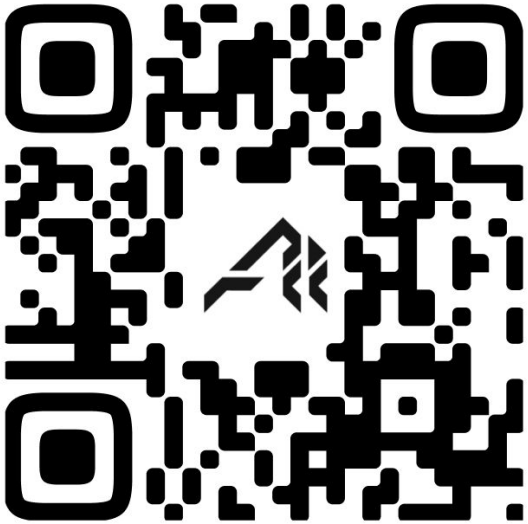
Лекция 2. Устройства полносвязных сетей - обучение MLP

Ссылка!



>> ТЕЛЕГРАММ КАНАЛ

>> YOUTUBE КАНАЛ

О лекторах

Душенев Даниил
@daniil_d_d

Карпов Назар
@rezvey

План лекции

- Повторение 1ой лекции
- Многослойные полносвязные нейросети
- Функции активации
- Forward и Backward propagation
- Autograd, что это такое и как реализован в PyTorch
- Autograd, считаем руками
- Градиентный спуск и его реализации
- Метрики оценки качества модели в задаче классификации

Вспоминаем основные понятия

Вспоминаем основные понятия

1. DL

Вспоминаем основные понятия

1. DL
2. Нейрон и линейный слой

Вспоминаем основные понятия

1. DL
2. Нейрон и линейный слой
3. Функция активации

Вспоминаем основные понятия

1. DL
2. Нейрон и линейный слой
3. Функция активации
4. Лосс

Вспоминаем основные понятия

1. DL
2. Нейрон и линейный слой
3. Функция активации
4. Лосс
5. Градиентный спуск

FNN

FNN (Feedforward Neural Network) - полносвязная нейронная сеть

С помощью нее, мы можем решать задачи с **нелинейной** зависимостью

Нелинейность добавляют **функции активации**

Функции активации

- Сигмоида
- Tanh
- ReLU
- Softmax

Как учить модель?

Мы считаем хотим изменить параметры модели так, чтобы значение лосс-функции уменьшалось

Чтобы это сделать, мы можем использовать градиентный спуск, но для этого нужно считать градиент (частную производную по каждому параметру модели).

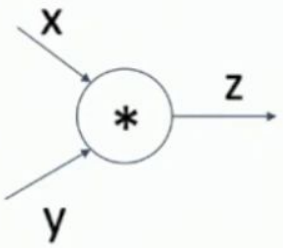
Как это сделать **эффективно**?

Метод обратного распространения ошибки

1. Строим вычислительный граф
2. Forward-pass, записываем локальные значения
3. Вспоминаем Chain Rule
4. DownStream = Local * Upstream
5. …
6. Профит!

Метод обратного распространения ошибки

Example: PyTorch Autograd Functions



(x,y,z are scalars)

```
class Multiply(torch.autograd.Function):
  @staticmethod
  def forward(ctx, x, y):
    ctx.save_for_backward(x, y)
    z = x * y
    return z
  @staticmethod
  def backward(ctx, grad_z):
    x, y = ctx.saved_tensors
    grad_x = y * grad_z   # dz/dx * dL/dz
    grad_y = x * grad_z   # dz/dy * dL/dz
    return grad_x, grad_y
```

Need to stash some values for use in backward

Upstream gradient

Multiply upstream and local gradients

Метод обратного распространения ошибки

PyTorch sigmoid layer

```
#ifndef TH_GENERIC_FILE
#define TH_GENERIC_FILE "THNN/generic/Sigmoid.c"
#else

void THNN_(Sigmoid_updateOutput)(
          THNNState *state,
          THTensor *input,
          THTensor *output)
{
  THTensor_(sigmoid)(output, input);
}
```

Forward

$$\sigma(x) = \frac{1}{1+e^{-x}}$$

```
void THNN_(Sigmoid_updateGradInput)(
          THNNState *state,
          THTensor *gradOutput,
          THTensor *gradInput,
          THTensor *output)
{
  THNN_CHECK_NELEMENT(output, gradOutput);
  THTensor_(resizeAs)(gradInput, output);
  TH_TENSOR_APPLY3(scalar_t, gradInput, scalar_t, gradOutput, scalar_t, output,
    scalar_t z = *output_data;
    *gradInput_data = *gradOutput_data * (1. - z) * z;
  );
}
```

Backward

$$(1-\sigma(x))\,\sigma(x)$$

Метод обратного распространения ошибки

Подробнее смотри тут:

Градиентный спуск (GD)

$$x_{k+1} = x_k - \alpha \nabla f(x_k),$$

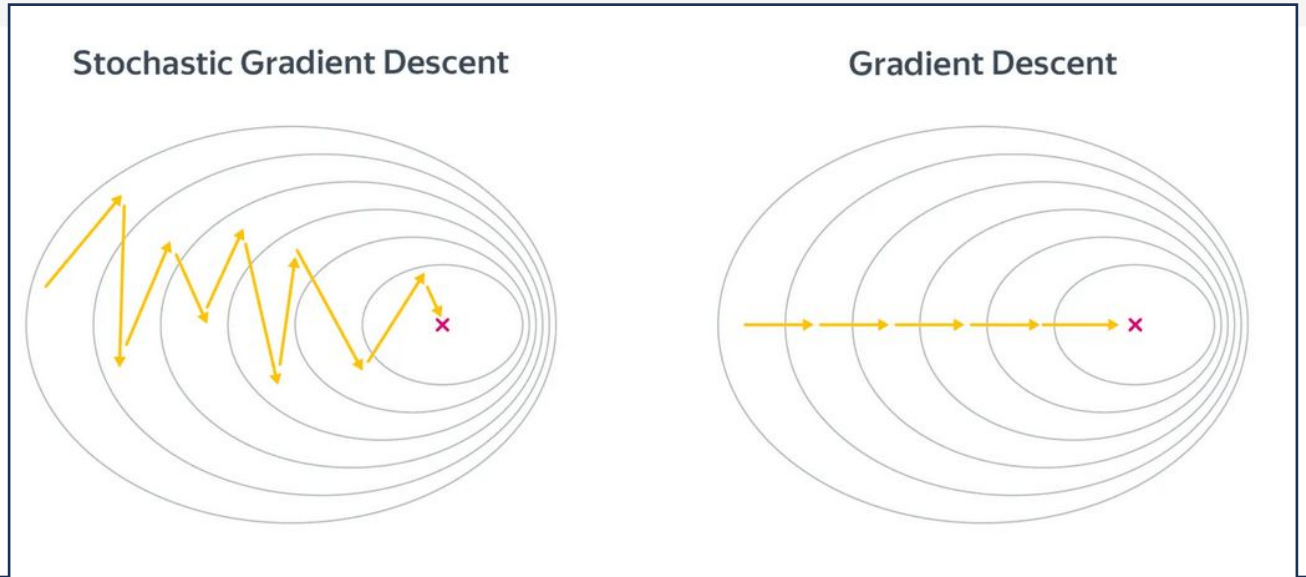
Градиентный спуск (GD)

$$x_{k+1} = x_k - \alpha \nabla f(x_k),$$

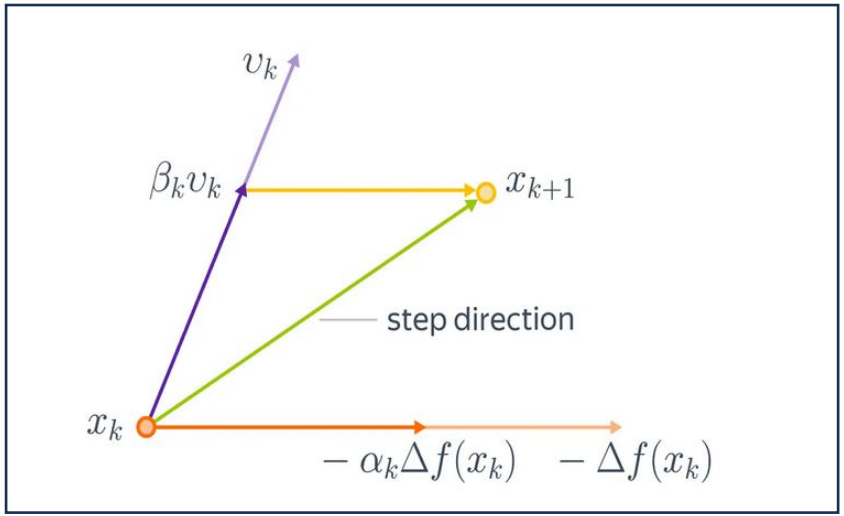
Не эффективный(

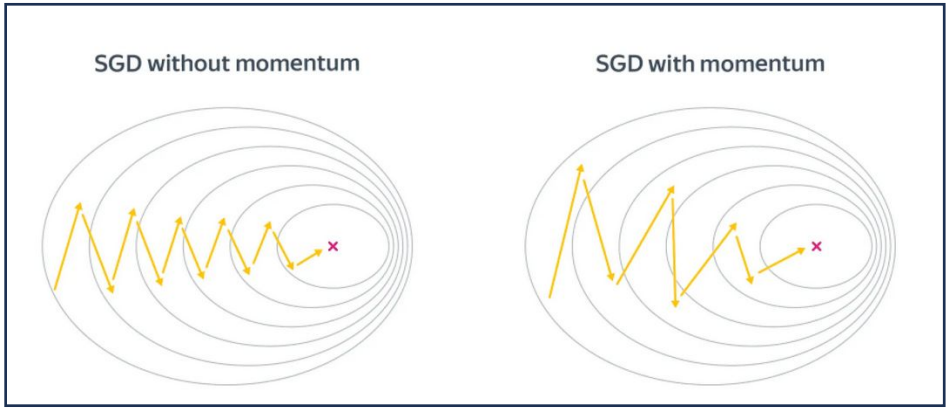
Стохастический градиентный спуск (SGD)

```
x = normal(0, 1)                          # инициализация
repeat E times:                           # цикл по количеству эпох
   for i = 0; i <= N; i += B:
        batch = data[i:i+B]
        h = grad_loss(batch).mean() # вычисляем оценку градиента как сред
        x -= alpha * h
```



Momentum

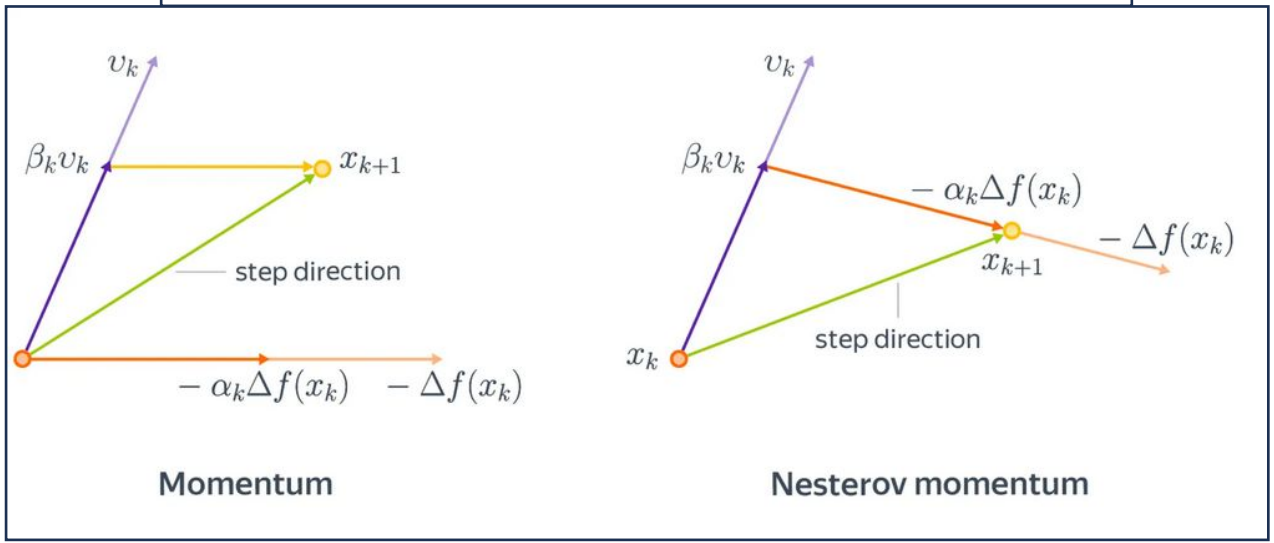




$$x_{k+1} = x_k - \alpha_k \nabla f(x_k) + \beta_k (x_k - x_{k-1}).$$

Nesterov Momentum

$$v_{k+1} = \beta_k v_k - \alpha_k \nabla f(x_k + \beta_k v_k)$$
$$x_{k+1} = x_k + v_{k+1}$$



Adagrad

$$G_{k+1} = G_k + (\nabla f(x_k))^2 \tag{5}$$

$$x_{k+1} = x_k - \frac{\alpha}{\sqrt{G_{k+1} + \varepsilon}} \nabla f(x_k). \tag{6}$$

RMSProp

$$G_{k+1} = \gamma G_k + (1-\gamma)(\nabla f(x_k))^2 \quad (7)$$

$$x_{k+1} = x_k - \frac{\alpha}{\sqrt{G_{k+1} + \varepsilon}} \nabla f(x_k). \quad (8)$$

Соединяем и получаем… Adam

$$v_{k+1} = \beta_1 v_k + (1 - \beta_1)\nabla f(x_k) \quad (9)$$

$$G_{k+1} = \beta_2 G_k + (1 - \beta_2)(\nabla f(x_k))^2 \quad (10)$$

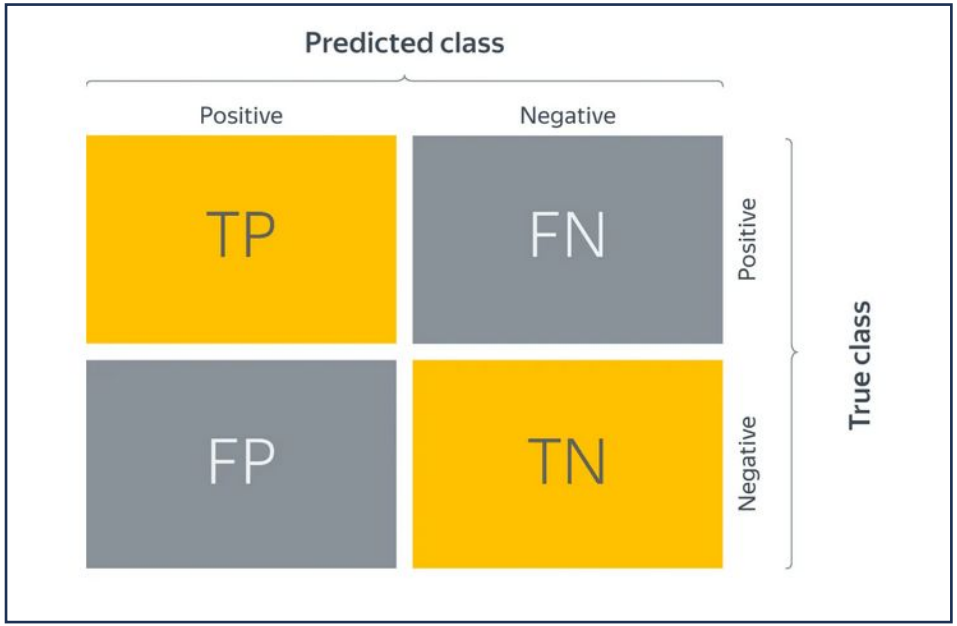
$$x_{k+1} = x_k - \frac{\alpha}{\sqrt{G_{k+1} + \varepsilon}} v_{k+1}. \quad (11)$$

Метрики классификации

1. Accuracy

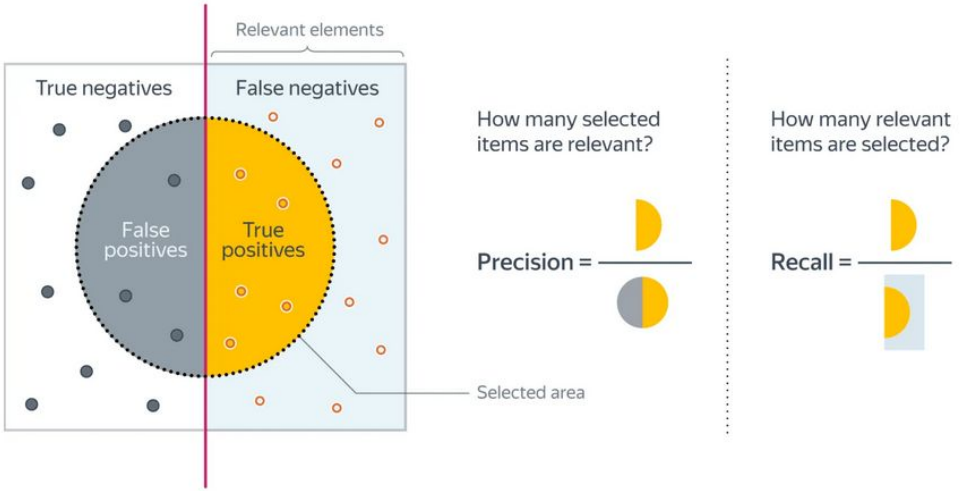
Метрики классификации

1. Accuracy
2. Confusion matrix



Метрики классификации

1. Accuracy
2. Confusion matrix
3. Precision, Recall



Метрики классификации

1. Accuracy
2. Confusion matrix
3. Precision, Recall
4. F1

$$F_1 = 2\frac{Recall \cdot Precision}{Recall + Precision}$$

Метрики классификации

1. Accuracy
2. Confusion matrix
3. Precision, Recall
4. F1
5. ROC-AUC

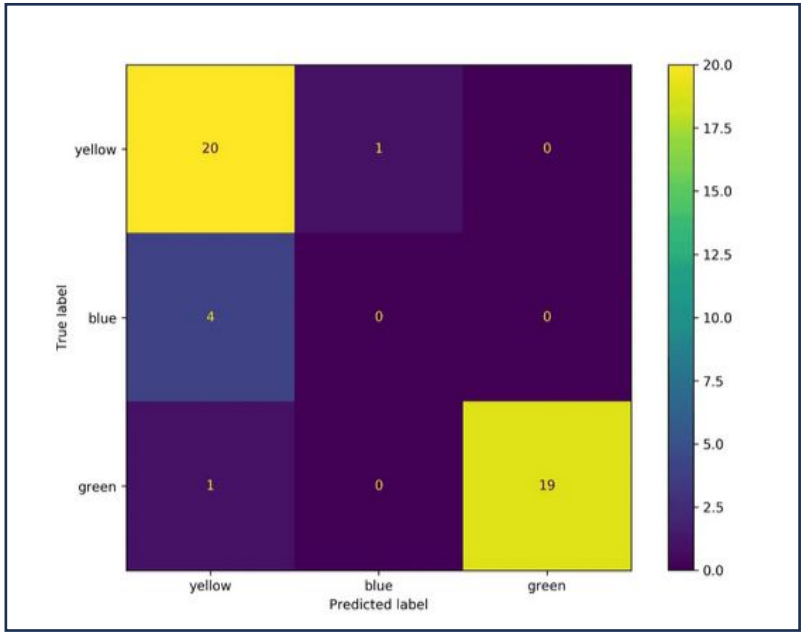
id	оценка	класс
4	0.6	1
1	0.5	0
6	0.3	1
3	0.2	0
5	0.2	1
2	0.1	0
7	0.0	0

Метрики классификации

1. Accuracy
2. Confusion matrix
3. Precision, Recall
4. F1
5. ROC-AUC
6. Микро- и Макроусреднения

1. Усредняем элементы матрицы ошибок (TP, FP, TN, FN) между бинарными классификаторами, например $TP = \frac{1}{K}\sum_{i=1}^{K} TP_i$. Затем по одной усреднённой матрице ошибок считаем Precision, Recall, F-меру. Это называют **микроусреднением**.
2. Считаем Precision, Recall для каждого классификатора отдельно, а потом усредняем. Это называют **макроусреднением**.

Считаем Precision micro и macro



1. С помощью микроусреднения получаем

$$\text{Precision} = \frac{\frac{1}{3}(20+0+19)}{\frac{1}{3}(20+0+19)+\frac{1}{3}(5+1+0)} = 0.87$$

2. С помощью макроусреднения получаем

$$\text{Precision} = \frac{1}{3}\left(\frac{20}{20+5}+\frac{0}{0+1}+\frac{19}{19+0}\right) = 0.6$$