



Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Московский государственный технический университет
имени Н.Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ _____ «Информатика и системы управления»

КАФЕДРА _____ «Теоретическая информатика и компьютерные технологии»

Лабораторная работа № 2
по курсу «Теория искусственных нейронных сетей»
«Разработка многослойного персептрона на основе обратного
распространения ошибки FFNN»

Студент группы ИУ9-71Б Баев Д.А

Преподаватель Каганов Ю. Т.

Москва 2023

1 Задание

1. Реализовать на языке высокого уровня многослойный персептрон и проверить его работоспособность на примере данных, выбранных из MNIST dataset.
2. Исследовать работу персептрона на основе использования различных целевых функций. (среднеквадратичная ошибка, перекрестная энтропия, дивергенция Кульбака-Лейблера).
3. Исследовать работу многослойного персептрона с использованием различных методов оптимизации (градиентный, Флетчера-Ривза (FR), Бroyдена-Флетчера-Гольдфарба-Шенно (BFGS)).
4. Провести исследование эффективности работы многослойного персептрона при изменении гиперпараметров (количества нейронов и количества слоев).
5. Подготовить отчет с распечаткой текста программы, графиками результатов исследования и анализом результатов.

2 Исходный код

Исходный код программы представлен в листингах 1- 6

Листинг 1: Подготовка датасета

```
1 import numpy as np
2 import pandas as pd
3 import matplotlib.pyplot as plt
4 import torchvision
5 from tqdm import tqdm
6
7 MNIST_train = torchvision.datasets.MNIST('./', download=True, train=True
8     )
9 MNIST_test = torchvision.datasets.MNIST('./', download=True, train=False
10    )
11
12 count = 480
13 count_test = 80
14
15 train_X = MNIST_train.data.numpy()[:count]
16 train_Y = MNIST_train.targets.numpy()[:count]
17 test_X = MNIST_test.data.numpy()[:count_test]
18 test_Y = MNIST_test.targets.numpy()[:count_test]
19
20 train_X = np.array(list(map(lambda x: x.flatten() / 256, train_X)))
21 train_Y = np.array([np.array([int(i == x) for i in range(10)]) for x in
22     train_Y])
23 test_X = np.array(list(map(lambda x: x.flatten() / 256, test_X)))
24 test_Y = np.array([np.array([int(i == x) for i in range(10)]) for x in
25     test_Y])
```

Листинг 2: Определение функций активации и функций ошибки

```
1 def softmax(x):
2     if np.linalg.norm(x) < 0.001:
3         return np.zeros(len(x))
4     x = x / np.linalg.norm(x)
5     return np.exp(x)/(np.exp(x)).sum() if (np.exp(x)).sum() > 0.01 else
6     np.zeros(len(x))
7
8 def relu(x):
9     return np.maximum(0, x)
10
11 def relu_derivative(x):
12     return np.where(x > 0, 1, 0)
```

```

13 def mse(y_true, y_pred):
14     return np.sum((y_true - y_pred) ** 2) / len(y_true)
15
16 def mse_derivative(y_true, y_pred):
17     return 2 * (y_pred - y_true) / len(y_true)
18
19 def sigmoid(x): return 1/(1+np.exp(-x))
20 def sigmoid_derivative(x): return sigmoid(x)*(1-sigmoid(x))
21
22 def cross_entropy(y_true, y_pred):
23     y_pred = np.clip(y_pred, 1e-8, 1 - 1e-8)
24     return -np.mean(y_true * np.log(y_pred))
25
26 def cross_entropy_derivative(y_true, y_pred):
27     y_pred = np.clip(y_pred, 1e-8, 1 - 1e-8)
28     res = y_pred - y_true
29     return res / np.linalg.norm(res)
30
31 def kl_divergence(y_true, y_pred):
32     y_true = np.clip(y_true, 1e-8, 1 - 1e-8)
33     y_pred = np.clip(y_pred, 1e-8, 1 - 1e-8)
34     return np.mean(y_true * np.log(y_true / y_pred))
35
36 def kl_divergence_derivative(y_true, y_pred):
37     y_pred = np.clip(y_pred, 1e-8, 1 - 1e-8)
38     res = y_pred - y_true
39     return res / np.linalg.norm(res)

```

Листинг 3: Класс линейного слоя

```

1 class LinearLayer:
2     def __init__(self, input_size, output_size, optimizer, model):
3         self.hessian = None
4         self.previous_weights = None
5         self.previous_grad = None
6         self.inputs = None
7         self.weights = np.random.rand(input_size + 1, output_size) - 0.5
8         self.optimizer = optimizer
9         self.model = model
10
11     def forward(self, inputs, train=True):
12         inputs = np.append([1], inputs)
13
14         if train:
15             self.inputs = inputs
16         return inputs @ self.weights
17

```

```

18 def calculate_forward(self):
19     return self.model.calculate_forward(self, self.inputs[1:])
20
21 def backward(self, grad):
22     accum_grad = (grad @ self.weights.T)[1:]
23     step_grad = np.outer(self.inputs, grad)
24     if np.linalg.norm(step_grad) != 0:
25         step_grad /= np.linalg.norm(step_grad)
26     step = None
27     match self.optimizer:
28         case 0:
29             # GD
30             step = step_grad
31         case 1:
32             # FR
33             if self.previous_grad is None:
34                 self.previous_grad = step_grad
35                 w = 0
36             else:
37                 grad = step_grad.flatten()
38                 previous_grad = self.previous_grad.flatten()
39                 w = max((np.linalg.norm(grad)/np.linalg.norm(
previous_grad)) ** 2, 0)
40                 if np.isnan(w) or w == np.Inf:
41                     w = 0
42                 step = (step_grad + w * self.previous_grad)
43                 self.previous_grad = step_grad
44         case 2:
45             # BFGS
46             if self.hessian is None:
47                 self.hessian = np.identity(len(step_grad))
48                 self.previous_grad = np.zeros(step_grad.shape)
49                 self.previous_weights = np.zeros(self.weights.shape)
50
51             step = self.hessian @ step_grad
52
53             s = self.weights - self.previous_weights
54             y = step_grad - self.previous_grad
55
56             w = np.identity(len(step_grad)) - s @ y.T
57             self.hessian = w @ self.hessian @ w.T
58
59             self.previous_grad = step_grad
60             self.previous_weights = self.weights
61
62

```

```

63         step *= self.model.lr
64
65         before = self.calculate_forward()
66         self.weights -= step
67         after = self.calculate_forward()
68         if before > after and abs(after - before) > 0.0001:
69             while before > after and abs(after - before) > 0.0001:
70                 self.weights -= step
71                 before = after
72                 after = self.calculate_forward()
73             self.weights += step
74         return accum_grad

```

Листинг 4: Класс слоя активации

```

1 class ActivationLayer:
2     def __init__(self, activation, activation_derivative):
3         self.inputs = None
4         self.activation = activation
5         self.activation_derivative = activation_derivative
6     def forward(self, inputs, train=True):
7         self.inputs = inputs
8         return self.activation(inputs)
9     def backward(self, grad):
10        return grad * self.activation_derivative(self.inputs)

```

Листинг 5: Класс перцептрона

```

1 class Perceptron:
2     def __init__(self, input_size, sizes, loss, loss_derivative,
3         optimizer, lr):
4         self.last_true = None
5         self.layers = []
6         prev_size = input_size
7         for size in sizes:
8             self.layers.append(LinearLayer(prev_size, size, optimizer,
9                 self))
10            self.layers.append(ActivationLayer(sigmoid,
11                sigmoid_derivative))
12            prev_size = size
13            self.layers.append(LinearLayer(prev_size, 10, optimizer, self))
14            self.layers.append(ActivationLayer(softmax, lambda x: softmax(x)
15                * (1 - softmax(x))))
16        self.loss = loss
17        self.loss_derivative = loss_derivative
18        self.lr = lr

```

```

16 def forward(self, inputs, train=True):
17     result = inputs
18     for layer in self.layers:
19         result = layer.forward(result, train)
20     return result
21
22 def backward(self, y_true, y_pred):
23     grad = self.loss_derivative(y_true, y_pred)
24     for layer in self.layers[::-1]:
25         grad = layer.backward(grad)
26
27 def fit(self, inputs, y_true):
28     self.last_true = y_true
29     y_pred = self.forward(inputs, True)
30     loss = self.loss(y_true, y_pred)
31     self.backward(y_true, y_pred)
32     return y_pred, loss
33
34 def calculate(self, layer, inputs):
35     place = self.layers.index(layer)
36     result = inputs
37     for layer in self.layers[place:]:
38         result = layer.forward(result, False)
39     return self.loss(self.last_true, result)
40
41
42 def train(self, epochs):
43     accuracy = []
44     loss_arr = []
45     for _ in tqdm(range(epochs)):
46         running_accuracy = 0
47         running_loss = 0
48         for inputs, y_true in zip(train_X, train_Y):
49             y_pred, loss = self.fit(inputs, y_true)
50             pred = np.argmax(y_pred)
51             running_loss += loss
52             running_accuracy += (np.argmax(y_true) == pred)
53         accuracy.append(running_accuracy / len(train_X))
54         loss_arr.append(running_loss / len(train_X))
55     return accuracy, loss_arr
56
57 def validate(self):
58     running_accuracy = 0
59     running_loss = 0
60     for inputs, y_true in zip(test_X, test_Y):
61         y_pred = self.forward(inputs, False)

```

```

62         loss = self.loss(y_true, y_pred)
63         pred = np.argmax(y_pred)
64         running_loss += loss
65         running_accuracy += (np.argmax(y_true) == pred)
66     return running_loss / len(test_X), running_accuracy / len(test_X
    )

```

Листинг 6 — Класс эксперимента

```

1 def experiment(learning_rate, layer_count, layer_neurons, epochs,
2   optimizer, loss, loss_der):
3     perceptron = Perceptron(28 * 28, [layer_neurons for _ in range(
4       layer_count)], loss, loss_der, optimizer, learning_rate)
5
6     accuracy, loss = perceptron.train(epochs)
7
8     plt.plot(np.arange(len(accuracy)), accuracy)
9     plt.title("Accuracy")
10    plt.show()
11    plt.plot(np.arange(len(loss)), loss)
12    plt.title("Loss")
13    plt.show()
14    print(perceptron.validate())

```

3 Результаты

В качестве начального эксперимента были выбраны следующие параметры: learning rate - 0.009, количество скрытых слоев (без учета входного и выходного слоев) - 1, количество нейронов в скрытом слое - 64, количество эпох - 40, функция потерь - перекрестная энтропия, оптимизатор - градиентный.

График точности этой модели от количества эпох приведен на рисунке 1

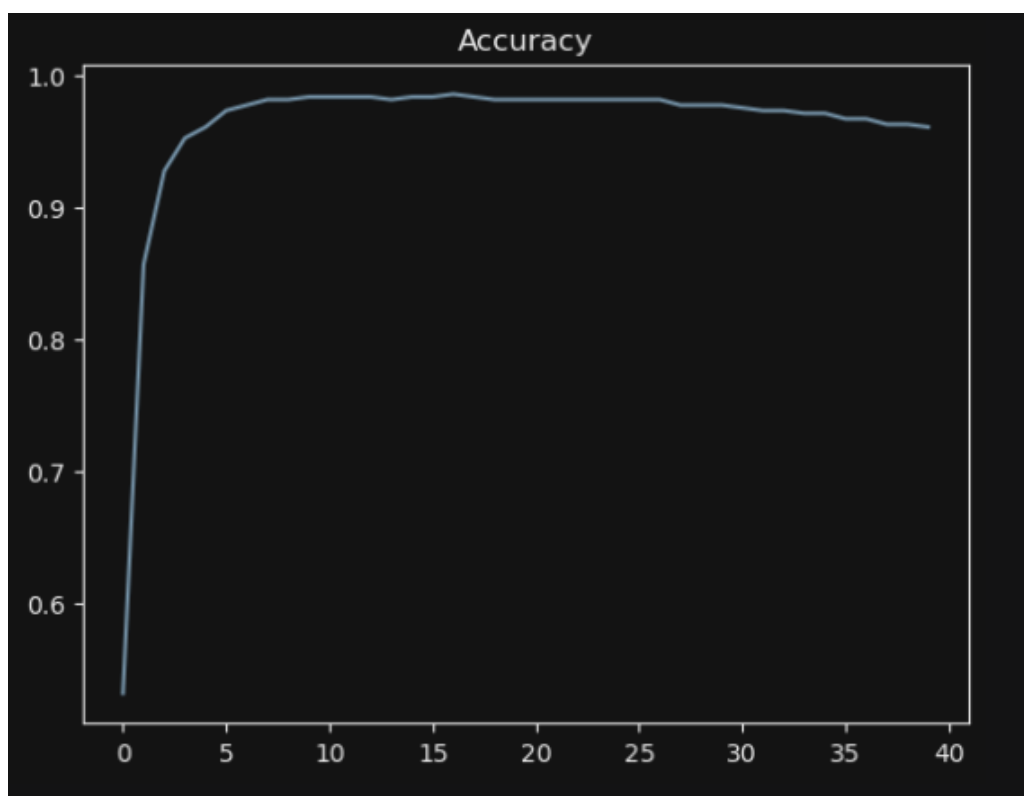


Рис. 1

Теперь функция потерь изменяется на среднеквадратичную ошибку. График точности измененной модели от количества эпох приведен на рисунке 2. Видно серьезное ухудшение результата.

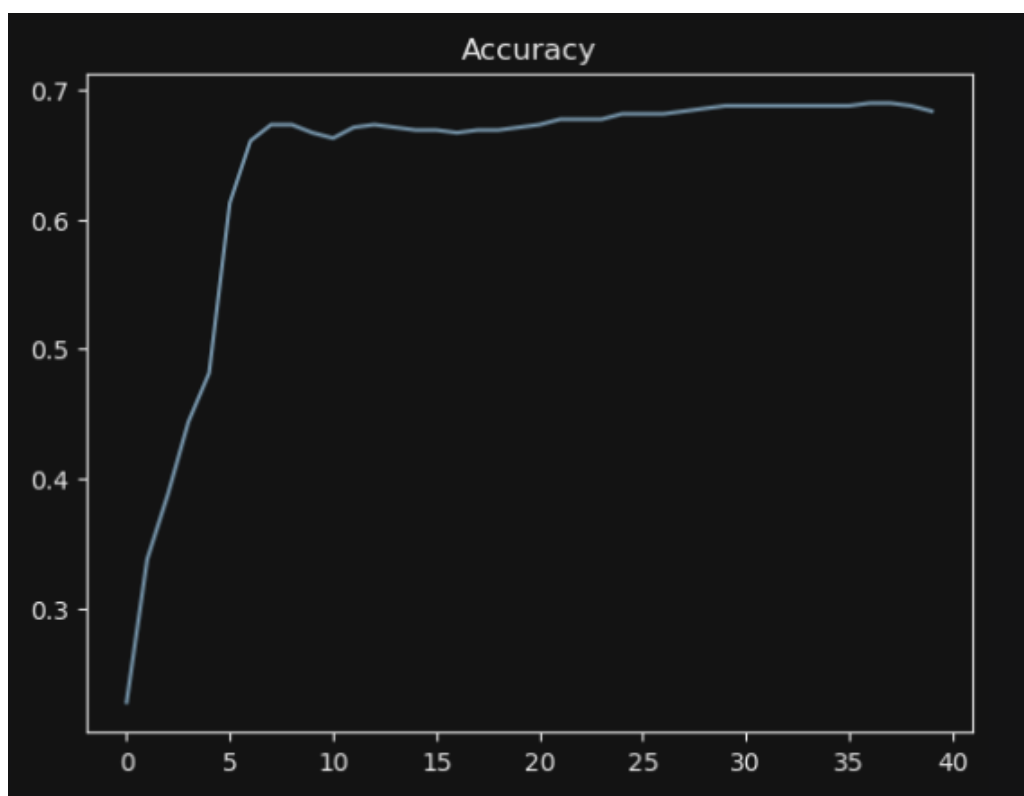


Рис. 2

Теперь функция потерь изменяется на дивергенцию Кульбака-Лейблера. График точности измененной модели от количества эпох приведен на рисунке 3. Результат примерно схож с перекрестной энтропией.

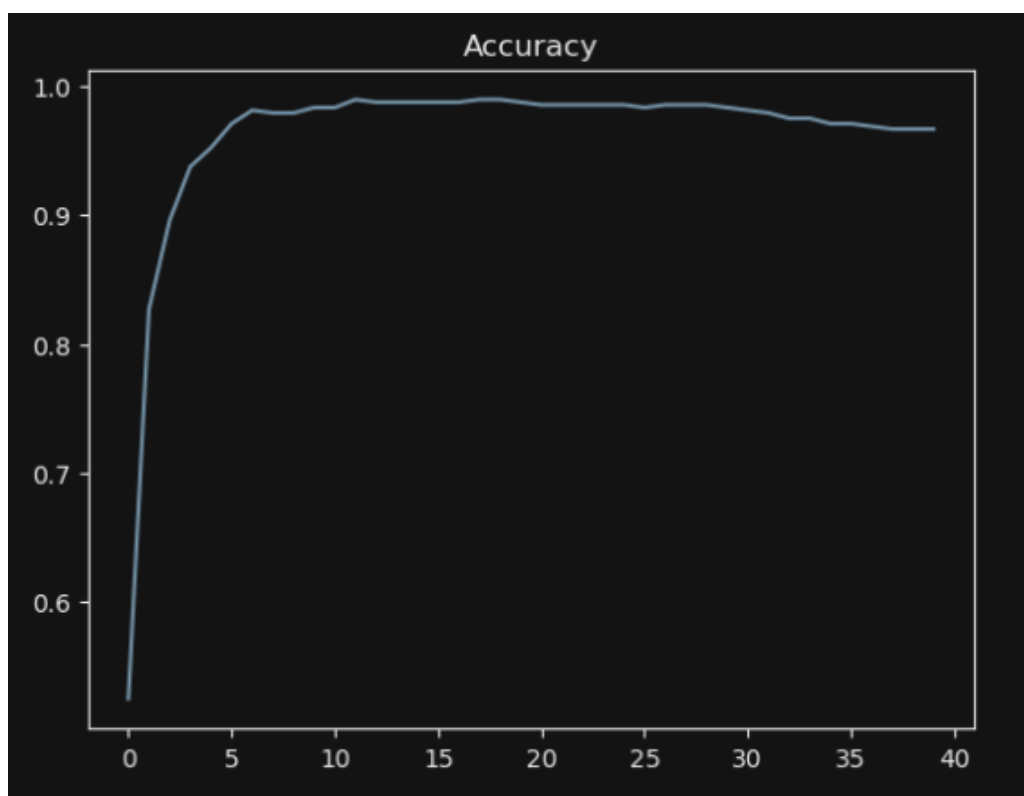


Рис. 3

Теперь в качестве функции потерь была возвращена перекрестная энтропия. Число скрытых слоев теперь - 0. График точности измененной модели от количества эпох приведен на рисунке 4. Видно ухудшение результата и расходимость обучения при большем количестве эпох.

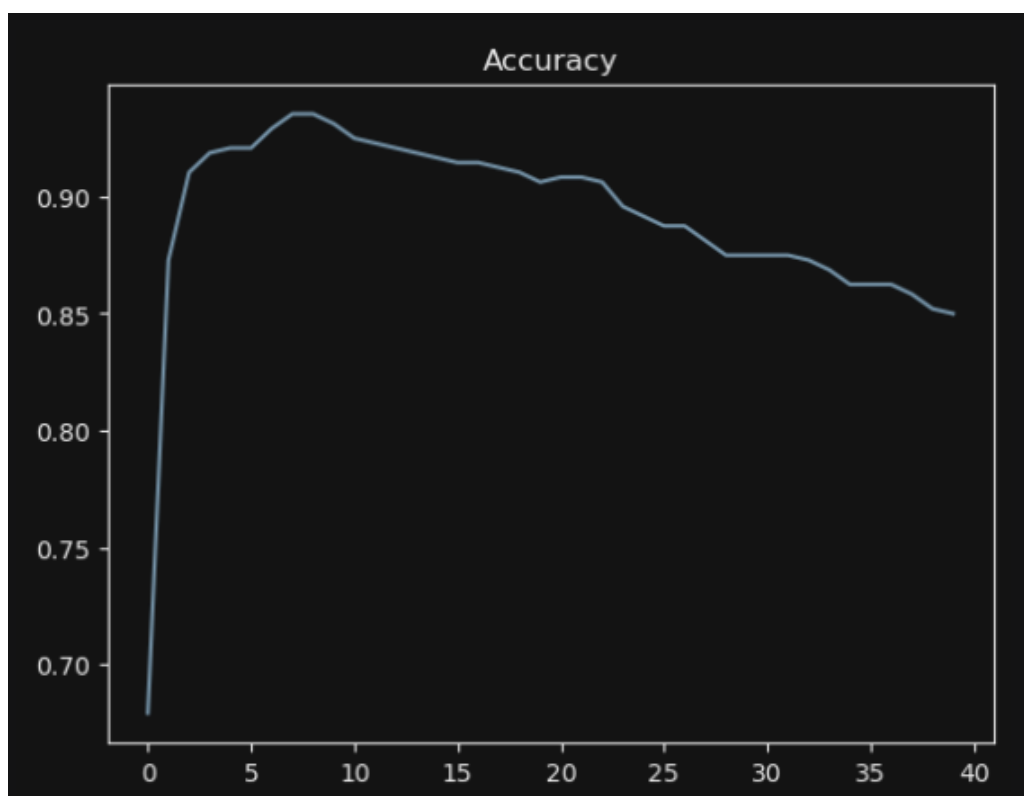


Рис. 4

Число скрытых слоев теперь - 2. График точности измененной модели от количества эпох приведен на рисунке 5. Видно ухудшение по сравнению с результатом с одним скрытым слоем.

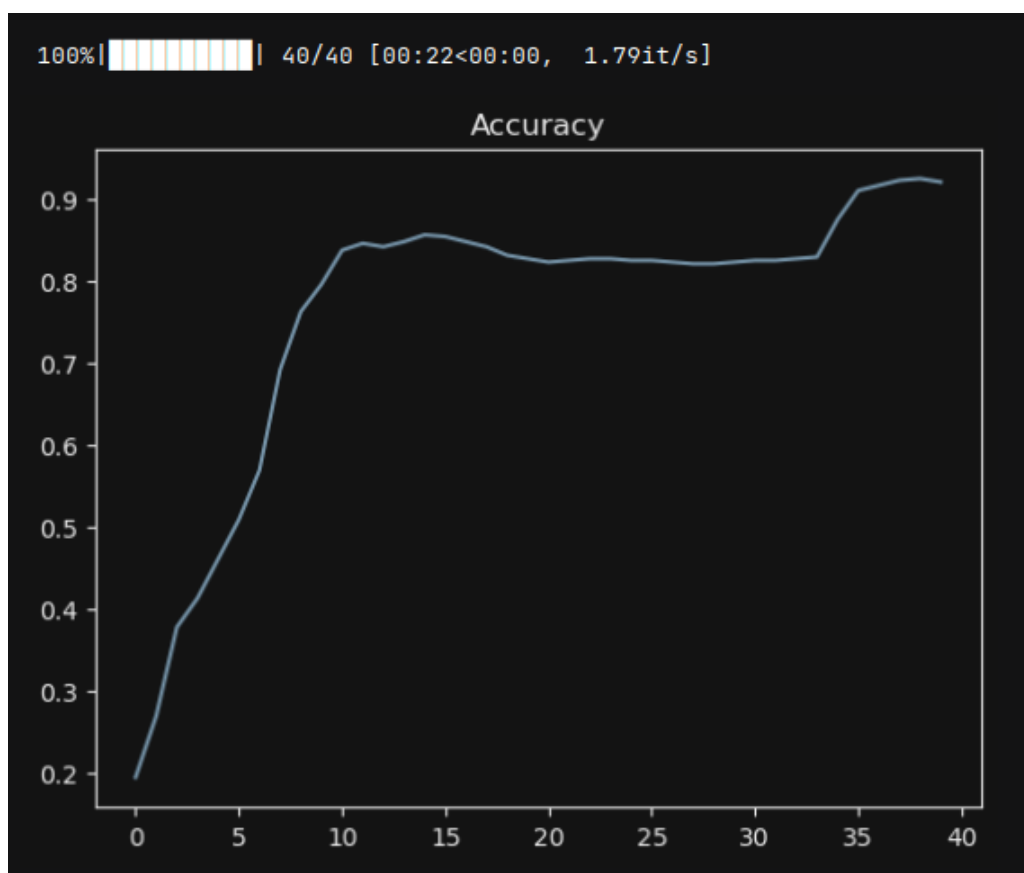


Рис. 5

Число скрытых слоев теперь - 1. Число нейронов в скрытом слое теперь - 128. График точности измененной модели от количества эпох приведен на рисунке 6. Результат примерно такой же, как и при 64 нейронах.

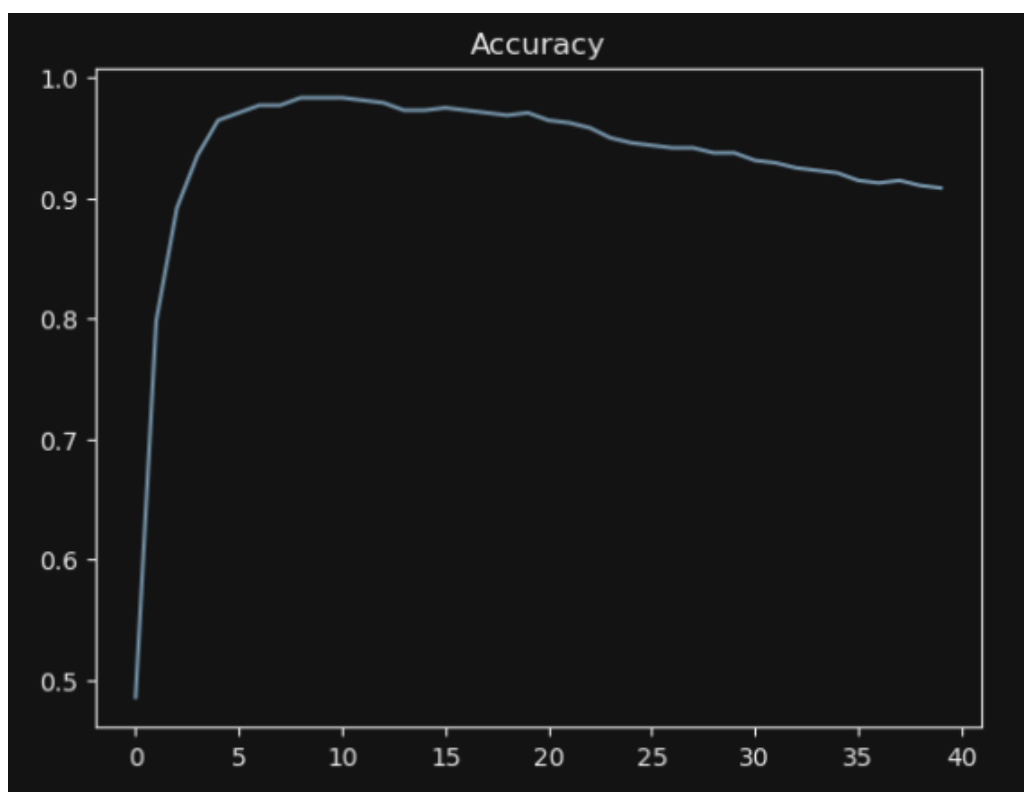


Рис. 6

Теперь все значения такие же, как и в первом эксперименте. В качестве оптимизатора был выбран FR. График точности измененной модели от количества эпох приведен на рисунке 7. Видно незначительное ухудшение результата.

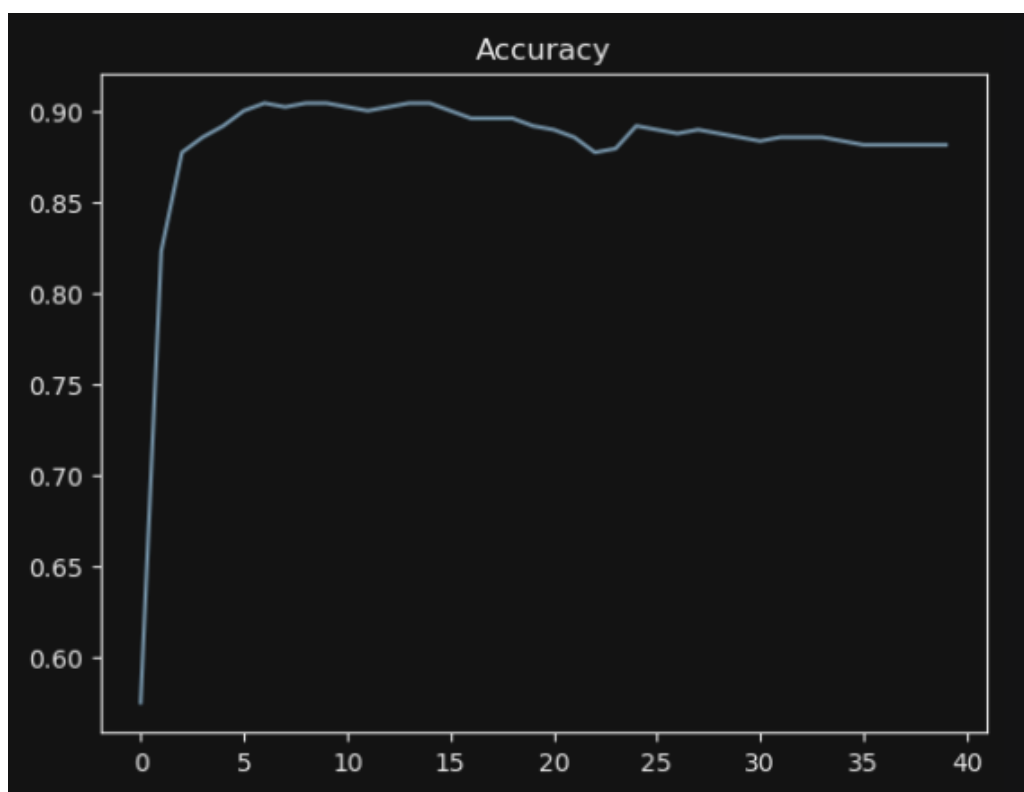


Рис. 7

Теперь в качестве оптимизатора был выбран BFGS. График точности измененной модели от количества эпох приведен на рисунке 8. Этот алгоритм оптимизации работает значительно дольше из-за обилия матричных вычислений при расчете Гессиана. Поэтому данная модель обучалась всего 5 эпох. С учетом этого факта, данный алгоритм показал достойный результат.

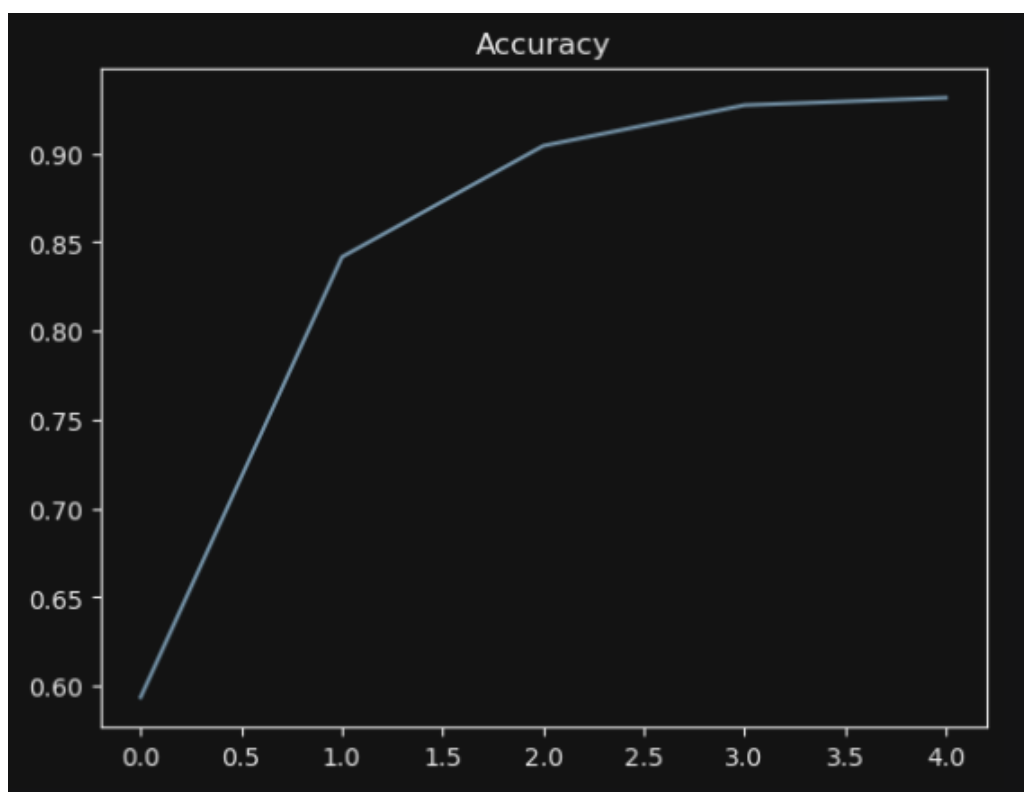


Рис. 8

В качестве финального эксперимента было проверено повышение количества скрытых слоев (3), чтобы продемонстрировать все большее ухудшение результата. График точности измененной модели от количества эпох приведен на рисунке 9.

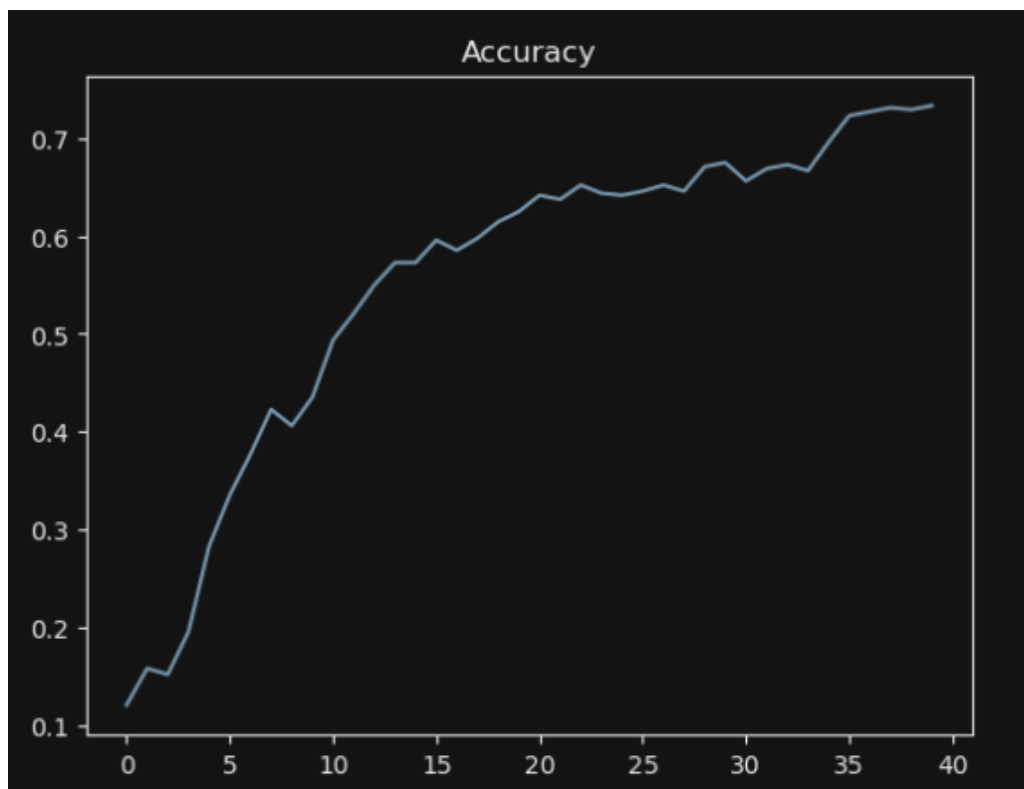


Рис. 9

4 Выводы

В рамках данной лабораторной работы был вручную реализован многослойный перцептрон, решающий задачу классификации рукописных цифр. Были реализованы различные целевые функции: среднеквадратичное отклонение, перекрестная энтропия, дивергенция Кульбака-Лейблера. Были реализованы различные методы оптимизации: градиентный, FR, BFGS. Наилучшие результаты показали перекрестная энтропия и дивергенция Кульбака-Лейбнера в сочетании с одним скрытым слоем и градиентным методом оптимизации. Также хорошо себя зарекомендовал метод BFGS, но в случае его использования происходит большой проигрыш в скорости.