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

誤差逆伝播法を実装したため, 課題 2 の時点と比較して関数の数が大幅に増加したため, 既存の関数を機能別でクラスにまとめることで可読性と再利用性を高めた.

0.0.1 params クラス

ソースコード 1: パラメータ W

```
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: $\dfrac{\partial E_n}{\partial W}$
- db: $\dfrac{\partial E_n}{\partial b}$

E: 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.

0.0.2 optimize クラス

ソースコード 2: パラメータ更新の手法選択

```
class optimize:
    def __init__(self, approach):
        self.approach = approach
        self.diff = 0
        if approach == 'default':
            self.eta = 0.01
        elif approach == 'SGD':
            self.eta = 0.01
            self.alpha = 0.9
        elif approach == 'AdaGrad':
            self.h = 1e−8
            self.eta = 0.001
        elif approach == 'RMSProp':
            self.h = 0
            self.eta = 0.001
            self.rho = 0.9
            self.epsilon = 1e−8
        elif approach == 'AdaDelta':
            self.h = 0
            self.s = 0
            self.rho = 0.95
            self.epsilon = 1e−6
        elif approach == 'Adam':
            self.t = 0
            self.m = 0
            self.v = 0
            self.alpha = 0.001
            self.beta1 = 0.9
            self.beta2 = 0.999
            self.epsilon = 1e−8

    def update(self, d_):
        if self.approach == 'default':
            self.diff = (−1) * self.eta * d_
        elif self.approach == 'SGD':
            self.diff = self.alpha * self.diff − self.eta * d_
        elif self.approach == 'AdaGrad':
            self.h = self.h + d_ * d_
            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)
                            * d_
46                      self.s = self.rho * self.s + (1 - self.rho) * self.diff * self.diff
47                  elif self.approach == 'Adam':
48                      self.t = self.t + 1
49                      self.m = self.beta1 * self.m + (1 - self.beta1) * d_
50                      self.v = self.beta2 * self.v + (1 - self.beta2) * d_ * d_
51                      m_hat = self.m / (1 - self.beta1 ** self.t)
52                      v_hat = self.v / (1 - self.beta2 ** self.t)
53                      self.diff = (-1) * self.alpha * m_hat / (np.sqrt(v_hat) + self.epsilon)
54                  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として初期化する.

update 関数

入力

- d_: $\dfrac{\partial E_n}{\partial W}$

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.

0.0.3 load 関数

ソースコード 3: W

```
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.

0.0.4 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, 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.

0.0.5 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.

0.0.6 matrix_operation クラス

ソースコード 6: 線形和の計算

```
class matrix_operation:
    def __init__(self, W, b):
        self.W = W
        self.b = b
        self.X = None

    def forward(self, X):
        self.X = X
        y = np.dot(X, self.W) + self.b
        return y

    def backward(self, back):
        dX = np.dot(back, self.W.T)
        dW = np.dot(self.X.T, back)
        db = np.sum(back, axis=0)
        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.

ソースコード 7: ソフトマックス関数の計算とその逆伝播

```
class softmax:
    def __init__(self, batch_size):
        self.y_pred = None
        self.batch_size = batch_size

    def forward(self, a):
        alpha = np.tile(np.amax(a, axis=1), 10).reshape(10, self.batch_size).T
        # print('max', alpha)
        exp_a = np.exp(a − alpha)
        # print('e', exp_a)
        sum_exp = np.tile(np.sum(exp_a, axis=1), 10).reshape(10, self.batch_size).T
        # print('sum', sum_exp)
        self.y_pred = exp_a / sum_exp
        return self.y_pred

    def backward(self, y_ans, B):
        da = (self.y_pred − y_ans) / B
        return da
```

ソフトマックス関数についてのクラスである． forward() でソフトマックス関数の適用を行い, backward() でその逆伝播の計算を行う. なお, 課題 2 のレポートでは alpha を計算する際に axis を指定していなかったため, alpha が行列 a の成分の中で最も大きい値を取っていたが, 本来は上記のコードのように行列 a の各行の中で最も大きい値を取り, それを行列 a と同じサイズに拡張した行列となる.

ソースコード 8: ニューラルネットワーク

```
class neural_network():
    def __init__(self, batch_size, epoch, middle_layer, last):
        self.batch_size = batch_size
        self.epoch = epoch
        self.middle_layer = middle_layer
        self.last = last

    def learning(self):
        params1 = params(self.middle_layer, 784)
        params2 = params(self.last, self.middle_layer)
        for i in range(self.epoch):
            loss = []
            for j in range(int(60000 / self.batch_size)):
                input_vec, image_size, batch_index, class_sum = input_layer2(train_X, j)
                batch_label = train_Y[batch_index[j]]
                y_ans = np.identity(10)[batch_label]

                W1, b1 = params1.W, params1.b
                mo1 = matrix_operation(W1, b1)
                t = mo1.forward(input_vec)
                # print('matrix', t)
                sig = sigmoid()
                y1 = sig.forward(t)
                # print('sigmoid', y1)
                W2, b2 = params2.W, params2.b
                mo2 = matrix_operation(W2, b2)
                a = mo2.forward(y1)
                # print('a', a)
                soft = softmax(self.batch_size)
```

```
30                          y2 = soft.forward(a)
31                          # print(y2)
32                          # binary_y = postprocessing(y2)
33                          # print(binary_y)
34                          E = cross_entropy_loss(y2, y_ans)
35                          loss.append(E)
36
37                          da = soft.backward(y_ans, self.batch_size)
38                          dX2, dW2, db2 = mo2.backward(da)
39                          dt = sig.backward(dX2)
40                          dX1, dW1, db1 = mo1.backward(dt)
41                          params1.update(dW1, db1)
42                          params2.update(dW2, db2)
43
44                      print(np.sum(loss) / len(loss))
45
46                  params1.save(1)
47                  params2.save(2)
48
49              def testing(self):
50                  input_vector, image_size, i, class_num = input_layer(test_X)
51                  # y_ans = np.identity(10)[test_Y[i]]
52                  W1, b1 = load(1)
53                  mo1 = matrix_operation(W1, b1)
54                  t = mo1.forward(input_vector)
55                  # print('matrix', y1)
56                  sig = sigmoid()
57                  y1 = sig.forward(t)
58                  # print('sigmoid', y1)
59                  W2, b2 = load(2)
60                  mo2 = matrix_operation(W2, b2)
61                  a = mo2.forward(y1)
62                  # print('a', a)
63                  soft = softmax(1)
64                  y2 = soft.forward(a)
65                  # print(y2)
66                  binary_y = postprocessing(y2)
67                  print(np.where(binary_y == 1)[1][0], test_Y[i])
```

ここまでに作成したクラスや関数を用いてニューラルネットワークを構築する関数である. __init__() でインスタンス化を行い, バッチサイズ batch_size, エポック epoch, 中間層のノード数 middle_layer, 出力層のノード数 (クラス数)last を指定する. learning() では 1 エポックを 60000/batch_size 回の繰り返しとし, パラメータの更新を実行する. testing() では保存された.npy ファイルから W1, W2, b1, b2 を読み込み, これらのパラメータを使ってテストデータの画像認識を行い, ニューラルネットワークの計算結果と正解のラベルを標準出力に出力する.

ソースコード 9: 課題 3 の実行

```
1   nn = neural_network(100, 100, 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()
```

neural_network クラスをインスタンス化し, 課題 3 を実行する.

実行結果

実行の結果, クロスエントロピー誤差が 2.296782644875204 から 0.2190637709925514 に減少し, 重み W1, W2 と切片ベクトル b1, b2 のファイルが作成された. さらに, これらのファイルは正常に読み込まれた.

工夫点

- この後の課題でも活用しやすくするために関数を機能別でクラスに集約し, 実装した.
- softmax 関数で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

- 課題 2 の時と比べるとクラスを活用することで改善されたことではあるが, バッチサイズを一箇所で管理できておらず, create_batch(X) 内の batch_size とは別で input_layer2(X) 内で input_vector を取得するためにバッチサイズである 100 をそのまま記述してしまっている. また, neural_network クラスでも改めて指定している.