深層学習前編（day1）レポート

2024/6/8

1　Section1: 入力層～中間層

図 1 に、入力層と中間層の模式図を示す。入力層では、数値ベクトルである入力 $\boldsymbol{x} = (x_1, x_2, \cdots)$ を受け取り、各入力値に重み $bmw = (w_1, w_2, \cdots)$ を掛けて次の中間層へ値を渡す。すなわち、入力データが n 次元だと、j 番目の中間層が受け取る値 u^j は式 1 となる。

$$u^j = w_{j1}x_1 + w_{j2}x_2 + \cdots w_{jn}x_n + b \tag{1}$$

中間層の値は、活性化関数 f を用いてさらに別の値に変換したうえで、出力層（またはさらなる中間層）へ出力されることになる。入力層 2 つ、中間層 1 つの実装演習

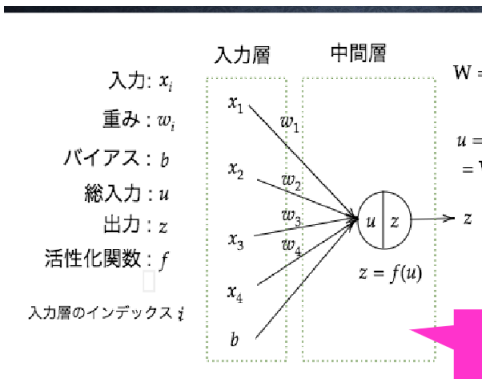


図 1: 入力層と中間層の模式図（講義用スライドより）

上では、図 2 が計算部分にあたる。なお、function_relu は活性化関数の 1 つである ReLU 関数である。出力結果は図 2 となり、中間層の値が $0.1 \times 2 + 0.2 \times 3 + 0.5 = 1.3$ と計算された。

```python
# 順伝播（単層・単ユニット）

# 重み
W = np.array([[0.1], [0.2]])

## 試してみよう_配列の初期化
#W = np.zeros(2)
#W = np.ones(2)
#W = np.random.rand(2)
#W = np.random.randint(5, size=(2))

print_vec("重み", W)


# バイアス
b = np.array(0.5)

## 試してみよう_数値の初期化
#b = np.random.rand() # 0~1のランダム数値
#b = np.random.rand() * 10 -5  # -5~5のランダム数値

print_vec("バイアス", b)

# 入力値
x = np.array([2, 3])
print_vec("入力", x)


# 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)
```

図 2: 実装演習上での入力層～中間層計算部分

```
*** 重み ***
[[0.1]
 [0.2]]
shape: (2, 1)

*** バイアス ***
0.5
shape: ()

*** 入力 ***
[2 3]
shape: (2,)

*** 総入力 ***
[1.3]
shape: (1,)

*** 中間層出力 ***
[1.3]
shape: (1,)
```

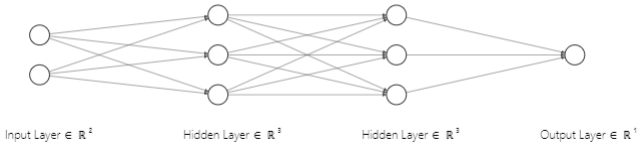
図 3: 図 2 の出力結果

1.1 確認テスト

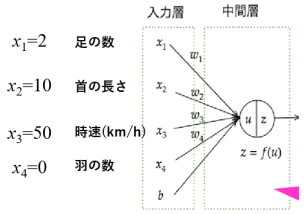
1.1.1 ディープラーニングは、結局何をやろうとしているか 2 行以内で述べよ。また、次の中のどの値の最適化が最終目的か。全て選べ。

多数の中間層を持つニューラルネットワークを用いて、学習を繰り返すことで入力値から出力値に変換する数学モデルを構築すること。
最適化の最終目的：重み、バイアス

1.1.2 次のネットワークをかけ。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



1.1.3 この図式に動物分類の実例を入れてみよう。



1.1.4 この数式を Python で書け。

u = np.dot(x,W) + b

1.1.5 1-1 のファイルから中間層の出力を定義しているソースを抜き出せ。

z2 = functions.relu(u2)

2 Section2: 活性化関数

活性化関数は中間層だけでなく、例えば分類問題などでは出力を 0 1 の範囲にする必要があることから、出力層でも用いられる。中間層、出力層それぞれでよく用いられる活性化関数は以下である。

- 中間層用の活性化関数
  - ReLU 関数
  - シグモイド（ロジスティック）関数
  - ステップ関数
- 出力層用の活性化関数
  - ソフトマックス関数
  - 恒等写像
  - シグモイド（ロジスティック）関数

なお、ここでの恒等写像とは、受け取った値をそのまま出力する関数、つまり $f(u) = u$ である。そのため、その他の関数について詳しく説明する。

2.1 ステップ関数

式 2 で表される、しきい値を超えたら発火する関数である。出力は常に 1 か 0 となる。0 1 の間の値を表現できないため、線形分離可能なものしか学習できない。

$$f(u) = \begin{cases} 1(u \geq 0) \\ 0(u < 0) \end{cases} \tag{2}$$

実装演習上（図 4）では、np.where を用いて、0 より大きい要素を 1 に、それ以外を 0 にして実装している。

```
# ステップ関数（閾値0）
def step_function(x):
    return np.where( x > 0, 1, 0)
```

図 4: 実装演習上でのステップ関数

2.2 シグモイド関数

式 3 で表される、0 1 の間を緩やかに変化する関数である。ステップ関数と異なり、0 1 の間の値を伝えられるようになった。しかし、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ことがあった。

$$f(u) = \frac{1}{1 + e^{-u}} \tag{3}$$

実装演習上（図 5）では、np.exp を用いて計算している。

```
# シグモイド関数（ロジスティック関数）
def sigmoid(x):
    return 1/(1 + np.exp(-x))
```

図 5: 実装演習上でのシグモイド関数

2.3 ReLU 関数

式 4 で表される、今最も使われている活性化関数である。勾配消失問題の回避とスパース化に貢献することができる。

$$f(u) = \begin{cases} u(u > 0) \\ 0(u \leq 0) \end{cases} \tag{4}$$

実装演習上（図 6）では、np.maximum を用いて、0 と受け取った値のうち大きい方を出力して表現している。

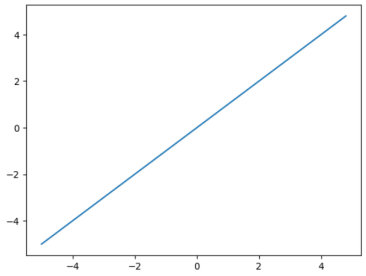
```
# ReLU関数
def relu(x):
    return np.maximum(0, x)
```

図 6: 実装演習上での ReLU 関数

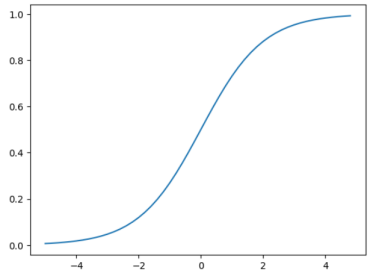
2.4 確認テスト

2.4.1 線形と非線形の違いを図にかいて簡易に説明せよ。

線形な関数は加法性：$f(x + y) = f(x) + f(y)$ と斉次性：$f(kx) = kf(x)$ を満たす。一方、非線形な関数は加法性・斉次性を満たさない。



線形な関数



非線形な関数

2.4.2　配布されたソースコードより該当する箇所を抜き出せ。

z1 = functions.relu(u1)
z2 = functions.relu(u2)

3　Section3：出力層

重みやバイアスの更新をするため、出力層で出力された値と実際の正解の値（訓練データの目的値）がどれくらい異なっているかを、誤差関数を用いて計算する必要がある。誤差関数として、式 5,6 で表される、二乗誤差や交差エントロピーが用いられる。ここで、n はデータ数、d は訓練データの目的値、y は出力層で出力された値である。

- 回帰問題の場合
  - 二乗誤差

$$E_n(w) = \frac{1}{2}\sum_{i=1}^{n}(y_i - d_i)^2 \tag{5}$$

- 二値分類、多クラス分類の場合
  - 交差エントロピー

$$E_n(w) = -\sum_{i=1}^{n} d_i \log y_i \tag{6}$$

交差エントロピーは、実装演習上（図 7) では、微小な値（1e-7）を足すことで、対数の総和を計算したときに負の無限大に発散しなように工夫している。

```
# クロスエントロピー
def cross_entropy_error(d, y):
    if y.ndim == 1:
        d = d.reshape(1, d.size)
        y = y.reshape(1, y.size)

    # 教師データが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)

    batch_size = y.shape[0]
    return -np.sum(np.log(y[np.arange(batch_size), d] + 1e-7)) / batch_size
```

図 7: 実装演習上での交差エントロピー

3.1 確認テスト

3.1.1 なぜ引き算ではなく 2 乗するか述べよ。下式の 1/2 はどういう意味を持つか述べよ。

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

- 2 乗する理由
  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、誤差の足し算をする際に全体の誤差を正しく表しにくい。そのため、2 乗してそれぞれのラベルでの誤差が正の値になるようにしている。
- 1/2 する理由
  実際にネットワークを学習するときに行う誤差逆伝搬の計算で、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。

3.1.2 1〜3 の数式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。

1. np.exp(x)/np.sum(np.exp(x))
2. np.exp(x)
3. np.sum(np.exp(x))

「if x.ndim == 2:」から始まる if 句は、x の次元数が 2 以上でもソフトマックス関数を計算できるようにした部分。本質的なソフトマックス関数の計算は「np.exp(x)/np.sum(np.exp(x), axis=0)」「np.exp(x)/np.sum(np.exp(x))」であり、「x=x-np.max(x)」はオーバーフロー対策に入れたプログラムを安定させる意味のコードである。

3.1.3 1 2 の数式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。

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

「if y.ndim == 1:」から始まる if 句は、y の次元数が 1 のときにも交差エントロピーを計算できるようにした部分である。本質的な交差エントロピーの計算は「np.sum(np.log(y[np.arange(batch_size), d] + 1e-7))」である。ここで、e^{-7} を足し算しているのは、自然対数が 0 に近い値をとると-∞に収束してしまい、計算上都合が悪いためである。

4 Section4：勾配降下法

出力層の誤差関数を最小化するように重みやバイアスを最適化するため、勾配降下法を用いて更新していく。勾配降下法とは、パラメータを少しずつ更新して勾配（微分値）が 0 になる点を探索するアルゴリズムであり、式 7 のように更新し

ていく。ここで、$\boldsymbol{w}^{(t)}$ は現在のパラメータ、$\boldsymbol{w}^{(t+1)}$ は更新後のパラメータ、ϵ は学習率である。

$$\boldsymbol{w}^{(t+1)} = \boldsymbol{w}^{(t)} - \epsilon \nabla E \tag{7}$$

$$where\ \nabla E = \frac{\partial E}{\partial \boldsymbol{w}} = [\frac{\partial E}{\partial w_1}, \cdots, \frac{\partial E}{\partial w_m}]$$

学習率について、値が小さい場合、パラメータが収束するまでに時間がかかる、または局所解に陥って、真の最適解に収束しない可能性がある。一方で学習率が大きい場合、最適解にたどり着けずに発散してしまう可能性がある。そのため、学習率の決定アルゴリズムは複数の論文で公開されており、以下がその例である。

- Momentum
- AdaGrad
- Adadelta
- Adam

誤差 ∇E について、通常の勾配降下法では全サンプルの平均誤差を用いる。他には、計算時間の短縮やオンライン学習を可能にするため、ランダムに抽出したサンプルの誤差を用いる確率的勾配降下法や、ランダムに分割したデータ集合に属するサンプルの平均誤差を用いるミニバッチ勾配降下法などがある。確率的勾配降下法は、実装演習上では図 8 のように実装している。ここでは以下の流れでパラメータ更新を行っている。

1. データ数 100,000 のうち、ランダムに 1,000 個抽出する。
2. 1,000 個のうち 1 つのデータについて出力層まで値を出し、誤差を計算する。
3. 式 7 に従って全てのパラメータを更新する（パラメータの更新方法は誤差逆伝搬法を用いている）。
4. 2 3 を残りの全てのデータについて行う。

1000 回の誤差の変遷は図 9 となる。1 つのデータについてしか誤差を計算していないが、1000 回行うことで誤差がほぼ 0 に近づいていることがわかる。

4.1 確認テスト

4.1.1 該当するソースコードを探してみよう。

network[key] -= learning1_rate * grad[key]

4.1.2 オンライン学習とは何か。2 行でまとめよ。

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。一方、バッチ学習では一度にすべての学習データを使ってパラメータ更新を行う。

```python
losses = []
# 学習率
learning_rate = 0.07

# 抽出数
epoch = 1000

# パラメータの初期化
network = init_network()
# データのランダム抽出
random_datasets = np.random.choice(data_sets, epoch)

# 勾配降下の繰り返し
for dataset in random_datasets:
    x, d = dataset['x'], dataset['d']
    z1, y = forward(network, x)
    grad = backward(x, d, z1, y)
    # パラメータに勾配適用
    for key in ('W1', 'W2', 'b1', 'b2'):
        network[key]  -= learning_rate * grad[key]

    # 誤差
    loss = functions.mean_squared_error(d, y)
    losses.append(loss)
```

図 8: 実装演習上での確率的勾配降下法

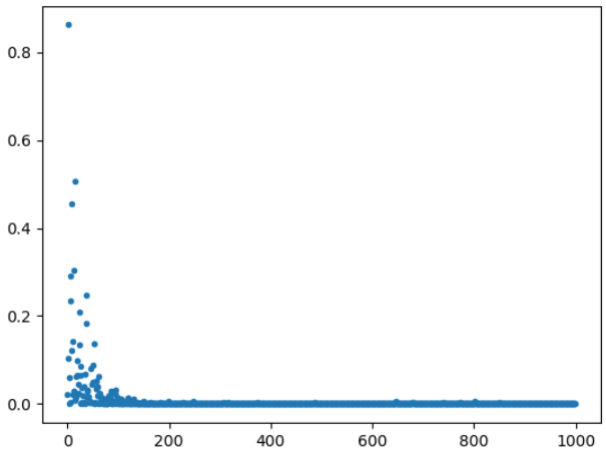
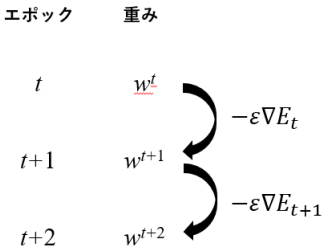


図 9: 出力層から得られた値と実際の値の誤差の変遷（1000 回）

4.1.3 この数式の意味を図に書いて説明せよ。

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

エポック t を経るごとに、誤差関数から算出された微分値分、重みの更新を行っていく。



5 Section5：誤差逆伝播法

勾配降下法の ∇E について、前述した全て or 一部のサンプルの誤差を用いるかだけでなく、出力層から離れたパラメータ（入力層側に近いパラメータ）をどのように計算するかも重要となってくる。出力層側に近いパラメータから 1 つ更新→新しいネットワークで再度誤差計算→次のパラメータを更新 $\cdots$ では時間がかかるため、微分の連鎖律を用いた誤差逆伝搬法を用いて一度の誤差計算で全てのパラメータを更新する。図 10 は、誤差逆伝搬法の模式図である。例えば、入力層から中間層への重み $\boldsymbol{w}^{(1)}$ についての偏微分は、微分の連鎖律から式 8 となる。

$$\frac{\partial E}{\partial \boldsymbol{w}^{(1)}} = \frac{\partial E}{\partial \boldsymbol{y}} \frac{\partial \boldsymbol{y}}{\partial \boldsymbol{u}^{(2)}} \frac{\partial \boldsymbol{u}^{(2)}}{\partial \boldsymbol{z}} \frac{\partial \boldsymbol{z}}{\partial \boldsymbol{u}^{(1)}} \frac{\partial \boldsymbol{u}^{(1)}}{\partial \boldsymbol{w}^{(1)}} \tag{8}$$

誤差逆伝搬法は、実装演習上では図 11 のように実装している。各パラメータの grad に入っている値が微分の連鎖律によって求めた結果となっている。

5.1 確認テスト

5.1.1 誤差逆伝搬法では不要な再帰的処理を避ける事ができる。既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。（答えは 12 ページ目）

5.1.2 2 つの空欄に該当するソースコードを探せ

出力層は以下。
delta2 = functions.d_mean_squared_error(d, y)
grad['W2'] = np.dot(z1.T, delta2)
中間層は以下。
delta1 = np.dot(delta2, W2.T) * functions.d_sigmoid(z1)
grad['W1'] = np.dot(x.T, delta1)

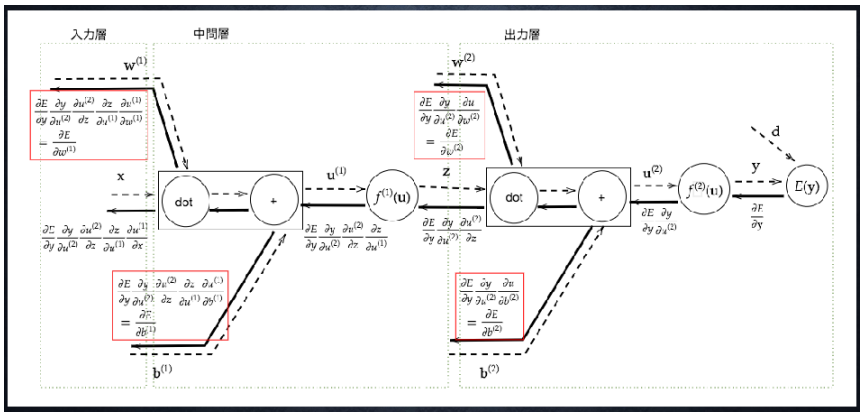


図 10: 誤差逆伝搬法（講義用スライドより）

```python
# 誤差逆伝播
def backward(x, d, z1, y):
    # print("¥n##### 誤差逆伝播開始 #####")

    grad = {}

    W1, W2 = network['W1'], network['W2']
    b1, b2 = network['b1'], network['b2']

    # 出力層でのデルタ
    delta2 = functions.d_mean_squared_error(d, y)
    # b2の勾配
    grad['b2'] = np.sum(delta2, axis=0)
    # W2の勾配
    grad['W2'] = np.dot(z1.T, delta2)
    # 中間層でのデルタ
    #delta1 = np.dot(delta2, W2.T) * functions.d_relu(z1)

    ## 試してみよう
    delta1 = np.dot(delta2, W2.T) * functions.d_sigmoid(z1)

    delta1 = delta1[np.newaxis, :]
    # b1の勾配
    grad['b1'] = np.sum(delta1, axis=0)
    x = x[np.newaxis, :]
    # W1の勾配
    grad['W1'] = np.dot(x.T, delta1)

    # print_vec("偏微分_重み1", grad["W1"])
    # print_vec("偏微分_重み2", grad["W2"])
    # print_vec("偏微分_バイアス1", grad["b1"])
    # print_vec("偏微分_バイアス2", grad["b2"])

    return grad
```

図 11: 実装演習上での誤差逆伝搬法

```
# 出力層でのデルタ
delta2 = functions.d_mean_squared_error(d, y)
# b2の勾配
grad['b2'] = np.sum(delta2, axis=0)
# W2の勾配
grad['W2'] = np.dot(z1.T, delta2)
# 中間層でのデルタ
#delta1 = np.dot(delta2, W2.T) * functions.d_relu(z1)

## 試してみよう
delta1 = np.dot(delta2, W2.T) * functions.d_sigmoid(z1)

delta1 = delta1[np.newaxis, :]
# b1の勾配
grad['b1'] = np.sum(delta1, axis=0)
x = x[np.newaxis, :]
# W1の勾配
grad['W1'] = np.dot(x.T, delta1)
```

確認テスト 5.1.1 の答え