性能評価

tkt914

2020 年 3 月 5 日

1 practice/practice1.py

1.1 内容

MNIST の画像 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する。

1.2 作成したプログラムの説明

作成したメソッドの意味を以下の表 1 に示す。

表 1: 作成したメソッドの説明

メソッド	説明
init_network()	初期化を行う。重みとバイアスは乱数で決定した。
forward(network, x)	順伝播で伝わっていく流れを行う。 引数の network には用いる重みやバイアスなどの情報が入っている。 x は画像データ。

common ディレクトリの function.py ファイルの中に sigmoid 関数と softmax 関数を表すメソッドを作成した。それらのメソッドの意味を以下の表 2 に示す。

表 2: 用いたメソッドの説明

メソッド	説明
sigmoid(x)	シグモイド関数である。
softmax(x)	ソフトマックス関数である。 バッチに対応するべく二次元のデータが渡された時にも対応するようにした。

各ノード数を定数として定義する。次に、用いる画像データを読み込む。次に、0 から 9999 までの数値を入力として受け取り、その番号にある画像データと用いる重みやバイアスが入ったデータを forward メソッドに渡す。最後は一番値が高い、すなわち一番可能性が高い番号を argmax を用いて出力させる。

1.3 実行結果

実行結果を以下の Listing1 に示す。

Listing 1: 実行結果

```
1    Please enter an integer between 0 and 9999
2    2
3    The most likely is 8
```

最初に入力を受け取るところでは 2 を入力した。

2 practice/practice2.py

2.1 内容

practice/practice1.py のコードをベースにミニバッチを入力可能とするように改良し、さらにクロスエントロピー誤差を計算するプログラムを作成する。

2.2 作成したプログラムの説明

作成したメソッドの意味を以下の表 3 に示す。

表 3: 作成したメソッドの説明

メソッド	説明
init_network()	初期化を行う。重みとバイアスは乱数で決定した。
predict(network, x)	順伝播で伝わっていく流れを行う。 引数の network には用いる重みやバイアスなどの情報が入っている。 x は画像データ。

common ディレクトリの function.py ファイルの中に新しくメソッドを作成した。それらのメソッドの意味を以下の表 4 に示す。なお、すでにその意味を示したものについては記載しない。

表 4: 用いたメソッドの説明

メソッド	説明
cross_entropy_error(y, t)	クロスエントロピー誤差を求める関数である。 引数の y は予想された結果を表す。 t は正解を表すデータを one-hot-vector 表現に直したものである。

各ノード数を定数として定義する。次に、用いる画像データを読み込む。次に、ランダムな値を生成し、その値を用いてバッチサイズ分のデータをランダムに取り出す。次に、重みやバイアスなどのデータとランダムに選ばれた 100 枚の画像データを predict メソッドに渡す。正解のラベルを記したデータを one-hot-vector 表現に直し、予想データと正解データを cross_entropy_error メソッドに渡してクロスエントロピー誤差を求める。

2.3 実行結果

実行結果を以下の 2 に示す。

Listing 2: 実行結果

```
1    cross_entropy_error : 2.4122362183864716
```

3 NeuralNetwork/NNSGD.py

3.1 内容

3層ニューラルネットワークのパラメータを学習するプログラムを作成する。

3.2 作成したプログラムの説明

使用するニューラルネットワークの各パラメータなどを設定したものを別ファイルに class として作り、各ノード数を引数として渡せばそのネットワークを使えるような設計にした。
また、順伝播と逆伝播のことを考慮する必要のある各関数や計算の処理を層として捉え、順伝播と逆伝播をメソッドとして持つようにし、それを common ディレクトリの layers.py にまとめた。
また、パラメータ更新の方法についても簡単に切り替えることができるようにパラメータ更新の手法を common ディレクトリの optimizer.py にまとめた。

LN.py に使用するニューラルネットワークを、common ディレクトリの layer.py に新たに必要となった層を、common ディレクトリの optimizer.py に新たに必要となった最適化手法を、NNSGD.py に全体の実装を施した。新しく作成したクラスのインスタンス変数の意味、メソッドの意味を以下の表 5 にしめす。

表 5: LayerNet クラスのインスタンス変数

インスタンス変数	説明
params	重みやバイアスなどの各パラメータを保持する。
layers	レイヤの情報を保持する。 順番付きにすることで順伝播、逆伝播を for 文一つでこなせるようにした。
lastLayer	最後のレイヤ。

表 6: LayerNet クラスのメソッド

メソッド	説明
__init__(self, INNODES, HNODES, ONODES)	初期化する。各パラメータを設定する。
predict(self, x)	推論を行う。 引数の x は画像データ。 追加したレイヤを for 文で回すだけで良い。
loss(self, x, t)	損失関数の値を求める。 引数の x は画像データ、t は正解ラベル。
accuracy(self, x, t)	認識精度を求める。
gradient(self, x, t)	重みパラメータに対する勾配を誤差逆伝播法で求める。 追加したレイヤを逆順にして for 文で回すだけで良い。

各関数の順伝播、逆伝播をまとめた層を表すクラスの説明を以下の表 7 に示す。

表 7: 関数別のクラスの説明

クラス名	説明
Sigmoid	シグモイド関数に関するクラスである。 初期化メソッド、順伝播メソッド、逆伝播メソッドを持つ。
Affine	行列計算に関するクラスである。 重みとバイアスを引数として与える。 初期化メソッド、順伝播メソッド、逆伝播メソッドを持つ。
SoftmaxWithLoss	softmax 関数とクロスエントロピー誤差の計算に関するクラスである。 初期化メソッド、順伝播メソッド、逆伝播メソッドを持つ。

パラメータ更新の手法をまとめたクラスの説明を以下の表 8 に示す。

表 8: 関数別のクラスの説明

クラス名	説明
SGD	確率的勾配降下法を実装したクラスである。 引数として学習率を渡す。

各ノード数を定数として定義する。各パラメータを定数として定義する。今回用いるニューラルネットワークの情報を記載したクラスを network 変数にいれる。パラメータ更新の手法として SGD(確率的勾配降下法) を設定する。次に、用いる画像データを読み込む。次に、ランダムな値を生成し、その値を用いてバッチサイズ分のデータをランダムに取り出す。次に、重みやバイアスなどのデータとランダムに選ばれた 100 枚の画像データを predict メソッドに渡す。正解のラベルを記したデータを one-hot-vector 表現に直し、予想データと正解データを cross_entropy_error メソッドに渡してクロスエントロピー誤差を求める。その際、同時に訓練データに対する正答率も表示させるようにした。

3.3 実行結果

実行結果を以下の Listing3 に示す。

Listing 3: 実行結果

```
1    1 : train_acc, cross_entropy_error | 10.716666666666667 %,
         2.3022416603586704
2    2 : train_acc, cross_entropy_error | 87.72 %, 0.5032862981802279
3    3 : train_acc, cross_entropy_error | 90.12833333333333 %,
         0.4823903638251616
4    4 : train_acc, cross_entropy_error | 90.81666666666666 %,
         0.27690497339207754
5    ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
6    27 : train_acc, cross_entropy_error | 95.81333333333333 %,
         0.1991737395218598
```

```
28 : train_acc, cross_entropy_error | 95.97333333333333 %,
    0.14366925184374962
29 : train_acc, cross_entropy_error | 96.06666666666666 %,
    0.13691907170279674
30 : train_acc, cross_entropy_error | 96.04166666666667 %,
    0.06836238884196702
```

勾配法による更新の回数は 18000 回とした。実行結果の最後の方は訓練データに対して大体 96 % の正答率であり、うまく動いていた。なお、結果が長かったため、途中は省略し最初と最後の 4 回を示した。これ以降も実行結果が長い場合はこのようにする。

4 NeuralNetwork/NNtestformnist.py

4.1 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する。

4.2 作成したプログラムの説明

重みのパラメータは NeuralNetwork/NNSGD.py で計算したパラメータを np.load を使って読み込んで用いた。それ以外は practice/practice1.py と同様である。

4.3 実行結果

以下の Listing4 に実行結果を示す。

Listing 4: 実行結果

```
Please enter an integer between 0 and 9999
3
result : 0
The answer is 0
test_acc : 95.36 %
```

0～9999 までの整数を入力させて結果の数字と正解の数字を出力させる。また、全体の正解・不正解から正答率も出るようにした。正答率は 95.36%となった。

5 NeuralNetwork/NNSGDforRelu.py

5.1 内容

Relu 関数を実装する。

5.2 作成したプログラムの説明

common ディレクトリの layers.py に Relu 関数の順伝播と逆伝播を表す層の実装を、LNforRelu.py に使用するネットワークの実装を、NNSGDforRelu.py に全体の実装を施した。以下の Listing5 にコードを示す。

Listing 5: Relu 関数の層の実装

```
class Relu:
def __init__(self):
    self.mask = None

def forward(self, x):

    self.x = x
    out = np.maximum(0, x)

    return out

def backward(self, dout):

    dx = dout * np.where(self.x > 0, 1, 0)

    return dx
```

作成したメソッドの意味を以下の表 9 に示す。

表 9: 作成したメソッドの説明

メソッド	説明
__init__(self)	初期化を行う。
forward(self, x)	順伝播を表す。 x は入ってくるデータを表す。
backward(self, dout)	逆伝播を表す。 引数の dout は上流から伝播してきたものを表す。

順伝播では 0 と x のうちの大きい方を出力する方法をとった。逆伝播では元の入力が 0 以上かどうかで 1 か 0 を np.where を用いて求め、それを dout にかけた。

5.3 実行結果

実行結果を以下の Listing6 に示す。

Listing 6: Relu 関数を用いた実行結果

```
1 : train_acc, cross_entropy_error | 15.228333333333332 %,
    13.18785770434647
2 : train_acc, cross_entropy_error | 34.403333333333336 %,
    1.7783033663347272
```

```
3     3 : train_acc, cross_entropy_error | 46.905 %, 1.502499277545619
4     4 : train_acc, cross_entropy_error | 47.46333333333334 %,
          1.2684076701907239
5     ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
6     27 : train_acc, cross_entropy_error | 55.81 %, 1.2290504640205264
7     28 : train_acc, cross_entropy_error | 49.35666666666666 %,
          1.1505136149778685
8     29 : train_acc, cross_entropy_error | 55.745 %, 1.1125242213640874
9     30 : train_acc, cross_entropy_error | 55.555 %, 1.012633186929583
```

実行結果が極めてよくなかった。テストデータに対する正答率は 10 %程度ですごく悪かった。これは初期値の問題かと推測した。調べたところによると sigmoid 関数は Xavier の初期値が適しているが、Relu 関数は He の初期値が適しているということであった。

そのため、次は初期値の標準偏差を変更して実行した。実行結果を以下の Listing7 に示す。

Listing 7: Relu 関数を用いた実行結果

```
1     1 : train_acc, cross_entropy_error | 10.643333333333334 %,
          14.198192978790187
2     2 : train_acc, cross_entropy_error | 48.31166666666667 %,
          1.4458970517433491
3     3 : train_acc, cross_entropy_error | 45.88833333333333 %,
          1.4138115421437671
4     4 : train_acc, cross_entropy_error | 60.11666666666666 %,
          1.0284964017676952
5     ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
6     27 : train_acc, cross_entropy_error | 66.715 %, 1.108267934551248
7     28 : train_acc, cross_entropy_error | 68.45166666666667 %,
          0.6973664590127113
8     29 : train_acc, cross_entropy_error | 69.32666666666667 %,
          0.6123580197540266
9     30 : train_acc, cross_entropy_error | 68.73333333333333 %,
          0.7354910420872506
```

訓練データに対する実行結果は先ほどよりは少しばかり良くなったように見える。しかし、テストデータに対する正答率は 10.1 %であまり変わらなかった。何が間違えているのかわからない。

6 NNSGDforDropout.py

6.1 内容

Dropout を実装する。

6.2 作成したプログラムの説明

common ディレクトリの layers に Dropout の関数を表すものを実装した。LNforDropout.py に使用したネットワークを実装した。NNSGDforDropout.py に全体を実装した。

以下の表 10 に Dropout クラスの説明を示す。

表 10: Dropout クラスのメソッド

メソッド	説明
__init__(self, dropout_ratio=0.5, train_flag=True)	初期化する。 引数の dropout_ratio はどの程度消去するかを決めるための基準。 train_flag は対象となるデータが訓練データかテストデータかを判定するフラグ。True で訓練データであることを表す。 呼び出し元で引数に加えて呼び出す。 引数に何も描かない場合は自動的に訓練データに対するものだと判定するようにした。
forward(self, x)	順伝播を表す。 引数の x は入ってくるデータ。
backward(self, dout)	逆伝播を表す。 引数の dout は上流から伝播してきたものを表す。

LNforDropout.py に Sigmoid 層の後に Dropout 層を挟んだものを用意した。Dropout 層を追加したこと以外は NNSGD.py 動かしたものと同じ構成である。NNSGDforDropout.py に作った構成を利用するファイルを用意した。

6.3 実行結果

実行結果を以下の Listing8 に示す。

Listing 8: Dropout を用いた実行結果

```
1    1 : train_acc, cross_entropy_error | 10.333333333333334 %,
         2.3342607288377706
2    2 : train_acc, cross_entropy_error | 73.09166666666667 %,
         1.1608827821470198
3    3 : train_acc, cross_entropy_error | 80.76 %, 0.8886753835719177
4    4 : train_acc, cross_entropy_error | 83.90166666666666 %,
         0.6368787606997498
5    ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
6    27 : train_acc, cross_entropy_error | 90.09666666666666 %,
         0.18761408490134263
7    28 : train_acc, cross_entropy_error | 90.17500000000001 %,
         0.32923917588718254
8    29 : train_acc, cross_entropy_error | 90.46666666666667 %,
         0.31215953769692856
9    30 : train_acc, cross_entropy_error | 90.815 %, 0.32118817589616705
```

テストデータに対しての正答率を以下の Listing9 にコードを示す。

Listing 9: Dropout を用いた実行結果

```
1    test_acc : 93.34 %
```

実行結果は 93.34%となった。

7 NNSGDforBN.py

7.1 内容

Batch Normalization を実装する。

7.2 作成したプログラムの説明

common ディレクトリの layers に BatchNormalization の関数を表すものを設計した。LNforBN.py に使用したネットワークを実装した。NNSGDforBN.py に全体を実装した。以下の表 11 に Batch-Normalization クラスの説明を示す。

表 11: BatchNormalization クラスのメソッド

メソッド	説明
__init__(self, gamma=1, beta=0, momentum=0.9, train_flag=True)	初期化する。 引数の gamma,beta は用いるパラメータ。 momenrum は移動平均・移動分散を計算する際に使用する。 詳しくはこの節の工夫点・問題点にて述べた。 train_flag は対象となるデータが訓練データかテストデータかを判定するフラグ。 True で訓練データであることを表す。 呼び出し元で引数に加えて呼び出す。 引数に何も描かない場合は自動的に訓練データに対する ものだと判定するようにした。
forward(self, x)	順伝播を表す。 引数の x は入ってくるデータ。
backward(self, dout)	逆伝播を表す。 引数の dout は上流から伝播してきたものを表す。

LNforBN.py に一つの Affine 層と Sigmoid 層の間に Batch Normalization 層を挟んだものを用意した。Batch Normalization 層を追加したこと以外は NNSGD.py で動かしたものと同じ構成である。NNSGDforBN.py に作った構成を利用するファイルを用意した。

7.3 実行結果

実行結果を以下の Listing10 に示す。

Listing 10: Batch Normalization を用いた実行結果

```
1     1 : train_acc, cross_entropy_error | 12.023333333333333 %,
          2.339778665440642
2     2 : train_acc, cross_entropy_error | 83.76666666666667 %,
          1.1051816272869288
3     3 : train_acc, cross_entropy_error | 86.81333333333333 %,
          0.8832146872495862
4     4 : train_acc, cross_entropy_error | 87.94999999999999 %,
          0.6474488563461622
5     ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
6     27 : train_acc, cross_entropy_error | 91.95 %, 0.4199221954333971
7     28 : train_acc, cross_entropy_error | 91.99166666666667 %,
          0.3372609974003518
8     29 : train_acc, cross_entropy_error | 92.05333333333333 %,
          0.28719901457548547
9     30 : train_acc, cross_entropy_error | 92.035 %, 0.2074839408240136
```

テストデータに対しての正答率を以下の Listing11 にコードを示す。

Listing 11: Batch Normalization を用いた実行結果

```
1     test_acc : 82.19 %
```

実行結果は 82.19%となった。

8 種々の最適化手法の実装

8.1 内容

種々の最適化手法を実装する。

8.2 NNMomentum.py

8.2.1 作成したプログラムの説明

common ディレクトリの optimizer.py に慣性項付き SGD を表すクラスを設計した。

NNMomentum.py に最適化手法として慣性項付き SGD を用いたものを用意した。なお、ここで用いたニューラルネットワークの構成は LayerNet/LN.py と同じ構成である。

8.2.2 実行結果

実行結果を以下の Listing12 に示す。

Listing 12: 慣性項付き SGD を用いた実行結果

```
1     1 : train_acc, cross_entropy_error | 10.716666666666667 %,
          2.3022416603586704
2     2 : train_acc, cross_entropy_error | 86.985 %, 0.4786249416406769
```

```
3     3 : train_acc, cross_entropy_error | 88.72833333333332 %,
          0.48682828311441356
4     4 : train_acc, cross_entropy_error | 87.86166666666666 %,
          0.3391920561953264
5     ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
6     27 : train_acc, cross_entropy_error | 90.36333333333333 %,
          0.381868619583981
7     28 : train_acc, cross_entropy_error | 90.44 %, 0.27865695247252015
8     29 : train_acc, cross_entropy_error | 89.97166666666668 %,
          0.3117181126573155
9     30 : train_acc, cross_entropy_error | 90.875 %, 0.224095889833865
```

テストデータに対しての正答率を以下の Listing13 にコードを示す。

Listing 13: Dropout を用いた実行結果

```
1     test_acc : 91.21000000000001 %
```

実行結果は 91.21%となった。

8.3 NNAdaGrad.py

8.3.1 作成したプログラムの説明

common ディレクトリの optimizer.py に AdaGrad を表すクラスを設計した。

NNAdaGrad.py に最適化手法として AdaGrad を用いたものを用意した。なお、ここで用いたニューラルネットワークの構成は LayerNet/LN.py と同じ構成である。

8.3.2 実行結果

実行結果を以下の Listing14 に示す。

Listing 14: AdaGrad を用いた実行結果

```
1     1 : train_acc, cross_entropy_error | 11.698333333333332 %,
          2.2041422671158277
2     2 : train_acc, cross_entropy_error | 81.39999999999999 %,
          0.9453251680010945
3     3 : train_acc, cross_entropy_error | 84.62833333333334 %,
          0.7806701852688275
4     4 : train_acc, cross_entropy_error | 86.265 %, 0.6163468760856798
5     ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
6     27 : train_acc, cross_entropy_error | 91.53333333333333 %,
          0.3951559557588784
7     28 : train_acc, cross_entropy_error | 91.57666666666667 %,
          0.30855008192601885
8     29 : train_acc, cross_entropy_error | 91.64 %, 0.2750605228279207
9     30 : train_acc, cross_entropy_error | 91.69166666666668 %,
          0.26015691731122037
```

テストデータに対しての正答率を以下の Listing15 にコードを示す。

Listing 15: AdaGrad を用いた実行結果

```
1    test_acc : 91.07 %
```

実行結果は 91.07%となった。

8.4 NNRMSProp.py

8.4.1 作成したプログラムの説明

common ディレクトリの optimizer.py に RMSProp を表すクラスを設計した。

NNRMSProp.py に最適化手法として RMSProp を用いたものを用意した。なお、ここで用いたニューラルネットワークの構成は LayerNet/LN.py と同じ構成である。

8.4.2 実行結果

実行結果を以下の Listing16 に示す。

Listing 16: RMSProp を用いた実行結果

```
1    1 : train_acc, cross_entropy_error | 20.11 %, 2.209134431990888
2    2 : train_acc, cross_entropy_error | 88.12166666666667 %,
         0.2672466311189515
3    3 : train_acc, cross_entropy_error | 88.95666666666666 %,
         0.33995589616260896
4    4 : train_acc, cross_entropy_error | 90.08500000000001 %,
         0.14500362981917128
5    ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
6    27 : train_acc, cross_entropy_error | 92.365 %, 0.2697922233490362
7    28 : train_acc, cross_entropy_error | 92.12 %, 0.18470638696130517
8    29 : train_acc, cross_entropy_error | 91.94666666666666 %,
         0.15328161459379663
9    30 : train_acc, cross_entropy_error | 92.08666666666666 %,
         0.10620706354015187
```

テストデータに対しての正答率を以下の Listing17 にコードを示す。

Listing 17: RMSProp を用いた実行結果

```
1    test_acc : 92.27 %
```

実行結果は 92.27%となった。

8.5 NNAdaDelta.py

8.5.1 作成したプログラムの説明

common ディレクトリの optimizer.py に AdaDelta を表すクラスを設計した。

NNAdaDelta.py に最適化手法として AdaDelta を用いたものを用意した。なお、ここで用いたニューラルネットワークの構成は LayerNet/LN.py と同じ構成である。

8.5.2 実行結果

実行結果を以下の Listing18 に示す。

Listing 18: AdaDelta を用いた実行結果

```
1     1 : train_acc, cross_entropy_error | 17.908333333333335 %,
          2.1088324976411212
2     2 : train_acc, cross_entropy_error | 90.43666666666667 %,
          0.2868945146764703
3     3 : train_acc, cross_entropy_error | 91.79 %, 0.27903486914617814
4     4 : train_acc, cross_entropy_error | 92.87833333333333 %,
          0.10589671491300251
5     ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
6     27 : train_acc, cross_entropy_error | 96.14833333333334 %,
          0.13425528028955724
7     28 : train_acc, cross_entropy_error | 96.35666666666667 %,
          0.10975251749698653
8     29 : train_acc, cross_entropy_error | 96.19833333333332 %,
          0.05152950401998373
9     30 : train_acc, cross_entropy_error | 96.20833333333333 %,
          0.037644912887046154
```

テストデータに対しての正答率を以下の Listing19 にコードを示す。

Listing 19: AdaDelta を用いた実行結果

```
1     test_acc : 95.82000000000001 %
```

実行結果は 95.82%となった。

8.6 NNAdam.py

8.6.1 作成したプログラムの説明

common ディレクトリの optimizer.py に Adam を表すクラスを設計した。

NNAdam.py に最適化手法として Adam を用いたものを用意した。なお、ここで用いたニューラルネットワークの構成は LayerNet/LN.py と同じ構成である。

8.6.2 実行結果

実行結果を以下の Listing20 に示す。

Listing 20: Adam を用いた実行結果

```
1     1 : train_acc, cross_entropy_error | 11.698333333333332 %,
          2.2041422671158277
2     2 : train_acc, cross_entropy_error | 90.425 %, 0.3228823942374737
3     3 : train_acc, cross_entropy_error | 91.545 %, 0.3565881938163395
4     4 : train_acc, cross_entropy_error | 92.01 %, 0.21899410182085335
5     ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

```
    27 : train_acc, cross_entropy_error | 94.51166666666667 %,
        0.307590600459141
    28 : train_acc, cross_entropy_error | 94.48333333333333 %,
        0.12913506303971486
    29 : train_acc, cross_entropy_error | 93.89666666666666 %,
        0.13786868914558398
    30 : train_acc, cross_entropy_error | 94.77499999999999 %,
        0.08540847200464921
```

テストデータに対しての正答率を以下の Listing21 にコードを示す。

Listing 21: Adam を用いた実行結果

```
    test_acc : 94.52000000000001 %
```

実行結果は 94.52%となった。

9 カラー画像

9.1 内容

カラー画像に拡張する。

9.2 作成したプログラムの説明

NNcolor.py に訓練データを読み込んで学習を行う設計をした。NNcolortest.py にテストデータに対してテストを行う設計をした。

ファイルを読み込む際には基本的にはサンプルで用意されていたコードを参考にしたが、訓練データに関しては 5 つのファイルを読み込む必要があったため、for 文でうまく読み込めるように調整した。

なお、ここでの最適化手法は SGD を用い、ニューラルネットワークの構成は LayerNet/LN.py を用いた。

9.3 実行結果

以下の Listing22 に実行結果を示す。

Listing 22: 訓練データでの学習

```
    1 : train_acc, cross_entropy_error | 10.038 %, 2.4053279452148155
    2 : train_acc, cross_entropy_error | 27.616000000000003 %,
        2.161321993071682
    3 : train_acc, cross_entropy_error | 30.753999999999998 %,
        2.0082028879208593
    4 : train_acc, cross_entropy_error | 32.196000000000005 %,
        1.9898638225024956
    ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    27 : train_acc, cross_entropy_error | 41.19 %, 1.743870407093964
```

```
    28 : train_acc, cross_entropy_error | 41.812 %, 1.7301771329293416
    29 : train_acc, cross_entropy_error | 41.842 %, 1.7178572053329282
    30 : train_acc, cross_entropy_error | 42.054 %, 1.6368166246949625
```

テストデータに対しての正答率を以下の Listing23 にコードを示す。

Listing 23: Adam を用いた実行結果

```
    test_acc : 37.74 %
```

実行結果は 37.74%となった。すごく低かった。

10 畳み込み層

10.1 内容

畳み込み層を実装する。

10.2 作成したプログラムの説明

common ディレクトリの layers.py に畳み込み層を表すものを設計した。以下の表 12 に Convolution クラスの説明を示す。

表 12: Convolution クラスのメソッド

メソッド	説明
__init__(self, W, b, stride, pad)	初期化する。 引数の W, b はフィルターのデータ、バイアスを表す。 stride はストライドを表す。 pad はパディングを表す。
forward(self, x)	順伝播を表す。 引数の x は入ってくるデータ。
backward(self, dout)	逆伝播を表す。 引数の dout は上流から伝播してきたものを表す。

forward メソッドでは x の形から入力データのバッチ数、チャンネル、高さ、幅を読み取る。次に W の形からフィルターの個数、チャンネル、高さ、幅を読み取る。それらのパラメータを用いて出力するデータの高さと幅を求める。

次に畳み込み層の演算を行うのですが、ここがはじめに自分が実装したものではどうにもうまくいかなかったので、参考書 [1] を大きく参考にした場所です。ファイルターにとって都合の良いように入力データを展開する関数 im2col を用いる。関数 im2col を common ディレクトリ以下の util.py に実装する。im2col は入力データに対してファイルターを適用する場所の領域を横方向に一列に展開するのをフィルターを適用する全ての場所で行うものである。これでデータを展開しそれをフィルターを展開したものと掛け合わせる。

形を直し、それを出力する。

Affine 層の順伝播と逆伝播にテンソルの形が流れてきても対応できるように Afine 層の順伝播・逆伝播をテンソルに対応させた。方法としてはただ元の x の形を別の変数の中に入れておいて x 自体の形を変えても必要な時に reshape で元の形になおしたりできるようにするものである。

backward メソッドは Affine 層と同じようにした。ただこちらでは先ほどの im2cal 関数の逆を行う col2im 関数を実装する必要があったのでそれを common ディレクトリ以下の util.py に実装した。

10.3 実行結果

次の Pooling 層の実装と合わせて用いるものだと思っていたのでそのようにした。なので、ここでの実行結果は次のプーリング層の実行結果とまとめて見る。

11 プーリング層

11.1 内容

プーリング層を実装する。

11.2 作成したプログラムの説明

参考書 [1] の畳み込みニューラルネットワークの章を参考にして実装をすることにした。

common ディレクトリの layers にプーリング層を表すものを設計した。以下の表 13 に Pooling クラスの説明を示す。

表 13: Pooling クラスのメソッド

メソッド	説明
__init__(self, pool_h, pool_w, stride, pad)	初期化する。 引数の pool_h はプーリングのウィンドウサイズの高さを表す。 引数の pool_w はプーリングのウィンドウサイズの幅を表す。 stride はストライドを表す。 pad はパディングを表す。
forward(self, x)	順伝播を表す。 引数の x は入ってくるデータ。
backward(self, dout)	逆伝播を表す。 引数の dout は上流から伝播してきたものを表す。

forward メソッドは入力データを展開、行ごとに最大値を求める、適切なサイズに整形するという流れで行った。展開の際には先ほどの畳み込み層の実装の際に作った im2col 関数を使った。

backward メソッドでは Relu レイヤでの逆伝播を参考にした。最後の出力を求める際には先ほどの畳み込み層の実装の際に作った col2im 関数を使った。

実行するニューラルネットワークを LNforCP.py に設計した。層としては Convolution 層→ Sigmoid 層→ Pooling 層→ Affine 層→ Softmax 層という形をとった。

実行するにあたり NNSGDforCP.py を用意した。このファイルでは畳み込み層ニューラルネットワークを用いるために mnist の画像を 1×28×28 の形で読み込むようにした。

11.3 実行結果

実装が終わりエラーを全て解決してようやく動かしたのだが、どうにも数値が現れなかった。そこで、エポックが終了するごとに今が何回目のエポックかを表すようにすると、1 回目のエポックの最終回にきているところまでは確認できた。しかし、なかなか待ってもその後訓練データに対する正答率とクロスエントロピー誤差を示すことはなかった。

参考文献

[1] 斎藤康毅『ゼロから作る Deep Learning: Python で学ぶディープラーニングの理論と実装』初版（株式会社オライリー・ジャパン，2017）

[2] Deep Learning における Batch Normalization の理解メモと、実際にその効果を見てみる - Qiita https://qiita.com/cfiken/items/b477c7878828ebdb0387

[3] BatchNormalization の初出論文メモ - 緑茶思考ブログ http://yusuke-ujitoko.hatenablog.com/entry/2017/06/17/164545

[4] TensorFlow の高レベル API を使った Batch Normalization の実装 - Qiita https://qiita.com/cometscome_phys/items/6d5d3c74d7000382efef