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

День 4: Важность пакетной нормализации

Для какой цели существуют нейронные сети? Нейронные сети - это обучаемые модели. Их главная цель - приблизиться или даже превзойти человеческие способности восприятия. Ричард Суттон говорит, что самый большой уро, который может быть получен с 70 годов разработки ИИ, в том, что общие методы использующие вычисления очень эффективны. В его эссе, Суттон говорит, что только модели без зашифрованного человеческого знания могут превзойти человеко-ориентированные подходы. Так, что нейронные сети в общем будут достаточны для использования вычислений. Затем, не удивительно, что они могут выставить миллионы обучаемых степеней свободы.

Самый большой вызов для нейронных сетей: 1) как обучить эти миллионы параметров, 2) как их интерпритировать. Пакетная нормализация(batchnorm) был представлен в качестве попытки сделать обучение еще эффективнее. Метод может сильно сократить число итераций обучения. Даже больше, batchnorm - возможо является ключем, который даст возможность обучать определенные архитектуры такие как бинарная нейронная сеть. Наконец, batchnorm одна из самых последних преимуществ нейронной сети.

Пакетная нормализация в кратце

Что такие пакет? До сих пор, вы смотрели на игрущечные наборы данных. Они были настолько малы, что могли быть легко помещены в память. Однако, в реальности существует огромные наборы занимающие сотни гигабайтов памяти, такие как Imagenet. Они часто не помещаются в память. В этом случае, имеет смысл разделить наборы данных в маленькие пакеты. Во время обучения обрабатывается только один пакет.

Как предполагает имя, batchnorm преобразование это действие над отдельными пакетами данных. Вывод линейного слоя может быть причиной деградации активационной функции.

Например: в случае ReLU `f(x)=max(0,x)` активация, все негативные значения будут приводить к нулевой активации. Отсюда, хорошая идея - нормализовать эти значения с помощью вычитания среднего `μ` пакета. Похожим образом, деление по стандартному отклоненю `√var` масштабирует амплитуту, которая особенно выгодня для активация сигмоидного вида.

Обучение и Batchnorm

Процеедура пакетной нормализации имеет отличия на этапах обучения и вывода. Во время обучения, каждый слой, где мы хотим применить batchnorm, сначала вычисляем средний мини пакет:

$$\mu = \langle \mathbf{X} \rangle = \frac{1}{m} \sum_{i=1}^{m} \mathbf{X}_i, \tag{1}$$

где `Xi` это `i` вектор-свойство идущий из прошлого слоя; `i=1…m`, где `m>1` это размер пакета. Так же получаем ди:сперсию для пакета

$$\mathrm{var} = \frac{1}{m} \sum_{i=1}^{m} (\mathbf{X}_i - \mu)^2. \tag{2}$$

Теперь batchnorm ядро, сама нормализация:

$$\hat{\mathbf{X}}_i = \frac{\mathbf{X}_i - \mu}{\sqrt{\mathrm{var} + \epsilon}}, \tag{3}$$

где маленькая постоянная `ϵ` добавляет числовую стабильность. Что если нормализация данного слоя была вредна? Алгоритм предоставит два обучаемых параметра, которые в худшем случае могут отменить эффект пакетной нормализации: смасштабирует параметр

`γ` и увеличит `β`. После применения таковых мы получим вывод слоя batchnorm:

$$\mathbf{Y}_i = \gamma * \hat{\mathbf{X}}_i + \beta. \qquad (4)$$

Отметим, что оба: среднее значениее `μ` и распределение `var` векторы количество которых такое же большое как и нейронов в данном скрытом слое. Оператор `*` говорит о поэлементном умножении.

Вывод

В стадии вывода лучше всего иметь одну выборку данных за раз. Как же сосчитать среднюю величину пакета если целый пакет это и есть выборка данных? Для правильной обраотки, во время обучения мы указываем среднее значение и распределение (`E[X]` and `Var[X]) для всех наборов обучения. Эти вектора заменят` μ `и` var` во время вывода, таким образом избегая проблемы нормализации единичного пакета.

Насколько эффективен Batchnorm

До этого момента мы успели поиграться с задачами, которые предоставляют свойство низкоразмерного ввода. Теперь, мы собираемся протестировать нейронную сеть проблемой по серьезнее. Мы применим наши навыки для автоматического распознавания написанных человеком цифр на основе известного MNIST набора данных. Эта проблем появляется в следствии нужды чтения кодов с конветов на почте.

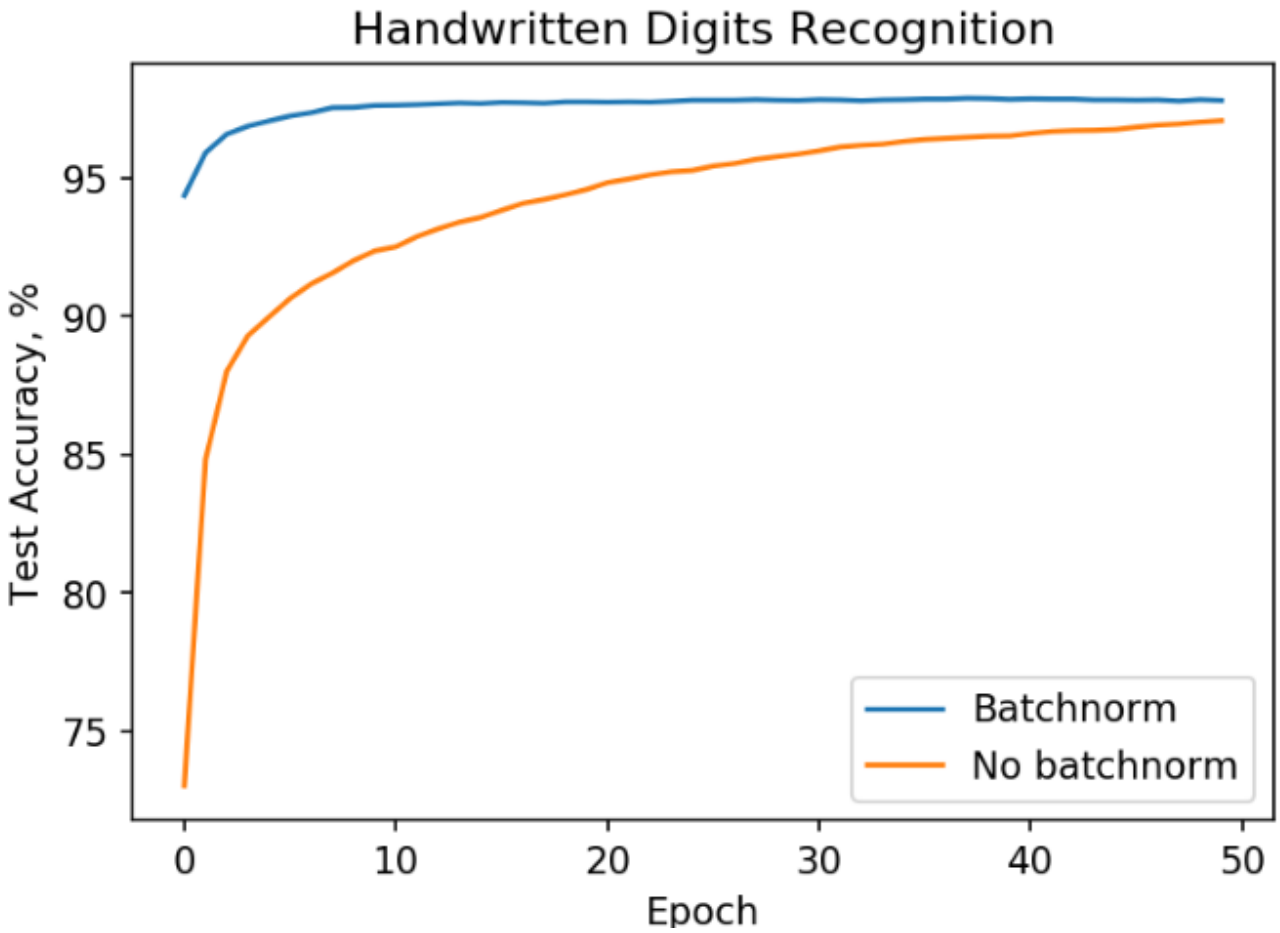
Мы создадим две нейронные сети, каждая бдует иметь два полностью связанных скрытых слоя (300 и 50 нейронов). Обе сети получают картинку размером `28×28=784` пикселей, и возвращает 10 чисел, распознанное число. Для внутненних слоев мы будем использовать ReLU `f(x)=max(0,x)`. Для получения возможных класси

$$\sigma(\mathbf{x})_i = \frac{\exp x_i}{\sum_{j=1} \exp x_j}$$

качестве функции активации мы будем использовать .

Одна из сетей в дополнение будет производить пакетную нормализацию перед ReLU. Затем,

мы будем обучать их использую стохастический градиентный спуск с коэфициентом обучения `α=0.01^3` и размером пакета `m=100`.



Обучение нейронной сети на MNIST данных.

Обучение с помощью пакетной нормализации (синий график) ведет к высокой точности быстрее чем без нее(ораньжевый график).

Из графика мы видим, что нейронная сеть с пакетной нормализацией достикает точности в 98% за 10 эпох, где второй пыатеся сделать это в течении 50!Похожие результаты могут быть получены другими архитектурами.

Надо упомянуть, что мы до сих пор может слабо недостаточно понимать как именно batchnorm помогает. В его оригинальном описание, предполагается, что batchnorm сокращает внутренее ковариантное смещение. Недавно, было показано, что это не обязательно правда. Лучшее на сегодняшний день объяснение то, что batchnorm делает оптимизационные сглаживания, которые делают обучение градиентным спуском эффективнее. Что, в свою очередь, позволяет ускорить процесс обучение с помощью batchnorm!

Реализация batchnorm

Мы воспользуемся кодом приведенным в дне 2. Первое, мы переопределим структура дынных `Layer` сделав его более детализированным:

```
data Layer a = -- Linear layer with weights and biases
               Linear (Matrix a) (Vector a)
               -- Same as Linear, but without biases
               | Linear' (Matrix a)
               -- Batchnorm with running mean, variance, and two
               -- learnable affine parameters
               | Batchnorm1d (Vector a) (Vector a) (Vector a) (Vector a)
               -- Usually non-linear element-wise activation
               | Activation FActivation
```

Превосходно! Теперь мы можем отделить несколько типов слоев: родственный(линейный), активационный и batchnorm. Так как batchnorm уже компенсирует смещения, нам не нужно использовать смещения на последующих слоях. Поэтому мы определяем `Linear` слой без смещений. Так же расширим Gradients для использования новой структуры слоёв:

```
data Gradients a = -- Weight and bias gradients
                   LinearGradients (Matrix a) (Vector a)
                   -- Weight gradients
                   | Linear'Gradients (Matrix a)
                   -- Batchnorm parameters and gradients
                   | BN1 (Vector a) (Vector a) (Vector a) (Vector a)
                   -- No learnable parameters
                   | NoGrad
```

Теперь, мы хотим расширить функцию распространения нейронной сети `_pass`, в зависимости от слоя. Это проще всего сделать с помощью с сопоставления с образцом. Вот как мы это сделаем в `Batchnorm1d` слое и его параметрах:

```
    _pass inp (Batchnorm1d mu variance gamma beta:layers)
        = (dX, pred, BN1 batchMu batchVariance dGamma dBeta:t)
      where
```

Как и раньше, `_pass` функция получает ввод `inp` и параметры слоя. Второй аргумент это образец с который мы и проводим сопоставление, создавая наш алгоритм, в данном случае, Batchnorm1D. Мы также указываем `_pass` для других видово слоев. Отсуда, мы получаем полиморфную функцию `_pass` для каждого слоя. Наконец, результат выражения является кортеж: градиенты для обратного распространения `dX`, предсказание `pred` и дополнительные данные `t` с вычисленными значениями `BN1` в этом слое(средний пакет `batchMu`, распределение `batchVariance` и параметры градиентов для обучения).

Следующий шаг показан ниже:

```
        -- Forward
        eps = 1e-12
        b = br (rows inp)  -- Broadcast (replicate) rows from 1 to batch size
        m = recip $ (fromIntegral $ rows inp)

        -- Step 1: mean from Equation (1)
        batchMu :: Vector Float
        batchMu = compute $ m `_scale` (_sumRows inp)

        -- Step 2: mean subtraction
        xmu :: Matrix Float
        xmu = compute $ inp .- b batchMu

        -- Step 3
        sq = compute $ xmu .^ 2

        -- Step 4: variance, Equation (2)
        batchVariance :: Vector Float
        batchVariance = compute $ m `_scale` (_sumRows sq)
```

```
        -- Step 5
        sqrtvar = sqrtA $ batchVariance `addC` eps

        -- Step 6
        ivar = compute $ A.map recip sqrtvar

        -- Step 7: normalize, Equation (3)
        xhat = xmu .* b ivar

        -- Step 8: rescale
        gammax = b gamma .* xhat

        -- Step 9: translate, Equation (4)
        out0 :: Matrix Float
        out0 = compute $ gammax .+ b beta
```

Обсуждая в статье "День 2", мы повторяли вызов получения градиентов следующего слоя, нейронная сеть предсказывала `pred` и вычисляла значение хвоста `t`:

```
        (dZ, pred, t) = _pass out layers
```

Я предпочитаю хранить обратную передачу без упрощений. Это помогает прояснить какой из шагов за что отвечает:

```
        -- Backward

        -- Step 9
        dBeta = compute $ _sumRows dZ

        -- Step 8
        dGamma = compute $ _sumRows (compute $ dZ .* xhat)
        dxhat :: Matrix Float
        dxhat = compute $ dZ .* b gamma

        -- Step 7
        divar = _sumRows $ compute $ dxhat .* xmu
        dxmu1 = dxhat .* b ivar

        -- Step 6
```

```
        dsqrtvar = (A.map (negate. recip) (sqrtvar .^ 2)) .* divar

        -- Step 5
        dvar = 0.5 `_scale` ivar .* dsqrtvar

        -- Step 4
        dsq = compute $ m `_scale` dvar

        -- Step 3
        dxmu2 = 2 `_scale` xmu .* b dsq

        -- Step 2
        dx1 = compute $ dxmu1 .+ dxmu2
        dmu = A.map negate $ _sumRows dx1

        -- Step 1
        dx2 = b $ compute (m `_scale` dmu)

        dX = compute $ dx1 .+ dx2
```

Часто, нам нужно произвести операции типа вычитания среднего `X–μ`, где, на практике, у нас есть матрица `X` и вектор `μ`. Как вычесть вектор из матрицы? Правильно, никак. Вычитать можно только 2 матрицы. Такие библиотеки как `Numpy` могут говорить о магии, которая упростит превращение вектора в матрицу. Это может быть полезно, но может выявить различные виды багов. Мы же взамен произведем явное преобразование вектора в матрицу. Для наших нужд, мы имеем скращение `b = br (rows inp)`, которое мы расширим вектор на то же число рядов как в `inp`. Где функция `br ('broadcast')`:

```
br rows' v = expandWithin Dim2 rows' const v
```

Это пример того, ка кработае br. Для начала, интерактивно запустим сессию и загрузим модуль `NeuralNetwork.hs`:

```
$ stack exec ghci
GHCi, version 8.2.2: http://www.haskell.org/ghc/  :? for help
Prelude> :l src/NeuralNetwork.hs
```

Затем проверим `br` функцию на векторе `[1, 2, 3, 4]`:

```
*NeuralNetwork> let a = A.fromList Par [1,2,3,4] :: Vector Float
*NeuralNetwork> a
Array U Par (Sz1 4)
  [ 1.0, 2.0, 3.0, 4.0 ]

*NeuralNetwork> let b = br 3 a
*NeuralNetwork> b
Array D Seq (Sz (3 :. 4))
  [ [ 1.0, 2.0, 3.0, 4.0 ]
  , [ 1.0, 2.0, 3.0, 4.0 ]
  , [ 1.0, 2.0, 3.0, 4.0 ]
  ]
```

Как можно увидеть, была получения новая матрица с тремя идентичными строками. Отметим, что `a` имеет тип `Array U Seq`, что значит, что данные хранятся простым распакованным масивом.Где результат типа `Array D Seq` так называемый отложенный массив. Этот отложенный массив, на самом деле не массив, но так понятнее становится, что это будет массивом в дальнейшейм. Чтобы разместить массив в памяти используйте `compute` :

```
*NeuralNetwork> compute b :: Matrix Float
Array U Seq (Sz (3 :. 4))
  [ [ 1.0, 2.0, 3.0, 4.0 ]
  , [ 1.0, 2.0, 3.0, 4.0 ]
  , [ 1.0, 2.0, 3.0, 4.0 ]
  ]
```

Подробную информацию можно получить из большой документации. Так же как и `br`, существует несколько более удобных функций, `rowsLike` и `colsLike`. Они полезны в связке с `_sumRows` и `_sumCols`:

```
-- | Sum values in each column and produce a delayed 1D Array
_sumRows :: Matrix Float -> Array D Ix1 Float
_sumRows = A.foldlWithin Dim2 (+) 0.0

-- | Sum values in each row and produce a delayed 1D Array
_sumCols :: Matrix Float -> Array D Ix1 Float
_sumCols = A.foldlWithin Dim1 (+) 0.0
```

Пример использования `_sumCols` и `colsLike` при вычислении `softmax` активационной функции $\sigma(\mathbf{x})_i = \frac{\exp x_i}{\sum_{j=1} \exp x_j}$:

```
softmax :: Matrix Float -> Matrix Float
softmax x =
  let x0 = compute $ expA x :: Matrix Float
      x1 = compute $ (_sumCols x0) :: Vector Float
      x2 = x1 `colsLike` x
  in (compute $ x0 ./ x2)
```

Заметьте, что `softmax` отличается от поэлементной активации. Взамен, `softmax` действует как полностью связанный слой, который получает и отдает вектор. Наконец, мы определяем нашу нейронную сеть, с двумя скрытыми слоями и пакетной нормализацией так:

```
  let net = [ Linear' w1
            , Batchnorm1d (zeros h1) (ones h1) (ones h1) (zeros h1)
            , Activation Relu
            , Linear' w2
            , Batchnorm1d (zeros h2) (ones h2) (ones h2) (zeros h2)
            , Activation Relu
            , Linear' w3
            ]
```

Число входов это произведение `28×28=784` пикселей картинки и а выход это число классов т.е. цифр. Мы случайным образом сгенерировали начальные веса `w1`, `w2`, and `w3`. И указали начальный состояние слоя пакетной нормализации следующим образом: среднее число - 0, распределение - 1, параметры масштабирования - 1, и смещение параметров - 0:

```
  let [i, h1, h2, o] = [784, 300, 50, 10]
  (w1, b1) <- genWeights (i, h1)
  let ones n = A.replicate Par (Sz1 n) 1 :: Vector Float
      zeros n = A.replicate Par (Sz1 n) 0 :: Vector Float
  (w2, b2) <- genWeights (h1, h2)
  (w3, b3) <- genWeights (h2, o)
```

Помним, что число пакетной нормализации должно быть равно числу нейронов. Это распространенная практика ставить пакетную нормализацию перед функцие активации, однако эта последовательность не обязательна, можно и на оборот. Для сравнения, мы составим нейронную сеть с двумя спрятанными слоями без пакетной нормализации.

```
  let net2 = [ Linear w1 b1
             , Activation Relu
             , Linear w2 b2
             , Activation Relu
             , Linear w3 b3
             ]
```

В обоих случаях выход активационной `softmax` избегается так как вычисляются совместно с градиентами потерь в конечном рекурсивном вызове `_pass`:

```
    _pass inp [] = (loss', pred, [])
      where
        pred = softmax inp
        loss' = compute $ pred .- tgt
```

Здесь `[]` с левой стороны значения это пустой список входных слоёв, а `[]` с правой стороны это пустой конец вычисленных значений в начале обратного прохода.

Подводные камни пакетной нормализации

Есть несколько потенциальных ловушек при использовании пакетной нормализации.

- Пакетная нормализация отличается во время обучения и взаимодействия. Что делает вашу реализацию более сложной.
- Пакетная нормализация может не получиться при обучении данных которые идут из разных наборов. Чтобы избежать этого, нужно просто убедиться, что любой пакет отражает целый набор данных. То есть все пакеты имеют одинаковую форму.

Выводы

Не смотря на подводные камни, пакетная нормализация важная идея и остается популярным методом в контексте глубого обучения. Сила пакетной нормализации в том, что она может существенно сократить число обучаемых эпох или даже помочь достичь лучше точности нейронной сети. После этого обсуждения, мы готовимся приступь к понятиям сверточные нейронные сети и обучение с подкреплением. Оставайтесь на связи.

Что почитать.

- Richard Sutton. The Bitter Lesson
- Using neural nets to recognize handwritten digits
- Batch normalization paper
- How Does Batch Normalization Help Optimization?
- On The Perils of Batch Norm

Haskell

- Why I think Haskell is the best general purpose language
- An opinionated beginner's guide to Haskell in mid 2019
- Massiv Array Library
- Alexey's presentation about massiv

Revision #8
Created 2022-09-03 08:52:13 UTC by gasick
Updated 2023-07-15 08:17:52 UTC by gasick