Конспект по теме "Обучение градиентным спуском"

Градиентный спуск для линейной регрессии

Повторим задачу обучения линейной регрессии:

$$w = \arg\min_{w} MSE(Xw, y)$$

Обучим модель градиентным спуском. Рапишем функцию потерь в векторном виде, чтобы найти её градиент. Представим *MSE* как скалярное произведение разности векторов на себя:

$$MSE(y, a) = \sum_{i=1}^{n} (a_i - y_i)^2 = \frac{1}{n}(a - y,\ a - y)$$

где *y* — вектор правильного ответа, *a* — вектор предсказания.

Преобразуем скалярное произведение в матричное:

$$(a - y,\ a - y) = (a - y)^T(a - y)$$

После транспонирования вектора, то есть его преобразования из столбца в строку, возможно умножение на другой вектор.

Объединим формулы *MSE* и линейной регрессии:

$$\text{MSE}(X w, y) = \frac{1}{n} (X w - y)^T (X w - y)$$

Найдём градиент функции по вектору параметров *w*. Градиенты векторных функций вычисляются, как и производные. Например, когда работаем с числами, производная $(xw - y)^2$ по вектору параметров *w* равна $2x(xw - y)$. Когда с векторами — от содержимого первой скобки остаётся только множитель при *w*, то есть X^T:

$$\nabla \text{MSE}(X w, y) = \frac{2}{n} X^T (X w - y)$$

Стохастический градиентный спуск

Время работы алгоритма уменьшится, если градиент вычислять на небольших частях обучающей выборки, которые называются **мини-батчами**, или **батчами**. А чтобы алгоритм всё-таки «увидел» полную обучающую выборку, её батчи на каждой итерации должны меняться случайным образом. За такое обучение отвечает **стохастический градиентный спуск по мини-батчам**, или **стохастический градиентный спуск (***stochastic gradient descent, SGD*).

Чтобы получить батчи, нужно перемешать все данные выборки и разбить её на части. В одном батче должно быть в среднем 100-200 объектов — это и есть **размер батча** (*batch size*). Когда алгоритм *SGD* прошёл один раз по всем батчам, значит, завершилась одна **эпоха** (*epoch*). Сколько всего будет эпох, зависит от размера обучающей выборки: одна-две эпохи — если выборка большая, несколько десятков — если маленькая. Количество батчей равно числу итераций для завершения одной эпохи.

Алгоритм стохастического градиентного спуска работает так:

1. На вход поступают гиперпараметры: размер батча, количество эпох и величина шага.
2. Определяются начальные значения весов модели.
3. Для каждой эпохи обучающая выборка разбивается на батчи.
4. Для каждого батча:
    - Вычисляется градиент функции потерь;

- Обновляются веса модели (к текущим значениям весов прибавляется антиградиент, умноженный на величину шага).

5. Алгоритм возвращает последние веса модели.

Вычислительную сложность *SGD* : $T(n, b, p) \sim np$, где:

- n — количество объектов во всей обучающей выборке;
- b — размер батча;
- p — количество признаков.

SGD на Python

Научимся передавать модели гиперпараметры. Для этого объявим её класс и создадим метод **«инициализатор класса»**. Записывается `__init__`:

```
# англ. SGD для модели линейной регрессии
class SGDLinearRegression:
    def __init__(self):
        ...
```

Добавим в инициализатор класса один гиперпараметр — величину шага `step_size`:

```
class SGDLinearRegression:
    def __init__(self, step_size):
        ...
```

Теперь размер шага можно передать модели при создании класса:

```
# размер шага выбрали произвольно
model = SGDLinearRegression(0.01)
```

Сохраним размер шага в атрибуте:

```
class SGDLinearRegression:
    def __init__(self, step_size):
        self.step_size = step_size
```

Тогда полностью реализация алгоритма стохастического градиентного спуска будет выглядеть так:

```
class SGDLinearRegression:
    def __init__(self, step_size, epochs, batch_size):
        self.step_size = step_size
```

```
        self.epochs = epochs
        self.batch_size = batch_size

    def fit(self, train_features, train_target):
        X = np.concatenate((np.ones((train_features.shape[0], 1)), train_features), axis=1)
        y = train_target
        w = np.zeros(X.shape[1])

        for _ in range(self.epochs):
            batches_count = X.shape[0] // self.batch_size
            for i in range(batches_count):
                begin = i * self.batch_size
                end = (i + 1) * self.batch_size
                X_batch = X[begin:end, :]
                y_batch = y[begin:end]

                gradient = 2 * X_batch.T.dot(X_batch.dot(w) - y_batch) / X_batch.shape[0]

                w -= self.step_size * gradient

        self.w = w[1:]
        self.w0 = w[0]

    def predict(self, test_features):
        return test_features.dot(self.w) + self.w0
```

Регуляризация линейной регрессии

Изменим функцию потерь, чтобы избавиться от переобучения. Уменьшить переобучение поможет **регуляризация** (*regularization*). Она «штрафует» модель, если значения параметров усложняют работу алгоритма. У моделей линейной регрессии регуляризация выражается в ограничении весов. Чем меньше значения весов, тем проще обучается алгоритм. Чтобы понять, насколько веса велики, вычисляют расстояние от вектора весов до вектора, состоящего из нулей. Например, евклидово расстояние $d_2(w, 0)$ равно скалярному произведению весов на себя: (w, w).

Чтобы ограничить значения весов, добавим скалярное произведение весов на себя в формулу функции потерь:

$$L(w) = MSE(Xw, y) + (w, w)$$

Производная (w, w) равна $2w$. Градиент функции потерь вычисляется так:

$$\nabla L(w) = \frac{2}{n} X^T (Xw - y) + 2w$$

Чтобы контролировать силу регуляризации, в формулу функции потерь добавляют **вес регуляризации** (*regularization weight*). Обозначается λ:

$$L(w) = MSE(Xw, y) + \lambda(w, w)$$

Вес регуляризации добавляется также в формулу вычисления градиента:

$$\nabla L(w) = \frac{2}{n} X^T (Xw - y) + 2\lambda w$$

Если для регуляризации весов применяется евклидово расстояние, то такая линейная регрессия называется **гребневой регрессией** (*ridge regression*).

С учётом регуляризации, стохасический градиентный спуск будет выглядеть так:

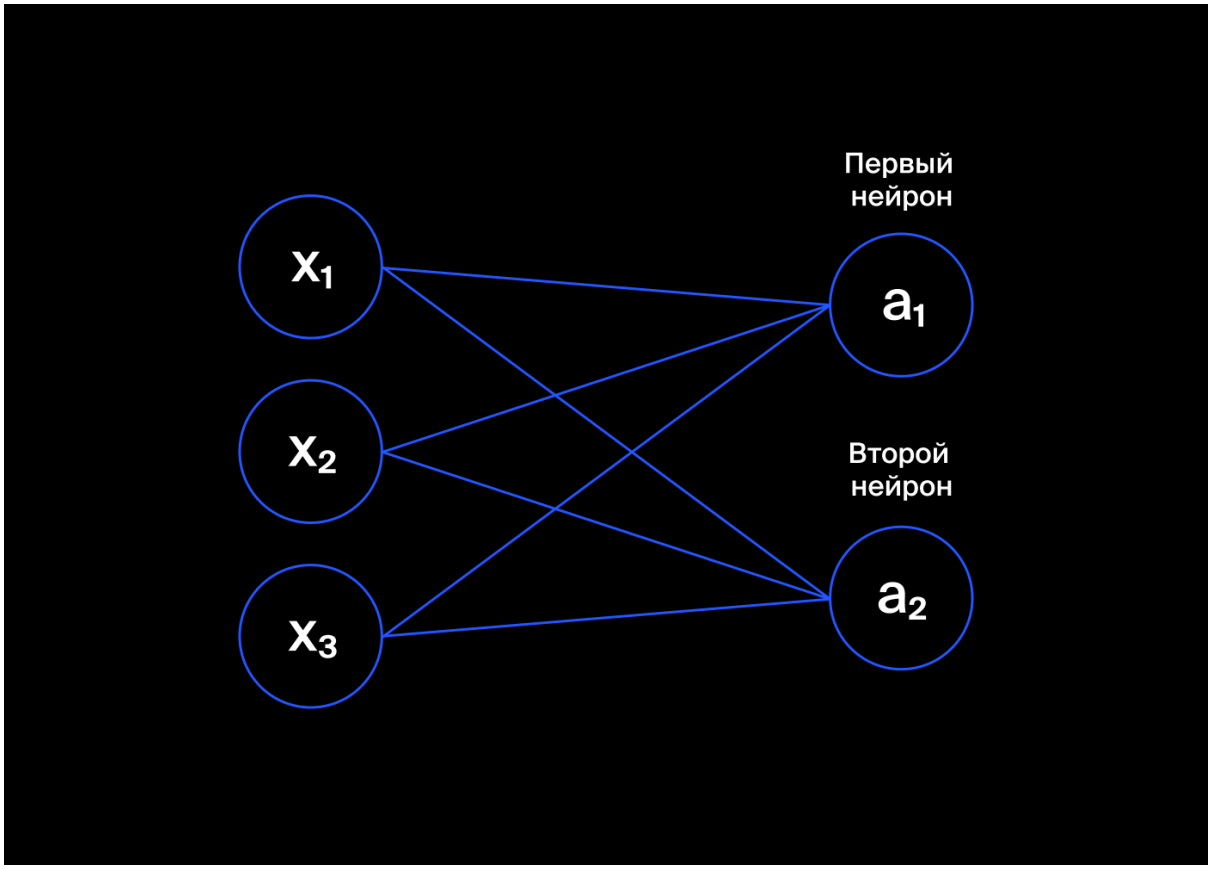
```
class SGDLinearRegression:
    def __init__(self, step_size, epochs, batch_size, reg_weight):
        self.step_size = step_size
        self.epochs = epochs
        self.batch_size = batch_size
        self.reg_weight = reg_weight

    def fit(self, train_features, train_target):
        X = np.concatenate((np.ones((train_features.shape[0], 1)), train_features), axis=1)
        y = train_target
        w = np.zeros(X.shape[1])

        for _ in range(self.epochs):
            batches_count = X.shape[0] // self.batch_size
            for i in range(batches_count):
                begin = i * self.batch_size
                end = (i + 1) * self.batch_size
                X_batch = X[begin:end, :]
                y_batch = y[begin:end]

                gradient = 2 * X_batch.T.dot(X_batch.dot(w) - y_batch) / X_batch.shape[0]
                reg = 2 * w.copy()
                reg[0] = 0
                gradient += self.reg_weight * reg
```

```
                w -= self.step_size * gradient

        self.w = w[1:]
        self.w0 = w[0]

    def predict(self, test_features):
        return test_features.dot(self.w) + self.w0
```

Основы нейронных сетей

Нейронная сеть (*neural network*) — это модель, которая состоит из множества простых моделей, например, линейных регрессий. Название пришло из биологии: искусственная нейронная сеть работает по принципу сетей нервных клеток, то есть из нейронов строит сложные зависимости между входными и выходными данными.

Рассмотрим пример нейронной сети с тремя входами x_1, x_2, x_3 и двумя выходами a_1, a_2:



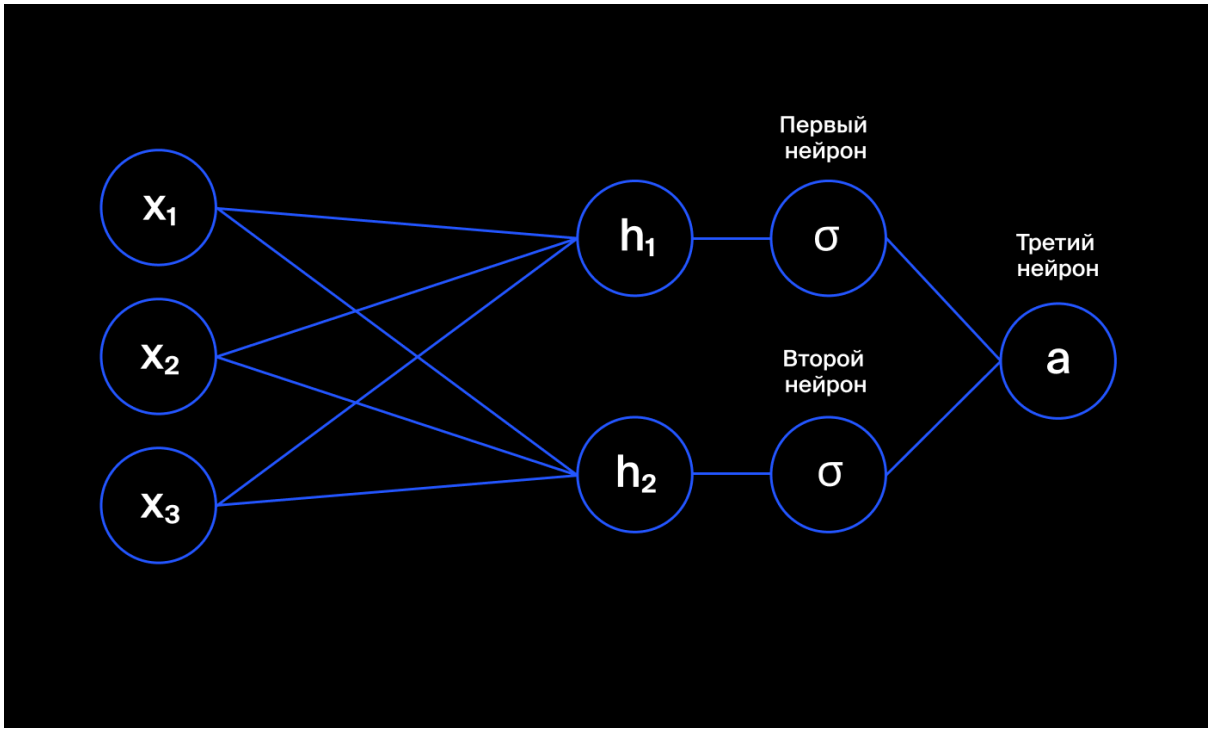
Значение на каждом выходе, или нейрон, вычисляется так же, как и предсказание линейной регрессии:

$$a_1 = xw_1$$

$$a_2 = xw_2$$

У каждого значения выхода — свои веса (w_1 и w_2).

Рассмотрим другую нейронную сеть. В ней три входа x_1, x_2, x_3, две промежуточных переменных h_1 и h_2 и один выход a.



Значения h_1 и h_2 передаются в логистическую функцию $\sigma(x)$:

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

где e — число Эйлера; приблизительно равно 2.718281828.

Логистическая функция в нейронной сети — это **функция активации** (*activation function*). Она добавляется в нейрон после умножения значений входов на веса, когда выходы нейрона становятся входами для других нейронов. Такими нейронами можно описать более сложные зависимости.

Промежуточная переменная (h_1, h_2) — это произведение значения входа на вес:

$$h_1 = xw_1$$

$$h_2 = xw_2$$

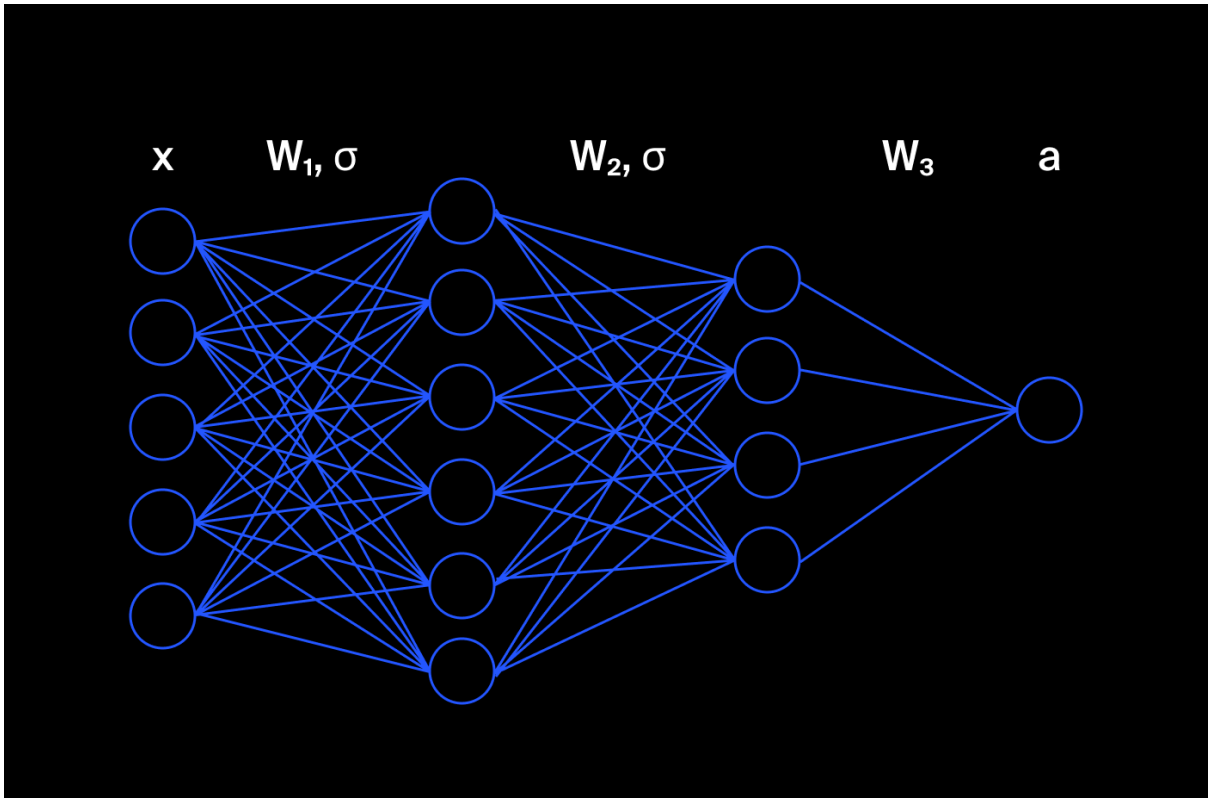
Для удобства промежуточные переменные h_1 и h_2 обозначим вектором h.
Предсказание нейронной сети вычисляется по формуле:

$$a = \sigma(h)w_3$$

Запишем эту формулу через векторную функцию. Веса w_1 и w_2 обозначим столбцами матрицы W. Получим такую векторную формулу:

$$a = \sigma(xW)w_3$$

Если в матрицы записать веса нескольких нейронов, можно получить ещё более сложную сеть, например, такую:



где:

- x — входной вектор размерностью p (количество признаков);
- W_1 — матрица размерностью $p \times m$;
- W_2 — матрица размерностью $m \times k$;
- W_3 — матрица размерностью $k \times 1$;
- a — предсказание модели (одно число).

Когда такая нейронная сеть рассчитывает предсказание, она последовательно выполняет все операции:

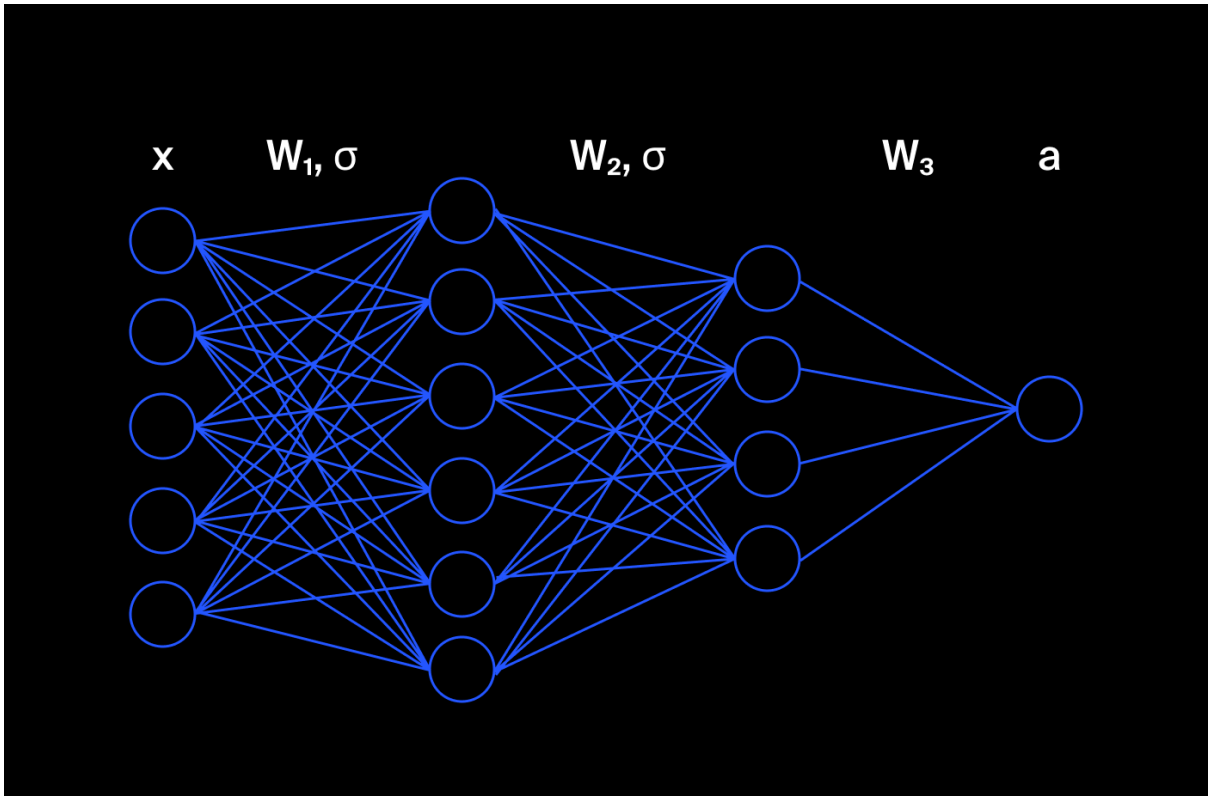
$$a = \sigma(\sigma(xW_1)W_2)W_3$$

Обучение нейронных сетей

Чтобы обучить нейронную сеть, нужно сформулировать задачу обучения. Любую нейронную сеть можно записать функцией от её входного вектора и параметров. Обозначим:

- *X* — признаки обучающей выборки;
- *P* — набор всех параметров нейронной сети;
- *N(X, P)* — функция нейронной сети.

Возьмём такую нейронную сеть:



Параметры нейронной сети — это веса в нейронах:

$$P = W_1, W_2, W_3$$

Функция нейронной сети такая:

$$N(X, P) = \sigma(\sigma(XW_1)W_2)W_3$$

Также обозначим:

- *y* — ответы обучающей выборки;
- *L(a, y)* — функция потерь (например, *MSE*).

Тогда задача обучения нейронной сети формулируется так:

$$\min_{P} L(N(X, P), y)$$

Минимум этой функции также находится алгоритмом *SGD*.

Алгоритм обучения нейронной сети такой же, как и алгоритм *SGD* для линейной регрессии. Только вместо градиента для линейной регрессии вычисляется градиент для нейронной сети:

$$\nabla L(N(X, P), y)$$