計算機科学実験及演習 4 画像処理 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 10 月 30 日

1　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.

1.1　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_sig = dEn_dX * differentiated_input
```

1.2　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

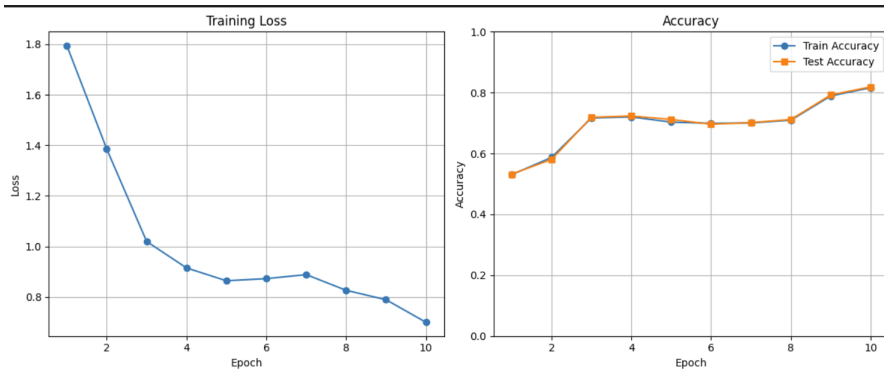

図 1　4-1 プロット結果 (ロード無し)

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

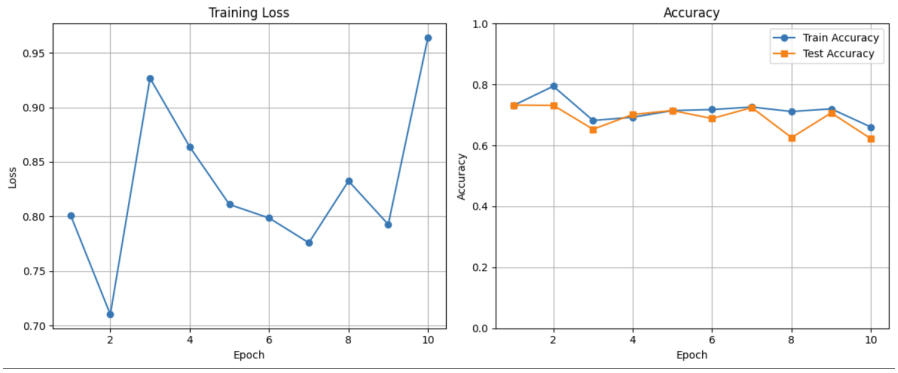

図 2　4-1 プロット結果 (ロード有り)

1.3　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。

1.4　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。

1.5　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、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。

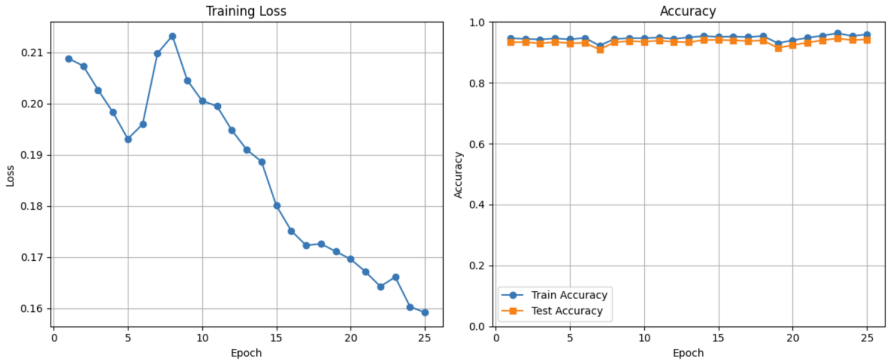

図 3　4-1 プロット結果 (ロード有り、修正後)

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。

2　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．

2.1 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 ---")
    random_selection = np.random.choice(np.arange(hidden_layer_size), size=ignore_number, replac
# テストデータに対する最終的な正答率を計算
    test_accuracy = calculate_accuracy_for_epoch(test_images, test_labels, weight1, bias1, weigh

    print(f"\n テストデータに対する最終正答率: {test_accuracy}")
    print(f"（ドロップアウト数 {ignore_number} 個）")
```

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):
    hidden_layer_input = np.dot(input_vector, weight1.T) + bias1
    hidden_layer_output = ReLU(hidden_layer_input)
    for index in ignore_number:
        hidden_layer_output[:, index] = 0 # 無視する値を 0 に
    output_layer_input = np.dot(hidden_layer_output, weight2.T) + bias2
    final_output = softmax(output_layer_input)
```

```
    return final_output, hidden_layer_input, hidden_layer_output

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_sig = 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。

2.2 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

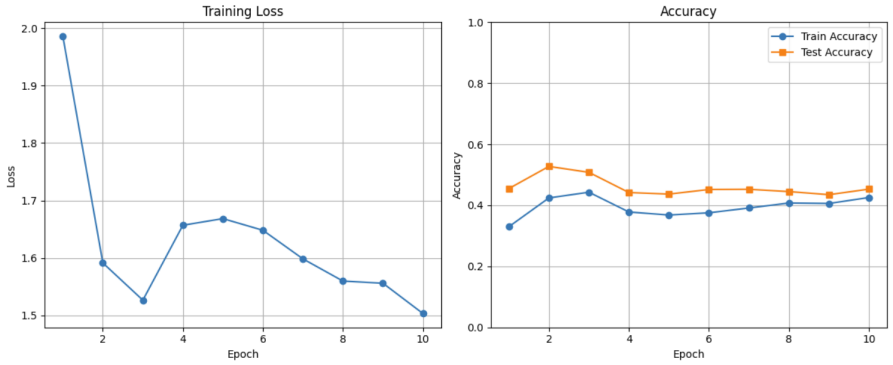

図 4　4-2 プロット結果 (ロード無し、Dropout = 20)

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

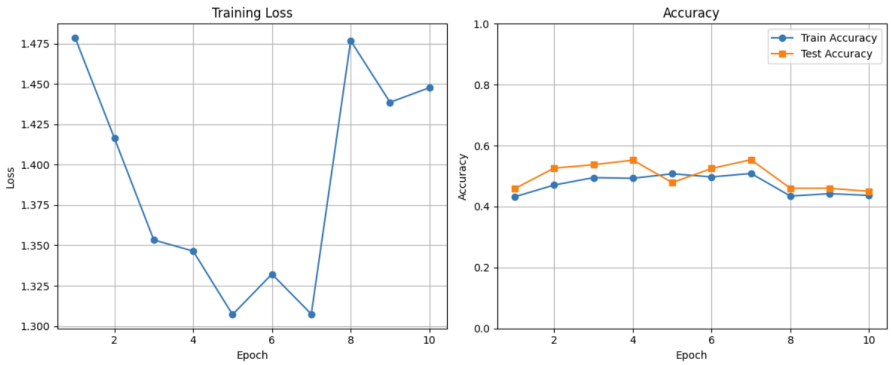

図 5　4-2 プロット結果 (ロード有り、Dropout = 20)

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)

2.3 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有力で受け取り、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。

2.4 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。

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、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。

3 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．

3.1 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。

リファクタリング前

```
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
```

リファクタリング後

```
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。

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: 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
```

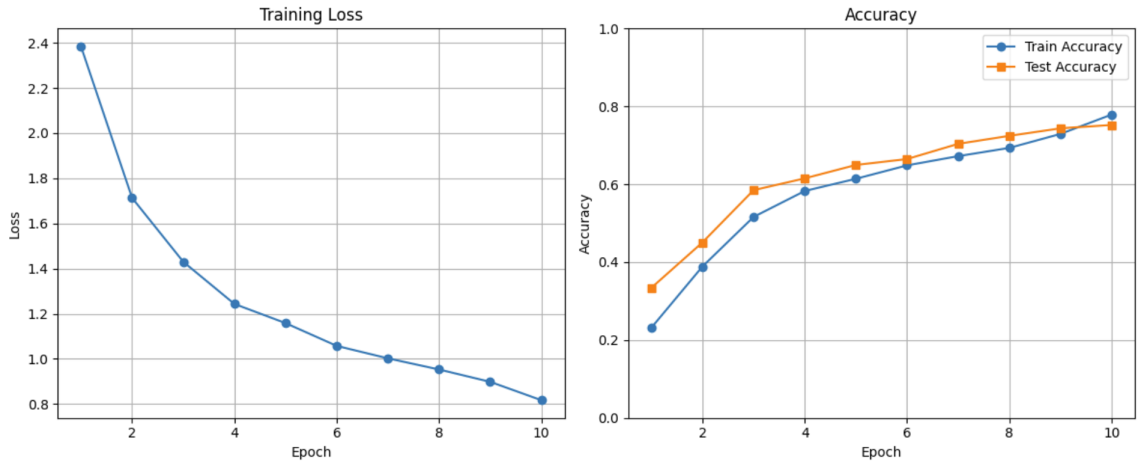

図 6　4-4(慣性項) プロット結果 (ロードなし、Dropout = 10)

```
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
```

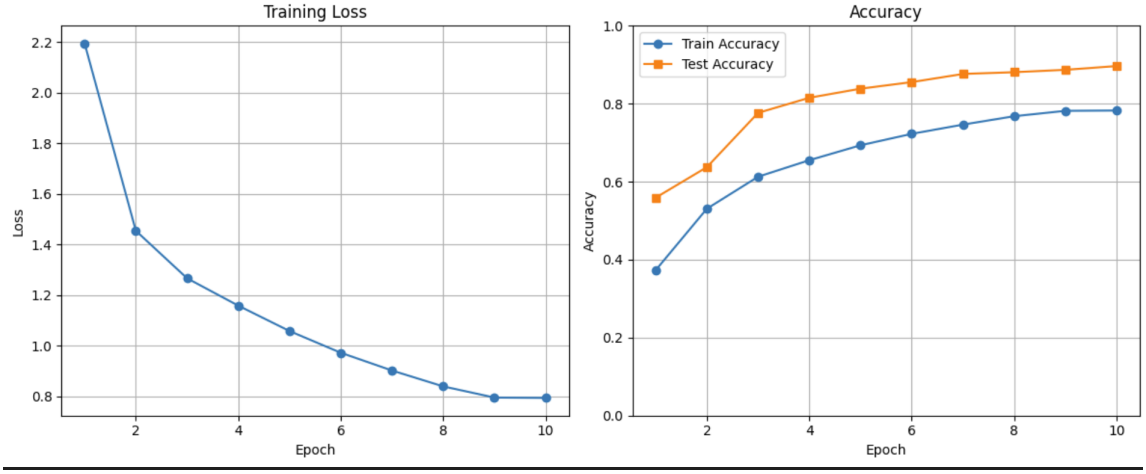

図 7　4-4(慣性項) プロット結果 (ロードなし、Dropout = 30)

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。

3.3　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。