

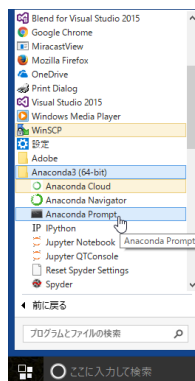
最も簡単なニューラルネットワーク その2

1 目的

- 前回入力したプログラム内部での変数の変化を確認し、動作の仕組みを確認します。
- `chainer.Variable`、`chainer.links`、`chainer.functions`、`chainer.optimizers` の使い方を確認します。

2 準備

2.1 実行環境の選択



Anaconda Prompt を選択します。

前回作った環境に入ります。

```
activate chenv  
cd Lesson
```

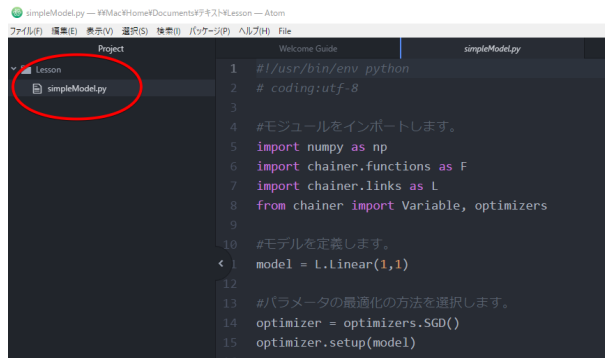
2.2 授業関連ファイルの取得

授業の関連ファイルをダウンロードします。

```
git clone https://github.com/k1nk/lesson.git
```

lesson フォルダが作成されます。その下に関連ファイルがあります。

2.3 ATOM の起動



ATOM を利用している場合には、画面左の Project の中に Lesson フォルダとそのフォルダに含まれるファイルが表示されます。ファイルを選択すると、ファイルの内容の確認、修正ができます。

3 サンプルコード

以下は、前回入力したプログラムです。

リスト 2.1 lesson/ch2/simpleModel.py

```
1  #!/usr/bin/env python
2  # coding:utf-8
3
4  #モジュールをインポートします。
5  import numpy as np
6  import chainer.functions as F
7  import chainer.links as L
8  from chainer import Variable, optimizers
9
10 #モデルを定義します。
11 model = L.Linear(1,1)
12
13 #パラメータの最適化の方法を選択します。
14 optimizer = optimizers.SGD()
15 optimizer.setup(model)
16
17 #学習の回数を設定します。
18 times = 50
19
20 #入力データ
21 #x = Variable(np.array([[1]], dtype=np.float32))
22 x = Variable(np.array([[1], [2], [7]], dtype=np.float32))
```

```

23
24 #教師データ
25 #t = Variable(np.array([[2]], dtype=np.float32))
26 t = Variable(np.array([[2],[4],[14]], dtype=np.float32))
27
28 #学習を行います。
29 for i in range(0,times):
30     #モデルの勾配データを初期化します。
31     model.cleargrads()
32     #optimizer.zero_grads()
33
34     #予測します。
35     y = model(x)
36
37     #モデルの出力を表示します。
38     print(y.data)
39
40     #予測と答えとの誤差を計算します。
41     loss = F.mean_squared_error(y,t)
42
43     #誤差逆伝搬を行い、勾配を計算します。
44     loss.backward()
45
46     #パラメータの更新を行います。
47     optimizer.update()
48
49 #学習の結果得られたパラメータに基づいて予測を行います。
50 print "result"
51 x = Variable(np.array([[3],[4],[5]], dtype=np.float32))
52 y = model(x)
53 print(y.data)

```

4 プログラムの動作の仕組み

4.1 変数の作成

Chainer では、外部からの入力データや正解データを扱うのに `chainer.Variable` を使います。プログラムの最初で、

```

8 from chainer import Variable, optimizers

```

として、インポートしているので、プログラムの中では、「chainer.Variable」を単に「Variable」として参照できます。

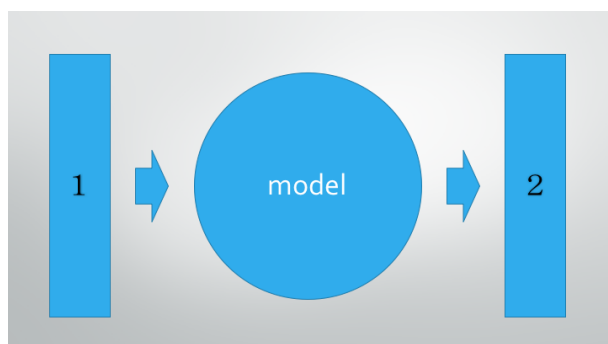
入力データや教師データは、以下のようにして作成できます。

#入力データ

```
x = Variable(np.array([[1],[2],[7]], dtype=np.float32))
```

#教師データ

```
t = Variable(np.array([[2],[4],[14]], dtype=np.float32))
```



この例では、入力が [1] のときの答えは [2]、入力が [2] のときの答えは [4]、入力が [7] のときの答えは [14] という3つのバッチデータを与えています。この例では入力と答えはサイズが1のベクトルになっています。

chainer.Variable が通常のいわゆる変数と異なる点は、変数の値 (data) だけでなく、勾配 (grad) を保持しているということです。

chainer.Variable の詳細

<https://docs.chainer.org/en/stable/reference/core/generated/chainer.Variable.html#chainer.Variable>

4.2 モデルの作成

#モデルを定義します。

```
model = L.Linear(1,1)
```

プログラムの最初で、

```
import chainer.links as L
```

として、インポートしているので、プログラムの中では、「chainer.links」を単に「L」として参照できます。

「chainer.links.Linear」は、

$$y = Wx + b \quad (1)$$

というレイヤーを表します。ここで W はウエイトを b はバイアスを表します。x が入力ベクトル、y が出力ベクトルです。入力のすべての要素（ノード）が出力のすべての要素（ノード）にウエイト W を通じて繋がっているため、全結合層 (fully-connected layer) とも言います。

model = L.Linear(<入力ベクトルのサイズ>, <出力ベクトルのサイズ>) としてモデルを作成します。この例では、入力・出力ともに、サイズが1のベクトルとなっています。そのため、先の入力データも [1] や [2] というサイズが1のベクトルで作成しています。

後に、`optimizer.update()` を使って、`W` や `b` のパラメータを更新することにより、このモデルを最適化していきます。`W` や `b` のようなモデルのパラメータも `Variable` と同様に、その値である `data` だけでなく、勾配 `grad` を保持しています。

`W` は平均 0、標準偏差

$$\sqrt{\frac{1}{\text{入力ベクトルのサイズ}}} \quad (2)$$

の正規分布に従った値で初期化されます。`b` はゼロで初期化されます。この値を初期値としてパラメータの更新を行っていきます。

「`chainer.links`」は、`W` や `b` のような更新可能なパラメータを持った関数（レイヤー）の集まりです。`chainer.links.Linear` 以外にも、よく使うレイヤーが `chainer.links` に多く定義されています。これらの定義されたレイヤーと使うことにより、モデルの定義が簡単になります。

`chainer.links.Linear` の詳細

<https://docs.chainer.org/en/stable/reference/generated/chainer.links.Linear.html>

`chainer.links` の詳細

<https://docs.chainer.org/en/stable/reference/links.html#module-chainer.links>

```
13 #パラメータの最適化の方法を選択します。
14 optimizer = optimizers.SGD()
15 optimizer.setup(model)
```

モデルのパラメータを最適化する方法を選択し、作成したモデルと関連づけておきます。

4.3 学習の開始

```
28 #学習を行います。
29 for i in range(0,times):
```

パラメータの更新を繰り返すことにより、学習を行います。この例では 50 回行います。

勾配の初期化 → 予測・誤差の計算 → 勾配の計算 → パラメータの更新

というステップを繰り返すことにより、誤差が小さくなるようにパラメータを更新していきます。

4.3.1 勾配の初期化

```
30 #モデルの勾配データを初期化します。
31 model.cleargrads()
```

Chainer では、勾配を計算して、アップデートする際に、以前の値に上書きされるのではなく、蓄積される仕様になっています。そのために、勾配の計算を行う前に、勾配の値を初期化しておく必要があります。

4.3.2 予測・誤差の計算

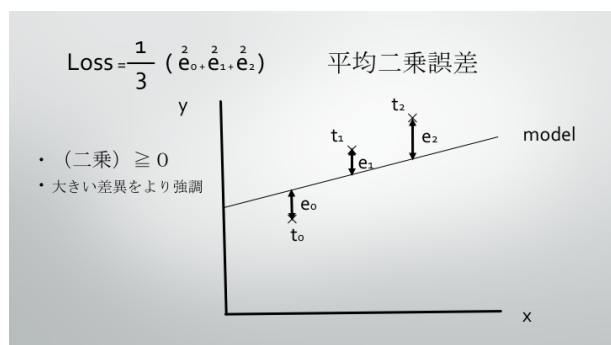
```
34 #予測します。
35 y = model(x)
```

```

36
37 #モデルの出力を表示します。
38 print(y.data)
39
40 #予測と答えとの誤差を計算します。
41 loss = F.mean_squared_error(y,t)

```

現在のモデルのパラメータの値を使って、入力データから出力データを予測します。そして、予測と答えとの誤差を計算します。誤差の計算方法として、ここでは平均二乗誤差を用いています。



平均二乗誤差は、二乗して誤差を求めるので、差がプラスであってもマイナスであっても、誤差は0以上となります。また、二乗して誤差を求めるので、より大きい誤差が強調されます。

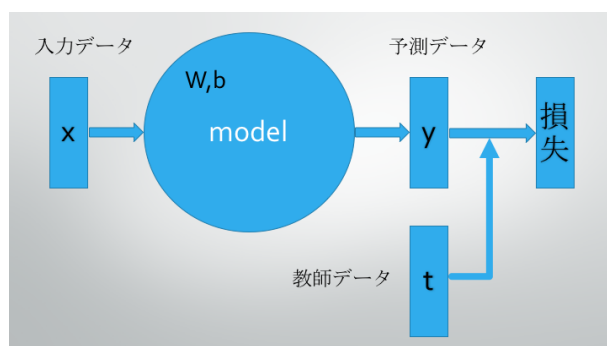
平均二乗誤差は、Chainer では、`chainer.functions.mean_squared_error` で定義されています。プログラムの最初で、

```

6 import chainer.functions as F

```

として、インポートしているので、プログラムの中では、「`chainer.functions`」を単に「`F`」として参照できます。`chainer.functions` は、更新可能なパラメータを持たない関数の集まりです。



ここで loss を定義することにより、入力データから loss までが、1つのチェーンのようにつながります。

`chainer.functions.mean_squared_error` の詳細
https://docs.chainer.org/en/stable/reference/generated/chainer.functions.mean_squared_error.html

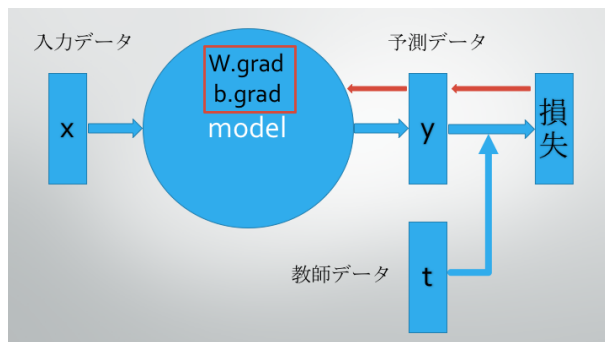
4.3.3 勾配の計算

43 **#誤差逆伝搬を行い、勾配を計算します。**

44 `loss.backward()`

先に求めた、loss を起点として、パラメータの勾配を計算します。パラメータの勾配とは、他の値は変化させずに、それぞれのパラメータの値を 1 単位変化させたときに、loss がどの程度増減するかを表す値です。パラメータの勾配を求める際に、予測を行ったときとは逆方向に計算を行っていきます。これを誤差逆伝搬法 (Backpropagation) と言います。

これにより、モデルのパラメータの勾配である、W.grad,b.grad が求められます。



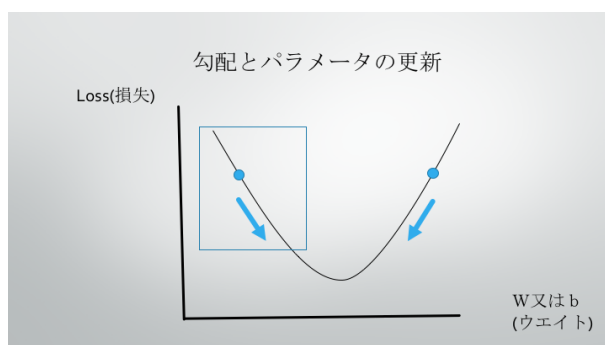
loss を起点として、パラメータの勾配を計算します。これにより、モデルのパラメータの勾配である、W.grad,b.grad が求められます。

4.3.4 パラメータの更新

46 **#パラメータの更新を行います。**

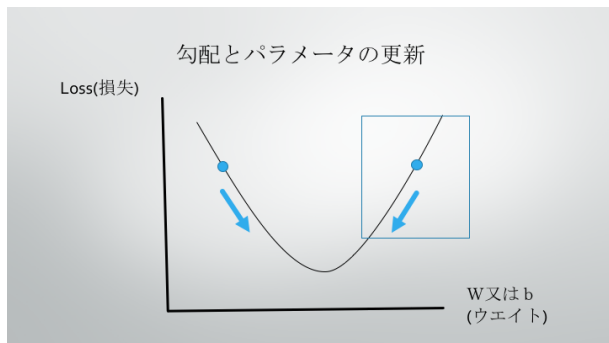
47 `optimizer.update()`

パラメータの勾配とは、他の値は変化させずに、それぞれのパラメータの値を 1 単位変化させたときに、loss がどの程度増減するかを表す値でした。グラフのように、パラメータの勾配がマイナスの場合、パラメータの値を増やしたほうが、loss の値が少なくなることになります。



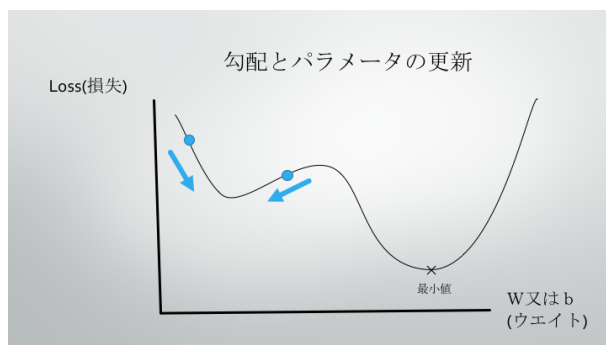
パラメータの勾配がマイナスの場合、パラメータの値を増やしたほうが、loss の値が少なくなる

一方、パラメータの勾配がプラスの場合、パラメータの値を減らしたほうが、loss の値が少なくなります。



パラメータの勾配がマイナスの場合、パラメータの値を増やしたほうが、loss の値が少なくなる

しかし、実際は、パラメータの変化は、loss の変化に対して下の図のようになることもあります。このように loss が局所的には極小となる部分を local minimum(ローカルミニマム)と言います。ローカルミニマムが、全体として最小となる部分(global minimum)になるとは限らないため、基本的には上記の考え方によりながらも、パラメータの値の更新方法としていくつかの方法が提案されています。また、パラメータの初期値がその収束結果に影響を与えることもわかつています。



ローカルミニマムが、全体として最小となる部分になるとは限らない。そのため、パラメータの更新方法にはいくつかの方法がある。

のプログラムでは、上記のパラメータの更新を訓練データごとに繰り返し行う、確率的勾配降下法(stochastic gradient descent, SGD)を用いています。

4.4 予測

モデルのパラメータの更新が終わったところで、学習後のパラメータに基づいて予測を行います。

```
49 #学習の結果得られたパラメータに基づいて予測を行います。
50 print "result"
51 x = Variable(np.array([[3],[4],[5]]), dtype=np.float32))
52 y = model(x)
53 print(y.data)
```

データとして、[3],[4],[5] というバッチデータを、chainer.Variable として作成します。そして、モデルに入力し予測を行います。そして、その予測した結果を表示します。

4.5 動作の確認

4.5.1 勾配の計算

それでは、勾配の計算とパラメータの更新の状態を実際に確認してみましょう。まず、勾配の計算の前後で、実際に勾配の値が変化していることを確認してみましょう。loss.backward() の前後で、W および b の勾配の値を確認します。W の勾配の値は、model.W.grad で確認できます。

```
43 #誤差逆伝搬を行い、勾配を計算します。
44 print "model.W.grad before backward:", model.W.grad
45 loss.backward()
46 print "model.W.grad after backward:", model.W.grad
```

プログラムを修正したら、修正後のプログラムを実行し、結果を確認します。

```
python simpleModel.py

model.W.grad before backward: None
model.W.grad after backward: [[-0.03739944]]
```

model.cleargrads() でモデルの勾配が None に設定されます。したがって、loss.backward() を呼ぶ前は勾配の値は None となります。backward() を呼ぶことにより、勾配に値が設定されていることがわかります。また、この例の場合、勾配の値は負になっています。

4.5.2 パラメータの更新

次に optimizer.update() の前後で、パラメータの更新が行われているか確認してみましょう。モデルのパラメータ W の値は、model.W.data で確認できます。

```
46 #パラメータの更新を行います。
47 print "model.W.data before update:", model.W.data
48 optimizer.update()
49 print "model.W.data after update:", model.W.data
```

プログラムを修正したら、修正後のプログラムを実行し、結果を確認します。

```
python simpleModel.py

model.W.grad before backward: None
model.W.grad after backward: [[-0.03557223]]
model.W.data before update: [[ 1.95189846]]
model.W.data after update: [[ 1.95225418]]
```

この例では、W の勾配は、「-0.03557223」と負になっています。勾配が負であるので、W の値は、update() により、1.95189846 から、1.95225418 へと増加していることがわかります。

バイアスの勾配 `model.b.grad` および、バイアスの値 `model.b.data` についても同様に値を確認してみましょう。

最後に、パラメータの更新によりモデルがどのように変化しているか、確認してみましょう。モデルのグラフを描くプログラムを追加します。

```
8 from chainer import Variable, optimizers
9 import matplotlib.pyplot as plt
10
11 def plotmodel(model):
12     x_for_plot = Variable(np.array([[1],[2],[7]], dtype=np.float32))
13     y_for_plot = model(x_for_plot)
14     plt.plot(x_for_plot.data, y_for_plot.data, "r-")
```

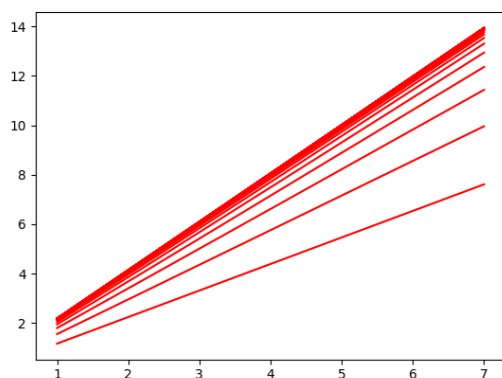
`update` の後に、モデルをプロットします。

```
46 #パラメータの更新を行います。
47 print "model.W.data before update:", model.W.data
48 optimizer.update()
49 print "model.W.data after update:", model.W.data
50 plotmodel(model)
```

最後にグラフを表示します。

```
52 y = model(x)
53 print(y.data)
54 plt.show()
```

プログラムを修正したら、修正後のプログラムを実行し、結果を確認します。



W と b の値が更新されることにより、入力データの倍の値を出力するモデルに近づいている様子か確認できます。