AI自動作曲プログラミング

第１２回

ディープラーニング
プログラミング

バックプロパゲーション
を実装した
ニューラルネットワーク
回帰

バックプロパゲーションを実装したニューラルネットワーク 回帰

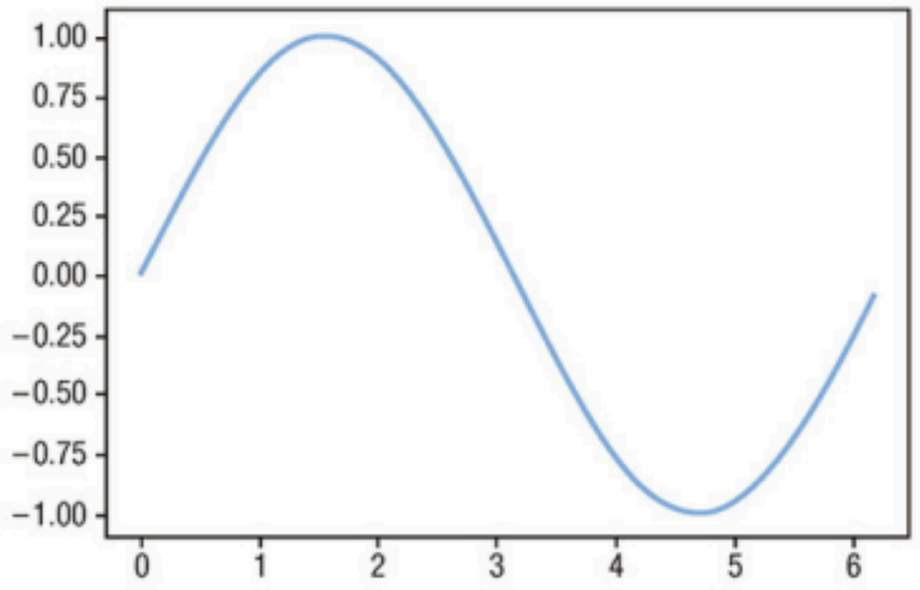
シンプルなニューラルネットワークにバックプロパゲーションを実装し、ネットワークが学習するプログラムを確認します。
ディープラーニングライブラリーの活用などはせずにPythonのみで実装します。

sin関数の学習

x座標を入力、y座標を出力とし、正解をsin(x)とします。

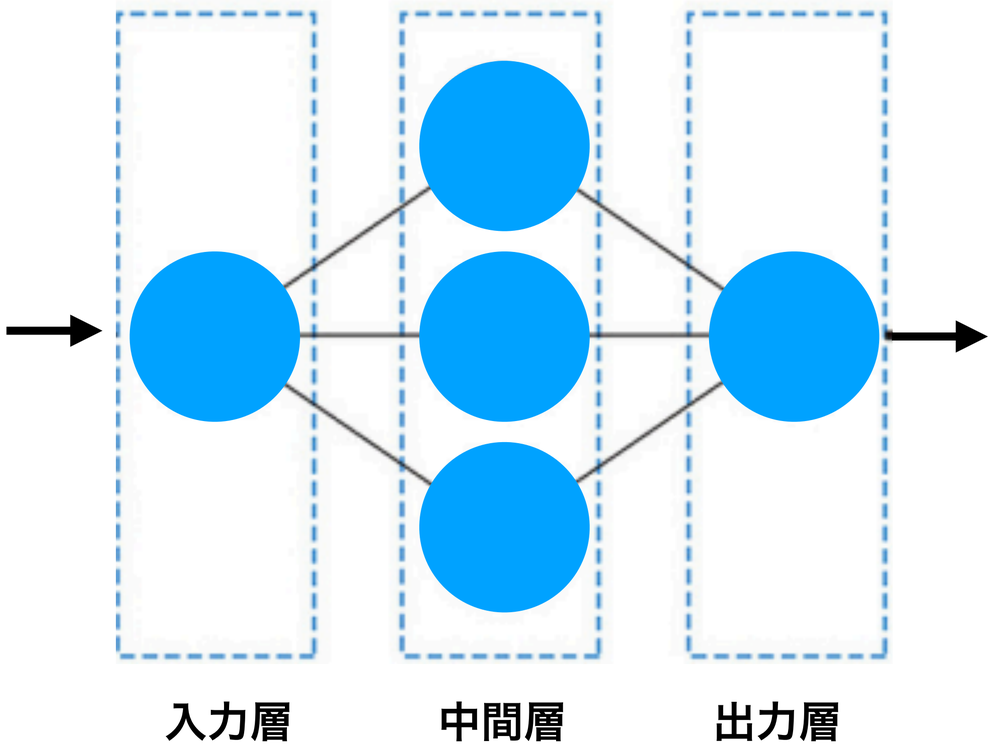
sin関数は連続した関数なので回帰問題となります。

誤差を逆伝播させ重みとバイアスを調整する事でsin関数の正解に近づきます。



バックプロパゲーションを実装したニューラルネットワーク 回帰

作成するニューラルネットワーク
入力層＝１
中間層＝３
出力層＝１
の３層のシンプルなニューラルネットワーク



中間層の活性化関数：シグモイド関数
出力層の活性化関数：恒等関数
損失関数：二乗和誤差
最適化アルゴリズム：確率的勾配降下法
バッチサイズ：１（オンライン学習＝逐次学習）

今回は訓練データのみ。テストデータはなし。

バックプロパゲーションを実装したニューラルネットワーク 回帰

データおよび入力の実装

バックプロパゲーションを実装したニューラルネットワーク 回帰

```
%matplotlib inline

import numpy as np
import matplotlib.pyplot as plt

# -- 入力と正解の用意 --
input_data = np.arange(0, np.pi*2, 0.1)  # 入力
correct_data = np.sin(input_data)  # 正解
input_data = (input_data-np.pi)/np.pi  # 入力を-1.0-1.0の範囲に収める
n_data = len(correct_data)  # データ数

# -- 各設定値 --
n_in = 1  # 入力層のニューロン数
n_mid = 3  # 中間層のニューロン数
n_out = 1  # 出力層のニューロン数

wb_width = 0.01  # 重みとバイアスの広がり具合
eta = 0.1  # 学習係数
epoch = 2000
interval = 100  # 経過の表示間隔
```

バックプロパゲーションを実装したニューラルネットワーク 回帰

```
%matplotlib inline

import numpy as np
import matplotlib.pyplot as plt
```

numpy、matpolotlibなど必要なモジュールのインポート。
配列作成やグラフ表示用です。
%matplotlib inlineはJupyternotebook、Google Colab使用の際必要です。

バックプロパゲーションを実装したニューラルネットワーク 回帰

```
# -- 入力と正解の用意 --
input_data = np.arange(0, np.pi*2, 0.1)  # 入力
correct_data = np.sin(input_data)  # 正解
input_data = (input_data-np.pi)/np.pi  # 入力を-1.0-1.0の範囲に収める
n_data = len(correct_data)  # データ数
```

numpyのarange関数を使用して0からpi x 2まで0.1ステップの配列を作成します。
正解データはnumpyのsin関数で作成します。
入力を表示をわかりやすくするために-1.0~1.0の範囲にします。
n_dataにデータ数を代入しインスタンス化します

バックプロパゲーションを実装したニューラルネットワーク 回帰

```
# -- 各設定値 --
n_in = 1  # 入力層のニューロン数
n_mid = 3  # 中間層のニューロン数
n_out = 1  # 出力層のニューロン数
```

ニューロン数の設定です。
入力層 x 1
中間層 x 3
出力層 x 1
です。

バックプロパゲーションを実装したニューラルネットワーク 回帰

```
wb_width = 0.01  # 重みとバイアスの広がり具合
eta = 0.1  # 学習係数
epoch = 2000
interval = 100  # 経過の表示間隔
```

wb_widthは分布の広がり具合です。
etaは学習率（学習係数）
epochで学習回数を設定
intervalで学習結果のグラフ表示の間隔を設定します。

バックプロパゲーションを実装したニューラルネットワーク 回帰

中間層の実装

バックプロパゲーションを実装したニューラルネットワーク 回帰

```
# -- 中間層 --
class MiddleLayer:
    def __init__(self, n_upper, n):  # 初期設定
        self.w = wb_width * np.random.randn(n_upper, n)  # 重み（行列）
        self.b = wb_width * np.random.randn(n)  # バイアス（ベクトル）

    def forward(self, x):  # 順伝播
        self.x = x
        u = np.dot(x, self.w) + self.b
        self.y = 1/(1+np.exp(-u))  # シグモイド関数

    def backward(self, grad_y):  # 逆伝播
        delta = grad_y * (1-self.y)*self.y  # シグモイド関数の微分

        self.grad_w = np.dot(self.x.T, delta)
        self.grad_b = np.sum(delta, axis=0)

        self.grad_x = np.dot(delta, self.w.T)

    def update(self, eta):  # 重みとバイアスの更新
        self.w -= eta * self.grad_w
        self.b -= eta * self.grad_b
```

出力層との違いは活性化関数がシグモイド関数である事と、変数deltaを求める際の計算式の違いです。

バックプロパゲーションを実装したニューラルネットワーク 回帰

```
class MiddleLayer:
    def __init__(self, n_upper, n):  # 初期設定
        self.w = wb_width * np.random.randn(n_upper, n)  # 重み（行列）
        self.b = wb_width * np.random.randn(n)  # バイアス（ベクトル）
```

コンストラクタ（__init__）でオブジェクトの初期設定を行う。
上の層（入力層または中間層）のニューロン数（n_upper）とこの層のニューロン数（n）を引数として受け取る。
重みはn_upper x nの行列
バイアスは要素数 n のベクトル
各要素（重みとバイアス）の初期値をnumpyのrandom.randn関数でランダムに作成。
randn関数は形状に合わせた各要素数のfloat型の配列を返す。（今回は行列とベクトル）
wb_widthは要素の分布の広がり具合。

バックプロパゲーションを実装したニューラルネットワーク 回帰

```
def forward(self, x):  # 順伝播
    self.x = x
    u = np.dot(x, self.w) + self.b
    self.y = 1/(1+np.exp(-u))  # シグモイド関数
```

forwardメソッドは順伝播のメソッド。
numpyのdot関数で入力と重みの行列積を求め、にバイアスを足しuを求め、活性化関数で出力を計算します。
中間層の活性化関数はシグモイド関数です。

バックプロパゲーションを実装したニューラルネットワーク 回帰

```
def backward(self, grad_y):  # 逆伝播
    delta = grad_y * (1-self.y)*self.y  # シグモイド関数の微分

    self.grad_w = np.dot(self.x.T, delta)
    self.grad_b = np.sum(delta, axis=0)

    self.grad_x = np.dot(delta, self.w.T)
```

backwardメソッドは逆伝播のメソッド。
deltaを求める計算はシグモイド関数の微分になるので
中間層の出力の勾配x（１ー中間層の出力）x中間層の出力
になります。
deltaを用いて
重みの勾配：grad_w
バイアスの勾配：grad_b
中間層の入力の勾配：grad_x
を求めます。

バックプロパゲーションを実装したニューラルネットワーク 回帰

```
def update(self, eta):  # 重みとバイアスの更新
    self.w -= eta * self.grad_w
    self.b -= eta * self.grad_b
```

upddateメソッドは重みとバイアスの更新用のメソッドです。
学習率：eta
重みの勾配：grad_w
バイアスの勾配：grad_b
それぞれの勾配に学習率をかけて更新量とし現在の値から引く事で更新します

バックプロパゲーションを実装したニューラルネットワーク 回帰

出力層の実装

バックプロパゲーションを実装したニューラルネットワーク 回帰

```python
# -- 出力層 --
class OutputLayer:
    def __init__(self, n_upper, n):  # 初期設定
        self.w = wb_width * np.random.randn(n_upper, n)  # 重み（行列）
        self.b = wb_width * np.random.randn(n)  # バイアス（ベクトル）

    def forward(self, x):  # 順伝播
        self.x = x
        u = np.dot(x, self.w) + self.b
        self.y = u  # 恒等関数

    def backward(self, t):  # 逆伝播
        delta = self.y - t

        self.grad_w = np.dot(self.x.T, delta)
        self.grad_b = np.sum(delta, axis=0)

        self.grad_x = np.dot(delta, self.w.T)

    def update(self, eta):  # 重みとバイアスの更新
        self.w -= eta * self.grad_w
        self.b -= eta * self.grad_b
```

バックプロパゲーションを実装したニューラルネットワーク 回帰

```
class OutputLayer:
    def __init__(self, n_upper, n):  # 初期設定
        self.w = wb_width * np.random.randn(n_upper, n)  # 重み（行列）
        self.b = wb_width * np.random.randn(n)  # バイアス（ベクトル）
```

コンストラクタ（__init__）でオブジェクトの初期設定を行う。
上の層（中間層）のニューロン数（n_upper）とこの出力層のニューロン数（n）を引数として受け取る。
重みはn_upper x nの行列
バイアスは要素数 n のベクトル
各要素（重みとバイアス）の初期値をnumpyのrandom.randn関数でランダムに作成。
randn関数は形状に合わせた各要素数のfloat型の配列を返す。（今回は行列とベクトル）
wb_widthは要素の分布の広がり具合。

バックプロパゲーションを実装したニューラルネットワーク 回帰

```
def forward(self, x):  # 順伝播
    self.x = x
    u = np.dot(x, self.w) + self.b
    self.y = u  # 恒等関数
```

forwardメソッドは順伝播のメソッド。
numpyのdot関数で入力と重みの行列積を求め、にバイアスを足しuを求め、活性化関数で出力を計算します。
出力層の活性化関数は恒等関数なのでそのまま出力します。

バックプロパゲーションを実装したニューラルネットワーク 回帰

```
def backward(self, t):  # 逆伝播
    delta = self.y - t

    self.grad_w = np.dot(self.x.T, delta)
    self.grad_b = np.sum(delta, axis=0)

    self.grad_x = np.dot(delta, self.w.T)
```

backwardメソッドは逆伝播のメソッド。
正解データ t を引数として受け取り**誤差を求めdelta**に代入します。
deltaを用いて
重みの勾配：grad_w
バイアスの勾配：grad_b
出力層の入力の勾配：grad_x
を求めます。

バックプロパゲーションを実装したニューラルネットワーク 回帰

```
    def update(self, eta):  # 重みとバイアスの更新
        self.w -= eta * self.grad_w
        self.b -= eta * self.grad_b
```

upddateメソッドは重みとバイアスの更新用のメソッドです。
学習率：eta
重みの勾配：grad_w
バイアスの勾配：grad_b
それぞれの勾配に学習率をかけて更新量とし現在の値から引く事で更新します

バックプロパゲーションを実装したニューラルネットワーク 回帰

```
# -- 各層の初期化 --
middle_layer = MiddleLayer(n_in, n_mid)
output_layer = OutputLayer(n_mid, n_out)

#クラスを使用して中間層を増やす事もできる
middle_layer = MiddleLayer(n_in, n_mid)
middle_layer2 = MiddleLayer(n_in2, n_mid2)
middle_layer3 = MiddleLayer(n_in3, n_mid3)
output_layer = OutputLayer(n_mid, n_out)
```

#クラスを使用して中間層を増やす事で４層以上（ディープラーニング）の実装も可能です
（例）同じMiddleLayerクラスを使用
middle_layer = MiddleLayer(n_in, n_mid)
middle_layer2 = MiddleLayer(n_in2, n_mid2)
middle_layer3 = MiddleLayer(n_in3, n_mid3)
output_layer = OutputLayer(n_mid, n_out)

バックプロパゲーションを実装したニューラルネットワーク 回帰

バックプロパゲーションの実装

バックプロパゲーションを実装したニューラルネットワーク 回帰

```
# -- 学習 --
for i in range(epoch):

    # インデックスをシャッフル
    index_random = np.arange(n_data)
    np.random.shuffle(index_random)

    # 結果の表示用
    total_error = 0
    plot_x = []
    plot_y = []

    for idx in index_random:

        x = input_data[idx:idx+1]  # 入力
        t = correct_data[idx:idx+1]  # 正解

        # 順伝播
        middle_layer.forward(x.reshape(1, 1))  # 入力を行列に変換
        output_layer.forward(middle_layer.y)

        # 逆伝播
        output_layer.backward(t.reshape(1, 1))  # 正解を行列に変換
        middle_layer.backward(output_layer.grad_x)

        # 重みとバイアスの更新
        middle_layer.update(eta)
        output_layer.update(eta)

        if i%interval == 0:

            y = output_layer.y.reshape(-1)  # 行列をベクトルに戻す

            # 誤差の計算
            total_error += 1.0/2.0*np.sum(np.square(y - t))  # 二乗和誤差

            # 出力の記録
            plot_x.append(x)
            plot_y.append(y)

    if i%interval == 0:

        # 出力のグラフ表示
        plt.plot(input_data, correct_data, linestyle="dashed")
        plt.scatter(plot_x, plot_y, marker="+")
        plt.show()

        # エポック数と誤差の表示
        print("Epoch:" + str(i) + "/" + str(epoch), "Error:" + str(total_error/n_data))
```

バックプロパゲーションを実装したニューラルネットワーク 回帰

```
# -- 学習 --
for i in range(epoch):

    # インデックスをシャッフル
    index_random = np.arange(n_data)
    np.random.shuffle(index_random)
```

学習はepoch数だけ繰り返されます。
最適化アルゴリズムの確率的勾配降下法に基づきepochごとに配列をランダムにシャッフルします。
numpyのshuffleメソッドを使用します。

バックプロパゲーションを実装したニューラルネットワーク 回帰

```
    # 結果の表示用
    total_error = 0
    plot_x = []
    plot_y = []
```

エラー総数の設定（仮引数0）しtotal_errorに代入。
x軸とy軸の空のリストを作成

```
    for idx in index_random:

        x = input_data[idx:idx+1]  # 入力
        t = correct_data[idx:idx+1]  # 正解
```

入力データと正解データをスライスで１つづつ増やしながら学習を繰り返す。

バックプロパゲーションを実装したニューラルネットワーク 回帰

```
# 順伝播
middle_layer.forward(x.reshape(1, 1))  # 入力を行列に変換
output_layer.forward(middle_layer.y)

# 逆伝播
output_layer.backward(t.reshape(1, 1))  # 正解を行列に変換
middle_layer.backward(output_layer.grad_x)
```

順伝播では入力を行列にreshapeメソッドで変換
forwardメソッドでデータを渡します。

逆伝播では正解データをreshapeメソッドで行列に変換
backwardメソッドで逆伝播にデータを渡します。

バックプロパゲーションを実装したニューラルネットワーク 回帰

```
        # 重みとバイアスの更新
        middle_layer.update(eta)
        output_layer.update(eta)
```

学習率に応じて重みとバイアスを更新します。

バックプロパゲーションを実装したニューラルネットワーク 回帰

```
if i%interval == 0:

    y = output_layer.y.reshape(-1)  # 行列をベクトルに戻す

    # 誤差の計算
    total_error += 1.0/2.0*np.sum(np.square(y - t))  # 二乗和誤差

    # 出力の記録
    plot_x.append(x)
    plot_y.append(y)
```

intervalで設定した回数でグラフ表示する設定をします。
reshapeメソッドでベクトルに戻し二乗和誤差で誤差を求めた数値をx軸とy軸に記録していきます

バックプロパゲーションを実装したニューラルネットワーク 回帰

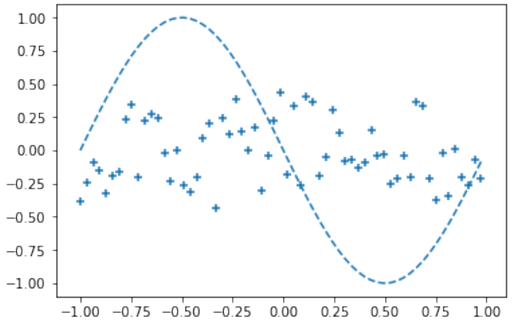
```
if i%interval == 0:

    # 出力のグラフ表示
    plt.plot(input_data, correct_data, linestyle="dashed")
    plt.scatter(plot_x, plot_y, marker="+")
    plt.show()

    # エポック数と誤差の表示
    print("Epoch:" + str(i) + "/" + str(epoch), "Error:" + str(total_error/n_data)
```

グラフの表示とepoch数、誤差の表示をします

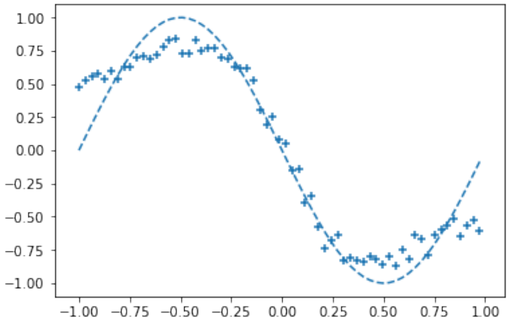
バックプロパゲーションを実装したニューラルネットワーク 回帰

学習の実行

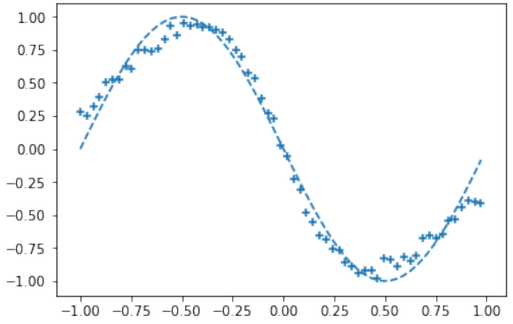
バックプロパゲーションを実装したニューラルネットワーク 回帰



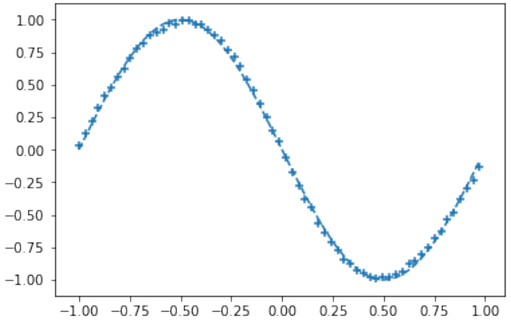
Epoch:0/2001 Error:0.2656120039625669



Epoch:100/2001 Error:0.01653801274242514



Epoch:400/2001 Error:0.005595691506846817



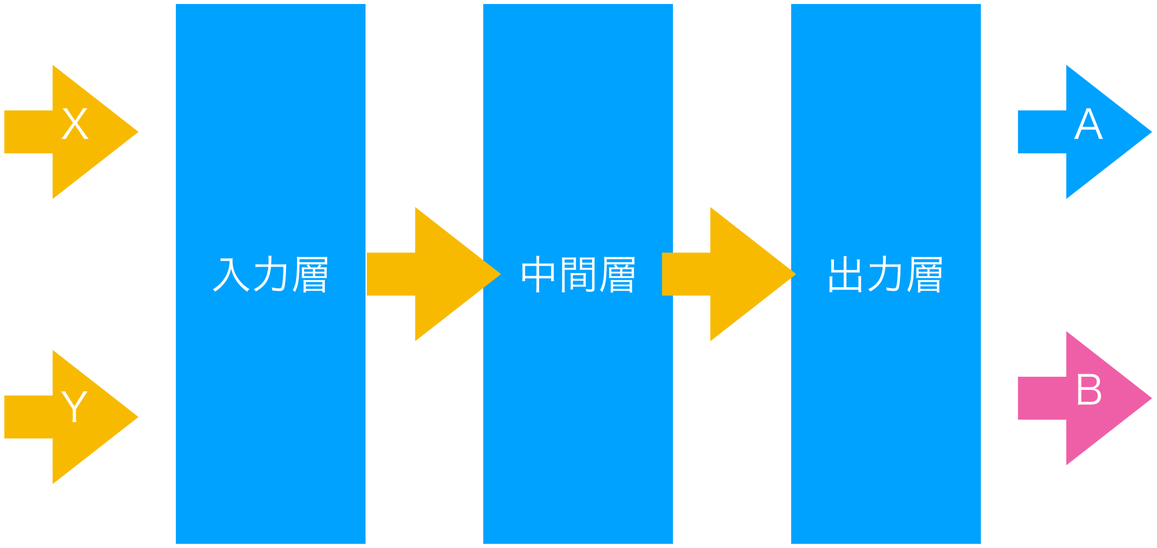
Epoch:1000/2001 Error:0.00028009429927842387

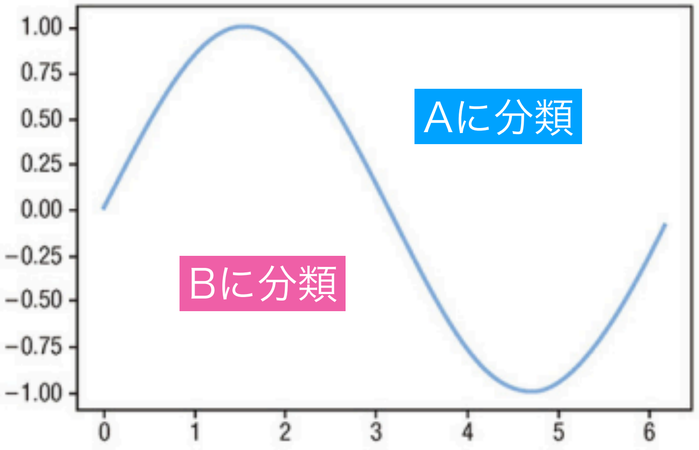
バックプロパゲーション
を実装した
ニューラルネットワーク
分類

バックプロパゲーションを実装したニューラルネットワーク 分類

シンプルなニューラルネットワークにバックプロパゲーションを実装し、ネットワークが学習するプログラムを確認します。
ディープラーニングライブラリーの活用などはせずにPythonのみで実装します。

x、y座標の入力がsin波の上に来るか下に来るかで分類
誤差を逆伝播させ重みとバイアスを調整する事で分類させる事ができます。
正解は[0, 1] または [1, 0]のone-hot表現となります。



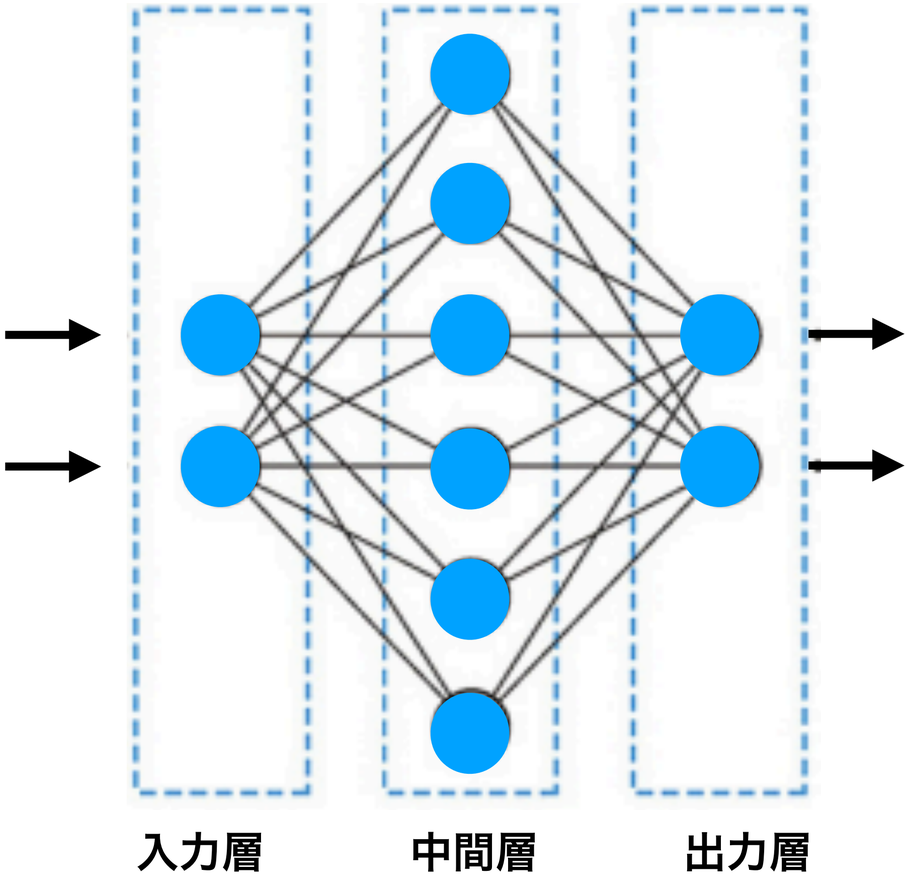


バックプロパゲーションを実装したニューラルネットワーク 分類

作成するニューラルネットワーク
入力層＝２
中間層＝６
出力層＝２
の３層のシンプルなニューラルネットワーク

中間層の活性化関数：シグモイド関数
出力層の活性化関数：ソフトマックス関数
損失関数：交差エントロピー誤差
最適化アルゴリズム：確率的勾配降下法
バッチサイズ：１（オンライン学習＝逐次学習）

今回は訓練データのみ。テストデータはなし。



バックプロパゲーションを実装したニューラルネットワーク 分類

データおよび入力の実装

バックプロパゲーションを実装したニューラルネットワーク 分類

```
%matplotlib inline

import numpy as np
import matplotlib.pyplot as plt

# -- 座標 --
X = np.arange(-1.0, 1.1, 0.1)
Y = np.arange(-1.0, 1.1, 0.1)

# -- 入力、正解データを作成 --
input_data = []
correct_data = []
for x in X:
    for y in Y:
        input_data.append([x, y])
        if y < np.sin(np.pi * x):  # y座標がsinカーブよりも下であれば
            correct_data.append([0, 1])  # 下の領域
        else:
            correct_data.append([1, 0])  # 上の領域

n_data = len(correct_data)  # データ数

input_data = np.array(input_data)
correct_data = np.array(correct_data)
```

バックプロパゲーションを実装したニューラルネットワーク 分類

```
%matplotlib inline

import numpy as np
import matplotlib.pyplot as plt
```

numpy、matpolotlibなど必要なモジュールのインポート。
配列作成やグラフ表示用です。
%matplotlib inlineはJupyternotebook、Google Colab使用の際必要です。

バックプロパゲーションを実装したニューラルネットワーク 分類

```
# -- 座標 --
X = np.arange(-1.0, 1.1, 0.1)
Y = np.arange(-1.0, 1.1, 0.1)
```

numpyのarange関数を使用して-1.0~1.1まで0.1ステップの配列を作成します。
XとYそれぞれの座標用に作成します。

バックプロパゲーションを実装したニューラルネットワーク 分類

```
# -- 入力、正解データを作成 --
input_data = []
correct_data = []
for x in X:
    for y in Y:
        input_data.append([x, y])
        if y < np.sin(np.pi * x):  # y座標がsinカーブよりも下であれば
            correct_data.append([0, 1])  # 下の領域
        else:
            correct_data.append([1, 0])  # 上の領域

n_data = len(correct_data)  # データ数

input_data = np.array(input_data)
correct_data = np.array(correct_data)
```

入力データと正解データ格納用の空のリストを作成
for文で入力データと正解データを作成
正解データはsinカーブの下[0,1]か上[1,0]で分類

バックプロパゲーションを実装したニューラルネットワーク 分類

```
# -- 各設定値 --
n_in = 2  # 入力層のニューロン数
n_mid = 6  # 中間層のニューロン数
n_out = 2  # 出力層のニューロン数

wb_width = 0.01  # 重みとバイアスの広がり具合
eta = 0.1  # 学習係数
epoch = 101
interval = 10  # 経過の表示間隔
```

ニューロン数の設定です。
入力層 x 2
中間層 x 6
出力層 x 2

wb_widthは分布の広がり具合です。
etaは学習率（学習係数）
epochで学習回数を設定
intervalで学習結果のグラフ表示の間隔を設定します。

バックプロパゲーションを実装したニューラルネットワーク 分類

中間層の実装

バックプロパゲーションを実装したニューラルネットワーク 分類

```
# -- 中間層 --
class MiddleLayer:
    def __init__(self, n_upper, n):  # 初期設定
        self.w = wb_width * np.random.randn(n_upper, n)  # 重み（行列）
        self.b = wb_width * np.random.randn(n)  # バイアス（ベクトル）

    def forward(self, x):  # 順伝播
        self.x = x
        u = np.dot(x, self.w) + self.b
        self.y = 1/(1+np.exp(-u))  # シグモイド関数

    def backward(self, grad_y):  # 逆伝播
        delta = grad_y * (1-self.y)*self.y  # シグモイド関数の微分

        self.grad_w = np.dot(self.x.T, delta)
        self.grad_b = np.sum(delta, axis=0)

        self.grad_x = np.dot(delta, self.w.T)

    def update(self, eta):  # 重みとバイアスの更新
        self.w -= eta * self.grad_w
        self.b -= eta * self.grad_b
```

回帰と同じプログラムです

バックプロパゲーションを実装したニューラルネットワーク 分類

```
class MiddleLayer:
    def __init__(self, n_upper, n):  # 初期設定
        self.w = wb_width * np.random.randn(n_upper, n)  # 重み（行列）
        self.b = wb_width * np.random.randn(n)  # バイアス（ベクトル）
```

コンストラクタ（__init__）でオブジェクトの初期設定を行う。
上の層（入力層または中間層）のニューロン数（n_upper）とこの層のニューロン数（n）を引数として受け取る。
重みはn_upper x nの行列
バイアスは要素数 n のベクトル
各要素（重みとバイアス）の初期値をnumpyのrandom.randn関数でランダムに作成。
randn関数は形状に合わせた各要素数のfloat型の配列を返す。（今回は行列とベクトル）
wb_widthは要素の分布の広がり具合。

バックプロパゲーションを実装したニューラルネットワーク 分類

```
def forward(self, x):  # 順伝播
    self.x = x
    u = np.dot(x, self.w) + self.b
    self.y = 1/(1+np.exp(-u))  # シグモイド関数
```

forwardメソッドは順伝播のメソッド。
numpyのdot関数で入力と重みの行列積を求め、にバイアスを足しuを求め、活性化関数で出力を計算します。
中間層の活性化関数はシグモイド関数です。

バックプロパゲーションを実装したニューラルネットワーク 分類

```
def backward(self, grad_y):  # 逆伝播
    delta = grad_y * (1-self.y)*self.y  # シグモイド関数の微分

    self.grad_w = np.dot(self.x.T, delta)
    self.grad_b = np.sum(delta, axis=0)

    self.grad_x = np.dot(delta, self.w.T)
```

backwardメソッドは逆伝播のメソッド。
deltaを求める計算はシグモイド関数の微分になるので
中間層の出力の勾配x（１ー中間層の出力）x中間層の出力
になります。
deltaを用いて
重みの勾配：grad_w
バイアスの勾配：grad_b
中間層の入力の勾配：grad_x
を求めます。

バックプロパゲーションを実装したニューラルネットワーク 分類

```
def update(self, eta):  # 重みとバイアスの更新
    self.w -= eta * self.grad_w
    self.b -= eta * self.grad_b
```

upddateメソッドは重みとバイアスの更新用のメソッドです。
学習率：eta
重みの勾配：grad_w
バイアスの勾配：grad_b
それぞれの勾配に学習率をかけて更新量とし現在の値から引く事で更新します

バックプロパゲーションを実装したニューラルネットワーク 分類

```
# -- 各層の初期化 --
middle_layer = MiddleLayer(n_in, n_mid)
output_layer = OutputLayer(n_mid, n_out)

#クラスを使用して中間層を増やす事もできる
middle_layer = MiddleLayer(n_in, n_mid)
middle_layer2 = MiddleLayer(n_in2, n_mid2)
middle_layer3 = MiddleLayer(n_in3, n_mid3)
output_layer = OutputLayer(n_mid, n_out)
```

#クラスを使用して中間層を増やす事で４層以上（ディープラーニング）の実装も可能です
（例）同じMiddleLayerクラスを使用
middle_layer = MiddleLayer(n_in, n_mid)
middle_layer2 = MiddleLayer(n_in2, n_mid2)
middle_layer3 = MiddleLayer(n_in3, n_mid3)
output_layer = OutputLayer(n_mid, n_out)