人工知能概論

12. 深層学習(1)
ニューラルネットワーク

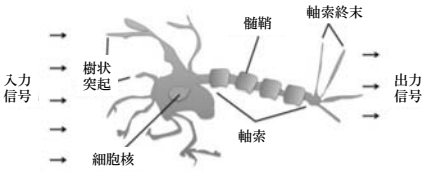
出典: Raschka, S. (2015) *Python Machine Learning*. Packt.

人工ニューロン

マカロフ・ピッツ(McCulloch-Pitts (MCP))のニューロンモデル:
(McCulloch, W.S. and & Pitts W. (1943). *A Logical Calculus of the Ideas Immanent in Nervous Activity.* The bulletin of mathematical biophysics, 5(4). pp. 115–133

ニューロンは脳内で相互接続される神経細胞

マカロフ・ピッツのモデル：神経細胞を「複数入力、二値出力」の単純な論理ゲートとみなす：入力の総和が特定の閾値を超えたら出力信号が生成され、軸索により伝達



パーセプトロン

F. Rosenblatt (1957) *The Perceptron, a Perceiving and Recognizing Automaton*. Cornell Aeronautical Laboratory.

マカロフ・ピッツのモデルに基づくパーセプトロンの学習規則の提案：自動で最適な重み係数を学習

学習の後、サンプルを2クラス分類

パーセプトロンのアルゴリズム

二値分類タスクとして捉える：１と－１の２つのクラス

活性化関数(activation function) $\varphi(z)$:

$z = w_1x_1 + \cdots + w_mx_m = \boldsymbol{w}^T\boldsymbol{x}$

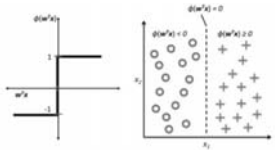
入力変数$\boldsymbol{x}$と重みベクトル $\boldsymbol{w}$ の線形和

($\boldsymbol{x}$はニューロンへの入力、$\boldsymbol{w}$は重み係数、$\boldsymbol{z}$はその総和)

パーセプトロンのアルゴリズムでは(θは閾値)

$$\varphi(z) = \begin{cases} 1 & if\ z \geq \theta \\ -1 & otherwise \end{cases}$$

単位ステップ関数、もしくヘヴィサイド関数



ローゼンブラットの初期のパーセプトロン学習規則

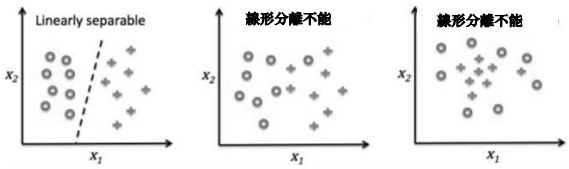
1. 重みを 0 または小さな乱数で初期化
2. 学習サンプル$\boldsymbol{x}^{(i)}$ごとに以下を実行:
   (註: $\boldsymbol{x}^{(i)}$は i 番目 の学習サンプル)
   ① 出力値 $\hat{y}$ を計算
   ② 重みを更新（修正）

- 重みベクトル$\boldsymbol{w}$の各重みw_jは同時に更新： $w_j \coloneqq w_j + \Delta w_j$
- 重みの更新に用いられるΔw_j の値: $\Delta w_j = \eta(y^{(i)} - \hat{y}^{(i)})x_j^{(i)}$
- ηは学習率(0.0から1.0の間の定数)、$y^{(i)}$はi番目の学習サンプルに対する本当のクラスラベル、$\hat{y}^{(i)}$はクラスラベルの予測値

予測が「正解」なら重みは更新(修正)されない。『不正解』のときだけ、入力値に比例した値が加算される

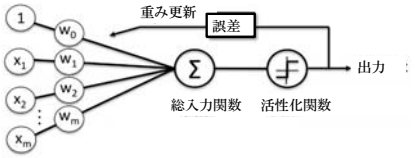
パーセプトロンの学習

- ーセプトロンが収束する ⇔ 二つのクラスが**線形分離可能**、かつ学習率が十分小さい
- 線形分離不能な場合に備えて、繰り返し回数の上限を設定、もしくは誤分類の許容範囲を設定しておく



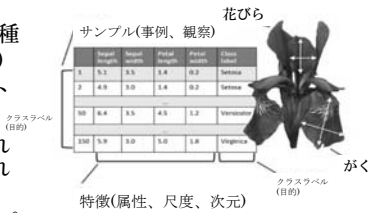
パーセプトロンの概念

パーセプトロンの入力$\boldsymbol{x}$と重み$\boldsymbol{w}$の線形和で総入力を計算
それが活性化関数に渡され、二値出力(-1か1)を生成する
この出力が予測誤りと重み更新の計算に使われる



使用する学習データ

- FisherのIris(アヤメ）データ: setosa, versicolor, virginicaの3種類のアヤメのデータ(各50例)
- データの記述に行列やベクトルを用いる
- 特徴行列において、それぞれのサンプルは行に、それぞれの特徴は個々の列に書く
- Irisデータは4特徴、150サンプルからなるので、150×4行列 $\boldsymbol{X} \in \boldsymbol{R}^{150\times4}$ で表す



Pythonによる学習データの作成

```
from sklearn import datasets
import numpy as np
iris = datasets.load_iris()       # Irisデータのロード(scikit-learn (=sklearn)にある)
X = iris.data[:, [2, 3]]                  # 2つの特徴量を抽出
y = iris.target                   # Irisの種別
# 学習結果のモデルを未知データにより評価するため、データを学習用と評価用に分ける
from sklearn.cross_validation import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=0)
# これにより配列XとyがX評価用30%(test_sizeで指定, 45個)、学習用70%(=105個)に分けられる
from sklearn.preprocessing import StandardScaler
sc = StandardScaler()                     # 標準化クラス(データの『標準化』：平均0, 分散1にデータ変換)
sc.fit(X_train)                           # X_trainを用いて学習データの標準化の準備
X_train_std = sc.transform(X_train)               # X_trainを標準化
X_test_std = sc.transform(X_test)                 # X_testを標準化
```

Pythonでゼロからつくる

```
class Perceptron(object):
  def __init__(self, eta=0.01, n_iter=10):
    self.eta = eta
    self.n_iter = n_iter

  def fit(self, X, y):  # 学習
    self.w_ = np.zeros(1 + X.shape[1])
    # ゼロで初期化
    self.errors_ = []
    for _ in range(self.n_iter):
      errors = 0
      for xi, target in zip(X, y):
        # Xとyそれぞれに対し
        update = self.eta * (target - self.predict(xi))
        # 重み更新
        self.w_[1:] += update * xi
        self.w_[0] += update
        errors += int(update != 0.0)
        # 誤分類の個数の記録
      self.errors_.append(errors)
    return self
```

```
  def net_input(self, X):      # 総入力の計算
      return np.dot(X, self.w_[1:]) + self.w_[0]

  def predict(self, X):        # 予測値の計算
      return np.where(self.net_input(X) >= 0.0, 1, -1)
          # 0.0以上なら1, そうでなければ-1に分類
```

Pythonによる実装（scikit learnモジュール利用)

```
from sklearn.linear_model import Perceptron
ppn = Perceptron(n_iter=40, eta0=0.1, random_state=0)  #　インスタンス作成
ppn.fit(X_train_std, y_train)             # fitメソッドにより学習
# ここで、eta0は学習率、n_iterはエポック数のパラメタ
# random_stateパラメタ：エポックごとのランダムシャッフルの初期値
y_pred = ppn.predict(X_test_std)     # 予測値を求める：predictメソッドによる
print('Misclassified samples: %d' % (y_test != y_pred).sum())
### 実行結果 ####
Misclassified samples: 4                  # 45例中4個の誤分類
```

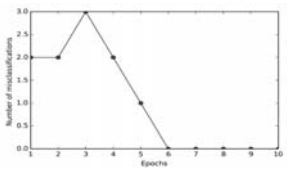
テストデータでは45例中4例を誤分類—8.9%

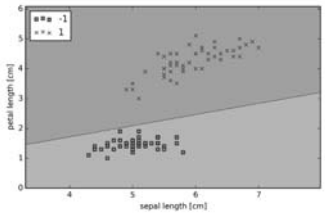
注: 誤分類の代わりに精度(accuracy)を報告するものもある。この例では

1 - 誤分類率 = 0.911　もしくは91.1%

学習過程の視覚化

```
import matplotlib.pyplot as plt
ppn = Perceptron(eta=0.1, n_iter=10)
ppn.fit(X, y)
plt.plot(range(1, len(ppn.errors_) + 1), ppn.errors_, marker='o')
plt.xlabel('Epochs')
plt.ylabel('Number of misclassifications')
plt.tight_layout()
plt.show()
```

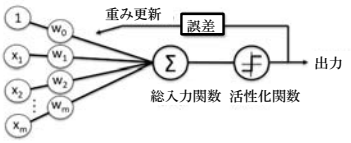




Widrow & Hoffによる改良:ADALINE

- ADAptive LInear NEuron (Adaline). (B. Widrow et al. (1960). Adaptive *"Adaline" neuron using chemical "memistors"*. Number Technical Report 1553-2. Stanford Electron. Labs. Stanford.)
- ADALINE アルゴリズム (Widrow-Hoff 学習アルゴリズム) ：コスト関数の定義と最小化
- ローゼンブラットのパーセプトロンとの違い: 重みの更新が線形ステップ関数ではなく、線形活性化関数$\varphi(z)$による----ADALINEアルゴリズムでは $\varphi(\boldsymbol{w}^T\boldsymbol{x}) = \boldsymbol{w}^T\boldsymbol{x}$
- ただし予測においては量子化関数（線形ステップ関数のようなもの）を用いる

パーセプトロンとADALINEの比較



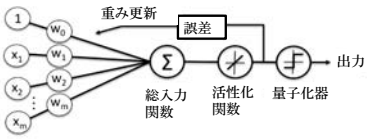
ADALINEの活性化関数

$\varphi(\boldsymbol{w}^T\boldsymbol{x}) = \boldsymbol{w}^T\boldsymbol{x}$

パーセプトロンの活性化関数

$$\varphi(z) = \begin{cases} 1 & if\ z \geq \theta \\ -1 & otherwise \end{cases}$$

$$z = w_1x_1 + \cdots + w_mx_m = \boldsymbol{w}^T\boldsymbol{x}$$



勾配降下法によるコスト関数の最小化

(パーセプトロンと比較した時の) ADALINEの特徴

重みの学習にコスト関数が定義できる

ADALINEの場合：計算値と真のクラスラベルとの間の二乗和誤差(Sum of Squared Errors , SSE) でコスト関数を定義

$$J(\boldsymbol{w}) = \frac{1}{2}\sum_i (y^{(i)} - \varphi(z^{(i)}))^2$$

勾配降下法 (*gradient descent*) によりコスト最小にする重みを見つけられる---$J(\boldsymbol{w})$は微分可能、かつ凸関数

勾配降下法(gradient descent method)

単純ながら強力な最適化

コスト最小となるまで、勾配と学習率によって決まる大きさが決まるステップで「坂を下る」

重みの更新： $\boldsymbol{w} := \boldsymbol{w} + \Delta\boldsymbol{w}$

$\Delta\boldsymbol{w} = -\eta\nabla J(\boldsymbol{w})$

∇ : ハミルトン演算子

$\nabla J(w_1, w_2, \cdots, w_m) \equiv \left(\frac{\partial J}{\partial w_1}, \frac{\partial J}{\partial w_2}, \ldots, \frac{\partial J}{\partial w_m}\right)$

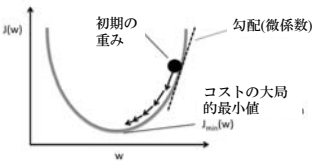
$z = w_1x_1 + \cdots + w_mx_m = \boldsymbol{w}^T\boldsymbol{x}$

$\nabla J(\boldsymbol{w})$ ：コスト関数 $J(\boldsymbol{w})$ の勾配 (w_jを重みとして)

$$\frac{\partial J}{\partial w_j} = -\sum_i (y^{(i)} - \varphi(z^{(i)}))x_j^{(i)}$$

つまり重みの更新値:

$$\Delta w_j = -\eta\frac{\partial J}{\partial w_j} = \eta\sum_i (y^{(i)} - \varphi(z^{(i)}))x_j^{(i)}$$



ADALINEは,バッチで(すべての学習データ用いて)、全ての重み(各w_j)を同時に更新

演習12-1

$$J(\boldsymbol{w}) = \frac{1}{2}\sum_i (y^{(i)} - \varphi(z^{(i)}))^2$$

$$z = w_1x_1 + \cdots + w_mx_m = \boldsymbol{w}^T\boldsymbol{x}$$

$$\varphi(\boldsymbol{w}^T\boldsymbol{x}) = \boldsymbol{w}^T\boldsymbol{x}$$

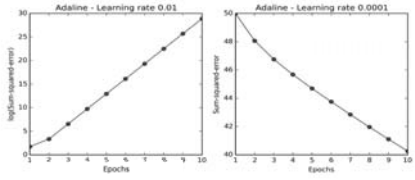
以上から

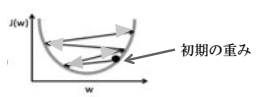
$\frac{\partial J}{\partial w_j} = -\sum_i (y^{(i)} - \varphi(z^{(i)}))x_j^{(i)}$　　となることを示せ.

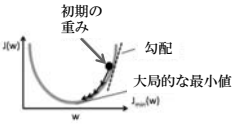
($x_j^{(i)}$、$y^{(i)}$はi番目の学習データで、スカラー定数)

ADALINEの学習過程

- 最適な値への収束には、学習率ηの値を求める実験が必要
- 学習率の値による違いの例：(η=0.01 と η=0.0001)

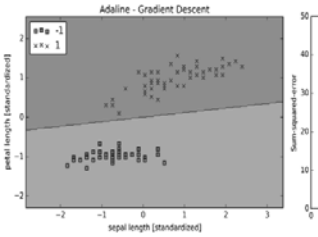


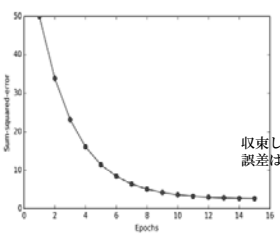




ADALINEの学習過程の視覚化

データを標準化し、学習率η=0.01 とした時の学習過程





収束しているが、
誤差は0にはならない

確率的勾配降下法

膨大なデータがある場合、バッチ勾配降下法ではコストが高い

∵ バッチ式の勾配降下法では、全てのデータを一々評価し直す

$$\Delta w_j = -\eta\frac{\partial J}{\partial w_j} = \eta\sum_i (y^{(i)} - \varphi(z^{(i)}))x_j^{(i)}$$

確率的勾配降下法(stochastic gradient descent,
オンライン勾配降下法とも言う)

学習サンプルごとに段階的に重みを更新

$$\Delta w_j = \eta(y^{(i)} - \varphi(z^{(i)}))x_j^{(i)}$$

確率的勾配降下法の特徴

バッチ式勾配降下法と比べて

- •はるかに収束が速い
- •データにはノイズが多く含まれ、それによって局所的最小値からの脱出が可能になる

注意点：

学習率は固定ではなく、時間とともに減少するようにする

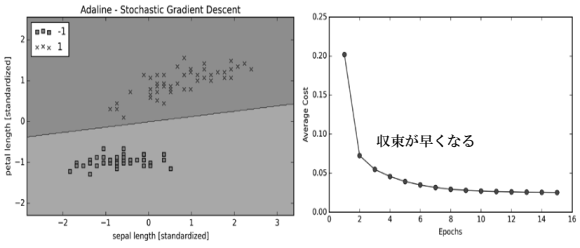
データをランダムに並び替える

ミニバッチ学習： バッチ勾配降下法と確率的勾配降下法の融合

学習データの一部に対しバッチ勾配降下法を適用する

確率的勾配降下法を用いたADALINE

少しの修正により、確率的勾配降下法による学習に変更可能



パーセプトロンからニューラルネットワークへ

パーセプトロンは、線形分離可能なクラス分類を学習できる

しかし、逆に言えば、

パーセプトロン・アルゴリズムは線形分離可能でないデータに対しては収束しない

⇒実際に使用するのは勧められない

ADALINEは「１つの入力層」と「１つの出力層」から構成

単層ネットワーク (出力層のノードが一つだけだから)

層をふやす…多層パーセプトロン、もしくはニューラルネットワーク

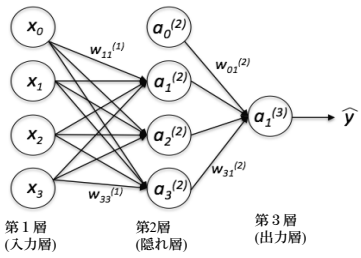
ニューラルネットワークについての学習項目

- 多層ニューラルネットワークの概念の理解
- 画像を分類するためのニューラルネットワークの訓練
- ニューラルネットワークを訓練するための強力なバックプロパゲーション・アルゴリズムの実装

多層ニューラルネットワーク・アーキテクチャ

任意の個数の隠れ層を追加することでより深いネットワークアーキテクチャが作成される

しかし層が増えると、誤差勾配が徐々に小さくなる「勾配消失」問題が起きる



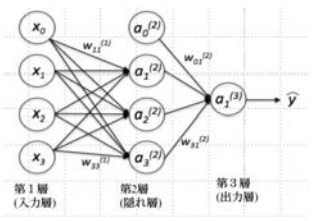
$a_i^{(k)}$: k番目の層のi番目の活性化ユニット

上記の図では、$a_0^{(1)}$と$a_0^{(2)}$はバイアスユニット(1に固定)

入力層(第1層)のユニットの活性化は、入力にバイアスユニットをたしたもの

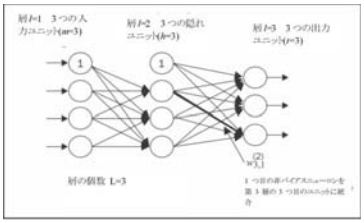
$$a^{(1)} = \begin{bmatrix} a_0^{(1)} \\ a_1^{(1)} \\ \vdots \\ a_m^{(1)} \end{bmatrix} = \begin{bmatrix} 1 \\ x_1^{(i)} \\ \vdots \\ x_m^{(i)} \end{bmatrix}$$

多層ニューラルネットワーク・アーキテクチャ



(入力)３－(隠れ)３－(出力)１の多層パーセプトロン　バイアスユニットはカウント外

「二値分類」なら出力のユニットは１で十分



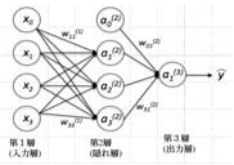
３－３－３の多層パーセプトロン

「多クラス分類」には出力のユニットを増やす

one-hot ベクトル表現---該当するユニットだけ1、他は0として表現

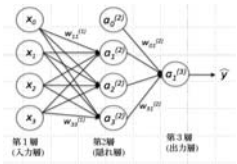
ニューラルネットワークの学習

1. 入力層を出発点とし, 学習データのパターンをネットワーク経由で順方向に伝播(forward propagation)させ、出力を生成
2. ネットワークの出力に基づき、コスト関数を使って誤差を計算。この誤差を最小化することが目的
3. 誤差を逆方向に伝播(backward propagation)させることで、ネットワーク内のそれぞれの重みに対する偏導関数を求め、モデルを更新



出力を生成：フォワードプロパゲーション

1. 入力層を出発点とし, 学習データのパターンをネットワーク経由で順方向に伝播(forward propagation)させ、出力を生成



活性化ユニット $a_1^{(2)}$ (第2層の1番目のユニットの活性)の計算：

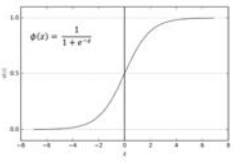
$$a_1^{(2)} = \varphi(z_1^{(2)}) \quad z_1^{(2)} = \sum_{i=0}^{m} a_i^{(1)} w_{1,i}^{(1)}$$

($z_1^{(2)}$は総入力、φは活性化関数)

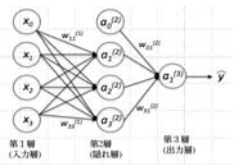
勾配降下法でニューロンの重みの学習を行うには、活性化関数が微分可能であることが必要

画像分類のような複雑なタスクのためには、非線形の活性化関数(例:シグモイド関数)が必要：

$$\varphi(z) = \frac{1}{1+e^{-z}}$$



行列を用いた活性化関数の記述



$\boldsymbol{a}^{(1)}$はm次元の学習データ$\boldsymbol{x}^{(i)}$にバイアスユニットを加えた

(m+1)×1次元の特徴ベクトル

隠れ層のユニット数をhとすると(バイアス除く)、

$\boldsymbol{W}^{(1)}$は $h \times (m+1)$次元の重み行列

これから隠れ層の総入力ベクトル: $z^{(2)} = \boldsymbol{W}^{(1)}\boldsymbol{a}^{(1)}$

隠れ層の活性化ベクトル: $\boldsymbol{a}^{(2)} = \varphi(z^{(2)})$

出力層のユニット数をtとすると(隠れ層のバイアス無しとして)

$\boldsymbol{W}^{(2)}$は $t \times h$ 次元の重み行列

出力層の総入力: $\boldsymbol{z}^{(3)} = \boldsymbol{W}^{(2)}\boldsymbol{a}^{(2)}$

ネットワークの出力： $\boldsymbol{a}^{(3)} = \varphi(\boldsymbol{z}^{(3)})$

ニューラルネットワークの学習：誤差

2. ネットワークの出力に基づき、コスト関数を使って誤差を計算。この誤差を最小化することが目的

コスト関数：ニューラルネットの出力と「正解」との誤差の評価関数。コスト関数を最小化することが学習の目的

ここでは以下で定義されるロジスティック関数を想定

$$J(\boldsymbol{w}) = -\left[\sum_{i=1}^{n}\sum_{j=1}^{nt} y_j^{(i)} \log a_j^{(i)} + \left(1 - y_j^{(i)}\right) \log\left(1 - a_j^{(i)}\right)\right]$$

パラメータ$\boldsymbol{w}$は「重み」コスト関数を最小にする$\boldsymbol{w}$を求めたい

i番目の学習データに対するj番目の出力ユニットの「正解」

i番目の学習データに対するj番目の出力ユニットの出力

値は0~1の範囲。出力が正解と一致すればコストは０，そうでなければ大きな値になる

実際には過学習しないように正則化

L1正則化

重みの絶対値の和を加えたコスト関数を最小化

$$L1 = \lambda\|\boldsymbol{w}\|_1^1 = \lambda\sum_{j=1}^{m}|w_j|$$

L2正則化

重みの２乗の和を加えたコスト関数を最小化

$$\mathrm{L2} = \frac{\lambda}{2}\|\boldsymbol{w}\|_2^2 = \lambda\sum_{j=1}^{m}|w_j|^2$$

誤差逆伝播(back propagation)

3. 誤差を逆方向に伝播(backward propagation)させることで、ネットワーク内のそれぞれの重みに対する偏導関数を求め、モデルを更新

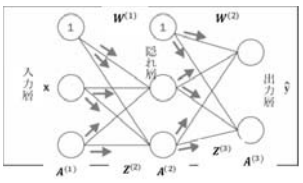
出力層の活性化ユニットを求める：順方向伝播の適用

$\boldsymbol{Z}^{(2)} = \boldsymbol{W}^{(1)}\left[\boldsymbol{A}^{(1)}\right]^T$ (隠れ層の入力)

$\boldsymbol{A}^{(2)} = \varphi(\boldsymbol{Z}^{(2)})$ (隠れ層の活性化)

$\boldsymbol{Z}^{(3)} = \boldsymbol{W}^{(2)}\boldsymbol{A}^{(2)}$ (出力層の総入力)

$\boldsymbol{A}^{(3)} = \varphi(\boldsymbol{Z}^{(3)})$ (出力層の活性化)



誤差逆伝播(back propagation)

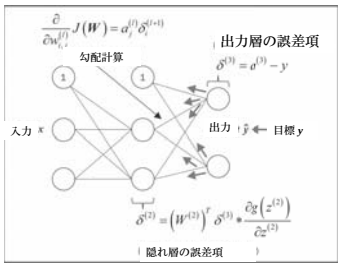
誤差逆伝播：誤差を右から左へ伝播(**y**は正しいクラスラベル)

出力層の誤差ベクトルを計算:

$$\delta^{(3)} = \boldsymbol{a}^{(3)} - \boldsymbol{y}$$

隠れ層の誤差項: $\delta^{(2)} = (\boldsymbol{W}^{(2)})^T\delta^{(3)} * \frac{\partial\varphi(z^{(2)})}{\partial z^{(2)}}$

活性化関数がシグモイドなら $\frac{\partial\varphi(z^{(2)})}{\partial z^{(2)}} = (\boldsymbol{a}^{(2)} * (1 - \boldsymbol{a}^{(2)}))$



演習12-2

活性化関数がφ(z) = $\frac{1}{1+e^{-z}}$ で与えられる時、次が成り立つことを示せ。

$$\frac{\partial\varphi(\mathrm{z})}{\partial\,\mathrm{z}} = \varphi(\mathrm{z})(1 - \varphi(\mathrm{z}))$$

註：

活性化関数がシグモイドなら $\frac{\partial\varphi(z^{(2)})}{\partial z^{(2)}} = (\boldsymbol{a}^{(2)} * (1 - \boldsymbol{a}^{(2)})$

と書いたのは、$\boldsymbol{a}^{(2)} = \varphi(z^{(2)})$ だからである

ニューラルネットの実装

MNIST 手書き数字のデータセットの利用

LeCunのウェブサイトから入手： 訓練用の画像とラベル(6万個)、テスト用の画像とラベル(1万個)

load_mnist関数を用いて学習用とテスト用を用意

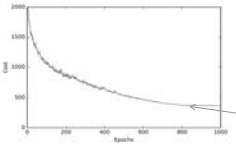
```
X_train, y_train = load_mnist('mnist', kind='train')
X_test, y_test = load_mnist('mnist', kind='t10k')
```

Scikit-learnのクラスを用いてニューラルネットを作成:

```
nn = NeuralNetMLP(n_output=10,          #出力ユニット数＝10
    n_features=X_train.shape[1],        #入力ユニット数=784
    n_hidden=50,                        # 隠れ層ユニット数=50
    l2=0.1,                             # L2正則化のλパラメタ=0.1
    l1=0.0,                             # L1正則化のλパラメタ=0.
    epochs=1000,                        # エポック数（訓練回数）=1000
    eta=0.001,                          # 学習率ηの初期値=0.001
    alpha=0.001,                        # モーメンタム学習の1つ前の勾配係数
    decrease_const=0.00001,             # 適応学習率の減少定数
    shuffle=True,                       # データをシャッフルする
    minibatches=50,                     # 各エポックのミニバッチ数=50
    random_state=1)                     # 乱数の初期化
```

ニューラルネットの学習と評価

nn.fit(X_train, y_train, print_progress=True)
により学習が行われ、その進行状況が図示化可能



ここらへんで収束

モデルの性能評価

```
y_test_pred = nn.predict(X_test)
acc = np.sum(y_test == y_test_pred, axis=0) / X_test.shape[0]
print('Test accuracy: %.2f%%' % (acc * 100))
```

Test accuracy: 95.62% (テストデータに対し95.62%)