numpy による CNN の実装

2020 年 11 月 26 日

目次

1　使用ファイル一覧

neural_network.py

　　ニューラルネットワークのクラスと学習に使う関数をまとめたファイル。

mnist_tool.py

　　mnist の学習で，共通する変数や関数を切り出したファイル．

mnist_accuracy.py

　　mnist の訓練データとテストデータに対して，それぞれ正答率を求める．

kadai3.py

　　課題 3 を実行するための main ファイル．

kadai4.py

　　課題 4 を実行するための main ファイル．

mnist_para.npz

　　mnist の学習で得られたパラメータを保存する．

cifar_tool.py

　　CIFAR-10 の学習で，共通する変数や関数を切り出したファイル．

cifar_accuracy.py

　　CIFAR-10 の訓練データとテストデータに対して，それぞれ正答率を求める．

cifar3.py

　　CIFAR-10 の学習の main ファイル．(kadai3.py に相当)

cifar4.py

　　CIFAR-10 のテストデータ 1 枚に対して推論を行うための main ファイル．(kadai4.py に相当)

cifar_para.npz

　　CIFAR-10 の学習で得られたパラメータを保存する．

2　実装内容

　課題 4 と発展課題 A1〜A7 に取り組んだ．ただし，A4(最適化) では Adam のみ実装した．最終的に実装したニューラルネットワークの構造を図 1 に示す．

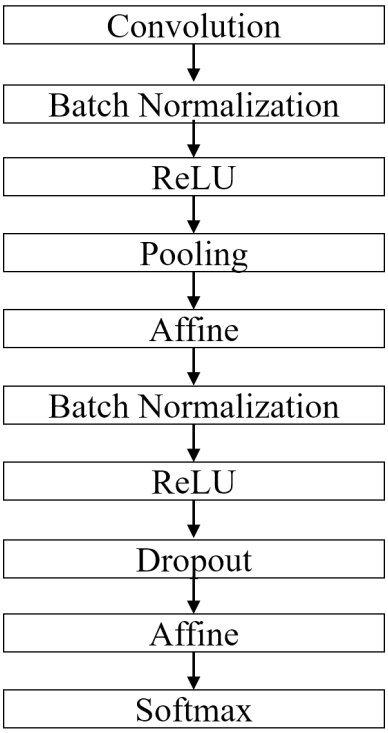


図 1　ニューラルネットワークの構造

まず，neural_network.py における各層の実装を発展課題から説明する．

3　発展課題 A1 ReLU

順伝播 (forward)

引数 t に対して，

$$a(t) = \max(0, t)$$

を返す．

```
class ReLU:
    def forward(self, t):
        self.x = t
        return np.maximum(0, t)
```

ソースコード 1　ReLU の順伝播

逆伝播

講義資料より，

$$\frac{\partial E_n}{dx} = \begin{cases} \frac{\partial E_n}{dy} & (x > 0) \\ 0 & (x \leq 0) \end{cases}$$

と表せるから，numpy.where で以下のように表現した．

```
class ReLU:
...
    def backward(self, dEn_dy):
        return np.where(self.x > 0, dEn_dy, 0)
```

ソースコード 2　ReLU の逆伝播

工夫点

多次元配列に対応するようにした.

4　発展課題 A2 Dropout

順伝播 (forward)

引数は，入力 x，学習時に True となるフラグ is_train，無視するノードの割合 rate(デフォルト値 0.5) である.

簡単のため，x と同次元の $[0, 1)$ の乱数配列を生成し，乱数が rate より大きいかどうかで学習時の無視しないノードを決める．self.is_valid は各ノードが無視されない場合に True となる.

テスト時は，x を (1-rate) 倍する.

```
class Dropout:
    def forward(self, x, is_train, rate = 0.5):
        if is_train:
            self.is_valid = np.random.rand(*x.shape) > rate
            y = np.where(self.is_valid, x, 0)
        else:
            y = x * (1 - rate)
        return y
```

ソースコード 3　Dropout の順伝播

逆伝播 (backward)

講義資料より,

$$\frac{\partial E_n}{dx} = \begin{cases} \frac{\partial E_n}{dy} & \text{(ノードが無視されない場合)} \\ 0 & \text{(ノードが無視された場合)} \end{cases}$$

と書けるから，順伝播で保存した self.is_valid を用いて以下のように計算する.

```
class Dropout:
...
    def backward(self, dEn_dy):
        return np.where(self.is_valid, dEn_dy, 0)
```

ソースコード 4　Dropout の逆伝播

工夫点

- 多次元配列に対応するようにした.
- 簡単のため，無視するノードを乱数の大小で決定するようにした.

5　発展課題 A4 Adam

コンストラクタ

学習するパラメータである t, m, v を 0 で初期化してインスタンス変数として持たせる.

```
class Adam:
    def __init__(self):
        self.t = 0
        self.m = 0
        self.v = 0
```

ソースコード 5　Adam のコンストラクタ

save/load

save はパラメータを保存するメソッド，load は読み込むメソッドである．ともに引数は para,name である．para は，key を str 型，value を numpy.ndarray とする dict 型である．name は自身に割り当てられた変数名の str 型が渡される．例えば，外部で「adam」という変数名で使われている場合，adam.save(para, 'adam') といった呼び出し方をする．このとき，「name + '_t'」という文字列はプログラム全体でたとえ，Adam が複数生成されていても，変数名は重複しなければ，一意に定まる．Adam だけでなく，全てのクラスでこのように save/load を定義することで，1 つの dict 型 para に学習に用いるパラメータを全て格納することができて，パラメータの管理が簡単になる．また，外部からは保存用の dict と自身の変数名を渡すだけで保存すべきパラメータが para に格納される.

save/load するのは，t, m, v であり，以下のように書く.

```
class Adam:
...
    def save(self, para, name):
        para[name + '_t'] = self.t
        para[name + '_m'] = self.m
        para[name + '_v'] = self.v

    def load(self, para, name):
        self.t = para[name + '_t']
        self.m = para[name + '_m']
        self.v = para[name + '_v']
```

ソースコード 6　Adam の save/load

重みの修正 (grad)

講義資料記載の以下の式をそのまま実装している．

$$
\begin{aligned}
t &\leftarrow t+1 \\
m &\leftarrow \beta_1 m + (1-\beta_1)\frac{\partial E_n}{\partial W} \\
v &\leftarrow \beta_2 v + (1-\beta_2)\frac{\partial E_n}{\partial W} \circ \frac{\partial E_n}{\partial W} \\
\hat{m} &\leftarrow m/(1-\beta_1^t) \\
\hat{v} &\leftarrow v/(1-\beta_2^t) \\
&\alpha\hat{m}/(\sqrt{\hat{v}}+\epsilon) \quad (\text{を返す})
\end{aligned}
$$

$\alpha, \beta_1, \beta_2, \epsilon$ は Adam の論文中での推奨値をデフォルト値とするデフォルト引数で設定した．

```
def grad(self, dEn_dW, alpha = 0.001, beta1 = 0.9, beta2 = 0.999, eps = 1e-8):
    self.t += 1
    self.m *= beta1
    self.m += (1 - beta1) * dEn_dW
    self.v *= beta2
    self.v += (1 - beta2) * dEn_dW * dEn_dW
    m_hat = self.m / (1 - np.power(beta1, self.t))
    v_hat = self.v / (1 - np.power(beta2, self.t))
    return alpha * m_hat / (np.sqrt(v_hat) + eps)
```

ソースコード 7　Adam の grad

工夫点

- Adam をいろいろなところに追加することを考え，簡単に使えるようクラスで実装した．
- 多次元配列に対応するようにした．
- 上記のようにパラメータの save/load を定義することで，パラメータの管理を容易にした．(ニューラルネットワーク全体での工夫点)

6　発展課題 A3 Batch Normalization

コンストラクタ

学習するパラメータは，gamma，beta，およびそれらの更新に使う Adam の adam_gamma,adam_beta である．また，ミニバッチの各ノードの出力の分散を学習のたびに足し合わせた var_sum，同様に平均を足し合わせた mean_sum，学習回数 count を保持する．ノード数を d とすると，var_sum と mean_sum は d 次元ベクトルとなる．これらをインスタンス変数に持たせる．また，小さな値として，eps = 1e-10 をクラス変数として定義している．gamma は 1，それ以外は 0 で初期化する．

```
class Batch_normalization:
    eps = 1e-10
    def __init__(self):
```

```
        self.gamma = 1
        self.beta = 0
        self.adam_gamma = Adam()
        self.adam_beta = Adam()
        self.var_sum = 0.
        self.mean_sum = 0.
        self.count = 0
```

ソースコード 8　Batch_normalization のコンストラクタ

save/load

save/load するパラメータは，上記のインスタンス変数である．また，adam_gamma と adam_beta に関しては，以下のように変数名の文字列を付け足して save や load を呼び出すことで簡単に書ける．

```
class Batch_normalization:
...
    def save(self, para, name):
        para[name + '_gamma'] = self.gamma
        para[name + '_beta'] = self.beta
        para[name + '_var_sum'] = self.var_sum
        para[name + '_mean_sum'] = self.mean_sum
        para[name + '_count'] = self.count
        self.adam_gamma.save(para, name + '_adam_gamma')
        self.adam_beta.save(para, name + '_adam_beta')

    def load(self, para, name):
        self.gamma  = para[name + '_gamma']
        self.beta = para[name + '_beta']
        self.var_sum = para[name + '_var_sum']
        self.mean_sum = para[name + '_mean_sum']
        self.count = para[name + '_count']
        self.adam_gamma.load(para, name + '_adam_gamma')
        self.adam_beta.load(para, name + '_adam_beta')
```

ソースコード 9　Batch_normalization の save/load

順伝播 (forward)

引数の入力 x は，ノード数を d，バッチサイズを B とすると，d 行 B 列の 2 次元配列で与えられる．is_train は学習時に True となるフラグである．

まず，学習時について述べる．ミニバッチの平均と分散は，ノードごと，つまり x に対して行ごとにとるので，np.mean,np.var の axis は 1 を設定している．x_hat や y は，numpy の計算の都合上，転置 T を多用している．(2 次元配列と 1 次元配列の演算を numpy でうまく行うため.) μ_B と σ_B の期待値を求めるため，var_sum と mean_sum にミニバッチの分散と平均を足す．ここで，不偏分散の期待値が母分散に一致するので，不偏分散に変換して足している．

```
class Batch_normalization:
...
    def forward(self, x, is_train):
        self.x = x
        if is_train:
```

```
            self.mean = np.mean(x, axis = 1)
            self.var = np.var(x, axis = 1)
            self.x_hat = ((x.T - self.mean) /
                          np.sqrt(self.var + Batch_normalization.eps)).T
            self.y = ((self.gamma * self.x_hat.T) + self.beta).T

            B = x.shape[1]
            self.var_sum += self.var * B / (B - 1)
            self.mean_sum += self.mean
```

ソースコード 10　Batch_normalization の順伝播 (訓練時)

次に，テスト時について述べる．学習時に足し合わせた分散と平均の和を学習回数で割ることでそれぞれの期待値を求める．ここで，学習回数 0 のときにテストを行うと 0 で割ることになるので，念のため場合分けを入れている．

```
class Batch_normalization:
...
    def forward(self, x, is_train):
        if if_train:
        ...
        else:
            var = self.var_sum / self.count if self.count > 0 else 0
            mean = self.mean_sum / self.count if self.count > 0 else 0

            c = self.gamma / np.sqrt(var + Batch_normalization.eps)
            self.y = (c * x.T + (self.beta - c * mean)).T
        return self.y
```

ソースコード 11　Batch_normalization の順伝播 (テスト時)

逆伝播 (backward)

学習回数を表す count を +1 する．講義資料記載の式に従って順に勾配を求め，Adam を通して gamma,beta を更新する．逆伝播もまた，numpy の計算の都合上，転置 T を多用している．

```
class Batch_normalization:
...
    def backward(self, dEn_dy):
        self.count += 1

        B = dEn_dy.shape[1]
        dEn_dx_hat = (dEn_dy.T * self.gamma).T
        dEn_dvar = np.sum(dEn_dx_hat * (self.x.T - self.mean).T, axis = 1)
                       * (-1 / 2) * (self.var + Batch_normalization.eps) ** (-3 / 2)
        c = 1 / np.sqrt(self.var + Batch_normalization.eps)
        dEn_dmean = -c * np.sum(dEn_dx_hat, axis = 1) +
                    dEn_dvar * (-2) * (np.mean(self.x, axis = 1) - self.mean)
        dEn_dx = (dEn_dx_hat.T * c +
                  dEn_dvar * 2 * (self.x.T - self.mean) / B + dEn_dmean / B).T
        dEn_dgamma = np.sum(dEn_dy * self.x_hat, axis = 1)
        dEn_dbeta = np.sum(dEn_dy, axis = 1)
        self.beta -= self.adam_beta.grad(dEn_dbeta)

        # 更新
```

```
        self.gamma -= self.adam_gamma.grad(dEn_dgamma)
        self.beta -= self.adam_beta.grad(dEn_dbeta)

        return dEn_dx
```

ソースコード 12　Batch_normalization の逆伝播

CNN での順伝播について

畳み込み層を入れる前までは，この実装で正しく動き，学習するにつれて精度も安定して上がっていったが，畳み込み層を入れると，途中から学習が不安定になり，一気に精度が落ちるという現象に直面した．クロスエントロピー誤差は順調に落ちる一方，訓練データに対する正答率が急激に低下した．そのため，テスト時に使用するミニバッチの平均と分散の期待値が畳み込み層を入れたことで適切な値に安定しなくなったのではないかと考えた．

そこで，以下のように期待値の求め方を算術平均から移動平均を用いたところ，精度が安定して上昇するように改善された．今得られた分散と平均を 0.01 倍し，過去のデータを 0.99 倍して足し合わせている．

```
def forward(self, x, is_train):
    self.x = x
    if is_train:
        ...
        # self.var_sum += self.var * B / (B - 1)
        # self.mean_sum += self.mean

        momentum = 0.01
        self.mean_sum *= (1 - momentum)
        self.mean_sum += momentum * self.mean
        self.var_sum *= (1 - momentum)
        self.var_sum += momentum * self.var
    else:
        # var = self.var_sum / self.count if self.count > 0 else 0
        # mean = self.mean_sum / self.count if self.count > 0 else 0

        var = self.var_sum
        mean = self.mean_sum

        c = self.gamma / np.sqrt(var + Batch_normalization.eps)
        self.y = (c * x.T + (self.beta - c * mean)).T
    return self.y
```

ソースコード 13　移動平均を使った順伝播

しかし，何故畳み込み層を入れたときに算術平均ではうまくいかなくなるのか，何故移動平均にするとうまくいくのか，に関する根本的な原因は見つけることができなかった．他のクラスが影響を及ぼしている可能性も考え，バグがないかや無限大や無限小になっていないかなどを確認したが，はっきりとしたものは見つからなかった．今後の課題にしたい．なお，これらのことは後の章で mnist の画像データの学習結果を報告する際にもう一度述べる．

以下，移動平均を選択するのに参照したものをあげる．

[1] Sergey Ioffe, Christian Szegedy(2015)「Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift」https://arxiv.org/abs/1502.03167

[2] Lei Mao 「Batch Normalization Explained」 https://leimao.github.io/blog/Batch-Normalization/ 閲覧日 2020 年 11 月 25 日

[1] の p.4, 右段,12〜13 行目の「Using moving averages instead, we can track the accuracy of a model as it trains.」と [2] の「Mathematical Definition → Training Phase」を参考にした.

工夫点

発展課題 A4 で実装した Adam を用いて β, γ を更新するようにした.

問題点

- x を d 行 B 列の行列としたため，ある d 次元ベクトルとバッチごと (列ごと) に処理する計算は，一度転置したあと再度転置をするというコードになり，非常に見にくく不自然になってしまった．このことから，画像バッチ x は B 行 d 列という次元のほうが numpy で自然に書けると思われる.
- 畳み込み層を実装した後，平均と分散の期待値を算術平均で求めると，精度が一気に落ちるという現象が起きた．算術平均ではなく，移動平均に変更した結果，この問題は解決できたが，詳しい原因は解明できなかった．今後の課題としたい.

7 発展課題 A6 畳み込み

Convolution クラスとして実装している.

コンストラクタ

行列 W とバイアス b の更新に用いる Adam を生成する.

```
class Convolution:
    def __init__(self):
        self.adam_W = Adam()
        self.adam_b = Adam()
```

ソースコード 14　Convolution のコンストラクタ

初期化 (init)

乱数で初期化する際に呼び出す．引数の ch,K,R,p,s は順に入力画像のチャンネル数，フィルタ枚数，フィルタサイズ，パディング幅，ストライドである．R,s は指定しなければ出力画像と元の画像の大きさが同じになるようにしている．self.W は講義資料の通りにフィルタから変換した K 行 $R \times R \times ch$ 列の行列である．b はバイアスベクトルである．W と b の生成方法は，全結合層と同様である.

```
class Convolution:
...
    def init(self, ch, K, R, p = -1, s = 1):
        self.R = R
        self.p = R // 2 if p < 0 else p
        self.s = s
```

```
        row = K
        col = R * R * ch
        self.W = random_ndarray(col, (row, col))
        self.b = random_ndarray(col, row)
...
# 乱数配列
def random_ndarray(N, shape):
    return np.random.normal(0, 1 / np.sqrt(N), shape)
```

ソースコード 15　Convolution の初期化

save/load

これまでと同様のやり方で，以下の変数を save/load する．

```
class Convolution:
...
    def save(self, para, name):
        para[name + '_R'] = self.R
        para[name + '_p'] = self.p
        para[name + '_s'] = self.s

        para[name + '_W'] = self.W
        para[name + '_b'] = self.b
        self.adam_W.save(para, name + '_adam_W')
        self.adam_b.save(para, name + '_adam_b')

    def load(self, para, name):
        self.R = para[name + '_R']
        self.p = para[name + '_p']
        self.s = para[name + '_s']

        self.W = para[name + '_W']
        self.b = para[name + '_b']
        self.adam_W.load(para, name + '_adam_W')
        self.adam_b.load(para, name + '_adam_b')
```

ソースコード 16　save/load

画像バッチから行列 X への変換 (convert_batch_into_X)

(B, ch, dy, dx) の画像バッチを講義資料記載の行列 X に変換するための関数である．まず，numpy.pad を用いてパディングを行う．次に，フィルタがずれる距離と出力画像のサイズを求める．そして，for 文で変換を行ったあと，$R \times R \times ch$ 行 $dw \times dh \times B$ 列の行列に整形する．

```
# 画像バッチから行列Xへの変換
def convert_batch_into_X(batch, R, p, s):
    # バッチサイズ，チャンネル，縦，横
    B, ch, dy, dx = batch.shape

    # パディング
    batch = np.pad(batch, ((0, 0), (0, 0), (p, p), (p, p)))
```

```
    # フィルタがずれる距離
    di = 2 * p + dy - R
    dj = 2 * p + dx - R

    # 出力画像のサイズ dw × dh
    dh = di // s + 1
    dw = dj // s + 1

    X = np.empty((B, ch, R, R, dh, dw))
    for i in range(R):
        for j in range(R):
            X[:, :, i, j, :, :] = batch[:, :, i:i+di+1:s, j:j+dj+1:s]

    # (R * R * ch, dw * dh * B) の X と次元数 (B, dh, dw) を返す
    X = X.transpose(1, 2, 3, 0, 4, 5).reshape(R * R * ch, -1)

    return X, (B, dh, dw)
```

ソースコード 17　画像バッチから行列 X への変換

ここで for 文で行っていることを説明する．例えば，パディング済みの 7×7 の 1 枚の画像に対し，フィルタサイズ 3，ストライド 1 で変換を行うことを考える．以下の図で緑で塗られた領域がフィルタだとすると，左の画像から 1 列のデータが得られる．

1,1	1,2	1,3	1,4	1,5	1,6	1,7
2,1	2,2	2,3	2,4	2,5	2,6	2,7
3,1	3,2	3,3	3,4	3,5	3,6	3,7
4,1	4,2	4,3	4,4	4,5	4,6	4,7
5,1	5,2	5,3	5,4	5,5	5,6	5,7
6,1	6,2	6,3	6,4	6,5	6,6	6,7
7,1	7,2	7,3	7,4	7,5	7,6	7,7

1,1
1,2
1,3
2,1
2,2
2,3
3,1
3,2
3,3

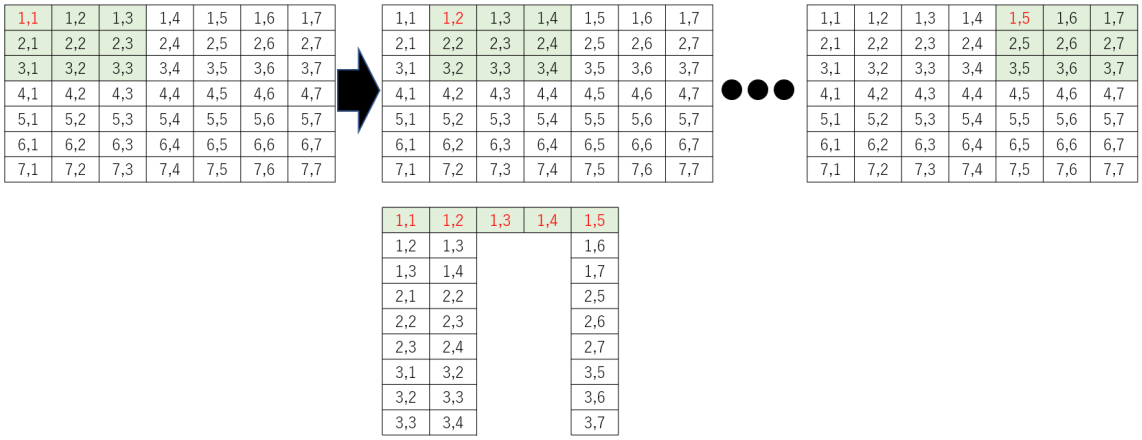
フィルタを 1 つ右にずらすと，さらに 1 列のデータが得られる．

1,1	1,2	1,3	1,4	1,5	1,6	1,7
2,1	2,2	2,3	2,4	2,5	2,6	2,7
3,1	3,2	3,3	3,4	3,5	3,6	3,7
4,1	4,2	4,3	4,4	4,5	4,6	4,7
5,1	5,2	5,3	5,4	5,5	5,6	5,7
6,1	6,2	6,3	6,4	6,5	6,6	6,7
7,1	7,2	7,3	7,4	7,5	7,6	7,7

1,1	1,2
1,2	1,3
1,3	1,4
2,1	2,2
2,2	2,3
2,3	2,4
3,1	3,2
3,2	3,3
3,3	3,4

ここで，得られるデータの列ではなく，行に着目する．フィルタ内の左上のマスの移動を追うと，以下のよ

うに 1 行のデータをなすことがわかる．



このことから，フィルタ内の相対位置 (i, j) の領域は，フィルタがずれる画像内の絶対位置の範囲 (i, j)〜$(i + di, j + dj)$ のデータを取り出すことがわかる．もし，$s > 1$ ならば，その範囲を s 個おきに抽出すればよい．これらの処理は，バッチとチャンネルに対して独立に行われ，その範囲のデータの 1 つ 1 つが出力画像の 1 マス分の畳み込み計算に使われることから，X = np.empty((B, ch, R, R, dh, dw)) に対して，あるフィルタの相対位置 (i,j) の領域を決めたときに，X[:, :, i, j, :, :] = batch[:, :, i:i+di+1:s, j:j+dj+1:s] は，画像データから畳み込みに必要なマスを取り出したものになる．これを全てのフィルタ内の相対位置に対して行い，X を整形すれば，目的の行列が得られる．

なお，この方法の理解に当たって以下のサイトを参考にした．

「深層学習／ im2col の実装の工夫に驚いた件」閲覧日 2020 年 11 月 25 日

https://qiita.com/jun40vn/items/d2e8711cabc9cfb1e0d5

順伝播 (forward)

convert_batch_into_X 関数を用いて，画像バッチを整形すると，単純な計算と整形で (B, K, dh, dw) 次元の出力画像が得られる．($dw \times dh$ が出力画像のサイズ)

```
def forward(self, x):
    # (R * R * ch, dw * dh * B)
    self.x, (B, dh, dw) = convert_batch_into_X(x, self.R, self.p, self.s)

    # 畳み込み演算
    y = (np.dot(self.W, self.x).T + self.b).T

    # (B, K, dh, dw)
    y = y.reshape(-1, B, dh, dw).transpose(1, 0, 2, 3)

    return y
```

ソースコード 18　Convolution の順伝播

行列 X から画像バッチへの変換 (convert_X_into_batch)

逆伝播の際には，convert_batch_into_X とは逆の変換を行う必要がある．これは単純に逆の操作を行えばよい．ただし，for 文の中身は，畳み込みによって分散した画像データの，勾配が元の位置に戻ってくるので，和を取っている．また，パディングを除く．

```
# 行列Xから画像バッチへの変換
def convert_X_into_batch(X, R, p, s, shape):
    B, dh, dw = shape

    # フィルタがずれた距離
    di = s * (dh - 1)
    dj = s * (dw - 1)

    # 元画像の縦横
    dy = di - 2 * p + R
    dx = dj - 2 * p + R

    # (B, ch, R, R, dh, dw)
    X = X.reshape(-1, R, R, B, dh, dw).transpose(3, 0, 1, 2, 4, 5)

    ch = X.shape[1]
    batch = np.zeros((B, ch, dy + 2 * p, dx + 2 * p))

    for i in range(R):
        for j in range(R):
            batch[:, :, i:i+di+1:s, j:j+dj+1:s] += X[:, :, i, j, :, :]

    # パディングを除く
    batch = batch[:, :, p:dy+p, p:dx+p]
    return batch
```

ソースコード 19　convert_X_into_batch

逆伝播 (backward)

順伝播と逆の整形を行い，逆伝播を行った後，上で定義した convert_X_into_batch 関数を使って，画像バッチの次元に戻す．

```
def backward(self, dEn_dY):
    B, K, dh, dw = dEn_dY.shape

    # (K, dw * dh * B)
    dEn_dY = dEn_dY.transpose(1, 0, 2, 3).reshape(K, -1)

    dEn_dX = np.dot(self.W.T, dEn_dY)
    dEn_dW = np.dot(dEn_dY, self.x.T)
    dEn_db = dEn_dY.sum(axis = 1)
    self.W -= self.adam_W.grad(dEn_dW)
    self.b -= self.adam_b.grad(dEn_db)

    # (B, ch, dy, dx)
    dEn_dX = convert_X_into_batch(dEn_dX, self.R, self.p, self.s, (B, dh, dw))
    return dEn_dX
```

ソースコード 20　Convolution の逆伝播

工夫した点

画像バッチと行列 X の変換を素朴に行うと，かなり時間がかかってしまうと思ったので，上記のような手法を用いた．

8　発展課題 A7 プーリング

max プーリングを Pooling クラスとして実装した．

初期化 (init)

プーリング幅 d を受け取る．

```
class Pooling:
    def init(self, d):
        self.d = d
```

ソースコード 21　Pooling の初期化

save/load

save/load するパラメータは，d である．

```
class Pooling:
...
    def save(self, para, name):
        para[name + '_d'] = self.d

    def load(self, para, name):
        self.d = para[name + '_d']
```

ソースコード 22　save/load

順伝播 (forward)

入力 x の次元を (B, ch, dy, dx) とすると，出力 y の次元は $(B, ch, dh, dw), dh = dy/d, dw = dx/d$ となる．ここで，$d \times d$ ごとに max をとるために，$d \times d$ 行 $dw \times dh \times ch \times B$ 列に整形し，列方向に max をとったあと，再度整形して，次元を (B, ch, dh, dw) にする．

例えば，4×4 のある 1 枚の画像に対して，$d = 2$ で処理する場合，以下のように動作する．

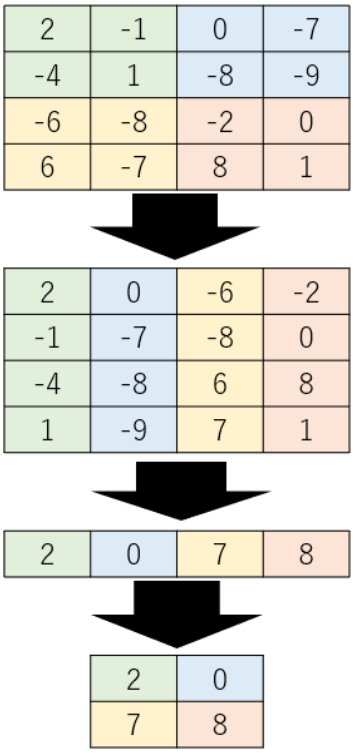


図 2　プーリングの動作例

このとき，逆伝播の際に必要な情報である，自身が max かどうかという情報も列方向にとっておき，self.mask として保持する．

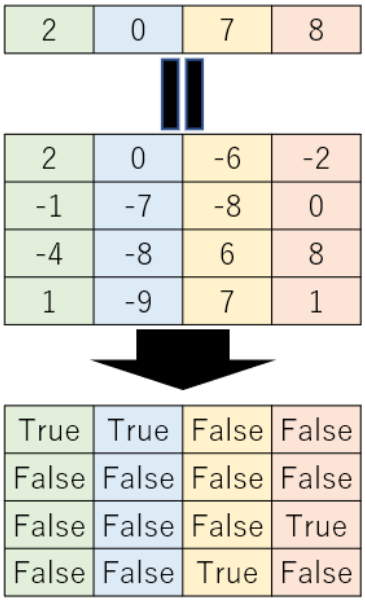


図 3　self.mask

```
class Pooling:
...
    def forward(self, x):
        self.x = x
        B, ch, dy, dx = x.shape

        # 出力画像のサイズ dw * dh
        dh = dy // self.d
        dw = dx // self.d

        # (d, d, B, ch, dh, dw)
        x = x.reshape(B, ch, dh, self.d, dw, self.d).transpose(3, 5, 0, 1, 2, 4)

        # (d * d, dw * dh * ch * B)
        x = x.reshape(self.d * self.d, -1)

        # dw * dh * ch * B
        y = x.max(axis = 0)

        #  (d * d, dw * dh * ch * B)
        self.mask = x == y

        # (B, ch, dh, dw)
        y = y.reshape(B, ch, dh, dw)

        return y
```

ソースコード 23　Pooling の順伝播

逆伝播 (backward)

引数である，(B, ch, dh, dw) 次元の $\frac{\partial E_n}{\partial Y}$ を $dw \times dh \times ch \times B$ 次元のベクトルに整形し，self.mask が True のときにその値，False のときに 0 としたものを $\frac{\partial E_n}{\partial X}$ とする．これは，$d \times d$ 行 $dx \times dy \times ch \times B$ 列の行列であり，(B, ch, dy, dx) に整形して返す．先ほどの例では，以下のようになる．

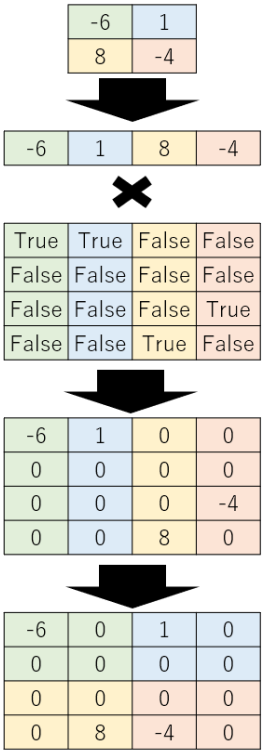


図 4　逆伝播の動作例

```
class Pooling:
...
    def backward(self, dEn_dY):
        B, ch, dh, dw = dEn_dY.shape

        # 元画像のサイズ dx * dy
        dy = dh * self.d
        dx = dw * self.d

        # dw *dh * ch * B
        dEn_dY = dEn_dY.reshape(-1)

        # (d * d, dw * dh * ch * B)
        dEn_dX = np.where(self.mask, dEn_dY, 0)

        # (B, ch, dy, dx)
        dEn_dX = dEn_dX.reshape(self.d, self.d, B, ch, dh, dw)
                       .transpose(2, 3, 4, 0, 5, 1).reshape(B, ch, dy, dx)
        return dEn_dX
```

ソースコード 24　Pooling の逆伝播

画像からベクトルへの整形

順伝播時に，プーリング層の出力を全結合層の入力にするために，次元 (B, ch, dy, dx) を $(dx \times dy \times ch, B)$ に変換する．逆伝播で元の次元が必要となるので，self.shape で記憶する．

```
class Pooling:
...
    # (B, ch, dy, dx) -> (dx * dy * ch, B)
    def convert_images_into_vectors(self, X):
        self.shape = X.shape
        B = self.shape[0]
        return X.transpose(1, 2, 3, 0).reshape(-1, B)
```

ソースコード 25　画像からベクトルへの整形

ベクトルから画像への整形

逆伝播時に，全結合層からのデータの次元を $(dx \times dy \times ch, B)$ から (B, ch, dy, dx) に変換する．変換には，順伝播時に記憶した self.shape を用いる．

```
class Pooling:
...
    # (dx * dy * ch, B) -> (B, ch, dy, dx)
    def convert_vectors_into_images(self, Y):
        B, ch, dy, dx = self.shape
        return Y.reshape(ch, dy, dx, B).transpose(3, 0, 1, 2)
```

ソースコード 26　ベクトルから画像への整形

工夫点

画像データを，max をとる領域ごとに列で並べた行列に変換することで max の演算を書きやすくした．

9　全結合層 (Affine)

コンストラクタ

重み行列 W とバイアスベクトル b を更新するための Adam を生成する．

```
class Affine:
    def __init__(self):
        self.adam_W = Adam()
        self.adam_b = Adam()
```

ソースコード 27　Affine のコンストラクタ

初期化

row 行 col 列の重み行列 W と row 次元のバイアスベクトル b を random_ndarray 関数を使って乱数で初期化する．col が手前の層のノード数である．

```
class Affine:
...
    def init(self, row, col):
        self.W = random_ndarray(col, (row, col))
```

```
        self.b = random_ndarray(col, row)
...
# 乱数配列
def random_ndarray(N, shape):
    return np.random.normal(0, 1 / np.sqrt(N), shape)
```

ソースコード 28　Affine の初期化

save/load

これまでと同様の方法で，W,b とそれらを更新する Adam のパラメータを save/load する．

```
class Affine:
...
    def save(self, para, name):
        para[name + '_W'] = self.W
        para[name + '_b'] = self.b
        self.adam_W.save(para, name + '_adam_W')
        self.adam_b.save(para, name + '_adam_b')

    def load(self, para, name):
        self.W = para[name + '_W']
        self.b = para[name + '_b']
        self.adam_W.load(para, name + '_adam_W')
        self.adam_b.load(para, name + '_adam_b')
```

ソースコード 29　save/load

順伝播 (forward)

$y = Wx + b$ を返す．W が (row, col)，x が (col, B) 次元の場合，Wx は (row,B) 次元となる．Wx の各列に row 次元ベクトル b を足すという計算を numpy で書くために，転置 T を行っている．

```
$
class Affine:
...
    def forward(self, x):
        self.x = x
        y = (np.dot(self.W, x).T + self.b).T
        return y
```

ソースコード 30　Affine の順伝播

逆伝播 (backward)

講義資料の式に基づいて勾配を求め，Adam を通して重みを更新する．

```
class Affine:
...
    def backward(self, dEn_dY):
        dEn_dX = np.dot(self.W.T, dEn_dY)
        dEn_dW = np.dot(dEn_dY, self.x.T)
        dEn_db = dEn_dY.sum(axis = 1)
```

```
        self.W -= self.adam_W.grad(dEn_dW)
        self.b -= self.adam_b.grad(dEn_db)
        return dEn_dX
```

ソースコード 31　Affine の逆伝播

10　ソフトマックス層 (Softmax)

順伝播 (forward)

出力を self.y に格納する．

```
class Softmax:
    def forward(self, a):
        ex = np.exp(a - np.max(a, axis = 0))
        self.y = ex / np.sum(ex, axis = 0)
        return self.y
```

ソースコード 32　Softmax の順伝播

逆伝播 (backward)

実装の都合上，クロスエントロピー誤差の計算も含めた逆伝播を行う．B はバッチサイズで，y は (one-hot vector 表記ではない)B 次元の正解データのベクトルである．

ここで，Softmax 関数の入力 $a_k(k = 1, 2, \cdots, C)$，出力 $y_k^{(2)}$ と正解データ $i \in \{0, 1, \cdots, 9\}$ に対して，$\frac{\partial E_n}{\partial a_k}$ は以下のように書ける．

$$\frac{\partial E_n}{\partial a_k} = \begin{cases} \frac{y_k^{(2)} - 1}{B} & (k = i) \\ \frac{y_k^{(2)}}{B} & (k \neq i) \end{cases}$$

そこで，まず self.y[y, cols] -= 1 で $k = i$ の場合の分子の-1 を計算した後，self.y /= B で全体を B で割ったものを返すようにした．

```
class Softmax:
...
    def backward(self, y, B):
        cols = np.arange(B)
        self.y[y, cols] -= 1
        self.y /= B
        return self.y
```

ソースコード 33　Softmax の逆伝播

工夫点

後で述べるように，正解データを one-hot vector 表記に変換するのは不便になったので，そのままの状態で正しく逆伝播できるようにした．

11　ニューラルネットワーク (Neural_network)

以上の層を組み合わせてニューラルネットワークを実装する．

初期化

乱数で初期化する際に呼び出す．各層を dict 型の self.layers で管理する．まず，create_layers 関数で各層を生成する．ニューラルネットワークの構成は，冒頭で示した図 1 のとおりである．次に，パラメータを設定して初期化する必要のある層に対して，init メソッドを呼び出し，初期化する．epoch_count は学習済みのエポック数で保存するためのパラメータである．

ch は入力画像のチャンネル数である．K,R,p,s はそれぞれ畳み込み層におけるフィルタ数，フィルタサイズ，パディング，ストライドである．d2 はプーリング幅である．d,M,C は 2 つの全結合層における次元数で，入力データが，1 つ目で d 次元から M 次元になり，2 つ目で M 次元から C 次元になる．

```
class Neural_network:
    # 初期化
    def init(self, d, M, C, ch, K, R, p, s, d2):
        self.layers = create_layers()
        self.layers['conv'].init(ch, K, R, p, s)
        self.layers['pooling'].init(d2)
        self.layers['affine1'].init(M, d)
        self.layers['affine2'].init(C, M)
        self.epoch_count = 0
...
# ネットワーク層生成
def create_layers():
    return {'conv': Convolution(),
            'bn1': Batch_normalization(),
            'relu1': ReLU(),
            'pooling': Pooling(),
            'affine1': Affine(),
            'bn2': Batch_normalization(),
            'relu2': ReLU(),
            'dropout': Dropout(),
            'affine2': Affine(),
            'softmax': Softmax()}
```

ソースコード 34　初期化

save

file_name で指定した.npz ファイルに学習に用いるパラメータを全て保存する．dict 型の para に各層のパラメータを格納したあと，np.savez の引数として展開する．self.layers の層で，save メソッドを持つものに対して，その変数名 name と para を渡し，各層で定義されている save メソッドによって，パラメータを para に格納する．name は一意に定まる key を生成するのに使われる．

```
class Neural_network:
...
    # パラメータ保存
```

```
    def save(self, file_name):
        para = {'epoch_count': self.epoch_count}

        # 各層で save を呼び出す
        for name, layer in self.layers.items():
            if hasattr(layer, 'save'):
                layer.save(para, name)

        np.savez(file_name, **para)
```

ソースコード 35　save

load

file_name で指定した.npz ファイルから学習に用いるパラメータを読み込む．まず，np.load でパラメータの dict を得る．次に，create_layers 関数で各層を生成後，load メソッドを持つ層に対して，その変数名 name と para を渡して load メソッドを呼び出し，それぞれの load の定義にしたがってパラメータが読み込まれる．何を保存すべきかは各層で定義されているので，外部からはこのように簡単に書ける．

```
class Neural_network:
...
    # パラメータ読み込み
    def load(self, file_name):
        para = np.load(file_name)

        # 各層で load を呼び出す
        self.layers = create_layers()
        for name, layer in self.layers.items():
            if hasattr(layer, 'load'):
                layer.load(para, name)

        self.epoch_count = para['epoch_count']
```

ソースコード 36　load

順伝播 (forward)

引数は，正規化された (N, ch, dy, dx) 次元の画像データ x と学習時に True になるフラグ is_train である．図 1 の構造に従って，順伝播を行う．ここで，畳み込み層の出力データの次元を (B, ch, dh, dw) から $(dw \times dh \times ch, B)$ に変換した後，Batch Normalization に通し，その後元に戻している．また，Pooling 層の convert_images_into_vectors メソッドでも同様の変換を行い，このとき形状を記憶しておく．

```
class Neural_network:
...
    # 順伝播
    def forward(self, x, is_train = False):
        # 畳み込み
        x = self.layers['conv'].forward(x)

        # Batch Normalization
        B, ch, dh, dw = x.shape
```

```
        # (dw * dh * ch, B)
        x = x.transpose(1, 2, 3, 0).reshape(-1, B)
        x = self.layers['bn1'].forward(x, is_train)

        # (B, ch, dh, dw)
        x = x.reshape(ch, dh, dw, B).transpose(3, 0, 1, 2)

        # ReLu と Pooling
        x = self.layers['relu1'].forward(x)
        x = self.layers['pooling'].forward(x)
        # (d, B)
        x = self.layers['pooling'].convert_images_into_vectors(x)

        x = self.layers['affine1'].forward(x)
        x = self.layers['bn2'].forward(x, is_train)
        x = self.layers['relu2'].forward(x)
        x = self.layers['dropout'].forward(x, is_train)
        x = self.layers['affine2'].forward(x)
        x = self.layers['softmax'].forward(x)
```

ソースコード 37　順伝播

ミニバッチ学習 (mini_batch_training)

mini_batch_training の入力 X は正規化された (N, ch, dy, dx) 次元の画像データ，Y は正解データの N 次元ベクトルである．まず，create_mini_batch 関数でミニバッチを生成する．次に，is_train を True にして forward メソッドを呼び出す．そして，cross_entropy 関数でクロスエントロピー誤差を求め，その平均と使用したミニバッチを返す．

ここで，講義資料記載の以下の式

$$E = \sum_{k=1}^{C} -y_k \log y_k^{(2)}$$

は正解 $y_k = 1$ となる k を i としたとき，

$$E = -\log y_i^{(2)}$$

と書ける．そこで，各列に対して i 番目のデータを取り出して log をとるようにした．正解データを one-hot vector 表記にすると，冗長な部分が生じたのでそのままの状態で求めるようにした．

```
class Neural_network:
...
    # ミニバッチ学習
    def mini_batch_training(self, X, Y, B):
        mini_batch = X, Y = create_mini_batch(X, Y, B)
        Y2 = self.forward(X, is_train = True)
        En = np.mean(cross_entropy(Y, Y2))
        return mini_batch, En
...
# ミニバッチ生成
def create_mini_batch(X, Y, B):
    N = len(X)
    indexes = np.random.choice(N, B, False)
    X = X[indexes]
    Y = Y[indexes]
```

```
    return X, Y

# クロスエントロピー誤差
def cross_entropy(y, y2):
    B = y2.shape[1]
    cols = np.arange(B)
    E = -np.log(y2[y, cols])
    return E
```

ソースコード 38　ミニバッチ学習

逆伝播 (bacward)

引数は，訓練データ X_train とその正解ベクトル Y_train，テスト用データ X_test とその正解ベクトル Y_test，バッチサイズ B，エポック数 epoch である．X_train は正規化された (N, ch, dy, dx) 次元の画像データである．X_test も同様の形式である．

訓練データの総数 N，B，epoch から学習回数を計算する．1 エポックあたりの E_n の和を En_sum で保持する．

rep 回の学習を for 文で繰り返す．まず，mini_batch_training メソッドでミニバッチ学習を行い，使用した mini_batch と E_n を得る．それから，各層で逆伝搬を行う．Pooling ではまず，convert_vectors_into_images メソッドで，順伝播時に記憶している形状に画像データを整形する．Batch Normalization では，順伝播と同様に，一旦，次元を $(dw \times dh \times ch, B)$ に変換してから逆伝播を行い，元に戻す．

1 エポックごとに E_n の平均と全ての訓練データ，テストデータに対する正答率をそれぞれ求め，出力する．check_accuracy メソッドは次で述べる..

```
# 誤差逆伝播
def backward(self, X_train, Y_train, X_test, Y_test, B, epoch):
    # 訓練データの総数
    N = len(X_train)
    #  1エポックあたりのミニバッチ学習の回数
    rep_per_epoch = N // B
    # ミニバッチ学習の総回数
    rep = epoch * rep_per_epoch

    # 損失関数の和
    En_sum = 0

    for i in range(rep):
        # ミニバッチ学習
        mini_batch, En = self.mini_batch_training(X_train, Y_train, B)
        # ミニバッチの正解データ
        Y = mini_batch[1]

        # 逆伝播
        dEn_dX = self.layers['softmax'].backward(Y, B)
        dEn_dX = self.layers['affine2'].backward(dEn_dX)
        dEn_dX = self.layers['dropout'].backward(dEn_dX)
        dEn_dX = self.layers['relu2'].backward(dEn_dX)
        dEn_dX = self.layers['bn2'].backward(dEn_dX)
        dEn_dX = self.layers['affine1'].backward(dEn_dX)

        # Pooling と ReLU
```

```
        dEn_dX = self.layers['pooling'].convert_vectors_into_images(dEn_dX)
        dEn_dX = self.layers['pooling'].backward(dEn_dX)
        dEn_dX = self.layers['relu1'].backward(dEn_dX)

        # Batch Normalization
        B, ch, dh, dw = dEn_dX.shape

        # (dw, dh * ch, B)
        dEn_dX = dEn_dX.transpose(1, 2, 3, 0).reshape(-1, B)
        dEn_dX = self.layers['bn1'].backward(dEn_dX)

        # (B, ch, dh, dw)
        dEn_dX = dEn_dX.reshape(ch, dh, dw, B).transpose(3, 0, 1, 2)

        # 畳み込み
        dEn_dX = self.layers['conv'].backward(dEn_dX)

        # 1エポックごとに、Enの平均と訓練データ，テストデータに対する正答率を出力
        if (i + 1) % rep_per_epoch == 0:
            self.epoch_count += 1
            En_avg = En_sum / rep_per_epoch

            # 正答率を求める
            accuracy_train = self.check_accuracy(X_train, Y_train)
            accuracy_test = self.check_accuracy(X_test, Y_test)

            print(self.epoch_count, En_avg, accuracy_train, accuracy_test)
            En_sum = 0
        else:
            En_sum += En
```

ソースコード 39　逆伝播

性能評価 (check_accuracy)

引数は，次元が (N, ch, dy, dx) の正規化された画像データ X とその正解データである N 次元ベクトル Y である．畳み込み層を入れると 60000 枚を一度に推論することは難しかったので，$B = 1000$ 枚ごとに行っている．ここで，forward メソッドの is_train が省略されているため，デフォルト値の False となり，テスト時の動作となるので，B 個の各データは独立に推論される．出力層の次元数を C，バッチサイズを B とすると，forward は C 行 B 列の行列を返す．列ごとに argmax をとることで，B 個のデータに対する推論が得られる．これと正解データ Y が一致するものの個数を数えて，総数 N で割ったものを返す．

```
class Neural_network:
...
    # 性能評価
    def check_accuracy(self, X, Y):
        N = len(X)
        B = 1000
        correct = sum(np.sum(self.forward(X[i:i+B]).argmax(axis = 0) == Y[i:i+B])
                      for i in range(0, N, B))
        return correct / N
```

ソースコード 40　性能評価

工夫点

- 各層をクラスで実装し，初期化，順伝播，逆伝播，save，load などの詳細な動作をそれぞれのクラスで定義することでニューラルネットワークの本体クラスである Neural_network クラス内で共通するメソッドを呼び出せばいいようにした．
- str 型を key とする dict 型 para に学習に使う全てのパラメータを格納するようにした．各層で, para と自身の変数名の文字列を引数とする save/load メソッドを定義し, 変数名から一意に定まる key を作り, save/load する必要のあるパラメータをやりとりするようにした．こうすることで，Neural_newwork クラスでは，単純な for 文で全てのパラメータを para に格納することができ，save/load が簡単になった．
- エポックごとに訓練データとテストデータの両方の正答率を出力するようにし，学習の進度を追えるようにした．

問題点

ニューラルネットワークの構造を固定して実装したので，柔軟に構造を変えることができない．外部から各層を任意に組み合わせることができるように実装できたらよいと感じた．

12 実行ファイル (mnist)

neural_network.py を使って，mnist の画像データの学習をするための実行ファイルについて説明する．

mnist_tool.py

このファイルは，mnist に関する課題の main ファイル内で from mnist_tool import *で import される．前半部でシードの設定と mnist のデータのファイル名と読み込み関数，ニューラルネットワークの各層の次元数，パラメータ保存用のファイル名を定義している．

```
import mnist
import numpy as np

# シードの設定
np.random.seed(100)

# 読み込み関数のエイリアス
load_file = mnist.download_and_parse_mnist_file

# データファイル
train_images_file = 'train-images-idx3-ubyte.gz'
train_labels_file = 'train-labels-idx1-ubyte.gz'
test_images_file = 't10k-images-idx3-ubyte.gz'
test_labels_file = 't10k-labels-idx1-ubyte.gz'

# 入力画像の次元数 dy * dx
dx = dy = 28
```

```
# 畳み込み層
ch = 1
K = 4
R = 3
p = R // 2
s = 1

# プーリング幅
d2 = 2

# 畳み込み後の次元数 dw * dh
dh = (2 * p + dy - R) // s + 1
dw = (2 * p + dx - R) // s + 1

# 全結合層の次元数
d = K * dh * dw // (d2 * d2)
M = 128
C = 10

# パラメータ保存用ファイル
parameters_file = 'mnist_para.npz'
```

ソースコード 41　グローバル変数

次に，前処理と後処理を定義している．前処理では，畳み込み層を実装した関係でチャンネル数が 1 であることを表現するために，(N, dy, dx) 次元を $(N, 1, dy, dx)$ 次元にしたあと，255 で割る．後処理は，argmax を返す．

```
# 前処理
def pre_process(X):
    # (N, 1, dy, dx)
    X = X[:, np.newaxis, :, :] / 255
    return X

# 後処理
def post_process(y):
    i = np.argmax(y)
    return i
```

ソースコード 42　前処理と後処理

kadai3.py

誤差逆伝播法によって mnist の画像データの学習を行う main ファイルである．mnist_tool.py で定義した load_file 関数や pre_process 関数を使い，訓練データとテスト用データを読み込み，前処理を行う．再学習をするか尋ね，load または init を行う．エポック数を標準入力から受け取り，バッチサイズ $B = 128$ で誤差逆伝播法を行った後，パラメータを保存する．

```
from mnist_tool import *
from neural_network import Neural_network

def main():
    # 訓練データの読み込み
    X_train = load_file(train_images_file)
```

```
    Y_train = load_file(train_labels_file)

    # テスト用データの読み込み
    X_test = load_file(test_images_file)
    Y_test = load_file(test_labels_file)

    # 前処理
    X_train = pre_process(X_train)
    X_test = pre_process(X_test)

    # ニューラルネットワーク生成
    nn = Neural_network()

    if input('再学習しますか␣>>␣[y,n]') == 'y':
        nn.load(parameters_file)
    else:
        nn.init(d, M, C, ch, K, R, p, s, d2)

    # バッチサイズ
    B = 128
    # エポック数
    epoch = int(input('エポック数␣>>␣'))

    # 誤差伝播
    nn.backward(X_train, Y_train, X_test, Y_test, B, epoch)

    # パラメータを保存
    nn.save(parameters_file)

if __name__ == '__main__':
    main()
```

ソースコード 43　kadai3.py

kadai4.py

課題 4 のための実行ファイルである．テストデータを読み込み，前処理したあと，0〜9999 の整数 i を 1 つ受け取り，i 番目の画像データを取り出す．ここで，ニューラルネットワークの実装の都合上，バッチサイズ 1 として順伝播を行う必要があり，None で次元を追加している．

ニューラルネットワークを生成し，パラメータを読み込み，順伝播，後処理を行って 0〜9 の推論結果を得る．ターミナルに推論結果と正解データを出力し，plt.imshow で入力画像を表示する．

```
from mnist_tool import *
from neural_network import Neural_network
import matplotlib.pyplot as plt
from pylab import cm

def main():
    # テストデータの読み込み
    X = load_file(test_images_file)
    Y = load_file(test_labels_file)

    # 前処理
    X = pre_process(X)
```

```
    # 入力
    i = int(input('0 ~ 9999から1つ入力してください >> '))
    x = X[i][None,:,:,:]

    # ニューラルネットワーク生成
    nn = Neural_network()
    nn.load(parameters_file)

    # 順伝播
    y = nn.forward(x)

    # 後処理
    y = post_process(y)

    # 結果の表示
    print('推論:',  y)
    print('正解:',  Y[i])
    plt.imshow(X[i].reshape(dy, dx), cmap = cm.gray)
    plt.show()

if __name__ == '__main__':
    main()
```

ソースコード 44　kadai4.py

neural_accuracy.py

学習したパラメータを用いて，全ての訓練データとテスト用データに対して推論を行い，正答率を求める．

```
from mnist_tool import *
from neural_network import Neural_network

def main():
    nn = Neural_network()
    nn.load(parameters_file)

    # 教師用データ
    X = load_file(train_images_file)
    Y = load_file(train_labels_file)
    X = pre_process(X)
    accuracy = nn.check_accuracy(X, Y)
    print(' 訓練データ', accuracy)

    # テスト用データ
    X = load_file(test_images_file)
    Y = load_file(test_labels_file)
    X = pre_process(X)
    accuracy = nn.check_accuracy(X, Y)
    print('テストデータ', accuracy)

if __name__ == '__main__':
    main()
```

ソースコード 45　neural_accuracy.py

13　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せよ．

実行方法

「0」と入力した結果，以下のように出力される．

```
$ python kadai4.py
0 ~ 9999から1つ入力してください >> 0
推論: 7
正解: 7
```

ソースコード 46　kadai4.py の実行方法

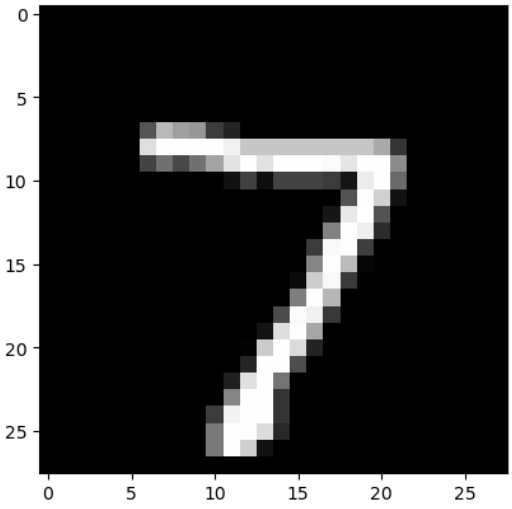


図 5　入力画像

工夫点

正解データおよび，plt.imshow による画像データの表示を行い，推論が正しいか分かるようにした．

14　mnist の学習

問題

Batch Normalization のところで述べたが，ニューラルネットワークに畳み込み層を加えると，クロスエントロピー誤差が順調に小さくなるのに対して，訓練データに対する正答率が急激に落ちるという現象が起きた．クロスエントロピー誤差は小さいままなので，訓練時とテスト時の動作の違いによって引き起こされると考えた．今回の実装で訓練時とテスト時で動作が異なるのは Batch Normalization と Dropout である．両者に対して，テスト時にも訓練時と同じ挙動で順伝播するようにしたところ，Batch Normalization のみそうすることで正答率が戻った (90 %後半)．このことから算術平均で求めている平均と分散の期待値が何らかの理由で適切な値になっていないと考えた．

そこで，Batch Normalization のところで述べたように，移動平均を使って平均と分散の期待値を求めるように変更したところ，非常に高い精度で学習が進んだ．これらについてまとめる．

実行結果

まず，畳み込み層を実装する前の状態で中間層の次元数 128 で 100 エポック学習させたときの「クロスエントロピー誤差」,「訓練データの正答率」を以下に示す．このように，正答率が急激に落ちるといったことは起きていない．

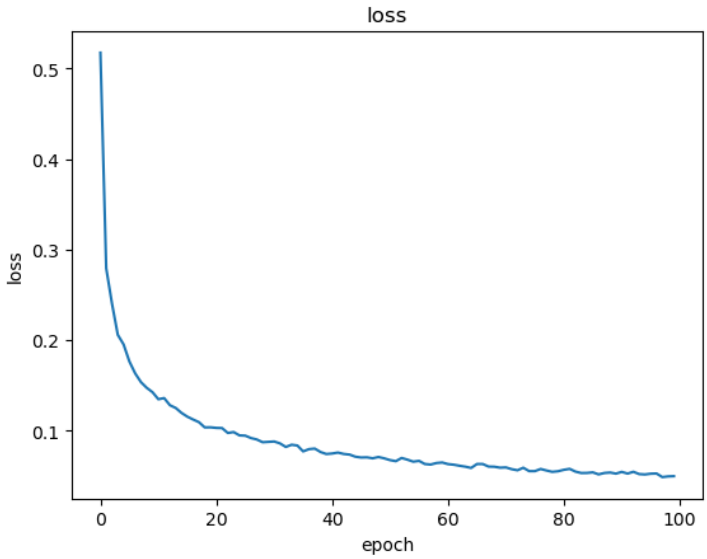


図 6　畳み込み無しの場合のクロスエントロピー誤差

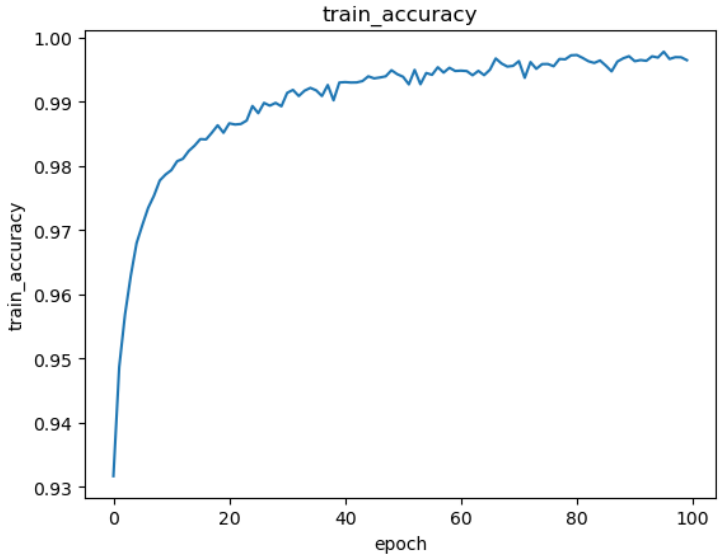


図 7　畳み込み無しの場合の訓練データの正答率

次に，畳み込み層とプーリング層を加える．フィルタ枚数 $K = 4$，フィルタサイズ $R = 3$，パディングとストライドは 1，プーリング幅は 2 とした．このとき，1 つ目の全結合層の入力の次元数は 728 で，2 つ目は 128 次元とした．50 エポック学習させたときの「クロスエントロピー誤差」,「訓練データの正答率」を以下に示す．このように，クロスエントロピー誤差が順調に小さくなっているのにも関わらず，訓練データの正答率が 30 エポックを過ぎたあたりから急落する．

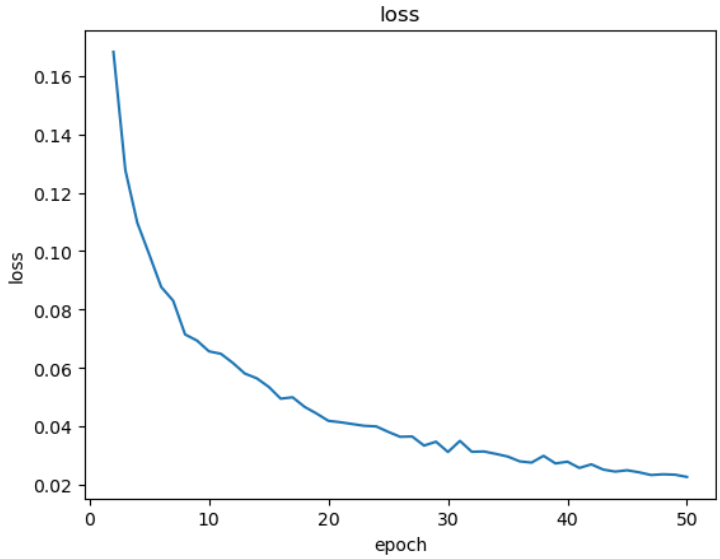


図 8　畳み込み有りの場合のクロスエントロピー誤差

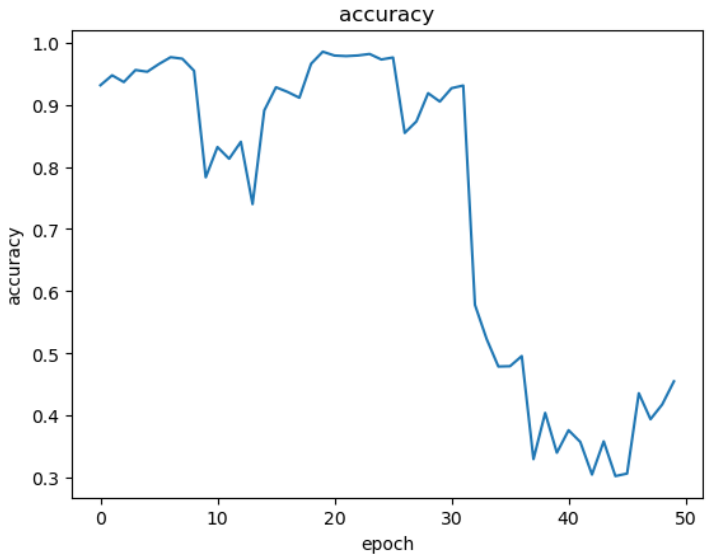


図 9　畳み込み有りの場合の訓練データの正答率

上記のネットワークに対して，Batch Normalization でのノードの平均と分散の期待値の計算に移動平均を使うように変更する．このときの「クロスエントロピー誤差」,「訓練データの正答率」は以下のようになる．学習時の挙動は上の場合と同じなので誤差のグラフは同一である．それに対し，正答率の推移が非常によくなったことが分かる．(1 エポック目の正答率は移動平均を使っているため，0.1 程度と低く，グラフが見にくくなるので，2 エポック目から表示している．)

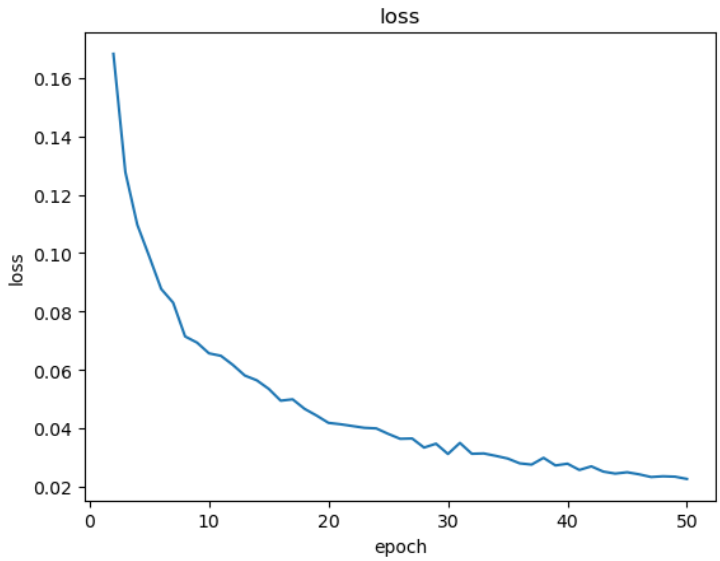


図 10　畳み込み有り，BN に移動平均，の場合のクロスエントロピー誤差

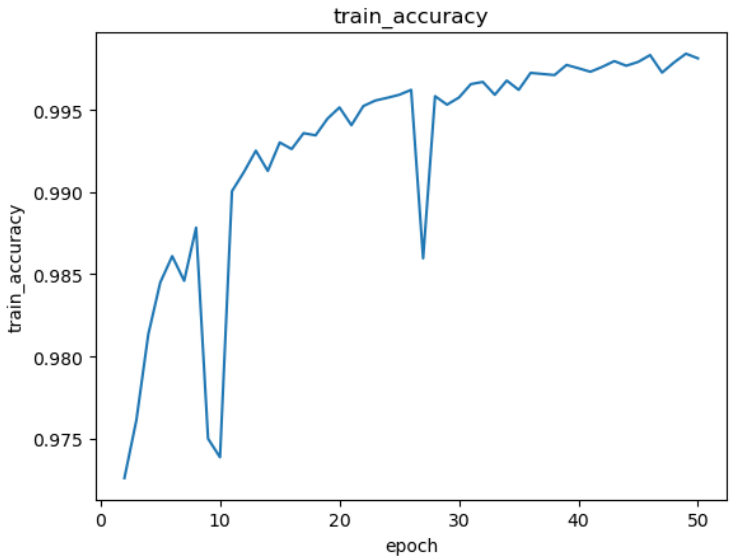


図 11　畳み込み有り，BN に移動平均，の場合の訓練データの正答率

また，このときの正答率を mnist_accuracy.py で求めると以下のようになった．

```
$ python mnist_accuracy.py
  訓練データ 0.9981333333333333
テストデータ 0.9834
```

ソースコード 47　正答率

結論

上記の結果から，畳み込み層を入れた場合には，Batch Normalization でのノードの分散と平均の期待値は，移動平均を使ったほうがよいという結果が得られた．

一方で，何故このような現象が起きるのか，詳しい理由は解明できなかった．もしかすると，別の場所にバグがあって移動平均を使ったところ，たまたま改善された可能性もある．今後の課題にしたい．

15　実行ファイル (CIFAR-10)

neural_network.py を使って，CIFAR-10 の画像データの学習をするための実行ファイルについて説明する．

cifar_tool.py

このファイルは，CIFAR-10 に関する課題の main ファイル内で from cifar_tool import *で import される．前半部でシードを設定し，ニューラルネットワークの各層の次元数，パラメータ保存用のファイル名を定義している．

```
import numpy as np
import pickle
```

```
# シードの設定
np.random.seed(100)

# プーリング幅
d2 = 2

# 畳み込み層
ch = 3
K = 8
R = 3
p = R // 2
s = 1

# 入力画像の次元数
dx = dy = 32

dh = (2 * p + dy - R) // s + 1
dw = (2 * p + dx - R) // s + 1

# 全結合層の次元数
d = K * dh * dw // (d2 * d2)
M = 256
C = 10

# パラメータ保存用ファイル
parameters_file = 'cifar_para.npz'
```

ソースコード 48　グローバル変数

次に訓練データの読み込み関数を定義している．unpickle は講義資料記載のものを利用している．load_train_file は，data_batch_1〜data_batch_5 を一度に読み込んで 50000 枚の訓練データを返す関数である．

```
def unpickle(file):
    with open(file, 'rb') as fo:
        dict = pickle.load(fo,encoding='bytes')
    X = np.array(dict[b'data'])
    X = X.reshape((X.shape[0],3,32,32))
    Y = np.array(dict[b'labels'])
    return X,Y

# 5つの訓練データを読み込む
def load_train_file():
    N, ch, dx, dy = 10000, 3, 32, 32
    n = 5

    X = np.empty((N * n, ch, dy, dx), dtype = np.int32)
    Y = np.empty(N * n, dtype = np.int32)

    for i in range(n):
        l = i * N
        r = l + N
        x, y = unpickle('./cifar-10-batches-py/data_batch_' + str(i + 1))
        X[l:r] = x
        Y[l:r] = y
```

```
    return X, Y
```

ソースコード 49　訓練データの読み込み

テストデータの読み込み関数も unpickle 関数を用いて以下のように定義している．

```
# テストデータを読み込む
def load_test_file():
    X, Y = unpickle('./cifar-10-batches-py/test_batch')
    return X, Y
```

ソースコード 50　テストデータの読み込み

前処理は入力データを 255 で割る．

```
# 前処理
def pre_process(X):
    X = X / 255
    return X
```

ソースコード 51　前処理

後処理は，ニューラルネットワークの出力ベクトルに argmax を取った後，対応するクラスの文字列を返す．

```
# 後処理
label = ['airplane', 'automobile', 'bird', 'cat' , 'deer',
         'dog', 'frog', 'horse', 'ship', 'truck']
def post_process(y):
    i = np.argmax(y)
    return label[i]
```

ソースコード 52　後処理

工夫点

5 つの訓練データを load_train_file 関数でまとめて読み込み，1 度に学習できるにした．

cifar3.py

課題 3(kadai3.py) に相当する．誤差逆伝播法によって CIFAR-10 の画像データの学習を行う main ファイルである．cifar_tool.py で定義した load_train_file 関数や load_test_file 関数で訓練データとテストデータを読み込み，pre_process 関数でそれぞれ前処理する．再学習をするか尋ね，load または init を行う．エポック数を標準入力から受け取り，バッチサイズ $B = 128$ で誤差逆伝播法を行った後，パラメータを保存する．

```
from cifar_tool import *
from neural_network import Neural_network

def main():
    # 訓練用データの読み込み
    X_train, Y_train = load_train_file()

    # テストデータの読み込み
    X_test, Y_test = load_test_file()

    # 前処理
    X_train = pre_process(X_train)
```

```
    X_test = pre_process(X_test)

    # ニューラルネットワーク生成
    nn = Neural_network()

    if input('再学習しますか␣>>␣[y,n]') == 'y':
        nn.load(parameters_file)
    else:
        nn.init(d, M, C, ch, K, R, p, s, d2)

    # バッチサイズ
    B = 128
    # エポック数
    epoch = int(input('エポック数␣>>␣'))

    # 誤差伝播
    nn.backward(X_train, Y_train, X_test, Y_test, B, epoch)

    # パラメータを保存
    nn.save(parameters_file)

if __name__ == '__main__':
    main()
```

ソースコード 53　cifar3.py

cifar4.py

mnist に対する課題 4(kadai4.py) に相当する．テストデータを読み込み，前処理したあと，0〜9999 の整数 i を 1 つ受け取り，i 番目の画像データを取り出す．ここで，ニューラルネットワークの実装の都合上，バッチサイズ 1 として順伝播を行う必要があり，None で次元を追加している．

ニューラルネットワークを生成し，パラメータを読み込み，順伝播，後処理を行って，10 クラスのうちの 1 クラスの文字列を得る．ターミナルに推論結果と正解クラスを出力し，plt.imshow で入力画像を表示する．

```
from cifar_tool import *
from neural_network import Neural_network
import matplotlib.pyplot as plt

def main():
    # 訓練用データの読み込み
    X, Y = load_test_file()

    # 前処理
    X = pre_process(X)

    # 入力
    i = int(input('0␣~␣9999から1つ入力してください␣>>␣'))
    x = X[i][np.newaxis, :, :, :]

    # ニューラルネットワーク生成
    nn = Neural_network()
    nn.load(parameters_file)

    # 順伝播
```

```
    y = nn.forward(x)

    # 後処理
    y = post_process(y)

    # 結果の表示
    print('推論:',  y)
    print('正解:',  label[Y[i]])
    plt.imshow(X[i].transpose(((1,2,0))))
    plt.show()

if __name__ == '__main__':
    main()
```

ソースコード 54　kadai4.py

cifar_accuracy.py

学習したパラメータを用いて，CIFAR-10 の全ての訓練データとテスト用データに対して推論を行い，正答率を求める．

```
from cifar_tool import *
from neural_network import Neural_network

def main():
    nn = Neural_network()
    nn.load(parameters_file)

    # 教師用データ
    X, Y = load_train_file()
    X = pre_process(X)
    accuracy = nn.check_accuracy(X, Y)
    print(' 訓練データ', accuracy)

    # テスト用データの読み込み
    X, Y = load_test_file()
    X = pre_process(X)
    accuracy = nn.check_accuracy(X, Y)
    print('テストデータ', accuracy)

if __name__ == '__main__':
    main()
```

ソースコード 55　neural_accuracy.py

16　発展課題 A5 CIFAR-10

課題内容

カラー画像である CIFAR-10 に対して 10 クラス識別を行う．

実行方法

cifar3.py

まず，cifar3.py で学習を行う．再学習を「n」，エポック数を「5」と入力すると以下のように出力される．各行は，「エポック」，「クロスエントロピー誤差」，「訓練データの正答率」，「テストデータの正答率」がスペース区切りで出力されている．

```
$ python cifar3.py
再学習しますか >> [y,n]n
エポック数 >> 5
1 1.460824580608643 0.51034 0.4835
2 1.1814402021760797 0.6153 0.5613
3 1.0830179405876499 0.65364 0.5978
4 1.0281391832276607 0.68398 0.6072
5 0.9656037821433356 0.70162 0.6094
```

ソースコード 56　cifar3.py の実行方法

cifar4.py

「11」と入力した結果，以下のように出力される．

```
$ python cifar4.py
0 ~ 9999から1つ入力してください >> 11
推論: truck
正解: truck
```

ソースコード 57　kadai4.py の実行方法

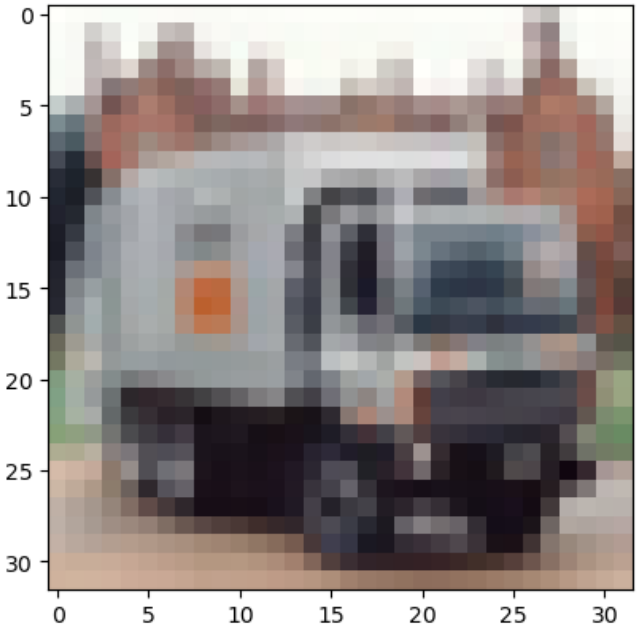


図 12　入力画像

工夫点

推論結果と正解データに対応するクラスの文字列を出力し，plt.imshow で表示されるカラー画像と比較できるようにした．

学習結果

畳み込み層のフィルタ枚数 $K = 8$，フィルタサイズ $R = 3$，パディングとストライドは 1，2 つ目の全結合層の入力次元数は 256 として，50 エポック学習させたときの，「クロスエントロピー誤差」，「訓練データとテストデータの正答率」を以下に示す．

クロスエントロピー誤差は順調に下がり，訓練データに対する正答率も 9 割を越えたが，テストデータに対する正答率は 6 割程度にとどまった．このことから，現在の畳み込みニューラルネットワークの構造では，単純に訓練データの数値の集まりを学習しているだけで，カラー画像に含まれる物体の特徴を学習できていないと思われる．

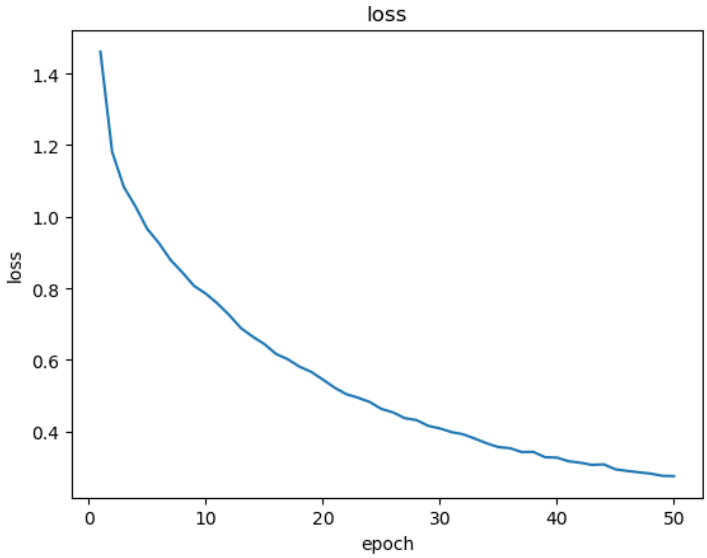


図 13　クロスエントロピー誤差

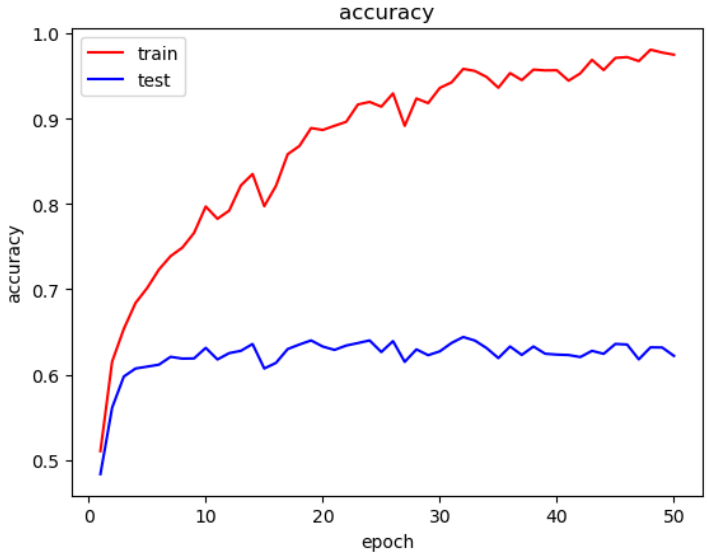


図 14　訓練データとテストデータの正答率

問題点

以上の結果より，現在の実装ではカラー画像を識別するのに十分な特徴を学習できていない．改善点として，畳み込み層とプーリング層をいくつも重ね，特徴を抽出していくといったことが重要だと考える．