ディープラーニングの実装課題

1 はじめに

近年、深層学習の発展に伴い、様々な深層学習ライブラリが登場している (e.g. caffe, theano, tensorflow, chainer, torch, etc...)．これらのライブラリは深層学習を簡単に行うことができるように作成されたライブラリであり、深層学習のアルゴリズムを知らなくても容易に学習を行うことができる．

本実験では、そのようなライブラリは使わずに深層学習の基本的なアルゴリズムを理解・実装してもらい、その仮定を通じて深層学習について理解することを目的とする．この資料は本実験に用いられるデータセットや課題、この実験のために作成された数値計算ライブラリ numj、深層学習フレームワーク dnn4j の説明を行う．

本実験では numj という数値計算ライブラリを用いる．これは Python の数値計算ライブラリである Numpy と似た API を提供している．このライブラリの JavaDoc は https://getumen.github.io/NumJ/doc/ にある．また、サンプルコードは https://github.com/kinoko/numj/blob/master/SampleCode.md にある．本課題に取り組む前に NumJ のサンプルコードを用いてテンソルの計算に慣れること．NumJ はこの実験課題のために作成されたものであるが、NumJ の API は NumPy の API と殆どのメソッドについて互換があるので、Python を用いたこの後の課題で NumPy を学習するための助けになる．dnn4j は https://github.com/kinoko/deep_learning_code にある．

2 データセットの説明

この実験では 0 から 9 までの手書き数字画像とその正解ラベルからなる Mnist データセットを用いる．手書き数字はグレイスケールの 28×28 ピクセルに加工してある．つまり、画像のチャンネル数は 1 であり、高さと幅は 28 である．Mnist データには訓練用データとして 60,000 サンプル、テスト用データとして 10,000 サンプルが用意されている．

この課題では手書き数字が与えられた時、0 から 9 の数字のどれを表すかを予測する深層学習モデルを実装する．

Mnist の手書き数字の例

3 実験で作成する深層学習モデルの概要

深層学習モデルではデータをモデルに入力し、各層を出力に向けて伝播させる順伝播 (forward propagation) により予測を行う．学習はモデルの勾配を計算し、その勾配を用いて確率的勾配降下法のような最適化法によりパラメータを更新して行う．

深層学習において勾配を効率的に計算することがより早くモデルを学習するために重要である．勾配の計算は誤差を出力から入力に向けて逆伝播 (backward propagation) させる誤差逆伝播法により行う．これにより複雑なモデルの微分を単純な実装で効率的に計算できる．この実験では一部の層を実装することで誤差逆伝播法につい

て理解を深める．

3.1 ネットワーク構造

この実験では図 1 のようなネットワークアーキテクチャを持つ深層学習モデルを作成する．

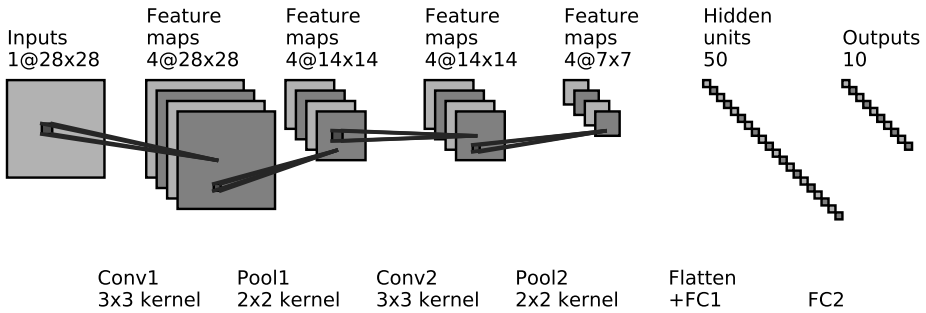

図 1　本実験で作成するアーキテクチャ

3.2 予測

深層学習モデルでは以下のように順番に入力を伝播していくこと (順伝播) で予測を行う．

Input　1 チャンネルの 28×28 ピクセルの画像を入力として受け取る．

Conv1　1. 4 つの 3×3 のフィルターを用いて入力画像に対して畳み込みを行う．このとき、ストライド幅、パディング幅を 1 としたため、特徴マップ (feature map) は 4 チャンネルの 28×28 の画像となる．

2. ReLU を用いて畳み込み層の出力を非線形に変換する．

Pooling1　2×2 のフィルターを用いて Max-pooling を行う．このとき、ストライド幅を 2、パディング幅を 0 としたため、特徴マップは 4 チャンネルの 14×14 の画像になる．

Conv1　1. 4 つの 3×3 のフィルターを用いて入力画像に対して畳み込みを行う．このとき、ストライド幅、パディング幅を 1 としたため、特徴マップ (feature map) は 4 チャンネルの 14×14 の画像となる．

2. ReLU を用いて畳み込み層の出力を非線形に変換する．

Pool2　2×2 のフィルターを用いて Max-pooling を行う．このとき、ストライド幅を 2、パディング幅を 0 としたため、特徴マップは 4 チャンネルの 7×7 の画像になる．

Flatten　畳み込み層やプーリング層では画像はチャンネル×高さ×幅のテンソルとして表されているが、これをベクトルに変換する (Flatten)

FC1　1. Unit Size が 50 の全結合層でアフィン変換をおこなう．

2. ReLU を用いて全結合層の出力を非線形に変換する．

FC2 1. Unit Size が 10 の全結合層でアフィン変換をおこなう．

2. Softmax 関数を用いて、ReLU 層の出力を各数字ラベルの確率に変換する．

3.3 課題 1

配布されたコードは一般的な Java アプリケーション開発におけるディレクトリ構造と同様になっている．

- /src/main/java アプリケーションについてのコードがある．実験においては基本的にこのディレクトリに含まれるファイルを書き換える．App.java がプログラムのエントリーポイントとなるファイルである．
- /src/main/resources Java コード以外のアプリケーションのためのファイルがある．配布されたコードには Mnist のバイナリデータが入っている
- /src/test/java 単体テスト用のコードが入っている．

Utils.computeOutputSize(int inputSize, int filterSize, int stride, int padding) を実装しなさい．computeOutputSize は例えば、出力の高さ=computeOutputSize(入力の高さ, フィルターの高さ, stride, padding) を計算する．実装の正しさは単体テストにより確認できる．テストは UtilsTest である．実装できたら、computeOutputSize のテストに@Test というアノテーションをつけて

```
./gradlew test
```

というコマンドを src や gradlew などのディレクトリやファイルがあるディレクトリで実行せよ．実験に必要な依存ファイルはこのコマンドを実行時にダウンロードされる．テストに成功した場合、SUCCESS と表示され、失敗した場合は詳細が html ファイルとして提供される．

3.4 学習

予測は以上のように行うことができた．深層学習モデルの学習を行うためにはモデルの出力結果と正解ラベルを見比べて、どれだけ間違っているかを損失関数により定量的に評価する．今回の実験では損失関数として交差エントロピー損失を用いる．

深層学習モデルの学習は確率的勾配降下法 (Stochastic Gradient Descent, SGD) などの勾配情報を用いてパラメータを更新するオプティマイザーにより行う．そのためには、パラメータを持つ畳み込み層や全結合層の重みパラメータやバイアス項についての勾配が必要になる．これらの勾配を求めるためには損失関数で求められた正解ラベルとの誤差を順伝播と逆方向に伝播させること (逆伝播) で効率的に計算することができる．

モデルの損失（クロスエントロピー損失, CE）はそれぞれの層の出力が次の層の入力となる関数の合成関数で表すことができる．

$$\mathrm{CrossEntropy} \circ \mathrm{Relu} \circ \mathrm{FC} \tag{1}$$

今、次のような入力 $\mathbf{X}$ について、FC と Relu の 2 層からなるモデルの出力を考える．図**??**のようなモデルにおいて、合成関数の各層が持つパラメータにおける微分を効率的に計算することを考える．

$$\mathbf{U} = \mathrm{FC}(\mathbf{X}) \tag{2}$$

$$\mathbf{Z} = \mathrm{ReLU}(\mathbf{U}) \tag{3}$$

パラメータを更新するためには出力から各層までの微分を計算する必要がある．この計算は以下のようにチェインルールを用いて計算することができる．

$$\frac{\partial \mathbf{Y}}{\partial \mathbf{U}} = \frac{\partial \mathbf{Z}}{\partial \mathbf{U}} \cdot \frac{\partial \mathbf{Y}}{\partial \mathbf{Z}} \tag{4}$$

$$\frac{\partial \mathbf{Y}}{\partial \mathbf{X}} = \frac{\partial \mathbf{U}}{\partial \mathbf{X}} \cdot \frac{\partial \mathbf{Y}}{\partial \mathbf{U}} \tag{5}$$

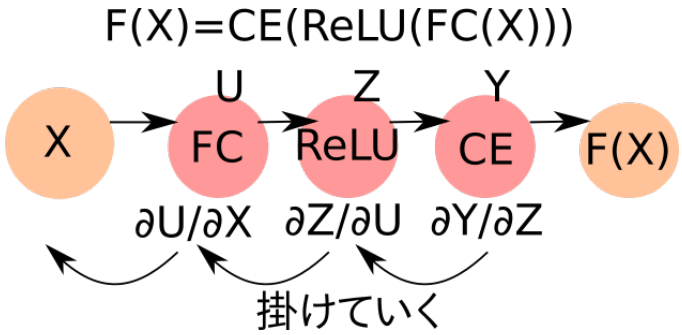

図 2　合成関数は順伝播、逆伝播について各層に分けて計算できる

$\frac{\partial Y}{\partial Z}$ の計算は出力側の層からチェインルールを用いて同様に計算することで得られる．このようにチェインルールを用いて、各層の出力を入力で微分して、出力層から伝播してくる微分に掛けあわせていくことで全ての層の勾配を求めることができる．

本実験では順伝播と逆伝播ともに一本道の単純な構造である．深層学習のライブラリの多くは本実験の実装モデルを拡張した計算グラフというものを用いることで、再帰構造や枝分かれを持つような複雑なネットワークを構築することができる．しかし、本実験で実装する各層自体は、計算グラフを用いた場合でも共通の実装である．

この実験では、活性化関数であるシグモイド関数、全結合層、畳み込み層の順方向と逆方向の伝播を実装する．

4　実験コードの説明と各レイヤーの実装課題

ここでは、深層学習モデルの各層を説明し、実装してもらう．

深層学習モデルの各レイヤーは jp.ac.tsukuba.cs.mdl.dnn4j.layers というパッケージにある．このパッケージ内のクラス図は図 3 である．このパッケージでは最終層以外の層は Layer というインターフェイスを実装している．また、最終層は LastLayer というインターフェイスを実装している．実験ではシグモイド関数、全結合層、畳み込み層の forward、backward を実装するが、以下では全ての層についてその役割を説明する．

各レイヤーの forward は図 1 において右向きの順伝播を行う．forward では左のレイヤーから順伝播してきた特徴量に対して何らかの変換を行い右のレイヤーの forward に伝播させる．

各レイヤーの backward は図 1 において左向きの逆伝播を行う．backward では右のレイヤーから逆伝播してきた誤差に対して forward で行った変換を入力で微分した値をかけて左のレイヤーの backward に伝播させる．加えて、backward ではもしレイヤーが重みとバイアスを持っていたらそれぞれのパラメータの勾配を更新する．

4.1　全結合層

jp.ac.tsukuba.cs.mdl.dnn4j.layers.FullyConnect は全結合層である．

forward(input) の入力の $\mathbf{X}$ はミニバッチサイズ×特徴量数の行列である (x.shape()==[miniBatch, feature])．順伝播で入力に対して以下のアフィン変換を行っている．

$$\mathbf{U} = \mathbf{X}\mathbf{W} + \mathbf{B} \tag{6}$$

ここで、$\mathbf{W}$ は入力サイズ×ユニットサイズの行列であり、$\mathbf{B}$ は大きさがミニバッチサイズ×ユニットサイズのである．

backward(dout) の入力である dout はミニバッチサイズ×ユニット数の行列である (dout.shape()==[miniBatch, unitNum])．逆伝播では図 1 で言うところの右の層から誤差 dout が与えられるので、これを用いて重み行列とバイアスの勾配を計算する．また、dout と全結合層のパラメータを用いて、図 1 で言うところの左の層に誤差を伝播させる．それぞれの勾配の計算と誤差の計算は式 6 を対応するパラメータで以下のように微分することで得ら

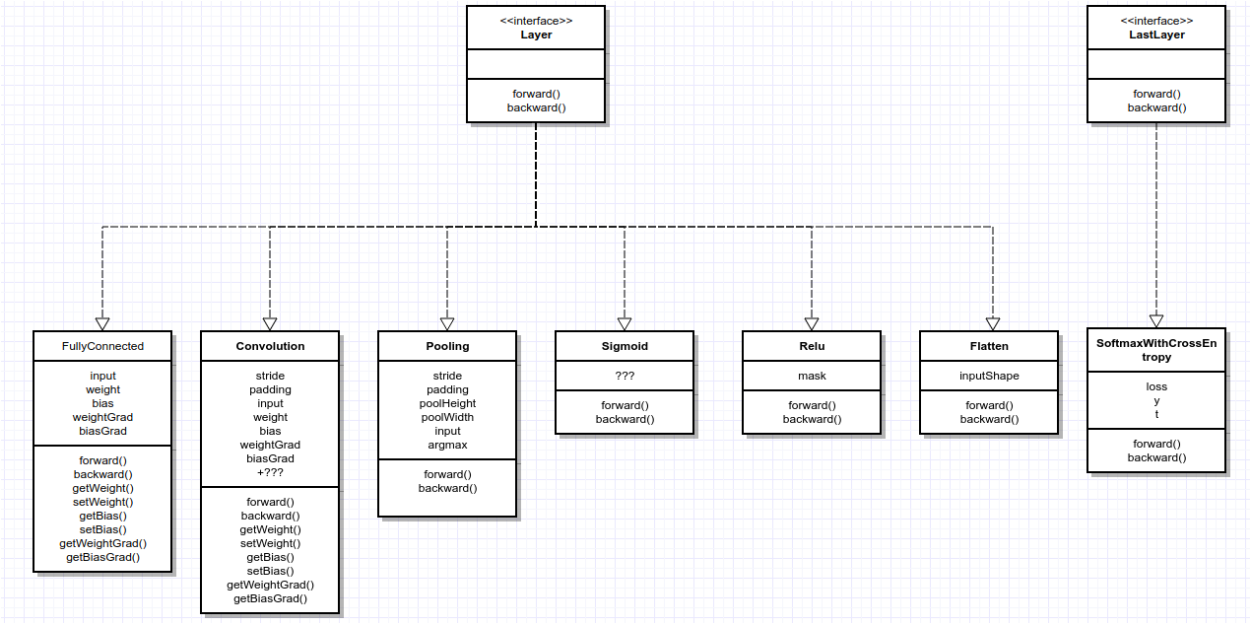

図 3　深層学習モデルのレイヤーのクラス図

れる．

$$\frac{\partial \mathbf{Y}}{\partial \mathbf{W}} = \mathbf{X}^T(\mathbf{DOUT}) \tag{7}$$

$$\frac{\partial \mathbf{Y}}{\partial \mathbf{B}} = \mathbf{1}^T(\mathbf{DOUT})//shape = [1, unitNum] \tag{8}$$

$$\frac{\partial \mathbf{Y}}{\partial \mathbf{X}} = (\mathbf{DOUT})\mathbf{W}^T \tag{9}$$

4.2　Pooling 層

jp.ac.tsukuba.cs.mdl.dnn4j.layers.Pooling 層はプーリング層である．

forward の入力はミニバッチサイズ×チャンネル数×高さ×幅のテンソルである (input.shape()==[miniBatch, channel, height, width])．Pooling の操作は図 4 の Pooling の部分に示す．順伝播では入力に対して Utils.im2col 関数を用いて画像テンソルを (ミニバッチサイズ×チャンネル) と (高さ×幅) の行列に変換する．そして、高さ×幅のサイズのプールから最大値を出力するという操作を行っている．さらに、Pooling の出力をテンソルにするという操作を行う．テンソルのサイズはミニバッチサイズ×チャンネル×高さ×幅であり、ミニバッチサイズは不変である．チャンネル数はモデルを構築するときに与えられる変数である．高さと幅は以前に実装した Utils.computeOutputSize により計算された入力サイズ、プールのサイズ、ストライド、パディングに依存する値である．

プーリングの forward を図の場合を例に説明する．図の Pooling の 3 × 3 の行列は 3 × 3 のプールを表している．この中で、最大値を取るのは (2, 2) の 3 である．forward ではこの 3 をプールの出力とする．

```
@Override
public NdArray forward(NdArray input) {
    // 入力を記憶しておく．
    this.input = input;
    int[] inputShape = input.shape();

    // 入力サイズ、プールのサイズ、ストライド、パディングから出力の画像サイズを計算する．
    int outHeight = Utils.computeOutputSize(inputShape[2], poolHeight, stride, padding);
```

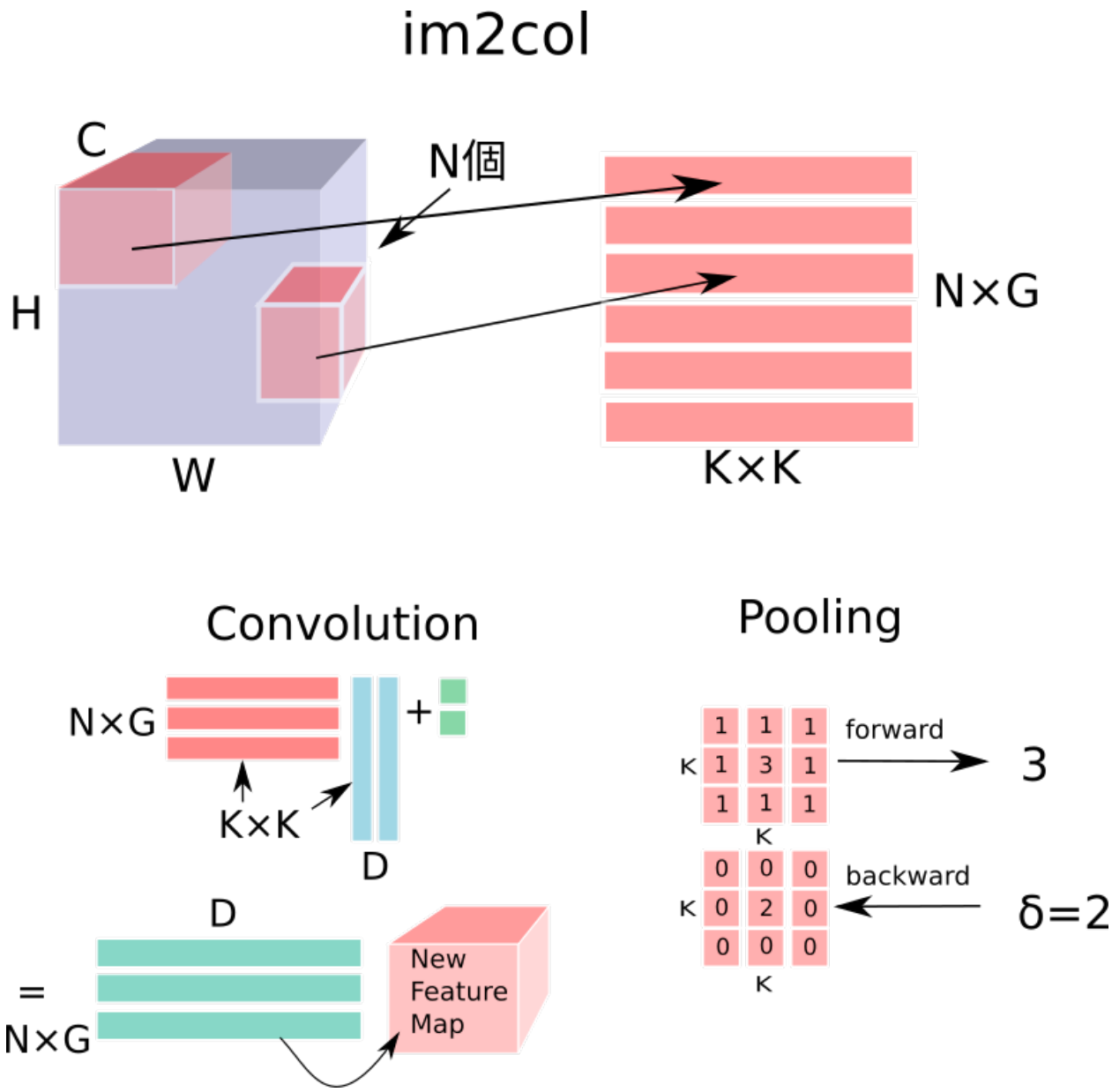

図 4　im2col の変換を可視化したもの．特徴マップのテンソルをベクトルに変換する部分に当たる．

```
int outWeight = Utils.computeOutputSize(inputShape[3], poolWidth, stride, padding);

// 各サンプルごとに画像のテンソルをベクトルに直す．
NdArray col = Utils.im2col(this.input, poolHeight, poolWidth, stride, padding);

// 画像ベクトルを（ミニバッチサイズ×チャンネル）と（プールの高さ×プールの幅）にリシェイプする．

col = col.reshape(col.size() / poolHeight / poolWidth, poolHeight * poolWidth);

// ここを実装．
// col の高さ×幅の部分の最大値を out に、そのインデックスを argmax に出力する．

// テンソルに直す．
out = out.reshape(inputShape[0], outHeight, outWeight, inputShape[1]).transpose(0, 3, 1, 2);
```

```
    return out;
}
```

im2col の詳細について説明する．im2col は図 4 のように特徴マップテンソルを特徴マップ行列に変換する．H は画像の高さ、W は画像の幅、C はチャンネル数、K はフィルターのサイズ、G はフィルターを特徴マップに適用した時のブロック数、D は出力チャンネル数×出力高さ×出力幅を表している．H × W × C のテンソルから K × K × 1 の窓を stride 幅だけ上下左右に動かして切り出して、特徴マップから特徴の行列を作る．特徴マップから特徴行列を作る際、特徴マップの端にあるものは端にないものに比べて読み取られる回数が少ないという問題が生じる．そこで、畳み込み層では特徴マップを全方向に padding 幅だけ 0 埋めして水増しし、全ての特徴マップの読み出し回数が同じになるようにする．プーリング層では一般にプールの大きさ K はストライドと同じ大きさにするため、全ての特徴マップの読み出し回数が 1 回となるので、パディングは用いられないことが多い．

im2col の定義は Utils.im2col(NdArray inputData, int kernelHeight, int kernelWidth, int stride, int padding) となっている．また、col2im の定義は Utils.col2im(NdArray col, int[] inputShape, int kernelHeight, int kernelWidth, int stride, int padding) となっている．この実装はすでに与えられているので、実験では利用するだけで良い．

backward の dout はミニバッチサイズ×チャンネル化×高さ×幅の行列である (dout.shape()==[miniBatch, channel, outHeight, outWidth])．逆伝播ではこの層の後ろの層から誤差 dout が与えられるので、最大値をとったフィルターの箇所に dout を代入し、その他のフィルターの箇所は 0 とする．これを Pooling 層の誤差として前の層に伝える．

Pooling の backward の実装では forward の変更を逆向きに適用している．つまり、forward では im2col を適用し、フィルターのプーリングを行い、transpose の変換を行う．backward では transpose によりデータを変換し、フィルターのプーリングの逆変換を行い、col2im を適用する．

具体的な操作を図 4 で見ていく．後ろの層から誤差 (出力側から伝播した微分) である $\delta = 2$ が与えられる．backward では、forward で最大値をとった (2, 2) に $\delta = 2$ を代入して、ほかは 0 で埋める．それを backward の出力とする．

```
@Override
public NdArray backward(NdArray dout) {
dout = dout.transpose(0, 2, 3, 1);
  int poolSize = poolHeight * poolWidth;
  NdArray dmax = NumJ.zeros(dout.size(), poolSize);

  for (int i = 0; i < argmax.size(); i++) {
      dmax.put(new int[]{i, (int) argmax.get(i)}, dout.get(i));
  }

  dmax = dmax.reshape(Ints.concat(dout.shape(), new int[]{poolSize}));

  NdArray dcol = dmax.reshape(
          dmax.shape()[0] * dmax.shape()[1] * dmax.shape()[2],
          dmax.size() / dmax.shape()[0] / dmax.shape()[1] / dmax.shape()[2]
  );
  return Utils.col2im(dcol, input.shape(), poolHeight, poolWidth, stride, padding);
}
```

4.3 畳み込み層

jp.ac.tsukuba.cs.mdl.dnn4j.layers.Convolution 層は畳み込みを行う．

forward の入力はミニバッチサイズ×チャンネル数×高さ×幅のテンソルである (input.shape()==[miniBatch, channel, height, width])．順伝播では入力に対して Utils.im2col(image to column の略) という関数を用いて画像をベクトル化し、そのベクトルに対してアフィン変換を行っている．さらに、出力としてベクトル化された特徴マップを元の画像を表すテンソルに直すために transpose により関する．

$$(\mathbf{COL}) = \mathrm{im2col}(\mathbf{x}) \tag{10}$$
$$\hat{\mathbf{U}} = (\mathbf{COL})\mathbf{W}^T + \mathbf{B} \tag{11}$$
$$\mathbf{U} = \hat{\mathbf{U}}.tranpose() \tag{12}$$

backward の dout はミニバッチサイズ×出力チャンネル×出力高さ×出力幅の行列である (dout.shape()==[miniBatch, channel, outHeight, outWidth])．逆伝播ではこの層の後ろの層から誤差 dout が与えられるので、dout を用いて、重み行列とバイアスの勾配を計算する．また、dout と全結合層のパラメータを用いて、出力する新たな dout を計算して出力する．それぞれの勾配の計算と誤差の計算は全結合層同様、対応するパラメータで微分することで得られる．

$$\frac{\partial \mathbf{Y}}{\partial \mathbf{W}} = (\mathbf{DOUT})^T(\mathbf{COL}) \tag{13}$$
$$\frac{\partial \mathbf{Y}}{\partial \mathbf{B}} = \mathbf{1}^T(\mathbf{DOUT})//shape = [1, unitNum] \tag{14}$$
$$\frac{\partial \mathbf{Y}}{\partial \mathbf{X}} = (\mathbf{DOUT})\mathbf{W} \tag{15}$$

4.4 Sigmoid 層

jp.ac.tsukuba.cs.mdl.dnn4j.layers.Sigmoid 層は順伝播では入力に対して要素ごとにシグモイド関数を適用する．

$$\mathbf{Z} = \sigma(U) \tag{16}$$
$$\sigma(x) = \frac{1}{1+\exp(-x)} \tag{17}$$

逆伝播ではこの層の後ろの層から誤差 dout が与えられるので、入力に関して要素ごとに微分して伝播させるべき誤差が得られる．

$$\frac{\partial \mathbf{Y}}{\partial \mathbf{U}} = \frac{\partial \mathbf{Z}}{\partial \mathbf{U}} \cdot (\mathbf{DOUT}) \tag{18}$$

4.5 ReLU 層

jp.ac.tsukuba.cs.mdl.dnn4j.layers.Relu 層は順伝播では入力に対して要素ごとに ReLU(Rectified Linear Unit) 関数を適用する．

$$\mathbf{Z} = ReLU(\mathbf{U}) \tag{19}$$
$$ReLU(x) = \max\{x, 0\} \tag{20}$$

ReLU 関数は微分不可能な関数であるため、誤差逆伝播のために劣微分を行う．劣微分は本実験の範囲を超えるため解説しない．逆伝播ではこの層の後ろの層から誤差 dout が与えられるので、入力に関して要素ごとに劣微分して伝播させるべき誤差が得られる．

4.6 Flatten 層

jp.ac.tsukuba.cs.mdl.dnn4j.layers.Flatten 層は順伝播で特徴マップを特徴ベクトルに変換する．逆伝播では特徴ベクトルを特徴マップに変換する．数学的な意味はなく、実装上の都合で (Convolution, Pooling) から (FullyConnect) 層に特徴量を渡すために用いられる．

4.7 課題 2

課題では今までに説明した Layer の内、Sigmoid 層、全結合層、畳み込み層、プーリング層を実装する．実装の正しさは単体テストにより確認できる．それぞれのテストは SigmoidTest, FullyConnectTest, ConvolutionTest である．それぞれのレイヤーを実装できたら、対応するテストに@Test というアノテーションをつけて

```
./gradlew test
```

というコマンドを src や gradlew などのファイルがあるディレクトリで実行せよ．テストに成功した場合、SUCCESS と表示され、失敗した場合は詳細が html ファイルとして提供される．

1. Sigmoid 層を実装せよ．
   （a）式 16 が Sigmoid の forward の出力となるように実装せよ．
   （b）式 18 が Sigmoid の backward の出力となるように実装せよ．
2. FullyConnected を実装せよ．
   （a）式 6 が FullyConnect の forward の出力となるように実装せよ．
   （b）式 7 を計算し、FullyConnect の重みの勾配を表すフィールドである weightGrad に代入せよ．
   （c）式 8 を計算し、FullyConnect のバイアスの勾配を表すフィールドである biasGrad に代入せよ．
   （d）式 9 が FullyConnect の backward の出力となるように実装せよ．
3. Convolution を実装せよ．
   （a）式 11 の計算権結果を Convolution の forward の out に代入せよ．
   （b）式 13 を計算し、Convolution の重みの勾配を表すフィールドである weightGrad に代入せよ．
   （c）式 14 を計算し、Convolution のバイアスの勾配を表すフィールドである biasGrad に代入せよ．
   （d）式 15 の計算結果を Convolution の backward の dcol に代入せよ．
4. Pooling を実装せよ．
   （a）Pooling 層の forward を実装せよ．
   （b）Pooling 層の backward を実装せよ．(実装はテキストにある．)

5 深層学習モデルの構築

ここでは、深層学習モデルを実行するためのコードを解説する．また、いくつかのモデルを作成して、性能評価を行ってもらう．

5.1 App.java の説明

本実験コードをアプリケーションとして動かすためには

```
./gradlew run
```

というコマンドを実行する．コマンドを実行することで、App.java の main 関数が実行される．

App.java の main 関数について上から見ていく．

5.1.1 データの準備

下のコードは Mnist データを読み取り、入力データの Shape や訓練データ、テストデータを用意するためのコードである．本実験ではこれを変更する必要はない．

```
Dataset dataset = new MnistDataset();
int[] inputShape = new int[]{
        dataset.getChannelSize(), dataset.getHeight(), dataset.getWidth()
};
NdArray xTrain = dataset.readTrainFeatures().reshape(
        dataset.getTrainSize(),
        dataset.getChannelSize(),
        dataset.getHeight(),
        dataset.getWidth()
);
NdArray tTrain = dataset.readTrainLabels();
NdArray xTest = dataset.readTestFeatures().reshape(
        dataset.getTestSize(),
        dataset.getChannelSize(),
        dataset.getHeight(),
        dataset.getWidth()
);
NdArray tTest = dataset.readTestLabels();
```

5.1.2 ネットワークアーキテクチャの設計

```
List<Map<String, Integer>> netArgList = constructNetArch();
```

constructNetArch() という関数でネットワークの設計した結果を得ている．constructNetArch() において、各レイヤーに関する情報を Map<String, Integer>にいれて、List<Map<String, Integer>>に順伝播の方向に加えていくことでネットワークを設計する．それぞれのレイヤーを構築するために必要な情報はそれぞれ異なっている．

- FullyConnect(UNIT_NUM)
- Convolution(FILTER_NUM, FILTER_SIZE, STRIDE, PADDING)
- Pooling(FILTER_SIZE, STRIDE, PADDING)
- Relu()
- Sigmoid()
- Flatten()

下に示したソースコードが網羅的な例となっている．これは Convolution, ReLU, Pooling, Flatten, FullyConect, Sigmoid, FullyConect, (Softmax, CrossEntropy) という構造のネットワークを設計している．MapBuilder<K, V>は HashMap<K, V>を作成するクラスである．

```
private static List<Map<String, Integer>> constructNetArch() {
      List<Map<String, Integer>> netArgList = new ArrayList<>();

      netArgList.add(
              new MapBuilder<String, Integer>()
```

```
                .put(NetArgType.LAYER_TYPE, LayerType.CONVOLUTION)
                .put(NetArgType.FILTER_NUM, 4)
                .put(NetArgType.FILTER_SIZE, 3)
                .put(NetArgType.STRIDE, 1)
                .put(NetArgType.PADDING, 1)
                .build()
);

netArgList.add(
        new MapBuilder<String, Integer>()
                .put(NetArgType.LAYER_TYPE, LayerType.RELU)
                .build()
);

netArgList.add(
        new MapBuilder<String, Integer>()
                .put(NetArgType.LAYER_TYPE, LayerType.POOLING)
                .put(NetArgType.FILTER_SIZE, 2)
                .put(NetArgType.STRIDE, 2)
                .put(NetArgType.PADDING, 0)
                .build()
);

netArgList.add(
        new MapBuilder<String, Integer>()
                .put(NetArgType.LAYER_TYPE, LayerType.FLATTEN)
                .build()
);

netArgList.add(
        new MapBuilder<String, Integer>()
                .put(NetArgType.LAYER_TYPE, LayerType.FULLY_CONNECTED)
                .put(NetArgType.UNIT_NUM, 50)
                .build()
);
netArgList.add(
        new MapBuilder<String, Integer>()
                .put(NetArgType.LAYER_TYPE, LayerType.SIGMOID)
                .build()
);
netArgList.add(
        new MapBuilder<String, Integer>()
                .put(NetArgType.LAYER_TYPE, LayerType.FULLY_CONNECTED)
                .put(NetArgType.UNIT_NUM, 50)
                .build()
```

```
        );
        return netArgList;
    }
```

5.1.3 最適化アルゴリズムのパラメータの設定

```
Map<String, Double> optimizerParams = Maps.newHashMap();
```

最適化パラメータを指定している．このコードでは単にからのマップを渡しているので、実際に用いられる最適化パラメータは jp.ac.tsukuba.cs.mdl.dnn4j.optimizers に定義されているデフォルトの値が用いられる．これらのデフォルトの値は殆どのディープラーニングフレームワークにおいて、同じ値に指定してあり、変更しないのが通例となっている．

com.google.common.collect.Maps.newHashMap() は new HashMap<>() と同じ意味である．Java で頻繁に用いられる Guava というライブラリが使われているが、深い意味はない．

5.1.4 ネットワークの構成

NetArgList を用いてネットワークを構築する．本実験では各レイヤーの重みパラメータにリッジ正則化を掛ける weight decay lambda 以外を変更する必要はない．

```
Net net = new NeuralNet(
        inputShape,
        netArgList,
        new SigmoidWithLoss(),
        0.01 // weight decay lambda
);
```

第一引数に入力データの形（チャンネル、高さ、幅）を与える．第二引数にネットワークの設計を与える．第三引数に最終層のインスタンスを与える．第四引数に weight decay のパラメータを与える．

5.1.5 訓練を行う

Trainer は構築された深層学習モデルのネットワークとデータを受け取り、訓練する．

学習では 1 エポック (epoch) に学習データをミニバッチサイズごとに切り分ける．そして、1 イテレーションごとに 1 つのミニバッチに対して順伝播、逆伝播を行い、勾配を計算し、パラメータを更新する．実験では epoch 数、minibatch size、optimizer を変更する．また、大きなネットワークを評価するとき、メモリ不足になることがある．その時は、evaluateBatchSize を小さくする．

verbase は true であるとき、各イテレーションの損失の値を出力する．

```
Trainer trainer = new TrainerImpl(
        net, // network
        xTrain, // input train data
        tTrain, // target train data
        xTest, // input test data
        tTest, // target test data
        5, // epoch num
        100, // mini batch size
        OptimizerType.SGD, // optimizer
        optimizerParams, // optimizer parameter
        1000, // evaluate batch size
```

```
        true // verbose
);
trainer.train();
```

Trainer のインターフェイスは以下のようになっている．訓練次精度やテスト時精度、損失関数の値はそれぞれの getter から得られる．

```
public interface Trainer {
    public List<Double> getTrainLossList();

    public List<Double> getTrainAccList();

    public List<Double> getTestAccList();

    public void train();
}
```

5.2 課題 3

1. FullyConnect(200), Sigmoid, FullyConect(100), Sigmoid, FullyConnect(10), (Softmax, CrossEntropy) というネットワークを設計せよ．
   損失関数の値を Train.getTrainLossList() から取得し、csv ファイルに出力せよ．
   また、各エポックごとの Accuracy を getTrainAccList(), getTestAccList() から取得し、csv ファイルに出力せよ．
   各々の csv ファイルを図示せよ．
2. 1 のネットワークの活性化関数を ReLU に置き換えて、精度を比較せよ．
3. 図 1 のネットワークを設計し、精度を比較せよ．