Лабораторная работа 6

Модель классификации рукописных цифр

В этой работе вы реализуете сетевую архитектуру, позволяющую реализовать классификацию рукописных цифр. Рассматриваемая здесь архитектура полносвязной сети далека от оптимальной для решения поставленной задачи, но позволяет продемонстрировать существенные результаты, полагаясь при этом только на концепции, которые вы изучили к настоящему времени в рамках данного курса.

Архитектура сети

Рассматриваемая модель работает с изображениями цифр, где каждое изображение содержит 784 (28×28) пикселя, поэтому в нашей сети должно быть 784 входных узла $x1$ – $x784$ (см. рис. 6.1).

Входные данные поступают на скрытый слой. Для примера мы принимаем, что данный слой состоит из 25 нейронов H0 – H24. Результат, получаемый на выходе скрытого слоя, поступает на выходной слой, состоящий из десяти нейронов Y0 – Y9, по одному на каждый класс распознаваемых цифр от 0 до 9 соответственно.

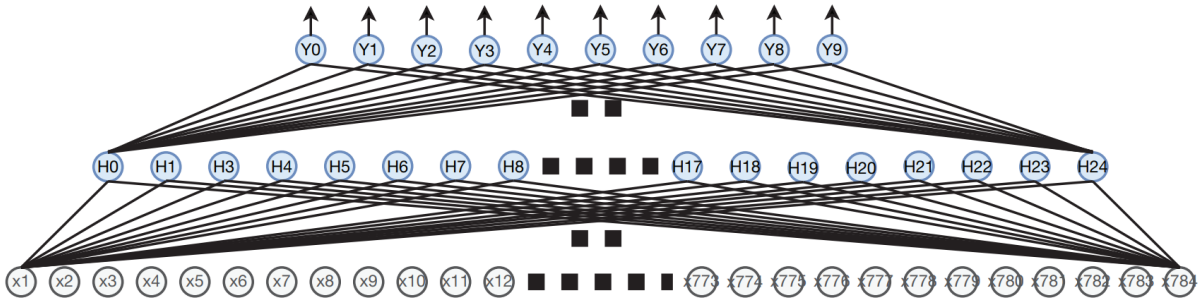

Рисунок 6.1 — Сеть для классификации цифр. Большое количество нейронов и связей между ними не показано на рисунке, чтобы сделать его менее загроможденным. В действительности каждый нейрон в слое связан со всеми нейронами в следующем слое

Мы используем функцию активации *tanh* для скрытых нейронов и логистическую сигмоидальную функцию для выходного слоя. Имея только один скрытый слой, эту сеть нельзя отнести к глубоким. Чтобы отнести сеть к DL, необходимо, чтобы она содержала по крайней мере два скрытых слоя.

Из рисунка 1.6 видно, что мы не используем в явном виде информацию о том, как пиксели связаны друг с другом в пространстве изображения. Разве не было бы полезно учитывать в каждом нейроне информацию о состоянии всех соседних пикселей? Из того, что они расположены на рисунке в виде одномерного вектора

вместо двухмерной сетки, кажется, что информация, относящаяся к тому, какие пиксели соседствуют друг с другом, теряется. Два пикселя, располагающиеся в изображении рядом друг с другом в направлении y, разделены 28 входными нейронами. Однако такое рассуждение не совсем верно. В полносвязной сети нет такого понятия, как "разделение пикселей", поскольку, например, в нашем случае все 25 скрытых нейронов «видят» все 784 пикселя, поэтому все пиксели одинаково близки друг к другу с точки зрения одного нейрона. С таким же успехом мы могли бы расположить пиксели и нейроны в двумерной сетке, но это не изменило бы фактических связей.

Однако действительно мы не сообщаем никаких предварительных знаний о том, какие пиксели соседствуют друг с другом, поэтому, если выгодно каким-то образом учитывать пространственные отношения между пикселями, то сети придется обучиться данным отношениям самостоятельно. В последующем, рассматривая свёрточные нейронные сети, вы узнаете, как проектировать модель таким образом, чтобы учитывать местоположение пикселя.

Функция потери для модели полиномиальной классификации

Когда мы решали проблему XOR, мы использовали среднеквадратичную ошибку (СКО, или MSE) в качестве функции потерь. В задаче классификации изображений можно использовать тот же подход, но с некоторыми изменениями, учитывающими, что теперь сеть имеет не один, а несколько выходов. Для этого можно определить функцию потерь (ошибок) как сумму квадратов ошибок для каждого отдельного выхода

$$Err = \frac{1}{m}\sum_{i=0}^{m-1}\sum_{j=0}^{n-1}\left(y_j^{(i)} - \hat{y}_j^{(i)}\right)^2,$$

где m — количество обучающих примеров, n — количество распознаваемых классов. То есть, в дополнение к внешней сумме по m, которая вычисляет среднее значение ошибки, теперь в формулу введена внутренняя сумма, которая суммирует квадрат ошибки для каждого выходного значения. Чтобы было предельно ясно, для одного обучающего примера мы получаем следующее

$$Err = \sum_{j=0}^{n-1}\left(y_j - \hat{y}_j\right)^2,$$

где $\hat{y}_j$ — значение на выходе нейрона Y_j. Чтобы далее прийти к упрощённому виду производную позже, поделим ошибку на 2. Минимизация функции потерь, масштабированной с коэффициентом 0,5, приведет к тому же процессу оптимизации, что и минимизация функции потерь без масштаба

$$Err: e(\hat{\boldsymbol{y}}) = \sum_{j=0}^{n-1}\frac{\left(y_j - \hat{y}_j\right)^2}{2}.$$

Здесь мы записали функцию ошибки как функцию от $\hat{\boldsymbol{y}}$, где $\hat{\boldsymbol{y}}$ теперь является вектором, потому что предполагаемая сеть имеет несколько выходных нейронов. Учитывая данную функцию потерь, мы можем вычислить значение ошибки для

каждого из n выходных нейронов, после чего алгоритм обратного распространения ничем не будет отличаться от того, как он был использован нами ранее. Следующее выражение показывает частную производную от ошибки по выходу $\hat{y}_1$ нейрона Y_1:

$$\frac{\partial e}{\partial \hat{y}_1} = \sum_{j=0}^{n-1} \frac{\partial \frac{(y_j - \hat{y}_j)^2}{2}}{\partial \hat{y}_1} = \frac{2(y_1 - \hat{y}_1)}{2} \cdot (-1) = -(y_1 - \hat{y}_1).$$

При вычислении частной производной функции потерь по конкретному выходу все остальные параметры являются константами (их производная равна 0), в результате сумма исчезает и условие ошибки для конкретного нейрона в конечном итоге оказывается таким же, как и в случае с одним выходом. То есть, функция потерь для нейрона Y_2 равна $-(y_2 - \hat{y}_2)$, или в общем случае ошибка для Y_j равна $-(y_j - \hat{y}_j)$.

Реализация модели полиномиальной классификации

Наша реализация эксперимента по классификации изображений рукописных цифр представляет собой модифицированную версию рассмотренного ранее примера обучения функции XOR ("Исключающее ИЛИ"). Единственное отличие состоит в том, что фрагмент кода 1.1 содержит несколько инициализаций, в которых указываются пути к обучающим и тестовым наборам данных. Мы также изменили значение коэффициента скорости обучения на 0,01 и ввели параметр EPOCHS — эпохи. Что такое эпоха, и почему изменилось значение коэффициента процесса обучения объясним ниже. Предполагается, что набор данных находится в каталоге .../data/ mnist/.

Фрагмент кода 1.1 — Инициализация

```
import numpy as np
import matplotlib.pyplot as plt
import idx2numpy
np.random.seed(7)
LEARNING_RATE = 0.01
EPOCHS = 20
# укажите пути к расположению обучающих и тестовых данных через
# TRAIN_IMAGE_FILENAME, TRAIN_LABEL_FILENAME,
# TEST_IMAGE_FILENAME и TEST_LABEL_FILENAME
def read_mnist():
    # добавьте в эту функцию чтение файлов с обучающими
    # и тестовыми наборами данных, их метками в train_images,
    # train_labels, test_images и test_labels
    # добавьте сюда же фрагменты кода 1.2 и 1.3, приведённые ниже
```

Далее масштабируем значения пикселей и центрируем их вокруг 0 — это называется стандартизацией данных (см. фрагмент кода 1.2). Теоретически в этом шаге нет необходимости, поскольку нейрон может принимать любое числовое

значение в качестве входных данных, но на практике такое масштабирование будет полезно. Сначала мы вычисляем среднее значение и стандартное отклонение всех обучающих значений. Мы стандартизируем данные, вычитая среднее значение из значения каждого пикселя и разделив его на стандартное отклонение.

Если вычесть среднее значение из значения каждого пикселя, новое среднее значение для всех пикселей будет равно 0. Стандартное отклонение — это мера того, насколько данные разбросаны (минимальные и максимальные значения). Деление на стандартное отклонение изменяет диапазон значений данных. Это означает, что значения данных после стандартизации будут находиться ближе к 0. В нашем случае мы начали со значениями пикселей от 0 до 255, и после стандартизации мы получим набор чисел с плавающей запятой, центрированных вокруг и намного ближе к 0.

Фрагмент кода 1.2 — Переформатирование и стандартизация данных

```
x_train = train_images.reshape(60000, 784)
mean = np.mean(x_train)
stddev = np.std(x_train)
x_train = (x_train - mean) / stddev
x_test = test_images.reshape(10000, 784)
x_test = (x_test - mean) / stddev
```

Чтобы упростить ввод входных данных в сеть, мы преобразуем изображения из двумерных в одномерные.

Фрагмент кода 1.3 — Преобразование массивов

```
y_train = np.zeros((60000, 10))
y_test = np.zeros((10000, 10))
for i, y in enumerate(train_labels):
        y_train[i][y] = 1
    for i, y in enumerate(test_labels):
        y_test[i][y] = 1
    return x_train, y_train, x_test, y_test
```

Теперь прочитаем обучающие и тестовые данные

Фрагмент кода 1.4 — Чтение данных

```
x_train, y_train, x_test, y_test = read_mnist()
index_list = list(range(len(x_train)))
```

Теперь перейдем к реализации слоев и созданию сети (фрагменте кода 1.5). Каждый нейрон скрытого слоя будет иметь 784 входа + смещение, каждый нейрон в выходном слое будет иметь 25 входов + смещение. Цикл `for` инициализирует значения, начиная с i=1 и, следовательно, не инициализирует смещение, оставляя его равным 0. Диапазон значений весов отличается от примера с XOR — теперь амплитуда значений составляет не 1, а 0.1.

Фрагмент кода 1.5 — Создание и инициализация всех нейронов в системе

```
def layer_w(neuron_count, input_count):
weights = np.zeros((neuron_count, input_count+1))
for i in range(neuron_count):
      for j in range(1, (input_count+1)):
      weights[i][j] = np.random.uniform(-0.1, 0.1)
return weights
# Объявляем матрицы и вектора, представляющие нейроны
hidden_layer_w = layer_w(25, 784)
hidden_layer_y = np.zeros(25)
hidden_layer_error = np.zeros(25)
output_layer_w = layer_w(10, 25)
output_layer_y = np.zeros(10)
output_layer_error = np.zeros(10)
```

Во фрагменте кода 1.6 приведены две функции, которые используются для индикации процесса выполнения работы модели и для визуализации процесса обучения. Функция `show_learning` вызывается несколько раз во время обучения и выводит данные о текущем процессе обучения, точности в результате тестирования, сохраняя эти значения в двух массивах.

Функция `plot_learning` вызывается в конце программы и использует полученные два массива для построения графика обучения и ошибок проверки на тестовых данных (1.0 минус ошибка) в процессе работы.

Фрагмент кода 1.6 — Функции для вывода сообщений о ходе работы в процессе обучения и тестирования

```
chart_x = []
chart_y_train = []
chart_y_test = []
def show_learning(epoch_no, train_acc, test_acc):
      global chart_x
      global chart_y_train
      global chart_y_test
      print('номер эпохи:', epoch_no, ', точность обучения: ',
      '%6.4f' % train_acc,
      ', точность тестирования: ', '%6.4f' % test_acc)
chart_x.append(epoch_no + 1)
chart_y_train.append(1.0 - train_acc)
chart_y_test.append(1.0 - test_acc)
def plot_learning():
       plt.plot(chart_x, chart_y_train, 'r-',
```

```
        label='ошибка обучения')
plt.plot(chart_x, chart_y_test, 'b-', label='ошибка тестирования')
plt.axis([0, len(chart_x), 0.0, 1.0])
plt.xlabel('эпохи обучения')
plt.ylabel('ошибка')
plt.legend()
plt.show()
```

Фрагмент кода 1.7 содержит функцию `forward_pass`, реализующую прямой проход. Функция `forward_pass` содержит два цикла. Первый обрабатывает все скрытые нейроны, представляя им одни и те же входные данные (пиксели). Также в данном цикле все выходные данные нейронов скрытого слоя вместе со значениями смещений собираются в массив, который затем может быть использован в качестве входных данных для нейронов выходного слоя. Аналогичным образом второй цикл представляет полученные выходные данные скрытого слоя нейронам выходного слоя и собирает выходные данные этого слоя в массив, который и возвращает функция `forward_pass`.

Фрагмент кода 1.7 — Функция прямого прохода при обучении сети

```
def forward_pass(x):
    global hidden_layer_y
    global output_layer_y
    # Функция активации для нейронов скрытого слоя
    for i, w in enumerate(hidden_layer_w):
        z = np.dot(w, x)
        hidden_layer_y[i] = np.tanh(z)
    hidden_output_array = np.concatenate(
        (np.array([1.0]), hidden_layer_y))
    # Функция активации для нейронов выходного слоя
    for i, w in enumerate(output_layer_w):
        z = np.dot(w, hidden_output_array)
        output_layer_y[i] = 1.0 / (1.0 + np.exp(-z))
```

Фрагмент кода 1.8 описывает функцию `back_pass`, реализующую обратный проход. Данная функция вычисляет производную функции ошибки, затем производную функции активации для выходного нейрона. Ошибка выходного нейрона вычисляется путем перемножения этих двух членов. Затем для каждого из двух нейронов скрытого слоя применяется обратное распространение ошибки путём вычисления производных их функций активации и умножения этих производных на ошибку выходного нейрона и на вес выходного нейрона.

Фрагмент кода 1.8 — Функция обратного прохода при обучении сети

```
def backward_pass(y_truth):
    global hidden_layer_error
    global output_layer_error
    # Обратное распространение ошибки для каждого выходного нейрона
    # и создание массива всех ошибок выходного нейрона.
    for i, y in enumerate(output_layer_y):
        error_prime = -(y_truth[i] - y) # Производная потерь
        derivative = y * (1.0 - y) # Производная логистической ф-ии
        output_layer_error[i] = error_prime * derivative
    for i, y in enumerate(hidden_layer_y):
        # Создание массива весов, соединяющих выход скрытого
        # нейрона i с с нейронами в выходном слое
        error_weights = []
        for w in output_layer_w:
            error_weights.append(w[i+1])
        error_weight_array = np.array(error_weights)
        # Обратное распространение для скрытых нейронов
        derivative = 1.0 - y**2 # производная ф-ии tanh
        weighted_error = np.dot(error_weight_array,
                                output_layer_error)
        hidden_layer_error[i] = weighted_error * derivative
```

Функции `forward_pass` и `back_pass` также косвенным образом определяют топологию сети.

Фрагмент кода 1.9 содержит реализацию функции `adjust_weights` корректировки значений весов. Функция корректирует веса для каждого из трёх нейронов. Коэффициент корректировки вычисляется путем умножения входного сигнала на скорость обучения и на член ошибки для данного нейрона.

Фрагмент кода 1.9 — Функция корректировки весов

```
def adjust_weights(x):
    global output_layer_w
    global hidden_layer_w
    for i, error in enumerate(hidden_layer_error):
        hidden_layer_w[i] -= (x * LEARNING_RATE
                              * error) # Обновляем все веса
    hidden_output_array = np.concatenate(
        (np.array([1.0]), hidden_layer_y))
    for i, error in enumerate(output_layer_error):
        output_layer_w[i] -= (hidden_output_array
```

```
                                * LEARNING_RATE
                                * error) # Обновляем все веса
```

Когда все эти части на месте, остается только цикл обучения, показанный в фрагменте кода 3.4, который несколько похож на цикл обучения для примера перцептрона в фрагменте кода 1.4.

Наконец, фрагмент кода 1.10 реализует цикл обучения сети. Если в примере с XOR обучение длилось до тех пор, пока мы не получали правильный вывод, теперь обучение происходит в течение фиксированного количество эпох.

Эпоха — одна итерация по всем обучающим данным.

Для каждого обучающего примера выполняется прямой проход, за которым следует обратный проход, а затем корректируются веса. Также отслеживается, сколько из обучающих примеров были спрогнозированы правильно. Затем перебираются все тестовые примеры и записывается, сколько из них было правильно спрогнозировано. Мы используем функцию `argmax` NumPy для определения индекса массива, соответствующего наибольшему значению; таким образом вектор декодируется в целое число.

Функция NumPy argmax() — это удобный способ найти элемент, который сеть прогнозирует как наиболее вероятный.

Перед тем, как передать примеры функции `forward_pass` и `adjust_weights`, каждый массив дополняется начальным элементом со значением 1.0, поскольку ожидается, что первой записью в массиве является постоянное смещение.

Мы не делаем никаких обратных проходов или корректировок значений для тестовых данных чтобы избежать утечки тестовых данных. В конце каждой эпохи происходит вывод на экран точности вывода сети как для данных обучения, так и для тестовых данных.

Фрагмент кода 1.10 — Цикл обучения для MNIST

```
# Цикл обучения сети
for i in range(EPOCHS): # Отсчёт эпох
    np.random.shuffle(index_list) # Случайный порядок
    correct_training_results = 0
    for j in index_list: # Обучение на всех примерах
          x = np.concatenate((np.array([1.0]), x_train[j]))
          forward_pass(x)
          if output_layer_y.argmax() == y_train[j].argmax():
                    correct_training_results += 1
          backward_pass(y_train[j])
          adjust_weights(x)
    correct_test_results = 0
    for j in range(len(x_test)): # Оценка сети
```

```
            x = np.concatenate((np.array([1.0]), x_test[j]))
            forward_pass(x)
            if output_layer_y.argmax() == y_test[j].argmax():
                  correct_test_results += 1
      # Вывод на экран прогресса обучения
      show_learning(i, correct_training_results/len(x_train),
                       correct_test_results/len(x_test))
plot_learning() # Вывод графика
```

Запустив программу мы должны получить вывод данных о результатах работы, с первыми строками вида:

```
номер эпохи: 0, точность обучения: 0.8563, точность тестирования: 0.9157
номер эпохи: 1, точность обучения: 0.9203, точность тестирования: 0.9240
номер эпохи: 2, точность обучения: 0.9275, точность тестирования: 0.9243
номер эпохи: 3, точность обучения: 0.9325, точность тестирования: 0.9271
номер эпохи: 4, точность обучения: 0.9342, точность тестирования: 0.9307
```

Как и прежде, ваши результаты могут немного отличаться от приводимых здесь из-за случайных отклонений. Когда программа завершится, она создаст диаграмму, показанную на рисунке 6.2.

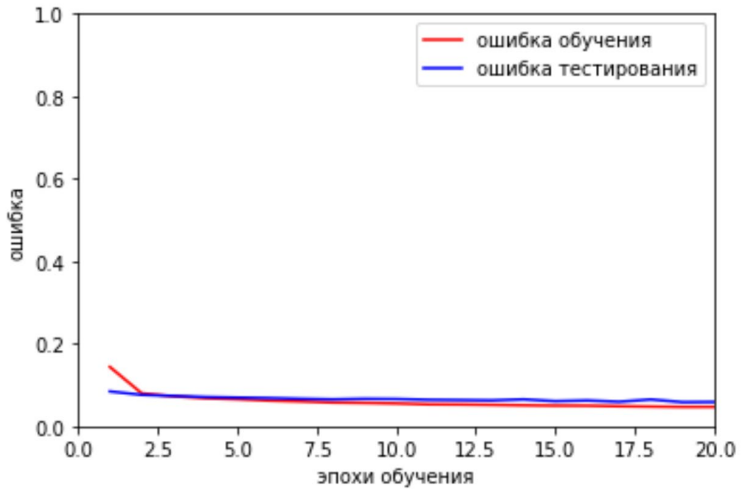

Рисунок 6.2 — Ошибка обучения и ошибка теста при обучении классификации цифр

Видно, что и количество ошибок обучения, и ошибок тестирования со временем уменьшается, при этом ошибка тестирования даже не начинает увеличиваться в правой части графика. Это является признаком того, что проблема избыточного подбора отсутствует. Видно также, что ошибок обучения меньше, чем ошибок тестирования. Это обычное явление и не является поводом для беспокойства, пока разница между двумя ошибками не существенна.

Как видно из результатов прогресса выполнения программы и диаграммы, значение ошибки тестирования быстро падает ниже 10% (точность выше 90%); то есть наша простая сеть может классифицировать больше, чем девять из десяти изображений правильно.

Задание

1. *Реализуйте полноценную программу по представленным фрагментам.*
2. *Попробуйте создать классификатор для набора данных MNIST, который достигает точности более 95,5% на тестовом наборе, подобрав значения гиперпараметров.*
3. *Напишите функцию, которая может сдвигать изображение MNIST в любом направлении (влево, вправо, вверх или вниз) на один пиксель. Затем для каждого изображения в обучающем наборе создайте четыре сдвинутые копии (по одной в каждом направлении) и добавьте их в обучающий набор. Обучите свою улучшенную модель на полученном расширенном обучающем наборе и измерьте её точность на тестовом наборе. Сравните результаты, полученные в трёх вариантах реализации модели (исходная сеть, улучшенная, улучшенная на расширенном наборе данных). Этот метод искусственного увеличения обучающей выборки называется увеличением данных или расширением обучающей выборки.*

В отчёте приведите:

- *фрагменты модифицированного и дополнительно разработанного кода с комментариями,*
- *значения гиперпараметров в улучшенной модели,*
- *результаты текстовые и графические по всем моделям.*