課題 4 レポート

羽路 悠斗

2022 年 1 月 23 日

1　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せよ。

2　作成したプログラムの説明

2.1　ディレクトリ構造

モデルの構造やパラメータを変えて実験したかったので、ディレクトリを以下のように構造化する。history ディレクトリには、訓練ごとに全エポックの loss と acc が格納されている。model ディレクトリには、訓練ごとに最適なエポックの重みが格納されている。log ディレクトリにはログが格納されている。

ex4_im
|–contest.py
|–ex_advanced.py
|–history
| |–0000_history.dump
|–log
| |–general.log
|–logger.py
|–model
| |–0000.npz

2.2　設計方針

オプティマイザ、レイヤ、モデルをそれぞれクラス化する。これによりプログラムの可読性が上がり、拡張性や可変性も上がる。特にコンテストにおいては、実験を繰り返す必要があるので、以上の性能は重要である。

また基本的に $(batch\_size, height, width)$ の ndarray でデータを保持する。平坦化した際も、先頭の次元が batch_size を表す。なお、畳み込みそうなどでチャンネルを表す次元が必要になるときは $(batch\_size, channel, height, width)$ を使う。

2.3 各層の説明

2.3.1 ベース

まずは層のベースとなる Layer クラスをソースコード 1 に示す。全てのレイヤは Layer クラスのサブクラスとして実装する。クラス内の各関数について簡単に説明する。

- コンストラクタは、層の重みとオプティマイザを初期化する。
- forward 関数は前の層の出力を受け取り、この層の出力を返す。逆伝搬処理で使うことがあるので、入出力をクラス変数として保持しておく。学習時と推論時で異なる動作をする層があるので、フラグも受け取る。
- backward 関数は次の層の勾配を受け取り、この層の勾配を返す。重みの更新で使うことがあるので、勾配をクラス変数として保持しておく。
- update 関数はオプティマイザに重みの香辛料を計算させ更新する。
- get_weight 関数はこの層の重みを表す辞書を返す。
- load_weight 関数は全ての層の重みを表す辞書を受け取り、自身に当てはまる重みをロードする。

ソースコード 1　ex_advanced.py

```
1  class Layer:
2    def __init__(self, *args, **kwds):
3      pass
4
5    def forward(self, x, *args, **kwds):
6      self.x = x
7      self.y = x
8      return self.y
9
10   def backward(self, grad, *args, **kwds):
11     self.grad_x = grad
12     return self.grad_x
13
14   def update(self, *args, **kwds):
15     pass
16
17   def get_weight(self, *args, **kwds) -> tuple:
18     return {}
19
20   def load_weight(self, weight_dic):
21     pass
```

2.3.2 入力層

入力層は MNIST 画像を指定された形状に変形する。すなわち、1 次元化するか 2 次元のまま扱うかである。

- コンストラクタには、データの形を指定する。
- forward 関数で、画像データを指定された形に変形する。

ソースコード 2　ex_advanced.py

```
class Input(Layer):
  def __init__(self, output_shape: tuple):
    self.output_shape = output_shape
    self.output_dim = len(output_shape)

  def forward(self, x, batch_size=100, mode='train'):
    self.x = x
    if self.output_dim == 1: # 次元化 1
      self.y = x.reshape((batch_size,) + self.output_shape).T
    else: # 画像データをそのまま扱う場合を想定
      self.y = x.reshape((batch_size,) + self.output_shape)
```

2.3.3 全結合層

全結合層の出力は、重みベクトルを W、バイアスベクトルを B とおくと、次のように計算される。

$$WX + B$$

逆伝搬は、クロスエントロピー誤差の平均を E_n として、次のように計算される。

$$\frac{\delta E_n}{\delta X} = W^T \frac{\delta E_n}{\delta Y} \frac{\delta E_n}{\delta W} = \frac{\delta E_n}{\delta Y} X^T \frac{\delta E_n}{\delta B} = rowSum(\frac{\delta E_n}{\delta Y})$$

- コンストラクタには中間層のノード数、入力サイズ、層の名前、オプティマイザとその引数を指定し、それらと重みを初期化する。
- forward 関数で、numpy の行列積を用いて出力を計算する。
- backward 関数で、勾配を計算する。こちらも numpy の行列積を用いる。

ソースコード 3　ex_advanced.py

```
class Dense(Layer):
  def __init__(self, units: int, input_size: int, name: str='dense', opt: str='SGD',
      opt_kwds: dict={}):
    self.units = units

    self.w = random.normal(loc=0, scale=np.sqrt(1/input_size), size=input_size*units).
        reshape(units, input_size)
    self.b = random.normal(loc=0, scale=np.sqrt(1/input_size), size=units)
    self.name = name
```

```
8       self.opt = get_opt(opt, **opt_kwds)
9
10    def forward(self, x, batch_size=100, mode='train'):
11      self.x = x
12      self.batch_size = batch_size
13      self.y = self.w@self.x + self.b.reshape(-1, 1)
14      return self.y
15
16    def backward(self, grad):
17      self.grad_x = self.w.T@grad
18      self.grad_w = grad@self.x.T
19      self.grad_b = np.sum(grad, axis=1)
20      return self.grad_x
21
22    def update(self):
23      dw, db = self.opt.update(self.grad_w, self.grad_b)
24      self.w += dw
25      self.b += db
26
27    def get_weight(self) -> tuple:
28      return {f'{self.name}_w': self.w, f'{self.name}_b': self.b}
29
30    def load_weight(self, weight_dic):
31      self.w = weight_dic[f'{self.name}_w']
32      self.b = weight_dic[f'{self.name}_b']
```

2.3.4 畳み込み層

畳み込み層によって画像を 2 次元のまま扱えるので、精度向上が見込める。なお im2col と col2im を用いて画像と行列を変換することで処理を高速化できるが、ここでは詳細は省く。

順伝搬は、im2col で処理した入力を用いると簡単に計算できる。

$$WX + B$$

逆伝搬も、im2col で処理した入力を用いると簡単に計算できる。

$$\frac{\delta E_n}{\delta X} = W^T \frac{\delta E_n}{\delta Y} \frac{\delta E_n}{\delta W} = \frac{\delta E_n}{\delta Y} X^T \frac{\delta E_n}{\delta B} = rowSum(\frac{\delta E_n}{\delta Y})$$

ただし入力の勾配は、col2im を通してから逆伝搬させることに注意。

- コンストラクタには、入力の形、フィルタの形、フィルタの枚数、層の名前、オプティマイザとその引数を指定し、初期化する。
- さらに、フィルタの重みとバイアスを初期化する。その他、再利用する数は計算しておく。
- フィルタはあらかじめ行列で表しておく。
- im2col は画像を受け取り、便利な形の行列に変換する内部関数である。
- col2im は im2col の逆の動作をする。
- forward 関数で、im2col、畳み込み演算、形を標準に戻す、の順に処理する。
- backward 関数で、次の層の勾配をフィルタと対応する形の行列に変形、この層の勾配を計算、の順に処理する。

ソースコード 4　ex_advanced.py

```
1  class Conv(Layer):
2    def __init__(self, *, input_shape: tuple, filter_shape: tuple, filter_num: int, name:
         str='conv', opt: str='SGD', opt_kwds: dict={}):
3      self.im_h = input_shape[0]
4      self.im_w = input_shape[1]
5      self.filter_h = filter_shape[0]
6      self.filter_w = filter_shape[1]
7      self.pad_h = math.floor(self.filter_h/2)
8      self.pad_w = math.floor(self.filter_w/2)
9      self.o_h = self.im_h-self.filter_h+2*self.pad_h+1
10     self.o_w = self.im_w-self.filter_w+2*self.pad_w+1
11     self.filter_num = filter_num
12     self.w = 0.05*random.randn(self.filter_num, self.filter_h*self.filter_w)
13     self.b = 0.05*random.randn(self.filter_num)
14     self.name = name
15     self.opt = get_opt(opt, **opt_kwds)
16
17   def _im2col(self, im, batch_size):
18     pad_wid = ((0, 0), (self.pad_h, self.pad_h), (self.pad_w, self.pad_w))
19     pad_im = np.pad(im, pad_wid)
```

```
20      col = np.empty((batch_size, self.filter_h, self.filter_w, self.o_h, self.o_w)) # (
            batch size, filter height, filter width, output height, output width)
21      for h in range(self.filter_h):
22        for w in range(self.filter_w):
23          col[:, h, w, :, :] = pad_im[:, h : h+self.o_h, w : w+self.o_w]
24      col = col.transpose(1, 2, 0, 3, 4).reshape(self.filter_w*self.filter_h, batch_size*
            self.o_w*self.o_h)
25      return col
26
27    def _col2im(self, col, batch_size):
28      col = col.reshape(self.filter_h, self.filter_w, batch_size, self.o_h, self.o_w).
            transpose(2, 0, 1, 3, 4)
29      im = np.zeros((batch_size, self.im_h+2*self.pad_h, self.im_w+2*self.pad_w))
30      for h in range(self.filter_h):
31        for w in range(self.filter_w):
32          im[:, h : h+self.o_h, w : w+self.o_w] += col[:, h, w, :, :]
33      return im
34
35    def forward(self, x, batch_size=100, mode='train'):
36      self.x = self._im2col(x, batch_size)
37      self.batch_size = batch_size
38      self.y = self.w@self.x + self.b.reshape(-1, 1)
39      self.y = self.y.reshape(self.filter_num, batch_size, self.o_h, self.o_w).transpose
            (1, 0, 2, 3)
40      return self.y
41
42    def backward(self, grad):
43      grad = grad.transpose(1, 0, 2, 3).reshape(self.filter_num, self.batch_size*self.o_h
            *self.o_w)
44      self.grad_x = self._col2im(self.w.T@grad, self.batch_size)
45      self.grad_w = grad@self.x.T
46      self.grad_b = np.sum(grad, axis=1)
47      return self.grad_x
48
49    def update(self):
50      dw, db = self.opt.update(self.grad_w, self.grad_b)
51      self.w += dw
52      self.b += db
53
54    def get_weight(self) -> tuple:
55      return {f'{self.name}_w': self.w, f'{self.name}_b': self.b}
56
57    def load_weight(self, weight_dic):
58      self.w = weight_dic[f'{self.name}_w']
59      self.b = weight_dic[f'{self.name}_b']
```

2.3.5 プーリング

プーリング層は畳み込み層で捉えられた特徴を「ぼかす」働きがある。具体的には 2*2max プーリングだと 2*2 の領域で最大のものを出力にする。これも処理を高速化するために im2col と col2im を用いる。畳み込みそうで用いられるものとは行列の形が少し異なる。

逆伝搬は、順伝搬時にプーリングで抽出されたインデックスに勾配を当てはめて、その他の勾配は 0 として伝搬する。

- コンストラクタには、入力の形とチャンネル数とプーリング層のサイズを指定する。
- forward 関数で、im2col、プーリングを順に処理する。逆伝播のために、プーリングで抽出されたインデックスを保持しておく。
- backward 関数で、勾配を行列で表してから、col2im を通す。

ソースコード 5　ex_advanced.py

```
1  class Pooling(Layer):
2    def __init__(self, *, input_shape: tuple, channel: int, pool_shape: tuple):
3      self.im_h = input_shape[0]
4      self.im_w = input_shape[1]
5      self.channel = channel
6      self.pool_h = pool_shape[0]
7      self.pool_w = pool_shape[1]
8      self.o_h = self.im_h//self.pool_h
9      self.o_w = self.im_w//self.pool_w
10
11   def _im2col(self, im, batch_size):
12     col = np.empty((batch_size, self.channel, self.pool_h, self.pool_w, self.o_h, self.
           o_w))
13     for h in range(self.pool_h):
14       for w in range(self.pool_w):
15         col[:, :, h, w, :, :] = im[:, :, h : h+self.pool_h*self.o_h : self.pool_h, w
               : w+self.pool_w*self.o_w : self.pool_w]
16     col = col.transpose(0, 1, 4, 5, 2, 3).reshape(batch_size*self.channel*self.o_h*
           self.o_w, self.pool_h*self.pool_w)
17     return col
18
19   def _col2im(self, col, batch_size):
20     col = col.reshape(self.batch_size, self.channel, self.o_h, self.o_w, self.pool_h,
           self.pool_w).transpose(0, 1, 4, 5, 2, 3)
21     im = np.zeros((batch_size, self.channel, self.im_h, self.im_w))
22     for h in range(self.pool_h):
23       for w in range(self.pool_w):
24         im[:, :, h : h+self.pool_h*self.o_h : self.pool_h, w : w+self.pool_w*self.o_w
               : self.pool_w] += col[:, :, h, w, :, :]
25     return im
```

```
26
27   def forward(self, x, batch_size=100, mode='train'):
28     self.x = self._im2col(x, batch_size)
29     self.batch_size = batch_size
30     self.ind = np.argmax(self.x, axis=1)
31     self.y = np.max(self.x, axis=1).reshape((batch_size, self.channel, self.o_h, self.
           o_w))
32     return self.y
33
34   def backward(self, grad):
35     self.grad_x = np.zeros((self.batch_size*self.channel*self.o_h*self.o_w, self.pool_h*
           self.pool_w))
36     self.grad_x[np.arange(self.ind.size), self.ind] = grad.flatten()
37     self.grad_x = self._col2im(self.grad_x, self.batch_size)
38     return self.grad_x
```

2.3.6 平坦化層

Flatten 層は、画像を全結合層の入力にできるように平坦化する。順伝搬は、バッチ次元はそのままで、他の次元を平坦化する。逆伝搬は、順伝播と逆の変形をする。

ソースコード 6　ex_advanced.py

```
1  class Flatten(Layer):
2    def __init__(self):
3      pass
4
5    def forward(self, x, batch_size=100, mode='train'):
6      self.x = x
7      self.batch_size = batch_size
8      self.y = x.reshape((batch_size, -1)).T
9      return self.y
10
11   def backward(self, grad):
12     self.grad_x = grad.T.reshape((self.batch_size, *self.x.shape[1:]))
13     return self.grad_x
```

2.3.7 シグモイド

シグモイド関数は、活性化関数の一種で、値を 0〜1 の範囲に変換する。順伝搬は次の式で表せる。なお、シグモイド関数はこの式からわかるように、絶対値の大きな数を扱うことが苦手である。そのため、後述する正規化が望ましい。

$$\frac{1}{1+exp(-x)}$$

逆伝搬は次の式で表せる。

$$\frac{\delta E_n}{\delta x} = \frac{\delta E_N}{\delta y}(1-y)y$$

- forward 関数で、オーバーフローに対処するために閾値を超えた値を丸める。

ソースコード 7　ex.advanced.py

```
1  class Sigmoid(Layer):
2    def __init__(self):
3      pass
4
5    def forward(self, x, batch_size=100, mode='train'):
6      sigmoid_range = 34.538776394910684
7      x = np.clip(x, -sigmoid_range, sigmoid_range)
8      self.x = x
9      self.y = 1/(1.0+np.exp(-x))
10     return self.y
11
12   def backward(self, grad):
13     self.grad_x = grad*(1-self.y)*self.y
14     return self.grad_x
```

2.3.8 ReLU

ReLU 関数は、活性化関数の 1 種で、0 以下の値を 0 に丸める。負の値をノイズと見做していると考えることができ、画像は値が正なので相性がいいとされる。

$$a(t) = \begin{cases} t(t>0) \\ 0(t \leq 0) \end{cases}$$

勾配は次の通り。

$$a(t)' = \begin{cases} 1(t>0) \\ 0(t \leq 0) \end{cases}$$

ソースコード 8　ex_advanced.py

```
1  class ReLU(Layer):
2    def __init__(self):
```

```
    pass

  def forward(self, x, batch_size=100, mode='train'):
    self.x = x
    self.y = np.where(x>0, x, 0)
    return self.y

  def backward(self, grad):
    self.grad_x = grad*np.where(self.x>0, 1, 0)
    return self.grad_x
```

2.3.9 バッチノーマリゼーション

バッチノーマリゼーションは、各ノードのミニバッチの出力が分散 1、平均 0 となるように正規化する処理である。順伝播と逆伝播の式の記述は、講義資料に任せる。コードの量は多いが、実装に難しい点はなく、式を愚直にプログラムに変換するのみである。1 点、numpy で sum をとると次元が小さくなるので、大きさ 1 の次元を np.expand_dims() で補完することに注意。

ソースコード 9　ex_advanced.py

```
class BatchNormalization(Layer):
  def __init__(self, units: int=96, name: str='bn', opt: str='SGD', opt_kwds: dict={})
      :
    self.name = name
    self.opt = get_opt(opt, **opt_kwds)
    self.gamma = np.ones((units,1))
    self.beta = np.zeros((units,1))
    self.mean_list = []
    self.variance_list = []

  def forward(self, x, batch_size=100, mode='train'):
    if mode == 'train':
      self.x = x
      self.batch_size = batch_size
      self.mean = np.expand_dims(np.sum(x, axis=1)/self.batch_size, axis=-1)
      self.variance = np.expand_dims(np.sum(np.square(x-self.mean), axis=1)/self.
          batch_size, axis=-1)
      self.x_normalized = (x-self.mean)/np.sqrt(self.variance+sys.float_info.epsilon)
      self.y = self.gamma*self.x_normalized + self.beta
      self.mean_list.append(self.mean)
      self.variance_list.append(self.variance)
      self.mean_expected = np.average(np.array(self.mean_list), axis=0)
      self.variance_expected = np.average(np.array(self.variance_list), axis=0)
      return self.y
    elif mode == 'inference':
      return self.gamma/np.sqrt(self.variance_expected+sys.float_info.epsilon)*x + (self
          .beta - self.gamma*self.mean_expected/np.sqrt(self.variance_expected+sys.
```

```
            float_info.epsilon))
25      else:
26        raise ValueError('mode␣is␣train␣or␣inference.')
27
28    def backward(self, grad):
29      self.grad_x_normalized = grad*self.gamma
30      self.grad_variance = np.expand_dims(np.sum(grad*(self.x-self.mean)*(-1)/2*np.power(
            self.variance+sys.float_info.epsilon, -3/2), axis=1), axis=-1)
31      self.grad_mean = np.expand_dims(np.sum(self.grad_x_normalized*(-1)/np.sqrt(self.
            variance+sys.float_info.epsilon), axis=1), axis=-1) + self.grad_variance*np.
            expand_dims(np.sum(-2*(self.x-self.mean), axis=1)/self.batch_size, axis=-1)
32      self.grad_x = self.grad_x_normalized/np.sqrt(self.variance+sys.float_info.epsilon) +
             self.grad_variance*2*(self.x-self.mean)/self.batch_size + self.grad_mean/self.
            batch_size
33      self.grad_gamma = np.expand_dims(np.sum(grad*self.x_normalized, axis=1), axis=-1)
34      self.grad_beta = np.expand_dims(np.sum(grad, axis=1), axis=-1)
35      return self.grad_x
36
37    def update(self):
38      dgamma, dbeta = self.opt.update(self.grad_gamma, self.grad_beta)
39      self.gamma += dgamma
40      self.beta += dbeta
41
42    def get_weight(self) -> tuple:
43      return {f'{self.name}_gamma': self.gamma, f'{self.name}_beta': self.beta}
44
45    def load_weight(self, weight_dic):
46      self.gamma = weight_dic[f'{self.name}_gamma']
47      self.beta = weight_dic[f'{self.name}_beta']
```

2.3.10 ドロップアウト

ドロップアウトは、層のノードをランダムに無視する手法である。過学習の抑制に効果的だとされる。

- forward 関数で、フラグによって学習時と推論時で動作を変えている。

$$a(t) = \begin{cases} t(\text{無視されない場合}) \\ 0(\text{無視される場合}) \end{cases}$$

勾配は次の通り。

$$a(t)' = \begin{cases} 1(\text{無視されない場合}) \\ 0(\text{無視される場合}) \end{cases}$$

ソースコード 10　ex_advanced.py

```
1 class Dropout(Layer):
2   def __init__(self, dropout: float=0.1):
```

```
    self.dropout = dropout

  def forward(self, x, batch_size=100, mode='train'):
    self.x = x
    if mode == 'train':
      self.mask = (np.random.rand(*self.x.shape) >= self.dropout)
      self.y = x*self.mask
      return self.y
    elif mode == 'inference':
      return x*(1-self.dropout)
    else:
      raise ValueError('mode is train or inference.')

  def backward(self, grad):
    self.grad_x = grad*self.mask
    return self.grad_x
```

2.3.11 ソフトマックス

ソフトマックス関数は多クラス分類で出力層の活性化関数に用いられる。各クラスに属する尤度を表す。αはオーバーフローを防ぐために導入する。

$$y_i = \frac{exp(x_i - \alpha)}{\Sigma_{j=1}^{C} exp(a_j - \alpha)}$$

$$\alpha = max\, a_i$$

逆伝搬は次の式で表せる。

$$\frac{\delta E_n}{\delta x_i} = \frac{y_i - y\_true_i}{batch\_size}$$

ソースコード 11　ex_advanced.py

```
class Softmax(Layer):
  def __init__(self):
    pass

  def forward(self, x, batch_size=100, mode='train'):
    self.batch_size = batch_size
    self.x = x
    self.y = np.exp(x-np.max(x, axis=0)) / (np.sum(np.exp(x-np.max(x, axis=0)), axis
        =0))
    return self.y

  def backward(self, true_y):
    grad_x = (self.y - true_y.T)/self.batch_size
    return grad_x
```

2.4 最適化

オプティマイザは、実装が単純なので、ベースクラスは作成していない。コンストラクタで、学習率などのパラメータを初期化する。update 関数で、重みとバイアスの勾配を受け取って、更新量を返す。それぞれの特性や式は、講義資料に説明があるので省略する。

2.4.1 SGD

ソースコード 12 ex_advanced.py

```
class SGD:
  def __init__(self, lr: float=0.01):
    self.lr = lr

  def update(self, grad_w, grad_b):
    dw = -self.lr*grad_w
    db = -self.lr*grad_b
    return dw, db
```

2.4.2 慣性項付き SGD

ソースコード 13 ex_advanced.py

```
class MomentumSGD:
  def __init__(self, lr: float=0.01, alpha: float=0.9):
    self.lr = lr
    self.alpha = alpha

  def update(self, grad_w, grad_b):
    self.dw = self.alpha*self.dw - self.lr*grad_w
    self.db = self.alpha*self.db - self.lr*grad_b
    return self.dw, self.db
```

2.4.3 AdaGrad

ソースコード 14 ex_advanced.py

```
class AdaGrad:
  def __init__(self, lr: float=0.001, h0: float=1e-8):
    self.lr = lr
    self.h_w = h0
    self.h_b = h0

  def update(self, grad_w, grad_b):
    self.h_w = self.h_w + grad_w*grad_w
    self.h_b = self.h_b + grad_b*grad_b
```

```
10      dw = -self.lr/np.sqrt(self.h_w)*grad_w
11      db = -self.lr/np.sqrt(self.h_b)*grad_b
12      return dw, db
```

2.4.4 RMSProp

ソースコード 15　ex_advanced.py

```
1  class RMSProp:
2    def __init__(self, lr: float=0.001, rho: float=0.9, epsilon: float=1e-8, h0: float=1
         e-8):
3      self.lr = lr
4      self.rho = rho
5      self.epsilon = epsilon
6      self.h_w = h0
7      self.h_b = h0
8
9    def update(self, grad_w, grad_b):
10     self.h_w = self.rho*self.h_w + (1-self.rho)*grad_w*grad_w
11     self.h_b = self.rho*self.h_b + (1-self.rho)*grad_b*grad_b
12     dw = -self.lr/np.sqrt(self.h_w)*grad_w
13     db = -self.lr/np.sqrt(self.h_b)*grad_b
14     return dw, db
```

2.4.5 AdaDelta

ソースコード 16　ex_advanced.py

```
1  class AdaDelta:
2    def __init__(self, rho: float=0.95, epsilon: float=1e-6):
3      self.rho = rho
4      self.epsilon = epsilon
5      self.h_w = 0
6      self.h_b = 0
7      self.s_w = 0
8      self.s_b = 0
9
10   def update(self, grad_w, grad_b):
11     self.h_w = self.rho*self.h_w + (1-self.rho)*grad_w*grad_w
12     self.h_b = self.rho*self.h_b + (1-self.rho)*grad_b*grad_b
13     dw = -np.sqrt(self.s_w+self.epsilon)/np.sqrt(self.h_w+self.epsilon)*grad_w
14     db = -np.sqrt(self.s_b+self.epsilon)/np.sqrt(self.h_b+self.epsilon)*grad_b
15     self.s_w = self.rho*self.s_w + (1-self.rho)*dw*dw
16     self.s_b = self.rho*self.s_b + (1-self.rho)*db*db
17     return dw, db
```

2.4.6 Adam

ソースコード 17　ex_advanced.py

```
class Adam:
  def __init__(self, alpha: float=0.001, beta_1: float=0.9, beta_2: float=0.999,
      epsilon: float=1e-8):
    self.alpha = alpha
    self.beta_1 = beta_1
    self.beta_2 = beta_2
    self.epsilon = epsilon
    self.t = 0
    self.m_w = 0
    self.m_b = 0
    self.v_w = 0
    self.v_b = 0

  def update(self, grad_w, grad_b):
    self.t = self.t+1
    self.m_w = self.beta_1*self.m_w + (1-self.beta_1)*grad_w
    self.m_b = self.beta_1*self.m_b + (1-self.beta_1)*grad_b
    self.v_w = self.beta_2*self.v_w + (1-self.beta_2)*grad_w*grad_w
    self.v_b = self.beta_2*self.v_b + (1-self.beta_2)*grad_b*grad_b
    self.m_w_hat = self.m_w/(1-np.power(self.beta_1, self.t))
    self.m_b_hat = self.m_b/(1-np.power(self.beta_1, self.t))
    self.v_w_hat = self.v_w/(1-np.power(self.beta_2, self.t))
    self.v_b_hat = self.v_b/(1-np.power(self.beta_2, self.t))
    dw = -self.alpha*self.m_w_hat/(np.sqrt(self.v_w_hat)+self.epsilon)
    db = -self.alpha*self.m_b_hat/(np.sqrt(self.v_b_hat)+self.epsilon)
    return dw, db
```

2.5 補助関数

one hot vector 表記に変換する補助関数を定義する。

ソースコード 18　ex_advanced.py

```
def to_one_hot_vector(labels):
  l = [[1 if i == label else 0 for i in range(10)] for label in labels]
  ohv = np.zeros((len(l), len(l[0])))
  ohv[:] = l
  return ohv
```

精度を計算する補助関数を定義する。正解の one hot vector 表記には非対応である。万が一、予測と正解の形が不正であれば、エラーを出す。

ソースコード 19　ex_advanced.py

```
def accuracy(true_y, pred_y): # ラベル表示に対応、one-hot に非対応 vector
  if true_y.shape != pred_y.shape:
    raise ValueError(f'true_y␣and␣pred_y␣must␣have␣the␣same␣shape.␣{true_y.shape},␣{
        pred_y.shape}')
  return np.count_nonzero(true_y == pred_y) / true_y.shape[0]
```

クロスエントロピー誤差を計算する補助関数を定義する。正解の one hot vector 表記に対応している。

ソースコード 20　ex_advanced.py

```
def cross_entropy(true_vec ,pred_vec): # one-hot に対応 vector
  return np.sum(np.sum(-1 * true_vec * np.log(pred_vec), axis=1)) / true_vec.shape[0]
```

グラフのプロットに用いるために、移動平均を計算する補助関数を定義する。

ソースコード 21　ex_advanced.py

```
def moving_average_curve(wave, window: int=2):
  mac = []
  for i in range(len(wave)):
    if i < window-1:
      mac.append(np.mean(wave[0:i+1]))
    else:
      mac.append(np.mean(wave[i-window+1:i+1]))
  return mac
```

2.6 モデル

2.6.1 Model クラス

モデルクラスを作成する。層のインスタンスのリストをインスタンス変数 layers に、層の重みを値、層の名前をキーとする辞書をインスタンス変数 weights_dic に保持する。

ソースコード 22　ex_advanced.py

```
class Model:
  def __init__(self, mode = 'train') -> None:
    self.batch_size = 100
    if mode not in ['train', 'inference']:
      raise ValueError('mode␣is␣train␣or␣inference.')
    self.mode = mode
    self.layers = []
    self.weights_dic = {}
```

モデルに層を追加していく add 関数を定義する。

ソースコード 23　ex_advanced.py

```
  def add(self, layer: Layer):
      self.layers.append(layer)
      self.weights_dic.update(layer.get_weight())
```

データセットからミニバッチを作成する前処理、尤度最大のクラスを取り出す後処理を定義する。

ソースコード 24　ex_advanced.py

```
  def preprocessing(self, train_x, train_y):
    '''学習データセットからバッチを作成する'''
    idx = random.randint(0, len(train_y), self.batch_size)
    tr_x = train_x[idx]
    tr_y = train_y[idx]
    return tr_x, tr_y # tr_x: (bs, 28, 28)

  def postprocessing(self, input_vec): # ミニバッチに対応、bs*class
    return np.argmax(input_vec, axis=1)
```

2.6.2 学習と検証

※この項目のプログラムは分量が多いです。ソースコード 28 を見ると概要が掴めます。

ミニバッチの処理を行う train_batch, valid_batch 関数を定義する。どちらも層の forward 関数を順に呼び出して順伝播を実装する。損失と精度を計算する。train_batch 関数では、層の backward 関数を逆順に呼び出して勾配を計算し、update 関数を呼び出して重みを更新する。

ソースコード 25　ex_advanced.py

```
  def valid_batch(self, val_x, val_y, lr):
```

```
    self.mode = 'inference'
    flag_input = True
    for layer in self.layers:
      if flag_input == True:
        x = val_x
        flag_input = False
      x = layer.forward(x, self.batch_size, self.mode)

    pred_y = self.layers[-1].y.T
    entropy = cross_entropy(to_one_hot_vector(val_y), pred_y)
    pred_y = self.postprocessing(pred_y)
    acc = accuracy(val_y, pred_y)

    return entropy, acc
```

ソースコード 26　ex_advanced.py

```
def train_batch(self, tr_x, tr_y, lr):
  self.mode = 'train'
  flag_input = True
  for layer in self.layers:
    if flag_input == True:
      x = tr_x
      flag_input = False
    x = layer.forward(x, self.batch_size, self.mode)

  pred_y = self.layers[-1].y.T
  entropy = cross_entropy(to_one_hot_vector(tr_y), pred_y)
  pred_y = self.postprocessing(pred_y)
  acc = accuracy(tr_y, pred_y)

  flag_output = True
  for layer in reversed(self.layers):
    if flag_output == True:
      grad = to_one_hot_vector(tr_y)
      flag_output = False
    grad = layer.backward(grad)

  for layer in self.layers:
    layer.update()

  weights_dic = {}
  for layer in self.layers:
    weights_dic.update(layer.get_weight())
  return weights_dic, entropy, acc
```

上記の関数を利用して学習と検証を行う。valid 関数は 1 エポック分の検証を行う。

ソースコード 27　ex_advanced.py

```
def valid(self, valid_x, valid_y, lr: float):
  entropies = []
  accuracies = []
  for i in range(valid_y.shape[0]//self.batch_size):
    val_x, val_y = self.preprocessing(valid_x, valid_y)
    entropy, acc = self.valid_batch(val_x, val_y, lr)
    entropies.append(entropy)
    accuracies.append(acc)
  entropy = sum(entropies)/len(entropies)
  acc = sum(accuracies)/len(accuracies)
  return entropy, acc
```

train 関数は指定された epoch 数の学習と検証を行う。検証は valid 関数を呼び出す。エポックごとの重み、損失、精度を辞書で残しておく。

ソースコード 28　ex_advanced.py

```
def train(self, train_x, train_y, valid_x=None, valid_y=None, valid: int=0, epochs:
    int = 10, lr: float = 0.01):
  '''
  train_x: 学習データの特徴量
  train_y: 学習データのラベル
  valid_x: バリデーションデータの特徴量
  valid_y: バリデーションデータのラベル
  valid: バリデーションの間隔 (0: バリデーションしない)
  epochs: エポック数
  lr: 学習率
  '''
  self.batch_size = 100
  history = []
  for i in tqdm(range(epochs)):
    entropies = []
    accuracies = []
    for j in range(60000//self.batch_size):
      tr_x, tr_y = self.preprocessing(train_x, train_y)
      self.weights_dic, entropy, acc = self.train_batch(tr_x, tr_y, lr)
      entropies.append(entropy)
      accuracies.append(acc)
    entropy = sum(entropies)/len(entropies)
    acc = sum(accuracies)/len(accuracies)

    # TODO: をので初期化しておき、リストに要素を追加 historylistdict
    if valid == 1:
      val_entropy, val_acc = self.valid(valid_x, valid_y, lr)
      print(f'Epoch {i+1}, loss: {entropy:.5f}, acc: {acc:.5f}, val_loss: {val_entropy
          :.5f}, val_acc: {val_acc:.5f}')
```

```
28          history.append((self.weights_dic, entropy, acc, val_entropy, val_acc))
29        elif valid >= 2:
30          if i%valid == 0:
31            val_entropy, val_acc = self.valid(valid_x, valid_y, lr)
32            print(f'Epoch␣{i+1},␣loss:␣{entropy:.5f},␣acc:␣{acc:.5f},␣val_loss:␣{
                  val_entropy:.5f},␣val_acc:␣{val_acc:.5f}')
33          else:
34            print(f'Epoch␣{i+1},␣loss:␣{entropy:.5f},␣acc:␣{acc:.5f}')
35          history.append((self.weight_dic, entropy, acc))
36        else:
37          print(f'Epoch␣{i+1},␣loss:␣{entropy:.5f},␣acc:␣{acc:.5f}')
38          history.append((self.weights_dic, entropy, acc))
39
40      history = dict(zip(['weight', 'loss', 'acc', 'val_loss', 'val_acc'], list(zip(*
            history))))
41      return history
```

2.6.3 推論

推論もほぼ上記と変わらないが、インスタンス変数 mode を inference に指定することと、バッチ処理を行わないこと等が異なる点である。

ソースコード 29　ex_advanced.py

```
1   def predict(self, test_x, test_y=None, valid=False, postprocess=True):
2     '''
3     train_x: テストデータの特徴量
4     test_y(option): テストデータのラベル
5     valid: テストデータのを計算するかどうかのフラグ loss
6     '''
7     self.mode = 'inference'
8     self.batch_size = 1 # 推論時はバッチ処理を行わない
9     if valid == False:
10      pred_y = []
11      for test_i in (test_x):
12        flag_input = True
13        for layer in self.layers:
14          if flag_input == True:
15            x = test_i
16            flag_input = False
17          x = layer.forward(x, self.batch_size, self.mode)
18        pred_y.append(self.layers[-1].y)
19      pred_y = np.array(pred_y)
20      if postprocess:
21        pred_y = self.postprocessing(pred_y)
22      return pred_y
23    elif valid == True:
```

```
      test_y = to_one_hot_vector(test_y)
      pred_y = []
      for test_i in (test_x):
        flag_input = True
        for layer in self.layers:
          if flag_input == True:
            x = test_i
            flag_input = False
          x = layer.forward(x, self.batch_size, self.mode)
        pred_y.append(self.layers[-1].y.flatten())
      pred_y = np.array(pred_y)
      entropy = cross_entropy(test_y, pred_y)
      if postprocess:
        pred_y = self.postprocessing(pred_y)
      return pred_y, entropy
    else:
      raise ValueError(f'please␣set␣bool␣not␣{valid}␣for␣valid.')
```

2.6.4 モデルと実行結果の保存、読み込み

検証データの損失に基づいて、インスタンス変数 weights_dic を、最適なエポックの重みに巻き戻すのが、load_best 関数である。

ソースコード 30　ex_advanced.py

```
def load_best(self, history):
  if 'val_loss' in history:
    epoch = np.nanargmin(history['val_loss'])
  else:
    epoch = np.nanargmin(history['loss'])
  print(f'back␣to␣epoch␣{epoch+1}') # はインデックスで保持している epoch
  self.weight_dic = history['weight'][epoch]
  for layer in self.layers:
    layer.load_weight(self.weights_dic)
```

現在の重みを保存するのが、save_model 関数である。load_best 関数を呼んでから使うと良い。

ソースコード 31　ex_advanced.py

```
def save_model(self, name):
  np.savez(f'model/{name}', **self.weights_dic)
```

保存された重みを読み込んで、層の名前でマッチングして読み込むのが、load_model 関数である。

ソースコード 32　ex_advanced.py

```
def load_model(self, name):
  print(f'loading␣{name}...')
  self.weights_dic = np.load(f'model/{name}.npz')
  for layer in self.layers:
```

```
5       layer.load_weight(self.weights_dic)
```

学習時に残しておいたエポックごとの重み、損失、精度が入った辞書を pickle で保存する。

ソースコード 33　ex_advanced.py

```
1   def save_history(self, name, history):
2     with open(f'history/{name}_history.dump', 'wb') as f:
3       pickle.dump(history, f)
```

2.7　ログ

後からモデルの損失や精度を比較できるようにログを残す。Logger クラスを読み込んで使う。コンストラクタでファイル名を指定し、ストリーミング形式で書き込んでいく。

ソースコード 34　logger.py

```
1  import logging
2
3  class Logger:
4    def __init__(self, filename='general.log'):
5      self.general_logger = logging.getLogger('general')
6      stream_handler = logging.StreamHandler()
7      file_general_handler = logging.FileHandler(f'./log/{filename}')
8
9      if len(self.general_logger.handlers) == 0:
10       self.general_logger.addHandler(stream_handler)
11       self.general_logger.addHandler(file_general_handler)
12       self.general_logger.setLevel(logging.INFO)
13
14   def info(self, message):
15     self.general_logger.info(message)
```

2.8 実行結果

実行スクリプトは次に示す通りである。実行の名前とメッセージを求める。データはニューラルネットで扱いやすいように、0 1 に正規化する。訓練を終えたら、損失と精度、最適なエポックの重みを保存する。モデルは、add 関数で層を追加して定義する。最適なエポックの重みを読み込んだモデルでの、損失と精度をログに残す。

```
# --- 実行---
if __name__ == '__main__':
  print('please␣input␣model␣name.␣(ex:␣0001)')
  run_name = input()
  print('please␣input␣message␣about␣this␣run.')
  run_msg = input()

  # --- データをロード---
  logger = Logger()
  train_x = mnist.download_and_parse_mnist_file('train-images-idx3-ubyte.gz')
  train_y = mnist.download_and_parse_mnist_file("train-labels-idx1-ubyte.gz")
  test_x = mnist.download_and_parse_mnist_file('t10k-images-idx3-ubyte.gz')
  test_y = mnist.download_and_parse_mnist_file('t10k-labels-idx1-ubyte.gz')
  # 正規化
  train_x = train_x.astype('float32')/255.0
  test_x = test_x.astype('float32')/255.0

  # --- モデルを定義---
  model = Model(mode='train')
  model.add(Input((28, 28)))
  model.add(Conv(input_shape=(28, 28), filter_shape=(5, 5), filter_num=32, opt='
      Adam', opt_kwds={}))
  model.add(ReLU())
  model.add(Pooling(input_shape=(28, 28), channel=32, pool_shape=(2, 2)))
  model.add(Flatten())
  model.add(Dropout(dropout=0.4))
  model.add(Dense(10, 14*14*32, opt='Adam', opt_kwds={}))
  model.add(Softmax())

  # --- 訓練---
  # model.load_model('0012')
  history = model.train(train_x, train_y, test_x, test_y, valid=1, epochs=30)
  model.load_best(history)
  if run_name != '':
    model.save_history(run_name, history)
    model.save_model(run_name)

```

```
37  # --- 推論---
38  pred_y, val_entropy = model.predict(test_x, test_y, valid=True)
39  print(pred_y)
40
41  # --- ログ出力---
42  if run_name != '':
43    logger.info(run_name)
44    logger.info(run_msg)
45    logger.info(f'val_loss:␣{val_entropy}')
46    logger.info(f'val_acc:␣{accuracy(test_y,␣pred_y)}')
47    logger.info('')
48  else:
49    print(f'val_loss␣{val_entropy}')
50    print(f'val_acc␣{accuracy(test_y,␣pred_y)}')
```

結果は次の通りである。

1 : 実行の名前とメッセージを入力

2 : 学習時の損失と精度とプログレスバーを表示

3 : ログ出力

2.9 プロット

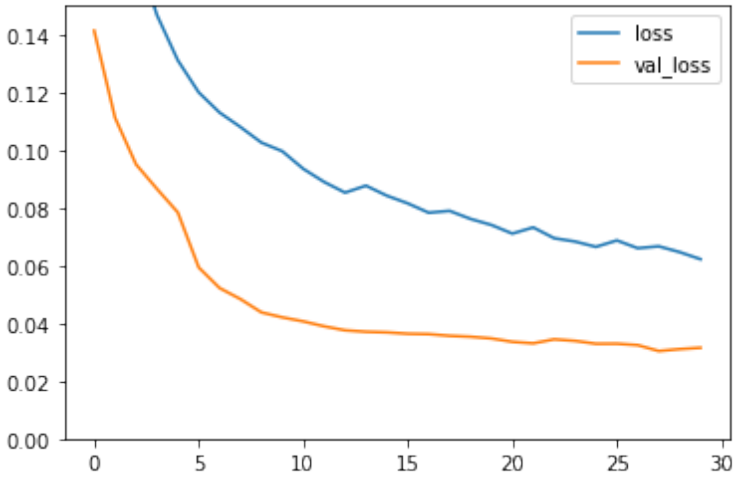
実行を終えると、エポック毎の損失と誤差のグラフを表示する。滑らかに描画するために、サイズ 2 の移動平均を使っている。

ソースコード 35　ex_advanced.py

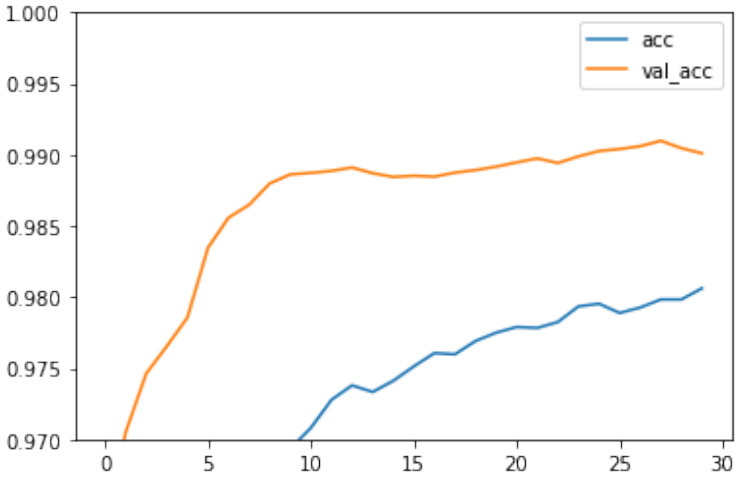
```
1  figure = plt.figure()
2  plt.plot(history['loss'], label='loss')
3  if 'val_loss' in history:
4    plt.plot(moving_average_curve(history['val_loss'], 2), label='val_loss')
5  plt.legend()
6  plt.ylim((0, 0.15))
7  plt.show()
8
9  figure = plt.figure()
```

```
10  plt.plot(history['acc'], label='acc')
11  if 'val_acc' in history:
12    plt.plot(moving_average_curve(history['val_acc'], 2), label='val_acc')
13  plt.legend()
14  plt.ylim((0.97, 1))
15  plt.show()
```

なお、学習後に損失と重みを pickle で保存しているので後からグラフの表示も可能である。plot.ipynb に実装している。損失のグラフの実例を示す。



精度のグラフの実例を示す。



3 実験

比較実験は様々行なったが、得られた主要なポイントを示す。

- 畳み込み層は非常に大きく精度を向上させる。
- ネットワークの構造を大きくすると、MNIST 程度のタスクだとすぐに過学習してしまう。
- 重みの初期値が大きいと、活性化関数でオーバーフローが起きやすい。
- ドロップアウトは過学習を抑制する。
- バッチノーマリゼーションとドロップアウトを同時に用いることは、必ずしも精度向上に寄与しない。
- プーリング層は必ずしも精度向上に寄与しない。
- 画像の正規化によって、活性化関数でオーバーフローが起きにくくなる
- Adam の方が高い精度を出すこともあれば、SGD と変わらないこともある。Adam は局所解を防ぐので、重みの初期値の影響を受けにくいからだと考えた。逆に言えば、初期値をうまく設定すれば、SGD も良い精度を出す。

実験の結果、検証データに対して単独で最も高い精度を出したモデルを示す。このモデルは、損失 0.03201、精度 0.99 を記録した。

Layer	Output	Parameter
Input	(BS, 28, 28)	output_shape=(28, 28)
Conv	(BS, 32, 28, 28)	input_shape=(28, 28), filter_shape=(5, 5), filter_num=32, opt='Adam', opt_kwds={}
ReLU	(BS, 32, 28, 28)	
Pooling	(BS, 32, 14, 14)	input_shape=(28, 28), channel=32, pool_shape=(2, 2)
Flatten	(BS, 6272(=32*14*14))	
Dropout	(BS, 6272)	dropout=0.4
Dense	(BS, 10)	units=10, input_size=32*14*14, opt='Adam', opt_kwds={}
Softmax	(BS, 10)	

4 コンテスト

学習データや検証データで良い精度を出すモデルであっても、コンテストのデータではあまり精度が出ない。これはコンテストのデータは学習データから特徴を捉えきれない、要は難しいデータであるからだと考えられる。そこで戦術として、

1. モデルの構造とパラメータをチューニングして可能な限りスコアを高める
2. データ拡張
3. アンサンブル

を用いて、特徴の取りこぼしを防ぐ。データ拡張は平行移動のみを用いた。アンサンブルについてはルールが曖昧だったが、アンサンブルによる精度向上が 1.9% であったことをここに記しておく。

5　工夫点

工夫点は数多くあるので、プログラムとともに詳細を説明してきた。特に力を入れて工夫したのが、コンテストに向けて比較実験を行いやすいように、モデルの拡張性、可変性を高めた点と、学習時の損失、精度の保存と、ログの出力である。

6　問題点

やれるだけのことをやり切った自信はあるが、強いて言うならば、データ拡張の種類をもう少し増やすことはできたかもしれない。

7　添付について

ディレクトリ構造を工夫したことを含めて見ていただきたいので、本実験のディレクトリごと添付した。主となるファイルは ex_advanced.py logger.py, plot.ipynb である。

8　感想

非常に楽しく、また勉強になりました。データ分析に興味があって、Keras や PyTorch を使っているものの、ニューラルネットの仕組みをきちんと理解できたのは、この実験のおかげです。畳み込みでうまくテンソル演算を組み合わせて for 分を減らすために、im2col と col2im が編み出されたことには感動しました。コンテスト暫定 1 位も非常にうれしく思っています。