深層学習 day1 レポート

導入

・確認テスト

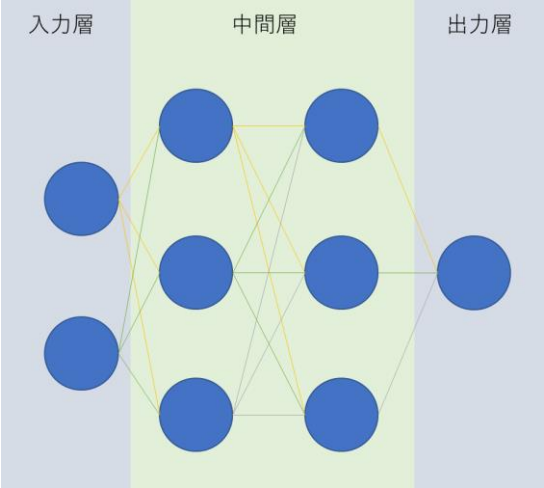
[ディープラーニングは、結局何をやろうとしているか 2行以内で述べよ。]

万能近似定理によりニューラルネットワークでの複雑な関数を表現し、入力データから目的関数となる出力を算出する。

[次の中のどの値の最適化が最終目的か]

③重み[W]、④バイアス[b]の最適化

[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]



Section1：入力層～中間層

入力層：ニューラルネットワークが最初に情報を受け取る層となる。
　　ニューラルネットワークに何かしらのインプットを行う層であり、各ノードで
　　入力より数値を受け取る。（生き物では電気信号で情報を受け取る）

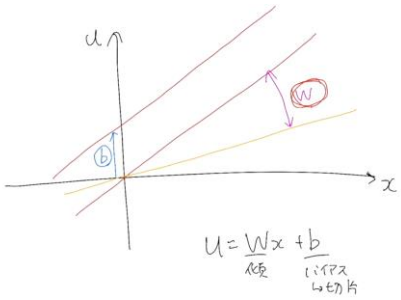
中間層：入力層から情報を受け継ぎ、さまざまな計算を行う層。
　　中間層が多いほど複雑な分析が可能（万能近似定理）
　　中間層は隠れ層と呼ばれる場合もある。

出力層：入力層、中間層で重みをかけ、活性化関数で処理された値が示される出力層。

重み：入力層の重要度を表現するもの。
　　入力の値をいい感じにシャッフルし出力するため重みを調整する。

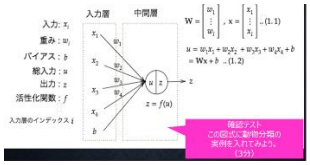
バイアス：w と x だけでは値をシフトすることができない。（傾きを表現している。）
　　バイアスを用いることでシフトすることができる。

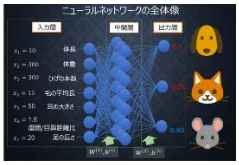
参考資料：Neural Network Console（SONY）

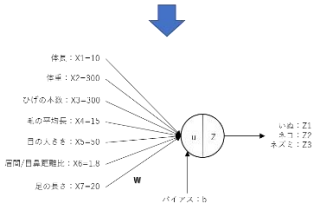


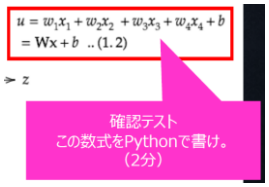
傾きと切片が決まれば、一次関数では関数を決めることができる。
ニューラルネットワークの学習ではこのｗとｂを学習することになる。

・確認テスト

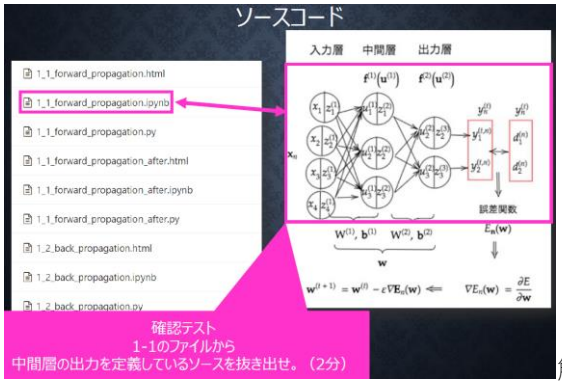








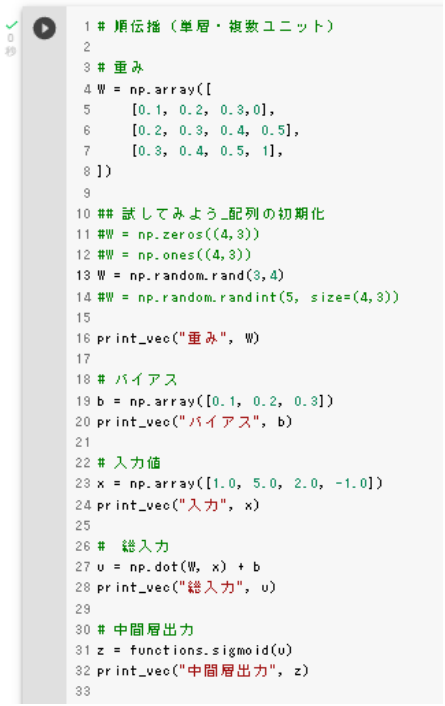
解答：
```
u = np.dot(W, x) + b
```



解答：
```
# 2層の総入力
u2 = np.dot(z1, W2) + b2

# 2層の総出力
z2 = functions.relu(u2)
```

・実習演習

順伝播（単層・複数ユニット）



```python
# 順伝播（単層・複数ユニット）

# 重み
W = np.array([
    [0.1, 0.2, 0.3,0],
    [0.2, 0.3, 0.4, 0.5],
    [0.3, 0.4, 0.5, 1],
])

## 試してみよう_配列の初期化
#W = np.zeros((4,3))
#W = np.ones((4,3))
W = np.random.rand(3,4)
#W = np.random.randint(5, size=(4,3))

print_vec("重み", W)

# バイアス
b = np.array([0.1, 0.2, 0.3])
print_vec("バイアス", b)

# 入力値
x = np.array([1.0, 5.0, 2.0, -1.0])
print_vec("入力", x)

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

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

```

支給ファイルの試してみようでは、W が 4x3 で記入されていたためエラーとなった。

W=np.random.rand(3,4)へ変更し、実行を行った。

```
*** 重み ***
[[0.233063   0.02587027 0.31105677 0.96633896]
 [0.95506073 0.34413852 0.85488792 0.64611252]
 [0.44716423 0.5759112  0.24649931 0.46086534]]
shape: (3, 4)

*** バイアス ***
[0.1 0.2 0.3]
shape: (3,)

*** 入力 ***
[ 1.  5.  2. -1.]
shape: (4,)

*** 総入力 ***
[0.11818892 3.93941664 3.65885349]
shape: (3,)

*** 中間層出力 ***
[0.52951288 0.98091188 0.97488498]
shape: (3,)
```

無事実行完了。

W と X で行列演算できる形へすることは重要と感じた。

Section2：活性化関数

活性化関数を使うことでニューラルネットワークの幅が広がる。

活性化関数は非線形の関数となっているため。

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（ほぼ使われることはない。パーセプトロンで利用された関数）

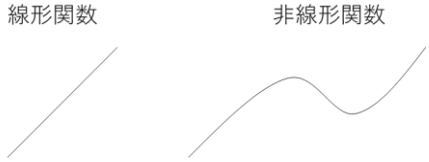
出力層用の活性化関数

- 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

活性化関数に微分を適用できるようになり、

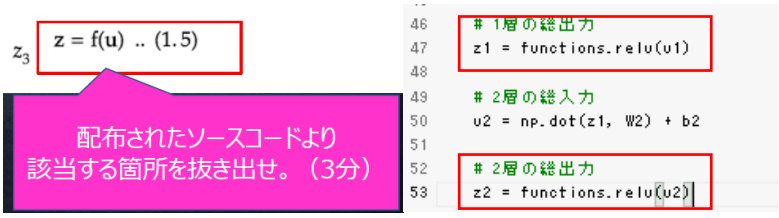
・確認テスト

[線形と非線形の違いを図にかいて簡易に説明せよ。]



※ 非線形関数は加法性・斉次性を満たさない（講師）

[配布されたソースコードより該当する箇所を抜き出せ。]



ここでは ReRU 関数を使用

※提供別ファイルで活性化関数は定義（common ファイル）

・実習演習

順伝播（3 層・複数ユニット）

活性化関数を ReLU 関数→シグモイド関数へ変更して出力の変化を確認してみる。

ReLU 関数

```
*** 出力 ***
[85.46151642 42.8777085  35.42770326 63.55549018]
shape: (4,)

出力合計: 227.3224183500664
```

シグモイド関数

```
# 1層の総出力
# z1 = functions.relu(u1)
z1 = functions.sigmoid(u1)

# 2層の総入力
u2 = np.dot(z1, W2) + b2

# 2層の総出力
#z2 = functions.relu(u2)
z2 = functions.sigmoid(u2)
```

```
*** 出力 ***
[3.01742837 2.32818806 3.23545458 3.25864259]
shape: (4,)

出力合計: 11.839713600998046
```

ReLU では 227.32 だった結果が、シグモイド関数では 11.83 となる。
これは入力層・中間層の出力が活性化関数により制限がかかったためと考える。

・その他
活性化関数について調べたところ多くの関数が活用されているとのこと。
参考に下記に記載する。
恒等関数／線形関数
ステップ関数
シグモイド関数
tanh 関数
ReLU
ソフトプラス関数
Leaky ReLU
PReLU／Parametric ReLU
ELU
SELU
Swish 関数
Mish 関数
ソフトマックス関数

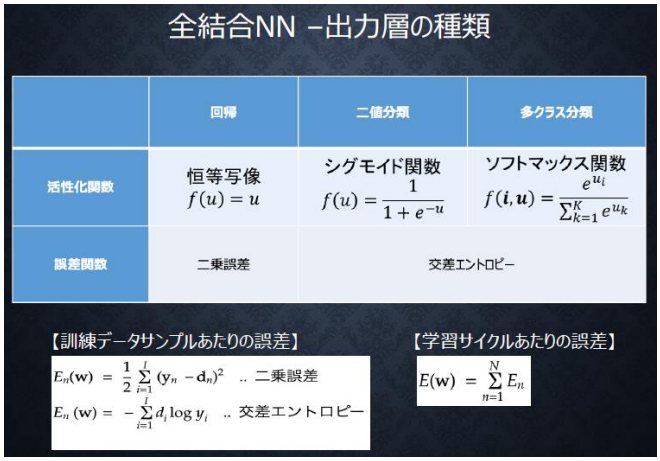
Section3：出力層

出力層：入力層、中間層で重みをかけ、活性化関数で処理された値が示される出力層。
　　私たち人間がほしい情報へ変換した出力を行ってくれる層となる。
　　動物分類のような問題では、イヌの確率やネコの確率などで出力する必要がある。
誤差関数：ニューラルネットワーク全体を学習させるために重要な関数となる。

訓練データとして入力データとそれに対する正解データをペアで用意し、
訓練データの入力データよりニューラルネットワークで予測を行う。
この結果を正解データと比べることで答え合わせを行う。
この際に使用する関数が誤差関数となる。
具体的には下記に示す二乗和誤差などがある。

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

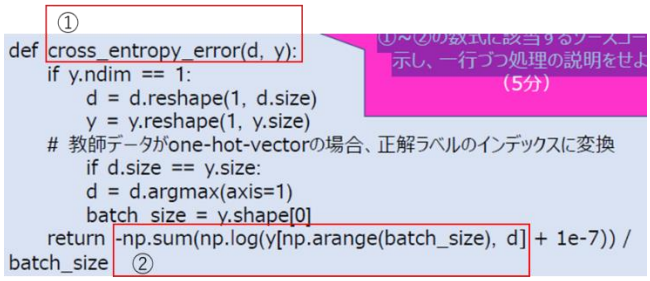
出力層の中間層との違い
　中間層 ：しきい値の前後で信号の強弱を調整
　出力層 ：信号の大きさ（比率）はそのままに変換
出力層と中間層で利用される活性化関数が異なる



・確認テスト
[なぜ、引き算でなく二乗するか述べよ]
誤差はプラス、マイナス両方発生するため、単純に引き算すると誤差を表現できないのでは。
二乗することで正の数にそろえることができる。
[下式の 1/2 はどういう意味を持つか述べよ]
ニューラルネットワークの学習の際にバックプロパゲーションを行うが、
その時に微分が必要となり、誤差関数の二乗により 2 が発生するため 1/2 を用いている。
[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
def softmax(x):
        if x.ndim == 2:
    x = x.T
    x = x - np.max(x, axis=0)
    y = np.exp(x) /np.sum(np.exp(x), axis=0)
    return y.T  ②                          ③
x = x - np.max(x) # オーバーフロー対策
    return np.exp(x) / np.sum(np.exp(x))
                ②                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。]



1e-7 は np.log(0)は計算できないため,0 とならないためにつけている。

・実習演習

回帰（2-3-2 ネットワーク）

誤差関数を二乗誤差とクロスエントロピー誤差で比較してみる。

```
54 # 入力値
55 x = np.array([1., 2., 3.])
56 network =  init_network()
57 y, z1 = forward(network, x)
58 # 目標出力
59 d = np.array([2., 4.])
60 # 誤差
61 loss = functions.mean_squared_error(d, y)
62 loss_2 = functions.cross_entropy_error(d, y)
63
64 ## 表示
65 print("¥n##### 結果表示 #####")
66 print_vec("中間層出力", z1)
67 print_vec("出力", y)
68 print_vec("訓練データ", d)
69 print_vec("二乗誤差",  loss)
70 print_vec("クロスエントロピー誤差",  loss_2)
```

```
*** 二乗誤差 ***
3613.87373486424
shape: ()

*** クロスエントロピー誤差 ***
-4.469187970953645
shape: ()
```

やはり二乗誤差とクロスエントロピー誤差では値が全く異なる。

・その他（参考：AVILEN AI Trend より）

線形回帰の損失関数：平均二乗誤差(meen squared error)

$$MSE(y_i, \hat{y}_i) = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

y_iは実値、$\hat{y}_i$は予測値を指す。

回帰問題に用いる損失関数：

平均二乗誤差(meen squared error)、

$$MSE(y_i, \hat{y}_i) = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

y_iは実値、$\hat{y}_i$は予測値を指す。

平均二乗誤差の性質として外れ値に対して敏感である。

外れ値を含むデータに適用すると、予測結果が不安定となる。（外れ値に引っ張られる）

平均絶対誤差 / Mean Absolute Error

$$MAE(y_i, \hat{y}_i) = \frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i|$$

y_i は実値、$\hat{y}_i$ は予測値を指す。

平均絶対誤差の性質として外れ値に強い

平均二乗対数誤差 / Mean Squared Logarithmic Error

$$MSLE(y_i, \hat{y}_i) = \frac{1}{n}\sum_{i=1}^{n}\{log(1+y_i) - log(1+\hat{y}_i)\}^2$$

y_i は実値、$\hat{y}_i$ は予測値を指す。

平均二乗対数誤差を用いたモデルは予測が実値を上回りやすくなるという傾向がある。

交差エントロピー誤差 / cross entropy error

$$E = -\sum_{k} t_k \log y_k$$

交差エントロピー誤差は実際のカテゴリーに対する予測確率のみを評価する。
異なる予測に対しては 0 を掛ける。

Section4：勾配降下法

ニューラルネットワークを学習させるための手法となる。
学習を通じて誤差を最小にするネットワークを作成するが、そのパラメータ更新に利用。

・学習率
　学習率が大きすぎた場合、最小値にいつまでもたどり着かず発散してしまう。
　学習率が小さい場合発散することはないが、収束するまでに時間がかかってしまう。
　また、極小解から抜け出せないことがある。

勾配降下法の学習率の決定、収束性向上のためのアルゴリズムについて複数の論文が公開され、よく利用されている。

- Momentum
- AdaGrad
- Adadelta
- Adam

※上記アルゴリズム詳細は Day2 で触れることになる。

誤差関数の値をより小さくする方向へ学習のサイクルをエポックと呼ぶ。
1 エポックでは 1 回の学習となる。

確率的勾配降下法（SGD）
勾配降下法では全サンプルの平均誤差で学習したが、SGD ではランダムに抽出したサンプルの誤差で学習する。下記メリットがある。

- データが冗長な場合の計算コストの軽減
- 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（バッチ学習の逆となる。）

ミニバッチ勾配降下法
SGD ではランダムに抽出したサンプルでの学習となるが、ミニバッチ勾配降下法では
ランダムに分割したデータ群に属するサンプル平均誤差により学習する方法となる。
確率的勾配降下法のメリットを損なわず、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 並列化のメリットがある。

誤差勾配の計算 ・・・ 微分
プログラムで微小な数値を生成し擬似的に微分を計算する一般的な手法 ← 計算量大
→ 誤差逆伝播法を利用する

・確認テスト

[該当するソースコードを探してみよう。]

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

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

```
# パラメータに勾配適用
for key in ('W1', 'W2', 'b1', 'b2'):
    network[key]  -= learning_rate * grad[key]
```

```
grad = backward(x, d, z1, y)
# パラメータに勾配適用
```

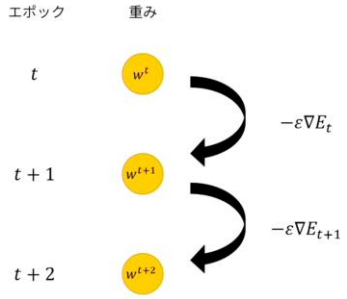
[オンライン学習とは何か]

学習毎にニューラルネットワークのパラメータの更新をする学習方法。

バッチ学習は学習データをまとめ平均誤差などを使ってパラメータの更新を行う方法。

[この数式の意味を図に書いて説明せよ。]

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



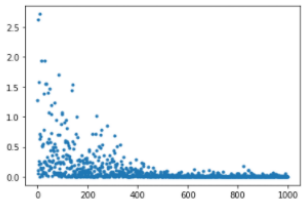
各学習毎にネットワーク（重み）を更新している。

・実習演習

1_3_stochastic_gradient_descent.ipynb

```
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)

print("##### 結果表示 #####")
lists = range(epoch)


plt.plot(lists, losses, '.')
# グラフの表示
plt.show()
```

```
##### 結果表示 #####
```



ここでは確率的勾配効果法による学習を実施する。

当たり前ではあるが、学習を繰り返すことで誤差が小さくなっている。

Section5：誤差逆伝播法

Deep Learning の学習においては、誤差関数の値がゼロになる（近づく）ようにパラメータを決めたい。この際に利用するのが誤差逆伝播法となる。

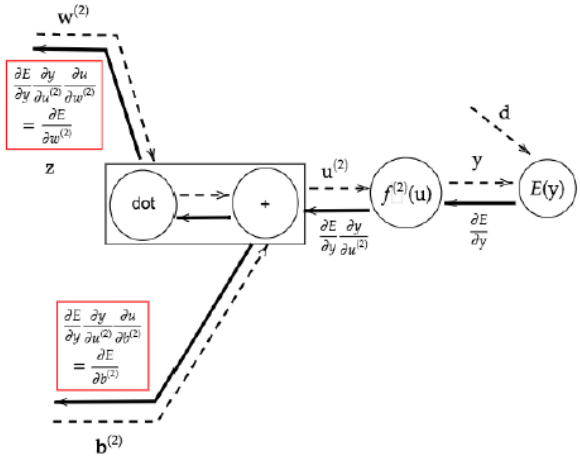
【誤差逆伝播法】

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播。

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

誤差勾配の計算では、誤差逆伝播法を用いる。

誤差から微分を逆算することで、不要な再帰的計算（計算量大）を避けて微分を算出できる



誤差逆伝播法の計算式

微分の連鎖律を用い計算量を減らしている。

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

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

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

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

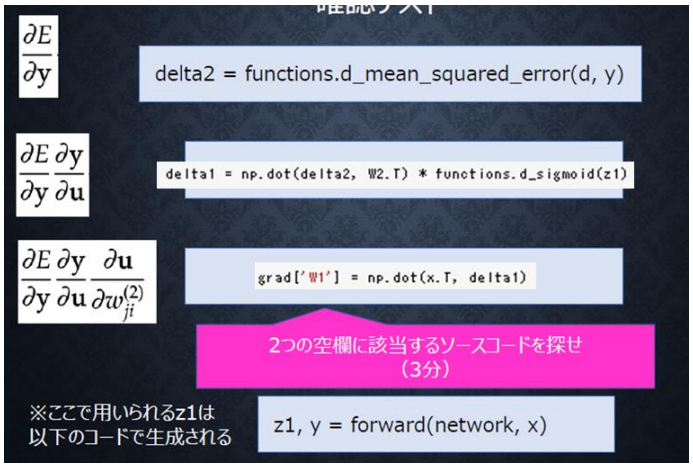
$$\frac{\partial E(\mathbf{y})}{\partial \mathbf{y}} = \frac{\partial}{\partial \mathbf{y}}\frac{1}{2}\|\mathbf{y}-\mathbf{d}\|^2 = \mathbf{y}-\mathbf{d}$$

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

・確認テスト

[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。]

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



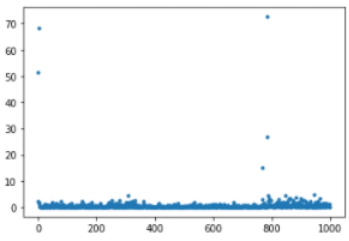
・実習演習

1_3_stochastic_gradient_descent.ipynb

学習率を 0.07 から 0.5 へ変更した場合を下記に示す。

```
losses = []
# 学習率
#learning_rate = 0.07
learning_rate = 0.5

# 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)

print("##### 結果表示 #####")
lists = range(epoch)


plt.plot(lists, losses, '.')
# グラフの表示
plt.show()
```

```
##### 結果表示 #####
```



Section4：勾配降下法で実施した際に比べ誤差が収束せず、800 エポック前後で誤差増加が発生した。このことから誤差逆伝播法での学習時の学習率は重要であることがわかる。