電気電子プログラミング及演習
総合課題レポート

2019 年 7 月 9 日 (火)

学生番号 xxxx-xx-xxxx れおまる

1. 学習/推論に用いるプログラムの紹介

学習プログラム Learning.c

```c
int main(void)
{
    float *train_x = NULL;
    unsigned char *train_y = NULL;
    int train_count = -1;

    float *test_x = NULL;
    unsigned char *test_y = NULL;
    int test_count = -1;

    int width = -1;
    int height = -1;

    load_mnist(&train_x, &train_y, &train_count,
               &test_x, &test_y, &test_count,
               &width, &height);

    // これ以降，３層NN の係数 A_784x10 および b_784x10 と，
    // 訓練データ train_x + 784*i (i=0,...,train_count-1), train_y[0]～train_y[train_count-1],
    // テストデータ test_x + 784*i (i=0,...,test_count-1), test_y[0]～test_y[test_count-1],
    // を使用することができる．

    srand(time(NULL));
    int c1,c2; //入力確認用
    int epoch = 10;
    int batch_size = 100;
    int opt_select = 0;
    int init_select = 0;

    float Loss_ave = 0;
    float Accuracy = 0;
    //float Loss_ave1 = 0;
    //float Accuracy1 = 0;
    float max_accuracy = 0;
```

はじめに nn.h の load_mnist を実行し、いくつかのパラメータを読み込む。
その後 rand 関数の seed を time(NULL)で指定し初期化の後に、試行を行うエポック数やバッチサイズなど main 関数内で使用しているいくつかの変数の初期化を行う。

```
float *y = malloc(sizeof(float)*10);

int *index = malloc(sizeof(int)*train_count);

float *A1 = malloc(sizeof(float)*784*50);
float *b1 = malloc(sizeof(float)*50);
float *A2 = malloc(sizeof(float)*50*100);
float *b2 = malloc(sizeof(float)*100);
float *A3 = malloc(sizeof(float)*100*10);
float *b3 = malloc(sizeof(float)*10);

float *dEdA1 = malloc(sizeof(float)*784*50);
float *dEdb1 = malloc(sizeof(float)*50);
float *dEdA2 = malloc(sizeof(float)*50*100);
float *dEdb2 = malloc(sizeof(float)*100);
float *dEdA3 = malloc(sizeof(float)*100*10);
float *dEdb3 = malloc(sizeof(float)*10);

float *dEdA1_ave = malloc(sizeof(float)*784*50);
float *dEdb1_ave = malloc(sizeof(float)*50);
float *dEdA2_ave = malloc(sizeof(float)*50*100);
float *dEdb2_ave = malloc(sizeof(float)*100);
float *dEdA3_ave = malloc(sizeof(float)*100*10);
float *dEdb3_ave = malloc(sizeof(float)*10);
```

```
//SGD
float learning_rate = 0.1;
//Momentum SGD
float *vA1 = malloc(sizeof(float)*784*50);
float *vb1 = malloc(sizeof(float)*50);
float *vA2 = malloc(sizeof(float)*50*100);
float *vb2 = malloc(sizeof(float)*100);
float *vA3 = malloc(sizeof(float)*100*10);
float *vb3 = malloc(sizeof(float)*10);
//AdaGrad
float *hA1 = malloc(sizeof(float)*784*50);
float *hb1 = malloc(sizeof(float)*50);
float *hA2 = malloc(sizeof(float)*50*100);
float *hb2 = malloc(sizeof(float)*100);
float *hA3 = malloc(sizeof(float)*100*10);
float *hb3 = malloc(sizeof(float)*10);
//Adam
float t = 1.0;

float *b_vA1 = malloc(sizeof(float)*784*50);
float *b_vb1 = malloc(sizeof(float)*50);
float *b_vA2 = malloc(sizeof(float)*50*100);
float *b_vb2 = malloc(sizeof(float)*100);
float *b_vA3 = malloc(sizeof(float)*100*10);
float *b_vb3 = malloc(sizeof(float)*10);

float *b_mA1 = malloc(sizeof(float)*784*50);
float *b_mb1 = malloc(sizeof(float)*50);
float *b_mA2 = malloc(sizeof(float)*50*100);
float *b_mb2 = malloc(sizeof(float)*100);
float *b_mA3 = malloc(sizeof(float)*100*10);
float *b_mb3 = malloc(sizeof(float)*10);
```

各層におけるパラメータ A,b やその勾配、ミニバッチにおける A,b の平均の動的メモリ確保を malloc 関数で行う。
同時に後述の optimizer で用いる配列について事前に全てメモリ確保を行ったのちに、このあとの選択によって必要なものだけを確保しておく。

```
do{
    printf("初期値を選択してください\n一様分布:1 ガウス分布(S.D.=0.01):2 Heの初期値:3 Xavierの初期値:4\nYour select:");
    c1 = scanf("%d",&init_select);
    if(c1 != 1){
        printf("Input Error!\n");
        scanf("%*s");
    }
}while(c1!=1||init_select<1||init_select>4);

do{
    printf("\nOptimizerを選択してください\nSGD:1 MomentumSGD:2 Adagrad:3 Adam:4\nYour select:");
    c2 = scanf("%d",&opt_select);
    if(c2 != 1){
        printf("Input Error!\n");
        scanf("%*s");
    }
}while(c2!=1||opt_select<1||opt_select>4);
printf("\n");
```

ここでユーザが初期化に用いる乱数分布と Optimizer について scanf 関数で選択することができる。c1,c2 はその選択を保存する変数であり、誤った入力がなされたときはエラーを表示して値を捨てることで正しい入力がなされるまで繰り返し入力を待つように工夫を行った。

```
初期値を選択してください
一様分布:1 ガウス分布(S.D.=0.01):2 Heの初期値:3 Xavierの初期値:4
Your select:3

Optimizerを選択してください
SGD:1 MomentumSGD:2 Adagrad:3 Adam:4
Your select:2
```

```
//Optimizerの選択
switch (opt_select)
{
case 1:{ //SGD
    free(vA1);free(vA2);free(vA3);free(vb1);free(vb2);free(vb3);
    free(hA1);free(hA2);free(hA3);free(hb1);free(hb2);free(hb3);
    free(b_vA1);free(b_vA2);free(b_vA3);free(b_vb1);free(b_vb2);free(b_vb3);
    free(b_mA1);free(b_mA2);free(b_mA3);free(b_mb1);free(b_mb2);free(b_mb3);
    break;
    }
case 2:{ //Momentum SGD
    init(784*50,0,vA1);
    init(50,0,vb1);
    init(50*100,0,vA2);
    init(100,0,vb2);
    init(100*10,0,vA3);
    init(10,0,vb3);
    free(hA1);free(hA2);free(hA3);free(hb1);free(hb2);free(hb3);
    free(b_vA1);free(b_vA2);free(b_vA3);free(b_vb1);free(b_vb2);free(b_vb3);
    free(b_mA1);free(b_mA2);free(b_mA3);free(b_mb1);free(b_mb2);free(b_mb3);
    break;
    }
case 3:{ //Adagrad
    init(784*50,1e-7,hA1);
    init(50,1e-7,hb1);
    init(50*100,1e-7,hA2);
    init(100,1e-7,hb2);
    init(100*10,1e-7,hA3);
    init(10,1e-7,hb3);
    free(vA1);free(vA2);free(vA3);free(vb1);free(vb2);free(vb3);
    free(b_vA1);free(b_vA2);free(b_vA3);free(b_vb1);free(b_vb2);free(b_vb3);
    free(b_mA1);free(b_mA2);free(b_mA3);free(b_mb1);free(b_mb2);free(b_mb3);
    break;
    }
case 4:{ //Adam
    init(784*50,0,b_vA1);
    init(50,0,b_vb1);
    init(50*100,0,b_vA2);
    init(100,0,b_vb2);
    init(100*10,0,b_vA3);
    init(10,0,b_vb3);

    init(784*50,0,b_mA1);
    init(50,0,b_mb1);
    init(50*100,0,b_mA2);
    init(100,0,b_mb2);
    init(100*10,0,b_mA3);
    init(10,0,b_mb3);
    free(vA1);free(vA2);free(vA3);free(vb1);free(vb2);free(vb3);
    free(hA1);free(hA2);free(hA3);free(hb1);free(hb2);free(hb3);
    break;
    }
}
```

```
switch (init_select) //初期値の選択
{
case 1:{ //rand関数による一様分布
    rand_init(784*50,A1);
    rand_init(50,b1);
    rand_init(50*100,A2);
    rand_init(100,b2);
    rand_init(100*10,A3);
    rand_init(10,b3);
    break;
    }
case 2:{ //標準偏差0.01のガウス分布
    normal_init(784*50,A1);
    normal_init(50,b1);
    normal_init(50*100,A2);
    normal_init(100,b2);
    normal_init(100*10,A3);
    normal_init(10,b3);
    break;
    }
case 3:{ //Heの初期値
    he_init(784*50,A1);
    he_init(50,b1);
    he_init(50*100,A2);
    he_init(100,b2);
    he_init(100*10,A3);
    he_init(10,b3);
    break;
    }
case 4:{ //Xavierの初期値
    xavier_init(784*50,A1);
    xavier_init(50,b1);
    xavier_init(50*100,A2);
    xavier_init(100,b2);
    xavier_init(100*10,A3);
    xavier_init(10,b3);
    break;
    }
}
```

先ほどのユーザからの選択に応じて、その乱数分布に従った初期化を各パラメータに対してそれぞれ行う。Optimizer の選択に応じて事前に確保しておいた必要のないメモリ領域は開放する。

乱数分布それぞれに対する乱数生成関数を作成しており、その詳細は 2.で述べる。

```
for(int i=0;i<epoch;i++){
    for(int o=0;o<train_count;o++){
        index[o] = o;
    }
    shuffle(train_count,index);


    for(int j=0;j<train_count/batch_size;j++){
        init(784*50,0,dEdA1_ave);
        init(50,0,dEdb1_ave);
        init(50*100,0,dEdA2_ave);
        init(100,0,dEdb2_ave);
        init(100*10,0,dEdA3_ave);
        init(10,0,dEdb3_ave);

        for(int k=0;k<batch_size;k++){
            printf("\rEpoch%3d:[%3d/100%%]",i,((k + 1 + batch_size * j) * 100/ train_count));
            backward6(A1,A2,A3,b1,b2,b3,train_x + 784*index[batch_size*j+k],
                train_y[index[batch_size*j+k]],y,dEdA1,dEdA2,dEdA3,dEdb1,dEdb2,dEdb3);
            add(784*50,dEdA1,dEdA1_ave);
            add(50,dEdb1,dEdb1_ave);
            add(50*100,dEdA2,dEdA2_ave);
            add(100,dEdb2,dEdb2_ave);
            add(100*10,dEdA3,dEdA3_ave);
            add(10,dEdb3,dEdb3_ave);
        }
        scale(784*50,1/((float)batch_size),dEdA1_ave);
        scale(50,1/((float)batch_size),dEdb1_ave);
        scale(50*100,1/((float)batch_size),dEdA2_ave);
        scale(100,1/((float)batch_size),dEdb2_ave);
        scale(100*10,1/((float)batch_size),dEdA3_ave);
        scale(10,1/((float)batch_size),dEdb3_ave);
```

エポック毎に配列 index を shuffle 関数によりランダムに並び替えたのちに、エポック毎にミニバッチ学習を train_count/batch_size(ここでは 600 回)行う。
ミニバッチ学習では各パラメータでの平均勾配配列(dEdA_ave,dEdb_ave)をまず 0 に初期化したあとに、配列 index から batch_size 個順番に取り出すことで、もれなくランダムに train_count 個の訓練データで学習を行うことができる。
batch_size 回 backward6 関数によって各層での dEdA,dEdb を計算し、dEdA_ave, dEdb_ave に加えることで最後に batch_size 回でその和を割れば平均勾配が得られる。

```
switch (opt_select)
{
case 1:{ //SGD
    scale(784*50,-learning_rate,dEdA1_ave);
    scale(50,-learning_rate,dEdb1_ave);
    scale(50*100,-learning_rate,dEdA2_ave);
    scale(100,-learning_rate,dEdb2_ave);
    scale(100*10,-learning_rate,dEdA3_ave);
    scale(10,-learning_rate,dEdb3_ave);
    add(784*50,dEdA1_ave,A1);
    add(50,dEdb1_ave,b1);
    add(50*100,dEdA2_ave,A2);
    add(100,dEdb2_ave,b2);
    add(100*10,dEdA3_ave,A3);
    add(10,dEdb3_ave,b3);
    break;
    }
case 2:{ //Momentum SGD
    momentum(784*50,vA1,dEdA1_ave,A1);
    momentum(50,vb1,dEdb1_ave,b1);
    momentum(50*100,vA2,dEdA2_ave,A2);
    momentum(100,vb2,dEdb2_ave,b2);
    momentum(100*10,vA3,dEdA3_ave,A3);
    momentum(10,vb3,dEdb3_ave,b3);
    break;
    }
case 3:{ //Adagrad
    AdaGrad(784*50,hA1,dEdA1_ave,A1);
    AdaGrad(50,hb1,dEdb1_ave,b1);
    AdaGrad(50*100,hA2,dEdA2_ave,A2);
    AdaGrad(100,hb2,dEdb2_ave,b2);
    AdaGrad(100*10,hA3,dEdA3_ave,A3);
    AdaGrad(10,hb3,dEdb3_ave,b3);
    break;
    }
case 4:{ //Adam
    Adam(784*50,b_mA1,b_vA1,dEdA1_ave,A1,t);
    Adam(50,b_mb1,b_vb1,dEdb1_ave,b1,t);
    Adam(50*100,b_mA2,b_vA2,dEdA2_ave,A2,t);
    Adam(100,b_mb2,b_vb2,dEdb2_ave,b2,t);
    Adam(100*10,b_mA3,b_vA3,dEdA3_ave,A3,t);
    Adam(10,b_mb3,b_vb3,dEdb3_ave,b3,t);
    t++;
    break;
    }
}
```

求めた平均勾配に対して選択した optimizer を用いて損失関数が減るようにパラメータ A,b を更新し、正解率の向上を図る。

各 optimizer でのパラメータ更新手法は 3.で詳しく述べる。

```
        float sum = 0;
        float loss_sum = 0;
        //float sum_train = 0;
        //float loss_train = 0;
        for(int m=0;m<test_count;m++){
            if(inference6(A1,A2,A3,b1,b2,b3,test_x+m*784,y) == test_y[m]){
                sum++;
            }
            loss_sum += cross_entropy_error(y,test_y[m]);
        }
        /* for(int m=0;m<train_count;m++){
            if(inference6(A1,A2,A3,b1,b2,b3,train_x+m*784,y) == train_y[m]){
                sum_train++;
            }
            loss_train += cross_entropy_error(y,train_y[m]);
        }
        */
        printf("\nLoss Average: %f (%+.3f)\n",loss_sum/test_count,loss_sum/test_count-Loss_ave);
        printf("Accuracy: %f (%+5.2f)\n",sum*100.0/test_count,sum*100.0/test_count-Accuracy);
        //printf("Loss Average(train): %f (%+.3f)\n",loss_train/train_count,loss_train/train_count-Loss_ave1);
        //printf("Accuracy(train): %f (%+.2f)\n",sum_train*100.0/train_count,sum_train*100.0/train_count-Accuracy1);
        if(sum*100.0/test_count > max_accuracy){
            max_accuracy = sum*100.0/test_count;
            copy(784*50,A1,best_A1);
            copy(50,b1,best_b1);
            copy(50*100,A2,best_A2);
            copy(100,b2,best_b2);
            copy(100*10,A3,best_A3);
            copy(10,b3,best_b3);
        }
        printf("Max Accuracy: %.2f%%\n\n",max_accuracy+0.001);
        Loss_ave = loss_sum/test_count;
        Accuracy = sum*100.0/test_count;
        //Loss_ave1 = loss_train/train_count;
        //Accuracy1 = sum_train*100.0/train_count;
    }

    save("fc1.dat",50,784,best_A1,best_b1);
    save("fc2.dat",100,50,best_A2,best_b2);
    save("fc3.dat",10,100,best_A3,best_b3);

    printf("finish!\n");

  return 0;
}
```

学習後 inference6 関数を用いて test_count 個のテストデータについての平均損失関数、正解率を表示する。学習中も進捗をアニメーション的に表示させることができる。
現時点での最良モデルのパラメータを best_A1～best_A3, best_b1～best_b3 に copy 関数を用いて一時的に配列に保存しておき、全てのエポックが終了したのちに save 関数を用いて最も正解率が高かったものを保存する。
Train データに対する正解率を求めるのに多少時間がかかるため、過学習を確認するときには用いて、提出時点ではコメントアウトしている。

```
0%       50%       100%
+---------+---------+
#######
Epoch 2: 43%
Loss Average: 0.123357 (-0.060)
Accuracy: 96.280000 (+1.98))
Max Accuracy: 96.28%
```

```
Epoch  1:[100/100%]
Loss Average: 0.263140 (-0.138)
Accuracy: 92.270000 (+4.06)
Max Accuracy: 92.27%

Epoch  2:[100/100%]
Loss Average: 0.215767 (-0.047)
Accuracy: 93.570000 (+1.30)
Max Accuracy: 93.57%

Epoch  3:[100/100%]
Loss Average: 0.176119 (-0.040)
Accuracy: 94.810000 (+1.24)
Max Accuracy: 94.81%
```

推論プログラム Inference.c

```c
int main(int argc, char const *argv[])
{

    float *train_x = NULL;
    unsigned char *train_y = NULL;
    int train_count = -1;

    float *test_x = NULL;
    unsigned char *test_y = NULL;
    int test_count = -1;

    int width = -1;
    int height = -1;

    load_mnist(&train_x, &train_y, &train_count,
               &test_x, &test_y, &test_count,
               &width, &height);

    //係数の初期化
    float *A1 = malloc(sizeof(float)*784*50);
    float *A2 = malloc(sizeof(float)*50*100);
    float *A3 = malloc(sizeof(float)*100*10);
    float *b1 = malloc(sizeof(float)*50);
    float *b2 = malloc(sizeof(float)*100);
    float *b3 = malloc(sizeof(float)*10);
    float *y = malloc(sizeof(float)*10);


    //文法確認
    if (argc < 5){
        printf("inference fc1.dat fc2.dat fc3.dat pic.bmpの形式で実行してください\n");
        return -1;
    }
    //行列データを読み込んで代入
    load(argv[1], 50, 784, A1, b1);
    load(argv[2], 100, 50, A2, b2);
    load(argv[3], 10, 100, A3, b3);

    //画像ファイルの読み込み
    float *x = load_mnist_bmp(argv[4]);

    int predict = 0;
    predict = inference6(A1, A2, A3, b1, b2, b3, x ,y);
    printf("  0    1    2    3    4    5    6    7    8    9  \n");
    for(int i=0;i<10;i++){
        printf("%3d%% ",(int)(y[i]*100.0));
    }
    printf("\nPicture's Number is %d.\n", predict);

  return 0;
}
```

inference.exe fc1.dat fc2.dat fc3.dat picture.bmp の形式で実行し、先頭３つで各パラメータの読み込みを行い、４つ目に推論を行うビットマップデータ（画像）を指定する。load 関数により各ファイルに保存されたパラメータ成分が行列として読み込まれ、inference6 で使用する A1,A2,A3,b1,b2,b3 となる。また load_mnist_bmp 関数により指定した画像データは 784 次元の入力データとなり、inference6 で用いる入力 x となる。出力層からの成分のうち最も高かったものが予測値であり、どの数字だと推論されたのか表示させる。

また、どのような確率で推論がなされたのか簡単に確認できるように softmax 層(出力層)からの各成分を百分率で表示させている。

```
(base) ReonoiMac-2:cpro reo_g$ ./inference fc1.dat fc2.dat fc3.dat train_00000.bmp
  0    1    2    3    4    5    6    7    8    9
  0%   0%   0%   2%   0%  97%   0%   0%   0%   0%
Picture's Number is 5.
```

2. 関数の説明

行列積演算(y=ax+b)

```
void fc(int m,int n,const float *x,const float *A,const float *b,float *y){
    for(int i=0;i<m;i++){
        y[i] = b[i];
        for(int j=0;j<n;j++){
            y[i] += A[n*i+j] * x[j];
        }
    }
}
```

関数 fc は行列積 y=ax+b を行う関数であり、int m に行列の行数 int n に行列の列数を指定する。そのため x に指定する const float *x は n 個の成分を持ち、const float *A に指定する行列 A はm×n 行列, const float *b に指定する行列 b はm×1 行列でなければならない。float *y には出力先の行列を指定し、m 個の成分を持つ。

$$y_k = \boldsymbol{a}_k \boldsymbol{x} + b_k \ (k = 1, \ldots, m) \qquad (1)$$

ReLU 関数

```
void relu(int n,const float *x,float *y){
    for(int i=0;i<n;i++){
        y[i] = (x[i]>0) ? x[i] : 0;
    }
}
```

関数 relu は活性化層における活性化関数 ReLU(ランプ関数)の計算を行う関数である。
要素数を int n に,要素数 n 個の入力を const float *x に出力を float *y を指定する。
下式からもわかるように、当然出力成分 yk はすべて非負であることが明らかである。

$$y_k = \begin{cases} x_k & (x_k > 0) \\ 0 & (otherwise) \end{cases} \qquad (2)$$

Softmax 関数

```
void softmax(int n,const float *x,float *y){
    float x_max = 0;
    float exp_sum = 0;
    for(int i=0;i<n;i++){
        if(x_max < x[i]){
            x_max = x[i];
        }
    }
    for(int i=0;i<n;i++){
        exp_sum += exp(x[i]-x_max);
    }
    for(int i=0;i<n;i++){
        y[i] = exp(x[i]-x_max) / exp_sum;
    }
}
```

関数 softmax は出力層のソフトマックス関数を計算する関数であり、入力の要素数を int n に指定する。n 次元の入力を float *x に出力を float *y に指定する。
式(3)と式(4)は式として等価であることから、オーバーフロー対策として要素の最大値を取得した上で exp の各要素に対して引く操作を行う。

$$y_k = \frac{\exp(x_k)}{\sum_i \exp(x_i)} \qquad (3) \qquad y_k = \frac{\exp(x_k - x_{\max})}{\sum_i \exp(x_i - x_{\max})} \qquad (4)$$

クロスエントロピー関数(損失関数計算)

```
float cross_entropy_error(const float *y,int t){
    return -1*log(y[t]+1e-7);
}
```

関数 cross_entropy_error の引数は softmax 関数による出力データを const float *y に int t に本来の正解データ(0〜9)を指定する。

$$E = -\sum_k t_k \log y_k$$

上式における tk は one-hot 表現であるので、t 番目のみ計算すればよい。

Softmax 層/誤差逆伝搬

```
void softmaxwithloss_bwd(int n,const float *y,unsigned char t,float *dEdx){
    for(int i=0;i<n;i++){
        dEdx[i] = (i==t)? y[i] - 1.0 : y[i];
    }
}
```

関数 softmaxwithloss_bwd は softmax 層における誤差逆伝搬計算を行う関数であり、int n に要素数,unsigned char t に正解のデータ(0〜9)を指定し、softmax 層での出力 const float *y を入力することで、損失関数計算に対する勾配 float *dEdx が求められる。

$$\frac{\partial E}{\partial x_k} = y_k - t_k$$

損失関数に対する入力の偏微分は上式のように表される。クロスエントロピー関数の場合と同様に tk は one-hot 表現であるので、t 番目のみ y から 1.0 を引けばよい。

ReLU 層/誤差逆伝搬

```
void relu_bwd(int n,const float *x,const float *dEdy,float *dEdx){
    for(int i=0;i<n;i++){
        dEdx[i] = (x[i]>0)? dEdy[i] : 0;
    }
}
```

関数 relu_bwd は ReLU 層における誤差逆伝播計算を行う関数であり、int n に要素数を入力する。その勾配は下式のように求められ、要素ごとに Relu 層での入力 const float *x が正であれば上流側の勾配 float *dEdy の値をそのまま float *dEdx とすることで計算される。

$$\frac{\partial E}{\partial x_k} = \frac{\partial E}{\partial y_k}\frac{\partial y_k}{\partial x_k} = \begin{cases} \frac{\partial E}{\partial y_k} & (x_k > 0) \\ 0 & (otherwise) \end{cases}$$

FC 層/誤差逆伝搬

```
//fc層の誤差逆伝搬
void fc_bwd(int m,int n,const float *x,const float *dEdy,const float *A,float *dEdA,float *dEdb,float *dEdx){
    for(int i=0;i<m;i++){
        dEdb[i] = dEdy[i];
        for(int j=0;j<n;j++){
            dEdA[n*i+j] = dEdy[i] * x[j];
        }
    }

    for(int i=0;i<n;i++){
        dEdx[i] = 0;
        for(int j=0;j<m;j++){
            dEdx[i] += A[n*j+i] * dEdy[j];
        }
    }
}
```

関数 fc_bwd は全結合層における損失関数に対するパラメータ A,b の勾配を求める誤差逆伝播計算を行う関数であり、下式のように求められる。

$$\frac{\partial E}{\partial \boldsymbol{a}_k} = \frac{\partial E}{\partial y_k}\frac{\partial y_k}{\partial \boldsymbol{a}_k} = \frac{\partial E}{\partial y_k}\boldsymbol{x}^\top$$

$$\frac{\partial E}{\partial b_k} = \frac{\partial E}{\partial y_k}\frac{\partial y_k}{\partial b_k} = \frac{\partial E}{\partial y_k}$$

$$\frac{\partial E}{\partial x_k} = \sum_j \frac{\partial E}{\partial y_j}\frac{\partial y_j}{\partial x_k} = \boldsymbol{a}_k^\top \frac{\partial E}{\partial \boldsymbol{y}}$$

int m に順伝播の際の出力次元,int n に入力次元を指定するために、fc 層での const float *A は m×n 行列でありその勾配である float *dEdA も m×n 行列である。const float *x に要素数 n の順伝播での入力 x を指定し、const float *dEdy には上流からの勾配を指定する。
これにより上式の計算によって損失関数に対するパラメータ A の勾配 float *dEdA, パラメータ b の勾配 float *dEdb, x の勾配 float *dEdx が計算される。

シャッフル関数

```
void shuffle(int n,int *x){
    int t = 0;
    for(int i=0;i<n;i++){
        t = genrand_int32() % n;
        int temp = x[i];
        x[i] = x[t];
        x[t] = temp;
    }
}
```

関数 shuffle の引数は index の要素数を int n に,シャッフルさせる int 型の数列 index を int *x に指定する。

推論関数 inference6

```
int inference6(const float *A1,const float *A2,const float *A3,
const float *b1,const float *b2,const float *b3,float *x,float *y){
    float *y1 = malloc(sizeof(float)*50);
    float *y2 = malloc(sizeof(float)*100);
    float temp = 0;
    int index;

    fc(50,784,x,A1,b1,y1);
    relu(50,y1,y1);
    fc(100,50,y1,A2,b2,y2);
    relu(100,y2,y2);
    fc(10,100,y2,A3,b3,y);
    softmax(10,y,y);

    for(int i=0;i<=9;i++){
        if(temp < y[i]){
            temp = y[i];
            index = i;
        }
    }
    free(y1);free(y2);
    return index;
}
```

関数 inference6 は推論を行う関数であり、学習を行ったパラメータ A,b が 6 層 NN が入力 float *x に対してどの数字である確率が高いか推論を行う。
const float *A1 には 1 段目の FC 層のパラメータ A(50×784)、const float *A2 には 2 段目の FC 層のパラメータ A(100×50)、const float *A3 には 3 段目の FC 層のパラメータ A(10×100)を指定する。同様に const float *b1 には 1 段目の FC 層のパラメータ b(50 行の列ベクトル)、const float *b2 には 2 段目の FC 層のパラメータ b(100 行の列ベクトル)、const float *b3 には 3 段目の FC 層のパラメータ b(10 行の列ベクトル)を指定する。
出力層による要素数１０の出力が float *y であり、各文字(0〜9)に対する x の推論確率を示す。最も確率が高いと判断した数字を戻り値として呼び出し元へ出力する。

6 層誤差逆伝搬計算関数　Backward6

```
void backward6(const float *A1,const float *A2,const float *A3,
const float *b1,const float *b2,const float *b3,const float *x,
unsigned char t,float *y,float *dEdA1,float *dEdA2,float *dEdA3,float *dEdb1,float *dEdb2,float *dEdb3){
    float *relu1_before = malloc(sizeof(float)*50);
    float *relu2_before = malloc(sizeof(float)*100);
    float *fc2_before = malloc(sizeof(float)*50);
    float *fc3_before = malloc(sizeof(float)*100);

    fc(50,784,x,A1,b1,relu1_before);
    relu(50,relu1_before,fc2_before);
    fc(100,50,fc2_before,A2,b2,relu2_before);
    relu(100,relu2_before,fc3_before);
    fc(10,100,fc3_before,A3,b3,y);
    softmax(10,y,y);

    float *dx3 = malloc(sizeof(float)*10);
    float *dx2 = malloc(sizeof(float)*100);
    float *dx1 = malloc(sizeof(float)*50);
    float *dx0 = malloc(sizeof(float)*784);

    softmaxwithloss_bwd(10,y,t,dx3);
    fc_bwd(10,100,fc3_before,dx3,A3,dEdA3,dEdb3,dx2);
    relu_bwd(100,relu2_before,dx2,dx2);
    fc_bwd(100,50,fc2_before,dx2,A2,dEdA2,dEdb2,dx1);
    relu_bwd(50,relu1_before,dx1,dx1);
    fc_bwd(50,784,x,dx1,A1,dEdA1,dEdb1,dx0);

    free(relu1_before);free(relu2_before);
    free(fc2_before);free(fc3_before);
    free(dx0);free(dx1);free(dx2);free(dx3);
}
```

関数 backward6 は 6 層での誤差逆伝播を行う関数であり、各層における損失関数に対するパラメータ A,b の勾配を求める。その計算結果の出力先は 1 段目の FC 層パラメータ A に対する勾配は float *dEdA1(50×784),b に対しては float *dEdb1(50)、２段目の FC 層パラメータ A に対する勾配は float *dEdA2(100×50),b に対しては float *dEdb2(100)、3 段目の FC 層パラメータ A に対する勾配は float *dEdA3(10×100),b に対しては float *dEdb3(10)である。

なお誤差逆伝播における FC 層と ReLU 層では順伝播時の入力が計算に必要となるので、順伝播を行う必要があり、１段目の FC 層のパラメータ const float *A1(50×784), const float *b1(50)、2 段目の FC 層のパラメータ const float *A2 (100×50),const float *b2(100)、3 段目の FC 層のパラメータ const float *A3(10×100), const float *b3(10)を指定する。

unsigned char t は 0〜9 の正解データであり、ソフトマックス層での誤差逆伝播に用いる。

学習に使用する 784 次元の入力データを const float *x に、その順伝播での出力を float *y に出力する。関数呼び出し後に計算で動的メモリ確保していたいくつかの配列は最後に free 関数を用いてメモリ領域を開放する。

行列和・スカラー積・初期化

```
void add(int n,const float *x,float *o){
    for (int i = 0; i < n;i++){
        o[i] = x[i] + o[i];
    }
}

void scale(int n,float x,float *o){
    for (int i = 0; i < n;i++){
        o[i] = o[i] * x;
    }
}

void init(int n,float x,float *o){
    for (int i = 0; i < n;i++){
        o[i] = x;
    }
}
```

関数 add の引数は行列の要素数を int n，加える行列を const float *x に、加えた結果を保存する行列を float *o に指定する。
関数 scale の引数は行列の要素数を int n，処理を行う行列を float *o に、行列にかけるスカラーを float x に指定する。
関数 init の引数は行列の要素数を int n，処理を行う行列を float *o に、初期化する数字を float x に指定する。

行列のコピー

```
//行列データのコピー
void copy(int n,const float *x,float *o){
    for (int i = 0; i < n;i++){
        o[i] = x[i];
    }
}
```

関数 copy の引数は行列の要素数を int n，コピー元の行列を const float *x に、コピー先の行列を float *o に指定する。
学習において現時点で最も正答率の高かったパラメータ A,b をコピーしておくことで、試行したエポックの中で最良の結果を示したパラメータを最終的に保存するためにコピー関数を用意している。

一様分布による乱数生成(rand 関数を用いた)

```
void rand_init(int n,float *o){
    for (int i = 0; i < n;i++){
        o[i] = (float)(rand() - (RAND_MAX / 2)) / (RAND_MAX / 2); //[-1:1]
    }
}
```

関数 rand_init の引数は初期化する行列の要素数を int n に,出力先の行列を float *o に指定する。
C 言語の rand 関数を用いて比較的シンプルな方法で一様分布に従う疑似乱数を生成した。

ボックスミュラー法を用いた正規分布に従う乱数生成

```
//ボックスミュラー法を用いた正規分布に従う疑似乱数の生成
double rand_normal( double mu, double sigma ){
    double z=sqrt( -2.0*log(genrand_real3())) * sin( 2.0*M_PI*genrand_real3());
    return mu + sigma*z;
}
```

関数 rand_normal の引数は生成する乱数列の平均を double mu, 標準偏差を double sigma で指定する。関数を読み出すと生成した乱数を返り値として return する。
ボックスミュラー法を用いて一様分布に従う確率変数から正規分布に従う疑似乱数を生成する。ここでの一様分布は Mersenne Twister で一様実乱数(0,1)を生成する関数として用意されている genrand_real3()を用いた。
rand 関数よりも優れた乱数生成アルゴリズムである、メルセンヌ・ツイスタを使用するため、ネット上に公開されている http://www.sat.t.u-tokyo.ac.jp/~omi/code/MT.h を取得しインクルードして使用している。

標準偏差 0.01 のガウス分布に従う乱数での初期化

```
//標準偏差0.01のガウス分布に従う重みの初期化
void normal_init(int n,float *o){
    for(int i=0;i<n;i++){
        o[i]=rand_normal(0,0.01);
    }
}
```

関数 normal_init の引数は初期化する行列の要素数を int n に,出力先の行列を float *o に指定する。上でボックスミュラー法を用いた rand_normal 関数を呼び出すことで平均 0,標準偏差 0.01 の正規分布に従う疑似乱数を生成し行列の各要素としている。

Xavier の初期値

```
//Xavierの初期値
void xavier_init(int n,float *o){
    for(int i=0;i<n;i++){
        o[i]=rand_normal(0,sqrt(1.0/n));
    }
}
```

関数 xavier_init の引数は初期化する行列の要素数を int n に,出力先の行列を float *o に指定する。rand_normal 関数を呼び出すことで平均 0,標準偏差$\frac{1}{\sqrt{n}}$の正規分布に従う乱数を生成し行列の各要素としている。

He の初期値

```
//Heの初期値
void he_init(int n,float *o){
    for(int i=0;i<n;i++){
        o[i]=rand_normal(0,sqrt(2.0/n));
    }
}
```

関数 he_init の引数は初期化する行列の要素数を int n に,出力先の行列を float *o に指定する。rand_normal 関数を呼び出すことで平均 0,標準偏差$\sqrt{\frac{2}{n}}$ の正規分布に従う乱数を生成し行列の各要素としている。

データの保存

```
//係数の保存
void save(const char *filename,int m,int n,const float *A,const float *b){
    FILE *fp;
    fp = fopen(filename,"wb");
    fwrite(A,sizeof(float),m*n,fp);
    fwrite(b,sizeof(float),m,fp);
    fclose(fp);
}
```

関数 save の引数は保存するファイルの名前を const char *filename に文字列で指定し、保存する行列 A,b を const float *A const float *b に指定する。int m が行列の行数で int n が行列の列数であるので、行列 A はm×n 行列,行列 b はm×1 行列でなければならない。

データの読み込み

```
//データの読み込み
void load(const char *filename,int m,int n,float *A,float *b){
    FILE *fp;
    fp = fopen(filename, "rb");
    fread(A,sizeof( (size_t)4UL );
    fread(b,sizeof(float),m, fp);
    fclose(fp);
}
```

関数 load の引数は読み込むファイルの名前を const char *filename に文字列で指定し、データを反映させる行列 A,b を const float *A const float *b に指定する。int m が行列の行数で int n が行列の列数であるので、行列 A はm× n 行列,行列 b はm×1 行列でなければならない。

3. Optimizer の導入

★深層学習における勾配降下法において確率的勾配降下法(SGD)だけではなく様々な手法が用いられている。実際に広く用いられているいくつかの最適化アルゴリズムを実際に実装し、パラメータ更新の最適化手法によって本文字認識における正解率が向上するのか検証を行った。

Momentum SGD

```
void momentum(int n,float *v,float *o,float *t){
    float mu = 0.9;
    float r = 0.1;
    for(int i=0;i<n;i++){
        v[i] = mu * v[i] - r * o[i];
    }
    add(n,v,t);
}
```

関数 momentum は int n に要素数,float *o に勾配,float *t に更新するパラメータを指定する。float *v が momentum と呼ばれる項で前回の更新量を保存して次回に加算することでパラメータの更新をより慣性的に行うことを意図している。

$\eta = 0.1$, $\alpha = 0.9$ として一般的に用いられているパラメータで設定した。

下式のようにパラメータが更新される。

$$\mathbf{w}^{t+1} \leftarrow \mathbf{w}^{t} - \eta \frac{\partial E(\mathbf{w}^{t})}{\partial \mathbf{w}^{t}} \quad + \alpha \Delta \mathbf{w}^{t}$$

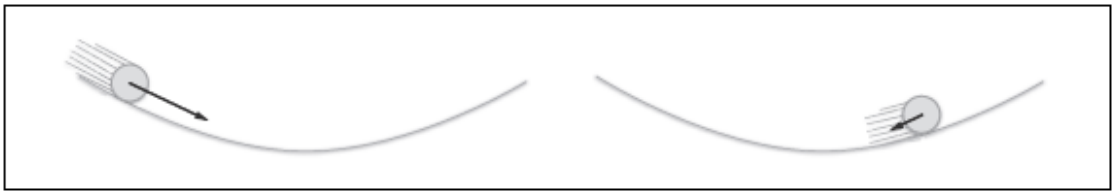


図 6-4　Momentum のイメージ：ボールが地面の傾斜を転がるように動く

(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から抜粋)

AdaGrad の実装

```
//Optimizer:AdaGrad
void AdaGrad(int n,float *h,float *i,float *o){
    float alpha = 0.001;
    float *ht = malloc(sizeof(float)*n);
    for(int j=0;j<n;j++){
        h[j] = h[j]+ i[j] * i[j];
        ht[j] = alpha / sqrt(h[j]);
        o[j] = o[j] - ht[j] * i[j];
    }
    free(ht);
}
```

Adagrad は学習係数の減衰を行い、学習係数を自動で調整してくれる最適化手法である。これまで経験した勾配の値を二乗和として float *h に保存するために、パラメータの更新の際に$\frac{1}{\sqrt{h}}$ で乗算することで、大きく更新された要素の学習率をパラメータの要素ごとに個別に減衰させることを可能にする。int n に要素数,float *i に勾配,float *o に更新するパラメータを指定する。

$\alpha = 0.001$ として一般的に用いられているパラメータで設定した。

下式のようにパラメータが更新される。

$$h_0 = \epsilon$$

$$h_t = h_{t-1} + \nabla E(\mathbf{w}^{t})^2$$

$$\eta_t = \frac{\eta_0}{\sqrt{h_t}}$$

$$\mathbf{w}^{t+1} = \mathbf{w}^{t} - \eta_t \nabla E(\mathbf{w}^{t})$$

Adam の実装

```
//Optimizer:Adam
void Adam(int n,float *m,float *v,float *i,float *o,float t){
    float beta1 = 0.9;
    float beta2 = 0.999;
    float alpha = 0.001;
    float *mh = malloc(sizeof(float)*n);
    float *vh = malloc(sizeof(float)*n);
    for(int j=0;j<n;j++){
        m[j] = beta1*m[j]+(1-beta1)*i[j];
        v[j] = beta2*v[j]+(1-beta2)*i[j]*i[j];

        mh[j] = m[j] / (1-pow(beta1,t));
        vh[j] = v[j] / (1-pow(beta2,t));

        o[j] = o[j] -alpha*mh[j]/(sqrt(vh[j])+1e-7);
    }
    free(mh);free(vh);
}
```

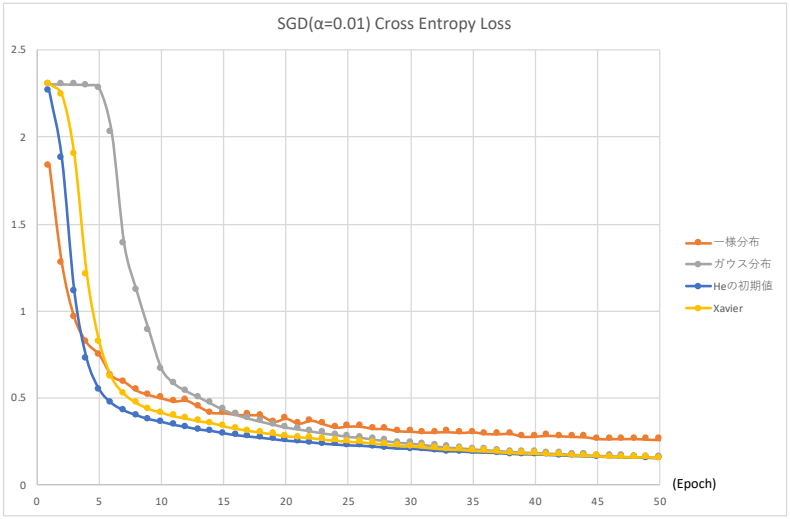
Adam は Momentum と AdaGrad を融合させたような最適化手法で、効率的にパラメータ空間の探索を行い、ハイパーパラメータのバイアス補正を行っている。勾配の一次モーメントと二次モーメントを float *m, float *v で保存する。
$\alpha = 0.001$とし、論文の初期値のまま$\beta_1 = 0.9, \beta_2 = 0.999, \epsilon = 10^{-8}$とした。
[Adam: A Method for Stochastic Optimization] https://arxiv.org/abs/1412.6980

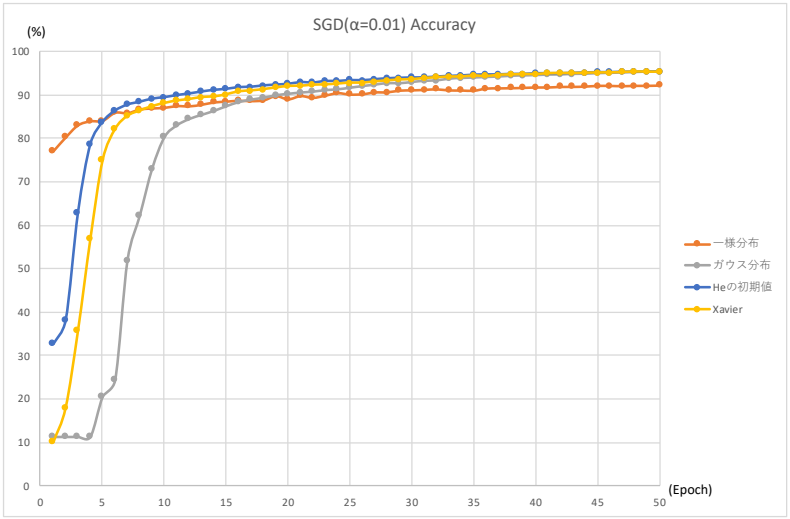
int n に要素数,float *i に勾配,float *o に更新するパラメータ、float t に時間を指定する。
下式のようにパラメータが更新される。

$$
\begin{aligned}
m_{t+1} &= \beta_1 m_t + (1-\beta_1)\nabla E(\mathbf{w}^t) \\
v_{t+1} &= \beta_2 v_t + (1-\beta_2)\nabla E(\mathbf{w}^t)^2 \\
\hat{m} &= \frac{m_{t+1}}{1-\beta_1^t} \\
\hat{v} &= \frac{v_{t+1}}{1-\beta_2^t} \\
\mathbf{w}^{t+1} &= \mathbf{w}^t - \alpha \frac{\hat{m}}{\sqrt{\hat{v}}+\epsilon}
\end{aligned}
$$

4. 様々なモデルの比較

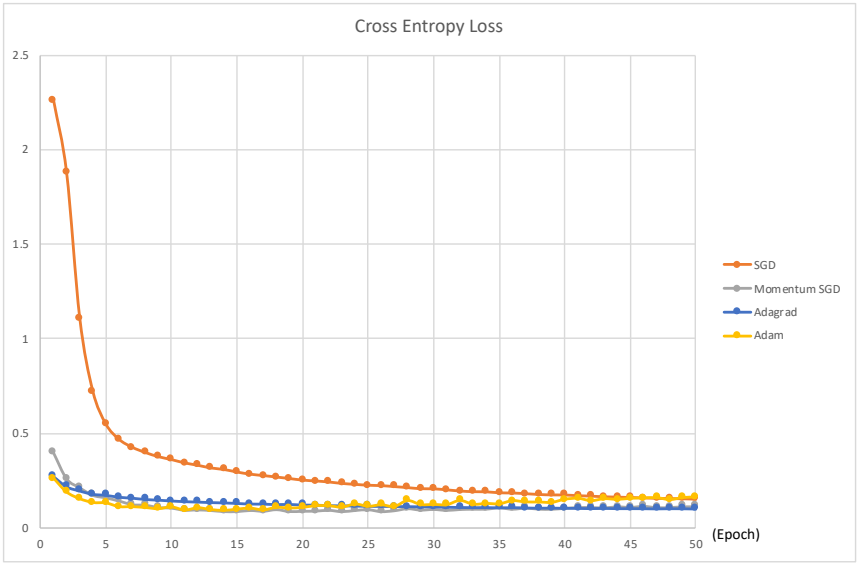
パラメータの重みの初期化に用いる乱数分布を一様分布、標準偏差 0.01 のガウス分布、He の初期値、Xavier の４種類から選択できるようにしていたが、活性化関数に ReLU を用いた場合標準偏差を$\sqrt{\frac{2}{n}}$ とする He の初期値を用いることが最良であるという本の記述が本当に正しいのか疑問に感じた。
そこで標準的なミニバッチ学習での確率的勾配降下法(SGD)において４パターンそれぞれ 50epoch の学習を行ったときの損失関数と正解率の推移を計測しグラフ化した。

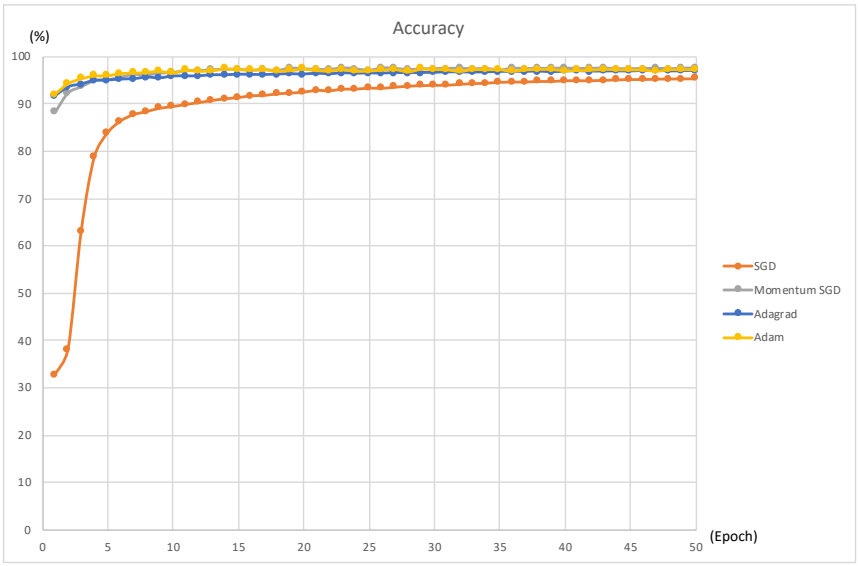




損失関数・正解率ともに He の初期値を用いた初期化のモデルが最も優れていることがわかる。明らかに一様分布は他３つのものよりも劣っていることが読み取れ、ガウス分布は初めこそもっとも劣っているものの、エポックを重ねると He の初期値や Xavier の初期値のパフォーマンスにほとんど近づいている。
シグモイド関数や tanh の場合に用いられる Xavier との差異はそこまで認められない。

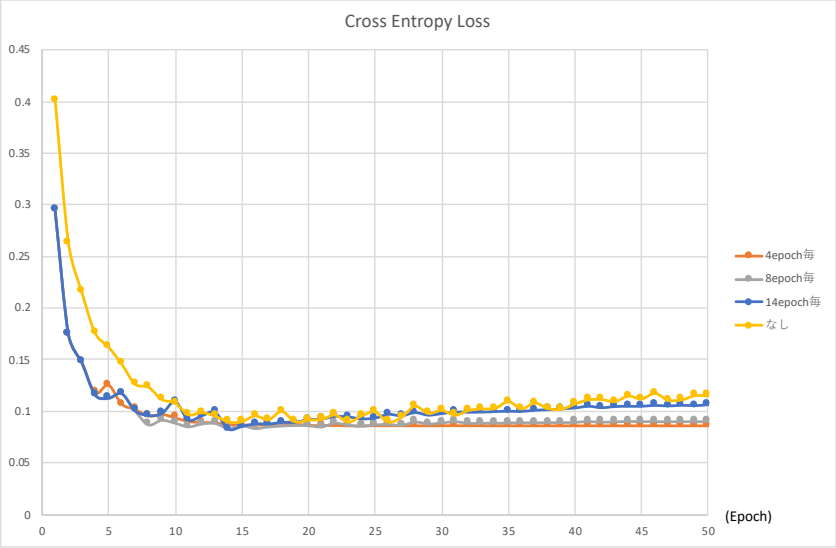
次に Optimizer による正解率の差はどの程度のものなのか検証を行った。
検証に用いるのは Learning.c で採用した SGD,Momentum SGD,AdaGrad,Adam の４種類の最適化手法で 50epoch の学習をおこなったときの損失関数と正解率の推移を同様に記録しグラフ化した。
なお各ハイパーパラメータは一般的に用いられているデフォルトの値のままで行った。



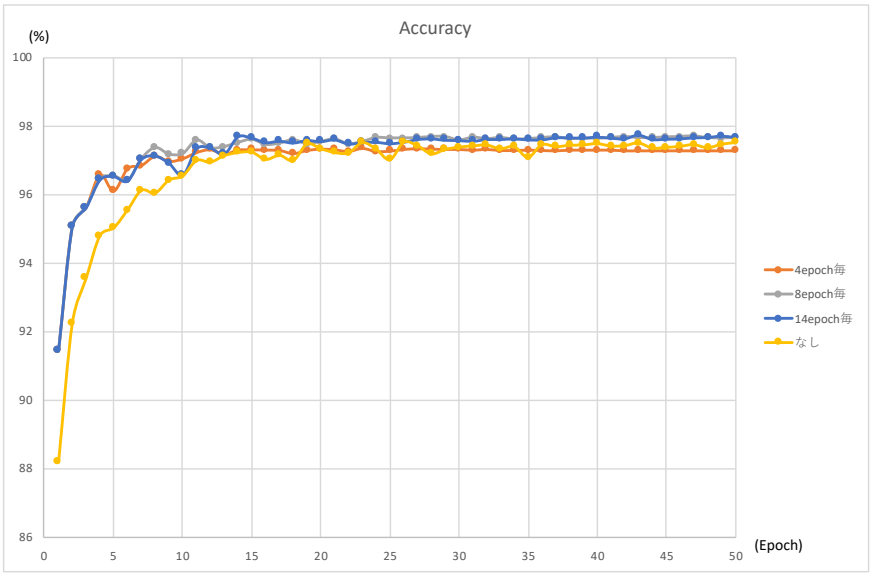


SGD を除いた 3 つの Optimizer を用いたモデルではどれも 50epoch 時点で 97.7%程度で学習が収束している。SGD は損失関数も正解率も初めはひどく劣っているように感じられるが、エポックとともに単調に改善していく様子が確認できる。Adam は初めはもっとも早いスピードで最良の学習を行っているが、15epoch を過ぎたあたりから損失関数は学習とともに増加しているのでハイパーパラメータの調整と過学習の抑制を加える必要があると考えられる。総合的に判断して今回のデータからは正解率が最も高かった Momentum SGD または AdaGrad を選択することが良いと感じられる。

最後に Momentum SGD において学習率をそのままにしておくのではなく、エポックとともに減衰させてみるとさらに良いパフォーマンスが得られるのではないかと考え実験を行った。
各 optimizer においてハイパーパラメータをいろいろといじってみたが、初めはある程度大きくないと学習が進まないし、大きすぎても正解率があまり高くないところで振動してしまうために定値ではなく減衰が必要ではないかと感じたためである。
指定したエポック毎に学習率を 2 分の 1 に半減させている。
減衰なしの条件で最も学習が進んでいた初期値$\eta = 0.02$, $\alpha = 0.9$で実験を行った。



4epoch 毎または 8epoch 毎に学習率を半減させたモデルではエポックを重ねても
クロスエントロピー誤差（損失関数）の値は増加することなく収束している。
正解率では 8epoch 毎のモデルが最も高く、何もしていないときよりも 0.5%程度改善していることが読み取れる。ハイパーパラメータの選択によりさらに改善は可能だと感じる。



5. 参考文献

1. 斉藤 家毅　「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」　オライリー社

2. C.M.ビショップ　「パターン認識と機械学習　上　ベイズ理論による統計的予測」　丸善出版

3. Panda 電気電子プログラミング演習及演習　総合課題レジュメ