МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ

Федеральное государственное бюджетное
образовательное учреждение высшего образования
«Тверской государственный университет»
Факультет прикладной математики и кибернетики
Направление 02.03.02 – «Фундаментальная информатика и информационные технологии»
Профиль подготовки: «Информатика и компьютерные науки»

Выпускная работа бакалавра

Тема: «Эволюционные алгоритмы в нейронных сетях»

Автор: студент 46 группы
Пурис Дмитрий Николаевич

Научный руководитель:
к.ф.-м.н., доцент
Дадеркин Дмитрий
Ольгердович

Допущен к защите:
Руководитель ООП:
_____ /Язенин А.В./
(подпись, дата)
Заведующий кафедрой: информатики
_____ /Дудаков С.М./
(подпись, дата)

Тверь, 2019

Оглавление

Введение

Актуальность

В последнее время современная техника становится настолько близка к человеку, что способна понимать его с полуслова. Трудно представить современный телефон или цифровую камеру без функции распознавания лиц. Различные рекламные сервисы , размещенные в интернете, настолько точно предоставляют нужную рекламу пользователю, насколько это возможно. Поисковые запросы в Google почти всегда дают самые нужные ссылки и информацию, основываясь на результатах работы нейронных сетей. Сегодня можно разговаривать с компьютером, и он будет разумно поддерживать диалог. Новейшие технологии в медицине позволяют предсказывать диагноз у пациентов. Все это стало возможным благодаря развитию машинного обучения, в том числе благодаря нейронным сетям и эволюционным алгоритмам. Концепция существует уже несколько десятилетий, но в последние годы приобрела огромную популярность благодаря передовым технологиям и аппаратному обеспечению.

Тема искусственного интеллекта на сегодняшний день является очень актуальной. В наши дни создаются основополагающие концепции и алгоритмы, связанные с машинным обучением. Не случайно самому распространенному алгоритму обучения нейронных сетей – алгоритму обратного распространения ошибки, нет даже и 10 лет. Также за последние 8 лет интерес к машинному обучению возрастает экспоненциально, что подтверждает официальная статистика Google. Сегодня существует определенный класс актуальных задач, решение которых без применения искусственных нейронных сетей (ИНС) невозможно или трудноосуществимо. Зачастую сюда относятся такие задачи, как классификация, прогнозирование и управление сложными системами. В последнее время быстро набирает популярность концепция глубокого обучения. По сути дела, в этой концепции нет ничего революционного, работы по изучению искусственных нейронных сетей ведутся с середины прошлого века, однако в последнее время уровень производительности персональных вычислительных средств и развитие параллельных вычислительных архитектур позволяет широкому кругу исследователей применять данные структуры более эффективно в области машинного обучения, это, в свою очередь, подстегивает очередной скачок интереса к нейронным сетям. Есть определенные основания полагать, что этот скачок окажется значительным и будет определять концепции развития техно-

логий машинного обучения в дальнейшем.

Цель работы

Разработать алгоритм, реализующий работу эволюционного алгоритма в нейронных сетях.

Задачи работы

- изучить обучающие алгоритмы;
- изучить виды нейронных сетей и методы их обучения;
- изучить применение эволюционных алгоритмов в нейронных сетях;
- разработать алгоритм, реализующий работу эволюционного алгоритма в нейронных сетях;

Основные результаты

Во время создания выпускной квалификационной работы были исследованы такие понятия как: нейронные сети; эволюционные алгоритмы; алгоритмы обучения нейронных сетей; обучение нейросетей с помощью эволюционных алгоритмов. Разработана нейронная сеть, обучающаяся с помощью эволюционного алгоритма, которая способна решать задачу классификации и давать правильный ответ.

Структура работы

В первой главе ВКР будут рассмотрены нейронные сети, их виды, применение и способы обучения. Вторая глава будет про эволюционные алгоритмы и применение их в нейронных сетях.В третьей главе пойдет речь о результатах работы, о разработанном программном обеспечении.

Глава 1

Нейронные сети

1.1 Применение нейронных сетей

Нейронные сети используются для решения сложных задач, которые требуют аналитических вычислений подобных тем, что делает человеческий мозг. Самыми распространенными применениями нейронных сетей являются:

Классификация — распределение данных по параметрам. Например, на вход дается набор людей и нужно решить, кому из них давать кредит, а кому нет. Эту работу может сделать нейронная сеть, анализируя такую информацию как: возраст, платежеспособность, кредитная история и т.д.

Предсказание — возможность предсказывать следующий шаг. Например, рост или падение акций, основываясь на ситуации на фондовом рынке.

Распознавание — в настоящее время, самое широкое применение нейронных сетей. Используется в Google, когда пользователи ищут фото или в камерах телефонов, когда оно определяет положение вашего лица и выделяет его и многое другое.

1.2 Основные определения

Алгоритм — Совокупность последовательных шагов, схема действий, приводящих к желаемому результату.

Генетический алгоритм — это эвристический алгоритм поиска, используемый для решения задач оптимизации и моделирования путём случайного подбора, комбинирования и вариации искомых параметров с использованием механизмов, аналогичных естественному отбору в природе. Популяция – это конечное множество особей.

Нейрон — узел искусственной нейронной сети, являющийся упрощённой моделью естественного нейрона. Математически, искусственный нейрон обычно представляют как некоторую нелинейную функцию от единственного аргумента — линейной комбинации всех входных сигналов. Данную функцию называют функцией активации или функцией срабатывания, передаточной функцией. Полученный результат посылается на единственный выход. Такие искусственные нейроны объединяют в сети — соединяют выходы одних нейронов с входами других.

Искусственная нейронная сеть (ИНС) — математическая модель, а также её программное или аппаратное воплощение, построенная по принципу организации и функционирования биологических нейронных сетей — сетей нервных клеток живого организма. После разработки алгоритмов обучения получаемые модели стали использовать в практических целях: в задачах прогнозирования, для распознавания образов, в задачах управления и т.д.

Эпоха — одна итерация в процессе обучения, включающая предъявление всех примеров из обучающего множества и, возможно, проверку качества обучения на контрольном множестве.

Алгоритм обратного распространения ошибки (Back propagation) — один из методов обучения многослойных нейронных сетей прямого распространения, называемых также многослойными персептронами.

1.3 Виды нейронных сетей

.

Нейронные сети прямого распространения (feed forward neural networks, FF или **FFNN)** и **перцептроны (perceptrons)** очень прямолинейны, они передают информацию от входа к выходу. Это означает, что в сети нет петель - информация всегда передается, никогда не возвращается. Если в сети были циклы, то получилась бы ситуация, когда вход в функцию зависит от выхода. Нейроны одного слоя не связаны между собой, а соседние слои обычно полностью связаны. Самая простая нейронная сеть имеет две входных клетки и одну выходную, и может использоваться в качестве модели логических вентилей. FFNN обычно обучается по методу обратного распространения ошибки, в котором сеть получает множества входных и выходных данных. Этот процесс называется обучением с учителем, и он отличается от обучения без учителя тем, что во втором случае множество выходных данных сеть составляет самостоятельно. Практически такие сети используются редко, но их часто комбинируют с другими типами для получения новых.

Однако есть и другие модели искусственных нейронных сетей, в которых возможны петли обратной связи. Эти модели называются **рекуррентными нейронными сетями**. Идея в этих моделях состоит в том, чтобы иметь нейроны, которые срабатывают в течение некоторого ограниченного периода времени, прежде чем станут спокойными. Это срабатывание может стимулировать другие нейроны, которые могут срабатывать немного позже, также в течение ограниченного периода времени. Это вызывает запуск еще большего количества нейронов, и поэтому со временем получается каскад запуска нейронов. Циклы не вызывают проблем в такой модели, поскольку выход нейрона влияет только на его вход через некоторое время, а не мгновенно.

Рекуррентные нейронные сети были менее влиятельными, чем сети с прямой связью, отчасти потому, что алгоритмы обучения для рекуррентных сетей (по крайней мере на сегодняшний день) менее эффективны. Но повторяющиеся сети все еще чрезвычайно интересны. По духу они гораздо ближе к тому, как работает наш мозг, чем к сетям прямой связи. И возможно, что повторяющиеся сети могут решить важные проблемы, которые могут быть решены с большим трудом только через сети прямой связи.

Нейронная сеть Хопфилда (Hopfield network, HN) — это полносвязная нейронная сеть с симметричной матрицей связей. Во время получения входных данных каждый узел является входом, в процессе обучения он становится скрытым, а затем становится выходом. Сеть обучается так: значения нейронов устанавливаются в соответствии с желаемым шаблоном, после чего вычисляются веса, которые в дальнейшем не меняются. После того, как сеть обучилась на одном или нескольких шаблонах, она всегда будет сводиться к одному из них (но не всегда — к желаемому). Она стабилизируется в зависимости от общей “энергии” и “температуры” сети. У каждого нейрона есть свой порог активации, зависящий от температуры, при прохождении которого нейрон принимает одно из двух значений (обычно -1 или 1, иногда 0 или 1). Такая сеть часто называется сетью с ассоциативной памятью; как человек, видя половину таблицы, может представить вторую половину таблицы, так и эта сеть, получая таблицу, наполовину зашумленную, восстанавливает её до полной.

Цепи Маркова (Markov chains, MC или discrete time Markov Chains, DTMC) — это предшественники машин Больцмана (BM) и сетей Хопфилда (HN). Их смысл можно объяснить так: каковы мои шансы попасть в один из следующих узлов, если я нахожусь в данном? Каждое следующее состояние зависит только от предыдущего. Хотя на самом деле цепи Маркова не являются НС, они весьма похожи. Также цепи Маркова не обязательно полносвязны.

Свёрточные нейронные сети (convolutional neural networks, CNN) и глубинные свёрточные нейронные сети (deep convolutional neural networks, DCNN) - тут надо их описать

1.4 Обучающие алгоритмы

Линейная регрессия - Алгоритм линейной регрессии используется для оценки реальных значений (стоимость домов, количество звонков, общий объем продаж и.т.д.). На основе непрерывных переменных. Здесь устанавливается связь между независимыми и зависимыми переменными, подбирая лучшую прямую. Эта прямая наилучшего соответствия называется линией регрессии и представлена линейным уравнением Y = a * X + b. Линейная регрессия бывает двух типов: простая линейная регрессия и множественная линейная регрессия. Простая линейная регрессия характеризуется одной независимой переменной. И, множественная линейная регрессия характеризуется множеством независимых переменных. Найдя наиболее подходящую прямую, можно подобрать полиномиальную или криволинейную регрессию.

Логистическая регрессия - Логистическая регрессия - это статистическая модель, используемая для прогнозирования вероятности возникновения некоторого события путём подгонки данных к логистической кривой. Он используется для оценки дискретных значений (двоичные значения, такие как 0/1, да / нет, истина / ложь) на основе заданного набора независимых переменных. Этот алгоритм предсказывает вероятность возникновения события путем подгонки данных к функции logit . Следовательно, это также известно как регрессия логита . Поскольку он предсказывает вероятность, его выходные значения лежат между 0 и 1.

Наивная байесовская классификация - Это метод классификации, основанный на теореме Байеса с предположением независимости между предикторами. Наивный байесовский классификатор предполагает, что наличие определенной функции в классе не связано с наличием любой другой функции. Например, фрукт можно считать яблоком, если оно красное, круглое и около 3 дюймов в диаметре. Даже если эти признаки зависят друг от друга или от наличия других признаков, наивный байесовский классификатор будет рассматривать все эти свойства независимо, чтобы повысить вероятность того, что этот фрукт является яблоком. Наивная байесовская модель проста в построении и особенно полезна для очень больших наборов данных. Известно, что наряду с простотой наивный байесовский метод превосходит даже самые сложные ме-

тоды классификации.

1.5 Перцептрон

В работе будет использоваться нейронная сеть, основанная на модели перцептрона. Данная модель выбрана по нескольким критериям: во-первых она достаточно проста в изучении, во-вторых данная модель подходит для решения задачи, в-третьих модель перцептрона имеет простую реализацию на языке Python. Перцептроны передают информацию от входа к выходу. Нейронные сети часто описываются в виде слоев нейронов, где каждый слой состоит из входных, скрытых или выходных клеток. Клетки одного слоя не связаны между собой, а соседние слои обычно полностью связаны (Рис. 1.1).

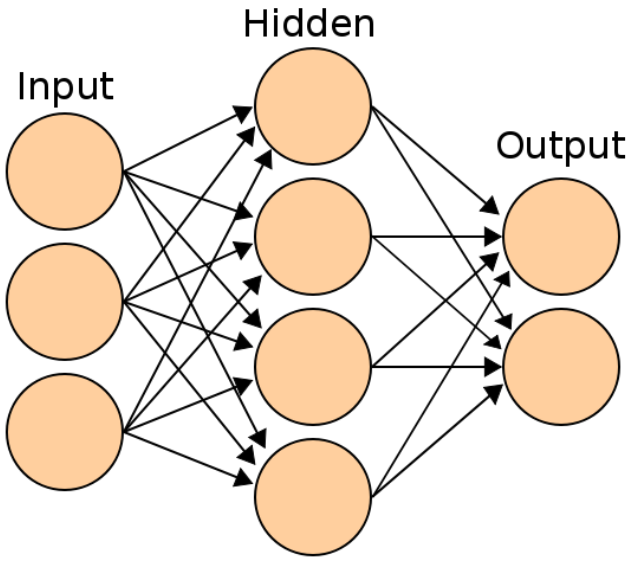

Рис. 1.1: Нейронная сеть

Самая простая нейронная сеть имеет две входных клетки и одну выходную, и может использоваться в качестве модели логических вентилей. Нейронная сеть прямого распространения обычно обучается по методу обратного распространения ошибки, в котором модель получает множества входных и выходных данных. Этот процесс называется обучением с учителем, и он отличается от обучения без учителя тем, что во втором случае множество выходных данных сеть составляет самостоятельно.

Модель перцептрона была предложена и популяризована Ф. Розенблаттом. В начале 1960-х. Розенблатт верил в свою модель, и надеялся, что с помощью перцептрона человечество продвинется в создании искусственного интеллекта. В те времена была большая надежда на эту модель. Но в 1969 году Марвин Минский и Сеймур Паперт выпустили книгу, в которой рассматривали вычислительные способности перцептронов – то, чему могут учиться и чему не мо-

гут учиться перцептроны. И в том числе показывали некоторые ограничения модели. И, как часто это бывает, завышенные ожидания сменились чрезмерным разочарованием. В итоге критика, которая заключалась в доказательстве ограничения такой модели как перцептрон была обобщена на все нейросетевые модели и привела к застою в исследовании данной области на долгие годы.

Обучение перцептрона заключается в том, что ему «показывают» примеры и правильные ответы для этих примеров. Сперва веса перцептрона инициализируются случайно. И поэтому перцептрон, когда ему на вход поступает какой-нибудь пример, дает случайный ответ. Но мы бы хотели, чтобы перцептрон от случайных весов перешел к «осмысленным» весам, а значит, и к осмысленным ответам.

Перцептрон (линейный нейрон) – это оператор вида:

$$f(\boldsymbol{x}, \boldsymbol{w}, b) = \begin{cases} 1, \text{ если } \sum_{i=1}^{n} \boldsymbol{w_i x_i} + b > 0 \\ 0 \text{иначе} \end{cases} \tag{1}$$

Где $f(\boldsymbol{x}, \boldsymbol{w}, b)$ - активационная функция;

$\sum_{i=1}^{n} \boldsymbol{w_i x_i} + b > 0$ - сумматорная функция;

$\boldsymbol{x}$ - вектор входных активаций;

$\boldsymbol{w}$ - вектор весов;

b - смещение;

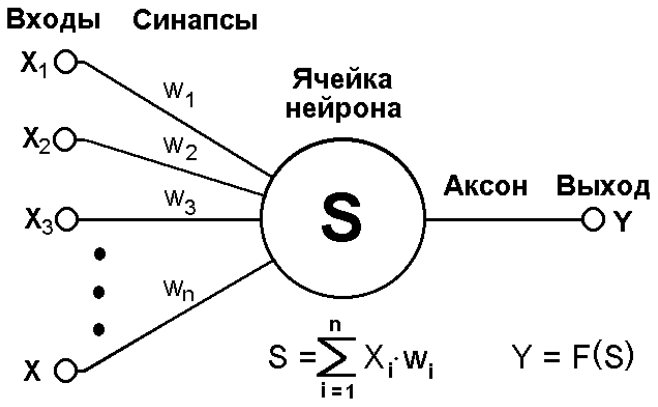

Рис. 1.2: Нейрон

Модель перцептрона принимает несколько вещественных входных данных и дает один двоичный выход. В модели перцептрона каждый вход x_i имеет вес, w_i связанный с ним. Веса указывают на важность ввода в процессе принятия решений. Выход модели определяется сумматорной функцией, если сумматорная функция больше нуля, то выход будет равен 1, иначе выход будет равен 0. Другими словами, модель будет срабатывать, если взвешенная сумма больше порога. Такой линейный нейрон способ решать лишь ряд простых задач, для

которых хватает линейной функции, например, предсказание стоимости квартиры от ее площади или роста собаки от ее веса.

К неприятным особенностям линейного нейрона можно отнести то, что небольшое изменение весов или смещений любого отдельного персептрона в сети может иногда приводить к тому, что выход этого персептрона полностью переворачивается, например, от 0 до 1. Этот скачок может сильно сказаться на поведении остальной сети.

1.5.1 Квадратичная целевая функция

$$J = \frac{1}{2}\sum_{i=1}^{n}(\hat{y}^{(i)} - y^{(i)})^2 \tag{2}$$

Где n - количество примеров на обучающей выборке
$\hat{y}^{(i)} = \sigma(w^T x^{(i)})$ - предсказанное моделью значение
$y^{(i)}$ - реальное значение целевой переменной на конкретном примере

Функцией потерь в данном случаю будет выражение $(\hat{y}^{(i)} - y^{(i)})^2$. В зарубежной литературе функция потерь пишется как *loss function* или *cost function*. Данная функция была выбрана потому что она проста и понятна. В дальнейшем удобство функции будет заметно при нахождении частных производных. Если взглянуть на функцию внимательно, можно заметить что она зависит от двух аргументов: с одной стороны от данных($\hat{y}^{(i)}$), с другой стороны от параметров модели($y^{(i)}$). Данные - это то, что фиксировано, а параметры - то, что будет меняться. Задача заключается в подборке таких параметров, на которых модель будет хорошо работать.

Целевая функция показывает на сколько сильно модель ошибалась. Чем меньше модель ошибается, тем лучше она справляется со своей задачей. Значит необходимо уменьшить эту ошибку. Для уменьшения ошибки хорошо подходит градиент, показывающий направление наибольшего роста функции. А значит отрицание градиента будет показывать нискорейшее убывание функции.

Данное понимание градиента можно применить к обучению линейного нейрона. Один шаг алгоритма будет уменьшать веса модели:

$$w_{j+1} = w_j - \alpha\nabla J \tag{3}$$

В данной формуле w_j - это текущий вес, α - коэффициент обучения(*learning rate*), а ∇J - градиент целевой функции. Этот шаг продолжается до тех пор, пока веса не перестанут меняться(*с какой-то точностью*), либо пока количество итераций не достигнет заданного максимума.

Дальнейшие вычисления удобно вести в матричном и векторном виде. Для такого же удобного использования выведенных формул будет использоваться библиотека NumPy для языка Python.

1.5.2 Обучение линейного нейрона

Чтобы обучать нейрон надо знать градиент целевой функции(∇J). В данном разделе будет выведена формула для ∇J. Значение целевой переменной будет находиться по формуле $\hat{y}^{(i)} = w^T x^{(i)}$. Где $x^{(i)}$ - i-ый пример из множества обучающей выборки.

$$\frac{\partial J}{\partial w_j} = \frac{\partial\Big(\frac{1}{2}\sum_{i=1}^{n}(\hat{y}^{(i)} - y^{(i)})^2\Big)}{\partial w_j} = \sum_{i=1}^{n}\frac{\partial\Big(\frac{1}{2}(\hat{y}^{(i)} - y^{(i)})^2\Big)}{\partial w_j}$$

Выполним следующий шаг:

$$\sum_{i=1}^{n}\frac{\partial\Big(\frac{1}{2}(\hat{y}^{(i)} - y^{(i)})^2\Big)}{\partial w_j} = \sum_{i=1}^{n}\frac{\partial\Big(\frac{1}{2}(\hat{y}^{(i)} - y^{(i)})^2\Big)}{\partial \hat{y}^{(i)}} \cdot \frac{\partial \hat{y}^{(i)}}{\partial w_j} \tag{4}$$

Вычисляем частную производную:

$$\frac{\partial\Big(\frac{1}{2}(\hat{y}^{(i)} - y^{(i)})^2\Big)}{\partial \hat{y}^{(i)}} = \frac{1}{2}\cdot 2 \cdot (\hat{y}^{(i)} - y^{(i)})^2 = \hat{y}^{(i)} - y^{(i)} \tag{5}$$

Так как $\hat{y}^{(i)} = w^T x^{(i)}$, то получаем:

$$\frac{\partial \hat{y}^{(i)}}{\partial w_j} = \frac{\partial(w_1 x_1^{(i)} + w_2 x_2^{(i)} + \ldots + w_j x_j^{(i)} + \ldots + w_m x_m^{(i)})}{\partial w_j} = \frac{\partial w_j x_j^{(i)}}{w_j} = x_j^{(i)} \tag{6}$$

Объединяем результаты (5) и (6) и получаем:

$$\frac{\partial J}{\partial w_j} = \sum_{i=1}^{n}(\hat{y}^{(i)} - y^{(i)}) \cdot x_j^{(i)} \tag{7}$$

Градиент целевой функции будет выглядеть следующим образом:

$$\nabla J = \begin{bmatrix} \ldots \\ (\hat{y}^{(i)} - y^{(i)}) \cdot x_j^{(i)} \\ (\hat{y}^{(i)} - y^{(i)}) \cdot x_{j+1}^{(i)} \\ \ldots \end{bmatrix} \tag{8}$$

Суммарная ошибка вычисляется по формуле (2). Средняя ошибка имеет следующий вид:

$$J = \frac{1}{2n}\sum_{i=1}^{n}(\hat{y}^{(i)} - y^{(i)})^2 \tag{9}$$

1.6 Сигмоидальный нейрон

Для решения более сложных задач используются нейроны с непрерывными дифференцируемыми активационными функциями, которые не являются линейными. Один из таких нейронов – сигмоидальный нейрон.

Введение функций сигмоидального типа было обусловлено ограниченностью нейронных сетей с пороговой функцией активации нейронов — при такой функции активации любой из выходов сети равен либо нулю, либо единице, что ограничивает использование сетей в задачах, не связанных с классификацией. Использование сигмоидальных функций позволило перейти от бинарных выходов нейрона к непрерывным. Функции передачи такого типа, как правило, присущи нейронам, находящимся во внутренних слоях нейронной сети.

Сигмоидальный нейрон описывается математически следующим образом:

$$f(\boldsymbol{x}, \boldsymbol{w}, b) = \sigma(\boldsymbol{w}\boldsymbol{x} + b)$$

Где $\sigma(x) = \dfrac{1}{1 + e^{-x}}$ - логистическая функция (Рис 1.3)

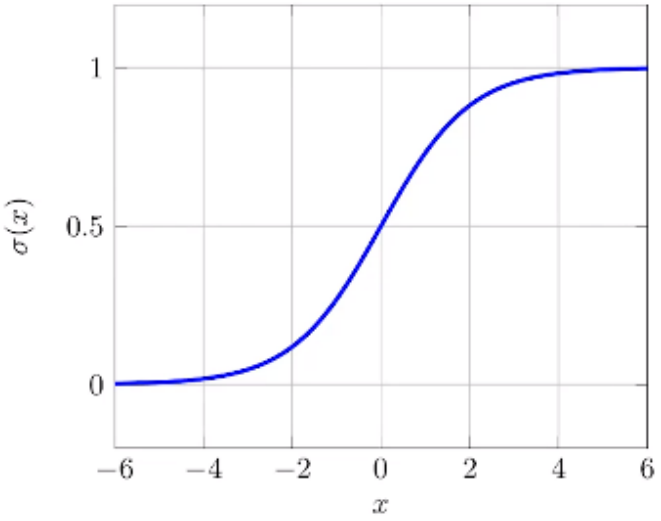

Рис. 1.3: График логистической функции

Особенностью нейронов с логистической функцией является то, что они усиливают сильные сигналы существенно меньше, чем слабые, поскольку области сильных сигналов соответствуют пологим участкам характеристики. Это позволяет предотвратить насыщение от больших сигналов.

Сигмовидные нейроны похожи на персептроны, но модифицированы так, что небольшие изменения их веса и смещения вызывают только небольшое изменение их выхода. Это решающий факт, который позволит сети сигмовидных нейронов учиться.

1.6.1 Вывод правил обучения для логистического нейрона

Используя данные полученные в главе 1.5.2, выведем правила обучения для сигмоидального нейрона. Целевая функция остается такой же как и для линейного нейрона. Меняется только активационная функция, она будет иметь вид:

$$\hat{y}^{(i)} = \sigma(w^T x^{(i)}) \tag{10}$$

Следовательно меняется и частная производная:

$$\frac{\partial J^{(i)}}{\partial w_j} = \frac{\partial J^{(i)}}{\partial \hat{y}^{(i)}} \cdot \frac{\partial \hat{y}^{(i)}}{\partial S} \cdot \frac{\partial S}{\partial w_j} \tag{11}$$

Где $S = w^T x^{(i)}$ - сумматорная функция. Находим частные производные:

$$\frac{\partial J^{(i)}}{\partial \hat{y}^{(i)}} = \sigma(w^T x^{(i)}) - y^{(i)} \tag{12}$$

$$\frac{\partial \hat{y}^{(i)}}{\partial S} = \frac{\partial \sigma(w^T x^{(i)})}{\partial (w^T x^{(i)})} = \sigma(w^T x^{(i)}) \cdot (1 - \sigma(w^T x^{(i)})) \tag{13}$$

$$\frac{\partial S}{\partial w_j} = x_j^{(i)} \tag{14}$$

Используя результаты (12), (13) и (14), находим градиент функции потерь:

$$\nabla J = \frac{1}{n} \cdot \sum_{i=1}^{n} \left(\sigma(w^T x^{(i)}) - y^{(i)}\right) \cdot \sigma(w^T x^{(i)}) \cdot (1 - \sigma(w^T x^{(i)})) \cdot x_j^{(i)} \tag{15}$$

1.7 Стохастический градиентный спуск

Стохастический градиентный спуск (stochastic gradient descent, SGD) - 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 нескольких случайно выбранных элементов.

Алгоритм стохастического градиентного спуска:

- Случайным образом выбираем часть примеров из всех имеющихся
- Считаем $\hat{y}$ для каждого из них

- Вычисляем градиент целевой функции по весам для каждого из них
- Суммируем то, что получилось
- Обновляем веса
- Проверяем критерии остановки алгоритма. Если хотя бы один из них отработал - выходим из цикла

Реализованный алгоритм стохастического градиентного спуска на языке Python:

```
import numpy as np
import math

def SGD(self, X, y, batch_size, learning_rate, eps, max_steps):
    indexes = list(range(len(X)))
    for s in range(max_steps):
        sample_ind = np.random.choice(indexes, batch_size, replace = False)
        X_example = X[sample_ind]
        y_answer = y[sample_ind]

        res = self.update_mini_batch(X_example, y_answer, learning_rate, eps)
        if res:
            return 1
    return 0

def update_mini_batch(self, X, y, learning_rate, eps):
    nabla_j = compute_grad_analytically(self, X, y)
    j_old = J_quadratic(self, X, y)
    delta_w = -1 * learning_rate * nabla_j
    self.w += delta_w
    j_new = J_quadratic(self, X, y)

    if math.fabs(j_old - j_new) < eps:
        return 1
    else:
        return 0
```

Листинг 1.1: Стохастический градиентный спуск

1.8 Многослойный перцептрон Румельхарта

Многослойный перцептрон — частный случай перцептрона Розенблатта, в котором один алгоритм обратного распространения ошибки обучает все слои. Название по историческим причинам не отражает особенности данного вида перцептрона, то есть не связано с тем, что в нём имеется несколько слоёв (так как несколько слоёв было и у перцептрона Розенблатта). Особенностью является наличие более чем одного обучаемого слоя (как правило — два или три).

w^l_{jk} - вес, который соединяет нейрон с номером j из слоя с номером l с нейроном с номером k из слоя $l-1$

z^l_j - результат сумматорной функции нейрона j из слоя l

a^l_j - результат активационной функции нейрона j из слоя l

b^l_j - смещение нейрона j из слоя l

W^l - матрица весов входящих в нейроны в с номером слоя l

1.8.1 Алгоритм обратного распространения ошибки

Определим ошибку в нейроне с номером j в слое l следующим образом:

$$\delta^l_j = \frac{\delta J}{\delta z^l_j} \tag{16}$$

Тогда частная производная целевой функции по сумматорной будет иметь вид:

$$\frac{\delta J}{\delta z^l_j} = (\sigma(z_j) - \sigma^2(z_j)) \cdot (\sigma(z_j) - y_j) \tag{17}$$

.

Будем обозначать $\nabla_a J$ - градиент целевой функции по активациям входного слоя, а $\odot$ - произведение Адамара(*поэлементное умножение векторов*).

Следовательно из формул (16) и (17) можно получить вектор ошибок выходного слоя:

$$\delta_L = \nabla_a J \odot \sigma'(z^L) \tag{18}$$

Суть алгоритма обратного распространения ошибки - зная ошибку на слое с номером l, узнать ошибку на $l-1$ слое. Рассмотрим любой нейрон с номером j в слое l. Этот нейрон связан со всеми нейронами из слоя $l+1$. Соответственно, если увеличить значение его активационной функции, то это изменение домноженное на соответствующие веса пойдет в нейроны следующего слоя. Допустим значение активационной функции было a^l_j и стало $a^l_j + \Delta$. Можно узнать как изменится значение сумматорной функции каждого из нейронов следующего слоя. До изменения активация нейрона равнялась $a^l_j w^l_{jk} + C$, где C - константа, а после добавления Δ получаем, что активация изменилась на

$$\Delta \cdot w^l_{jk} \tag{19}$$

. Известно, что изменение функции равно произведению градиента на разницу вектора аргументов $\Delta J \approx \nabla J \cdot \Delta x$. Если вектор изменения входных активаций слоя $l+1$ равен $(\Delta \cdot w^{l+1}_{1k}, \ldots, \Delta \cdot w^{l+1}_{n_{l+1}k})$ и Δ мало, то тогда получаем:

$$\Delta J = \Delta \cdot (W^{l+1})^T_k \cdot \delta^{l+1}$$

Где $(W^{l+1})^T_k$ - k-й столбец матрицы весов между слоем l и $l+1$.

Выразим ошибку δ^l через ошибку δ^{l+1}. Из формулы (18) известно, что

$$\delta_l = \nabla_{a^l} J \odot \sigma'(z^l) \tag{20}$$

Осталось выразить $\nabla_{a^l} J$ через δ^{l+1}. $\nabla_{a^l} J$ состоит из частной производной $\frac{\partial J}{\partial a_j^l}$.

$$\Delta J = \Delta \cdot (W^{l+1})_k^T \cdot \delta^{l+1} \tag{21}$$

Где $(W^{l+1})_k^T$ - это k-й столбец матрицы W^{l+1}. Следовательно, $\frac{\partial J}{\partial a_i^l} = (W^{l+1})_k^T \cdot \delta^{l+1}$. Из формулы (21) получаем, что $\nabla_{a^l} J = (W^{l+1})_k^T \cdot \delta^{l+1}$. Итого:

$$\delta^l = ((W^{l+1})^T \cdot \delta^{l+1}) \odot \sigma'(z^l) \tag{22}$$

Сумматорную функцию нейрона j в слое l можно выразить через активационную функцию нейронов из слоя $l-1$.$z_j^l = W_j^l \cdot a_{l-1} + b_j^l$

Тогда частняя производная целевой функции по весу $\frac{\partial J}{\partial W_{jk}^l}$ будет равна $a_k^{l-1} \cdot \delta_j^l$, а по смещению: $\frac{\partial J}{\partial w_{jk}^l} = \delta_j^l$.

Четыре кита backpropogation

Все нужные формулы получены. Теперь можно выделить 4 основных, которые будут участвовать в алгоритме обратного распространения ошибки:

1) $\delta_L = \nabla_a J \odot \sigma'(z^L)$

2) $\delta^l = ((W^{l+1})^T \cdot \delta^{l+1}) \odot \sigma'(z^l)$

3) $\frac{\partial J}{\partial w_{jk}^l} = \delta_j^l$

4) $\frac{\partial J}{\partial W_{jk}^l} = a_k^{l-1} \cdot \delta_j^l$

Первая формула из списка помогает получить вектор ошибок выходного слоя, зная градиент целевой функции по активациям нейронов выходного слоя $\nabla_a J$. Хочу заметить, что хоть σ' записывается как производная сигмоидальной функции, однако, на самом деле, это не обязательно сигмоида. На ее месте может быть любая подходящая активационная функция.

Второй кит помогает, зная ошибки в слое с номером l, вычислить ошибки в слое $l-1$. Это делает возможным осуществить шаг алгоритма с дальнего слоя к ближнему.

Третья и четвертая формула, зная ошибки, вычисленные на (1) и (2) шаге, позволяют вычислить нужные производные(*по смещению и по любому из весов*).

Шаги алгоритма обратного распространения ошибки

Используя формулы из предыдущего пункта, получаем алгоритм обратного распространения ошибки:

- Выбрать m примеров, на основании которых будет производиться изменение весов
- Осуществить прямое распространение активации для каждого примера
- Использовать первого кита, чтобы посчитать ошибки нейронов в выходном слое по каждому примеру
- Использовать второго кита - теперь есть ошибки нейронов во всех слоях, по каждому примеру
- Использовать третьего и четвёртого кита, теперь есть градиенты по параметрам и смещениям
- Обновить параметры
- Проверить критерии остановки алгоритма

Нормализация

Для правильно работы алгоритма обратного распространения ошибки важно чтобы веса были разные. Действительно, если все веса одинаковые, то все нейроны из первого скрытого слоя получат одинаковую активацию. Потому что каждый нейрон связан с каждым входом скрытого слоя. Соответственно, никакой разницы между нейронами первого скрытого слоя не существует. Эту проблему можно решить, инициализируя веса случайными значениями. Например, нормальный шум или равномерный шум.

Большие векторы весов, также как и больше векторы входных активаций могут привести к тому, что производная будет близка к нулю. Соответственно, можно получить состояние *ступора сети*. Если активации имеют разный масштаб, то и градиенты будут иметь разный масштаб. Поэтому очень важно правильно нормализовать входы.

Глава 2

Эволюционные алгоритмы

2.1 Виды эволюционных алгоритмов

Эволюционные алгоритмы — это направление в компьютерных науках, использующее принципы биологической эволюции для решения задач искусственного интеллекта. Основной принцип биологической эволюции — это сочетание естественного отбора, мутаций и воспроизводства. Хотя эволюционные алгоритмы и пытаются имитировать биологическую эволюцию, они более схожи с искусственным разведением животных, то есть скрещиванием самых лучших представителей, отбором их лучших потомков и повторным скрещиванием уже этих потомков.

Виды алгоритмов

- генетические алгоритмы — эвристический алгоритм поиска, используемый для решения задач оптимизации и моделирования путём случайного подбора, комбинирования и вариации искомых параметров;
- генетическое программирование — автоматическое создание или изменение программ с помощью генетических алгоритмов;
- эволюционное программирование — аналогично генетическому программированию, но структура программы постоянна, изменяются только числовые значения;
- эволюционные стратегии — похожи на генетические алгоритмы, но в следующее поколение передаются только положительные мутации;
- нейроэволюция — аналогично генетическому программированию, но геномы представляют собой искусственные нейронные сети, в которых происходит эволюция весов при заданной топологии сети, или помимо эволюции весов также производится эволюция топологии;

2.2 Понятие естественного отбора

Процесс естественного отбора начинается с выбора наиболее приспособленных животных из популяции. Они производят потомство, которое наследует характеристики родителей и будет добавлено к следующему поколению. Если родители подготовлены лучше, то их дети имеют больше шансов на выживание. Этот процесс продолжает повторяться, и в конце, поколение с самыми подходящими животными будет найдено.

Это понятие может быть применено к проблеме обучения нейронной сети. В работе рассматривается набор решений для проблемы и выбирается из них набор лучших.

Пять фаз эволюционного алгоритма

- Начальная популяция
- Фитнес-функция
- Выбор
- Кроссовер
- Мутация

Начальная популяция

Процесс начинается с набора нейронный сетей. Каждая нейроная сеть(индивидуум) умеет решать поставленную задачу с каким-то уровнем успеха.

Индивидуум характеризуется набором параметров (переменных), известных как гены. Гены объединяются в цепочку, образуя хромосому.

В эволюционном алгоритме набор генов индивида представлен строкой в алфавитном порядке. Обычно используются двоичные значения (строка из 1 и 0). Обычно говорят, что таким образом кодируются гены в хромосоме.

Финтес-функция

Функция пригодности(финтес-функция) определяет , насколько подходит индивидуум (способность нейросети конкурировать с другими нейронными сетями). Это дает оценку пригодности для каждой сети. Вероятность того, что конкретная нейронная сеть будет выбрана для размножения, основана на ее оценке пригодности.

Идея фазы отбора состоит в том, чтобы отобрать наиболее подходящих особей и позволить им передать свои гены следующему поколению. Две пары индивидуумов (родителей) выбираются на основе их показателей пригодности. Нейронный сети с высоким процентом ответов имеют больше шансов быть отобранными для размножения.

Кроссовер

Кроссовер является наиболее значимой фазой в генетическом алгоритме. Для каждой пары родителей, которые должны быть спарены, точка пересечения (crossover point) выбирается случайным образом из генов. Например, рассмотрим точку пересечения 3, как показано ниже.

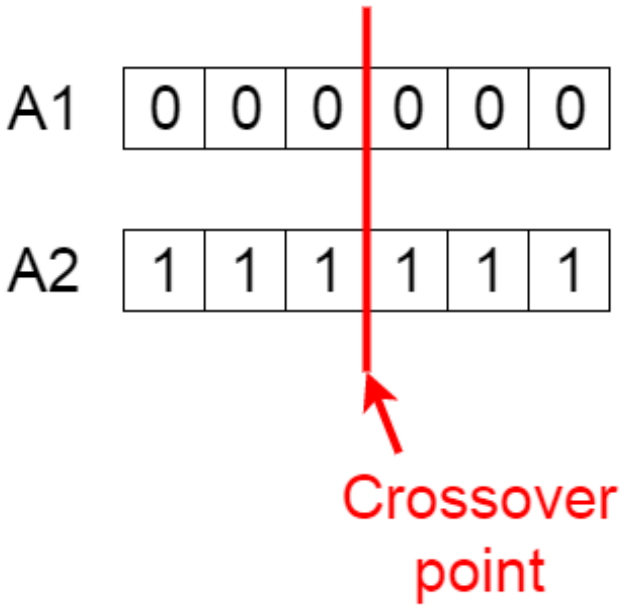

Рис. 2.1: Точка пересечения(crossover point)

Потомки создаются путем обмена генами родителей между собой, пока не будет достигнута точка пересечения.

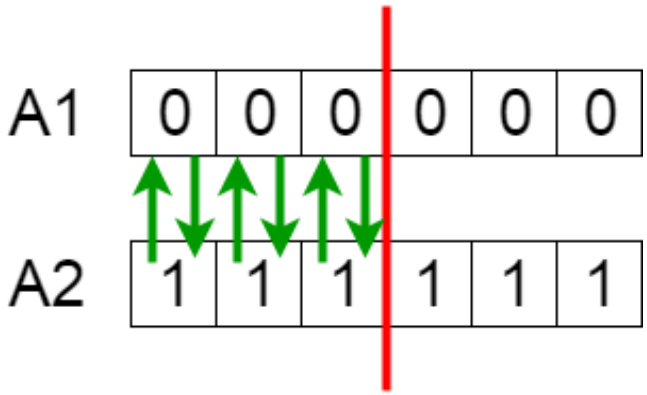

Рис. 2.2: Обмен генами

В итоге получаем новое потомство.

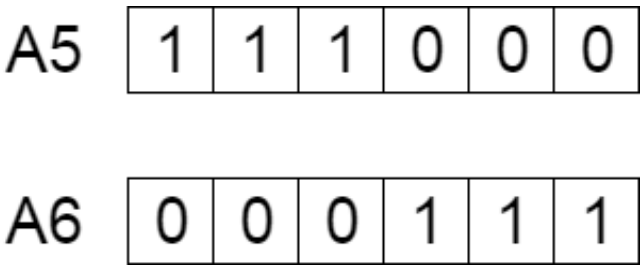

Рис. 2.3: Потомки

Мутация

У некоторых новых потомков отдельные гены могут подвергаться мутации с низкой случайной вероятностью. Это подразумевает, что некоторые биты в битовой строке могут быть перевернуты.

Рис. 2.4: Мутация

Мутация происходит для поддержания разнообразия в популяции и предотвращения преждевременной конвергенции. Алгоритм завершается, если популяция не производит потомство, которое значительно отличается от предыдущего поколения.

Глава 3

Разработанное ПО

Цель создания ПО

Программное обеспечение разрабатывалось в учебных целях для исследования работы эволюционных алгоритмов в нейронных сетях и тестирования его эффективности.

Выбор средств для создания ПО

Для написания программы использовался язык программирования Python в связке со средой разработки Jupyter Notebook. Этот язык был выбран для написания выпускной работы бакалавра в связи с большим количеством библиотек, в которых реализованы основные методы для работы с матрицами, строками и файлами. Среда разработки Jupyter Notebook позволяет удобно работать с кодом, подсвечивая синтаксис языка. Главной интересной функцией среды разработки является возможность компилировать код построчно, что позволяет увидеть ошибку сразу.

Описание созданного ПО

Нужно научиться решать такую задачу: Зная начальное направление (угол от поверхности земли) и скорость мяча, определить, где этот мяч упадёт. Или иначе: попадёт ли мяч в мишень, находящуюся на земле, при заданных начальных условиях.

Рис. 3.1: Удар по мячу

В данных у нас скорость удара, угол удара(угол между направлением полёта мяча и землёй) и расстояние до мишени. Скорость броска от 0 и до 50. Угол принимает значения из промежутка от 0 до $\frac{\pi}{2}$.

Загружаем файл с данными.

```
data = np.loadtxt("data.csv", delimiter=",")
```

Разница в масштабах настолько существенна, что имеет смысл нормализовать наши данные, чтобы при обучении не отвлекаться на попытки скомпенсировать масштаб данных масштабом весов.

```
means = data.mean(axis=0)
means[-1] = 0
stds = data.std(axis=0)
stds[-1] = 1
data = (data - means) / stds
```

Чтобы была возможность следить за успешностью обучения, сразу отделим часть данных в тестовое множество.

```
np.random.seed(42)
test_index = np.random.choice([True, False], len(data), replace=True, p=[0.25, 0.75])
test  = data[test_index]
train = data[np.logical_not(test_index)]
```

Приведем данные в тот вид, в котором они понимаются нейросетью. Для обучения нам нужно, чтобы ответ был в формате one-hot: вектор длины 3 (общее количество классов), состоящий из нулей и одной единицы на месте правильного класса наблюдения. Мы сделаем это с помощью np.eye: для единицы, стоящей на i-м месте, нужно создать вектор np.eye(3, 1, k=-i). Соответственно, когда мы будем итерироваться по нашим входным данным, искомое i для примера - это последний элемент строки с этим примером, то есть d(-1).

```
train = [(d[:3][:, np.newaxis], np.eye(3, 1, k=-int(d[-1]))) for d in train]
test =  [(d[:3][:, np.newaxis], d[-1]) for d in test]
```

Зададим параметры нейронной сети:

- inputCount - количество нейронов входного слоя
- hiddenCount - количество нейронов скрытого слоя
- outputCount - количество нейронов выходного слоя

- SIZE - размер популяции нейронных сетей

```
input_count  = 3
hidden_count = 7
output_count = 3
SIZE = 10
```

Следующим этапом будет создание популяции из 10 особей нейронных сетей и применение эволюционного алгоритма к этой популяции.

Эволюционный алгоритм:

- Создаем популяцию из 10 нейронных сетей с рандомными весами
- Обучаем каждую особь из популяции на тренировочной выборке и смотрим какой процент правильных ответов она выдает
- Выбираем 5 лучших особей по проценту правильных ответов
- Скрещиваем 2 самые лучшие особи и 4 пары из 5 лучших (получается 10 новых нейронных сетей)
- Проделываем пункты 2-4 пока по количеству эпох, в нашем случае будет 100 эпох (поколений нейросетей)

```
nn = []

for i in range(SIZE):
  network = Network([input_count, hidden_count, output_count]) percent =
  network.SGD(training_data=train, epochs=1, mini_batch_size=5,
  eta=1, test_data=test)
  nn.append((network, percent))

for epoch in range(100):
  nn.sort(key=lambda x: x[1])
  nn = nn[5:]
  nnChild = []
  indexes = np.random.randint(0, 5, 10)
  np.random.shuffle(indexes)

  nn[indexes[4]][0].genetic(nn[indexes[3]][0], nnChild)

  for x in range(4):
    nn[indexes[x]][0].genetic(nn[indexes[x + 5]][0], nnChild)
  nn = nnChild
```

Каждая наша нейронная сеть имеет 3 слоя – входной, скрытый и выходной. Это означает, что веса нейронной сети будут храниться в 2-х массивах:

- Первый массив – веса между входным и скрытым слоем, размерности 7×3
- Второй массив – веса между скрытым и выходным слоем, размерности 3×7

Алгоритм скрещивания работает следующим образом:

- На вход подается 2 особи (нейронных сети)
- Выбираем случайным образом индекс из матрицы весов
- Создаем 2 новых потомка, таких, что в первом потомке будут веса из первой нейронной сети до номера случайного индекса, остальная часть из матрицы весов второй нейронной сети, а у второго потомка наоборот
- Обучаем алгоритмом обратного распространения ошибки и смотрим результат на тестовой выборке
- Возвращаем 2 новых нейронных сети с их результатами

После обучения выбираем нейронную сеть с самым лучшим результатом – это и будет итоговая сеть, которую можно использовать для решения реальных задач.

Результаты тестирования ПО

В ходе работы программы были выведены результаты обучения нейронной сети с использованием эволюционного алгоритма. Нейронная сеть обучалась на тренировочной выборке 100 раз. Ниже представлены результаты начальной эпохи, 51-й эпохи и последней эпохи.

В начале первое поколение нейронных сетей дает следующий результат. Как можно заметить, после первого обучения нейронные сети дают в среднем 40% правильных ответов.

```
Эпоха  0
Процент верных ответов:  39 %
Процент верных ответов:  45 %
Процент верных ответов:  41 %
Процент верных ответов:  37 %
Процент верных ответов:  48 %
Процент верных ответов:  43 %
Процент верных ответов:  39 %
Процент верных ответов:  40 %
Процент верных ответов:  46 %
Процент верных ответов:  44 %
```

Рис. 3.2: Начальное поколение нейронных сетей

Поколение с номером 51 дает уже 73% правильных ответов. Заметен существенный прирост в количестве верных ответов.

```
Эпоха  51
Процент верных ответов:  72 %
Процент верных ответов:  71 %
Процент верных ответов:  77 %
Процент верных ответов:  76 %
Процент верных ответов:  74 %
Процент верных ответов:  72 %
Процент верных ответов:  73 %
Процент верных ответов:  76 %
Процент верных ответов:  76 %
Процент верных ответов:  74 %
```

Рис. 3.3: 51 поколение нейронных сетей

Последнее поколение будет отвечать на поставленную задачу правильно в 87% случаев.

```
Эпоха  99
Процент верных ответов:  84 %
Процент верных ответов:  86 %
Процент верных ответов:  87 %
Процент верных ответов:  85 %
Процент верных ответов:  85 %
Процент верных ответов:  87 %
Процент верных ответов:  84 %
Процент верных ответов:  82 %
Процент верных ответов:  86 %
Процент верных ответов:  85 %
```

Рис. 3.4: 51 поколение нейронных сетей

После прохождения 100 эпох получилась рабочая нейронная сеть, способная отвечать правильно на поставленную задачу в 87% случаях.

Литература

1. Нейронные сети, генетические алгоритмы и нечеткие системы: Пер. с польск. И. Д. Рудинского. – 2-е изд., стереотип. – М.: Горячая линия – Телеком, 2013. – 384 с.: ил. ISBN 978-5-9912-0320-3.

2. Нейронные сети: распознавание, управление, принятие решений. — М.: Финансы и статистика, 2004. — 176 с: ил. — (Прикладные информационные технологии). ISBN 5-279-02757-X

3. Python для сложных задач: наука о данных и машинное обучение. — СПб.: Питер, 2018. — 576 с.: ил. — (Серия «Бестселлеры O'Reilly»). ISBN 978-5-496-03068-7.

4. Дрейпер Н., Смит Г. Прикладной регрессионный анализ / Пер. с англ. – М.: Издательский дом «Вильямс», 2007. – 912 с.: ил. ISBN: 978-5-8459-0963-3

```
import math
import doctest
import random
import matplotlib.pyplot as plt
import time
import numpy as np

def sigma(x):
    return 1 / (1 + np.exp(-x))

def sigma_prime(x):
    return sigma(x) * (1 - sigma(x))

def J_quadro(neuron, X, y):
    return 0.5 * np.mean((neuron.vector_forward_pass(X) - y) ** 2)

def J_quadro_derivative(y, y_hat):
    return (y_hat - y) / len(y)

def compute_gradient(neuron, X, y, J_prime=J_quadro_derivative):
    # Compute activation vector

    z = neuron.summatory_function(X)
    y_hat = neuron.activation(z)

    dy_dyhat = J_prime(y, y_hat)
    dyhat_dz = neuron.activation_derivative(z)

    dz_dw = X
    grad = ((dy_dyhat * dyhat_dz).T).dot(dz_dw)
    grad = grad.T
    return grad

class Network:

    def __init__(self, sizes, output=True):
        self.num_layers = len(sizes)
        self.sizes = sizes
        self.biases = [np.random.randn(y,1) for y in sizes[1:]]
        self.weights = [np.random.randn(y, x) for x, y in zip(sizes[:-1], sizes
    [1:])]
        self.output = output

    def feedforward(self, a):
        for b, w in zip(self.biases, self.weights):
            a = sigma(np.dot(w, a) + b)
        return a
```

```
48      def SGD(self, training_set, epochs, mini_batch_size, eta,
            test_set=None):
50
        if test_set is not None: n_test = len(test_set)
52      n = len(training_set)
        success_tests = 0
54
        for j in range(epochs):
56          random.shuffle(training_set)
            mini_batches = [
58              training_set[k:k+mini_batch_size]
                for k in range(0, n, mini_batch_size)]
60
            for mini_batch in mini_batches:
62              self.update_mini_batch(mini_batch, eta)

64          if test_set is not None and self.output:
                success_tests = self.evaluate(test_set)
66              print("Percent of correct answers: ", int(success_tests /  n_test *
    100),"%")

68      if test_set is not None:
            return success_tests / n_test
70
    def update_mini_batch(self, mini_batch, eta):
72      #Update biases and weights neural network, make one step of SGD with mini
    batch

74      nabla_biases = [np.zeros(b.shape) for b in self.biases]
        nabla_weigths = [np.zeros(w.shape) for w in self.weights]
76      for x, y in mini_batch:
            delta_nabla_biases, delta_nabla_weigths = self.back_propogation(x, y)
78          nabla_biases = [nb+dnb for nb, dnb in zip(nabla_biases,
    delta_nabla_biases)]
            nabla_weigths = [nw+dnw for nw, dnw in zip(nabla_weigths,
    delta_nabla_weigths)]
80
        epsilon = eta / len(mini_batch)
82      self.weights = [w - epsilon * nw for w, nw in zip(self.weights,
    nabla_weigths)]
        self.biases  = [b - epsilon * nb for b, nb in zip(self.biases,
    nabla_biases)]
84
    def back_propogation(self, x, y):
86      nabla_biases = [np.zeros(b.shape) for b in self.biases]
        nabla_weigths = [np.zeros(w.shape) for w in self.weights]
88
        activations = [x]
```

```
90          activation = x
            zs = []
92          for b, w in zip(self.biases, self.weights):
                # Compute activations
94              z = w.dot(activation) + b
                zs.append(z)
96              activation = sigma(z)
                activations.append(activation)
98              pass

100         # Back propogation
            delta = (activations[-1]  - y) * activations[-1] * (1 - activations[-1])
102         nabla_biases[-1] = delta
            nabla_weigths[-1] =  delta.dot(activations[-2].T)
104
            for l in range(2, self.num_layers):
106             delta = (self.weights[-l+1]).T.dot(delta) * activations[-l] * (1 -
    activations[-l]) # error on layer L-l
                nabla_biases[-l] =  delta
                    # derivative J by L-l layer
108             nabla_weigths[-l] =  delta.dot(activations[-l-1].T)
            return nabla_biases, nabla_weigths
110
        def evaluate(self, test_set):
112         #Return number of tests with correct answers
            test_results = [(np.argmax(self.feedforward(x)), y)
114                         for (x, y) in test_set]
            return sum(int(x == y) for (x, y) in test_results)
116
        def evolution_algorithm(self, network, neural_network_child):
118         input_count = 3
            hidden_count = 7
120         output_count = 3

122         first = Network([input_count, hidden_count, output_count])
            second = Network([input_count, hidden_count, output_count])
124
            a = self.weights[0]
126         b = self.weights[1]
            c = network.weights[0]
128         d = network.weights[1]

130         a = a.reshape(1, 21)
            b = b.reshape(1, 21)
132         c = c.reshape(1, 21)
            d = d.reshape(1, 21)
134
            first_arr = np.empty((0,1), int)
```

```
        second_arr = np.empty((0,1), int)

        for i in range(len(a)):
            first_arr = np.append(first_arr, a[i])
            second_arr = np.append(second_arr, c[i])
        for i in range(len(b)):
            first_arr = np.append(first_arr, b[i])
            second_arr = np.append(second_arr, d[i])

        point = random.randint(0, len(first_arr))

        for i in range(point):
            first_arr[i], second_arr[i] = second_arr[i], first_arr[i]

        a = first_arr[:a.shape[1]].reshape(7,3)
        b = first_arr[b.shape[1]:].reshape(3,7)
        c = second_arr[:c.shape[1]].reshape(7,3)
        d = second_arr[d.shape[1]:].reshape(3,7)

        first.weights[0] = a
        first.weights[1] = b
        second.weights[0] = c
        second.weights[1] = d

        first_result = first.SGD(training_set=train, epochs=1, mini_batch_size=5,
    eta=1, test_set=test)
        second_result = second.SGD(training_set=train, epochs=1, mini_batch_size=5,
     eta=1, test_set=test)

        neural_network_child.append((first, first_result))
        neural_network_child.append((second, second_result))

data = np.loadtxt("data.csv", delimiter=",")

means = data.mean(axis=0)
means[-1] = 0
stds = data.std(axis=0)
stds[-1] = 1
data = (data - means) / stds

np.random.seed(42)
test_index = np.random.choice([True, False], len(data), replace=True, p=[0.25,
    0.75])
test = data[test_index]
train = data[np.logical_not(test_index)]

train = [(d[:3][:, np.newaxis], np.eye(3, 1, k=-int(d[-1]))) for d in train]
test =  [(d[:3][:, np.newaxis], d[-1]) for d in test]
```

```
182 input_count = 3
    hidden_count = 7
184 output_count = 3
    SIZE = 10
186 epoch = 1

188 random.seed(1)
    np.random.seed(1)
190 neural_network = [] #array of neural network
    for i in range(SIZE):
192     #create neural network, learn  and save results
        network = Network([input_count, hidden_count, output_count])
194     percent = network.SGD(training_set=train, epochs=1, mini_batch_size=5, eta=1,
        test_set=test)

196     #add pair of neural network in array
        neural_network.append((network, percent))
198
    # start evolution algorithm
200 for epoch in range(epoch):
        print("\nEpoch ", epoch, )
202
        neural_network.sort(key=lambda x: x[1])      # sorting by number of correct
        answers
204     neural_network = neural_network[5:]          # pick top 5
        neural_network_child = []                    # array of childrens
206     indexes = np.random.randint(0, 5, 10)        # array of random indexes
        np.random.shuffle(indexes)
208
        #pick top 2 neural network
210     neural_network[indexes[4]][0].evolution_algorithm(neural_network[indexes
        [3]][0], neural_network_child)

212     for x in range(4):
            neural_network[indexes[x]][0].evolution_algorithm(neural_network[indexes[x
        + 5]][0], neural_network_child)
214
        neural_network = neural_network_child
```