ГЛАВА X1.
Нейронные сети

Ошибка лежит на поверхности, и ее замечаешь сразу, а истина скрыта в глубине, и не всякий может отыскать ее.

И.В. Гёте

Нейрон – простейшая нейросеть

Рассмотрим задачу обучения с размеченными данными (классификации или регрессии). Дана обучающая выборка $\{(x_i, y_i)\}_{i=1}^m$, состоящая из пар «объект» $x_i \in X$, «метка» $y_i \in Y$, необходимо построить алгоритм $a(x;w)$, который реализует отображение из множества объектов $X = \mathbb{R}^n$ во множество меток Y:

$$a(x;w): X \to Y,$$

w – параметры алгоритма (отображения). Такой алгоритм должен для нового объекта $x \in X$ достаточно точно предсказывать его метку $y = y(x)$. Ошибка предсказания обычно формализуется с помощью функции ошибки $L(y, a(x;w))$, а алгоритм строится с помощью процедуры обучения, в котором минимизируется эмпирический риск (суммарная ошибка на обучающей выборке). В минимизируемый функционал часто добавляют регуляризатор $R(w)$, чтобы избежать переобучения:

$$\frac{1}{m}\sum_{t=1}^{m} L(y_t, a(x_t;w)) + R(w) \to \min_w. \qquad (0)$$

Простейшей моделью алгоритмов в классическом машинном обучении является линейная, в ней ответ алгоритма является деформацией линейной комбинации значений признаков

$$a(x) = \varphi(b + w_1 x_{[1]} + \ldots + w_n x_{[n]}).$$

($x_{[i]} \in \mathbb{R}$ – значение i-го признака объекта x[1]). Сейчас такой алгоритм будем называть **нейроном (моделью нейрона)**, функцию $\varphi: \mathbb{R} \to \mathbb{R}$ – **функцией активации**, $w_0 = b, w_1, \ldots, w_n$ – **весами связей (параметрами)**, b – **смещением (bias)**. Геометрически нейрон можно представить как **граф вычислений**[2], где входные значения $x_{[1]}, \ldots, x_{[n]}$ умножаются на соответствующие веса $w_1, \ldots, w_n$, суммируются с добавлением смещения $w_0 = b$, а затем применяется функция активации φ. Пример такого графа приведён на рисунке.

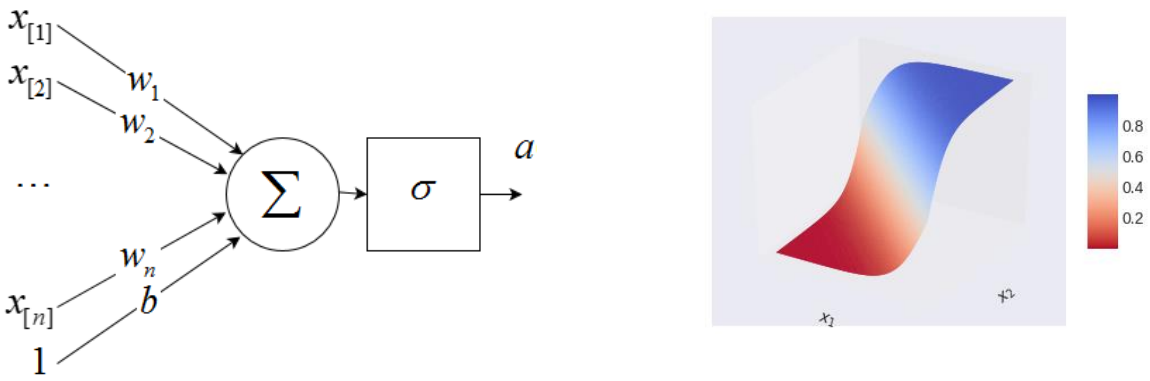


Рис. Модель нейрона и график его значений.

При различных функциях активации мы получаем разные классические линейные модели:

1. **Тождественная функция** (линейная[3] / linear activation function) – линейная регрессия,

$$f(z) = z, \quad \frac{\partial f(z)}{\partial z} = 1.$$

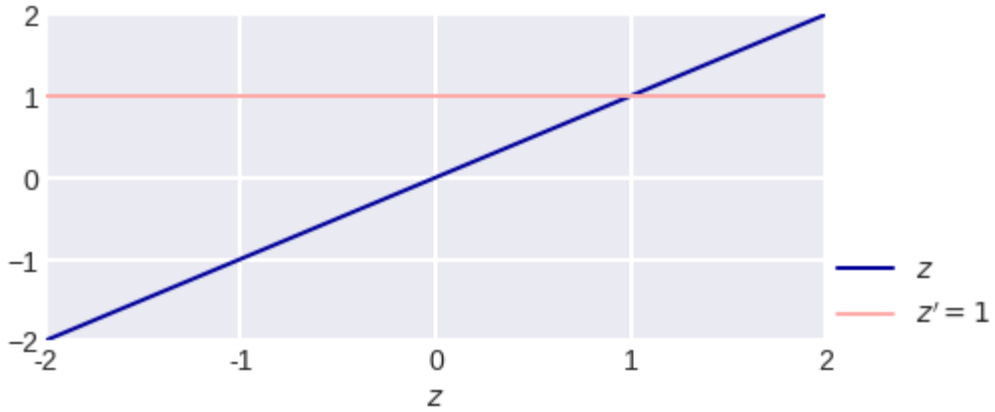


Рис. График тождественной активации и её производной.

[1] Чтобы не путать с *i*-м объектом обучающей выборки.

[2] У нас часто сейчас будет возникать этот термин. Пока понимаем его интуитивно.

[3] Её часто называют линейной, хотя она реализует тождественную функцию.

2. **Пороговая функция** (threshold function) – линейный классификатор,

$$\text{th}(z) = I[z > 0], \quad \frac{\partial \text{th}(z)}{\partial z} = 0 .$$

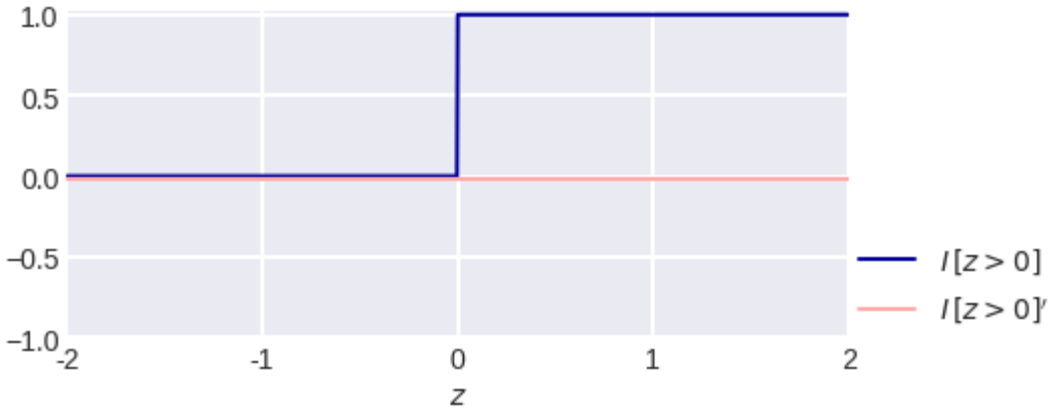


Рис. График пороговой активации и её производной.

3. **Сигмоида** (sigmoid activation function) – логистическая регрессия

$$\sigma(z) = \frac{1}{1+e^{-z}} \in (0,1), \quad \frac{\partial \sigma(z)}{\partial z} = \sigma(z)(1-\sigma(z)) > 0$$

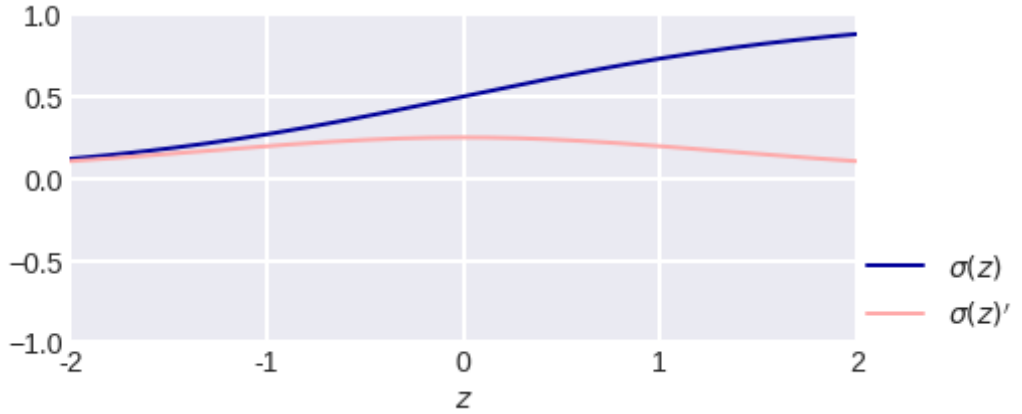


Рис. График сигмоиды и её производной.

Мы не просто так посчитали здесь производные функций активаций и привели их графики – они нам понадобится в дальнейшем. Обратим внимание на «приятное» свойство сигмоиды: производная выражается через саму функцию. Таким же свойством обладает и гиперболический тангенс (hyperbolic tangent):

$$\tanh(z) = \frac{2}{1+e^{-2z}} - 1 = \frac{e^{+z} - e^{-z}}{e^{+z} + e^{-z}} = \frac{e^{+2z} - 1}{e^{+2z} + 1}, \quad \frac{\partial \tanh(z)}{\partial z} = 1 - \tanh^2(z)$$

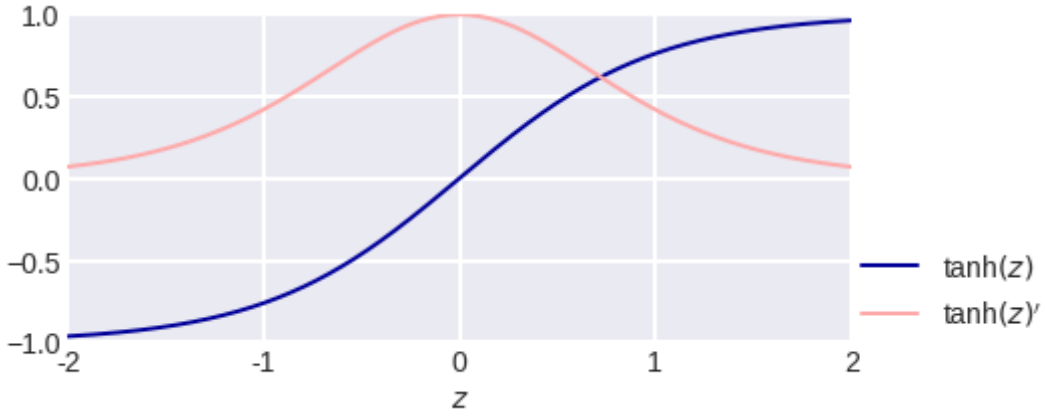


Рис. График гиперболического тангенса и его производной.

Также вспомним, что сигмоида использовалась в логистической регрессии для решения задач бинарной классификации. В задачах с k непересекающимися классами сигмоида имеет естественное обобщение[1]: функцию **softmax** («мягкий максимум»),

$$\text{softmax}(z_1,\ldots,z_k)=\frac{1}{\sum_{t=1}^{k}\exp(z_t)}(\exp(z_1),\ldots,\exp(z_k))^{\mathrm{T}},$$

которая преобразует вектор вещественных чисел $(z_1,\ldots,z_k)$ в вектор неотрицательных элементов[2], которые в сумме дают 1. Это позволяет интерпретировать результат как распределение вероятностей на множестве из k классов. При этом максимальный элемент после мягкого максимума остаётся максимальным:

$$[0.5,\ 0.5,\ 0.1,\ 0.7]\to[0.257,\ 0.257,\ 0.172,\ 0.314],$$

$$[-1.0,\ 0,\ 1.0,\ 0,\ -1.0]\to[0.07,\ 0.18,\ 0.5,\ 0.18,\ 0.07],$$

$$[1.0,\ 1.0,\ 1.0,\ 2.0,\ 1.0]\to[0.15,\ 0.15,\ 0.15,\ 0.4,\ 0.15]$$

(максимальный элемент выделен красным), поэтому для определения класса можно не считать softmax, а найти максимальный элемент.

Покажем, как реализовать модель нейрона в PyTorch[3]. Базовый класс для всех моделей в PyTorch – nn.Module, поэтому проводим наследование от него.

[1] Если предположить, что оценки классов нормально распределены с одинаковыми дисперсиями и классы равновероятны.

[2] Даже положительных.

[3] Код предложен с помощью модели DeepSeek.

Линейный модуль[1] реализуется с помощью модуля nn.Linear, а сигмоида – с помощью модуля nn.Sigmoid. Обратим внимание, что на вход сети подали сразу два объекта и сразу получили ответы на каждом объекте.

```
import torch
import torch.nn as nn

# Определяем класс нейрона
class SigmoidNeuron(nn.Module):
    def __init__(self, input_dim):
        super(SigmoidNeuron, self).__init__()
        # Линейный слой: y = wx + b
        self.linear = nn.Linear(input_dim, 1)
        # Сигмоидная функция активации
        self.activation = nn.Sigmoid()

    def forward(self, x):
        # Применяем линейное преобразование
        z = self.linear(x)
        # Применяем сигмоидную активацию
        a = self.activation(z)
        return a

# Пример использования
input_dim = 3  # Размерность входного вектора
model = SigmoidNeuron(input_dim)

# Создаем случайный входной вектор (батч из 2 примеров)
x = torch.tensor([[0.5, 1.0, -0.5], [1.0, -1.0, 0.0]], dtype=torch.float32)

# Прямой проход через нейрон
output = model(x)
print("Входные векторы:\n", x)
```

```
Входные векторы:
tensor([[ 0.5000,  1.0000, -0.5000],
        [ 1.0000, -1.0000,  0.0000]])
```

```
print("Выходы нейрона:\n", output)
```

```
Выходы нейрона:
tensor([[ 0.5000,  1.0000, -0.5000],
        [ 1.0000, -1.0000,  0.0000]])
```

Значения параметров нейрона можно вывести следующим способом:

```
model.linear.weight, model.linear.bias
```

```
(Parameter containing:
 tensor([[-0.1076,  0.0099,  0.2709]], requires_grad=True),
 Parameter containing:
 tensor([-0.2058], requires_grad=True))
```

[1] В PyTorch реализация функции, которая производит вычисления называется модулем. Она наследуется от базового класса nn.Module. Заметим, что нейросеть состоит из модулей и сама является модулем.

В PyTorch сигмоидную активацию можно реализовать двумя основными способами: с использованием модуля nn.Sigmoid и функции torch.sigmoid. В первом способе используется готовый модуль:

```
import torch
import torch.nn as nn

# Создаём модуль сигмоидной активации
sigmoid_layer = nn.Sigmoid()

# Пример входных данных
x = torch.tensor([1.0, 0.0, -1.0])

# Применяем сигмоидную активацию
output = sigmoid_layer(x)
```

Во втором – функция torch.sigmoid (эта функция может быть использована как в составе нейронных сетей, так и вне их).

```
import torch

# Пример входных данных
x = torch.tensor([1.0, 0.0, -1.0])

# Применяем сигмоидную функцию
output = torch.sigmoid(x)
```

Результат будет одинаковый. Модуль удобнее использовать для создания нейронных сетей, например при конкатенировании серии модулей с помощью nn.Sequential. Отдельную функцию – для разового применения. Можно также самостоятельно реализовать сигмоиду в виду Python-функции:

```
def custom_sigmoid(x):
    return 1 / (1 + torch.exp(-x))
```

От нейрона к суперпозициям

Линейный классификатор, а следовательно, и нейрон, способен решать задачи, соответствующие логическим операциям И, ИЛИ и НЕ. Эти задачи можно интерпретировать как задачи классификации вершин единичного квадрата.

Например, логическое И и ИЛИ соответствуют определённым способам разделения этих вершин на два класса, см. рис.

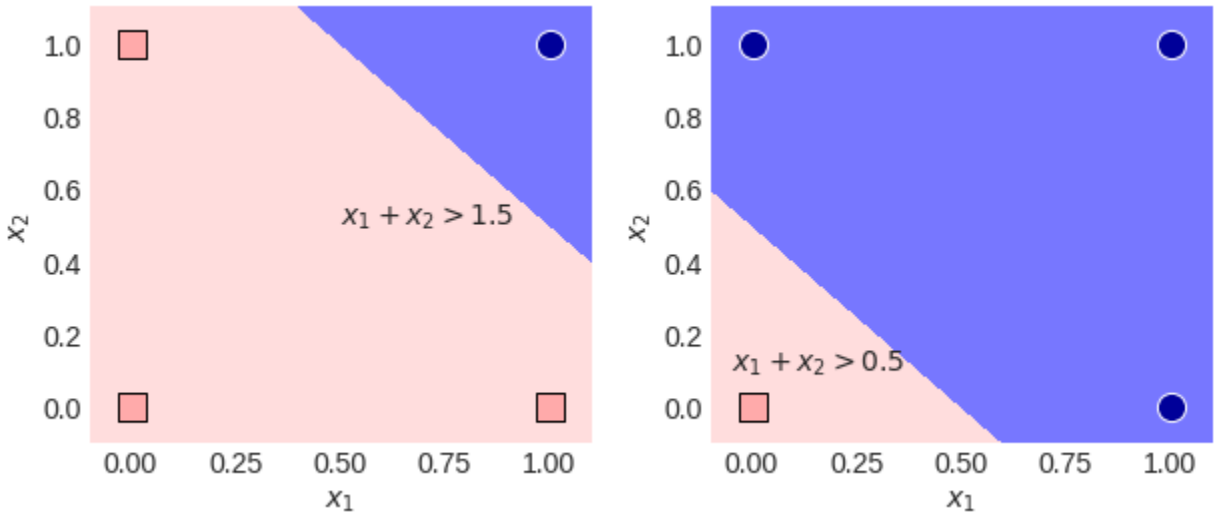


Рис. Задача логического И (слева) и ИЛИ (справа).

Однако не все классификации вершин могут быть реализованы с помощью одного нейрона. В частности, задачи исключающего ИЛИ (XOR) и эквивалентности не могут быть решены одним нейроном, как показано на рисунке[1].

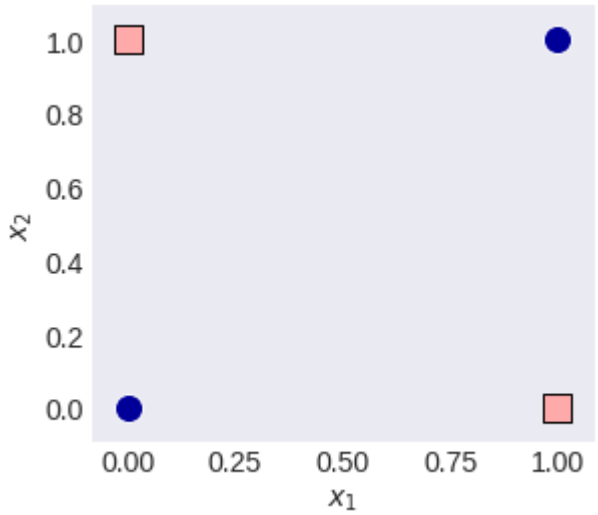


Рис. Задача исключающего ИЛИ и эквивалентности.

Для решения таких задач требуется суперпозиция (композиция) нескольких нейронов. Например, задача эквивалентности / XOR может быть решена с помощью следующей суперпозиции:

[1] Т.к. их выпуклые оболочки разных классов пересекаются. В задачах XOR и эквивалентность противоположные концы диагоналей квадрата имеют одну метку, различаются задачи тем, какой диагонали соответствует метка 0, а какой – 1.

$$\text{th}(\text{th}(x_{[1]} + x_{[2]} - 1.5) + \text{th}(-x_{[1]} - x_{[2]} + 0.5) - 0.5),$$

в чём легко убедиться подстановкой значений:

$$\text{th}(\text{th}(0+0-1.5) + \text{th}(-0-0+0.5) - 0.5) = \text{th}(0+1-0.5) = 1,$$

$$\text{th}(\text{th}(0+1-1.5) + \text{th}(-0-1+0.5) - 0.5) = \text{th}(0+0-0.5) = 0,$$

$$\text{th}(\text{th}(1+0-1.5) + \text{th}(-1-0+0.5) - 0.5) = \text{th}(0-0-0.5) = 0,$$

$$\text{th}(\text{th}(1+1-1.5) + \text{th}(-1-1+0.5) - 0.5) = \text{th}(1+0-0.5) = 1.$$

Геометрический смысл суперпозиции упрощённо показан на рис. Один нейрон отделяет одну точку класса 1 от остальных, второй – другую, а третий «объединяет» результаты первых двух, формируя решающую поверхность. Также можно посмотреть, как классифицирует все точки пространства построенная суперпозиция, см. рис: полоса, ограниченная двумя параллельными прямыми, относится к классу 1, а остальные точки – к классу 0.

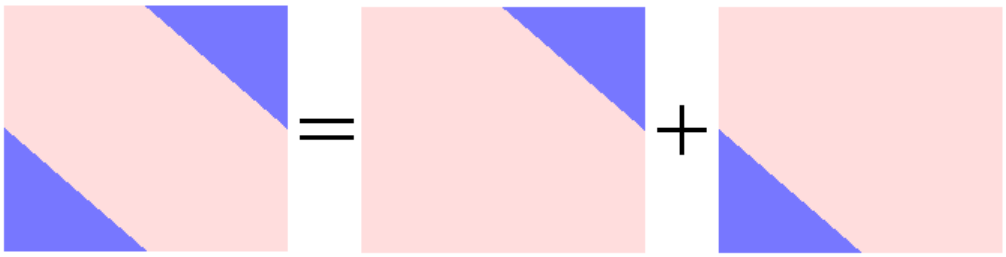


Рис. Геометрический смысл суперпозиции.

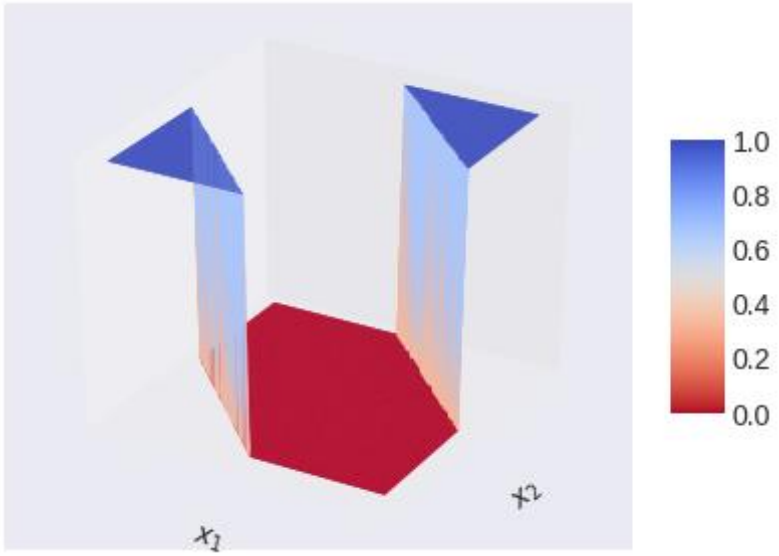


Рис. Решающая поверхность суперпозиции нейронов.

Пороговая функция может быть приближена сигмоидой:

$$\sigma_c(z)=\frac{1}{1+e^{-cz}}\xrightarrow[c\to+\infty]{}\text{th}(z),$$

поэтому можно рассмотреть такую суперпозицию:

$$\sigma_c(\sigma_c(x_{[1]}+x_{[2]}-1.5)+\sigma_c(-x_{[1]}-x_{[2]}+0.5)-0.5),$$

которая позволяет в задаче XOR получать решающие поверхности «разной сглаженности», см. рис.

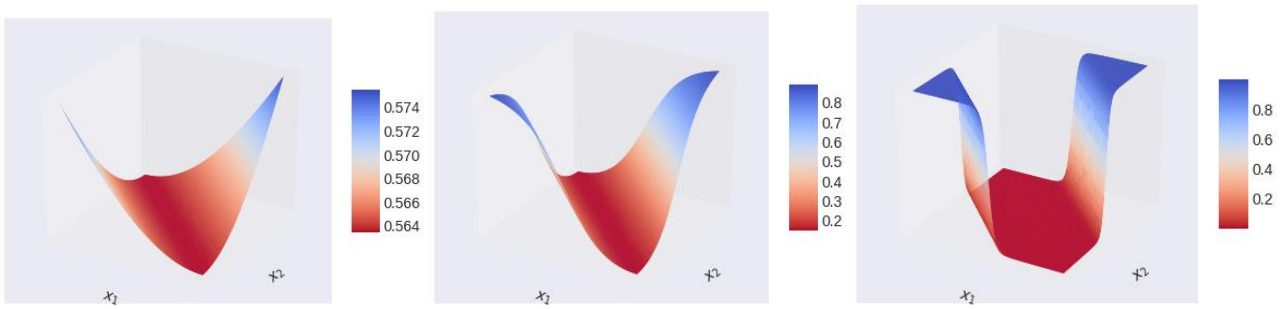
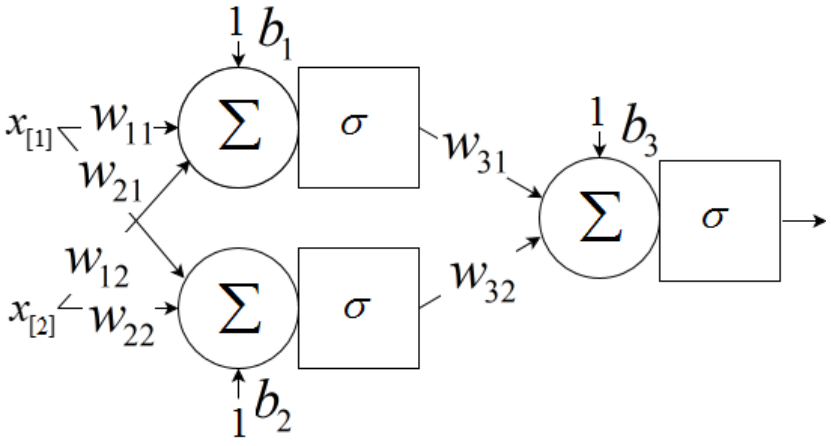


Рис. Решающие поверхности при использовании параметрической сигмоиды.

При этом сигмоида всюду дифференцируема, что пригодится нам в дальнейшем. Построенная суперпозиция

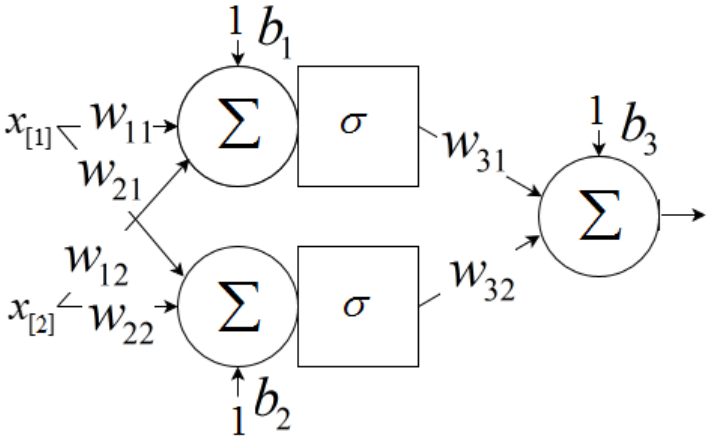
$$a=\sigma(b_3+w_{31}\sigma(b_1+w_{11}x_{[1]}+w_{12}x_{[2]})+w_{32}\sigma(b_2+w_{21}x_{[1]}+w_{22}x_{[2]})) \qquad (1)$$

имеет следующий граф вычислений



Заметим, что подобный граф можно получить и **в задаче регрессии**, но тогда **в последнем нейроне логично использовать тождественную функцию активации**:

$$a=b_3+w_{31}\sigma(b_1+w_{11}x_{[1]}+w_{12}x_{[2]})+w_{32}\sigma(b_2+w_{21}x_{[1]}+w_{22}x_{[2]}) \qquad (2)$$



Это простейшие примеры двуслойных нейронных сетей: два первых нейрона образуют первый слой – получают на вход признаковые описания объекта, а последний нейрон – второй слой – получает результаты работы первого слоя. Также отметим, что формулу (1) можно переписать в матричном виде:

$$\sigma\left(\begin{bmatrix} w_{31} & w_{32} & b_3 \end{bmatrix}\begin{bmatrix} \sigma\left(\begin{bmatrix} w_{11} & w_{12} & b_1 \\ w_{21} & w_{22} & b_2 \end{bmatrix}\begin{pmatrix} x_{[1]} \\ x_{[2]} \\ 1 \end{pmatrix}\right) \\ 1 \end{bmatrix}\right),$$

здесь используются только операции:

- конкатенации (добавление единицы к вектору),
- матричного умножения,
- покомпонентного применения функции активации:

выражение $\sigma(z)$, где $z = (z_1, \ldots, z_k)^{\mathrm{T}}$ интерпретируем как

$$(\sigma(z_1), \ldots, \sigma(z_k))^{\mathrm{T}}.$$

Суперпозиции нейронов, по сути, и являются нейронными сетями. Теорема об универсальной аппроксимации [Hornik, 1991] утверждает, что простой нейронной сети (2) достаточно для решения большинства задач регрессии.

Теорема. Любую непрерывную функцию можно с любой точностью приблизить нейросетью глубины 2 с сигмоидной функцией активации на скрытом слое и линейной функции на выходном слое

Есть также любопытный результат, в котором получен критерий на функцию активации: теорема об универсальной аппроксимации Пинкуса[1].

Теорема. Нейросеть глубины два с фиксированной функцией активации в первом слое и линейной функцией активации во втором может равномерно аппроксимировать (м.б. при увеличении числа нейронов в первом слое) любую непрерывную функцию на компактном множестве тогда и только тогда, когда функция активации неполиномиальная.

Можно продемонстрировать способность аппроксимировать сложные функции при увеличении числа нейронов в первом слое с помощью следующей иллюстрацией из [https://udlbook.github.io/udlbook/] Рассмотрим двухслойную сеть с функцией активацией $\mathrm{ReLU} = \max(z, 0)$:

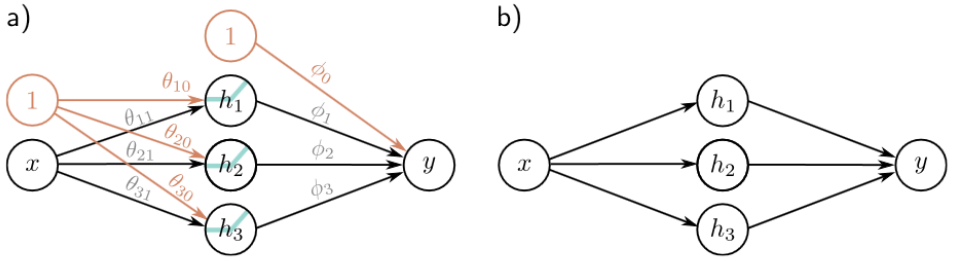


Figure 3.4 Depicting neural networks. a) The input x is on the left, the hidden units h_1, h_2, and h_3 in the center, and the output y on the right. Computation flows from left to right. The input is used to compute the hidden units, which are combined to create the output. Each of the ten arrows represents a parameter (intercepts in orange and slopes in black). Each parameter multiplies its source and adds the result to its target. For example, we multiply the parameter ϕ_1 by source h_1 and add it to y. We introduce additional nodes containing ones (orange circles) to incorporate the offsets into this scheme, so we multiply ϕ_0 by one (with no effect) and add it to y. ReLU functions are applied at the hidden units. b) More typically, the intercepts, ReLU functions, and parameter names are omitted; this simpler depiction represents the same network.

При увеличении числа нейронов в первом слое она реализует кусочно-линейную функцию с большим числом изломов. На рис. показано, как улучшается аппроксимация при увеличении числа нейронов / изломов.

[1] http://www2.math.technion.ac.il/~pinkus/papers/neural.pdf

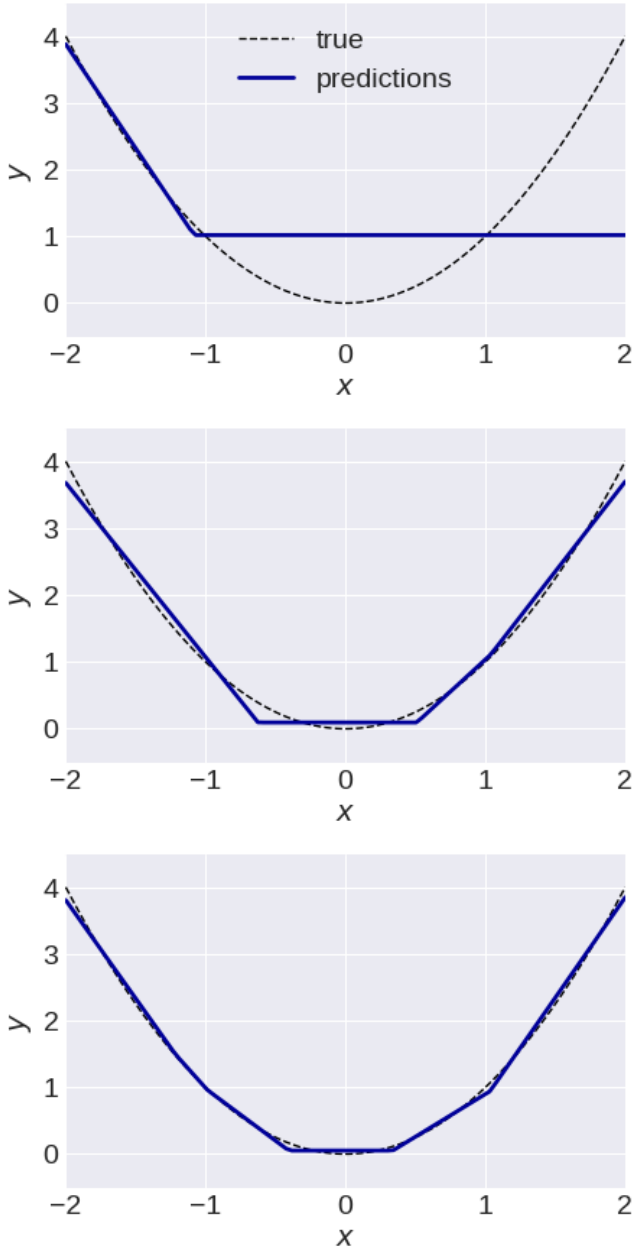


Рис. Аппроксимация квадратичной зависимости с помощью нейросети с разным числом нейронов в первом слое: 2, 4, 8.

Разберём подробно, как работает нейросеть. Первый линейный модуль реализует функции вида $w_i x + b_i$, т.е. прямые, они показаны на рис. (слева). Второй модуль – функция активация реализует функции $\max(w_i x + b_i, 0)$, которые показаны на рис. (справа). Их линейная комбинация даёт результат, показанный на рис.

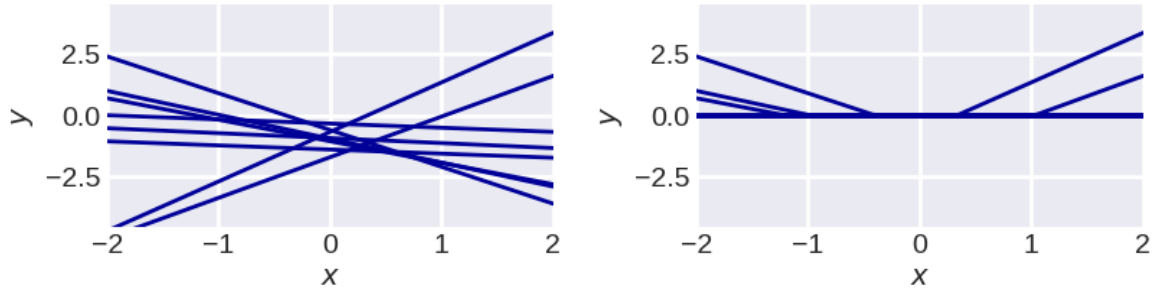


Рис. Выход первого и второго модулей.

Иллюстрация эксперимента, в котором квадратичную зависимость пытаемся приблизить ломаной с помощью обучения нейросети.

```
import torch
import torch.nn as nn
import torch.optim as optim
import matplotlib.pyplot as plt
import numpy as np

# Генерация данных
x = torch.linspace(-2, 2, 100).view(-1, 1)  # Входные данные (от -2 до 2)
y = x ** 2  # Целевые значения (y = x^2)

# Определение модели
class QuadraticNet(nn.Module):
    def __init__(self, k=2):
        super(QuadraticNet, self).__init__()
        self.fc1 = nn.Linear(1, k)  # Первый полносвязный слой
        self.fc2 = nn.Linear(k, 1)   # Выходной слой
        self.relu = nn.ReLU()          # Функция активации ReLU

    def forward(self, x):
        x = self.relu(self.fc1(x))  # Применяем ReLU к первому слою
        x = self.fc2(x)             # Выходной слой (без активации)
        return x

# Создание модели, функции потерь и оптимизатора
model = QuadraticNet(k=2)
criterion = nn.MSELoss()  # Функция потерь (среднеквадратичная ошибка)
optimizer = optim.Adam(model.parameters(), lr=0.1)  # Оптимизатор Adam

# Обучение модели
num_epochs = 100

for epoch in range(num_epochs):
    # Прямой проход
    outputs = model(x)
    loss = criterion(outputs, y)
```

```
    # Обратный проход и оптимизация
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

# Финальная визуализация
plt.figure(figsize=(5, 3))
plt.plot(x.numpy(), y.numpy(), "k--", label="true", lw=1)
plt.plot(x.numpy(), model(x).detach().numpy(), label="predictions",
color="#000099", lw=2)
# plt.title("Финальные предсказания")
plt.xlabel("$x$")
plt.ylabel("$y$")
plt.legend()
```

Несмотря на теоретические обоснования применимости двуслойных сетей, в них упущено несколько важных моментов:

1) сколько нейронов может понадобиться для решения реальных задач,
2) как обучать подобные сети,
3) не потребуется ли для достижения требуемой точности использование аномально больших по модулю весов (а это может повлиять на устойчивость модели).

В настоящее время используются многослойные сети (для ни тоже есть теоретические обоснования).

Многослойная нейронная сеть

Естественным обобщение рассмотренной ранее двухслойной сети является **многослойная нейронная сеть**, см. рис. В такой сети признаковые описания поступают на вход нейронов первого слоя, а результаты работы нейронов каждого k-го слоя передаются на вход нейронов $(k+1)$-го слоя. В зависимости от задачи, последний слой может быть организован следующим образом:

- один нейрон с сигмоидной функцией активации — для задач бинарной классификации,
- l нейронов с функцией softmax — для задач классификации с l непересекающимися классами,

- нейроны с тождественной функцией активации — для задач (многомерной) регрессии.

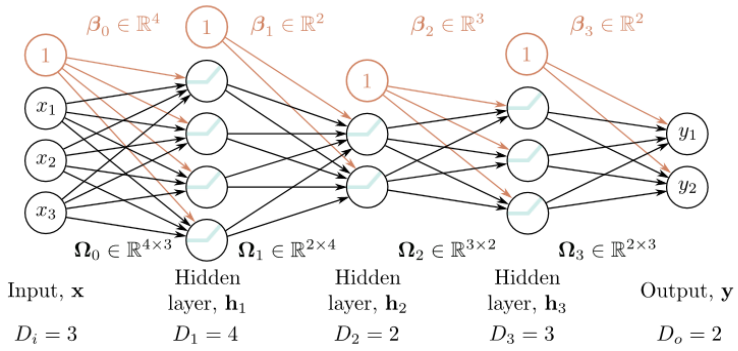


Figure 4.6 Matrix notation for network with $D_i = 3$-dimensional input $\mathbf{x}$, $D_o = 2$-dimensional output $\mathbf{y}$, and $K = 3$ hidden layers $\mathbf{h}_1, \mathbf{h}_2$, and $\mathbf{h}_3$ of dimensions $D_1 = 4$, $D_2 = 2$, and $D_3 = 3$ respectively. The weights are stored in matrices $\mathbf{\Omega}_k$ that multiply the activations from the preceding layer to create the pre-activations at the subsequent layer. For example, the weight matrix $\mathbf{\Omega}_1$ that computes the pre-activations at $\mathbf{h}_2$ from the activations at $\mathbf{h}_1$ has dimension 2×4. It is applied to the four hidden units in layer one and creates the inputs to the two hidden units at layer two. The biases are stored in vectors $\boldsymbol{\beta}_k$ and have the dimension of the layer into which they feed. For example, the bias vector $\boldsymbol{\beta}_2$ is length three because layer $\mathbf{h}_3$ contains three hidden units.

Рис. Сеть прямого распространения[1].

Описанная сеть называется **сетью прямого распространения** (Feedforward Neural Network, FFN). В её вычислительном графе отсутствуют циклы[2]. Существует некоторая неоднозначность в определении числа слоёв: часто нулевым слоем называют входные данные (признаковые описания), а последний слой — выходным. Все промежуточные слои называются **скрытыми слоями** (hidden layers).

Нейронные сети называют **глубокими**, если количество слоёв «достаточно велико». Хотя это определение неформально, развитие теории и практики глубокого обучения тесно связано с увеличением глубины сетей (числа слоёв).

Отметим интересную интерпретацию функционирования глубокой нейросети нейросети. Первый слой получает признаковое описание объекта $x \in \mathbb{R}^n$, второй слой – вектор

$$h_1 = \varphi_1(W_1 \cdot x + b_1),$$

здесь φ_1 – функция активации в первом слое (считаем, что она совпадает у всех нейронов первого слоя), $W_1 \in \mathbb{R}^{n_1 \times n}$ – $n_1 \times n$-матрица весов нейронов первого слоя (n_1 – число нейронов первого слоя), $b_1 \in \mathbb{R}^{n_1}$ – вектор смещений нейронов первого слоя. Заметим, что вся остальная часть нейросети знает об объекте лишь значения, записанные в векторе

$$h_1 \in \mathbb{R}^{n_1},$$

[1] Источник: [https://udlbook.github.io/udlbook/]

[2] Что отличает её от рекуррентных сетей, которые мы изучим дальше.

исходное признаковое описание x она не знает. Таким образом, первый слой трансформировал исходное признаковое описание в новое:

$$f_1 : x \to h_1 = \varphi_1(W_1 \cdot x + b_1).$$

Аналогично делает каждый слой:

$$h_{k+1} = \varphi_{k+1}(W_{k+1} \cdot h_k + b_{k+1}).$$

Глубокая нейронная сеть последовательно трансформирует признаковое описание с помощью нейронов, в результате получается выход сети:

$$\varphi_K(W_K \cdot \ldots \cdot \varphi_2(W_2 \cdot \varphi_1(W_1 \cdot x + b_1) + b_2) \ldots + b_K).$$

Вместо добавления векторов смещений можно было использовать конкатенацию векторов, как раньше. Если опустить смещения (иногда используют слои без смещений), то формула упростится:

$$\varphi_K(W_K \cdot \ldots \cdot \varphi_2(W_2 \cdot \varphi_1(W_1 \cdot x)) \ldots)^{1}.$$

Сейчас наука глубокого обучения, в основном, занимается тем, чтобы понять, как правильно представлять и преобразовывать признаковые пространства с помощью глубоких сетей[2].

Почему нужны именно глубокие нейронные сети? Сейчас на этот вопрос мы ответить не сможем, но дадим несколько примеров, в которых «более сложные» сети предпочтительнее простых. На рис. показаны гистограммы распределений ошибок сетей с разным числом нейронов. Интересно, что более **сложные сети не только имеют меньшую среднюю ошибку** (середина соответствующего колокола), **но и меньшую дисперсию ошибки** (его ширина).

[1] В некоторых научных статьях нейросети определяют с помощью таких формул.

[2] Далее мы это увидим, например, в таких моделях как кодировщик. Понятно, что на преобразования логично накладывать определённые требования.

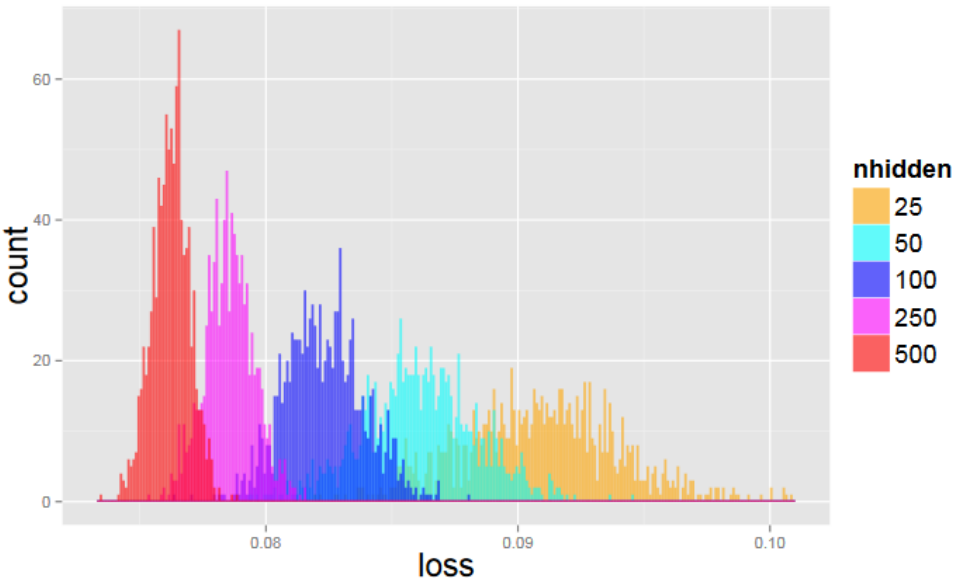


Рис. Из Anna Choromanska, et. al. «The Loss Surfaces of Multilayer Networks» 2015, https://arxiv.org/abs/1412.0233

На рис. отмечается, что большим языковым моделям[1] требуется меньшее число токенов для достижения фиксированной точности (по сравнению с малыми языковыми моделями). Это не означает, что они быстрее обучаются[2], но можно сказать, что **большим моделям нужно меньше данных** (по крайней мере, в некоторых практически важных случаях).

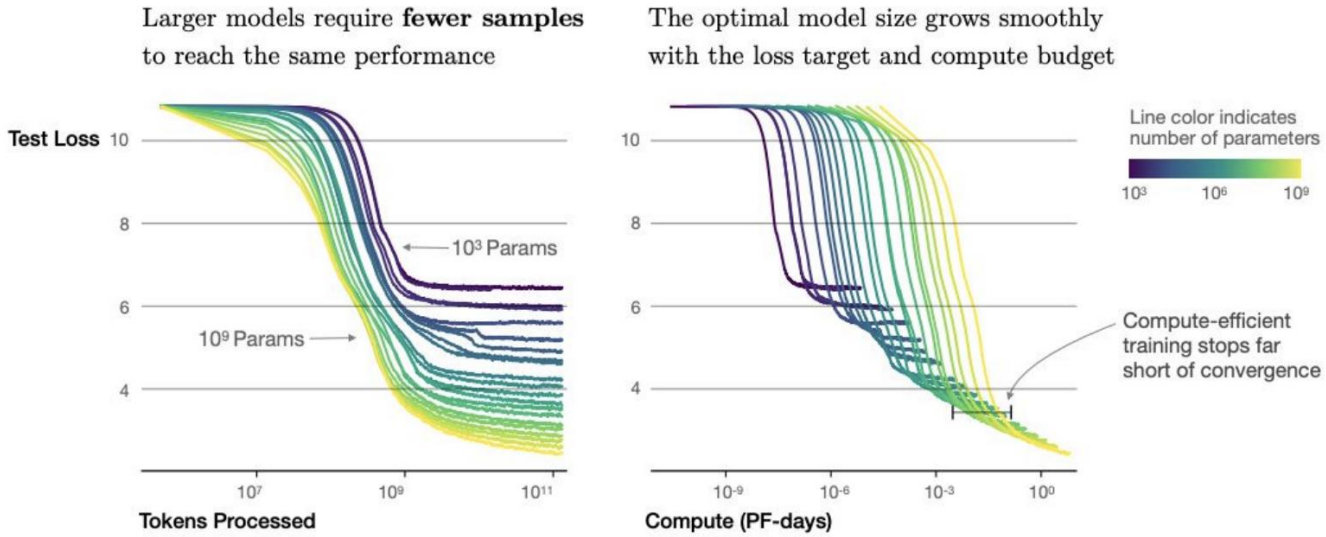


Рис. Уменьшение ошибки при обучении языковых моделей.

Также упомянем **эмерджентность** – свойство сложных систем, при котором возникают новые качества, свойства или поведение, не присущие отдельным

[1] Это отдельный вид нейросетей, с которым мы познакомимся позже.

[2] Они сложнее и результат получают (дальше будем называть это прямым проходом) дольше.

элементам системы, но проявляющиеся при их взаимодействии[1]. Нейросети её блестяще демонстрируют (как мы увидим в дальнейшем, простое функционирование нейронов приводит, например, к детектированию объектов на изображениях). При этом ярко оно появляется при достижении сложности модели нейросети некоторого порога, см. рис.

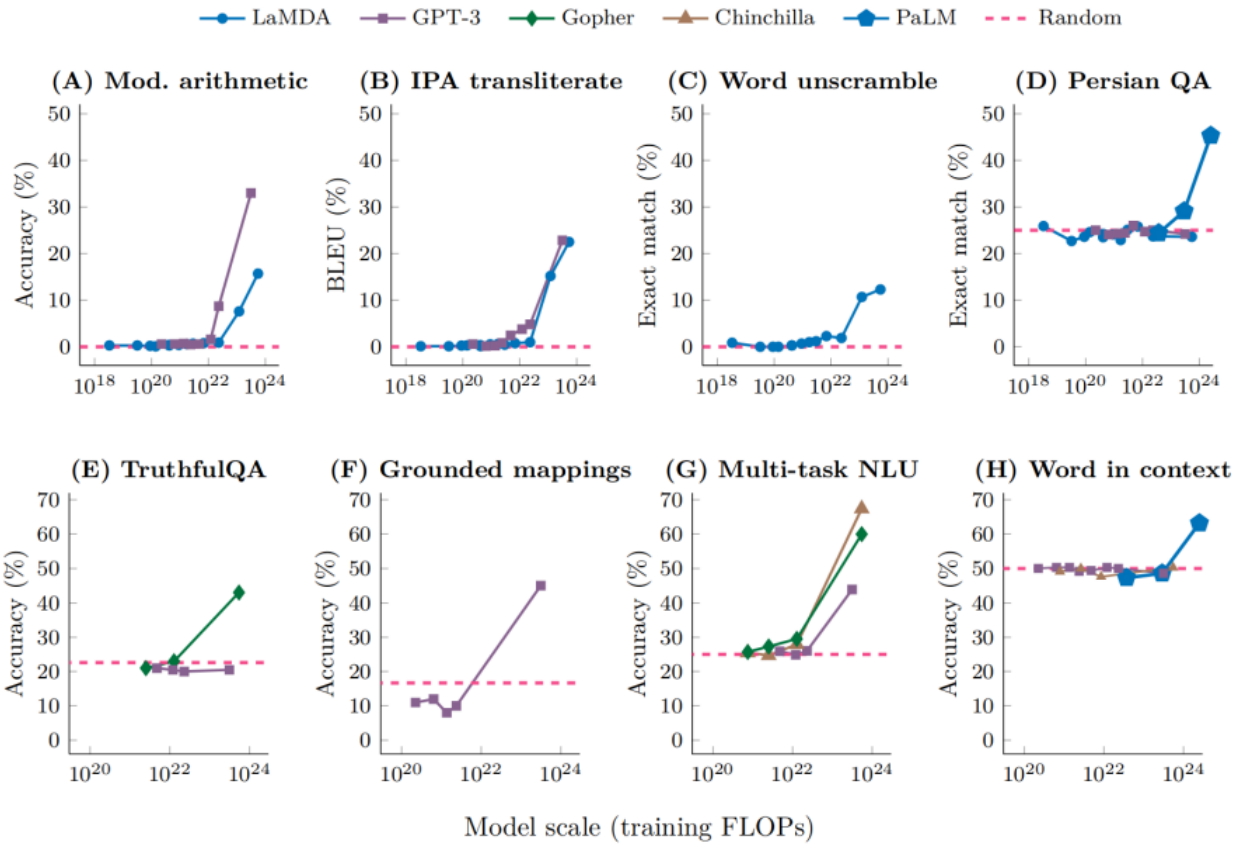


Рис. Качество решения NLP-задач различными нейросетями.

К сожалению, это означает, что нейросети нельзя хорошо исследовать на домашнем оборудовании и игрушечных датасетах (в отличие от многих классических моделей). Поскольку модели аналогичной архитектуры, но более сложные, которые обучены на большем объёме данных, показываю совершенно другие свойства.

Написать многослойную нейросеть прямого распространения в PyTorch не сильно сложнее, чем однослойную. Есть несколько способов. Самый простой – с помощью **nn.Sequential** сконкатенировать нужные модули (чередуем

[1] Неформально, это когда «объединение больше, чем просто сумма слагаемых». Отвлечённый пример – рынок. Простые действия отдельных участников (покупателей и продавцов) приводят к возникновению сложных экономических явлений, таких как спрос, предложение и цены.

линейные модули и модули активаций).

```
import torch
import torch.nn as nn
import torch.optim as optim

# Определение модели с помощью nn.Sequential
model = nn.Sequential(
    nn.Linear(784, 256),  # Полносвязный слой с 784 входами и 256 выходами
    nn.Sigmoid(),            # Функция активации ReLU
    nn.Linear(256, 128),  # Второй полносвязный слой
    nn.Sigmoid(),            # Функция активации ReLU
    nn.Linear(128, 10)    # Выходной слой для 10 классов
)

# Пример входных данных (батч из 32 примеров, каждый размером 784)
x = torch.randn(32, 784)

# Прямой проход через модель
output = model(x)
```

Второй способ более гибкий и рекомендуется для сложных архитектур: создать класс, наследуемый от nn.Module. В конструкторе определяются слои, а в методе forward – как функционирует сеть.

```
import torch
import torch.nn as nn

# Определение модели как подкласса nn.Module
class DeepNeuralNetwork(nn.Module):
    def __init__(self):
        super(DeepNeuralNetwork, self).__init__()
        self.fc1 = nn.Linear(784, 256)  # Первый полносвязный слой
        self.fc2 = nn.Linear(256, 128)  # Второй полносвязный слой
        self.fc3 = nn.Linear(128, 10)   # Выходной слой
        self.sigmoid = nn.Sigmoid()           # Функция активации

    def forward(self, x):
        x = self.sigmoid(self.fc1(x))  # Применяем активацию к первому слою
        x = self.sigmoid(self.fc2(x))  # Применяем активацию ко второму слою
        x = self.fc3(x)              # Выходной слой (без активации)
        return x

# Создание экземпляра модели
model = DeepNeuralNetwork()

# Пример входных данных (батч из 32 примеров, каждый размером 784)
x = torch.randn(32, 784)
```

```
# Прямой проход через модель
output = model(x)
```

В более сложном способе, который нужен в исключительных случаях, например, если архитектура сети определяется на этапе выполнения (например, количество слоев задаётся параметром) используется nn.ModuleList.

```
import torch
import torch.nn as nn

# Определение модели с динамическим созданием слоев
class DeepNeuralNetwork(nn.Module):
    def __init__(self, layer_sizes):
        super(DeepNeuralNetwork, self).__init__()
        self.layers = nn.ModuleList()
        for i in range(len(layer_sizes) - 1):
            self.layers.append(nn.Linear(layer_sizes[i], layer_sizes[i +
1]))
            if i < len(layer_sizes) - 2:  # Добавляем активацию, кроме
последнего слоя
                self.layers.append(nn.Sigmoid())

    def forward(self, x):
        for layer in self.layers:
            x = layer(x)
        return x

# Создание экземпляра модели с заданной архитектурой
layer_sizes = [784, 256, 128, 10]  # Архитектура сети
model = DeepNeuralNetwork(layer_sizes)

# Пример входных данных (батч из 32 примеров, каждый размером 784)
x = torch.randn(32, 784)

# Прямой проход через модель
output = model(x)
```

Обучение нейронных сетей

По аналогии с классическим машинным обучением, нейронные сети обучаются с помощью минимизации эмпирического риска. Такая задача оптимизации, вообще говоря, невыпуклая, поэтому нахождение локального минимума не гарантирует глобальный минимум[1]. Число слагаемых в сумме ошибок (0) равно

[1] Как отметим дальше, не факт, что его надо находить.

объёму обучающей выборки и, как правило, очень большое, поэтому используют метод стохастического градиентного спуска[1] (Stochastic Gradient Descent, SGD):

1. Случайная инициализация весов $w^{(0)} \sim \text{norm}(0, \sigma^2)$ [2].
2. Цикл по t до сходимости
   a. Выбираем случайный объект x_i, $i = \text{random\_form}(\{1, 2, \ldots, m\})$.
   b. Вычисляем градиент на этом объекте
   $$\nabla[L(a(x_i; w^{(t)}), y_i) + \lambda R(w^{(t)})].$$
   c. Производим адаптацию весов
   $$w^{(t+1)} = w^{(t)} - \eta \nabla[L(a(x_i; w^{(t)}), y_i) + \lambda R(w^{(t)})].$$

Гиперпараметр $\eta > 0$ называется **скоростью или темпом обучения (learning rate)**. Дальше мы уточним этот базовый метод и предложим более продвинутые методы оптимизации, но пока нам будет достаточно понимания, что **для обучения нейронных сетей необходимо вычислять градиенты в точках выборки некоторой функции**.

Рассмотрим популярную в классификации на непересекающиеся классы функцию ошибки: **кросс-энтропию** (CrossEntropyLoss), запишем её как функцию от логитов (суперпозицию softmax и кроссэнтропии[3]):

$$L(y, (a_1, \ldots, a_l)) = -\log \frac{\exp(a_y)}{\sum_{j=1}^{l} \exp(a_j)} = -a_y + \log \sum_{j=1}^{l} \exp(a_j).$$

Прямое вычисление кросс-энтропии по приведённой формуле может быть **вычислительно неустойчивым** из-за возможного переполнения или обнуления при экспоненциальных операциях. Для повышения устойчивости используется следующий приём:

[1] На самом деле, оптимизация производится по батчам, но об этом дальше.

[2] Инициализацию также будем подробно обсуждать дальше.

[3] Именно так и сделано в одной из реализаций pytorch-а.

$$\frac{\exp(a_y)}{\sum_{j=1}^{l}\exp(a_j)} = \frac{\exp(a_y)}{\sum_{j=1}^{l}\exp(a_j)}\frac{\exp(-a_t)}{\exp(-a_t)} = \frac{\exp(a_y - a_t)}{\sum_{j=1}^{l}\exp(a_j - a_t)},$$

поэтому

$$L(\cdot,\cdot) = -a_y + \max\{a_j\}_{j=1}^{l} + \log\left(\sum_{j=1}^{l}\exp(a_j - \max\{a_j\}_{j=1}^{l})\right).$$

На практике такие трюки уже встроены в функции вычисления ошибки.

Возможность использовать «самописные» функции ошибки придаёт нейросетевому подходу к решению задач машинного обучения большую гибкость. Важно, чтобы функция ошибки была:

- дифференцируема,
- устойчива к вычислительным ошибкам.

Для реализации нужно опять наследовать класс nn.Module и в методе forward написать вычисление ошибки, приведём пример собственной реализации.

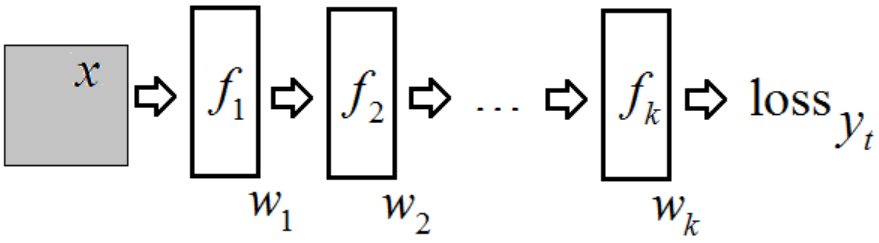
```
import torch
import torch.nn as nn

class CustomMSELoss(nn.Module):
    """
    Пользовательская функция потерь: среднеквадратичная ошибка (MSE)
    """
    def __init__(self, l2_lambda=0.01):
        super(CustomMSELoss, self).__init__()

    def forward(self, predictions, targets):
        """
        Вычисляет значение функции потерь.

        Параметры:
            predictions (torch.Tensor): Предсказания модели (выход сети).
            targets (torch.Tensor): Истинные значения (целевые значения).

        Возвращает:
            torch.Tensor: Значение функции потерь.
        """
        # Вычисляем среднеквадратичную ошибку (MSE)
        mse_loss = torch.mean((predictions - targets) ** 2)
```

```
        return mse_loss
```

Теперь вычислить ошибку можно, например, так

```
import torch
import torch.nn as nn

# Определение модели
model = nn.Sequential(
    nn.Linear(100, 10),
    nn.Sigmoid(),
    nn.Linear(10, 1)
)

# Пример входных данных (батч из 32 примеров)
x = torch.randn(32, 100)
y = torch.randn(32, 1)

# Прямой проход через модель
output = model(x)

# Создаём экземпляр пользовательской функции потерь
criterion = CustomMSELoss()

# Вычисление функции потерь
loss = criterion(output, y)
```

Запишем нейросеть в виде вычислительного графа



,

который состоит из модулей[1] $f_1,\ldots,f_k$, зависящих соответственно от параметров $w_1,\ldots,w_k$. Исходный объект x проходит через суперпозицию модулей

$$a(x) = f_k(\ldots f_2(f_1(x;w_1);w_2)\ldots;w_k),$$

[1] Модуль – центральное понятие PyTorch. Традиционный слой в FFN это конкатенация линейного модуля и модуля активации.

на объекте вычисляется ошибка

$$L(y, a(x)) = L(y, f_k(\dots f_2(f_1(x; w_1); w_2)\dots; w_k)).$$

Важно, что она имеет значение из $\mathbb{R}$ (это просто вещественное число, даже вещественное неотрицательное). Для простоты уберём некоторые параметры и будем считать, что функция ошибки представляется в виде суперпозиции

$$L(w) \sim L(f_k(\dots f_2(f_1(w))\dots)), \quad (3)$$

которую мы хотим минимизировать по параметру w (вообще говоря, это вектор параметров). Вычислим градиент по правилу вычисления градиента сложной функции:

$$\nabla L = \frac{\partial f_1}{\partial w}(w) \cdot \frac{\partial f_2}{\partial f_1}(f_1(w)) \cdot \dots \cdot \frac{\partial L}{\partial f_k}(f_k(\dots f_2(f_1(w))\dots)).$$

К этому выражению можно относиться как к формальному, хотя отметим, что правый множитель является градиентом, а остальные матрицами Якоби. Но к этому мы позже вернёмся. Если все функции одномерные одного аргумента, тогда в этом выражении везде стоят производные. Синим цветом мы выделили, в каких точках вычисляются производные / градиенты / матрицы Якоби.

Прямым проходом по вычислительному графу сети называется последовательное вычисление суперпозиции: подставляем в (3) значение w (на самом деле, ещё и объект), получаем сначала $f_1(w)$, затем подставляем это значение в $f_2(\cdot)$ и получаем $f_2(f_1(w))$ и т.д.:

$$f_1(w) \to f_2(f_1(w)) \to f_3(f_2(f_1(w))) \to \dots \to f_k(\dots f_2(f_1(w))\dots),$$

синим цветом здесь подсвечено вычисленное на предыдущем этапе.

Обратным проходом называется вычисление градиента функции ошибки по вектору параметров w, поскольку оно производится в противоположную сторону:

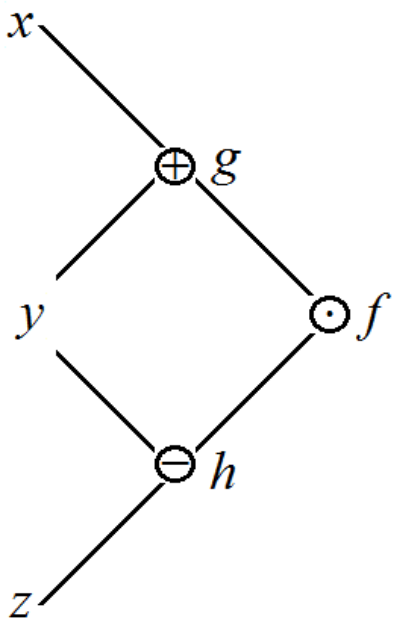
$$\frac{\partial L}{\partial f_k}(f_k(\dots f_2(f_1(w))\dots)) \to \frac{\partial L}{\partial f_{k-1}} = \frac{\partial L}{\partial f_k}\frac{\partial f_k}{\partial f_{k-1}}(f_{k-1}(\dots f_2(f_1(w))\dots)) \to$$

$$\dots$$

$$\to \frac{\partial L}{\partial f_1} = \frac{\partial L}{\partial f_k}\frac{\partial f_k}{\partial f_{k-1}} \cdot \dots \cdot \frac{\partial f_3}{\partial f_2}\frac{\partial f_2}{\partial f_1}(f_1(w)) \to \frac{\partial L}{\partial w} = \frac{\partial L}{\partial f_k} \cdot \dots \cdot \frac{\partial f_2}{\partial f_1}\frac{\partial f_1}{\partial w}(w). \quad (4)$$

здесь опять синим подсвечено вычисленное на предыдущем шаге, а красным – вычисленное на прямом проходе (не всегда указано). Таким образом, **чтобы вычислять градиенты по параметрам, надо хранить результаты прямого прохода.** Ниже мы это продемонстрируем на простом примере.

До сих пор мы активно использовали понятие вычислительный граф. В простейшем случае, это направленный граф, в котором вершинам соответствуют переменные и функции, а рёбрам – зависимости. Например, выражению $f=(x+y)\cdot(y-z)$ соответствует граф на рис.



Здесь мы ввели ещё обозначение для внутренних вершин:

$$g(x,y)=(x+y),$$

$$h(y,z)=(y-z),$$

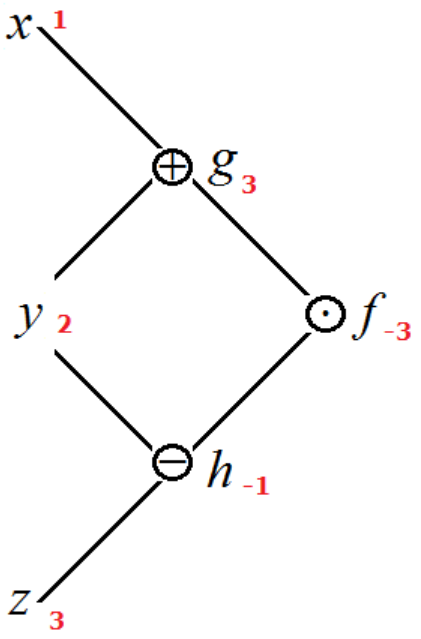
$$f(x,y,z)=g(x,y)\cdot h(y,z).$$

Прямой проход по графу соответствует вычислению выражения, например в точке $x=1,\ y=2,\ z=3$:

$$g(1,2)=1+2=3,$$

$$h(2,3)=2-3=-1,$$

$$f(x,y,z)=3\cdot(-1)=-3.$$



.

На обратном подходе вычисляются производные, например,

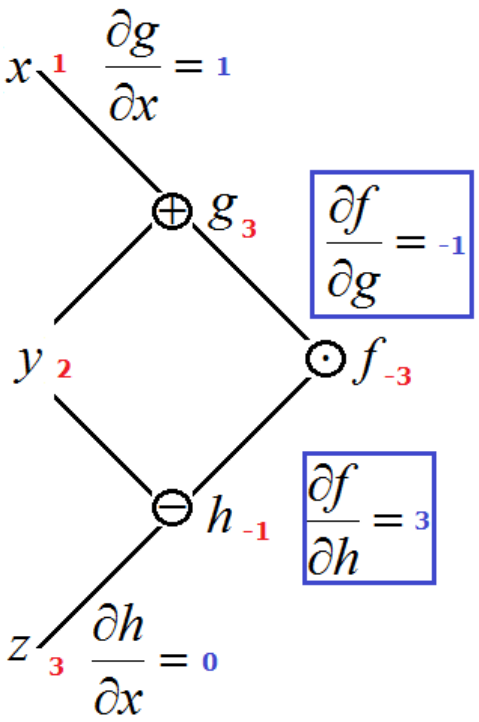
$$\frac{\partial f}{\partial x}=\frac{\partial f}{\partial g}\frac{\partial g}{\partial x}+\frac{\partial f}{\partial h}\frac{\partial h}{\partial x},$$

Причём можно заранее вычислить, что:

$$\frac{\partial f}{\partial g}=h,\frac{\partial f}{\partial h}=g\,,\ \frac{\partial g}{\partial x}=1,\frac{\partial h}{\partial x}=0\,,$$

поэтому

$$\frac{\partial f}{\partial x}=h\cdot 1+g\cdot 0=-1\cdot 1=-1$$



Эти вычисления не зависит от конкретных значений переменных x, y, z, но обратный проход нельзя сделать без прямого, поскольку

$$\frac{\partial f}{\partial g} = h,$$

т.е. значение h надо вычислить и запомнить. Это иллюстрирует, что **при обучении нейросетей используется много памяти**, т.к. хранится сама нейросеть (её параметры), промежуточные результаты прямого прохода (на каждом модуле) и значения производных параметров[1].

На самом деле, PyTorch это библиотека не для написаний нейростей, а для работы с графами вычислений. Возможность эффективно работать с нейросетями получается как следствие этого, поскольку нейросеть является частным случаем графа вычислений. Продемонстрируем, как происходит работа с графами. Создание графа получается автоматически при вычислении выражения в PyTorch.

```
import torch
from torch.autograd import Variable
```

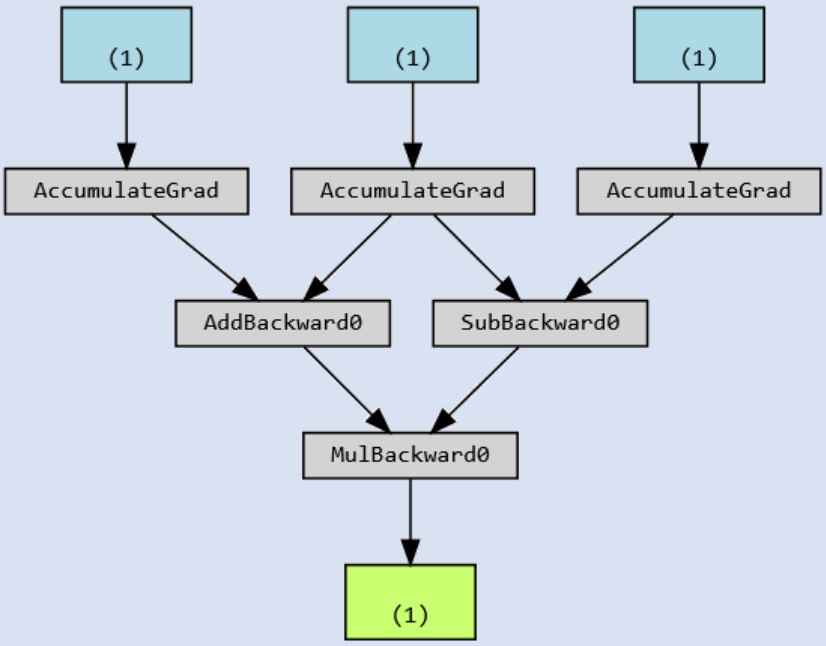
[1] Есть нейронные сети, в которых нет необходимости при обучении хранить промежуточные результаты вычислений. Например, обратимые нейронные сети (Reversible Neural Networks): RevNet и iRevNet, в которых промежуточные результаты могут быть **восстановлены** во время обратного прохода, что позволяет не хранить их в памяти.

```
# переменные
x = torch.tensor([1.], requires_grad=True)
y = torch.tensor([2.], requires_grad=True)
z = torch.tensor([3.], requires_grad=True)
f = (x + y) * (y - z) # прямой проход - вычисление

f.backward() # обратный проход - вычисление производных
x.grad, y.grad, z.grad # производные
```

```
(tensor([-1.]), tensor([2.]), tensor([-3.]))
```

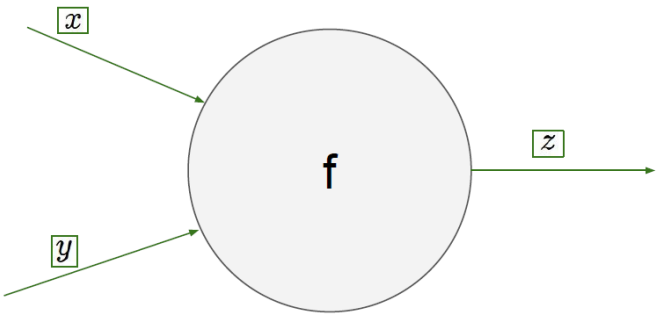
Здесь мы воспроизвели пример, разобранный выше. Создали три переменные. В PyTorch переменная это просто тензор, содержащий одно значение. При создании мы указали свойство requires_grad, чтобы PyTorch знал, что требуется вычисление производных по этим переменным. Прямой проход происходит при вычислении выражения для *f*, в этот момент создаётся граф вычислений, можно даже его изобразить.

```
from torchviz import make_dot

make_dot(f)
```



Таким образом, вычисления в PyTorch отличаются от вычислений в стандартных математических пакетах. Здесь выражение интерпретируется как граф вычислений. Нужные вычисления производных делаются с помощью обратного прохода, для реализации которого используют метод backward. При этом градиенты сохраняются только у тензоров со свойством requires_grad.

Приведём ещё одну известную иллюстрацию обратного прохода[1]. Пусть в графе вычислений есть вершина, которая считает функцию $z = f(x, y)$:



При прямом проходе в вершину приходят значения x и y, и вычисляется значение f при пришедших аргументах. Можно заранее посчитать

$$\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y},$$

эти значения называют также **локальными градиентами**. При обратном проходе в эту вершину приходит

$$\frac{\partial L}{\partial z},$$

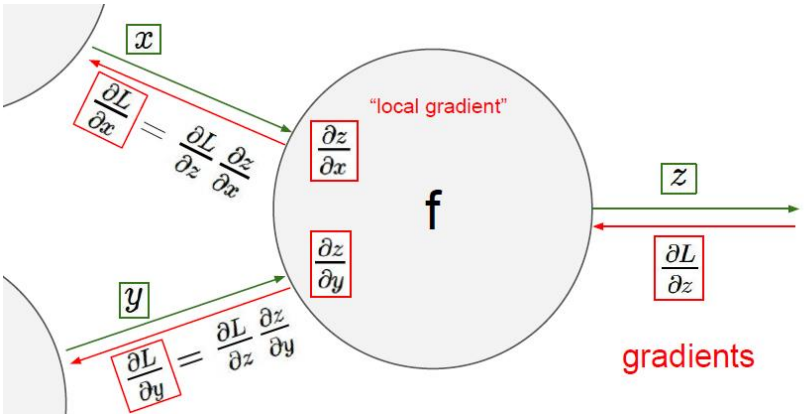
Необходимо умножить это значение на локальные градиенты и передать в вершины x и y, поскольку[2]

$$\frac{\partial L}{\partial x} = \frac{\partial z}{\partial x}\frac{\partial L}{\partial z},$$

$$\frac{\partial L}{\partial y} = \frac{\partial z}{\partial y}\frac{\partial L}{\partial z}:$$

[1] Из Стэнфордского курса «Deep Learning for Computer Vision».

[2] Рис. взяты из http://cs231n.stanford.edu/2017/index.html



Вернёмся к формулам для вычисления градиента с помощью обратного прохода (4). Напомним некоторые термины векторного дифференцирования[1], если есть зависимость $z(x)$, то в зависимости от размерности z и x говорят:

производная[2]	$x \in \mathbb{R}, z \in \mathbb{R} \Rightarrow$	$\frac{\partial z}{\partial x} \in \mathbb{R}$
градиент	$x \in \mathbb{R}^n, z \in \mathbb{R} \Rightarrow$	$\frac{\partial z}{\partial x} = \left(\frac{\partial z}{\partial x_1}, \ldots, \frac{\partial z}{\partial x_n} \right)^{\mathrm{T}} \in \mathbb{R}^n$
матрица Якоби	$x \in \mathbb{R}^n, z \in \mathbb{R}^m \Rightarrow$	$\frac{\partial z}{\partial x} = \left\| \frac{\partial z_j}{\partial x_i} \right\|_{n \times m} \in \mathbb{R}^{n \times m}$

Здесь матрица Якоби нестандартно записана[3] в виде:

$$\frac{\partial z}{\partial x} = \left\| \begin{matrix} \frac{\partial z_1}{\partial x_1} & \cdots & \frac{\partial z_m}{\partial x_1} \\ \cdots & \cdots & \cdots \\ \frac{\partial z_1}{\partial x_n} & \cdots & \frac{\partial z_m}{\partial x_n} \end{matrix} \right\|_{n \times m} \in \mathbb{R}^{n \times m},$$

но это упростит дальнейшие формулы. Рассмотрим простейший модуль активации:

[1] Очень полезно также изучить обзор по дифференцированию на компьютере: Atilim Gunes Baydin, Barak A. Pearlmutter, Alexey Andreyevich Radul, Jeffrey Mark Siskind «Automatic differentiation in machine learning: a survey» 2015-2018 https://arxiv.org/abs/1502.05767

[2] Здесь не совсем корректны используются знаки принадлежности. Но, надеемся, такое упрощение не повредит пониманию.

[3] Обычно используют транспонированную матрицу.

$$z = \varphi(x),\ x \in \mathbb{R}^n,\ z \in \mathbb{R}^n,$$

функция активации действует поэлементно на вектор x, тогда очевидно, что

$$\frac{\partial z}{\partial x} = \operatorname{diag}(\varphi'(x_1),\ldots,\varphi'(x_n)) = \begin{bmatrix} \varphi'(x_1) & 0 & \ldots & 0 \\ 0 & \varphi'(x_2) & \ldots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \ldots & \varphi'(x_n) \end{bmatrix} \quad (5)$$

и на эту матрицу умножается градиент при обратном проходе:

$$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial x}\frac{\partial L}{\partial z}.$$

Теперь рассмотрим линейный модуль:

$$z = Wx, \quad z \in \mathbb{R}^m, x \in \mathbb{R}^n, W \in \mathbb{R}^{m \times n}, \quad (6)$$

матрица Якоби

$$\frac{\partial z}{\partial x} = \frac{\partial Wx}{\partial x} = \frac{\partial \begin{bmatrix} w_{11}x_1 + \ldots + w_{1n}x_n \\ \cdots \\ w_{m1}x_1 + \ldots + w_{mn}x_n \end{bmatrix}}{\partial x} = \begin{bmatrix} w_{11} & \ldots & w_{m1} \\ \cdots & \cdots & \cdots \\ w_{1n} & \ldots & w_{mn} \end{bmatrix} = W^{\mathrm{T}}.$$

Конкатенация модулей приводит к перемножению матриц Якоби, например если

$$z = \mathrm{ReLU}(Wx), \quad z \in \mathbb{R}^m, x \in \mathbb{R}^n, W \in \mathbb{R}^{m \times n}, \text{ то}$$

$$\frac{\partial z}{\partial x} = \frac{\partial h}{\partial x}\frac{\partial z}{\partial h}\bigg|_{h=Wx} = W^{\mathrm{T}}_{n \times m} \operatorname{diag}(\varphi'(Wx))_{m \times m}.$$

Если теперь записать FFN:

$$z = \varphi(W_k(\ldots W_2\varphi(W_1 x))),$$

то

$$\frac{\partial z}{\partial x} = W_1^{\mathrm{T}} \operatorname{diag}(\varphi'(W_1 x)) \cdot W_2^{\mathrm{T}} \operatorname{diag}(\varphi'(\cdot)) \cdot \ldots \cdot W_k^{\mathrm{T}} \operatorname{diag}(\varphi'(\cdot)) \quad (7)$$

(для простоты опустили векторы, в которых вычисляются матрицы Якоби модулей активации).

В формулах выше было некоторое лукавство, ведь градиенты вычисляются для метода оптимизации SGD и берутся по параметрам нейронной сети. В модуле активации параметров нет, а вот в линейном модуле (6) матрица W и есть матрица параметров. Можно вывести, например, такую формулу[1]

$$\frac{\partial L}{\partial W} = \underset{m\times 1}{\frac{\partial L}{\partial z}}\frac{\partial z}{\partial W} = \underset{1\times n}{\frac{\partial L}{\partial z}} x^{\mathrm{T}} = \nabla_z L \cdot x^{\mathrm{T}}.$$

– это $m\times n$-матрица, ij-й элемент которой равен $\partial L / \partial w_{ij}$, где $W = \| w_{ij} \|$ (это нестандартное обозначение и использовано для удобства).

В этом разделе мы поняли, как с помощью обратного прохода по графу вычислять градиенты по параметрам для оптимизации ошибки нейронной сети. Отметим, что графы вычислений могут быть разные, см. рис.

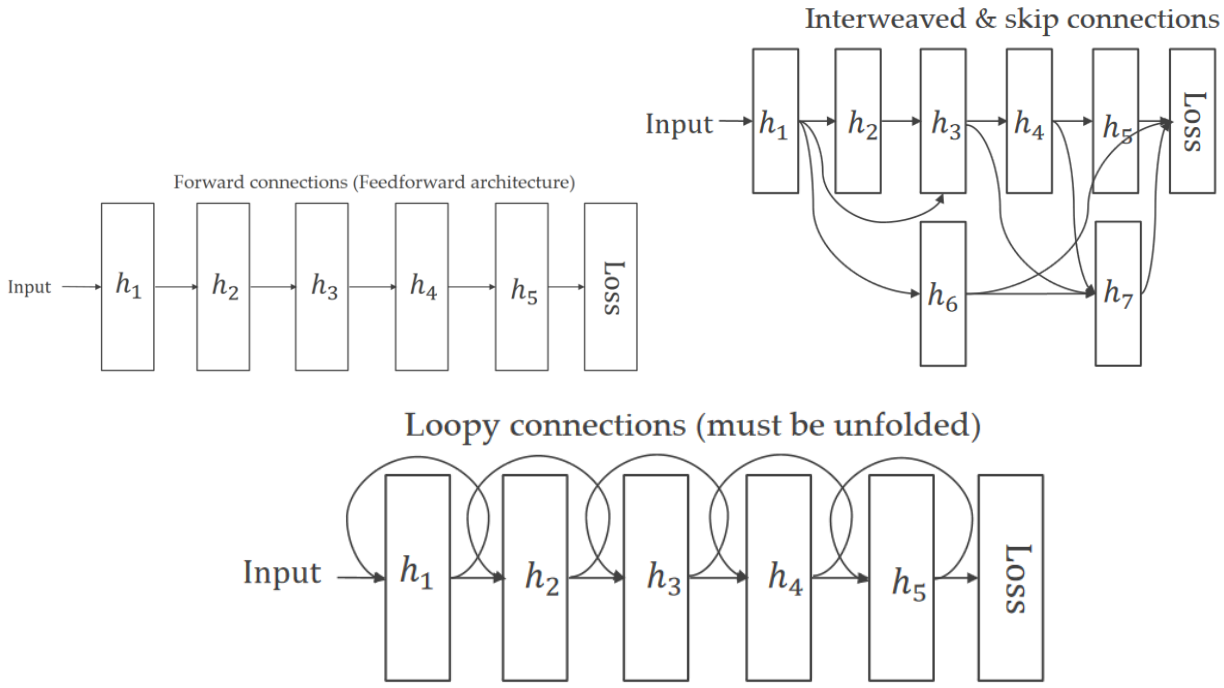


Рис. Графы вычислений[2].

[1] Вывод оставляем читателю.

[2] Источник: ???

Проблема затухания градиента

Если в нейросети присутствует модуль активации[1], тогда при вычислении градиента используется матрица Якоби (5) с производными функции активации на диагонали. Здесь нам и пригождаются производные, которые мы ранее вычислили для функций активации. Заметим, что у сигмоиды производная

$$\frac{\partial \sigma(z)}{\partial z} = \sigma(z)(1-\sigma(z)),$$

для больших по модулю значений близка к нулю. В глубокой сети с большим числом модулей активации с сигмоидой это может привести к тому, что градиент обратится в ноль (из-за точности вычислений) или станет очень маленьким (по модулю), что всё равно плохо, поскольку шаги градиентным методом будут очень небольшие. Это явление называется **затуханием градиента**.

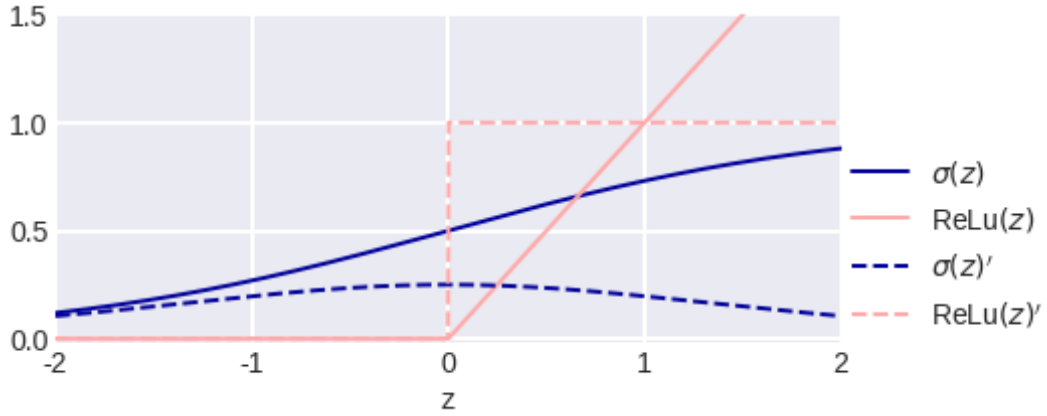


Рис. Две функции активации и их производные

Решить проблему затухания можно с помощью функции, у которой производная не так близка к нулю. Чаще используют ReLU (Rectified Linear Unit):

$$\mathrm{ReLU}(z) = \max(0, z),$$

$$\frac{\partial \mathrm{ReLU}(z)}{\partial z} = I[z > 0].$$

При её вычислении не надо брать экспоненту (она вообще быстрее вычисляется), на «половине» области определения производная равна константе 1, правда на другой она нулевая. Из-за последнего обстоятельства

[1] Он должен быть, чтобы решение получилось нелинейным.

появляется опасность наличия в сети т.н. **мёртвых нейронов (Dead Neurons)**, которые выдают ноль на любом объекте выборки. В реальных сетях такие нейроны есть, но их доля незначительна, и она уменьшается по мере обучения сети.

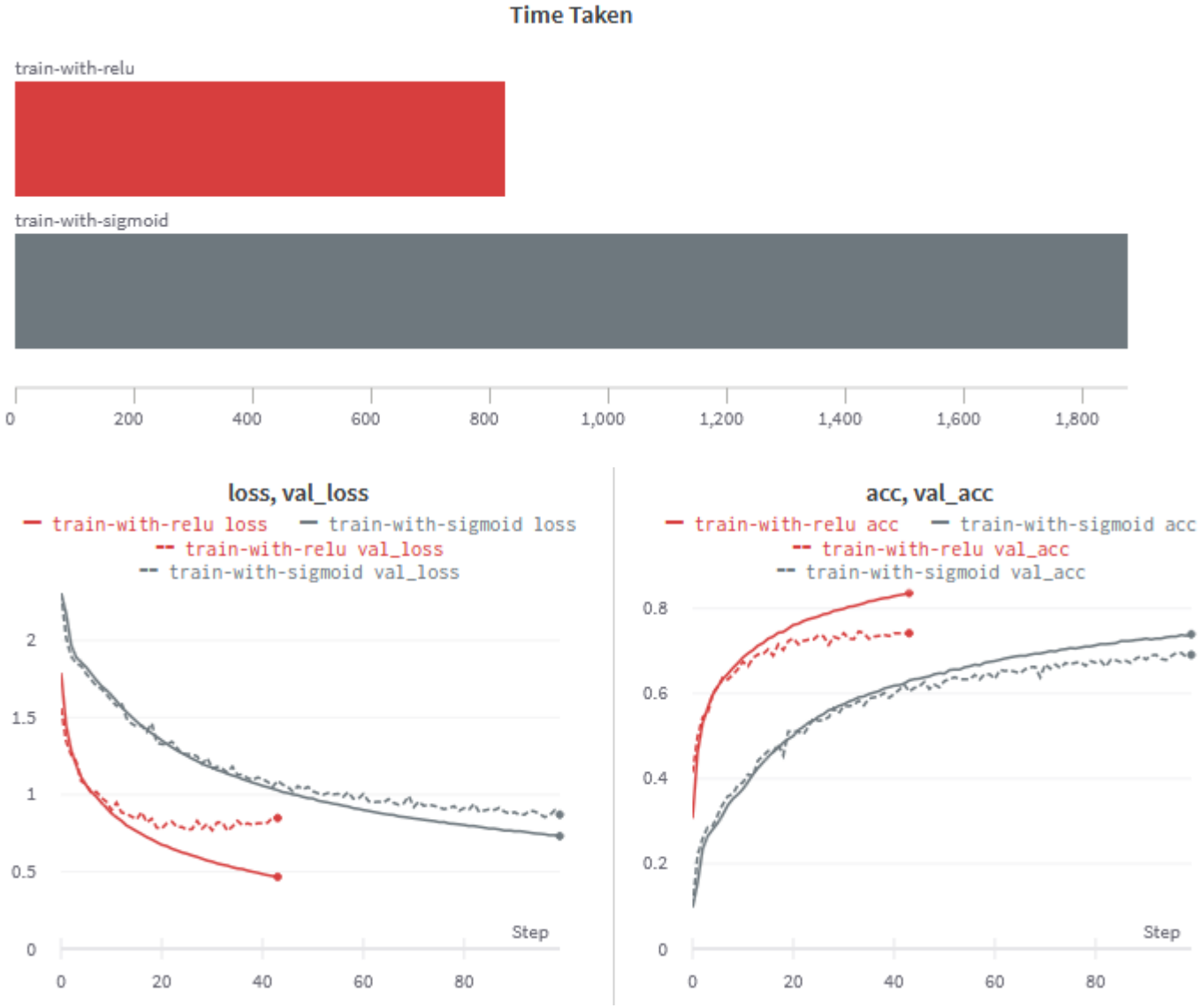


Рис. Сравнение сигмоиды и ReLU[1].

Функция активаций довольно много[2] – большинство из них решают проблемы затухания и стабильности вычислений. Приведём несколько популярных:

[1] https://wandb.ai/ayush-thakur/dl-question-bank/reports/ReLU-vs-Sigmoid-Function-in-Deep-Neural-Networks-Why-ReLU-is-so-Prevalent--VmlldzoyMDk0MzI

[2] См. например обзор Jagtap A. D., Karniadakis G. E. How important are activation functions in regression and classification? A survey, performance comparison, and future directions //Journal of Machine Learning for Modeling and Computing. – 2023. – Т. 4. – №. 1. // https://arxiv.org/pdf/2209.02681.pdf

LeakyReLU	$\text{LeakyReLU}(z) = \max(0.01z, z)$
Exponential Linear Unit	$\text{ELU}(z) = \begin{cases} z, & z \geq 0, \\ \alpha(e^z - 1), & z < 0. \end{cases}$
Softplus	$\text{softplus}(z) = \ln(1 + \exp(z))$
Swish	$\text{swish}(z) = x \cdot \sigma(\alpha x)$
Gaussian Error Linear Unit[1]	$\text{GELU}(z) = \frac{z}{2}\left(1 + \tanh\left(\sqrt{\frac{2}{\pi}}(z + \alpha z^3)\right)\right)$
Gated Linear Unit	$\text{GLU}(z) = \sigma(W^{\text{T}} z + w_0) \cdot (V^{\text{T}} z + v_0)$
Maxout[2]	$\text{Maxout}(z) = \max(w^{\text{T}} z + w_0, v^{\text{T}} z + v_0)$

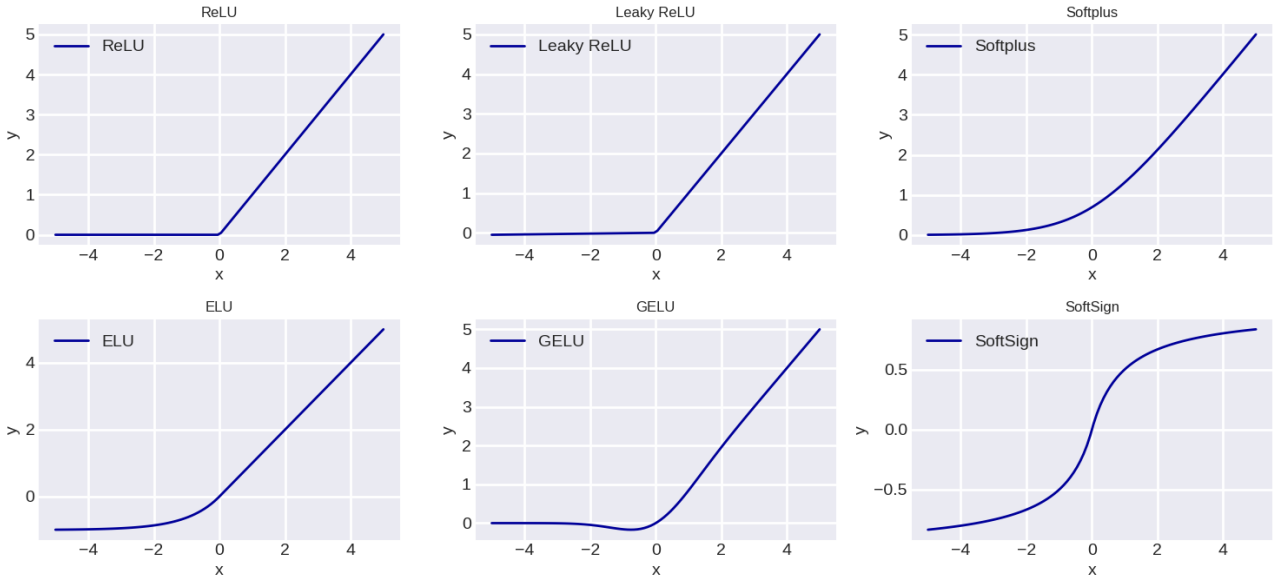


Рис. Различные функции активации.

[1] Приобрела популярность в трансформерах, α=0.044715.

[2] Maxout и GLU не совсем функции активации, т.к. имеет настраиваемые параметры. Традиционно функции активации таких параметров не имеют.

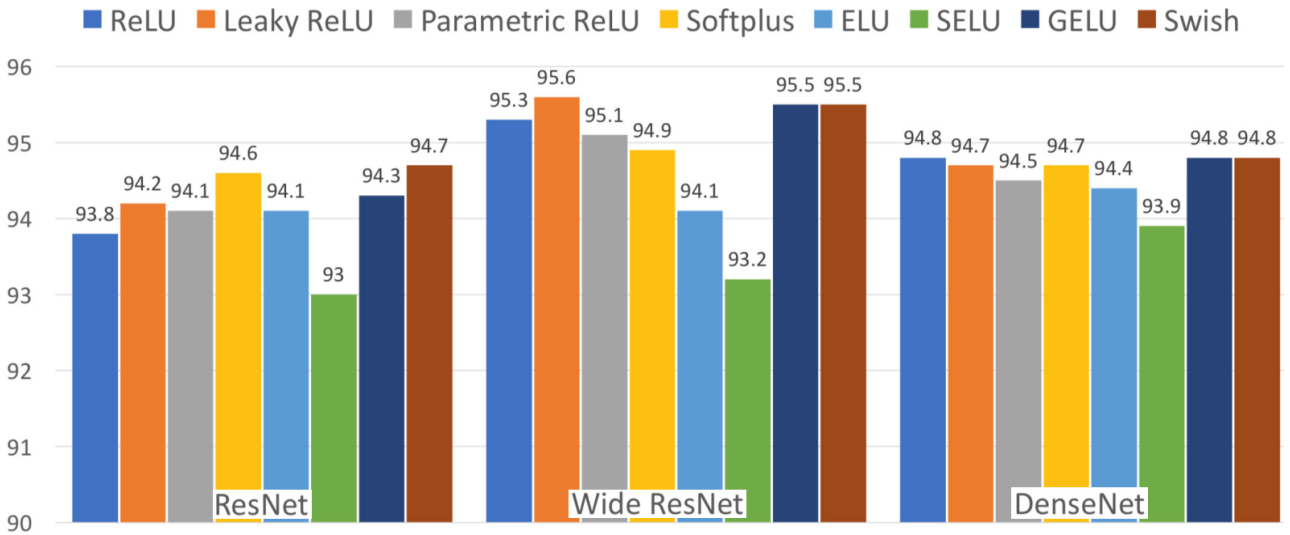


Рис. Сравнение разных функций активации на датасете CIFAR10[1].

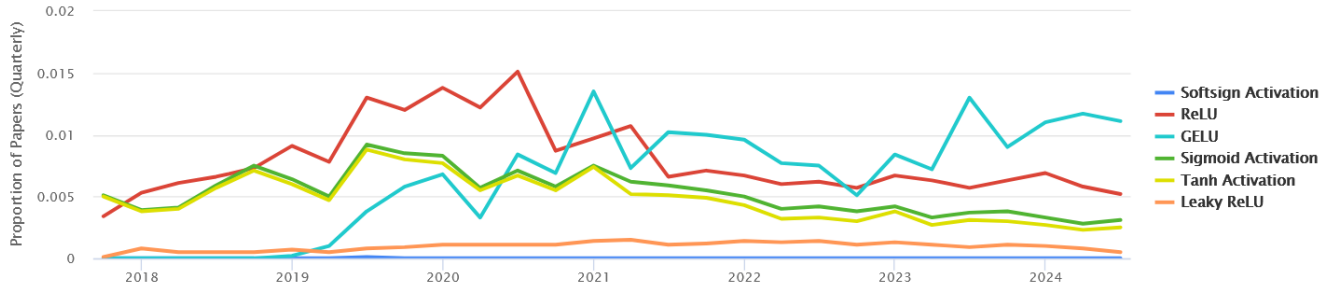


Рис. «Популярность» функций активации[2].

Функция активации также может быть реализована в виде модуля в Pytorch.

```
import torch
import torch.nn as nn

# Реализация ELU
class ELU(nn.Module):
    def __init__(self, alpha=1.0):
        super(ELU, self).__init__()
        self.alpha = alpha

    def forward(self, x):
        # Применяем ELU к каждому элементу тензора
        return torch.where(x > 0, x, self.alpha * (torch.exp(x) - 1))

# Пример использования
elu = ELU(alpha=1.0)
input_tensor = torch.tensor([-1.0, 0.0, 1.0, 2.0])
```

[1] [Juhstin Johnson] / https://arxiv.org/pdf/1710.05941.pdf

[2] https://paperswithcode.com/method/softsign-activation

```
output_tensor = elu(input_tensor)
print(output_tensor)
tensor([-0.6321,  0.0000,  1.0000,  2.0000])
```

Вопросы и задачи

1. Какой выход (выходной модуль) нейронной сети лучше использовать в задаче классификации, в которой каждый объект принадлежит ровно двум классам? Как обучать такую сеть (какую функцию ошибки использовать)?

2. Как реализовать любую булеву функцию с помощью нейросети, используя наименьшее число слоёв?

3. В записи нейронной сети FFN в виде суперпозиции

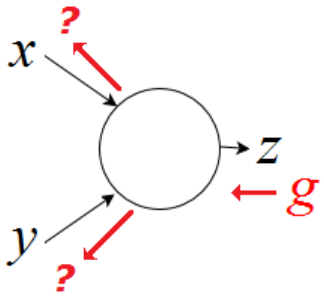
$$\varphi_K(W_K \cdot \ldots \cdot \varphi_2(W_2 \cdot \varphi_1(W_1 \cdot x))\ldots)$$

нет смещений в слоях. Корректно ли такое упущение?

4. Какое минимальное число нейронов достаточно для реализации XOR / эквивалентности (можно использовать нейросеть любой структуры)?

5. Напишите код на Pytorch, в котором по списку натуральных чисел создаётся нейросеть с соответствующим числом нейронов в каждом слое (список может быть любой длины).

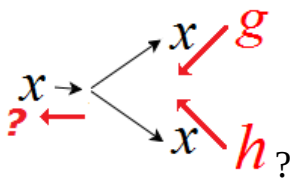
6. Что в таком подграфе графа вычислений передаётся при обратном проходе



если он реализует функцию:

- $f(x,y) = x + y$,
- $f(x,y) = x \cdot y$,
- $f(x,y) = \max(x,y)$?

Что передаётся для расщепления $f(x) = (x, x)$:



Итоги

- Нейросети – это
  - нелинейное обобщение линейных алгоритмов,
  - последовательное преобразование признакового пространства,
  - ансамбль алгоритмов,
  - суперпозиция «логистических регрессий[1]»,
  - граф вычислений.
- Нейросети обладают высокой функциональной выразимостью и являются универсальными аппроксиматорами[2].
- Нейросети обучаются градиентными методами. Далее ещё будем говорить про то, как делать эффективное обучение.

Спасибо за внимание к книге!
Замечания по содержанию, замеченные ошибки и неточности можно написать в телеграм-чате
https://t.me/Dyakonovsbook

[1] Если используется архитектура FFN и сигмоида в качестве активации на всех слоях.

[2] Рекомендуем почитать лучшую «классическую» книгу по DL: Ian Goodfellow, Yoshua Bengio, Aaron Courville «Deep Learning» http://www.deeplearningbook.org/
Также очень неплохую Simon J.D. Prince «Understanding Deep Learning» https://udlbook.github.io/udlbook/