Полезные ссылки:

- https://habr.com/ru/post/312450/
- https://habr.com/ru/post/313216/
- https://habr.com/ru/company/otus/blog/483466/

Прямое распространение (Feedforward)

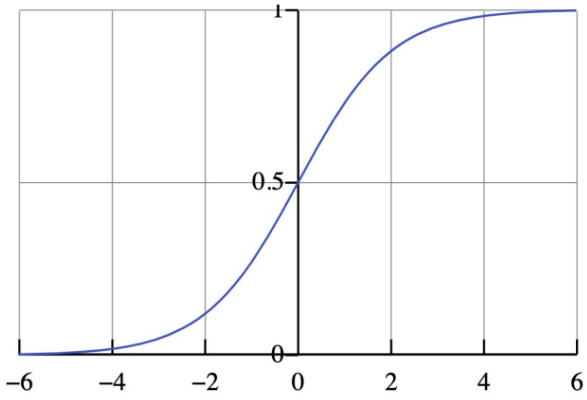
Прямое распространение - это интерактивный процесс расчета активаций для каждого слоя, начиная с входного и заканчивая выходным слоем.

Для простой сети, упомянутой в предыдущем разделе выше, мы можем рассчитать активации для второго уровня на основе входного уровня и наших сетевых параметров:

$$a_1^{(2)} = g\left(\Theta_{10}^{(1)}x_0 + \Theta_{11}^{(1)}x_1 + \Theta_{12}^{(1)}x_2 + \Theta_{13}^{(1)}x_3\right)$$

$$a_2^{(2)} = g\left(\Theta_{20}^{(1)}x_0 + \Theta_{21}^{(1)}x_1 + \Theta_{22}^{(1)}x_2 + \Theta_{23}^{(1)}x_3\right)$$

$$a_3^{(2)} = g\left(\Theta_{30}^{(1)}x_0 + \Theta_{31}^{(1)}x_1 + \Theta_{32}^{(1)}x_2 + \Theta_{33}^{(1)}x_3\right)$$



Активация выходного слоя будет рассчитана на основе активаций скрытого слоя:

$$h_\Theta(x) = a_1^{(3)} = g\left(\Theta_{10}^{(2)}a_0^{(2)} + \Theta_{11}^{(2)}a_1^{(2)} + \Theta_{12}^{(2)}a_2^{(2)} + \Theta_{13}^{(2)}a_3^{(2)}\right)$$

Где функция *g ()* может быть сигмовидной:

$$g(z) = \frac{1}{1+e^{-z}}$$

Векторизованная реализация прямого распространения

Теперь давайте преобразуем предыдущие вычисления в более сжатую векторизованную форму.

$$x = \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix}$$

Чтобы упростить предыдущие уравнения активации, давайте введем переменную z :

$$z_1^{(2)} = \Theta_{10}^{(1)} x_0 + \Theta_{11}^{(1)} x_1 + \Theta_{12}^{(1)} x_2 + \Theta_{13}^{(1)} x_3$$

$$z_2^{(2)} = \Theta_{20}^{(1)} x_0 + \Theta_{21}^{(1)} x_1 + \Theta_{22}^{(1)} x_2 + \Theta_{23}^{(1)} x_3$$

$$z_3^{(2)} = \Theta_{30}^{(1)} x_0 + \Theta_{31}^{(1)} x_1 + \Theta_{32}^{(1)} x_2 + \Theta_{33}^{(1)} x_3$$

$$z^{(2)} = \begin{bmatrix} z_1^{(2)} \\ z_2^{(2)} \\ z_3^{(2)} \end{bmatrix} = \Theta^{(1)} x = \Theta^{(1)} a^{(1)}$$

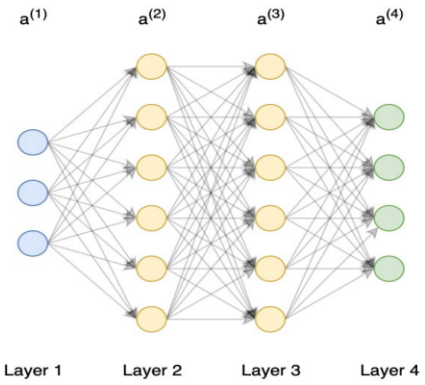
> Не забудьте добавить единицы смещения (активации) перед переходом к следующему слою. $a_0^{(2)} = 1,\ so\ that\ a^{(2)} \in R^4$

$$z^{(3)} = \Theta^{(2)} a^{(2)}$$

$$h_\Theta(x) = a^{(3)} = g\left(z^{(3)}\right)$$

Пример прямого распространения

В качестве примера возьмем следующую сетевую архитектуру с 4 слоями (входной слой, 2 скрытых слоя и выходной слой):



В этом случае шаги прямого распространения будут выглядеть следующим образом:

$$a^{(1)} = x$$
$$z^{(2)} = \Theta^{(1)} a^{(1)}$$
$$a^{(2)} = g\left(z^{(2)}\right),\ \left(add\ a_0^{(2)} = 1\right)$$
$$z^{(3)} = \Theta^{(2)} a^{(2)}$$
$$a^{(3)} = g\left(z^{(3)}\right),\ \left(add\ a_0^{(3)} = 1\right)$$
$$z^{(4)} = \Theta^{(3)} a^{(3)}$$
$$a^{(4)} = h_\Theta(x) = g\left(z^{(4)}\right)$$

Функция потерь (Loss function or cost function)

Функция стоимости нейронной сети очень похожа на функцию стоимости логистической регрессии.

$$J(\Theta) = -\frac{1}{m}\left[\sum_{i=1}^{m}\sum_{k=1}^{K} y_k^{(i)} log\left(h_\Theta(x^{(i)})\right)_k + \left(1 - y_k^{(i)}\right) log\left(1 - \left(h_\Theta(x^{(i)})\right)_k\right)\right] + \frac{\lambda}{2m}\sum_{l=1}^{L-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_{l+1}}\left(\Theta_{ji}^{(l)}\right)^2$$

$$h_\Theta(x) \in R^K$$

$$(h_\Theta(x))_i = i^{th}\ output$$

MSE (Mean Squared Error) – среднеквадратичная ошибка

$$\frac{(i_1-a_1)^2+(i_2-a_2)^2+\ldots+(i_n-a_n)^2}{n}$$

Root MSE – среднеквадратическое отклонение.

$$\sqrt{\frac{(i_1-a_1)^2+(i_2-a_2)^2+\ldots+(i_n-a_n)^2}{n}}$$

Кросс-энтропия (или логарифмическая функция потерь – log loss)

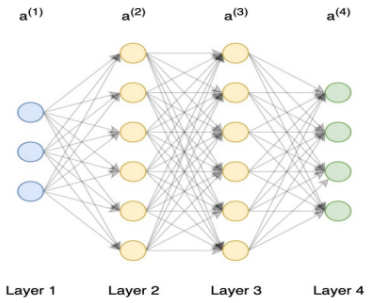
$$H(P,Q) = -\sum_x P(x)\log Q(x)$$

Обратное распространение

Вычисление градиента

Алгоритм обратного распространения ошибки имеет ту же цель, что и градиентный спуск для линейной или логистической регрессии - он корректирует значения тета, чтобы минимизировать функцию стоимости.
Другими словами, нам нужно иметь возможность вычислять частную производную функции стоимости для каждого тета.

$\frac{\partial}{\partial\Theta_{ij}^{(l)}}J(\Theta)$



Предположим, что:

$\delta_j^{(l)}$ – «ошибка» узла j в слое l.

Для каждого блока вывода (слой $L = 4$):

$$\delta_j^{(4)} = a_j^{(4)} - y_j$$

Или в векторизованном виде:

$$\delta^{(4)} = a^{(4)} - y$$

$$\delta^{(3)} = \left(\Theta^{(3)}\right)^T \delta^{(4)} .* g'\left(z^{(3)}\right)$$

$$\delta^{(2)} = \left(\Theta^{(2)}\right)^T \delta^{(3)} .* g'\left(z^{(2)}\right)$$

$\delta^{(1)}$ – *is for input layer, we can't change it*

g' - сигмовидный градиент.

$$g'(z) = \frac{\partial}{\partial z}g(z) = g(z)(1-g(z)),\ where\ g(z) = \frac{1}{1+e^{-z}}$$

Теперь мы можем рассчитать шаг градиента:

$$\frac{\partial}{\partial\Theta_{ij}^{(l)}}J(\Theta) = a_j^{(l)}\delta_i^{(l+1)}$$

Алгоритм обратного распространения ошибки — популярный алгоритм обучения плоскослоистых нейронных сетей прямого распространения (многослойных персептронов). Относится к методам обучения с учителем, поэтому требует, чтобы в обучающих примерах были заданы целевые значения. Также является одним из наиболее известных алгоритмов машинного обучения.

В основе идеи алгоритма лежит использование выходной ошибки нейронной сети

$$E = \frac{1}{2}\sum_{i=1}^{k}(y - y')^2$$

для вычисления величин коррекции весов нейронов в ее скрытых слоях, где k — число выходных нейронов сети, y — целевое значение, y' — фактическое выходное значение. Алгоритм является итеративным и использует принцип обучения «по шагам» (обучение в режиме on-line), когда веса нейронов сети корректируются после подачи на ее вход одного обучающего примера. На каждой итерации происходит два прохода сети — прямой и обратный. На прямом входной вектор распространяется от входов сети к ее выходам и формирует некоторый выходной вектор, соответствующий текущему (фактическому) состоянию весов. Затем вычисляется ошибка нейронной сети как разность между фактическим и целевым значениями. На обратном проходе эта ошибка распространяется от выхода сети к ее входам, и производится коррекция весов нейронов в соответствии с

правилом:

$$\Delta w_{j,i}(n) = -\eta\frac{\partial E_{av}}{\partial w_{ij}},$$

где $w_{j,i}$ — вес i-й связи j-го нейрона, η — параметр скорости обучения, который позволяет дополнительно управлять величиной шага коррекции $\Delta w_{j,i}$ с целью более точной настройки на минимум ошибки и подбирается экспериментально в процессе обучения (изменяется в интервале от 0 до 1)

Учитывая, что выходная сумма j-го нейрона равна

$$S_j = \sum_{i=1}^{n} w_{ij}x_i,$$

можно показать, что

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial S_j}\frac{\partial S_j}{\partial w_{ij}} = x_i\frac{\partial E}{\partial S_j}$$

Из последнего выражения следует, что дифференциал ∂S_j активационной функции нейронов сети f(s) должен существовать и не быть равным нулю в любой точке, т.е. активационная функция должна быть дифференцируема на всей числовой оси. Поэтому для применения метода обратного распространения используют сигмоидальные активационные функции, например, логистическую или гиперболический тангенс.

Для тренировочного набора

$$\left\{\left(x^{(1)}, y^{(1)}\right), \left(x^{(2)}, y^{(2)}\right), \ldots, \left(x^{(m)}, y^{(m)}\right)\right\}$$

Нам нужно установить:

$$\Delta_{ij}^{(l)} = 0 \ (for\ all\ l,\ i,\ j)$$

$$\begin{aligned}
&for\ i = 1\ to\ m \\
&\quad Set\ a^{(1)} = x^{(i)} \\
&\quad Perform\ forward\ propagation\ to\ compute\ a^{(l)}\ for\ l = 2, 3, \ldots, L \\
&\quad Using\ y^{(i)},\ compute\ \delta^{(L)} = a^{(L)} - y^{(i)} \\
&\quad Compute\ \delta^{(L-1)},\ \delta^{(L-2)}, \ldots, \delta^{2} \\
&\quad \Delta_{ij}^{(l)} := \Delta_{ij}^{(l)} + a_j^{(l)} \delta_i^{(l+1)} \left(or\ in\ vectorized\ form\ \Delta^{(l)} := \Delta^{(l)} + \delta^{(l+1)} \left(a^{(l)}\right)^T\right)
\end{aligned}$$

$$D_{ij}^{(l)} := \begin{cases} \frac{1}{m}\Delta_{ij}^{(l)} + \frac{\lambda}{m}\Theta_{ij}^{(l)} & if\ j \geq 1 \\ \frac{1}{m}\Delta_{ij}^{(l)} & if\ j = 0 \end{cases}$$

$$\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta) = D_{ij}^{(l)}$$

Оптимизаторы

- Пакетный градиентный спуск
- Мини-пакетный градиентный спуск
- Стохастический градиентный спуск
- Адаград
- RMSprop
- Адам

```
# loss function: Binary Cross-entropy and optimizer: Adam
model.compile(loss='binary_crossentropy', optimizer='adam')
```

или

```
# loss function: MSE and optimizer: stochastic gradient descent
model.compile(loss='mean_squared_error', optimizer='sgd')
```

Нейронная сеть на Python

```
class NeuralNetwork:
    def __init__(self, x, y):
        self.input      = x
        self.weights1   = np.random.rand(self.input.shape[1],4)
        self.weights2   = np.random.rand(4,1)
        self.y          = y
        self.output     = np.zeros(y.shape)
```

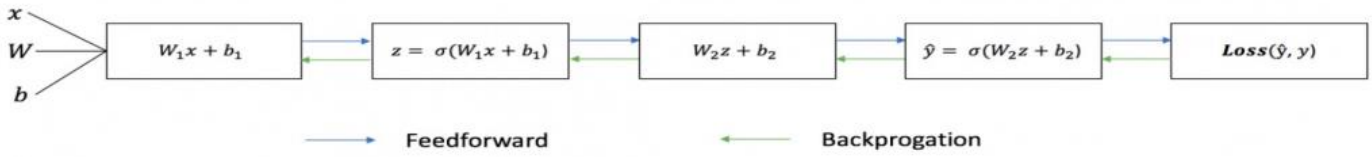
Вот её вывод в виде формулы:

$$\hat{y} = \sigma(W_2\sigma(W_1x + b_1) + b_2)$$

Веса W и смещения b — это только переменные, которые влияют на вывод — ŷ. Естественно, правильность значений и весов предопределяется предсказательной силой. Процесс тонкой настройки весов и смещений на основании входных данных известен как обучение нейронной сети. Каждая итерация обучения состоит из:

- Расчёта спрогнозированного вывода ŷ, известного как прямое распространение.
- Обновления весов и смещений, известного как обратное распространение.

Диаграмма ниже иллюстрирует обучение:



Прямое распространение

```
class NeuralNetwork:
    def __init__(self, x, y):
        self.input      = x
        self.weights1   = np.random.rand(self.input.shape[1],4)
        self.weights2   = np.random.rand(4,1)
        self.y          = y
        self.output     = np.zeros(self.y.shape)

    def feedforward(self):
        self.layer1 = sigmoid(np.dot(self.input, self.weights1))
        self.output = sigmoid(np.dot(self.layer1, self.weights2))
```

Обратное распространение

$$Loss(y,\hat{y}) = \sum_{i=1}^{n}(y - \hat{y})^2$$

$$\frac{\partial\, Loss(y,\hat{y})}{\partial W} = \frac{\partial Loss(y,\hat{y})}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial z} * \frac{\partial z}{\partial W} \quad \text{where } z = Wx + b$$

$$= 2(y - \hat{y}) * \text{derivative of sigmoid function} * x$$

$$= 2(y - \hat{y}) * z(1\text{-}z) * x$$

```
class NeuralNetwork:
    def __init__(self, x, y):
        self.input      = x
        self.weights1   = np.random.rand(self.input.shape[1],4)
        self.weights2   = np.random.rand(4,1)
        self.y          = y
        self.output     = np.zeros(self.y.shape)

    def feedforward(self):
        self.layer1 = sigmoid(np.dot(self.input, self.weights1))
        self.output = sigmoid(np.dot(self.layer1, self.weights2))

    def backprop(self):
        # application of the chain rule to find derivative of the loss function wit
        d_weights2 = np.dot(self.layer1.T, (2*(self.y - self.output) * sigmoid_deri
        d_weights1 = np.dot(self.input.T,  (np.dot(2*(self.y - self.output) * sigmo

        # update the weights with the derivative (slope) of the loss function
        self.weights1 += d_weights1
        self.weights2 += d_weights2
```

Результат

Набор данных:

X1	X2	X3	Y
0	0	1	0
0	1	1	1
1	0	1	1
1	1	1	0

Получим:

Prediction	Y (Actual)
0.023	0
0.979	1
0.975	1
0.025	0

Стохастический градиентный спуск (SGD) для логарифмической функции потерь (LogLoss) в задаче бинарной классификации

```
import pandas as pd
data = pd.read_csv("https://raw.githubusercontent.com/DLSchool/dlschool_old/master/
data.head(10)
```

```
import pandas as pd
data = pd.read_csv("https://raw.githubusercontent.com/DLSchool/dlschool_old/master/materials/homeworks/hw04/data/apples_pears.csv")
data.head(10)
```

	yellowness	symmetry	target
0	0.779427	0.257305	1.0
1	0.777005	0.015915	1.0
2	0.977092	0.304210	1.0
3	0.043032	0.140899	0.0
4	0.760433	0.193123	1.0
5	0.796064	0.040560	1.0
6	0.903320	0.170235	1.0
7	0.981188	0.091016	1.0
8	0.407194	0.465379	0.0
9	0.790900	0.291975	0.0

Пусть: x1 — yellowness, x2 — symmetry, y = targer Составим функцию y = w1 * x1 + w2 * x2 + w0 (w0 будем считать смещением(англ. — bias)) Теперь наша задача сводится к поиску весов w1, w2 и w0, которые наиболее точным образом описывают зависимость y от x1 и x2.

Используем логарифмическую функцию потерь:

$$LogLoss(\hat{y}, y) = -\frac{1}{n}\sum_{i=1}^{n} y_i \log(\hat{y_i}) + (1 - y_i)\log(1 - \hat{y_i}) = -\frac{1}{n}\sum_{i=1}^{n} y_i \log(\sigma(w \cdot x_i)) + (1 - y_i)\log(1 - \sigma(w \cdot x_i))$$

Выберем SGD

$$LogLoss(\hat{y}, y) = -y_i \log(\hat{y_i}) + (1 - y_i)\log(1 - \hat{y_i}) = -y_i \log(\sigma(w \cdot x_i)) + (1 - y_i)\log(1 - \sigma(w \cdot x_i))$$

Подготовим данные:

```
import pandas as pd
import numpy as np

X = data.iloc[:,:2].values  # матрица объекты-признаки
y = data['target'].values.reshape((-1, 1))  # классы (столбец из нулей и единиц)

x1 = X[:, 0]
x2 = X[:, 1]

def sigmoid(x):
    return 1 / (1 + np.exp(-x))
```

```
import random
np.random.seed(62)
w1 = np.random.randn(1)
w2 = np.random.randn(1)
w0 = np.random.randn(1)

print(w1, w2, w0)

# form range 0..999
idx = np.arange(1000)

# random shuffling
np.random.shuffle(idx)
x1, x2, y = x1[idx], x2[idx], y[idx]
# learning rate
lr = 0.001

# number of epochs
n_epochs = 10000

for epoch in range(n_epochs):

  i = random.randint(0, 999)

  yhat = w1 * x1[i] + w2 * x2[i] + w0

  w1_grad = -((y[i] - sigmoid(yhat)) * x1[i])

  w2_grad = -((y[i] - sigmoid(yhat)) * x2[i])

  w0_grad = -(y[i] - sigmoid(yhat))

  w1 -= lr * w1_grad

  w2 -= lr * w2_grad

  w0 -= lr * w0_grad

print(w1, w2, w0)
```

[0.49671415] [-0.1382643] [0.64768854]
[0.87991625] [-1.14098372] [0.22355905]

$$\frac{\partial Loss}{\partial w_j} = -\left(\frac{y_i}{\sigma(w \cdot x_i)} - \frac{1-y_i}{1-\sigma(w \cdot x_i)}\right)(\sigma(w \cdot x_i))'_{w_j} = -\left(\frac{y_i}{\sigma(w \cdot x_i)} - \frac{1-y_i}{1-\sigma(w \cdot x_i)}\right)\sigma(w \cdot x_i)(1-\sigma(w \cdot x_i))x_{ij} = -(y_i - \sigma(w \cdot x_i))\,x_{ij}$$

Для свободного члена w0 — опускается множитель x(принимается равным единице).

Проверка верных ответов:

```
i = 0
correct = 0
incorrect = 0
for item in y:
  if(np.around(x1[i] * w1 + x2[i] * w2 + w0) == item):
    correct += 1
  else:
    incorrect += 1
  i = i + 1

print(correct, incorrect)
```

925 75

np.around(x) — округляет значение x. Для нас: если x > 0.5, то значение равно 1. Если x ≤ 0.5, то значение равно 0.

```
import torch
import numpy as np
from torch.nn import Linear, Sigmoid

def make_train_step(model, loss_fn, optimizer):
    def train_step(x, y):
        model.train()
        yhat = model(x)
        loss = loss_fn(yhat, y)
        loss.backward()
        optimizer.step()
        optimizer.zero_grad()
        return loss.item()
    return train_step

X = torch.FloatTensor(data.iloc[:,:2].values)

y = torch.FloatTensor(data['target'].values.reshape((-1, 1)))

from torch import optim, nn

neuron = torch.nn.Sequential(
    Linear(2, out_features=1),
    Sigmoid()
)
print(neuron.state_dict())

lr = 0.1
n_epochs = 10000
loss_fn = nn.MSELoss(reduction="mean")
optimizer = optim.SGD(neuron.parameters(), lr=lr)
train_step = make_train_step(neuron, loss_fn, optimizer)

for epoch in range(n_epochs):
    loss = train_step(X, y)

print(neuron.state_dict())
print(loss)
```

OrderedDict([('0.weight', tensor([[-0.4148, -0.5838]])), ('0.bias', tensor([0.5448]))])
OrderedDict([('0.weight', tensor([[5.4915, -8.2156]])), ('0.bias', tensor([-1.1130]))])
0.03930133953690529

Демонстрация нейронной сети для классификации

```
# To make debugging of multilayer_perceptron module easier we enable imported modules autoreloading feature.
# By doing this you may change the code of multilayer_perceptron library and all these changes will be available here.
%load_ext autoreload
%autoreload 2

# Add project root folder to module loading paths.
import sys
sys.path.append('../..')
```

```
# Import 3rd party dependencies.
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
import math

# Import custom multilayer perceptron implementation.
from homemade.neural_network import MultilayerPerceptron
```

```
# Load the data.
data = pd.read_csv('../../data/fashion-mnist-demo.csv')

# Laets create the mapping between numeric category and category name.
label_map = {
    0: 'T-shirt/top',
    1: 'Trouser',
    2: 'Pullover',
    3: 'Dress',
    4: 'Coat',
    5: 'Sandal',
    6: 'Shirt',
    7: 'Sneaker',
    8: 'Bag',
    9: 'Ankle boot',
}

# Print the data table.
data.head(10)
```

	label	pixel1	pixel2	pixel3	pixel4	pixel5	pixel6	pixel7	pixel8	pixel9	...	pixel775	pixel776	pixel777	pixel778	pixel779	pixel780	pixel781	pixel782
0	2	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0
1	9	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0
2	6	0	0	0	0	0	0	0	5	0	...	0	0	0	30	43	0	0	0
3	0	0	0	0	1	2	0	0	0	0	...	3	0	0	0	0	1	0	0
4	3	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0
5	4	0	0	0	5	4	5	5	3	5	...	7	8	7	4	3	7	5	0
6	4	0	0	0	0	0	0	0	0	0	...	14	0	0	0	0	0	0	0
7	5	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0
8	4	0	0	0	0	0	0	3	2	0	...	1	0	0	0	0	0	0	0
9	8	0	0	0	0	0	0	0	0	0	...	203	214	166	0	0	0	0	0

10 rows × 785 columns

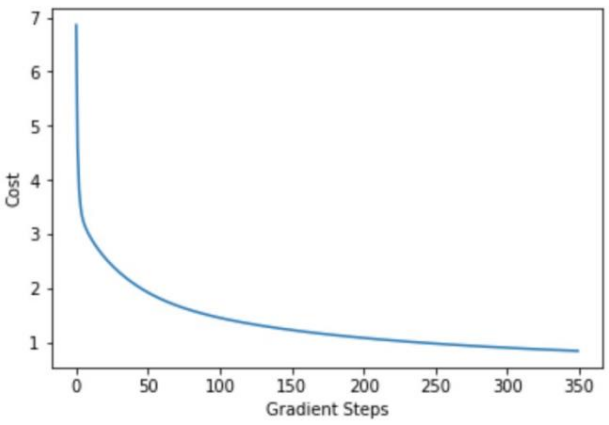
```
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 5000 entries, 0 to 4999
Columns: 785 entries, label to pixel784
dtypes: int64(785)
memory usage: 29.9 MB
```

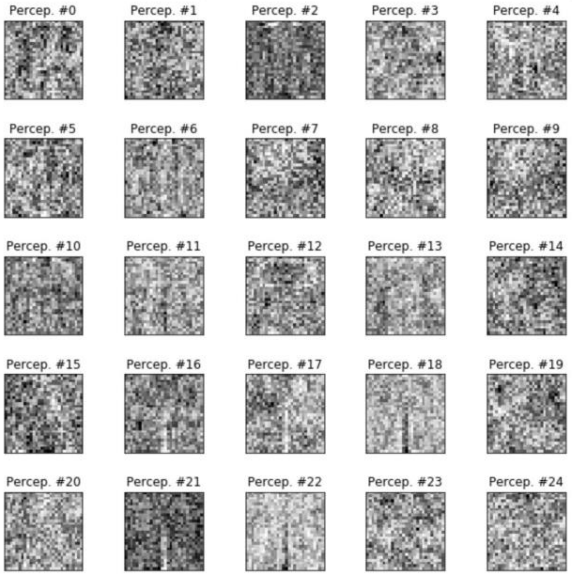
```
# How many images to display.
numbers_to_display = 25

# Calculate the number of cells that will hold all the images.
num_cells = math.ceil(math.sqrt(numbers_to_display))

# Make the plot a little bit bigger than default one.
plt.figure(figsize=(10, 10))

# Go through the first images in a training set and plot them.
for plot_index in range(numbers_to_display):
    # Extract image data.
    digit = data[plot_index:plot_index + 1].values
    digit_label = digit[0][0]
    digit_pixels = digit[0][1:]

    # Calculate image size (remember that each picture has square proportions).
    image_size = int(math.sqrt(digit_pixels.shape[0]))

    # Convert image vector into the matrix of pixels.
    frame = digit_pixels.reshape((image_size, image_size))

    # Plot the image matrix.
    plt.subplot(num_cells, num_cells, plot_index + 1)
    plt.imshow(frame, cmap='Greys')
    plt.title(label_map[digit_label])
    plt.tick_params(axis='both', which='both', bottom=False, left=False, labelbottom=False, labelleft=False)

# Plot all subplots.
plt.subplots_adjust(hspace=0.5, wspace=0.5)
plt.show()
```

```
# Split data set on training and test sets with proportions 80/20.
# Function sample() returns a random sample of items.
pd_train_data = data.sample(frac=0.8)
pd_test_data = data.drop(pd_train_data.index)

# Convert training and testing data from Pandas to NumPy format.
train_data = pd_train_data.values
test_data = pd_test_data.values

# Extract training/test labels and features.
num_training_examples = 1000

x_train = train_data[:num_training_examples, 1:]
y_train = train_data[:num_training_examples, [0]]

x_test = test_data[:, 1:]
y_test = test_data[:, [0]]
```

```
# Configure neural network.
layers = [
    784, # Input layer - 28x28 input pixels.
    25,  # First hidden layer - 25 hidden units.
    10,  # Output layer - 10 labels, from 0 to 9.
];
normalize_data = True  # Flag that detects whether we want to do features normalization or not.
epsilon = 0.12  # Defines the range for initial theta values.
max_iterations = 350  # Max number of gradient descent iterations.
regularization_param = 2  # Helps to fight model overfitting.
alpha = 0.1  # Gradient descent step size.

# Init neural network.
multilayer_perceptron = MultilayerPerceptron(x_train, y_train, layers, epsilon, normalize_data)

# Train neural network.
(thetas, costs) = multilayer_perceptron.train(regularization_param, max_iterations, alpha)

plt.plot(range(len(costs)), costs)
plt.xlabel('Gradient Steps')
plt.ylabel('Cost')
plt.show()
```




```
# Setup the number of layer we want to display.
# We want to display the first hidden layer.
layer_number = 1

# How many perceptrons to display.
num_perceptrons = len(thetas[layer_number - 1])

# Calculate the number of cells that will hold all the images.
num_cells = math.ceil(math.sqrt(num_perceptrons))

# Make the plot a little bit bigger than default one.
plt.figure(figsize=(10, 10))

# Go through the perceptrons plot what they've learnt.
for perceptron_index in range(num_perceptrons):
    # Extract perceptron data.
    perceptron = thetas[layer_number - 1][perceptron_index][1:]

    # Calculate image size (remember that each picture has square proportions).
    image_size = int(math.sqrt(perceptron.shape[0]))

    # Convert image vector into the matrix of pixels.
    frame = perceptron.reshape((image_size, image_size))

    # Plot the image matrix.
    plt.subplot(num_cells, num_cells, perceptron_index + 1)
    plt.imshow(frame, cmap='Greys', vmin=np.amin(frame), vmax=np.amax(frame))
    plt.title('Percep. #%s' % perceptron_index)
    plt.tick_params(axis='both', which='both', bottom=False, left=False, labelbottom=False, labelleft=False)

# Plot all subplots.
plt.subplots_adjust(hspace=0.5, wspace=0.5)
plt.show()
```




```
# Make training set predictions.
y_train_predictions = multilayer_perceptron.predict(x_train)
y_test_predictions = multilayer_perceptron.predict(x_test)

# Check what percentage of them are actually correct.
train_precision = np.sum(y_train_predictions == y_train) / y_train.shape[0] * 100
test_precision = np.sum(y_test_predictions == y_test) / y_test.shape[0] * 100

print('Training Precision: {:5.4f}%'.format(train_precision))
print('Test Precision: {:5.4f}%'.format(test_precision))
```

```
Training Precision: 93.8000%
Test Precision: 80.6000%
```

```
# How many images to display.
numbers_to_display = 64

# Calculate the number of cells that will hold all the images.
num_cells = math.ceil(math.sqrt(numbers_to_display))

# Make the plot a little bit bigger than default one.
plt.figure(figsize=(15, 15))

# Go through the first images in a test set and plot them.
for plot_index in range(numbers_to_display):
    # Extract digit data.
    digit_label = y_test[plot_index, 0]
    digit_pixels = x_test[plot_index, :]

    # Predicted label.
    predicted_label = y_test_predictions[plot_index][0]

    # Calculate image size (remember that each picture has square proportions).
    image_size = int(math.sqrt(digit_pixels.shape[0]))

    # Convert image vector into the matrix of pixels.
    frame = digit_pixels.reshape((image_size, image_size))

    # Plot the image matrix.
    color_map = 'Greens' if predicted_label == digit_label else 'Reds'
    plt.subplot(num_cells, num_cells, plot_index + 1)
    plt.imshow(frame, cmap=color_map)
    plt.title(label_map[predicted_label])
    plt.tick_params(axis='both', which='both', bottom=False, left=False, labelbottom=False, labelleft=False)

# Plot all subplots.
plt.subplots_adjust(hspace=0.5, wspace=0.5)
plt.show()
```

Радиальная базисная функция

Используется для разбиения пространства входных данных окружностями или (в общем случае) гиперсферами.

Отличия сетей RBF от сетей MLP:

- моделируют произвольную нелинейную функцию с помощью всего одного промежуточного слоя
- сети с RBF обучается на порядок быстрее MLP
- «групповое» представление пространства модели (в отличие от «плоскостного»)
- при удалении от обучающего множества значение функции отклика быстро спадает до нуля
- сети с RBF испытывают трудности, когда число входов велико.

Обучение RBF-сети проходит в несколько этапов:

1. Определяются центры и отклонения для радиальных элементов
2. Оптимизируются параметры линейного выходного слоя.

Расположение центров должно соответствовать кластерам, реально присутствующим в исходных данных . Рассмотрим два наиболее часто используемых метода:

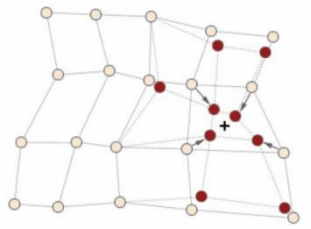
- выборка из выборки

- алгоритм К-средних.

После того, как определено расположение центров, нужно найти отклонения. Величина отклонения определяет «остроту» гауссовой функции. Используют несколько методов:

- явный
- изотропный
- К ближайших соседей

Сеть Кохонена



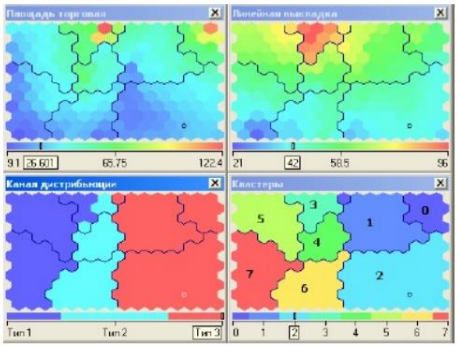
Сеть рассчитана на неуправляемое обучение.

Возможности применения:

- Распознавать кластеры в данных
- Устанавливать близость классов
- Выявлять новизну данных.

В процессе обучения и функционирования сеть выполняет 3 процесса:

- Конкуренция
- Объединение
- подстройка весов



Обучение сети Кохонена:

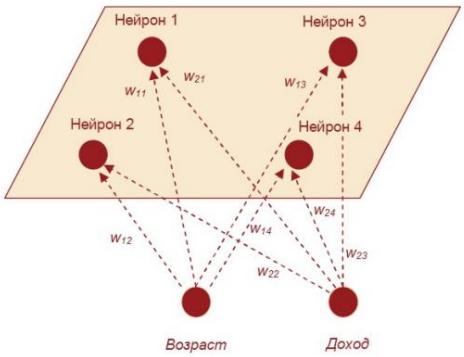
$$\omega_{\text{ij,новое}} = \omega_{\text{ij,текущее}} + \eta(x_{ni} - \omega_{ij,текущее})$$

$0 < \eta < 1$ - коэффициент скорости обучения.

Процесс обучения сети можно разделить на две фазы:
- фаза грубой подстройки;
- фаза точной подстройки.

После того, как сеть обучена распознаванию структуры данных, ее можно использовать как средство визуализации при анализе данных, при котором можно определить разбивается ли карта на отдельные кластеры.
Далее, как только кластеры выявлены, нейроны топологической карты помечаются содержательными по смыслу метками (в некоторых случаях помечены могут быть и отдельные наблюдения).

Пример работы сети Кохонена



Задача:

Имеется множество данных, в которых содержатся атрибуты «Возраст» и «Доход», которые были предварительно нормализованы

$$w_{11} = 0{,}9 \quad w_{21} = 0{,}8 \quad w_{12} = 0{,}9 \quad w_{22} = 0{,}2$$

$$w_{13} = 0{,}1 \quad w_{23} = 0{,}8 \quad w_{14} = 0{,}1 \quad w_{24} = 0{,}2$$

N	x_{i1}	x_{1j}	Описание
1	$x_{11} = 0,8$	$x_{12} = 0,8$	Пожилой человек с высоким доходом
2	$x_{21} = 0,8$	$x_{22} = 0,1$	Пожилой человек с низким доходом
3	$x_{31} = 0,2$	$x_{32} = 0,8$	Молодой человек с высоким доходом
4	$x_{41} = 0,1$	$x_{42} = 0,8$	Молодой человек с низким доходом