PyTorch и полносвязные нейронные сети

Повторение

In [1]:

```
from IPython.display import clear_output
import matplotlib.pyplot as plt
import seaborn as sns
sns.set(palette='Set2', font_scale=1.5)

import numpy as np
from sklearn.datasets import load_boston
import torch
from torch import nn
import torch.nn.functional as F
print(torch.__version__)
```

```
1.12.1+cu113
```

Простой пример обучения нейронной сети

1 Цикл обучения модели

Пусть у нас есть вход $x \in \mathbb{R}^d$. Мы построили модель (нейронную сеть) $model$, состоящую из обучаемых параметров w и b. На выходе модель возвращает некоторый ответ $\hat{y} = model(x)$. Для обучения такой сети мы задаем функцию, которую будем минимизировать. Тогда процесс обучения задается так :

- **Прямой проход / Forward pass**
  Считаем $\hat{y}$ и также запоминаем значения выходов всех слоев;
- **Вычисление оптимизируемой функции**
  Вычисляем оптимизируемую функцию на текущем наборе объектов;
- **Обратный проход / Backward pass**
  Считаем градиенты по всем обучаемым параметрам и запоминаем их;
- **Шаг оптимизации**
  Делаем шаг градиентного спуска, обновляя все обучаемые веса.

1.1 Линейная регрессия

Сделаем одномерную линейную регрессию на датасете boston.

Скачиваем данные.

In [2]:

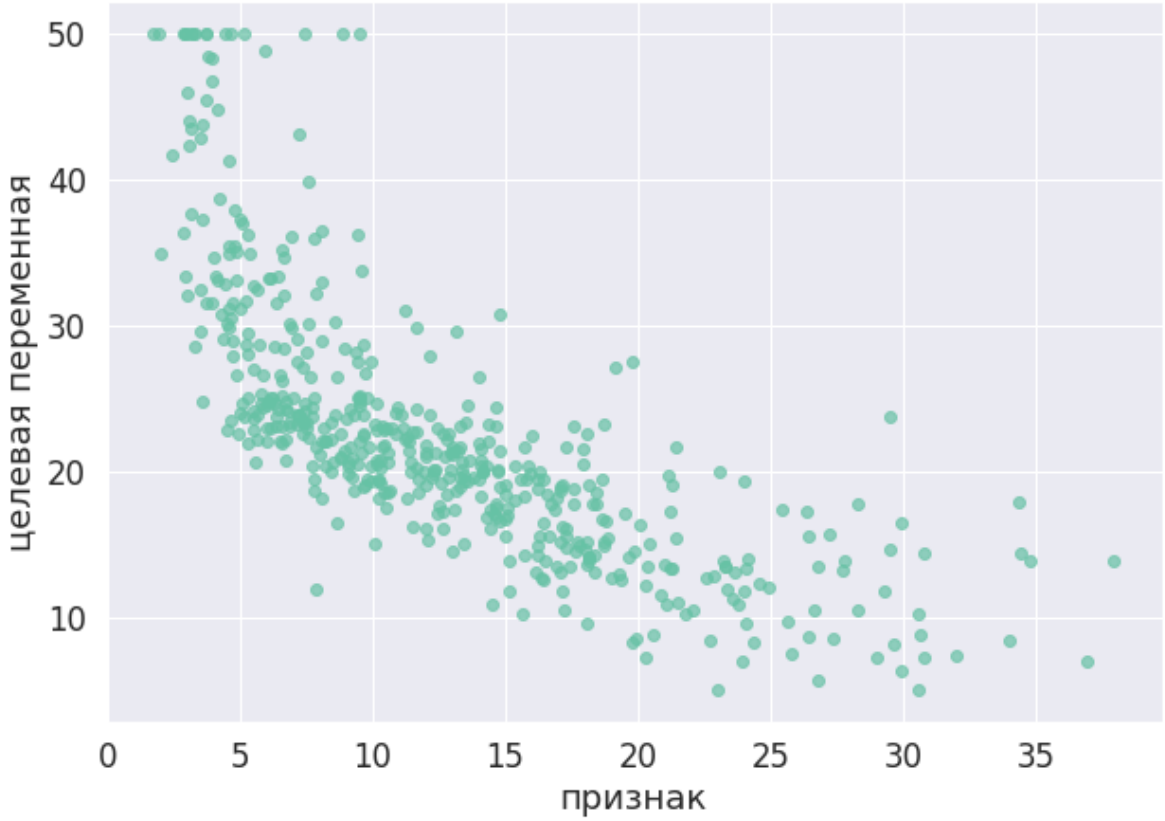
```
1 boston = load_boston()
```

```
/usr/local/lib/python3.7/dist-packages/sklearn/utils/deprecation.py:8
7: FutureWarning: Function load_boston is deprecated; `load_boston` is
deprecated in 1.0 and will be removed in 1.2.

    The Boston housing prices dataset has an ethical problem. You can
refer to
    the documentation of this function for further details.

    The scikit-learn maintainers therefore strongly discourage the use
of this
    dataset unless the purpose of the code is to study and educate abo
ut
    ethical issues in data science and machine learning.

    In this special case, you can fetch the dataset from the original
    source::

        import pandas as pd
        import numpy as np


        data_url = "http://lib.stat.cmu.edu/datasets/boston"
        raw_df = pd.read_csv(data_url, sep="\s+", skiprows=22, header=
None)
        data = np.hstack([raw_df.values[::2, :], raw_df.values[1::2, :
2]])
        target = raw_df.values[1::2, 2]

    Alternative datasets include the California housing dataset (i.e.
    :func:`~sklearn.datasets.fetch_california_housing`) and the Ames h
ousing
    dataset. You can load the datasets as follows::

        from sklearn.datasets import fetch_california_housing
        housing = fetch_california_housing()

    for the California housing dataset and::

        from sklearn.datasets import fetch_openml
        housing = fetch_openml(name="house_prices", as_frame=True)

    for the Ames housing dataset.

  warnings.warn(msg, category=FutureWarning)
```

Будем рассматривать зависимость таргета от последнего признака в данных.

In [3]:

```
plt.figure(figsize=(10,7))
plt.scatter(boston.data[:, -1], boston.target, alpha=0.7)
plt.xlabel('признак')
plt.ylabel('целевая переменная')
plt.grid(':');
```



В данном случае ответ модели задается следующим образом:

$$\widehat{y}(x) = wx + b.$$

Объявляем обучаемые параметры: тут у нас всего 2 скалярных параметра (w, b).
Также задаем вход x и отклики y в виде torch-тензоров.

In [4]:

```
# создаем два тензора размера 1 с заполнением нулями,
# для которых будут вычисляться градиенты
w = torch.zeros(1, requires_grad=True)
b = torch.zeros(1, requires_grad=True)

# Данные оборачиваем в тензоры, по которым не требуем вычисления градиента
x = torch.FloatTensor(boston.data[:, -1] / 10)
y = torch.FloatTensor(boston.target)

# по-другому:
# x = torch.tensor(boston.data[:, -1] / 10, dtype=torch.float32)
# y = torch.tensor(boston.target, dtype=torch.float32)
```

In [5]:

```
print(x.shape)
print(y.shape)
```

```
torch.Size([506])
torch.Size([506])
```

Задалим оптимизируемую функцию — MSE, и сделаем обратный проход `loss.backward()` :

$$MSE(\hat{y}, y) = \frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2.$$

In [6]:

```
def optim_func(y_pred, y_true):
    return torch.mean((y_pred - y_true) ** 2)
```

In [7]:

```
# Прямой проход
y_pred = w * x + b

# Подсчет лосса
loss = optim_func(y_pred, y)

# Вычисление градиентов
# с помощью обратного прохода по сети
# и сохранение их в памяти сети
loss.backward()
```

`loss` — значение функции MSE, вычисленное на этой итерации.

In [8]:

```
loss
```

Out[8]:

```
tensor(592.1469, grad_fn=<MeanBackward0>)
```

К градиентам для параметров, которые требуют градиента (`requires_grad=True`), теперь можно обратиться следующим образом:

In [9]:

```
print("dL/dw =", w.grad)
print("dL/b =", b.grad)
```

```
dL/dw = tensor([-47.3514])
dL/b = tensor([-45.0656])
```

Если мы посчитаем градиент M раз, то есть M раз вызовем `loss.backward()` , то градиент будет накапливаться (суммироваться) в параметрах, требующих градиента. Иногда это бывает удобно.

Убедимся на примере, что именно так все и работает.

In [10]:

```
y_pred = w * x + b
loss =  optim_func(y_pred, y)
loss.backward()

print("dL/dw =", w.grad)
print("dL/b =", b.grad)
```

```
dL/dw = tensor([-94.7029])
dL/b = tensor([-90.1312])
```

Видим, что значения градиентов стали в 2 раза больше, за счет того, что мы сложили одни и те же градиенты 2 раза.

Если же мы не хотим, чтобы градиенты суммировались, то нужно **занулять градиенты** между итерациями после того как сделали шаг градиентного спуска. Это можно сделать с помощью функции `zero_` для градиентов.

In [11]:

```
w.grad.zero_()
b.grad.zero_()
w.grad, b.grad
```

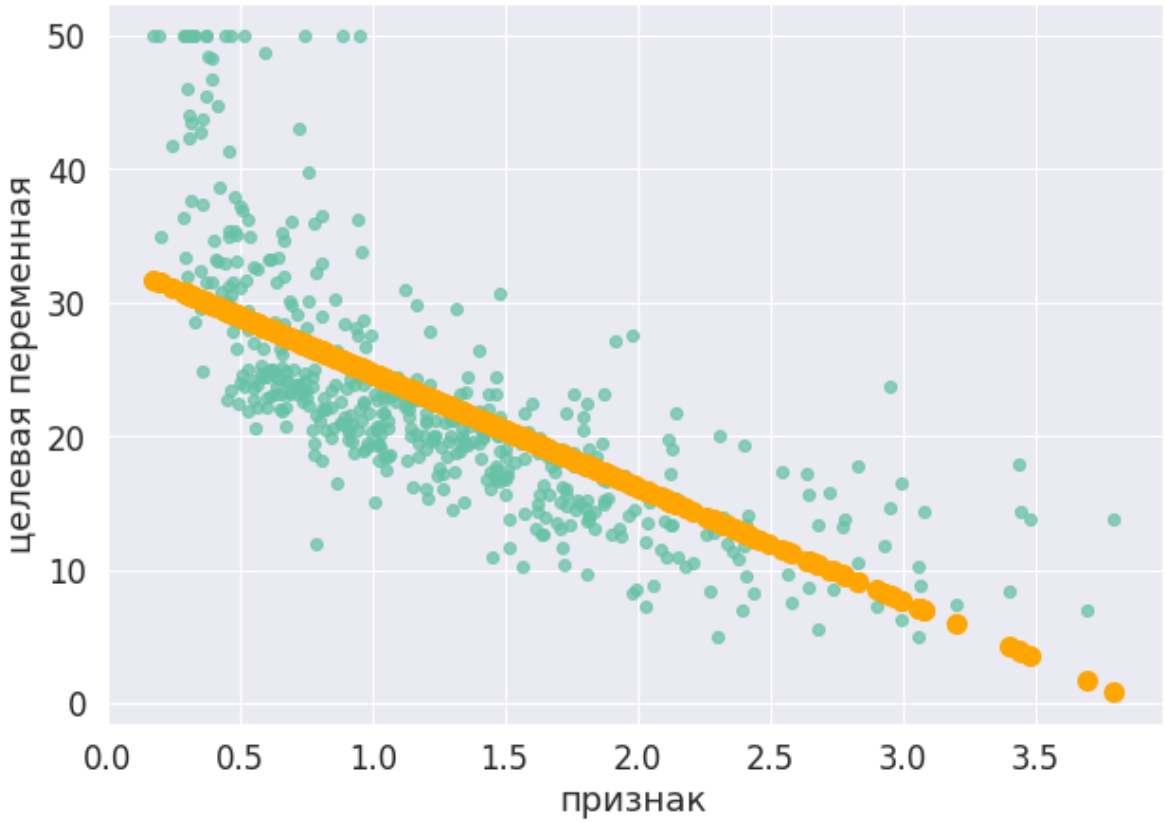
Out[11]:

```
(tensor([0.]), tensor([0.]))
```

Напишем код, обучающий нашу модель.

In [12]:

```python
# Инициализация параметров
w = torch.zeros(1, requires_grad=True)
b = torch.zeros(1, requires_grad=True)

# Количество итераций
num_iter = 1000

# Скорость обучения для параметров
lr_w = 0.01
lr_b = 0.05

for i in range(num_iter):

    # Forward pass: предсказание модели
    y_pred = w * x + b

    # Подсчет оптимизируемой функции (MSE)
    loss = optim_func(y_pred, y)

    # Обратный проход: подсчет градиентов
    loss.backward()

    # Оптимизация: обновение параметров
    w.data -= lr_w * w.grad.data
    b.data -= lr_b * b.grad.data

    # Зануление градиентов
    w.grad.zero_()
    b.grad.zero_()


    # График + вывод MSE
    if (i + 1) % 5 == 0:

        # Избавимся от градиентов перед отрисовкой графика
        y_pred = y_pred.detach()

        # Превратим тензор размерности 0 в число, для краисивого отображения
        loss = loss.item()

        clear_output(True)
        plt.figure(figsize=(10,7))
        plt.scatter(x, y, alpha=0.75)
        plt.scatter(x, y_pred, color='orange', linewidth=5)
        plt.xlabel('признак')
        plt.ylabel('целевая переменная')
        plt.show()


        print("MSE = ", loss)
        if loss < 39:
            print("Done!")
            break
```




```
MSE =  38.977848052978516
Done!
```

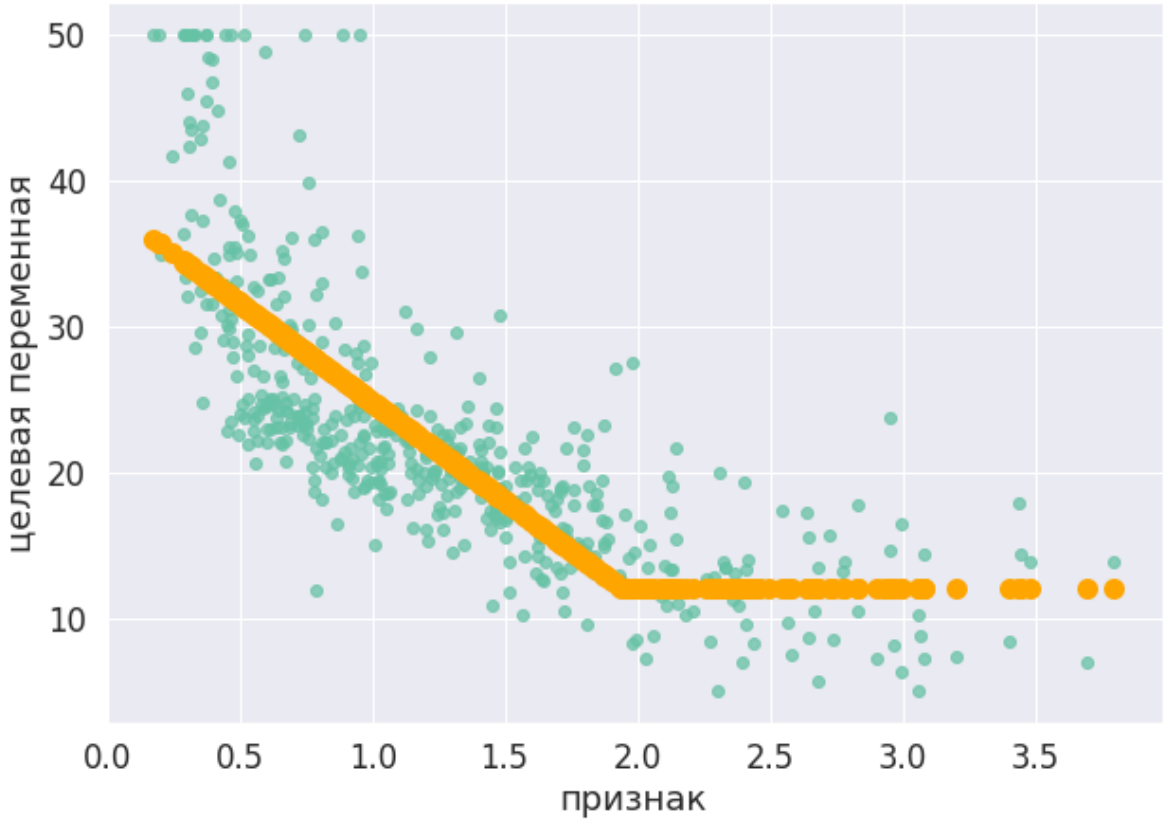
2.2 Улучшение модели

Попробуем усложнить модель. Сделаем еще один слой.

In [13]:

```
# Инициализация параметров
w0 = torch.ones(1, requires_grad=True)
b0 = torch.ones(1, requires_grad=True)
w1 = torch.ones(1, requires_grad=True)
b1 = torch.ones(1, requires_grad=True)

# Функция активации
def act_func(x):
    return x * (x >= 0)

# Количество итераций
num_iter = 1000

# Скорость обучения для параметров
lr_w = 0.01
lr_b = 0.05

for i in range(num_iter):

    # Forward pass: предсказание модели
    y_pred =  w1 * act_func(w0 * x + b0) + b1

    # Подсчет оптимизируемой функции (MSE)
    loss = optim_func(y_pred, y)

    # Bakcward pass: подсчет градиентов
    loss.backward()

    # Оптимизация: обновление параметров
    w0.data -= lr_w * w0.grad.data
    b0.data -= lr_b * b0.grad.data
    w1.data -= lr_w * w1.grad.data
    b1.data -= lr_b * b1.grad.data

    # Зануление градиентов
    w0.grad.zero_()
    b0.grad.zero_()
    w1.grad.zero_()
    b1.grad.zero_()

    # График + вывод MSE
    if (i+1) % 5 == 0:

        # Избавимся от градиентов перед отрисовкой графика
        y_pred = y_pred.detach()

        # Превратим тензор размерности 0 в число, для краисивого отображения
        loss = loss.item()

        clear_output(True)
        plt.figure(figsize=(10,7))
        plt.scatter(x, y, alpha=0.75)
        plt.scatter(x, y_pred, color='orange', linewidth=5)
        plt.xlabel('признак')
        plt.ylabel('целевая переменная')
        plt.show()

        print("MSE = ", loss)
        if loss < 33:
```

```
            print("Done!")
            break
```




```
MSE =  32.993534088134766
Done!
```

Полученная модель лучше описывает данные.

На практике нейронные сети так не пишут, пользуются готовыми модулями. Напишем такую же нейросеть, но теперь с помощью pytorch. Для этого будем пользоваться `torch.nn` .

In [14]:

```
model = nn.Sequential( # собираем модули в последовательность
    nn.Linear(1, 1), # кол-во признаков во входном слое 1, в выходном тоже 1
    nn.ReLU(), # та же ф-ция активации, что и раньше, только из pytorch
    nn.Linear(1, 1) # кол-во признаков во входном слое 1, в выходном тоже 1
)
model
```

Out[14]:

```
Sequential(
  (0): Linear(in_features=1, out_features=1, bias=True)
  (1): ReLU()
  (2): Linear(in_features=1, out_features=1, bias=True)
)
```

Для того, чтобы работать с данной моделью, нам понадобится поменять размерность x и y.

In [15]:

```
x_new = x.reshape(-1, 1)
y_new = y.reshape(-1, 1)
```

Применим модель к нашим данным.

In [16]:

```
model(x_new)[:10] # результаты для первых 10 элементов
```

Out[16]:

```
tensor([[0.3273],
        [0.3273],
        [0.3273],
        [0.3273],
        [0.3273],
        [0.3273],
        [0.3273],
        [0.3273],
        [0.3273],
        [0.3273]], grad_fn=<SliceBackward0>)
```

Посмотрим на параметры модели с помощью функции `named_parameters`, которая кроме парамтеров, выдает также их названия.

In [17]:

```
for name, param in model.named_parameters():
    print(name)
    print(param.data)
```

```
0.weight
tensor([[0.0842]])
0.bias
tensor([-0.7547])
2.weight
tensor([[0.6612]])
2.bias
tensor([0.3273])
```

Инициализируем параметры так же, как мы делали для подобной модели ранее. На этот раз воспользуемся функцией `parameters`, она возвращает только параметры.

In [18]:

```
for p in model.parameters():
    p.data = torch.FloatTensor([[1]])
    print(p.data)
```

```
tensor([[1.]])
tensor([[1.]])
tensor([[1.]])
tensor([[1.]])
```

Ранее мы оптимизацию производили самостоятельно. Теперь же сделаем это с помощью оптимизатора SGD из pytorch. Установим скорость обучения на уровне 0.01 для всех параметров сразу. Также заменим нашу написанную MSE функцию на соответствуюшую из pytorch.

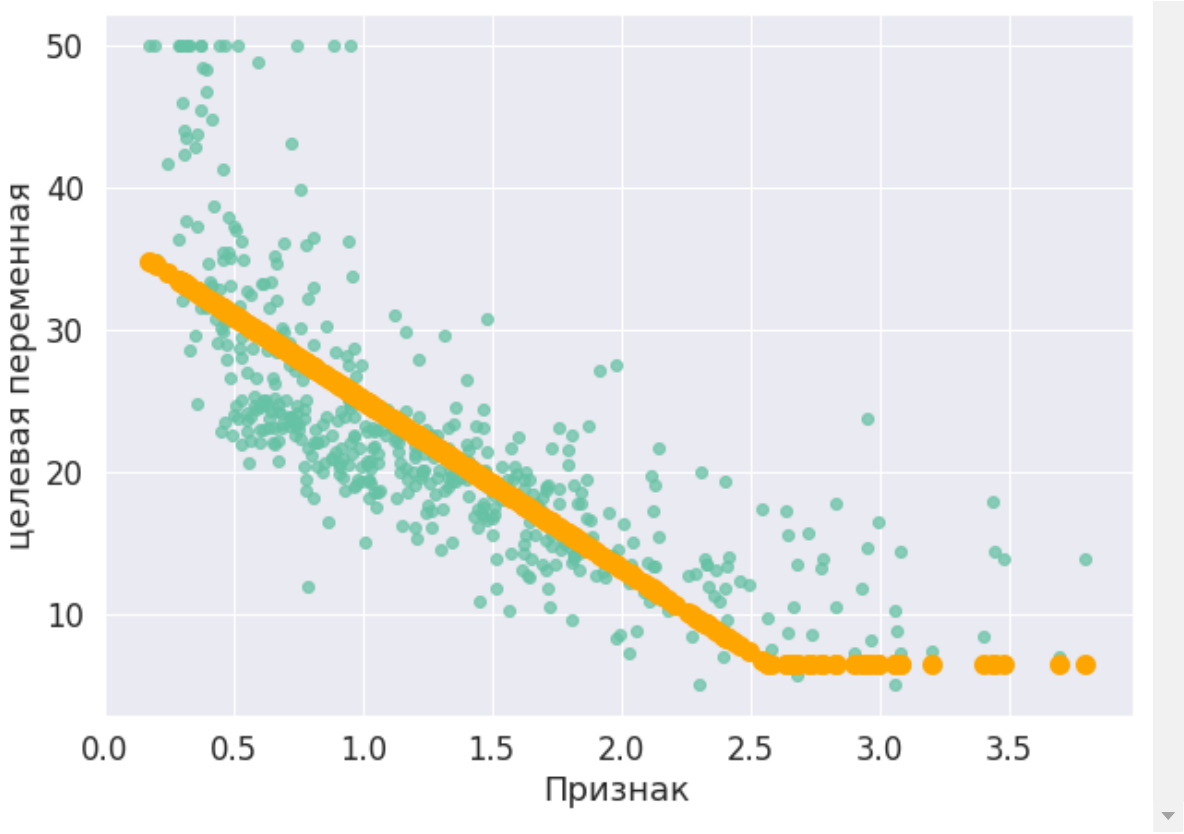
In [19]:

```
optimizer = torch.optim.SGD(model.parameters(), lr=0.01)
optim_func = nn.MSELoss()
```

Обучим полученную модель на наших данных. Теперь обновления значений параметров происходят с помощью вызова `optimzer.step()`, а зануление градиентов — `optimizer.zero_grad()`.

In [20]:

```
# Количество итераций
num_iter = 1000

for i in range(num_iter):

    # Forward pass: предсказание модели
    y_pred =  model(x_new)

    # Подсчет оптимизируемой функции (MSE)
    loss = optim_func(y_pred, y_new)

    # Bakcward pass: подсчет градиентов
    loss.backward()

    # Оптимизация: обновление параметров
    optimizer.step()

    # Зануление градиентов
    optimizer.zero_grad()

    # График + вывод MSE
    if (i+1) % 5 == 0:

        # Избавимся от градиентов перед отрисовкой графика
        y_pred = y_pred.detach()

        # Превратим тензор размерности 0 в число, для краисивого отображения
        loss = loss.item()

        clear_output(True)
        plt.figure(figsize=(10,7))
        plt.scatter(x, y, alpha=0.75)
        plt.scatter(x, y_pred, color='orange', linewidth=5)
        plt.xlabel('Признак')
        plt.ylabel('целевая переменная')
        plt.show()

        print(f"MSE = {loss:.2f}")
        if loss < 33:
            print("Готово!")
            break
```




```
MSE = 35.92
```

Полученная модель довольно хорошо приближает данные, однако дольше сходится к оптимумум за счет меньшей скорости обучения для параметров сдвига.

In [20]:

```
1
```