Учреждение образования

«БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ»

Кафедра информатики

Отчет по предмету:

«Машинное обучение»

По лабораторной работе №4

«Нейронные сети»

Выполнил: Сенькович Дмитрий Сергеевич

магистрант кафедры информатики

группы №858642

Проверил: Стержанов Максим Валерьевич

доцент, кандидат технических наук

Минск 2019

**Оглавление**

[1 Введение 2](#_Toc25708913)

[2 Распознавание цифр 4](#_Toc25708914)

[2.1 Постановка задачи 4](#_Toc25708915)

[2.2 Чтение данных 4](#_Toc25708916)

[2.3 Реализация прямого распространения 4](#_Toc25708917)

[2.4 Вычисление процента правильных классификаций. Сравнение результатов с результатами обучения логистической регрессии 5](#_Toc25708918)

[2.5 One-hot кодировка 6](#_Toc25708919)

[2.6 Реализация функции стоимости. L2-регуляризация 6](#_Toc25708920)

[2.7 Реализация функции вычисления производной функции активации 7](#_Toc25708921)

[2.8 Инициализация весов случайными числами 8](#_Toc25708922)

[2.9 Реализация обратного распространения. L2-регуляризация 8](#_Toc25708923)

[2.10 Проверка функции обратного распространения 10](#_Toc25708924)

[2.11 Обучение нейронной сети. Подсчет правильных классификаций 12](#_Toc25708925)

[2.12 Визуализация скрытого слоя 13](#_Toc25708926)

[2.13 Подбор параметра регуляризации. Визуализация скрытого слоя с различными параметрами 14](#_Toc25708927)

# 1 Введение

Нейронные сети - еще один алгоритм машинного обучения. Зачем нужны нейронные сети, если уже есть, скажем, логистическая регрессия? Для ответа можно привести следующий пример неразделимых линейно данных:

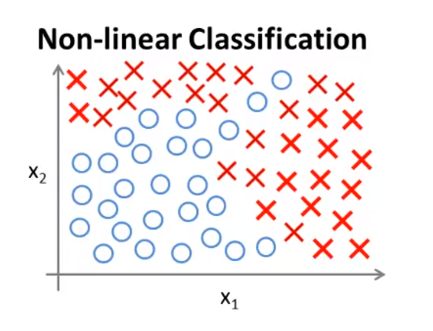


Рисунок 1.1. Пример задачи нелинейной классификации

Как мы знаем из предыдущих лабораторных работ, логистическую регрессию можно использовать и для решения такой задачи путем добавления фич различного порядка, например, так:

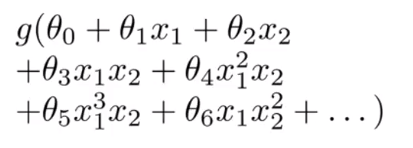


Рисунок 1.2. Модель логистической регрессии для некоторой нелинейной зависимости

В принципе, такая модель вполне неплохо может апроксимировать некоторую нелинейную зависимость. Но заметим, что для этого пришлось добавить огромное, по сравнению с исходными двумя, количество фич.

В реальности же таких фич обычно совсем не две. Порядок количества фич обычно исчисляется сотнями, тысячами и даже десятками тысяч. Понятно, что добавление такого огромного количества фич становится невозможным с точки зрения вычислительной сложности. Например, в случае всего лишь 100 фич и добавления фич, полученных умножением оригинальных фич до второй степени (x1^2, x1\*x2, x1\*x3, …, x100^2) у нас получится порядка 5000 фич. Можно включить гораздо меньше фич, но этого может не хватить.

Решением данной проблемы являются нейронные сети, построенные на вычислительных юнитах, называемых нейронами, сгруппированными в слои. Каждый нейрон принимает на вход вектор некоторой длины. С каждым нейроном ассоциирован вектор весов. Нейрон выполняет простейшую операцию: перемножает поэлементно эти вектора, суммирует получившееся значение и передает его в функцию активации, например, сигмоиду:

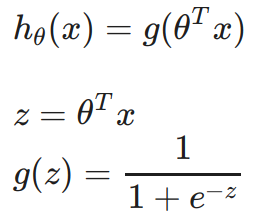


Рисунок 1.3. Функция активации - логистическая функция

Такие нейроны группируются в слои, а слои в свою очередь формируют нейронную сеть. Нейронной сети подается на вход вектор (на первый слой - входной), каждый нейрон подсчитывает свою преобразование данного вектора, к получившимся данным добавляется bias - еще один элемент, добавляемый первым в таком выходном векторе, и в таком виде данные передаются дальше, результируя в некотором конечном значении. Этот процесс называется forward propagation - прямое распространение. После этого выход сети (обработанные данные на последнем - выходном слое) сравнивается с целевыми данными и нейронная сеть подстраивает свои веса для уменьшения этой ошибки с помощью некоторого метода обучения, например, с помощью back propagation - обратное распространение. После обучения, нейронная сеть способна некоторое целевое значение. Используя определенное количество слоев и нейронов на них можно апрокисмировать сколь угодно сложную нелинейную зависимость.

# 2 Распознавание цифр

### 2.1 Постановка задачи

Даны 5000 изображений цифр 20x20 в оттенках серого. Нужно обучить нейронную сеть распознавать новые изображения.

### 2.2 Чтение данных

Считаем данные:

data = loadmat("ex4data1.mat")

x = data['X']

y = data['y']

weights\_data = loadmat("ex4weights.mat")

w0 = weights\_data['Theta1']

w1 = weights\_data['Theta2']

wt = [w0.transpose(), w1.transpose()]

Получили массив изображений 5000x400 (каждая строка - изображение), метки изображений, т.е. цифры на соответствующих изображениях и веса слоев нейронной сети. В итоге имеем следующую структуру сети: сеть имеет 3 слоя, а именно входной, скрытый и выходной. На первом слое будут 400 нейрон (размерность входного вектора), на втором 25, на третьем 10 (количество классов - цифр). При этом, для каждого нейрона второго слоя вектор его весов будет иметь размерность 400 + 1 = 401 (включая bias), а для третьего - 25 + 1 = 26 (включая bias).

### 2.3 Реализация прямого распространения

Прямое распространение представляет собой преобразование входного вектора в выходной по мере прохождения слоев нейронной сети.

Первый шаг - подсчитывание суммы поэлементного произведения векторов весов и входного вектора:



Рисунок 2.3.1. Формула первого шага преобразования входных данных a слоем j с весами Θ

Вторым шагом значением передается в функцию активации:



Рисунок 2.3.2. Формула второго шага преобразования входных данных слоем j

Данное преобразование производится каждым слоем нейронной сети, за исключением входного, используя в качестве входного вектора выходной вектор предыдущего уровня с bias элементом в качестве нулевого.

Реализуем данную логику:

def sigmoid(z):

return 1.0 / (1 + np.exp(-z))

def hypothesis(x, wt):

a1 = np.column\_stack([[1]\*(x.shape[0]), x])

z2 = a1.dot(wt[0])

a2 = sigmoid(z2)

a2 = np.column\_stack([[1]\*(a2.shape[0]), a2])

h = sigmoid(a2.dot(wt[1]))

return h

### 2.4 Вычисление процента правильных классификаций. Сравнение результатов с результатами обучения логистической регрессии

Подсчитаем процент правильных классификаций:

wt = [w0.transpose(), w1.transpose()]

output = hypothesis(x, wt)

predictions = [np.argmax(probabilities) + 1 for probabilities in output]

success\_count = 0

m = x.shape[0]

for i in range(m):

prediction\_i = predictions[i]

yi = y[i, 0]

if yi == prediction\_i:

success\_count += 1

print("success rate: %s" % ((success\_count / m)\*100))

В случае нейронной сети он составил 97.52%. Результат обучения классификатора на базе логистической регрессии давал 95.28% правильных ответов.

### 2.5 One-hot кодировка

Реализуем one-hot кодировку:

def to\_one\_hot(y, size):

converted = []

for i in range(y.shape[0]):

converted\_i = [0] \* size

yi = y[i, 0]

converted\_i[yi - 1] = 1

converted.append(converted\_i)

return np.array(converted)

Такая кодировка конвертирует конкретный класс в вектор из 0 и 1, где 1-ей является элемент вектора, индекс которого соответствует классу.

### 2.6 Реализация функции стоимости. L2-регуляризация

Функция стоимости с L2-регуляризации для нейронной сети выглядит следующим образом:

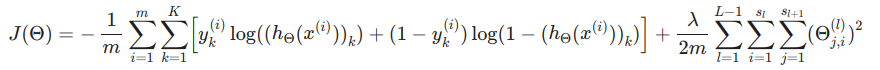


Рисунок 2.6.1. Формула стоимости с L2-регуляризацией для нейронной сети

Логика составления функции схожа с логикой составления функции для логистической регрессии. Сейчас ошибка считается по каждому вычислительному юниту на каждом слое нейронной сети, а регуляризация проводится по всем весовым параметрам.

Реализуем данную функция стоимости:

def regularization\_part(wt, m, l):

weights\_sum = regularization\_weights\_sum(wt[0]) + regularization\_weights\_sum(wt[1])

return weights\_sum \* l / (2 \* m)

def regularization\_weights\_sum(wt):

return np.sum(np.power(wt, 2))

def cost(unrolled\_weights, x, y, l):

w = from\_unrolled(unrolled\_weights, weight\_sizes)

wt = [w[0].transpose(), w[1].transpose()]

calculated\_hypothesis = hypothesis(x, wt)

m = x.shape[0]

cost\_first\_part = np.multiply(y, np.log(calculated\_hypothesis))

cost\_second\_part = np.multiply(1 - y, np.log(1-calculated\_hypothesis))

return np.sum(cost\_first\_part + cost\_second\_part) / -m + regularization\_part(wt, m, l)

def from\_unrolled(unrolled, sizes):

first\_matrix = np.reshape(unrolled[:sizes[0][0]\*sizes[0][1]], sizes[0], order = 'F')

second\_matrix = np.reshape(unrolled[sizes[0][0]\*sizes[0][1]:], sizes[1], order = 'F')

return [first\_matrix, second\_matrix]

Здесь, параметры нейронной сети преобразуются из вектора, содержащего все параметры (unrolled параметры), после чего подсчитывается функция по формуле выше.

### 2.7 Реализация функции вычисления производной функции активации

Функция производной активации получается следующим образом:

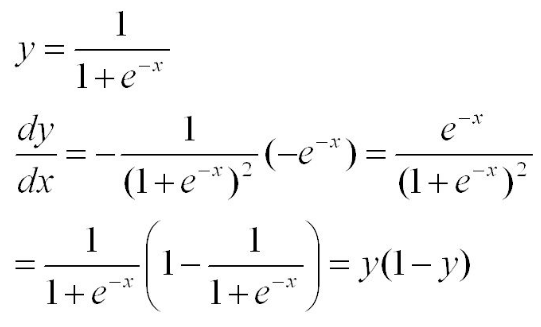


Рисунок 2.7.1. Формула производной функции активации

Реализуем производную функции активации:

def activation\_derivative(a):

return np.multiply(a, (1 - a))

Здесь a - значение функции активации.

### 2.8 Инициализация весов случайными числами

Инициализировать веса нулями, как в случае с линейной и логистической регрессиями не получится, иначе во время обратного распространения весы всех нейронов будут иметь одинаковое приращение:

def build\_randomized\_weights(size):

eps = 1

return np.random.rand(size[0], size[1]) \* (2\*eps) - eps

### 2.9 Реализация обратного распространения. L2-регуляризация

Алгоритм обратного распространения нужен для обучения сети, т.е. изменения весов слоев. Для этого подсчитываются delta матрицы изменения весов для каждого слоя. Каждый экземпляр вкладывает свою часть в это изменение. Для каждого экземпляра сначала выполняется прямое распространение для получения значений al - значения функций активаций на каждом из слоев. Затем рассчитываются δ - ошибки каждого слоя - по следующим формулам:





Рисунок 2.9.1. Формулы вычисления ошибок. Первая формула - для последнего слоя, вторая - для остальных (кроме первого)

После этого, вычисляются приращения для каждого из слоя:



Рисунок 2.9.2. Формула вычисления изменения весов слоя

Эта формула является аналогией изменения параметра линейной регрессии, т.е. правое слагаемое представляет собой частную производную функции стоимости по элементу вектора веса некоторого нейрона в некотором слое (здесь приведена формула векторной реализации, но смысл именно такой).

После вычисления таких частичных изменений весов, мы подсчитываем матрицу весов изменения всех весов в нейронной сети как взвешенную сумму этих приращений:

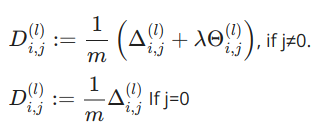


Рисунок 2.9.3. Формула приращения весов нейронной сети после обратного распространения

Это приращение является матрицей частичных производных функции стоимости по всем весам нейронной сети:



Рисунок 2.9.4. Смысл подсчитанного значения в обратном распространении

Реализуем данную логику:

def back\_propagation(unrolled\_weights, x, y, l):

w = from\_unrolled(unrolled\_weights, weight\_sizes)

delta1 = np.zeros(w[0].shape)

delta2 = np.zeros(w[1].shape)

w1 = w[0]

w2 = w[1]

wt1 = w1.transpose()

wt2 = w2.transpose()

x = np.column\_stack([[1]\*(x.shape[0]), x])

for i in range(x.shape[0]):

a1 = x[i].reshape(-1, 1).T

z2 = a1.dot(wt1)

a2 = sigmoid(z2)

a2 = np.insert(a2, 0, [1])

a3 = sigmoid(a2.dot(wt2))

yi = y[i]

d3 = a3 - yi

d2 = np.multiply(wt2.dot(d3), activation\_derivative(a2))

delta2 += d3.reshape(-1, 1).dot(a2.reshape(-1, 1).T)

delta1 += d2.reshape(-1, 1)[1:, :].dot(a1)

m = x.shape[0]

delta1 /= m

delta2 /= m

delta1[:, 1:] = delta1[:, 1:] + l\*w1[:, 1:]/m

delta2[:, 1:] = delta2[:, 1:] + l\*w2[:, 1:]/m

return unroll([delta1, delta2])

Заметим, что, как и функция стоимости, функция обратного распространения также принимает веса в unrolled виде, возвращая матрицы приращения в таком же.

### 2.10 Проверка функции обратного распространения

Заметим, что имеет место следующее равенство:

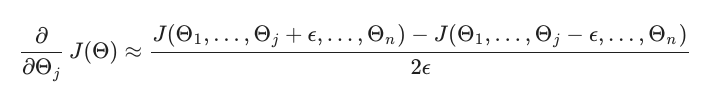


Рисунок 2.10.1. Подсчет примерного значения частной производной

Это следует из определения производной: производная - это отношение приращения функции к приращению аргумента при приращении аргумента, стремящемся к 0. Если взять *ϵ* достаточно малым, мы сможем получить примерную оценку производной, получаемой в обратном распространении:

def gradient\_checking(w, x, y, unrolled\_delta, l):

unrolled\_w = unroll(w)

n = len(unrolled\_w)

eps = 10\*\*(-4)

for \_ in range(5):

theta\_index = int(np.random.rand() \* n)

theta\_value = unrolled\_w[theta\_index]

unrolled\_w[theta\_index] = theta\_value + eps

cost\_above\_derivative = cost(unrolled\_w, x, y, l)

unrolled\_w[theta\_index] = theta\_value - eps

cost\_under\_derivative = cost(unrolled\_w, x, y, l)

approximated\_gradient = (cost\_above\_derivative - cost\_under\_derivative) / float(2 \* eps)

print('approximated gradient: %s, back propagation gradient: %s' % (approximated\_gradient, unrolled\_delta[theta\_index]))

Здесь мы подсчитываем приблизительное значение частной производной в отдельно взятом элементе вектора весов некоторого нейрона некоторого слоя и сравниваем его с соответствующим значением, полученным в обратном распространении. Если значения приблизительно равны - обратное распространение реализовано правильно.

Проверим значения подсчитанного градиента:

yoh = to\_one\_hot(y, 10)

rand\_w = [build\_randomized\_weights(w0.shape), build\_randomized\_weights(w1.shape)]

unrolled\_delta = back\_propagation(unroll(rand\_w), x, yoh, l)

gradient\_checking(rand\_w, x, yoh, unrolled\_delta, 0)

В зависимости от значений весов результат может выглядеть так:

approximated gradient: 0.002471721982644226, back propagation gradient: 0.002471721989888378

approximated gradient: -0.020684211330390667, back propagation gradient: -0.020684211517870806

approximated gradient: -0.00012591301334907712, back propagation gradient: -0.0001259129844161116

approximated gradient: 0.029689629306162146, back propagation gradient: 0.029689624169513075

approximated gradient: -0.017820162954151897, back propagation gradient: -0.0178201750458366

Видим, что обратное распространение реализовано правильно.

### 2.11 Обучение нейронной сети. Подсчет правильных классификаций

Обучим нейронную сеть с использованием библиотечного градиентного спуска и подсчитаем количество правильно классифицированных изображений:

yoh = to\_one\_hot(y, 10)

rand\_w = [build\_randomized\_weights(w0.shape), build\_randomized\_weights(w1.shape)]

unrolled\_w = unroll(rand\_w)

unrolled\_optimal\_w = optimize.fmin\_cg(maxiter=50, f=cost, x0=unrolled\_w, fprime=back\_propagation, args=(x, yoh, 1), disp=True)

optimal\_w = from\_unrolled(unrolled\_optimal\_w, weight\_sizes)

optimal\_wt = [optimal\_w[0].transpose(), optimal\_w[1].transpose()]

output = hypothesis(x, optimal\_wt)

predictions = [np.argmax(probabilities) + 1 for probabilities in output]

success\_count = 0

m = x.shape[0]

for i in range(m):

prediction\_i = predictions[i]

yi = y[i, 0]

if yi == prediction\_i:

success\_count += 1

print("success rate: %s" % ((success\_count / m)\*100))

В случае параметра регуляризации 1 удалось получить 95.28% правильно классифицированных изображений.

### 2.12 Визуализация скрытого слоя

Визуализируем веса скрытого слоя нашей нейронной сети, взяв значения весов в качестве значений яркости:

def visualize(w1):

sample\_width = 20

plot\_dim = 5

fig, axises = plt.subplots(plot\_dim, plot\_dim, figsize=(10, 10))

axises = axises.ravel()

for i in range(axises.shape[0]):

axis = axises[i]

axis.imshow(w1[i].reshape(sample\_width, sample\_width, order='F'), cmap='gray')

axis.axis('off')

plt.show()

visualize(optimal\_w[0][:, 1:])

В случае параметра регуляризации 1 получим следующее изображение:

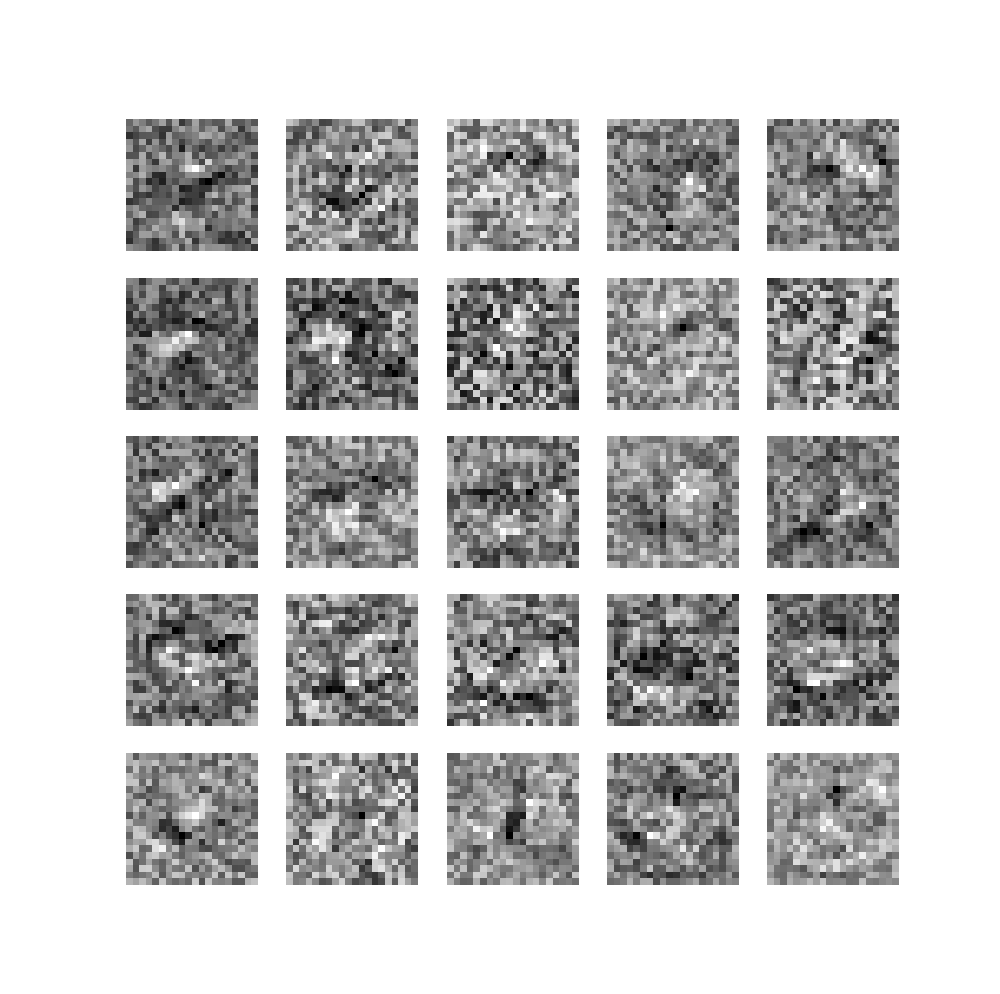


Рисунок 2.12.1. Визуализация скрытого слоя. Параметр регуляризации 1

### 2.13 Подбор параметра регуляризации. Визуализация скрытого слоя с различными параметрами

Подбор параметра регуляризации будет строится следующим образом: мы будет исследовать некоторый интервал значения параметра в поисках наилучшего, в нашем случае такого, который дает наибольший процент правильных классификаций:

m = x.shape[0]

lambdas = np.linspace(0.01, 0.5, 50)

errors = []

success\_rates = []

yoh = to\_one\_hot(y, 10)

for l in lambdas:

rand\_w = [build\_randomized\_weights(w0.shape), build\_randomized\_weights(w1.shape)]

unrolled\_w = unroll(rand\_w)

unrolled\_optimal\_w, error, func\_calls, grad\_calls, warnflag = optimize.fmin\_cg(maxiter=50, f=cost, x0=unrolled\_w, fprime=back\_propagation, args=(x, yoh, l), disp=True, full\_output=true)

optimal\_w = from\_unrolled(unrolled\_optimal\_w, weight\_sizes)

errors.append(error)

optimal\_wt = [optimal\_w[0].transpose(), optimal\_w[1].transpose()]

output = hypothesis(x, optimal\_wt)

predictions = [np.argmax(probabilities) + 1 for probabilities in output]

success\_count = 0

m = x.shape[0]

for i in range(m):

prediction\_i = predictions[i]

yi = y[i, 0]

if yi == prediction\_i:

success\_count += 1

success\_rates.append(((success\_count / m)\*100))

best\_lambda\_index = np.argmin(errors)

best\_lambda = lambdas[best\_lambda\_index]

min\_error = errors[best\_lambda\_index]

print("best\_lambda %s, min\_error %s" % (best\_lambda, min\_error))

plt.plot(lambdas, errors, c="r")

plt.show()

best\_lambda\_index\_by\_success\_rate = np.argmax(success\_rates)

best\_lambda = lambdas[best\_lambda\_index\_by\_success\_rate]

max\_success\_rate = success\_rates[best\_lambda\_index\_by\_success\_rate]

print("best\_lambda %s, max\_success\_rate %s" % (best\_lambda, max\_success\_rate))

plt.plot(lambdas, success\_rates, c="r")

plt.show()

Поиск будет вестись в двух интервалах: от 0.1 до 5 (50 значений) и от 0.01 до 0.5 (также 50 значений). Код выше строит два графика-показателя перформанса нашей нейронной сети: зависимость ошибки и количества правильных классификаций от параметра регуляризации.

Посмотрим первые графики:

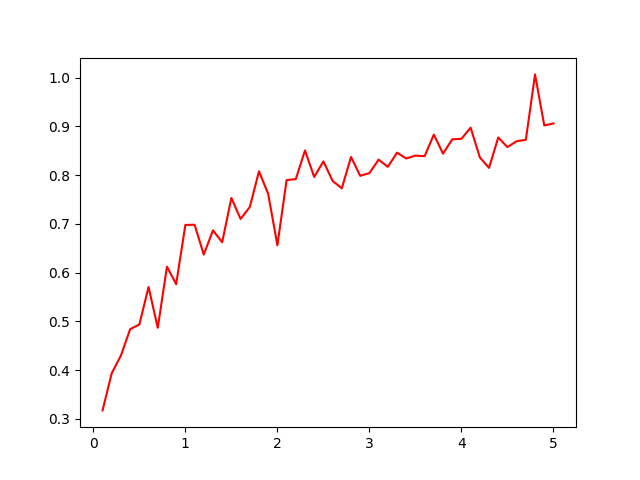


Рисунок 2.13.1. График ошибки от параметра регуляризации. Интервал [0.1; 5]

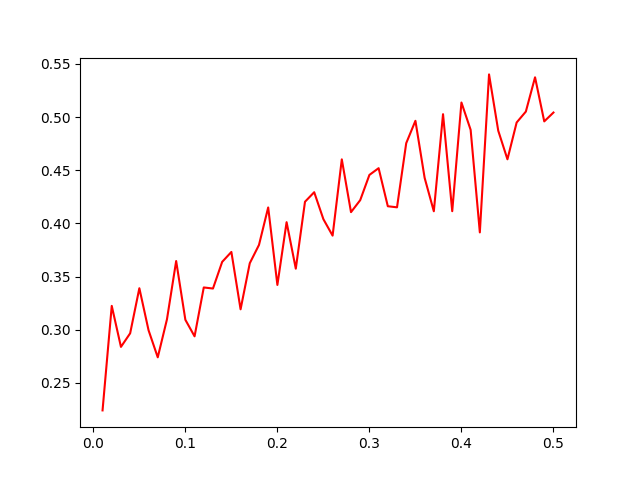


Рисунок 2.13.2. График ошибки от параметра регуляризации. Интервал [0.01; 0.5]

Теперь посмотрим на графики зависимости количества правильных классификаций:

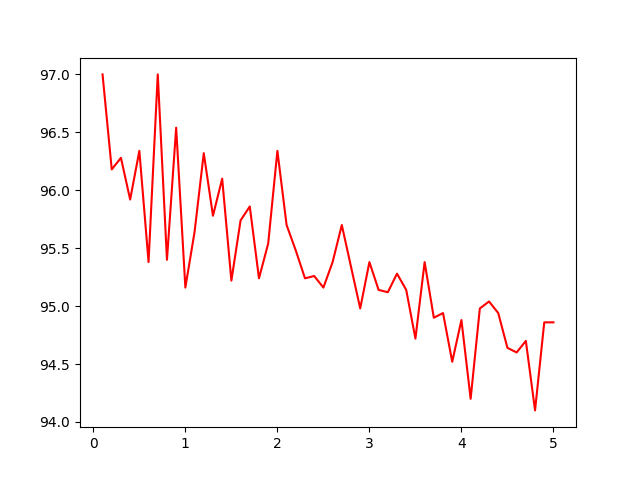


Рисунок 2.13.3. График процента правильных классификаций в зависимости от параметра регуляризации. Интервал [0.1; 5]

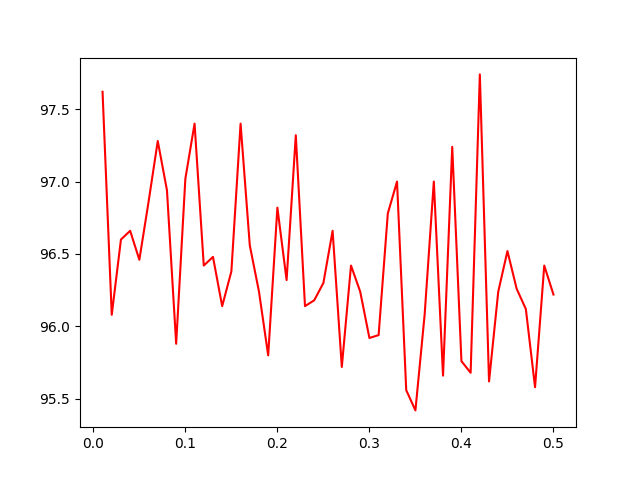


Рисунок 2.13.4. График процента правильных классификаций в зависимости от параметра регуляризации. Интервал [0.01; 0.5]

В результате поиска остановимся на значении параметра регуляризации 0.42, при котором достигается максимальная точность классификации в 97.74%.

Визуализируем скрытый слой в зависимости от нескольких значений параметра регуляризации: 0, 0.42, 1, 10, 25, 50, 70 и 100:

|  |  |
| --- | --- |
|  |  |
|  |  |
|  |  |
|  |  |

Рисунок 2.13.5. Изображение скрытого слоя при различных параметрах регуляризации по возрастанию параметра

Заметим, что с возрастанием параметра регуляризации шума на изображениях становится меньше и изображения весов отдельных нейронов даже напоминают цифры.