

ラビット・チャレンジ

<実装演習レポート 深層学習後編(day3, day4)>

メールアドレス : mtop.jp@gmail.com

受講者名 : 山崎 英和

受講種別 : ラビット・チャレンジ

## ■ 深層学習後編(day3)

<Section1: 再帰型ニューラルネットワークの概念>

### ◆ 要点まとめ

#### 1-1 RNN 全体像

再帰型ニューラルネットワーク (Recurrent Neural Network : 以下、RNN) とは、**時系列データに対応可能なニューラルネットワーク(NN)**である。時系列データとは、時間的順序を追って一定間隔ごとに観察されて、しかも**相互に統計的依存関係が認められる**ようなデータの系列。

例：音声データ、テキストデータ、株価のデータ…etc.

図 1 のニューラルネットワーク構造図とおり、基本的な構造はニューラルネットワークと同じで、入力層、中間層、出力層からなる。しかし、RNN の中間層では、時系列データを扱えるように中間層の出力を、再度、中間層の入力へ入れることにより、**過去の時系列データの情報を保持(時間的に記憶)しながら最終的な出力方向へ順次伝搬**することを可能にしている。RNN の模式図を図 1-2 に示す。

このように時系列モデルを扱うには、初期の状態と過去の時間  $t-1$  の状態を保持し、そこから次の時間  $t$  を再帰的に求める再帰構造が必要になる。

RNN の特徴には、中間層間の伝搬(出力  $\Rightarrow$  次の入力)があるため、それに対する重みもあり、**重みとバイアスは一つ増えて三か所**となり、これらのパラメータの学習を行うことが目的となる。

図 1 : ニューラルネットワーク構造図

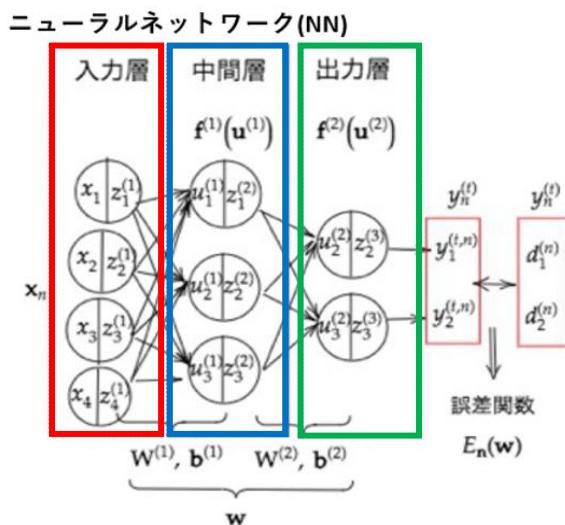
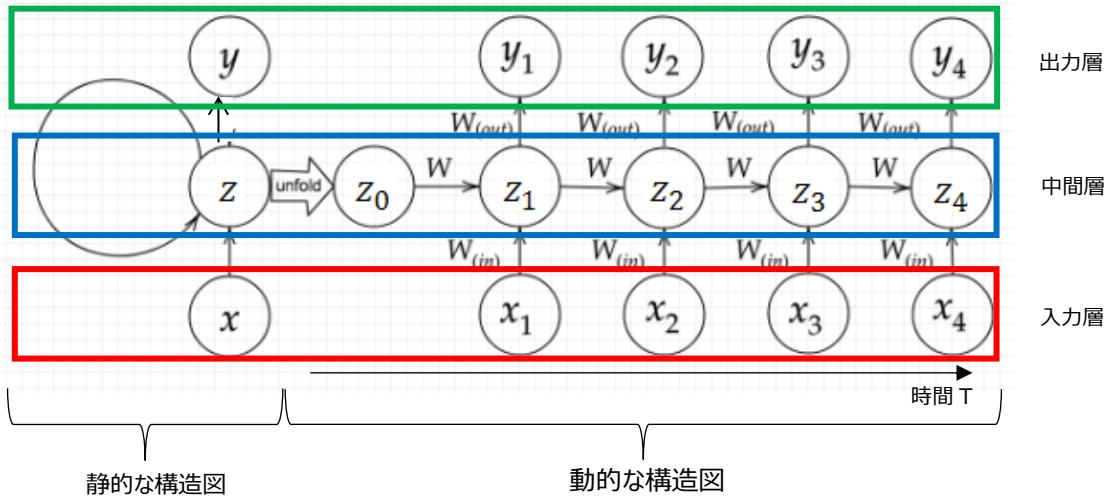


図 1-2 : RNN の模式図



### 数式とコード

$u^t$  が時間  $t$  での中間層への入力データ、 $v^t$  が出力層への入力データ。 $u^t$  式の中に  $z^{t-1}$  があるが、これが、時間  $t-1$  での中間層からの出力データである。

$$\left. \begin{aligned} u^t &= W_{in} \cdot x^t + W \cdot z^{t-1} + b \\ z^t &= f(u^t) = f(W_{in} \cdot x^t + W \cdot z^{t-1} + b) \\ v^t &= W_{out} \cdot z^t + c \\ y^t &= g(v^t) = g(W_{out} \cdot z^t + c) \end{aligned} \right\} \text{式①}$$

```

u[:,t+1] = np.dot(X, W_in) + np.dot(z[:,t].reshape(1, -1), W)
z[:,t+1] = functions.sigmoid(u[:,t+1])
np.dot(z[:,t+1].reshape(1, -1), W_out)
y[:,t] = functions.sigmoid(np.dot(z[:,t+1].reshape(1, -1), W_out))

```

### 1-2 BPTT

RNN の場合、中間層の時間経過を考慮しなければならぬので、そのままでは誤差逆伝播法を適用できない。そこで、RNN 用に時間経過を表現できる形に改良された手法の一つとして、誤差が時間をさかのぼって逆伝播する Backpropagation Through Time (以下、BPTT) があり、パラメータ調整方法の一種となる。

### 数式とコード

RNN では 3 つの重み、 $W_{(in)}$ 、 $W_{(out)}$ 、 $W$  があるため、それぞれ、誤差からの微分によって重みを更新する必要がある。

まず、 $W_{(in)}$  は誤差逆伝播法より  $\frac{\partial E}{\partial W_{(in)}}$  から更新するが、RNN の場合は、1 回の学習ごとに、

$T=t, t-1, \dots, 0$  の場合の  $W_{(in)}$  を更新する必要があるため、簡略化のため  $[ ]^T$  で表現する。

式①より、 $W_{(in)}$ は、 $u^t$ の式であるため、 $u^t$ を $W_{(in)}$ で微分し $x^t$ となる。

また、 $\frac{\partial E}{\partial u^t}$ は、式①より、出力層からの出力値 $y^t$ 、出力層への入力値 $v^t$ 、中間層からの出力値 $z^t$ と連鎖律

の原理で微分することで求まる。ここでは数式を簡略化するため、 $\frac{\partial E}{\partial u^t}$ を $\delta^t$ と置く。他の $W_{(out)}$ 、 $W$ についても同

様に表現する。これにより、3つの重みを更新する式は以下の式となる。

また、バイアスの  $b$  と  $c$  を更新する式は以下の式となる。

$$\frac{\partial E}{\partial W_{(in)}} = \left( \frac{\partial E}{\partial u^t} \frac{\partial u^t}{\partial W_{(in)}} \right)^T = \delta^t [x^t]^T$$

`np.dot(X.T, delta[:,t].reshape(1,-1))`

$$\frac{\partial E}{\partial W_{(out)}} = \left( \frac{\partial E}{\partial v^t} \frac{\partial v^t}{\partial W_{(out)}} \right)^T = \delta^{out,t} [z^t]^T$$

`np.dot(z[:,t+1].reshape(-1,1), delta_out[:,t].reshape(-1,1))`

$$\frac{\partial E}{\partial W} = \left( \frac{\partial E}{\partial u^t} \frac{\partial u^t}{\partial W} \right)^T = \delta^t [z^{t-1}]^T$$

`np.dot(z[:,t].reshape(-1,1), delta[:,t].reshape(1,-1))`

$$\frac{\partial E}{\partial b} = \frac{\partial E}{\partial u^t} \frac{\partial u^t}{\partial b} = \delta^t$$

コード記載なし（本講座のsimple RNNコードでは簡略化のため省略）

$$\frac{\partial E}{\partial c} = \frac{\partial E}{\partial v^t} \frac{\partial v^t}{\partial c} = \delta^{out,t}$$

コード記載なし（本講座のsimple RNNコードでは簡略化のため省略）

$$\frac{\partial E}{\partial u^t} = \frac{\partial E}{\partial v^t} \frac{\partial v^t}{\partial u^t} = \frac{\partial E}{\partial v^t} \frac{\partial \{ W_{(out)} f(u^t) + c \}}{\partial u^t} = f'(u^t) W_{(out)}^T \delta^{out,t} = \delta^t$$

$$\delta^{t-1} = \frac{\partial E}{\partial u^{t-1}} = \frac{\partial E}{\partial u^t} \frac{\partial u^t}{\partial u^{t-1}} = \delta^t \left\{ \frac{\partial u^t}{\partial z^{t-1}} \frac{\partial z^{t-1}}{\partial u^{t-1}} \right\} = \delta^t \{ W f'(u^{t-1}) \}$$

$$\delta^{t-z-1} = \delta^{t-z} \{ W f'(u^{t-z-1}) \}$$

`delta[:,t] = (np.dot(delta[:,t+1].T, W.T) + np.dot(delta_out[:,t].T, W_out.T)) * functions.d_sigmoid(u[:,t])`

## パラメータの更新式

1回の学習当たりの重みは3つ、バイアス2つに対するパラメータ更新式を以下に示す。重みについては、時間  $T=t, t-1, \dots, 0$  の場合について重みを更新する必要がある。

また、 $\epsilon$ は学習率のハイパーパラメータである。

$$W_{(in)}^{t+1} = W_{(in)}^t - \epsilon \frac{\partial E}{\partial W_{(in)}} = W_{(in)}^t - \epsilon \sum_{z=0}^{T_t} \delta^{t-z} [x^{t-z}]^T$$

W\_in -= learning\_rate \* W\_in\_grad

$$W_{(out)}^{t+1} = W_{(out)}^t - \epsilon \frac{\partial E}{\partial W_{(out)}} = W_{(out)}^t - \epsilon \delta^{out,t} [z^t]^T$$

W\_out -= learning\_rate \* W\_out\_grad

$$W^{t+1} = W^t - \epsilon \frac{\partial E}{\partial W} = W_{(in)}^t - \epsilon \sum_{z=0}^{T_t} \delta^{t-z} [z^{t-z-1}]^T$$

W -= learning\_rate \* W\_grad

$$b^{t+1} = b^t - \epsilon \frac{\partial E}{\partial b} = b^t - \epsilon \sum_{z=0}^{T_t} \delta^{t-z} \quad \text{コード記載なし}$$

$$c^{t+1} = c^t - \epsilon \frac{\partial E}{\partial c} = c^t - \epsilon \delta^{out,t} \quad \text{コード記載なし}$$

### ◆ 実装演習結果キャプチャ

#### 演習チャレンジ 1

以下は再帰型ニューラルネットワークにおいて構文木を入力として再帰的に文全体の表現ベクトルを得るプログラムである。ただし、ニューラルネットワークの重みパラメータはグローバル変数として定義してあるものとし、\_activation 関数はなんらかの活性化関数であるとする。木構造は再帰的な辞書で定義しており、root が最も外側の辞書であると仮定する。(<) にあてはまるのはどれか。

```
def traverse(node):
    ...
    node: tree node, recursive dict, {left: node', right: node'}
    if leaf, word embedded vector, (embed_size,)

    W: weights, global variable, (embed_size, 2*embed_size)
    b: bias, global variable, (embed_size,)
    ...
    if not isinstance(node, dict):
        v = node
    else:
        left = traverse(node['left'])
        right = traverse(node['right'])
        v = _activation(left + right)  (<)
    return v
```

- (1) W.dot(left + right)
- (2) W.dot(np.concatenate([left, right]))
- (3) W.dot(left \* right)
- (4) W.dot(np.maximum(left, right))

正解 : (2)

### 【解説】

隣接単語（表現ベクトル）から表現ベクトルを作るという処理は、隣接している表現 left と right を合わせたものを特徴量としてそこに重みを掛けすることで実現する。つまり、 $W \cdot \text{dot}(\text{np.concatenate}([left, right]))$  である。他の選択肢では、left と right 両方の表現が失われるため間違いである。

### 演習チャレンジ 2

下図は BPTT を行うプログラムである。なお簡単化のため活性化関数は恒等関数であるとする。また、`calculate_dout` 関数は損失関数を出力に関して偏微分した値を返す関数であるとする。**(お)** にあてはまるのはどれか。

```
def bptt(xs, ys, W, U, V):
    """
    ys: labels, (batch_size, n_seq, output_size)
    """

    # 順伝播
    hiddens, outputs = rnn_net(xs, W, U, V)
    # 損失関数のW, Y, Vに関する偏微分値
    dW = np.zeros_like(W) # dL/dW
    dU = np.zeros_like(U) # dL/dU
    dV = np.zeros_like(V) # dL/dV
    # 損失関数の出力値に関する偏微分値
    do = _calculate_d0(outputs, ys) # dL/do, (batch_size, n_seq, output_size)

    batch_size, n_seq = ys.shape[:2]
    # 時間を逆方向にたどり、パラメータの偏微分値を計算（バックプロパゲーション）-
    # dL/dV = do/dV · dL/do = h · dL/do
    # dL/dW = do/dW · dL/do
    # dL/dU = do/dU · dL/do
    # do/dW = (dh_{\{t\}}/dW · d/dh_{\{t\}}+dh_{\{t-1\}}/dW · d/dh_{\{t-1\}}+\dots)o_{\{t\}}-
    #         = (x_{\{t\}}+x_{\{t-1\}} \cdot U+x_{\{t-2\}} \cdot U^2+\dots) \cdot V
    # do/dU = (dh_{\{t\}}/dU · d/dh_{\{t\}}+dh_{\{t-1\}}/dU · d/dh_{\{t-1\}}+\dots)o_{\{t\}}-
    #         = (h_{\{t-1\}}+h_{\{t-2\}} \cdot U+h_{\{t-3\}} \cdot U^2+\dots) \cdot V
    for t in reversed(range(n_seq)):
        dV += np.dot(do[:, t].T, hiddens[:, t]) / batch_size
        delta_t = do[:, t].dot(V)
        # 時間tの出力は時間t以前の中間層すべてに依存するため
        # W, Uはさらに遡って計算
        for bptt_step in reversed(range(t+1)):
            dW += np.dot(delta_t.T, xs[:, bptt_step]) / batch_size
            dU += np.dot(delta_t.T, hiddens[:, bptt_step-1]) / batch_size
            delta_t = do[:, t].dot(U)
    return dW, dU, dV
```

- (1)  $\delta_t \cdot \text{dot}(W)$
- (2)  $\delta_t \cdot \text{dot}(U)$
- (3)  $\delta_t \cdot \text{dot}(V)$
- (4)  $\delta_t \cdot V$

正解：(2)

### 【解説】

RNN では中間層出力  $h_{\{t\}}$  が過去の中間層出力  $h_{\{t-1\}}, \dots, h_{\{1\}}$  に依存する。

RNN において損失関数を重み  $W$  や  $U$  に関して偏微分するときは、それを考慮する必要があり、 $dh_{\{t\}}/dh_{\{t-1\}} = U$  であることに注意すると、過去に遡るたびに  $U$  が掛けられる。つまり、 $\delta_t = \delta_t \cdot \text{dot}(U)$  となる。

## 実装演習

<3\_1\_simple\_RNN\_after.ipynb>

以下のとおり各種パラメータを変更して学習結果を確認した。

1. weight\_init\_std、learning\_rate 及び hidden\_layer\_size のパラメータを変更して結果を確認

- 結果としては、学習率 2 倍の場合が一番良い結果になった

1-1. Default : 収束までの学習回数は 3000 回付近

1-2. 重み初期値 2 倍 : 収束までの学習回数が多くなり 6000 回付近で収束

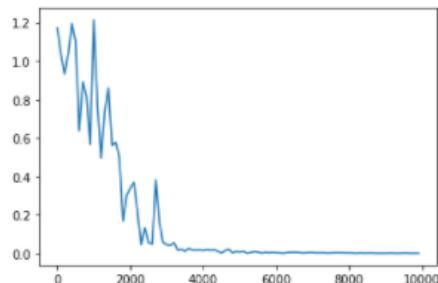
1-3. 学習率 2 倍 : 収束までの学習回数が少くなり 2000 回付近で収束

1-4. 中間層 2 倍 : 収束までの学習回数が多くなり 6000 回付近で収束

### 1-1.Default

```
weight_init_std = 1  
learning_rate = 0.1  
hidden_layer_size = 16
```

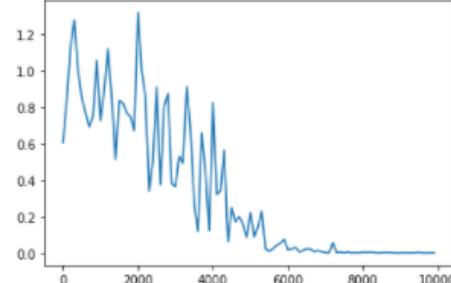
```
iters:9900  
Loss:0.00037489862254820946  
Pred:[1 0 0 1 1 0 0 0]  
True:[1 0 0 1 1 0 0 0]  
125 + 27 = 152
```



### 1-2.重み初期値 2 倍

```
weight_init_std = 2  
learning_rate = 0.1  
hidden_layer_size = 16
```

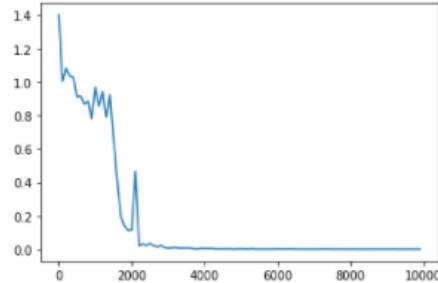
```
iters:9900  
Loss:0.0013353027789628944  
Pred:[1 0 0 1 0 0 0 0]  
True:[1 0 0 1 0 0 0 0]  
39 + 105 = 144
```



### 1-3.学習率 2 倍

```
weight_init_std = 1  
learning_rate = 0.2  
hidden_layer_size = 16
```

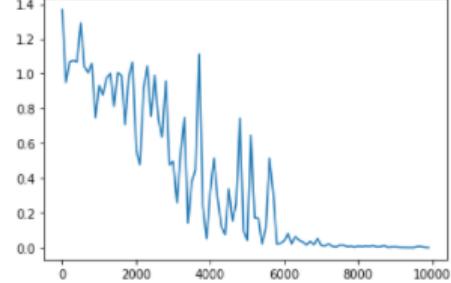
```
iters:9900  
Loss:0.00013839568632886788  
Pred:[1 0 0 0 0 1 1 0]  
True:[1 0 0 0 0 1 1 0]  
39 + 35 = 134
```



### 1-4.中間層 2 倍

```
weight_init_std = 1  
learning_rate = 0.1  
hidden_layer_size = 32
```

```
iters:9900  
Loss:0.002552554026104608  
Pred:[0 1 0 1 0 1 0 0]  
True:[0 1 0 1 0 1 0 0]  
28 + 56 = 84
```

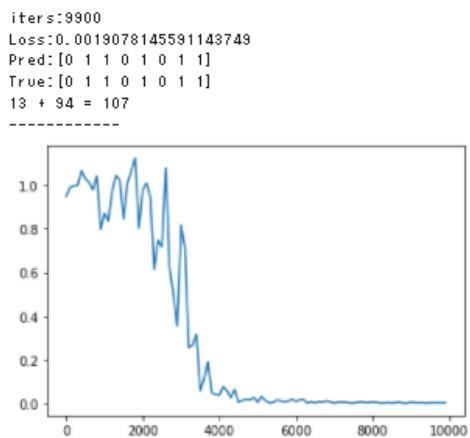


## 2. 重みの初期化方法を変更して結果を確認

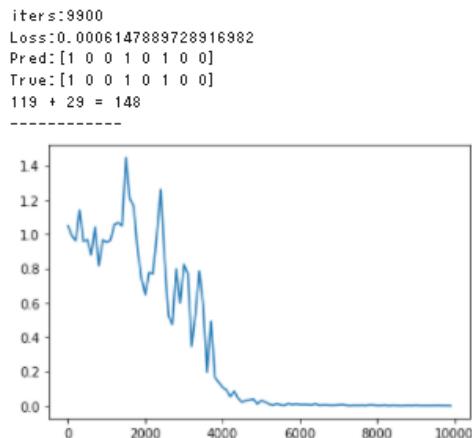
- 結果としては、Default の場合が一番良い結果になった

- 2-1. Default(random) : 収束までの学習回数は 3000 回付近
- 2-2. Xavier : 収束までの学習回数が多くなり 4000 回付近で収束
- 2-3. He : Xavier 同様、4000 回付近で収束

### 2-2.Xavier



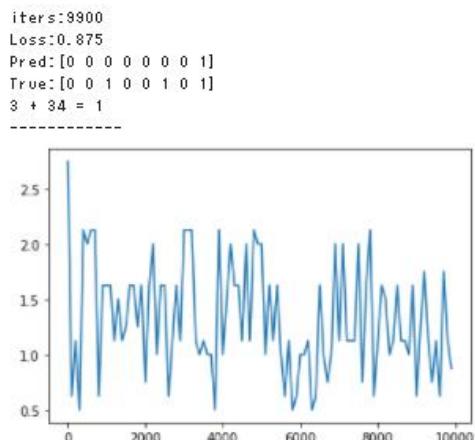
### 2-3.He



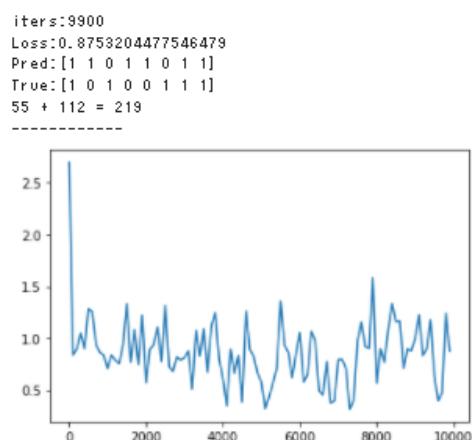
### 3. 中間層の活性化関数を変更して結果を確認

- 結果としては、Default の場合が一番良い結果になった
- 3-1. Default(sigmoid) : 収束までの学習回数は 3000 回付近
- 3-2. ReLU : 発散し収束しなかった
- 3-3. tanh : 発散し収束しなかった

### 3-2.ReLU



### 3-3.tahn



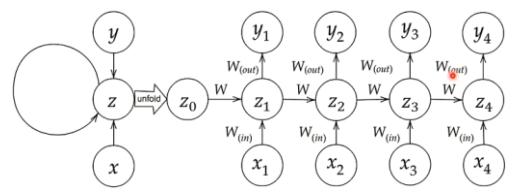
#### ◆確認テストなどの考察結果

確認テスト結果を表 1 にまとめた。

表 1: 確認テスト

	問題	解答	考察
--	----	----	----

①	<p>サイズ <math>5 \times 5</math> の入力画像を、サイズ <math>3 \times 3</math> の フィルタで畳み込んだ時の出力画像サイズを答 えよ。</p> <p>なおストライドは 2、パディングは 1 とする。</p>	<p><math>3 \times 3</math></p> <p>出力画像サイズ(幅、高さ) <math>=(5+2\times 1 - 3)/2 + 1</math> <math>=3</math></p>	<p>深層学習前編 (day2) で学習し た、畳み込み後の 出力画像サイズを 求める公式を使 用了した</p>
②	<p>RNN のネットワークには大きく分けて 3 つの重 みがある。</p> <p>1 つは入力から現在の中間層を定義する際に 掛けられる重み、1 つは中間層から出力を定 義する際に掛けられる重みである。</p> <p>残り 1 つの重みについて説明せよ。</p>	<p>時間的に 1 つ前の中間層から 現在の中間層を定義する際に 掛けられる重み</p>	<p>3 番目の重みが必 要なことが、ニューラ ルネットワークと異な る点である</p>
③	<p>連鎖律の原理を使い、<math>\frac{dz}{dx}</math> を求めよ。</p> <p><math>z = t^2</math></p> <p><math>t = x + y</math></p>	$\begin{aligned} \frac{dz}{dx} &= \frac{dz}{dt} \times \frac{dt}{dx} \\ &= 2t \times 1 \\ &= 2(x + y) \end{aligned}$	<p>連鎖律の原理を用 いて、複数の関数が 合成された合成関 数に対して、合成 関数の微分を各関 数の微分の積で求 めることが出来る</p>
④	<p>下の図の <math>y_1</math> を <math>x_1, z_0, z_1, W_{in}, W, W_{out}</math> を用 いて式で表せ。※バイアスは任意の文字で 定義せよ。</p> <p>※また、中間層の出力にシグモイド関数 <math>g(x)</math> を作用させよ。</p>	<p><math>y_1 = g(W_{out} \cdot z_1 + c)</math></p> <p><math>z_1 = f(W_{in} \cdot x_1 + W \cdot z_0 + b)</math></p> <p>※ <math>c</math> と <math>b</math> はバイアス</p>	<p>中間層でも活性化 関数として <math>f(x)</math> を作 用させた</p>



## <Section2: LSTM>

### ◆要点まとめ

#### ・RNN の課題

##### - 勾配消失

誤差逆伝播法では、下位層に進んでいくに連れて、勾配がどんどん緩やかになっていく。そのため、勾配降下法による更新では下位層のパラメータはほとんど変わらず、学習は最適値に収束しなくなる。

RNN でも時系列を遡れば遡るほど、勾配が消失していき、長い時系列の学習が困難となる。例えば 8 回分の記憶は 8 層の中間層と同様になることから顕著に勾配消失が発生する。

### <解決策>

これまでの講義で触れた勾配消失の解決方法とは異なり、RNN では時系列データを記憶する関係で、そもそも構造自体を変えて **LSTM** (Long Short-Term Memory: 長・短期記憶) により課題を解決している。

##### - 勾配爆発

勾配爆発は、勾配が層を逆伝播するごとに指数関数的に大きくなっていく、勾配消失とは反対の現象で、原因は次の更新式において

$$\text{次バッチの重み} = \text{現バッチの重み} - \text{学習率} \times \text{勾配}$$

重みも勾配も大きな負の値と大きな正の値を行ったり来たりして、振幅もどんどん大きくなり発散してしまうのです。恒等関数を用いた場合に多く発生する。

### <解決策>

研究・実践して得られた学習率の推奨値を用いることで対応する。あるいは、実装演習で紹介された勾配クリッピングの手法を使う。

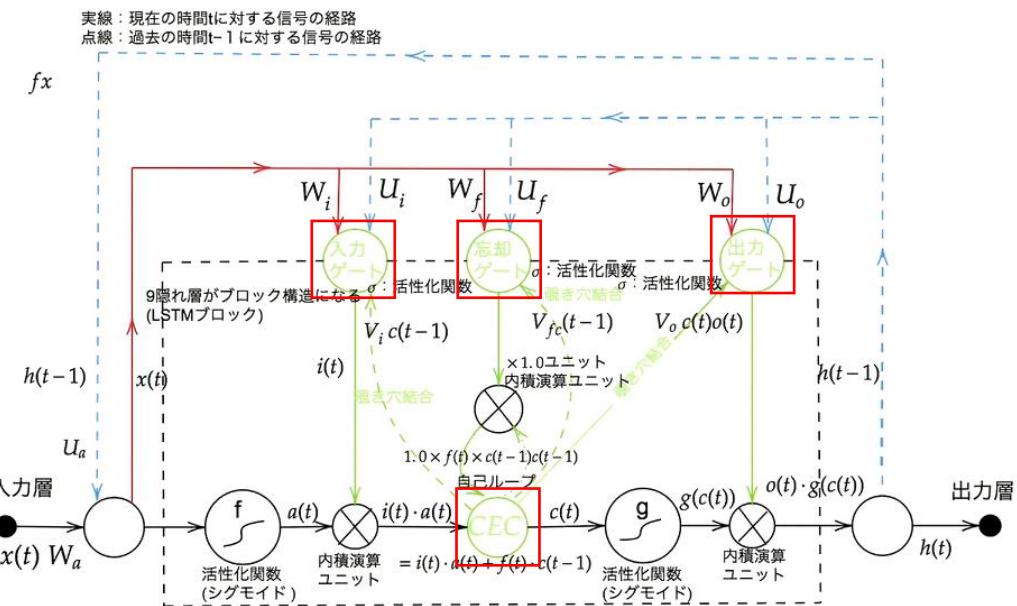
RNN では時系列データを記憶する関係で、そもそも構造自体を変えて **LSTM** (Long Short-Term Memory: 長・短期記憶) により課題を解決している。

## LSTM 全体像

以下に LSTM 構成図を示す。黒枠内がすべて中間層である。RNN の一種であるため、青点線で示すように中間層の出力を再帰的に中間層に入力する。

LSTM は、学習の状態を記憶する CEC と、学習を行う入力ゲートと出力ゲート、不要になった記憶を削除する忘却ゲートで構成することに特徴がある。以下ではこれらの構成要素を順番に説明する。

図 2:LSTM 構成図



## 2-1 CEC (Constant Error Carousel)

勾配消失および勾配爆発の解決方法として、**勾配が"1"**であれば解決できる。

$$\delta^{t-z-1} = \delta^{t-z} \{ W f'(u^{t-z-1}) \} = 1$$

$$\frac{\partial E}{\partial c^{t-1}} = \frac{\partial E}{\partial c^t} \frac{\partial c^t}{\partial c^{t-1}} = \frac{\partial E}{\partial c^t} \frac{\partial}{\partial c^{t-1}} \{ a^t - c^{t-1} \} = \frac{\partial E}{\partial c^t}$$

CEC の出力 \$c(t)\$ は、入力層からの入力を \$a(t)\$、入力ゲートからの入力を \$i(t)\$、忘却ゲートからの入力を \$f(t)\$、そして、CEC 自身の自己ループからの入力を \$c(t-1)\$ とすると、以下の式となる。

**CECの出力**

$$c(t) = \frac{i(t) \cdot a(t)}{\text{入力}} + \frac{f(t) \cdot c(t-1)}{\text{自己ループ}} \quad \dots \text{式②}$$

CEC は記憶機能のみを担っているため、入力データについて、CEC は時間依存度に関係なく重みが一律となる課題がある。CEC だけではニューラルネットワークの学習特性が無くなってしまう。

そのため、考える=学習機能として入力ゲートと出力ゲートを CEC の周りに置くようにしている。

## 2-2 入力ゲートと出力ゲート

入力・出力ゲートを中間層に追加することで、それぞれのゲートへの入力値の重みを、重み行列 \$w\$ と \$u\$ で調整可能とし学習機能を持たせることで、CEC の課題を解決している。

入力ゲートは、今回の入力値と前回の入力値から、それぞれの重み \$w\_i\$ と \$u\_i\$ を学習し、CEC へ今回の入力

$a(t)$ をどのくらい覚えさせるかを $i(t)$ により制御している。※式②参照

出力ゲートは、今回の入力値と前回の入力値から、それぞれの重み $w_o$ と $u_o$ を学習し、CEC の出力 $c(t)$ をどの程度使うか $o(t)$ により制御している。

## 2-3 忘却ゲート

LSTM の課題として、CEC は過去の情報を全て保管するため、過去の情報が不要になった場合、削除することはできず、保管され続ける課題があった。

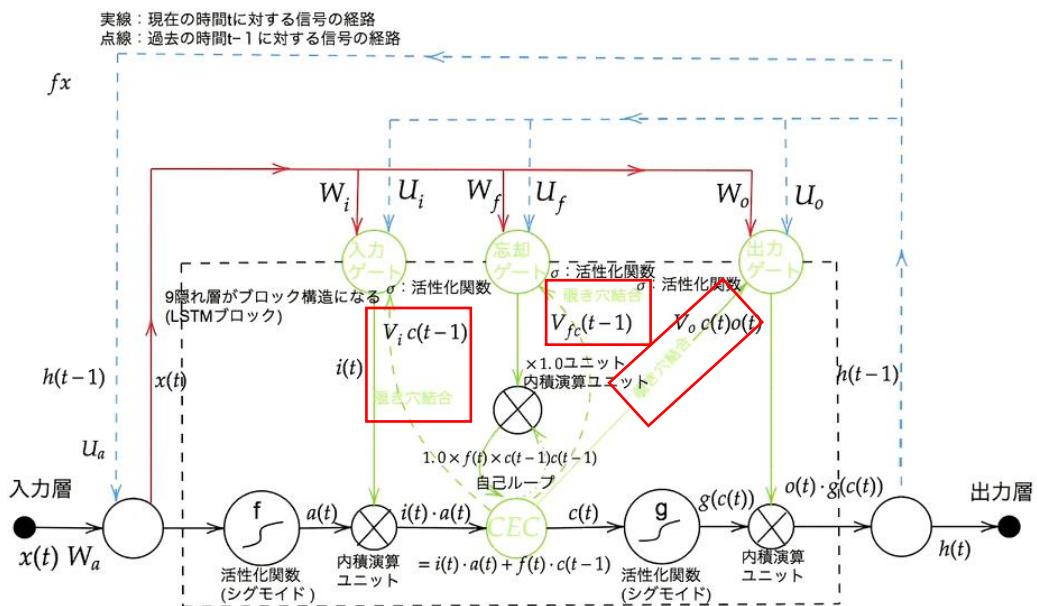
解決策として、過去の情報が不要になった場合、そのタイミングで情報を忘却する機能を忘却ゲートで実現している。

忘却ゲートは、今回の入力値と前回の入力値から、それぞれの重み $w_f$ と $u_f$ を学習し、CEC の自己ループからの入力 $c(t-1)$ をどの程度使うか $f(t)$ により制御している。※式②参照

## 2-4 覗き穴結合

LSTM ブロックの課題として、CEC に保存されている過去の情報を任意のタイミングで他のノードに伝播させたり、あるいは任意のタイミングで忘却させたい。CEC 自身はゲート制御に影響を与えていない。

そこで、CEC 自身の値に重み行列を介して他のゲートに伝播可能にした構造とする覗き穴結合が考えられたが、効果はあまりないので使われていない。



### ◆ 実装演習結果キャプチャ

#### 演習チャレンジ 1

NN や深いモデルでは勾配の消失または爆発が起こる傾向がある。

勾配爆発を防ぐために勾配のクリッピングを行うという手法がある。具体的には勾配のノルムがしきい値を超えたたら、勾配のノルムをしきい値に正規化するというものである。以下は勾配のクリッピングを行う関数である。

(さ) にあてはまるのはどれか。

(1)  $\text{gradient} * \text{rate}$

- (2) gradient / norm
- (3) gradient / threshold
- (4) np.maximum(gradient, threshold)

```
def gradient_clipping(grad, threshold):
    """
    grad: gradient
    """
    norm = np.linalg.norm(grad)
    rate = threshold / norm
    if rate < 1:
        return [rate * grad]
    return grad
```

正解：(1)

#### 【解説】

勾配のノルムがしきい値より大きいときは、勾配のノルムをしきい値に正規化するので、クリッピングした勾配は、  
勾配×(しきい値/勾配のノルム)と計算される。つまり、gradient \* rate である。

- ノルムの正規化は、Python の norm 関数を用いて計算している
- rate 比率に応じて、1 より小さければ gradient \* rate の戻り値となり、  
1 以上であれば gradient の戻り値となる

## 演習チャレンジ 2

以下のプログラムは LSTM の順伝播を行うプログラムである。ただし sigmoid 関数は要素ごとにシグモイド関数を作用させる関数である。**(け)** にあてはまるのはどれか。

- (1) output\_gate\* a + forget\_gate\* c
- (2) forget\_gate\* a + output\_gate\* c
- (3) input\_gate\* a + forget\_gate\* c
- (4) forget\_gate\* a + input\_gate\* c

```
def lstm(x, prev_h, prev_c, W, U, b):
    """
    x: inputs, (batch_size, input_size)
    prev_h: outputs at the previous time step, (batch_size, state_size)
    prev_c: cell states at the previous time step, (batch_size, state_size)
    W: upward weights, (4*state_size, input_size)
    U: lateral weights, (4*state_size, state_size)
    b: bias, (4*state_size,)
    """
    # セルへの入力やゲートをまとめて計算し、分離
    lstm_in = _activation(x.dot(W.T) + prev_h.dot(U.T) + b)
    a, i, f, o = np.hsplit(lstm_in, 4)

    # 録を変換、セルへの入力:(-1, 1), ゲート:(0, 1)
    a = np.tanh(a)
    input_gate = _sigmoid(i)
    forget_gate = _sigmoid(f)
    output_gate = _sigmoid(o)

    # セルの状態を更新し、中間層の出力を計算
    c = [forget_gate * np.tanh(a)]
    h = output_gate * np.tanh(c)
    return c, h
```

正解：(3)

【解説】

新しいセルの状態は、計算されたセルへの入力と 1 ステップ前のセルの状態にそれぞれ入力ゲートと忘却ゲートの出力を掛けて足し合わせたものと表現される。

つまり、CEC の出力の式の  $\text{input\_gate} * a + \text{forget\_gate} * c$  である。

### 実装演習

<predict\_word.ipynb>

該当する実装演習がないため、"predict\_word.ipynb"のコードを実施して確認した。

Google Colaboratory で Tensorflow を使用しようとするとエラーが出るため、以下の様に Tensorflow のバージョン切替を実施した。

変更方法は以下コードを実行後「メニュー→ランタイム→ランタイムを再起動」すると反映される。

TensorFlow 2.x系をやめて1.x系で動かす

```
1. %tensorflow_version 1.x
```

このコードは、以下の様に、"some of them looks like"の文章の次に来る単語を予測するコードであり、登録済みの単語に対する出現確率を表示し、最後に、予測結果を表示する。ここでは、単語"et"が予測された。

```
# 保存したモデルを使って単語の予測をする
ln.predict("some of them looks like")
```

ストリーミング出力は最後の 5000 行に切り捨てされました。

```
beating : 1.3192024e-14
authentic : 1.490178e-14
glow : 1.5293678e-14
oy : 1.468018e-14
emotion : 1.4983662e-14
delight : 1.4004107e-14
nuclear : 1.4859943e-14
dropped : 1.4963213e-14
hiroshima : 1.3928321e-14
beings : 1.582246e-14
tens : 1.5364727e-14
burned : 1.3809626e-14
homeless : 1.5643702e-14
albert : 1.447877e-14
```

```
Prediction: some of them looks like et
```

◆確認テストなどの考察結果

確認テスト結果を表 2 にまとめた。

表 2: 確認テスト

問題	解答	考察
<p>① シグモイド関数を微分した時、入力値が 0 の時に最大値をとる。その値として正しいものを選択肢から選べ。</p> <p>(1) 0.15          (2) 0.25          (3) 0.35          (4) 0.45</p>	<p>(2) 0.25</p> <p>図：シグモイド関数</p> <p>図：シグモイド関数微分値</p>	<p>シグモイド関数の微分結果は以下となり、0 の時に最大値、0.25 となる</p>
<p>② 以下の文書を LSTM に入力し空欄に当てはまる単語を予測したいとする。</p> <p>文中の「とても」という言葉は空欄の予測においてなくなっても影響を及ぼさないと考えられる。このような場合、どのゲートが作用すると考えられるか。</p> <p>「映画おもしろかったね。ところで、とてもお腹が空いたから何か_____。」</p>	<p>忘却ゲート</p>	<p>忘却ゲートにより LSTM(=CEC) に記憶する情報を任意の期間で記憶することができるため、不要になつた情報を削除することができる。</p>

### <Section3: GRU>

#### ◆要点まとめ

LSTM の課題としては、LSTM では構成が複雑になり、パラメータ数が多く、計算負荷が高くなる問題があった。

GRU は、上述の LSTM の課題解決のため、そのパラメータを大幅に削減し、精度は同等またはそれ以上が望める様になった構造とすることで、計算負荷が低くするメリットを得た。

※NN の分野では、最初は計算量を度外して仕組みを作り、その後に大きくなり過ぎた NW の精度を極力維持しながら、軽量化の改良を加えるアプローチが多い。

図 3:GRU 構成図

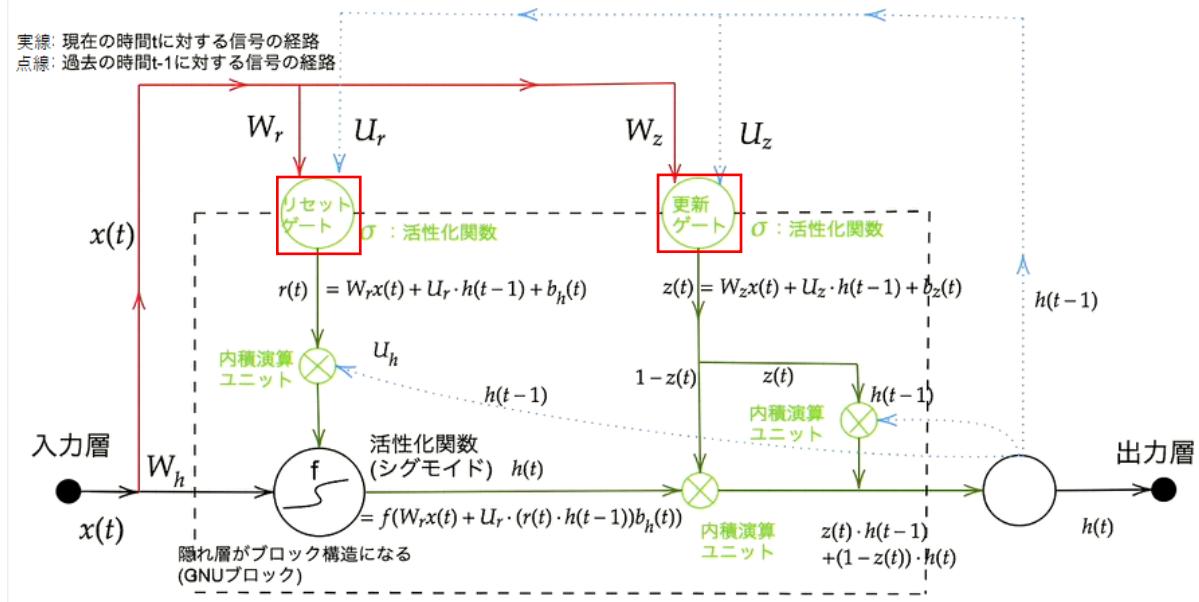


図 3 は GRU の全体像を表すデータフロー図である。GRU では、CEC、LSTM の入力ゲート、出力ゲート、および忘却ゲートの代わりに、リセットゲートと更新ゲートを導入することでシンプル化が図られた。

活性化関数の出力  $h(t)$  は以下の式となる。

**活性化関数(シグモイド)の出力**

$$h(t) = f(W_r x(t) + U_r \cdot (r(t) \cdot h(t-1)) + b_h(t)) \cdots \text{式③}$$

$r(t)$  がリセットゲートの出力、 $z(t)$  が更新ゲートの出力となり、これらのパラメータを使って、LSTM と同様に、今回と前回の入力値をどのくらいの割合で組み合わせて出力するかを制御している。

#### ◆実装演習結果キャプチャ

##### 演習チャレンジ 1

GRU(Gated Recurrent Unit)も LSTM と同様に RNN の一種であり、単純 RNN において問題となる勾配消失問題を解決し、長期的な依存関係を学習することができる。LSTM に比べ変数の数やゲートの数

が少なく、より単純なモデルであるが、タスクによっては LSTM より良い性能を発揮する。以下のプログラムは GRU の順伝播を行うプログラムである。ただし \_sigmoid 関数は要素ごとにシグモイド関数を作成する関数である。**(c)** にあてはまるのはどれか。

- (1)  $z * h_{\text{bar}}$
- (2)  $(1-z) * h_{\text{bar}}$
- (3)  $z * h * h_{\text{bar}}$
- (4)  $(1-z) * h + z * h_{\text{bar}}$

```
def gru(x, h, W_r, U_r, W_z, U_z, W, U):
    """
    x: inputs, (batch_size, input_size)
    h: outputs at the previous time step, (batch_size, state_size)
    W_r, U_r: weights for reset gate
    W_z, U_z: weights for update gate
    U, W: weights for new state
    """
    # ゲートを計算
    r = _sigmoid(x.dot(W_r.T) + h.dot(U_r.T))
    z = _sigmoid(x.dot(W_z.T) + h.dot(U_z.T))

    # 次状態を計算
    h_bar = np.tanh(x.dot(W.T) + (r * h).dot(U.T))
    h_new = [ ]          (c)
    return h_new
```

正解：(4)

#### 【解説】

新しい中間状態は、1 ステップ前の中間表現と計算された中間表現の線形和で表現される。つまり更新ゲート  $z$  を用いて、 $(1-z) * h + z * h_{\text{bar}}$  と書ける。

図 3 の GRU の出力、 $h(t)$  を求める計算式が該当箇所となる

#### 実装演習

<predict\_word.ipynb>

TensorFlow を使用したコードである。TensorFlow は NN を実行するための高度なライブラリであり、LSTM や GRU を容易に実装できる。

このコードは、以下の様に、“some of them looks like”の文章の次に来る単語を予測するコードであり、登録済みの単語に対する出現確率を表示し、最後に、予測結果を表示する。ここでは、単語“et”が予測された。

```
# 保存したモデルを使って単語の予測をする  
In.predict("some of them looks like")
```

ストリーミング出力は最後の 5000 行に切り捨てられました。

```
beating : 1.3192024e-14  
authentic : 1.490178e-14  
glow : 1.5293678e-14  
oy : 1.468018e-14  
emotion : 1.4983662e-14  
delight : 1.4004107e-14  
nuclear : 1.4859943e-14  
dropped : 1.4963213e-14  
hiroshima : 1.3928321e-14  
beings : 1.582246e-14  
tens : 1.5364727e-14  
burned : 1.3809626e-14  
homeless : 1.5643702e-14  
albert : 1.447877e-14
```

```
Prediction: some of them looks like et
```

### ◆確認テストなどの考察結果

確認テスト結果を表 3 にまとめた。

表 3: 確認テスト

問題	解答	考察
① LSTM と CEC が抱える課題について、それぞれ簡潔に述べよ。	<ul style="list-style-type: none"><li>LSTM 4つの構成要素(CEC、入力ゲート、出力ゲート、忘却ゲート)があり、複雑かつパラメータが多いため計算量が多い</li><li>CEC 勾配が 1 で学習能力がない</li></ul>	LSTM は複雑で計算量が多いことが課題
② LSTM と GRU の違いを簡潔に述べよ。	<ul style="list-style-type: none"><li>LSTM CEC と入力ゲート、出力ゲート、忘却ゲートの 3 つのゲートがあり計算量が多い</li><li>GRU CEC なく、リセットゲートと更新ゲートの 2 つのゲートで実現。計算量は LSTM より少ない</li></ul>	GRU により、LSTM に対して予測精度はほぼ同等で、計算量を少なくすることが可能となった

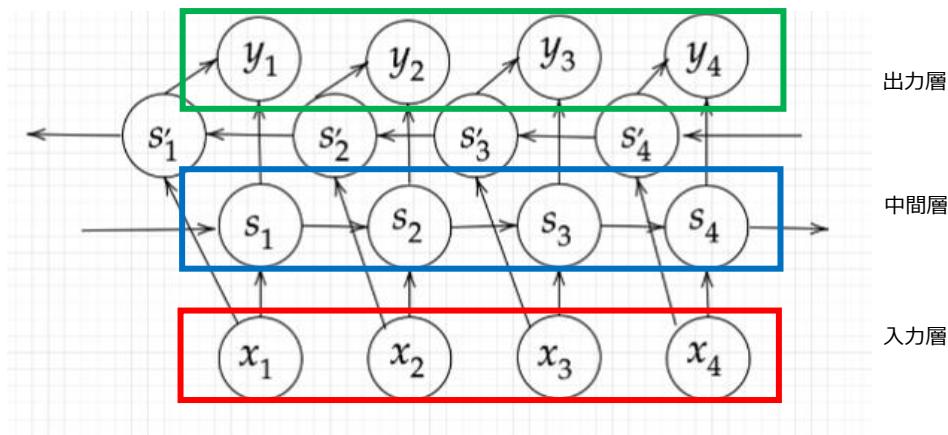
## <Section4: 双方向 RNN>

### ◆要点まとめ

双方向 RNN(Bidirectional RNN)とは、過去の情報だけでなく未来の情報を加味することで、両方向に伝播するネットワークのこと、精度を向上させるためのモデル。

実用事例としては、入力時に過去の情報と未来の情報が一度に与えられる、文章の推敲や機械翻訳等があるが、過去から未来までのすべての情報を入力してはじめて予測するため、双方向 RNN の応用範囲は限定される。

図 4: 双方向 RNN 構成図



### ◆実装演習結果キャプチャ

#### 演習チャレンジ 1

以下は双方向 RNN の順伝播を行うプログラムである。順方向については、入力から中間層への重み  $W_f$ , 1 ステップ前の中間層出力から中間層への重みを  $U_f$ 、逆方向に関しては同様にパラメータ  $W_b$ ,  $U_b$  を持ち、両者の中間層表現を合わせた特徴から出力層への重みは  $V$  である。`_rnn` 関数は RNN の順伝播を表し中間層の系列を返す関数であるとする。(か) にあてはまるのはどれか

- (1)  $h_f + h_b[:: -1]$
- (2)  $h_f * h_b[:: -1]$
- (3) `np.concatenate([h_f, h_b[:: -1]], axis=0)`
- (4) `np.concatenate([h_f, h_b[:: -1]], axis=1)`

```

def bidirectional_rnn_net(xs, W_f, U_f, W_b, U_b, V):
    """
    ...
    W_f, U_f: forward rnn weights, (hidden_size, input_size)
    W_b, U_b: backward rnn weights, (hidden_size, input_size)
    V: output weights, (output_size, 2*hidden_size)
    ...
    xs_f = np.zeros_like(xs)
    xs_b = np.zeros_like(xs)
    for i, x in enumerate(xs):
        xs_f[i] = x
        xs_b[i] = x[::-1]
    hs_f = _rnn(xs_f, W_f, U_f)
    hs_b = _rnn(xs_b, W_b, U_b)
    hs = [ ] (か) for h_f, h_b in zip(hs_f, hs_b)]
    ys = hs.dot(V.T)
    return ys

```

正解：(4)

【解説】

双方向 RNN では、順方向と逆方向に伝播したときの中間層表現をあわせたものが特徴量となるので、

`np.concatenate([h_f, h_b[::-1]], axis=1)`

である。

各重みの  $W_f, U_f, W_b, U_b, V$  の使い方とデータフローをイメージしながら判断する。

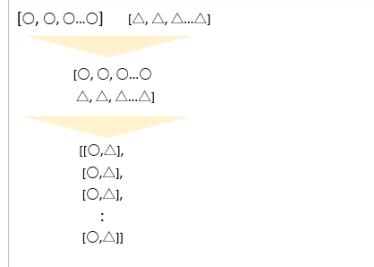
`concatenate` で使っている `axis` の使い方は以下のとおりとなる。`axis` パラメータにより、`concatenate` による 2 つの配列の合体の仕方が変わり、`axis=0` では 2 つの配列を合体させるのに対して、`axis=1` では二つ配列要素が同じ場所の者同士を合体させる。

このコードでは、`axis=1` により、同じ時間の値どうしを合体させている。

`axis=0`



`axis=1`



## 実装演習

該当する実装演習がないため省略

### ◆確認テストなどの考察結果

双方向 RNN は未来の情報がわかっていないと使えないため、文書の推敲や、機械翻訳、フレーム間の補完などの用途に向いている。

該当する確認テストはなかったため省略

## <Section5: Seq2Seq>

### ◆要点まとめ

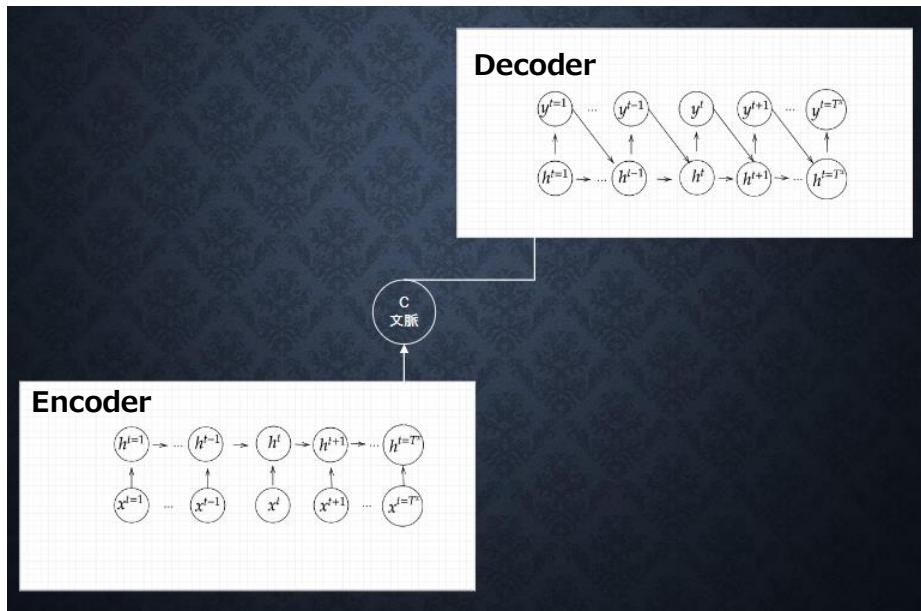
#### seq2seq

seq2seq は、機械対話や、機械翻訳などに使用される **Encoder-Decoder モデルの一種**である。

Encoder と Decoder の二つのモデルのドッキングしたもので、例えば機械翻訳の場合、

- ①Encoder で単語列が入力されると順次記憶し、文脈の解釈を可能にする
  - ②それを Decoder で文脈の解釈から翻訳する言語の表現を逆変換で生成する
- のようなシーケンスからシーケンスを作り出すことで実現している。

図 5:seq2seq 構成図



### 5-1 Encoder RNN

自然言語処理の場合、Encoder RNN は、ユーザーがインプットした文を、単語等のトークンへ区切って (Taking)、**分散表現ベクトル**に変換(Embedding)し、それを RNN に入力することで、入力した文の意味を蓄積(RNN)する処理を繰り返す構造となる。

- Taking : 文章を単語等のトークン毎に分割し、トークンごとの ID に分割する
- Embedding : ID から、そのトークンを表す分散表現ベクトルに変換
- RNN : 分散表現ベクトルとそれまでの hiddenstate により、新たな hiddenstate を蓄積

自然言語処理の場合の文書を分散表現ベクトルに変換する概要を図 5-1 に示す。

1. 入力された単語を ID (one-hot ベクトル) に分割する (数十万規模)
2. one-hot ベクトル表現の冗長を解消するために embedding を行う (数百規模に圧縮)  
embedding への変換は NN を用いて行う。その際に単語の似ているものは embedding 値が近いものとして処理して embedding 表現として単語の意味を表すようにする

図 5-1: 文書を分散表現ベクトルに変換

分散表現ベクトルへ変換				
単語	ID	one-hot	ebedding	
私	1	[0, 0, 0 ... 0]	[0.2, 0.4, 0.6 ... 0.1]	
は	2	[0, 1, 0 ... 0]	[ ... ]	
刺身	3	[0, 0, 1 ... 0]	[ ... ]	
昨日	4	[ ... ]	[ ... ]	
:	:	:	:	
xxxxx	10.000	[0, 0, 0 ... 1]	[ ... ]	

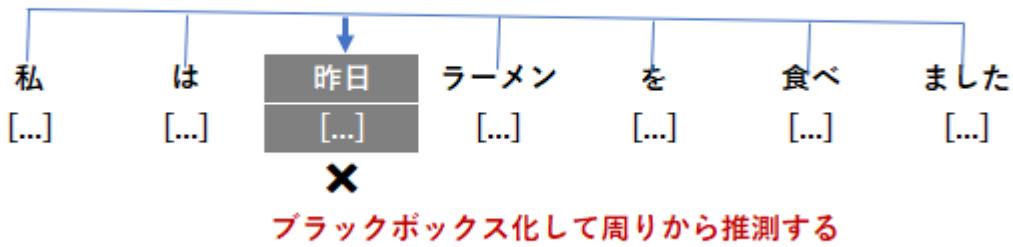
Encoder RNN は、分散表現ベクトル（vec1）を入力とし、hidden state を出力する。この hidden state と次の入力（vec2）を、また RNN に入力し、hidden state を出力するという流れを繰り返す。

最後の分散表現ベクトル（vec）を入れたときの hidden state を final state としてとておく。この final state が thought vector と呼ばれ、入力した文の意味を表すベクトルとなる。

Encoder RNN では、embedding で似たような単語は似たような分散表現ベクトルになるように機械学習で特徴量抽出を行っている。ここが Encoder RNN の肝となる。

この分野では Google の **BERT** (Bidirectional Encoder Representations from Transformers) が有名である。BERT のようなモデルでは、MLM (Masked Language MODEL) が良く使われている。

### MLM: Masked Language MODEL



例えば、上のように「昨日」の部分をブラックボックス化して、前後の関係から同じ意味のものを機械学習させることで、人の介在なしに単語・意味ベクトルの教師無し学習を行っている。これにより、飛躍的に大量のデータを学習させることができた。

## 5-2 Decoder RNN

Decoder RNN は、以下の様に、Encoder RNN が出力した文の意味を表す情報（thought vector）を入力とし、アウトプットデータを単語等のトークンごとに生成する構造となっている。隠れ層から出力した単語を、再度解釈して文が成り立つように次の単語の出力しており、会話のシーケンスから別のシーケンスを作り出すことが可能となった。

- DecoderRNN : Encoder RNN の final state (thought vector) から、各 token の生成確率を出力
- Sampling : 生成確率にもとづいて token をランダムに選ぶ
- Embedding : Sampling で選ばれた token を Embedding して Decoder RNN への次の入力とする
- Detokenize : Decoder,Sampling,Embedding を繰り返し、得られた token を、分散表現ベクトルから文字列に変換して出力し、文として出力する

### 5-3 HRED

seq2seq の課題としては、一問一答しかできなく、問に対して文脈も何もなく、ただ応答を行うことしかできないことである。

一つ一つの過去の文、状態と次の状態をコンテキスト化して過去  $n-1$  個の発話から次の発話を生成するのが **HRED** (the hierarchical recurrent encoder-decoder) である。

HRED の構成を図 5-1 に示す。その構成は seq2Seq (青枠部分) + Context RNN (赤枠部分) である。赤線の過去のコンテキストを Context RNN の入力とすることで実現している。

Context RNN: Encoder のまとめた各文章の系列をまとめて、  
これまでの会話コンテキスト全体を表すベクトルに変換する構造。  
これにより、過去の発話の履歴を加味した返答が可能になる

図 5-1:HRED 構成図

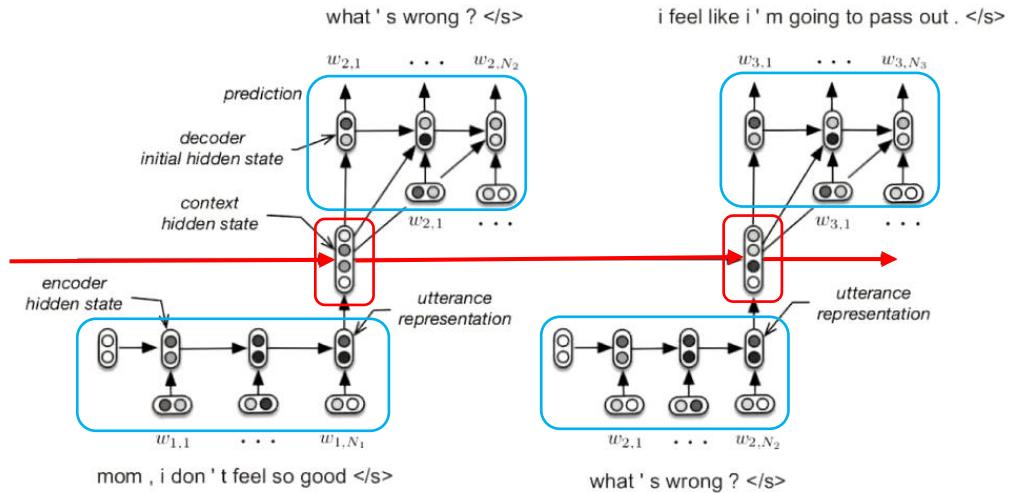


Figure 1: The computational graph of the HRED architecture for a dialogue composed of three turns. Each utterance is encoded into a dense vector and then mapped into the dialogue context, which is used to decode (generate) the tokens in the next utterance. The encoder RNN encodes the tokens appearing within the utterance, and the context RNN encodes the temporal structure of the utterances appearing so far in the dialogue, allowing information and gradients to flow over longer time spans. The decoder predicts one token at a time using a RNN. Adapted from Sordoni et al. (2015a).

### HRED の課題

HRED は確率的な多様性が字面にしかなく、会話の「流れ」のような多様性が無い。同じコンテキスト

(発話リスト) を与えられても、答えの内容が毎回会話の流れとしては同じものしか出せない。

HRED は、短いよくある答えを学習してしまう傾向があるため、短く、情報量に乏しい答えをしがちである。

例：「うん」「そうだね」「…」など

システム： インコかわいいよね。

ユーザー： うん

システム： インコかわいいのわかる。

※ありがちな答えになる

## 5-4 VHRED

**VHRED** (Variational Hierarchical Recurrent Encoder Decoder) は、HRED へ VAE (Variational AutoEncoder) の**潜在変数の概念**を追加したもので、過去の会話のコンテキストを汎用化することでより汎用的な表現力を保持し、HRED の課題である短い単調な答えになることを解決した構造である。次の章で VAE について説明する。

## 5-5 VAE (Variational AutoEncoder)

### 5-5-1 オートエンコーダ(AutoEncoder)

オートエンコーダとは、学習時の入力データの訓練データのみで教師データを利用しない教師なし学習の一つである。オートエンコーダ具体例として、MNIST の場合、 $28 \times 28$  の数字の画像を入れて、同じ画像を出力するニューラルネットワークなどがある。

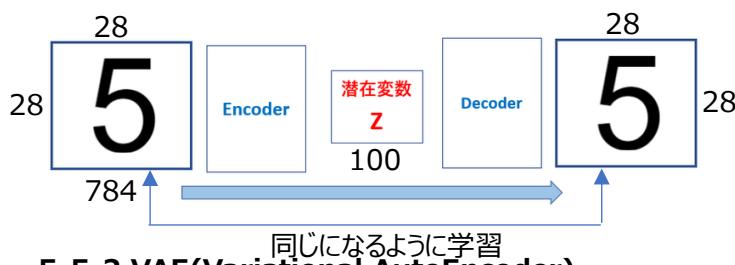
オートエンコーダ構造としては、

Encoder： 入力の画像データから**潜在変数 z** に変換するニューラルネットワーク

Decoder： 逆に**潜在変数 z** をインプットとして元画像を復元するニューラルネットワーク

で構成し、次元削減を行えるメリットがある。

以下の様に、z の次元(=100)が入力データ(=784)より小さい場合、次元削減とみなすことができる。

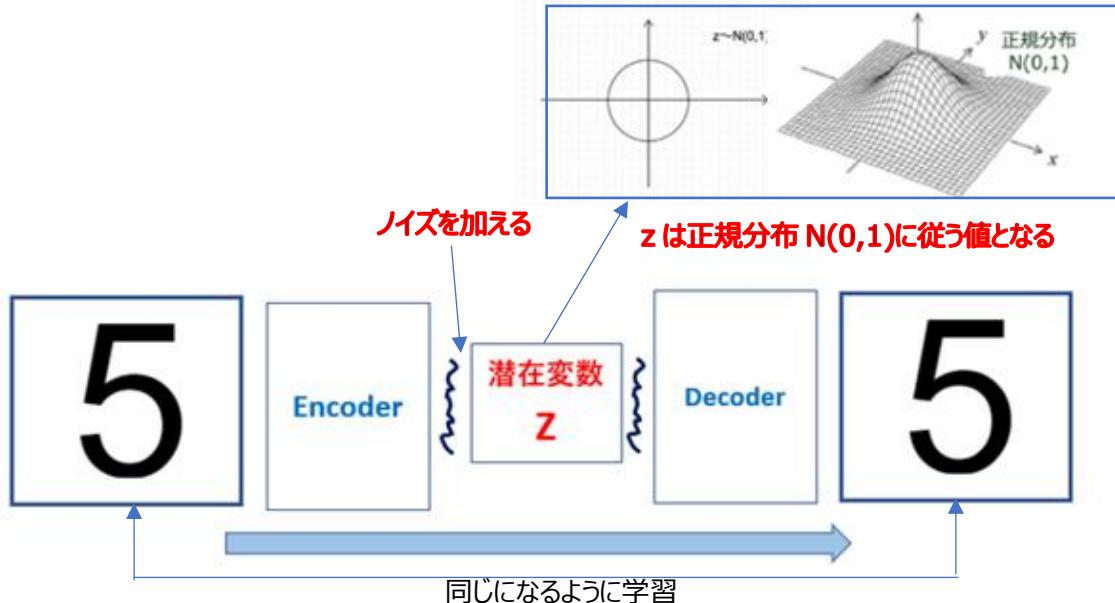


### 5-5-2 VAE(Variational AutoEncoder)

通常のオートエンコーダの場合、Encode 対象の情報を何かしら潜在変数 **z** に押し込んでいるものの、その **z** がどのような状態かわからない。**VAE** ではこの**潜在変数 z に確率分布  $z \sim N(0,1)$  を仮定**したものである。

MNIST の例では、オートエンコーダの場合、Encode 対象の数値が「1」と「7」で似通っており、「2」や「4」は似通っていないというような元の数値の類似度は学習できなかった。そのため **VAE** では、データを**潜在変数 z の確率分布**に変換することを可能にしている。これにより、類似している数値は似通った確率分布として表現することが可能となった。

具体的には、ランダムなノイズを潜在変数へ印加し、ノイズがある潜在変数から元の画像に Decode するように学習することにより、平均 0、分散 1 の潜在変数  $z$  を出力し、汎用性を高めている。



#### ◆実装演習結果キャプチャ

##### 演習チャレンジ 1

機械翻訳タスクにおいて、入力は複数の単語から成る文（文章）であり、それぞれの単語は one-hot ベクトルで表現されている。Encoder において、それらの単語は単語埋め込みにより特徴量に変換され、そこから RNN によって（一般には LSTM を使うことが多い）時系列の情報をもつ特徴へとエンコードされる。以下は、入力である文（文章）を時系列の情報をもつ特徴量へとエンコードする関数である。ただし\_activation 関数はなんらかの活性化関数を表すとする。（き）にあてはまるのはどれか。

- (1)  $E \cdot dot(w)$
- (2)  $E.T \cdot dot(w)$
- (3)  $w \cdot dot(E.T)$
- (4)  $E * w$

```
def encode(words, E, W, U, b):
    """
    words: sequence words (sentence), one-hot vector, (n_words, vocab_size)
    E: word embedding matrix, (embed_size, vocab_size)
    W: upward weights, (hidden_size, hidden_size)
    U: lateral weights, (hidden_size, embed_size)
    b: bias, (hidden_size,)
    """

    hidden_size = W.shape[0]
    h = np.zeros(hidden_size)
    for w in words:
        e = (き)
        h = _activation(W.dot(e) + U.dot(h) + b)
    return h
```

正解:(1)

【解説】

単語  $w$  は one-hot ベクトルでありそれを別の特徴量  $e$  に変換する。これは埋め込み行列  $E$  を用いて、 $E \cdot e$  の内積計算と書ける。 $E$  は単語と単語の意味の対応表である。

$E$  : word embedding matrix, ( $\text{embed\_size}$ ,  $\text{vocab\_size}$ )

$w$  : upward weight, ( $\text{hidden\_size}$ ,  $\text{hidden\_size}$ )

実装演習

該当する実装演習がないため省略

◆確認テストなどの考察結果

確認テスト結果を表 5 にまとめた。

表 5: 確認テスト

	問題	解答	考察
①	<p>下記の選択肢から、seq2seq について説明しているものを選べ。</p> <p>(1) 時刻に関して順方向と逆方向の RNN を構成し、それら 2 つの中間表現を特徴量として利用するものである。</p> <p>(2) RNN を用いた Encoder-Decoder モデルの一種であり、機械翻訳などのモデルに使われる。</p> <p>(3) 構文木などの木構造に対して、隣接単語から表現ベクトル（フレーズ）を作るという演算を再帰的に行い（重みは共通）、文全体の表現ベクトルを得るニューラルネットワークである。</p> <p>(4) RNN の一種であり、単純な RNN において問題になる勾配消失問題を CEC とゲートの概念を導入することで解決したものである。</p>	(2)	(1)は双方向 RNN の説明 (3)は構文木の説明 (4)は LSTM の説明
②	seq2seq と HRED、HRED と VHRED の違いを簡潔に述べよ。	• seq2seq と HRED seq2seq では会話の文脈無視で応答がなされるのに対し、HRED では前の単語の流れに	—

		<p>即して応答されるため、より人間らしい文書が生成される</p> <ul style="list-style-type: none"> <li>• HRED と VHRED</li> </ul> <p>HRED は seq2seq+context RNN で構成され、VHRED は HRED に VAE の潜在変数の概念を追加したもの</p>	
③	<p>VAE に関する下記の説明文中の空欄に当てはまる言葉を答えよ。</p> <p>自動符号化器の潜在変数に _____ を導入したもの</p>	確率分布	<p>確率分布を導入することで、似通ったデータは、似たような確率分布になるようにしている</p>

## <Section6: word2Vec>

### ◆要点まとめ

RNN の課題として、単語のような可変長の文字列を NN に与えることができず、構造上、固定長形式で単語を表す必要がある。

その解決策として、embedding による分散表現ベクトルを使用した word2vec がある。これは既に seq2seq で解説しているが、学習データからボキャブラリを作成する際に、one-hot ベクトルが辞書の単語数分できるのを、embedding により分散表現ベクトルに変換することで回避している。

word2vec により、大規模データの分散表現の学習が、現実的な計算速度とメモリ量で実現可能にした。

以下は、" I want to eat apples. I like apples."というボキャブラリを入力した場合の例である。

1. 辞書には 6 つの単語を登録する

Ex) I want to eat apples. I like apples.



辞書 : {apples, eat, I, like, to, want}

2. apples を入力する場合は、入力層には以下のベクトルを入力する

※本来は辞書の単語数だけ one-hot ベクトルで入力

Ex)

1…apples

0…eat

0…I

0…like

3. 入力されたベクトルを分散表現ベクトルに変換

この変換表を機械学習により、うまく作成することが重要

$$[0,1,0,0,0,0,0,0,0,\dots] \xrightarrow{\begin{bmatrix} w_{11}^{(l)} & \dots & w_{1l}^{(l)} \\ \vdots & \ddots & \vdots \\ w_{j1}^{(l)} & \dots & w_{jl}^{(l)} \end{bmatrix}} =[1,13,15]$$

↓  
ボキャブラリ数×単語ベクトルの次元数の重み行列。

### ◆実装演習結果キャプチャ

該当する実装の説明とコードがなかつたため、「word2vec を Colab 環境で使うための 5 行」のページで、gensim を利用して簡易的に「単語のベクトル化」を行う方法を確認した。

[https://qiita.com/Ninagawa\\_Izumi/items/6c38160e041b6c333905](https://qiita.com/Ninagawa_Izumi/items/6c38160e041b6c333905)

### ◆確認テストなどの考察結果

該当する確認テストがないため省略

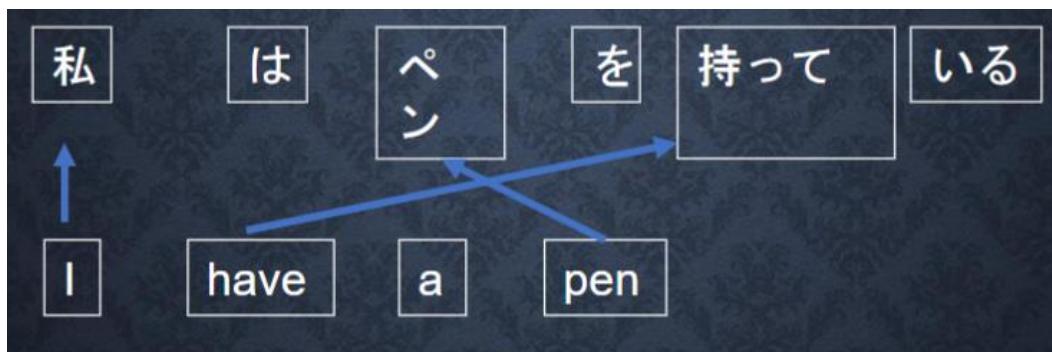
## <Section7: Attention Mechanism>

### ◆要点まとめ

seq2seq の課題として、長い文章への対応が難しいことである。seq2seq では、入力が 2 単語でも、100 単語でも、作成したモデルの中間層が固定化されているため、固定次元ベクトルにより入力しなければならない。

文章が長くなるほど、動的にそのシーケンスの内部表現の次元が大きくなっていく仕組みが必要で、Attention Mechanism では「入力と出力との単語が関連しているのか」の**関連度を学習する仕組み**で実現している。

以下の例では、「a」については関連度が低く、「I」については「私」との関連度が高いことを学習できるようにしている。



### ◆実装演習結果キャプチャ

該当する実装の説明とコード提供がないため省略

### ◆確認テストなどの考察結果

確認テスト結果を表 7 にまとめた。

表 7: 確認テスト

問題	解答	考察
① NN と word2vec、seq2seq と Attention の違いを簡潔に述べよ。	<ul style="list-style-type: none"><li>• RNN は時系列データの処理に適している手法</li><li>• word2vec は単語の分散表現ベクトルを得る手法</li><li>• seq2seq は一つの時系列データから別の時系列データを得るネットワーク</li><li>• Attention Mechanism は時系列データの中身の関連性に重みをつける手法</li></ul>	—

## ■深層学習後編(day4)

### <Section1: 強化学習>

#### ◆要点まとめ

##### 1-1. 強化学習

機械学習を学習スタイルから大きく3つに分類すると以下となる。強化学習はそのうちの一つである。

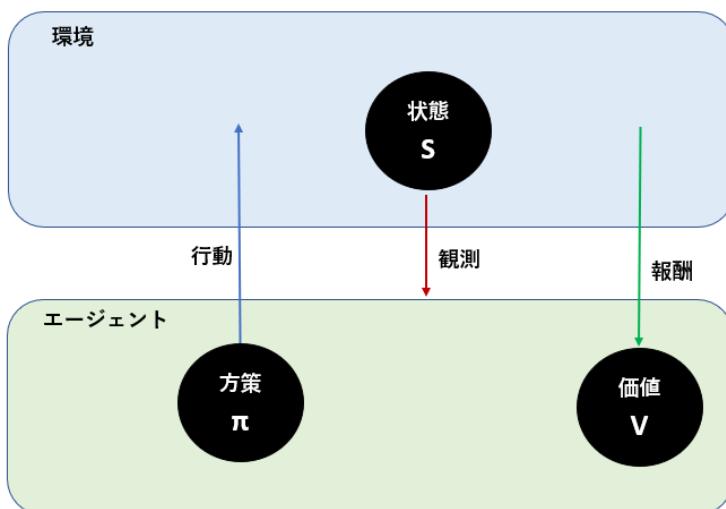
教師あり学習

教師なし学習

##### ★強化学習

強化学習は、長期的に報酬を最大化できるように環境のなかで行動を選択できるエージェントを作ることを目標とする機械学習の一分野である。エージェントは、行動の結果として得られる利益（報酬）をもとに、利益を最大化する行動を学習する。

以下に、強化学習のイメージを示す。主人公はエージェントとなる。エージェントは環境=仕事場で行動することになるが、方策となる方針に従って行動し、仕事を頑張ってよりよい報酬（価値）を得ようと方策を学習することに例えられる。環境は状態を持つため変わりうる。



##### 1-2. 強化学習の応用例

###### マーケティングの場合

環 境:	会社の販売促進部
エージェント:	プロフィールと購入履歴に基づいて、キャンペーンメールを送る顧客を決めるソフトウェアである。
行 動:	顧客ごとに送信、非送信のふたつの行動を選ぶことになる。
報 酬:	キャンペーンのコストという負の報酬とキャンペーンで生み出されると推測される売上という正の報酬を受けるマーケティングの場合

上表のように実際の職場に例えれば、エージェントは

### 目標：売上を最大化し、コストを最小化する

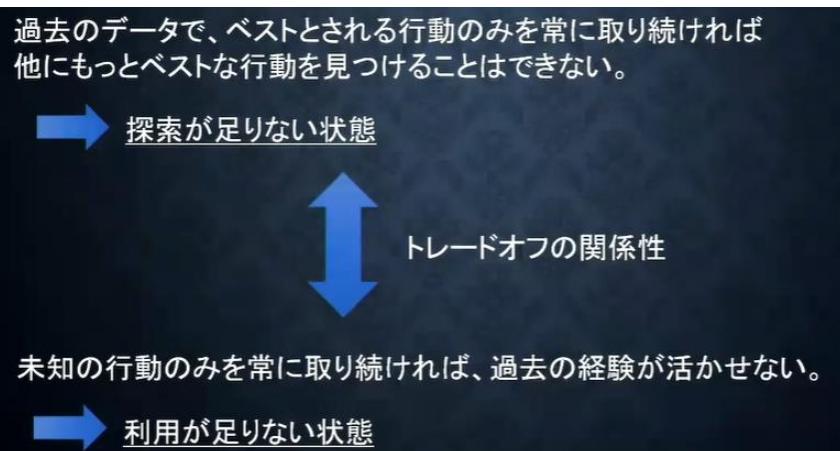
を目指す行動をすることにより、キャンペーンのコストを抑えつつ、セット販売などを増加させるような顧客に対してメールを送信することとなる。

強化学習の最初は顧客選定の知見がないので、無作為はじめることになるが、行動の結果としてキャンペーンの効果を学習することで顧客選定の方策を学ぶことになる。

### 1-3. 探索と利用のトレードオフ

環境について事前に完璧な知識があれば、最適な行動を予測し決定することは可能だが、最初から知識があるわけではないため、上記仮定は成り立たないとする。

強化学習では、不完全な知識を元に行動しながら、データを収集。最適な行動を見つけていく。最初はランダムに行動を開始し、その後、過去の経験とトレードオフしながら学習を進めていく。



### 1-4. 強化学習のイメージ

方策と価値の判断は学習の中で実行されるが、強化学習では、環境下の状態( $s$ )と行動( $a$ )との関数として表すことができる。

方策関数 :  $\pi(s,a)$

行動価値関数 :  $Q(s,a)$

強化学習では、方策関数と行動価値関数が最適になるように学習を行う。

### 1-5. 強化学習の差分

強化学習と通常の教師あり、教師なし学習との違いは目標が異なることである。

- 教師あり、教師なし学習では、データに含まれるパターン（特徴）を見つけ出す  
および、そのデータから予測することが目標
- 強化学習では、優れた方策を見つけることが目標

近年、計算速度の向上、および、関数近似法と Q 学習を組み合わせる手法の登場により大きく発展している。

Q学習:	行動価値関数を、行動する毎に更新することにより学習を進める方法
関数近似法:	価値関数や方策関数を関数近似する手法のこと

## 1-6. 行動価値関数

価値を表す価値関数としては以下の2種類あり、最近は行動価値関数が使われる。

状態価値関数 : 環境（状態）で価値が変化する関数

ある状態の価値に注目する場合は、状態価値関数を使う

行動価値関数 : 環境（状態）と行動で価値が変化する関数

(Q学習) ある状態で行動した時の価値に注目する場合は、行動価値関数を使う

## 1-7. 方策関数

方策関数とは、方策ベースの強化学習手法において、ある状態でどのような行動を探るのかの確率を与える関数のこと。エージェントの行動を決める関数で、エージェントはある環境下で方策関数に基づいて行動する。

方策関数 :  $\pi(s) = a$

関数の関係

エージェントは方策に基づいて行動する

$\pi(s, a) : V$  や  $Q$  を基にどういう行動をとるか

⇒ その瞬間、その瞬間の行動をどうするか

$$\left. \begin{array}{l} V^\pi(s) : \text{状態関数} \\ Q^\pi(s, a) : \text{状態+行動関数} \end{array} \right\} \begin{array}{l} \text{ゴールまで今の方策を続けたときの} \\ \text{報酬の予測値が得られる} \end{array}$$

⇒ やり続けたら最終的にどうなるか

囲碁の場合は、

方策関数：一手一手どこに打つかを決める

価値関数：最終的に勝てるかの価値を出す

となる。

## 1-8. 方策勾配法

方策関数の学習は、方策関数を NN として学習することで行う。具体的には方策をモデル化して方策勾配法で最適化する手法である。

方策勾配法：

$$\theta^{(t+1)} = \theta^{(t)} + \epsilon \nabla J(\theta)$$

$\theta$  が NN の重みに対応し、 $\theta$  は学習率  $\epsilon$  と、 $\theta$  での行動の累積によって得られる価値を表す関数  $J(\theta)$  の微分値

の積で更新して最適化を行う。

NN での重みの最適化と異なるのは、 $J(\theta)$  の微分値を加算する点である。NN では、正解ラベルとの誤差を最小化することが目的であったが、強化学習ではなるべく多くの報酬を得ることが目的であるため加算している。

$J(\theta)$  は平均報酬や割引報酬和で定義する。 $\nabla J(\theta)$  は、 $J(\theta)$  の定義から行動価値関数  $Q(s, a)$  を使った以下の式から求まる。

$$\nabla_{\theta} J(\theta) = \nabla_{\theta} \sum_{a \in A} \pi_{\theta}(a|s) Q^{\pi}(s, a) \quad \text{ある行動を取るときの報酬}$$

$$\nabla_{\theta} J(\theta) = E_{\pi_{\theta}}[(\nabla_{\theta} \log \pi_{\theta}(a|s) Q^{\pi}(s, a))] \quad \cdots \text{方策勾配定理}$$

※方策勾配定理を求める式は、加点レポートで別途確認する。

#### ◆ 実装演習結果キャプチャ

該当する実装の説明とコード提供がないため、

「強化学習（DQN）チュートリアル」

[https://colab.research.google.com/github/YutaroOgawa/pytorch\\_tutorials\\_jp/blob/main/notebook/4\\_RL/4\\_1\\_reinforcement\\_q\\_learning\\_jp.ipynb](https://colab.research.google.com/github/YutaroOgawa/pytorch_tutorials_jp/blob/main/notebook/4_RL/4_1_reinforcement_q_learning_jp.ipynb)

を確認した。

エージェント (Agent) は、カート(土台の車)に乗っているポールを、できるかぎり直立した状態を維持することを目指し、行動を選択することを学習するチュートリアルであるが、講義で学習した用語は出てきたが、どこに実装されているかわからないコードになっている。PyTorch で実装されているため、さらなる学習が必要である。

#### ◆ 確認テストなどの考察結果

該当する確認テストがないため省略

#### ◆ 加点レポート

以下資料により、方策勾配定理導出の確認を行った。

[http://seiya-kumada.blogspot.com/2018/02/blog-post\\_13.html](http://seiya-kumada.blogspot.com/2018/02/blog-post_13.html)

## <Section2: AlphaGo>

### ◆要点まとめ

**AlphaGo** は、強化学習により注目を集めた碁のゲーム用モデルである。

#### **AlphaGo(Lee)**

#### **AlphaGo Zero**

について比較しながら解説する。

<b>AlphaGo Lee</b>	David Silver Mastering the game of Go with deep neural networks and tree search nature 27 January 2016 <a href="https://www.nature.com/articles/nature16961">https://www.nature.com/articles/nature16961</a>
<b>AlphaGo Zero</b>	David Silver Mastering the game of Go without human knowledge nature 18 October 2017 <a href="https://www.nature.com/articles/nature24270">https://www.nature.com/articles/nature24270</a>

#### **AlphaGo(Lee)**

基本的には強化学習と同様な考え方であり、方策関数と価値関数に該当する 2 つの畳み込み NW で構成される。

方策関数 → **PolicyNet**

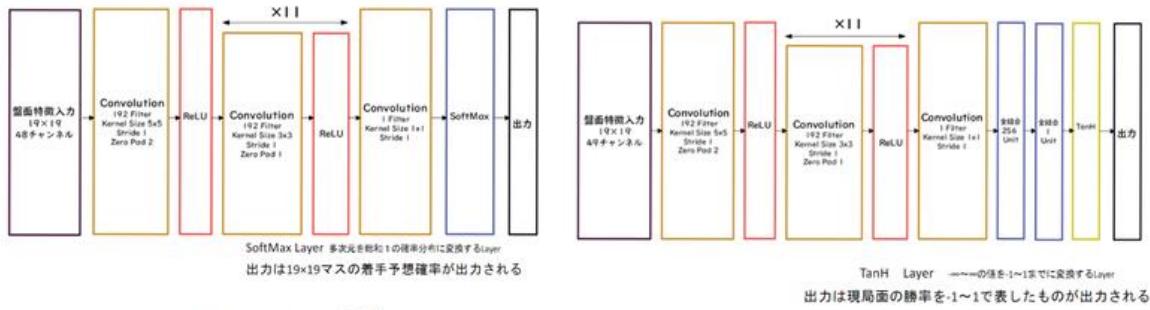
価値関数 → **ValueNet**

方策関数に該当する **Policy Net** は、次の一手を確率的に導くための処理で、図 9-1 のとおり基盤特徴入力として碁盤の目にあたる  $19 \times 19$  の 48 チャネルをとる。48 チャンネルの内訳は、図 9-1 の左下の表に示す。例えば、石の状態は、自石、敵石、空白の 3 チャンネルで表す。

このように二次元データ化すると畳み込みが可能となり、畳み込みと活性化関数として ReLU を繰り返して、最後に、SoftMax Layer で碁盤  $19 \times 19$  の予想指し手の確率（総和は 1）が出力される。これにより**次の一手を予測**している。

価値関数に該当する **ValueNet** は、最終的に勝てそうかを予測するため、入力として  $19 \times 19$  の 49 チャンネルを取る。49 チャンネル目は図 9-1 の左下の表のとおり"手番"が追加されている。畳み込み後に Flatten 層により 1 チャンネル（一次元）に変換している。これは 2017 年頃には定石になった方法である。最後に、Tanh Layer を用意することで、 $-\infty \sim +\infty$  の値を、(負け)-1～+1(勝ち)に変換し、**現局面の勝率**を予測している。

図 9-1: PolicyNet と ValueNet



Policy Net、Value Netの入力

特徴	チャンネル数	説明
石	3	自石、敵石、空白の3チャンネル
オール1	1	全盤1
着手履歴	8	8手前までに石が打たれた場所
呼吸点	8	該当の位置に石がある場合、その石を含む達の呼吸点の数
取れる石の数	8	該当の位置に石を打った場合、取れる石の数
取られる石の数	8	該当の位置に石を打たれた場合、取られる石の数
着手後の呼吸点の数	8	該当の位置に石を打った場合、その石を含む達の呼吸点の数
着手後にシショウで取れるか？	1	該当の位置に石を打った場合、シショウで隣接達を取れるかどうか
着手後にシショウで取られるか？	1	該当の位置に石を打たれた場合、シショウで隣接達を取られるかどうか
合法手	1	合法手であるかどうか
オール0	1	全盤0
手番	1	現在の手番が黒番であるか？(PolicyNetにはこの入力はなく、ValueNetのみ)

NNではなく線形の方策関数  
探索中に高速に着手確率を出すために使用される

RollOutPolicy

赤字の特徴はrollOut時に使用されず、TreePolicyの初期値として使用されるときに使われる

上記の特徴が19x19マス分あり、出力はそのマスの着手予想確率となる

## AlphaGo の学習

AlphaGo の学習で特徴的なことは、強化学習に先立ち、**RollOutPolicy** と **PolicyNet** を教師あり学習することで効率化を図っている。具体的には、**RollOutPolicy** では、通常 3ms 掛かる次の一手の計算が 3μs と 1000 倍に改善されており、何億回の手順計算に時間要する計算量の削減を、精度を一定に保ちつつ行っている。

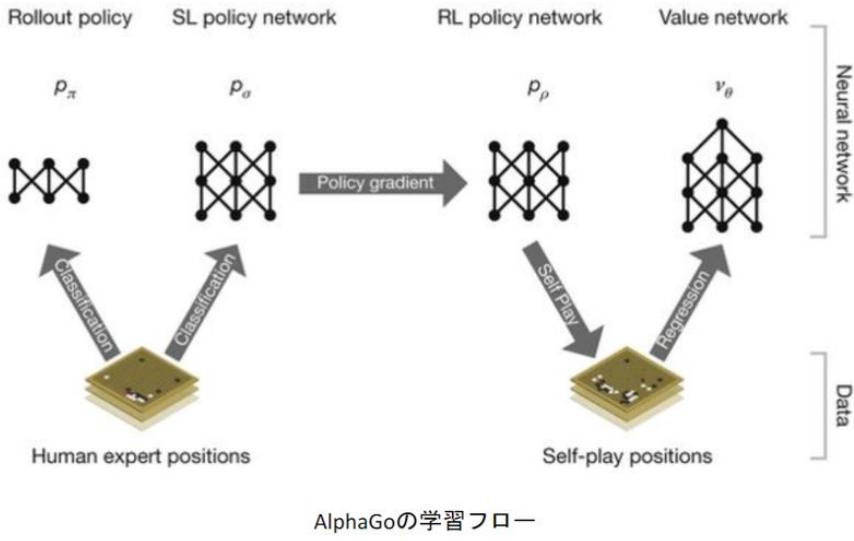
## 2. AlphaGo の学習フロー

全体の学習フローは、

1. 教師あり学習による RollOutPolicy と PolicyNet の学習
2. 強化学習による PolicyNet の学習
3. 強化学習による ValueNet の学習 Alpha Go の学習

となり下図のように説明されている。

図 9-2: AlphaGo 学習フロー



AlphaGoの学習フロー

出典: David Silver Mastering the game of Go with deep neural networks and tree search  
nature 27 January 2016  
<https://www.nature.com/articles/nature16961>

Rollout Policy は、NN でなく線形の方策関数である。次の一手を探索時に高速に指し手確率を出すために使用する。

## 2-1. PolicyNet の教師あり学習

KGS Go Server (ネット囲碁対局サイト) の棋譜データから 3000 万局面分の教師データを用意し、教師と同じ指し手を予測できるよう学習を行った。具体的には、教師が指した手を 1 とし残りを 0 とした  $19 \times 19$  次元の配列を教師データとし、それを分類問題として学習した。

この学習で作成した **PolicyNet** は 57% ほどの精度である。

## 2-2. RollOutPolicy の教師あり学習

PolicyNet と同様の方法で学習する。作成した RollOutPolicy は 24% ほどの精度であった。

## 2-3. PolicyNet の強化学習

現状の PolicyNet と PolicyPool からランダムに選択された PolicyNet と対局シミュレーションを行い、その結果を用いて方策勾配法で学習を実施。

PolicyPool とは、PolicyNet の強化学習の過程を 500 Iteration ごとに記録し保存しておいたものである。現状の PolicyNet 同士の対局ではなく、PolicyPool に保存されているものを使用する理由は、対局に幅を持たせて過学習を防ごうというのが主である。この学習を minibatch size 128 で 1 万回実施。

## 2-4. ValueNet の学習

PolicyNet を使用して対局シミュレーションを行い、その結果の勝敗を教師として学習した。

教師データ作成の手順は以下となる。

1. まず SL PolicyNet(教師あり学習で作成した PolicyNet)で N 手まで打つ。

2. N+1 手目の手をランダムに選択し、その手で進めた局面を S(N+1)とする。
  3. S(N+1)から RL PolicyNet(強化学習で作成した PolicyNet)で終局まで打ち、その勝敗報酬を R とする。
- S(N+1)と R を教師データ対とし、損失関数を平均二乗誤差とし、回帰問題として学習。この学習を minibatch size 32 で 5000 万回実施。
- N 手までと N+1 手からの PolicyNet を別々にしてある理由は、過学習を防ぐためであると論文では説明されている。

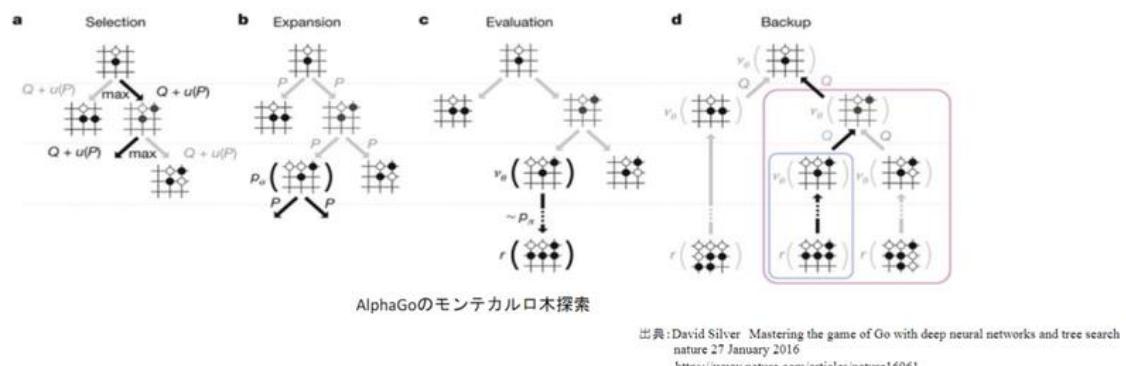
ValueNet の学習では、事前に学習した RollOutPolicy モデルを使って、盤面での一手一手の最終局面をモンテカルロ木探索で推定して Q 学習を早く終わらせている。このように計算量多い強化学習を教師あり学習で補うことで、ニューラルネットワークを強化しつつ学習を最適化している。

## 2-5. AlphaGo(Lee)のモンテカルロ木探索

モンテカルロ木探索は価値関数の学習で使用する。現局面の価値や勝率予想を出すのは困難であるため、現局面から最終局面まで PlayOut と呼ばれるランダムシミュレーションを多数回行い、その勝敗を集計して指し手の優劣を決定する手法である。

モンテカルロ木探索（図 9-4）は、該当手のシミュレーション回数が一定数を超えた後、その指し手の局面を開始局面として探索木の成長を行うことで、一定条件下で探索結果は最善手を返すということが論理的に証明されている。

図 9-4: AlphaGo(Lee)のモンテカルロ木探索



## AlphaGo(Lee) と AlphaGo Zero の違い

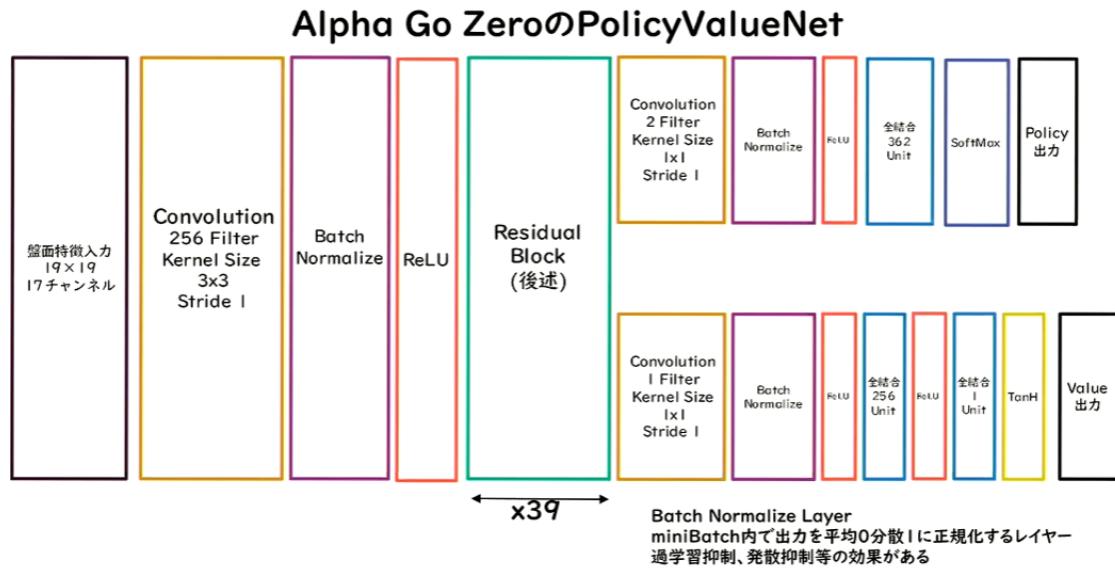
主な違いは次のとおりである。

- 1 .教師あり学習を一切行わず、**強化学習のみで作成**
- 2 .特徴入力から**ヒューリスティック**（経験的）な要素を排除し、**石の配置のみ**にした  
※事前に用意する**ハイパーパラメータ**がなくなった
- 3 .PolicyNet と ValueNet を 1 つのネットワークに統合した  
※ニューラルネットワークでは、似通った部分の**ネットワークの結合、分岐**を行うことは多々ある。
- 4 .**Residual Net** を導入した  
※画像データのように扱う

5. モンテカルロ木探索から RollOut シミュレーションをなくした

以下に、AlphaGo Zero の NW 構成を示す。

図 9-5: AlphaGo Zero の NW 構成



AlphaGo Zero では、AlphaGo(Lee)の構成要素に対して、新たに、Batch Normalize Layer と Residual Block Layer を追加している。

Batch Normalize Layer はミニバッチ内で出力を平均 0 分散 1 に正規化するレイヤーである。これにより、過学習抑制、発散抑制等の効果を得ている。

## 2-6. Residual Network

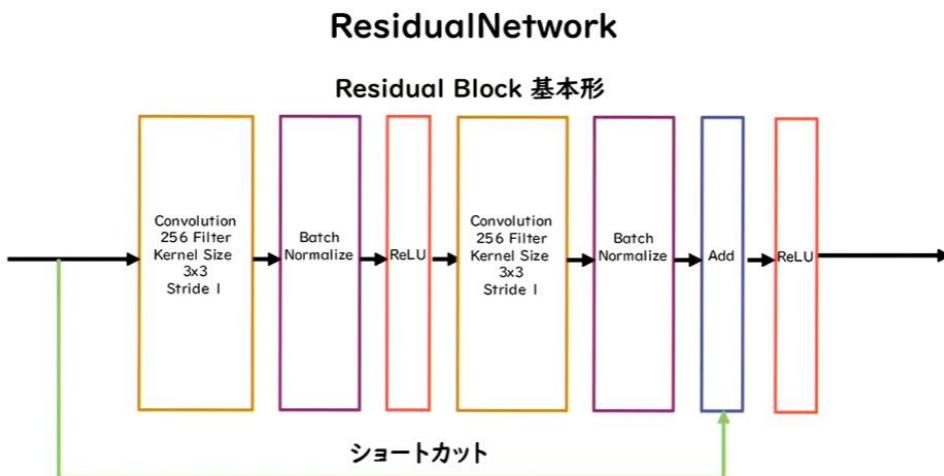
ネットワークにショートカット構造を追加して、勾配の爆発、消失を抑える効果を狙ったもの。Residual Network を使うことにより、100 層を超えるネットワークでも安定した学習が可能となった。

基本構造は、

Convolution→BatchNorm→ReLU→Convolution→BatchNorm→Add→ReLU  
の Block を 1 単位にして積み重ねる形 (AlphaGo Zero では 39 段) となる。

また、Residual Network を使うことにより、層数の違う Network のアンサンブル効果が得られているという説もある。※ブロック 39 段でショートカットするパスとショートカットしないパスが出来ることで異なるネットワークが表現され、複雑な学習が可能になることをアンサンブル効果となる。

図 9-6: Residual Network 構成



## 2-7. Residual Network の派生形

Residual Network で工夫したポイントを図 9-7 に示す。基本的に、本質的に重要で深層学習の発展へ寄与したのは以下の要素である。

- ・畠み込み
- ・プーリング
- ・RNN
- ・アテンション
- ・活性化関数

が基本的なパートとなっており、他はこれらの組合せである。

図 9-7: Residual Network 派生形の工夫点

### ResidualNetworkの派生形

**Residual Blockの工夫**  
Bottleneck  
1×1 KernelのConvolutionを利用し、1層目で次元削減を行って3層目で次元を復元する3層構造にして、2層のものと比べて計算量はほぼ同じだが1層増やせるメリットがある、としたもの

**PreActivation**  
ResidualBlockの並びをBatchNorm→ReLU→Convolution→BatchNorm→ReLU→Convolution→Addとすることにより性能が上昇したとするもの

**Network構造の工夫**  
WideResNet  
ConvolutionのFilter数をk倍にしたResNet。1倍→k倍×ブロック→2\*k倍×ブロックと段階的に幅を増やしていくのが一般的。Filter数を増やすことにより、浅い層数でも深い層数のものと同等以上の精度となり、またGPUをより効率的に使用できるため学習も早い

**PyramidNet**  
WideResNetで幅が広がった直後の層に過度の負担がかかり精度を落とす原因となっているとし、段階的にではなく、各層でFilter数を増やしていくResNet。

## Residual Network の性能

以下は、Residual Network を ImageNet の画像セットの分類を使ったときのネットワーク構造、および

エラーレートの評価結果である。18、34、50、101、152Layer で評価している。

性能は従来の VGG-16 と比較して、Top5 で 5.71 と 4%程度ものエラー率の低減に成功している。

layer name	output size	18-layer	34-layer	50-layer	101-layer	152-layer	model	top-1 err.	top-5 err.
conv1	112×112			7×7, 64, stride 2 3×3 max pool, stride 2			VGG-16 [41]	28.07	9.33
conv2_x	56×56	$\begin{bmatrix} 3 \times 3, 64 \\ 3 \times 3, 64 \end{bmatrix} \times 2$	$\begin{bmatrix} 3 \times 3, 64 \\ 3 \times 3, 64 \end{bmatrix} \times 3$	$\begin{bmatrix} 1 \times 1, 64 \\ 3 \times 3, 64 \\ 1 \times 1, 256 \end{bmatrix} \times 3$	$\begin{bmatrix} 1 \times 1, 64 \\ 3 \times 3, 64 \\ 1 \times 1, 256 \end{bmatrix} \times 3$	$\begin{bmatrix} 1 \times 1, 64 \\ 3 \times 3, 64 \\ 1 \times 1, 256 \end{bmatrix} \times 3$	GoogLeNet [44]	-	9.15
conv3_x	28×28	$\begin{bmatrix} 3 \times 3, 128 \\ 3 \times 3, 128 \end{bmatrix} \times 2$	$\begin{bmatrix} 3 \times 3, 128 \\ 3 \times 3, 128 \end{bmatrix} \times 4$	$\begin{bmatrix} 1 \times 1, 128 \\ 3 \times 3, 128 \\ 1 \times 1, 512 \end{bmatrix} \times 4$	$\begin{bmatrix} 1 \times 1, 128 \\ 3 \times 3, 128 \\ 1 \times 1, 512 \end{bmatrix} \times 4$	$\begin{bmatrix} 1 \times 1, 128 \\ 3 \times 3, 128 \\ 1 \times 1, 512 \end{bmatrix} \times 8$	PReLU-net [13]	24.27	7.38
conv4_x	14×14	$\begin{bmatrix} 3 \times 3, 256 \\ 3 \times 3, 256 \end{bmatrix} \times 2$	$\begin{bmatrix} 3 \times 3, 256 \\ 3 \times 3, 256 \end{bmatrix} \times 6$	$\begin{bmatrix} 1 \times 1, 256 \\ 3 \times 3, 256 \\ 1 \times 1, 1024 \end{bmatrix} \times 6$	$\begin{bmatrix} 1 \times 1, 256 \\ 3 \times 3, 256 \\ 1 \times 1, 1024 \end{bmatrix} \times 23$	$\begin{bmatrix} 1 \times 1, 256 \\ 3 \times 3, 256 \\ 1 \times 1, 1024 \end{bmatrix} \times 36$	plain-34	28.54	10.02
conv5_x	7×7	$\begin{bmatrix} 3 \times 3, 512 \\ 3 \times 3, 512 \end{bmatrix} \times 2$	$\begin{bmatrix} 3 \times 3, 512 \\ 3 \times 3, 512 \end{bmatrix} \times 3$	$\begin{bmatrix} 1 \times 1, 512 \\ 3 \times 3, 512 \\ 1 \times 1, 2048 \end{bmatrix} \times 3$	$\begin{bmatrix} 1 \times 1, 512 \\ 3 \times 3, 512 \\ 1 \times 1, 2048 \end{bmatrix} \times 3$	$\begin{bmatrix} 1 \times 1, 512 \\ 3 \times 3, 512 \\ 1 \times 1, 2048 \end{bmatrix} \times 3$	ResNet-34 A	25.03	7.76
	1×1			average pool, 1000-d fc, softmax			ResNet-34 B	24.52	7.46
		FLOPs	$1.8 \times 10^9$	$3.6 \times 10^9$	$3.8 \times 10^9$	$7.6 \times 10^9$	ResNet-34 C	24.19	7.40
							ResNet-50	22.85	6.71
							ResNet-101	21.75	6.05
							ResNet-152	<b>21.43</b>	<b>5.71</b>

### ImageNetのエラー率

出典: Kaiming He Deep Residual Learning for Image Recognition

arXiv.org 10 December 2015

<https://arxiv.org/abs/1512.03385>

## AlphaGo Zero の学習フロー

AlphaGo の学習は、自己対局による教師データの作成、学習、ネットワークの更新の 3ステップで構成される。

### 自己対局による教師データの作成

現状のネットワークでモンテカルロ木探索を用いて自己対局を行う。

まず30手までランダムで打ち、そこから探索を行い勝敗を決定する。

自己対局中の各局面での着手選択確率分布と勝敗を記録する。

教師データの形は(局面、着手選択確率分布、勝敗)が1セットとなる。

### 学習

自己対局で作成した教師データを使い学習を行う。

NetworkのPolicy部分の教師に着手選択確率分布を用い、Value部分の教師に勝敗を用いる。

損失関数はPolicy部分はCrossEntropy、Value部分は平均二乗誤差。

### ネットワークの更新

学習後、現状のネットワークと学習後のネットワークとで対局テストを行い、学習後の

ネットワークの勝率が高かった場合、学習後のネットワークを現状のネットワークとする。

### ◆実装演習結果キャプチャ

該当する実装の説明とコード提供がないため省略

### ◆確認テストなどの考察結果

該当する確認テストがないため省略

### ◆加点レポート

Wikipedia より、モンテカルロ木探索アルゴリズムの確認を行った。

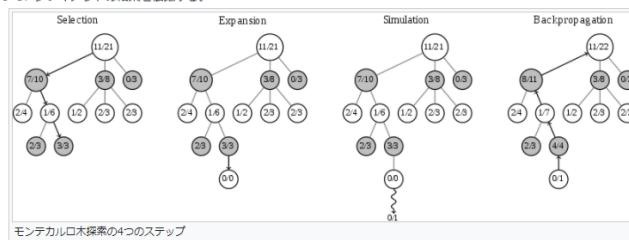
## アルゴリズム [編集]

モンテカルロ木探索は、最も良い手を選択するために使われ、ランダムサンプリングの結果に基づいて探索木を構築する。ゲームでのモンテカルロ木探索は、最後までプレイしたシミュレーション結果に基づいて構築する。ゲームの勝敗の結果に基づいてノードの値を更新して、最終的に勝率が高いことが見込まれる手を選択する。

最も単純な方法は、それぞれの有効な選択肢に、同数ずつプレイアウトの回数と一緒に割り振って、最も勝率が良かった手を選択する方法である<sup>[5]</sup>。これは単純なモンテカルロ木探索 (pure Monte Carlo tree search) と呼ばれる。過去のプレイアウト結果に基づき、よりプレイヤーの勝利に結びつく手にプレイアウトの回数をより多く割り振ると探索効率が向上する。

モンテカルロ木探索は4つのステップからなる<sup>[32]</sup>。

- 選択：根から始めて、葉ノードにたどり着くまで、子ノードを選択し続ける。根が現在のゲームの状態で、葉ノードはシミュレーションが行われていないノード。より有望な方向に木が展開していくように、子ノードの選択を片寄せる方法は、モンテカルロ木探索で重要なことであるが、探索と知識利用の所で後述する。
- 展開：ループ勝負を決するノードでない限り、叶から有効手の子ノードの中から1つ選ぶ。
- シミュレーション：Cから完全なランダムプレイアウトを行う。これはロールアウトとも呼ばれる。単純な方法としては、一様分布から手を選択してランダムプレイアウトを実行する。
- バックプロパゲーション：CからRへのパスに沿って、プレイアウトの結果を伝搬する。



上記のグラフは各ステップの選択を表している。ノードの数字は、そのノードからのプレイアウトの"勝った回数/プレイアウトの回数"を表している<sup>[33]</sup>。Selectionのグラフでは、今、黒の手番である。根ノードの数字は白が21回中11回勝利していることを表している。裏を返すと黒が21回中10回勝利していることを表していて、根ノードの下の3つのノードは手が3種類あることを表していて、数字を合算すると10/21になる。

シミュレーションで白が負けたとする。白の0/1ノードを追加して、そこから根ノードまでのパスの全てのノードの母分 (プレイアウトの回数) に1加算して、分子 (勝った回数) は黒ノードだけ加算する。引き分けの際は、0.5加算する。こうすることで、プレイヤーは最も有望な手を自分の手番で選択することが出来る。

計算の制限時間に到達するまで、これを反復し、最も勝率が高い手を選択する。

### <Section3: 軽量化・高速化技術>

#### ◆要点まとめ

##### I. 学習高速化技術

###### 3-1. 並列化処理

###### ・分散深層学習とは

- 深層学習は多くのデータを使用したり、パラメータ調整のために多くの時間を使用したりするため、高速な計算が求められる。
- 複数の計算資源(ワーカー)を使用し、並列的にニューラルネットを構成することで、効率の良い学習を行いたい。
- データ並列化、モデル並列化、GPUによる高速技術は不可欠である。

現在、並列化処理による効率的な学習はビジネス的に解決したい課題である。例えば、スマホ、パソコンを含むいろいろなコンピュータの有休時の計算資源(ワーカー)を最大限活用するなどのモデルもある。

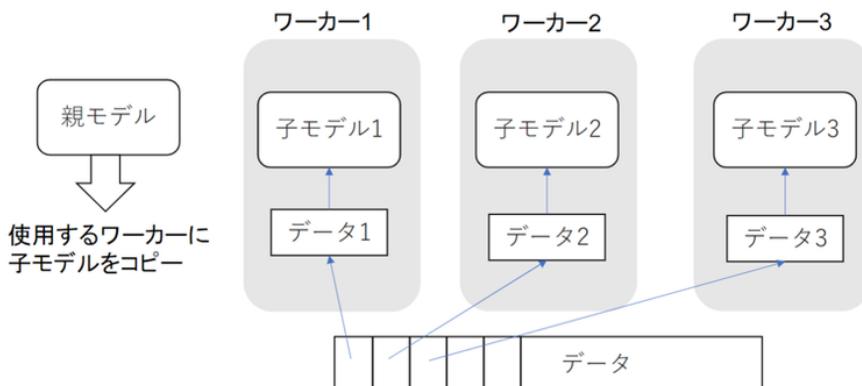
ハードウェアは、**ムーアの法則**で言われるように半導体業界において、一つの集積回路（ICチップ）に実装される素子の数は**18ヶ月(1年6ヶ月)**ごとに倍増するという経験則があるが、それに対して、ソフトウェアによる学習モデルは、**毎年10倍の速度でモデルの複雑性やデータ量が増加**していると言われ、ハードウェアの進化が追いついていない。

基本的な解決策としては、

- コンピュータ(ハードウェア)の台数を増やして通信ネットワークを介して、いろいろなコンピュータでデータ並列・分散処理をする。
- 通信ネットワークを介すると遅延要素となるので、コンピュータ(ハードウェア)内部でCPU、GPU及びTPUを増設して、それらを組み合わせて計算力を強化するなど、何かしらの工夫しながらモデル学習させている。

###### 3-2. データ並列

- ・親モデルを各ワーカーに子モデルとしてコピー
- ・データを分割し、ワーカーごとに計算させる



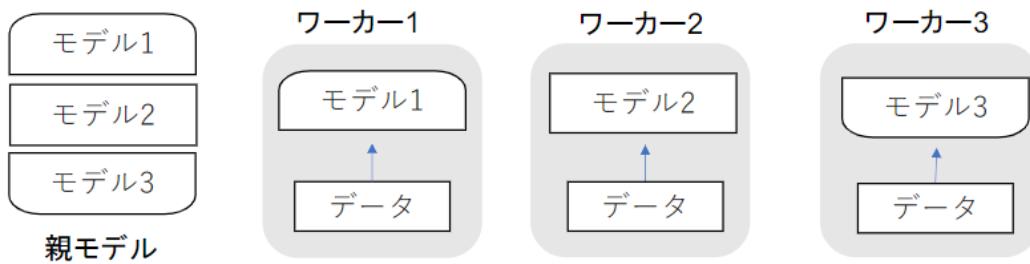
データ並列化は各ワーカーから得られたパラメータの更新方法で、同期型か非同期型かが決まる。表 10-1 に、同期型と非同期型の特徴を示す。

表 10-1: データ並列化での同期型と非同期型の特徴

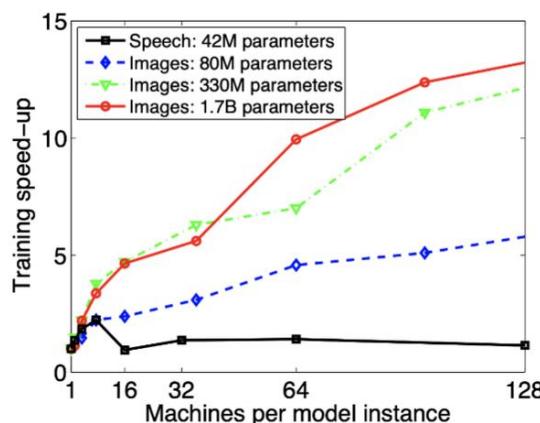
	同期型	非同期型
概要	<p>パラメータ更新の流れ :</p> <p><b>各ワーカーの計算が終わるのを待ち、全ワーカーの勾配が出たところで勾配の平均を計算し、親モデルのパラメータを更新する。</b></p> <p>再度、学習する際には、更新後のモデルを各ワーカーに子モデルとしてコピーする。</p>	<p>パラメータ更新の流れ :</p> <p><b>各ワーカーはお互いの計算を待たず、子モデルごとに更新を行う。</b></p> <p>学習が終わった子モデルはパラメータサーバに Push する。</p> <p>新たに学習を始める時は、パラメータサーバから pop した最新のモデルに対して学習していく。</p>
処理イメージ		
メリット/デメリット	<ul style="list-style-type: none"> <li>・処理スピードは、全ワーカーの計算完了を待つため、一番遅いワーカーに律速し、非同期型より遅くなる</li> <li>・常に最新のパラメータで学習できるため、安定した学習が可能</li> <li>→現在は、同期型の方が、精度が良いことが多いため主流</li> </ul>	<ul style="list-style-type: none"> <li>・処理スピードは、お互いのワーカーの計算完了を待たないため同期型より早い</li> <li>・最新のモデルのパラメータが利用できない場合があるため、学習が不安定になることがある</li> </ul>

### 3-3. モデル並列

- ・親モデルを各ワーカーに分割し、それぞれのモデルを学習させる。全てのデータで学習が終わった後で、一つのモデルに復元。
- 実際には、1回の学習ごとに誤差によるフィードバックが必要なため、PCに複数接続された GPU を使ってモデル並列を行う
- ・モデルが大きいときはモデル並列化を、データが大きいときはデータ並列化をすると良い。



下図の様に、モデルのパラメータ数が多い（モデルが大きい）ほど、モデル並列化によるスピードアップ効率も向上する。逆にモデルが小さいと、ワーカー間の通信がボトルネックになってスピードが出ない。



(Jeffrey Dean et al. 2016)  
Large Scale Distributed Deep Networks

#### 参考論文

- Large Scale Distributed Deep Networks
- Google 社が 2016 年に出した論文
- Tensorflow の前身といわれている

この論文では、

- 並列コンピューティングを用いることで大規模なネットワークを高速に学習させる仕組みを提案。
- 主にモデル並列とデータ並列(非同期型)の提案をしている。

#### 3-4. GPU

ニューラルネットワークの学習は、CPU、または、GPU などにより行うが、表 10-2 に示す特性により、CPU より GPU を使う方が高速化可能である。元々の使用目的であるグラフィックス以外の用途で使用される GPU の総称として、GPGPU (General Purpose GPU) と呼ぶ。

表 10-2: CPU と GPU の特性

CPU	GPU
-----	-----

<ul style="list-style-type: none"> <li>・高性能なコアで構成。コア数は少数</li> <li>・複雑で連続的な処理が得意</li> </ul>	<ul style="list-style-type: none"> <li>・比較的低性能なコアが多数</li> <li>・簡単な並列処理が得意</li> </ul> <p>→ニューラルネットワークの学習は単純な行列演算が多いので、GPUによる高速化が可能</p>
--	---

GPGPUによる開発環境としては、表 10-3 に示す環境がある。

表 10-3: GPGPU 開発環境

CUDA	OpenCL	Deep Learning フレームワーク
<ul style="list-style-type: none"> <li>・GPU上で並列コンピューティングを行うためのプラットフォーム</li> <li>・NVIDIA 社が開発している GPUのみで使用可能</li> <li>・Deep Learning 用に提供されているので使いやすい →CUDA が主流となっており、利用者は並列化の実装を意識することなく利用可能</li> </ul>	<ul style="list-style-type: none"> <li>・オープンな並列コンピューティングのプラットフォーム</li> <li>・NVIDIA 社以外の会社 (Intel、AMD、ARMなど) の GPUでも使用可能</li> <li>・Deep Learning 用の計算に特化しているわけではない</li> </ul>	<ul style="list-style-type: none"> <li>・Deep Learning フレームワーク (Tensorflow、Pytorch) 内で実装されているので、使用する際は指定すれば良い</li> </ul>

## II. 軽量化技術

モデルの軽量化とは、モデルの精度を維持しつつパラメータや演算回数を低減する手法の総称。ディープラーニングでは、高メモリ負荷や高い演算性能が必要であるため、低メモリで低演算性能のスマホや IoT などのデバイスに向けに使われる技術である。モデルの軽量化は計算の高速化と省メモリ化を行うためモバイル、IoT 機器と相性が良い手法となる。

代表的な軽量化の手法として下記の 3 つがある。

- **量子化**：重みの精度を下げることにより計算の高速化と省メモリ化を行う技術
- **蒸留**：複雑で精度の良い教師モデルから軽量な生徒モデルを効率よく学習を行う技術
- **プルーニング**：寄与の少ないニューロンをモデルから削減し高速化と省メモリ化を行う技術

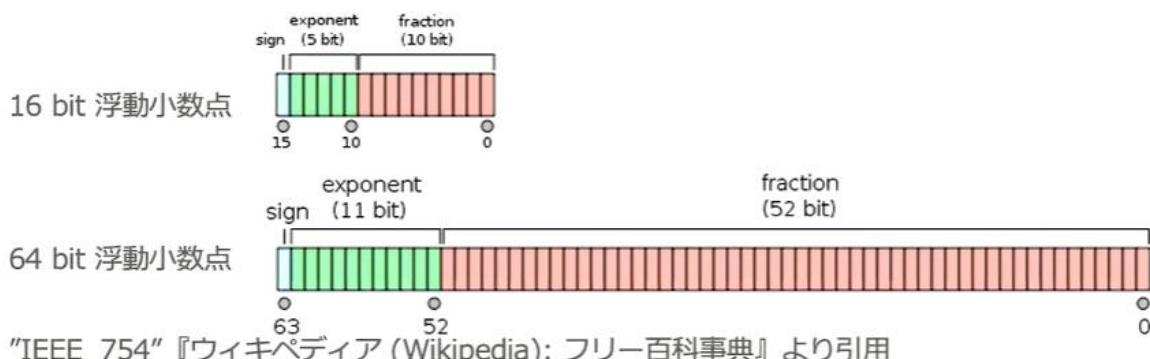
### 3-5. 量子化 (Quantization)

ネットワークが大きくなると大量のパラメータが必要になり学習や推論に多くのメモリと演算処理が必要となる。そこで、通常のパラメータの 64 bit 浮動小数点を 32 bit など下位の精度に落とすことでメモリと演算処理の削減を行う。

例えば、機械翻訳などの自然言語処理を行うための BERT の学習モデルはいろいろあるが、小さいモデルでも数百 MB のメモリを必要とする。ニーズとしてはスマホのメモリ上で動作させたいが、重みの情報だけでも大量となるので、そのパラメータをメモリ上に保持するだけで大変である。そこで、量子化によりスマホに実装を行う。

浮動小数点は下図のとおり、ビット配列で実現されており、符号部 (sign) 、指数部 (exponent) 、力

仮数部 (fraction) で表現されており、このようにビットで表現することを**量子化**という。



量子化の利点と欠点は以下となる。



計算の高速化として、倍精度演算(64 bit)と单精度演算(32 bit)とでは、演算性能が大きく違うため、量子化により精度を落とすことにより多くの計算をすることができる。

以下に、NVIDIA の单精度と倍精度の GPU 性能を示す。さらに、半精度だと 100T-FLOPS ぐらいとなる。16bit が半精度、32bit が单精度、64bit が倍精度となる。

FLOPS( Floating point number Operations Per Second)は、名称が示す通り、1 秒間に浮動小数点演算が何回できるかの指標値、ひいては性能値の事を指す。 (Wikipedia より)

	单精度	倍精度
NVIDIA® Tesla V100™	15.7 TeraFLOPS	7.8 TeraFLOPS
NVIDIA® Tesla P100™	9.3 TeraFLOPS	4.7 TeraFLOPS

<https://www.nvidia.com/ja-jp/data-center/tesla-v100/>  
<https://www.nvidia.co.jp/object/tesla-p100-jp.html>

学習結果が 0 と 1 しかないような極端なケースを除けば、半精度、16bit のモデルで実用上は十分であるが、極端な量子化は精度が落ちるためバランスが必要である。例えば表現できる値が 0, 1 の 1 bit の場合で、 $a=0.1$  が真値の時、関数  $y(x) = ax$  を近似する場合を考えると、学習によって  $a$  が 0.1 を得る必要がある。しかし、量子化によって  $a$  が表現できる値が 0 か 1 であるため、求められる式は、

$$y(x) = 0 \text{ または } y(x) = x$$

となり、誤差の大きな式になってしまう。

そのため、量子化する際は極端に精度が落ちない程度に量子化をしなければならない。

速度や精度に関するいろいろな実験が行われているので、その事例を以下に示す。

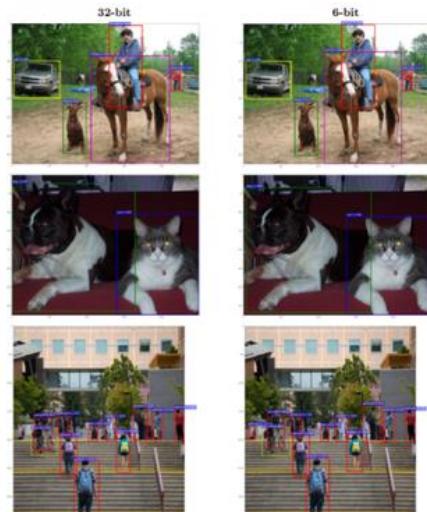
事例 1 :

## 速度の実験結果

右の図は 32bit と 6bit のモデルで検出を行った画像になる

それぞれの検出の時間は下記の表のようになる

32 bit	6 bit
0.507s	0.098s
0.441s	0.106s
32.269s	6.113s



Penghang Yin , Shuai Zhang , Yingyong Qi , and Jack Xin (2017) 「Quantization and Training of Low Bit-Width Convolutional Neural Networks for Object Detection」 <https://arxiv.org/pdf/1612.06052.pdf>

事例 2 :

## 精度の実験結果

右の図は量子化を行い検出した矩形の図になる。  
量子化することにより目的のオブジェクト以外の領域が多くなっている



下記の図は量子化を行わない場合と量子化を行った際の精度になる。

R-FCN, ResNet-50	mAP	R-FCN, ResNet-101	mAP
4-bit LBW	74.37%	4-bit LBW	76.79%
5-bit LBW	76.99%	5-bit LBW	77.83%
6-bit LBW	77.05%	6-bit LBW	78.24%
32-bit full-precision	77.46%	32-bit full-precision	78.94%

Penghang Yin , Shuai Zhang , Yingyong Qi , and Jack Xin (2017) 「Quantization and Training of Low Bit-Width Convolutional Neural Networks for Object Detection」 <https://arxiv.org/pdf/1612.06052.pdf>

事例 3 :

学習に特化するため、Google TPU では、bf16 という浮動小数点のフォーマットを定義している。

bfloat16 形式は、符号ビットが 1 つ、指数ビットが 8 つ、仮数ビットが 7 つ、暗黙の仮数ビットが 1 つの [1:8:7] の形式です。これに対し、標準の 16 ビット浮動小数 (fp16) 形式は [1:5:10] です。fp16 形式の場合、指数ビットは 5 つのみです。このような特性のため、bfloat16 の方が fp16 よりダイナミックレンジが広くなります。bfloat16 のレンジは、fp16 のダイナミックレンジの外にある可能性があるため損失のスケーリングが必要な勾配などに役立ちます。bfloat16 は、このような勾配を直接表現できます。さらに bfloat16 形式を使用すると、すべての整数 [-256, 256] を正確に表現できます。つまり、精度を失わずに int8 を bfloat16 でエンコードできます。

次の図は、3 つの浮動小数点形式を示しています。

- fp32 - IEEE の単精度浮動小数点
- fp16 - IEEE の半精度浮動小数点
- bfloat16 - 16 ビットの brain 浮動小数点



bfloat16 のダイナミックレンジは、fp16 のダイナミックレンジよりも大きくなります。

### 3-5. 蒸留 (Distillation)

蒸留は、以下の教師モデル-生徒モデルで構成され、すでに学習済みの教師モデルが一種の正解ラベルとなり、両者の誤差から生徒モデルが重みパラメータ学習して教師モデルから生徒モデルへモデルの引継ぎを行うことである。

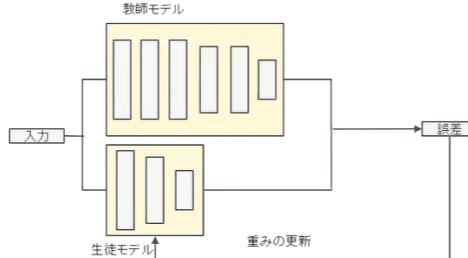
#### 蒸留

モデルの簡約化		
• 学習済みの精度の高いモデルの知識を軽量なモデルへ継承させる • 知識の形勝により、軽症でありながら複雑なモデルに匹敵する精度のモデルを得ることが期待できる		
• 精度の高いモデルはニューロンの規模が大きいモデルになっている、そのため、推論に多くのメモリと演算処理が必要となる	知識の継承 →	• 規模の大きなモデルの知識を使って軽量なモデルの作成を行う
精度の高いモデル  精度の高いモデル 精度 - 高い 計算コスト - 高い		軽量なモデル  軽量なモデル 精度 - 普通 計算コスト - 低い

## 教師モデルと生徒モデル

教師モデルの重みを固定し生徒モデルの重みを更新していく

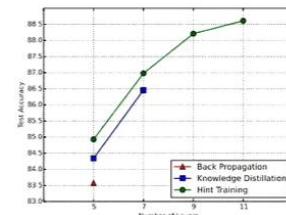
誤差は教師モデルと生徒モデルのそれぞれの誤差を使い重みを更新していく



以下のグラフは、Cifar10 データセットで学習を行ったレイヤー数と精度のグラフであり、軽量化しているモデル間の関係が読み取れる。赤が計量モデルの通常学習時、青が教師モデル—生徒モデルでの軽量モデルの学習時、緑はこの論文の手法適用時の計量モデルの結果である。大規模のモデルは、数百層となるが、蒸留によって 11 層の小さいモデルへ変換している。

表の back propagation は通常の学習、Knowledge Distillation は先に説明した蒸留手法、Hint Training は蒸留は引用論文で提案された蒸留手法

図から蒸留によって少ない学習回数でより精度の良いモデルを作成することができている

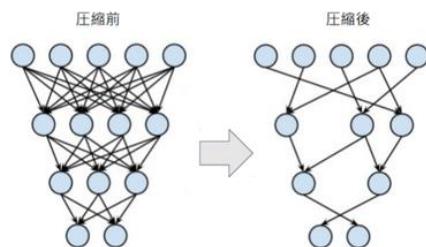


Adriana Romero, Nicolas Ballas, Samira Ebrahimi Kahou, Antoine Chassang, Carlo Gatta, Yoshua Bengio (2015) 「FITNETS: HINTS FOR THIN DEEP NETS」, <https://arxiv.org/pdf/1412.6550.pdf>.

## 3-6. プルーニング (Pruning)

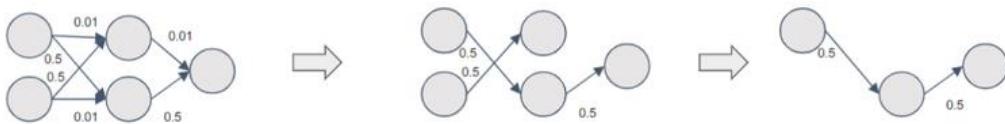
プルーニングは、大きなニューラルネットワークにおいて、以下の様に、使っていないパラメータを削除してスカスカの軽量化を図ること。

寄与の少ないニューロンの削減を行いモデルの圧縮を行うことで高速化に計算を行うことができる



Song Han, Jeff Pool, John Tran, William J. Dally (2015) 「Learning both Weights and Connections for Efficient Neural Networks」, <https://arxiv.org/pdf/1506.02626.pdf>.

ニューロンの削減の手法は、重みが閾値以下の場合ニューロンを削減し再学習を行う。下記の例は重みが 0.1 以下のニューロンを削減した例である。重み 0.1 以下のニューロンを削減することで、出力に関与しないノードも合わせて不要な NW を消している。



プルーニングにより精度をあまり劣化させずに軽量化を行える。以下の CaffeNet の例では、閾値のパラメータ  $\alpha$  を 0.5 から 2.6 まで変更して、全結合層全体を約 50%から 99%まで削減したとしても、精度は 91.66%から 87.08%とあまり変わらないことがわかる。

下記の表は Oxford 102 category flower dataset を CaffeNet で学習したモデルのプルーニングの閾値による各層と全体のニューロンの削減率と精度をまとめたものになる。 $\alpha$  は閾値のパラメータ、閾値は各層の標準偏差  $\sigma$  とパラメータの積

閾値を高くするとニューロンは大きく削減できるが精度も減少する

パラメータ削減率 (%)				
$\alpha$	全結合層 1	全結合層 2	全結合層 3	全結合層全体
0.5	49.54	47.13	57.21	48.86
0.6	57.54	55.14	64.99	56.86
0.8	70.96	69.04	76.44	70.42
1.0	80.97	79.77	83.74	80.62
1.3	90.51	90.27	90.01	90.43
1.5	94.16	94.28	92.45	94.18

パラメータ削減率 (%)				
$\alpha$	全結合層 1	全結合層 2	全結合層 3	全結合層全体
1.6	95.44	95.64	93.38	95.49
1.7	96.41	96.66	94.20	96.47
1.8	97.17	97.43	94.88	97.23
1.9	97.75	98.02	95.45	97.81
2.0	98.20	98.45	95.94	98.26
2.2	98.80	99.03	96.74	98.85
2.4	99.19	99.40	97.41	99.24
2.6	99.46	99.64	97.95	99.50

佐々木健太、佐々木勇和、鬼塚 真（2015）「ニューラルネットワークの全結合層における パラメータ削減手法の比較」, <https://db-event.ipn.org/deim2017/papers/62.pdf>.

#### ◆ 実装演習結果キャプチャ

該当する実装の説明とコード提供がないため省略

#### ◆ 確認テストなどの考察結果

該当する確認テストがないため省略

## <Section4: 応用モデル>

### ◆要点まとめ

#### 4-1. MobileNet

##### 論文タイトル

MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications

##### 提案手法

- ディープラーニングモデルは精度が良いが、その分ネットワークが深くなり計算量が増える
- 計算量が増えると、多くの計算リソースが必要で、お金がかかってしまう
- ディープラーニングモデルの軽量化・高速化・高精度化を実現  
(その名の通りモバイルなネットワーク)

<https://qiita.com/HiromuMasuda0228/items/7dd0b764804d2aa199e4>

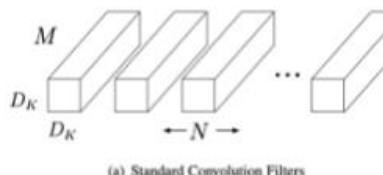
近年の画像認識タスクに用いられる最新のニューラルネットワークアーキテクチャは、多くのモバイルおよび組み込みアプリケーションの実行環境を上回る高い計算資源を必要とされる。

<https://arxiv.org/pdf/1704.04861.pdf>

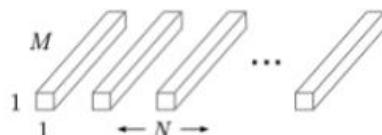
図 11-1: MobileNet 構成図



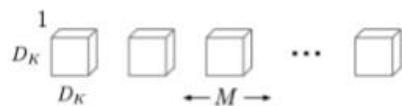
Figure 1. MobileNet models can be applied to various recognition tasks for efficient on device intelligence.



(a) Standard Convolution Filters



(c) 1 × 1 Convolutional Filters called Pointwise Convolution in the context of Depthwise Separable Convolution



(b) Depthwise Convolutional Filters

Figure 2. The standard convolutional filters in (a) are replaced by two layers: depthwise convolution in (b) and pointwise convolution in (c) to build a depthwise separable filter.

## 一般的な畳み込みレイヤーの計算量

一般的な畳み込みレイヤーの計算量を下図に示す。

図 11-2: 一般的な畳み込みレイヤーの計算量

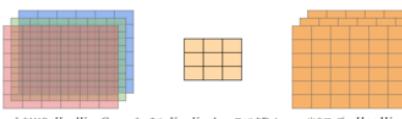
### 一般的な畳み込みレイヤー

<ul style="list-style-type: none"> <li>・入力特徴マップ(チャネル数) :</li> <li>・畳込みカーネルのサイズ :</li> <li>・出力チャネル数(フィルタ数) :</li> <li>・ストライド 1 でパディングを適用した場合の 畳み込み計算の計算量?</li> <li>・入力MAPカーネルフィルタ数出力マップ この点を計算するための計算量は小さい造れる</li> </ul>	<p>入力MAP <math>H \times W \times C</math> カーネル <math>K \times K \times C</math> フィルタ数 <math>M</math> 出力マップ <math>H \times W \times M</math></p>
<ul style="list-style-type: none"> <li>・入力特徴マップ(チャネル数) :</li> <li>・畳込みカーネルのサイズ :</li> <li>・出力チャネル数(フィルタ数) :</li> <li>・ストライド 1 でパディングを適用した場合の この点を計算するための計算量は <math>K \times K \times C \times M</math> ※普通の畳み込みで、3 6 ピクセル、3 チャンネル フィルターでいろいろな個性の表現可能 3 つの情報で3カーネル</li> </ul>	<p>入力MAP <math>H \times W \times C</math> カーネル <math>K \times K \times C</math> フィルタ数 <math>M</math> 出力マップ <math>H \times W \times M</math></p>
<ul style="list-style-type: none"> <li>・入力特徴マップ(チャネル数) :</li> <li>・畳込みカーネルのサイズ :</li> <li>・出力チャネル数(フィルタ数) :</li> <li>・ストライド 1 でパディングを適用した場合の この点を計算するための計算量は <math>H \times W \times K \times K \times C \times M</math> ※一回だけの畳み込みの計算量を表した図</li> </ul>	<p>入力MAP <math>H \times W \times C</math> カーネル <math>K \times K \times C</math> フィルタ数 <math>M</math> 出力マップ <math>H \times W \times M</math></p>

## MobileNets: Depthwise Convolution と Pointwise Convolution

図 11-2 のように、一般的な畳み込みレイヤーは全部の畳み込み計算を行うので計算量が多くなる。そこで、**MobileNets** では **Depthwise Convolution** と **Pointwise Convolution** の組み合わせで軽量化を実現している。

Depthwise Convolution	Pointwise Convolution
<p>仕組み</p> <ul style="list-style-type: none"> <li>・入力マップのチャネルごとに畳み込みを実施</li> <li>・出力マップをそれらと結合 (入力マップのチャネル数と同じになる)</li> <li>・通常の畳み込みカーネルは全ての層にかかっていることを考えると計算量が大幅に削減可能</li> <li>・各層ごとの畳み込みなので層間の関係性は全く考慮されない。通常はPW畳み込みとセットで使うことで解決</li> <li>　　出力マップ(一般的な畳み込み層)の計算量は <math>H \times W \times K \times K \times C \times M</math></li> <li>　　出力マップの計算量は? <math>H \times W \times C \times K \times K</math></li> </ul>	<p>仕組み</p> <ul style="list-style-type: none"> <li>・<math>1 \times 1</math> convとも呼ばれる(正確には<math>1 \times 1 \times C</math>)</li> <li>・入力マップのポイントごとに畳み込みを実施</li> <li>・出力マップ(チャネル数)はフィルタ数分だけ作成可能 (任意のサイズが指定可能)</li> </ul> <p>出力マップの計算量は <math>H \times W \times C \times M</math></p>



・**Depthwise Convolution** は、**フィルタを1固定にして1チャンネル毎に処理し**、3チャンネル同時にしない。Mを固定化 $\Rightarrow$ 消したことにより計算量が減り、入力と出力が同じサイズで出力される。

・**Pointwise Convolution** は、カーネルを $1 \times 1$ としてフィルタMを変化させることにより、 $K \times K$ の部分が削減されて計算量が減る。

この両方を組み合わせることで一般的な畳み込みと同等にすると共に、個々の削減を活かしながらトータルで計算量を削減できないかの発想で考え出された。

## 4-2. DenseNet

### 論文タイトル

- Densely Connected Convolutional Networks. G. Huang et., al. 2016

<https://arxiv.org/pdf/1608.06993.pdf>

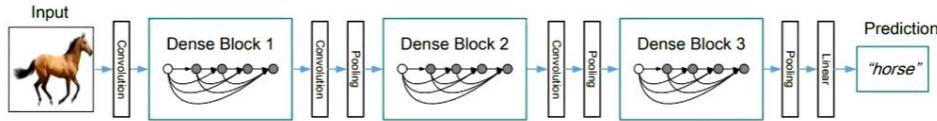
<https://www.slideshare.net/harmonylab/densely-connected-convolutional-networks>

### 概要

- Dense Convolutional Network (以下、DenseNet) は、畳み込みニューラルネットワーク (以下、CNN) アーキテクチャの一種である。
- ニューラルネットワークでは層が深くなるにつれて、学習が難しくなるという問題があったが、Residual Network (以下、ResNet) などの CNN アーキテクチャでは前方の層から後方の層へアイデンティティ接続を介してパスを作ることで問題を対処した。
- DenseNet も DenseBlock と呼ばれるモジュールを用いた ResNet と同じようなアーキテクチャの一つである。

図 11-3: DenseNet 構成図

- 初期の畠み込み
- Dense ブロック
- 変換レイヤー
- 判別レイヤー

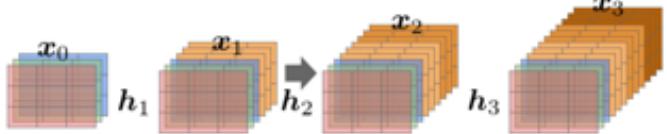
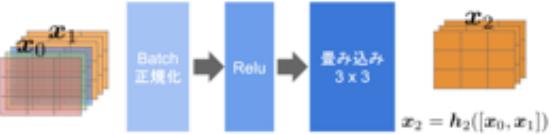
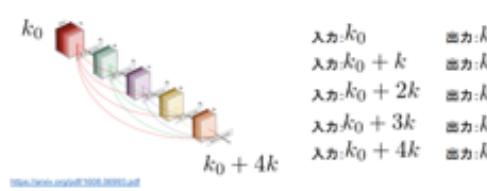
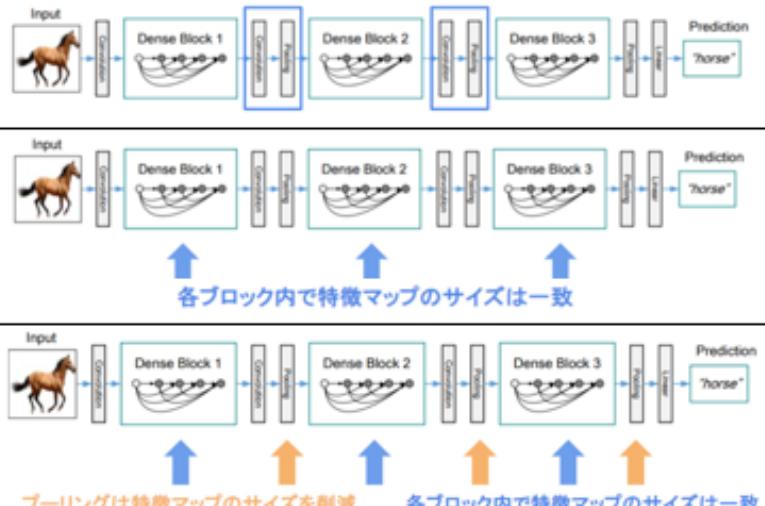


<https://arxiv.org/pdf/1608.06993.pdf>

DenseNet では Dense ブロックを持つことが特徴的で、図 11-4 で Dense ブロックの働きを示す。1 つの Dense ブロックでは、 $k_0$  チャンネルのインプットに対して、 $k$  チャンネルずつ追加する処理を行っている。4 つの NW を通ると出力チャンネルは、 $k_0+4k$  となる。

こうして増えたチャンネル数を、Dense ブロックをつなぐ、Convolution と Pooling から構成される Transition Layer で、チャンネル数を減らすダウンサンプリングを行う。Transition Layer により、各 Dense ブロックの特徴マップサイズは同じである。

図 11-4: Dense ブロックの働き

<ul style="list-style-type: none"> <li>出力層に前の層の入力を足し合わせる</li> <li>層間の情報の伝達を最大にするために全ての同特徴量サイズの層を結合する</li> </ul>											
<ul style="list-style-type: none"> <li>特徴マップの入力に対し、下記の処理で出力を計算</li> <li>-Batch正規化</li> <li>-Relu関数による変換</li> <li>-<math>3 \times 3</math>畳み込み層による処理</li> </ul>											
<ul style="list-style-type: none"> <li>前スライドで計算した出力に入力特徴マップを足し合わせる</li> <li>-入力特徴マップのチャンネル数が<math>l \times k</math>だった場合、出力は<math>(l+1) \times k</math>となる</li> <li>第<math>l</math>層の出力をとすると右式</li> </ul>	$x_l = H_l([x_0, x_1, \dots, x_{l-1}])$ 										
<ul style="list-style-type: none"> <li><math>k</math>をネットワークのgrowth rateと呼ぶ</li> <li><math>k</math>が大きくなるほど、ネットワークが大きくなるため、小さな整数に設定するのがよい</li> </ul>	 <table border="1"> <tr> <td>入力: <math>k_0</math></td> <td>出力: <math>k</math></td> </tr> <tr> <td>入力: <math>k_0 + k</math></td> <td>出力: <math>k</math></td> </tr> <tr> <td>入力: <math>k_0 + 2k</math></td> <td>出力: <math>k</math></td> </tr> <tr> <td>入力: <math>k_0 + 3k</math></td> <td>出力: <math>k</math></td> </tr> <tr> <td>入力: <math>k_0 + 4k</math></td> <td>出力: <math>k</math></td> </tr> </table> <p><a href="https://arxiv.org/pdf/1608.06993.pdf">https://arxiv.org/pdf/1608.06993.pdf</a></p>	入力: $k_0$	出力: $k$	入力: $k_0 + k$	出力: $k$	入力: $k_0 + 2k$	出力: $k$	入力: $k_0 + 3k$	出力: $k$	入力: $k_0 + 4k$	出力: $k$
入力: $k_0$	出力: $k$										
入力: $k_0 + k$	出力: $k$										
入力: $k_0 + 2k$	出力: $k$										
入力: $k_0 + 3k$	出力: $k$										
入力: $k_0 + 4k$	出力: $k$										
<ul style="list-style-type: none"> <li>Transition Layer</li> <li>-CNNでは中間層でチャネルサイズを変更し</li> <li>-特徴マップのサイズを変更し、ダウンサンプリングを行うため、Transition Layerと呼ばれる層でDense blockをつなぐ</li> </ul>	 <p><a href="https://arxiv.org/pdf/1608.06993.pdf">https://arxiv.org/pdf/1608.06993.pdf</a></p>										

### DenseNet と ResNet の違い

DenseBlock では前方の各層からの出力全てが後方の層への入力として用いられる。一方、ResidualBlock では前 1 層の入力のみ後方の層へ入力する違いがある。

※ResNet は前の層のみからに対して DenseBlock は全部の層から入力される

### 成長率 (Growth Rate)

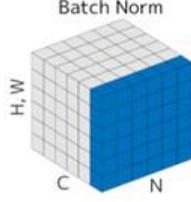
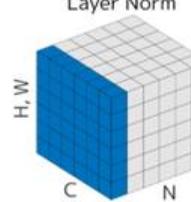
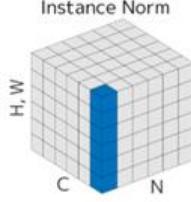
DenseNet 内で使用される DenseBlock と呼ばれるモジュールでは成長率 (Growth Rate) と呼ばれるハイパーパラメータが存在する。DenseBlock 内のノード毎に  $k$  個ずつ特徴マップのチャネル数が増加してい

く時、k を成長率と呼ぶ。

### 4-3. 正規化

以下に 3 種類の正規化の手法を示す。

図 11-5: 正規化の手法

正規化		
Batch Norm	Layer Norm	Instance Norm
ミニバッチに含まれるsampleの同一チャネルが同一分布に従うよう正規化 	それぞれのsampleの全てのpixelsが同一分布に従うよう正規化 	さらにchannelも同一分布に従うよう正規化 

N: ミニバッチ数C: ChannelH, W: Height / Widthをまとめた

論文参照: <https://arxiv.org/pdf/1803.08494.pdf>

#### 4-3-1. Batch Norm

BatchNorm は、

- レイヤー間を流れるデータの分布を、**ミニバッチ単位で平均が 0、分散が 1 になるように正規化**
- Batch Normalization はニューラルネットワークにおいて**学習時間の短縮や初期値への依存低減、過学習の抑制など効果**

がある。

Batch Norm の問題点は、Batch Size が小さい条件下では、学習が収束しないことがあり、代わりに Layer Normalization などの正規化手法が使われることが多い。

- Batch Norm は、ミニバッチ化単位で処理する必要があり、扱いづいため、あまり使われない。
- $H \times W \times C$  の画像が N 個あった場合に、N 個の同一チャネル( $H \times W \times N$ )が正規化の単位となる RGB の 3 チャンネルの sample が N 個の場合は、それぞれのチャネルの平均と分散を求め 正規化を実施（図 11-5 の青い部分）。チャネルごとに正規化された特徴マップを出力。
- **ミニバッチのサイズを大きく取れない場合には、効果が薄くなってしまう。**

論文: <https://arxiv.org/pdf/1803.08494.pdf>

#### 4-3-2. Layer Norm

-N 個の sample のうち一つに注目。 $H \times W \times C$  の全ての pixel が正規化の単位。

-RGB の 3 チャンネルの sample が N 個の場合は、ある sample を取り出し、

全てのチャネルの平均と分散を求め正規化を実施（図 11-5 の青い部分）。

特徴マップごとに正規化された特徴マップを出力。

-ミニバッチ数に依存しないので、Batch Norm の問題を解消できている。

論文: <https://arxiv.org/pdf/1803.08494.pdf>

Layer Norm は、入力データや重み行列に対して、以下の操作を施しても、出力があまり変わらない（ロバストである）ことが知られている。

-入力データのスケールに関してロバスト（堅牢性）

-重み行列のスケールやシフトに関してロバスト

-詳細は下記のスライド参照

■ <https://www.slideshare.net/KeigoNishida/layer-normalizationnips>

#### 4-3-3. Instance Norm

-各 sample の各チャンネル( $H \times W$ )が正規化の対象。

何個も sample を集めないで、個々を正規化して、うまくデータの特徴を合わせる。

※Batch Norm のバッチサイズが 1 の場合と等価

-RGB の 3 チャンネルの sample が N 個の場合は、ある sample を取り出し、

そのうち 1 チャンネルの平均と分散を求め正規化を実施（図 11-5 の青い部分）。

Instance Norm は、コントラストの正規化に寄与・画像のスタイル転送やテクスチャ合成で利用される。

論文: <https://blog.cosnomi.com/posts/1493/>

<https://gangango.com/2019/06/16/post-573/>

[https://blog.albert2005.co.jp/2018/09/05/group\\_normalization/](https://blog.albert2005.co.jp/2018/09/05/group_normalization/)

#### 4-4. Wavenet

- Wavenet

- Aaron van den Oord et. al., 2016 により提案

- AlphaGo のプログラムを開発しており、2014年に google に買収される

- 生の音声波形を生成する深層学習モデル

- Pixel CNN(高解像度の画像を精密に生成できる手法)を音声に応用したもの

- 関連記事

- <https://gigazine.net/news/20171005-wavenet-launch-in-google-assistant/>

- <https://qiita.com/MasaEguchi/items/cd5f7e9735a120f27e2a>

- [https://www.slideshare.net/NU\\_I\\_TODALAB/wavenet-86493372](https://www.slideshare.net/NU_I_TODALAB/wavenet-86493372)

Wavenet は音声生成モデルで用いられる。畳み込みは入力データに対する受容野を広く出来る利点があるため、Wavenet でも、音声の時系列データに対して畳み込みを適用し受容野を広めている。Wavenet で

は畳み込みを層が深くなるにつれて時間間隔を離すことで、より広い範囲の特徴を学習できる。

- Wavenetのメインアイディア

- 時系列データに対して畳み込み(Dilated convolution)を適用する
- Dilated convolution
  - 層が深くなるにつれて畳み込むリンクを離す
  - 受容野を簡単に増やすことができるという利点がある
  - 図(右)では、Dilated = 1,2,4,8となっている

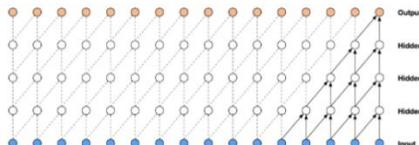


Figure 2: Visualization of a stack of causal convolutional layers.

既存の畳み込み

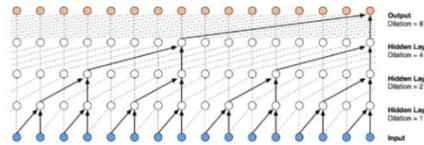


Figure 3: Visualization of a stack of *dilated* causal convolutional layers.

畳み込み (Dilated)

<https://arxiv.org/pdf/1609.03499.pdf>

### ◆実装演習結果キャプチャ

該当する実装の説明とコード提供がないため省略

### ◆確認テストなどの考察結果

該当する確認テストがないため、講義中の演習問題の確認結果を記載する。

## 演習問題 1

### MobileNet のアーキテクチャ

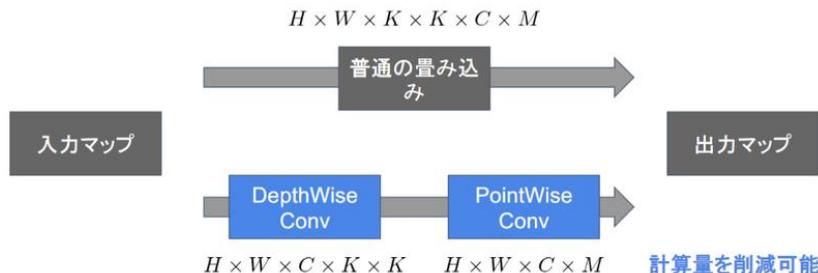
- Depthwise Separable Convolution という手法を用いて計算量を削減している。  
通常の畳み込みが空間方向とチャネル方向の計算を同時に行うのに対して、  
Depthwise Separable Convolution ではそれを Depthwise Convolution と  
Pointwise Convolution と呼ばれる演算によって個別に行う。
- Depthwise Convolution はチャネル毎に空間方向へ畳み込む。すなわち、  
チャネル毎に  $D \times D \times 1$  のサイズのフィルタをそれぞれ用いて計算を行うため、  
その計算量は (い) となる。
- 次に Depthwise Convolution の出力を Pointwise Convolution によって  
チャネル方向に畳み込む。すなわち、出力チャネル毎に  $1 \times 1 \times M$  サイズのフィルタを  
それぞれ用いて計算を行うため、その計算量は (う) となる。

### 解答

- (い)  $H \times W \times C \times K \times K \times 1$  チャネルずつ畳み込みを行う
- (う)  $H \times W \times C \times M$   $\times 1 \times 1$  のフィルタを用いて畳み込みを行う

## MobileNet

Depth-wise separable convolutions



第3世代のMobileNetは、VGG-Netでメモリサイズを300～600Mから14Mに圧縮している。

## 演習問題2

### Wavenet

深層学習を用いて結合確率を学習する際に、効率的に学習が行えるアーキテクチャを提案したことがWavenetの大きな貢献の1つである。

提案された新しいConvolution型アーキテクチャは**(あ)**と呼ばれ、結合確率を効率的に学習できるようになっている。

選択肢

- ・Dilated causal convolution
- ・Depthwise separable convolution
- ・Pointwise convolution
- ・Deconvolution

解答

### Dilated causal convolution

畳み込み層の深い層では間隔を離して畳み込みを行うことでより広い範囲の特徴を学習できる。

Depthwise separable convolutionとPointwise convolutionは、MobileNetで実装されている畳み込み手法である。Deconvolutionは、画像の解像度をアップサンプリングするときに使用される手法である。

## 演習問題3

**(あ)**を用いた際の大きな利点は、単純なConvolution layerと比べて**(い)**ことである。

選択肢

- ・パラメータ数に対する受容野が広い
- ・受容野あたりのパラメータ数が多い
- ・学習時に並列計算が行える

- ・推論時に並列計算が行える

解答

**パラメータ数に対する受容野が広い**

## <Section5: Transformer>

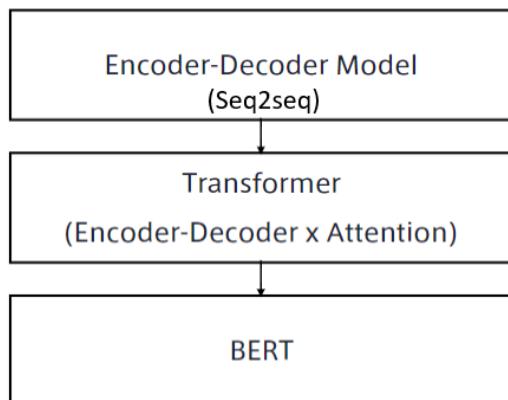
### ◆要点まとめ

本単元は、次の3つから構成されており、最終的にはBERTの理解を深める

- Encoder - Decoder Model (Seq2Seq)
- Transformer (Encoder - Decoder × Attention)
- BERT

BERTまでのロードマップ

BERTを理解するために必要な材料



### 5.1. seq2seq

seq2seqとは、系列(Sequence)を入力として、系列を出力するもので、Encoder-Decoderモデルとも呼ばれる。入力系列がEncode(内部状態に変換)され、内部状態からDecode(系列に変換)する。

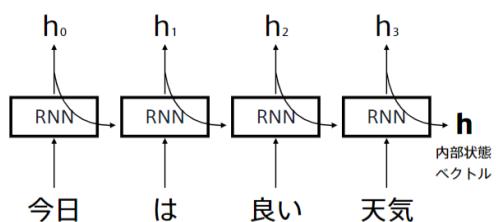
実応用上も、入力・出力共に系列情報であるものは多い。

- 翻訳 (英語→日本語)
- 音声認識 (波形→テキスト)
- チャットボット (テキスト→テキスト)

seq2Seqは、系列(Sequence)情報が扱えるEncoderとDecoderから構成される言語モデルを二つ連結した形になっている。

系列(Sequence)情報である時系列データを扱えるのはRNNである。RNNとは系列データを読み込むために再帰的に動作するニューラルネットワークである。

RNNは再帰処理で時間軸方向に展開でき、系列情報を舐めて**内部状態ベクトル**へ変換できるのがポイントである。

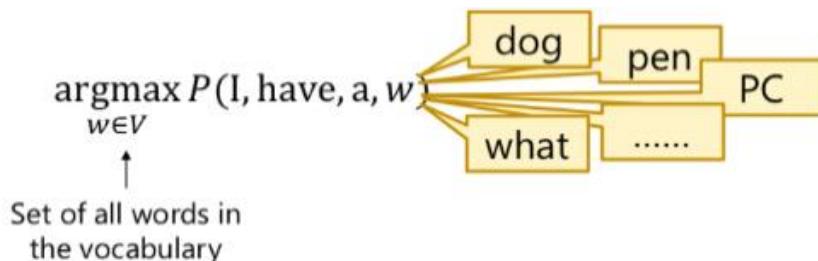


言語モデルは、時刻  $t-1$  までの情報で、**時刻  $t$  の事後確率**を求めることが目標となる。これで文章の同時確率を事後確率として分解して表せる。

- 単語の並びに対して尤度(それがどれだけ起こり得るか)、すなわち、文章として自然かを確率で評価する
- 例)
  - You say goodbye → 0.092 (自然)
  - You say good die → 0.00000032 (不自然)
- 数式的には同時確率を事後確率に分解して表せる

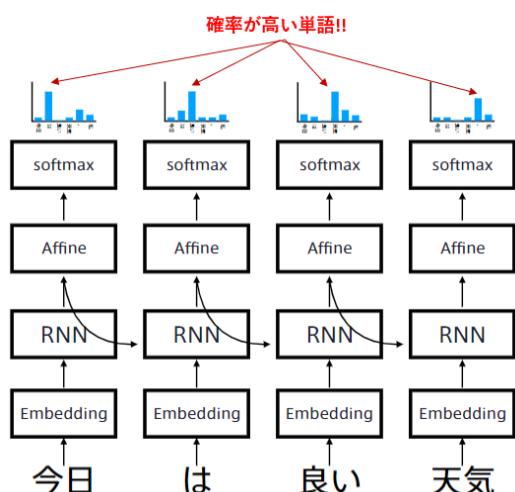
$$P(w_1, \dots, w_m) = \prod_{i=1}^m P(w_i | w_1, \dots, w_{i-1})$$

“I have a”と来た時に、次に続く単語を事後確率から予測できる。

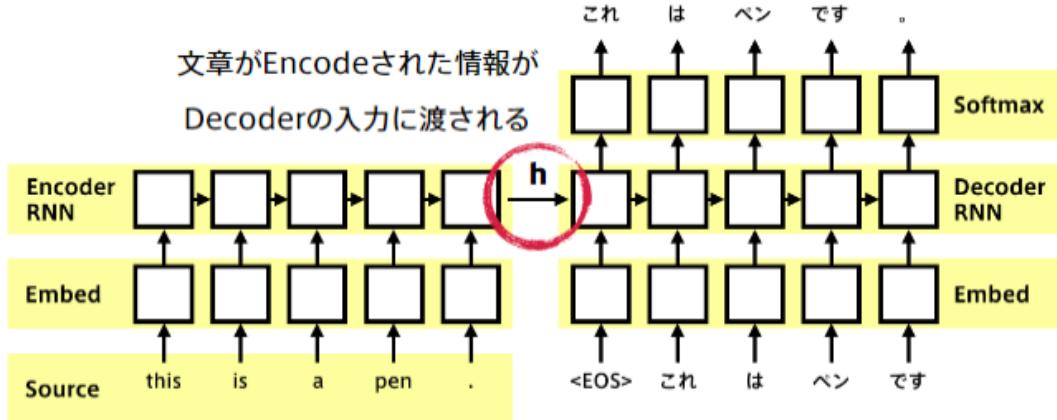


RNN × 言語モデルのように組み合わせることに、各地点で次にどの単語が来れば自然(**事後確率最大**)かを下図のように出力できるになる。

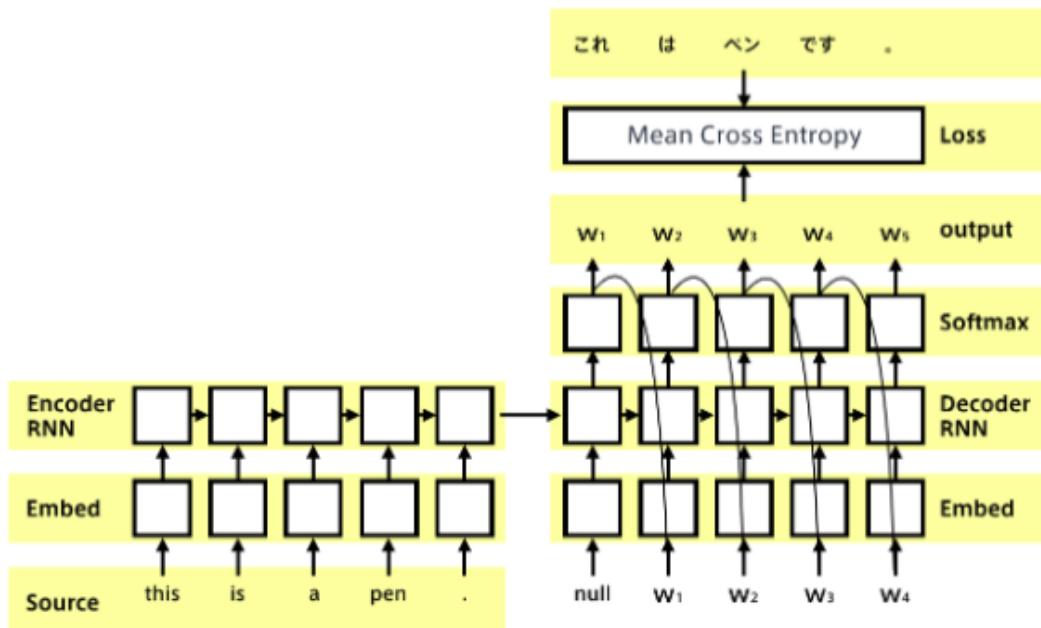
- RNN は系列情報を内部状態に変換することができる。文章の各単語が現れる際の同時確率は、事後確率で分解でき、その事後確率を求めることが RNN の目標にする
- 言語モデルを再現するように **RNN の重みが学習**されていれば、ある時点の次の単語を予測することができる
- **先頭単語(初期値)を与えれば文章を生成**することも可能



seq2seq は、Encoder から Decoder に渡される内部状態ベクトルが鍵となる。Decoder 側の構造は言語モデル RNN とほぼ同じだが、隠れ状態の初期値に Encoder 側の内部状態  $h$  を受け取る。この内部状態ベクトル  $h$  を Decoder へ入力することにより翻訳などの自然言語の分野で飛躍的な発展を遂げた。



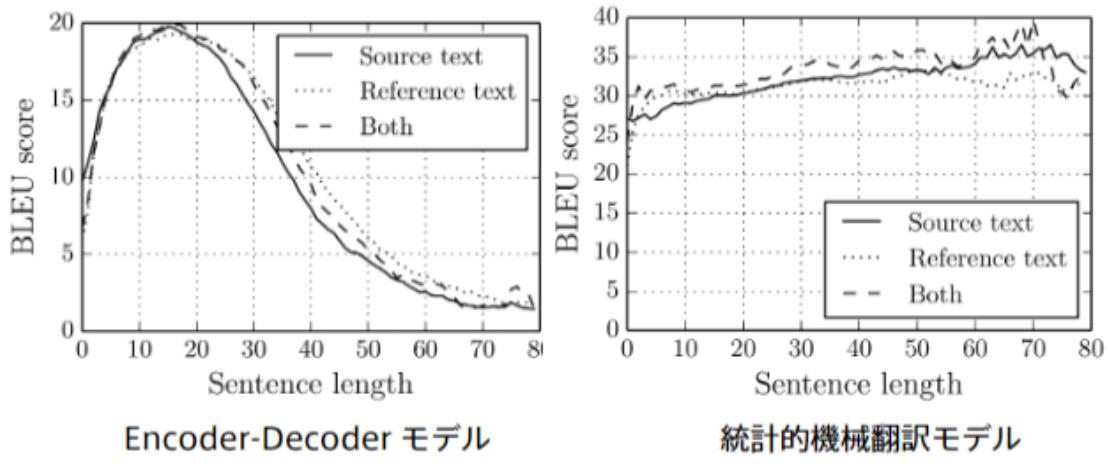
Decoder の output 側に正解ラベルとの誤差を比較すれば、正解ラベルを直接 Decoder へ入力する教師あり学習となり、差分(誤差)から微分が可能となるため、end-to-end で学習することができる。



## 5.2. Transformer

Self-Attention(自己注意機構)に焦点を当てる。

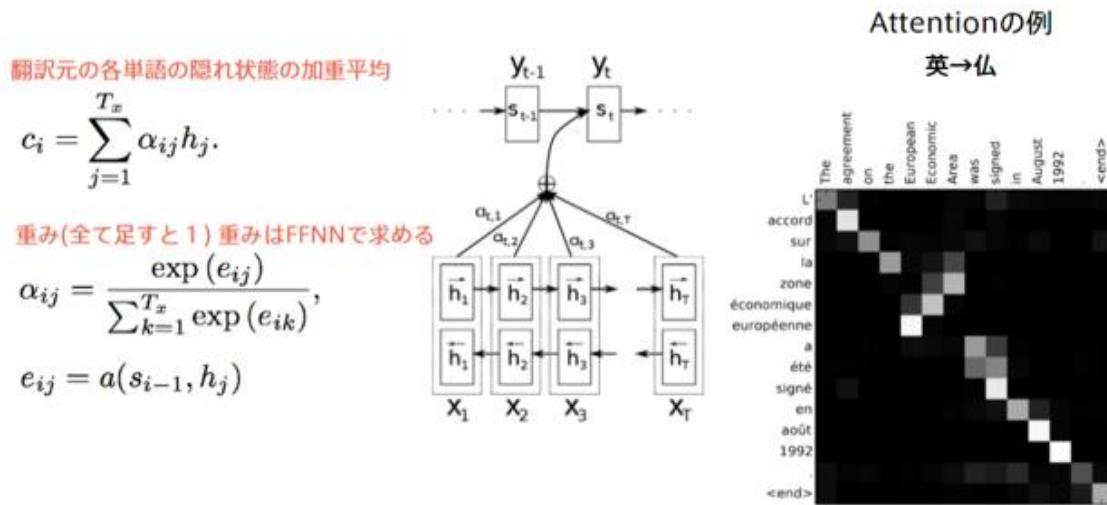
ニューラル機械翻訳(Encoder-Decoderモデル)の問題点としては、文の長さに弱いことである。下図の左のグラフに示すように、文が長くなると BLEU が低下していく。この原因は、翻訳元の文を **ひとつの固定ベクトル** で表現するので、文長が長くなると **表現力が足りなくなり**、下図のグラフのように統計的機械翻訳モデルと比較しても文長と翻訳精度の関係性が悪化し、**BLEU が低下**する。



その対策として、Attention (注意機構) (Bahdanau et al., 2015)がある。Attention は、翻訳先の各単語を選択する際に、翻訳元の文中の各単語の隠れ状態を利用する。Attention (注意機構) は、何に注意すべきか、何に注意しなくてもよいのかの注意の分配を行っている（図 12-1）。

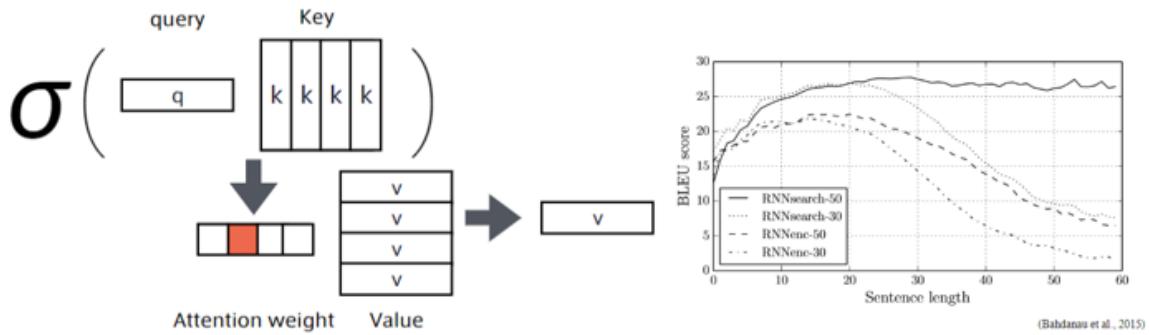
図 12-1 の右側の Attention の事例は英→仏の翻訳であるが、数字などの自明な関係は白色で良く反応しているが、複数の個所で黒色となり対応関係が悪化している。

図 12-1. Attention



Attention は辞書オブジェクトと同様に、**query(検索クエリ)**に一致する **key** を索引し、対応する **value** を取り出す操作であると見なすことができる（図 12-2）。図から文長と翻訳精度について、文長が長くなつても翻訳精度が落ちないことがグラフから確認できる。

図 12-2. Attention の機能イメージと性能

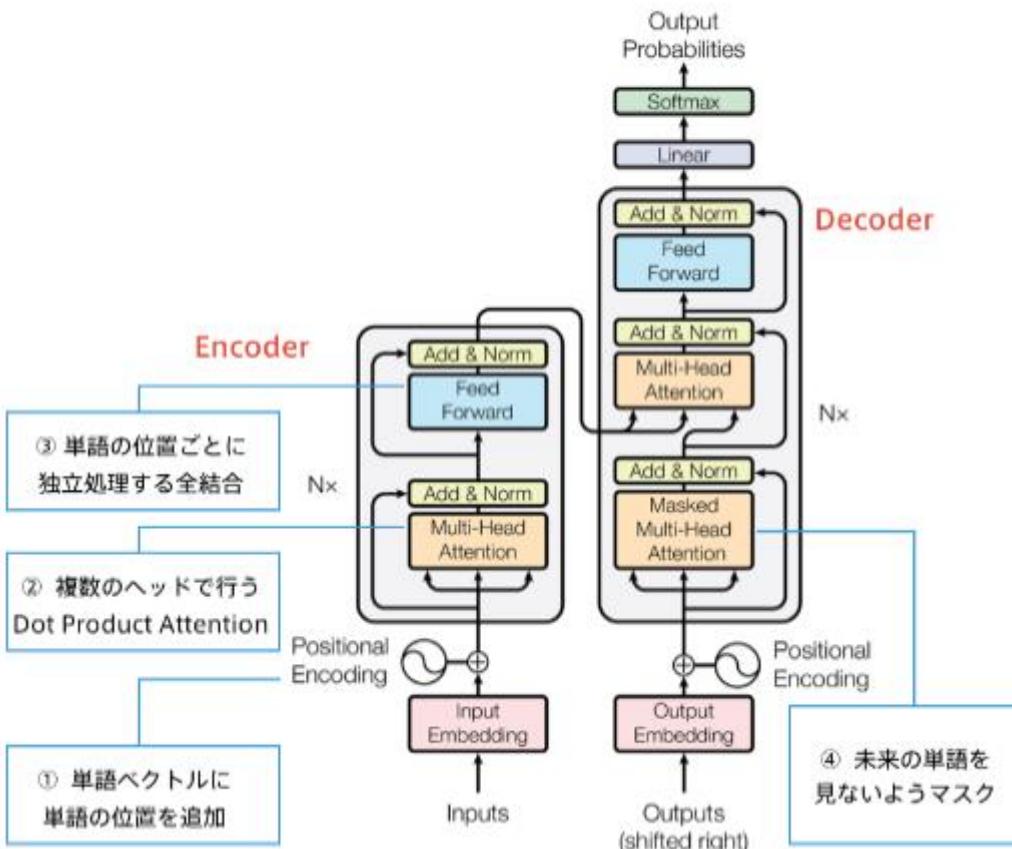


### Transformer (Vaswani et al., 2017) ~Attention is all you need~

2017年6月に登場し、RNNを使用せず、必要なのはAttentionだけである。当時のSOTAをはるかに少ない計算量で実現し、英仏(3600万文)の学習を8GPUにより3.5日で完了した。図12-3にTransformerの構成図を示す。

図12-3. Transformer構成図

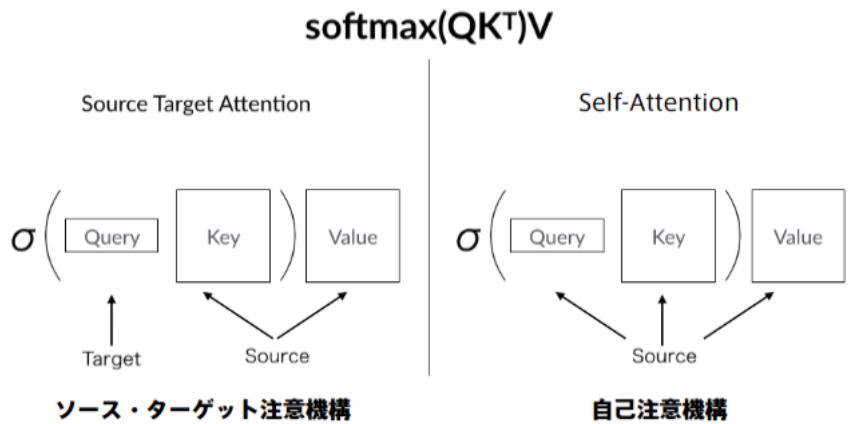
## Transformer主要モジュール



ソース・ターゲット注意機構がもともと異なる用途からスタートしたが同じような発想で狙うべきターゲットをQueryに変えて自己注意機構として発展した。アテンションベクトル(query, key, value)だけ注意を払う。

## Attention

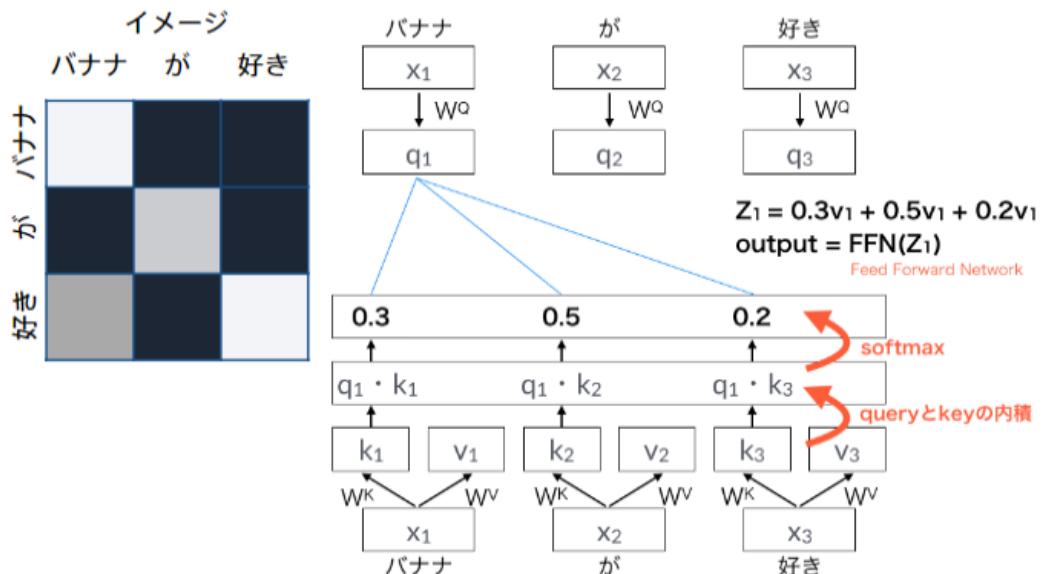
注意機構には二種類ある



**Attention (注意機構)** では、文脈を考慮して各単語をエンコード可能で、入力だけで学習的に注意箇所を決めていく。注意箇所を判定する内部状態は内積により処理されており、CNN のウィンドウサイズのイメージとなる。

### Self-Attentionが肝

入力を全て同じにして学習的に注意箇所を決めていく

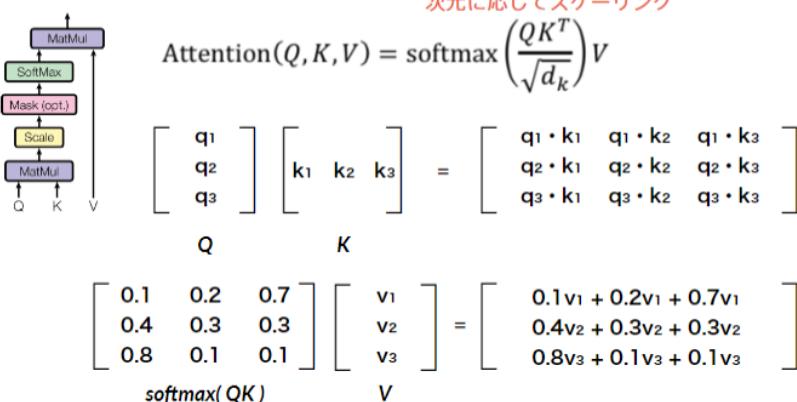


Encoder では、文中の単語の位置情報を保持して Encode しており、各単語の qkv を線形変換して行列からスカラーに変換して Attention を計算している。また、アンノウンやパッドなど評価したくないところをマスクする実装も行って精度を向上している。

## Scaled dot product attention

全単語に関するAttentionをまとめて計算する

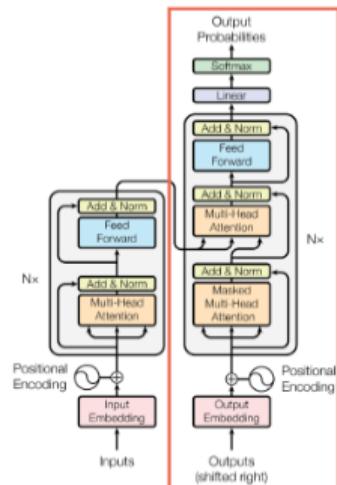
Scaled Dot-Product Attention



Decoder の基本的構造は Encoder と同じ構成要素を使っており類似している。

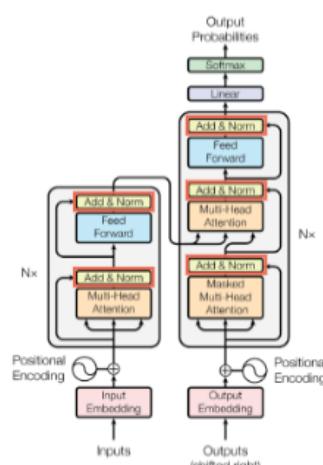
## Decoder

- Encoderと同じく6層
  - 各層で二種類の注意機構
  - 注意機構の仕組みはEncoderとほぼ同じ
- 自己注意機構
  - 生成単語列の情報を収集
    - 直下の層の出力へのアテンション
    - 未来の情報を見ないようにマスク
- Encoder-Decoder attention
  - 入力文の情報を収集
    - Encoderの出力へのアテンション



Add & Norm は seq2seq で学んだ機能を提供しており、この機能を追加すると性能が向上するため、お作法的なテクニックで用いられている。

- Add (Residual Connection)
  - 入出力の差分を学習させる
  - 実装上は出力に入力をそのまま加算するだけ
  - 効果：学習・テストエラーの低減
- Norm (Layer Normalization)
  - 各層においてバイアスを除く活性化関数への入力を平均0、分散1に正則化
  - 効果：学習の高速化



Attention は、単語の位置情報を保持しないため、Position Encoding により位置情報を Embedding 情報に付加することで **Attention (注意機構)** 内へインプットしている。

## Position Encoding

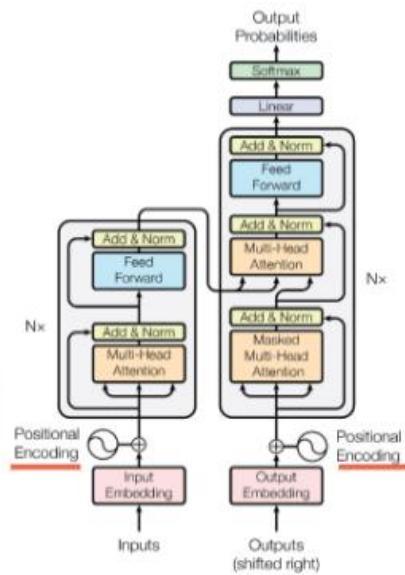
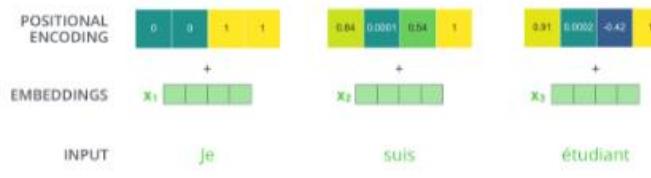
### RNNを用いないので単語列の語順情報を追加する必要がある

- 単語の位置情報をエンコード

$$PE_{(pos,2i)} = \sin\left(\frac{pos}{10000^{2i/512}}\right)$$

$$PE_{(pos,2i+1)} = \cos\left(\frac{pos}{10000^{2i/512}}\right)$$

- posの(ソフトな)2進数表現
- 動作イメージ↓



### ◆実装演習結果キャプチャ

#### 実装演習 1. seq2seq

< lecture\_chap1\_exercise\_public.ipynb >

データセットを読み込んだ結果

```
[8] ! ls data
dev.en dev.ja test.en test.ja train.en train.ja
```

抽出した単語

```
[18] vocab_size_X = len(vocab_X.id2word)
vocab_size_Y = len(vocab_Y.id2word)
print('入力言語の語彙数：', vocab_size_X)
print('出力言語の語彙数：', vocab_size_Y)
```

入力言語の語彙数： 3725  
出力言語の語彙数： 4405

IDへの変換

```
[19] print('train data', train_X[0])
print('valid data', valid_X[0])

train data [132, 321, 28, 290, 367, 12, 2]
valid data [8, 98, 3532, 36, 236, 13, 284, 4, 2]
```

学習結果

学習回数が増えるごとに、訓練データとテストデータに対する誤差が減り、BLEU が上がっていることが確認できた。

```
[34] 5分
    # validationデータでBLEUが改善した場合にはモデルを保存
    if valid_bleu > best_valid_bleu:
        ckpt = model.state_dict()
        torch.save(ckpt, ckpt_path)
        best_valid_bleu = valid_bleu

    print('Epoch {}: train_loss: {:.5f} train_bleu: {:.2f} valid_loss: {:.5f} valid_bleu: {:.2f}'.format(
        epoch, train_loss, train_bleu, valid_loss, valid_bleu))

    print('-'*80)

Epoch 1: train_loss: 52.67 train_bleu: 3.03 valid_loss: 48.77 valid_bleu: 4.84
-----
Epoch 2: train_loss: 44.39 train_bleu: 7.67 valid_loss: 45.10 valid_bleu: 9.54
-----
Epoch 3: train_loss: 40.22 train_bleu: 11.15 valid_loss: 42.38 valid_bleu: 10.56
-----
Epoch 4: train_loss: 37.24 train_bleu: 14.20 valid_loss: 41.05 valid_bleu: 12.21
-----
Epoch 5: train_loss: 35.17 train_bleu: 16.28 valid_loss: 40.39 valid_bleu: 15.31
-----
Epoch 6: train_loss: 33.01 train_bleu: 19.13 valid_loss: 40.14 valid_bleu: 15.10
-----
Epoch 7: train_loss: 31.76 train_bleu: 20.63 valid_loss: 40.06 valid_bleu: 16.49
-----
Epoch 8: train_loss: 30.37 train_bleu: 22.51 valid_loss: 39.77 valid_bleu: 16.04
-----
Epoch 9: train_loss: 28.82 train_bleu: 24.97 valid_loss: 40.56 valid_bleu: 17.76
-----
Epoch 10: train_loss: 27.89 train_bleu: 26.48 valid_loss: 40.94 valid_bleu: 18.23
```

## 学習モデルの確認結果

入力を入れ替えて結果を確認した。

```
[40] 0秒
    output = model(batch_X, lengths_X, max_length=20)
    output = output.max(dim=-1)[1].view(-1).data.cpu().tolist()
    output_sentence = ' '.join(ids_to_sentence(vocab_Y, trim_eos(output)))
    output_sentence_without_trim = ' '.join(ids_to_sentence(vocab_Y, output))
    print('out: {}'.format(output_sentence))
    print('without trim: {}'.format(output_sentence_without_trim))

src: show your own business .
tgt: 自分 の 事を しろ 。
out: 自分 の 仕事 は よけい しなさい 。
without trim: 自分 の 仕事 は よけい しなさい 。 </S> </S>
```

```
# 生成
batch_X, batch_Y, lengths_X = next(test_dataloader)
sentence_X = ' '.join(ids_to_sentence(vocab_X, batch_X.data.cpu().numpy()[:-1, 0]))
sentence_Y = ' '.join(ids_to_sentence(vocab_Y, batch_Y.data.cpu().numpy()[:-1, 0]))
print('src: {}'.format(sentence_X))
print('tgt: {}'.format(sentence_Y))

output = model(batch_X, lengths_X, max_length=20)
output = output.max(dim=-1)[1].view(-1).data.cpu().tolist()
output_sentence = ' '.join(ids_to_sentence(vocab_Y, trim_eos(output)))
output_sentence_without_trim = ' '.join(ids_to_sentence(vocab_Y, output))
print('out: {}'.format(output_sentence))
print('without trim: {}'.format(output_sentence_without_trim))

src: i can 't swim at all .
tgt: 私 は 少し も 泳 げ な い 。
out: 私 は 泳 げ な い い 。
without trim: 私 は 泳 げ な い い 。 </S> 。 </S> </S>
```

## 学習モデルの BLEU スコア

```
✓ 5 秒
# BLEUの計算
test_dataloader = DataLoader(test_X, test_Y, batch_size=1, shuffle=False)
refs_list = []
hyp_list = []

for batch in test_dataloader:
    batch_X, batch_Y, lengths_X = batch
    pred_Y = model(batch_X, lengths_X, max_length=20)
    pred = pred_Y.max(dim=-1)[1].view(-1).data.cpu().tolist()
    refs = batch_Y.view(-1).data.cpu().tolist()
    refs_list.append(refs)
    hyp_list.append(pred)
bleu = calc_bleu(refs_list, hyp_list)
print(bleu)

17.799568608262103
```

## 実装演習 2. Transformer

<lecture\_chap2\_exercise\_public.ipynb>

データセットを読み込んだ結果

```
✓ 0 秒
[8] # データセットの中身を確認
print('train_X:', train_X[:5])
print('train_Y:', train_Y[:5])

train_X: [['wheres', 'shall', 'we', 'eat', 'tonight', '?'], ['i', 'made', 'a', 'big', 'mistake', 'in', 'choosing', 'my', 'wife', '.'],
train_Y: [['今夜', 'は', 'どこ', 'で', '食事', 'を', 'し', 'よ', 'う', 'か', '。'], ['僕', 'は', '妻', 'を', '選', 'ぶ', 'の', 'に']]
```

ID への変換

```
✓ 0 秒
[11] train_X = [sentence_to_ids(vocab_X, sentence) for sentence in train_X]
      train_Y = [sentence_to_ids(vocab_Y, sentence) for sentence in train_Y]
      valid_X = [sentence_to_ids(vocab_X, sentence) for sentence in valid_X]
      valid_Y = [sentence_to_ids(vocab_Y, sentence) for sentence in valid_Y]
```

学習結果

学習回数が増えるごとに、訓練データとテストデータに対する誤差が減り、BLEU が上がっていることが確認できた。精度は、seq2seq を上回っている。

```

[35]     elapsed_time = (time.time()-start) / 60
      print('Epoch {} [ {:.1f}min]: train_loss: {:.5f} train_bleu: {:.2f} valid_loss: {:.5f} valid_bleu: {:.2f}'.format(
          epoch, elapsed_time, train_loss, train_bleu, valid_loss, valid_bleu))
      print('*'*80)

Epoch 1 [0.7min]: train_loss: 77.51 train_bleu: 4.77 valid_loss: 41.49 valid_bleu: 11.09
-----
Epoch 2 [0.7min]: train_loss: 39.29 train_bleu: 12.29 valid_loss: 32.12 valid_bleu: 17.82
-----
Epoch 3 [0.7min]: train_loss: 31.84 train_bleu: 18.05 valid_loss: 27.96 valid_bleu: 22.16
-----
Epoch 4 [0.7min]: train_loss: 28.12 train_bleu: 21.90 valid_loss: 25.67 valid_bleu: 24.86
-----
Epoch 5 [0.7min]: train_loss: 25.69 train_bleu: 24.62 valid_loss: 24.21 valid_bleu: 27.26
-----
Epoch 6 [0.7min]: train_loss: 23.88 train_bleu: 26.86 valid_loss: 22.98 valid_bleu: 28.98
-----
Epoch 7 [0.7min]: train_loss: 22.46 train_bleu: 28.57 valid_loss: 22.13 valid_bleu: 30.35
-----
Epoch 8 [0.7min]: train_loss: 21.26 train_bleu: 30.20 valid_loss: 21.55 valid_bleu: 31.29
-----
Epoch 9 [0.7min]: train_loss: 20.24 train_bleu: 31.56 valid_loss: 20.83 valid_bleu: 32.17
-----
Epoch 10 [0.7min]: train_loss: 19.36 train_bleu: 32.88 valid_loss: 20.36 valid_bleu: 33.42
-----
Epoch 11 [0.7min]: train_loss: 18.58 train_bleu: 34.02 valid_loss: 19.92 valid_bleu: 33.70
-----
Epoch 12 [0.7min]: train_loss: 17.83 train_bleu: 35.24 valid_loss: 19.80 valid_bleu: 34.11
-----
Epoch 13 [0.7min]: train_loss: 17.24 train_bleu: 36.05 valid_loss: 19.46 valid_bleu: 34.70
-----
Epoch 14 [0.7min]: train_loss: 16.66 train_bleu: 37.03 valid_loss: 19.32 valid_bleu: 34.87
-----
Epoch 15 [0.7min]: train_loss: 16.12 train_bleu: 37.94 valid_loss: 19.05 valid_bleu: 35.52
-----
```

## 学習モデルの確認結果

```

1 src, tgt = next(test_dataloader)
2
3 src_ids = src[0][0].cpu().numpy()
4 tgt_ids = tgt[0][0].cpu().numpy()
5
6 print('src: {}'.format(' '.join(ids_to_sentence(vocab_X, src_ids[1:-1]))))
7 print('tgt: {}'.format(' '.join(ids_to_sentence(vocab_Y, tgt_ids[1:-1]))))
8
9 preds, enc_slf_attns, dec_slf_attns, dec_enc_attns = test(model, src)
10 pred_ids = preds[0].data.cpu().numpy().tolist()
11 print('out: {}'.format(' '.join(ids_to_sentence(vocab_Y, trim_eos(pred_ids)))))

12 src: i can 't swim at all .
13 tgt: 私は少しも泳げない。
14 out: 私はまったく泳げない。
```

## 学習モデルの BLEU スコア

seq2seq より高いスコアが出ている

```

1 # BLEUの評価
2 test_dataloader = DataLoader(
3     test_X, test_Y, 128,
4     shuffle=False
5 )
6 refs_list = []
7 hyp_list = []

8 for batch in test_dataloader:
9     batch_X, batch_Y = batch
10    preds, *_ = test(model, batch_X)
11    preds = preds.data.cpu().numpy().tolist()
12    refs = batch_Y[0].data.cpu().numpy()[:, 1:].tolist()
13    refs_list += refs
14    hyp_list += preds
15 bleu = calc_bleu(refs_list, hyp_list)
16 print(bleu)

24.65371113758429
```

◆確認テストなどの考察結果

確認テストの出題なしのため省略

## <Section6: 物体検知・セグメンテーション>

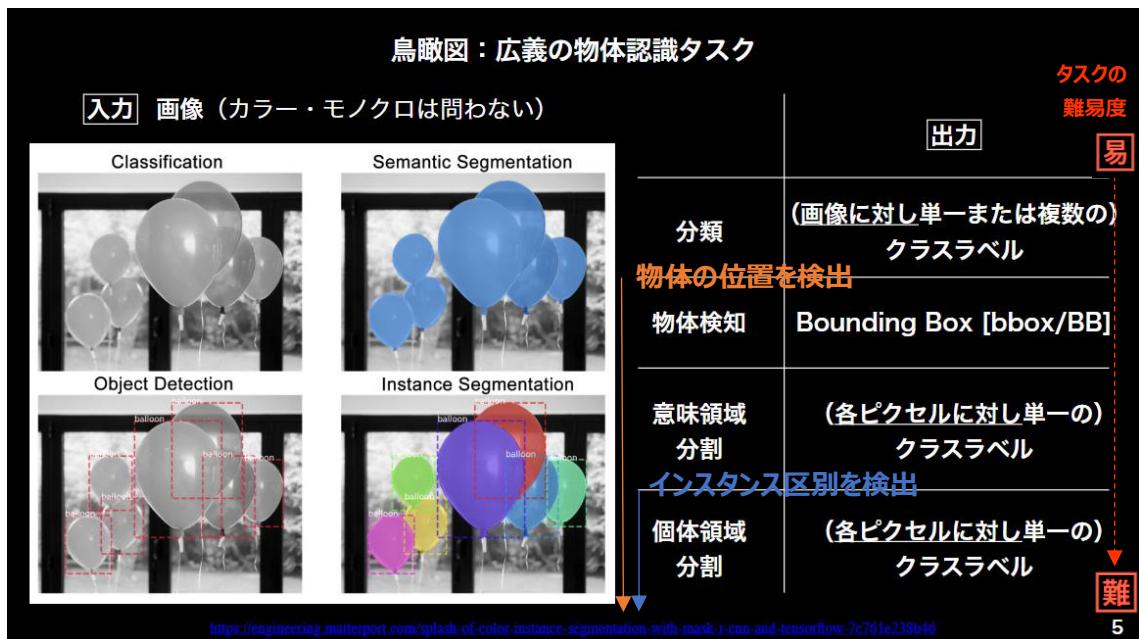
### ◆要点まとめ

#### 1. Introduction

広義の物体認識タスクの定義を図 13 に示す。物体認識は、分類 (Classification)、物体検知 (ObjectDetection)、意味領域分割 (Semantic Segmentation)、個体領域分割 (Instance Segmentation) の 4 つに分類できる。

本講義では「**物体検知**」、「**意味領域分類**」について詳しく解説する。

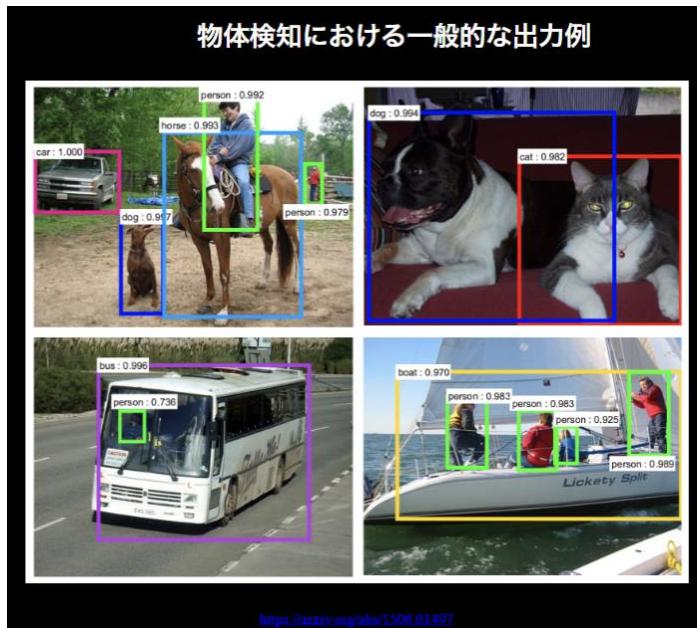
図 13 : 広義の物体認識タスクの定義



#### 1-1. 物体検知

物体検知では、検出対象の位置に興味があり、そのラベルと**位置をバンディングボックス（検知枠）**で捉えて、**検出精度を confidence（信頼度確率）**として表す。物体検知の検出例を図 13-2 に示す。

図 13-2：物体検知の検出例



## 1-2. 代表的データセット

表 13-3 は、いずれも物体検出コンペティションで用いられたデータセットで、それらは目的に応じて用意されており、使い分けが必要である。

データセットを選択する際の指標としては、「クラス」、「サンプル数(Train+Val)」及び「Box/画像」がある。「Box/画像」は 1 枚の画像に物体が映っている物体数を表しており、少ないと学習時の効率が悪く、多すぎると識別が大変になる。

このような指標や、クラス vs Box/画像のポジショニングマップを参考にして、目的に応じた学習のためのデータセットを選択することになる。

表 13-3：代表データセット

代表的データセット				ポジショニングマップ Box/画像
	クラス	Train+Val	Box/画像	
VOC12	20	11,540	2.4	Instance Annotation
ILSVRC17	200	476,668	1.1	Instance Annotation
MS COCO18	80	123,287	7.3	Instance Annotation
OICOD18	500	1,743,042	7.0	Instance Annotation

小
Box/画像
大

アイコン的な映り  
日常感とはかけ離れやすい
部分的な重なり等も見られる  
日常生活のコンテキストに近い

◆ PASCAL VOC Object Detection Challenge ◆

- VOC = Visual Object Classes
- 主要貢献者が2012年に亡くなったことに伴いコンペも終了

◆ ILSVRC Object Detection Challenge ◆

- コンペは2017年に終了（後継：Open Images Challenge）
- ILSVRC = ImageNet Scale Visual Recognition Challenge
- ImageNet (21,841 クラス/1400万枚以上) のサブセット

◆ MS COCO Object Detection Challenge ◆

- COCO = Common Object in Context
- 物体位置推定に対する新たな評価指標を提案（後述）

◆ Open Images Challenge Object Detection ◆

- ILSVRCやMS COCOとは異なるannotation process
- Open Images V4 (6000 クラス以上/900万枚以上) のサブセット

MS COCO18  
ILSVRC18  
VOC12  
OICOD18  
MS COCO18  
ILSVRC17  
VOC12  
OICOD18  
ILSVRC17  
Laptop  
Notebook

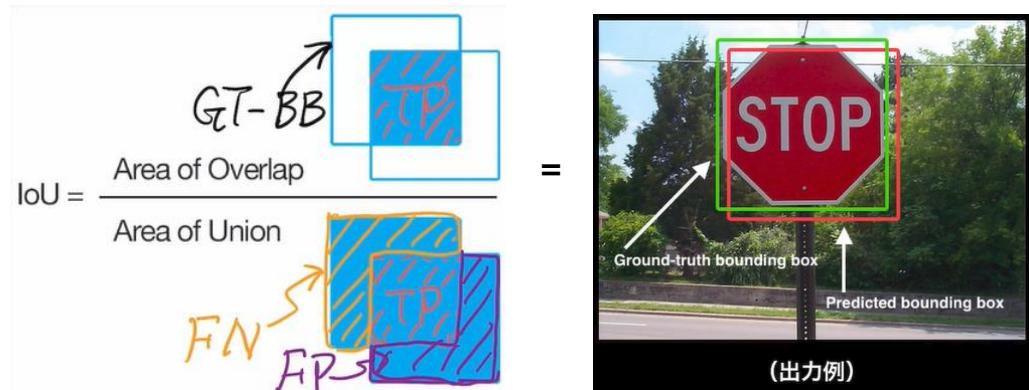
・目的に応じたBox/画像の選択を！  
・クラスが大きいことは嬉しいのか？

ILSVRC17 以外のデータセットには、Instance Annotation、個々の物体にラベルが与えられている

### 1-3. 評価指標

検出精度の評価では、基本は確率統計で用いられる Confusion Matrix（混同行列）を使用する。評価では「クラス分類」、「物体検出」の confidence 値と IOU 値に対し、各閾値変化が「Ground-Truth」に対してどのような値になるかを確認しながら進める。

IOU (Intersection Over Union) は、物体検出においてはクラスラベルだけでなく、物体位置の予測精度も評価したいため、下式のとおり「混同行列」の要素を用いて評価する。Ground Truth Bounding Box (GT-BB) が真の bbox としたとき、GT-BB と予測した bbox が重なっている部分が TP、GT-BB に対して予測した bbox が重なっていない部分が FN、予測した bbox に対して GT-BB が重なっていない部分が FP となる。



※IOU が TP/GT-BB や、TP/Predicted-BB でないことに注意

1 枚の画像から検出した bbox に対する Precision と Recall は、検出した bbox の confidence と IOU を、confidence 閾値と IOU 閾値により TP か FP かを判定して行う。以下に検出した bbox の confidence 値と IOU 値から Precision と Recall を算出する例を示す。以下の例では Precision は 0.5、Recall は 1.0 である。

また、複数画像から検出した同じクラスに対する Precision と Recall は、各 bbox の confidence と IOU を、confidence 閾値と IOU 閾値により TP か FP かを判定して行う。以下に入クラス bbox の confidence 値と IOU 値から Precision と Recall を算出する例を示す。以下の例では Precision は 0.5、Recall は 0.75 である。GT-BB4 に対して bbox が未検出であるため FN となることに注意が必要である。

また、confidence 閾値に対する Average Precision (AP) もよく使う指標である。AP は以下の式で表す。AP は PR 曲線の下側面積となる。

$$AP = \int_0^1 P(R)dR$$

個々のクラスの AP から、全クラスの mean Average Precision (mAP) が求まる。mAP は以下の式で表す。ここで C はクラス数である。

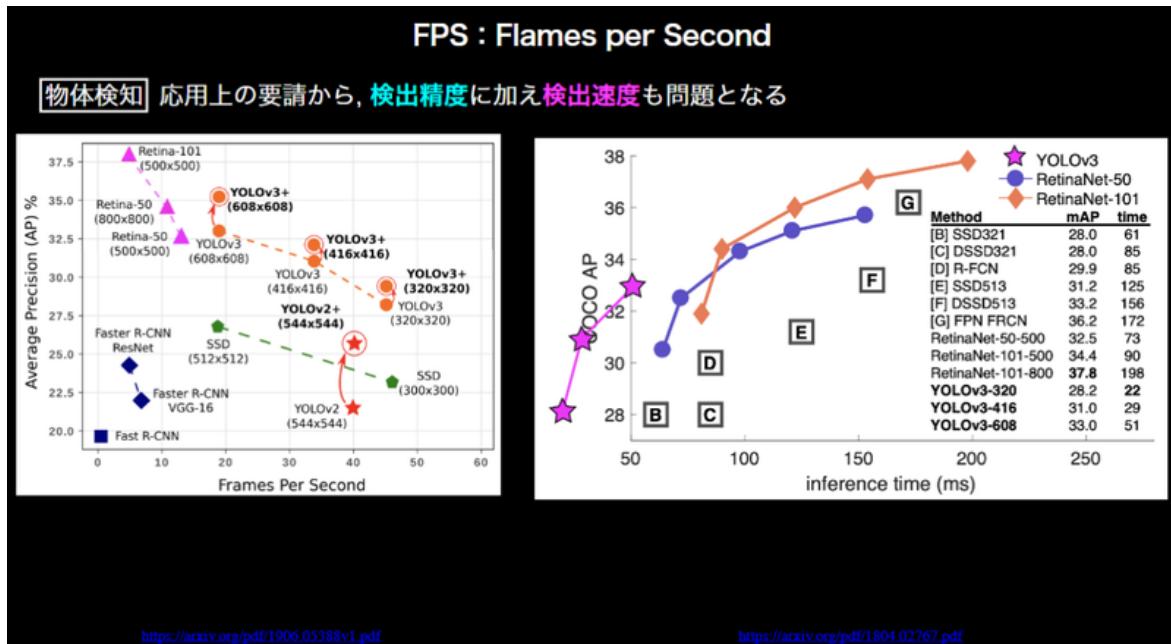
$$mAP = \frac{1}{C} \sum_{i=1}^C AP_i$$

MS COCO では、検知位置精度を評価する以下の mAPcoco が導入された。 $mAP_{0.5}$  とは IOU 閾値

0.5 の時の mAP であり、閾値を 0.05 刻みで加算した mAP の算術平均から求める。

$$mAP_{coco} = \frac{mAP_{0.5} + mAP_{0.55} + \cdots + mAP_{0.95}}{10}$$

また、別の指標として、応用上の要請から、検出精度に加え検出速度（スピード）も問題となるため、「FPS (Flames per Second)」を用いて評価する。論文などで以下のような指標として使用されている。FPS だけでなく、inference time が指標として使用される場合もある。



## 2. 物体検知の大枠

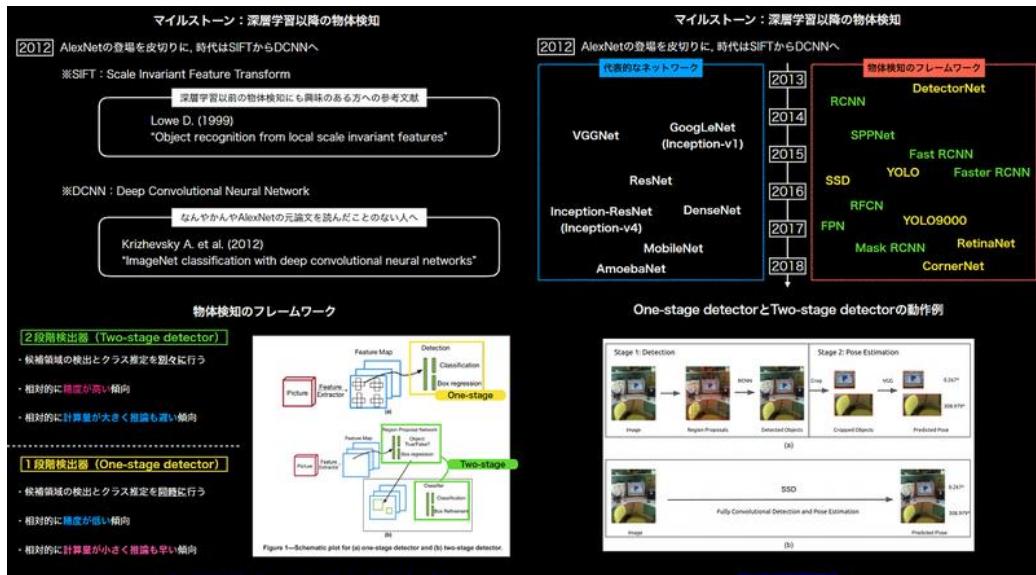
2012 年の AlexNet の登場を皮切りに、時代は SIFT (Scale Invariant Feature Transfor) から DCNN (Deep Convolutional Neural Network) へ変遷している。

物体検知のフレームワークとしては、1 段階検出器と 2 段階検出器に大きく分類することができる。図 13-4 の右上の黄色文字のフレームワークが 1 段階検出器、緑色文字のフレームワークが 2 段階検出器である。構成を図 13-4 の左下に、処理イメージを図 13-4 の右下に示す。

1 段階検出器と 2 段階検出器のそれぞれの特徴は以下となる。

- **1 段階検出器** (One-stage detector)
  - ・候補領域の検出とクラス推定を**同時**に行う
  - ・2 段階検出に比べ相対的に**精度が低い傾向**
  - ・2 段階検出に比べ相対的に**計算量が小さく推論も早い傾向**
- **2 段階検出器** (Two-stage detector)
  - ・候補領域の検出とクラス推定を**別々**に行う
  - ・1 段階検出に比べ相対的に**精度が高い傾向**
  - ・1 段階検出に比べ相対的に**計算量が大きく推論も遅い傾向**

図 13-4：物体検知フレームワーク



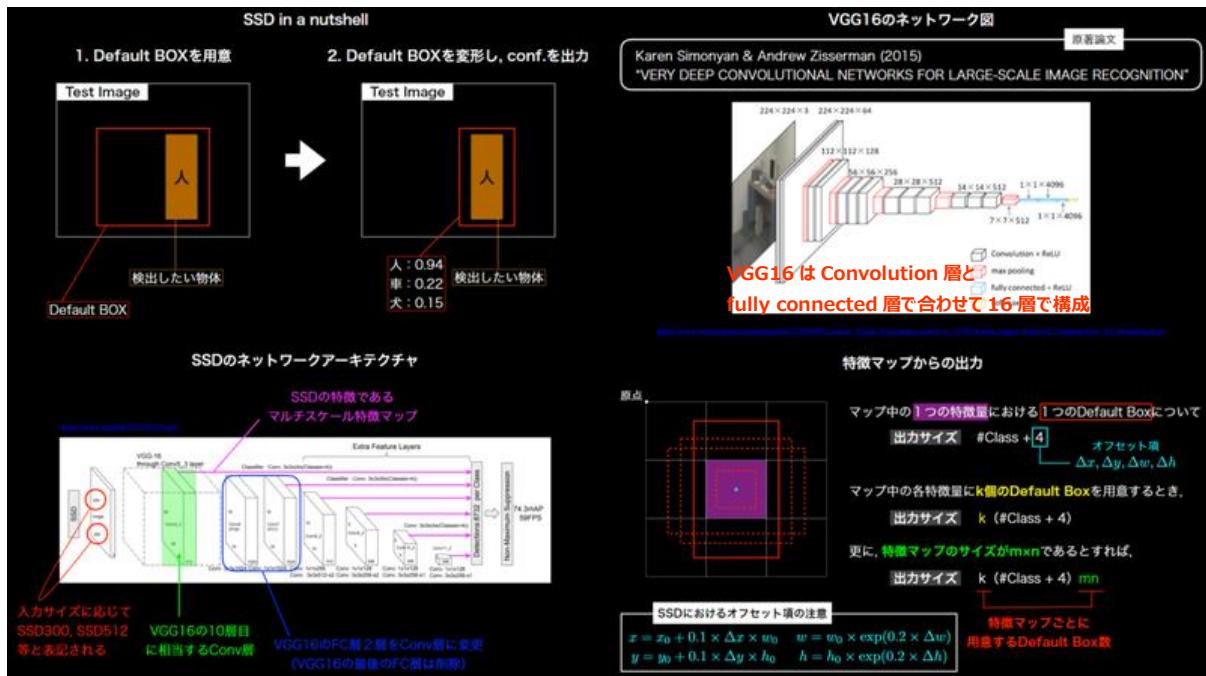
## 2-1. SSD (Single Shot Detector)

SSD では、"Single Shot" のとおり、1 度の CNN 演算で物体の「領域候補検出」と「クラス分類」の両方を行って、物体検出処理の高速化を可能にしている。

- ・検出は Default BOX を用意し、Default BOX を学習により変形し、confidence (conf.) を出力
- ・SSD は VGG16 (図 13-5 右上) をベースとして、ネットワークアーキテクチャは、
  - VGG16 の Fully Connected 層 2 層を Conv 層に変更 (最後の FC 層は削除)
  - SSD の特徴であるマルチスケール特徴マップを Detection 層に出力 (38x38, 19x19, 10x10, 5x5, 3x3, 1x1)

となっている

図 13-5：SSD 構成と特徴



特徴マップからの出力サイズは、特徴マップごとに用意するDefault Box数に関係して次式で求められる。

$$\text{出力サイズ: } k \times (\#Class + 4) \times mn$$

#Class : クラス数  $\times +4$  はオフセット項

k : マップ中の各特徴量に対して k 個の Default Box を用意

mn : 特徴マップのサイズが  $m \times n$

で求められる。ここで特徴マップごとに用意する Default Box 数は  $k \times mn$  となる。

図 13-6 の左上に示すように、SSD の特徴マップサイズは、8732 個の Default Box と、21 個のクラス数 + 4 を掛け合わせたサイズとなる。クラス数は VOC データセットのクラス数 20 に背景クラスが 1 足されていることに注意が必要である。

多数の Default Box を用意したことによる問題への対処として

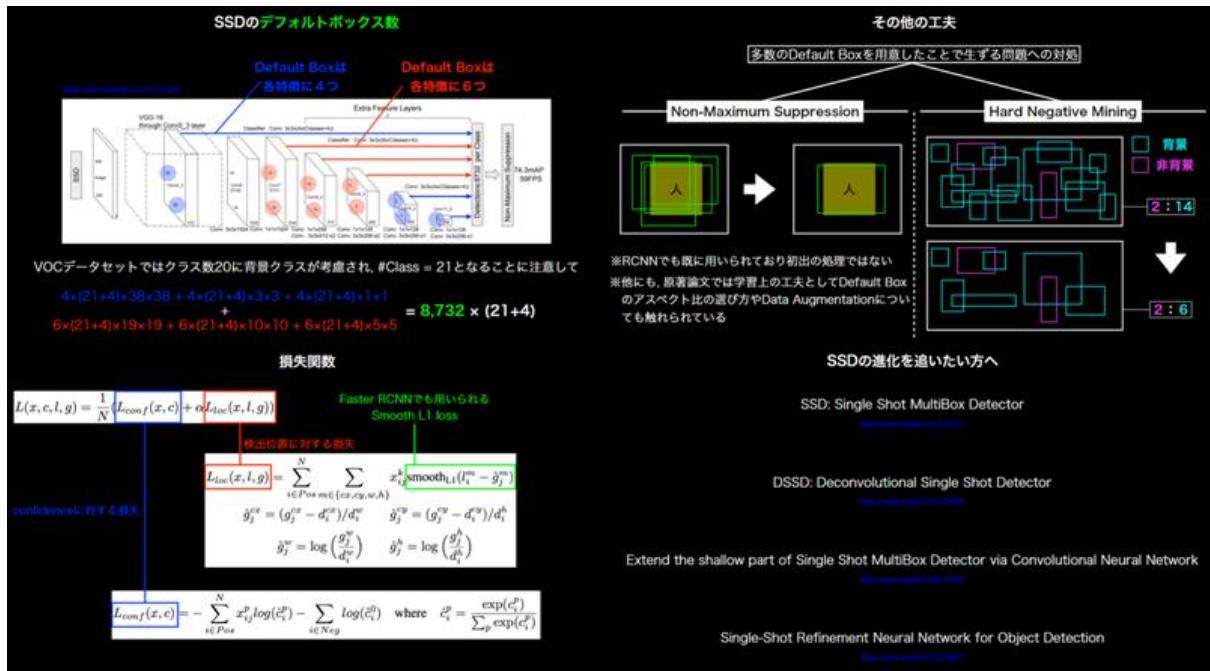
- **Non-Maximum Suppression** : IOU と conf.により最も適切な bbox を選択する

- **Hard Negative Mining** : 背景クラスと判断される Negative の bbox が多数検出されるため、

Positive の bbox に対して Negative の bbox が最大でも 3 倍までになるように削減するにより、bbox 数の削減を行っている（図 13-6 右上）。

学習には、ネットワークから出力された値が教師ラベルとどれほどズレているかという指標として損失関数を用いる。損失は **クラス分類予測に起因するもの**と、**バウンディングボックスの位置予測に起因するものの和**となり、前者には **交差エントロピー誤差関数**を用い、後者には **Smooth L1 損失関数**を用いる（図 13-6 左下）。

図 13-6 : SSD 特徴と損失関数



### 3. Semantic Segmentation

#### 3-1. Semantic Segmentation の概略

Semantic Segmentation は、各 Pixel に対し、单一のクラスラベルを与える処理であるため、Convolution + Pooling 処理で解像度を下げた画像（図 13-7 の右上）に対してクラスラベルを与えることはできない。

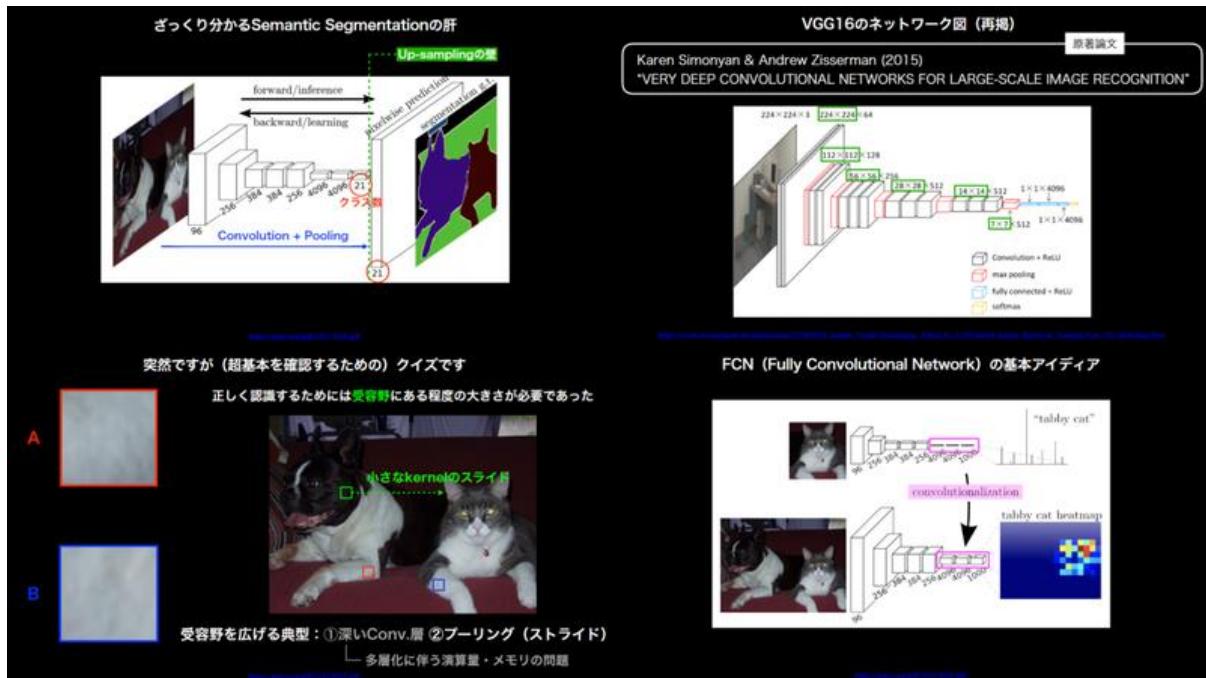
そのため、元の解像度まで戻す必要があるが、Up-sampling なんて面倒なことが必要になるなら、そもそも Pooling しなければ良いのではと考えられる。しかし、物体を正しく認識するためには受容野にある程度の大きさが必要になる。受容野を広げる方法としては

- ①深いConv.層
- ②プーリング（ストライド）

があり、ネットワークの多層化に伴う演算量やメモリ量の削減にはプーリングが必須である。

Up-sampling の基本アイディアとして FCN (Fully Convolutional Network) があり、全結合層を持たず、ネットワークが畳み込み層のみで構成されており、クラス分類の結果がヒートマップとして出力されるようになる（図 13-7 の右下）。

図 13-7 : Semantic Segmentation 概略



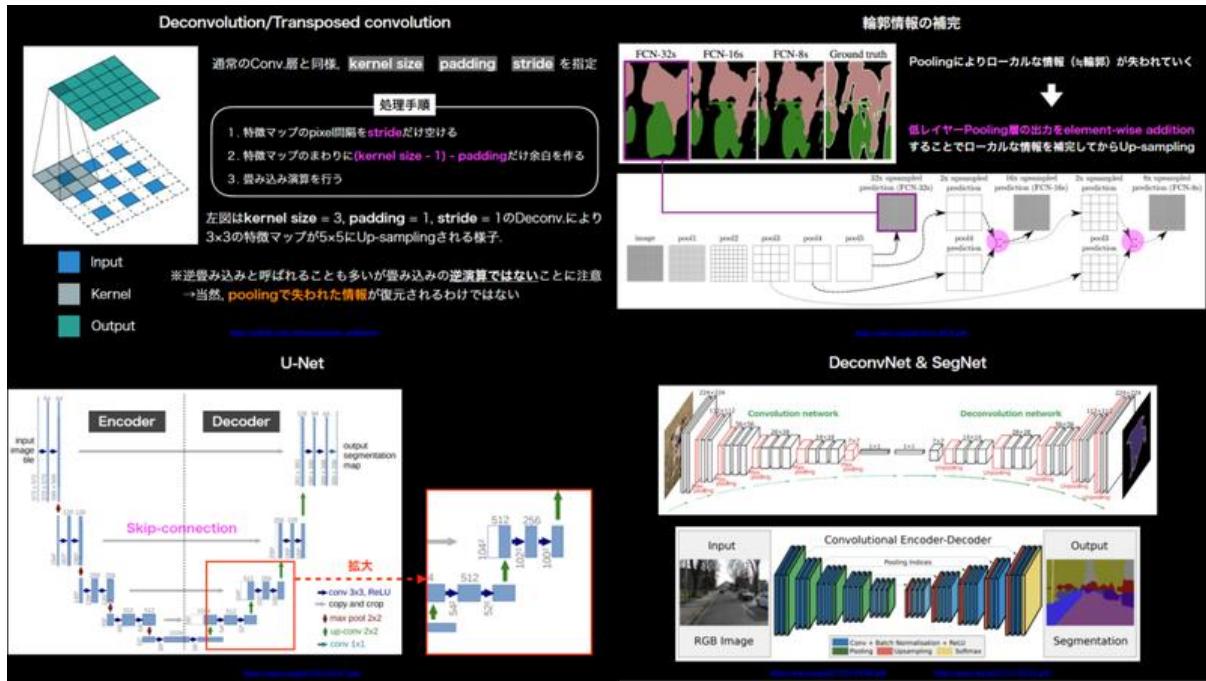
ヒートマップからのアップサンプリングには逆畳み込み（Deconvolution）という処理を施す（図 13-8 左上）。※逆演算でない。

処理手順としては以下となる。

- 1.ヒートマップの pixel 間隔を stride だけ空ける
- 2.ヒートマップのまわりに  $(\text{kernel size} - 1) - \text{padding}$  だけ余白を作る
- 3.畳み込み演算を行う

ヒートマップをアップサンプリングで入力画像と同サイズに拡大するだけでは、Pooling 層で失われた情報を復元できないため、semantic segmentation の結果は物体の輪郭がぼやけたものとなる。そこで、途中の層で出力される Pooling 層のサイズの大きいヒートマップを利用することで、輪郭補完された物体の詳細な情報を捉えた semantic segmentation が可能となる（図 13-8 右上）。

図 13-8 : Deconvolution と U-Net



他に有名なのが U-Net である。U-Net で行われる処理は、

- 画像全体のクラス分類 (MNIST など) から FCN と deconvolution を使い、  
**物体の位置情報を出力**
- skip-connection を用いて畳み込みによって失われる位置情報を保持しておき、  
**より精密な領域を出力**

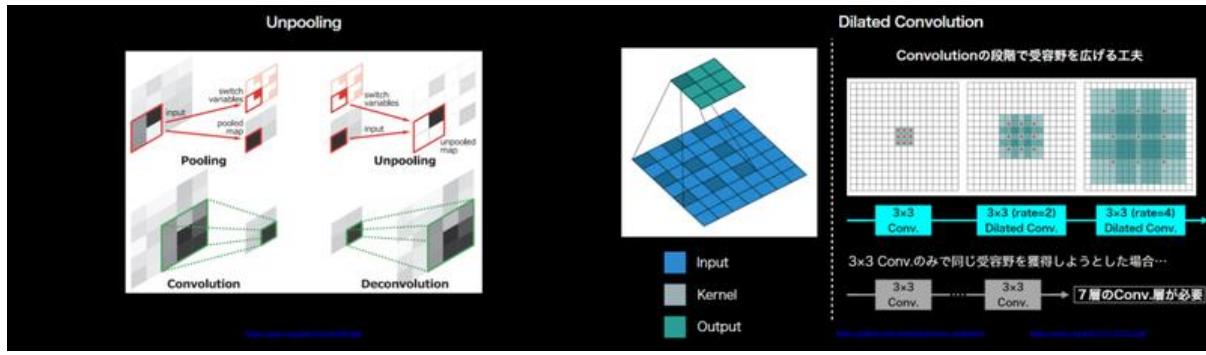
となり、上記 2 つを上手く組み合わせて精度を高めている。

他にも、学習モデルには、以下がある。

- **DeconvNet:** DCN では大まかなセグメンテーション出力が発生する可能性があった  
ラベルマップを DeconvNet により出力  
ラベルマップは、段階的なデコンボリューションと **Unpooling** によって取得
- **SegNet:** 画像セグメンテーションのための深層畳み込みエンコーダ・デコーダーアーキテクチャ

他の需要野を広げる工夫：

- **Dilated Convolution:** マルチスケールのコンテキスト集約



#### ◆実装演習結果キャプチャ

物体検知・セグメンテーションに対する実装演習対象のコード説明とコード提供なしのため、代わりに「【python AI】pytorch SSD 物体検出の実装方法 -学習と確認-」により、SSD のモデルの学習、および、モデルを使った検知の実装と動作を確認した。

<https://hituji-ws.com/code/python/python-od2/>

#### ◆確認テストなどの考察結果

確認テストの出題なしのため省略