

Московский государственный университет имени М. В. Ломоносова



Факультет Вычислительной Математики и Кибернетики
Кафедра Математических Методов Прогнозирования

**Эссе студентки 317 группы
«Нейронные сети»**

Выполнила:
студентка 3 курса 317 группы
Батшева Анастасия Васильевна

Москва, 2021

Содержание

1 Введение	2
2 Аннотация	3
3 Нейронные сети	4
3.1 Простейшая нейросеть	4
3.2 Функции активации	5
3.3 Функциональная выразимость нейрона	8
3.4 Теорема об универсальной аппроксимации	10
3.5 Сеть прямого распространения	11
3.6 Необходимость глубоких сетей	12
3.7 Обучение	12
3.8 Функции ошибки	14
3.9 Обратное распространение градиента	14
3.10 Нейросеть – вычислительный граф	15
3.11 Вычисление градиента на графике	16
3.12 Проблема затухания градиента	17
3.13 Производные на компьютере	18
4 Заключение	19
Список литературы	20

1 Введение

Идея искусственных нейронных сетей возникла ещё в начале 20 века. Пережив несколько взлетов и падений, настоящего пика популярности механический разум достиг только к нынешнему времени. Техническое оборудование, вычислительные мощности и ресурсы только сейчас позволили по-настоящему развернуться данному направлению. Нейронные сети, продолжая тему машинного обучения, работают с большими данными - колоссальными объемами информации, такой как потоки изображений, видео, числовых данных. Область применения технологии сетей охватывает такие задачи как прогнозирование, распознавание, выявление закономерностей, генерация данных, и многое другое. Более того, успех применения данной технологии настолько велик, что начиная с 2015 года она превзошла человека в точности распознавания изображений. Однако достижения сегодняшнего дня - только ступени к будущим открытиям и возможностям.

2 Аннотация

Данная статья является введением в тему нейросетей. Ознакомление с данной темой предполагает знание базовых терминов и алгоритмов машинного обучения, таких как обучаемая выборка, обучение с учителем, линейные алгоритмы классификации и регрессии и многое другое. В статье приводятся описания первых этапов создания нейросетей и процессов их обучения.

3 Нейронные сети

3.1 Простейшая нейросеть

Вспомним один из столпов машинного обучения - линейную регрессию. Линейная регрессия основана на аппроксимации истинной функции y некоторой линейной функцией от входных аргументов - векторов признакового описания:

$$y \simeq \langle w, x \rangle + b$$

$X, Y = (x_i, y_i)_{i=1}^N$ – обучающая выборка

$y \in Y \subset \mathbb{R}$ – целевая переменная

$x \in X \subset \mathbb{R}^n$ – вектор признаков

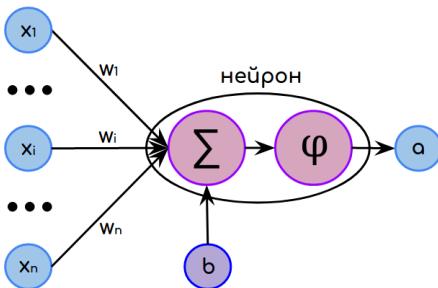
$w \in \mathbb{R}^n$ – вектор весов

$b \in \mathbb{R}^n$ – вектор смещений

$N \times n$ – размерность матрицы признаков X

Обозначим $a(x) = \varphi(\langle w, x \rangle + b)$.

Представим функцию a в виде схемы, которую будем называть **нейрон**.



Нейрон состоит из:

1. Набора входов x , которым приписаны веса w и смещения b
2. Сумматора $\langle w, x \rangle + b$
3. Функции выхода или активации φ

Тогда:

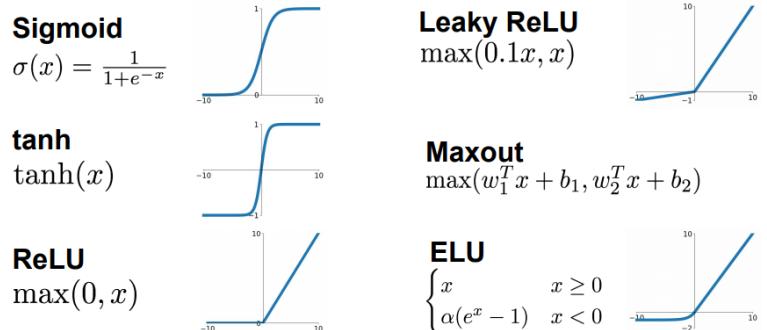
1. Если $\varphi(x) = x$, то $a(x)$ - **линейная регрессия**
2. Если $\varphi(x) = \begin{cases} 1 & x > \text{threshold} \\ 0 & x < \text{threshold} \end{cases}$, то $a(x)$ - **линейная классификация**
3. Если $\varphi(x) = \frac{1}{1+e^{-x}}$, то $a(x)$ - **логистическая регрессия**

3.2 Функции активации

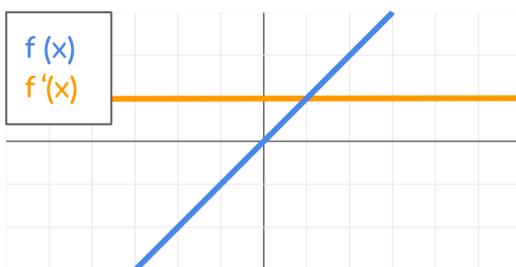
Применяя к одному и тому же нейрону различные функции активации, можно получать различные семейства моделей.

Приведём основные используемые функции активации:

1. Тождественная функция
2. Пороговая функция
3. Сигмоида
4. Гиперболический тангенс
5. ReLU



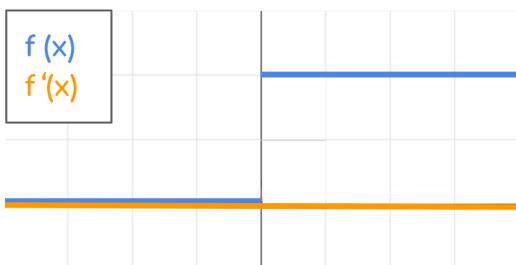
Тождественная функция (Linear Activation function):



$$\varphi(x) = x \Rightarrow \frac{\partial \varphi(x)}{\partial x} = 1$$

Легко проверить, что производная тождественной функции всюду равна единице.

Пороговая функция (Threshold function):

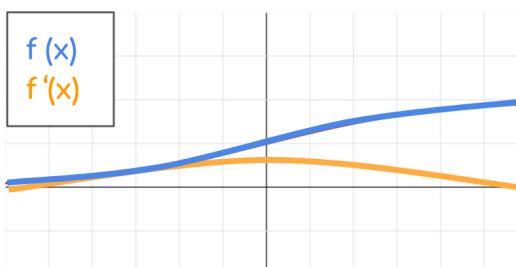


$$\varphi(x) = \mathbb{I}[x > 0] \Rightarrow \frac{\partial \varphi(x)}{\partial x} = 0$$

Пороговая функция выражается через индикатор: если входное значение лежит в заданном промежутке (в нашем случае требование - строгая положительность), индикатор принимает значение 1, иначе 0.

Для индикатора производная всюду равна нулю, и это плохое свойство для функции активации, поскольку без производных не получится обучать сеть градиентным спуском.

Сигмоида (Sigmoid):



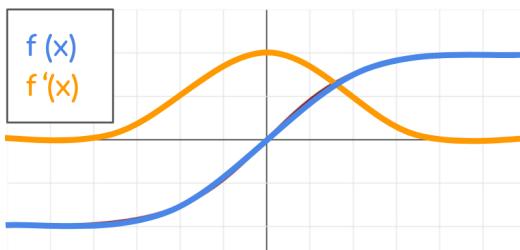
Замечательное свойство сигмоиды состоит в том, что её значение заключено в промежутке $(0, 1)$, а её производная выражается через саму функцию. Это позволяет существенно уменьшить затраты на вычисление градиента. При этом значения сигмоиды могут быть интерпретированы как сила выходного сигнала или же уверенность нейрона.

$$\varphi(x) = \frac{1}{1+e^{-x}} \in (0, 1) \Rightarrow \frac{\partial \varphi(x)}{\partial x} = \varphi(x)(1-\varphi(x)) > 0$$

Недостатки сигмоиды:

1. Слишком быстро уменьшаются градиенты
2. Выходы не отцентрированы
3. Вычисление экспоненты слишком дорого

Гиперболический тангенс (Tanh):



Гиперболический тангенс решает одну из проблем классической сигмоиды - у этой функции активации отцентрированы выходы, то есть значения функции лежат в интервале между -1 и 1 .

Однако гиперболический тангенс похож на сигмоиду, поскольку производная так же выражается через значение самой функции, а графики производных представляют собой колоколообразные функции.

$$\varphi(x) = \frac{2}{1 + e^{-2x}} - 1 = \frac{e^{+x} - e^{-x}}{e^{+x} + e^{-x}} = \frac{e^{+2x} - 1}{e^{+2x} + 1} \in (-1, 1) \Rightarrow \frac{\partial \varphi(x)}{\partial x} = 1 - \varphi^2(z)$$

Тем не менее, требуется заметить, что гиперболический тангенс не решает основную проблему сигмоиды - график его производной достаточно быстро выходит на свои асимптоты, и производные становятся близки к нулю.

Softmax:

В задачах классификации часто используются функции активации вида softmax: она зависит от n переменных и сумма её выходов всего равна 1, а сами выходы положительны, что позволяет интерпретировать их как вероятности.

$$\text{softmax}(x_1, x_2, \dots, x_n) = \frac{1}{\sum_{j=1}^k e^{x_j}} (e^{x_1}, e^{x_2}, \dots, e^{x_n})^T$$

Примеры применения softmax к векторам:

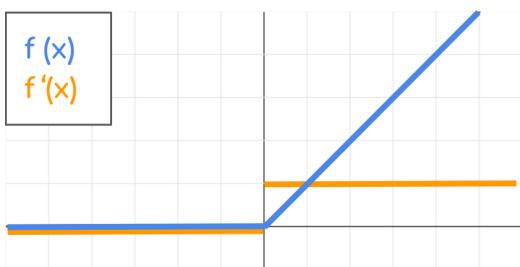
$$[0.5, 0.5, 0.1, \color{red}{0.7}] \rightarrow [0.257, 0.257, 0.172, \color{red}{0.314}]$$

$$[-1.0, 0, \color{red}{1.0}, 0, -1.0] \rightarrow [0.07, 0.18, \color{red}{0.5}, 0.18, 0.07]$$

$$[1.0, 1.0, 1.0, \color{red}{2.0}, 1.0] \rightarrow [0.15, 0.15, 0.15, \color{red}{0.4}, 0.15]$$

Функция активации сохраняет номер максимального элемента (то есть максимальный элемент в исходном векторе будет максимальным и в результирующем).

ReLU (Rectified Linear Unit):



Чтобы решить проблему затухания насыщенного сигнала, часто используют другие виды функций активации, например ReLU. ReLU совмещает в себе пороговую и тождественные функции активации:

$$\varphi(x) = \mathbb{I}(x > 0)x = \max(0, x) \Rightarrow \frac{\partial \varphi(x)}{\partial x} = \mathbb{I}(x > 0)$$

Сети, использующие ReLU работают быстрее, чем сети с сигмоидой или тангенсом, однако ReLU всё так же обладает недостатком в виде нулевой производной.

Существует несколько модификаций классического ReLU, решающих эту проблему:

1. **Leaky ReLU** = $\max(0.1x, x)$: Функция добавляет небольшую поправку, чтобы градиент в отрицательной области не обращался в ноль
2. **ELU (Exponential Linear Unit)** = $\begin{cases} x, & x \geq 0 \\ a(e^x - 1), & x < 0 \end{cases}$: Это улучшение меняет функцию градиента в отрицательной области, сглаживая её и убирая явный излом функции
3. **SELU (Scaled Exponential Linear Unit)** = $\lambda \cdot \text{ELU}(x)$: Добавляет шкалирование в ELU при помощи выбора подходящих коэффициентов λ
4. **GELU (Gaussian Error Linear Unit)** = $\frac{x}{2} \left(1 + \tanh \left(\sqrt{\frac{2}{\pi}} (x + ax^2) \right) \right)$
 $a = 0.44715$: Часто используется в архитектурах-трансформерах (например, Google's BERT или Open AI's GPT-2)

Maxout:

Пусть Maxout и не является в полной мере функцией активации из-за наличия параметризации, она является естественным обобщением ReLU-подобных функций и представляет собой операцию взятия максимума из двух линейных функций:

$$\text{Maxout}(x) = \max(w^T x + w_0, v^T x + v_0)$$

Из-за необходимости вводить большое число параметров, эту функцию активации применяют достаточно редко.

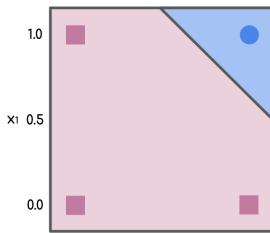
3.3 Функциональная выразимость нейрона

Но на что всё-таки способен один нейрон?

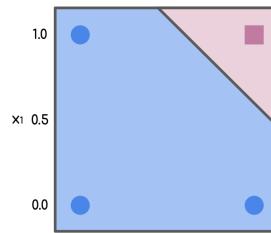
Для простоты будем использовать пороговую функцию активации. При всех её недостатках, с её помощью получиться просто выразить некоторые логические функции.

Выход нейрона с пороговой функцией активации - это 0 или 1.

Покажем, что можно выразить им логические функции "**И**"(слева) и "**ИЛИ**"(справа):



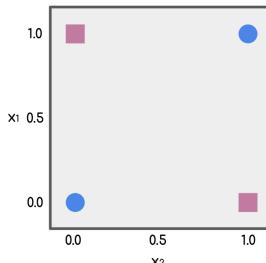
$$x_1 + x_2 > 1.5$$



$$x_1 + x_2 > 0.5$$

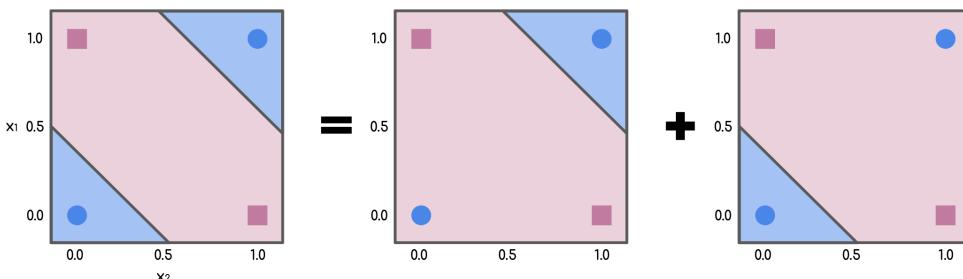
Заметим, что нейроны отличаются только значениями bias-ов: веса в обоих случаях равны [1, 1].

Как известно, чтобы два множества были разделимы гиперплоскостью, требуется, чтобы их выпуклые оболочки не пересекались. Этот критерий накладывает ограничение на множество логических функций, реализуемых одним нейроном: например, "Исключающее ИЛИ" одним нейроном уже реализовать не получится.

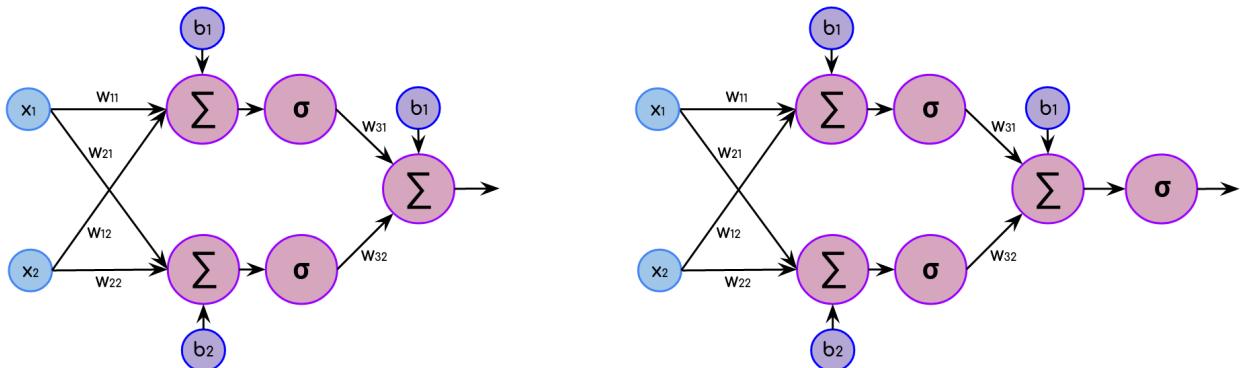


Однако можно сделать это при помощи суперпозиции из трёх нейронов:

$$\text{th}(\text{th}(x_1 + x_2 - 1.5) + \text{th}(-x_1 - x_2 + 0.5) - 0.5)$$



Такая суперпозиция может быть представлена в виде двухслойной нейронной сети:



Формула для регрессии (слева):

$$a = b_3 + w_{31}\sigma(b_1 + w_{11}x_1 + w_{12}x_2) + w_{32}\sigma(b_2 + w_{21}x_1 + w_{22}x_2)$$

Формула для классификации (справа):

$$a = \sigma(b_3 + w_{31}\sigma(b_1 + w_{11}x_1 + w_{12}x_2) + w_{32}\sigma(b_2 + w_{21}x_1 + w_{22}x_2))$$

При возможности менять число нейронов в первом слое такой нейронной сети хватит для выражения любых логических функций. Однако пороговая функция обладает множеством описанных выше недостатков: возможно ли использовать другую функцию активации, например, сигмоиду?

Рассмотрим сигмоиду с параметром C :

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

При увеличении параметра C такая сигмоида будет стремиться к пороговой функции, при уменьшении же решение будет всё более и более гладким в смысле графика. Нейросети, использующие сигмоиду вместо пороговой функции активации реализуют тот же класс функций с поправкой на то, что сигмоида принимает значение не в отрезке $[0, 1]$, а в интервале $(0, 1)$.

Можно выразить функциональность двухслойной сети с двумя нейронами на первом слое в виде матричного выражения:

$$\sigma \left[\begin{bmatrix} w_{31} & w_{32} & b_3 \end{bmatrix} \left[\sigma \left(\begin{bmatrix} w_{11} & w_{12} & b_1 \\ w_{21} & w_{22} & b_2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ 1 \end{bmatrix} \right) \right] \right],$$

Причём сигмоиду вычисляются покомпонентно.

3.4 Теорема об универсальной аппроксимации

Известна следующая теорема:

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

Позднее, Allan Pinkus доказал ещё более общий случай этой теоремы:

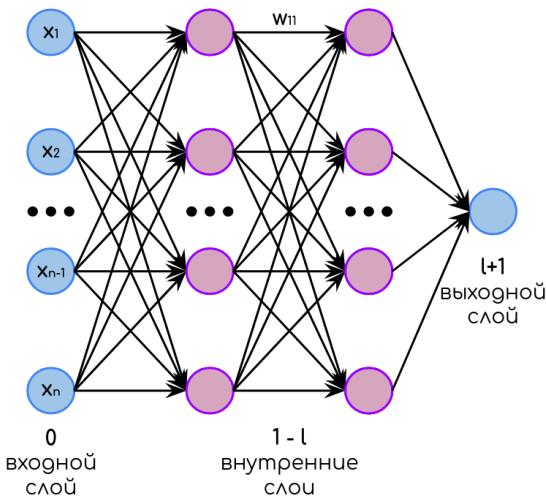
Нейросеть глубины два с фиксированной функцией активации в первом слое и линейной функцией активации во втором может равномерно аппроксимировать (возможно, при увеличении числа нейронов в первом слое) любую непрерывную функцию на компактном множестве тогда и только тогда, когда функция активации неполиномиальна.

Идею доказательства теоремы можно сформулировать так: если функция активации в скрытом слое представляет собой полином k -ой степени, то выходной слой будет представлять собой линейную комбинацию полиномов k -ой степени, то есть являться полиномом не выше, чем k -ой степени, что противоречит условию.

3.5 Сеть прямого распространения

Однако было показано, что при использовании всего двух слоев сети, необходимое число нейронов в этих слоях растет очень быстро. Чем больше размерность пространства признаков, тем больше нужно нейронов в слоях. Поэтому дальнейшее развитие нейросетей происходило за счет их углубления.

Глубокие сети подразумевают несколько (от 2 до $n \sim 200$) слоев. На рисунке 1 представлен простейший пример такой сети с $l = 2$ внутренними слоями. 2 крайних слоя иногда включаются в список слоев сети под номерами 0 и $l+1$.



Подобная сеть есть ничто иное как ориентированный граф. В качестве **вершин** выступают переменные или нейроны, в качестве **ребер** - зависимости с метками весов этих зависимостей w_{ij} .

Важно, что в простейшем случае в сети нет циклов и все нейроны предыдущего слоя связаны со всеми нейронами следующего слоя.

Тогда такая сеть называется **сетью прямого распространения FNN**.

В таком случае работа сети описывается формулой:

$$\varphi_{l+1}(W_{l+1} \cdot \varphi_l(W_{l-1} \cdot \dots \varphi_1(W_1 \cdot X)) \dots)$$

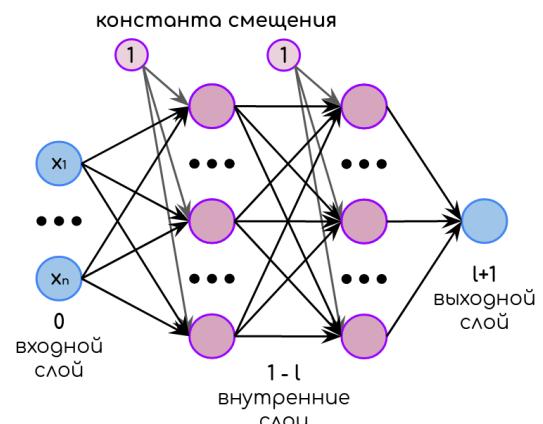
В сущности, **принцип действия сети заключается в последовательном преобразовании пространства признаков**. Каждый слой обрабатывает входящие в него данные (которые, в свою очередь, являются выходными данными предыдущего слоя) и отображает каждый объект из этих данных в новое пространство признаков. Поэтому действие сети эквивалентно суперпозиции линейных операторов, последовательно отображающих пространство признаков X с учетом функции активации:

$$X \xrightarrow{\varphi_1(W_1 \cdot X)} X_1 \xrightarrow{\varphi_2(W_2 \cdot X_1)} X_2 \dots \xrightarrow{\varphi_{l+1}(W_{l+1} \cdot X_l)} X_{l+1}$$

Однако в оригинальном прохождении через каждый нейрон слоя к результату добавляется смещение b , поэтому суперпозиция должна была бы выглядеть так:

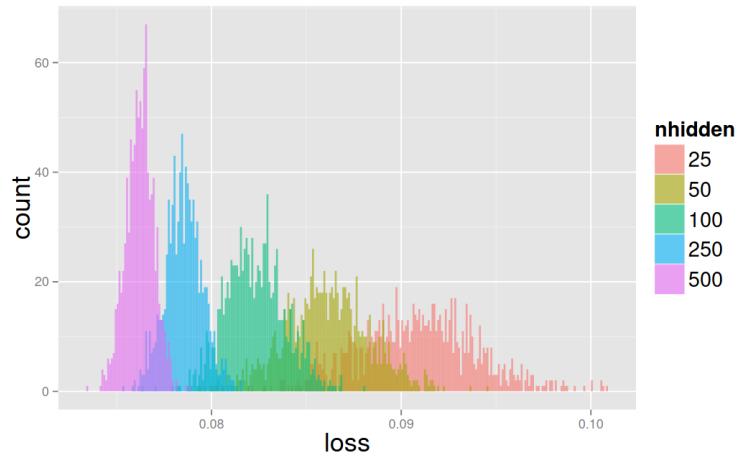
$$X \xrightarrow{\varphi_1(W_1 \cdot X + b_1)} X_1 \xrightarrow{\varphi_2(W_2 \cdot X_1 + b_2)} X_2 \dots \xrightarrow{\varphi_{l+1}(W_{l+1} \cdot X_l + b_{l+1})} X_{l+1}$$

Но на самом деле фиктивные единицы могут быть воссозданы сетью автоматически (например с помощью обнуления весов определенного нейрона), поэтому отсутствие констант в формуле композиции при правильной настройке сети может не привести к плохому результату.



3.6 Необходимость глубоких сетей

В качестве аргумента за использование глубоких сетей приведены результаты научной работы [1].



На данном графике изображены гистограммы распределения потерь для каждой группы экспериментов с нейронными сетями различной глубины.

Видно, что дисперсия уменьшается с увеличением размера сети. Причина заключается в том, что у небольшой сети риск попасть в локальный минимум вместо глобального выше, чем у большой сети.

3.7 Обучение

Классическим подходом к вычислению оптимальных параметров системы является минимизация эмпирического риска. Эмпирический риск - это средняя величина ошибки алгоритма на обучающей выборке $\{(x_i, y_i)\}_1^n$. Поэтому задача оптимизации при обучении алгоритма (сети) a по прецедентам, как в нашем случае, выглядит так:

$$Q(a, X, Y) = \frac{1}{n} \sum_{i=1}^n L(a(x_i, W), y_i) \rightarrow \min_W$$

W - параметры сети (веса).

Как известно, линейные операторы без регуляризации могут приводить к переобучению из-за очень больших коэффициентов (весов), чувствительных к малейшим изменениям входных данных. Поэтому к функционалу ошибки добавляется регуляризация $\lambda R(W)$ - Ridge, Lasso и прочее:

$$Q(a, X, Y) = \frac{1}{n} \sum_{i=1}^n L(a(x_i, W), y_i) + \lambda R(W) \rightarrow \min_W$$

Гиперпараметр лямбда отвечает за то, насколько сильно регуляризация будет влиять на модель.

Данная задача оптимизации невыпуклая, так как имеет локальные минимумы.

Решением данной задачи является **метод градиентного спуска** - то есть вычисление градиента по параметрам W - $\nabla Q(W)$ и итеративное приближение к минимуму функционала Q :

$$W^{i+1} = W^i - \eta \nabla Q(W^i)$$

η - параметр, называемый темпом алгоритма или скоростью обучения (learning rate).

Однако в большинстве реальных случаев расчёты на каждом шаге будут слишком медленными. Ведь обучающая выборка может содержать миллионы объектов, а сеть - миллионы настраиваемых параметров. Потому вычисление общего направления градиента - очень трудоемкая задача.

Более верным подходом будет использование **стохастического** градиентного спуска **SGD**. Отличие в том, что на каждом шаге потери и градиент вычисляются не по всему датасету, а лишь по небольшому подмножеству (возможно, из одного элемента) из датасета — мини-пакете или батче (batch):

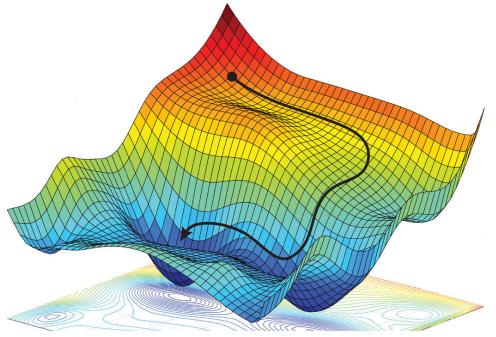
$$W^{t+1} = W^t - \eta \nabla Q(W^t)$$

W^t - некоторое подмножество обновляемых весов W .

Это значительно ускоряет процесс оптимизации. К тому же, как показывает практика, подобный подход (пускай и с некоторыми улучшениями) уменьшает вероятность остановки в локальном минимуме.

Метод стохастического градиента подразумевает следующие шаги:

1. Инициализация весов W .
2. Цикл по t до сходимости:
 - (a) Выбор случайного объекта или батча X_i
 - (b) Вычисление градиента для X_i : $\nabla Q(W^t)$
 - (c) Обновление весов: $W^{t+1} = W^t - \eta \nabla Q(W^t)$



Рассмотрим первый пункт алгоритма подробнее. Инициализация весов может быть различной: начальные значения можно положить равными нулю, единицам, случайным значениям из различных распределений и так далее. Правильной инициализацией считается выбор значений из нормального распределения $\mathcal{N}(0, \sigma^2)$, так разнообразие весов с большей вероятностью приведет к правильному ответу: недостатки плохо подобранных весов скомпенсируются успехом хорошо подобранных, в то время как нейроны с одинаковыми весами будут обучаться одинаково и в случае изначальной ошибки (неудачно подобранного значения весов) не дадут хорошего результата. Эта особенность проистекает из свойств ансамблевых алгоритмов.

3.8 Функции ошибки

Вернемся к функции ошибки L . Для задачи классификации наиболее логично в качестве L взять **логистическую ошибку**. Наиболее логично это по причине того, что логистическая ошибка напрямую следует из решения методом максимального правдоподобия [2]. Тогда **LogLoss** или **CrossEntropyLoss** для одного объекта (x, y) из обучающей выборки при многоклассовой классификации ($1 - k$) определен так:

$$L(a(x), y) = - \sum_{j=1}^k y \log a_j$$

Где a_j - j -компоненты выхода $a(x)$. Тогда для алгоритма a с $\varphi_{l+1}(x) = \text{softmax}(x)$ верно следующее:

$$L(a(x), y) = L(a_1, a_2, \dots, a_k, y) = -\log \frac{e^{a_y}}{\sum_{j=1}^k e^{a_j}} = -a_y + \log \sum_{j=1}^k e^{a_j}$$

Обратим внимание, что в зависимости от a_j экспоненты в сумме могут быть достаточно большими. Большие значения опасны выходами за допустимые пределы числовых типов данных в представлении компьютера. Для устранения этого эффекта применяется вычисление максимального элемента $a_{max} = \max(a_1, a_2, \dots, a_k)$.

$$L(a(x), y) = -\log \frac{e^{a_y - a_{max}}}{\sum_{j=1}^k e^{a_j - a_{max}}} = -a_y - a_{max} + \log \sum_{j=1}^k e^{a_j - a_{max}}$$

В таком случае все степени $a_j - a_{max} \leq 0 \Rightarrow$ все экспоненты $e^{a_j - a_{max}} \leq 1$.

Такой подход позволяет вычислять значения функции без опасения превышения лимита числовых данных.

3.9 Обратное распространение градиента

Данный параграф более детально рассматривает вычисление градиентов $\nabla Q(W^t)$ для SGD.

Из предыдущих пунктов имеем:

1. $a(W, X) = \varphi_{l+1}(W_{l+1} \cdot \varphi_l(W_{l-1} \cdot \dots \cdot \varphi_1(W_1 \cdot X)) \dots)$
2. $X \xrightarrow{\varphi_1(W_1 \cdot X)} X_1 \xrightarrow{\varphi_2(W_2 \cdot X_1)} X_2 \dots \xrightarrow{\varphi_{l+1}(W_{l+1} \cdot X_l)} X_{l+1}$
3. $Q(a, X, Y) \rightarrow \min_W$

Так как Q, a, φ - сложные функции, то по теореме о дифференцировании сложных функций можно провести такую цепочку рассуждений:

$$\begin{aligned} \boxed{\nabla Q(W) = \frac{\partial Q}{\partial a} \nabla a(W)} &\rightarrow \boxed{\nabla a(W) = \frac{\partial a}{\partial \varphi_{l+1}} \nabla \varphi_{l+1}(W)} \rightarrow \boxed{\nabla \varphi_{l+1}(W) = \frac{\partial \varphi_{l+1}}{\partial \varphi_l} \nabla \varphi_l(W)} \rightarrow \dots \\ \dots &\rightarrow \boxed{\nabla \varphi_2(W) = \frac{\partial \varphi_2}{\partial \varphi_1} \nabla \varphi_1(W)} \rightarrow \boxed{\nabla \varphi_1(W) = \langle W_1, dX \rangle} \end{aligned}$$

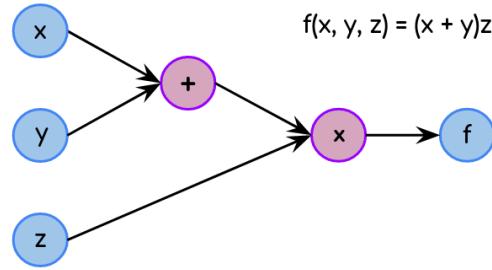
Суть данной последовательности рассуждений заключается в том, что каждый слой нейросети, начиная с последнего, вносит свой вклад в подсчет градиента, поскольку каждый слой оперирует выходом суперпозиции предыдущих слоев. Соответственно алгоритм рекурсивно использует правило дифференцирования сложной функции для вычисления градиента и потому носит название **обратного распространения градиента**. **Обратное распространение градиента = SGD + дифференцирование сложных функций**.

3.10 Нейросеть – вычислительный граф

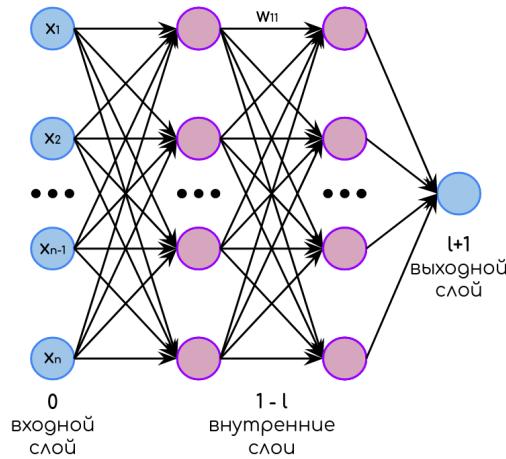
Интерпретировать и визуализировать полученный выше результат можно с помощью вычислительных графов.

Вычислительный график — это иллюстрированная запись какой-либо функции, состоящая из вершин и рёбер. Вершины (иногда их ещё называют узлы) — это вычислительные операции, которые необходимо выполнить, а рёбра связывают их в определённую последовательность [3].

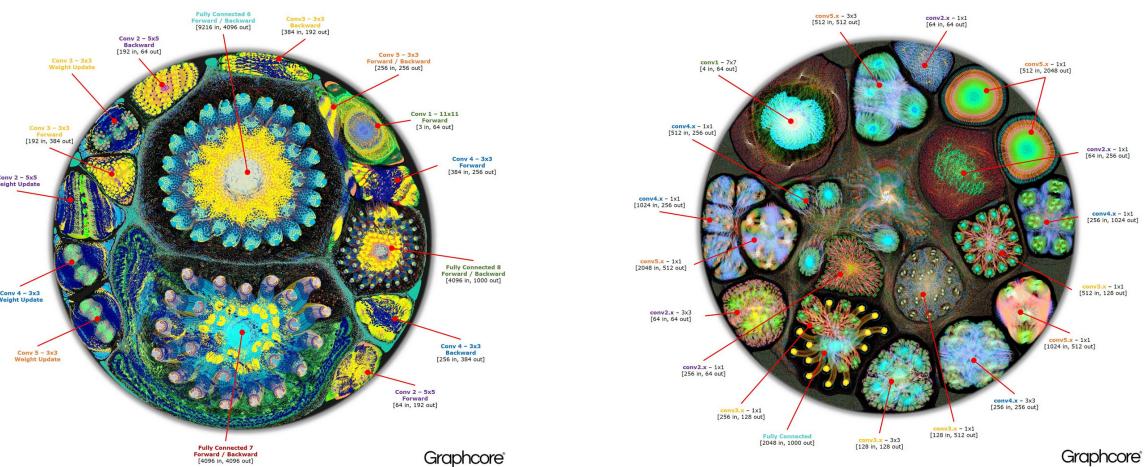
В качестве простейшего примера можно рассмотреть вычислительный график функции $f(x, y, z) = (x + y) \times z$:



Вычислительный график нейросети уже был приведен в начале работы, но рассмотрим его еще раз:



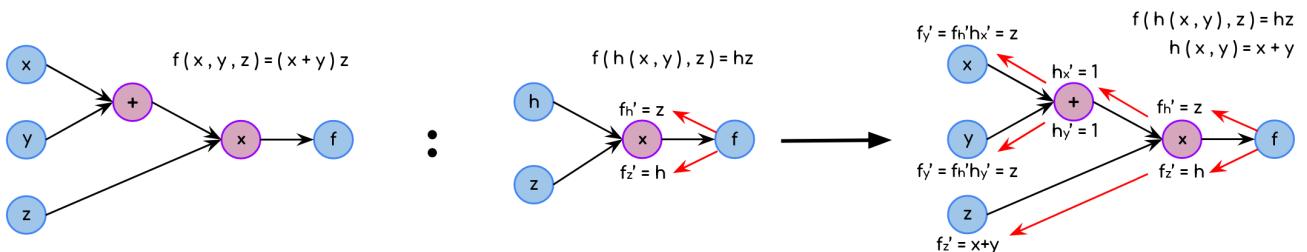
Чуть более сложные примеры графов для сетей AlexNet (слева) и ResNet-50 (справа) [4]. В этих сетях миллиарды параметров, кластеризованных в зависимости от расположения в сети.



3.11 Вычисление градиента на графе

Вычисление градиента по графу, в сущности, основано на том же принципе, что и обратное распространение градиента. Действительно, каждая вершина графа отвечает какому-то из нейронов, на вход которому подается выход суперпозиции предыдущих слоев или, в терминах графа, значения в вершинах ребер, входящих в этот график.

Наглядный пример вычисления градиента на графике представлен на следующем рисунке. В качестве изучаемой функции взята функция из предыдущего параграфа $f(x, y, z) = (x + y)z$.



Значение производной для узла, который следует за текущим, называется **восходящий градиент**, а для предыдущего узла — **локальный градиент**. Например, для $(+)$ восходящим градиентом является производная $f'_h = \frac{df}{dh}$, а локальным — производные $f'_x = \frac{df}{dx}$ и $f'_y = \frac{df}{dy}$.

Реализация алгоритма подобных вычислений представлена в модуле autograd библиотеки pytorch. torch.autograd — это механизм автоматической дифференциации PyTorch, который обеспечивает обучение нейронной сети [5].

Пример работы функций из данного модуля для функции $Q(x, y) = 3x^3 + y^2$:

```
import torch

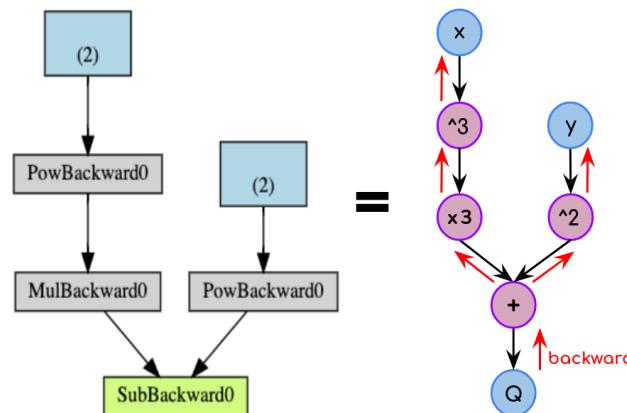
x = torch.tensor([2., 3.], requires_grad=True)
y = torch.tensor([6., 5.], requires_grad=True)

Q = 3*x**3 + y**2

external_grad = torch.tensor([1., 1.])
Q.backward(gradient=external_grad)

x: расчет autograd: [36.0, 81.0] истинное значение: [36.0, 81.0] соответствие: [True, True]
y: расчет autograd: [12.0, 10.0] истинное значение: [12.0, 10.0] соответствие: [True, True]
```

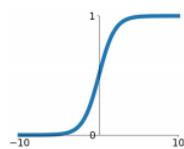
Вычислительный график данной функции:



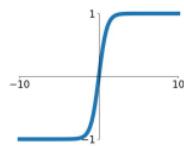
3.12 Проблема затухания градиента

Вспомним наиболее популярные функции активации φ : [3]

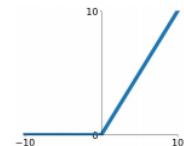
Sigmoid
 $\sigma(x) = \frac{1}{1+e^{-x}}$



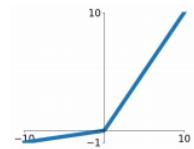
tanh
 $\tanh(x)$



ReLU
 $\max(0, x)$



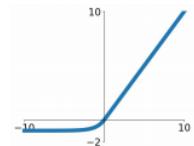
Leaky ReLU
 $\max(0.1x, x)$



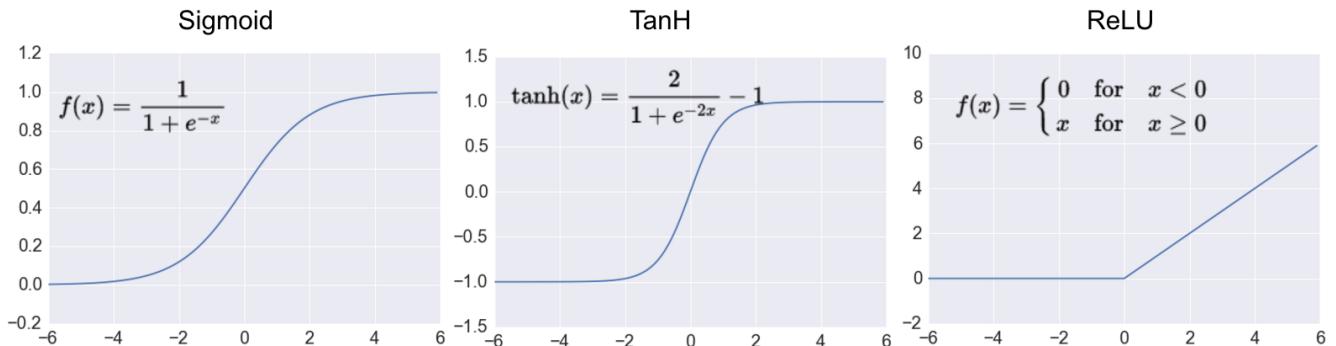
Maxout
 $\max(w_1^T x + b_1, w_2^T x + b_2)$

ELU

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$

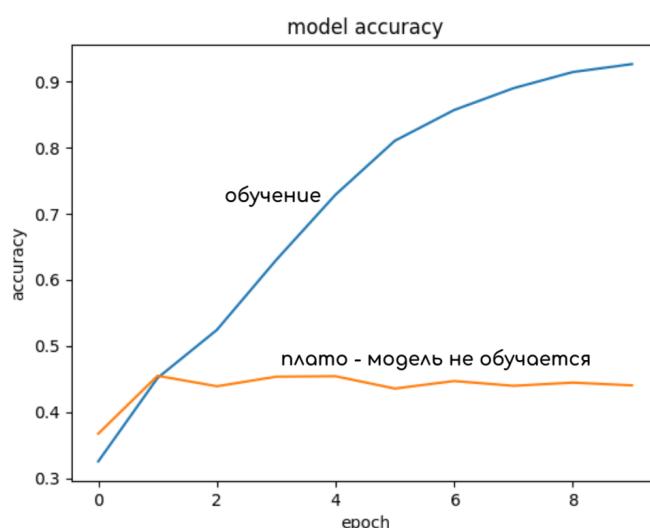


Обратим внимание на те функции, которые содержат практически горизонтальную часть: sigmoid, tanh, ReLU [6]:



Все эти функции объединяет общая проблема, которая носит название **затухание градиента**. Из-за наличия у них почти горизонтальных частей: область насыщенного сигнала у sigmoid и tanh и область отрицательных x у ReLU, - градиент в этих областях практически нулевой (в определенный момент машинной точности не хватит и градиент станет нулевым). Тогда во все последующие узлы в обратном распространении будут передаваться нулевые производные, что и **убьет** градиентный поток. Смерть потока приведет к остановке обучения, так как веса перестанут обновляться.

Пример выхода на плато:



К другим популярным проблемам функций активации относятся:

1. Нецентрированность нулем
2. Затратность вычисления экспоненты

Нецентрированность нулем приводит к такой проблеме: пусть исходные данные полностью положительны, тогда все градиенты во время обратного распространения будут одного знака. Это приведёт к тому, что веса при обновлении также будут одновременно либо увеличены, либо уменьшены, и градиентный поток станет зигзагообразным, испытывая лишние колебания вокруг нужного вектора направления.

Затратность вычисления экспоненты актуальна для функций sigmoid и ELU. В сравнении с вычислениями скалярных произведений это не самая большая проблема, однако так или иначе она добавляет больше вычислительной нагрузки, нежели остальные функции активации.

3.13 Производные на компьютере

Основная информация для данного параграфа была взята из работы [7].

Существуют 4 основных способа вычисления производных для решения описанных выше задач:

1. Аналитический
2. Численный
3. Символьный
4. Алгоритмический

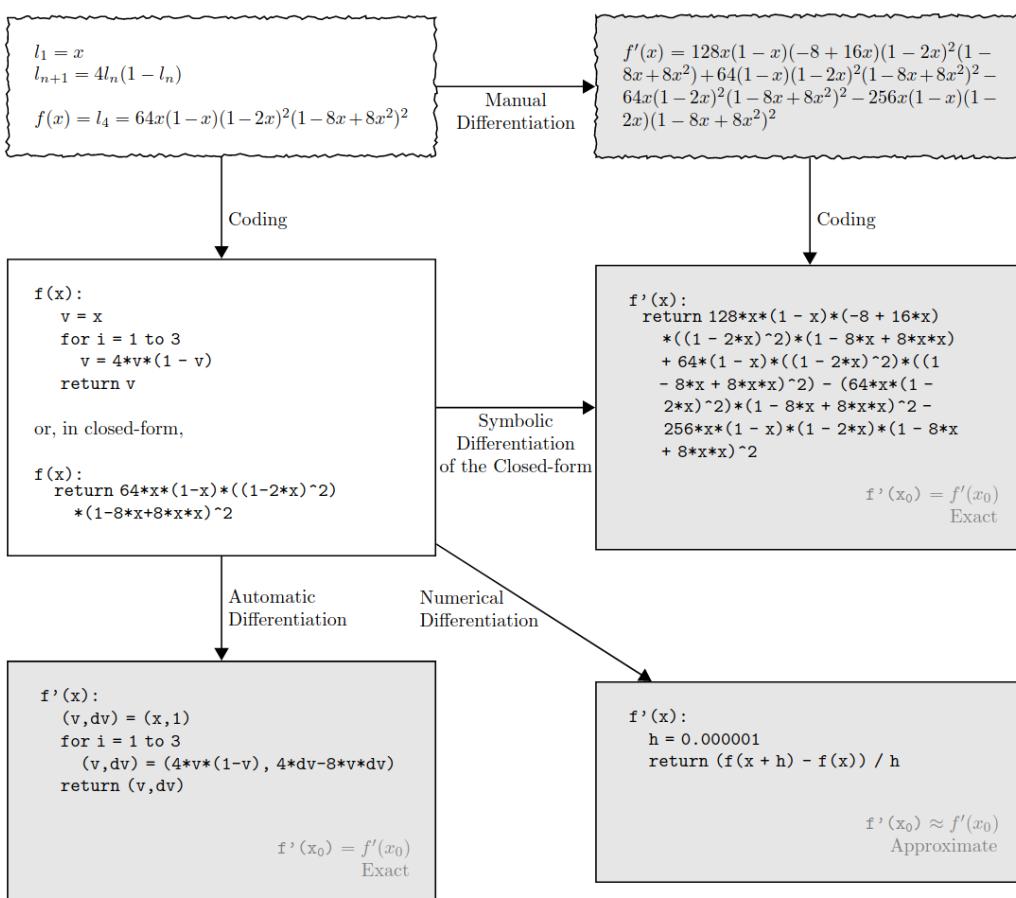
Аналитический метод подразумевает наличие специальной функции $g(x)$, отвечающей производной исходной функции $f(x)$ по x . Тогда вычисление производной в данной точке x^* сводится к вычислению значения $g(x^*)$. Например, если имеются две функции: $f(x) = x^2 + 2x + 1$ и $g(x) = 2x + 2$, где функция g является производной функции $f(x)$ ($g(x) = \frac{df(x)}{dx}$), тогда значение производной в точке $x = 3$ равно $g(3) = 6 + 2 = 8$.

Численный метод предполагает аппроксимацию производных конечными разностями с использованием значений исходной функции, оцененных в некоторых точках выборки. В своей простейшей форме он основан на предельном определении производной. Например, для функции из верхнего примера $f(x) = x^2 + 2x + 1$ производная может в точке x^* быть оценена как $\frac{df(x)}{dx}(x^*) = \frac{f(x^*+h)-f(x^*)}{h}$, где $h > 0$ - небольшой размер шага.

Символьный метод - это автоматическая манипуляция выражениями для получения производных, выполняемая путем применения преобразований, представляющих правила дифференцирования, такие как $\frac{d(f(x)+g(x))}{dx} = \frac{df(x)}{dx} + \frac{dg(x)}{dx}$ и $\frac{d(f(x)g(x))}{dx} = \frac{df(x)}{dx}g(x) + \frac{dg(x)}{dx}f(x)$. Формулы представлены как структуры данных, символически различающие дерево выражений. Тогда вычисление представляет собой совершенно механистический процесс. Это реализовано в современных системах компьютерной алгебры, таких как Mathematica, Maxima и Maple.

Алгоритмический подход основан на использовании таких структур как, например, вычислительные графы. Исходная функция представляется в виде графа с узлами - подфункциями или переменными и ребрами - выражениями, связывающими эти подфункции. Вычисление производной некой функции сводится к последовательному вычислению производных в узлах графа. Вычисление ведется в обратном направлении. Пример работы таких алгоритмов был приведен выше в разделе **Вычисление градиента на графике**.

Иллюстрация, демонстрирующая различные подходы: - В центре справа: Символьное вычисление. Дает точные результаты, но требует ввода закрытой формы и страдает от чрезмерного увеличения экспрессии - Внизу справа: Численное вычисление. Имеет проблемы с точностью из-за ошибок округления и усечения - Внизу слева: Автоматическое вычисление. Так же точно, как и символьное дифференцирование, только с постоянным коэффициентом накладных расходов и поддержкой потока управления



4 Заключение

В данной статье были рассмотрены основные принципы нейронных сетей. Подробно описано устройство основных элементов сети - нейронов, показано, как нейроны собираются в слои и зачем эти слои необходимы. Приведены ссылки на несколько полезных при ознакомлении с темой работ. Так же описаны процессы обучения и настройки. Предоставлено множество иллюстраций для лучшего понимания.

Список литературы

- [1] Anna Choromanska, Mikael Henaff, Michael Mathieu, Gérard Ben Arous, Yann LeCun «The Loss Surfaces of Multilayer Networks» 2015
- [2] А.Г.Дьяконов «Логистическая функция ошибки» 2018
- [3] Стэнфордский курс: лекция 4. «Введение в нейронные сети» 2019
- [4] «Как выглядят глубокие нейронные сети и почему они требуют так много памяти» 2017
- [5] «A Gentle Introduction to torch.autograd»
- [6] «Применение сверточных нейронных сетей для задач NLP» 2018
- [7] «Atilim Gunes Baydin, Barak A. Pearlmutter, Alexey Andreyevich Radul, Jeffrey Mark Siskind «Automatic differentiation in machine learning: a survey» 2015-2018