計算機科学実験及演習 4 画像処理 最終レポート

工学部 情報学科 計算機科学コース
学生番号: 1029358455　　氏名: 登古紘平

2025 年 11 月 21 日

1　課題 3 以前で実装したプログラムの説明

課題 3 までの過程で、3 層ニューラルネットワークを構築し、順伝播、逆伝播、およびクロスエントロピー誤差、正答率計算を実装した。以下に示すのは、それらのコードとその説明である。

```
# ...（データ読み込みと初期化設定）

# --- データ準備とユーティリティ関数 ---
def get_shaffled_index():
    index = np.arange(len(train_images))
    np.random.shuffle(index)
    return index

def get_batch(random_index):
    batch_images = train_images[random_index].reshape(len(random_index), -1)
    batch_labels = train_labels[random_index]
    return batch_images, batch_labels

def get_one_hot_label(batch_labels, output_layer_size):
    one_hot_labels = np.zeros((batch_labels.size, output_layer_size))
    one_hot_labels[np.arange(batch_labels.size), batch_labels] = 1
    return one_hot_labels

# --- 活性化関数と出力関数 ---
def sigmoid(x):
    """シグモイド活性化関数"""
    return 1 / (1 + np.exp(-x))

def softmax(x):
    """ソフトマックス関数"""
    alpha = np.max(x, axis=-1, keepdims=True)
    exp_x = np.exp(x - alpha)
    return exp_x / np.sum(exp_x, axis=-1, keepdims=True)

# --- 順伝播と評価 ---
def forward_propagation(input_vector, weight1, bias1, weight2, bias2):
    hidden_layer_input = np.dot(input_vector, weight1.T) + bias1
    hidden_layer_output = sigmoid(hidden_layer_input)
```

```
    output_layer_input = np.dot(hidden_layer_output, weight2.T) + bias2
    final_output = softmax(output_layer_input)
    return final_output, hidden_layer_output

def get_predicted_class(output_probabilities):
    if output_probabilities.ndim == 1:
        return np.argmax(output_probabilities)
    else:
        return np.argmax(output_probabilities, axis=1)

def get_accuracy(y_prop, y_true):
    y_pred = get_predicted_class(y_prop)
    accuracy = np.sum(y_pred == y_true) / len(y_prop)
    return accuracy

def get_cross_entropy_error(y_pred, y_true):
    delta = 1e-7
    loss = -np.sum(y_true * np.log(y_pred + delta))
    batch_size = y_pred.shape[0]
    return loss / batch_size

# --- 逆伝播と更新 ---
def backward_propagation_and_update(batch_image_vector, hidden_layer_output,
output_probabilities, one_hot_labels, weight1, bias1, weight2, bias2, learning_rate):
    """逆伝播法を用いて勾配を計算し、全パラメータを更新する関数。"""
    current_batch_size = batch_image_vector.shape[0]

    # 1. 出力層の誤差伝播 (softmax + cross entropy)
    dEn_dak = (output_probabilities - one_hot_labels) / current_batch_size
    # 2. 第 2 層（中間層 -> 出力層）の勾配
    dEn_dW_1 = np.dot(dEn_dak.T, hidden_layer_output)
    dEn_db_1 = np.sum(dEn_dak, axis = 0)

    # 3. 中間層の誤差伝播（シグモイドの微分）
    dEn_dX = np.dot(dEn_dak, weight2)
    dEn_dX_sig = dEn_dX * (hidden_layer_output * (1 - hidden_layer_output))

    # 4. 第 1 層（入力層 -> 中間層）の勾配
    dEn_dW_2= np.dot(dEn_dX_sig.T, batch_image_vector)
    dEn_db_2 = np.sum(dEn_dX_sig, axis=0)
```

```
# 5. パラメータ更新
weight1 -= dEn_dW_2 * learning_rate
bias1   -= dEn_db_2 * learning_rate
weight2 -= dEn_dW_1 * learning_rate
bias2   -= dEn_db_1 * learning_rate

return weight1, bias1, weight2, bias2
```

1.1 各関数の説明

1.1.1 データ処理・ユーティリティ

- `get_shaffled_index` 関数: 訓練データのインデックス（0 から 59999）を生成し、ミニバッチ学習のためにランダムにシャッフルする。これにより、重複なくデータを学習及びテストに用いることができる。
- `get_batch` 関数: シャッフルされたインデックスの配列に基づき、対応する画像データとラベルのミニバッチを取得する。画像データの形状は (バッチサイズ, 28, 28) であり、reshape により、画像データは（バッチサイズ, 784）の形状に変換される。-1 で残りの次元を自動的に計算している。
- `get_one_hot_label` 関数: np.zeros((batch_labels.size, output_layer_size)) で、出力層の数 (10 個) 分の長さの配列がバッチサイズ分さらに配列をなし、それがすべて 0 で埋められたものを取得する。次に、正解ラベル番目の値を 1 に変換する。これにより、真のラベルを、出力層の次元に対応した One-Hot ベクトルに変換する。

1.1.2 活性化関数と出力関数

- `sigmoid` 関数: 中間層に適用されるシグモイド関数 $\sigma(x) = 1/(1 + e^{-x})$ を定義する。
- `softmax` 関数: 出力層に適用されるソフトマックス関数を定義する。

$$y_i = \frac{\exp(a_i - \alpha)}{\sum_{j=1}^{K} \exp(a_j - \alpha)}$$
$$\alpha = \max a_i$$

axis=-1 で、行方向の最大値の計算を表し、keepdims=True で、計算後の形状を、(バッチサイズ, 1) に保っている。

1.1.3 順伝播と評価

- `forward_propagation` 関数: 入力ベクトルを受け取り、全結合、活性化関数 (シグモイド)、全結合、出力 (ソフトマックス) と計算する流れを一つにした関数である。最終的に、出力確率と中間層の出力を返す。
- `get_predicted_class` 関数: ソフトマックス関数から得られた出力確率ベクトルのうち、最も値が大きい要素のインデックス（すなわち予測されたクラス）を返す。

- `get_accuracy` 関数: get_predicted_class 関数を作用させ得たインデックスと、真のラベルが一致した個数を、バッチサイズで割ることで正答率を計算する。
- `get_cross_entropy_error` 関数: $E = \sum_{k=1}^{C} -y_k log y_k^{(2)}$
  予測確率 $\mathbf{y}_{\text{pred}}$ と真のラベル $\mathbf{y}_{\text{true}}$ を用いて、損失関数であるクロスエントロピー誤差を計算する。対数の引数がゼロになるのを避けるため、微小値 $\delta = 1e-7$ を加算している。これらの和をバッチサイズで割り、その平均をとっている。

1.1.4 逆伝播とパラメータ更新

- `backward_propagation_and_update` 関数: クロスエントロピー誤差から開始し、連鎖律を用いて各層の重み $\mathbf{W}$ とバイアス $\mathbf{B}$ に対する勾配（偏微分）を計算する。特に、中間層の逆伝播ではシグモイド関数の微分 $y(1-y)$ を適用する。計算された勾配に学習率 η を乗じてパラメータを更新する。

2 4-1

課題内容

活性化関数としてシグモイド関数の他に次式で挙げる **ReLU** もよく用いられる.

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

ReLU の微分は,

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

で与えられる.

2.1 作成したプログラムの説明

このプログラムでは、活性化関数として、シグモイド関数の代わりに ReLU を用いる。これは、入力の値によって出力の仕方が変化するので、中間層に対する入力である hidden_layer_input を順伝播実行時に出力させ、それを逆伝播及び ReLU の微分に用いた。今回作成した ReLU 関数は以下の通りである。

```
def ReLU(arr):
    new_arr = np.where(arr > 0, arr, 0)
    return new_arr
```

また、逆伝播における微分を以下のように表した。

```
# dEn_dX_sig = dEn_dX * (hidden_layer_output * (1 - hidden_layer_output))
differentiated_input = np.where(hidden_layer_input > 0, 1, 0)
# ReLU に入力する hidden_input_layer の微分
dEn_dX_ReLU = dEn_dX * differentiated_input
```

np.where(arr $>$ 0, arr, 0) は、配列の要素が 0 より大きいかどうかを条件とし、これが真であればそのまま、偽であれば 0 を代入する。np.where(hidden_layer_input $>$ 0, 1, 0) は、ReLU の微分を表している。

2.2 実行結果

ロードしますか? yes or no: no

1 **エポック目**
平均クロスエントロピー誤差: 1.794267525319996
テストデータに対する正答率: 0.5322
学習データに対する正答率: 0.426833333333334
2 **エポック目**
平均クロスエントロピー誤差: 1.385299717909186
テストデータに対する正答率: 0.5804
学習データに対する正答率: 0.5111833333333332
...
10 **エポック目**
平均クロスエントロピー誤差: 0.7004218257765509
テストデータに対する正答率: 0.8188
学習データに対する正答率: 0.7939166666666663

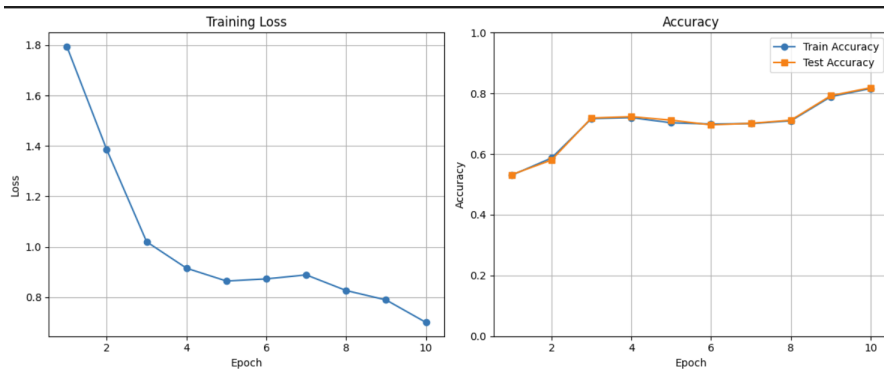

図 1　4-1 プロット結果 (ロード無し)

ロードしますか? yes or no: yes

1 エポック目

平均クロスエントロピー誤差：0.8011985461454941

テストデータに対する正答率：0.7324

学習データに対する正答率：0.7313666666666673

2 エポック目

平均クロスエントロピー誤差：0.7101978870361783

テストデータに対する正答率：0.7315

学習データに対する正答率：0.794383333333333

...

10 エポック目

平均クロスエントロピー誤差：0.9641420967176714

テストデータに対する正答率：0.6221

学習データに対する正答率：0.6602499999999994

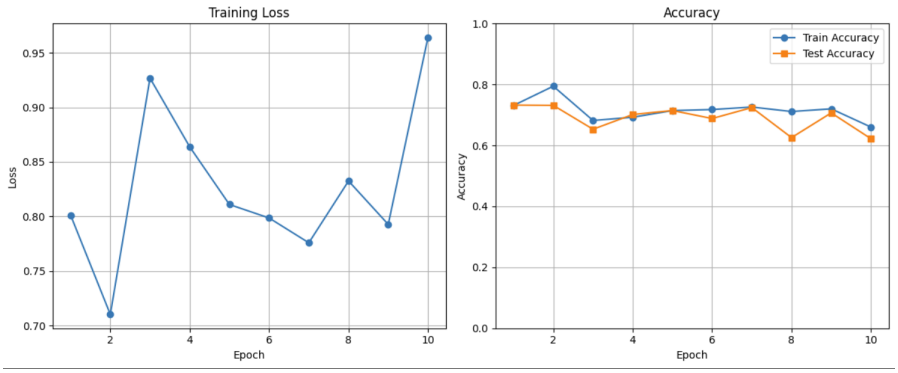

図 2　4-1 プロット結果 (ロード有り)

2.3　工夫点

中間層の活性化関数をシグモイド関数から ReLU に変更するにあたり、逆伝播時の勾配計算に必要な情報を正確に伝達する工夫を施した。具体的には、ReLU の微分が $t > 0$ で 1、$t \leq 0$ で 0 となる性質を利用するため、順伝播の際に ReLU への入力値 (`hidden_layer_input`) を保持し、戻り値として追加した。これにより、逆伝播において `hidden_layer_input` を基に、ReLU の正しい勾配を計算し、元のコードの主要な構造を維持したまま機能変更を達成することができた。

2.4　問題点と考察

プログラムの動作自体は確認できたが、実行結果には学習の効率と安定性に関して複数の問題点が観察された。

- **正答率の低下**: パラメータのロードの有無にかかわらず、ReLU 実装後の正答率が以前のシグモイド実装時と比較して低下する傾向が見られた。

- **学習の不安定化（ロード有り）**：既存のパラメータをロードして学習を再開した場合、平均クロスエントロピー誤差と正答率がエポック間で激しく上下し、結果が不安定になった。

これらの問題を踏まえ、今後は、学習の安定化のために学習率を調整するなどの工夫が必要であると感じた。

2.5　リファクタリング

リファクタリングには、Gemini(2.5 Flash) を用いた。

- プロンプト: (コード、プロット結果貼り付け) これは、３層ニューラルネットワークにおいて、活性化関数として ReLU を実装したものである。これに論理エラーはあるか
- 目的: 認識できていない論理エラーの検出、ReLU 関数への理解向上のため

結論として、ReLU 関数はうまく実装できており、正答率や平均クロスエントロピー誤差が不安定な理由として 4 つの理由が挙げられた。

- 学習率が大きすぎる可能性
- 重みの初期化が不適切（過度に大きい、小さいなど）
- バッチサイズとエポック数の影響
- 汎化性能の問題

これを踏まえ、学習したパラメータの初期化、学習率を 0.005 に、エポック数を 25 に変更した。その結果が以下である。

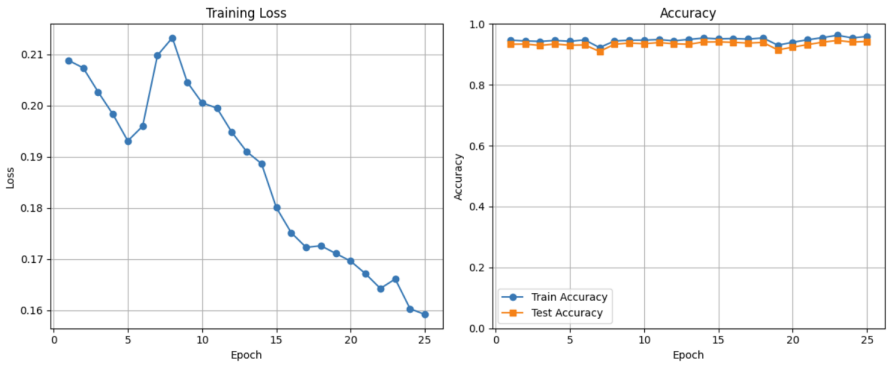

図 3　4-1 プロット結果 (ロード有り、修正後)

以上より、正答率、平均クロスエントロピー誤差ともに安定したことがわかる。修正前後で比較すると、1 エポック目の値に注目すると、ロードしていたパラメータに問題があったのではないかと考えられる。

3　4-2

課題内容

Dropout は学習時に中間層のノードをランダムに選び，その出力を無視（出力 $= 0$）して学習する手法である．無視するノードの選択は学習データ毎にランダムに行い，中間層全体のノード数 $\times \rho$ 個のノードの出力を

無視する.

テスト時は，全てのノードの出力を無視せず，代わりに元の出力に $(1-\rho)$ 倍したものを出力として用いる．このように，Dropout は学習時とテスト時で振る舞いが異なるので，学習時かテスト時かを判定するフラグを用意しておく必要がある．

Dropout を活性化関数の一種と考えると，

$$a(t) = \begin{cases} t & (\text{ノードが無視されない場合}) \\ 0 & (\text{ノードが無視された場合}) \end{cases} \tag{1}$$

となり，Dropout の微分は，

$$a(t)' = \begin{cases} 1 & (\text{ノードが無視されない場合}) \\ 0 & (\text{ノードが無視された場合}) \end{cases} \tag{2}$$

で与えられる．

3.1 作成したプログラムの説明

今回、訓練時とテスト時でプログラムの振る舞いが異なる。そのため、訓練時かテスト時かどうかを標準有力として受け取り、それを mode という変数に代入し、その値に応じて条件分岐させた。

```
mode = str(input('実行モードを入力してください (train or test): '))
if mode not in ['train', 'test']:
    print("無効なモードです。'train' または 'test' を入力してください。")
    exit()

ignore_number = int(input('Dropout の個数を 0 ~ 100 で入力してください: '))
if not (0 <= ignore_number <= hidden_layer_size):
    print("無効なドロップアウト数です。0 から 100 の範囲で入力してください。")
    exit()

# 訓練モードの場合にのみ学習を実行
if mode == 'train':
    print("\n--- 訓練モード実行中 ---")
...

    # テストモードの場合にのみ予測を実行
elif mode == 'test':
    print("\n--- テストモード実行中 ---")
...
```

また、今回新たに、Dropout を考慮した、訓練時、テスト時の順伝播、訓練時の逆伝播の 3 つの関数を、既存のコードをベースに作成した。

```
def forward_propagation_train(input_vector, weight1, bias1, weight2, bias2, ignore_number):
...
    for index in ignore_number:
        hidden_layer_output[:, index] = 0 # 無視する値を0に
...

def forward_propagation_test(input_vector, weight1, bias1, weight2, bias2, ignore_number):
    ...
    hidden_layer_output *= 1 - (len(ignore_number) / hidden_layer_size)
    ...

def backward_propagation_and_update_train(batch_image_vector, hidden_layer_input,
    hidden_layer_output, output_probabilities, one_hot_labels,  weight1, bias1, weight2,
    bias2, learning_rate, ignore_number):

    ...
    # dEn_dX_sig = dEn_dX * (hidden_layer_output * (1 - hidden_layer_output))
    differentiated_input = np.where(hidden_layer_input > 0, 1, 0)
    # ReLU に入力する hidden_input_layer の微分
    for index in ignore_number:
        dEn_dX[:, index] = 0
        differentiated_input[:, index] = 0
    dEn_dX_ReLU = dEn_dX * differentiated_input
    ...
```

以上の通り、順伝播において、訓練時は、ランダムに取得したインデックス配列を for 文で回し、各要素を 0 にしており、テスト時は、各要素の値を $(1-\rho)$ 倍にしている。逆伝播においても同様である。

3.2 実行結果

```
ロードしますか? yes or no: no
実行モードを入力してください (train or test): train
Dropout の個数を 0 ~ 100 で入力してください: 20

--- 訓練モード実行中 ---
1 エポック目
  平均クロスエントロピー誤差: 1.9865031046553099
  学習データに対する正答率: 0.3299000000000002
  テストデータに対する正答率: 0.4549
2 エポック目
```

平均クロスエントロピー誤差: 1.5910906225606625
学習データに対する正答率: 0.4241833333333337
テストデータに対する正答率: 0.5275
...
10 エポック目
平均クロスエントロピー誤差: 1.50288702684242
学習データに対する正答率: 0.42561666666666603
テストデータに対する正答率: 0.4531

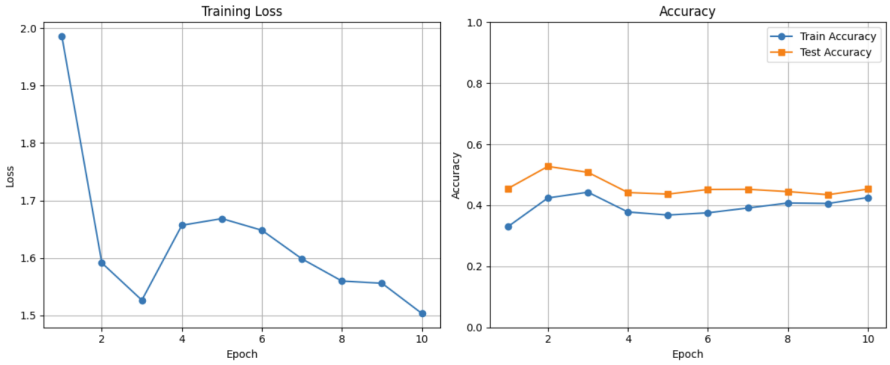

図 4　4-2 プロット結果 (ロード無し、Dropout = 20)

ロードしますか? yes or no: yes
実行モードを入力してください (train or test): train
Dropout の個数を 0 ~ 100 で入力してください: 20

--- 訓練モード実行中 ---
1 エポック目
平均クロスエントロピー誤差: 1.4786787110387787
学習データに対する正答率: 0.4321666666666662
テストデータに対する正答率: 0.4589
2 エポック目
平均クロスエントロピー誤差: 1.4163873100601898
学習データに対する正答率: 0.47068333333333345
テストデータに対する正答率: 0.5262
...
10 エポック目
平均クロスエントロピー誤差: 1.4476414952641528
学習データに対する正答率: 0.4366666666666658
テストデータに対する正答率: 0.4498

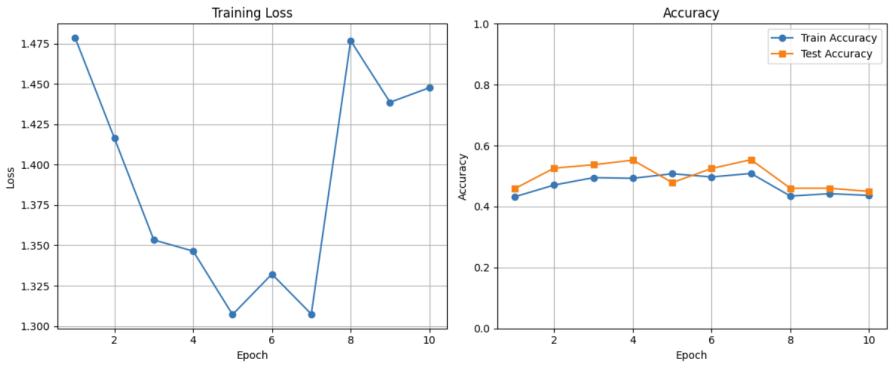

図 5　4-2 プロット結果 (ロード有り、Dropout = 20)

ロードしますか? yes or no: no
実行モードを入力してください (train or test): test
Dropout の個数を 0 ~ 100 で入力してください: 10

--- テストモード実行中 ---

テストデータに対する最終正答率: 0.1062
(ドロップアウト数 10 個)

ロードしますか? yes or no: no
実行モードを入力してください (train or test): test
Dropout の個数を 0 ~ 100 で入力してください: 80

--- テストモード実行中 ---

テストデータに対する最終正答率: 0.1057
(ドロップアウト数 80 個)

ロードしますか? yes or no: yes
実行モードを入力してください (train or test): test
Dropout の個数を 0 ~ 100 で入力してください: 10

--- テストモード実行中 ---

テストデータに対する最終正答率: 0.4505
(ドロップアウト数 10 個)

3.3 工夫点

訓練時かテスト時かどうか判定するフラグを標準入力で受け取り、その文字列を関数が受け取ることで、内部で用いる関数を変化させた。これにより、それぞれの場合での関数実行の流れが明確になっただけでなく、直接文字列を指定して、正答率計算を簡潔に記述することができた。

3.4 問題点と考察

正答率計算においては、工夫点として上にあげたように簡潔に条件分岐を記述することができたが、正答率計算関数の内部で用いた順伝播関数は、訓練時、テスト時それぞれ分けて記述されており、記述量が多くなり冗長なコードとなってしまった。また、Dropout において、出力を無視する要素を選択するメソッドが少しわかりにくくなってしまった。また、条件分岐の際に exit() を用いたため、カーネルがクラッシュしてしまった。

3.5 リファクタリング

リファクタリングには、Gemini(2.5 Flash) を用いた。

- プロンプト: (コード、プロット結果貼り付け) これは、3 層ニューラルネットワークにおいて、Dropout を実装したものである。これに論理エラーはあるか。
- 目的: 認識できていない論理エラーの検出

結果、現在のコードはスケーリングが施されておらず、期待値計算として好ましくないと回答された。ドロップアウトにおいてスケーリング（$\frac{1}{1-p}$ や $1-p$ を乗じる処理）を行う目的は、ニューロンの出力の期待値を維持することであり、これは、訓練時とテスト時の間で、ニューラルネットワークの出力量のスケールが急激に変化するのを防ぎ、予測性能を安定させるために不可欠な処理であるという。今回の課題では、スケーリングの実装は要求されていないため、リファクタリングの結果として、カーネルのクラッシュを改善した。

- プロンプト: (コード貼り付け) これは、3 層ニューラルネットワークにおいて、Dropout を実装したものである。不正な値が入力された際にカーネルがクラッシュしてしまう。原因と解決策を教えて
- 目的: カーネルがクラッシュしてしまう原因の究明とその改善

リファクタリング前

```
mode = str(input('実行モードを入力してください (train or test): '))
if mode not in ['train', 'test']:
    print("無効なモードです。'train' または 'test' を入力してください。")
    exit()

ignore_number = int(input('Dropout の個数を 0 ~ 100 で入力してください: '))
if not (0 <= ignore_number <= hidden_layer_size):
    print("無効なドロップアウト数です。0 から 100 の範囲で入力してください。")
```

```
        exit()
```

リファクタリング後

```
    while True:
        mode = str(input('実行モードを入力してください (train or test): '))
        if mode in ['train', 'test']:
            break
        print("無効なモードです。'train' または 'test' を入力してください。")

    while True:
        try:
            ignore_number = int(input(f'Dropout の個数を 0 ~ {hidden_layer_size} で入力
してください: '))
            if 0 <= ignore_number <= hidden_layer_size:
                break
            else:
                print(f"無効なドロップアウト数です。0 から{hidden_layer_size}の範囲で入力し
てください。")
        except ValueError:
            print("無効な入力です。整数を入力してください。")
```

exit() を用いずに、while でループさせることによって、カーネルのクラッシュを回避した。また、Dropout の個数の指定を、"0 から 100"としていたが、中間層の個数が変化した場合にも対応するため、"0 から hidden_layer_size"とした。

4 4-4 慣性項

課題内容

慣性項 (Momentum) 付き SGD 慣性項 (Momentum) 付き SGD では，パラメータ W の更新量に前回の更新量の α 倍を加算する手法である．事前に設定する必要のあるパラメータは学習率 η と α である．($\eta = 0.01$, $\alpha = 0.9$ くらいがおすすめ)

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

$$W \leftarrow W + \Delta W$$

なお，ΔW の初期値は $\mathbf{0}$ とする．

4.1 作成したプログラム

この課題では、訓練時、逆伝播において重みを更新する backward_propagation_and_update_train のプロセスに変更を加えた。以下がそのコードの差分である。

変更前

```
def backward_propagation_and_update_train(..., ignore_number):

    ...

    weight1 -= dEn_dW_2 * learning_rate
    bias1   -= dEn_db_2 * learning_rate
    weight2 -= dEn_dW_1 * learning_rate
    bias2   -= dEn_db_1 * learning_rate

    return weight1, bias1, weight2, bias2
```

変更後

```
def backward_propagation_and_update_train(..., ignore_number, momentum, prev_delta_W1, prev_delta_W2

    ...

    weight1 = weight1 + (momentum * prev_delta_W1 - dEn_dW_2 * learning_rate )
    bias1   -= dEn_db_2 * learning_rate
    weight2 = weight2 + (momentum * prev_delta_W2 - dEn_dW_1 * learning_rate )
    bias2   -= dEn_db_1 * learning_rate

    params_delta_W1 = dEn_dW_2 * learning_rate
    params_delta_W2 = dEn_dW_1 * learning_rate

    return weight1, bias1, weight2, bias2, params_delta_W1, params_delta_W2
```

新たに慣性項パラメータ $\alpha = 0.9$ 、前回の重みの変化量である params_delta_W1, params_delta_W2 を導入し、資料に従って重みの更新を行った。重みの変化量を、

```
params_delta_W1 = dEn_dW_2 * learning_rate
params_delta_W2 = dEn_dW_1 * learning_rate
```

とするか、

```
params_delta_W1 = momentum * prev_delta_W1 - dEn_dW_2 * learning_rate
```

```
params_delta_W2 = momentum * prev_delta_W2 - dEn_dW_1 * learning_rate
```

とするか 2 通りの実装の方法が考えられたが、今回は前者の形をとっている。また、メイン処理部分において、params_delta_W1, params_delta_W2 = 0 の初期化を行っている。

4.2 実行結果

```
ロードしますか? yes or no: no
実行モードを入力してください (train or test): train
Dropout の個数を 0 ~ 100 で入力してください: 10

--- 訓練モード実行中 ---
1 エポック目
  平均クロスエントロピー誤差: 2.384880129041223
  学習データに対する正答率: 0.2312666666666665
  テストデータに対する正答率: 0.3338
2 エポック目
  平均クロスエントロピー誤差: 1.7131512353370284
  学習データに対する正答率: 0.3887166666666666
  テストデータに対する正答率: 0.4506
  ...
10 エポック目
  平均クロスエントロピー誤差: 0.8172133287892329
  学習データに対する正答率: 0.7791333333333331
  テストデータに対する正答率: 0.7523
```

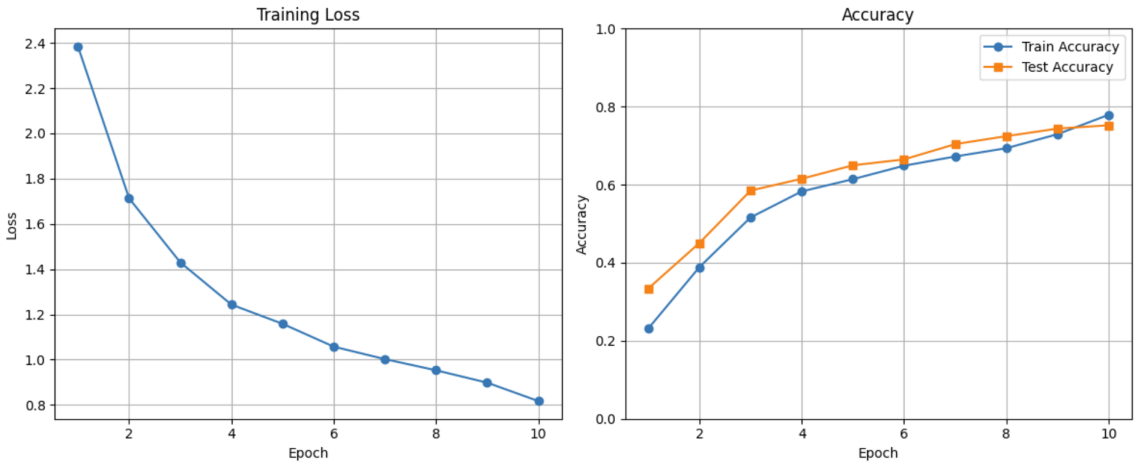

図 6　4-4(慣性項) プロット結果 (ロードなし、Dropout = 10)

```
ロードしますか? yes or no: no
```

```
実行モードを入力してください（train or test）: train
Dropout の個数を 0 ~ 100 で入力してください: 30

--- 訓練モード実行中 ---
1 エポック目
  平均クロスエントロピー誤差: 2.19495728323395
  学習データに対する正答率: 0.3730166666666668
  テストデータに対する正答率: 0.559
2 エポック目
  平均クロスエントロピー誤差: 1.45440449015992
  学習データに対する正答率: 0.530683333333333
  テストデータに対する正答率: 0.6373
  ...
10 エポック目
  平均クロスエントロピー誤差: 0.7935305138366348
  学習データに対する正答率: 0.7825833333333335
  テストデータに対する正答率: 0.8966
```

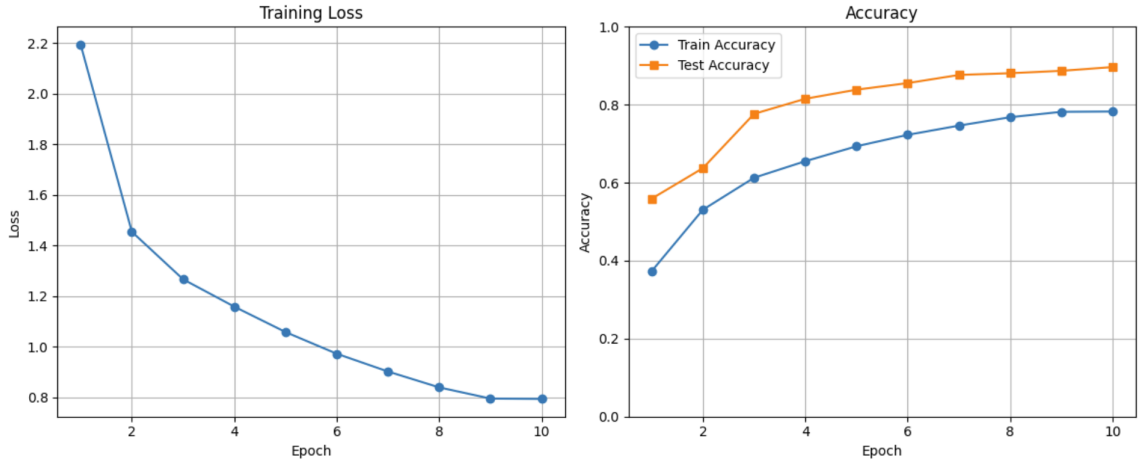

図 7　4-4(慣性項) プロット結果 (ロードなし、Dropout = 30)

プロット結果を見ると、今まで不安定だったグラフが、着実に精度を上げ、大幅に完全されたことがわかる。また、Dropout の数によって、正答率や、訓練データとテストデータの差が大きく変化することもわかる。ただ、この差は、前回の課題で述べたスケーリングを実装すれば減少するのではないかと考えた。

4.3　リファクタリング

リファクタリングには、Gemini(2.5 Flash) を用いた。

- プロンプト: (コード貼り付け) これは、３層ニューラルネットワークにおいて、慣性項を実装したもの

である。これは慣性項を正しく実装できているか、また、慣性項を実装する利点を教えて

- 目的: 論理エラーの改善、慣性項に対する理解向上

この結果、慣性項の実装がうまく行えていることが分かった。また、慣性項には以下の利点があることが分かった。

- 学習の高速化: Momentum は、更新ベクトル（ΔW）に「慣性」を持たせることで、同じ方向の勾配が続く場合、更新が加速される。これにより、最適解に素早く到達する。
- 最適解への収束性の向上: Momentum は、振動方向の勾配を相殺し、一貫性のある谷底への方向に加速するため、効率的な収束を助ける。

5　5 畳み込み層の実装

5.1　作成したプログラム

本課題では、従来の全結合層によるネットワークの前に畳み込み層（Conv 層）を導入し、画像データに特化した階層構造を持つニューラルネットワークを構築した。Conv 層の計算は、Im2col（Image to Column）変換により行列演算に置き換えられ、既存の全結合層の順伝播・逆伝播メカニズムと統合されている。

データ取得部分のメソッドの説明は割愛する。また、今回、フィルタサイズは 3 × 3、ストライドは 1、パディングは縦横それぞれ 1 マス行っている。

- padding_data 関数：データの形状変換とパディングを行うための前処理関数

```
def padding_data(train_images, test_images, pad=1):

train_imgs = train_images.reshape(-1, 3, 32, 32).transpose(0, 2, 3, 1)
test_imgs = test_images.reshape(-1, 3, 32, 32).transpose(0, 2, 3, 1)

    padded_train = np.pad(
        train_imgs,
        ((0, 0), (pad, pad), (pad, pad), (0, 0)),
        mode="constant",
        constant_values=0.0,
    )
    padded_test = np.pad(
        test_imgs,
        ((0, 0), (pad, pad), (pad, pad), (0, 0)),
        mode="constant",
        constant_values=0.0,
```

```
    )

    return padded_train, padded_test
```

reshape、transpose を用いて、(画像数 N, 3072) の形状を (画像数 N, チャンネル数 3, 高さ 32, 幅 32) の形状に変換し、0 番目（画像数）を維持しつつ、1 番目（チャンネル）と 2, 3 番目（高さ、幅）を入れ替え、最終的に (N, H, W, C) の順序にしている。その後、numpy の pad を用いて、縦横を 1 マスずつ 0 で埋めてパディングしている。パディングしたマスに constant_values=0.0 で 0 を、mode="constant" で同じ値 (0) ですべて埋めることを表している。

- im2col 関数: 入力画像 $\mathbf{X}$ を、フィルタとの行列積として畳み込み計算が可能な 2 次元行列 $\mathbf{X}_{\text{col}}$ に変換する機能を持つ。

```
def im2col(padding_data, filter_size, stride=1, pad=0):
    N, H, W, C = padding_data.shape
    out_h = (H - filter_size) // stride + 1
    out_w = (W - filter_size) // stride + 1

    col = np.zeros((N, out_h, out_w, filter_size, filter_size, C))

    for i in range(out_h):
        i_start = i * stride
        i_end = i_start + filter_size
        for j in range(out_w):
            j_start = j * stride
            j_end = j_start + filter_size
            col[:, i, j, :, :, :] = padding_data[:, i_start:i_end, j_start:j_end, :]

    col = col.reshape(-1, filter_size * filter_size * C).T

    return col
```

まず、入力データ padding_data の形状 ($\mathbf{N}, \mathbf{H}, \mathbf{W}, \mathbf{C}$)、フィルタサイズ (filter_size)、およびストライド (stride) から、畳み込み後の出力特徴マップの高さ (out_h) と幅 (out_w) を計算する。次に、col を、($\mathbf{N}$, out_h, out_w, filter_size, filter_size, $\mathbf{C}$) の形状でゼロ初期化する。これは、出力マップの各位置に対応するフィルタサイズのパッチを抽出するための領域である。その後、for ループにより、パッチの切り出しと格納を行う。フィルタサイズ、ストライドにより切り出し位置を定め、padding_data からこの領域 [:, i_start:i_end, j_start:j_end, :] をスライスして抽出し、col の該当位置 [:, i, j, :, :, :] に代入する。最後に、reshape を用いて col を (N*out_h*out_w, R*R*C) の形状に変形し、転置を行う。これにより、入力画像 $\mathbf{X}$ を、フィルタとの行列積として畳み込み計算が可能な 2 次元行列に変換する。

- set_filter_weights 関数: 畳み込みフィルタ（重み行列 $\mathbf{W}$）を初期化する。活性化関数として ReLU が使用されているため、He の初期化（He initialization）と呼ばれる手法を適用している。

```
def set_filter_weights():
    K = 32  # フィルタ枚数
    R = 3   # フィルタサイズ
    ch = 3  # 入力チャネル数
    input_node_count = R * R * ch
    std_dev = np.sqrt(2.0 / input_node_count)
    W = std_dev * np.random.randn(K, R * R * ch)

    return W, R
```

- set_biases 関数: 畳み込み層のバイアスベクトル $\mathbf{B}$ を初期化する。今回、初期値に関する記述が資料になかったため、$\mathbf{B}$ の初期値を、平均が 0、標準偏差が 0.01 の正規分布で与えている。

```
def set_biases():
    K = 32  # フィルタ枚数
    b = np.random.normal(loc=0.0, scale=0.01, size=K)
    b_vector = b.reshape(-1, 1) # (K, 1) に整形

    return b_vector
```

- conv_forward 関数: 畳み込み層の順伝播 $\mathbf{Y} = \mathbf{W}\mathbf{X}_{\text{col}} + \mathbf{B}$ を実行する。

```
def conv_forward(padded_data, conv_W, conv_b_vector, filter_size, stride=1):

    N, H_prime, W_prime, C = padded_data.shape
    K, _ = conv_W.shape

    col = im2col(padded_data, filter_size, stride=stride, pad=0)

    conv_out = np.dot(conv_W, col)

    conv_out += conv_b_vector

    out_h = (H_prime - filter_size) // stride + 1
    out_w = (W_prime - filter_size) // stride + 1

    output = conv_out.T.reshape(N, out_h, out_w, K)
```

```
    return output, col
```

まず、入力データに対して im2col 関数を呼び出し、行列 $\mathbf{X}_{\text{col}}$（col）を生成する。その後、フィルタ重み $\mathbf{W}$ と $\mathbf{X}_{\text{col}}$ の行列積を計算し、バイアス $\mathbf{B}$ を加算する。計算結果 $\mathbf{Y}$ を、全結合層への入力に適した 4 次元の特徴マップ形状 $(\mathbf{N}, \mathbf{out\_h}, \mathbf{out\_w}, \mathbf{K})$ に整形して返す。$\mathbf{K}$ は、フィルタサイズを表している。

- forward_propagation 関数: 順伝播を行う。

```
def forward_propagation_train(input_data_4d, conv_W, conv_b_vector, conv_R,
weight2, bias2, ignore_number):

    # 畳み込み層
    conv_output_pre_relu, col = conv_forward(input_data_4d, conv_W,
    conv_b_vector, conv_R, stride=1)

    # ReLU
    hidden_layer_output_conv = ReLU(conv_output_pre_relu)

    # 全結合層への入力のために平坦化
    input_vector_fc = hidden_layer_output_conv.reshape(hidden_layer_output_conv.shape[0], -1)

    # ドロップアウト適用
    hidden_layer_output = input_vector_fc.copy()
    for index in ignore_number:
        hidden_layer_output[:, index] = 0

    # 全結合層
    output_layer_input = np.dot(hidden_layer_output, weight2.T) + bias2
    final_output = softmax(output_layer_input)

    # 逆伝播に必要な情報を返す
    return final_output, hidden_layer_output, conv_output_pre_relu, col

def forward_propagation_test(input_data_4d, conv_W, conv_b_vector, conv_R,
weight2, bias2, ignore_number):
...
    hidden_layer_output = input_vector_fc * (1 - (len(ignore_number) / fc_input_size))
...
```

順伝播全体 (畳み込み層の順伝播、ReLU、全結合、出力 (ソフトマックス関数)) の流れを関数として一つにまとめた。ReLU 関数を通過した hidden_layer_output_conv のデータの形状は、 $(\mathbf{N}, \mathbf{out\_h}, \mathbf{out\_w}, \mathbf{K})$ である。これを、全結合層への入力のために、(バッチサイズ, 残りの全ての次元) という形にする $(\mathbf{N}, \mathbf{out\_h} * \mathbf{out\_w} * \mathbf{K})$。weight2 は (10, 32728) であるので $(\textbf{出力層の数}, \mathbf{out\_h} * \mathbf{out\_w} * \mathbf{K})$、全結合後ソフトマックス関数に入力するデータの形状は、$(\mathbf{N}, \textbf{出力層の数})$ となる。また、返り値として、最終結果だけでなく、逆伝播に必要な hidden_layer_output, conv_output_pre_relu, col を含めている。これはドロップアウトを実装しているので、訓練時とテスト時で実装が異なる。

- backward_propagation_and_update_train 関数: ReLU 関数までの逆伝播を行う。

```
def backward_propagation_and_update_train(hidden_layer_output, output_probabilities,
one_hot_labels, weight2, bias2, learning_rate, ignore_number, momentum,
prev_delta_W2,conv_output_pre_relu, train_images_col):

    current_batch_size = hidden_layer_output.shape[0]

    dEn_dak = (output_probabilities - one_hot_labels) / current_batch_size  # (N, Output_K)

    dEn_dW_2 = np.dot(dEn_dak.T, hidden_layer_output)  # (Output_K, FC_Input_Size)
    dEn_db_2 = np.sum(dEn_dak, axis=0)                # (Output_K,)

    dY_conv_vector = np.dot(dEn_dak, weight2)          # (N, FC_Input_Size)

    for index in ignore_number:
        dY_conv_vector[:, index] = 0

    delta_W2 = momentum * prev_delta_W2 - dEn_dW_2 * learning_rate

    weight2 += delta_W2
    bias2 -= dEn_db_2 * learning_rate

    N, fc_input_size = dY_conv_vector.shape
    conv_output_h = 32
    conv_output_w = 32
    conv_K = 32

    dY_conv_4d = dY_conv_vector.reshape(N, conv_output_h, conv_output_w, conv_K)

    relu_diff_conv = np.where(conv_output_pre_relu > 0, 1, 0)
```

```
    dY_conv_4d *= relu_diff_conv

    return weight2, bias2, delta_W2, dY_conv_4d
```

出力層の勾配計算、全結合層、ドロップアウトの逆伝播とパラメータ更新を行った後、畳み込み層への誤差伝播を計算している。得られた勾配ベクトル $\mathbf{dY}_{\text{conv\_vector}}$ を、畳み込み層の出力と同じ形状の $\mathbf{dY}_{\text{conv\_4d}}$ に復元する。

$$(\mathbf{N}, \mathbf{out\_h} * \mathbf{out\_w} * \mathbf{K}) \xrightarrow{\text{reshape}} (\mathbf{N}, \text{out\_h}, \text{out\_w}, \mathbf{K})$$

これに、ReLU の微分を適用する。こうして得られた $\mathbf{dY}_{\text{conv\_4d}}$ が、次の畳み込み層の逆伝播に渡される。

- conv_backward 関数: 畳み込み層の逆伝播を計算し、フィルタ重み $\mathbf{W}$ とバイアス $\mathbf{B}$ の勾配を計算する。

```
def conv_backward(dY_4d, col):

    N, out_h, out_w, K = dY_4d.shape

    dY = dY_4d.transpose(3, 0, 1, 2).reshape(K, N * out_h * out_w)

    dW = np.dot(dY, col.T)

    db_vector = np.sum(dY, axis=1, keepdims=True)

    return dW, db_vector
```

上流からの誤差 $\frac{\partial E_n}{\partial \mathbf{Y}}$（dY_4d）を受け取り、これを $(\mathbf{K}, \mathbf{N} \cdot \mathbf{out\_h} \cdot \mathbf{out\_w})$ の形状に整形し、フィルタ重み $\mathbf{W}$ とバイアス $\mathbf{B}$ の勾配を計算する。

5.2 実行結果

```
ロードしますか? yes or no: no
実行モードを入力してください (train or test): train
Dropout の個数を 0 ~ 32768 で入力してください: 0

--- 訓練モード実行中 ---
1 エポック目
 平均クロスエントロピー誤差: 1.7635114039454238
 学習データに対する正答率: 0.4381600000000005
 テストデータに対する正答率: 0.4708
```

2 エポック目
 平均クロスエントロピー誤差: 1.3573994327561356
 学習データに対する正答率: 0.5528999999999998
 テストデータに対する正答率: 0.5116
...
10 エポック目
 平均クロスエントロピー誤差: 0.9591971991252656
 学習データに対する正答率: 0.6976999999999998
 テストデータに対する正答率: 0.5853

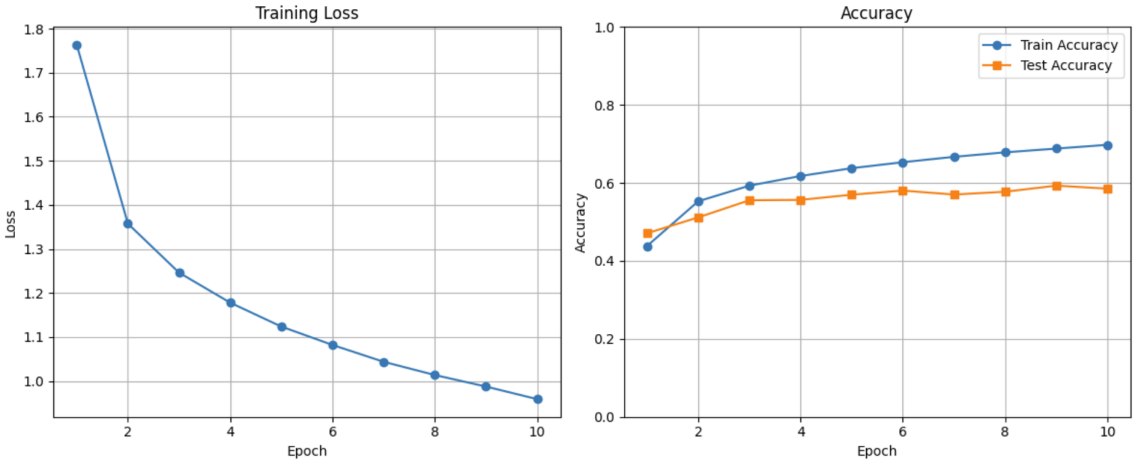

図 8　5 プロット結果 (ロードなし、Dropout = 0)

ロードしますか? yes or no: no
実行モードを入力してください (train or test): train
Dropout の個数を 0 ~ 32768 で入力してください: 2000

--- 訓練モード実行中 ---
1 エポック目
 平均クロスエントロピー誤差: 1.745491363659473
 学習データに対する正答率: 0.4426599999999998
 テストデータに対する正答率: 0.4711
2 エポック目
 平均クロスエントロピー誤差: 1.3649674217797447
 学習データに対する正答率: 0.5500400000000009
 テストデータに対する正答率: 0.5126
...
10 エポック目
 平均クロスエントロピー誤差: 0.978346438719636

学習データに対する正答率： `0.6916799999999997`
テストデータに対する正答率： `0.591`

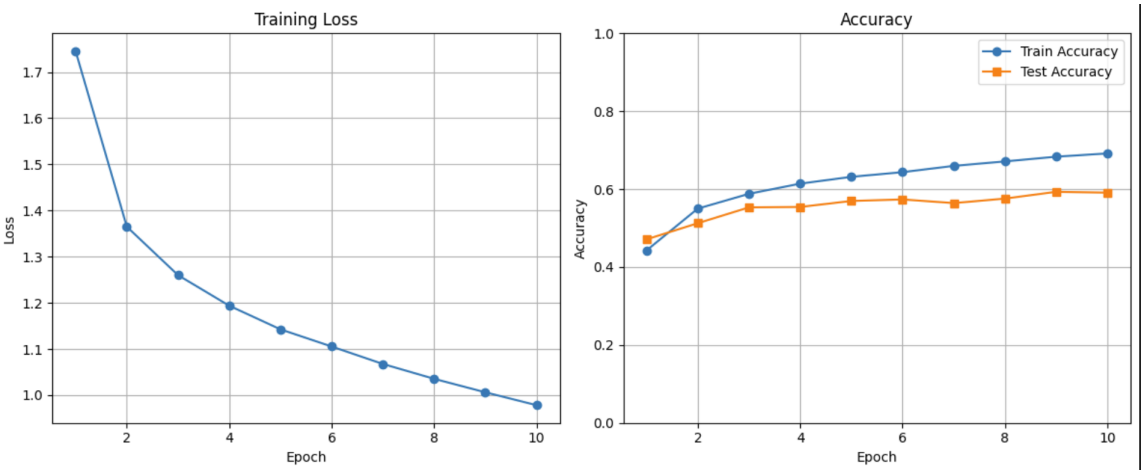

図 9　5 プロット結果 (ロードなし、Dropout = 2000)

結果を見ると、カラー画像に拡張した結果、やはり誤差は大きくなり、正答率は低くなった。しかし、エポック数を重ねると順調に精度が上がっていることがわかる。また、今回、このプログラムの実行時間は約 10 倍となった。前回の MNIST データでは入力次元数が $28 \times 28 = 784$ であったのに対し、今回のデータでは Conv 層の出力（32×32 の特徴マップが 32 枚）を平坦化するため、全結合層への入力次元は $32,768$ となっている。これにより、行列積の計算コストが増大したことが、実行時間増大の主因であると考えられる。また、今回のドロップアウトのサイズとして、全結合層の入力サイズとしている。

5.3　工夫点

im2col 関数により、4 次元の入力画像データ を、フィルタ重み $\mathbf{W}$ との行列積 $\mathbf{W}\mathbf{X}_{\text{col}}$ として計算できる 2 次元行列 $\mathbf{X}_{\text{col}}$ に変換し、畳み込み操作を効率的に処理した。また、畳み込みフィルタ $\mathbf{W}$ の初期化に ReLU 活性化関数と相性の良い He の初期化を採用した。

5.4　問題点と考察

やはり、実行時間の著しい増加が問題として挙げられる。この解決策として、課題 6 で示されているプーリング層の導入が挙げられる。プーリングにより、特徴マップの空間サイズ（高さと幅）をダウンサンプリングし、全結合層への入力次元を削減する。これにより行列積の計算量を減らすことで、実行時間を短縮できると考えられる。

5.5　リファクタリング

リファクタリングには、Gemini(2.5 Flash) を用いた。

- プロンプト: (コード貼り付け) これは、畳み込み層を導入し、カラー画像に対応するニューラルネット

ワークを実装したものである。これに論理エラーはあるか、また、実行結果が大幅に遅くなった理由は何か。

- 目的: 論理エラーの改善、Conv 層導入に伴う計算量の理解

結果、以下のことが得られた。

- 計算コスト増大の妥当性: 実行速度が約 10 倍に低下した原因として、全結合層への入力次元が 784（前回）から 32,768（今回）へと大幅に増加したこと、および Conv 層自体の計算（Im2col 変換と行列積）が加わったことが特定された。
- 論理エラー（逆伝播の欠落）の指摘: conv_backward 関数について、フィルタ重み $\mathbf{W}$ とバイアス $\mathbf{B}$ の勾配（$\mathbf{dW}$, $\mathbf{db}$）は正しく計算されていることがわかった。しかし、畳み込み層の入力データ $\mathbf{X}$ に対する誤差（勾配）$\mathbf{dX}$（次の層へ伝播させる誤差）を計算し、col2im 変換により元の形状に戻す処理が欠落していることが指摘された。今回のネットワーク構成では畳み込み層が先頭であり入力画像 $\mathbf{X}$ を更新しないため問題はない。

6　6 プーリング層の実装

課題内容

プーリング層は多チャンネルの畳み込み層の出力を入力として受け取り, 多チャンネルの画像を出力する層である. 具体的には K チャンネル $d_x \times d_y$ の多チャンネル画像 (3 次元配列) を入力として受け取り, K チャンネル $d_x/d \times d_y/d$ の多チャンネル画像を出力する. ここで d はプーリング層のサイズである. また, チャンネルサイズ K は畳み込み層の出力のチャンネル層のサイズである. すなわち畳み込みのフィルタ枚数である.

ここでは, プーリング層のなかでも最も良く用いられている max プーリングについて説明する. 入力を 3 次元配列 $X[c,i,j]$ $(c = 1,...,K,\ i = 1,...,d_x,\ j = 1,...,d_y)$ とし, 出力を 3 次元配列 $Y[c,i,j]$ $(c = 1,...,K,\ i = 1,...,\lfloor d_x/d \rfloor,\ j = 1,...,\lfloor d_y/d \rfloor)$ とする. 出力 Y は,

$$Y[c,i,j] = \max_{0 \le k < d, 0 \le l < d} X[c, id+k, jd+l] \quad \text{where } 0 \le k < d, 0 \le l < d$$

によって得られる.

プーリング層において学習されるパラメータは存在しない. 逆伝播は,

$$\frac{\partial E_n}{\partial X[c,id+k,jd+l]} = \begin{cases} \frac{\partial E_n}{\partial Y[c,i,j]} & \text{if } X[c,id+k,jd+l] = Y[c,i,j] \\ 0 & \text{otherwise} \end{cases}$$

によって得ることができる.

6.1　作成したプログラム

プーリング層の実装にあたり、max_pooling_forward（順伝播）と max_pooling_backward（逆伝播）の 2 つの関数を新たに作成した。プーリング層は学習するパラメータを持たないため、重みやバイアスの更新は行わない。

- pool_forward 関数: 入力された特徴マップを Max プーリングする。この際、逆伝播で使用するために、最大値がどこにあったかの情報を保持する。

```
def max_pooling_forward(conv_output_4d, pool_h=POOL_SIZE,
pool_w=POOL_SIZE, stride=POOL_STRIDE):

    N, H, W, C = conv_output_4d.shape
    out_h = (H - pool_h) // stride + 1
    out_w = (W - pool_w) // stride + 1

    col = np.zeros((N, out_h, out_w, pool_h, pool_w, C))

    for i in range(out_h):
        i_max = i * stride + pool_h
        for j in range(out_w):
            j_max = j * stride + pool_w
            col[:, i, j, :, :, :] = conv_output_4d[:, i * stride:i_max, j * stride:j_max, :]

    col_max_calc = col.reshape(N * out_h * out_w, pool_h * pool_w, C)

    out_flat = np.max(col_max_calc, axis=1)
    max_idx = np.argmax(col_max_calc, axis=1)

    out = out_flat.reshape(N, out_h, out_w, C)

    return out, max_idx, conv_output_4d.shape
```

入力形状 $(\mathbf{N}, \mathbf{H}, \mathbf{W}, \mathbf{C})$ とプーリングパラメータ (pool_h, pool_w, stride) から、出力の高さ out_h と幅 out_w を計算し、col を $(\mathbf{N}, \text{out\_h}, \text{out\_w}, \text{pool\_h}, \text{pool\_w}, C)$ の形状でゼロ初期化する。for ループを用いて、入力特徴マップ $\mathbf{X}$ からプーリング領域 (pool_h × pool_w) を重複なく切り出し、col に格納する。col を $(\text{パッチの総数}, \text{pool\_size}^2, C)$ の形状（col_max_calc）に整形し、プーリング領域軸（axis = 1）で最大値 np.max を計算する。これにより、各プーリング領域の最大値が抽出され、出力 out_flat を得る。Max プーリングの逆伝播では、最大値がどこにあったかのインデックス情報が必要である。np.argmax により、プーリング領域内の最大値の位置インデックス max_idx を計算し、保持する。得られた out_flat を、最終的な出力特徴マップの形状 $(\mathbf{N}, \text{out\_h}, \text{out\_w}, \mathbf{C})$ に整形し、max_idx と共に出力する。

- max_pooling_backward 関数: 上流から伝播された誤差 ($\frac{\partial E_n}{\partial \mathbf{Y}}$) を受け取り、順伝播時に保持した情報を使って、Max プーリング層の入力側へ誤差を伝播させる。最大値であった要素にのみ誤差を伝える。

```
def max_pooling_backward(dY, max_idx, input_shape, pool_h=POOL_SIZE,
pool_w=POOL_SIZE, stride=POOL_STRIDE):

    N, H, W, C = input_shape
    out_h = dY.shape[1]
    out_w = dY.shape[2]

    dY_flat = dY.reshape(-1, C)

    dX_col = np.zeros((dY_flat.shape[0], pool_h * pool_w, C))


    idx_flat = np.arange(dY_flat.shape[0])[:, None]

    dX_col[idx_flat, max_idx, np.arange(C)[None, :]] = dY_flat

    dX_col = dX_col.reshape(N, out_h, out_w, pool_h, pool_w, C)

    dX = np.zeros(input_shape)

    for i in range(out_h):
        i_max = i * stride + pool_h
        for j in range(out_w):
            j_max = j * stride + pool_w
            dX[:, i * stride:i_max, j * stride:j_max, :] += dX_col[:, i, j, :, :, :]

    return dX
```

入力誤差 $\mathbf{dY}$ を dY_flat = (パッチ総数, $\mathbf{C}$) に平坦化し、ゼロ行列 $\mathbf{dX}_{\text{col}}$ を (パッチ総数, pool_size^2, $\mathbf{C}$) の形状で準備する。保持しておいた最大値インデックス max_idx を使用し、平坦化された誤差 dY_flat の値を、$\mathbf{dX}_{\text{col}}$ 内の順伝播で最大値であった要素の位置にのみコピーする。idx_flat は dY_flat の各行（プーリング領域）を識別する。dX_col[idx_flat, max_idx, np.arange(C)[None, :]] = dY_flat が、最大値位置への誤差の選択的配置を実行する。$\mathbf{dX}_{\text{col}}$ を $(\mathbf{N}, \text{out\_h}, \text{out\_w}, \text{pool\_h}, \text{pool\_w}, \mathbf{C})$ の形状に戻し、最終的な勾配 $\mathbf{dX}$ をゼロ初期化する。for ループを用いて、$\mathbf{dX}_{\text{col}}$ のパッチを元の入力 $\mathbf{X}$ の形状（input_shape）に戻しながら、重複する領域の誤差を加算していく。

また、pooling 層の実装に伴い、順伝播、逆伝播関数の実装は以下のように変化した。

順伝播:

```
def forward_propagation_train(input_data_4d, conv_W, conv_b_vector, conv_R,
```

```
weight2, bias2, ignore_number):

...
    pool_output, pool_mask, pool_input_shape = max_pooling_forward(conv_output_pre_relu)

    relu_conv_output = ReLU(pool_output)

    input_vector_fc = relu_conv_output.reshape(relu_conv_output.shape[0], -1)

...
    return ...,  pool_mask, pool_input_shape, pool_output
```

逆伝播:

```
def backward_propagation_and_update_train(hidden_layer_output, output_probabilities,
one_hot_labels, weight2, bias2, learning_rate, ignore_number, momentum,
prev_delta_W2, conv_output_pre_relu, col, pool_mask, pool_input_shape, pool_output):
...
    dEn_dX_pool = np.dot(dEn_dak, weight2)

    for index in ignore_number:
         dEn_dX_pool[:, index] = 0
...

    N = current_batch_size

    pool_output_h = 16
    pool_output_w = 16
    conv_K = 32

    dPool_output = dEn_dX_pool.reshape(N, pool_output_h, pool_output_w, conv_K)

    dRelu_input = dPool_output * np.where(pool_output > 0, 1, 0)

    dY_conv_4d = max_pooling_backward(dRelu_input, pool_mask, pool_input_shape)

    return weight2, bias2, delta_W2, dY_conv_4d
```

6.2　実行結果

ロードしますか? yes or no: no
実行モードを入力してください (train or test): train
Dropout の個数を 0 ~ 100 で入力してください: 0

--- 訓練モード実行中 ---
1 エポック目
　平均クロスエントロピー誤差: 1.713209226643652
　学習データに対する正答率: 0.46294000000000074
　テストデータに対する正答率: 0.5255
2 エポック目
　平均クロスエントロピー誤差: 1.3115280307892117
　学習データに対する正答率: 0.5676199999999997
　テストデータに対する正答率: 0.5394
...
10 エポック目
　平均クロスエントロピー誤差: 0.9460997437346031
　学習データに対する正答率: 0.7011599999999998
　テストデータに対する正答率: 0.6214

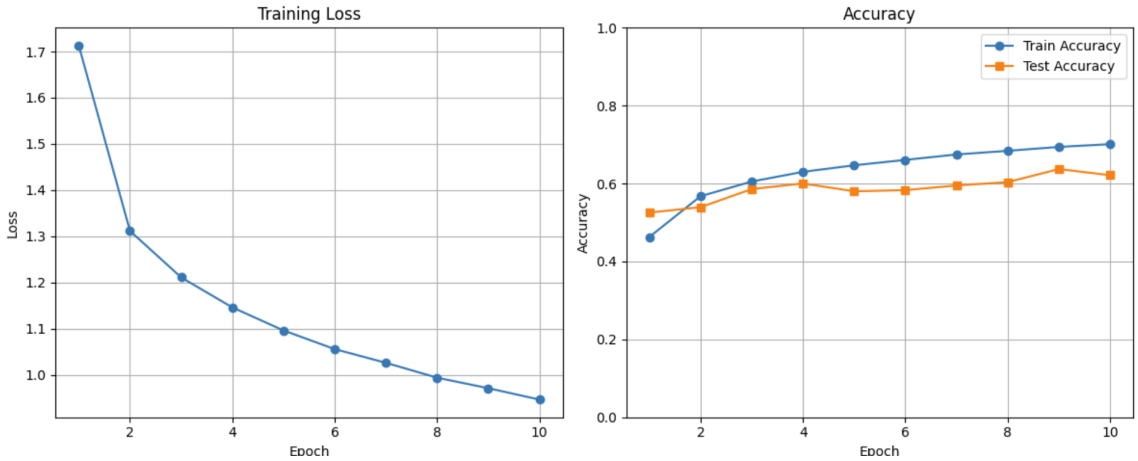

図 10　6 プロット結果 (ロードなし、Dropout = 0)

ロードしますか? yes or no: no
実行モードを入力してください (train or test): train
Dropout の個数を 0 ~ 100 で入力してください: 2000

--- 訓練モード実行中 ---

1 エポック目
平均クロスエントロピー誤差: 1.751038416835169
学習データに対する正答率: 0.4210400000000002
テストデータに対する正答率: 0.5056
2 エポック目
平均クロスエントロピー誤差: 1.4149346373033214
学習データに対する正答率: 0.5290600000000003
テストデータに対する正答率: 0.5266
...
10 エポック目
平均クロスエントロピー誤差: 1.0759614707715426
学習データに対する正答率: 0.6549999999999999
テストデータに対する正答率: 0.6155

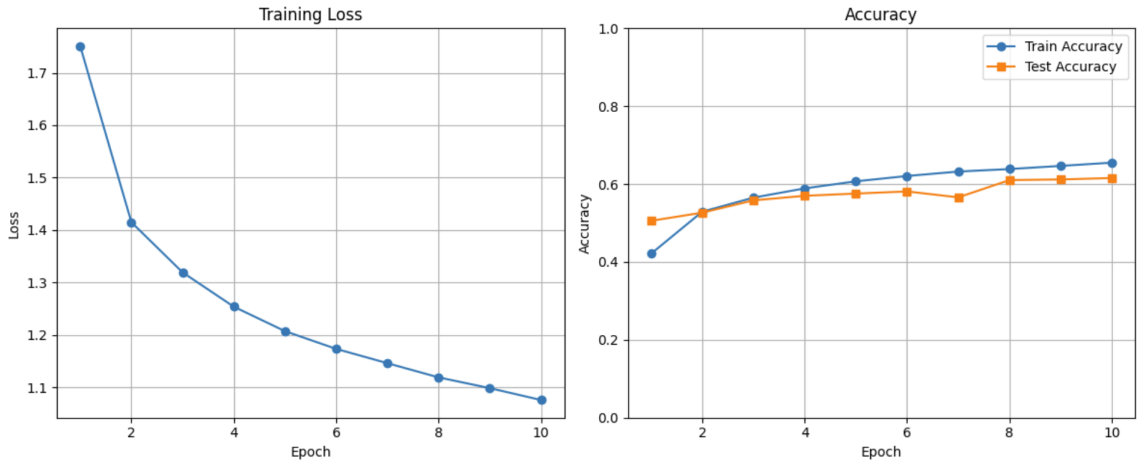

図 11　6 プロット結果 (ロードなし、Dropout = 2000)

6.3　工夫点

プーリング層は、最大値を出力した位置にのみ誤差を伝播させる特性を持つ。 max_pooling_forward 関数内で np.argmax を使用して最大値インデックス (max_idx) を正確に計算・保持することで、max_pooling_backward における誤差の正確なルーティングを可能にした。

6.4　問題点と考察

前回の考察で、カラー画像に対応するよう実装したことによって、全結合層への入力の次元数が劇的に増加し、それに伴い実行時間が増加したと結論付けた。プーリング層の導入により、前課題で指摘された「実行時間の著しい増加」を改善し、実行時間は約 1/3 に短縮されると考えたが、実際の実行時間は 3/4 程度にしかならなかった。つまりこれは、実行時間増加の主因が全結合層の次元数増加だけではないことを示唆している。

6.5 リファクタリング

リファクタリングには、Gemini(2.5 Flash) を用いた。

- プロンプト: (コード貼り付け) これは、畳み込み層を導入し、カラー画像に対応するニューラルネットワークを実装し、さらに maxpooling 層を実装したものである。これに論理エラーはあるか
- 目的: 論理エラーの改善

  結果、ReLU の適用位置が、非標準的と指摘された。現在のプログラムにおける順伝播は、畳み込み→ maxpooling → ReLU →全結合→ソフトマックス関数となっているが、一般的には、畳み込み→ ReLU → maxpooling →... であるという。しかし、課題 5、6 の結果を比較すると、最終的な精度は悪化しておらず、むしろ少し良くなっている。そのため、この ReLU と pooling 層の適用順番の違いが結果に与える差は大きくないと考察できる。

7 課題 7

この課題 7 では、今までに実装した、活性化関数 (シグモイド、 ReLU)、dropout、 慣性項が、3 層ニューラルネットワーク全体の計算の精度にどのような影響を与えているのかを検証する。この検証に伴い、慣性項の論理エラーと思われる箇所を発見した。畳み込み層におけるフィルタ重み更新で、慣性項を作用させていなかったため、修正を行った。また、標準入力で、活性化関数の選択 (シグモイド、 ReLU)、dropout の個数、慣性項の有無を選択できるようにした。

7.1 実行結果

活性化関数を `**ReLU**` **に設定しました。**
パラメータ更新に慣性項 `**0.9**` **を使用します。**

`--- 訓練モード実行中 ---`
1 **エポック目**
平均クロスエントロピー誤差: `1.7045962979428306`
学習データに対する正答率: `0.42950000000000016`
テストデータに対する正答率: `0.514`
2 **エポック目**
平均クロスエントロピー誤差: `1.3295393672008848`
学習データに対する正答率: `0.547`
テストデータに対する正答率: `0.538`
...
10 **エポック目**
平均クロスエントロピー誤差: `0.9472025557074539`
学習データに対する正答率: `0.6860600000000003`

テストデータに対する正答率: 0.6168

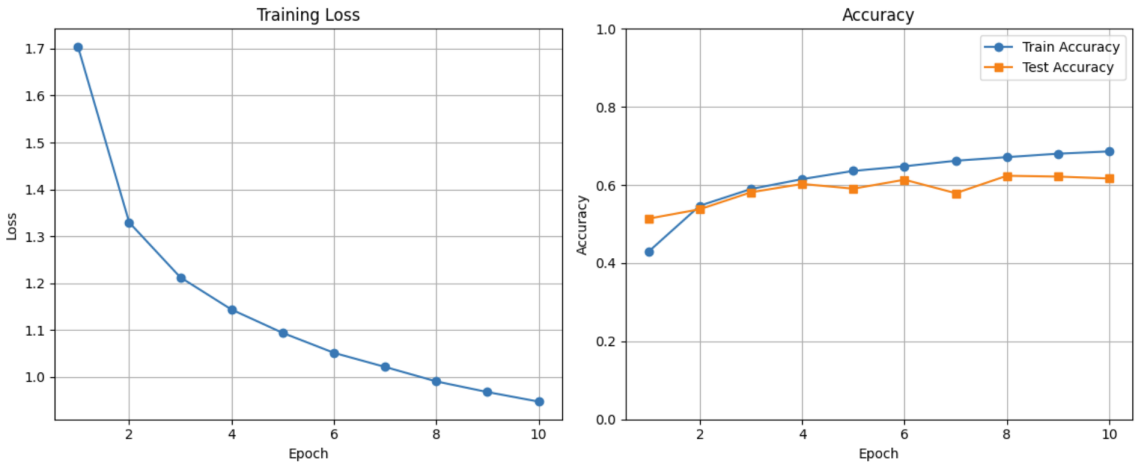

図 12　7 プロット結果 (ロードなし、ReLU 使用、Dropout = 0、 慣性項有)

活性化関数を **ReLU** に設定しました。
パラメータ更新に慣性項 **0.9** を使用します。

--- 訓練モード実行中 ---
1 エポック目
平均クロスエントロピー誤差: 1.6689724236079329
学習データに対する正答率: 0.4330799999999999
テストデータに対する正答率: 0.5086
2 エポック目
平均クロスエントロピー誤差: 1.362358214188421
学習データに対する正答率: 0.5401000000000006
テストデータに対する正答率: 0.5251
...
10 エポック目
平均クロスエントロピー誤差: 1.0128924665881593
学習データに対する正答率: 0.6639799999999997
テストデータに対する正答率: 0.64

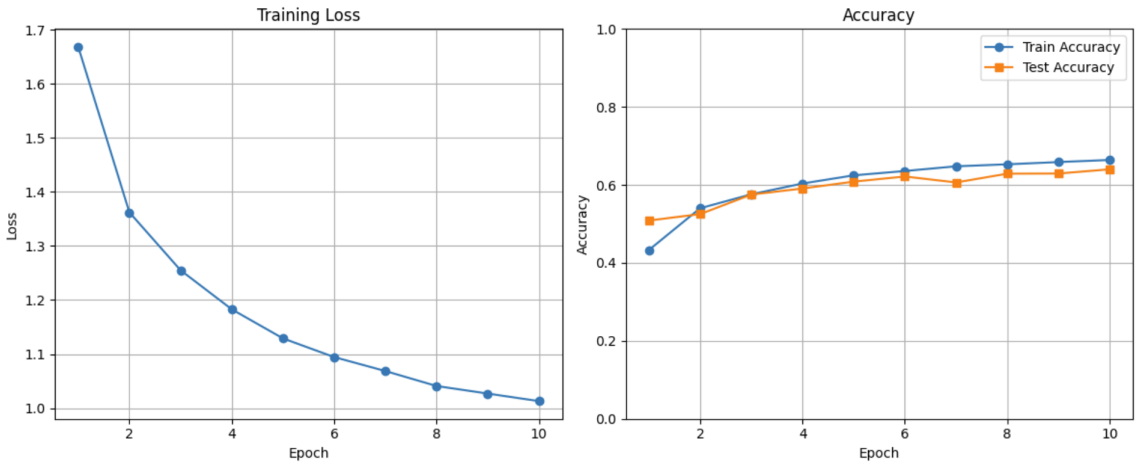

図 13　7 プロット結果 (ロードなし、ReLU 使用、Dropout = 2000、 慣性項有)

活性化関数を **ReLU** に設定しました。
パラメータ更新に慣性項 **0.9** を使用します。

--- 訓練モード実行中 ---
1 エポック目
 平均クロスエントロピー誤差: 1.750711362510338
 学習データに対する正答率: 0.40347999999999934
 テストデータに対する正答率: 0.4786
2 エポック目
 平均クロスエントロピー誤差: 1.4678865461684192
 学習データに対する正答率: 0.5009600000000003
 テストデータに対する正答率: 0.5352
...
10 エポック目
 平均クロスエントロピー誤差: 1.1184993329186732
 学習データに対する正答率: 0.6251000000000008
 テストデータに対する正答率: 0.6133

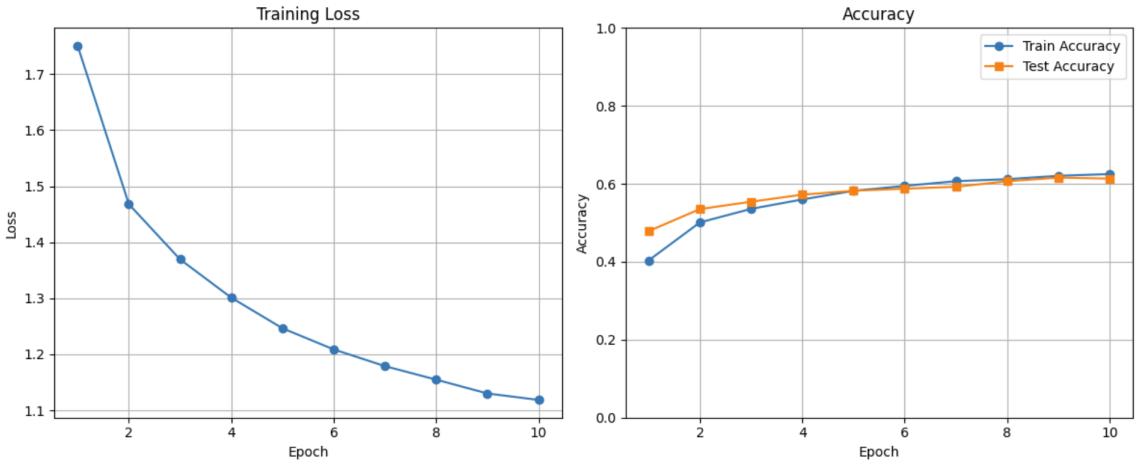

図 14　7 プロット結果 (ロードなし、ReLU 使用、Dropout = 4000、 慣性項有)

活性化関数を **ReLU** に設定しました。
パラメータ更新に慣性項 **0.9** を使用します。

--- 訓練モード実行中 ---
1 エポック目
 平均クロスエントロピー誤差: 1.9019736664516413
 学習データに対する正答率: 0.3500999999999998
 テストデータに対する正答率: 0.401
2 エポック目
 平均クロスエントロピー誤差: 1.660694886399369
 学習データに対する正答率: 0.4331399999999996
 テストデータに対する正答率: 0.4783
...
10 エポック目
 平均クロスエントロピー誤差: 1.3031418295035508
 学習データに対する正答率: 0.55622
 テストデータに対する正答率: 0.5747

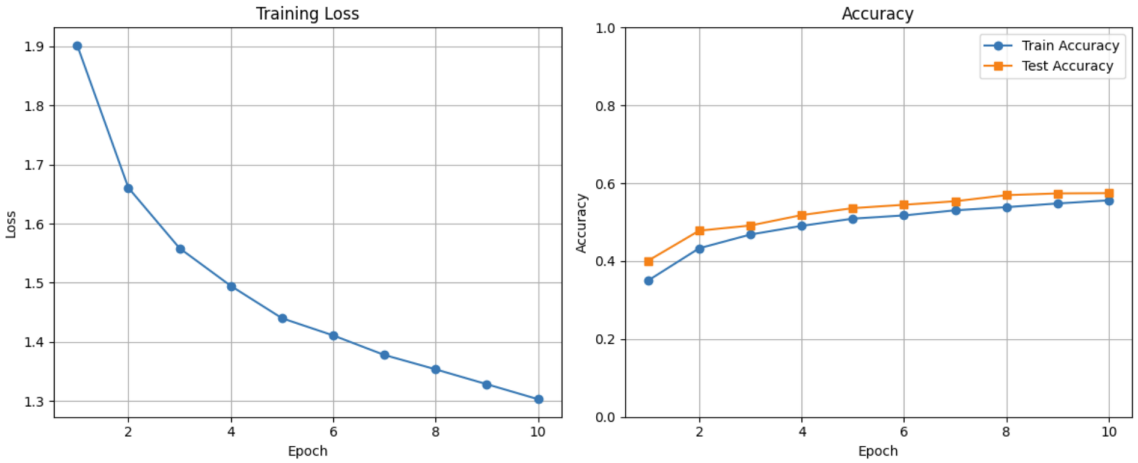

図 15　7 プロット結果 (ロードなし、ReLU 使用、Dropout = 6000、 慣性項有)

活性化関数を **ReLU** に設定しました。
パラメータ更新に慣性項は使用しません (Momentum=0.0)。

--- 訓練モード実行中 ---
1 エポック目
 平均クロスエントロピー誤差: 1.904174959020718
 学習データに対する正答率: 0.39303999999999983
 テストデータに対する正答率: 0.3861
2 エポック目
 平均クロスエントロピー誤差: 1.632643408958925
 学習データに対する正答率: 0.4752199999999997
 テストデータに対する正答率: 0.4545
...
10 エポック目
 平均クロスエントロピー誤差: 1.2233542749282833
 学習データに対する正答率: 0.61042
 テストデータに対する正答率: 0.5709

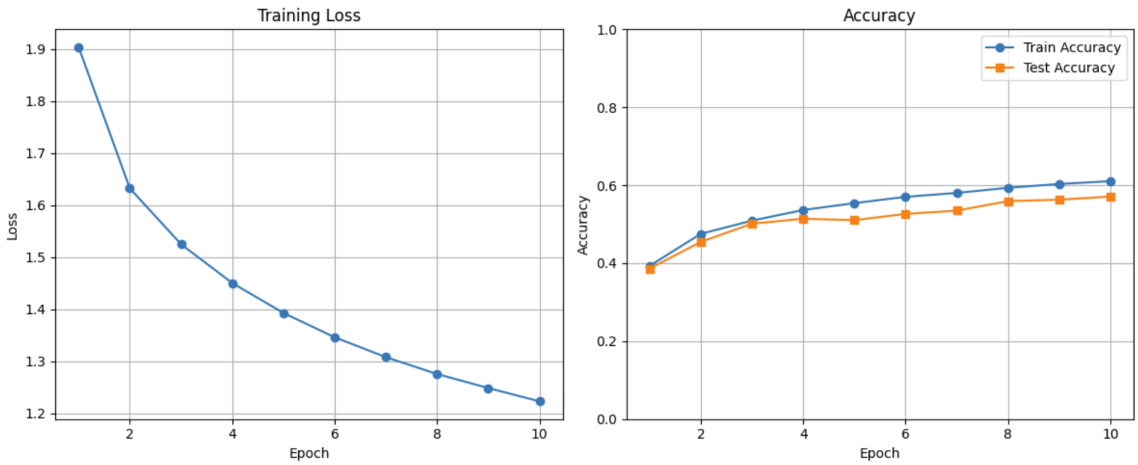

図 16　7 プロット結果 (ロードなし、ReLU 使用、Dropout = 0、 慣性項無)

活性化関数を **Sigmoid** に設定しました。
パラメータ更新に慣性項 **0.9** を使用します。

--- 訓練モード実行中 ---
1 エポック目
  平均クロスエントロピー誤差: 2.4134444161315596
  学習データに対する正答率: 0.3109199999999998
  テストデータに対する正答率: 0.3724
2 エポック目
  平均クロスエントロピー誤差: 1.7966734490499516
  学習データに対する正答率: 0.3815799999999995
  テストデータに対する正答率: 0.3661
...
10 エポック目
  平均クロスエントロピー誤差: 1.3477915977783728
  学習データに対する正答率: 0.5383200000000007
  テストデータに対する正答率: 0.5139

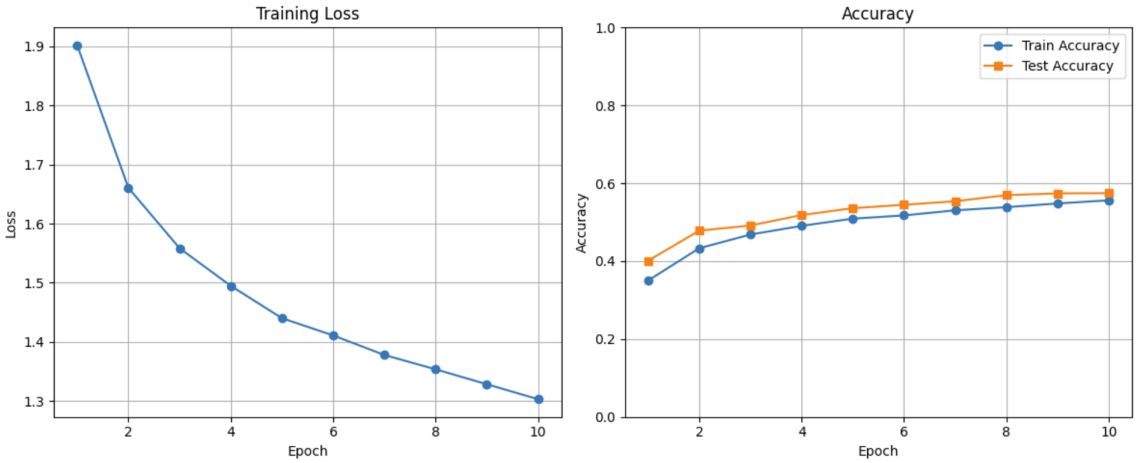

図 17　7 プロット結果 (ロードなし、Sigmoid 使用、Dropout = 0、 慣性項有)

7.2　考察

表 1　慣性項（Momentum）の影響

設定	慣性項	10 エポック目 訓練精度	10 エポック目 テスト精度	傾向
ReLU, Dropout 0	有 (0.9)	0.6861	0.6168	訓練・テスト共に高精度。収束が速い傾向。
ReLU, Dropout 0	無 (0.0)	0.6104	0.5709	全体的に精度が低い。学習が遅い傾向。

慣性項を導入（Momentum=0.9）することで、学習データに対する収束速度と、最終的なテスト精度の両方が向上する傾向が見られる。これは、慣性項が勾配の振動を抑制し、より効率的に最適解に近づいたことを示唆している。特にテスト精度では、慣性項「有」の方が約 4.5 ポイント高い結果となった。

表 2　Dropout の影響 (ReLU, Momentum 有)

Dropout 個数	訓練精度	テスト精度	訓練/テスト差	傾向
0	**0.6861**	0.6168	0.0693	訓練精度は最高だが、過学習の懸念が最も高い。
2000	0.6640	**0.6400**	**0.0240**	**テスト精度は最高**。汎化性能が最も高い。
4000	0.6251	0.6133	0.0118	訓練とテストの差が非常に小さい。良好な汎化。
6000	0.5562	0.5747	**-0.0185**	訓練・テスト共に最も低い精度。過度な学習抑制。

Dropout を増やすほど、訓練データに対する精度は低下しするが、これは Dropout の正規化効果（学習を抑制する効果）による自然な傾向である。しかし、Dropout を 2000 個設定した場合、訓練精度は低下するものの、テスト精度は最高値（0.6400）を達成した。また、訓練精度とテスト精度の差が最も小さくなっており、過学習の抑制に最も成功している傾向が強く見られる。Dropout を 4000 個以上に増やすと、学習抑制が強くなりすぎ、訓練・テスト精度ともに低下し始めた。特に 6000 個では精度が大きく悪化しており、モデルの表現力が過度に失われたと考えられる。結論として、適切な数の Dropout（この場合 2000 個程度）は、訓練精度を多少犠牲にしても、汎化性能（テスト精度）を向上させる上で非常に効果的であることが示唆される。

表 3　活性化関数（ReLU vs Sigmoid）の影響

設定	活性化関数	10 エポック目 訓練精度	10 エポック目 テスト精度	傾向
ReLU, Dropout 0	**ReLU**	**0.6861**	**0.6168**	訓練・テスト共に高精度で収束が速い。
Sigmoid, Dropout 0	**Sigmoid**	0.5383	0.5139	訓練・テスト共に低精度。収束が遅い。

ReLU を使用した方が、Sigmoid を使用した場合に比べて、訓練・テストの両方で大幅に高い精度を達成している。

7.3　結論

- 慣性項 (Momentum)：導入により学習が安定し、最終的なテスト精度が向上する傾向がある。
- 活性化関数：ReLU は Sigmoid よりもはるかに速く、高い最終精度を達成する傾向があり、高性能化に不可欠であることが示唆される。
- Dropout：
  Dropout なし（0 個）は訓練精度を最高にするものの、過学習の懸念が残る。
  適切な Dropout 数（2000 個）は、訓練精度とテスト精度の差を最も小さくし、最高の汎化性能（テスト精度）をもたらした。
  過度な Dropout（6000 個）は、モデルの学習能力を低下させ、精度を大きく悪化させた。

結論として、これらのパラメーターの中では、特に ReLU の選択と慣性項の導入がベースラインの精度を大きく向上させ、適切な Dropout 数の設定がモデルの汎化性能を最大化する鍵であることが示唆される。