Теоретичні відомості

Розробка простої нейронної мережі на мові Python.

Нейромережа, яка тренується через зворотне поширення (backpropagation) та намагається використовувати вхідні дані для передбачення вихідних.

Дано:

Вхідні дані			Вихідні дані
0	0	1	0
1	1	1	1
1	0	1	1
0	1	1	0

Потрібно передбачити, як буде виглядати колонка «Вихідні дані» на основі вхідних даних.

Завдання може бути вирішено, підрахувавши статистичну відповідність між ними. З вихідними даними на 100% корелює лівий стовпець. Для створення моделі, в найпростішому випадку, подібну статистику розраховує зворотне поширення.

Приклад коду на Python №1.

Код запускатися в ipython notebook.

```
X = np.array([ [0,0,1],[0,1,1],[1,0,1],[1,1,1] ])
y = np.array([[0,1,1,0]]).T
syn0 = 2*np.random.random((3,4)) - 1
syn1 = 2*np.random.random((4,1)) - 1
for j in xrange(60000):
    l1 = 1/(1+np.exp(-(np.dot(X,syn0))))
    l2 = 1/(1+np.exp(-(np.dot(l1,syn1))))
    l2_delta = (y - l2)*(l2*(1-l2))
    l1_delta = l2_delta.dot(syn1.T) * (l1 * (1-l1))
    syn1 += l1.T.dot(l2_delta)
    syn0 += X.T.dot(l1_delta)
```

Нейромережа в два шари

Змінні та їх опис.

X - матриця вхідного набору даних; рядки - тренувальні приклади.
y - матриця вихідного набору даних; рядки - тренувальні приклади.
l0 - перший шар мережі, визначений вхідними даними.

l1 - другий шар мережі, або прихований шар.

syn0 - перший шар ваг, Synapse 0, об'єднує l0 з l1.

"*" - поелементне множення - два вектори одного розміру перемножують відповідні значення, і на виході виходить вектор такого ж розміру.

"-" - поелементне віднімання векторів.

x.dot (y) - якщо x і y - це вектори, то на виході вийде скалярний добуток. Якщо це матриці, то вийде множення матриць. Якщо матриця та вектор - це множення вектора і матриці.

Приклад коду на Python №2 нейромережа в два шари (рус. слоя)**.**

```
import numpy as np # библиотека линейной алгебры

# Сигмоида
def nonlin(x,deriv=False):
    if(deriv==True):
        return f(x)*(1-f(x))
    return 1/(1+np.exp(-x))

# набор входных данных
X = np.array([  [0,0,1],
                [0,1,1],
                [1,0,1],
                [1,1,1] ])

# выходные данные
y = np.array([[0,0,1,1]]).T

# сделаем случайные числа более определёнными
np.random.seed(1)

# инициализируем веса случайным образом со средним 0
syn0 = 2*np.random.random((3,1)) - 1

for iter in xrange(10000):

    # прямое распространение
    l0 = X
    l1 = nonlin(np.dot(l0,syn0))

    # насколько мы ошиблись?
    l1_error = y - l1

    # перемножим это с наклоном сигмоиды
```

```
    # на основе значений в l1
    l1_delta = l1_error * nonlin(l1,True) # !!!

    # обновим веса
    syn0 += np.dot(l0.T,l1_delta) # !!!

print "Выходные данные после тренировки:"
print l1
```

Вихідні дані після тренування:

```
[[ 0.00966449]
 [ 0.00786506]
 [ 0.99358898]
 [ 0.99211957]]
```

Нейромережа в три шари

Вхідні дані			Вихідні дані
0	0	1	0
0	1	1	1
1	0	1	1
1	1	1	0

Треба передбачити вихідні дані на основі трьох вхідних стовпців. Жоден з вхідних стовпців не корелює на 100% з вихідним. Третій стовпець взагалі ні з чим не пов'язаний, оскільки в ньому всі одиниці. Однак, якщо в одному з двох перших стовпців (але не в обох відразу) міститься 1, то результат також буде дорівнювати 1.

Це нелінійна схема, оскільки не існує прямої відповідності стовпців один до одного. Відповідність будується на комбінації стовпців 1 і 2 вхідних даних. Щоб її отримати потрібно додати ще один шар. Перший шар комбінує вхід, другий призначає відповідність виходу, використовуючи в якості вхідних даних вихідні дані першого шару (див. таблицю нижче).

Вхід (l0)			Прихована вага (l1)				Вихід (l2)
0	0	1	0.1	0.2	0.5	0.2	0
0	1	1	0.2	0.6	0.7	0.1	1
1	0	1	0.3	0.2	0.3	0.9	1
1	1	1	0.2	0.1	0.3	0.8	0

Вага призначається випадковим чином, так отримуються приховані значення для шару №1. У другого стовбця прихованих ваг вже є невелика кореляція з виходом. І це теж є важливою частиною процесу тренування мережі. Тренування буде тільки посилювати цю кореляцію. Вона буде оновлювати syn1, щоб призначити її відповідність вихідним даним, і syn0, щоб краще отримувати дані з входу.

Змінні та їх опис.

X - матриця вхідного набору даних; рядки - тренувальні приклади.

y - матриця вихідного набору даних; рядки - тренувальні приклади.

l0 - перший шар мережі, визначений вхідними даними.

l1 - другий шар мережі, або прихований шар.

l2 - фінальний шар, прийнята гіпотеза. По мірі тренування повинен наближатися до правильної відповіді.

syn0 - перший шар ваг, Synapse 0, об'єднує l0 з l1.

syn1 - другий шар ваг, Synapse 1, об'єднує l1 з l2.

l2_error - промах мережі в кількісному виразі.

l2_delta - помилка мережі, в залежності від впевненості передбачення. Майже збігається з помилкою, за винятком упевнених прогнозів.

l1_error - зважуючи l2_delta вагою з syn1, підраховується помилка в середньому/прихованому шарі.

l1_delta - помилки мережі з l1, масштабовані за впевненістю прогнозів. Майже збігається з l1_error, за винятком впевнених прогнозів.

"*" - поелементне множення - два вектори одного розміру перемножують відповідні значення, і на виході виходить вектор такого ж розміру.

"-" - поелементне віднімання векторів.

x.dot (y) - якщо x і y - це вектори, то на виході вийде скалярний добуток. Якщо це матриці, то вийде множення матриць. Якщо матриця та вектор - це множення вектора і матриці.

Код - це просто попередня реалізація мережі, складена в два шари один над іншим. Вихід першого шару l1 - це вхід другого шару. Щось нове є лише в наступному рядку:

```
l1_error = l2_delta.dot(syn1.T)
```

Використовує помилки, зважені на впевненості передбачень з l2, щоб підрахувати помилку для l1. Отримує помилку, зважену за вкладами - підраховується, який внесок в помилки в l2 вносять значення в вузлах l1. Цей крок і називається зворотним поширенням помилок. Потім оновлюється syn0, використовуючи той же алгоритм, що і у варіанті з нейромережею з двох шарів.

Приклад коду на Python №3 нейромережа в три шари

```
import numpy as np

def nonlin(x,deriv=False):
        if(deriv==True):
         return f(x)*(1-f(x))

        return 1/(1+np.exp(-x))

X = np.array([[0,0,1],
          [0,1,1],
          [1,0,1],
          [1,1,1]])

y = np.array([[0],
                      [1],
                      [1],
                      [0]])

np.random.seed(1)

# случайно инициализируем веса, в среднем - 0
syn0 = 2*np.random.random((3,4)) - 1
syn1 = 2*np.random.random((4,1)) - 1

for j in xrange(60000):

        # проходим вперёд по слоям 0, 1 и 2
   l0 = X
   l1 = nonlin(np.dot(l0,syn0))
   l2 = nonlin(np.dot(l1,syn1))

   # как сильно мы ошиблись относительно нужной величины?
   l2_error = y - l2

   if (j% 10000) == 0:
      print "Error:" + str(np.mean(np.abs(l2_error)))

   # в какую сторону нужно двигаться?
   # если мы были уверены в предсказании, то сильно менять его
не надо
   l2_delta = l2_error*nonlin(l2,deriv=True)

   # как сильно значения l1 влияют на ошибки в l2?
```

```
        l1_error = l2_delta.dot(syn1.T)

        # в каком направлении нужно двигаться, чтобы прийти к l1?
        # если мы были уверены в предсказании, то сильно менять его
не надо
        l1_delta = l1_error * nonlin(l1,deriv=True)

        syn1 += l1.T.dot(l2_delta)
    syn0 += l0.T.dot(l1_delta)
```

```
Error:0.496410031903
Error:0.00858452565325
Error:0.00578945986251
Error:0.00462917677677
Error:0.00395876528027
Error:0.00351012256786
```

Корисні посилання

1. Нейросеть в 11 строчек на Python https://habr.com/ru/post/271563/
2. Инструкция: Создание нейронной сети без навыков программирования https://vc.ru/selectel/41002-instrukciya-sozdanie-neyronnoy-seti-bez-navykov-programmirovaniya
3. Нейросеть на Python, часть 2: градиентный спуск https://habr.com/ru/post/272679/
4. Как создать собственную нейронную сеть с нуля на языке Python https://neurohive.io/ru/tutorial/kak-sozdat-nejronnuju-set-s-nulja-na-jazyke-python/
5. Нейронные сети, или Как обучить искусственный интеллект http://internetinside.ru/neyronnye-seti-ili-kak-obuchit-iskuss/
6. Bouchard, G.Accelerating Stochastic Gradient Descent via Online Learning to Sample / Guillaume Bouchard, Theo Trouillon, Julien Perez, Adrien Gaidon // https://arxiv.org/pdf/1506.09016v1.pdf
7. Sutskever, I. On the importance of initialization and momentum in deep learning / Ilya Sutskever, James Martens, George Dahl, Geoffrey Hinton // http://proceedings.mlr.press/v28/sutskever13.pdf
8. Алгоритм імітації відпалу https://uk.wikipedia.org/wiki/Алгоритм_імітації_відпалу
9. Simulated annealing https://en.wikipedia.org/wiki/ Simulated_annealing
10. Rasdi Rere, L.M. Simulated Annealing Algorithm for Deep Learning / L.M. Rasdi Rere, Mohamad IvanFananyAniati MurniArymurthy // Procedia Computer Science Volume 72, 2015, P. 137-144 https://www.sciencedirect.com/science/article/pii/S1877050915035759

11. Метод стохастичного градієнта https://uk.wikipedia.org/wiki/Метод_стохастичного_градієнта
12. Broyden–Fletcher–Goldfarb–Shanno (BFGS)Алгоритм Бройдена — Флетчера — Гольдфарба — Шанно https://uk.wikipedia.org/wiki/Алгоритм_Бройдена_—_Флетчера_—_Гольдфарба_—_Шанно
13. Limited-memory BFGS https://en.wikipedia.org/wiki/Limited-memory_BFGS
14. ADADELTA: An Adaptive Learning Rate Method https://arxiv.org/abs/1212.5701
15. Adam: A Method for Stochastic Optimization https://arxiv.org/abs/1412.6980