

計算機科学実験及演習 4 画像認識 レポート 4

高橋駿一 2018 年度入学 1029-30-3949

提出日: 2020 年 11 月 26 日

課題 4

課題内容

MNIST のテスト画像 1 枚を入力とし, 3 層ニューラルネットワークを用いて, 0~9 の値のうち 1 つを出力するプログラムを作成した.

作成したプログラムの説明

機能ごとにクラスや関数を実装している.

1. params クラス

ソースコード 1: パラメータ W と b に関する処理

```
1 class params:
2     def __init__(self, M, d):
3         np.random.seed(seed=32)
4         self.W = np.random.normal(0, 1/d, (d, M))
5         self.b = np.random.normal(0, 1/d, (1, M))
6         self.eta = 0.01
7         approach = 'Adam'
8         self.op1 = optimize(approach)
9         self.op2 = optimize(approach)
10
11     def update(self, dW, db):
12         self.W += self.op1.update(dW)
13         self.b += self.op2.update(db)
14
15     def save(self, i):
16         np.save('./w{}'.format(i), self.W)
17         np.save('./b{}'.format(i), self.b)
```

重み W とバイアス b の初期化, 更新, 保存を行うクラスである.

`__init__`(コンストラクタ)

入力

- M: 次の層のノード数
- d: 前の層のノード数

メンバ変数

- W: 重み
- b: バイアス
- eta: 学習率
- op1, op2: W, b の最適化を行うクラスのインスタンス

W, b を M, d に基づいて初期化し, それぞれを最適化するためのインスタンスを作成する.

update 関数

入力

- dW: $\frac{\partial E_n}{\partial W}$
- db: $\frac{\partial E_n}{\partial b}$

E_n : 損失関数である. op1, op2 によって W, b の値を更新する.

save 関数

入力

- i: 文字列

入力 i を受けて W, b をそれぞれ wi, bi という名前の.npy ファイルで保存する.

2. optimize クラス

ソースコード 2: パラメータ更新の手法選択

```
1 class optimize:
2     def __init__(self, approach):
3         self.approach = approach
4         self.diff = 0
5         if approach == 'default':
6             self.eta = 0.01
7         elif approach == 'SGD':
8             self.eta = 0.01
9             self.alpha = 0.9
10        elif approach == 'AdaGrad':
11            self.h = 1e-8
12            self.eta = 0.001
13        elif approach == 'RMSProp':
14            self.h = 0
15            self.eta = 0.001
16            self.rho = 0.9
17            self.epsilon = 1e-8
18        elif approach == 'AdaDelta':
19            self.h = 0
20            self.s = 0
21            self.rho = 0.95
22            self.epsilon = 1e-6
23        elif approach == 'Adam':
24            self.t = 0
25            self.m = 0
26            self.v = 0
27            self.alpha = 0.001
28            self.beta1 = 0.9
29            self.beta2 = 0.999
30            self.epsilon = 1e-8
31
32        def update(self, d_):
33            if self.approach == 'default':
34                self.diff = (-1) * self.eta * d_
35            elif self.approach == 'SGD':
36                self.diff = self.alpha * self.diff - self.eta * d_
37            elif self.approach == 'AdaGrad':
38                self.h = self.h + d_ * d_
39                self.diff = (-1) * self.eta / np.sqrt(self.h) * d_
```

```

40         elif self.approach == 'RMSProp':
41             self.h = self.rho * self.h + (1 - self.rho) * d_ * d_
42             self.diff = (-1) * self.eta / (np.sqrt(self.h) + self.epsilon) * d_
43         elif self.approach == 'AdaDelta':
44             self.h = self.rho * self.h + (1 - self.rho) * d_ * d_
45             self.diff = (-1) * np.sqrt(self.s + self.epsilon) / np.sqrt(self.h + self.epsilon)
46                 * d_
47             self.s = self.rho * self.s + (1 - self.rho) * self.diff * self.diff
48         elif self.approach == 'Adam':
49             self.t = self.t + 1
50             self.m = self.beta1 * self.m + (1 - self.beta1) * d_
51             self.v = self.beta2 * self.v + (1 - self.beta2) * d_ * d_
52             m_hat = self.m / (1 - self.beta1 ** self.t)
53             v_hat = self.v / (1 - self.beta2 ** self.t)
54             self.diff = (-1) * self.alpha * m_hat / (np.sqrt(v_hat) + self.epsilon)
55         return self.diff

```

パラメータ更新の手法を選択するためのクラスである.

`__init__`(コンストラクタ)

入力

- approach: 最適化の手法を表す文字列

メンバ変数 (共通)

- approach: 最適化の手法を表す文字列
- diff: パラメータの更新前後の差分

入力 approach を元にパラメータ更新の手法を選択し, それぞれの計算に必要な変数をメンバ変数を初期化する.

update 関数

入力

- d.: $\frac{\partial E_n}{\partial W}$

メンバ変数 approach によって更新手法を制御し, それぞれのメンバ変数と入力 d_ を元に diff を計算する.

3. load 関数

ソースコード 3: W と b の.npy ファイルを読み込み

```

1     def load(i):
2         W_loaded = np.load('./w{}.npy'.format(i))
3         b_loaded = np.load('./b{}.npy'.format(i))
4         return W_loaded, b_loaded

```

.npy ファイルを読み込むための関数である.

入力

- i: 文字列

入力 i を受けて wi.npy, bi.npy を読み込む.

4. create_batch 関数

ソースコード 4: ミニバッチを作成

```
1 def create_batch(X):
2     batch_size = 100
3     np.random.seed(seed=32)
4     batch_index = np.random.choice(len(X), (600, batch_size))
5     return batch_index
```

ミニバッチのインデックスを作成するための関数である.

入力

- X: 元の画像データ

シードを固定し, 0 以上入力 X の長さ (画像データの数) 以下の非負整数を 600*batch_size の行列に割り当てる.

5. input_layer_train

ソースコード 5: 学習時の入力層の処理

```
1 def input_layer_train(X, j):
2     batch_index = create_batch(X)
3     input_images = X[batch_index[j]] / 255
4     image_size = 784
5     class_num = 10
6     input_vector = input_images.reshape(100, image_size)
7     return input_vector, image_size, batch_index, class_num
```

学習時の入力層の処理を行う関数である.

入力

- X: 元の画像データ
- j: batch_index の何行目を参照するかを指定する int

入力 X を引数に create_batch 関数を実行し, batch_index の j 行目に対応する X を正規化し input_images に格納し, これを batch_size*image_size の行列に変形する.

6. input_layer_test

ソースコード 6: テストの入力層の処理

```
1 def input_layer_test(X, i):
2     input_image = X[i] / 255
3     image_size = input_image.size
4     image_num = len(X)
5     class_num = 10
6     input_vector = input_image.reshape(1, image_size)
7     return input_vector, image_size, i, class_num
```

テスト時の入力層の処理を行う関数である。

入力

- X: 元の画像データ
- i: 画像データの何番目を参照するかを指定する int

入力 X の i 番目のデータを正規化しベクトルに変換する。

7. matrix_operation クラス

ソースコード 7: 線形和の計算

```
1 class matrix_operation:
2     def __init__(self, W, b):
3         self.W = W
4         self.b = b
5         self.X = None
6
7     def forward(self, X):
8         self.X = X
9         y = np.dot(X, self.W) + self.b
10        return y
11
12    def backward(self, back):
13        dX = np.dot(back, self.W.T)
14        dW = np.dot(self.X.T, back)
15        db = np.sum(back, axis=0)
16        return dX, dW, db
```

線形和の計算に関するクラスである。

__init__(コンストラクタ)

入力

- W: 重み
- b: バイアス

メンバ変数

- W: 重み
- b: バイアス
- X: 順伝播の入力

forward 関数

入力

- X: 順伝播の入力

X, W, b で線形和を計算する.

backward 関数

入力

- back: $\frac{\partial E_n}{\partial y}$

順伝播の出力 y による偏微分 back を入力として受け取り, これを元に $\frac{\partial E_n}{\partial X}$, $\frac{\partial E_n}{\partial W}$, $\frac{\partial E_n}{\partial b}$ を計算する.

8. sigmoid クラス

ソースコード 8: シグモイド関数の計算

```
1 class sigmoid:
2     def __init__(self):
3         self.y = None
4
5     def forward(self, t):
6         self.y = (1 / (1 + np.exp(-1 * t)))
7         return self.y
8
9     def backward(self, back):
10        dt = back * (1 - self.y) * self.y
11        return dt
```

シグモイド関数に関するクラスである.

__init__(コンストラクタ)

メンバ変数

- y: 順伝播の出力

forward 関数

入力

- t: 順伝播の 1 つ前の演算結果

シグモイド関数の関数適用を行う.

backward 関数

入力

- back: $\frac{\partial E_n}{\partial y}$

順伝播の出力 y による偏微分 back を入力として受け取り, これを元に $\frac{\partial E_n}{\partial t}$ を計算する.

9. ReLU クラス

ソースコード 9: ReLU 関数の計算

```
1 class ReLU():
2     def __init__(self):
3         self.a = None
4
5     def forward(self, t):
6         self.a = np.where(t > 0, t, 0)
7         return self.a
8
9     def backward(self, back):
10        dt = back * np.where(self.a > 0, 1, 0)
11        return dt
```

ReLU 関数に関するクラスである.

`__init__`(コンストラクタ)

メンバ変数

- a: 順伝播の出力

forward 関数

入力

- t: 順伝播の 1 つ前の演算結果

ReLU 関数の関数適用を行う.

backward 関数

入力

- back: $\frac{\partial E_n}{\partial a}$

順伝播の出力 a による偏微分 back を入力として受け取り, これを元に $\frac{\partial E_n}{\partial t}$ を計算する.

10. Dropout クラス

ソースコード 10: Dropout 関数の計算

```
1  def __init__(self, rho):
2      self.rho = rho
3      self.mask = None
4
5  def forward(self, t, train_flag=1):
6      if train_flag == 1:
7          self.mask = np.random.rand(*t.shape) > self.rho
8          a = t * self.mask
9          return a
10     else:
11         a = t * (1 - self.rho)
12         return a
13
14 def backward(self, back):
15     dt = back * self.mask
16     return dt
```

Dropout 関数に関するクラスである.

`__init__`(コンストラクタ)

メンバ変数

- rho: 無視するノードの割合
- mask: 無視しない/する要素の位置をそれぞれ True/False とした行列

`forward` 関数

入力

- t: 順伝播の 1 つ前の演算結果
- train_flag: 順伝播が学習時かテスト時かを制御するフラグ

Dropout 関数の関数適用を行う. なお, train_flag が 1 のときに学習時である.

`backward` 関数

入力

- back: $\frac{\partial E_n}{\partial a}$

順伝播の出力 a による偏微分 back を入力として受け取り, これを元に $\frac{\partial E_n}{\partial t}$ を計算する.

11. Batch_Normalization クラス

ソースコード 11: Batch_Normalization に関する計算

```
1 class Batch_Normalization():
2     mean_list = []
3     var_list = []
4
5     def __init__(self):
6         self.batch_size = None
7         self.gamma = 1
8         self.beta = 0
9         self.x = None
10        self.mean = None
11        self.var = None
12        self.normalized_x = None
13        self.epsilon = 1e-7
14        self.op1 = optimize('Adam')
15        self.op2 = optimize('Adam')
16
17    def forward(self, x, train_flag=1):
18        self.x = x
19        if train_flag == 1:
20            self.batch_size = x.shape[0]
21            self.mean = np.mean(x, axis=0)
22            self.var = np.var(x, axis=0)
23            # print('x:', x.shape)
24            self.normalized_x = (x - self.mean) / np.sqrt(self.var + self.epsilon)
25            y = self.gamma * self.normalized_x + self.beta
26        else:
27            y = self.gamma / np.sqrt(np.mean(Batch_Normalization.var_list, axis=0) +
28                                     self.epsilon) * x + \
29                (self.beta - self.gamma * np.mean(Batch_Normalization.mean_list, axis=0) /
30                np.sqrt(np.mean(Batch_Normalization.var_list, axis=0) +
31                             self.epsilon))
32
33        return y
34
35    def backward(self, back):
36        dn_x = back * self.gamma
37        dvar = np.sum(dn_x * (self.x - self.mean) * (-1 / 2) * (self.var + self.epsilon)
38                      ** (-3 / 2), axis=0)
39        dmean = np.sum(dn_x * (-1) / np.sqrt(self.var + self.epsilon), axis=0) + dvar
40                * np.sum(-2 * (self.x - self.mean), axis=0) / self.batch_size
41        dx = dn_x / np.sqrt(self.var + self.epsilon) + dvar * 2 * (self.x - self.mean) /
42              self.batch_size + dmean / self.batch_size
43        dgamma = np.sum(back * self.normalized_x, axis=0)
44        dbeta = np.sum(back, axis=0)
45        self.gamma += self.op1.update(dgamma)
46        self.beta += self.op2.update(dbeta)
47        return dx
```

Batch_Normalization に関するクラスである。

クラス変数

- mean_list: ミニバッチの平均を格納するためのリスト
- var_list: ミニバッチの分散を格納するためのリスト

__init__(コンストラクタ)

メンバ変数

- batch_size: バッチサイズ
- gamma: 正規化後に出力を調整するためのパラメータ
- beta: 正規化後に出力を調整するためのパラメータ
- x: 順伝播の 1 つ前の演算結果
- mean: ミニバッチの平均
- var: ミニバッチの分散
- normalized_x: x を正規化したもの
- epsilon: 分母=0 を防ぐための微小量
- op1, op2: gamma, beta の最適化を行うクラスのインスタンス

forward 関数

入力

- x: 順伝播の 1 つ前の演算結果
- train_flag: 順伝播が学習時かテスト時かを制御するフラグ

Batch_Normalization の順伝播の計算を行う。なお, train_flag が 1 のときに学習時である。

backward 関数

入力

- back: $\frac{\partial E_n}{\partial y}$

順伝播の出力 y による偏微分 back を入力として受け取り, これを元に $\frac{\partial E_n}{\partial x}$, $\frac{\partial E_n}{\partial \gamma}$, $\frac{\partial E_n}{\partial \beta}$ を計算する。また, gamma と beta の更新を行う。

12. softmax クラス

ソースコード 12: ソフトマックス関数の計算

```

1      class softmax:
2          def __init__(self, batch_size):
3              self.y_pred = None
4              self.batch_size = batch_size
5
6          def forward(self, a):
7              alpha = np.tile(np.amax(a, axis=1), 10).reshape(10, self.batch_size).T
8              exp_a = np.exp(a - alpha)
9              sum_exp = np.tile(np.sum(exp_a, axis=1), 10).reshape(10, self.batch_size).T
10             self.y_pred = exp_a / sum_exp
11             return self.y_pred
12
13         def backward(self, y_ans, B):
14             da = (self.y_pred - y_ans) / B
15             return da

```

softmax 関数に関するクラスである。

`__init__`(コンストラクタ)

メンバ変数

- `y_pred`: 出力層の出力
- `batch_size`: バッチサイズ

forward 関数

入力

- `a`: 順伝播の 1 つ前の演算結果

`softmax` 関数の関数適用を行う.

backward 関数

入力

- `y_ans`: 正解クラスを one-hot vector 表記にしたもの
- `B`: バッチサイズ

`y_ans`, `B` 入力として受け取り, これを元に $\frac{\partial E_n}{\partial a}$ を計算する.

13. `postprocessing` 関数

ソースコード 13: 後処理

```
1 def postprocessing(y):
2     binary_y = np.where(y == np.amax(y, axis=1), 1, 0)
3     return binary_y
```

後処理を行うための関数である.

入力

- `y`: 出力層の出力

`y` を受け取り, 行ごとに最も大きい値を 1, それ以外を 0 にする.

14. `cross_entropy_loss` 関数

ソースコード 14: クロスエントロピー誤差の計算

```
1 def cross_entropy_loss(y_pred, y_ans):
2     B = len(y_pred)
3     E = 1 / B * np.sum((-1) * y_ans * np.log(y_pred))
4     return E
```

クロスエントロピー誤差を計算するための関数である.

入力

- `y_pred`: 出力層の出力
- `y_ans`: 正解クラスを one-hot vector 表記にしたもの

`y_pred` と `y_ans` によってクロスエントロピー誤差を計算する.

15. neural_network クラス

ソースコード 15: 3 層ニューラルネットワークの構成

```
1 class neural_network():
2     def __init__(self, batch_size, epoch, middle_layer, last):
3         self.batch_size = batch_size
4         self.epoch = epoch
5         self.middle_layer = middle_layer
6         self.last = last
7
8     def learning(self):
9         params1 = params(self.middle_layer, 784)
10        params2 = params(self.last, self.middle_layer)
11        for i in range(self.epoch):
12            loss = []
13            for j in range(int(60000 / self.batch_size)):
14                input_vec, image_size, batch_index, class_sum = input_layer_train(
15                    train_X, j)
16                batch_label = train_Y[batch_index[j]]
17                y_ans = np.identity(10)[batch_label]
18
19                W1, b1 = params1.W, params1.b
20                mo1 = matrix_operation(W1, b1)
21                t = mo1.forward(input_vec)
22                # print('matrix', t)
23
24                bn = Batch_Normalization()
25                y_bn = bn.forward(t)
26                # y_bn = t
27
28                # sig = sigmoid()
29                # y1 = sig.forward(t)
30                re = ReLU()
31                y_re = re.forward(y_bn)
32                # print('sigmoid', y1)
33
34                dr = Dropout(0.2)
35                y1 = dr.forward(y_re)
36
37                W2, b2 = params2.W, params2.b
38                mo2 = matrix_operation(W2, b2)
39                a = mo2.forward(y1)
40                # print('a', a)
41                soft = softmax(self.batch_size)
42                y2 = soft.forward(a)
43                # print(y2)
44                # binary_y = postprocessing(y2)
45                # print(binary_y)
46                E = cross_entropy_loss(y2, y_ans)
47                loss.append(E)
48
49            da = soft.backward(y_ans, self.batch_size)
50            dX2, dW2, db2 = mo2.backward(da)
51
52            dt_dr = dr.backward(dX2)
```

```

53         # dt = sig.backward(dX2)
54         dt_re = re.backward(dt_dr)
55
56         dt = bn.backward(dt_re)
57         # dt = dt_re
58
59         dX1, dW1, db1 = mo1.backward(dt)
60         params1.update(dW1, db1)
61         params2.update(dW2, db2)
62         Batch_Normalization.mean_list = np.append(Batch_Normalization.mean_list
63             , bn.mean, axis=0)
64         Batch_Normalization.var_list = np.append(Batch_Normalization.var_list, bn.
65             var, axis=0)
66         # Batch_Normalization.var_list.append(bn.var)
67         print(np.sum(loss) / len(loss))
68
69         params1.save(1)
70         params2.save(2)
71
72     def testing(self, all_flag=1):
73         # input_vector, image_size, i, class_num = input_layer(test_X)
74         ans = []
75         Batch_Normalization.mean_list = Batch_Normalization.mean_list.reshape([self.
76             epoch, self.middle_layer])
77         Batch_Normalization.var_list = Batch_Normalization.var_list.reshape([self.epoch,
78             self.middle_layer])
79         # print(Batch_Normalization.mean_list.shape)
80         if all_flag == 1:
81             for k in range(test_Y.shape[0]):
82                 input_vector, image_size, i, class_num = input_layer_test(test_X, k)
83                 # y_ans = np.identity(10)[test_Y[i]]
84                 W1, b1 = load(1)
85                 mo1 = matrix_operation(W1, b1)
86                 t = mo1.forward(input_vector)
87
88                 bn = Batch_Normalization()
89                 y_bn = bn.forward(t, 0)
90                 # y_bn = t
91
92                 # print('matrix', y1)
93                 # sig = sigmoid()
94                 # y1 = sig.forward(t)
95                 re = ReLU()
96                 y_re = re.forward(y_bn)
97                 # print('sigmoid', y1)
98
99                 dr = Dropout(0.2)
100                 y1 = dr.forward(y_re, 0)
101
102                 W2, b2 = load(2)
103                 mo2 = matrix_operation(W2, b2)
104                 a = mo2.forward(y1)
105                 # print('a', a)
106
107                 soft = softmax(1)
108                 y2 = soft.forward(a)
109                 # print(y2)
110                 binary_y = postprocessing(y2)
111                 # print(np.where(binary_y == 1)[1][0], test_Y[i])
112                 eq = 1 if np.where(binary_y == 1)[1][0] == test_Y[i] else 0
113                 ans.append(eq)
114             print(np.mean(ans))
115         else:
116             k = int(input('テストデータの何番目を試すか入力してください'))
117             input_vector, image_size, i, class_num = input_layer_test(test_X, k)
118             # y_ans = np.identity(10)[test_Y[i]]
119             W1, b1 = load(1)
120             mo1 = matrix_operation(W1, b1)

```

```

117         t = mo1.forward(input_vector)
118
119         bn = Batch_Normalization()
120         y_bn = bn.forward(t, 0)
121         # y_bn = t
122
123         # print('matrix', y1)
124         # sig = sigmoid()
125         # y1 = sig.forward(t)
126         re = ReLU()
127         y_re = re.forward(y_bn)
128         # print('sigmoid', y1)
129
130         dr = Dropout(0.2)
131         y1 = dr.forward(y_re, 0)
132
133         W2, b2 = load(2)
134         mo2 = matrix_operation(W2, b2)
135         a = mo2.forward(y1)
136         # print('a', a)
137
138         soft = softmax(1)
139         y2 = soft.forward(a)
140         # print(y2)
141         binary_y = postprocessing(y2)
142         print('予測結果:', np.where(binary_y == 1)[1][0], '正解:', test_Y[k])

```

ニューラルネットワークの構成に関するクラスである.

`__init__`(コンストラクタ)

メンバ変数

- `batch_size`: バッチサイズ
- `epoch`: エポック数
- `middle_layer`: 中間層のノード数
- `last`: 出力層のノード数

メンバ変数それぞれを設定する.

learning 関数 学習を行う. 入力層 → 線形和 → Batch_Normalization → ReLU → Dropout → 中間層 → 線形和 → 出力層 → softmax(→ クロスエントロピー誤差) という構成で順伝播, 逆伝播の計算を行い, パラメータを更新し, 1 エポックごとにクロスエントロピー誤差と Batch_Normalization の平均, 分散をそれぞれリストに格納する. そして, 学習終了後に重み `W` とバイアス `b` を `.npy` ファイルとして保存する.

testing 関数

入力

- `all_flag`: テストデータ全てを活用するか 1 つを活用するか制御するフラグ

学習時に保存した.npy ファイルを読み込み, 全テストデータに関してニューラルネットワークの計算を行う. `all_flag` が 1 であればテストデータの全てに関して計算を実行し, 最後に正解率を標準出力に出力する. 一方 `all_flag` が 0 であれば標準入力でテストデータの何番目を活用するか受け取り, 最後に計算による予測結果と正解を標準出力に出力する.

学習とテストの実行

ソースコード 16: 学習とテストの実行

```
1 nn = neural_network(100, 20, 50, 10)
2 print('学習を開始します.␣')
3 nn.learning()
4 print('テストを開始します.␣')
5 nn.testing(all_flag=0)
```

`neural_network` クラスをインスタンス化し, 学習とテストを実行する. なお, バッチサイズ: 100, エポック数: 20, 中間層のノード数: 50, 出力層のノード数: 10 で実行した.

実行結果

クロスエントロピー誤差が 0.7203779372223761 から 0.05723036650842045 まで減少し, 多くのテストデータについて正解を導けた. なお, `all_flag=1` とし, 全てのテストデータに関して実行すると正解率は 95.87%となった.

工夫点

- 機能ごとクラスを作成することで可読性と保守性を高めた.
- 様々な行列に関する演算を実装する際に `for` 文を使わず, `numpy` の機能を活用することで計算時間を抑えた.

問題点

- バッチサイズを一箇所で管理できていない.