In [1]:

```
from collections import defaultdict
from collections import OrderedDict
import torch
from torch import nn
from torch.utils.data import Dataset
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline

import seaborn as sns
sns.set(palette='Set2', font_scale=1.2)

from IPython.display import clear_output
from sklearn.datasets import load_boston
```

```
---------------------------------------------------------------
--------------
ModuleNotFoundError                       Traceback (most rec
ent call last)
<ipython-input-1-94fe9400ed0c> in <module>
      1 from collections import defaultdict
      2 from collections import OrderedDict
----> 3 import torch
      4 from torch import nn
      5 from torch.utils.data import Dataset

ModuleNotFoundError: No module named 'torch'
```

In [3]:

```
boston = load_boston()
boston['data'];
boston['target'];
```

```
/usr/local/lib/python3.7/dist-packages/sklearn/utils/deprecat
ion.py:87: 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 discourag
e the use of this
    dataset unless the purpose of the code is to study and ed
ucate about
    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=2
2, header=None)
        data = np.hstack([raw_df.values[::2, :], raw_df.value
s[1::2, :2]])
        target = raw_df.values[1::2, 2]

    Alternative datasets include the California housing datas
et (i.e.
    :func:`~sklearn.datasets.fetch_california_housing`) and t
he Ames housing
    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)
```

Сформируем тренировочную и валидационную выборки

In [4]:

```
np.random.seed(1)
indeces = np.arange(len(boston.target))
np.random.shuffle(indeces)
```

In [5]:

```
train_ind = indeces[:int(len(indeces) * 0.8)]
test_ind = indeces[int(len(indeces) * 0.2):]
```

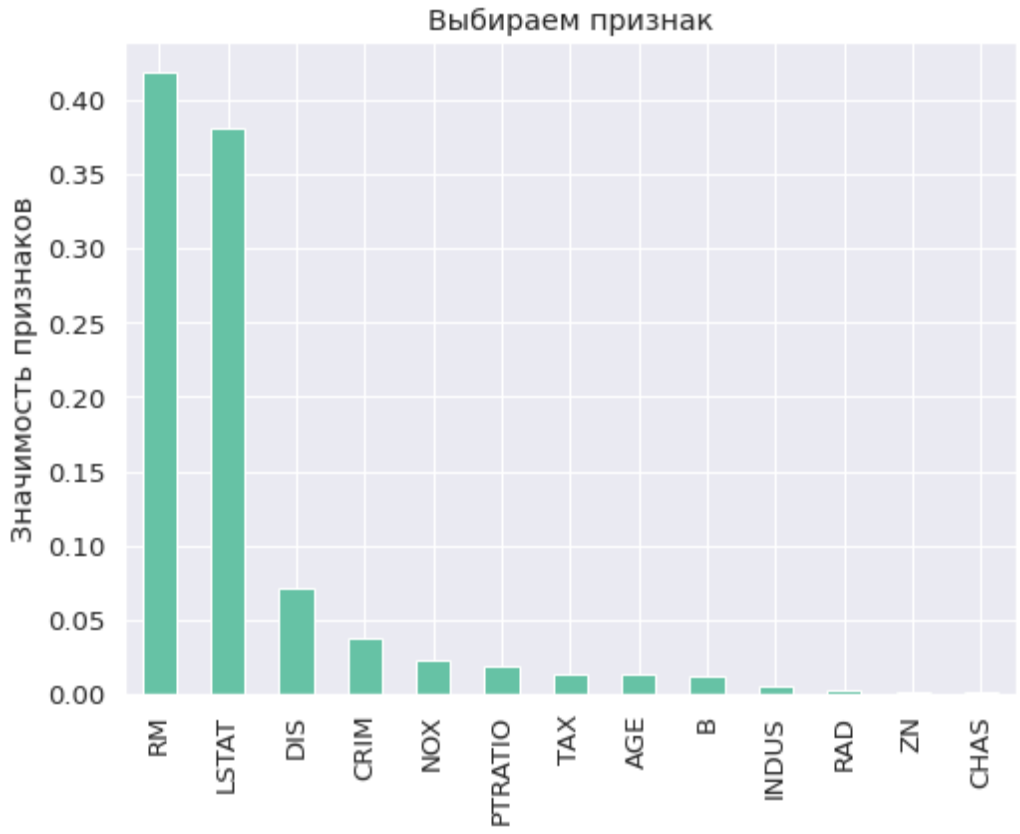
Выбор признака

Для выбора обучим случайный лес на датасете и с помощью него посмотрим важность признаков. В качестве второго выберем признак с наибольшей статистической значимостью

In [28]:

```
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split

from sklearn.ensemble import RandomForestRegressor
import pandas as pd

forest = RandomForestRegressor(random_state=12)
forest.fit(boston.data, boston.target) #Обучим случайный лес на всей выбор
ке
```

```
feature_importance = pd.Series(forest.feature_importances_, boston.feature
_names).sort_values(ascending=False)
plt.figure(figsize = (8,6))
feature_importance.plot(kind='bar', title='Выбираем признак')
plt.ylabel('Значимость признаков');
```



Берем признак RM. Это 6 признак.

Далее сформируйте датасет в pytorch-обертке

In [53]:

```
class BostonDataset(Dataset):
    def __init__(self, boston_dataset, indeces, train_split_ratio, train:
bool):
        self.indeces = indeces
        self.train_split_ratio = train_split_ratio
        self.train = train

        self.ll = len(boston_dataset.data)
```

```
        train_ind = self.indeces[:int(self.ll * self.train_split_ratio)]
        test_ind = self.indeces[int(self.ll * self.train_split_ratio):]

        if train:
            self.data = boston_dataset.data[:, [-1,5]][train_ind]
            self.labels = boston_dataset.target[train_ind]
        else:
            self.data = boston_dataset.data[:, [-1,5]][test_ind]
            self.labels = boston_dataset.target[test_ind]

    def __len__(self):
        return len(self.labels)

    def __getitem__(self, idx):
        sample = self.data[idx]/10
        label = self.labels[idx]
        return sample, label
```

In [54]:

```
trainset = BostonDataset(boston, train_ind, 1, True)
testset = BostonDataset(boston, test_ind, 0, False)
```

Создайте и обучите нейросеть

Делаем нейросеть как на семинаре

Для начала разбиваем нашу выборку на батчи

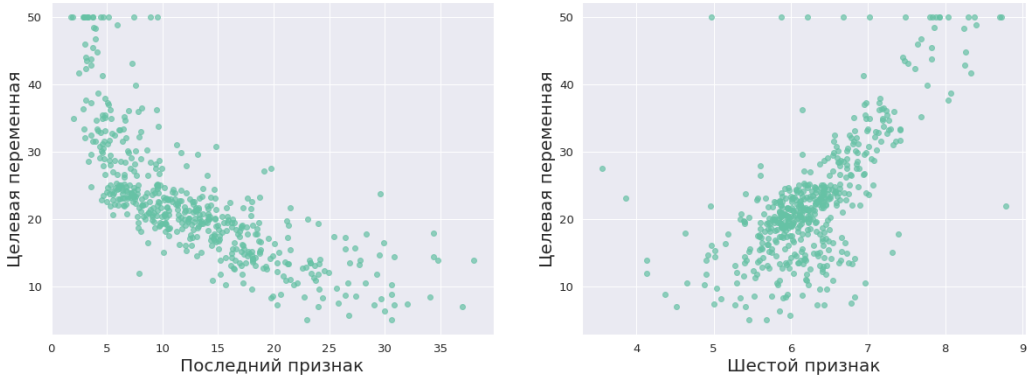
In [55]:

```
batch_size = 202

trainloader = torch.utils.data.DataLoader(trainset,
                                          batch_size=batch_size,
                                          shuffle=True)

testloader = torch.utils.data.DataLoader(testset,
                                         batch_size= testset.__len__(),
                                         shuffle=False)
```

Теперь пишем саму модель. Чтобы понять, какие слои стоит добавить, посмотрим как выглядят данные.

In [32]:

```
fig = plt.figure(figsize=(20,7))

ax = fig.add_subplot(1,2,1)
ax.scatter(boston.data[:, -1], boston.target, alpha=0.7)
ax.set_xlabel('Последний признак', fontsize = 20)
ax.set_ylabel('Целевая переменная', fontsize = 20)

ax = fig.add_subplot(1,2,2)
ax.scatter(boston.data[:, 5], boston.target, alpha=0.7)
ax.set_xlabel('Шестой признак', fontsize = 20)
ax.set_ylabel('Целевая переменная', fontsize = 20);
```



Видно, что подойдут слои с семинара

In [33]:

```
torch.manual_seed(2)
# Создаем последовательную нейронную сеть
model = nn.Sequential()

# Добавляем линейный слой с выходным размером 1.
# Размер входа равен произведению размерностей данных.
model.add_module('linear_1', nn.Linear(2, 3))
model.add_module('relu_', nn.ReLU())
model.add_module('linear_2', nn.Linear(3, 1))
```

В качестве критерия возьмём MSE

In [34]:

```
criterion = nn.MSELoss()
```

Инициализируем веса

In [35]:

```
# функция для инициализаци весов
def init_weights(model):
    if isinstance(model, nn.Linear):
        torch.nn.init.normal_(model.weight)

# Инициализация весов
model.apply(init_weights)
```

Out[35]:

```
Sequential(
  (linear_1): Linear(in_features=2, out_features=3, bias=Tru
e)
  (relu_): ReLU()
  (linear_2): Linear(in_features=3, out_features=1, bias=Tru
e)
)
```

Переводим модель на GPU

In [36]:

```
device = 'cuda:0'
model.to(device);
```

Возьмём SGD в качестве оптимизатора

In [37]:

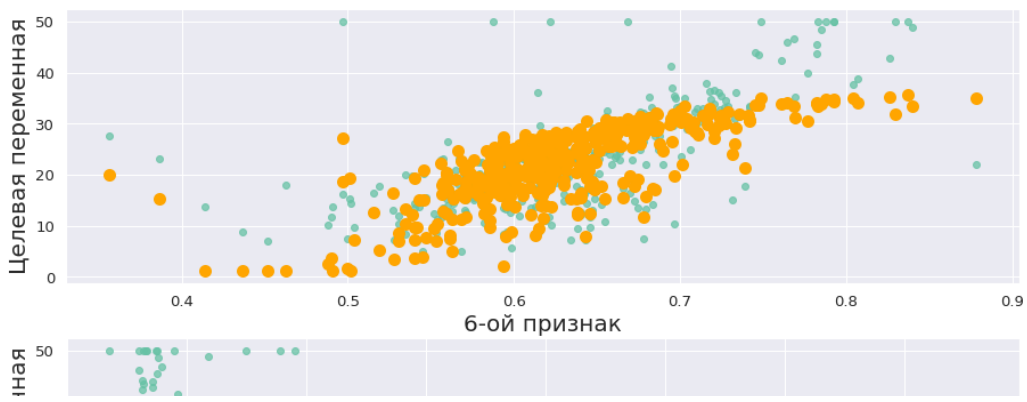
```
opt = torch.optim.SGD(model.parameters(), lr=0.005)
```

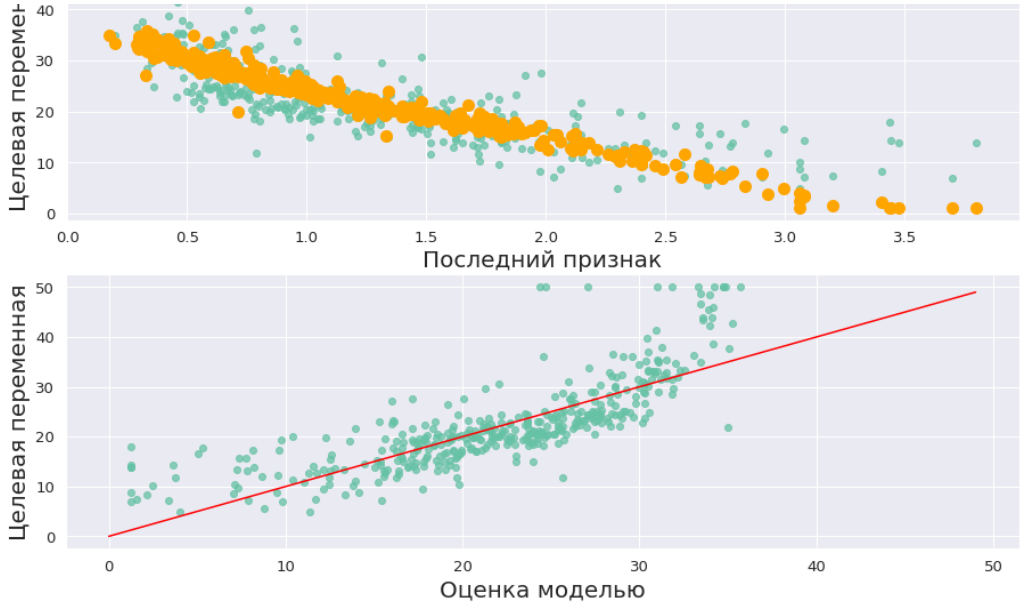
Теперь обучим нашу нейронную сеть. Посмотрим на графики и лосс.

In [58]:

```
num_epochs = 100  # общее кол-во полных проходов ("эпох") по обучаемым данным

num_train_batches = (trainset.__len__()) // batch_size
num_val_batches = 1 # так как для валидационного набора батч - вся выборка (мы так задали размер батча)

history = defaultdict(lambda: defaultdict(list))

for epoch in range(num_epochs):
    print(f'epoch {epoch}')

    train_loss = 0
    val_loss = 0


    # Устанавливаем режим обучения
    model.train(True)

    # На каждой "эпохе" делаем полный проход по данным
    for _, data in enumerate(trainloader):

        X_batch = data[0].to(device).float()
        y_batch = torch.unsqueeze(data[1].to(device), 1).float()

        # Обучаемся на батче (одна "итерация" обучения нейросети)
        outs = model(X_batch)
        loss = criterion(outs, y_batch)

        # Обратный проход, шаг оптимизатора и зануление градиентов
        loss.backward()

        opt.step()
        opt.zero_grad()


        train_loss += loss.detach().cpu().numpy()
        y_pred_np = outs.detach().cpu().numpy()
        y_batch_np = y_batch.cpu().numpy()

    # Подсчитываем лоссы и сохраням в "историю"
    train_loss /= num_train_batches
    history['loss']['train'].append(train_loss)
```

```
    # Устанавливаем режим тестирования
    model.eval()

    # Полный проход по валидации
    with torch.no_grad(): # Отключаем подсчет градиентов, то есть detach н
е нужен
        for _, data_test in enumerate(testloader):
            X_batch = data_test[0].to(device).float()
            #X_batch = torch.unsqueeze(X_batch, 1).float()
            y_batch = torch.unsqueeze(data_test[1].to(device).float(), 1)

            outs = model(X_batch)
            loss = criterion(outs, y_batch)

            val_loss += loss.cpu().numpy().sum()
            y_pred_np = outs.cpu().numpy()
            y_batch_np = y_batch.cpu().numpy()

    # Подсчитываем лоссы и сохраням в "историю"
    val_loss /= num_val_batches
    history['loss']['val'].append(val_loss)

    # График + вывод лосса
    clear_output(True)
    fig, ax = plt.subplots(3, 1, figsize = (15, 15))

    ax[0].scatter(X_batch[:,1].cpu().numpy(), y_batch.cpu().data.numpy(),
alpha=0.75)
    ax[0].scatter(X_batch[:,1].cpu().numpy(), y_pred_np, color='orange', l
inewidth=5)
    ax[0].set_xlabel('6-ой признак', fontsize = 20)
    ax[0].set_ylabel('Целевая переменная', fontsize = 20)

    ax[1].scatter(X_batch[:,0].cpu().numpy(), y_batch.cpu().data.numpy(),
alpha=0.75)
    ax[1].scatter(X_batch[:,0].cpu().numpy(), y_pred_np, color='orange', l
inewidth=5)
    ax[1].set_xlabel('Последний признак', fontsize = 20)
    ax[1].set_ylabel('Целевая переменная', fontsize = 20)

    ax[2].scatter(y_pred_np, y_batch.cpu().data.numpy(),  alpha=0.75)
    ax[2].plot(np.arange(y_batch.cpu().max()), np.arange(y_batch.cpu().max
()),  color = 'red')
    ax[2].set_xlabel('Оценка моделью', fontsize = 20)
    ax[2].set_ylabel('Целевая переменная', fontsize = 20)

    plt.show()
```

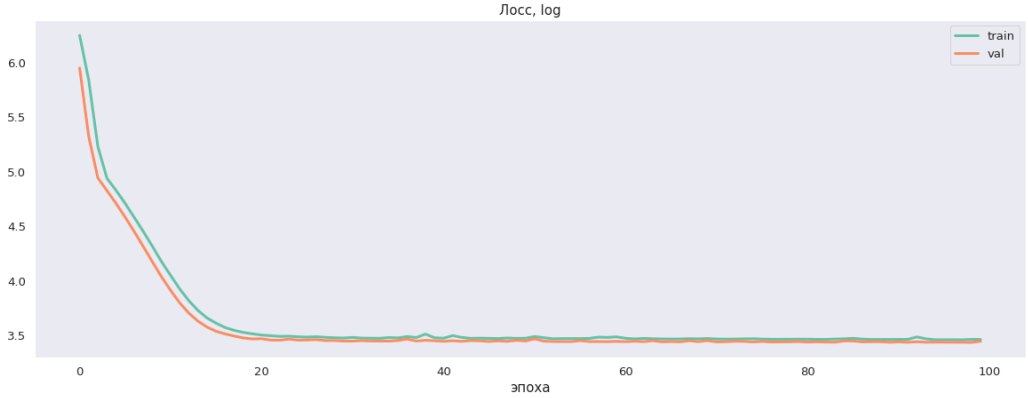




Посмотрим на кривую обучения

In [39]:

```
fig = plt.figure(figsize=(20,7))

plt.subplot(1,1,1)
plt.title('Лосс, log', fontsize=15)
plt.plot(np.log(history['loss']['train']), label='train', linewidth = 3)
plt.plot(np.log(history['loss']['val']), label='val', linewidth = 3)
plt.grid()
plt.xlabel('эпоха', fontsize=15)
plt.legend();
```



Задача 2

С начало разберемся, какие вообще есть гиперпараметры: manual_seed, размеры train/test выборок, batch_size, слои нейросети, начальная инициализация, количество эпох, оптимизатор и его параметры, . Рассмотрим каждый по отдельности.

1)Manual_seed - случайный параметр. Нет смысла рассматривать его как гиперпараметр.

2)Размеры выборок играют только на силу переобучения и значения лоса, не так сильно на веса модели. Данные параметры можно выбрать из качественных соображений, рассматривать их дальше тоже не буедм.

3)От batch_size зависит только время/вычислительная сложность работы. Сколько в итоге вся выборка пройдёт через нейросеть определяется не этим параметром, следовательно, на веса он тоже не влияет.

4)Слои нейрости должны тоже выбираться из качественных соображений. Исследовать их как гиперпараметр, особенно в такой простой задаче, где, по-сути, любые слои дадут правильный ответ - глупо.

5)Начальная инициализация, если он плохая, может привести нас не в глобальный, а локальный минимум, например. Но, если, всё-таки, этого не случается, то менять её тоже не стоит и от неё мало чего зависит

6)Количество эпох также не очень важный параметр. Надо просто поставить критерий остановки, при котором наши итирации закончатся, а количество эпох делать достаточно большим. Поэтому, рассматривать как гиперпараметр это тоже не будем.

7)Оптимизатор и его параметры сильно вляют на результат, в чем я убедился в предыдущей задаче. Как видно из пункта 4 задачи три мне даже удалось переобучить двухслойную нейронную сеть, неправильно выбрав оптимизатор и его параметр. Главный параметр оптимизатора SGD - learning rate. Посмотрим как он влияет на результаты модели.

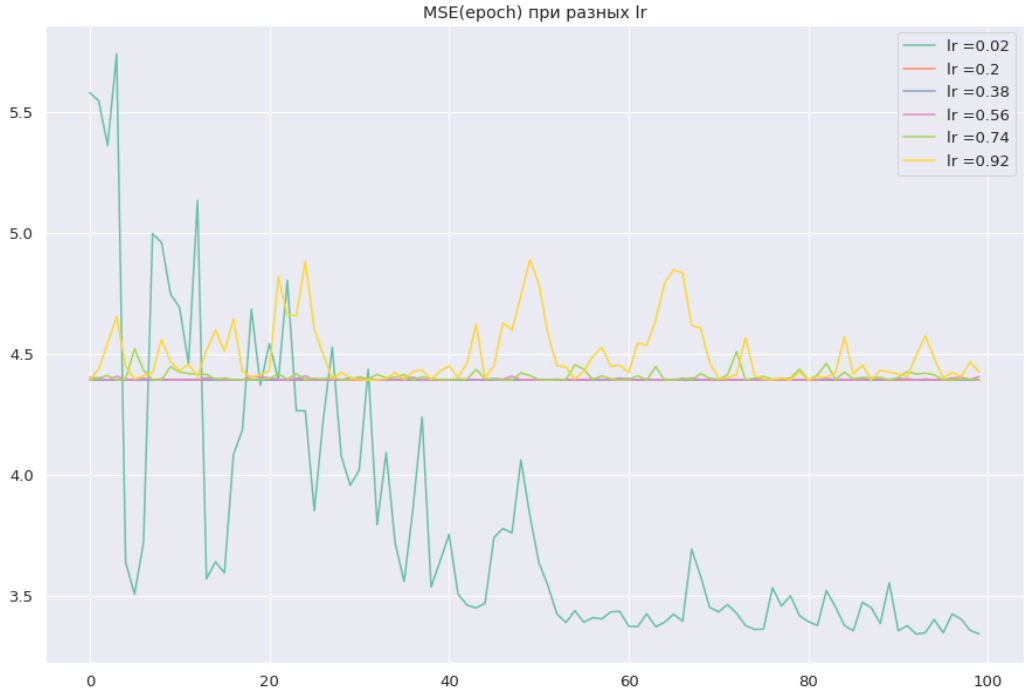
In [112]:

```
#Собрал весь код, чтобы увидеть весь пайплан сразу
#Train и test выборка
trainset = BostonDataset(boston, train_ind, 1, True)
testset = BostonDataset(boston, test_ind, 0, False)
#Батчи
batch_size = 202
trainloader = torch.utils.data.DataLoader(trainset, batch_size=batch_size,
shuffle=True)
testloader = torch.utils.data.DataLoader(testset, batch_size= testset.__le
n__(), shuffle=False)
torch.manual_seed(2)
# Создаем последовательную нейронную сеть
model = nn.Sequential()
# Добавляем линейный слой с выходным размером 1.
# Размер входа равен произведению размерностей данных.
model.add_module('linear_1', nn.Linear(2, 3))
model.add_module('relu_', nn.ReLU())
model.add_module('linear_2', nn.Linear(3, 1))
criterion = nn.MSELoss()
# Инициализация весов
model.apply(init_weights)
#Переход на cuda
model.to(device)
#Обучаем нейросеть
num_epochs = 100  # общее кол-во полных проходов ("эпох") по обучаемым дан
ным
num_train_batches = (trainset.__len__()) // batch_size #Количество train б
атчей
num_val_batches = 1 #Кол-во тест батчей
MSE = np.zeros((50,100)) #Создаём пустой массив MSE, чтобы потом посмотрет
ь на зависимости

for i, lr in enumerate(np.linspace(0,1,50)):
  history = defaultdict(lambda: defaultdict(list)) #Каждый раз нужна чиста
я история
```

```
  #Выбор оптимизатора
  opt = torch.optim.SGD(model.parameters(), lr=lr)

  for epoch in range(num_epochs):
      train_loss = 0
      val_loss = 0
      # Устанавливаем режим обучения
      model.train(True)
      # На каждой "эпохе" делаем полный проход по данным
      for _, data in enumerate(trainloader):
          X_batch = data[0].to(device).float()
          y_batch = torch.unsqueeze(data[1].to(device), 1).float()
          # Обучаемся на батче (одна "итерация" обучения нейросети)
          outs = model(X_batch)
          loss = criterion(outs, y_batch)
          # Обратный проход, шаг оптимизатора и зануление градиентов
          loss.backward()
          opt.step()
          opt.zero_grad()

          train_loss += loss.detach().cpu().numpy()
          y_pred_np = outs.detach().cpu().numpy()
          y_batch_np = y_batch.cpu().numpy()

      # Подсчитываем лоссы и сохраням в "историю"
      train_loss /= num_train_batches
      history['loss']['train'].append(train_loss)

      # Устанавливаем режим тестирования
      model.eval()

      # Полный проход по валидации
      with torch.no_grad(): # Отключаем подсчет градиентов, то есть detach
не нужен
          for _, data_test in enumerate(testloader):
              X_batch = data_test[0].to(device).float()
              y_batch = torch.unsqueeze(data_test[1].to(device).float(), 1
)
              outs = model(X_batch)
              loss = criterion(outs, y_batch)
              val_loss += loss.cpu().numpy().sum()
              y_pred_np = outs.cpu().numpy()
              y_batch_np = y_batch.cpu().numpy()
      # Подсчитываем лоссы и сохраням в "историю"
      val_loss /= num_val_batches
      history['loss']['val'].append(val_loss)
  MSE[i] = np.array(history['loss']['val'])
```

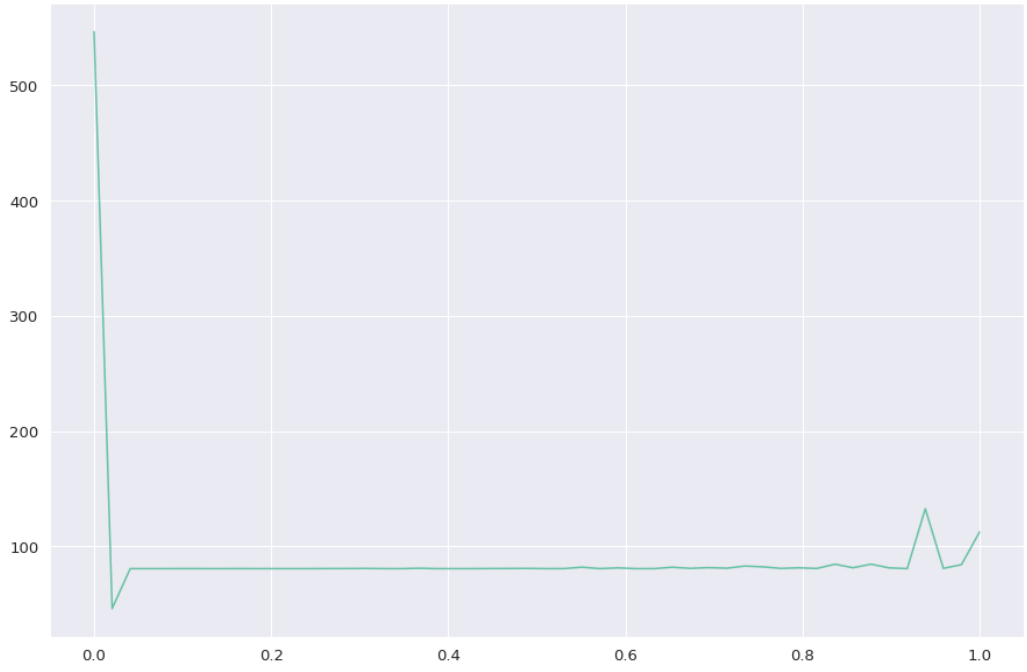
In [127]:

```
plt.figure(figsize=(15,10))
for i in range(1,50,9):
  plt.plot([i for i in range(100)], np.log(MSE[i]), label = 'lr =' + str(i
* 1 / 50))
plt.title('MSE(epoch) при разных lr')
plt.legend();
```



In [136]:

```
plt.figure(figsize = (15,10))
plt.plot(np.linspace(0,1,50),MSE[:,49], label = 'MSE(lr)');
```



Из очевидного - лосс при нулевом LR очень высок, а при LR->1 начинает расти из-за переобучения. В определенном интервале лосс почти не меняется, что связано с тем, что за 100 эпох наша сеть попадает в локальный экстремум при любой LR в данном интервале.

Виден отчетливый минимум в области нуля. Следовательно, при определенном значении LR наша модель попадет не в локальный, а глобальный экстремум.

Задача 3

1. Переобучение - подгон под тренировочную выборку. Существует классический пример с многочленами. Допустим, у нас есть 10 точек, которые лежат примерно на одной прямой. Логично было бы профитировать прямую. Но можно профитировать многочлен 10 степени. В итоге, на тренировочной выборке точность будет 100%, но, очвеидно, если добавить пару точек, наш многочлен даст очень плохой результат.
2. Оно может произойти при неправильном выборе модели(можно взять слишком сложную модель) и сильном подгоне под тренировочную выборку(высокий лернинг рейт).
3. Нет, не всегда. Если у нас уже выбрана модель, то может быть такое, что она просто не может идеально выдавать значения такие же, как на трейне(просто по формуле).
4. Да. Просто добавить столько слоёв, сколько всего точек, например. Но даже это не обязательно. Поставив слишком высокий лернинг рейт в первой задаче получил следующее.