

電気電子プログラミング及演習
総合課題レポート

京都大学工学部電気電子工学科

二回生

中島魁人

1026331261

1. プログラムの概要

学習部分と推論部分でプログラムを分けた。学習は `Training_6Layer_Gauss.c` で行い、推論は `Inference_6Layer.c` で行う。また、関数のほとんどは `Training_6Layer_Gauss.c` に記述されており、`Inference_6Layer.c` には必要最低限の関数を記述した。

まず、プログラム `Training_6Layer_Gauss.c` について説明する。

関数は全部で 22 個ある。まず `main` 関数がコマンドライン引数として 3 つの `dat` ファイルを受け取る。これには最終的に学習した後の `A1, b1` などの行列データ(パラメータ)を書き込むことになる。

まずは `main` 関数内で必要な動的配列の宣言を行う。そして、エポック回数やミニバッチサイズ、学習率などの初期化を行う。次に `A1, b1` などの用意した動的配列を平均 0、分散 2/入力次元 のガウス分布によって初期化する。次にエポック回数だけ次の計算を繰り返す。まず、訓練データを指定する配列 `index` を訓練データ分用意し、重複が無いようにシャッフルする。次にミニバッチ学習を `train_count / minibatchsize` 回繰り返す。ミニバッチ学習では平均勾配を 0 で初期化し、ミニバッチサイズ回だけ、誤差逆伝搬法をして平均勾配を更新していく事を繰り返す。それが終わると、`A1, b1` などの行列データを $A1 = A1 - (\text{学習率} / \text{ミニバッチサイズ}) * \text{平均勾配}$ という式に従って更新していく。1 エポック終わると、推論関数によって学習を行った行列データを用いた推論と正解データとの誤差を計算し、正解率、誤差関数の表示を行う。ここまでがエポック回繰り返す内容である。最後にコマンドライン引数で指定した `dat` ファイルに学習した行列を書き込み、`main` 関数の初めで宣言していた動的配列のメモリを開放すると終了する。

次に、プログラム `Inference_6Layer.c` について説明する。

まず、`main` 関数で必要な動的配列の宣言を行う。次にコマンドライン引数から受け取った 3 つの `dat` ファイルを配列にロードし、6 層の推論を行う。推論では `fc1 層`→`relu1 層`→`fc2 層`→`relu2 層`→`fc3 層`→`softmax 層`と順に計算していき、最後に戻り値として推論した数字を返す。`main` 関数でその推論した数字を表示して、宣言した動的配列を開放すると終了する。

2. 関数の説明

引数に受け取る配列について説明しやすくするために 配列(`m, n`) という風に記述する。`m` は行、`n` は列である。配列の場合要素数が `m*n` となることで行列を表現している。

まず、`Training_6Layer_Gauss.c` について上から順に説明する。

- print

戻り値無しで、引数に行列の行に相当する整数 $m>0$ 、列に相当する整数 $n>0$ 、配列 $x(m, n)$ を受け取る。二重ループによって m 行 n 列の行列を表示する。

- fc

戻り値無しで、引数に行列の行に相当する整数 $m>0$ 、列に相当する整数 $n>0$ 、配列 $x(n, 1)$, $A(m, n)$, $b(m, 1)$, $y(m, 1)$ を受け取る。 $y=Ax+b$ という計算を行いたいので、二重ループにして、一段目で行を操作して、 y に b を足していき二段目で列を操作して m 行目の A と列ベクトル x をかけてそれを y に足していく事で更新していく。

- relu

戻り値無しで、引数に列ベクトルの行数に相当する整数 $n>0$ 、配列 $x(n, 1)$, $y(n, 1)$ を受け取る。Relu 演算では 0 以上ではそのまま、0 以下だと 0 にするので、if-else 分を用いて n 回 for ループを回して計算していく。

- softmax

戻り値無しで、引数に列ベクトルの行数に相当する整数 $n>0$ 、配列 $x(n, 1)$, $y(n, 1)$ を受け取る。最大値を記録する float 型の変数 x_Max 、和を記録する float 型の変数 S を用意し初期化しておく。for ループを n 回回して x の最大値を x_Max に記録する。次に for ループを n 回回して S を計算する。最後に y に $\exp(x-xMax)/S$ を計算したものを代入していく。

- softmaxwithloss_bwd

戻り値無しで、引数に列ベクトルの行数に相当する整数 $n>0$ 、配列 $y(n, 1)$, $dEdx(n, 1)$ 、正解の整数 $t(0\sim9)$ を受け取る。for ループを n 回繰り返す。配列 $i+1$ 行目と正解データの数字が一致するなら t は 1 を返し、それ以外なら 0 を返すので(one-hot 表現)その処理を if-else 分によって記述していく。

- relu_bwd

戻り値無しで、引数に列ベクトルの行数に相当する整数 $n>0$ 、配列 $x(n, 1)$, $dEdy(n, 1)$, $dEdx(n, 1)$ を受け取る。for ループを n 回繰り返して $x>0$ なら $dEdx = dEdy$ それ以外なら $dEdx = 0$ となるように if-else 文を用いて記述していく。

- fc_bwd

戻り値無しで行列の行と列に相当する整数 $m>0$, $n>0$ 、配列 $x(1, n)$, $dEdy(m, 1)$, $A(m, n)$, $dEdA(m, n)$, $dEdb(m, 1)$, $dEdx(1, n)$ を受け取る。まず二重ループで $dEdA$ の行列の計算を行う。二つ目の一つ目のループで A の行と $dEdy$ の行を動かし、二つ目のループで列を動かす。 A の $i+1$ 行目の $j+1$ 列目を $dEdy$ の $i+1$ 行目と x の $j+1$ 列目を掛け合わせることで計算できる。次の for ループでは $dEdb$ の $i+1$ 行目に $dEdy$ の $i+1$ 行目を代入していく。次の二重ループでは初めの for ループで $dEdx$ の $i+1$ 列目を初期化した後(n 回)、中の for ループで $dEdx$ の $i+1$ 列目を A の $j+1$ 行目 $i+1$ 列目と $dEdy$ の $j+1$ 行目を掛け合わせることを m 回繰り返す。

- shuffle

戻り値無しで、引数に繰り返し回数 $n > 0$ 、配列 x を受け取る。一時的な `int` 型変数 t を宣言し、以下を変数 i で n 回繰り返す。`int` 型変数 j を $0 \sim n$ の整数を返すように設定する。一時的な変数 t に $x[i]$ を記録し、 $x[i]$ に $x[j]$ を記録、 $x[j]$ に t を記録して $x[i]$ と $x[j]$ をスワップする。これにより重複なくある範囲の連続した整数をシャッフルできる。

- `cross_entropy_error`

戻り値が損失関数 $\log(y + 1e-7)$ で引数に配列 $y(10, 1)$ 、正解ラベル t を受け取る。 t の時の one-hot 表現を考えると例えば $t=2$ の時は $t_k(0, 0, 1, 0, 0, 0, 0, 0, 0, 0)$ のようにして考えるとよいので、 t_k の値が 1 になるようなときの $\log(y + 1e-7)$ の値を返す。

- `add`

戻り値が無し、引数に繰り返し回数 n 、配列 `const float x, float o` を受け取る。以下の for ループを変数 i として n 回繰り返す。 o の $i+1$ 行目に x の $i+1$ 行目を加える。

- `scale`

戻り値無し、引数に繰り返し回数 n 、整数 x 、配列 `float o` を受け取る。以下を変数 i として n 回繰り返す。 o の $i+1$ 行目に x をかける。

- `init`

戻り値無し、引数に繰り返し回数 n 、整数 `const float x`、配列 `float o` を受け取る。以下の for ループを変数 i で n 回繰り返す。 o の $i+1$ 要素目を x で書き換える。これにより配列 o は全ての行が x で初期化される。

- `rand_init`

戻り値無し、引数に繰り返し回数 n 、配列 `float o` を受け取る。以下の for ループを変数 i で n 回繰り返す。 o の $i+1$ 要素目を乱数で生成した $[0:1]$ の実数を入れていく。`Rand()` は $0 \sim \text{RAND_MAX}$ を取るので先に 2.0 をかけておいて `double` 型に変換した後それを RAND_MAX で割るとランダムに $[0:2]$ の実数を返すようになる。さらに -1 を足すことで $[-1:1]$ の実数をランダムに返すようになる。

- `Mersenne_Twister1`

戻り値は `double` 型の関数、引数無し。メルセンヌツイスター法により非常に長い周期 $2^{19937}-1$ ($\approx 4.315 \times 10^{6001}$) で一様乱数を生成している。ヘッダファイル `MT.h` に `genrand_real3()` が定義されている。これにより高い精度の一様乱数 $[0:1]$ を生成して戻り値で返す。

- `Mersenne_Twister2`

`Mersenne_Twister1` と同様。ボックスミュラー法で用いる。

- `rand_normal`

戻り値 `double` 型の正規分布、引数に平均 n 、標準偏差 σ を受け取る。ボックスミュラー法により正規分布を一様分布を用いて生成できる。

- `gauss_init`

戻り値無し、引数に繰り返し回数 n 、初期化する配列 o を受け取る。以下を変数 i で n 回繰り返す。 o の $i+1$ 要素目を `rand_normal` により平均 0、標準偏差 $\sqrt{2.0/n}$ (試行回数) を計

算して書き換える。

`inference3, backward3` は最終的に使っていないが、関数自体の説明だけ載せておく。

・ `inference3`

戻り値は `int` 型の推論結果 `inference`、引数に配列 `A, b, x, y` を受け取る。`A` は 784×10 の配列、`b` は ReLU の出力として用いる 10 行の列ベクトル、`x` は教師データの画像を読み込む配列で 28×28 の行列を表し、`y` は `softmax` の出力として用いる 10 行の列ベクトルである。

動的配列 `y1` を用意し、`fc→relu→softmax` の順に実行し、`y` の最大値を記録する `int` 型変数 `max` を用意しておき、`for` ループと `if` 文を組み合わせで最大値を推論結果として返す。

・ `backward3`

戻り値は無し、引数に配列 `A, b, x, y, dEdA, dEdb` 教師データ `x` に対応する正解ラベル `int` 型変数 `t` を受け取る。`A` は 784×10 の配列、`b` は ReLU の出力として用いる 10 行の列ベクトル、`x` は教師データの画像を読み込む配列で 28×28 の行列を表し、`t` は教師データの画像に対する正解ラベル。`y` は `softmax` の出力として用いる 10 行の列ベクトル、`dEdA` は 784×10 の配列、`dEdb` は 10 行の列ベクトル。`y1` は FC の出力として保存しておき逆誤差伝搬法で用いるための配列。動的配列 `y1` を用意し `fc, relu, softmax` を行って、`y1, y` の更新をする。`dEdx1, dEdx2` を用意し、逆誤差伝搬法を行う。

・ `inference6`

戻り値は `int` 型の推論結果 `inference`、引数に配列 `A1, b1, A2, b2, A3, b3, x, y` を受け取る。`A1` は FC1 で 28 行 28 列 ($28 \times 28 = 784$) の画像を読み取り 50 行の列ベクトルとして ReLU1 に受け渡すので、配列のサイズは 784×50 、`b1` は 50 行の列ベクトル、`A2` は FC2 で 50 行の列ベクトルを受け取り、100 行の列ベクトルとして ReLU2 に受け渡すので、配列のサイズは 50×100 、`b2` は 100 行の列ベクトル、`A3` は FC3 で 100 行の列ベクトルを受け取り、10 行の列ベクトルとして `Softmax` に受け渡すので、配列のサイズは 100×10 、`b3` は 10 行の列ベクトル、`x` は教師データの画像を読み込む配列で 28×28 の行列を表し、`y` は `softmax` の出力として用いる 10 行の列ベクトルである。

動的配列 `y1, y2` を用意し(一時的に用いるため)、`fc1, relu1, fc2, relu2, fc3, softmax` と計算を行っていく。そして最後に更新された `y` の値の最大値を保存しておく `int` 型変数 `max` を用意しておき、`for` ループを 10 回回すことで `y` の最大値が 0~9 のどの整数で取るのか `if` 文と `max` を用いながら調べて最大値をとる添え字 `i` を推論結果として保存しておく。動的配列を開放した後、推論結果を戻り値として返す。

・ `backward6`

戻り値無し、引数に配列 `A1, b1, A2, b2, A3, b3, x, y, dEdA1, dEdb1, dEdA2, dEdb2, dEdA3, dEdb3` 教師データ `x` に対応する正解ラベル `int` 型変数 `t` を受け取る。`dEdA1, dEdb1, dEdA2, dEdb2, dEdA3, dEdb3` は `A1-b3` の勾配で配列の大きさは勾配を取る前に等しい。

動的配列 `yf1, yr1, yf2, yr2` を用意しておき `fc1, relu1, rc2, relu2, fc3, softmax` を行う。

yf1,yr1,yf2,yr2 はそれぞれ FC1,ReLU1,FC2,ReLU2 の出力を保存しておき逆誤差伝搬法で使うための配列である。ここではあえて inference6 を用いず fc や relu や softmax を用いて計算していく(関数 inference6 の引数を減らして見通しを良くするため)。inference6 部分が終わると動的配列 dEdx1, dEdx2, dEdx3, dEdx4 を用意しておき inference6 の時とは逆向きの順番で softmax,fc,bwd を実行していき dEdA1, dEdb1 などの勾配を逆誤差伝搬法によって更新していく。最後に、もう必要ない動的配列を開放すると終了する。

- save

戻り値無し、引数に書き込む dat ファイルの名前(main 関数内でコマンドライン引数で指定)、行列の行数 m 列数 n、そして保存したい配列の A, b を受け取る。まず、ファイルポインタを用意し、もし、ファイルが開けなかったらエラー文を表示した後、異常終了する。開けたら A, b の中身を fwrite 関数を用いてファイルポインタに書き込む。そしてファイルポインタを閉じて終了する。

Inference_6Layer.c について説明する。Training_6Layer_Gauss.c から新規に追加した load 関数のみ説明する。

- load

戻り値無し、引数に読み込む dat ファイルの名前(main 関数内でコマンドライン引数で指定)、行列の行数 m 列数 n、そして読み込んだ配列の A, b を受け取る。まず、ファイルポインタを用意し、もし、ファイルが開けなかったらエラー文を表示した後、異常終了する。開けたら fread によって A, b にファイルポインタの中身を書き込む。そしてファイルポインタを閉じて終了する。

3. 工夫・拡張・考察事項

- ハイパーパラメータを変えてみる。

- 学習率

今回実施した機械学習では確率的勾配降下法を用いており、そのパラメータの更新では損失関数が最小になるように変化していくようになっている。パラメータは特に機械学習ではハイパーパラメータと呼ばれることが多く、特にエポック数やミニバッチサイズ、学習率などの人が工夫して最適値を見つけ出さないといけないようなものを指している。最適化が進むにつれて損失値を求めるグラフの勾配が緩やかになっていくと考えられ、大きい学習率を用いると確かに学習の初期は学習が早く進むと考えられるが、ある時点から収束せずにグラフを行ったり来たりして最適値にたどり着けないという問題が起きてしまう。一方、学習率をはじめから小さくしすぎると本当の最適値ではなく局所解にとらわれてしまいそこから学習が進まなくなってしまうという問題が起こることがある。そこで、

学習の初期はある程度の大きさの学習率を取っておき、学習が進むにつれてその値を徐々に小さくしていくと収束の効率が上がるということが分かった。そこで私は正解率に基づいて学習率の更新を行うような式を用意してそれを組み込んでみた。以下では if 文を用いて簡単な分岐によって学習率を更新している。

```
if (sum * 100.0 / test_count >= 95.0 && sum * 100.0 / test_count < 96.0)
    learningrate = 0.05;
if (sum * 100.0 / test_count >= 96.0 && sum * 100.0 / test_count < 97.0)
    learningrate = 0.025;
if (sum * 100.0 / test_count >= 97.0)
    learningrate = 0.01;
```

・ミニバッチサイズ

ミニバッチサイズを変更してみて学習の様子がどのように遷移するのか確認してみた。ミニバッチの単位が小さければ小さいほど、1 つ 1 つのデータに敏感に反応すると考えることができます。逆にミニバッチの単位が大きければ大きいほど、平均化されるので、1 つ 1 つのデータよりもミニバッチ全体の特徴を捉えるということになります。[1]
以上のような特徴があるということなので実際にミニバッチサイズを 1000,100,50,15,10,5,1 と減らしていくとどのようなようになるのかを試した。

今回行ったディープラーニングはエポック回数ミニバッチ学習を繰り返すものであり、index をシャッフルして、エポックごとに同じ index が出ないように設定がされているため、1 エポック内ですべての教師データを用いることが出来るように、ミニバッチサイズはトレインカウント(60000)の約数となるように選んだ。また、この検証は他の検証よりも後に行ったため、他の検証においてはミニバッチサイズ 100 として行っている。

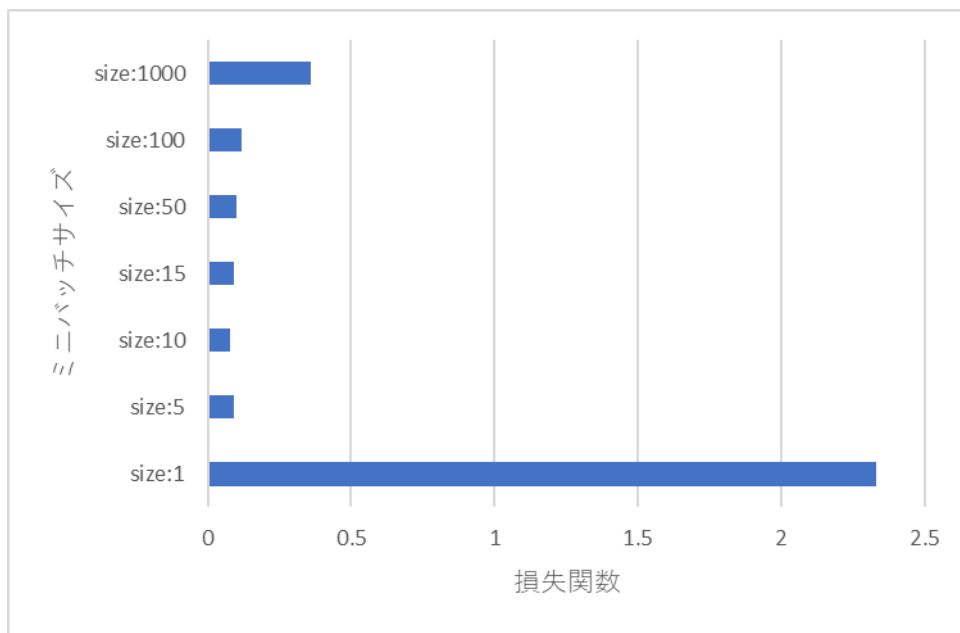


図 1 10 エポック後の損失関数

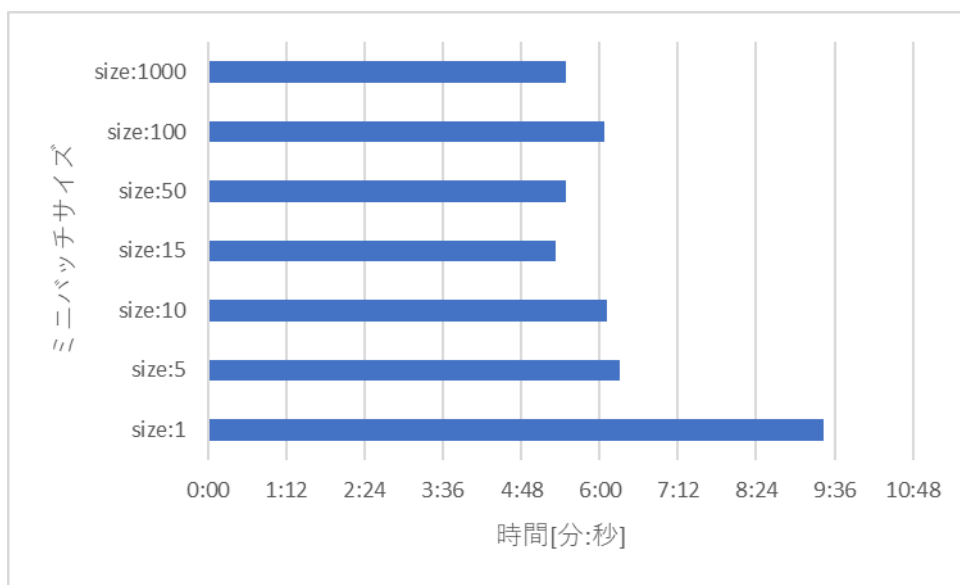


図 2 10 エポック後の経過時間

図 1, 2 にミニバッチサイズを変えた時の学習結果を示した。サイズ 1000 のときは学習の速度はそこそこ早くなったが、ミニバッチサイズが大きすぎるためか学習初期の正解率、損失関数が著しく悪かったため、エポック 10 後の損失関数も相対的に悪い結果となった。

サイズ 100~5 に関しては大きな違いは存在しなかったが、細かい違いに着目すると、特

にサイズ 15,10 辺りでは学習速度が速いうえ、学習初期の正解率、損失関数が良く、また、測定したデータの中では学習速度はサイズ 15 において最小値を取ったので、およそ今回作成したディープラーニングの最適値に近いミニバッチサイズは 15 であると考えられた。

また、サイズを 1 にすると学習が一向に進まず、時間もかなりかかってしまった。これはパラメータの更新が小さすぎて局所解にとらわれてしまったことが原因だと思われる。

・3 層と 6 層の学習の遷移の比較

せっかく 3 層と 6 層のニューラルネットワークを作成したのでそれぞれの学習を比較してみてもそこからわかることを考察してみようと思う。

体感的にわかることは明らかに 3 層の時の学習が速いということは感じていた。

今回はエポック 10、ミニバッチサイズ 100、初期学習率 0.1、He の初期値(後述)として比較対照実験を行った。

3 層の時は初期の正解率、損失関数にばらつきが出た。正解率は 30%~50%程度の初期値であることが多く、損失関数はそれに応じて 1.8~1.3 程度であることが多かった。また、10 エポックに達するまでに 1 分かからないで 50 秒程度で学習が終了していた。最終的に正解率は 70~87%程度に達し、損失関数は 0.7~0.3 程度になった。

6 層の時は初期の正解率、損失関数にばらつきが少なかった。正解率は 90%程度の初期値であることが多く、損失関数はそれに応じて 0.4~0.3 程度であることが多かった。また、10 エポックに達するまでに 6 分程度かかって学習が終了した。最終的に正解率は 96~97%程度に達し、損失関数は 0.12~0.10 になった。

これらのことはパラメータの数によってある程度説明できる。

3 層の場合は $(784 \times 10 + 10) = 7850$ のパラメータ、一方 6 層では $(784 \times 50 + 50) + (50 \times 100 + 100) + (100 \times 10 + 10) = 45460$ ものパラメータがある。パラメータの比を単純計算すると $45460 / 7850 = 5.79$ となっており、計算速度の観点から見ると確かに 50 秒程度→6 分程度となって整合性の取れる結果となっている。また、パラメータが増えるとそれだけディープラーニングの分析の柔軟性や、結果の表現力が上がると考えられるため、層を増やした結果、これだけ精度が上がったのは当然の結果であったといえる。これ以上層を増やすとどのようなになるか予想してみる。ある程度層を増やしたところで限界に達し、そこからはむしろ層を増やしたところで精度が下がっていくのではないかと考えられる。また、3 層の場合において初期値にばらつきが大きかった説明としては、パラメータの数が少なすぎて He の初期化がきれいに行われず、適切な分布にならなかったことが原因ではないかと考えられた。分布については後述する部分において考察を行っている。

・C 言語でガウス分布を用いる

後に用いる平均 0、分散 $\text{sqrt}(2/\text{入力次元})$ のガウス分布を用いるために、C 言語でガウス分布を用いるための拡張について述べたいと思う。C 言語による乱数生成という WEB サイト[2]を参考にして、rand による一様乱数生成法より精度が高いメルセンヌ素数を用いた一様乱数生成法であるメルセンヌ・ツイスタ(MT)を用いた。その後、一様分布をもとにしてガウス分布を生成できる、ボックスミュラー法を用いた。

ヘッダファイル MT.h を上記のサイトからダウンロード後、ヘッダファイルを含める。

```
include "MT.h"
```

ヘッダファイル MT.h の中には、`genrand_real3()` という (0:1) の一様乱数を返す関数が定義されているため、それを用いる。まず、`void gauge_init(int n, float *o)` という関数を用意し o という n 次元の配列を for ループによってある値で初期化するという処理を繰り返す。

その際、それぞれの配列は `rand_normal(0, sqrt(2.0/n))` という第一引数に平均値、第二引数に標準偏差を取るような関数によって初期化する。

`rand_normal` ではボックス＝ミュラー法[3]により (0:1) の一様分布を

のような変換によって標準正規分布に変換できる。今回は特に z_2 の式を用いた。ここでの X, Y はそれぞれ (0:1) の一様分布である。また、生成した標準正規分布は平均値 0、標準偏差 1 であるのでこれを線形変換 $X = \sigma Z + \mu$ (σ : 標準偏差、 μ : 平均) によって任意の正規分布に変換することが出来る。一様分布は初めに述べたようなメルセンヌ・ツイスタによって生成できるので、これらの関数を組み合わせることによって用いたい分布を得ることが出来た。

・パラメータの初期値を変えた場合について考察を行う

一様分布、すべて 0、ガウス分布(He の初期値)のそれぞれを用いてパラメータを初期化した場合について、エポックごとの正解率、損失関数を以下に示した。

ハイパーパラメータは、エポック 10、学習率 0.1、ミニバッチサイズ 100 としている。

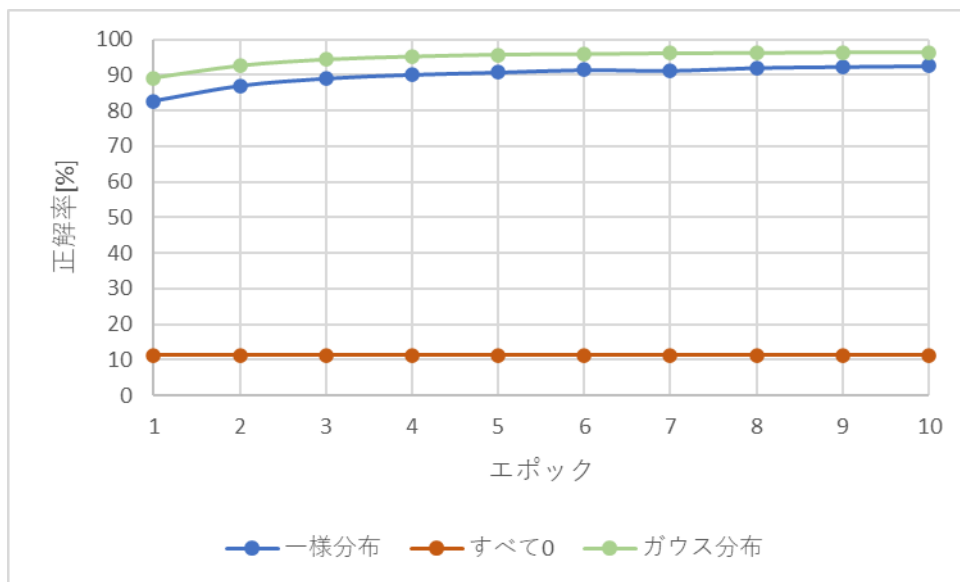


図 3 正解率の推移

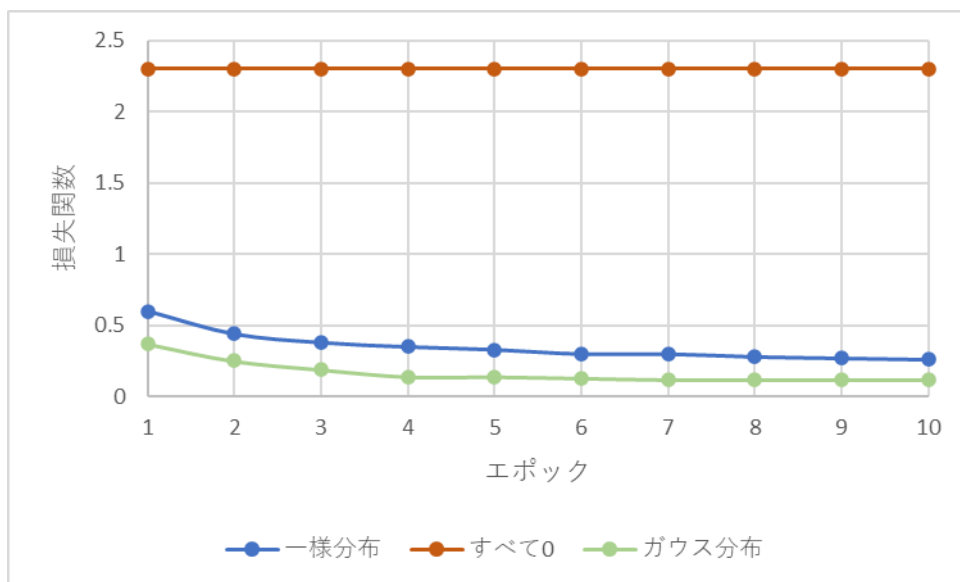


図 4 損失関数の推移

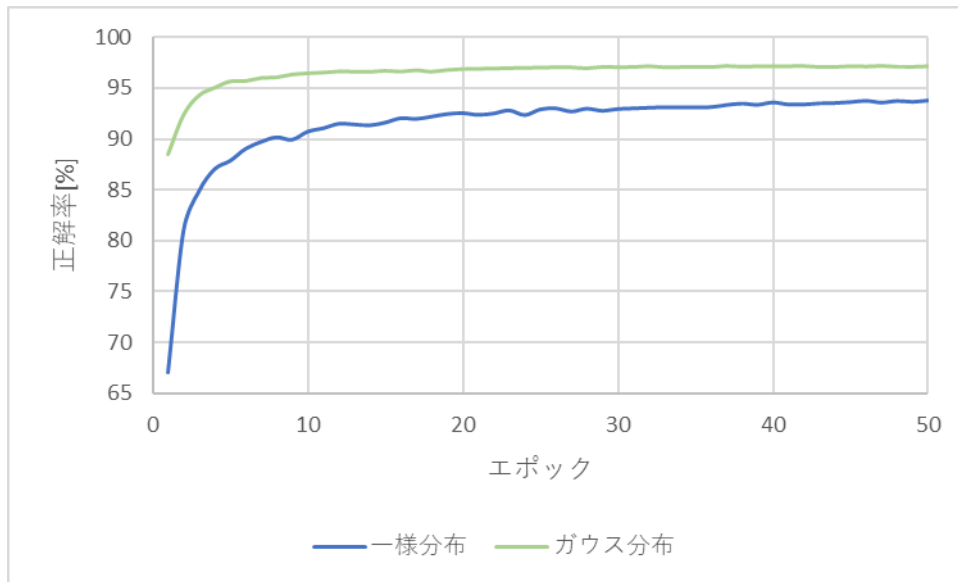


図 5 エポック 50 までの正解率の推移(図 1,2 の時とは別で実行)

確率的勾配降下法では重みの更新を繰り返すことによって最適な重みを求めるが、重みの更新のためにはあらかじめ重みの初期値を設定する必要がある。この重みの初期値によって学習結果が変わってくるため、適切な初期値を設定する必要がある。もし、適切な初期値を設定できなかったら、逆伝搬での勾配がどんどん小さくなって消えてしまう勾配消失や、複数のニューロンが同じ値を出力してしまう表現力の消失などの問題が生じてしまう。

・一様乱数による分布

まず用いたものは講義資料をそのまま実装した一様乱数による分布だ。 $[-1:1]$ の一様分布で平均は 0、分散は $1/3$ である。50 エポックほど学習を繰り返した結果、正解率は 93.75% 弱、損失関数は 0.24 ほどに収束した。

・すべて 0 で初期化した分布

次に、パラメータをすべて 0 で初期化した場合を考える。平均値、分散はともに 0 である。このパラメータをもちいた学習は失敗に終わった。正解率、学習率が初期値のまま変化しなかった。なぜこのようなことが起きたのか疑問に思ったので、ゼロから作る

DeepLearning[4]を参照してみると、以下のような記述があった。

たとえば、2 層のニューラルネットワークにおいて、1 層目と 2 層目の重みが 0 だと仮定します。そうすると、順伝搬時には、入力層の重みが 0 であるため、2 層目のニューロンにはすべて同じ値が伝達されます。2 層目のニューロンですべて同じ値が入力されるということは、逆伝搬のときに 2 層目の重みはすべて同じように更新されるということになり

ます。そのため、均一の値で更新され、重みは対称的な値を持つようになってしまいます。

つまり、パラメータをすべて 0 にしてしまうとパラメータの偏りが大きすぎることによって、学習の際に層ごとにデータをうまく伝達することが出来なくなってしまっていたという事と考えられる。

・ガウス分布を用いてパラメータを初期化したとき(He の初期値)

そして、最後に考えたのがガウス分布を用いてパラメータを初期化する方法だ。この方法は一般に He の初期値と呼ばれているようである。

平均 0、標準偏差 $\sqrt{2/(\text{入力次元})}$ のガウス分布を用いてパラメータを初期化すると、エポックが早い段階から正解率が高く、損失関数が小さく出ることが分かった。また、学習の進み具合が一様分布を用いた場合より早く、最終的に収束する正解率、損失関数の値も一様分布を用いた場合より改善された。エポック 30 ほどで収束し正解率 97.07%、損失関数 0.09 だった。

この方法は分散に特徴がある。もともと Xavier の初期値と呼ばれるディープラーニングで標準的に用いられるものがあり、その方法と He の初期値とではほぼ同じ分布を用いる手法であるのだが、Xavier の初期値では標準偏差を $1/\sqrt{(\text{入力次元})}$ としていたのに対し、He の初期値では $\sqrt{2/(\text{入力次元})}$ として学習を行う。Xavier の初期値がシグモイド関数などの活性化関数を用いる際に適しているのに対し、He の初期値は今回の総合課題でも用いた ReLU 関数という活性化関数を用いた際に適しているとされている。 $\sqrt{2}$ をかけている理由として、シグモイド関数は y 軸対称な関数である一方、ReLU 関数は x が負の領域では 0 になるので、よりパラメータの分布に広がりを持たせるために補正をかけているということだ。

まとめると、適切な初期値の設定には、分布が偏りすぎないことや分散の値が大きすぎず小さすぎないようにする必要があること。また、適切な初期値を設定することが出来れば初期段階の精度、収束速度の改善や、最終的な収束値などに大きな変化をもたらすことが分かった。

・フォントによる認識制度についての考察

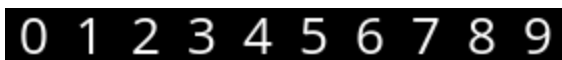
ImageMagick によって Windows11 に標準搭載されているフォントを用いて、28x28 サイズの画像をいくつか作成した。それらの画像データを学習したパラメータでどれほど正しく認識できるのか試すことにより、今回作成した手書き文字認識のニューラルネットワークがどれほどの汎用性があるのか調べてみた。

用いた学習パラメータはエポック 10、ミニバッチサイズ 100、初期学習率 0.1、He の初

期値を用いて学習したものである。

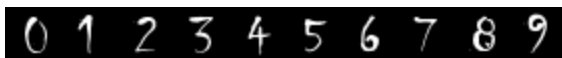
また、作成した画像ファイルは word に貼り付ける際に bmp ファイルから png ファイルに変換している。

・ Windows11 のデフォルトフォント



推論結果は順に 0,1,2,3,4,5,6,7,8,9
このときの正解率は 10/10 だった。

・ Chiller



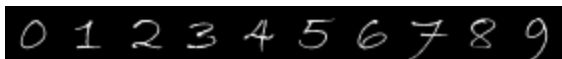
推論結果は順に 0,8,2,3,4,5,6,3,6,8
このときの正解率は 6/10 だった。

・ Impact



推論結果は順に 8,8,8,2,8,8,8,2,8,8
このときの正解率は 1/10 だった。

・ Bradhite



推論結果は順に 0,5,2,3,4,5,6,7,8,9
このときの正解率は 9/10 だった。

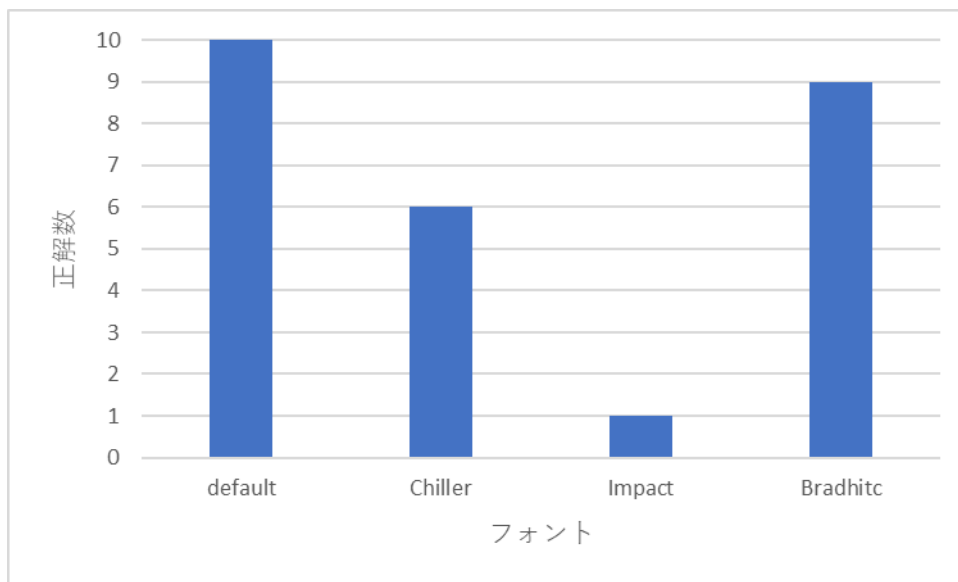


図 6 フォントによる正解率

図 6 にフォントによる正解率のグラフを示した。これらの分析結果からわかるのは基本的な Windows のフォントは認識が上手くいったが、これはクセがなく文字の太さや形状が訓練データから得られたパラメータに近いものだったからだといえる。つまり、訓練データにはいろいろな数字の書き方があるものの、それらの文字は平均すると全体としてデフォルトフォントに近づいていくような文字であると考えられる。そういった特徴をもつ訓練データを用いて学習していたからこそ標準的な文字をうまく認識することが出来たと考えられる。一方、Impact という標準的な数字とは形状がかなり異なり、文字の太さが太いという特徴があるようなフォントではほぼ 8 という認識結果を返した。これはフォント自体の全体に対する占有面積が大きいためには起きていると考えられる。すなわち、訓練データの数字は全体に占める数字の面積が大きいときに、8 を返すようにパラメータが学習されているためにこういう結果になっているのではないかと考えられた。次に用いた Chiller というフォントはマジックで描いたようなフォントであり比較的正解率が高く出たものの、Bradley Hand ITC font という万年筆でかいたような手書き文字のフォントでは高い正解率を示した。

これらのことから訓練データが手書き文字セットだったことが影響して、他の手書き文字、特にペン先が細いペンを用いて書かれたようなフォントが高い精度で推論できるようなパラメータを学習できたと結論付けられる。

参考文献

[1] Batch size をどうやって決めるかについてまとめる

<https://www.st-hakky-blog.com/entry/2017/11/16/161805>

[2] C 言語による乱数生成

https://omitakahiro.github.io/random/random_variables_generation.html

[3] ボックス＝ミュラー法 Wikipedia

<https://ja.wikipedia.org/wiki/%E3%83%9C%E3%83%83%E3%82%AF%E3%82%B9%E3%82%9F%E3%83%9F%E3%83%A5%E3%83%A9%E3%83%BC%E6%B3%95#:~:text=%E3%83%9C%E3%83%83%E3%82%AF%E3%82%B9%E3%82%9F%E3%83%A5%E3%83%A9%E3%83%BC%E6%B3%95%E3%82%88%E3%83%9C%E3%83%83%E3%82%AF%E3%82%B9,%E7%99%BA%E7%94%9F%E3%81%AB%E5%BF%9C%E7%94%A8%E3%81%95%E3%82%8C%E3%82%8B%E3%80%82>

[4] 斎藤 康毅: ゼロから作る Deep Learning -Python で学ぶディープラーニングの理論と実装, オライリージャパン, 2016