知能プログラミング演習 I 演習課題

1 準備

1.1 各自の環境で実施している場合

- Moodle から課題を各自任意のフォルダにダウンロードし, 展開したフォルダの中に以下のものがすべて入っていることを確認
  - NN.py
  - DNN.py
  - report.pdf
  - report.tex
  - task.pdf

1.2 CSE で実施している場合

- まだ演習用のフォルダを作っていない人は DLL のフォルダを作成
  - ホームディレクトリに演習用のディレクトリを作成

    step1: `mkdir -p DLL`
- 作業ディレクトリ DLL に移動

  step1: `cd ./DLL`
  - 展開したフォルダの中に, 以下のものがすべて入っていることを確認
    - NN.py
    - DNN.py
    - report.pdf
    - report.tex
    - task.pdf
- Lec2 へ移動

  step1: `cd ./Lec2`

2 課題 1

0 から 3 までの手書き文字 (28 × 28 ピクセル) の 4 クラス分類を実装する．以下のプログラムを作成せよ．ただし, NN.py にコードを保存すること．NN.py ではデバッグ中の実行時間を短くするために，データの一部のみ使用するようにしてある．最後の課題では全てのデータを使うこと (NN.py は初期状態では未定義の関数がある影響で途中でエラーが出るが課題を進めていくと最後まで実行されるようになる).

1. 以下の関数を定義せよ.
   (a) 実数 x に対して, ReLU とその微分は以下のように書ける.

$$
\begin{aligned}
f(x) &= \max\{0, x\} \\
&= \begin{cases} x & x > 0 \\ 0 & \text{その他} \end{cases} \\
f'(x) &= \begin{cases} 1 & x > 0 \\ 0 & \text{その他} \end{cases}
\end{aligned}
$$

   NN.py 内の関数 ReLU を参考に ReLU の微分を計算する関数 dReLU を作成せよ．入力がベクトルの場合は，要素ごとの結果をベクトルとして返すこと．NN.py 内で関数 ReLU は以下のように定義されている.

```
def ReLU(x):
    return x*(x>0)
```

   python では真偽値 (True, False) による演算は 1, 0 として解釈されるため，$x > 0$ が True の場合は `x*1` に，False の場合は `x*0` になる．また，この関数はベクトルが入力されると要素単位で演算が実行されようになっている．例えば，関数 ReLU が定義された状態で，以下のように実行するとベクトルに対する挙動が確認できるのでどのような振る舞いをしているのか理解しておくこと.

```
In [1]: r = np.random.normal(0,1,10) # 乱数ベクトルを生成

In [2]: r
Out[2]:
array([ 0.91338191, -0.92811437,  0.90809788, -1.95660232,  1.59452886,
       -0.15433484,  0.31451213,  1.09889458,  0.7726738 ,  0.04000455])

In [3]: ReLU(r) # 乱数ベクトルを関数 ReLU に渡す
Out[3]:
array([ 0.91338191, -0.        ,  0.90809788, -0.        ,  1.59452886,
       -0.        ,  0.31451213,  1.09889458,  0.7726738 ,  0.04000455])
```

   (b) 中間層の出力 $\boldsymbol{z}$ と，定数ユニット以外の中間層の活性化関数の引数に関する微分 $\nabla f(W\boldsymbol{x})$ を関数 `forward` 内で計算し，返り値として返せ.

$$
\boldsymbol{z} = \begin{bmatrix} 1 \\ z_1 \\ \vdots \\ z_q \end{bmatrix} = \begin{bmatrix} 1 \\ f(\boldsymbol{w}_1^\top \boldsymbol{x}) \\ \vdots \\ f(\boldsymbol{w}_q^\top \boldsymbol{x}) \end{bmatrix} = \begin{bmatrix} 1 \\ f(W\boldsymbol{x}) \end{bmatrix}
$$

$$\nabla f(W\boldsymbol{x}) = \begin{bmatrix} f'(\boldsymbol{w}_1^\top \boldsymbol{x}) \\ \vdots \\ f'(\boldsymbol{w}_q^\top \boldsymbol{x}) \end{bmatrix}$$

$f(W\boldsymbol{x})$ や $\nabla f(W\boldsymbol{x})$ はベクトル $W\boldsymbol{x}$ の各要素に f や f' を適用したものである．これらの計算は

$$W\boldsymbol{x} = \begin{bmatrix} \boldsymbol{w}_1^\top \boldsymbol{x} \\ \vdots \\ \boldsymbol{w}_q^\top \boldsymbol{x} \end{bmatrix}$$

を計算してから (参考: 前回の課題で行なった `np.dot` による行列とベクトルの掛け算の計算), 各要素の f や f' を求めるとよい．今回の場合, 活性化関数 f とその微分 f' は上の (a) で作成した ReLU と dReLU であるが，ベクトル入力についても動作するように定義したことを利用すると簡単である．関数 `forward` は，引数として $\boldsymbol{x}$, $\boldsymbol{w}$ にくわえ，ReLU とその微分 dReLU を関数引数として受け取っていることに注意せよ (NN.py 内のコメント参照．返り値を複数返す関数や，関数を引数としてとる関数については講義ノート 1.4 の後半を参照)．また，$\boldsymbol{z}$ の作成で，1 と $f(W\boldsymbol{x})$ が結合したベクトルを作るには `np.append` を利用できる．“$\boldsymbol{z}$ を保存する変数 = `np.append`(1, $f(W\boldsymbol{x})$ に相当する np.array の変数)” とすればよい．

(c) ソフトマックス関数を定義して, 出力層の最終出力

$$g(V\boldsymbol{z})$$

を計算せよ．ただし, ソフトマックス関数は m 次元の実ベクトル $\boldsymbol{x} = (x_1, \ldots, x_m)^\top$ に対して, 以下で定義される．

$$g(\boldsymbol{x}) = \frac{1}{\sum_{k=1}^m e^{x_k}} \begin{bmatrix} e^{x_1} \\ \vdots \\ e^{x_m} \end{bmatrix}$$

(d) クロスエントロピー関数を定義して, 誤差関数の値を計算せよ．なお, クロスエントロピーは, 上で作成した最終層の出力を $\boldsymbol{g} = g(V\boldsymbol{z})$ として，m 次元の出力ラベル $\boldsymbol{y} = (y_1, \ldots, y_m)^\top$ に対して以下で定義される．

$$E(\boldsymbol{g}, \boldsymbol{y}) = -\sum_{j=1}^m y_j \log g_j.$$

ただし, g_j と y_j は $\boldsymbol{g}$ と $\boldsymbol{y}$ のそれぞれの j 番目の成分を表す．ここまで作成すると，error.pdf に誤差の値が保存されるようになるはずである (パラメータの更新がまだ作成されてないので初期誤差が定数としてプロットされる)．

(e) スライド 12page を参考に, $\boldsymbol{\delta}_2$ と $\boldsymbol{\delta}_1$ を計算するコードを作成せよ．$\boldsymbol{\delta}_1$ の計算は関数 `backword` 内で行うこと (呼び出し側はコメントで記載されている．引数に与えられている `V[:,1:]` は `V` の最初の列を省いた行列であることに注意)．また，$\boldsymbol{\delta}_1$ 内の行列の転置 $\top$ の存在に注意すること．np.array では変数の後ろに ‘.T’ をつけることで転置が表現できる．適当な $a \times b$ 行列 M, 長さ b のベクトル $\boldsymbol{x}$, 長さ a のベクトル $\boldsymbol{y}$ があったとき np.array では $M\boldsymbol{x}$ は `np.dot(M, x)` であり, 一方 $M^\top \boldsymbol{y}$ は `np.dot(M.T, y)` となる．例えば，以下のようなコードを実行すると上で述べたことを確認できる．各自手元で実行して，意味を理解すること．

```
In [1]: A = np.random.normal(0,1,(2,3))  # 2 x 3 の乱数行列を生成

In [2]: A
Out[2]:
array([[ 1.02831053, -0.42482729,  1.53931409],
       [-0.57573832,  1.46109948, -1.02279812]])

In [3]: A[:,1:] # A の 1 列目を省いた行列の作成
Out[3]:
array([[-0.42482729,  1.53931409],
       [ 1.46109948, -1.02279812]])

In [4]: np.dot(A,np.ones(3)) # A と 長さ 3 のベクトル (1, 1, 1)^T の掛け算
Out[4]: array([ 2.14279733, -0.13743696])

In [5]: np.dot(A.T,np.ones(2)) # A の転置 と 長さ 2 のベクトル (1, 1)^T の掛け算
Out[5]: array([0.45257221, 1.03627219, 0.51651597])

In [6]: np.dot(A,np.ones(2)) # A と 長さ 2 のベクトル (1, 1)^T の
                             # 掛け算を行うとサイズが合わずエラー
ValueError
...
ValueError: shapes (2,3) and (2,) not aligned: 3 (dim 1) != 2 (dim 0)

In [7]: np.dot(A[:,1:],np.ones(2))       # A の 1 列目を省くと長さ 2 のベクトル (1, 1)^T との
Out[7]: array([1.1144868 , 0.43830136])  # 掛け算は可能になる（A が 2x2 になるため）
```

(f) 同様にスライド 12page を参考に, W と V の更新を学習率 eta_t の確率的勾配降下法で行え（eta_t はすでに定義されているものを使えばよい）. 注: $\boldsymbol{\delta}_2 \boldsymbol{z}^{(t)\top}$ や $\boldsymbol{\delta}_1 \boldsymbol{x}^\top$ のような, 縦ベクトルと横ベクトルの掛け算には np.outer を使う（講義ノート 1.2.3「配列の操作」補足 1 の直後あたり参照）. ここまでで誤差逆伝播が定義されたため, error.pdf に変化が起こるはずである. 各自, 確認すること.

2. 学習終了後, テストデータの出力と正解に対して, confusion matrix を計算するプログラムを完成させよ. 変数 true_label と predict_label に正解ラベルと予測されたラベルが保存されている (print で中身を見てみること). 例えば, 以下のようにすると, 正解ラベルが 0 で予測ラベルも 0 である数をカウントできる:

   ```
   np.sum((true_label == 0) & (predict_label == 0)),
   ```

   (np.array との '==' は要素ごとの真偽値が返る. さらに, &で要素ごとの論理積がとられ, sum で True の数がカウントされる）. ConfMat[i,j] には true_label が i, predict_label が j であるものの数を保存すればよい[*1]

3. NN.py を実行し, error.pdf に保存される訓練誤差とテスト誤差が減少している様子を確認せよ. また, confusion.pdf で分類結果がどのようなものかも確認すること. このとき, NN.py の先頭付近

[*1] 正解と予測ともに, 0-1 の n_test×m で表現されていれば, 行列の掛け算だけで ConfMat を作ることもできる. 余裕がある人は考えてみるとよい.

use_small_data を False にし，全てのデータを使うように変更すること．さらに，plot_misslabeled を True にすると，誤分類されたデータがどのようなものか図に保存されるようになるのでこちらも観察してみること．

注意: NN.py 中のこの部分は全てうめているので, 結果を確認するだけで良い.

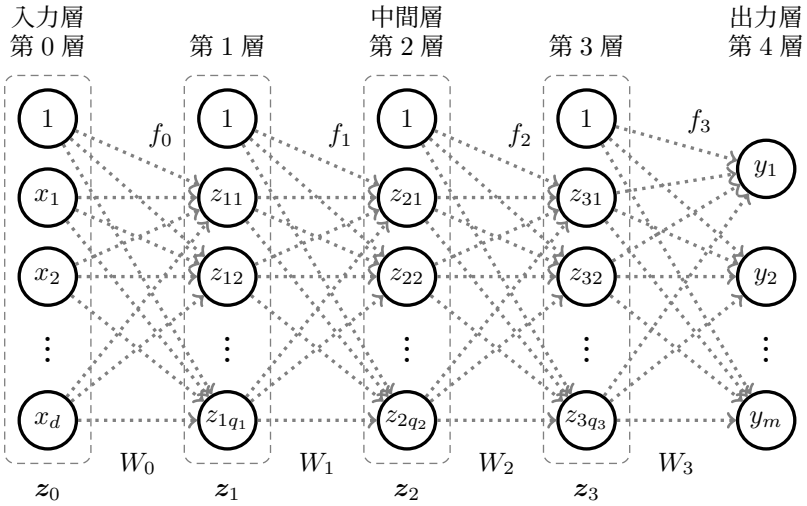

図 1　$r = 3$ のネットワーク

3　課題 2

先と同じ手書き文字 (28× 28 ピクセル) の多値分類を深層ニューラルネットワークで実装する.

1. 図 1 に示すような $r = 3$ のネットワークを以下の手順にしたがって作成する．ただし，各中間ユニットの数は $q_1 = 100, q_2 = 50, q_3 = 10$ とする．DNN.py にはこの数に合わせて，パラメータ行列 W_0 $(d \times 100)$, W_1 (100×50), W_2 (50×10) W_3 $(10 \times m)$ がすでに乱数で初期化され `W0`, `W1`, `W2`, `W3` という名前で作られている.
   (a) DNN.py には活性化関数としてすでに ReLU が作成してある．この関数は以前作成したものと違い，返り値として，ReLU とその微分の値の二つを返している．これを参考にシグモイド関数 `sigmoid(x)`，ハイパボリックタンジェント `Tanh(x)` について，その値と微分の二つを返す関数を作成せよ．また，ベクトルが入力された場合，それぞれの値について活性化関数が適用されたものが返される必要がある（典型的な作り方をすれば，勝手にこの条件は満たされるはずであるが，きちんとそのようになっているか確認すること）．例えば，ReLU が定義されている状態で，コンソールから以下のような振る舞いが確認できる

```
In [1]: a = np.array(range(-5,5)) # -5 から 4 までの数値からなるベクトル

In [2]: a
Out[2]: array([-5, -4, -3, -2, -1,  0,  1,  2,  3,  4])

In [3]: z, dz = ReLU(a) # a に対する ReLU とその微分をそれぞれ，z と dz に代入

In [4]: z
Out[4]: array([0, 0, 0, 0, 0, 0, 1, 2, 3, 4])
```

```
In [5]: dz
Out[5]: array([0, 0, 0, 0, 0, 0, 1, 1, 1, 1])
```

ヒント: シグモイドとハイパボリックタンジェント，またそれぞれの微分は以下で定義される

$$\begin{aligned} \text{シグモイド関数: } f(x) &= 1/(1+e^{-x}) \\ f'(x) &= f(x)(1-f(x)) \\ \text{ハイパボリックタンジェント: } f(x) &= \tanh x \\ f'(x) &= 1-f(x)^2 \end{aligned}$$

また，numpy ではハイパボリックタンジェントは np.tanh で計算できる．

ヒント: 適当な np.array 配列 x に対して各要素の 2 乗は x**2 で計算できる．

(b) 関数 forward(z_in, W_k, actfunc) に順伝播の処理を定義せよ（スライド 28page, 2.2 の処理に相当）．引数 z_in は $\boldsymbol{z}_k$，引数 W は W_k に相当する．また，引数 actfunc は上で作成した活性化関数のいずれかを受け取る関数型の引数とする．返り値はベクトル $\boldsymbol{z}_{k+1}$ と，ベクトル $\nabla f_k(W_k\boldsymbol{z}_k)$ の二つを返すものとする．NN.py で作成したものとほぼ同じようなプログラムになるが，ここでは actfunc が活性化関数の値と，その微分を同時に返すように作成されている点が異なることに注意せよ（例えば，上の ReLU の実行例のように，actfunc の返り値をカンマでつないだ二つの変数に渡せば活性化関数の値とその微分をそれぞれ別の変数に格納できる）．

(c) 関数 forward(z_in, W_k, actfunc) を使って，順伝播を完成させよ．出力層を除いて，活性化関数は全ての層で ReLU とする．出力層の softmax(x) 関数は課題 1 で作成したものをそのまま使えばよい．CrossEntoropy(g, y) を定義して誤差の評価が保存される場所まで作成すること（CrossEntoropy 関数も課題 1 のものをそのまま使えばよい）．訓練とテストで，順伝播処理は 2 回出てくるので注意すること（処理はほぼ同じ）．訓練のクロスエントロピーが配列 e に，テストのクロスエントロピーが e_test に保存される．ヒント: 直前の課題で作成した関数 forward(z_in, W_k, actfunc) は任意の $k = 0, \ldots, r-1$ に対する $\boldsymbol{z}_k$ と W_k で使用できることを利用して，関数 forward(z_in, W_k, actfunc) を繰り返し適用していけばよい（スライド 28page, 2.2 の処理の流れをよく理解しておくこと）．

ここまで作成すると，DNN.py 実行後に誤差関数の値の推移が error.pdf に保存されるはずである（パラメータ更新部がまだ作成されてないので，変化しない定数の誤差が plot される）．

(d) スライド 28page の 3.1, 3.2 を参考に $\boldsymbol{\delta}_3$, $\boldsymbol{\delta}_2$, $\boldsymbol{\delta}_1$, $\boldsymbol{\delta}_0$ を作成せよ．$\boldsymbol{\delta}_2$, $\boldsymbol{\delta}_1$, $\boldsymbol{\delta}_0$ については関数 backward(W_tilde, delta, derivative) を使って，逆伝播を行い計算する．関数 backward(W_tilde, delta, derivative) 自体は課題 1 とほぼ同様となるが，層を逆順に辿って適用していく手順に注意すること．

(e) パラメータ W_3, W_2, W_1, W_0 の更新を学習率 eta_t の確率勾配降下法で行うコードを作成せよ（eta_t はすでに定義されているものを使えばよい）．ここまで作成すると，DNN.py 実行後に保存される error.pdf の誤差関数の値の推移が変化するする．前回同様，DNN.py の先頭付近，use_small_data で False を True にするとデータ全体を使うようになる．このようにした時の結果（error.pdf など）を確認すること．また，plot_misslabeled を True にすることで，分類誤りがでたサンプルを確認できるので，こちらも観察してみること．

2. 作成した DNN.py の深層ニューラルネットワークの様々な設定を自由に変えて，結果を観察せよ．変

更する設定には，例えば，以下の設定がある．

- 分類するクラス数 (または，どの数字を分類するか)
- 活性化関数: シグモイド関数, ReLU, ハイパボリックタンジェント
  - 一つのネットワークで複数の異なる活性化関数を使っても構わない
- 層の数
- 中間層のユニット数
- 学習率 `eta_t` の設定方法
- epoch 数

解析結果のレポートを作成し, pdf ファイルで提出せよ．tex や word など何を使用して作成してもよいが，大まかにでよいので report.pdf のような体裁で整えること（report.pdf を生成した tex も含まれているので，使いたければこれを使ってもよい）．以下のことに留意すること．

- 設定した中間層の数や, 中間層ごとのユニット数, 各層で用いた活性化関数などの実験設定を正確に記述すること．
- 解析結果の図 (誤差関数の推移や confusion matrix など) も用いること (confusion matrix は初期状態では作成されないが，課題 2 と全く同じコードをコピーして変数 `ConfMat` の値を適切に設定すれば confusion.pdf に保存される)
- 解析結果に対する考察を述べること．

4 課題の提出

Moodle を使ってファイルを提出してください. 提出方法は以下の通りです.

- Moodle にログインし, 知能プログラミング演習のページへ移動.
- Lec2 の項目に, NN.py, DNN.py, レポートの PDF をアップロードする.

6/21(金) の 17:00 を提出期限とします.