ADSC
Applied Data Science Center

Тренды в анализе данных. Статья 2

О цикле

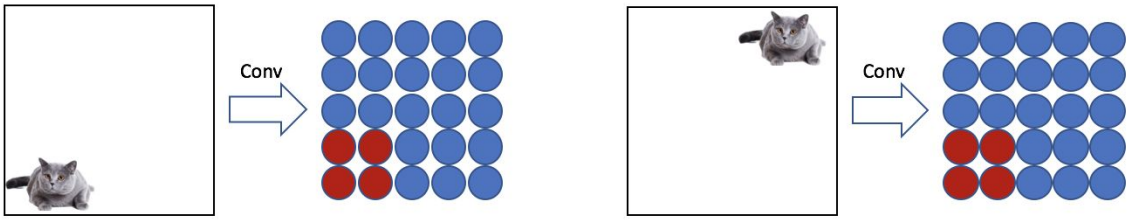
Данный цикл статей основан на семинарских занятиях открытого курса Data Mining in Action по направлению “тренды в анализе данных”. На семинарах (и в статьях по ним) мы будем говорить о последних достижениях в области data science.
Общую лекцию для всего потока можно посмотреть здесь, а презентацию к ней скачать здесь.

Сегодня в статье:

1. Инвариантность и эквивариантность в нейросетях
2. Spatial Transformer Networks (STN)
3. Capsule Networks (CapsNet)

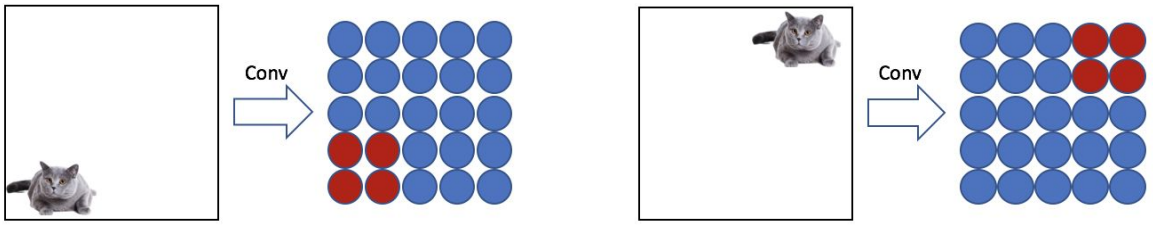
Инвариантность и эквивариантность в нейросетях

Инвариантность: $f(T(x)) = f(x)$



Для начала вспомним, что такое **инвариантность**. Пусть у нас есть некоторая картинка с котом. Синие и красные кружки на картинке - это карта признаков (feature map) после свертки. Кроме этого, как обычно происходит в CNN, у нас есть какие-то нейроны, которые активируются, когда видят кота.
Перемещаем кота из нижнего левого угла изображения в правый верхний угол - активируются те же нейроны. Это и есть инвариантность. Если говорить формально, то инвариантность - это такое свойство функции, при котором, если мы применим трансформацию ко входу, выход функции никак не изменится.

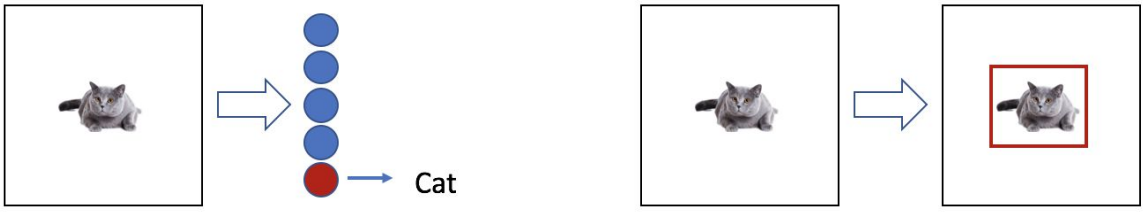
Эквивариантность: $f(T(x)) = T(f(x))$



В свою очередь **эквивариантность** - это такое свойство, что если мы переместим кота на картинке в другое место, то у нас активизируются другие нейроны. Формально: если мы применим некоторую трансформацию на вход, то, применив ту же трансформацию на выход функции, мы получим то же самое.

Чтобы классифицировать объект нужна инвариантность.

Чтобы сегментировать объект нужна эквивариантность.

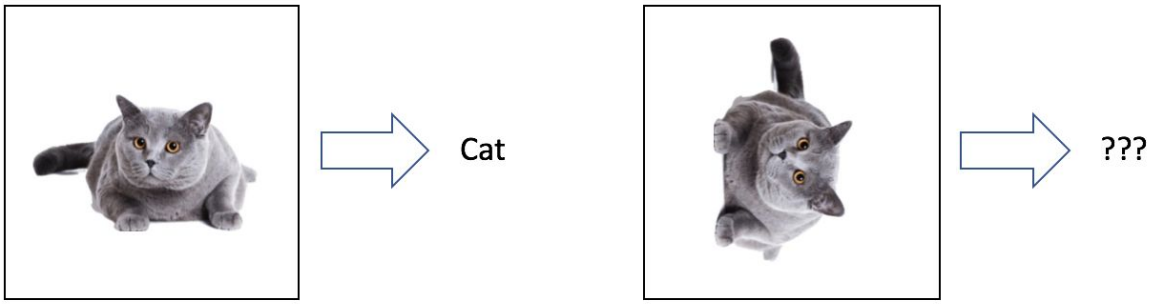


В задачах классификации инвариантность очень полезна, так как мы можем двигать кота как угодно: если он есть на картинке, то мы его найдем. А если мы хотим сегментировать что-то (в нашем случае - локализовать кота), то тогда нужна эквивариантность.

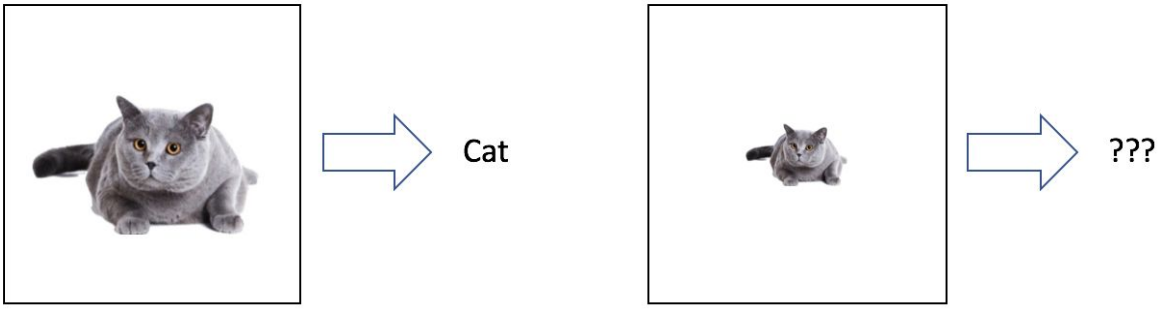
Теперь поговорим про инвариантность и эквивариантность в свертках. Свертки эквивариантны к **сдвигам**: если мы подвинем кота, зажгутся другие нейроны. Сама сеть, в свою очередь, инвариантна к сдвигам, так как выход будет тем же - это кот!

Возникает вопрос: а как сделать саму операцию свертки инвариантной к сдвигам? Можем использовать pooling, и тогда свертка будет инвариантна к малым сдвигам. Например, у нас есть окно 2x2, и в нем зажегся один из четырех нейронов. Мы пулим их, то есть, берем максимум из четырех. И если мы немного подвинем кота, то зажжется другой нейрон, но максимум будет такой же. Работает это, правда, только с маленьким сдвигом, так как у нас

маленькое окно. А брать большое окно - плохо, в таком случае теряется много информации.

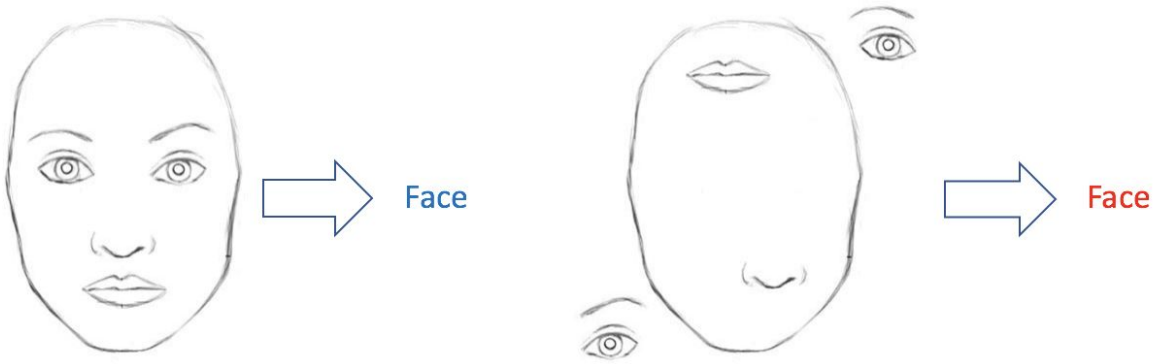


Кроме этого, у сверток есть проблемы - например, **повороты**. Конечно, если обучаться на ImageNet, это не будет проблемой, ведь там много позиций одного и того же объекта. Но если мы выкачаем из гугла картинки котов, то они, в основном, будут вертикальны. И когда мы обучимся на таких данных и дадим нейронной сети повернутого кота - сеть может не справиться.



Еще одна проблема - **изменение масштаба**. Если у вас очень глубокая сеть, то все будет в порядке: в ней много параметров, и она может выучить такие комбинации low-level и high-level признаков, что сможет найти вам кота, какой бы он ни был по размеру. Однако если у вас маленькая сеть, например, два слоя convolution, global pool и softmax, то она может не распознать маленького кота. Получается, что такая сеть инвариантна к сдвигам, но не инвариантна к поворотам и изменению масштаба. Это проблему мы и хотим решить.

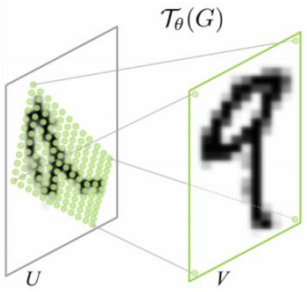
Примитивный способ, как сделать нейросеть инвариантной к поворотам и изменениям масштаба - это data augmentation. Мы расширяем и дополняем нашу выборку, генерируя дополнительные примеры из изображений путем применения к ним разных деформаций (поворотом, изменения масштаба). Это работает, но учится долго и допускает ошибки. А что делать, если нам нельзя ошибаться, и мы хотели бы полагаться на более обоснованные теоретические методы, нежели data augmentation? Например, в медицине нам совершенно не хочется иметь false negative (то есть, не детектировать какую-то серьезную болезнь). Именно поэтому люди придумывают интересные решения.



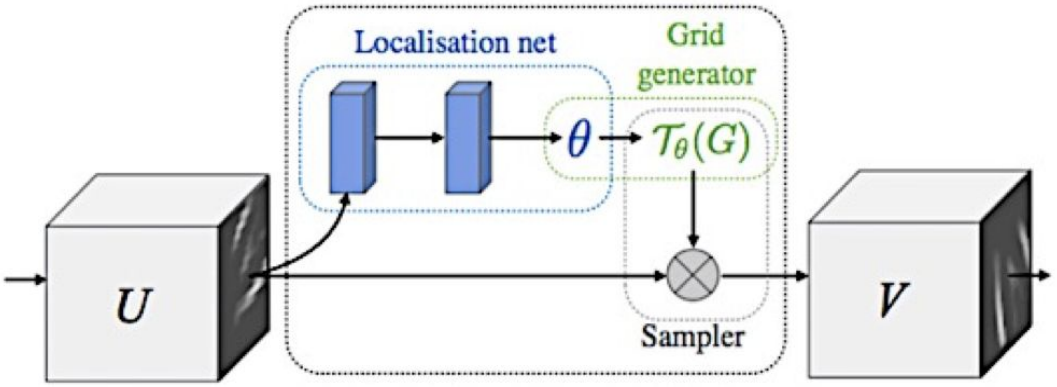
Еще один вопрос, который редко появляется на практике, но представляет научный интерес, - это поведение нейросети при изменении взаимного расположения компонентов одного объекта. Если мы в компонентах какого-то объекта изменим их взаимное расположение, то CNN, скорее всего, сообщит вам, что это тот же объект. Разумеется, если у нас очень глубокая сеть, то она может выучить разные репрезентации лица и понять, что объект на правой части картинки - не лицо. У нее будет конкретная карта признаков, по которой мы сделаем global pool, и он скажет - лицо это или нет.
В то же время простая сеть, вероятно, проверит только наличие частей лица, а не их взаимное расположение. У нее есть карта признаков на части лица, и, когда сеть их заметит и сделает global pool, то softmax уверенно скажет, что если на изображении есть нос, рот и глаза, то это точно лицо.

Spatial Transformer Networks (STN)

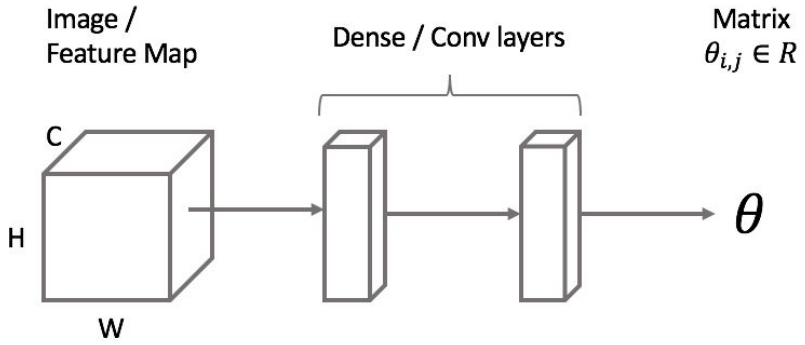
Бывает, что пишут такие статьи: что если просто прикрутить еще одну сеть, которая будет делать то, что мы хотим? Иногда это поразительно хорошо работает. Мы рассмотрим статью [1], в которой предлагается сделать сеть инвариантной к поворотам и изменению масштаба. Идея следующая: у нас нет инвариантности к поворотам и изменению масштаба, и это плохо. В таком случае просто сделаем еще одну сеть-модуль (Spatial Transformer), которая поможет нам получить инвариантность и будет учиться end-to-end (то есть ничего нового делать не надо, просто вставить этот модуль в сеть). Тогда мы получим асимптотически инвариантную к поворотам и изменению масштаба сеть. Разберемся, как это работает.



Если говорить коротко, то Spatial transformer решает нашу проблему отсутствия инвариантности следующим образом. Перед тем как подать в основную сеть картинку, он преобразовывает ее до некоторого нормального вида, к которому привыкла сеть: например, растягивает или поворачивает.



Посмотрим на архитектуру. Здесь вы подаете какое-то изображение (или стек из карт признаков), после чего оно идет в сеть Localisation net (любой архитектуры), и сеть выдает какую-то матрицу параметров θ. Далее эта матрица идет в Grid Generator, который по ней выдает новое изображение.



In: изображение или Feature Map размера (H, W, C).

Out: матрица θ размера (6,).

Pipeline: вы сделали пару сверток, получили стек из карт признаков и подали их в модуль Spatial transformer. После, например, идет два полносвязных слоя. На выходе сеть выдает матрицу θ.

Есть ограничение: матрица θ должна быть размера шесть (2x3). Почему именно шесть? Когда мы умножаем картинку на какую-то матрицу, мы делаем аффинное преобразование (сдвинуть, обрезать, повернуть, растянуть). И авторы статьи утверждают, что, если мы умножаем картинку на матрицу 2x3, это позволит нам сделать **любое** аффинное преобразование.
После того как мы вычислили θ, мы хотим взять исходное изображение и умножить на эту матрицу, чтобы получить новое изображение. В этом нам поможет Grid Generator.

Output from Localization Net

$$\begin{bmatrix} x^s \\ y^s \end{bmatrix} = \begin{bmatrix} \theta_{11} & \theta_{12} & \theta_{13} \\ \theta_{21} & \theta_{22} & \theta_{23} \end{bmatrix} \begin{bmatrix} x^t \\ y^t \\ 1 \end{bmatrix}$$

s – source image/FM, *t* – target image/FM

Рассмотрим, как он работает. Здесь x^t, y^t - это просто сетка (grid) от 0 до 256 (если изображение 256x256) для каждого пикселя на новом изображении. Мы растягиваем 256x256 на вектор размера nx1, где каждый n - это x^t, y^t. Перемножив этот вектор на θ, получаем координаты x^s, y^s для каждого x^t, y^t, которые мы будем использовать, чтобы построить новое изображение.
Например, x^t, y^t - это первый пиксель (1, 1). Допустим, после перемножения на θ у нас получится (2, 2). То есть, x^s, y^s - это координаты (2, 2) в исходном изображении. Тогда для первого пикселя из нашей сетки (grid) мы возьмем пиксель (2, 2) из изначального изображения. По факту мы ищем обратное аффинное образование.
Но что делать, если у нас не получатся целочисленные координаты, и какое брать значение пикселя?

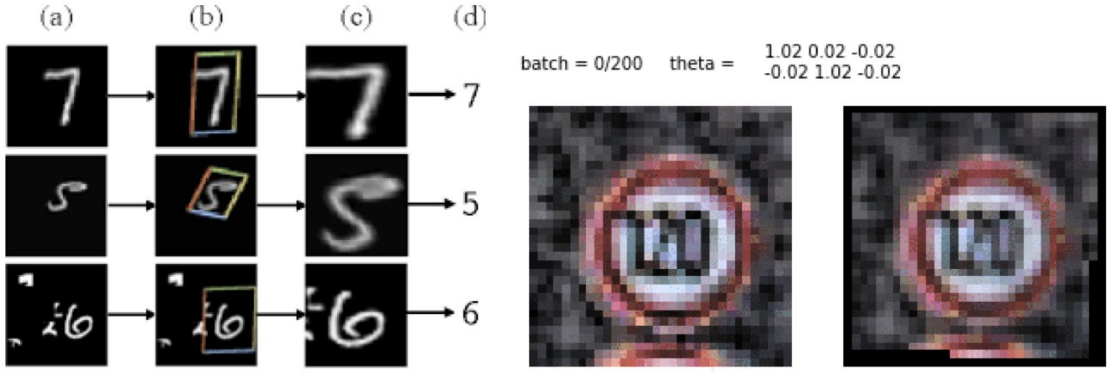
$$pixel\ value\ at\ (x^t, y^t) = pixel\ value\ at\ (k(x^s, y^s))$$

k – интерполяция (например, билинейная).

Мы уже знаем координаты, но они могут быть не целочисленные, и если мы хотим узнать значения пикселей в этих координатах, то придется интерполировать. Например, у нас получилось (x^s, y^s) = (1.5, 2.5). Тогда мы

берем четыре соседних пикселя и усредняем по ним значения (если интерполяция билинейная). Но если нам повезло сразу получить целочисленную координату, то интерполировать не нужно. Авторы статьи не указывают, как обрабатывать ситуацию, если мы получили в координатах минус, но обычно это просто обрезается до изначального изображения.
Отметим, что интерполяция может быть какой угодно, главное - дифференцируемой.

Результат работы Spatial Transformer:



Хочется заметить, что самое прекрасное в этом решении то, что оно end-to-end, и никаких дополнительных телодвижений производить не нужно. Но есть свои нюансы: на сложных датасетах обучение становится несколько нестабильным. Конечно, люди начинают придумывать разные приемы, чтобы его стабилизировать, но их в рамках этой статьи мы рассматривать не будем.

Capsule Networks (CapsNet)

Теперь рассмотрим классификацию в условиях изменения взаимного расположения компонентов объекта, предложенную в статье [2]. Вспомним, что мы рассматривали эту проблему на примере классификации изображений лица. Нам хочется, чтобы сеть обращала внимание не только на присутствие каких-то элементов, но и на их взаимное расположение.
Эта хорошая идея, но появилась она не из-за того, что мы часто ошибаемся на “измененных” объектах вроде лица с перемещенными чертами - мы можем справиться с этим, просто сделав более глубокую сеть. Идея зародилась у Джеффри Хинтона, автора статьи, когда он занимался автоэнкодерами. В своей статье 2011 года он говорил, что было бы интересно понимать не только то, что мы реконструируем, но и где это находится на картинке. Тогда эта идея не получила развития, так как модель требует много вычислений.

Капсула – группа нейронов, чей вектор характеризует:

- Уверенность сети в том, что объект (или его часть) присутствует в какой-то части картинки.
- Параметры расположения этого объекта относительно других объектов.
- Параметры освещения, деформации, поворота и т.п.

Его предложение - капсульные нейросети. Капсула - это группа нейронов (то есть, вектор), где уверенность в присутствии объекта на картинке выражается не самим значением капсулы, а ее L2-нормой. Тогда, по предположению автора, элементы этого вектора каким-то образом хранят информацию о параметрах объекта: например, его освещении, деформации, взаимном расположении с другими объектами и так далее. Отдельно отметим, что мы сами не закладываем в эти элементы какой-то смысл, однако на деле они действительно могут оказаться интерпретируемыми. Иными словами, в значениях капсулы будет отражена не только информация о наличии объекта, но и какие-то важные параметры.
Автор предполагает, что эту идею можно реализовать многими способами, после чего описывает один из них.

Итак, с нейронами у нас обычно есть какая-то нелинейность вроде ReLU; мы знаем, как это работает только для одного нейрона, но не для капсулы. Очевидно, что теперь нам необходима какая-то общая нелинейность для вектора капсулы. Автор предлагает следующее:

$$\mathbf{v}_j = \frac{||\mathbf{s}_j||^2}{1 + ||\mathbf{s}_j||^2} \frac{\mathbf{s}_j}{||\mathbf{s}_j||}$$

s_j – вход капсулы j, v_j – вектор капсулы j

Такая нелинейность приводит очень длинный вектор к длине около 1, а очень маленький к длине около 0.

Здесь s_j - это j-й вход капсулы, который мы нормализуем (приводим к норме 1). Формула дает нам следующее: если норма s_j очень большая, то мы получим единицу, а если очень маленькая, то ноль. Таким образом мы сохраняем

свойство капсул отражать уверенность в присутствии объекта на определенном участке изображения и быть в интервале от 0 до 1. Конечно, здесь можно попробовать сделать и другую нелинейность с вектором, но Хинтон в своей статье говорит, что это самый простой способ (и это работает).

$$\mathbf{s}_j = \sum_i c_{ij} \hat{\mathbf{u}}_{j|i} \ , \qquad \hat{\mathbf{u}}_{j|i} = \mathbf{W}_{ij} \mathbf{u}_i$$

Дальше посмотрим на то, как капсулы связываются между собой. В правой формуле u_i - это выход предыдущей капсулы (вектор размера n). W - обычная матрица весов, которая учится backprop'ом. Если бы не левая формула, все было бы почти так же, как в обычной CNN. Но когда мы получаем входной вектор из капсул, умноженных на матрицу весов, то мы говорим, что хотели бы их как-то взвесить. Например, если капсула на верхнем уровне сообщает, что на изображении есть лицо с какими-то параметрами, а на ранних уровнях мы обнаружили рот, нос, глаз и хвост, то мы хотим взвесить их так, чтобы нос, глаз и рот были с бóльшим весом, нежели хвост (если он не находится прямо на лице). Таким образом, c_{ij} - это коэффициент взвешивания (от 0 до 1). И тогда мы умножаем трансформированный с помощью матрицы весов вход предыдущего слоя на коэффициент c_{ij} и получаем вектор s_j.
Идея этих коэффициентов c_{ij} основана на том, насколько часто эти объекты встречаются вместе (сеть это учит сама). На эти коэффициенты есть ограничение: для связок u_i со всеми s_j в сумме коэффициенты равны единице. Это работает так же, как в softmax.

Procedure 1 Routing algorithm.

procedure ROUTING($\hat{\boldsymbol{u}}_{j|i}, r, l$)
  for all capsule i in layer l and capsule j in layer $(l+1)$: $b_{ij} \leftarrow 0$.
  for r iterations **do**
    for all capsule i in layer l: $\mathbf{c}_i \leftarrow \texttt{softmax}(\mathbf{b}_i)$
    for all capsule j in layer $(l+1)$: $\mathbf{s}_j \leftarrow \sum_i c_{ij} \hat{\mathbf{u}}_{j|i}$
    for all capsule j in layer $(l+1)$: $\mathbf{v}_j \leftarrow \texttt{squash}(\mathbf{s}_j)$
    for all capsule i in layer l and capsule j in layer $(l+1)$: $b_{ij} \leftarrow b_{ij} + \hat{\mathbf{u}}_{j|i}.\mathbf{v}_j$
  return $\mathbf{v}_j$

$$\text{squash}\,(s_j) = \frac{||\mathbf{s}_j||^2}{1 + ||\mathbf{s}_j||^2} \frac{\mathbf{s}_j}{||\mathbf{s}_j||}$$

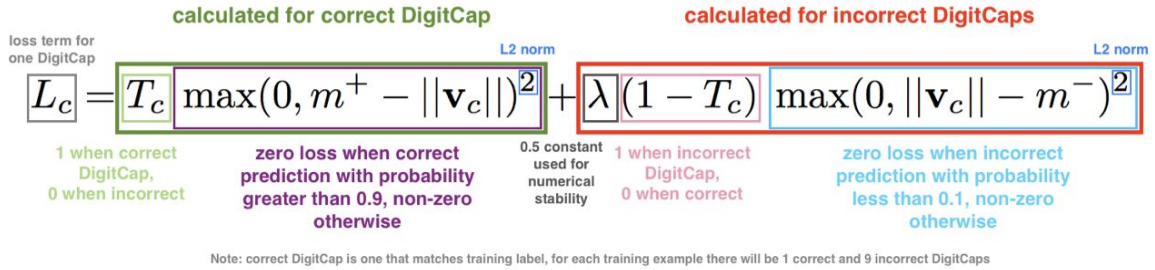
Как подбираются веса между капсулами c_i? Посмотрим на код, где описана процедура Routing. Разберем построчно, что здесь происходит.
2: Сначала инициализируем нулями все коэффициенты для того, чтобы C были одинаковы. Коэффициенты b отвечают за то, чтобы C были от 0 до 1 (как в softmax). Это считается только во время прямого прохода.
3: Мы говорим, что у нас есть r итераций функции роутинга.
4: Для всех капсул i (это наши примитивные капсулы на предыдущем слое) инициализируем c_i, чтобы в сумме получилась единица, а каждый коэффициент был от 0 до 1.
5: После этого для каждой капсулы j мы взвешиваем выходы предыдущих капсул, складываем их и получаем вход.
6: Получаем выход капсулы, сделав нелинейную операцию.
7: А дальше обновляем b, добавляя к ним скалярное произведение между выходами предыдущих капсул и выходами этих капсул.

Рассмотрим пример. Пусть у нас есть примитивная капсула, и направление ее вектора показывает, где находится нос. Далее у нас есть следующая капсула, которая показывает на лицо. Если направления этих векторов совпадает, то этот нос относится к этому лицу.
Если применить модель к изображению, где нос и лицо находится в разных местах, то скалярное произведение будет минус бесконечность, так как у векторов разное направление. Это все нужно для того, чтобы реализовать некоторое согласование капсул нижнего и верхнего уровня и понимать, где находится главный объект (то, что мы ищем).
Можно интерпретировать по-другому. Например, на картинке есть лицо в тени, а глаз будет находится на резкой вспышке. Тогда вектора освещения будут указывать в разные стороны. И когда мы прибавим к некоторому b, который был равен нулю, отрицательное число и пропустим через softmax, то мы получим число, близкое к нулю. А значит мы не будем обращать внимания на предыдущую капсулу (глаз), когда будем складывать все входы и делать вывод, что там есть лицо.
Отметим, что нам не нужно оптимизировать коэффициенты C и учить их. Каждый раз при прямом проходе мы считаем их заново. А когда идем backprop'ом, мы считаем их константами. Таким образом, мы считаем C, чтобы понять, на что из предыдущего слоя нам нужно обратить внимание на этом слое.

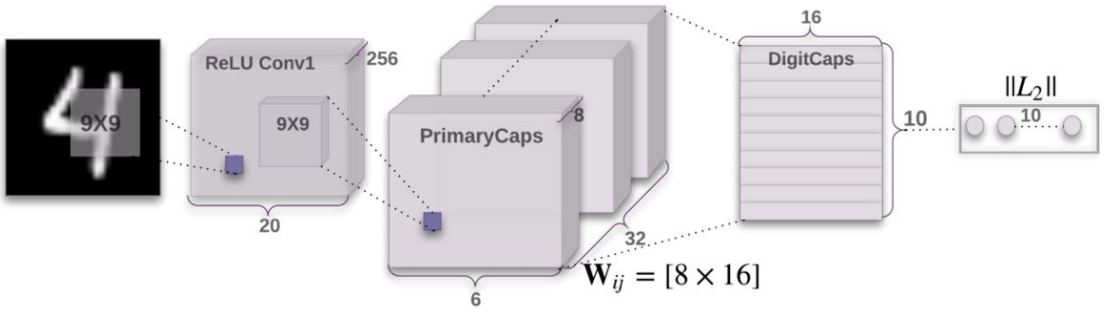
Теперь о том, как выглядят финальные капсулы. Если у нас классификация, на 10 классов, то на выходе будет 10 капсул, которые хранят информацию о параметрах финальных классов, и нормы их векторов отражают уверенность капсул в присутствии на изображении каждого из классов.

$$L_k = T_k \ \max(0, m^+ - ||\mathbf{v}_k||)^2 + \lambda \ (1 - T_k) \ \max(0, ||\mathbf{v}_k|| - m^-)^2$$

В статье автор предлагает свою функцию потерь, которая выглядит как квадратичный hinge и кастомные пороги. И эти пороги просто подбираются.

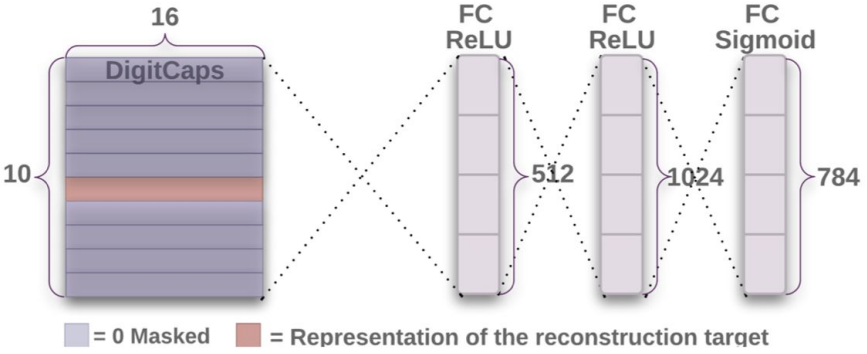


Здесь T_c, как в обычной логистической функции потерь, - ноль или единица. Дальше идет ReLU. Норма v_c - это норма нашей финальной капсулы, то есть, наше предсказание. Коэффициенты m+ = 0.9, а m- = 0.1. Соответственно, для корректного класса потеря будет нулевая, если предсказали 0.9 и выше. Это очень похоже на hinge, но при этом мы возводим все в квадрат. По экспериментам можно увидеть, что, если квадрат не ставить, учиться будет чуть быстрее. Однако так мы не дотягиваем последние доли процентов. Кроме того, автор утверждает, что с λ = 0.5 все учится стабильнее. Этот коэффициент нужен, чтобы негатив не перевесил позитив. Заметим, что функция потерь считается не только для нужной капсулы, но и для остальных тоже. То есть, если классов 10, то мы проходимся 10 раз для одного примера, а потом складываем.



Посмотрим на архитектуру в целом. Мы подаем на вход некоторую картинку из MNIST размера 28x28 и проходимся по ней обычными свертками 9x9 (со страйдом 1). Получаем карту признаков 20x20x256. Далее в игру вступают первые капсулы, которые по сути - та же самая свертка по входам. Мы делаем большое количество карт признаков - 32x8. Каждая группа из 32-х - это

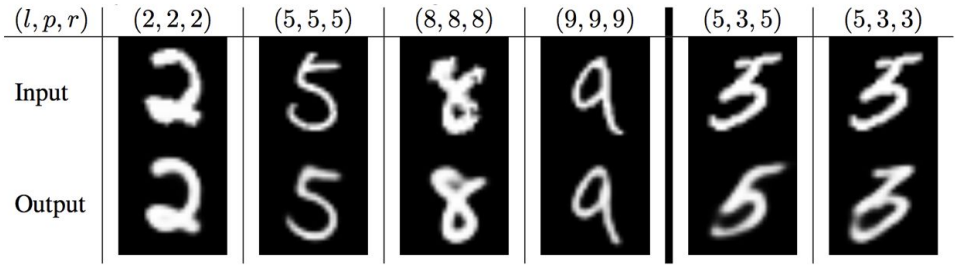
“архикапсула”, которая состоит из 36 примитивных капсул и выдает важную информацию.
Дальше есть матрица 8x16, которую мы перемножаем с каждой примитивной капсулой (всего их получается 1152).



И, наконец, есть финальные 10 капсул, в каждую из которых мы подаем по 1152 капсулы. Именно на этом этапе вступает в силу то, что мы умножаем на веса и подбираем C. В итоге, норма финальных капсул - это ответ, а значения - какие-то параметры.

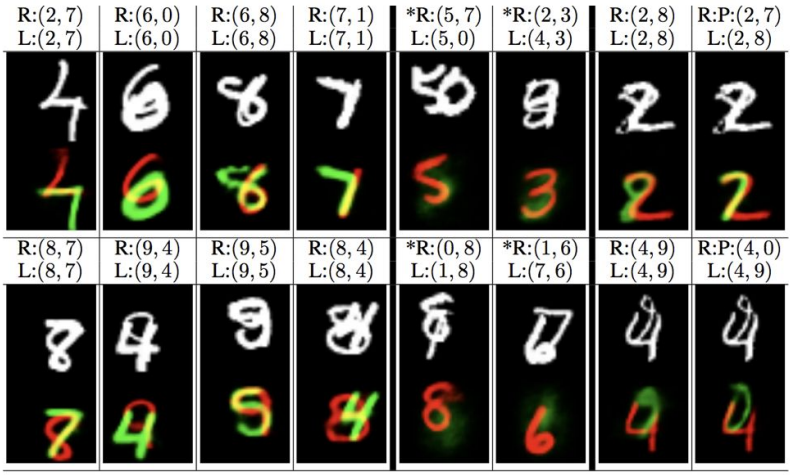
Дальше предлагается добавить регуляризацию. Вспомним, что вся идея зародилась от автоэнкодеров. Попробуем из этих капсул реконструировать оригинальное изображение. Есть финальная капсула для какого-то класса. Мы берем ее и, так как там есть какие-то параметры, попробуем реконструировать изображение. А дальше потерю этой реконструкции с коэффициентом 0.005 добавим к функции потерь, которую мы уже разбирали.

В итоге, эта сеть довольно сложно и долго учится и, к сожалению, пока что работает хорошо только на простых датасетах. Но этот метод бьет state-of-the-art на MNIST.

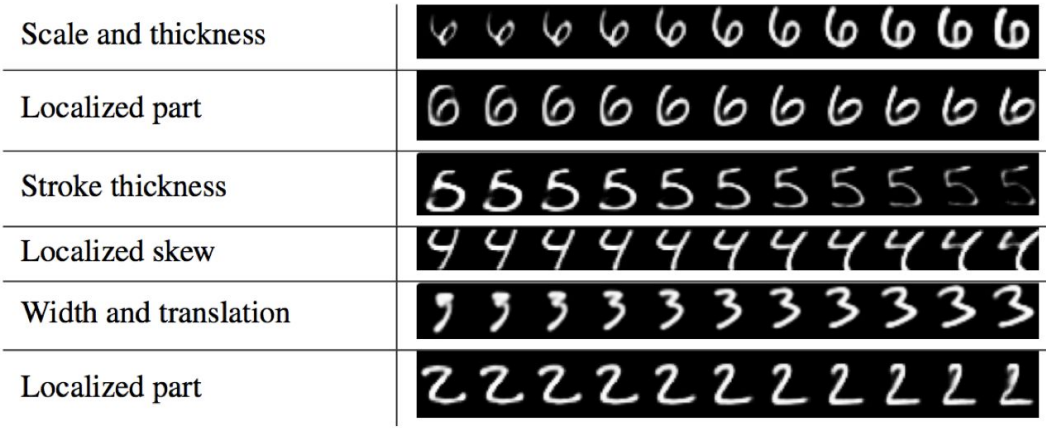


Тут тройки (*l, p, r*) - это корректный класс, предсказание модели и цель реконструкции. У сети получатся хорошие реконструкции из векторов капсул (четыре примера слева). Две правые картинки, на которых сеть ошиблась в классификации (предсказала 3 вместо 5), позволяют понять, почему модель

ошиблась на этом примере. В случае 5,3,5 сеть предсказала 3, корректный класс был 5, и мы попытались реконструировать изображение из капсулы для 5. На самой правой картинке мы попросили сеть реконструировать изображение из капсулы для 3.



Если подавать сети наложенные картинки из MNIST, то сеть очень хорошо сегментирует эти наложения.



Вспомним, что еще мы получили вектор, в котором есть некоторые параметры для каждой цифры. Оказывается, что если начать варьировать их, то по полученным деформациям мы видим, что параметры действительно что-то характеризуют. К примеру, мы меняем какую-то координату и получаем что-то похожее на искривление, смещение, регулировку толщины.

Смысл не в том, чтобы взять изображение четверки и сделать ее кривой, а в том, что, если сеть разучивает что-то интерпретируемое, мы можем узнать об объектах больше и понять, как это работает. Deep learning обычно называют черным ящиком - но не потому, что мы не знаем как он работает (мы знаем), а потому, что мы не знаем, какой вклад вносит каждый нейрон. Изложенный

метод дает нам в финале какую-то интерпретируемость. И это очень интересный результат для глубокого обучения.

Оставить отзыв по прочитанной статье вы можете здесь. По всем вопросам пишите на почту курса: dmia@applieddatascience.ru

Список литературы

Оригинальные статьи

1. Spatial Transformer Networks
2. Dynamic Routing Between Capsules

Дополнительная литература

1. Базовый туториал по ST
2. Объяснение ST на русском
3. Отличный туториал по CapsNet
4. Объяснение CapsNet в 4-х частях
5. Страничка с тысячей ресурсов по CapsNet