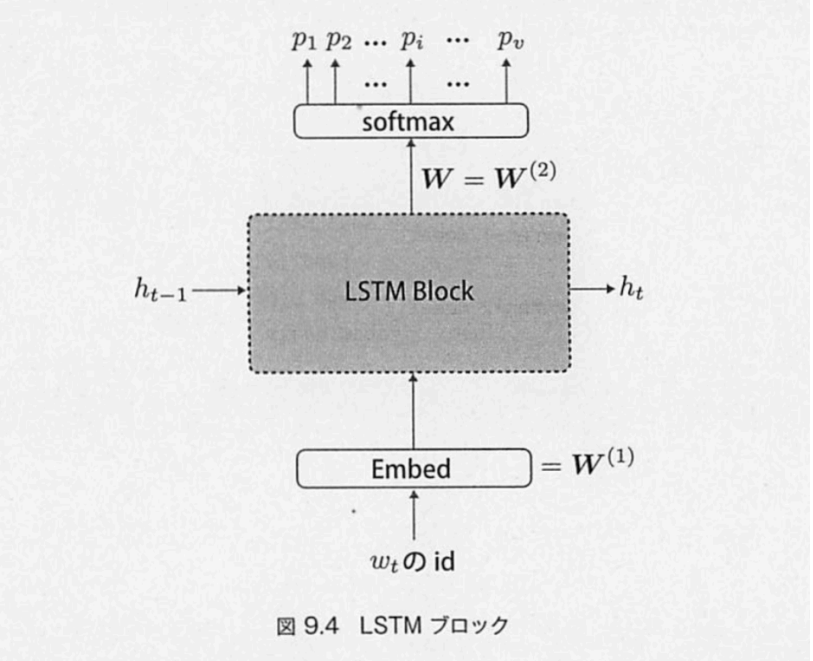


図 9.4　LSTM ブロック

- ブロックの中身を構成する要素：
- 記憶セル，入力ゲート，出力ゲート，忘却ゲートの 4 つである

9.6 LSTM

- x_t： 時刻tにおける*LSTM*ブロックへの入力は１つ下の層
- h_{t-1}:時刻$t-1$いおける*LSTM*ブロックの出力
- 素の*RNN*と同じく，以下のように変換されます：

$$\bar{z}_t = W_z x_t + R_z h_{t-1} + b_z$$

$$z_t = \tanh(\bar{z}_t)$$

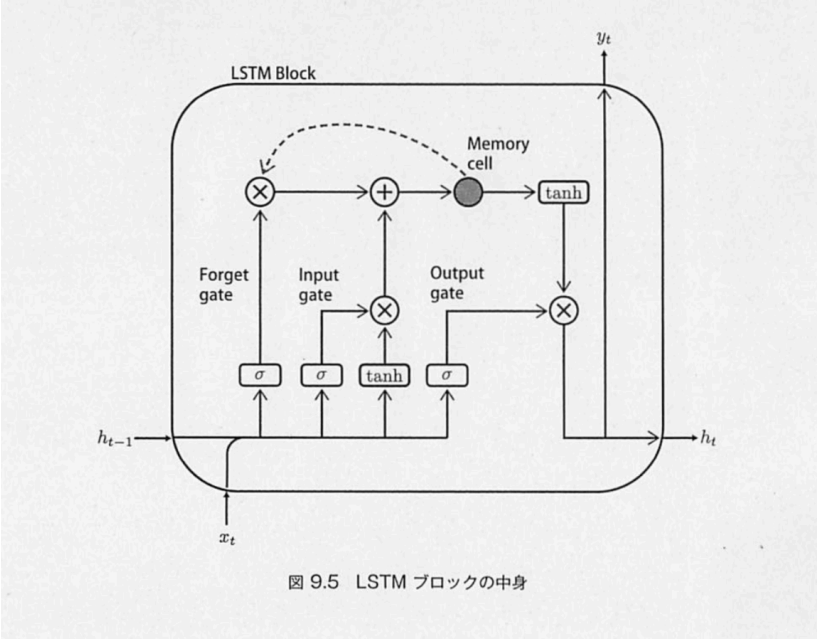


図 9.5 LSTM ブロックの中身

x_t： *LSTM*ブロックへの第１層からの入力

y_t： *LSTM*の出力

9.6 LSTM

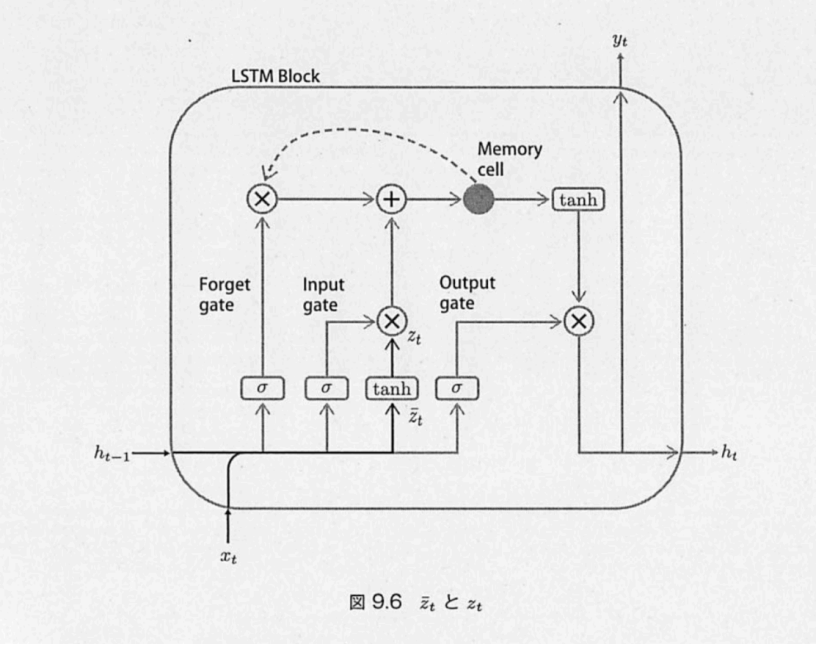


図 9.6 $\bar{z}_t$ と z_t

入力ゲートにおける変換：

$$\bar{i}_t = W_i x_t + R_i h_{t-1} + b_i$$

$$i_t = \sigma(\bar{i}_t)$$

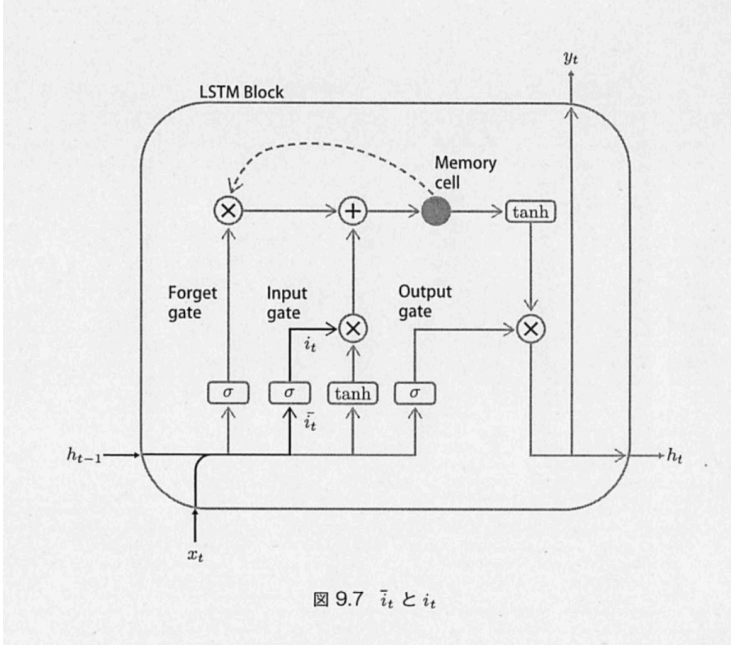


図 9.7 $\bar{i}_t$ と i_t

9.6 LSTM

- 忘却ゲートにおける変換：

$$\bar{f}_t = W_f x_t + R_f h_{t-1} + b_f$$
$$f_t = \sigma(\bar{f}_t)$$

- 記憶セルにおける変換☹
  （LSTMのポイントです）

$$c_t = i_t \otimes z_t + f_t \otimes c_{t-1}$$

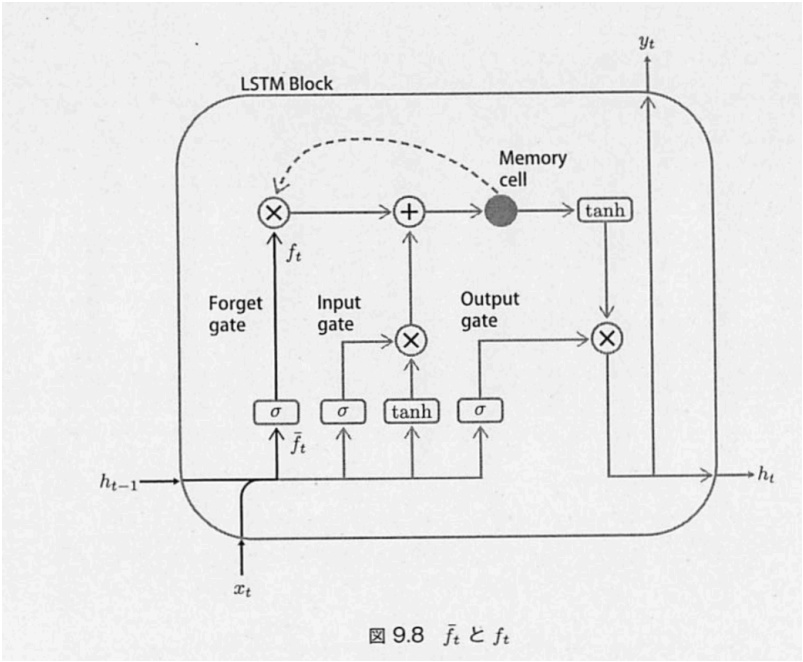


図 9.8 $\bar{f}_t$ と f_t

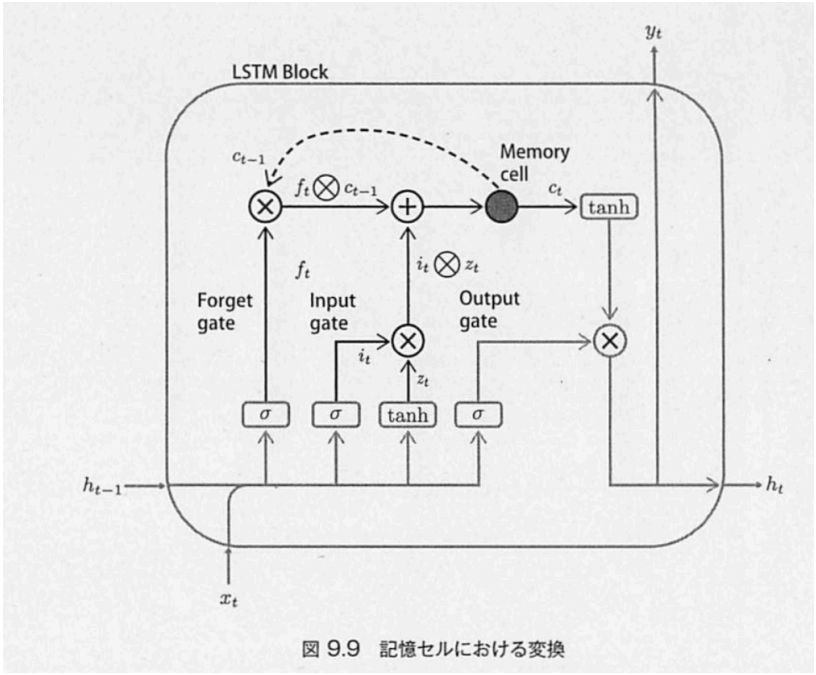


図 9.9 記憶セルにおける変換

9.6 LSTM

- 出力ゲートにおける変換：

$$\bar{o}_t = W_o x_t + R_o h_{t-1} + b_o$$
$$o_t = \sigma(\bar{o}_t)$$

- LSTMブロックの出力：

$$y_t = h_t = o_t \otimes \tanh(c_t)$$

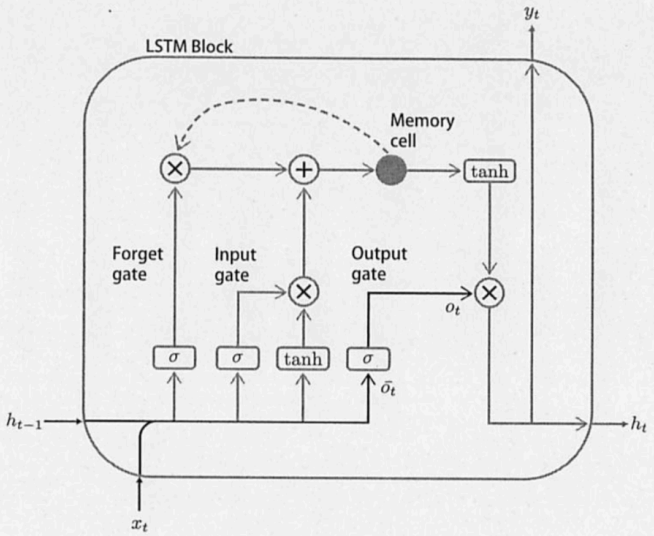


図 9.10 $\bar{o}_t$ と o_t

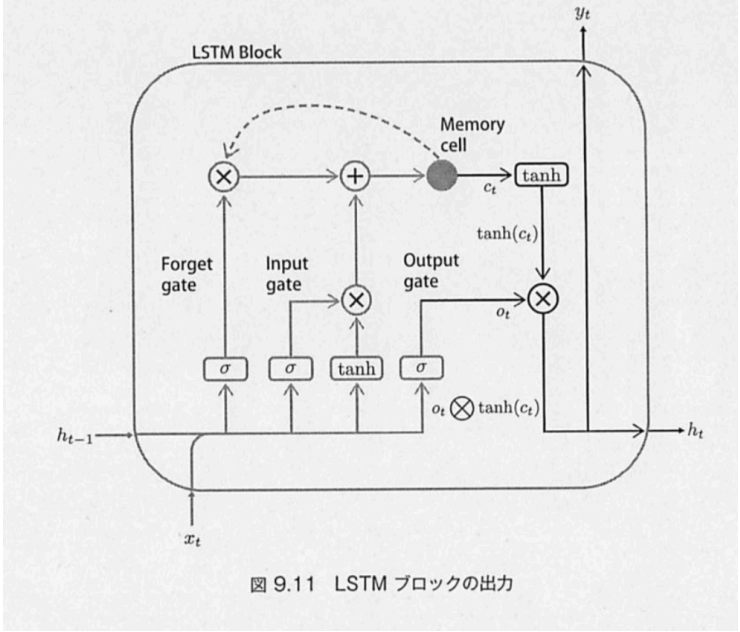


図 9.11 LSTM ブロックの出力

9.7 ChainerによるLSTMの実装

- まず，パラメータのある関数を___init___部分で定義します：

```
class MyLSTM(chainer.Chain):
    def __init__(self, v, k):
        super(MyLSTM, self).__init__(
            embed = L.EmbedID(v, k),
            Wz = L.Linear(k, k),
            Wi = L.Linear(k, k),
            Wf = L.Linear(k, k),
            Wo = L.Linear(k, k),
            Rz = L.Linear(k, k),
            Ri = L.Linear(k, k),
            Rf = L.Linear(k, k),
            Ro = L.Linear(k, k),
            W = L.Linear(k, v),
        )
```

- 損失関数___call___部分で定義する：

```
class MyLSTM(chainer.Chain):
    def __init__(self, v, k):
        <前述しているので省略>

    def __call__(self, s):
        accum_loss = None
        v, k = self.embed.W.data.shape
        h = Variable(np.zeros((1,k), dtype=np.float32))
        c = Variable(np.zeros((1,k), dtype=np.float32))
        for i in range(len(s)):
            next_w_id = eos_id if (i == len(s) - 1) \
                               else s[i+1]
            tx = Variable(np.array([next_w_id], \
                               dtype=np.int32))
            x_k = self.embed(Variable(np.array([s[i]], \
                               dtype=np.int32)))
            z0 = self.Wz(x_k) + self.Rz(h)
            z1 = F.tanh(z0)
            i0 = self.Wi(x_k) +  self.Ri(h)
            i1 = F.sigmoid(i0)
            f0 = self.Wf(x_k) +  self.Rf(h)

            f1 = F.sigmoid(f0)
            c = i1 * z1 + f1 * c
            o0 = self.Wo(x_k) +  self.Ro(h)
            o1 = F.sigmoid(o0)
            h = o1 * F.tanh(c)
            loss = F.softmax_cross_entropy(self.W(h), tx)
            accum_loss = loss if accum_loss is None \
                              else accum_loss + loss
        return accum_loss
```

9.7 ChainerによるLSTMの実装

- 学習する部分：

```
for epoch in range(5):
    s = []
    for pos in range(len(train_data)):
        id = train_data[pos]
        s.append(id)
        if (id == eos_id):
            model.cleargrads()
            loss = model(s)
            loss.backward()
            if (len(s) > 29):              # 文の長さ30以上で
                loss.unchain_backward()  # unchainを実行
            optimizer.update()
            s = []
    outfile = "lstm0-" + str(epoch) + ".model"
    serializers.save_npz(outfile, model)
```

- 学習に対しての改善：
- 入力ゲートと忘却ゲートに入ってくるの出力$ht-1$にドロップアウトというノイズを挿入する関数$F.dropout$をかぶせることで，学習が改善されます
- パラメータの減って，学習の時間を減少できます

9.8 システムから提供されている関数の利用

素のRNNでは，中間層の再帰部分に，以下のような線形変換を使います：

$$H = L.Linear(k, k)$$

この部分が以下になるだけです：

$$H = L.LSTM(k, k)$$

L.LSTMの第 1 引数はLSTMブロックへの入力のベクトルの次元数，第 2 引数はLSTMブロックから出力のベクトルの次元数である

9.9 GRU

- 勾配消失問題に対して，LSTMよりもも少し単純な解決法である

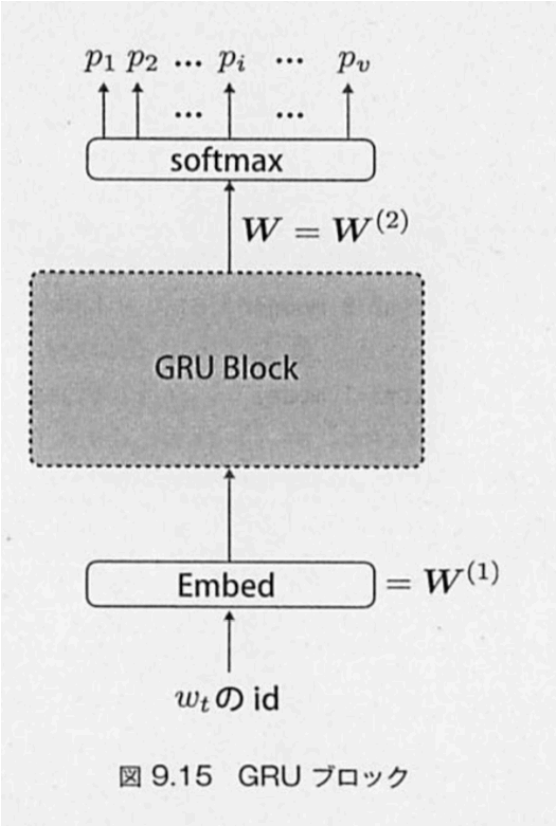


図 9.15　GRU ブロック

プログラムも単にL.LSTMの部分をL.StatefulGRUに変更するだけです

H = L.StatefulGRU(k, k)

9.10 RNNのミニバッチ処理

- データの集合の中の最も長さの長いデータに他のデータの長さを合わせることです．長さが足りない部分には0ベクトルとなるデータを入れておきます

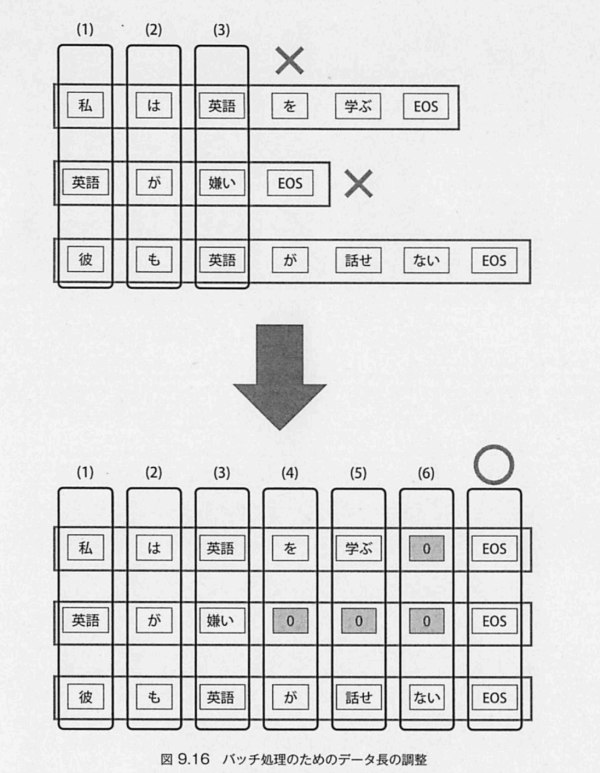


図 9.16　バッチ処理のためのデータ長の調整

モデルの部分では，*ignore_label*のオプションに*zero_id*を設定する

```
class MyLSTM(chainer.Chain):
    def __init__(self, v, k):
        super(MyLSTM, self).__init__(
            embed = L.EmbedID(v, k, ignore_label = zero_id),
            H = L.LSTM(k, k),
            W = L.Linear(k, v),
        )
    def __call__(self, s):
        <ここはlstm2.pyと同じなので省略>
```

9.11 NStepLSTMによるミニバッチ処理

- モデルの部分：

```
class MyLSTM(chainer.Chain):
    def __init__(self, layer, v, k, dout):
        super(MyLSTM, self).__init__(
            embed = L.EmbedID(v, k),
            H = L.NStepLSTM(layer, k, k, dout),
            W = L.Linear(k, v),
        )
    def __call__(self, hx, cx, xs, t):
        accum_loss = None
        xembs = [ self.embed(x) for x in xs ]
        xss = tuple(xembs)
        hy, cy, ys = self.H(hx, cx, xss)
        y = [self.W(item) for item in ys]
        for i in range(len(y)):
            tx = Variable(np.array(t[i], dtype=np.int32))
            loss = F.softmax_cross_entropy(y[i], tx)
            accum_loss = loss if accum_loss is None \
                              else accum_loss + loss
        return accum_loss
```

LSTMとの大きな違いは，L.LSTMの代わりにL.NStepLSTMを使うところです，第１引数は層の数です．第２引数，第３引数はL.LSTMの第１引数，第２引数に対応するもので，第４引数のdoutはLSTMで使うdropoutの比率です．

```
H = L.NStepLSTM(layer, k, k, dout)
```

9.11 NStepLSTMによるミニバッチ処理

- 学習の部分：

```
for pos in range(len(train_data)):
    id = train_data[pos]
    if (id != eos_id):
        s += [ id ]
    else:
        bc += 1
        next_s = s[1:]
        next_s += [ eos_id ]
        xs += [ np.asarray(s, dtype=np.int32) ]
        t += [ np.asarray(next_s, dtype=np.int32) ]
        s = []
        if (bc == 10):
            model.cleargrads()
            hx = Variable(np.zeros((layer, len(xs), demb), \
                    dtype=np.float32))
            cx = Variable(np.zeros((layer, len(xs), demb), \
                    dtype=np.float32))
            loss = model(hx, cx, xs, t)
            loss.backward()
            optimizer.update()
            xs = []
            t = []
            bc = 0
```

比較：

表 9-1　ミニバッチの効果

lstm2.py	lstm2-minibatch.py	nsteplstm.py
約 343 分	約 68 分	約 33 分