## Описание метода обратного распространения ошибки

В задаче реализуется классификация изображений из датасета MNIST.

—входной вектор из 784 значений, является преобразованным изображением из датасета 28 на 28 пикселей;

— выходной вектор из 10 правильных значений;

— вектор предсказанных значений;

– матрица весов между входным и скрытым слоями;

– смещение для весов скрытого слоя;

– матрица весов между скрытым и выходным слоями;

– смещение для весов скрытого слоя.

Использовалась функция активации Softmax

и функция потерь категориальная кросс-энтропия

Алгоритм работы сети обратного распространения ошибки:

1. Прямое распространение.
2. Вычисление потерь.
3. Обратное распространение ошибки  
    – ошибка на выходе

– градиенты для весов и смещений выхожного слоя

– ошибка на скрытом слое

*–* градиенты для весов и смещений скрытого слоя

1. Обновление весов.

# Реализация программы

Для начала загрузим библиотеки необходимые для выполнения работы

import numpy as np

import matplotlib.pyplot as plt

from tensorflow.keras.datasets import mnist

from tensorflow.keras.utils import to\_categorical

* numpy для работ с матрицами
* matplotlib для работы с графиками
* tensorflow для загрузки датасета MNIST и формулы для преобразования вектора в one-hot метки

Реализация функции потерь и ошибок

def relu(self, z):

"""Функция активации ReLU"""

return np.maximum(0, z)

def relu\_derivative(self, z):

"""Производная функции ReLU"""

return np.where(z > 0, 1, 0)

def softmax(self, z):

"""Функция активации softmax с защитой от переполнения"""

exp\_z = np.exp(z - np.max(z, axis=1, keepdims=True))

return exp\_z / np.sum(exp\_z, axis=1, keepdims=True)

def categorical\_cross\_entropy(self, y\_pred, y\_true):

"""Функция потерь - категориальная кросс-энтропия"""

m = y\_true.shape[0] # количество примеров

log = -np.log(y\_pred[np.arange(m), np.argmax(y\_true, axis=1)])

loss = np.sum(log) / m

return loss

Реализуем функцию для загрузки датасета

def load\_mnist\_data():

"""Загрузка и предобработка данных MNIST"""

print("Загрузка MNIST")

(x\_train, y\_train), (x\_test, y\_test) = mnist.load\_data()

# Нормализация изображений

x\_train = x\_train.astype('float32') / 255.0

x\_test = x\_test.astype('float32') / 255.0

# Преобразование в одномерные векторы

x\_train\_flat = x\_train.reshape(x\_train.shape[0], -1)

x\_test\_flat = x\_test.reshape(x\_test.shape[0], -1)

# Преобразование меток в one-hot encoding

y\_train\_one\_hot = to\_categorical(y\_train, 10)

y\_test\_one\_hot = to\_categorical(y\_test, 10)

print(f"Размер обуч выборки: X={x\_train\_flat.shape}, Y={y\_train\_one\_hot.shape}")

print(f"Размер тест выборки: X={x\_test\_flat.shape}, Y={y\_test\_one\_hot.shape}")

return x\_train\_flat, y\_train\_one\_hot, x\_test\_flat, y\_test\_one\_hot, y\_test, x\_test

Реализация класса нейронной сети согласно описанным выше формулам

class NeuralNetwork:

"""Реализация двухслойной нейронной сети для классификации изображений MNIST"""

def \_\_init\_\_(self, input\_size, hidden\_size, output\_size):

"""Инициализация весов и смещений"""

# Инициализация весов нормальным распределением

self.W1 = np.random.randn(input\_size, hidden\_size) \* 0.01

self.b1 = np.zeros((1, hidden\_size))

self.W2 = np.random.randn(hidden\_size, output\_size) \* 0.01

self.b2 = np.zeros((1, output\_size))

# Для хранения промежуточных результатов

self.hist = {}

self.loss\_history = []

self.accuracy\_history = []

def forward(self, X):

"""Прямое распространение"""

# Скрытый слой

self.hist['Z1'] = X @ self.W1 + self.b1

self.hist['A1'] = self.relu(self.hist['Z1'])

# Выходной слой

self.hist['Z2'] = self.hist['A1'] @ self.W2 + self.b2

A2 = self.softmax(self.hist['Z2'])

self.hist['A2'] = A2

return A2

def backward(self, X, y\_true):

"""Обратное распространение ошибки"""

m = X.shape[0]

A1 = self.hist['A1']

A2 = self.hist['A2']

Z1 = self.hist['Z1']

# Градиенты выходного слоя

dZ2 = A2 - y\_true

self.dW2 = (1/m) \* (A1.T @ dZ2)

self.db2 = (1/m) \* np.sum(dZ2, axis=0, keepdims=True)

# Градиенты скрытого слоя

dA1 = dZ2 @ self.W2.T

dZ1 = dA1 \* self.relu\_derivative(Z1)

self.dW1 = (1/m) \* (X.T @ dZ1)

self.db1 = (1/m) \* np.sum(dZ1, axis=0, keepdims=True)

def update\_parameters(self, learning\_rate):

"""Обновление весов и смещений"""

self.W1 -= learning\_rate \* self.dW1

self.b1 -= learning\_rate \* self.db1

self.W2 -= learning\_rate \* self.dW2

self.b2 -= learning\_rate \* self.db2

def predict(self, X):

"""Предсказание классов"""

A2 = self.forward(X)

return np.argmax(A2, axis=1)

def predict\_proba(self, X):

"""Предсказание вероятностей классов"""

A2 = self.forward(X)

return A2

def evaluate(self, X\_test, y\_test\_orig):

"""Оценка точности модели"""

predictions = self.predict(X\_test)

accuracy = np.mean(predictions == y\_test\_orig)

return accuracy

Функция обучения модели

def train(self, X\_train, y\_train, X\_test, y\_test\_orig,

epochs=20, batch\_size=64, learning\_rate=0.1):

"""Обучение модели"""

print(f"\nНачало обучения: {epochs} эпох, размер батча {batch\_size}, скорость обучения {learning\_rate}")

for epoch in range(epochs):

epoch\_loss = 0

indices = np.random.permutation(X\_train.shape[0])

X\_train\_shuffled = X\_train[indices]

y\_train\_shuffled = y\_train[indices]

for i in range(0, X\_train.shape[0], batch\_size):

X\_batch = X\_train\_shuffled[i:i+batch\_size]

y\_batch = y\_train\_shuffled[i:i+batch\_size]

# Прямое и обратное распространение

y\_pred\_batch = self.forward(X\_batch)

loss = self.categorical\_cross\_entropy(y\_pred\_batch, y\_batch)

epoch\_loss += loss \* X\_batch.shape[0]

self.backward(X\_batch, y\_batch)

self.update\_parameters(learning\_rate)

# Расчет средней потери и точности

avg\_epoch\_loss = epoch\_loss / X\_train.shape[0]

self.loss\_history.append(avg\_epoch\_loss)

accuracy = self.evaluate(X\_test, y\_test\_orig)

self.accuracy\_history.append(accuracy)

print(f"Эпоха {epoch+1}/{epochs} - Потери: {avg\_epoch\_loss:.4f} - Точность на тесте: {accuracy:.4f}")

final\_accuracy = self.evaluate(X\_test, y\_test\_orig)

print(f"\nИтоговая точность на тестовом наборе: {final\_accuracy:.4f}")

Итоговая точность на тестовом наборе: 0.9787

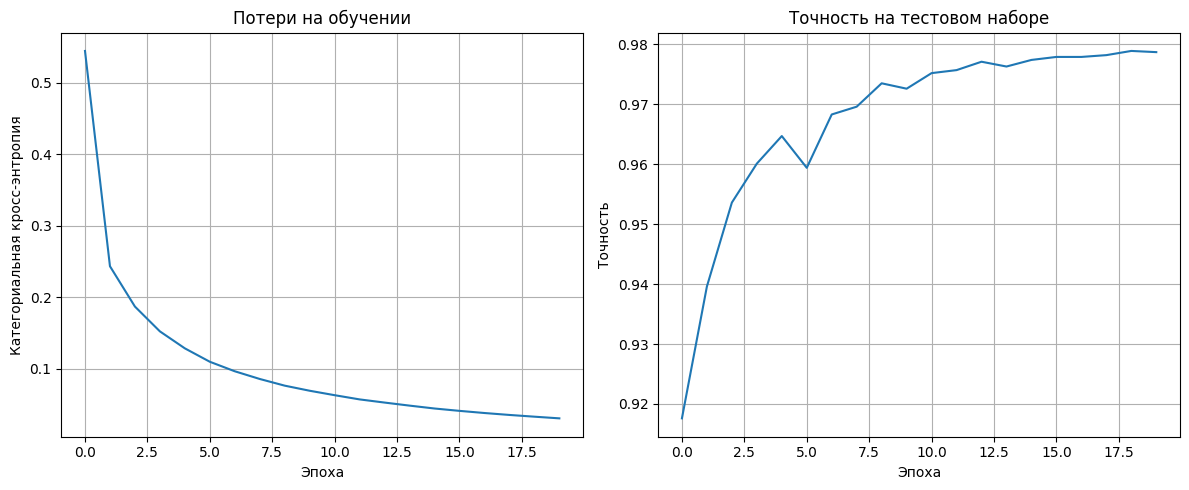


Рисунок 1– Графики обучения модели

Проверка работы нейронной сети на нескольких изображениях

def visualize\_predictions(model, x\_test, y\_test, x\_test\_flat, num\_images=10):

"""Визуализация предсказаний модели на тестовых изображениях"""

# Выбираем случайные изображения из тестового набора

indices = np.random.choice(range(len(x\_test)), num\_images, replace=False)

sample\_images = x\_test[indices]

sample\_labels = y\_test[indices]

sample\_images\_flat = x\_test\_flat[indices]

# Получаем предсказания модели

predictions = model.predict(sample\_images\_flat)

proba = model.predict\_proba(sample\_images\_flat)

# Создаем график

plt.figure(figsize=(15, 8))

rows = int(np.ceil(num\_images / 5))

for i in range(num\_images):

plt.subplot(rows, 5, i+1)

# Отображаем изображение

plt.imshow(sample\_images[i], cmap='gray')

plt.axis('off')

# Определяем цвет текста в зависимости от правильности предсказания

pred\_color = 'green' if predictions[i] == sample\_labels[i] else 'red'

# Добавляем информацию о предсказании

title = f"True: {sample\_labels[i]}\nPred: {predictions[i]}\nProba: {proba[i][predictions[i]]:.2f}"

plt.title(title, color=pred\_color, fontsize=10)

plt.tight\_layout()

plt.show()

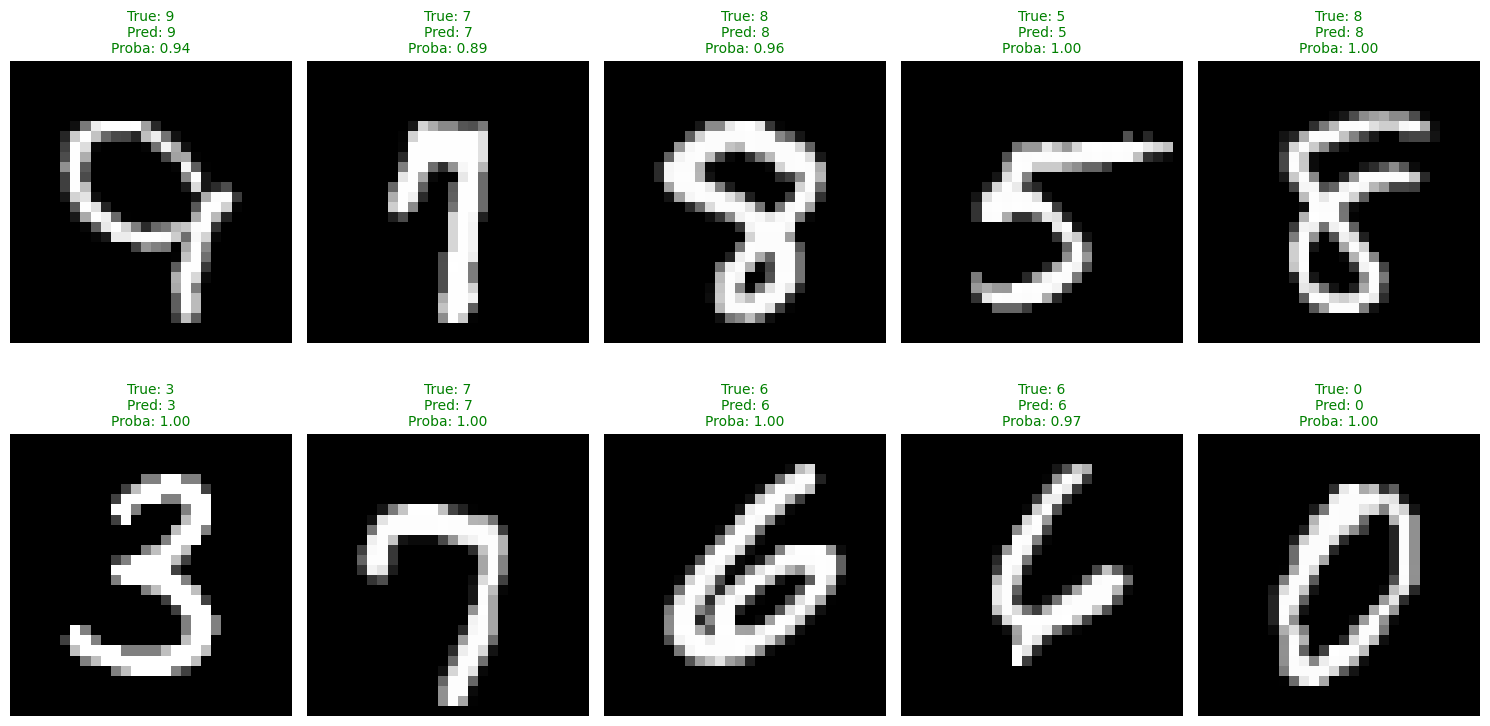


Рисунок 2 – Тестирование работы нейросети

Данный код можно запустить по ссылке <https://github.com/Aftershock280/Neuron>, скачать файл: 1Лаба.ipynb и запустить его в Google Colab.