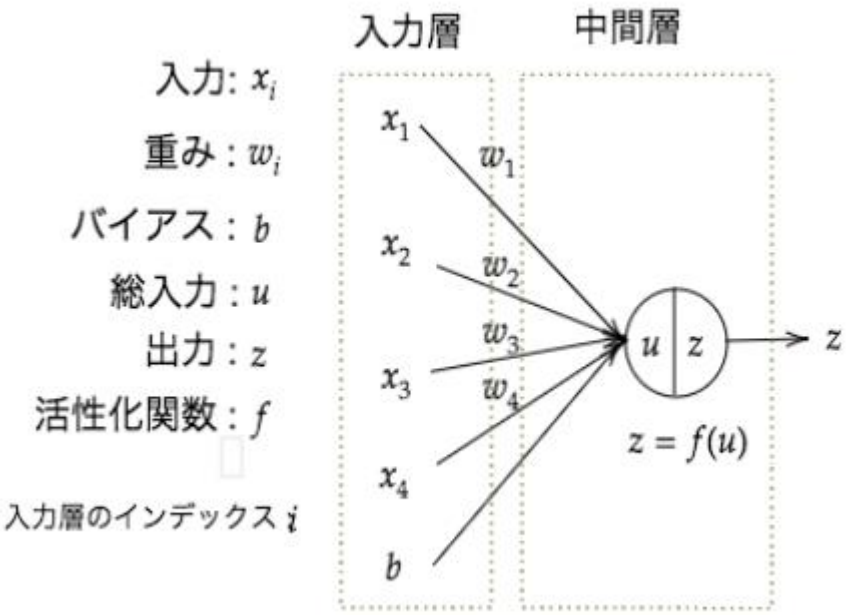
Section1：入力層～中間層

入力層から中間層の構造図は以下のようになる。



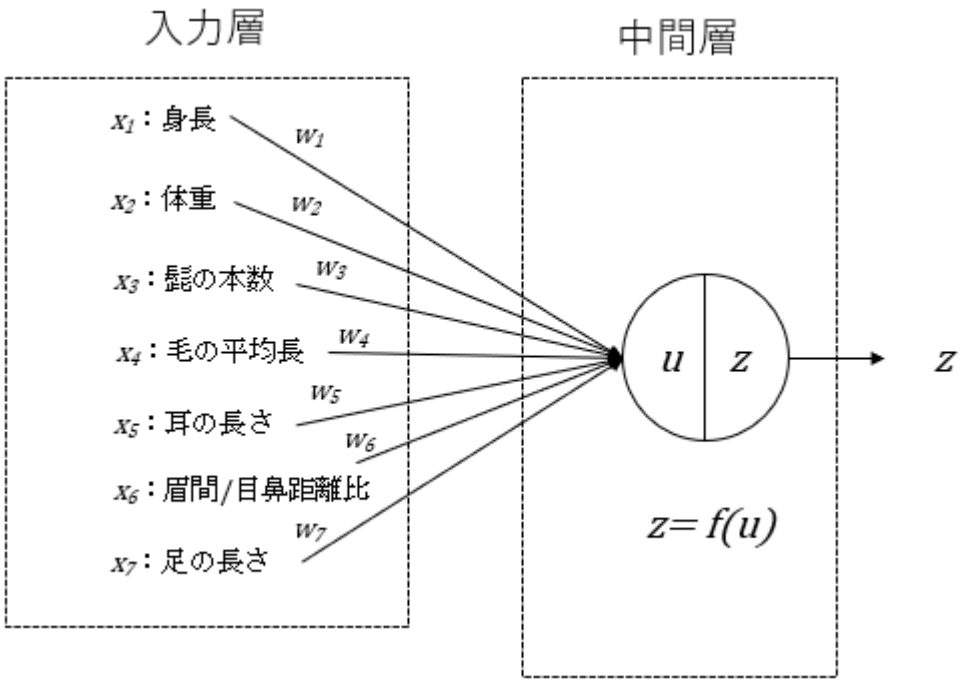
上記を式にすると以下となる。

$$u = w_1x_1 + w_2x_2 + w_3x_3 + w_4x_4 + b = Wx + b$$

$$W = \begin{bmatrix} w_1 \\ \vdots \\ w_I \end{bmatrix}, \quad x = \begin{bmatrix} x_1 \\ \vdots \\ x_I \end{bmatrix}$$

<確認テスト 1>

Q. この図式に動物分類の実例を入れてみよう。



入力：x_iは動物を分類する際に参照する情報。

重み：w_iは参照する情報について、分類する動物にとって特徴的なものほど大きくなる。

例えばウサギを分類するのであれば、ウサギは耳が長いのが特徴なので耳の長さの重みを大きくするなど。

<確認テスト 2>

Q. この数式を Python で書け。

$$u = w_1x_1 + w_2x_2 + w_3x_3 + w_4x_4 + b = Wx + b$$

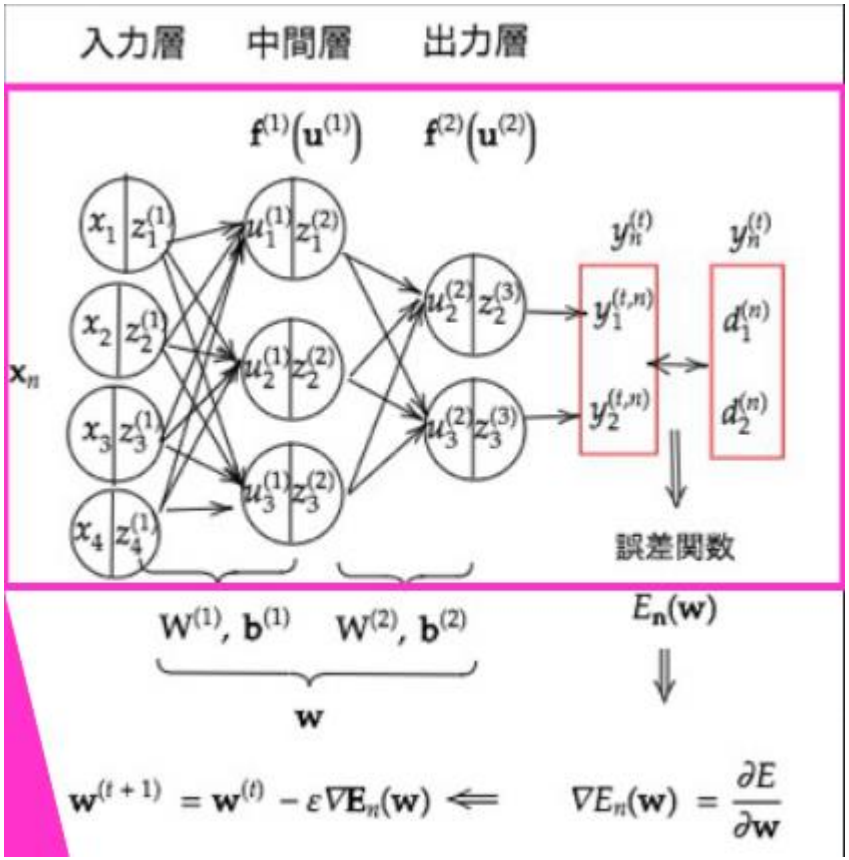
A. u1 = np.dot(x, W) + b

np.dot()は numpy ライブラリの dot()のコールであり、第 1 引数と第 2 引数の行列積を計算する。

$Wx + b$は、行列：xと行列：Wの行列積：Wxに bを足したものなので、上記の式が回答となる。

<確認テスト 3>

Q. 1-1 ファイルから中間層の出力を定義しているソースコードを抜き出せ。



対象のソースコードは、1_1_forward_progration.ipynb の順伝播(3 層・複数ユニット)

A. u2= np.dot(z1, W2) + b2
z2= funcions.relu (u2)

$u_i^1 = W^1 x_i + b^1$により計算されたu_i^1を入力として受け取り、

$u_i^2 = W^2 u_i^1 + b^2$を計算している部分であるため、2 層の総入力と 2 層の総出力の処理を記述している回答箇所が該当する。

<実装演習>

1_1_forward_progration.ipynb

Section2：活性化関数

ニューラルネットワークにおいて、次の層への出力の大きさを決める非線形の関数。

非線形な関数のため、加法性・斉次性を持たない。

加法性：$f(x+y) = f(x) + f(y)$を満たすこと。

斉次性：$f(kx) = kf(x)$を満たすこと。

活性化関数の種類

中間層用と出力層用に分けられる。

中間層用の活性化関数

・ステップ関数

閾値を超えたら発火する関数であり、出力は常に 1 か 0。

パーセプトロン(ニューラルネットワークの前身)で利用された関数。

$$f(x) = \begin{cases} 1\ (x \geq 0) \\ 0\ (x < 0) \end{cases}$$

0, 1 のみの表現のため、線形分離可能なものしか学習できない。

・シグモイド関数(ロジスティック関数)

0〜1 の間を緩やかに変化する関数で、ステップ関数が ON/OFF の状態のみなのに対して、信号の強弱を伝えられる。

$$f(u) = \frac{1}{1+e^{-u}}$$

大きな値では出力の変化が微小なため、勾配消失問題を引き起こす。

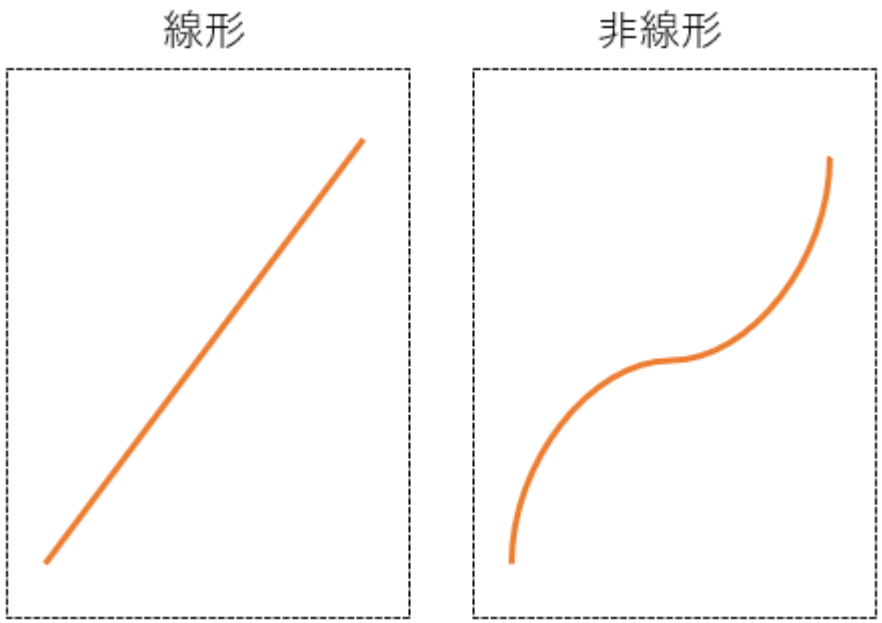
・ReLU 関数

最も使用されている活性化関数。勾配消失問題の回避とスパース化に貢献することで良い成果をもたらしている。

$$f(x) = \begin{cases} x\ (x > 0) \\ 0\ (x \leq 0) \end{cases}$$

<確認テスト 1>

Q. 線形と非線形の違いを図にかいて簡単に説明せよ。



加法性・斉次性の有無が線形と非線形の違い。

砕けた言い方をすると直線系が線形、直線系でないのが非線形。

<確認テスト 2>

Q. 配布されたソースコードより$z = f(u)$に該当する箇所を抜き出せ。

対象のソースコードは、1_1_forward_progration.ipynb の順伝播(3 層・複数ユニット)

A. z1 = functions.relu(u1)

z2 = functions.relu(u2)

中間層の活性化関数は次の層への出力の大きさを決めるための関数であるため、$u = Wx + b$の計算で算出した値に対して処理している箇所になる。

上記の計算は np.dot()で行われるため、u1, u2 にこの計算値が設定されており、u1, u2 の値を利用した計算をしているのは回答の箇所になる。

functions.relu()は common ディレクトリ下の functins.py で記述されている relu()をコールしており、ReLU 関数の処理が記述されている。

そのため、中間層の活性化関数には ReLU 関数が利用されていることがわかる。

<実装演習>

1_1_forward_propagation_after.ipynb

Section3：出力層

誤差関数とは算出した値(予測値)と正解値の誤差を計算するための関数のこと。

代表的なものとして、二乗誤差と交差エントロピーの 2 つがある。

二乗誤差

$$E_n(w) = \frac{1}{2}\sum_{j=1}^{J}\left(y_j - d_j\right)^2 = \frac{1}{2}\|(y-d\|^2$$

y_j：算出した値(予測値)、d_j：正解値

1/2 する理由：誤差逆伝播の計算で、誤差関数の微分を用いるため、その際の計算式を簡単にするため。本質的な意味はない。

2 乗する理由：各ラベルでの誤差で正負の両方の値が発生するため、全体誤差を誤差の和から判断することができないため。

交差エントロピー

$$E_n(w) = -\sum_{i=1}^{I} d_i \log y_i$$

y_i：算出した値(予測値)、d_i：正解値

誤差が大きいほど値が大きくなるため学習効率が二乗誤差よりも良い。

出力層の活性化関数

中間層の活性化関数は閾値の前後で信号の強弱を調整するが、出力層の活性化関数は信号の大きさ(比率)はそのままで変換する。

分類問題の場合、出力層の出力は 0～1 の範囲に限定し、総和が 1 となるようにする必要があるのでこのような変換が必要になる。

出力層用の活性化関数

・恒等写像

回帰の場合に使用する。誤差関数には二乗誤差を使用。

$f(u) = u$

・シグモイド関数(ロジスティック関数)

二値分類の場合に使用する。誤差関数には交差エントロピーを使用。

$$f(u) = \frac{1}{1 + e^{-u}}$$

・ソフトマックス関数

多クラス分類の場合に使用する。誤差関数には交差エントロピーを使用。

$$f(i, u) = \frac{e^{u_i}}{\sum_{k=1}^{K} e^{u_k}}$$

訓練データサンプルあたりの誤差

$E_n(w) = \frac{1}{2}\sum_{i=1}^{I}(y_i - d_i)$　　二乗誤差

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

学習サイクルあたりの誤差

$$E(w) = \sum_{n=1}^{N} E_n$$

活性化関数の効果により、一部の出力は弱く、一部の出力は強く伝播される。

<確認テスト 1>

Q. なぜ、引き算ではなく二乗するのか述べよ。

A. 引き算を行うだけでは、各ラベルでの誤差で正負両方の値が発生し、全体の誤差を正しく表すのに都合が悪い。

2 乗してそれぞれのラベルでの誤差を正の値になるようにする。

正負の両方の値がある場合、総和を計算する際に都合が悪い。2 乗すれば元の値の正負によらず正の値になるため、総和の計算がしやすくなる。

Q. $E_n(w) = \frac{1}{2}\sum_{j=1}^{J}\left(y_j - d_j\right)^2 = \frac{1}{2}\|(y-d\|^2$の 1/2 はどういう意味を持つか述べよ。

A. 実際にネットワーク学習するときに行う誤差逆伝播の計算で、誤差関数の微分を用いるが、その際の計算式を簡単にするため。
本質的な意味はない。

誤差関数を微分して誤差の最小値を求める必要があり、$\left(y_j - d_j\right)^2 = (y_j^2 - 2y_j d_j + d_j^2)$を微分するので、1/2 の係数だと計算が簡単になる。

<確認テスト 2>

Q. 以下の数式の①～③に該当するソースコードを示し、一行づつ処理の説明をせよ。

$f(i,u) = \frac{e^{u_i}}{\sum_{k=1}^{K} e^{u_k}}$　　①：$f(i,u)$、②：e^{u_i}、③：$\sum_{k=1}^{K} e^{u_k}$

※対象ソースコードは common/function.py の softmax()

A. ①np.exp(x) / np.sum(np.exp(x), axis=0)　②np.exp(x)　③np.sum(np.exp(x), axis=0)
np.exp()は numpy の exp()をコールする記述であり、これはネイピア数 e を底とする指数関数を計算する関数なのでe^{u_i}の処理を行っている。
nu.sum()は numpy の sum()をコールする記述であり、これは要素の和を計算する関数なので③のe^{u_i}の総和を計算する処理を行っている。
①　はe^{u_i}/e^{u_i}の総和を計算しているため、$f(i,u)$の処理を行っている。
axis は和を計算する軸の方向を指定する引数であり、axis=0 は 2 次元配列を例にすると列方向での和の指定。

<確認テスト 3>

Q. 以下の数式の①～②に該当するソースコードを示し、一行づつ処理の説明をせよ。

$E_n(w) = -\sum_{i=1}^{I} d_i \log y_i$　　①：$E_n(w)$、②：$-\sum_{i=1}^{I} d_i \log y_i$

※対象ソースコードは common/function.py の cross_entropy_error()

A. ①、②共に-sum(np.log(y[np.arange(batch_size), d] + 1e-7))

np.log()は numpy の log()をコールする記述であり、これはネイピア数 e を底とする対数を計算する処理である。
np.arange()は numpy の arange()をコールする記述であり、0 以上、第 1 引数未満に対して間隔 1 で等差数列の配列を生成する。
np.log(y[np.arange(batch_size), d]が$d \log y$を計算する処理に該当する。
log(x)は x=0 だと-∞となるため、微小値である 1e-7 を足すことで log(0)となることを防止している。

<実装演習>

1_2_back_propagation.ipynb

Section4：勾配降下法

勾配降下法

学習を通して誤差を最小にするネットワークを作成すること。
誤差$E(w)$を最小化するパラメータ wを発見すること。

$$w^{(t+1)} = w^t - \varepsilon \nabla E_n(w)$$

$$\nabla E_n(w) = \frac{\partial E}{\partial w} = \left[\frac{\partial E}{\partial w_1} \cdots \frac{\partial E}{\partial w_M}\right]$$

ε：学習率

学習率の決定、収束性向上のためのアルゴリズム
・Momentum
・AdaGrad
・Adadelta
・Adam ※よく使用されるアルゴリズム。

エポック(epoch)
誤差関数の値をより小さくする方向に重み W 及びバイアス b を更新し次の周(エポック)に反映する。

確率的勾配降下法(SGD)
勾配降下法は全サンプルの平均誤差なのに対して、ランダムに抽出したサンプルの誤差を計算する。

$$w^{(t+1)} = w^t - \varepsilon \nabla E_n$$

以下のメリットがある。
・データが冗長な場合の計算コストが軽減される。
・望まない局所極小解に収束するリスクが軽減される。
・オンライン学習ができる。

オンライン学習
　学習データが入ってくるたびに都度パラメータを更新し、学習を進めていく方法。
バッチ学習

ミニバッチ勾配降下法
一般的に使用されている。
勾配降下法は全サンプルの平均誤差なのに対して、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平均誤差。

$$w^{(t+1)} = w^t - \varepsilon \nabla E_t$$

$$E_t = \frac{1}{N_t} \sum_{n \in D_t} E_n$$

$$N_t = |D_t|$$

以下のメリットがある。
・確率的勾配降下法のメリットを損なわずに、計算機の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(Single Instruction Multi Data)並列化。

誤差勾配の計算

$$\nabla E_n(w) = \frac{\partial E}{\partial w} = \left[\frac{\partial E}{\partial w_1} \cdots \frac{\partial E}{\partial w_M}\right]$$

$$\frac{\partial E}{\partial w_m} \approx \frac{E(w_m + h) - E(w_m - h)}{2h}$$

上記の微分式から数値微分により求める。

数値微分

プログラムで微小な数値を生成し疑似的に微分を計算する一般的な手法。

各パラメータ w_m それぞれについて $E(w_m + h)$ や $E(w_m - h)$ を計算するため、順伝播の計算を繰返し行う必要があり負荷が大きいデメリットがある。

→数値微分ではなく、誤差逆伝播法により計算する。

<確認テスト 1>

Q. $w^{(t+1)} = w^t - \varepsilon \nabla E_n(w)$ に該当するソースコードを探してみよう。

※対象のソースコードは 1_3_stochastic_gradient_descent.ipynb

A. network[key] -= learning_rate * grad[key]

grad = backward(x, d, z1, y)

learning_rate は 0.07 の値が設定されている学習率：ε に該当する。

backward()は、誤差逆伝播を計算する関数として定義されており、grad[]には key に対応した $\nabla E_n(w)$ が設定されているため、

learning_rate * grad[key]で $\varepsilon \nabla E_n(w)$ が計算できるため、w^t が設定されている network[]から”-=”の演算子で減算することで求めている。

<確認テスト 2>

Q. オンライン学習とは何か 2 行でまとめよ。

A. 学習データが入ってくる度に都度パラメータを更新し、学習を進めていく方法。

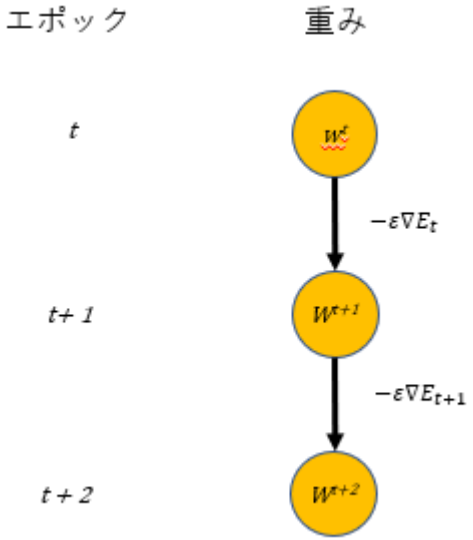
一方、バッチ学習では一度に全ての学習データを使用してパラメータ更新を行う。

最近の深層学習ではオンライン学習というのがかなり重要な要素になっている。

<確認テスト 3>

Q. $w^{(t+1)} = w^t - \varepsilon \nabla E_t$ の数式の意味を図に書いて説明せよ。

A.



<実装演習>

1_3_stochastic_gradient_descent.ipynb

Section5：誤差逆伝播法

算出された誤差を出力層側から順番に微分し、前の層前の層へと伝播し、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。

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。

誤差関数に二乗誤差関数、出力層の活性化関数に恒等写像を利用している場合は、以下のように計算できる。

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

$y = u^{(L)}$　　出力層の活性化関数　　：恒等写像

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

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

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

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

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

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

<確認テスト 1>

Q. 誤差逆伝播法では不要な再帰的処理を避けることができる。すでに行った計算結果を保持しているソースコードを抽出せよ。

※対象のソースコードは、1_3_stochastic_gradient_descent.ipynb

A. delta2 = functions.d_mean_squared_error(d, y)

d_mean_squared_error()は、$\frac{\partial E(y)}{\partial y} = \frac{\partial}{\partial y}\frac{1}{2}\|y-d\|^2 = y-d$の計算結果を返すため、delta2 には$\frac{\partial E(y)}{\partial y}$が設定される。

$\frac{\partial E(y)}{\partial y}$は、$\frac{\partial E}{\partial w^{(2)}} = \frac{\partial E}{\partial y}\frac{\partial y}{\partial u}\frac{\partial u}{\partial w^2}$の計算や$\frac{\partial E}{\partial b^{(2)}} = \frac{\partial E}{\partial y}\frac{\partial y}{\partial u}\frac{\partial u}{\partial b^{(2)}}$の計算に利用されるため、

delta2 に計算結果を設定しておき、delta2 に相当する部分の再計算せずに処理できるようにしている。

<確認テスト 2>

Q. 2 つの空欄に該当するソースコードを探せ。

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

$\frac{\partial E}{\partial y}\frac{\partial y}{\partial u}$　　空欄①

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

※ここで用いられる z1 は以下のコードで生成される。

z1, y = forward(network, x)

※対象のソースコードは、1_3_stochastic_gradient_descent.ipynb

A. 空欄①：delta1 = np.dot(delta2, W2.T) * functions.d_sigmoid(z1)

空欄②：grad['W2'] = np.dot(z1.T, delta2)

$\frac{\partial E(y)}{\partial y}\frac{\partial y}{\partial u}\frac{\partial u}{\partial w_{ji}^{(2)}} = (y-d)\cdot\begin{bmatrix}0\\ \vdots\\ z_i\\ \vdots\\ 0\end{bmatrix} = (y_j - d_j)z_i$より、空欄②には grad['W2']への値設定を記述している部分が該当する。

出力層の活性化関数にはシグモイド関数を利用しているため、

$y = \frac{1}{1+e^{-u}}$ より、$\frac{\partial y}{\partial u} = \left(1 - \frac{1}{1+e^{-u}}\right)\frac{1}{1+e^{-u}}$となるため、空欄①にはシグモイド関数の微分の計算値を掛けているdelta1 の部分が該当する。

<実装演習>

1_4_1_mnist_sample.ipynb