Тренды в анализе данных. Статья 6

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. Super-resolution
2. Texture Synthesis
3. Style Transfer
4. Conditional Generation
5. PixelRNN и PixelCNN

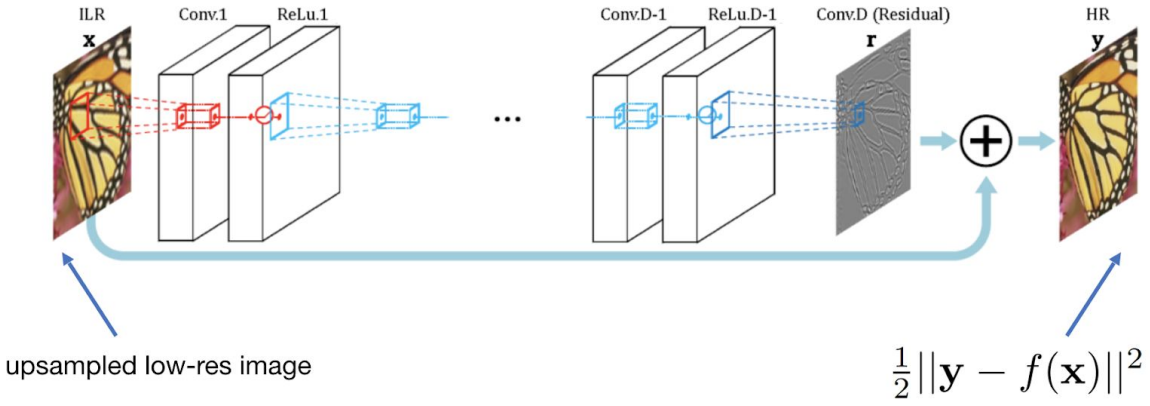
Статья посвящена генеративным моделям - таким, которые могут генерировать новые изображения или изменять старые.

Если мы посмотрим на фундаментальную задачу генеративных моделей, то это окажется unsupervised learning. Тем не менее, конкретные задачи могут отличаться друг от друга.
Зачем нам вообще unsupervised learning? Общая задача - понять какую-то скрытую структуру данных, с которыми мы имеем дело. Это можно использовать для чего угодно: например, для генерации или кластеризации. А вот в unsupervised есть еще много интересного, что не реализовано.
Вот что забавно: любая задача, когда у нас мало размеченных данных, но много неразмеченных может решается смесью one-shot learning и unsupervised learning.

Super-resolution

Начнем с простого. У нас есть какое-то маленькое изображение (например, 32x32), и мы хотим увеличить его (например, до 128x128) так, чтобы качество было очень хорошим. Если мы сделаем какую-то простую интерполяцию, то качество хорошим не будет.
Классическая постановка задачи: есть картинка в хорошем качестве. Мы хотим сжать ее и после этого прогнать через некоторую модель, чтобы снова

получить большую картинку, которая будет по качеству такой же, как и исходная.



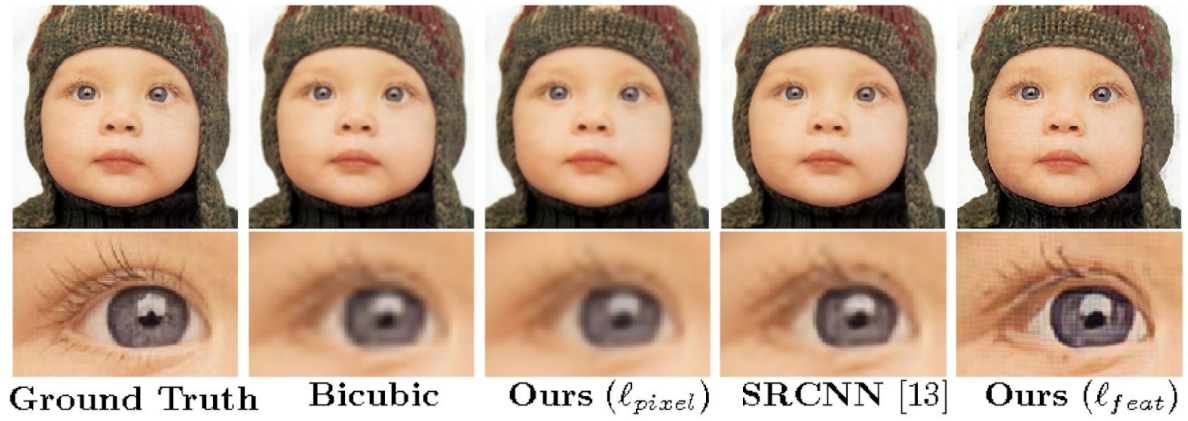
Приятная особенность задачи:
Не нужна разметка данных

Итак, у нас есть маленькое изображение 32x32, которое подается на вход. Мы увеличиваем его до размера, который нас интересует (например, 128x128). Можно делать это по-разному: первые работы увеличивали бикубической интерполяцией или unpooling. В современных статьях используют transpose convolution. Мы делаем upsampling изображения и оставляем тензор то есть, мы сохраняем матрицу HxWx3 (3 - RGB). И при этом его же пропускаем через мощную сеть вроде Fully Convolutional. После этого мы получаем стек карт признаков, складываем его с оригинальным изображением (как в ResNet) и получаем аутпут. Учится это обычно на MSE.

Идея в том, что мы хотели бы делать upsampling так, чтобы изображение было максимально похожим на оригинальное. У нас была картинка бабочки 128x128, мы ее сжали до 32x32, пропустили через сетку и теперь хотим, чтобы полученное изображение было таким же, как исходное. Поэтому мы используем MSE. Самое приятное в том, что это задача unsupervised, и размечать ничего не нужно.

Сейчас в некоторых несложных задачах (когда нет большой вариации в том, что изображено на картинках: например, мы имеем дело только с картинками котов) Super-resolution по ощущениям людей выглядит лучше, чем исходные картинки. Проводят эксперименты по сравнению, и людям кажется, что они более настоящие.

Иногда сначала делают Super-resolution для того, чтобы машина лучше классифицировала изображения. Но это работает тогда, когда исходные изображения маленькие.



Разумеется, даже простыми методами можно сделать намного лучше, чем бикубическая интерполяция, но в этой сфере этот метод считается классикой, поэтому с ним часто сравниваются.

Как это обучается

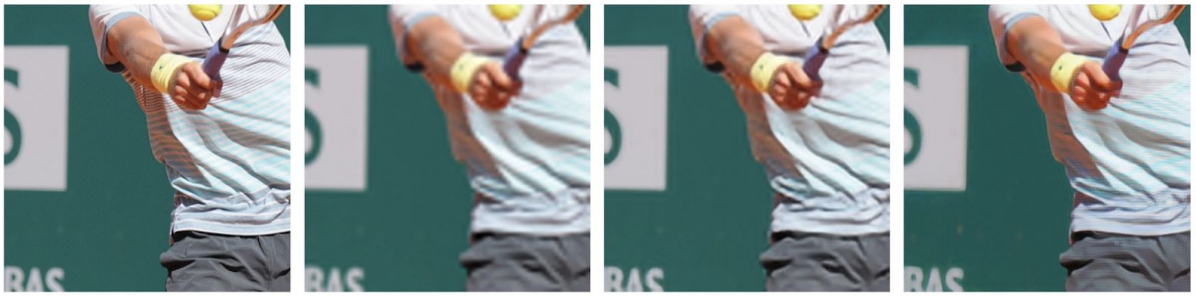
Pixel-wise MSE	Perceptual loss
$\sum_i \|f(x_i;\theta) - y_i\|^2$	$\frac{1}{C_j H_j W_j} \|\phi_j(\hat{y}) - \phi_j(y)\|_2^2$

Пиксельная функция потерь. Мы действительно берем каждый пиксель из оригинального изображения и сравниваем с тем, что мы получили, и считаем MSE по всем этим пикселям. Однако есть проблема. Вспомним l2-регуляризацию, она размазывает веса. А MSE размывает картинку. Он чувствителен к среднему по картинке, поэтому будут смазаны все грани

изображения и картинка в целом. Таким образом, если мы обучаем на MSE, у нас получаются размытые картинки.

Perceptual loss. Посмотрим на формулу. Первый множитель - это нормализация. Мы делим единицу на перемноженную размерность тензора (стека карт признаков) - это наш нормировочный коэффициент. А дальше φ_j - это выход предпоследнего слоя какой-то предобученной, например на ImageNet, нейросети. Вектор такого выхода будет содержать признаки, относящиеся к этому изображению. Как правило, используют VGG: так сложилось исторически, потому что такая сеть обладает достаточной репрезентативной мощностью для задания и экстрактирует лучше интерпретируемые признаки, чем, например, ResNet (но это спорно).
Пропускаем оригинальное изображение через VGG, получаем тензор. Пропускаем то изображение, которое получилось после Super-resolution, снова получаем тензор. И потом просто берем евклидову дистанцию между этими тензорами. То есть, если у вас были одинаковые фотографии, то должны быть и одинаковые признаки.

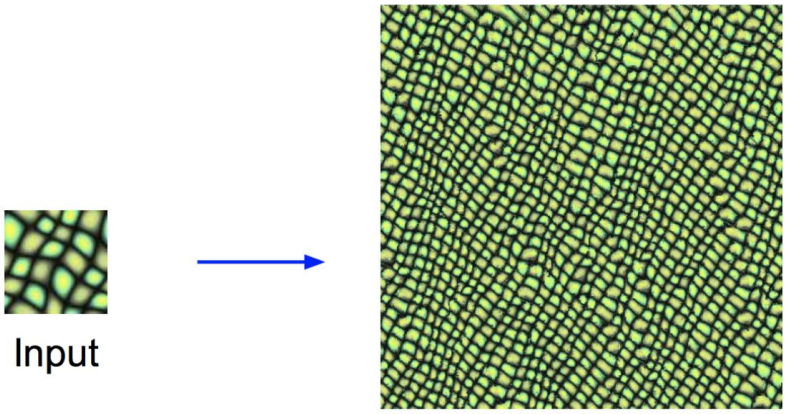
Почему это должно помогать в Super-resolution? В этом векторе могут содержаться признаки, которые косвенно отвечают за resolution. Значит, если мы в этом пространстве будем минимизировать ошибку, то это будет гораздо лучше, чем пытаться что-то делать с помощью MSE.



(SRCNN - это сеть с MSE)

Texture Synthesis

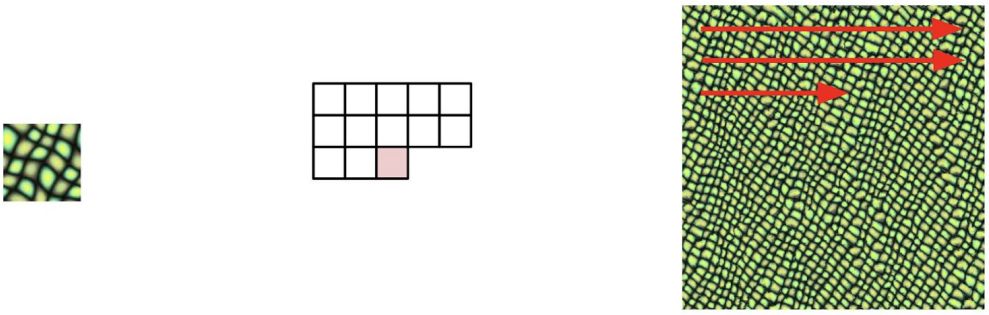
Задача следующая: у нас есть маленькая часть текстуры (например, мы сфотографировали только часть чего-то), и хотелось бы это как-то разумно размножить, чтобы это смотрелось естественно. Главное, чтобы это не была часть собаки, потому что это будет выглядеть странно. Нам нужна именно текстура - например, галька или кирпичи.



Это востребованная задача уже довольно давно: например, игровая индустрия готова отдать большие деньги за прорывы в этой области. В играх много текстур, и было бы очень удобно прорисовывать их автоматически.

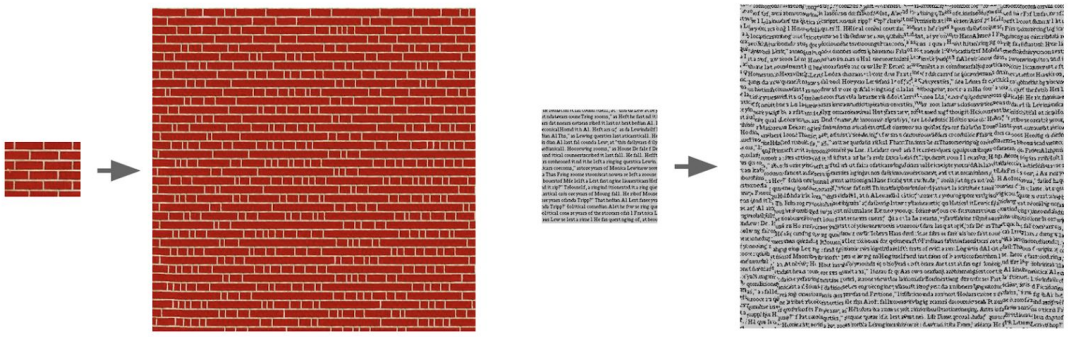
Nearest Neighbor

Первый метод, который мы рассмотрим, - это Nearest Neighbor (около 2000 г). Как он работает?



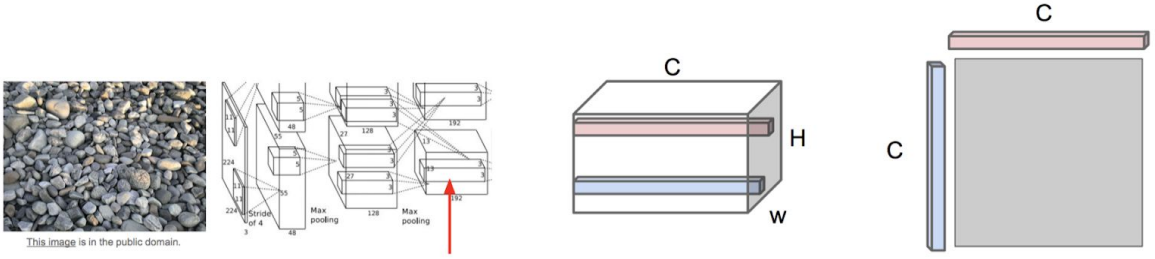
Мы берем начальное изображение и вставляем его в угол какого-то большого пустого изображения. Потом убираем один пиксель из него (красный на картинке) - его мы хотим сэмплировать. Для этого мы обозначим размер ее окрестности и ищем точки с такой же окрестностью. Например, обозначим окрестность пикселя, который нужно сгенерировать, как три пикселя: слева, сверху и слева-сверху. Таким образом получим вектор из значений пикселей в окрестности. Далее ищем самый схожий этому вектору кусочек на исходном изображении. Когда найдем - возьмем из этого батча пиксель, который стоит на той же позиции, как и тот, который мы хотели засемплировать и вставим его на место искомого.

Это работает нормально на каких-то простых текстурах, однако плохо справляется с более сложными.



Gram Metrix

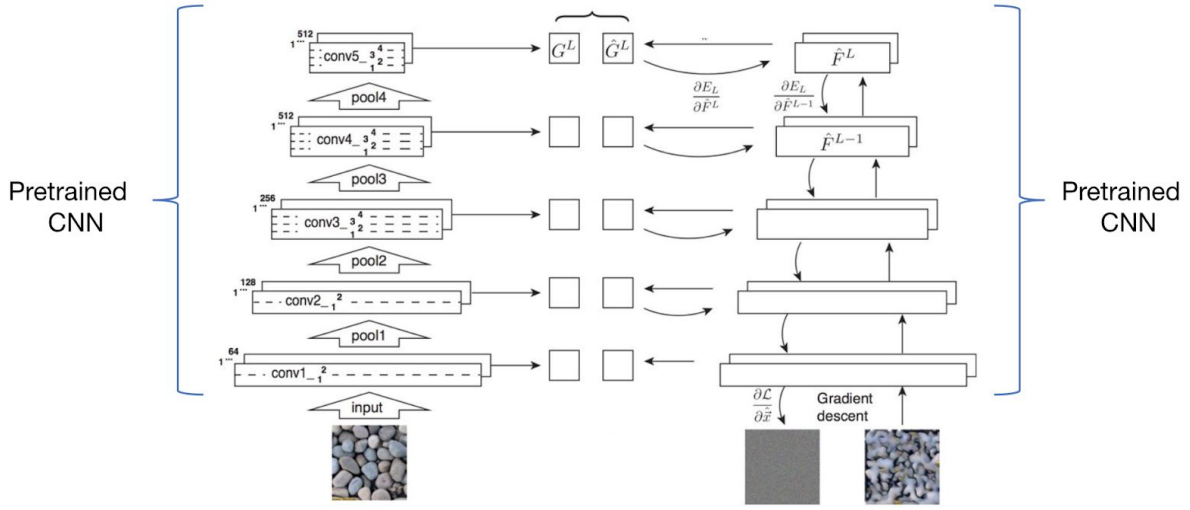
Далее рассмотрим Gram Matrix. Идея: каждый стек карт признаков дает нам тензор, и к этому тензору можно относится как к сетке (grid) C-размерных векторов.



Каждый слой CNN дает тензор C x H x W
или сетку H x W векторов размерности C.

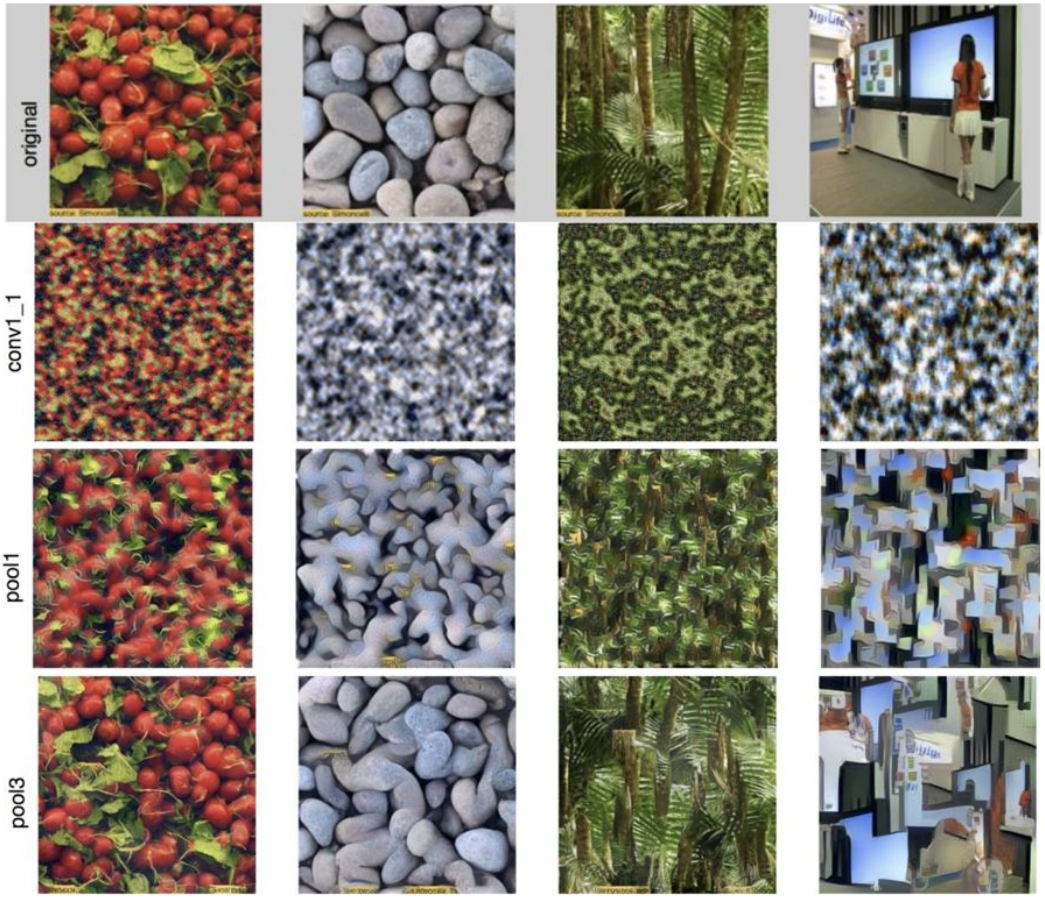
Далее: если мы возьмем два этих вектора (красный и синий) и посчитаем внешнее произведение между ними, то мы получим матрицу (grid), которая показывает со-встречаемость паттернов. Ведь по измерению C у нас каждая карта признаков отвечает за какой-то конкретный признак.

Это довольно интересный трюк, который позволяет очень быстро посчитать приближение матрицы ковариаций разных паттернов к друг другу, усредненную по картам признаков.



Теперь мы берем какую-то нейросеть, например, VGG, обученную на ImageNet. И пропускаем через нее наши оригинальные изображения. После этого берем какой-то шум, пропускаем его через ту же сеть и на каждом слое считаем матрицу Грама. Когда мы пропускаем оригинальное изображение, мы получаем 5 матриц Грама (после каждого слоя) и тоже самое с картинкой шума.

Мы хотели бы, чтобы матрицы Грама у этих двух изображений совпадали. И теперь мы оптимизируем не параметры сети, а входное изображение. То есть, мы делаем градиентный спуск на картинке - изменяем именно пиксели на картинке во время обратного прохода. Соответственно, минимизируя эту дистанцию (по MSE) между матрицами Грама, после 500 итераций мы получаем достаточно неплохие изображения. Примерно так работают современные методы.

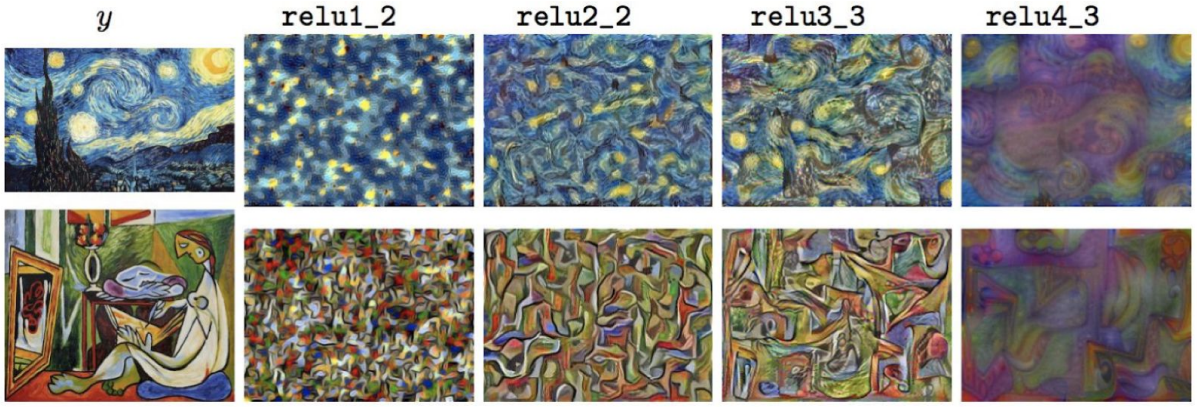


На этой картинке мы можем увидеть, что меняется в зависимости от того, с какого слоя мы берем матрицы Грама. Сразу бросается в глаза: чем больше матриц Грама мы будем брать, тем будут получаться бóльшие паттерны. А на самом нижнем уровне VGG разучиваются самые примитивные признаки - например, изменения цвета, угол, линии. И если мы будем брать матрицу Грама там, мы будем разучивать очень маленькие детали. Когда мы делаем много много карт признаков, мы увеличиваем размер окна, по которому мы агрегируем информацию в сетке.

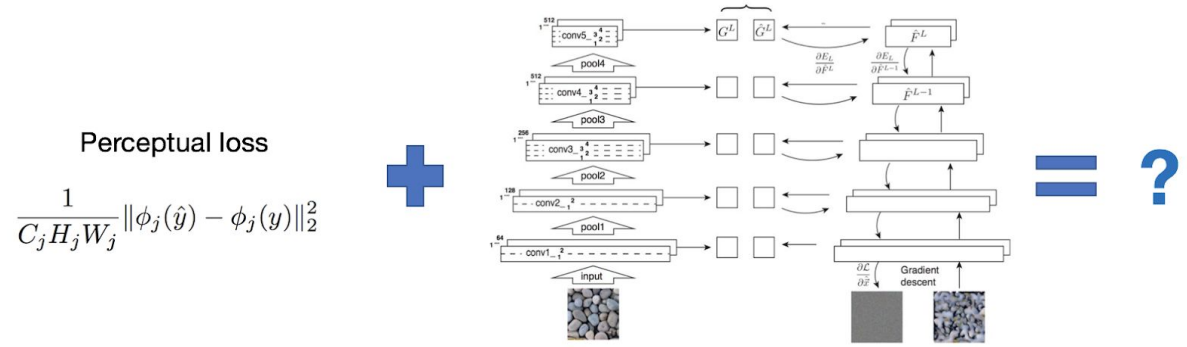
Когда мы считаем матрицу Грама на нижних слоях - у вас очень маленькое окно (например 3x3). Вы смотрите только на него и пытаетесь минимизировать такие дистанции. Но чем выше мы поднимаемся, тем больше у нас окно, и мы начинаем смотреть на объекты текстуры целиком (например, камень). Чтобы получить совпадающее матрицы Грама на этом уровне, нужно действительно хорошо синтезировать текстуру. Чем выше мы поднимаемся в сети и берем матрицы Грама, тем лучше получаются изображения.

Если мы дадим маленький вес матрицам Грама с нижних уровней, то будет мало мелких деталей, и все будет немного размыто. А если дадим больший вес - все будет четче.

Если мы будем синтезировать текстуры от произведений искусств, это будет выглядеть так:



Сходство получается не очень большое, так как картина - это не текстура. Однако, если брать очень много слоев, можно получить что-то осмысленное.



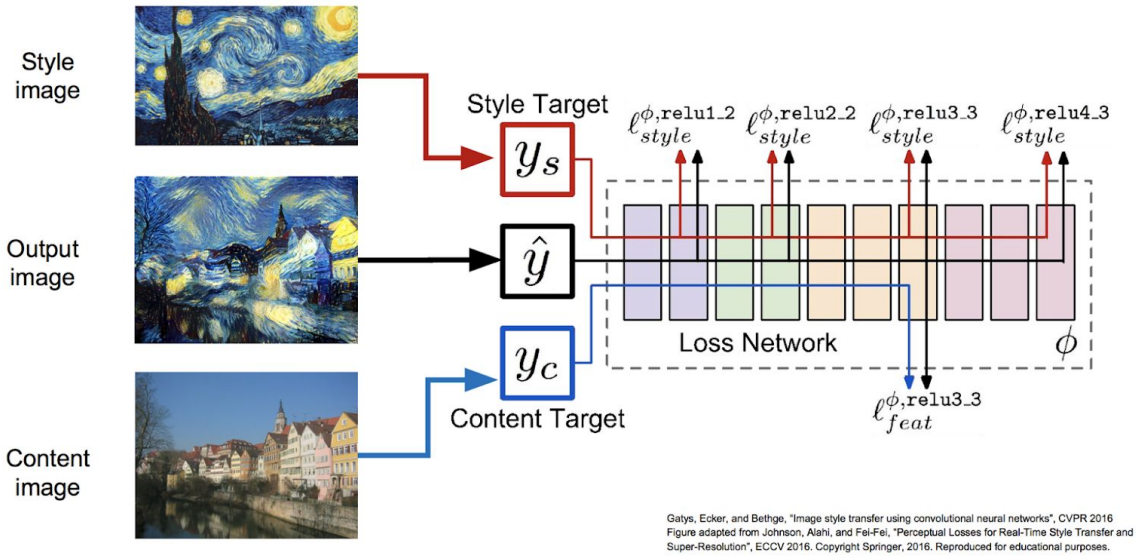
Что будет, если мы объединим Perceptual loss и texture synthesis?

Style Transfer

Известное приложение Prisma - это как раз Style transfer.

Здесь мы объединяем две идеи: синтез текстур, чтобы повторить паттерны, которыми рисовали известные художники, и Perceptual loss, чтобы сохранять исходные изображения.

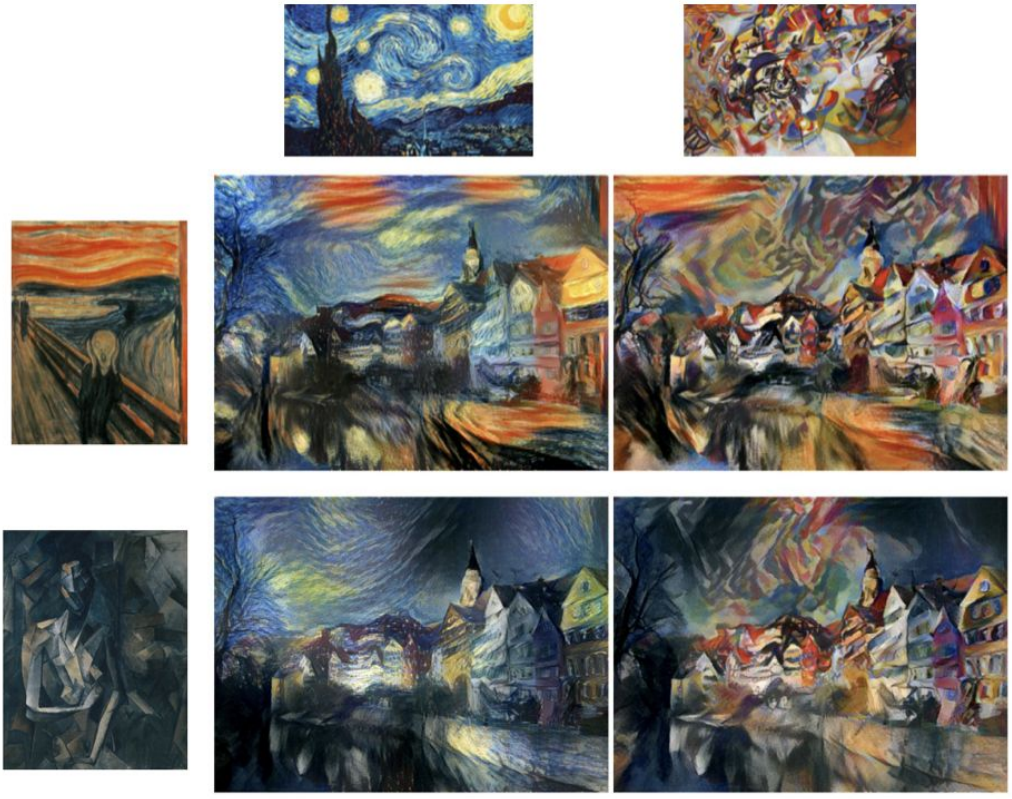
Мы минимизируем взвешенное между этими функциями потерь - дистанцию между матрицами Грама на разных слоях сети и Perceptual loss.



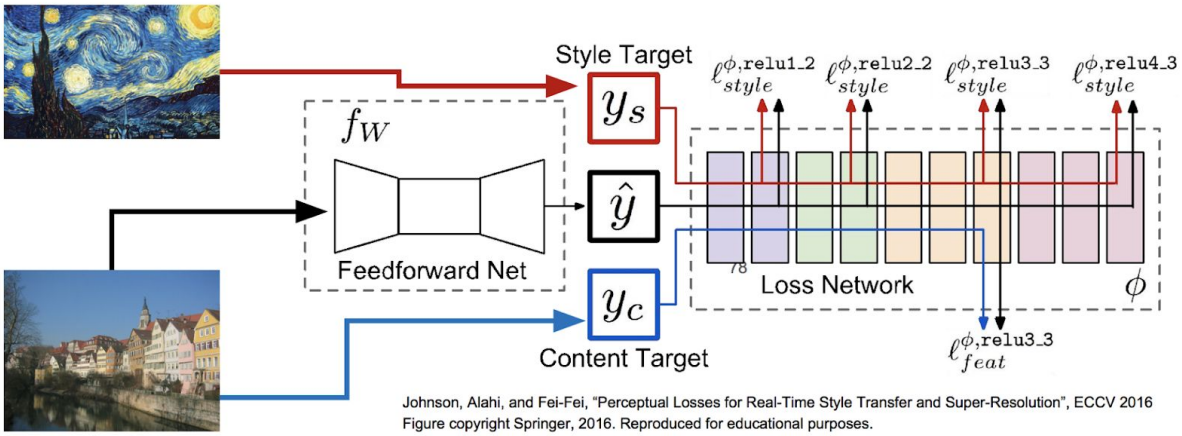
Выглядит это так:

Здесь оптимизируется входная картинка. Кроме этого, мы можем дать больший вес одной или другой функции потерь и таким образом получить либо более хорошую реконструкцию, либо что-то более абстрактное, но более похожее на художника.

Более того, стили можно как-то смешивать. Например, подавать на вход не одно изображение художника, а два разных, и оптимизировать не две функции потерь, а три.



Однако есть проблема. В оригинальном алгоритме нам нужно было пятьсот раз изменять изображение. Это долго. А в приложении Prisma это работает моментально. Как мы будем избавляться от пятисот итераций? Можем заменить процесс оптимизации на одну feed-forward сеть, которая сразу будет понимать что ей делать с изображением:

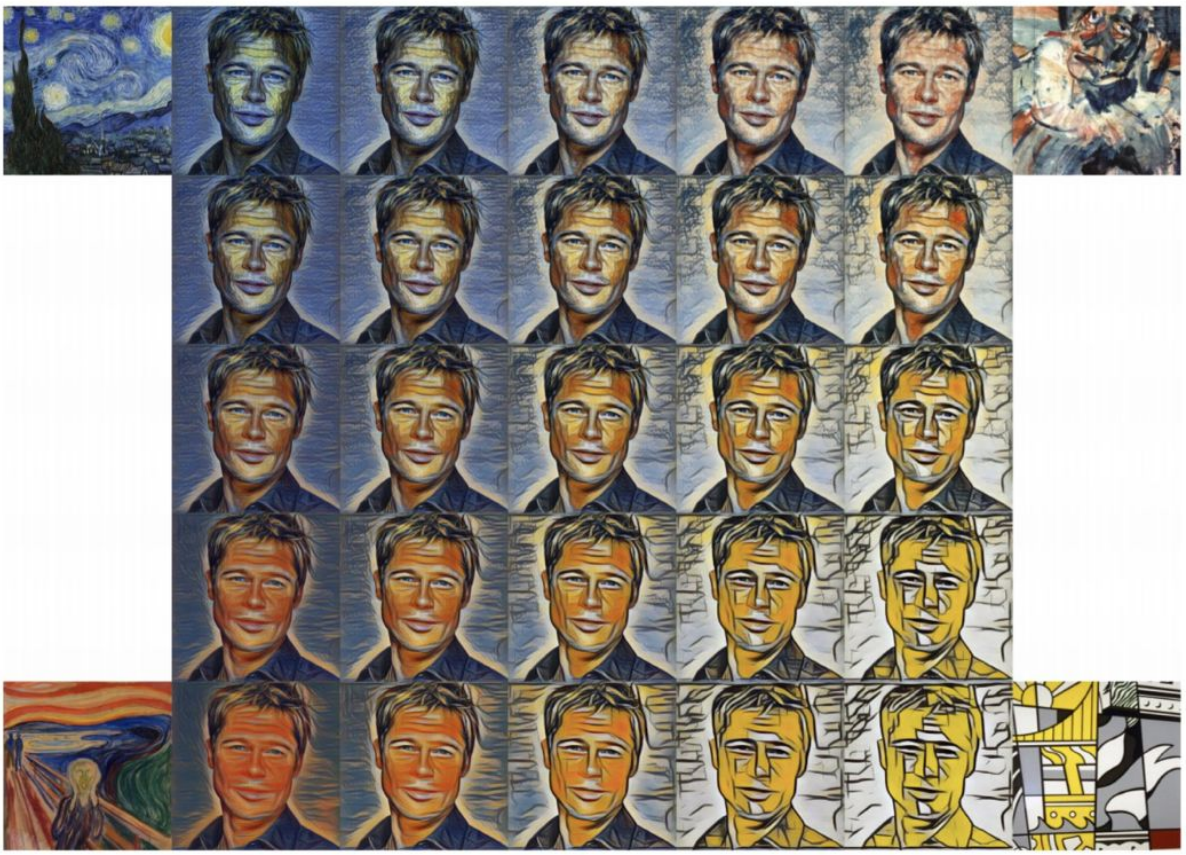


Johnson, Alahi, and Fei-Fei, "Perceptual Losses for Real-Time Style Transfer and Super-Resolution", ECCV 2016
Figure copyright Springer, 2016. Reproduced for educational purposes.

Теперь вместо шума мы пропускаем наше входное изображение через сеть. Когда она научится генерировать стилизованное изображение, - нужно будет всего один раз пропустить через нее изображение и получить красивую картинку.

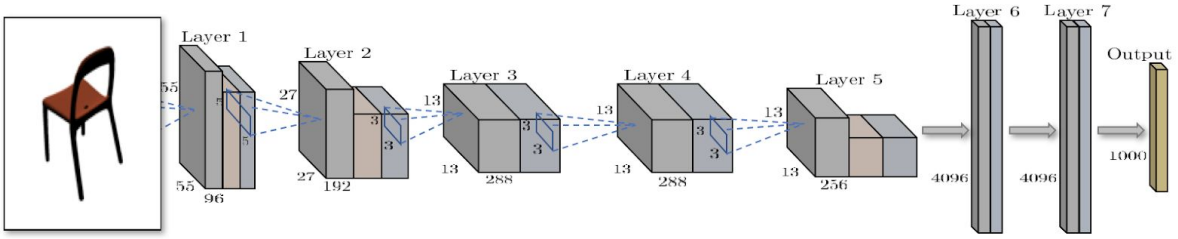
feedforward optimization

Можно делать также совмещение стилей, и получится неплохо:

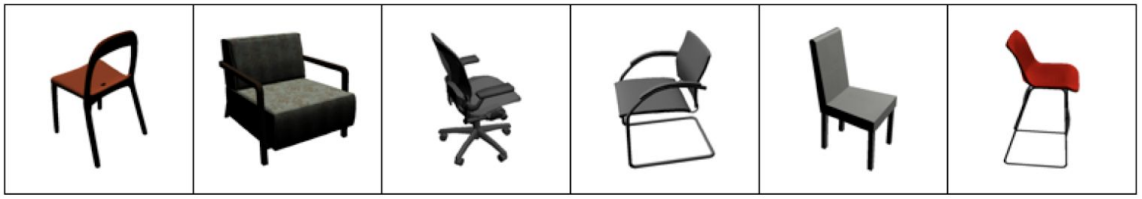


Conditional Generation

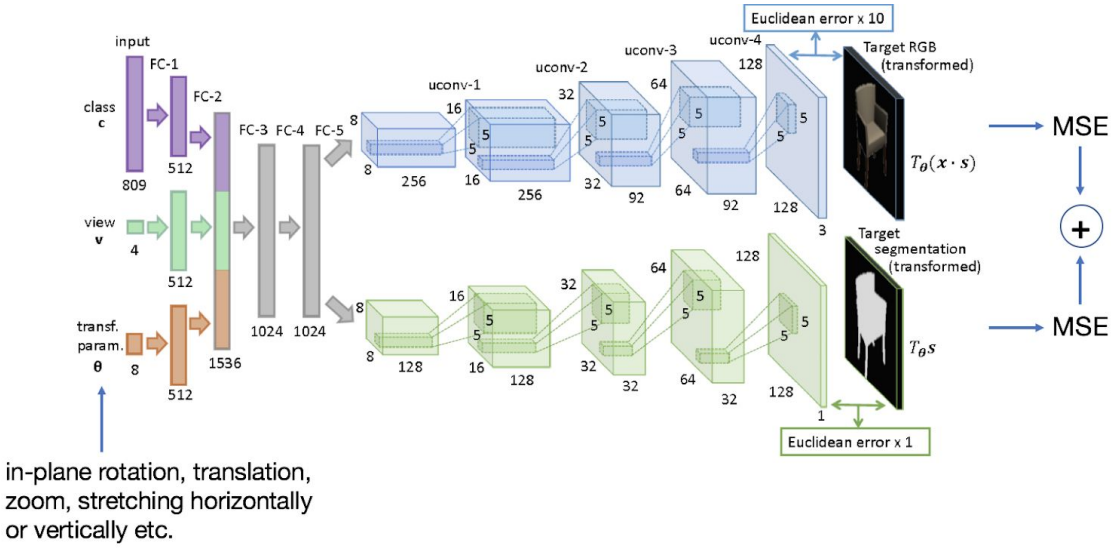
Классическая задача: у вас есть изображение стула, и вы хотите его классифицировать (например, как в каталоге IKEA).
А можно пойти в обратную сторону: из лейблов генерировать стулья и делать их максимально похожими на те стулья, которые должны были быть с этим лейблом.



Кроме лейблов, у нас есть много параметров, связанных с тем, с какой стороны мы смотрим на стул. Тут таких параметров восемь: угол, поворот и так далее.



А как получить такие параметры, если никто их не размечает? Можно научиться на синтетических данных. Например взять Unity и сгенерировать там стулья с заданными параметрами. Так мы получим очень много данных помимо сырого лейбла.



Это обычная нейросеть, которая принимает на вход вектор взгляда на стул (view) размерности четыре, параметры трансформации (transf. param.) размера восемь и класс (здесь вектор класса размерности 809, потому что применялся one-hot-encoding). Далее все пропускается через Fully-connected слои и transpose convolution (но здесь используется просто unpooling).

В итоге мы повышаем размерность до 128x128x3 и подгоняем выход нейросети под изображение стула, который нас интересует. Еще есть другая часть сети, которая предсказывает маску стула, это тоже помогает. В обоих случаях минимизируется MSE: вы их взвешенно складываете и минимизируете.

Можно смешивать модели стульев. Здесь слева одна модель стула, справа другая, а посередине - смешанная. Это может облегчить жизнь дизайнерам IKEA.

PixelCNN и PixelRNN

Рассмотрим еще один класс несложных генеративных моделей.

$$p(x) = \prod_{i=1}^{n} p(x_i | x_1, \ldots, x_{i-1})$$

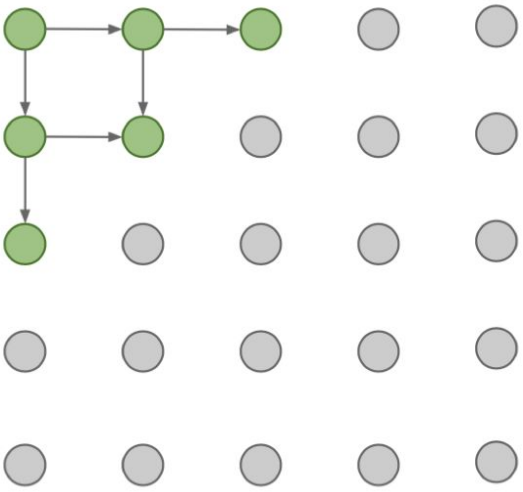
Likelihood of image x

Probability of i'th pixel value given all previous pixels

Давайте попробуем выразить, например, правдоподобие ваших данных как какое-то сложное распределение. Это сложное распределение означает то, что у вас i-тый пиксель зависит от каких-то предыдущих пикселей. От каких - мы определяем сами, потому что мы задаем распределение. Есть какое-то семейство распределений, в котором каждое конкретное распределение обладает собственным порядком. Если мы говорим, что i-ый пиксель зависит только от одного пикселя левее - это один порядок, а от десяти пикселей левее

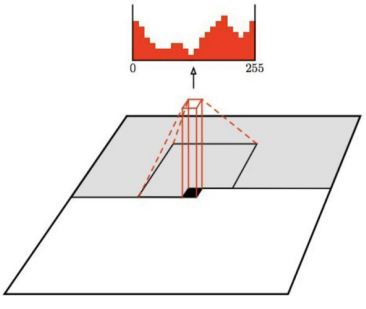
- другой. Дальше нужно просто максимизировать правдоподобие наших данных.

А как мы будем моделировать распределение этих пикселей?
Мы принимаем на вход предыдущие пиксели, которые стоят в картинке раньше, и хотим выдать распределение вероятности по значениям от 0 до 255 для некоторого пикселя, который мы еще не сгенерировали.



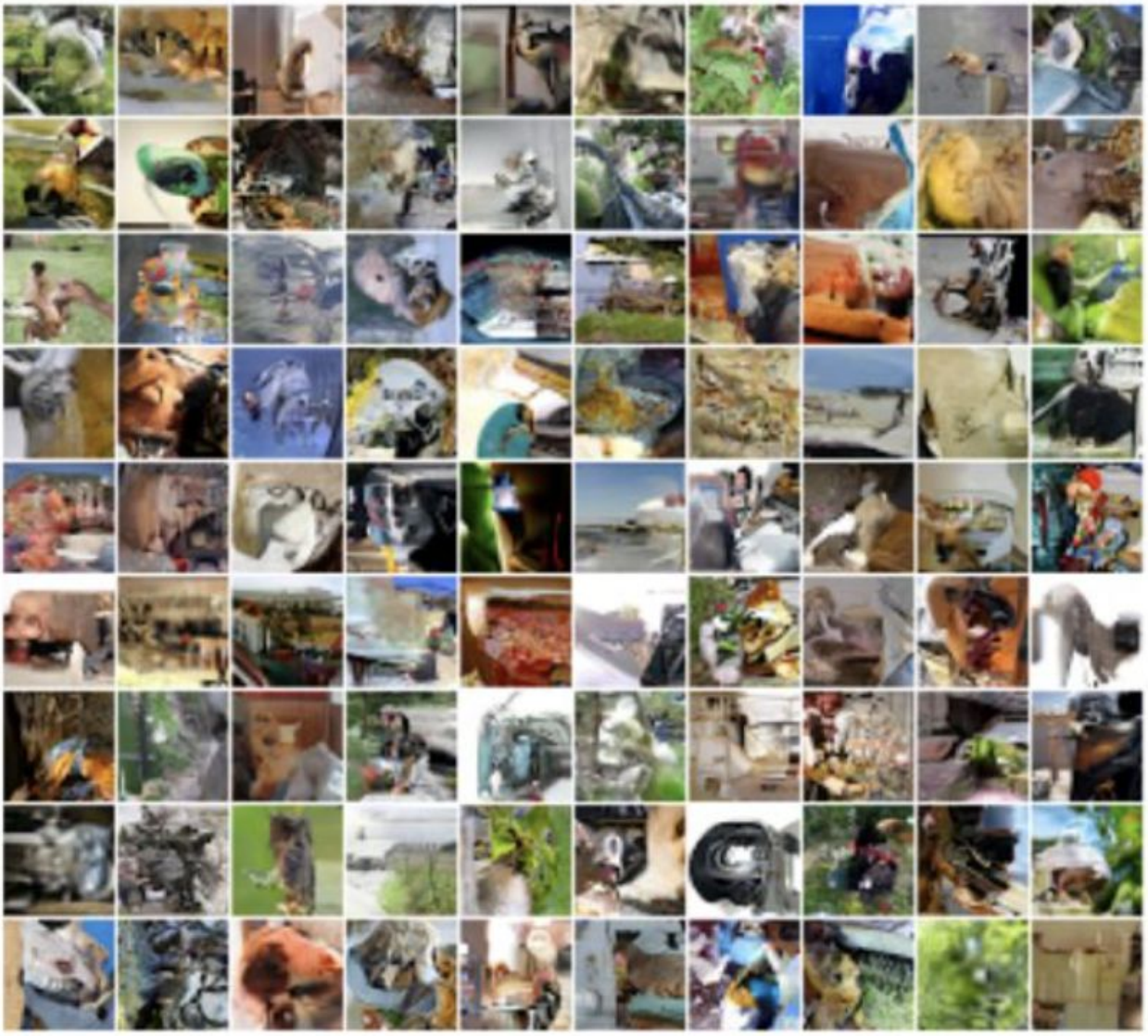
Это просто нейронная сеть, которая будет принимать на вход значения пикселей вокруг (порядок заложен в нее) и выдавать значение, которое мы хотим. Обучаем следующим образом: сеть смотрит на предыдущие пиксели и выдает значение нового, но мы хотим, чтобы оно было именно таким, как в реальных данных. То есть, это максимизация правдоподобия данных. Обычно используют окрестность слева и выше (10 пикселей, например). Идея в том, что порядок должен соответствовать тому, как мы генерируем изображение.
Результат работы получается не слишком хорошим, но на самом деле этот подход нашел отличное применение в несколько иной задаче.

Основная проблема: это очень долго обучается, потому что мы проходим по всей картинке постепенно. Поэтому можно попробовать взять CNN и смотреть на соседей фильтром.

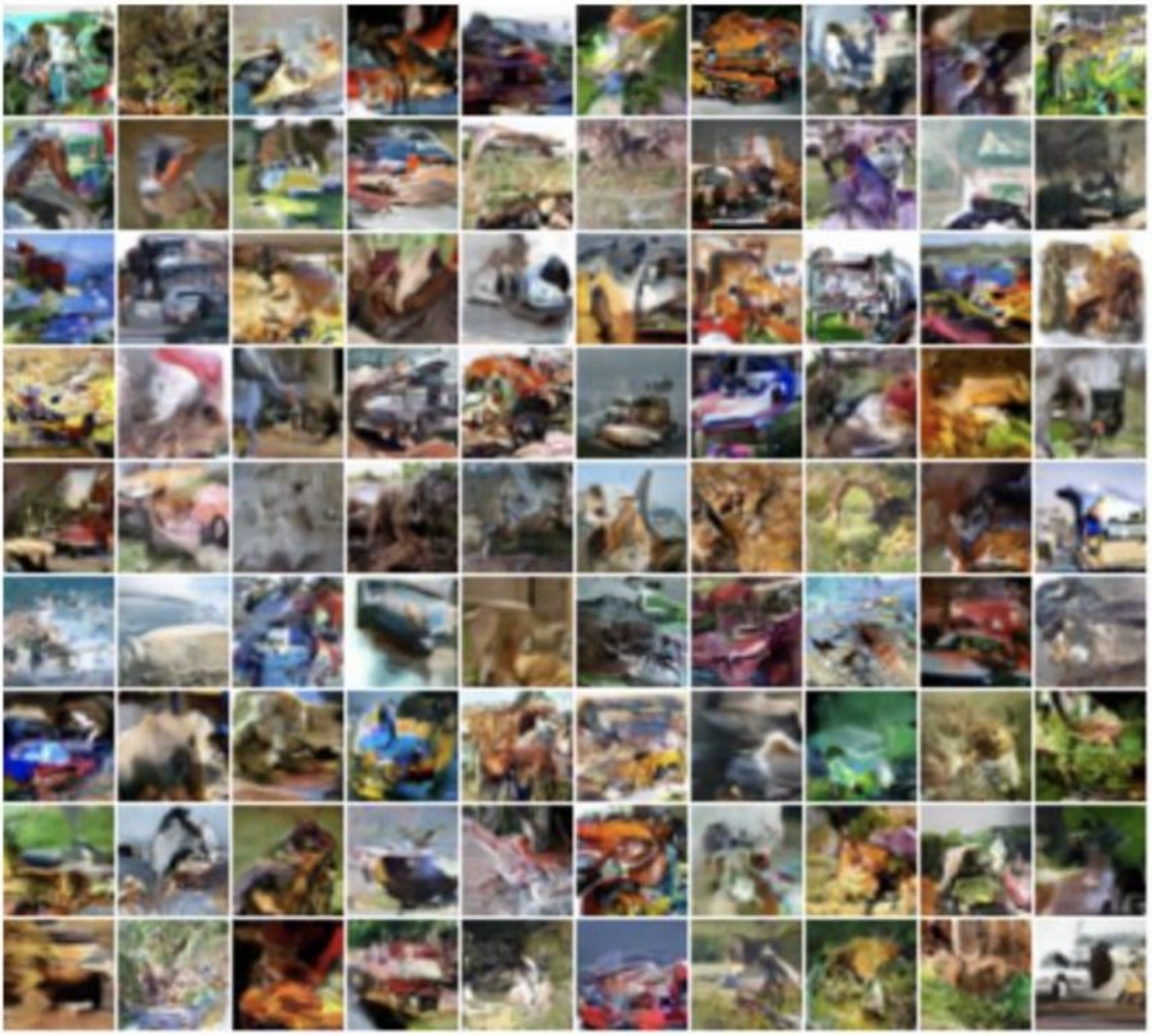


Это тренируется быстро, потому что мы можем распараллелить процесс (в отличие от RNN, где важен порядок), но генерирует все еще медленно (по одному пикселю).

Посмотрим на результаты:



32x32 ImageNet



32x32 CIFAR-10

В плане генерации эта модель оказалась не очень удачной. Однако ее начали использовать для двух важных вещей: во-первых, эти модели неплохо дорисовывают изображения (10-20 пикселей, к примеру). Во-вторых, их хорошо использовать для энкодинга изображений: мы берем все наши пиксели и учитываем все предыдущие значения. Это позволяет нам закодировать некую структуру того, что было вокруг каждого пикселя в его же значении. И это хорошо работает для некоторого класса задач.

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

Список литературы

Видео

Лекция базируется на видео лекциях Stanford University School of Engineering (Fei-Fei Li, Justin Johnson, Serena Yeung). Часть слайдов взята из презентации. Оригинальные лекции можно посмотреть здесь и здесь, а здесь и здесь скачать к ним слайды.