計算機科学実験4 画像処理
課題4・発展課題 レポート

1029311501 坂井優斗

提出日：2022年1月24日

1 課題 4

1.1 課題内容

MNIST のテスト画像 1 枚を入力として，3 層ニューラルネットワークを用いて，0 9 の値のうち, どの値が書かれているかを識別して出力するプログラムを作成する。プログラムの仕様は以下の通りである。

- ニューラルネットワークの重みパラメータ $W^{(1)}, W^{(2)}, \boldsymbol{b}^{(1)}, \boldsymbol{b}^{(2)}$ は課題 3 で学習した重みを利用する。
- その他の仕様は課題 1 と同じとする。

1.2 プログラムの説明

課題 4 は **task4.py** に実装されている。

Code 1: main 関数

```
def main():
  # 画像データの準備
  test_images = mnist.download_and_parse_mnist_file("t10k-images-idx3-ubyte.gz")
  test_labels = mnist.download_and_parse_mnist_file("t10k-labels-idx1-ubyte.gz")

  # パラメータの準備
  load_parameters("parameters_task3.npz")

  i = int(input('Input␣the␣number:␣'))
  image = test_images[i]
  processed_image = output_layer(inner_layer(input_layer(image)))
  prediction = np.argmax(processed_image)
  print(f"prediction:␣{prediction}")
  print(f"correct␣label:␣{test_labels[i]}")
  plt.imshow(test_images[i], cmap=cm.gray)
  plt.show()
```

基本の動作は課題 1 のプログラムと同じである。異なる点は 7 行目で load_parameters 関数が実行されているところである。これによって, 課題 3 で学習した重みを読み込んで利用できるようにしている。今回は, parameters_task3.npz というファイルに保存されたパラメータを利用している。重みパラメータを読み込んだあとは, 入力された番号の画像に書かれた数字を識別し, モデルによって識別された結果と, 真のラベルと, さらに該当する画像の 3 つを出力する。

1.3 実行結果

`python task4.py` でスクリプトを実行する。以下に画像の番号として 0 を入力したときの結果を例として示す。

```
$ python task4.py
  Input the number : 0
  prediction : 7
  correct label : 7
```

2　発展課題 A1

2.1　課題内容

活性化関数として ReLU 関数を実装してニューラルネットワークを構築する。

2.2　プログラムの説明

発展課題 A1 は **taskA1.py** に実装されている。
ReLU 関数は次式で定義される。

$$a(t) = \begin{cases} t & (t > 0) \\ 0 & (t \leq 0) \end{cases}$$

これをプログラム中で実装したのが次に示す ReLU_function である。

Code 2: ReLU_function

```
1   def ReLU_function(t):
2     global ReLU_mask
3     ReLU_mask = np.where(t>0, 1, 0)
4     return np.maximum(t, 0)
```

ReLU 関数は正の数はその値を, 0 以下の数は 0 を返す関数であるので, numpy の maximum 関数を利用して, 引数 t として入力された配列の maximum(t, 0) を返すように実装している。また, 3 行目で numpy の where 関数を利用して, 正の数であるところは 1, 0 以下であるところは 0 を埋めた配列 ReLU_mask を生成して保持している。これは後に ReLU 関数の微分を逆伝搬で計算するときに使う。

また, ReLU 関数の微分は次式のように計算される。

$$a'(t) = \begin{cases} 1 & (t > 0) \\ 0 & (t \leq 0) \end{cases}$$

従って, ReLU 関数の入力を $\boldsymbol{x} = (x_1, x_2, \ldots, x_n)$, 出力を $\boldsymbol{y} = (y_1, y_2, \ldots, y_n)$ とすると, クロスエントロピー誤差 E_n を $\boldsymbol{x}$ で微分した勾配 ∇E_n は次式で与えられる。

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

これを実装したのが calc_derivative_ReLU である。

Code 3: calc_derivative_ReLU

```
1   def calc_derivative_ReLU(derivative_y):
2     return derivative_y * ReLU_mask
```

引数として $\frac{\partial E_n}{\partial y}$ が渡されるので, それと ReLU_mask の要素ごとの積を返す。

最後に, これまでシグモイド関数を使用していたところを上で定義した関数に取り替えることで, 中間層の活性化関数を ReLU 関数にしたニューラルネットワークが構築できた。

2.3 実行結果

下の図 1 に 10 エポック学習させたときの損失値の推移を示す。活性化関数としてシグモイド関数を利用したときの損失を青色, ReLU 関数を利用したときの損失を橙色で描いている。

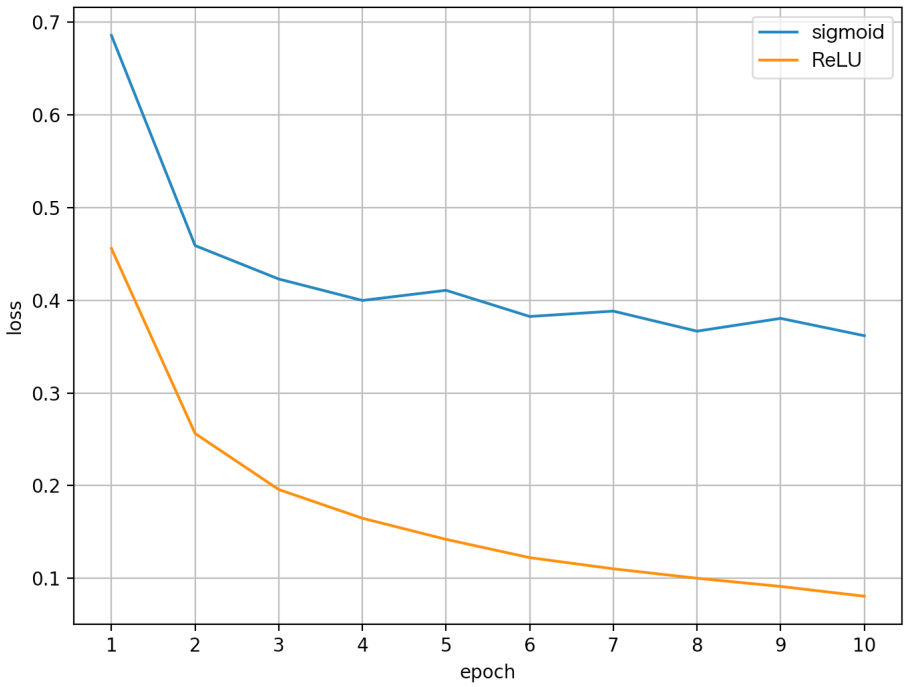

図 1: シグモイド関数と ReLU 関数による損失値の比較

図 1 を見ると, シグモイド関数の場合に比べて, ReLU 関数の場合の方が loss の落ちる速度が大きく, 加えてより小さい損失値まで落としていることがわかる。また, ReLU 関数を利用したときの各エポックごとの損失は以下のようになっている。

```
$ python taskA1.py
The loss in epoch1 is 0.45831989361167125.
The loss in epoch2 is 0.25697229483213585.
The loss in epoch3 is 0.20503483034133851.
The loss in epoch4 is 0.166003790112126.
The loss in epoch5 is 0.14295108062377038.
The loss in epoch6 is 0.12745600828935305.
The loss in epoch7 is 0.11665758228071545.
The loss in epoch8 is 0.10108669181540034.
The loss in epoch9 is 0.09536119086802915.
The loss in epoch10 is 0.08646191427193999.
```

3　発展課題 A2

3.1　課題内容

学習時に一定割合でランダムにノードを選び出力を無視することで, 過学習を防ぐ手法 Dropout を実装する。ノードの不活性化率 ρ は, 次のように定める。

- 入力層：$\rho_{input} = 0.2$
- 中間層：$\rho_{inner} = 0.5$

3.2　プログラムの説明

発展課題 A2 は **taskA2.py** に実装されている。
はじめに, 変数についての説明をまとめる。

- training_flag：学習とテストを判別するためのフラグ. 学習時は True, テスト時は False を示す。
- input_nonactive_ratio：入力層のノードの不活性化率 $\rho_{input}(= 0.2)$
- inner_nonactive_ratio：入力層のノードの不活性化率 $\rho_{inner}(= 0.5)$
- dropout_mask：活性ノードを True, 不活性ノードを False で表した配列

Dropout を関数として捉えると次式のように定義できる。

$$a(t) = \begin{cases} t & (\text{活性ノード}) \\ 0 & (\text{不活性ノード}) \end{cases}$$

これを実装したのが dropout_layer である。

Code 4: dropout_layer

```
1   def dropout_layer(x, nonactive_ratio):
2     if training_flag:
3       mask = np.random.rand(*x.shape) > nonactive_ratio
4       return x * mask, mask
5     else:
6       return (1-nonactive_ratio) * x
```

学習時は numpy の random 関数を利用して, 各ノードに擬似的に 0~1 の値を割り当てる。そして, その値が ρ より大きいノードは活性ノード, ρ 以下のノードは不活性ノードとみなすことで, ランダムにノードの活性化を実現している。なお, この部分でノードの活性, 不活性を True, False で表すマスクを生成しておく。
また, テスト時は学習時とは異なり, すべてのノードが活性化状態であるが, 出力が $(1-\rho)$ 倍となるので, training_flag が False であるときは入力 x に $(1-\rho)$ を掛けて返り値としている。

Dropout の関数表現の微分は次の式で与えられる。

$$a'(t) = \begin{cases} 1 & (\text{活性ノード}) \\ 0 & (\text{不活性ノード}) \end{cases}$$

従って, dropout 層の入力を $\boldsymbol{x} = (x_1, x_2, \ldots, x_n)$, 出力を $\boldsymbol{y} = (y_1, y_2, \ldots, y_n)$ とすると, クロスエントロピー誤差 E_n を $\boldsymbol{x}$ で微分した勾配 ∇E_n は次式で与えられる。

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

これは ReLU 関数の微分と同一であるので, calc_derivative_ReLU と同様に calc_derivative_dropout を実装した。

Code 5: calc_derivative_dropout

```
1  def calc_derivative_dropout(derivative_y):
2    return derivative_y * dropout_mask
```

最後に, 入力層では画像の形状変化, 画素値の正規化の後に dropout の処理を, 中間層では ReLU 関数に通したあとに dropout の処理を追加した。

3.3 実行結果

下の図 2 に 10 エポック学習させたときの損失値の推移を示す。Dropout 無しを青色, Dropout 有りを橙色で描いている。なお, 中間層の活性化関数はどちらも ReLU 関数である。

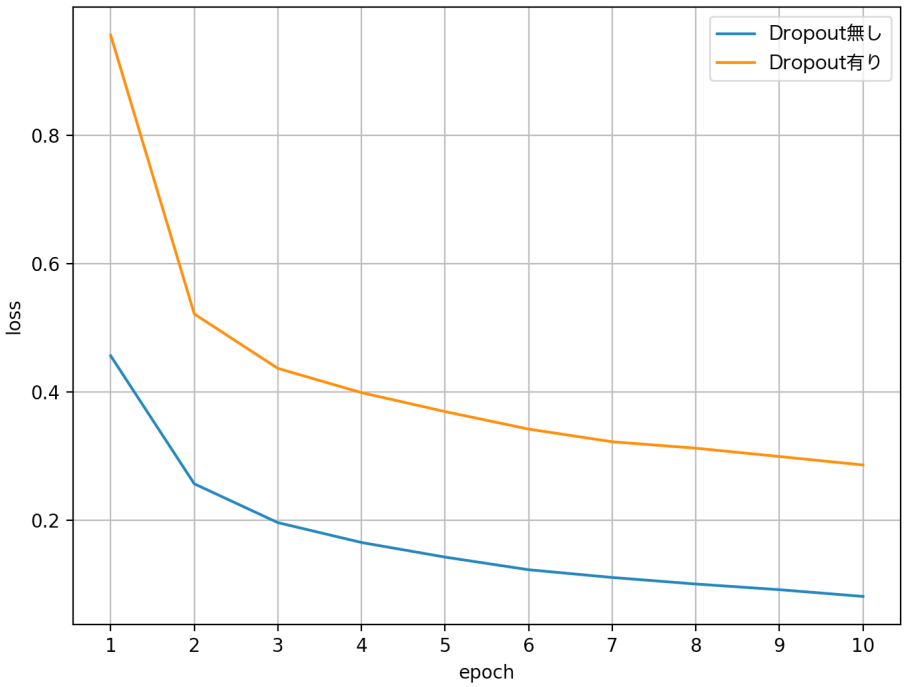

図 2: Dropout の有無による損失値の比較

図 2 を見ると, Dropout を入れたときのほうが無いときに比べて loss の落ちる速度が緩やかで, 学習終了時の loss も大きいことがわかる。汎化性能が高まったためにこのような結果になったと考えたが, テストデータに対する accuracy も Dropout 無しが 97.08%, Dropout 有りが 95.26%と,

Dropout 無しのほうが高かった。
また, 各エポックごとの損失は以下のようになっている。

```
$ python taskA2.py
The loss in epoch1 is 0.9713138300566404.
The loss in epoch2 is 0.521897431433509.
The loss in epoch3 is 0.44275020385478175.
The loss in epoch4 is 0.39382777992847146.
The loss in epoch5 is 0.36561734463654116.
The loss in epoch6 is 0.3518418752318544.
The loss in epoch7 is 0.32424740110556244.
The loss in epoch8 is 0.3099968254216254.
The loss in epoch9 is 0.29788842126892284.
The loss in epoch10 is 0.28765560854737243.
```

4 発展課題 A3

4.1 課題内容

ミニバッチに対して各ノードの出力を平均 0, 分散 1 に正規化する Batch Normalization を実装する。

4.2 プログラムの説明

発展課題 A3 は **taskA3.py** に実装されている。
ミニバッチサイズを B, ミニバッチデータに対するあるノードの出力を $x_i(i = 1, 2, \ldots, B)$ とする。
学習時, Batch Normalization は以下の式で x_i を y_i へと変換して出力する。

$$\mu_B \leftarrow \frac{1}{B}\sum_{i=1}^{B} x_i \tag{1}$$

$$\sigma_B^2 \leftarrow \frac{1}{B}\sum_{i=1}^{B} (x_i - \mu_B)^2 \tag{2}$$

$$\hat{x_i} \leftarrow \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}} \tag{3}$$

$$y_i \leftarrow \gamma \hat{x}_i + \beta \tag{4}$$

また, ミニバッチごとに計算される μ_B, σ_B^2 の期待値を $E[x_i], Var[x_i]$ とすると, テスト時は y_i を

$$y_i \leftarrow \frac{\gamma}{\sqrt{\mathrm{Var}\left[x_i\right] + \epsilon}} \cdot x_i + \left(\beta - \frac{\gamma E\left[x_i\right]}{\sqrt{\mathrm{Var}\left[x_i\right] + \epsilon}}\right) \tag{5}$$

で出力する。これを実装したのが次に示す batch_normalization である。

Code 6: batch_normalization

```
1   def batch_normalization(x):
2     global BN_x, BN_x_hat, ave_list, var_list
3     delta = 1e-7
4     if training_flag:
5       BN_x = x
6       ave = BN_x.mean(axis = 0)
7       ave_list = np.append(ave_list, ave)
8       var = BN_x.var(axis = 0)
9       var_list = np.append(var_list, var)
10      BN_x_hat = (BN_x - ave)/(np.sqrt(var + delta))
11      y = gamma * BN_x_hat + beta
12      return y
13    else:
14      ave = np.mean(ave_list)
15      var = np.mean(var_list)
16      return gamma * x / (np.sqrt(var+delta)) + beta - (gamma * ave / (np.sqrt(var+delta)
          ))
```

学習時は 5 行目から 11 行目が実行される。6 行目で (1) 式の計算を行う。numpy の mean 関数の引数 axis を 0 に指定することで各バッチの x_i の平均をとっている。また, 7 行目で今回のバッチの μ_B を記録するための変数 ave_list に ave($=\mu_B$) を格納する。次に 8 行目で (2) 式の計算を行う。μ_B の計算と同じように numpy の var 関数の引数 axis を 0 に指定することで各バッチの x_i の分散をとっている。また, 9 行目で今回のバッチの σ_B^2 を記録するための変数 var_list に var($=\sigma_B^2$) を格納する。そして, 10 行目で $\hat{x_i}$, 11 行目で y_i を計算している。
テスト時は 14 行目から 16 行目が実行される。14, 15 行目でバッチごとに求めて記録しておいた μ_B, σ_B^2 の期待値を計算し, 16 行目で (5) 式の計算をしている。

Batch Normalization の逆伝搬は以下の式で与えられる。

$$\frac{\partial E_n}{\partial \hat{x_i}} = \frac{\partial E_n}{\partial y_i} \cdot \gamma \tag{6}$$

$$\frac{\partial E_n}{\partial \sigma_B^2} = \sum_{i=1}^{B} \frac{\partial E_n}{\partial \hat{x_i}} \cdot (x_i - \mu_B) \cdot \frac{-1}{2} \left(\sigma_B^2 + \epsilon\right)^{-3/2} \tag{7}$$

$$\frac{\partial E_n}{\partial \mu_B} = \left(\sum_{i=1}^{B} \frac{\partial E_n}{\partial \hat{x_i}} \cdot \frac{-1}{\sqrt{\sigma_B^2 + \epsilon}} \right) + \frac{\partial E_n}{\partial \sigma_B^2} \cdot \frac{\sum_{i=1}^{B} -2\,(x_i - \mu_B)}{B} \tag{8}$$

$$\frac{\partial E_n}{\partial x_i} = \frac{\partial E_n}{\partial \hat{x_i}} \cdot \frac{1}{\sqrt{\sigma_B^2 + \epsilon} + \frac{\partial E_n}{\partial \sigma_B^2} \cdot \frac{2\,(x_i - \mu_B)}{B} + \frac{\partial E_n}{\partial \mu_B} \cdot \frac{1}{B}} \tag{9}$$

$$\frac{\partial E_n}{\partial \gamma} = \sum_{i=1}^{B} \frac{\partial E_n}{\partial y_i} \cdot \hat{x_i} \tag{10}$$

$$\frac{\partial E_n}{\partial \beta} = \sum_{i=1}^{B} \frac{\partial E_n}{\partial y_i} \tag{11}$$

これを実装したのが次に示す calc_derivative_BN である。

Code 7: calc_derivative_BN

```
1   def calc_derivative_BN(derivative_y):
2     delta = 1e-7
3     ave = ave_list[-1]
4     var = var_list[-1]
```

```
5       derivative_x_hat = derivative_y * gamma
6       derivative_var = np.sum((derivative_x_hat * (BN_x - ave) * (-1/2) * ((var + delta)
          **(-3/2))), axis=0)
7       derivative_ave = (np.sum((derivative_x_hat * (-1) * ((var + delta)**(-1/2))), axis=0)
          ) + (derivative_var * (1/batch_size) * (np.sum(((-2) * (BN_x - ave)), axis=0)))
8       derivative_x = (derivative_x_hat * ((var + delta)**(-1/2))) + (derivative_var * (2/
          batch_size) * (BN_x - ave)) + ((1/batch_size) * derivative_ave)
9       derivative_gamma = np.sum((derivative_y * BN_x_hat), axis=0)
10      derivative_beta = np.sum(derivative_y, axis=0)
11      return derivative_x, derivative_beta, derivative_gamma
```

1 行目で定義しているのは, 分母が 0 になるのを防ぐための微小値である。

2, 3 行目で最も直近のバッチに対する μ_B, σ_B^2 を求める。そして 5 行目で上の (6) 式 $\dfrac{\partial E_n}{\partial \hat{x}_i}$, 6 行目で (7) 式 $\dfrac{\partial E_n}{\partial \sigma_B^2}$, 7 行目で (8) 式 $\dfrac{\partial E_n}{\partial \mu_B}$, 8 行目で (9) 式 $\dfrac{\partial E_n}{\partial x_i}$, 9 行目で (10) 式 $\dfrac{\partial E_n}{\partial \gamma}$, 10 行目で (11) 式 $\dfrac{\partial E_n}{\partial \beta}$ を計算している。

最後に, 中間層でアフィン変換を行って, ReLU 関数に通す処理の間に Batch Normalization を挟むように変更した。

4.3 実行結果

下の図 3 に 10 エポック学習させたときの損失値の推移を示す。Batch Normalization 無しを青色, Batch Normalization 有りを橙色で描いている。なお, 中間層の活性化関数はどちらも ReLU 関数である。

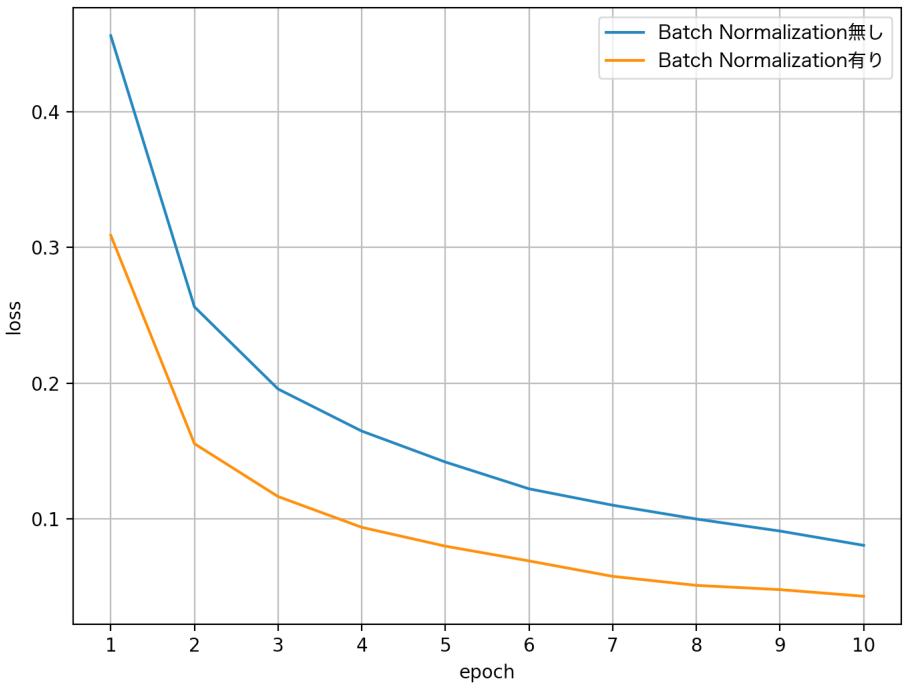

図 3: Batch Normalizaiton の有無による損失値の比較

図 3 を見ると, Batch Normalization によって loss の値がより早く小さい値へ落ちていることがわかる。しかし, テストデータに対する accuracy は Batch Normalization の処理を入れたときが 91.45%に対して, Batch Normalization の処理を行わないときは 97.08%という結果になった。
また, 各エポックごとの損失は以下のようになっている。

```
$ python taskA3.py
The loss in epoch1 is 0.3027437971918727.
The loss in epoch2 is 0.15926088412230485.
The loss in epoch3 is 0.11846851009444552.
The loss in epoch4 is 0.09574224918045819.
The loss in epoch5 is 0.08357365221852066.
The loss in epoch6 is 0.07148749235518706.
The loss in epoch7 is 0.06272362046586331.
The loss in epoch8 is 0.056778851376282774.
The loss in epoch9 is 0.0499809040628355.
The loss in epoch10 is 0.043804498986508 99.
```

5 発展課題 A4

5.1 課題内容

重みパラメータの学習手法の最適化を行う。最適化手法は以下の 5 つを実装する。

1. 慣性項付き SGD
2. AdaGrad
3. RMSProp
4. AdaDelta
5. Adam

5.2 プログラムの説明

5.2.1 慣性項付き SGD

慣性項付き SGD では, 従来の SGD におけるパラメータ W の更新式に慣性項を加えて定義される。

$$\Delta W \leftarrow \alpha \Delta W - \eta \frac{\partial E_n}{\partial W} \tag{12}$$

$$W \leftarrow W + \Delta W \tag{13}$$

なお, パラメータ α, η は $\alpha = 0.9, \eta = 0.01$ に設定し, ΔW の初期値は 0 としている。

慣性項付き SGD による学習は **taskA4_momentum.py** に実装されていて, 該当部分のコードが次のようになっている。

Code 8: Momentum SGD

```
1   delta_W1 = alpha * delta_W1 - Momentum_SGD_lr * derivative_W_1
2   delta_W2 = alpha * delta_W2 - Momentum_SGD_lr * derivative_W_2
3   delta_b1 = alpha * delta_b1 - Momentum_SGD_lr * derivative_b_1
4   delta_b2 = alpha * delta_b2 - Momentum_SGD_lr * derivative_b_2
5   parameters["W_1"] += delta_W1
6   parameters["W_2"] += delta_W2
7   parameters["b_1"] += delta_b1
8   parameters["b_2"] += delta_b2]
```

ソースコードにおける alpha が 更新式のパラメータ α, Momentum_SGD_lr がパラメータ η を意味する。1~4 行目で (12) 式に当たる ΔW を計算し, 5~8 行目で (13) 式に当たる W の更新を行っている。

5.2.2 AdaGrad

AdaGrad は学習率が自動的に小さくなるよう調整される学習手法であり, 次式で定義される。

$$h \leftarrow h + \frac{\partial E_n}{\partial W} \circ \frac{\partial E_n}{\partial W} \tag{14}$$

$$W \leftarrow W - \eta \frac{1}{\sqrt{h}} \frac{\partial E_n}{\partial W} \tag{15}$$

なお, $\circ$ はアダマール積を意味し, パラメータ η は, $\eta = 0.01$, h の初期値 h_0 は $h_0 = 10^{-8}$ としている。

AdaGrad による学習は **taskA4_AdaGrad.py** に実装されていて, 該当部分のコードが次のようになっている。

Code 9: AdaGrad

```
1   h_W1 += derivative_W_1 * derivative_W_1
2   h_W2 += derivative_W_2 * derivative_W_2
3   h_b1 += derivative_b_1 * derivative_b_1
4   h_b2 += derivative_b_2 * derivative_b_2
5   parameters["W_1"] -= AdaGrad_lr * derivative_W_1 / np.sqrt(h_W1)
6   parameters["W_2"] -= AdaGrad_lr * derivative_W_2 / np.sqrt(h_W2)
7   parameters["b_1"] -= AdaGrad_lr * derivative_b_1 / np.sqrt(h_b1)
8   parameters["b_2"] -= AdaGrad_lr * derivative_b_2 / np.sqrt(h_b2)
```

ソースコードにおける AdaGrad_lr が η を意味する。1~4 行目で (14) 式に当たる h を計算し, 5~8 行目で (15) 式に当たる W の更新を行っている。

5.2.3 RMSProp

RMSProp は AdaGrad の改良版である。AdaGrad では h の更新式を 1 つ前の h に勾配の 2 乗を足すことで定義していたのに対して, RMSProp では 1 つ前の h と勾配の 2 乗の指数移動平均をとることで更新する手法で, 次式で定義される。

$$h \leftarrow \rho h + (1-\rho) \frac{\partial E_n}{\partial W} \circ \frac{\partial E_n}{\partial W} \tag{16}$$

$$W \leftarrow W - \eta \frac{1}{\sqrt{h} + \epsilon} \frac{\partial E_n}{\partial W} \tag{17}$$

なお, パラメータ η, ρ, ϵ は, $\eta = 0.01, \rho = 0.9, \epsilon = 10^{-8}$, h の初期値 h_0 は $h_0 = 0$ としている。

RMSProp による学習は **taskA4_RMSProp.py** に実装されていて, 該当部分のコードが次のようになっている。

Code 10: RMSProp

```
1  h_W1 = rho * h_W1 + (1-rho) * derivative_W_1 * derivative_W_1
2  h_W2 = rho * h_W2 + (1-rho) * derivative_W_2 * derivative_W_2
3  h_b1 = rho * h_b1 + (1-rho) * derivative_b_1 * derivative_b_1
4  h_b2 = rho * h_b2 + (1-rho) * derivative_b_2 * derivative_b_2
5  parameters["W_1"] -= RMSProp_lr * derivative_W_1 / (np.sqrt(h_W1) + epsilon)
6  parameters["W_2"] -= RMSProp_lr * derivative_W_2 / (np.sqrt(h_W2) + epsilon)
7  parameters["b_1"] -= RMSProp_lr * derivative_b_1 / (np.sqrt(h_b1) + epsilon)
8  parameters["b_2"] -= RMSProp_lr * derivative_b_2 / (np.sqrt(h_b2) + epsilon)
```

ソースコードにおける RMSProp_lr がパラメータ η を, rho, epsilon がパラメータ ρ, ϵ を意味する。1~4 行目で (16) 式に当たる h を計算し, 5~8 行目で (17) 式に当たる W の更新を行っている。

5.2.4 AdaDelta

AdaDelta は, RMSProp や AdaGrad の改良版である。RMSProp, AdaGrad とは異なり, 学習率の初期値を定めることが不要であるのがポイントである。AdaDelta の更新式は次式で定義される。

$$h \leftarrow \rho h + (1-\rho)\frac{\partial E_n}{\partial W} \circ \frac{\partial E_n}{\partial W} \tag{18}$$

$$\Delta W \leftarrow -\frac{\sqrt{s+\epsilon}}{\sqrt{h+\epsilon}}\frac{\partial E_n}{\partial W} \tag{19}$$

$$s \leftarrow \rho s + (1-\rho)\Delta W \circ \Delta W \tag{20}$$

$$W \leftarrow W + \Delta W \tag{21}$$

なお, パラメータ ρ, ϵ は, $\rho = 0.95, \epsilon = 10^{-6}$, h, s の初期値 h_0, s_0 は $h_0 = 0, s_0 = 0$ としている。

AdaDelta による学習は **taskA4_AdaDelta.py** に実装されていて, 該当部分のコードが次のようになっている。

Code 11: AdaDelta

```
1  h_W1 = rho * h_W1 + (1-rho) * derivative_W_1 * derivative_W_1
2  h_W2 = rho * h_W2 + (1-rho) * derivative_W_2 * derivative_W_2
3  h_b1 = rho * h_b1 + (1-rho) * derivative_b_1 * derivative_b_1
4  h_b2 = rho * h_b2 + (1-rho) * derivative_b_2 * derivative_b_2
5  delta_W1 = -np.sqrt((s_W1 + epsilon)/(h_W1 + epsilon)) * derivative_W_1
6  delta_W2 = -np.sqrt((s_W2 + epsilon)/(h_W2 + epsilon)) * derivative_W_2
7  delta_b1 = -np.sqrt((s_b1 + epsilon)/(h_b1 + epsilon)) * derivative_b_1
8  delta_b2 = -np.sqrt((s_b2 + epsilon)/(h_b2 + epsilon)) * derivative_b_2
9  s_h1 = rho * s_W1 + (1-rho) * delta_W1 * delta_W1
10 s_h2 = rho * s_W2 + (1-rho) * delta_W2 * delta_W2
11 s_b1 = rho * s_b1 + (1-rho) * delta_b1 * delta_b1
12 s_b2 = rho * s_b2 + (1-rho) * delta_b2 * delta_b2
13 parameters["W_1"] += delta_W1
14 parameters["W_2"] += delta_W2
15 parameters["b_1"] += delta_b1
16 parameters["b_2"] += delta_b2
```

ソースコードにおける rho, epsilon がパラメータ ρ, ϵ を意味する。1~4 行目で (18) 式に当たる h を, 5~8 行目で (19) 式に当たる ΔW を, そして 9~12 行目で (20) 式に当たる s を計算している。最後に, 13~16 行目で (21) 式に当たる W の更新を行っている。

5.2.5 Adam

Adam は, AdaDelta の改良版である。2 つの慣性項を用いて更新が行われる手法で, 次式で定義される。

$$t \leftarrow t + 1 \tag{22}$$

$$m \leftarrow \beta_1 m + (1 - \beta_1) \frac{\partial E_n}{\partial W} \tag{23}$$

$$v \leftarrow \beta_2 v + (1 - \beta_2) \frac{\partial E_n}{\partial W} \circ \frac{\partial E_n}{\partial W} \tag{24}$$

$$\hat{m} \leftarrow \frac{m}{1 - \beta_1^t} \tag{25}$$

$$\hat{v} \leftarrow \frac{v}{1 - \beta_2^t} \tag{26}$$

$$W \leftarrow W - \frac{\alpha \hat{m}}{\sqrt{\hat{v}} + \epsilon} \tag{27}$$

なお, パラメータ $\alpha, \beta_1, \beta_2, \epsilon$ は, $\alpha = 0.001, \beta_1 = 0.9, \beta_2 = 0.999, \epsilon = 10^{-8}$ に設定して, t, m, v の初期値はすべて 0 としている。

Adam による学習は **taskA4_Adam.py** に実装されていて, 該当部分のコードが次のようになっている。

Code 12: Adam

```
1   t += 1
2   m_W1 = beta_1 * m_W1 + (1-beta_1) * derivative_W_1
3   m_W2 = beta_1 * m_W2 + (1-beta_1) * derivative_W_2
4   m_b1 = beta_1 * m_b1 + (1-beta_1) * derivative_b_1
5   m_b2 = beta_1 * m_b2 + (1-beta_1) * derivative_b_2
6   v_W1 = beta_2 * v_W1 + (1-beta_2) * derivative_W_1 * derivative_W_1
7   v_W2 = beta_2 * v_W2 + (1-beta_2) * derivative_W_2 * derivative_W_2
8   v_b1 = beta_2 * v_b1 + (1-beta_2) * derivative_b_1 * derivative_b_1
9   v_b2 = beta_2 * v_b2 + (1-beta_2) * derivative_b_2 * derivative_b_2
10  m_W1_hat = m_W1/(1-beta_1**t)
11  m_W2_hat = m_W2/(1-beta_1**t)
12  m_b1_hat = m_b1/(1-beta_1**t)
13  m_b2_hat = m_b2/(1-beta_1**t)
14  v_W1_hat = v_W1/(1-beta_2**t)
15  v_W2_hat = v_W2/(1-beta_2**t)
16  v_b1_hat = v_b1/(1-beta_2**t)
17  v_b2_hat = v_b2/(1-beta_2**t)
18  parameters["W_1"] -= alpha * m_W1_hat / (np.sqrt(v_W1_hat) + epsilon)
19  parameters["W_2"] -= alpha * m_W2_hat / (np.sqrt(v_W2_hat) + epsilon)
20  parameters["b_1"] -= alpha * m_b1_hat / (np.sqrt(v_b1_hat) + epsilon)
21  parameters["b_2"] -= alpha * m_b2_hat / (np.sqrt(v_b2_hat) + epsilon)
```

ソースコードにおける alpha, beta_1, beta_2, epsilon がパラメータ $\alpha, \beta_1, \beta_2, \epsilon$ を意味する。
はじめに 1 行目で (22) 式の t のインクリメントを行っている。次に, 2~5, 6~9 行目で (23), (24) 式に当たる m と v の計算をしている。そして, 10~13, 14~17 行目で (25), (26) 式の $\hat{m}$ と $\hat{v}$ の計算を行う。最後に, 18~21 行目で (27) 式に当たる W の更新を行っている。

5.3 実行結果

通常の学習アルゴリズム SGD に加えて, 発展課題 A4 で実装した 5 つの最適化手法による 10 エポックの学習での損失値の推移を下の図 4 に示す。

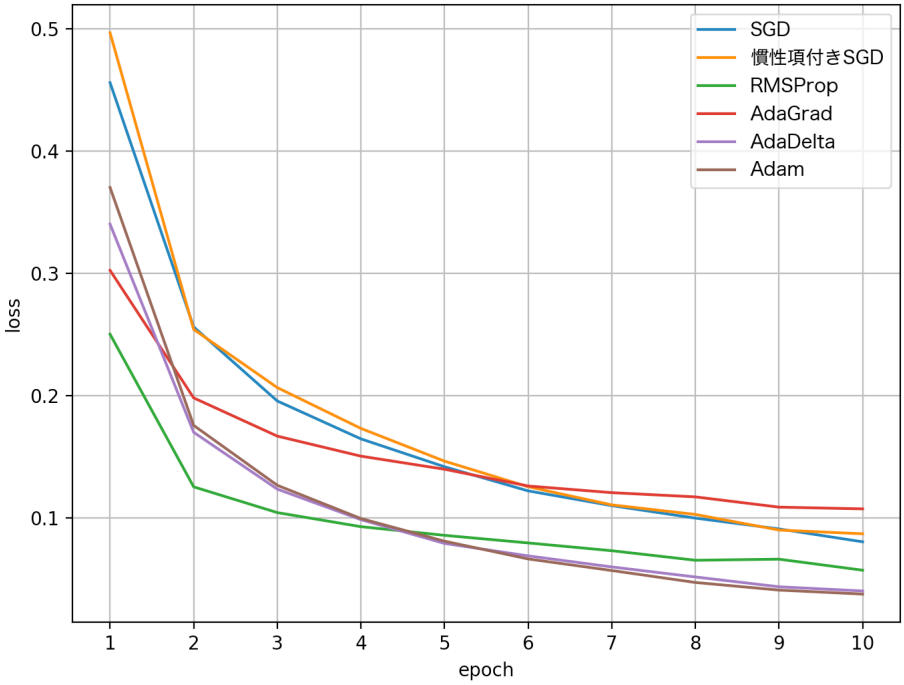

図 4: SGD と様々な最適化手法による学習の損失値の比較

また, 10 エポック学習後のパラメータを利用してのモデルで MNIST のテストデータ 10000 件を識別すると accuracy は次のようになった。

- SGD → 97.08%
- 慣性項付き SGD → 97.02%
- AdaGrad → 96.54%
- RMSProp → 97.15%
- AdaDelta → 97.67%
- Adam → 97.64%

上の結果より, 損失値の推移, accuracy どちらの観点から見ても 6 手法の中では AdaDelta と Adam の 2 つが他の 4 つに比べて優れていることがわかる。とはいえ, 他の 4 つの手法が全く駄目な訳ではなく, 最終的な accuracy の間にも 1%もないほどのわずかな差であった。学習データによっては, これらの手法の間に顕著な差が生まれるのかもしれない。

最後に, 発展課題 A4 で実装した 5 手法による 10 エポックの学習の損失値を下にまとめておく。

慣性項付き SGD

```
$ python taskA4_momentum.py
The loss in epoch1 is 0.4967264184355564.
```

```
The loss in epoch2 is 0.2543247047140201.
The loss in epoch3 is 0.20675356319296445.
The loss in epoch4 is 0.17339135496673308.
The loss in epoch5 is 0.1466429924946291.
The loss in epoch6 is 0.1257278908476439.
The loss in epoch7 is 0.11090184616481391.
The loss in epoch8 is 0.10309014551841046.
The loss in epoch9 is 0.0903342288724449.
The loss in epoch10 is 0.08727391506332602.
```

AdaGrad

```
$ python taskA4_AdaGrad.py
The loss in epoch1 is 0.30257788817526043.
The loss in epoch2 is 0.19833257041554195.
The loss in epoch3 is 0.1670942934312417.
The loss in epoch4 is 0.1507246349409675.
The loss in epoch5 is 0.14002715110296637.
The loss in epoch6 is 0.12633951232338275.
The loss in epoch7 is 0.12089119764282599.
The loss in epoch8 is 0.11743728843634367.
The loss in epoch9 is 0.10905445378389973.
The loss in epoch10 is 0.10764741996928061.
```

RMSProp

```
$ python taskA4_RMSProp.py
The loss in epoch1 is 0.25038696281178535.
The loss in epoch2 is 0.12563720623270544.
The loss in epoch3 is 0.10459468389615117.
The loss in epoch4 is 0.09309341354053348.
The loss in epoch5 is 0.0860103114891031.
The loss in epoch6 is 0.07979857537479187.
The loss in epoch7 is 0.07344573192631004.
The loss in epoch8 is 0.06563711179470264.
The loss in epoch9 is 0.06651788005460688.
The loss in epoch10 is 0.05751099149810582.
```

AdaDelta

```
$ python taskA4_AdaDelta.py
The loss in epoch1 is 0.34033926030150974.
```

```
The loss in epoch2 is 0.17022704189598553.
The loss in epoch3 is 0.12378065352145504.
The loss in epoch4 is 0.0988294408749616.
The loss in epoch5 is 0.07938981076234958.
The loss in epoch6 is 0.06922675227642665.
The loss in epoch7 is 0.06018709170340347.
The loss in epoch8 is 0.051975593566414104.
The loss in epoch9 is 0.04395829980545176.
The loss in epoch10 is 0.04053391058399345.
```

Adam

```
$ python taskA4_Adam.py
The loss in epoch1 is 0.3703141260333902.
The loss in epoch2 is 0.17586503475128465.
The loss in epoch3 is 0.12691378483829052.
The loss in epoch4 is 0.09974245141426769.
The loss in epoch5 is 0.08123205623441054.
The loss in epoch6 is 0.06671134073683999.
The loss in epoch7 is 0.05724521713296505.
The loss in epoch8 is 0.047473480413203525.
The loss in epoch9 is 0.04125423933213625.
The loss in epoch10 is 0.03798538252785112.
```

6 工夫点

- ニューラルネットワークにおける各機能をソースコード内で関数として切り分けて定義することでプログラムの可読性を高めた。また, それに伴いプログラムの拡張や変更もやりやすくなっている。
- バッチの処理を行うときに, 各画像を 1 枚ずつニューラルネットワークに通すのではなく, 行列にまとめてから演算を行うようなプログラムにすることで処理の高速化を行った。
- パラメータの学習プログラムを実行するときに外部ファイルからパラメータをロードするかどうかと, 外部ファイルに学習したパラメータを保存するかどうかを、コンソールから指定できるようにした。また, どちらも外部ファイルを使用する際はファイル名を入力可能にすることで利便性を高めた。
- 画素値 $0 \sim 255$ のままデータを入力すると e の冪乗を計算するときに overflow を起こしてしまうので, はじめに画素値を $0 \sim 1$ に正規化することで overflow を防ぐようにした。
- 各エポック終了時のクロスエントロピー誤差を記録しておき, 学習後に記録した損失の値の推移をグラフに表示するようにした。そうすることで、発展課題で様々な手法を実装するときに, どの手法がどれくらいモデルに影響を与えるのかを視覚的に捉えやすくなった。

- tqdm というライブラリを利用して, 学習状況をプログレスバーで表示するというビジュアル方面での工夫を行った。現在どの程度学習を終えているのかや, どれくらいの時間で学習を終えたかが一目でわかるようになった。その様子を以下の図 5 に示す。

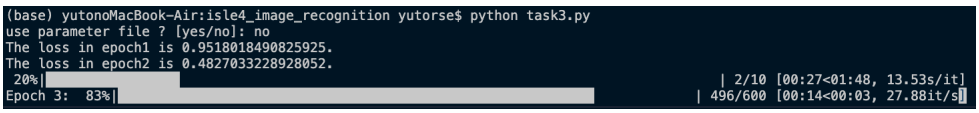

図 5: tqdm によるプログレスバー

7 問題点, 反省点

- 発展課題で実装した機能をすべてまとめたのが **taskA.py** なのだが, ニューラルネットワークの構造を変更するときにプログラムを書き換える必要が存在していて, 必要以上に手間がかかってしまう。例えば, Batch Normalization を無効にするには該当部分のコードを消す, ないしはコメントアウトして実行させないようにしなくてはならない。
  → 汎用性を高めるためにプログラム実行時にコンソールからモデルを指定できるように改善する余地がある。

- CNN の実装までたどり着けなかった。扱っているデータが 2 次元画像であるので CNN の実装で精度はかなり上がるのではないかと考えていたが間に合わなかった。

- 最終デモで話に上がっていたが、学習用データの形状を変換するという発想ができなかった。