Научно-исследовательская работа
2 курс, 1 семестр

Тема курсовой работы: задача распознавания образов

Гусев Сергей Александрович 21Б13-мм

Отчет по учебной практике

Научный руководитель:
Старший научный сотрудник кафедры прикладной кибернетики Мокаев Руслан Назирович

Тема курсовой работы: ”Задача распознавания образов”.

Выполнил: Гусев Сергей Александрович, 21Б13-мм

Научный руководитель: Старший научный сотрудник кафедры прикладной кибернетики Руслан Назирович Мокаев

Распознавание образов – это научная дисциплина, целью которой является классификация объектов по нескольким категориям или классам. Объекты называются образами.

Классификация основывается на прецедентах.

Прецедент – это образ, правильная классификация которого известна.

Прецедент – ранее классифицированный объект, принимаемый как образец при решении задач классификации. Идея принятия решений на основе прецедентности – основополагающая в естественно-научном мировоззрении.

Будем считать, что все объекты или явления разбиты на конечное число классов. Для каждого класса известно и изучено конечное число объектов – прецедентов. Задача распознавания образов состоит в том, чтобы отнести новый распознаваемый объект к какому-либо классу.

Задача распознавания образов является основной в большинстве интеллектуальных систем. Рассмотрим примеры интеллектуальных компьютерных систем.

1. Машинное зрение. Это системы, назначение которых состоит в получении изображения через камеру и составление его описания в символьном виде (какие объекты присутствуют, в каком взаимном отношении находятся и т.д.).
2. Символьное распознавание – это распознавание букв или цифр.
3. Диагностика в медицине.
4. Геология.
5. Распознавание речи.
6. в дактилоскопии (отпечатки пальцев), распознавание лица, подписи, жестов.

В теме курсовой рассмотрим следующие 4 основных метода распознавания образов:

1. Алгоритм k-средних (k-means)
2. Алгоритм максимина (maxmin)
3. Метод персептрона
4. Алгоритм классификации на основе нейронной сети

В 1-ой части курсовой рассмотрим принципы работы каждого из алгоритмов, включая их математические модели.

Во 2 части курсовой применим эти алгоритмы и сравним их эффективность между собой. В качестве задачи рассмотрим символьное распознавание на наборе рукописных цифр в датасете MNIST.

Часть 1.1 Алгоритм k-средних (k-means)

Алгоритм k-средних – это метод кластерного анализа, цель которого является разделение m наблюдений (из пространства R^n) на k кластеров, при этом каждое наблюдение относится к тому кластеру, к центру (центроиду) которого оно ближе всего.

В качестве меры близости используется Евклидово расстояние:

$$p(x,y) = ||x-y|| = \sqrt{\sum_{p=1}^{n}(x_p - y_p)^2},\ x,\ y \in R^n$$

Итак, рассмотрим ряд наблюдений $(x^{(1)}, x^{(2)}, \dots, x^{(m)}),\ x^{(j)} \in R^n$

Метод k-средних разделяет m наблюдений на k групп (или кластеров) (k ≤ m) $S = \{S_1, S_2, \dots, S_k\}$, чтобы минимизировать суммарное квадратичное отклонение точек кластеров от центроидов этих кластеров:
$min\left[\sum_{i=1}^{k}\sum_{x^{(j)}\in S_i}||x^{(j)} - \mu_i||^2\right]$, где $x^{(j)} \in R^n,\ \mu_i \in R^n$

μ_i - центроид для кластера S_i

Алгоритм.

Итак, если мера близости до центроида определена, то разбиение объектов на кластеры сводится к определению центроидов этих кластеров. Число кластеров k задается исследователем заранее.

Рассмотрим первоначальный набор k средних (центроидов) $\mu_1, \mu_2, \dots, \mu_k$ в кластерах $S_1, S_2, \dots, S_k$. На первом этапе центроиды кластеров выбираются случайно или по определенному правилу (например, выбрать центроиды, максимизирующие начальные расстояния между кластерами).

Относим наблюдения к тем кластерам, чье среднее (центроид) к ним ближе всего. Каждое наблюдение принадлежит только к одному кластеру, даже если его можно отнести к двум и более кластерам.

Затем центроид каждого i-го кластера перевычисляется по следующему правилу: $\mu_i = \frac{1}{S_i} \sum_{x^{(j)} \in S_i} x^{(j)}$

Таким образом, алгоритм k-средних заключается в перевычислении на каждом шаге центроида для каждого кластера, полученного на предыдущем шаге.

Алгоритм останавливается, когда значения μ_i не меняются: $\mu_i^{\text{шаг } t} = \mu_i^{\text{шаг } t+1}$

Важно: Неправильный выбор первоначального числа кластеров k может привести к некорректным результатам. Именно поэтому при использовании метода k-средних важно сначала провести проверку подходящего числа кластеров для данного набора данных.

Итак, еще раз подчеркнем некоторые особенности метода k-средних:

1. В качестве метрики используется Евклидово расстояние
2. Число кластеров заранее не известно и выбирается исследователем заранее
3. Качество кластеризации зависит от первоначального разбиения

Проблемы k-средних:

1. Не гарантируется достижение глобального минимума суммарного квадратичного отклонения V, а только одного из локальных минимумов.
2. Результат зависит от выбора исходных центров кластеров, их оптимальный выбор неизвестен.
3. Число кластеров надо знать заранее.

Прочая информация:

1. Вычислительная сложность - O(nkl), где k – число кластеров, l – число итераций
2. Форма кластеров - гиперсфера
3. Входные данные - число кластеров
4. Результат - центры кластеров

Часть 1.2 Алгоритм максимина $(\mathbf{maxmin})$

Метод максимина предназначен для разделения объектов на кластеры, причем количество кластеров заранее неизвестно; оно определяется автоматически в процессе разбиения объектов.

Принцип работы метода следующий. Выбирается один из объектов (любой); он становится прототипом первого кластера. Находится объект, наиболее удаленный от выбранного; он становится прототипом второго кластера. Все объекты распределяются по двум кластерам; каждый объект относится к кластеру, представленному ближайшим прототипом. Затем в каждом из кластеров находится объект, наиболее удаленный от своего прототипа. Если расстояние между этим объектом и прототипом кластера оказывается значительным (превышающим некоторую предельную величину), то объект становится новым прототипом, т.е. образуется новый кластер. После этого распределение объектов по кластерам выполняется заново. Процесс продолжается, пока не будет получено такое разбиение на кластеры, при котором расстояние от каждого объекта до прототипа кластера не будет превышать заданную предельную величину. Приведем пошаговый алгоритм реализации метода максимина.

1. Выбирается любой из объектов, например, первый в списке объектов (X_1). Он становится прототипом первого кластера: $P_1 = X_1$. Количество кластеров принимается равным единице: K = 1.
2. Определяются расстояния от объекта P_1 до всех остальных объектов: $D(P_1, \ldots, X_j),\ \ j = 1, \ldots, N$.
3. Определяется объект, наиболее удаленный от P_1, т.е. объект X_f, для которого выполняется условие: $D(P_1, X_f) = max_j D(P_1, X_j)$. Этот объект становится прототипом второго кластера: $P_2 = X_f$. Количество кластеров принимается равным двум: $K = 2$.
4. Определяется пороговое расстояние. Оно принимается равным половине расстояния между прототипами P_1 и P_2: $T = \frac{D(P_1, P_2)}{2}$. Эта величина будет использоваться для проверки условия окончания алгоритма.
5. Находятся расстояния от каждого из анализируемых объектов до каждого из имеющихся объектов-прототипов. Выполняется отнесение каждого объекта к ближайшему кластеру, т.е. кластеру, для которого расстояние между этим объектом и прототипом кластера минимально.
6. В каждом кластере определяется объект, наиболее удаленный от прототипа своего кластера. Обозначим эти объекты как Y_k, $k = 1, \ldots, K$ (здесь k – номер кластера, K – количество кластеров).
7. Для каждого из наиболее удаленных объектов, найденных на шаге 7, проверяется условие: $D(P_k, Y_k) < T,\ k = 1, \ldots, K$. Если это условие выполняется для всех кластеров, то алгоритм завершается. Если для некоторого объекта Y_k это условие не выполняется, то он становится прототипом нового кластера, и количество кластеров увеличивается на единицу ($K = K + 1$). В результате этого шага количество кластеров K увеличивается на число, равное количеству новых кластеров.

8. Находится новое пороговое расстояние. Оно определяется как половина среднего арифметического всех расстояний между прототипами:
$$T = \frac{\sum_{i=1}^{K-1} \sum_{j=i+1}^{K} D(P_i, P_j)}{K * (K-1)}$$
9. Выполняется возврат к шагу 6.

Таким образом, окончательным является разбиение, для которого во всех кластерах расстояние от прототипа кластера до каждого из объектов, входящих в этот кластер (даже до самого удаленного), не превышает некоторой предельной величины (порогового расстояния).

Часть 1.3 Метод персептрона

Персептрон — простейший вид нейронных сетей. В основе лежит математическая модель восприятия информации мозгом, состоящая из сенсоров, ассоциативных и реагирующих элементов. Предлагаю для начала разобрать структуру нейронной сети как таковую (а в части 1.4 разберем поподробнее активационные функции, обучение нейронной сети методом обратного распространения ошибки, рекомендации обучения и метрики оценки качества)

Хорошим примером биологической нейронной сети является человеческий мозг. Наш мозг — сложнейшая биологическая нейронная сеть, которая принимает информацию от органов чувств и каким-то образом ее обрабатывает (узнавание лиц, возникновение ощущений и т.д.). Мозг же, в свою очередь, состоит из нейронов, взаимодействующих между собой.

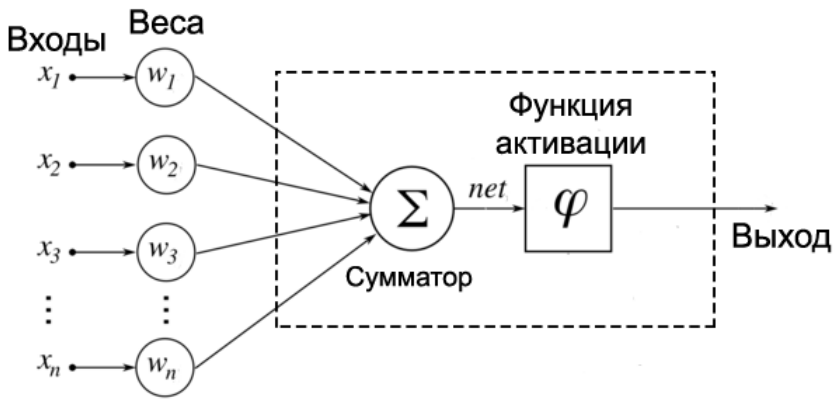
Для построения искусственной нейронной сети будем использовать ту же структуру. Как и биологическая нейронная сеть, искусственная состоит из нейронов, взаимодействующих между собой, однако представляет собой упрощенную модель. Так, например, искусственный нейрон, из которых состоит ИНС, имеет намного более простую структуру: у него есть несколько входов, на которых он принимает различные сигналы, преобразует их и передает другим нейронам. Другими словами, искусственный нейрон — это такая функция $R^n \to R$, которая преобразует несколько входных параметров в один выходной.

Как видно на рисунке ниже, у нейрона есть n входов x_i, у каждого из которого есть вес w_i, на который умножается сигнал, проходящий по связи. После этого взвешенные сигналы $x_i * w_i$ направляются в сумматор, который аггрегирует все сигналы во взвешенную сумму. Эту сумму также называют net. Таким образом, $net = \sum_{i=1}^{i=n} w_i * x_i = w^T * x$

Просто так передавать взвешенную сумму net на выход достаточно бессмысленно — нейрон должен ее как-то обработать и сформировать адекватный выходной сигнал. Для этих целей используют функцию активации, которая преобразует взвешенную сумму в какое-то число, которое и будет являться выходом нейрона. Функция активации обозначается $\phi(net)$. Таким образом, выходов искусственного нейрона является $\phi(net)$.

Для разных типов нейронов используют самые разные функции активации, но одними из самых популярных являются:

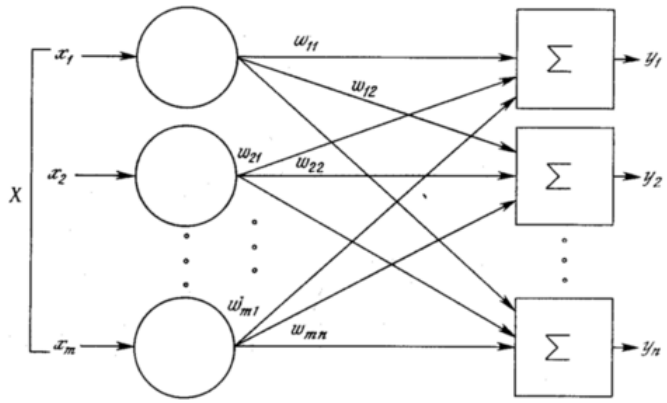
1. Функция единичного скачка. Если $net > threshold, \phi(net) = 1$, а иначе 0;
2. Сигмоиданльая функция. $\phi(net) = \frac{1}{1+exp(-a*net)}$, где параметр a характеризует степень крутизны функции;
3. Гиперболический тангенс. $\phi(net) = tanh(\frac{net}{a})$, где параметр a также определяет степень крутизны графика функции;



Разберем, какие виды нейронных сетей бывают:

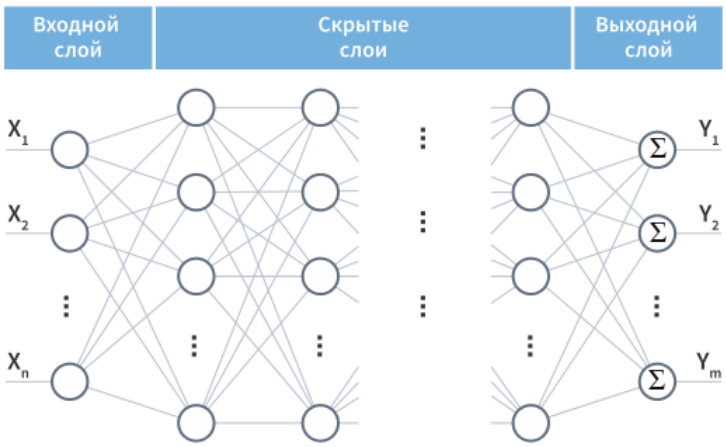
1. Однослойная нейронная сеть (англ. Single-layer neural network) — сеть, в которой сигналы от входного слоя сразу подаются на выходной слой, который и преобразует сигнал и сразу же выдает ответ.

Как видно из схемы однослойной нейронной сети, представленной ниже, сигналы $x_1, x_2, \ldots, x_n$ поступают на входной слой (который не считается за слой нейронной сети), а затем сигналы распределяются на выходной слой обычных нейронов. На каждом ребре от нейрона входного слоя к нейрону выходного слоя написано число — вес соответствующей связи.



2. Многослойная нейронная сеть (англ. Multilayer neural network) — нейронная сеть, состоящая из входного, выходного и расположенного(ых) между ними одного (нескольких) скрытых слоев нейронов.

Помимо входного и выходного слоев эти нейронные сети содержат промежуточные, скрытые слои. Такие сети обладают гораздо большими возможностями, чем однослойные нейронные сети, однако методы обучения нейронов скрытого слоя были разработаны относительно недавно.



3. Сети с обратными связями - их мы рассматривать не будем, т.к. в задачах кластеризации и распознавания достаточно использовать сети прямого распространения, в которых сигнал распространяется строго от входного слоя к выходному, но никогда в обратном направлении.

Обучение нейронной сети заключается в поиске такого набора весовых коэффицентов, при котором входной сигнал после прохода по сети преобразоуется в нужный нам выходной. В обучении используется несколько входных сигналов, т.к. ждем от сети способности обощать какие-то признакии и решать задачу на различных входных данных.

Обучение сети проходит на обучающей выборке - конечном наборе входных сигналов (иногда вместе с правильными выходными сигналами), по которым происходит обучение сети.

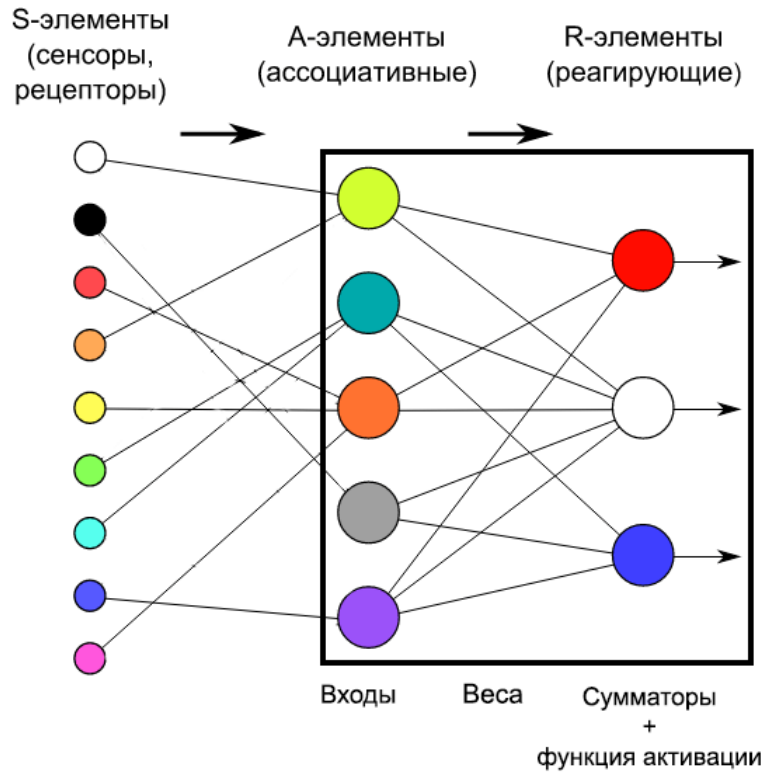
После обучения, но перед ее практическим применением, производят оценку качества работы нейронной сети на тестовой выборке - конечном наборе входных сигналов (иногда вместе с правильными выходными сигналами), по которым происходит оценка качества работы сети.

Наконец, перейдем к понятию персептрона.

Персептрон (англ. Perceptron) — простейший вид нейронных сетей.

В основе персептрона лежит математическая модель восприятия информации мозгом. Разные исследователи по-разному его определяют. В самом общем своем виде (как его описывал Розенблатт) он представляет систему из элементов трех разных типов:

сенсоров, ассоциативных элементов и реагирующих элементов.



Принцип работы персептрона следующий:

1. Первыми в работу включаются S-элементы. Они могут находиться либо в состоянии покоя (сигнал равен 0), либо в состоянии возбуждения (сигнал равен 1);
2. Далее сигналы от S-элементов передаются A-элементам по так называемым S-A связям. Эти связи могут иметь веса, равные только -1, 0 или 1;
3. Затем сигналы от сенсорных элементов, прошедших по S-A связям, попадают в A-элементы, которые еще называют ассоциативными элементами;

- Одному A-элементу может соответствовать несколько S-элементов;
- Если сигналы, поступившие на A-элемент, в совокупности превышают некоторый его порог θ, то этот A-элемент возбуждается и выдает сигнал, равный 1;
- В противном случае (сигнал от S-элементов не превысил порога A-элемента), генерируется нулевой сигнал;

4. Далее сигналы, которые произвели возбужденные A-элементы, направляются к сумматору (R-элемент), действие которого нам уже известно. Однако, чтобы добраться до R-элемента, они проходят по A-R связям, у которых тоже есть веса (которые уже могут принимать любые значения, в отличие от S-A связей);
5. R-элемент складывает друг с другом взвешенные сигналы от A-элементов, а затем

- если превышен определенный порог, генерирует выходной сигнал, равный 1;
- если порог не превышен, то выход персептрона равен -1.

Для элементов персептрона используют следующие названия: S-элементы называют сенсорами; A-элементы называют ассоциативными; R-элементы называют реагирующими.

Классификация персептронов

Персептрон с одним скрытым слоем (элементарный персептрон, англ. elementary perceptron) — персептрон, у которого имеется только по одному слою S, A и R элементов.

Однослойный персептрон (англ. Single-layer perceptron) — персептрон, каждый S-элемент которого однозначно соответствует одному A-элементу, S-A связи всегда имеют вес 1, а порог любого A-элемента равен 1. Часть однослойного персептрона соответствует модели искусственного нейрона.

Его ключевая особенность состоит в том, что каждый S-элемент однозначно соответствует одному A-элементу, все S-A связи имеют вес, равный +1, а порог A элементов равен 1. Часть однослойного персептрона, не содержащая входы, соответствует искусственному нейрону, как показано на картинке. Таким образом, однослойный персептрон — это искусственный нейрон, который на вход принимает только 0 и 1.

Однослойный персептрон также может быть и элементарным персептроном, у которого только по одному слою S,A,R-элементов.

Многослойный персептрон по Розенблатту (англ. Rosenblatt multilayer perceptron) — персептрон, который содержит более 1 слоя A-элементов.

Многослойный персептрон по Румельхарту (англ. Rumelhart multilater perceptron) — частный случай многослойного персептрона по Розенблатту, с двумя особенностями:

S-A связи могут иметь произвольные веса и обучаться наравне с A-R связями; Обучение производится по специальному алгоритму, который называется обучением по методу обратного распространения ошибки.

Задача обучения перцептрона — подобрать такие $w_0, w_1, w_2, \ldots, w_n$, чтобы $sign(\sigma(w_0 + w_1 \cdot x_1 + w_2 \cdot x_2 + \ldots + w_n \cdot x_n))$ как можно чаще совпадал с $y(x)$ — значением в обучающей выборке (здесь σ — функция активации). Для удобства, чтобы не тащить за собой свободный член w_0, добавим в вектор x лишнюю «виртуальную размерность» и будем считать, что $x = (1, x_1, x_2, \ldots, x_n)$. Тогда $w_0 + w_1 \cdot x_1 + w_2 \cdot x_2 + \ldots + w_n \cdot x_n$ можно заменить на $w^T \cdot x$.

Чтобы обучать эту функцию, сначала надо выбрать функцию ошибки, которую потом можно оптимизировать градиентным спуском. Число неверно классифицированных примеров не подходит на эту кандидатуру, потому что эта функция кусочно-гладкая, с массой разрывов: она будет принимать только целые значения и резко меняться при переходе от одного числа неверно классифицированных примеров к другому. Поэтому использовать будем другую функцию, так называемый критерий персептрона:

$E_P(w) = -\sum_{x \in M} y(x)(\sigma(w^T \cdot x))$, где M — множество примеров, которые персептрон с весами w классифицирует неправильно.

Иначе говоря, мы минимизируем суммарное отклонение наших ответов от правильных, но только в неправильную сторону; верный ответ ничего не вносит в функцию ошибки. Умножение на $y(x)$ здесь нужно для того, чтобы знак произведения всегда получался отрицательным: если правильный ответ −1, значит, перцептрон выдал положительное число (иначе бы ответ был верным), и наоборот. В результате у нас получилась кусочно-линейная функция, дифференцируемая почти везде, а этого вполне достаточно.

Теперь $E_P(w)$ можно оптимизировать градиентным спуском. На очередном шаге получаем:
$w^{(\tau+1)} = w^{(\tau)} - \eta \bigtriangledown_w E_P(w)$.

Алгоритм такой — мы последовательно проходим примеры $x_1, x_2, \ldots$ из обучающего множества, и для каждого x_n:

если он классифицирован правильно, не меняем ничего; а если неправильно, прибавляем $\eta \bigtriangledown_w E_P(w)$. Ошибка на примере x_n при этом, очевидно, уменьшается, но, конечно, совершенно никто не гарантирует, что вместе с тем не увеличится ошибка от других примеров. Это правило обновления весов так и называется — правило обучения персептрона, и это было основной математической идеей работы Розенблатта.

Часть 1.4 Алгоритм классификации на основе нейронной сети

Т.к. базовая теория для понимания нейронных сетей была изложена в части 1.3, расскажем о градиенте, градиентном спуске, функциях активации, функциях ошибок, переобучении, структуре полносвязной нейронной сети, методе обратного распространения ошибки, нейронах смещения, рекомендациях обучения и метриках оценки качества поподробнее.

Градиент — вектор, своим направлением указывающий направление возрастания (а антиградиент - убывания) некоторой скалярной величины φ, (значение которой меняется от одной точки пространства к другой, образуя скалярное поле), а по величине (модулю) равный скорости роста этой величины в этом направлении.

С математической точки зрения на градиент можно смотреть как на:

1. Коэффициент линейности изменения значения функции многих переменных от изменения значения аргумента;
2. Вектор в пространстве области определения скалярной функции многих переменных, составленный из частных производных;
3. Строки матрицы Якоби содержат градиенты составных скалярных функций из которых состоит векторная функция многих переменных.

Пространство, на котором определена функция и её градиент, может быть, вообще говоря, как обычным трёхмерным пространством, так и пространством любой другой размерности любой физической природы или чисто абстрактным (безразмерным).

Стандартное обозначение - $grad\,\varphi$

Для случая трёхмерного пространства градиентом дифференцируемой в некоторой области скалярной функции координат $\varphi = \varphi(x, y, z)$ называется векторная функция с компонентами
$\frac{\partial \varphi}{\partial x}, \frac{\partial \varphi}{\partial y}, \frac{\partial \varphi}{\partial z}$.

Или, использовав для единичных векторов по осям прямоугольных декартовых координат
$\overrightarrow{e_x}, \overrightarrow{e_y}, \overrightarrow{e_z}$:

$grad\,\varphi = \bigtriangledown\varphi = \frac{\partial \varphi}{\partial x}\overrightarrow{e_x} + \frac{\partial \varphi}{\partial y}\overrightarrow{e_y} + \frac{\partial \varphi}{\partial z}\overrightarrow{e_z}$

Если φ - функция n переменных $x_1, \ldots, x_n$, то ее градиентом называется n-мерный вектор

$(\frac{\partial \varphi}{\partial x_1}, \ldots, \frac{\partial \varphi}{\partial x_n})$,

компоненты которого равны частным производным φ по всем её аргументам.

Размерность вектора градиента определяется, таким образом, размерностью пространства (или многообразия), на котором задано скалярное поле, о градиенте которого идёт речь. Оператором градиента называется оператор, действие которого на скалярную функцию (поле) даёт её градиент. Этот оператор иногда коротко называют просто «градиентом».

Смысл градиента любой скалярной функции f в том, что его скалярное произведение с бесконечно малым вектором перемещения dx даёт полный дифференциал этой функции при соответствующем изменении координат в пространстве, на котором определена f, то есть линейную (в случае общего положения она же главная) часть изменения f при смещении на dx. Применяя одну и ту же букву для обозначения функции от вектора и соответствующей функции от его координат, можно написать:

$$df = \frac{\partial f}{\partial x_1}dx_1 + \frac{\partial f}{\partial x_2}dx_2 + \frac{\partial f}{\partial x_3}dx_3 + \ldots = \sum_i \frac{\partial f}{\partial x_i}dx_i = (grad\, f * dx)$$

Стоит здесь заметить, что поскольку формула полного дифференциала не зависит от вида координат x_i, то есть от природы параметров x вообще, то полученный дифференциал является инвариантом, то есть скаляром, при любых преобразованиях координат, а поскольку dx — это вектор, то градиент, вычисленный обычным образом, оказывается ковариантным вектором, то есть вектором, представленным в дуальном базисе, какой только и может дать скаляр при простом суммировании произведений координат обычного (контравариантного), то есть вектором, записанным в обычном базисе. Таким образом, выражение (вообще говоря — для произвольных криволинейных координат) может быть вполне правильно и инвариантно записано как:

$df = \sum_i (\partial_i f) dx^i$, или, опуская по правилу Эйнштейна знак суммы,

$df = (\partial_i f) dx^i$.

В ортонормированном базисе мы можем писать все индексы нижними, как мы и делали выше. Однако градиент оказывается настоящим ковариантным вектором в любых криволинейных координатах.

Используя интегральную теорему

$\int\limits_V \int \int \nabla \varphi dV = \int\limits_S \varphi ds$, градиент можно выразить в интегральной форме:

$$\nabla \varphi = \lim_{V \to 0} \frac{1}{V} (\int\limits_S \int \varphi ds),$$

здесь S — замкнутая поверхность охватывающая объём V, ds — нормальный элемент этой поверхности.

Градиентный спуск - численный метод нахождения локального минимума или максимума функции с помощью движения вдоль градиента, один из основных численных методов современной оптимизации.

Пусть объекты задаются n числовыми признаками $f_j : X \to R, j = 1 \ldots n$ и пространство признаковых описаний в таком случае $X = R^n$. Пусть Y - конечное множество меток классов и задана обучающая выборка пар «объект-ответ» $\{(x_1, y_1), \ldots, (x_l, y_l)\}$. Пусть семейство алгоритмов $a(x, w)$ имеет параметр вектор весов w. И пускай мы выбрали какую-нибудь функцию потерь. Для i-го объекта выборки для алгоритма с весами w обозначим ее $L_i(w)$. Необходимо минимизировать эмпирический риск, т.е. $Q(w) = \sum_i L_i(w) \to \min\limits_w$. Если функция потерь принадлежит классу $C_1(X)$, то можно применить метод градиентного спуска. Выберем $w^{(0)}$ - начальное приближение. Тогда каждый следующий вектор параметров будет вычисляться как $w^{(t+1)} = w^{(t)} - h \sum_{i=1}^{l} \nabla L_i(w^{(t)})$, где h - градиентный шаг, смысл которого заключается в том, насколько сильно менять вектор весов в направлении градиента. Остановка алгоритма будет определяться сходимостью Q или w.

Стохастический градиентный спуск (англ. stochastic gradient descent) - оптимизационный алгоритм, отличающийся от обычного градиентного спуска тем, что градиент оптимизируемой функции считается на каждом шаге не как сумма градиентов от каждого элемента выборки, а как градиент от одного, случайно выбранного элемента.

Проблема предыдущего алгоритма заключается в том, что чтобы определить новое приближение вектора весов необходимо вычислить градиент от каждого элемента выборки, что может сильно замедлять алгоритм. Идея ускорения алгоритма заключается в использовании только одного элемента, либо некоторой подвыборки для подсчета нового приближения весов. То есть теперь новое приближение будет вычисляться как $w^{(t+1)} = w^{(t)} - h \nabla L_i(w^{(t)})$, где i - случайно выбранный индекс. Так как теперь направление изменения w будет определяться за $O(1)$, подсчет Q на каждом шаге будет слишком дорогостоящим. Для того, чтобы ускорить оценку Q, будем использовать приближенную рекуррентную формулу. Можно выбрать одну из следующих формул:

- среднее арифметическое: $\overline{Q_m} = \frac{1}{m}\varepsilon_m + \frac{1}{m}\varepsilon_{m-1} + \frac{1}{m}\varepsilon_{m-2} + \ldots = \frac{1}{m}\varepsilon_m + (1 - \frac{1}{m}\overline{Q_{m-1}})$;

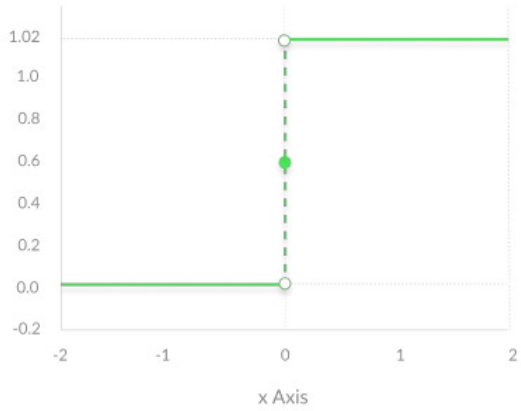
- экспоненциальное скользящее среднее: $\overline{Q_m} = \lambda\varepsilon_m + (1-\lambda)\varepsilon_{m-1} + (1-\lambda)^2\varepsilon_{m-2} + \ldots = \lambda\varepsilon_m + (1-\lambda)\overline{Q_{m-1}}$, где λ -

Функция активации (англ. activation function) a(x) определяет выходное значение нейрона в зависимости от результата взвешенной суммы входов и порогового значения.

Рассмотрим нейрон, у которого взвешенная сумма входов: $z = \sum_i w_i x_i + b$, где w_i и x_i — вес и входное значение i-ого входа, а b — смещение. Полученный результат передается в функцию активации, которая решает рассматривать этот нейрон как активированный, или его можно игнорировать.

Рассмотрим 6 наиболее популярных функций активации

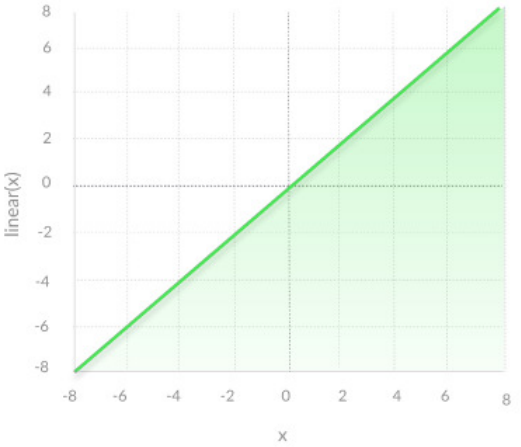
1. Ступенчатая функция (англ. binary step function) является пороговой функцией активации. То есть если z больше или меньше некоторого значения, то нейрон становится активированным. Такая функция отлично работает для бинарной классификации. Но она не работает, когда для классификации требуется большее число нейронов и количество возможных классов больше двух.



2. Линейная функция (англ. linear function) представляет собой прямую линию, то есть $a(x) = \sum_i c_i x_i$, а это значит, что результат этой функции активации пропорционален переданному аргументу. В отличии от предыдущей функции, она позволяет получить диапазон значений на выходе, а не только бинарные 0 и 1, что решает проблему классификации с большим количеством классов. Но у линейной функции есть две основных проблемы:

- Невозможность использования метода обратного распространения ошибки. Так как в основе этого метода обучения лежит градиентный спуск, а для того чтобы его найти, нужно взять производную, которая для данной функции активации — константа и не зависит от входных значений. То есть при обновлении весов нельзя сказать улучшается ли эмпирический риск на текущем шаге или нет.
- Рассмотрим нейронную сеть с несколькими слоями с данной функцией активации. Так как для каждого слоя выходное значение линейно, то они образуют линейную комбинацию, результатом которой является линейная функция. То есть финальная функция активации на последнем слое зависит только от входных значений на первом слое. А это значит, что любое количество слоев может быть заменено всего одним слоем, и, следовательно, нет смысла создавать многослойную сеть.

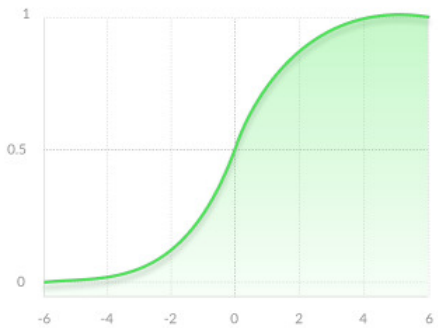
Главное отличие линейной функции от остальных в том, что ее область значений не ограничена: $(-\infty; +\infty)$. Следовательно, ее нужно использовать, когда выходное значение нейрона должно $\in R$, а не ограниченному интервалу.



3. Сигмоидная функция (англ. sigmoid function), которую также называет логистической (англ. logistic function), является гладкой монотонно возрастающей нелинейной функцией: $\sigma(z) = \frac{1}{1+e^{-z}}$. И так как эта функция нелинейна, то ее можно использовать в нейронных сетях с множеством слоев, а также обучать эти сети методом обратного распространения ошибки. Сигмоида ограничена двумя горизонтальными асимптотами $y = 1$ и $y = 0$, что дает нормализацию выходного значения каждого нейрона. Кроме того, для сигмоидной функции характерен гладкий градиент, который предотвращает "прыжки" при подсчете выходного значения. Помимо всего этого, у этой функции есть еще одно преимущество, для значений $x > 2$ и $x < -2$, y "прижимается" к одной из асимптот, что позволяет делать четкие предсказания классов.

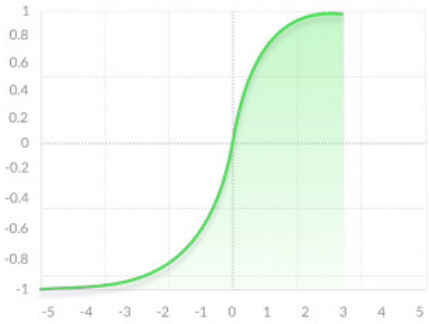
Несмотря на множество сильных сторон сигмоидной функции, у нее есть значительный недостаток. Производная такой функции крайне мала во всех точках, кроме сравнительно небольшого промежутка. Это сильно усложняет процесс улучшения весов с помощью градиентного спуска. Более того, эта проблема усугубляется в случае, если модель содержит много слоев. Данная проблема называется проблемой исчезающего градиента.

Что касается использования сигмоидной функции, то ее преимущество над другими — в нормализации выходного значения. Иногда, это бывает крайне необходимо. К примеру, когда итоговое значение слоя должно представлять вероятность случайной величины. Кроме того, эту функцию удобно применять при решении задачи классификации, благодаря свойству "прижимания" к асимптотам.



4. Функция гиперболического тангенса (англ. hyperbolic tangent) имеет вид: $tanh(z) = \frac{2}{1+e^{-2z}} - 1$. Эта функция является скорректированной сигмоидной функцей $tanh(z) = 2 \cdot sigma(2z) - 1$, то есть она сохраняет те же преимущества и недостатки, но уже для диапазона значений $(-1; 1)$.

Обычно, $tanh$ является предпочтительнее сигмоиды в случаях, когда нет необходимости в нормализации. Это происходит из-за того, что область определения данной функции активации центрирована относительно нуля, что снимает ограничение при подсчете градиента для перемещения в определенном направлении. Кроме того, производная гиперболического тангенса значительно выше вблизи нуля, давая большую амплитуду градиентному спуску, а следовательно и более быструю сходимость.

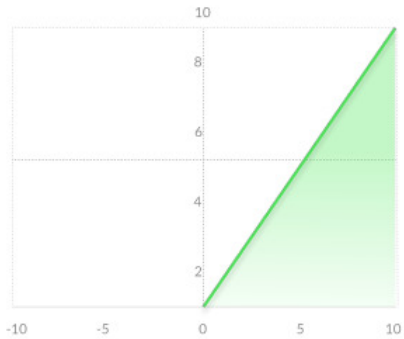


5. Rectified Linear Unit — это наиболее часто используемая функция активации при глубоком обучении. Данная функция возвращает 0, если принимает отрицательный аргумент, в случае же положительного аргумента, функция возвращает само число. То есть она может быть записана как $f(z) = max(0, z)$. На первый взгляд может показаться, что она линейна и имеет те же проблемы что и линейная функция, но это не так и ее можно использовать в нейронных сетях с множеством слоев. Функция ReLU обладает несколькими преимущества перед сигмоидой и гиперболическим тангенсом:

- Очень быстро и просто считается производная. Для отрицательных значений -0, для положительных -1.
- Разреженность активации. В сетях с очень большим количеством нейронов использование сигмоидной функции или гиперболического тангенса в качестве активационный функции влечет активацию почти всех нейронов, что может сказаться на производительности обучения модели. Если же использовать ReLU, то количество включаемых нейронов станет меньше, в силу характеристик функции, и сама сеть станет легче.

У данной функции есть один недостаток, называющийся проблемой умирающего ReLU. Так как часть производной функции равна нулю, то и градиент для нее будет нулевым, а то это значит, что веса не будут изменяться во время спуска и нейронная сеть перестанет обучаться.

Функцию активации ReLU следует использовать, если нет особых требований для выходного значения нейрона, вроде неограниченной области определения. Но если после обучения модели результаты получились не оптимальные, то стоит перейти к другим функциям, которые могут дать лучший результат.



6. Одной из проблем стандартного ReLU является затухающий, а именно нулевой, градиент при отрицательных значениях. При использовании обычного ReLU некоторые нейроны умирают, а отследить умирание нейронов не просто. Чтобы решить эту проблему иногда используется подход ReLU с «утечкой» (leak) — график функции активации на отрицательных значениях образует не горизонтальную прямую, а наклонную, с маленьким угловым коэффициентом (порядка $0,01$). То есть она может быть записана как

$$f(x) = \begin{cases} 0.01x, & \text{if x} < 0 \\ x, & \text{otherwise} \end{cases}$$

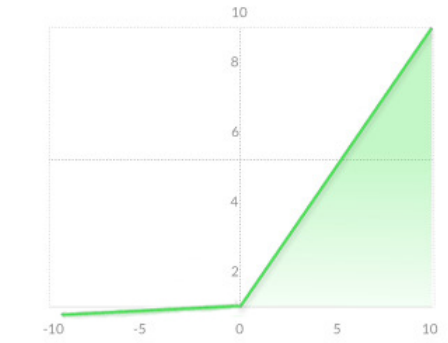
Такое небольшое отрицательное значение помогает добиться ненулевого градиента при отрицательных значениях. Однако, функция Leaky ReLU имеет некоторые недостатки:

- Сложнее считать производную, по сравнению со стандартным подходом (так как значения уже не равны нулю), что замедляет работу каждой эпохи.
- Угловой коэффициент прямой также является гиперпараметром, который надо настраивать.
- На практике, результат не всегда сильно улучшается относительно ReLU.

Стоит отметить, что помимо проблемы умирающих нейронов, у ReLU есть и другая — проблема затухающего градиента. При слишком большом количестве слоев градиент будет принимать очень маленькое значение, постепенно уменьшаясь до нуля. Из-за

этого нейронная сеть работает нестабильно и неправильно. Leaky ReLU (LReLU) решает первую проблему, но в по-настоящему глубоких сетях проблема затухания градиента все еще встречается и при использовании этого подхода.

На практике LReLU используется не так часто. Практический результат использования LReLU вместо ReLU отличается не слишком сильно. Однако в случае использования Leaky требуется дополнительно настраивать гиперпараметр (уровень наклона при отрицательных значениях), что требует определенных усилий. Еще одной проблемой является то, что результат LReLU не всегда лучше чем при использовании обычного ReLU, поэтому чаще всего такой подход используют как альтернатива. Довольно часто на практике используется PReLU (Parametric ReLU), который позволяет добиться более значительных улучшений по сравнению с ReLU и LReLU. Также, в случае параметрической модификации ReLU, угол наклона не является гиперпараметром и настраивается



Чтобы изменить содержимое ячейки, дважды нажмите на нее (или выберите "Ввод")

7. Функция softmax, также известная как softargmax или нормализованная экспоненциальная функция, преобразует вектор из K действительных чисел в распределение вероятностей K возможных исходов. Это обобщение логистической функции на несколько измерений, и используется в мультиномиальной логистической регрессии. Функция softmax часто используется в качестве функции последней активации нейронной сети для нормализации выходных данных сети до распределения вероятностей по прогнозируемым выходным классам на основе аксиомы выбора Люса.

Функция softmax принимает в качестве входных данных вектор z из K действительных чисел и нормализует его в распределение вероятностей, состоящее из K вероятностей, пропорциональных экспонентам входных чисел. То есть до применения softmax некоторые компоненты вектора могут быть отрицательными или больше единицы; и могут не суммироваться до 1; но после применения softmax каждый компонент будет находиться в интервале $(0, 1)$, и компоненты будут складываться до 1, так что их можно интерпретировать как вероятности. Кроме того, большие входные компоненты будут соответствовать большим вероятностям.

Стандартная (единичная) функция softmax $\sigma : R^K \to (0,1)^K$ определяется, когда $K >= 1$ по формуле

$$\sigma(z)_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \; for\; i,\; \ldots,\; K\; and\; z = (z_1, z_2, \ldots, z_K) \in R^K.$$

Простыми словами, она применяет стандартную экспоненциальную функцию к каждому элементу z_i входного вектора z и нормализует эти значения путем деления на сумму всех этих экспонент; эта нормализация гарантирует, что сумма компонентов выходного вектора $\sigma(z)$ равна 1.

Вместо e можно использовать другое основание $b > 0$. Если $0 < b < 1$, меньшие входные компоненты приведут к увеличению выходных вероятностей, а уменьшение значения b создаст распределения вероятностей, которые в большей степени сосредоточены вокруг позиций наименьших входных значений. И наоборот, если $b > 1$, большие входные компоненты приведут к увеличению выходных вероятностей, а увеличение значения b создаст распределения вероятностей, которые более сконцентрированы вокруг позиций наибольших входных значений. Запись $b = e^{\beta}$ или $e^{-\beta}$ (для реального β) дает выражения:

$\sigma(z)_i = \frac{e^{\beta z_i}}{\sum_{j=1}^{K} e^{\beta z_j}}$ or $\sigma(z)_i = \frac{e^{-\beta z_i}}{\sum_{j=1}^{K} e^{-\beta z_j}}$ for $i = 1, \ldots, K$.

В некоторых полях база фиксирована, соответствует фиксированному масштабу, в то время как в других параметр β изменяется.

Несколько слов о функции ошибок и переобучении нейронной сети.

Функция ошибок - это целевая функция, требующая минимизации в процессе управляемого обучения нейронной сети. С помощью функции ошибок можно оценить качество работы нейронной сети во время обучения. Например, часто используется сумма квадратов ошибок. От качества обучения нейронной сети зависит ее способность решать поставленные перед ней задачи.

Переобучение, или чрезмерно близкая подгонка - излишне точное соответствие нейронной сети конкретному набору обучающих примеров, при котором сеть теряет способность к обобщению.

Переобучение возникает в случае слишком долгого обучения, недостаточного числа обучающих примеров или переусложненной структуры нейронной сети.

Переобучение связано с тем, что выбор обучающего (тренировочного) множества является случайным. С первых шагов обучения происходит уменьшение ошибки. На последующих шагах с целью уменьшения ошибки (целевой функции) параметры подстраиваются под особенности обучающего множества. Однако при этом происходит "подстройка" не под общие закономерности ряда, а под особенности его части - обучающего подмножества. При этом точность прогноза уменьшается.

Один из вариантов борьбы с переобучением сети - деление обучающей выборки на два множества (обучающее и тестовое).

На обучающем множестве происходит обучение нейронной сети. На тестовом множестве осуществляется проверка построенной модели. Эти множества не должны пересекаться.

С каждым шагом параметры модели изменяются, однако постоянное уменьшение значения целевой функции происходит именно на обучающем множестве. При разбиении множества на два мы можем наблюдать изменение ошибки прогноза на тестовом множестве параллельно с наблюдениями над обучающим множеством. Какое-то количество шагов ошибки прогноза уменьшается на обоих множествах. Однако на определенном шаге ошибка на тестовом множестве начинает возрастать, при этом ошибка на обучающем множестве продолжает уменьшаться. Этот момент считается концом реального или настоящего обучения, с него и начинается переобучение.

Прежде, чем перейти к рассмотрению метода обратного распространения ошибки, упомянем классическую архитектуру нейронной сети - полносвязную нейросеть прямого распространения, или Fully Connected Feed-Forward Neural Network, FNN, за рамки которой в теме курсовой выходить не будем.

FNN, как следует из названия, передает сигнал только в одну сторону, и каждый нейрон связан со всеми нейронами предыдущего слоя. Данная архитектура показала хорошие результаты в теме классификации, но есть некоторые проблемы:

- Много параметров.

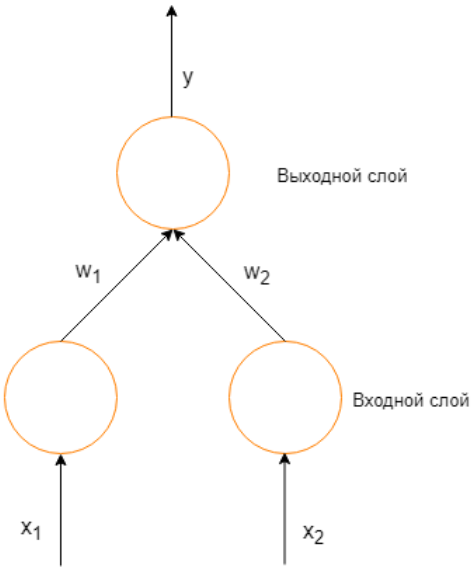
Например, если взять нейросеть из 3 скрытых слоев, которой нужно обрабатывать картинки 100*100 ps, это значит, что на входе будет 10 000 ps, и они заводятся на 3 слоя. В общем, если честно посчитать все параметры, у такой сети их будет порядка миллиона. Это на самом деле много. Чтобы обучить нейросеть с миллионом параметров, нужно очень много обучающих примеров, которые не всегда есть.

Кроме того, сеть, у которой много параметров, имеет дополнительную склонность переобучаться. Она может заточиться на то, чего в реальности не существует, например, на некоторый шум. Даже если, в конце концов, сеть запомнит примеры, но на тех, которых она не видела, потом не сможет нормально использоваться.

- Затухающие градиенты. Данная проблема была описана выше, в теме функций активации.

Метод обратного распространения ошибок (англ. backpropagation) — метод вычисления градиента, который используется при обновлении весов в нейронной сети.

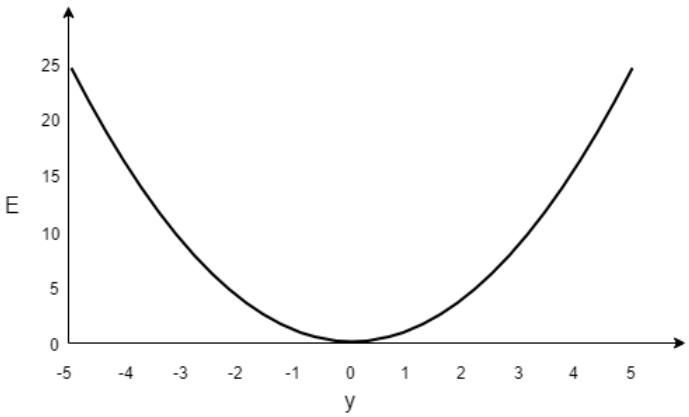
Рассмотрим простую нейронную сеть без скрытых слоев, с двумя входными вершинами и одной выходной, в которых каждый нейрон использует линейную функцию активации, (обычно, многослойные нейронные сети используют нелинейные функции активации, линейные функции используются для упрощения понимания) которая является взвешенной суммой входных данных.



Изначально веса задаются случайно. Затем, нейрон обучается с помощью тренировочного множества, которое в этом случае состоит из множества троек (x_1, x_2, t), где x_1 и x_2 — это входные данные сети и t — правильный ответ. Начальная сеть, приняв на вход x_1 и x_2, вычислит ответ y, который вероятно отличается от t. Общепринятый метод вычисления несоответствия между ожидаемым t и получившимся y ответом — квадратичная функция потерь:

$E = (t - y)^2$, где E ошибка.

В качестве примера, обучим сеть на объекте $(1, 1, 0)$, таким образом, значения x_1 и x_2 равны 1, а t равно 0. Построим график зависимости ошибки E от действительного ответа y, его результатом будет парабола.



Минимум параболы соответствует ответу y, минимизирующему E. Если тренировочный объект один, минимум касается горизонтальной оси, следовательно ошибка будет нулевая и сеть может выдать ответ y, равный ожидаемому ответу t. Следовательно, задача преобразования входных значений в выходные может быть сведена к задаче оптимизации, заключающейся в поиске функции, которая даст минимальную ошибку.

График ошибки для нейрона с линейной функцией активации и одним тренировочным объектом В таком случае, выходное значение нейрона — взвешенная сумма всех его входных значений:

$y = x_1 w_1 + x_2 w_2$, где w_1 и w_2 - веса на ребрах, соединяющих входные вершины с выходной. Следовательно, ошибка зависит от весов ребер, входящих в нейрон. И именно это нужно менять в процессе обучения. Распространенный алгоритм для поиска набора весов, минимизирующего ошибку — градиентный спуск. Метод обратного распространения ошибки используется для вычисления самого "крутого" направления для спуска.

Рассмотрим дифференцирование для однослойной сети

Метод градиентного спуска включает в себя вычисление дифференциала квадратичной функции ошибки относительно весов сети. Обычно это делается с помощью метода обратного распространения ошибки. Предположим, что выходной нейрон один, (их может быть несколько, тогда ошибка — это квадратичная норма вектора разницы) тогда квадратичная функция ошибки:

$E = \frac{1}{2}(t - y)^2$, где E - квадратичная ошибка, t - требуемый ответ для обучающего обазца, y - действительный ответ сети.

Множитель $\frac{1}{2}$ добавлен чтобы предотвратить возникновение экспоненты во время дифференцирования. На результат это не повлияет, потому что позже выражение будет умножено на произвольную величину скорости обучения (англ. learning rate).

Для каждого нейрона j, его выходное значение o_j определено как $o_j = \varphi(net_j) = \varphi(\sum k = 1^n w_{kj} o_k)$.

Входные значения net_j нейрона — это взвешенная сумма выходных значений o_k предыдущих нейронов. Если нейрон в первом слое после входного, то o_k входного слоя — это просто входные значения x_k сети. Количество входных значений нейрона n. Переменная w_{kj} обозначает вес на ребре между нейроном k предыдущего слоя и нейроном j текущего слоя.

Функция активации φ нелинейна и дифференцируема. Одна из распространенных функций активации — сигмоида: $\varphi(z) = \frac{1}{1+e^{-z}}$ у

$\frac{d\varphi(z)}{}$

Найдем производную ошибки

Вычисление частной производной ошибки по весам w_{ij} выполняется с помощью цепного правила:

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial o_j} \frac{\partial o_j}{\partial net_j} \frac{\partial net_j}{\partial w_{ij}}$$

Только одно слагаемое в net_j зависит от w_{ij}, так что

$$\frac{\partial net_j}{\partial w_{ij}} = \frac{\partial}{\partial w_{ij}} (\sum_{k=1}^{n} w_{kj} o_k) = \frac{\partial}{\partial w_{ij}} w_{ij} o_i = o_i$$

Если нейрон в первом слое после входного, то o_i — это просто x_i.

Производная выходного значения нейрона j по его входному значению — это просто частная производная функции активации (предполагается что в качестве функции активации используется сигмоида):

$$\frac{\partial o_j}{\partial net_j} = \frac{\partial}{\partial net_j} \varphi(net_j) = \varphi(net_j)(1 - \varphi(net_j))$$

По этой причине данный метод требует дифференцируемой функции активации. (Тем не менее, функция ReLU стала достаточно популярной в последнее время, хоть и не дифференцируема в 0)

Первый множитель легко вычислим, если нейрон находится в выходном слое, ведь в таком случае $o_j = y$ и

$$\frac{\partial E}{\partial o_j} = \frac{\partial E}{\partial y} = \frac{\partial}{\partial y} \frac{1}{2} (t - y)^2 = y - t$$

Тем не менее, если j произвольный внутренний слой сети, нахождение производной E по o_j менее очевидно.

Если рассмотреть E как функцию, берущую на вход все нейроны $L = u, v, \ldots, w$ получающие на вход значение нейрона j,

$$\frac{\partial E(o_j)}{\partial o_j} = \frac{\partial E(net_v, net_\vartheta, \ldots, net_\omega)}{\partial o_j}$$

И взять полную производную по o_j, получим рекурсивное выражение для производной:

$$\frac{\partial E}{\partial o_j} = \sum_{l \in L} (\frac{\partial E}{\partial net_l} \frac{\partial net_l}{\partial o_j}) = \sum_{l \in L} (\frac{\partial E}{\partial o_l} \frac{\partial o_l}{\partial net_l} w_{jl})$$

Следовательно, производная по o_j может быть вычислена если все производные по выходным значениям o_l следующего слоя известны.

Если собрать все месте:

$\frac{\partial E}{\partial w_{ij}} = \delta_j o_i$, и

$$\delta_j = \frac{\partial E}{\partial o_j} \frac{\partial o_j}{\partial net_j} = \begin{cases} (o_j - t_j) o_j (1 - o_j) & \text{если j является нейроном выходного слоя,} \\ (\sum_{l \in L} w_{jl} \delta_l) o_j (1 - o_j) & \text{если j является нейроном внутреннего слоя.} \end{cases}$$

Чтобы обновить вес w_{ij} используя градиентный спуск, нужно выбрать скорость обучения, $\eta > 0$. Изменение в весах должно отражать влияние E на увеличение или уменьшение в w_{ij}. Если $\frac{\partial E}{\partial w_{ij}} > 0$, увеличение w_{ij} увеличивает E; наоборот, если $\frac{\partial E}{\partial w_{ij}} < 0$, увеличение w_{ij} уменьшает E. Новый $\triangle w_{ij}$ добавлен к старым весам, и произведение скорости обучения на градиент, умноженный на -1, гарантирует, что w_{ij} изменения будут всегда уменьшать E. Другими словами, в следующем уравнении, $-\eta \frac{\partial E}{\partial w_{ij}}$ всегда изменяет w_{ij} в такую сторону, что E уменьшается:

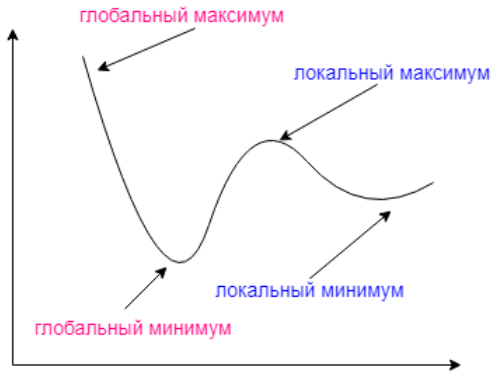
$\triangle w_{ij} = -\eta \frac{\partial E}{\partial w_{ij}} = -\eta \delta_j o_i$

О недостатках алгоритма

Несмотря на многочисленные успешные применения обратного распространения, оно не является универсальным решением. Больше всего неприятностей приносит неопределённо долгий процесс обучения. В сложных задачах для обучения сети могут потребоваться дни или даже недели, она может и вообще не обучиться. Причиной может быть одна из описанных ниже.

- Паралич сети

В процессе обучения сети значения весов могут в результате коррекции стать очень большими величинами. Это может привести к тому, что все или большинство нейронов будут функционировать при очень больших выходных значениях, а производная активирующей функции будет очень мала. Так как посылаемая обратно в процессе обучения ошибка пропорциональна этой производной, то процесс обучения может практически замереть.

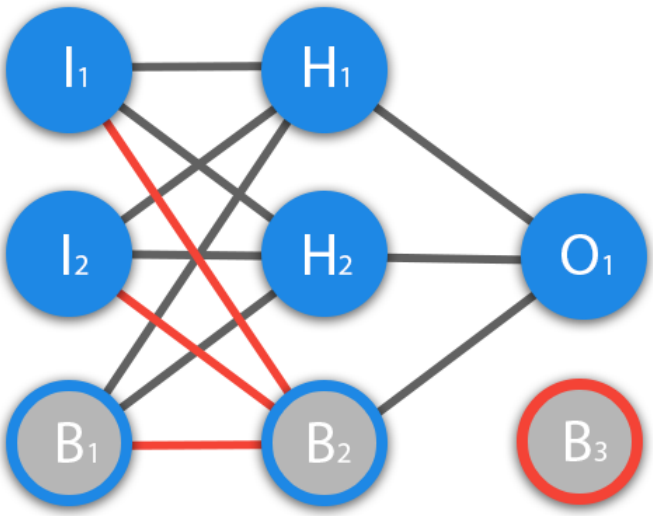


- Локальные минимумы

Градиентный спуск с обратным распространением ошибок гарантирует нахождение только локального минимума функции; также, возникают проблемы с пересечением плато на поверхности функции ошибки.

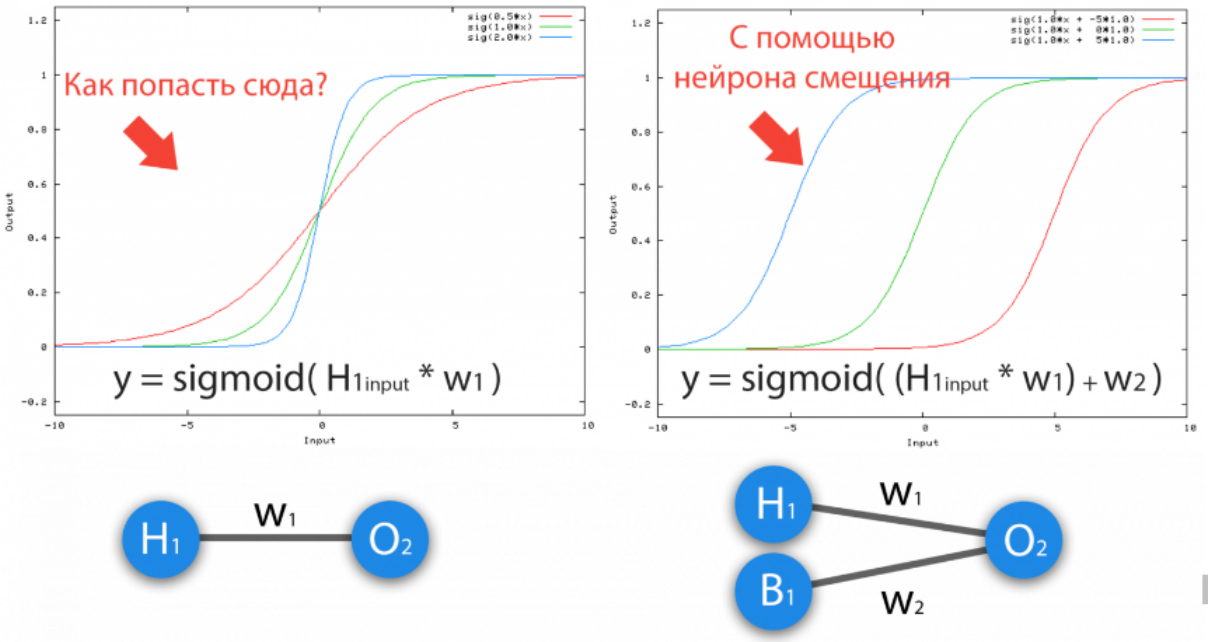
Стоит затронуть так же один из необходимых для понимания нейронов.

Нейрон смещения или bias нейрон — это вид нейронов, используемый в большинстве нейросетей. Особенность этого типа нейронов заключается в том, что его вход и выход всегда равняются 1 и они никогда не имеют входных синапсов.



Нейроны смещения могут либо присутствовать в нейронной сети по одному на слое, либо полностью отсутствовать, 50/50 быть не может (красным на схеме обозначены веса и нейроны которые размещать нельзя). Соединения у нейронов смещения такие же, как у обычных нейронов — со всеми нейронами следующего уровня, за исключением того, что синапсов между двумя bias нейронами быть не может. Следовательно, их можно размещать на входном слое и всех скрытых слоях, но никак не на выходном слое, так как им попросту не с чем будет формировать связь.

Нейрон смещения нужен для того, чтобы иметь возможность получать выходной результат, путем сдвига графика функции активации вправо или влево. Если это звучит запутанно, давайте рассмотрим простой пример, где есть один входной нейрон и один выходной нейрон.



Тогда можно установить, что выход O2 будет равен входу H1, умноженному на его вес, и пропущенному через функцию активации (формула на фото слева). В нашем конкретном случае, будем использовать сигмоид.

Когда в ходе обучения, мы регулируем веса скрытых и выходных нейронов, мы меняем наклон функции активации. Однако, регулирование веса нейронов смещения может дать нам возможность сдвинуть функцию активации по оси X и захватить новые участки. Иными словами, если точка, отвечающая за ваше решение, будет находиться, как показано на графике слева, то ваша НС никогда не сможет решить задачу без использования нейронов смещения. Поэтому, вы редко встретите нейронные сети без нейронов смещения.

Также нейроны смещения помогают в том случае, когда все входные нейроны получают на вход 0 и независимо от того какие у них веса, они все передадут на следующий слой 0, но не в случае присутствия нейрона смещения. Наличие или отсутствие нейронов смещения — это гиперпараметр (об этом чуть позже). Одним словом, вы сами должны решить, нужно ли вам использовать нейроны смещения или нет, прогнав НС с нейронами смешения и без них и сравнив результаты.

Стоило бы рассказать о методах оптимизации, в частности, алгоритмах обучения с оптимизацией по Adam и Нестерову для ускорения обучения НС, но ввиду некоторой теоретической избыточности ограничимся тем, что они помогают решать проблему медленной сходимости на пологих участках функции в градиентных алгоритмах.

После данной выше теории, можно перечислить рекомендации обучения:

1. Запускать алгоритм для разных начальных значений весовых коэффицентов. И, зачем, отобрать лучший вариант. Начальные значения генерируем случайным образом в окрестности нуля, кроме тех, что относятся к bias.
2. Запускаем алгоритм обучения с оптимизацией по Adam или Нестерову для ускорения обучения НС.
3. Выполнять нормировку входных значений и запоминать нормировочные параметры min, max из обучающей выборки.
4. Помещать в обучающую выборку самые разнообразные данные примерно равного количества.
5. Наблюдения на вход сети подавать случайным образом, корректировать веса после серии наблюдений, разбитых по mini-batch.
6. Проблема переобучения - использовать минимальное необходимое число нейронов в нейронной сети.
7. Разбивать все множество наблюдений на три выборки: обучающую, валидации и тестовую.
8. При малом числе слоев можно использовать гиперболическую и сигмоидальную функции активации или ReLU, при числе слоев от 8 и более - ReLU и ее вариации.
9. Для задач регрессии у выходных нейронов использовать линейную (linear) функцию активации, для задач классификации не пересекающихся классов - softmax.

Наконец, завершим теорию, затронув последнюю тему - метрики оценки качества, но только основные из них - Accuracy, precision, recall и F-мера.

Перед переходом к самим метрикам необходимо ввести важную концепцию для описания этих метрик в терминах ошибок классификации — confusion matrix (матрица ошибок). Допустим, что у нас есть два класса и алгоритм, предсказывающий принадлежность каждого объекта одному из классов, тогда матрица ошибок классификации будет выглядеть следующим образом:

	$y = 1$	$y = 0$
$\hat{y} = 1$	True Positive (TP)	False Positive (FP)
$\hat{y} = 0$	False Negative (FN)	True Negative (TN)

Здесь $\hat{y}$ — это ответ алгоритма на объекте, а y — истинная метка класса на этом объекте. Таким образом, ошибки классификации бывают двух видов: False Negative (FN) и False Positive (FP).

- Accuracy

Интуитивно понятной, очевидной и почти неиспользуемой метрикой является accuracy — доля правильных ответов алгоритма:

$$accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$

Эта метрика бесполезна в задачах с неравными классами, и это легко показать на примере.

Допустим, мы хотим оценить работу спам-фильтра почты. У нас есть 100 не-спам писем, 90 из которых наш классификатор определил верно (True Negative = 90, False Positive = 10), и 10 спам-писем, 5 из которых классификатор также определил верно (True Positive = 5, False Negative = 5). Тогда accuracy:

$$accuracy = \frac{5 + 90}{5 + 90 + 10 + 5} = 86,4$$

Однако если мы просто будем предсказывать все письма как не-спам, то получим более высокую accuracy:

$$accuracy = \frac{0 + 100}{0 + 100 + 0 + 10} = 90,9$$

При этом, наша модель совершенно не обладает никакой предсказательной силой, так как изначально мы хотели определять письма со спамом. Преодолеть это нам поможет переход с общей для всех классов метрики к отдельным показателям качества классов.

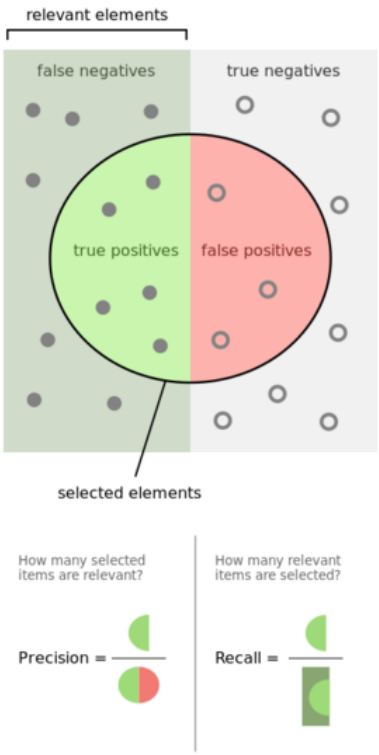
- Precision, recall и F-мера

Для оценки качества работы алгоритма на каждом из классов по отдельности введем метрики precision (точность) и recall (полнота).

$$precision = \frac{TP}{TP + FP}$$

$$recall = \frac{TP}{TP + FN}$$

Precision можно интерпретировать как долю объектов, названных классификатором положительными и при этом действительно являющимися положительными, a recall показывает, какую долю объектов положительного класса из всех объектов положительного класса нашел алгоритм.



Именно введение precision не позволяет нам записывать все объекты в один класс, так как в этом случае мы получаем рост уровня False Positive. Recall демонстрирует способность алгоритма обнаруживать данный класс вообще, а precision — способность отличать этот класс от других классов.

Как мы отмечали ранее, ошибки классификации бывают двух видов: False Positive и False Negative. В статистике первый вид ошибок называют ошибкой I-го рода, а второй — ошибкой II-го рода. В нашей задаче по определению оттока абонентов, ошибкой первого рода будет принятие лояльного абонента за уходящего, так как наша нулевая гипотеза состоит в том, что никто из абонентов не уходит, а мы эту гипотезу отвергаем. Соответственно, ошибкой второго рода будет являться "пропуск" уходящего абонента и ошибочное принятие нулевой гипотезы.

Precision и recall не зависят, в отличие от accuracy, от соотношения классов и потому применимы в условиях несбалансированных выборок. Часто в реальной практике стоит задача найти оптимальный (для заказчика) баланс между этими двумя метриками. Классическим примером является задача определения оттока клиентов. Очевидно, что мы не можем находить всех уходящих в отток клиентов и только их. Но, определив стратегию и ресурс для удержания клиентов, мы можем подобрать нужные пороги по precision и recall. Например, можно сосредоточиться на удержании только высокодоходных клиентов или тех, кто уйдет с большей вероятностью, так как мы ограничены в ресурсах колл-центра.

Обычно при оптимизации гиперпараметров алгоритма (например, в случае перебора по сетке GridSearchCV) используется одна метрика, улучшение которой мы и ожидаем увидеть на тестовой выборке. Существует несколько различных способов объединить precision и recall в агрегированный критерий качества. F-мера (в общем случае F_β) — среднее гармоническое precision и recall:

$$F_\beta = (1 + \beta^2) * \frac{precision * recall}{(\beta^2 * precision) + recall}$$

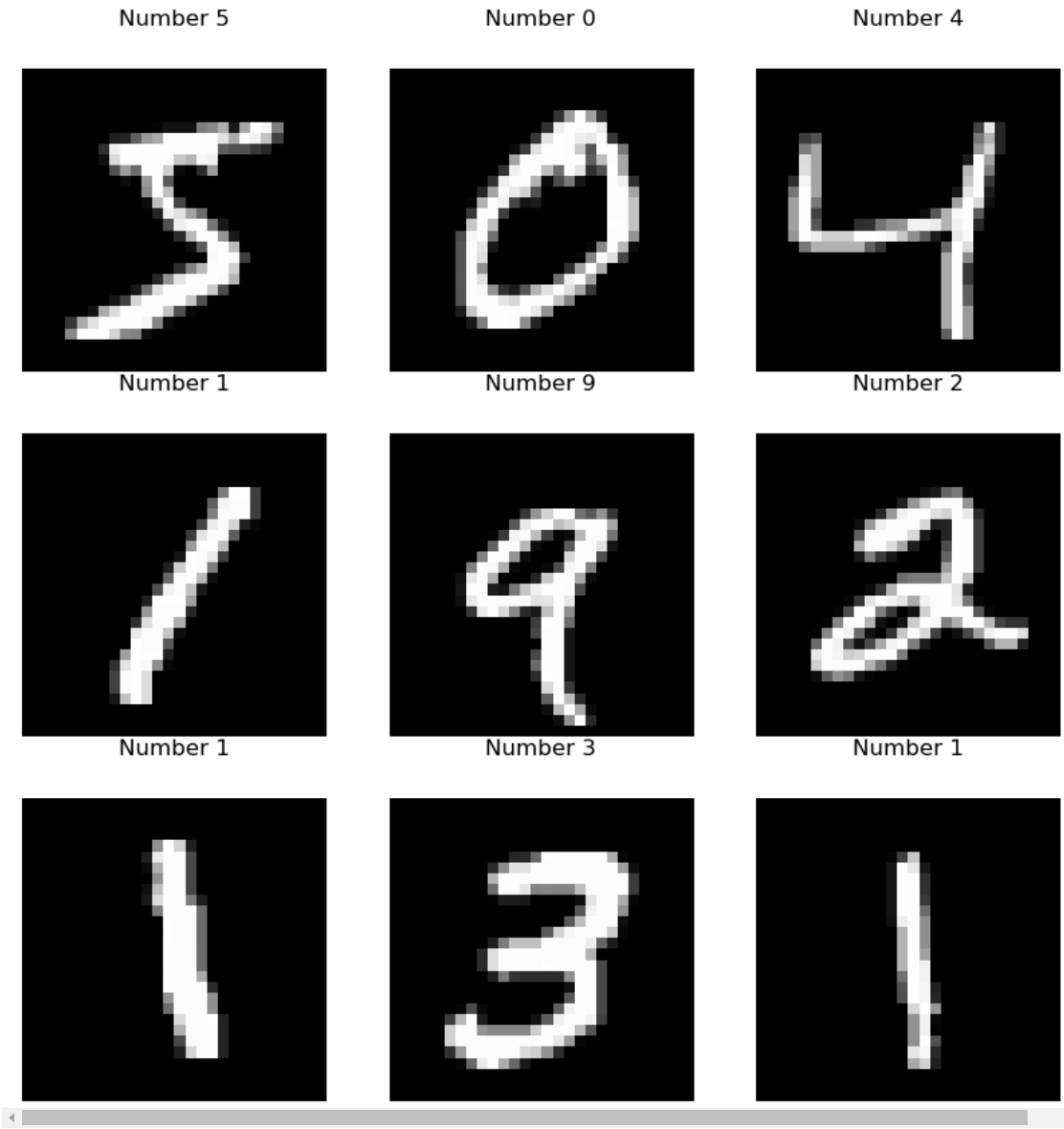
β в данном случае определяет вес точности в метрике, и при $\beta = 1$ это среднее гармоническое (с множителем 2, чтобы в случае precision = 1 и recall = 1 иметь $F_1 = 1$) F-мера достигает максимума при полноте и точности, равными единице, и близка к нулю, если один из аргументов близок к нулю.

Часть 2 Применение алгоритмов в задаче символьного распознавания на наборе рукописных цифр в датасете MNIST; сравнение эффективности а

```
# k-means by sklearn

import sys
import sklearn
import matplotlib
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline



from keras.datasets import mnist

(x_train, y_train), (x_test, y_test) = mnist.load_data()
```

```
fig, axs = plt.subplots(3, 3, figsize = (10, 10))
plt.gray()

for i, ax in enumerate(axs.flat):
    ax.matshow(x_train[i])
    ax.axis('off')
    ax.set_title('Number {}'.format(y_train[i]))

fig.show()
```

```
/tmp/ipykernel_510/3399184502.py:13: UserWarning: Matplotlib is currently using module://matplotlib_inline.backend_inline, whic
  fig.show()
```




```
# preprocessing the images

# convert each image to 1 dimensional array
X = x_train.reshape(len(x_train),-1)
Y = y_train

# normalize the data to 0 - 1
X = X.astype(float) / 255.

print(X.shape)
print(X[0].shape)
```

```
(60000, 784)
(784,)
```

```
from sklearn.cluster import MiniBatchKMeans

n_digits = len(np.unique(y_test))
print(n_digits)
```

```
# Initialize KMeans model
kmeans = MiniBatchKMeans(n_clusters = n_digits)

# Fit the model to the training data
kmeans.fit(X)
```

```
    10
    MiniBatchKMeans(n_clusters=10)
```

```
def infer_cluster_labels(kmeans, actual_labels):
    """
    Associates most probable label with each cluster in KMeans model
    returns: dictionary of clusters assigned to each label
    """

    inferred_labels = {}

    for i in range(kmeans.n_clusters):

        # find index of points in cluster
        labels = []
        index = np.where(kmeans.labels_ == i)

        # append actual labels for each point in cluster
        labels.append(actual_labels[index])

        # determine most common label
        if len(labels[0]) == 1:
            counts = np.bincount(labels[0])
        else:
            counts = np.bincount(np.squeeze(labels))

        # assign the cluster to a value in the inferred_labels dictionary
        if np.argmax(counts) in inferred_labels:
            # append the new number to the existing array at this slot
            inferred_labels[np.argmax(counts)].append(i)
        else:
            # create a new array in this slot
            inferred_labels[np.argmax(counts)] = [i]

        #print(labels)
        #print('Cluster: {}, label: {}'.format(i, np.argmax(counts)))

    return inferred_labels

def infer_data_labels(X_labels, cluster_labels):
    """
    Determines label for each array, depending on the cluster it has been assigned to.
    returns: predicted labels for each array
    """

    # empty array of len(X)
    predicted_labels = np.zeros(len(X_labels)).astype(np.uint8)

    for i, cluster in enumerate(X_labels):
        for key, value in cluster_labels.items():
            if cluster in value:
                predicted_labels[i] = key

    return predicted_labels


# test the infer_cluster_labels() and infer_data_labels() functions
cluster_labels = infer_cluster_labels(kmeans, Y)
X_clusters = kmeans.predict(X)
predicted_labels = infer_data_labels(X_clusters, cluster_labels)
print(predicted_labels[:20])
print(Y[:20])
```

```
    [8 0 4 1 9 2 1 8 1 7 3 1 3 6 1 7 2 8 6 7]
    [5 0 4 1 9 2 1 3 1 4 3 5 3 6 1 7 2 8 6 9]
```

```
from sklearn import metrics

def calculate_metrics(estimator, data, labels):

    # Calculate and print metrics
    print('Number of Clusters: {}'.format(estimator.n_clusters))
    print('Inertia: {}'.format(estimator.inertia_))
    print('Homogeneity: {}'.format(metrics.homogeneity_score(labels, estimator.labels_), average='weighted'))
```

```
clusters_first_pack = [10, 32, 64]
clusters_second_pack = [256, 512, 1024]

def params_and_metrics(pack):
    # test different numbers of clusters
    for n_clusters in pack:
        estimator = MiniBatchKMeans(n_clusters = n_clusters)
        estimator.fit(X)

        # print cluster metrics
        calculate_metrics(estimator, X, Y)

        # determine predicted labels
        cluster_labels = infer_cluster_labels(estimator, Y)
        predicted_Y = infer_data_labels(estimator.labels_, cluster_labels)

        # calculate and print metrics
        print('Accuracy: {}'.format(metrics.accuracy_score(Y, predicted_Y)))
        print('Recall: {}'.format(metrics.recall_score(Y, predicted_Y, average='micro')))
        print('Precision: {}'.format(metrics.precision_score(Y, predicted_Y, average='micro')))
        print('F-measure: {}\n'.format(metrics.f1_score(Y, predicted_Y, average='weighted')))


params_and_metrics(clusters_first_pack)

    Number of Clusters: 10
    Inertia: 2397048.216262334
    Homogeneity: 0.4516858598388123
    Accuracy: 0.5633
    Recall: 0.5633
    Precision: 0.5633
    F-measure: 0.5167827205827648

    Number of Clusters: 32
    Inertia: 1996547.3443251778
    Homogeneity: 0.6881892978649621
    Accuracy: 0.7737833333333334
    Recall: 0.7737833333333334
    Precision: 0.7737833333333334
    F-measure: 0.770412870572983

    Number of Clusters: 64
    Inertia: 1803543.0406960966
    Homogeneity: 0.7401452106788614
    Accuracy: 0.8095333333333333
    Recall: 0.8095333333333333
    Precision: 0.8095333333333333
    F-measure: 0.8072613065160603



params_and_metrics(clusters_second_pack)

    Number of Clusters: 256
    Inertia: 1504261.5917454856
    Homogeneity: 0.8408049476509997
    Accuracy: 0.8935333333333333
    Recall: 0.8935333333333333
    Precision: 0.8935333333333333
    F-measure: 0.8933432160949372

    Number of Clusters: 512
    Inertia: 1375370.2614004884
    Homogeneity: 0.8799979338618343
    Accuracy: 0.9236166666666666
    Recall: 0.9236166666666666
    Precision: 0.9236166666666666
    F-measure: 0.9236462865431903

    Number of Clusters: 1024
    Inertia: 1253919.6686961018
    Homogeneity: 0.9008438873915924
    Accuracy: 0.9315
    Recall: 0.9315
    Precision: 0.9315
    F-measure: 0.9314339844468495



# As you can see, as the number of clusters increases, the performance of the model also improves, but after 256 clusters, model fitt
# We will focus on 1024 clusters as the most efficient number of those considered.

# test kmeans algorithm on testing dataset
# convert each image to 1 dimensional array
X_test = x_test.reshape(len(x_test),-1)
```

```
# normalize the data to 0 - 1
X_test = X_test.astype(float) / 255.

# initialize and fit KMeans algorithm on training data
kmeans = MiniBatchKMeans(n_clusters = 1024)
kmeans.fit(X)
cluster_labels = infer_cluster_labels(kmeans, Y)

# predict labels for testing data
test_clusters = kmeans.predict(X_test)
predicted_labels = infer_data_labels(kmeans.predict(X_test), cluster_labels)

# calculate and print metrics
print('Accuracy: {}'.format(metrics.accuracy_score(y_test, predicted_labels)))
print('Recall: {}'.format(metrics.recall_score(y_test, predicted_labels, average='micro')))
print('Precision: {}'.format(metrics.precision_score(y_test, predicted_labels, average='micro')))
print('F-measure: {}\n'.format(metrics.f1_score(y_test, predicted_labels, average='weighted')))
```

```
Accuracy: 0.9345
Recall: 0.9345
Precision: 0.9345
F-measure: 0.9343987205837733
```

```
# perceptron by sklearn

from sklearn import datasets
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import Perceptron      #Single-layer perceptron
from sklearn.neural_network import MLPClassifier #Multilayer perceptron
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
from sklearn.metrics import recall_score
from sklearn.metrics import precision_score
from sklearn.metrics import f1_score


mnist = datasets.load_digits()
n_samples = len(mnist.images)
X = mnist.images.reshape((n_samples, -1))
y = mnist.target
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3)


# Train the scaler, which standarizes all the features to have mean=0 and unit variance
sc = StandardScaler()
sc.fit(X_train)
```

```
StandardScaler()
```

```
# Apply the scaler to the X training data
X_train_std = sc.transform(X_train)


# Apply the SAME scaler to the X test data
X_test_std = sc.transform(X_test)


# Create a single-layer perceptron object with the parameters: 40 iterations (epochs) over the data, and a learning rate of 0.1
ppn = Perceptron(max_iter=40, eta0=0.1, random_state=0)
# Create a multilayer perceptron object
mppn = MLPClassifier(solver='lbfgs', alpha=1e-5, hidden_layer_sizes=(256, 512, 128), random_state=1)


# Train the perceptrons
ppn.fit(X_train_std, y_train)
mppn.fit(X_train_std, y_train)
```

```
MLPClassifier(alpha=1e-05, hidden_layer_sizes=(256, 512, 128), random_state=1,
              solver='lbfgs')
```

```
# Apply the trained perceptrons on the X data to make predicts for the y test data
y_pred = ppn.predict(X_test_std)
multi_y_pred = mppn.predict(X_test_std)


# View the accuracies of the model
print('Single-layer perceptron accuracy: %.4f' % accuracy_score(y_test, y_pred))
print('Single-layer perceptron recall: %.4f' % recall_score(y_test, y_pred, average='micro'))
print('Single-layer perceptron precision: %.4f' % precision_score(y_test, y_pred, average='micro'))
print('Single-layer perceptron f-measure: %.4f\n' % f1_score(y_test, y_pred, average='micro'))
```

```
print('Multilayer perceptron accuracy: %.4f' % accuracy_score(y_test, multi_y_pred))
print('Multilayer perceptron recall: %.4f' % recall_score(y_test, multi_y_pred, average='micro'))
print('Multilayer perceptron precision: %.4f' % precision_score(y_test, multi_y_pred, average='micro'))
print('Multilayer perceptron f-measure: %.4f\n' % f1_score(y_test, multi_y_pred, average='micro'))
```

```
Single-layer perceptron accuracy: 0.9241
Single-layer perceptron recall: 0.9241
Single-layer perceptron precision: 0.9241
Single-layer perceptron f-measure: 0.9241

Multilayer perceptron accuracy: 0.9759
Multilayer perceptron recall: 0.9759
Multilayer perceptron precision: 0.9759
Multilayer perceptron f-measure: 0.9759
```

```
# neural network by keras

from tensorflow import keras
from keras.datasets import mnist
from keras.layers import Dense, Flatten


(x_train, y_train), (x_test, y_test) = mnist.load_data()
x_train = x_train / 255
x_test = x_test / 255


y_train_cat = keras.utils.to_categorical(y_train, 10)
y_test_cat = keras.utils.to_categorical(y_test, 10)


# As you can already see, the use of recall, precision and f-score does not provide important additional information, so we will lim

model_28 = keras.Sequential([
    Flatten(input_shape=(28, 28, 1)),
    Dense(28, activation='relu'),
    Dense(10, activation='softmax')
])

model_128 = keras.Sequential([
    Flatten(input_shape=(28, 28, 1)),
    Dense(128, activation='relu'),
    Dense(10, activation='softmax')
])

model_784 = keras.Sequential([
    Flatten(input_shape=(28, 28, 1)),
    Dense(784, activation='relu'),
    Dense(10, activation='softmax')
])

model_28.compile(optimizer='adam',
loss='categorical_crossentropy',
metrics=['accuracy'])

model_128.compile(optimizer='adam',
loss='categorical_crossentropy',
metrics=['accuracy'])

model_784.compile(optimizer='adam',
loss='categorical_crossentropy',
metrics=['accuracy'])

model_28.fit(x_train, y_train_cat, batch_size=32, epochs=10, validation_split=0.2)
```

```
Epoch 1/10
1500/1500 [==============================] - 5s 3ms/step - loss: 0.4108 - accuracy: 0.8839 - val_loss: 0.2203 - val_accuracy: 0
Epoch 2/10
1500/1500 [==============================] - 4s 3ms/step - loss: 0.2025 - accuracy: 0.9413 - val_loss: 0.1844 - val_accuracy: 0
Epoch 3/10
1500/1500 [==============================] - 4s 3ms/step - loss: 0.1632 - accuracy: 0.9514 - val_loss: 0.1593 - val_accuracy: 0
Epoch 4/10
1500/1500 [==============================] - 4s 2ms/step - loss: 0.1393 - accuracy: 0.9591 - val_loss: 0.1509 - val_accuracy: 0
Epoch 5/10
1500/1500 [==============================] - 4s 3ms/step - loss: 0.1234 - accuracy: 0.9634 - val_loss: 0.1394 - val_accuracy: 0
Epoch 6/10
1500/1500 [==============================] - 4s 2ms/step - loss: 0.1110 - accuracy: 0.9681 - val_loss: 0.1389 - val_accuracy: 0
Epoch 7/10
1500/1500 [==============================] - 4s 2ms/step - loss: 0.1017 - accuracy: 0.9699 - val_loss: 0.1388 - val_accuracy: 0
Epoch 8/10
1500/1500 [==============================] - 3s 2ms/step - loss: 0.0933 - accuracy: 0.9721 - val_loss: 0.1349 - val_accuracy: 0
Epoch 9/10
1500/1500 [==============================] - 3s 2ms/step - loss: 0.0861 - accuracy: 0.9742 - val_loss: 0.1267 - val_accuracy: 0
```

```
Epoch 10/10
1500/1500 [==============================] - 3s 2ms/step - loss: 0.0793 - accuracy: 0.9765 - val_loss: 0.1297 - val_accuracy: 0
<keras.callbacks.History at 0x7f631c677430>
```

```
model_128.fit(x_train, y_train_cat, batch_size=32, epochs=10, validation_split=0.2)
```

```
Epoch 1/10
1500/1500 [==============================] - 5s 3ms/step - loss: 0.2825 - accuracy: 0.9191 - val_loss: 0.1514 - val_accuracy: 0
Epoch 2/10
1500/1500 [==============================] - 5s 3ms/step - loss: 0.1274 - accuracy: 0.9623 - val_loss: 0.1127 - val_accuracy: 0
Epoch 3/10
1500/1500 [==============================] - 5s 3ms/step - loss: 0.0854 - accuracy: 0.9747 - val_loss: 0.1148 - val_accuracy: 0
Epoch 4/10
1500/1500 [==============================] - 5s 3ms/step - loss: 0.0633 - accuracy: 0.9812 - val_loss: 0.0895 - val_accuracy: 0
Epoch 5/10
1500/1500 [==============================] - 5s 3ms/step - loss: 0.0484 - accuracy: 0.9852 - val_loss: 0.0981 - val_accuracy: 0
Epoch 6/10
1500/1500 [==============================] - 5s 3ms/step - loss: 0.0384 - accuracy: 0.9887 - val_loss: 0.0863 - val_accuracy: 0
Epoch 7/10
1500/1500 [==============================] - 5s 3ms/step - loss: 0.0293 - accuracy: 0.9910 - val_loss: 0.0920 - val_accuracy: 0
Epoch 8/10
1500/1500 [==============================] - 5s 3ms/step - loss: 0.0233 - accuracy: 0.9928 - val_loss: 0.0887 - val_accuracy: 0
Epoch 9/10
1500/1500 [==============================] - 5s 3ms/step - loss: 0.0190 - accuracy: 0.9942 - val_loss: 0.0914 - val_accuracy: 0
Epoch 10/10
1500/1500 [==============================] - 5s 3ms/step - loss: 0.0164 - accuracy: 0.9955 - val_loss: 0.1097 - val_accuracy: 0
<keras.callbacks.History at 0x7f631c684a30>
```

```
model_784.fit(x_train, y_train_cat, batch_size=32, epochs=10, validation_split=0.2)
```

```
Epoch 1/10
1500/1500 [==============================] - 15s 10ms/step - loss: 0.2096 - accuracy: 0.9376 - val_loss: 0.1113 - val_accuracy:
Epoch 2/10
1500/1500 [==============================] - 14s 10ms/step - loss: 0.0847 - accuracy: 0.9734 - val_loss: 0.0963 - val_accuracy:
Epoch 3/10
1500/1500 [==============================] - 15s 10ms/step - loss: 0.0528 - accuracy: 0.9839 - val_loss: 0.0912 - val_accuracy:
Epoch 4/10
1500/1500 [==============================] - 14s 10ms/step - loss: 0.0369 - accuracy: 0.9878 - val_loss: 0.0806 - val_accuracy:
Epoch 5/10
1500/1500 [==============================] - 14s 10ms/step - loss: 0.0269 - accuracy: 0.9913 - val_loss: 0.0942 - val_accuracy:
Epoch 6/10
1500/1500 [==============================] - 14s 10ms/step - loss: 0.0205 - accuracy: 0.9937 - val_loss: 0.0889 - val_accuracy:
Epoch 7/10
1500/1500 [==============================] - 15s 10ms/step - loss: 0.0185 - accuracy: 0.9937 - val_loss: 0.1024 - val_accuracy:
Epoch 8/10
1500/1500 [==============================] - 14s 10ms/step - loss: 0.0128 - accuracy: 0.9957 - val_loss: 0.0933 - val_accuracy:
Epoch 9/10
1500/1500 [==============================] - 14s 10ms/step - loss: 0.0130 - accuracy: 0.9954 - val_loss: 0.0943 - val_accuracy:
Epoch 10/10
1500/1500 [==============================] - 14s 10ms/step - loss: 0.0105 - accuracy: 0.9962 - val_loss: 0.1061 - val_accuracy:
<keras.callbacks.History at 0x7f6318bffa90>
```

```
# The best indicators came from a neural network with 784 neurons on a hidden layer. Let's summarize the results:

# K-means accuracy: 0.9345; fit-time: 1 m 47 s

# Single-layer perceptron accuracy: 0.9241; fit-time: 3 s
# Multilayer perceptron accuracy: 0.9759; fit-time: 3 s

# Neural network accuracy: 0.9955; fit-time: 47 s
```

- Источники теоретических материалов:

Математические методы распознавания образов: НОУ ИНТУИТ

Обзор алгоритмов кластеризации данных

Алгоритм k-средних-1

Алгоритм k-средних-2

Алгоритм k-средних-3

Алгоритм k-средних-4

Алгоритм максимина-1

Алгоритм максимина-2

Алгоритм максимина-3

Алгоритм максимина-4

Алгоритм максимина-5

Нейронные сети, персептрон

Метод персептрона

Обработка данных

Градиент

Градиентный спуск

Стохастический градиентный спуск

Метод обратного распространения ошибки-1

Метод обратного распространения ошибки-2

Практики реализации нейронных сетей

Softmax

Введение в архитектуры нейронных сетей

Функции ошибок и переобучение

Нейронные сети для начинающих

Методы оптимизации нейронных сетей

Метрики в задачах машинного обучения

- Код:

MNIST by k-means

MNIST by perceptron

MNIST by neural network