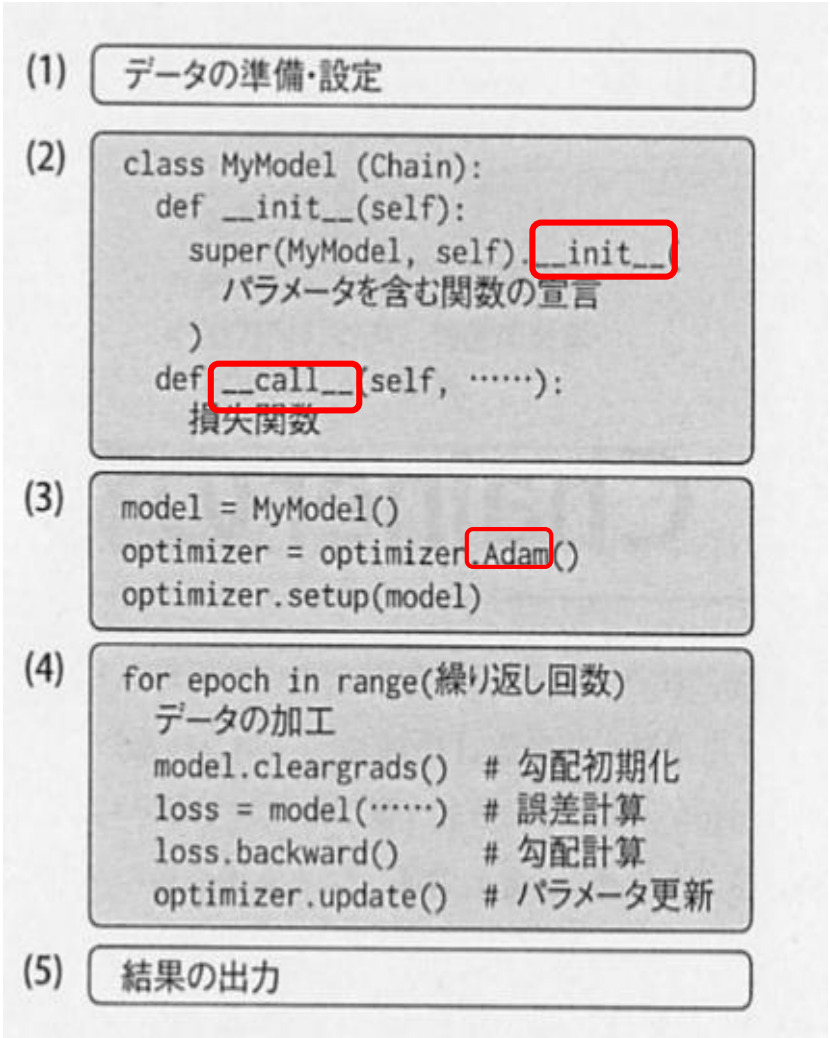
第４章

Chainerの利用例

白　静

4.1　全体図



（1）で学習データの準備

（2）はモデルの記述

__init__ と __call__の部分は必須です。

（3）はモデルと最適化アルゴリズムを設定する部分です。

（4）が学習の部分です。

（5）で結果の出力です。

4.2　Irisデータ

Irisデータ

機械学習でよく用いられるサンプルデータ

150個のデータからなり、各データはアヤメのデータ

各データにはアヤメの花の種類

setosa (0)、versicolor (1)、virgmica (2)に

応じて、その数値がラベルとして与えられている

こからはプログラムfoo.pyなどの形で書いて実行する形を想定しています。

```
> python f00.py
```

iris0.py

```
>>> from sklearn import datasets
>>> iris = datasets.load_iris()
>>> X = iris.data.astype(np.float32)
>>> Y = iris.target
>>> N = Y.size
>>> Y2 = np.zeros(3 * N).reshape(N,3).astype(np.float32)
>>> for i in range(N):
...     Y2[i,Y[i]] = 1.0
...
>>> index = np.arange(N)
>>> xtrain = X[index[index % 2 != 0],:]
>>> ytrain = Y2[index[index % 2 != 0],:]
>>> xtest = X[index[index % 2 == 0],:]
>>> yans = Y[index[index % 2 == 0]]
```

教師信号にあたるytrainの要素は、3次元のベクトル

例:　　クラスが 1

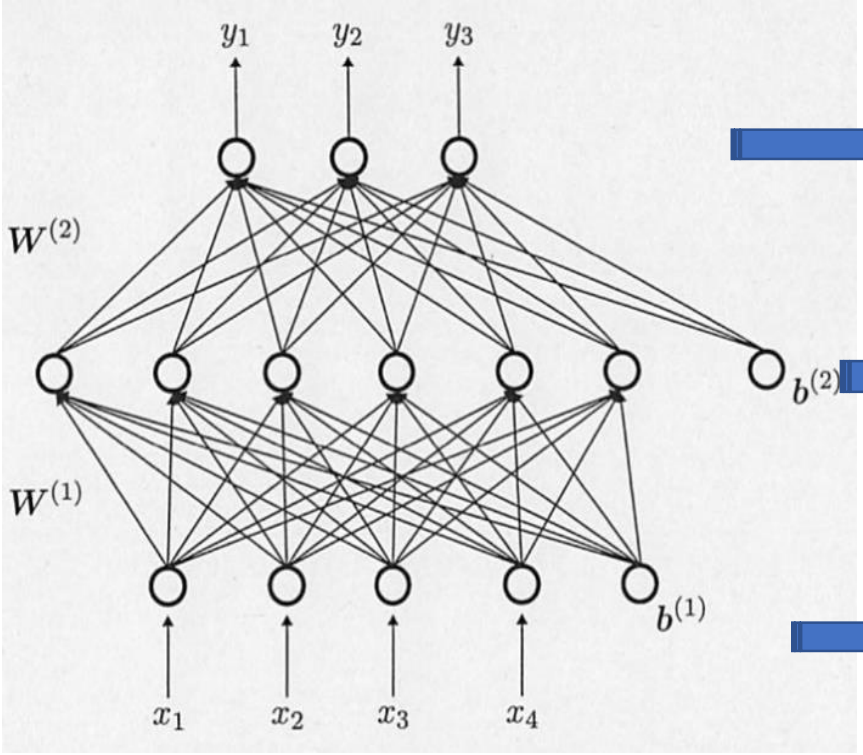
(0.0, 1.0, 0.0)

>>>from sklearn import datasets

Irisデータだけを取り出すのに、本書のサンプルプログラム集の中にiris-x. txt およびiris-y. txtを使って、変数XとYを作る

```
>>> X = np.loadtxt('iris-x.txt').astype(np.float32)
>>> Y = np.loadtxt('iris-y.txt').astype(np.int32)
```

4.3　基本的なプログラム



出力層:　3つのユニット

中間層:　6つのユニット

入力層:　4つのユニット

```
class IrisChain(Chain):
    def __init__(self):
        super(IrisChain, self).__init__(
            l1=L.Linear(4,6),
            l2=L.Linear(6,3),
        )

    def __call__(self,x,y):
        return F.mean_squared_error(self.fwd(x), y)

    def fwd(self,x):
         h1 = F.sigmoid(self.l1(x))
         h2 = self.l2(h1)
         return h2
```

Chainのクラスを設定します。
IrisChainと名付けました

```
model = IrisChain()
optimizer = optimizers.SGD()
optimizer.setup(model)

# Learn

for i in range(10000):
    x = Variable(xtrain)
    y = Variable(ytrain)
#    model.zerograds()
    model.cleargrads()
    loss = model(x,y)
    loss.backward()
    optimizer.update()
```

パラメータの学習

```
xt = Variable(xtest)
yy = model.fwd(xt)

ans = yy.data
nrow, ncol = ans.shape
ok = 0
for i in range(nrow):
    cls = np.argmax(ans[i,:])
    print ans[i,:], cls
    if cls == yans[i]:
        ok += 1

print ok, "/", nrow, " = ", (ok * 1.0)/nrow
```

```
68 / 75  =  0.906666666667
```

4.4　ミニバッチ

```
↓
n = 75                      #データのサイズ↓
bs = 25                     #バッチのサイズ↓
for j in range(5000):       #全体データの学習回数↓
    sffindx = np.random.permutation(n)↓
    for i in range(0, n, bs):↓
        x = Variable(xtrain[sffindx[i:(i+bs) if (i+bs) < n else n]])↓
        y = Variable(ytrain[sffindx[i:(i+bs) if (i+bs) < n else n]])↓
        model.cleargrads()↓
        loss = model(x,y)↓
        loss.backward()↓
        optimizer.update()↓
↓
```

バッチ処理：1回パラメータ更新ごとに75個の訓練データ全てを使う

ミニバッチ：1回のパラメータ更新にはランダムに取り出した25個の訓練データを使う

```
72 / 75   =   0.96
```

4.5　誤差の累積

Chainerでは累積された誤差の総計から勾配を求め

結果的には、各データに対して、勾配を求め、それら勾配の総和を取ったものを累積された誤差の勾配としている

```
n = 75                  #データのセイズ↓
bs = 25                 #累積するセイズ↓
for j in range(2000):   #全体データの学習回数↓
    accum_loss = None     ↓
    sffindx = np.random.permutation(n)↓
    for i in range(0, n, bs):↓
        x = Variable(xtrain[sffindx[i:(i+bs) if (i+bs) < n else n]])↓
        y = Variable(ytrain[sffindx[i:(i+bs) if (i+bs) < n else n]])↓
        model.cleargrads()↓
        loss = model(x,y)↓
        accum_loss = loss if accum_loss is None else accum_loss + loss
    loss.backward()↓
    optimizer.update()↓
```

```
51 / 75  =  0.68
```

4.6 softmax

```
>>> ans[0]
array([1.01875758, -0.02290851, 0.00886351], dtype=float32)
```

$$\mathrm{Softmax}(x_i)=\frac{\exp(x_i)}{\sum_j \exp(x_j)}$$

i：次元の番目

x_i：第i次元の値

$\mathrm{softmax}(x_i)$：確率値に変換する関数

```
class IrisChain(Chain):
      def  __init__(self):
                ...
      def  __call__(self,x,y):
                ...
      def fwd(self,x):
                h1 = F.sigmoid(self.l1(x))
                h2 = self.l2(h1)
                h3 = F.softmax(h2)

                return h3
```

softm関数はfunctions内で提供されているので、出力層からの出力に活性化関数としてこの関数を呼び出すだけで、出力値を確率値に変換できる

4.7 softmax cross entropy

```
X = iris.data.astype(np.float32)↓
Y = iris.target.astype(np.int32)↓
N = Y.size↓
index = np.arange(N)↓
xtrain = X[index[index % 2 != 0],:]↓
ytrain = Y[index[index % 2 != 0]]      #教師信号は整数値
```

```
class IrisChain(Chain):↓
    def __init__(self):↓
        super(IrisChain, self).__init__(↓
            l1=L.Linear(4,6),↓
            l2=L.Linear(6,3),↓
        )↓
        ↓
    def __call__(self,x,y):↓
        return F.softmax_cross_entropy(self.fwd(x), y)
↓
    def fwd(self,x):↓
        h1 = F.sigmoid(self.l1(x))↓
        h2 = self.l2(h1)↓
        return h2↓
```

```
71 / 75   =   0.946666666667
```

- 出力層からの出力にsoftmax関数を利用場合、教師信号はラベルなので、cross entropyで損失を測る
- Chainerには損失関数として

F.softmax_cross_entropyが用意されている

- 教師信号はラベルに対する整数値を指定する

4.8　ロジスティクス回帰

ロジスティック回帰：

- 分類問題を解く機械学習の手法
- NNを利用して実現できる
- 識別先のクラスが2値

  3値以上の場合は**多項ロジスティック回帰**と呼ばれる

識別先のクラスを$C = \{c_1, c_2 \ldots, c_k\}$、入力のベクトルを

$x = (x_1, x_2 \ \ldots, x_n)^t$としたとき、多項ロジスティック回帰のモデルは以下の式で表せる

$$p(c_k \mid x) = \pi(a_k)$$

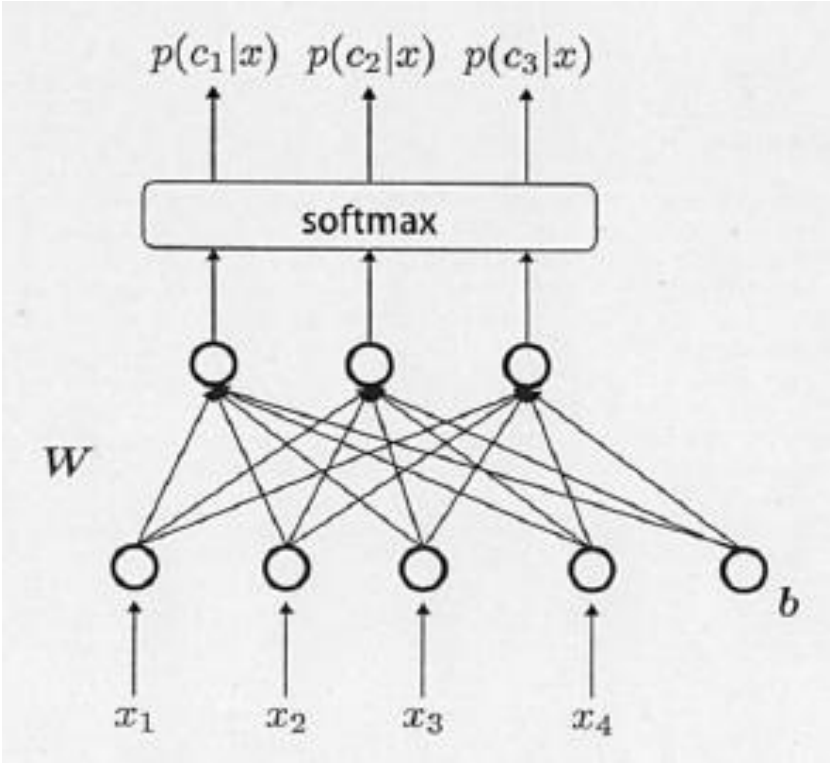
$$a_k = (Wx + b)_k$$

W：K行 n 列の行列

b ：K次元ベクトル

Π：はsoftmax関数

$(Wx + b)_k$：$Wx + b$の k 次元の値




```
class IrisRogi(Chain):
    def __init__(self):
        super(IrisRogi, self).__init__(
            l1=L.Linear(4,3),
        )

    def __call__(self,x,y):
        return F.mean_squared_error(self.fwd(x), y)

    def fwd(self,x):
        return F.softmax(self.l1(x))


model = IrisRogi()
optimizer = optimizers.Adam()
optimizer.setup(model)
```