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 17 日

課題内容

[課題 2] のコードをベースに，3 層ニューラルネットワークのパラメータ $W^{(1)}, W^{(2)}, b^{(1)}, b^{(2)}$ を学習するプログラムを作成せよ．

- ネットワークの構造，バッチサイズは課題 2 と同じで良い．
- 学習には MNIST の学習データ 60,000 枚を用いること．
- 繰り返し回数は自由に決めて良い．教師データの数を N，ミニバッチのサイズを B としたとき，N/B 回の繰り返しを 1 エポックと呼び，通常はエポック数とミニバッチサイズで繰り返し回数を指定する．
- 1 エポック内の処理では学習データ 60,000 枚を重複なく利用すること．（図 10 参照．単純にミニバッチをランダムに取得する課題 2 のコードを N/B 回実行すればよいわけではない．）
- 学習率 η は自由に決めて良い（0.01 あたりが経験的に良さそう）
- 各エポックの処理が終了する毎に，1 エポック内の平均クロスエントロピー誤差，学習データに対する正答率，及びテストデータに対する正答率を標準出力に出力すること（これにより学習がうまく進んでいるかどうか，どの程度で学習を打ち切って良さそうか，を確認することができる）．この時の出力は，単に 1 エポック内の最後のミニバッチに対するクロスエントロピー誤差や正答率とはしないこと．これらの 3 種の値については，値を保存しておき，エポック毎にプロットすることを薦める．
- 学習終了時に学習したパラメータをファイルに保存する機能を用意すること．フォーマットは自由．パラメータを numpy の配列 (ndarray) で実装している場合は numpy.save() や numpy.savez() 関数が利用できる．
- ファイルに保存したパラメータを読み込み，再利用できる機能を用意すること．（numpy.load() 関数が利用できる）

1　作成したプログラムの説明

このプログラムは、誤差逆伝播法による 3 層ニューラルネットワークの学習を実装したものである。今回は新たに以下の二つの関数を実装した。

- get_shaffled_index：学習データ分の長さの、0 から始まるインデックスの配列を生成し、そのインデックスをシャッフルしその配列を返す関数。課題 2 では、学習データからランダムに 100 枚を選んで用いたが、課題 3 では、学習データ全てを重複なく用いるため、この関数を用いて、ループごとにランダムなデータを用いることを実現した。
- get_accuracy：正答率を計算する関数。順伝播の出力から予測結果を取得する get_predicted_class 関数をすでに実装していたので、この関数では、順伝播の出力と正解ラベルを受け取り、予測結果と正解ラベルが一致した個数を計算し、それを配列の長さで割ることで正答率を出力している。

今回、中間層の個数は 10 個、学習率は 0.01、エポック数は 10、バッチサイズは 100 としている。for 文を用いてループ処理を行い、1 エポック毎に (学習データ ÷ バッチサイズ) 回のループを更に行っている。1 エポック毎の実行の流れとしては、get_shaffled_index 関数から出力された配列から先頭の値をバッチサイズ分取り出す。それを用いて、順伝播、クロスエントロピー誤差を計算する。その後、資料の通り実装した逆伝播、パ

ラメータ更新を行う。1 エポックにおける学習のループが終了すると、get_accuracy 関数を用いて正答率を計算し、その結果を配列に保存する。また、平均クロスエントロピー誤差、正答率を標準出力に表示させる。すべてのループが終了すると、matplotlib.pyplot を用いて、それらをプロットしたものを表示する。更新されたパラメータは自動でファイルに保存される。このプログラムの実行時、’ロードしますか？ yes or no: ’と表示され、yes と入力すると、保存されたパラメータをロードし、それを初期値とする。

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: `1.6683479264819345`
　テストデータに対する正答率: `0.7029`
　学習データに対する正答率: `0.6917`
2 **エポック目**
　平均クロスエントロピー誤差: `1.290009013291961`
　テストデータに対する正答率: `0.7726`
　学習データに対する正答率: `0.7649833333333333`
...
10 **エポック目**
　平均クロスエントロピー誤差: `0.6334008819285666`
　テストデータに対する正答率: `0.8411`
　学習データに対する正答率: `0.8377833333333333`

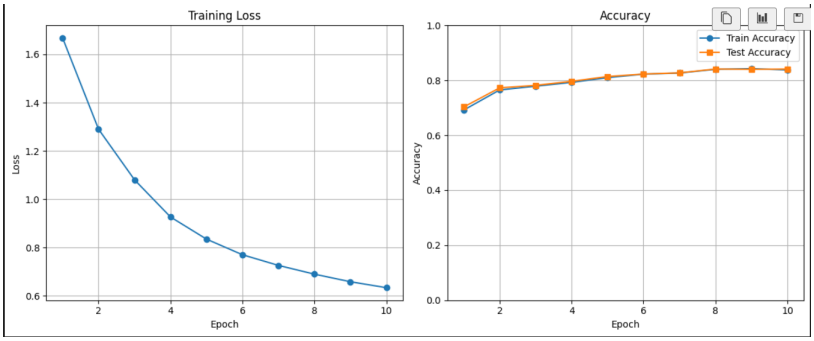

図 1　プロット結果 (ロード無し)

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: `0.6074049847542984`
　テストデータに対する正答率: `0.858`

学習データに対する正答率：0.8572166666666666

2 エポック目

平均クロスエントロピー誤差：0.5954952880557283

テストデータに対する正答率：0.8582

学習データに対する正答率：0.8531333333333333

...

10 エポック目

平均クロスエントロピー誤差：0.5157515634899839

テストデータに対する正答率：0.8636

学習データに対する正答率：0.8630166666666667

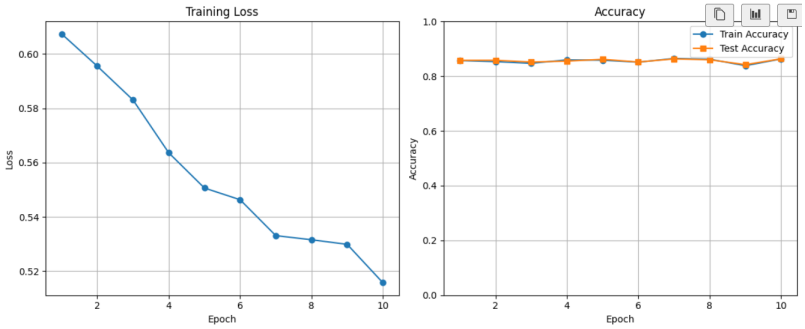

図 2　プロット結果 (ロード有り)

実行結果より、パラメータをロードしなかった場合、エポックを重ねるごとに訓練誤差が減少し、訓練データ・テストデータ双方の正答率が順調に向上しており、学習が正しく進んでいることが確認できる。一方、ロードした場合は、学習済みの状態から開始するため初期精度が高く、さらに学習を進めることで正答率が 86% 以上に達した。これにより、パラメータの保存・読み込み機能も正しく動作していることがわかる。

3　工夫点

get_shaffled_index 関数により出力された配列の先頭から順に値を取り出し、重複なく学習を行えるよう実装したのは工夫した点であるといえる。また、すでに実装していた get_predicted_class 関数を再利用し、記述量を削減した。ただ、正答率計算の際、データの形を整え、順伝播を行った出力を関数は入力として受け取るが、そうするのであれば、データと正解ラベルを直接受け取り、正答率計算の関数内で順伝播も行うように実装した方が、より簡潔になったかもしれない。

4　問題点

今回のプログラムを実行すると、RuntimeWarning: overflow encountered in exp return 1 / (1 + np.exp(-x)) という警告文が出てしまった。これは、シグモイド関数の出力に対する警告文である。また、実行結果を見ると、ロードを行った方は、途中で正答率が逆に下がっている部分がある。今回ロードしたパラメータはすでに何回か学習していたものであるので、過学習が起き始めている可能性がある。このプログラムでは、保存

するファイル、読み込むファイルが固定されているため、中間層の数や学習率を変化させて学習した場合でもパラメータが更新されてしまう。そのため、ロードするか否かの外部入力だけでなく、ファイルを指定できるようにし、また、パラメータの削除 (初期化) まで実装すればさらに良いプログラムになると感じた。

5 リファクタリング

リファクタリングには、Gemini(2.5 Pro) を用いた。

- プロンプト: (コード貼り付け) これは、誤差逆伝播法による 3 層ニューラルネットワークの学習を表している。逆伝播、正答率計算部分のコードを、関数を定義して、現在記述されている関数及びコードを用いて簡潔にまとめろ。
- 目的: 関数をまとめて再定義し、メイン処理部分の記述量を削減しコードの可読性を高める、また、自分が認識していない論理エラーや冗長性を発見すること。

このリファクタリングにより、メインループ内の逆伝播とパラメータ更新のロジックが backward_propagation_and_update 関数に、性能評価のロジックが evaluate_model 関数にそれぞれ整理された。以下にメインループ部分のリファクタリング前後を示す。

リファクタリング前 (メイン処理部分):

```
# --- 逆伝播 ---
# ソフトマックス関数とクロスエントロピー誤差の合成関数の微分,
誤差 (予測-正解) をバッチサイズで割ったもの (バッチサイズ, 10)
dEn_dak = (output_probabilities - one_hot_labels) / batch_size
# 中間層の出力 (h_out) に対する勾配 (バッチサイズ, 10) @ (10, 100) ->
(バッチサイズ, 100)
dEn_dX = np.dot(dEn_dak, weight2)
# weight2 の勾配 (10, バッチサイズ) @ (バッチサイズ, 100) -> (10, 100)
dEn_dW_1 = np.dot(dEn_dak.T, hidden_layer_output)
# bias2 の勾配 (バッチサイズ, 10) の行列を列ごとに合計 -> (10,)
dEn_db_1 = np.sum(dEn_dak, axis = 0)
# 中間層の入力に対する勾配, シグモイド関数の微分の値 y*(1-y) を乗算する
(バッチサイズ, 100) * (バッチサイズ, 100) -> (バッチサイズ, 100).. 要素ごとの積
dEn_dX_sig = dEn_dX * (hidden_layer_output * (1 - hidden_layer_output))

# weight1 の勾配 (100, バッチサイズ) @ (バッチサイズ, 784) -> (100, 784)
dEn_dW_2= np.dot(dEn_dX_sig.T, batch_image_vector)
# bias1 の勾配 (バッチサイズ, 100) の行列を列ごとに合計 -> (100,)
dEn_db_2 = np.sum(dEn_dX_sig, axis=0)

weight1 -= dEn_dW_2 * learning_rate
```

```
            bias1 -= dEn_db_2 * learning_rate
            weight2 -= dEn_dW_1 * learning_rate
            bias2 -= dEn_db_1 * learning_rate

        num_batches = len(train_images) // batch_size

        train_loss_list.append(error_sum / num_batches)

        train_images_vector = train_images.reshape(len(train_images), -1)
        train_prob, _ = forward_propagation(train_images_vector, weight1, bias1, weight2, bias2)
        train_accuracy = get_accuracy(train_prob, train_labels)
        train_acc_list.append(train_accuracy)

        test_images_vector = test_images.reshape(len(test_images), -1)
        test_prob, _ = forward_propagation(test_images_vector, weight1, bias1, weight2, bias2)
        test_accuracy = get_accuracy(test_prob, test_labels)
        test_acc_list.append(test_accuracy)
```

リファクタリング後:
関数再定義

```
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。
    更新後のパラメータを返す。
    """
    current_batch_size = batch_image_vector.shape[0]

    # --- 逆伝播 ---
    dEn_dak = (output_probabilities - one_hot_labels) / current_batch_size
    dEn_dX = np.dot(dEn_dak, weight2)
    dEn_dW_1 = np.dot(dEn_dak.T, hidden_layer_output)
    dEn_db_1 = np.sum(dEn_dak, axis = 0)
    dEn_dX_sig = dEn_dX * (hidden_layer_output * (1 - hidden_layer_output))
    dEn_dW_2= np.dot(dEn_dX_sig.T, batch_image_vector)
    dEn_db_2 = np.sum(dEn_dX_sig, axis=0)

    # --- 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

def calculate_accuracy(images, labels, weight1, bias1, weight2, bias2):
    """
    指定されたデータセットに対するモデルの正答率を計算する関数。
    """
    # 1. データ全体を（枚数，784）のベクトルに変換
    images_vector = images.reshape(len(images), -1)

    # 2. 順伝播で予測確率を計算
    probabilities, _ = forward_propagation(images_vector, weight1, bias1, weight2, bias2)

    # 3. 正答率を計算
    accuracy = get_accuracy(probabilities, labels)

    return accuracy
```

メイン処理部分

```
            # --- 逆伝播 ---
            weight1, bias1, weight2, bias2 = backward_propagation_and_update(
                batch_image_vector, hidden_layer_output, output_probabilities,
                one_hot_labels,
                weight1, bias1, weight2, bias2, learning_rate
            )

        train_accuracy = calculate_accuracy(train_images, train_labels,
        weight1, bias1, weight2, bias2)
        test_accuracy = calculate_accuracy(test_images, test_labels,
        weight1, bias1, weight2, bias2)

        num_batches = len(train_images) // batch_size
        train_loss_list.append(error_sum / num_batches)
        train_acc_list.append(train_accuracy)
        test_acc_list.append(test_accuracy)
```