深層学習（day1）・要点まとめ

Section0： ニューラルネットワークの全体像

《確認テスト》： DL は結局何をやろうとしているのか。また、その値の最適化が最終目的か。

⇒ 入力値 x に対して、その出力値 y と目的値 d の誤差を最小化（最適化）するように、重み w とバイアス b の値を学習すること。

【事前に用意する情報】

入力データ： 説明変数データ $\boldsymbol{x_n} = [x_{n1} \cdots x_{ni}]$

目的変数データ $\boldsymbol{d_n} = [d_{n1} \cdots d_{ni}]$

【多層ネットワークのパラメータ】

$\boldsymbol{w^{(l)}}$ ・・・ 1) **重み**： $\boldsymbol{W^{(l)}} = \begin{bmatrix} w_{11}^{(l)} & \cdots & w_{1i}^{(l)} \\ \vdots & \ddots & \vdots \\ w_{j1}^{(l)} & \cdots & w_{ji}^{(l)} \end{bmatrix}$

2) **バイアス**： $\boldsymbol{b^{(l)}} = [b_1^{(l)} \cdots b_j^{(l)}]$

活性化関数： $\boldsymbol{f^{(l)}(u^{(l)})} = [f^{(l)}(u_1^{(l)}) \cdots f^{(l)}(u_j^{(l)})]$

中間層出力： $\boldsymbol{z^{(l)}} = [z_1^{(l)} \cdots z_K^{(l)}] = \boldsymbol{f^{(l)}(u^{(l)})}$

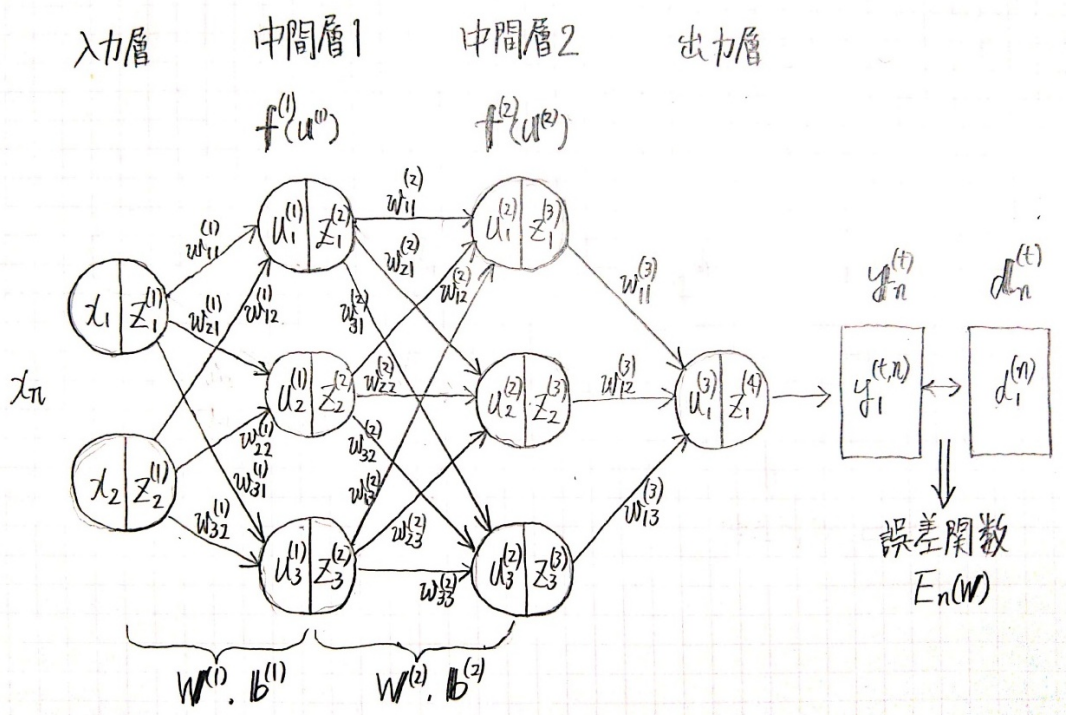
総入力： $\boldsymbol{u^{(l)}} = \boldsymbol{W^{(l)} z^{(l-1)} + b^{(l)}}$

出力： $\boldsymbol{y_n^{(t)}} = [y_{n1}^{(t)} \cdots y_{nK}^{(t)}] = \boldsymbol{z^{(L)}}$

誤差関数： $\boldsymbol{E_n(w)}$

入力層ノードのインデックス： i $(=1 \cdots I)$
中間層ノードのインデックス： j $(=1 \cdots J)$
出力層ノードのインデックス： k $(=1 \cdots K)$
層ノードのインデックス： l $(=1 \cdots L)$
入力データのインデックス： n $(=1 \cdots N)$
試行回数のインデックス： t $(=1 \cdots T)$

《確認テスト》： 入力層：2 ノード 1 層、中間層：3 ノード 2 層、出力層：1 ノード 1 層

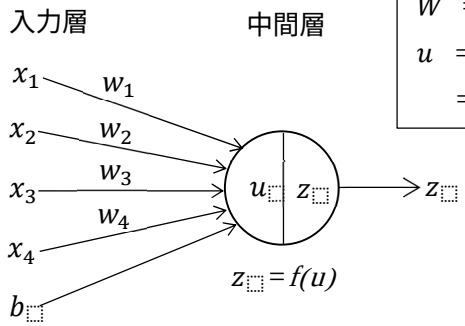


深層学習（day1）・要点まとめ

Section1： 入力層～中間層

▣ 入力層から中間層

入力：x_i
重み：w_i
バイアス：b
総入力：u
出力：z
活性化関数：f
入力層ノードのインデックス： i



$$W = [w_1 \cdots w_i]^T, \quad x = [x_1 \cdots x_i]^T$$
$$u = w_1x_1 + w_2x_2 + w_3x_3 + w_4x_4 + b$$
$$= Wx + b$$

《確認テスト》： 上の図式に動物分類の実例を入れると、入力層が以下のようになる。

$x_1 = 10$	体長	$x_2 = 300$	体重
$x_3 = 300$	ひげの本数	$x_4 = 15$	毛の平均長
$x_5 = 50$	耳の大きさ	$x_6 = 1.8$	眉間／目鼻距離比
$x_7 = 20$	足の長さ		

《確認テスト》： $u = Wx + b$ を Python で書くと以下のようになる。

```
u1 = np.dot(x, W1) + b1
```

《確認テスト》： 中間層の出力を定義しているソースコードの部分（1_1_forward_propagation.ipynb）

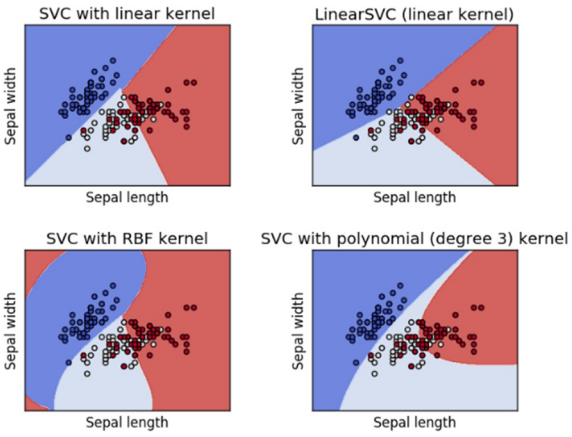
```
# 順伝播（単層・単ユニット）

# 総入力
u = np.dot(x, W) + b
print_vec("総入力", u)

# 中間層出力
z = functions.relu(u)
print_vec("中間層出力", z)
```

Section2： 活性化関数

《確認テスト》： 線形と非線形の違い

⇒ クラス分類において、右図の上2つのように直線によって境界線を構成できるものが**線形**、右図の下2つのように曲線でしか境界線を構成できないものが**非線形**である。

DL では、モデルを非線形にするために非線形な活性化関数を各層に設ける。



深層学習（day1）・要点まとめ

▣ 中間層の活性化関数

活性化関数： $\boldsymbol{f}^{(l)}(\boldsymbol{u}^{(l)}) = [f^{(l)}(u_1^{(l)}) \cdots f^{(l)}(u_j^{(l)})]$

中間層の活性化関数･･･ReLu 関数、シグモイド関数、ステップ関数を用いることが多い

中間層出力： $\boldsymbol{z}^{(l)} = [z_1^{(l)} \cdots z_K^{(l)}] = \boldsymbol{f}^{(l)}(\boldsymbol{u}^{(l)})$

《確認テスト》： 中間層出力$\boldsymbol{z}^{(l)} = \boldsymbol{f}^{(l)}(\boldsymbol{u}^{(l)})$ の該当箇所（1_1_forward_propagation.ipynb）

```
z = functions.sigmoid(u)
```

▣ 出力層の活性化関数

最終層の活性化関数･･･シグモイド関数、ソフトマックス関数、恒等写像を用いることが多い

Section3： 出力層

▣ 誤差関数

《確認テスト》： $\boldsymbol{E_n}(\boldsymbol{w}) = \frac{1}{2}\sum_{j=1}^{J}(y_j - d_j)^2 = \frac{1}{2}\|(\boldsymbol{y}-\boldsymbol{d})\|^2$ において、なぜ、引き算でなく二乗するのか？

1/2 はどのような意味を持つのか？

⇒ 予測値と目標値の差を目標にするには符号の違いを無くす必要があり、絶対値では微分不能な点が発生するため。1/2 は、微分した時に係数を 1 にするためである。

▣ 出力層の活性化関数

- 出力層の種類

	回帰	二値分類	多クラス分類
活性化関数	恒等写像 $f(u) = u$	シグモイド関数 $f(u) = \frac{1}{1+e^{-u}}$	ソフトマックス関数 $f(i,u) = \frac{e^{-u_i}}{\sum_{k=1}^{K} e^{-u_k}}$
誤差関数	二乗誤差	交差エントロピー	交差エントロピー

[誤差データサンプル当たりの誤差]　　[学習サイクル当たりの誤差]

$\boldsymbol{E_n}(\boldsymbol{w}) = \frac{1}{2}\sum_{j=1}^{J}(y_j - d_j)^2$ ･･････ 二乗誤差　　$\boldsymbol{E_n}(\boldsymbol{w}) = \sum_{n=1}^{N} E_n$

$\boldsymbol{E_n}(\boldsymbol{w}) = -\sum_{i=1}^{n} d_i \log y_i$ ･･････ 交差エントロピー

《確認テスト》： ソフトマックス関数に該当するソースコードを示し、一行ずつ処理の説明をせよ。

（/common/functions.py）

```
 # 出力層の活性化関数
# ソフトマックス関数
def softmax(x):
    # ndim :次元数 → N 個の入力データをまとめて処理する場合は次元数が 2
    if x.ndim == 2:  # ndim :次元数 → N 個の入力データに対して処理する場合は次元数が 2
        x = x.T       # 出力層のノードの値を第 0 軸にすべく転置
        x = x - np.max(x, axis=0)        # 0<exp 値<=1 にするオーバーフロー対策
        y = np.exp(x) / np.sum(np.exp(x), axis=0)    # y=f(i,x) ソフトマックス計算
```

深層学習（day1）・要点まとめ

```
        return y.T       # x の配列と合わせるために転置して元に戻す

    x = x - np.max(x) # オーバーフロー対策
    return np.exp(x) / np.sum(np.exp(x))
```

《確認テスト》： 交差エントロピーに該当するソースコードを示し、一行ずつ処理の説明をせよ。
（/common/functions.py）

```
   # クロスエントロピー
def cross_entropy_error(d, y):       # E(w)=−Σd(i)*log{y(i)}
    if y.ndim == 1:      # y の次元数が 1 の場合に、d と y を共に次元数 2 に合わせる
        d = d.reshape(1, d.size)     # d の次元数 2 にする
        y = y.reshape(1, y.size)     # y の次元数 2 にする

    # 教師データ d が one-hot-vector の場合、正解ラベルのインデックスに変換
    if d.size == y.size:
        d = d.argmax(axis=1)     # d の最大要素のインデックスを返す

    batch_size = y.shape[0]      # バッチ数 N
    # 各データ毎の交差エントロピーを求め、バッチ全体の和を取り正規化（オーバーフロー対策含む）
    return -np.sum(np.log(y[np.arange(batch_size), d] + 1e-7)) / batch_size
```

Section4： 勾配降下法

▣ 勾配降下法

以下のように微分値を利用して、誤差$E(w)$を最小化するパラメータwを見つけること（最適化）

勾配降下法 $w^{(t+1)} = w^{(t)} - \varepsilon \nabla E$ （ε：学習率）　　$\nabla E = \frac{\partial E(w)}{\partial w} = \left[\frac{\partial E}{\partial w_1} \cdots \frac{\partial E}{\partial w_M}\right]$

《確認テスト》： 勾配降下法に該当するコード（1_2_back_propagation.ipynb）

```
# 訓練データ
x = np.array([[1.0, 5.0]])
# 目標出力
d = np.array([[0, 1]])
#  学習率
learning_rate = 0.01
network =  init_network()
y, z1 = forward(network, x)
```

```
grad = backward(x, d, z1, y)
for key in ('W1', 'W2', 'b1', 'b2'):
    network[key]  -= learning_rate * grad[key]
```

▣ 確率的勾配降下法（SGD）

確率的勾配降下法	勾配降下法
$w^{(t+1)} = w^{(t)} - \varepsilon \nabla E_n$	$w^{(t+1)} = w^{(t)} - \varepsilon \nabla E$
ランダムに抽出したサンプルの誤差	全サンプルの平均誤差

深層学習（day1）・要点まとめ

【確率的勾配降下法のメリット】：

- データが冗長な場合の計算コストの削減
- 望まない局所極小解に収束するリスクの軽減
- オンライン学習ができる

《確認テスト》： オンライン学習とは？

⇒ 新たにデータを取得した場合などにおいて、リアルタイムに追加で学習ができる。

▣ ミニバッチ勾配降下法)

ミニバッチ勾配降下法	確率的勾配降下法
$w^{(t+1)} = w^{(t)} - \varepsilon \nabla E_t$ $E_t = \frac{1}{N_t} \sum_{n \in D_t} E_n$ $N_t = \lvert D_t \rvert$	$w^{(t+1)} = w^{(t)} - \varepsilon \nabla E_n$
ランダムに分割したデータの集合（ミニバッチ）D_t に属するサンプルの誤差	ランダムに抽出したサンプルの平均誤差

【ミニバッチ勾配降下法のメリット】：

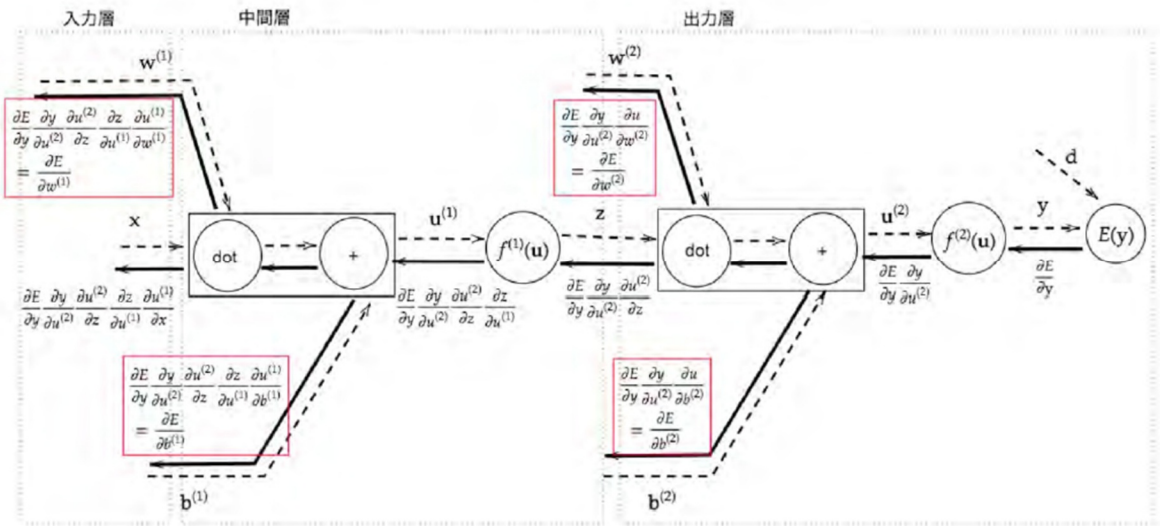
- 確率的勾配降下法のメリットを損なわず、計算資源を有効利用できる → CPU を利用したスレッド並列化や GPU を利用した SIMD 並列化

Section5： 誤差逆伝搬法

▣ 誤差勾配の計算 ー 誤差逆伝搬法

誤差逆伝搬法： 算出された誤差を、出力層側から順に微分し、前の層前の層へと伝搬。最小限の計算で各パラメータでの微分値を解析的に計算する手法。

［メリット］：計算結果（＝誤差）から微分を逆算することで、不要な再帰的計算を避けて微分を算出できる



《確認テスト》： 誤差逆伝搬法では、最終面から最初の面に向けて、前の面の勾配データと順伝搬で計算した中間層出力を保持することで対象の面の勾配データを計算することによって、再帰的処理を避けることができる。（関連するソースコードは以下の通り）

深層学習（day1）・要点まとめ

```
# 誤差逆伝播
def forward(network, x):
    W1, W2 = network['W1'], network['W2']
    b1, b2 = network['b1'], network['b2']

    u1 = np.dot(x, W1) + b1
    z1 = functions.relu(u1)# z1:中間層出力 1
   ·····························
    return y, z1           # 中間層出力 z1 を返す

# 誤差逆伝播
def backward(x, d, z1, y):
    grad = {}

    W1, W2 = network['W1'], network['W2']
    b1, b2 = network['b1'], network['b2']
    #  出力層でのデルタ＝(∂E/∂y)*(∂y/∂u2)⇒交差エントロピーとシグモイド関数の複合導関数
    delta2 = functions.d_sigmoid_with_loss(d, y)    # delta2:(1,2)
    #  b2 の勾配
    grad['b2'] = np.sum(delta2, axis=0)    # delta2 の(1,2)をベクトル化(2,)する
    #  W2 の勾配:中間層の出力 z1 は順伝搬で計算･保存
    grad['W2'] = np.dot(z1.T, delta2)    # grad['W2']:(3,2)←(3,1)×(1,2)
    #  中間層でのデルタ                delta1:(1,3)←(1,2)×(2,3)
    delta1 = np.dot(delta2, W2.T) * functions.d_relu(z1)
    # b1 の勾配
    grad['b1'] = np.sum(delta1, axis=0)
    #  W1 の勾配
    grad['W1'] = np.dot(x.T, delta1)    # grad['W1']:(2,3)←(2,1)×(1,3)

    return grad
```

[考察]

- 行列の積を実装する場合、前の行列の列と後の行列の行の大きさを等しく、結果の行列が目的とする行列の大きさと合うように、積の順序と転置の有無を考慮することがポイント。

▣ 誤差逆伝搬法の計算式

$\boldsymbol{E_n(w)}=\frac{1}{2}\sum_{j=1}^{J}(y_j-d_j)^2=\frac{1}{2}\|(\boldsymbol{y}-\boldsymbol{d})\|^2$　：誤算関数＝二乗誤差関数

$\boldsymbol{y}=\boldsymbol{u^{(L)}}$　：出力層の活性化関数＝恒等写像

$\boldsymbol{u^{(l)}}=\boldsymbol{w^{(l)}z^{(l-1)}}+b^{(l)}$　：総入力の計算

$$\frac{\partial E}{\partial w_{ji}^{(2)}}=\frac{\partial E}{\partial y}\frac{\partial y}{\partial \boldsymbol{u}}\frac{\partial \boldsymbol{u}}{\partial w_{ji}^{(2)}}$$

$$\frac{\partial E}{\partial y}=\frac{\partial}{\partial y}\frac{1}{2}\|(y-d)\|^2=y-d$$

$$\frac{\partial y}{\partial u}=\frac{\partial u}{\partial u}=1$$

深層学習（day1）・要点まとめ

$$\frac{\partial \boldsymbol{u}(\boldsymbol{w})}{\partial w_{ji}}=\frac{\partial}{\partial w_{ji}}(\boldsymbol{w}^{(l)}\boldsymbol{z}^{(l-1)}+b^{(l)})=\frac{\partial}{\partial w_{ji}}\left(\begin{pmatrix} w_{11}z_1+\cdots+w_{1i}z_i+\cdots w_{1I}z_I \\ \vdots \\ w_{j1}z_1+\cdots+w_{ji}z_i+\cdots w_{jI}z_I \\ \vdots \\ w_{J1}z_1+\cdots+w_{Ji}z_i+\cdots w_{JI}z_I \end{pmatrix}+\begin{pmatrix} b_1 \\ \vdots \\ b_j \\ \vdots \\ b_J \end{pmatrix}\right)=\begin{pmatrix} 0 \\ \vdots \\ z_i \\ \vdots \\ 0 \end{pmatrix}$$

$$\frac{\partial E}{\partial w_{ji}^{(2)}}=\frac{\partial E}{\partial y}\frac{\partial y}{\partial \boldsymbol{u}}\frac{\partial \boldsymbol{u}}{\partial w_{ji}^{(2)}}=(y-d)\cdot\begin{pmatrix} 0 \\ \vdots \\ z_i \\ \vdots \\ 0 \end{pmatrix}=(y_j-d_j)\,z_i$$

《確認テスト》： ソースコード 1_3_stochastic_gradient_descent.ipynb から、誤差逆伝搬法における勾配計算部分を探す。

$\frac{\partial E}{\partial y}$ ： `delta2 = functions.d_mean_squared_error(d, y)`

$\frac{\partial E}{\partial y}\frac{\partial y}{\partial u}$ ： `delta2 = functions.d_mean_squared_error(d, y)` ⟨出力層は恒等写像の為、y=u⟩

$\frac{\partial E}{\partial y}\frac{\partial y}{\partial \boldsymbol{u}}\frac{\partial \boldsymbol{u}}{\partial w_{ji}^{(2)}}$ ： `grad['W1'] = np.dot(x.T, delta1)`

ここで用いられる z1 は右のコードで生成される： `z1, y = forward(network, x)`