Если вы видите что-то необычное, просто сообщите мне.

День 1: Изучение Нейронной сети сложным путем.

Нейронная сеть - тема которая переодически появляется в моей жизни. Однажды, когда я был студентом, я был поглощен идеей построить интеллектуальную машину. Я провел пару бессонных ночей в думах. Я прочел несколько эссе проливающих свет на этот философский вопрос, среди которых самый видны, возможно, были записки Марвина Мински(Marvin Minsky). Как результат, я наткнулся на идею нейронной сети. Это был 2010 год, и глубокое обучение не было насколько популярным, как сейчас. Более того, ни кто не приложил много усилий, чтобы связать нейронную сеть в курсах математики или линейной алгебры. Даже ребята делая классическую оптимизацию и статистику иногда казались в затруднении слыша о нейронных сетях. Через несколько лет. я реализовал простую нейронную сеть с сигмоидной активацией как часть курса "принятия решения". В это же время я понял. что существующее положение нашего знания был до сих пор недостатком при создании "думающешго компьютера" на практике.

Это был 2012, конференция в Крыме, Украина где я присутствовал при разговоре с проф. Лореном Ларгером(Laurent Larger). Он объяснил как построить высоко-скоростное оборудование для распознавания речи используя лазер. Разговор меня вдохновил и годом позже я начал докторскую работу с целью разработать пластовые вычисления, рекурсивные нейронные сети реализуемые напрямую в железе. Наконец, теперь я использую нейронные сети как часть своей работы.

В это серии постов, я буду освещать некоторые интерсные детали проблемы решаемые с помощью нейронной сети. Я очень против повторяющихся усилий, поэтому я постараюсь обойти повторения сторон описанных много раз где-то еще. Для тех же, кто новичек в

данной теме, вместо вступления в нейронные сети я буду ссылаться на главу 1.2 своей статьи: Theory and Modeling of Complex Nonlinear Delay Dynamics Applied to Neuromorphic Computing. Здесь, я всего лишь подведу итог, что нейронные сети были вдохновлены биологическими нейронами. Походя на своего билогического двойника, искуственный нейрон получает множество входных данных, проивзодит нелинейную транформацию, и

$$y = f(w_1 x_1 + w_2 x_2 + \cdots + w_N x_N) = f(\sum_i w_i x_i), \qquad (1)$$

где `N` это число входов `x`, `w` вес синапсиса, а `y` это результат. Удивительно, как может показаться, что в современной нейронной сети `f` может быть практически любой нелинейной. Эта нелинейная фнукия часто называемая функцией активации. Поздравляю, вы дошли до нейроной сети 1950 года.

Для продолжения чтения этой статьи, будут полезны, базовые математические знания, но не обязательно. Разрабатывать может любой у кого есть интуиция на тему нейронных сетей!

Слово о Haskell

Это серия постов практически иллюстрирует все концепты на языке программирования Haskell. Чтобы смотивировать вас на это, вот несколько вопросов и ответов, которые вы хотели знать.

Вопрос: Есть что, что делает Haskell практически лучше для нейронной сети или вы просто делаете это потому что вы предпочли Haskell?

Ответ: Неронные сети это просто "функциональный объекты". Сеть это просто большая композиция функци. А это естественная вещь для функционального языка.

Вопрос: Так как я абсолютный новичек в Haskell, хочу знать преимуства испольвания Haskell перед python или другим языком?

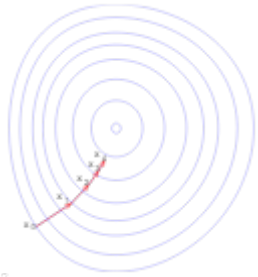
Ответ: Вот основые преимущества:

1. Прост в понимании что делает ваша программа
2. Он хорош в переработке существующей кодовой базы.
3. Haskell направляет ваше мышление непосредственно к решению проблемы практическим путем.
4. Программы на Haskell быстры.

Вопрос: Мой недостаток знания Haskell будет мешать в чтении кода, но ваши примеры кода до сих пор имеют смысл для меня.

Ответ: Спасибо! Я нахожу Haskell интуитивно понятным при объяснени нейронных сеток.

Гадиентный спуск: Первый курс CS



Основная идея обучения нейронных сетей и глубокого обучения это логические оптимизационные методы известные как "Градиентный спуск". Для тех, кому сложно вспомнить первый курс, просто посмотрите видео объясняющее идею.

Как идея, простая как градиентный спуск может работать для нейронной сети? Пожалуй, нейронная сеть это всего лишь функция, сопоставляющая вход и выход. Сравнивая вывод нейронной сети с некоторым желаемым выводом, онам ожет получить другую функцию, известную как ошибочная функция. Это ошибочная фунцкия имеет конкретный *ландшафт ошибок*, как горы. Используя градиентный спуск, мы изменяем, нашу нейронную сеть так, будто мы спускаемся с этого ландафта. Далее, наша цель найти ошибку минимума. Ключевая идея метода оптимизации это, то что градиент ошибки дает нам направление в котором нам необходимо изменить нашу сеть. Похожим образом мы сможем спустится с холма покрытого густым туманом. В обоих случаях локальный градиент(наклон) нам

доступен.

Градиентный спуск может быть описан формулой:

$$x_{n+1} = x_n - \gamma \cdot \nabla F(x_n), \qquad (2)$$

где постоянная `γ` это то что называется скорость обучения в машинном обучении, таким образо количество обучения в одной итерации - `n`. В простом случае `x` скалярная величина. Метод градиентного спуска может быть применен в нескольких строках кода.

```
descend gradF iterN gamma x0 = take iterN (iterate step x0)
  where
    step x = x - gamma * gradF(x)
gradF_test x = 2 * (x - 3)


main = print (descend gradF_test iterN gamma 0.0)
where
iterN = 10
gamma = 0.4
```

На выходе мы получим следующую последовательность:

```
[0.0,2.4000000000000004,2.88,2.976,2.9952,2.99904,2.999808,2.9999616,2.99999232,2.999998464]
```

Выходит. значение минимизации функции `f(x)` равно 3, то есть `min f(x) = f(3) = (x-3)^2 = 0`. А последовательность постепенно сходится к этому числу. Давайте посмотрим пристально на то, что делает код выше.

- Линии 1-3: Мы определяем метод градиентного спуска, который пошагово применяет функцию `step` реализующую выражение 2. Мы предоставляем промежуточные результаты берущие первые `iterN` значения.
- Линии 5: Предположим, мы хотим оптимизировать функцию `f(x) = (x -3)^2`. Тогда этот градиент `gradF_test` становится:

$$\nabla f(x) = 2 \cdot (x - 3).$$

- Линии 7-10: И вот, мы запустили наш градиентный спуск установив сколкрость обучения $\gamma = 0.4$.

Ключевой момент - понять как значение γ действует на сходимость. Когда γ слишком маленькое, алгоритм произведет множество итераций для схождения, однако если γ слишком большое, алгоритм никогда не сойдется. Я не пытаюсь показать простой путь как определить γ для заданной проблемы. Отсюда, должны использоваться различные значения γ. Вы можете поиграться с кодом выше и посмотреть что получается. Для примера вы можете попробовать различные значения γ, к примеру: 0.01, 0.1, 1.1. Метод применим ко множеству N. В далнейшем мы заменим `gradF_test` функцию скорей векторной, чем скалярной.

Части нейронной сети для задачи классификации

После того как мы поняли, как градиентный спуск работает, мы может быть захотим научить умеренно полезуню сетку. Скажем, мы хотим определять типа ириса используя явные черты: длина чашелистика, ширина чашелистика, длинна лепестка, ширина лепестка. Существует три типа цветов мы хотим узнавать:

- Ирис щетинестый(Setosa)

- Ирис разноцветный(Versicolour)
- Ирис вергинский(Virginica) Теперь проблема закодировать эти классы чтобы наша нейронная сеть могда обработать их.

Наивное решение и почему оно не работает.

Самое простое решение указать на отличия это использовать натуральные числа. Для примера, ирис щетинистый может быть зашифрован как `1`, разноцветный `2`, виргинский `3`. Однако, проблема с этим типом кодирования заключается в том что он предвзятый. Первое кодируются в качестве числа, мы навязали линейный порядок для трех класов. Это значит, что мы начинаем наш отсчет с ириса щетинистого, затем разноцветный и только потом виргинский. Однако, в реальности это не важно чем мы заканчиваем виргинским или разноцветным. Второе, мы так же предполагаем, что расстояние между виргинским и щетинистым ирисами: `3-1=2` - больше чем между виргинским и разноцветным ирисами `3-2=1`, что априори не правильно.

Одно горячее кодирование.

Какой же тип кодировки нам нужен? Первое, мы не хотим влиять на ограничения порядка, и второе, мы хотим уровнять разницу расстояний между типами. Далее, мы предпочитаем кодировать каждый тип ортогонально, то есть независимо от двух других. Это становится возможно если мы используем вектор трех измерений(так как у нас три типа). Теперь, ирис щетинистый закодирован как [1,0,0], разноцветный [0,1,0] и виргинский [0,0,1]. Эвклидово расстояние между любыми парми классов равны `sqrt(2)` Для примера, расстояние между щетинистым и разноцветным вычисляется как:

$$\sqrt{(1-0)^2+(0-1)^2+(0-0)^2}=\sqrt{2}.$$

Складываем всё вместе.

Так как мы уже знакомы с базой нейронной сети и градиентного спуска и так же имеем некоторые данные, чтобы поиграться, пусть веселуха начнется!

Сначала мы создаем сеть из трех нейронов. Чтобы это сделать, мы обощим формулу (1):

$$y_i = f(\sum_k w_{ik} x_k), \tag{3}$$

где $(x_1, x_2, x_3, x_4) = \mathbf{x}$ четырех мерный входной вектор, $w_{ik} \in \mathbf{W}, i = 1 \dots 3, k = 1 \dots 4$ веса матрицы синопсов, и результирующий $(y_1, y_2, y_3) = \mathbf{y}$ трехмерный вектор. Говоря в общем, мы производим матрично-векторную манипуляцию с последующей поэлементной активацией.

$$\mathbf{y} = f(\mathbf{W}\mathbf{x}). \tag{4}$$

Как не линейную активационную функцию `f` мы будем использовать функцию *сигмоид* $\sigma(x) = [1 + e^{-x}]^{-1}$. Мы будем использовать `hmatrix` Haskell библиотеку для операций линейной алгебры таких как работа с матрицами. С помощью `hmatrix`, выражение (4) может быть переписано как:

```
import Numeric.LinearAlgebra as LA

sigmoid = cmap f
where
f x = recip $ 1.0 + exp (-x)


forward x w =
let h = x LA.<> w
y = sigmoid h
in [h, y]
```

где `<>` обозначает матричный результат функции из модуля `LA`. Отметим что `x` может быть вектором, но он так же может быть матричным набором. В последнем случае, `forward` будет преобразовывать наш полный набор данных. Отметим, что мы предоставляем не только результат наших вычислений `y`, но так же промежуточный шаг `h` так как он будет использован повтороно позднее для `w` градиентных вычислений.

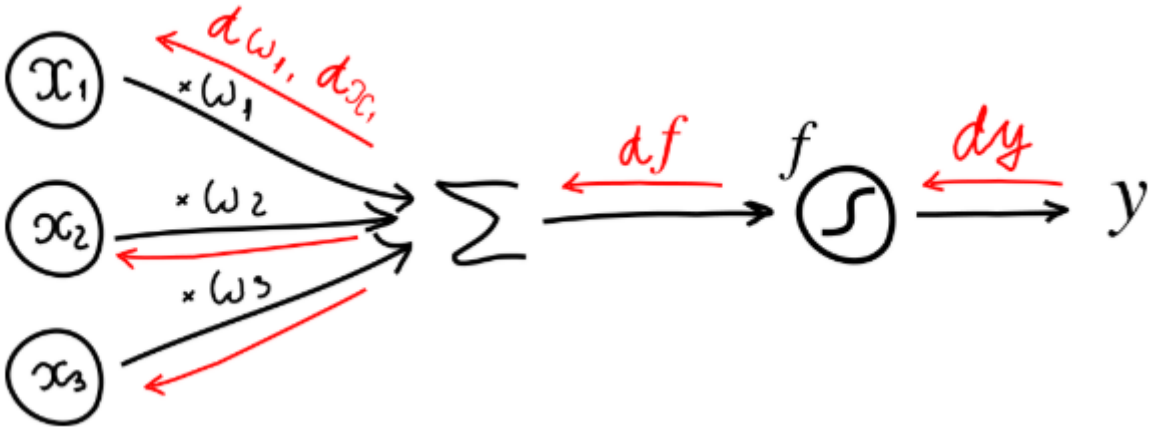
Какждый из нейронов `y` предположительно сработает когда он "думает" что он заметил один из трех видов. Напрмер, когда мы имеет вывод `[0.89, 0.1, 0.2]`, мы предполагаем что первый нейрон самый "правилный", то есть, мы интерпретируем результат как ирис щетинистый. Другими словами, этот вывод воспринимаетя как `[1,0.0]`. Как вы можете увидеть, максимальный элемент был установлен еденицей, а остальные два нулями. Это так называемое правило "Победитель забирает всё".

Перед обучением нейронной сети, нам нужно некоторые измерения для уменьшения ошибок или функции потерь. Для примера, мы можем использовать Эвклидову потерю: $\text{loss} = \sum_i (\hat{y}_i - y_i)^2 \text{ where } \hat{y}_i$ где $\hat{y}_i$ - предсказание, а y_i реальный ответ из нашего набора данных:

```
loss y tgt =
  let diff = y - tgt
  in sumElements $ cmap (^2) diff
```

Для иллюстрации градиентного скуска, мы повторно используем `descend` функцию определенную выше. Теперь, мы можем определить градиентную функцию для выражения (4) нашей неронной сети. Мы используем. то что называется методом *обратного распространия*, который естественно результат дифференцирования сложной функции и показан для отдельного нейрона на рискунке ниже:

Обратный проход



Обратное распространение для единичного нейрона.
Первое, в прямом прохождении, начальные выводы `y` вычеслены. Затем, этот вывод сравнивается с некоторым желаемым выводом, и градиент ошибки `dy` передается назад. Затем, градиент активационной функции `df`использует полученное `dy`. Это приводит к оставшимся градиентам:

$$dw_1, dx_1, dw_2,$$

Теперь мы можем посчитать веса градиента `dW` используя метод *обратного распространения* выше:

```
grad (x, y) w = dW
  where
    [h, y_pred] = forward x w
    dE = loss' y_pred y
    dY = sigmoid' h dE
    dW = linear' x dY
```

Здесь `linear'`, `sigmoid'`, `loss'` градиенты ленейной операции(перемножения), активация сигмоида $\sigma(x)$ и функции потерь. Отметим что операциями над матрицами мы обычно вычисляем не скалярную величину а градиент вектора `dW` обозначенный как любой вес синопса dw_i. Ниже эти "направленные" функциональный определения в haskell используют `hmatrix` библиотеку:

```
linear' x dy = cmap (/ m) (tr' x LA.<> dy)
  where
    m = fromIntegral $ rows x
```

```
sigmoid' x dY = dY * y * (ones - y)
where
y = sigmoid x
ones = (rows y) >< (cols y) $ repeat 1.0


loss' y tgt =
let diff = y - tgt
in cmap (* 2) diff
```

Чтобы проверить нашу сетку, мы качаем набор данных (два файла: `x.dat` и `y.dat`) и сам код. Как указано в коментарии файла, мы запускаем нашу программу:

```
$ stack --resolver lts-10.6 --install-ghc runghc --package hmatrix-0.18.2.0 Iris.hs
Initial loss 169.33744797846379
Loss after training 61.41242708538934
Some predictions by an untrained network:
(5><3)
[ 8.797633210095851e-2, 0.15127581829026382, 0.9482351750129188
,  0.11279346747947296,  0.1733431584272155, 0.9502442520696124
,  0.10592462402394615, 0.17057190568339017, 0.9367875655363787
,  0.10167941966201806, 0.20651101803783944, 0.9300343579182122
, 8.328154248684484e-2, 0.15568011758813116,  0.940816298954776 ]
Some predictions by a trained network:
(5><3)
[ 0.6989749292681016, 0.14916793398555747,  0.1442697900857393
,  0.678406436711954,  0.1691062984304366,  0.2052955124240905
, 0.6842327447503195, 0.16782087736820395, 0.16721778476233148
, 0.6262988163006756, 0.19656943129188192, 0.17521133197774072
, 0.6905553549763312, 0.15299944611286123, 0.12910826989854146 ]
Targets
(5><3)
[ 1.0, 0.0, 0.0
, 1.0, 0.0, 0.0
, 1.0, 0.0, 0.0
, 1.0, 0.0, 0.0
, 1.0, 0.0, 0.0 ]
```

На сегодня всё. Вы можете поиграться с кодом. **Подсказка:** вы можете заметить что `grad` вызывает `sigmoid` дважды на один и тот же набор данных: один раз в `forward` и один раз в `sigmoid`. Попробуйте оптимизировать код чтобы избежать эту избыточность.

Как только вы поймете основы нейронных сетей, можете переходит ко второму дню. В следующей статье, вы научитесь использовать ваше нейронную сеть. Во-первых, мы выделим важность многослоной структуры. МЫ так же показывать как нелинейные активаторы важны. Во-вторых, мы улучшим обучение нейронной сети и обсудим начальные веса синопсов.

Revision #11
Created 17 September 2020 09:29:24 by gasick
Updated 3 September 2022 09:34:01 by gasick