NATIONAL RESEARCH UNIVERSITY
HIGHER SCHOOL OF ECONOMICS
FACULTY OF SOCIAL SCIENCE

MEGA PAPER

PZDataAnalysis

Political and Physical Science

Student
Anastasia Uspenskaya
Arina Puchkova
Sergey Zakharov
Igor Kolesnikov

Spasibo chto est'
Evgeniy Sokolov

Moscow, 2021

Содержание

1 Опишите две компоненты, из которых состоит perceptual loss (стилевую и контентную). Как можно использовать этот функционал, чтобы делать перенос стиля?

Для контекста:

Мы хотим понять есть ли содержательные отличия между картинками, а не просто ли у нас одинаковые пиксели. Именно поэтому мы не используем Евклидово расстояние, например.

Идея:

используем предобученную сетку VGG для определения различий между изображениями.

1.1 Content

$$L^l_{content}(A, B) = \sum_{i,j}(A^l_{i,j} - B^l_{i,j})^2 \tag{1}$$

В 1 следующие обозначения:

- A^ℓ_{ij} — значение i-го фильтра на j-й позиции в слое ℓ для изображения A
- Чем дальше слой, тем меньше он чувствителен к небольшим изменениям в изображении
- Можем попробовать градиентным спуском из картинки с белым шумом найти изображение, минимизирующее $L^\ell_{\text{content}}(A, B)$

Рис. 1: Нотация для первой компоненты Perceptual loss

Считаем для слоя l, он конкретный. Прогоняю картинку A в VGG беру слой нейросети l, там i канал (свертка i) и смотрю ее значение на позиции j

(позиция здесь просто индекс для всех возможных положений фильтра на картинке).

Чем больше слой, тем более Loss будет отражать содержательные различия, а не какие-то непонятные мелкие признаки.

Возьмем значение этого лосса и будем ее минимизировать по B (пиксель), то есть мы будем менять изображение, чтобы добиться наименьшего лосса.

Лучше брать последние, если нам интересно содержимое. Картинка B меняется попиксельно.

1.2 Style

$$L^l_{style}(A, B) = \sum_{i,j}(G^l_{i,j}(A) - G^l_{i,j}(B))^2 \tag{2}$$

Чтобы посчитать G для A, Мы берем два канала i и j, затем мы рассматриваем k-ю позицию фильтра на указанных каналах. Перемножаем выходы с филтьтров и суммируем по всем k.

$$G^l_{i,j}(A) = \sum_k A^l_{i,k} \times A^l_{j,k} \tag{3}$$

Что по сути происходит в 3: эта штука показывает то, насколько перекликаются (насколько похожи) i и j канал с точки зрения выходов фильтра, проецируемеого на картинку A.

Обобщение всего лосса стилевого: он требует, чтобы корреляции каналов между собой (в нашем случае i,j) совпадали для картинки A и картинки B.

Затем предлагается взять таким образом полученные лосс для отдельных слоев l и просуммировать их с определенными весами. (В этом отличие

от контентного лосса, там мы считали для одного слоя, а тут мы складываем по всем слоям).

$$L_{style}(A, B) = \sum_{l=0}^{L} w_l L^l_{style}(A, B) \tag{4}$$

Как это понимать:

Допустим, что фильтр 1 детектирует человека (то есть там, тем больше выход, тем больше изображение похоже на человеческое), а фильтр 2 - синий цвет детектирует. Если стилевая картинка в синей гамме, то тогда корреляция между фильтрами будет высокая, поскольку тогда скалярное произведение будет большим.

Итоговый лосс будет выглядить следующим образом:

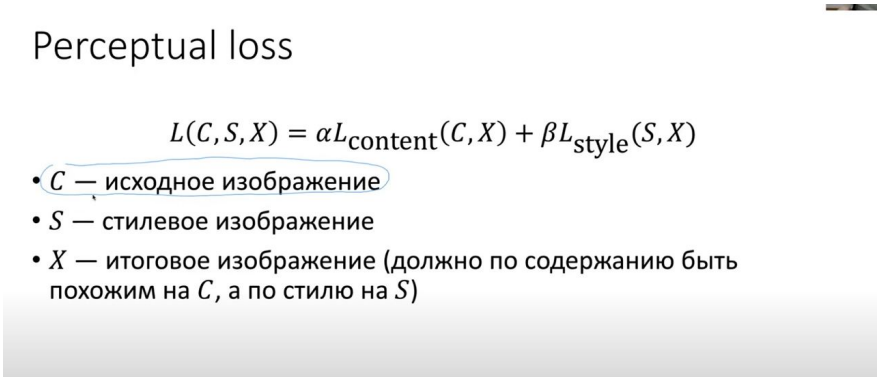


Рис. 2: Полный лосс

Нам дается "исходное"изображение, то есть которое мы пытаемся переделать (до этого это был белый шум). Хотим сделать похожим на S. X-результат. Как и с B, мы постоянно обновляем X для уменьшения лосса. a и β - гиперпараметры модели. Минимизируем по X.

2 Как можно сделать перенос стиля более быстрым, если стилевое изображение известно и не будет меняться?

В чем вообще проблема: минимизация Perceptual Loss напрямую для каких-то адекватных размеров изображений занимает большое время (порядок времени — 4 минуты для изображения 1024x1024). Это с точки зрения пользователя плохая история.

В том случае, если стилевое изображение зафиксировано, мы можем не просто напрямую минимизировать Perceptual Loss, а обучить нейросеть на выборке таким образом, чтобы она принимала на вход изображение x, а ее выход минимизировал Perceptual Loss.

Изначально было: $L(x, S, \hat{y}) \to \min\limits_{\hat{y}}$

Теперь стало: $L(x, S, a_\theta(x)) \to \min\limits_{\theta}$

Здесь x — входное изображение (пытаемся приблизиться к нему по контенту), S — стилевое изображение (пытаемся приблизиться к нему по стилю), которое мы и фиксируем, a_θ — модель с параметрами θ, принимающая на вход входное изображение и выдающее стилизованное.

То есть фактически мы подбираем модель, которая на выходе бы давала картинку, хорошую с точки зрения Perceptual Loss. Вместо прямой оптимизации мы пытаемся обучить какое-то преобразование.

Подобный подход не позволит нам выиграть времени на обучении (все равно придется обучать сетку, а это требует время), но зато затем, с обученной сеткой, создание картинки с нужным стилем будет значительно быстрее.

Однако, если мы поменяем стилевое изображение, то сетку придется обучать новую (что в целом логично).

3 Что в вариационном автокодировщике выдают кодировщик и декодировщик?

Общая идея:

Обычный автокодировщик кодировал нам как-то картинку в вектора. Этот вектор являлся "характеристиками"этого изображения.

Хотим чтобы характеристики были не просто какими-то числовыми значениями, но вероятностными распределениями. Этим мы делаем две вещи: включаем в модель неопределенность (пример: плачущий мальчик и Мона Лиза: насчет первого мы точно знаем, что он не улыбается, его распределение будет сдвинуто, а само распределение прижато к среднему, в тоже самое время насчет Мона Лизы мы до конца не понимаем, улыбается она или нет, и раздвигая границы распределения "улыбки"мы этот момент учитываем).

Задача:

Хотим построить векторное представление картинок $\mathbb{R}^d$, но каждая отдельная картинка соответствует не отдельному числу, но распределению определенному в этом векторном пространстве. Следствие: картинка будет описываться не отдельным числом, но некоторой средней и дисперсией нормального распределения. Получившиееся представление есть набор из d распределений со своими средними и дисперсиями.

Что мы делаем:

Сэмплируем вектор Z из распределения для картинки x, построеного для конкретной картинки. После этого мой декодировщик берет z и превращает его в $\bar{x}$. Добиваемся того, чтобы раскодированная картинка была, как можно ближе к x.

3.1 Кодировщик

Кодировщик представляет картинку в виде набора распределений, которые описываются средним и дисперсией (во время обучения используется логарифм дисперсии).

3.2 Декодировщик

Из построенного распределения сэмплируем значения и декодировщик выдает нам картинку, которая похожа на оригинальную.

4 Как устроен функционал ошибки в вариационном автокодировщике? За что отвечает каждое из слагаемых?

$$\sum_{i=1}^{l}(\mathbb{E}_{q(z|x_i)}\log p(x_i|z) - \mathrm{KL}(q(z|x)||\mathcal{N}(0,1))) \to max$$

$q(z|x)$: q – это кодировщик (полносвязная или сверточная НС), на вход принимает x (картинку), и выдает z, то есть вектор средних и дисперсий. То есть он выдает распределение представлений при условии, что на вход пришла такая-то картинка.

$p(z|x)$: p – это декодировщик. Он принимает на вход скрытый вектор и раскодирует картинку. По сути также представляет собой НС. Декодер выдает нормальное распределение с центром в том, что он раскодировал, плюс ε, то есть некий случайный шум.

$\mathbb{E}_{q(z|x_i)}\log p(x_i|z)$: взяли распределение на скрытых векторах ('облачко'), из него нагенерили кучу точек. Из каждой этой точки развернули новую картинку ($\tilde{x}_1, \tilde{x}_2, \tilde{x}_3$ и т.д.). Далее оценили, насколько каждая из этих раскодированных картинок похожа на оригинальную. По сути мы тут считаем среднюю ошибку рекострукции. Это слагаемое функционала отвечает за то, что мы хотим, чтобы картинки из распределения были похожи на оригинал (оно называется reconstruction likelihood).

KL – это дивергенция Кульбака-Лейблера, мера сходства двух распределений (если KL = 0, то распределения одинаковые). Тут мы просим, чтобы распределение, которым кодируется картинка, было как можно больше

похоже на стандартное нормальное. Зачем нам это надо? А затем, что кодировщик очень хочет сделать нам вырожденное распределение, и представить его как точку (то есть сделать распределение с очень узким пиком). Это ведет к переобучению – тогда декодер заучит, во что нужно развернуть именно эту точку, и картинки не особо будут отличаться между собой. Тогда KL – это что-то типа регуляризатора в нашем функционале. KL позволяет нам расшакалить распределение, на выходе мы получим что-то с центром в 0, но с дисперсией пошире. Если взять только reconstruction likelihood – получим пустоты в скрытом векторономпространстве, все будет описано почти что точками, не понятно будет, во что развернуть какую-то случайно выбранную точку между ними. Если взять только KL – всякие разные картинки перемешаются между собой, так как они все будут описываться стандартным нормальным распределением. Если вместе – разные картинки будут друг от друга отстоять, но их распределения будут более размазаны.

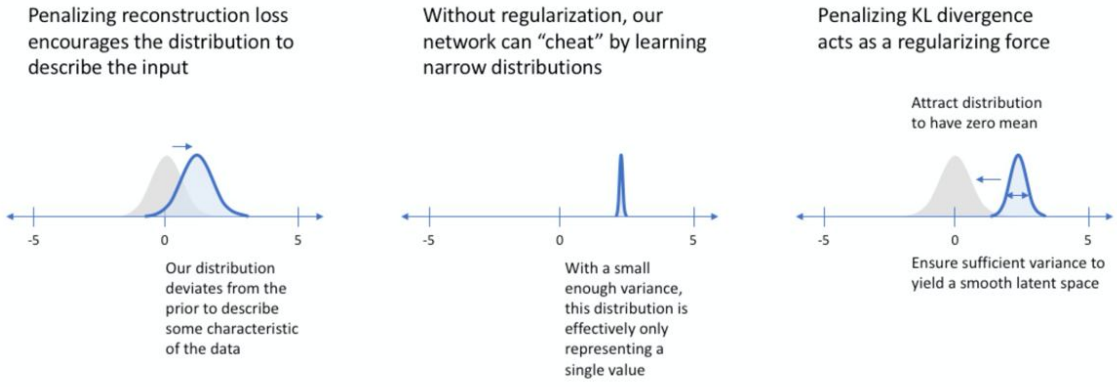


Рис. 3: Иллюстрация, что делает KL

5 В чём состоит идея трюка репараметризации (reparametrization trick)?

Кодировщик выдает нам μ и σ и мы по ним генерируем какой-то вектор z из этого распределения. Проблема: непонятно, как посчитать производную производную выхода по μ и σ, поскольку там происходит вероятностный процесс. Вместо этого сделаем так:

Кодировщик выдает нам μ и σ (это конкретные детерминированные числа), а еще мы будем генерировать стандартное нормальное число ε. И чтобы получить конкретное представление z мы сделаем $z = \mu + \sigma \odot \varepsilon$ (здесь $\odot$ — поэлементное умножение). То есть фактически мы разделяем случайность (запихиваем ее в ε), при этом делая ее без параметров (и никакие градиенты туда не нужны), и выдачу параметров кодировщиком (см. рисунок 4 в качестве иллюстрации).

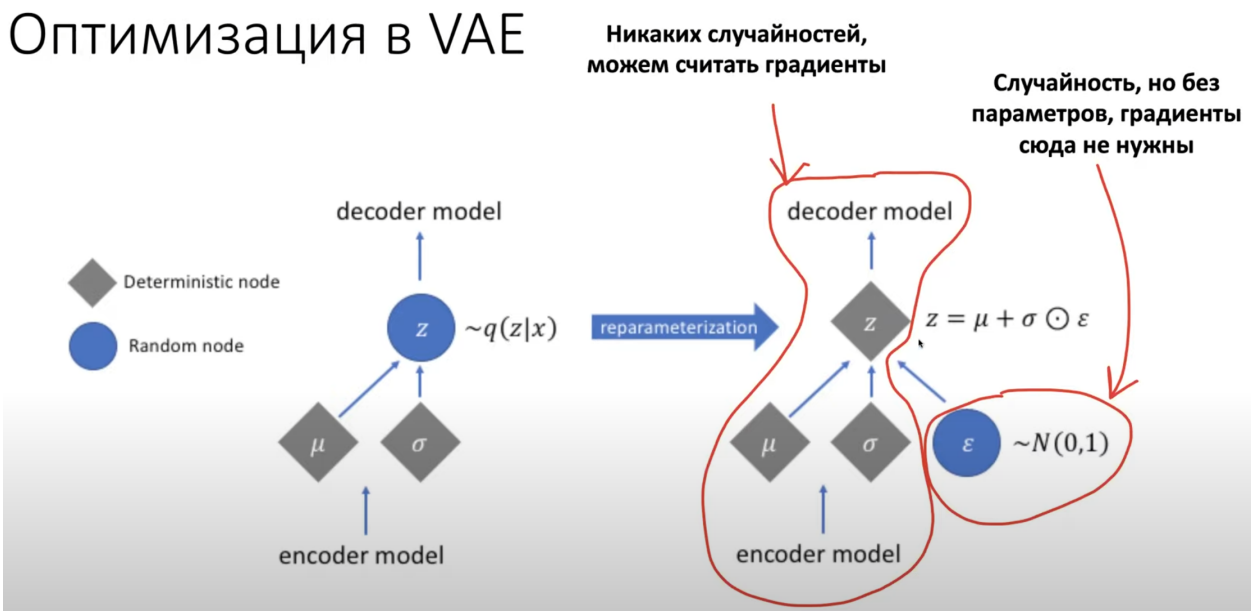


Рис. 4: Трюк репараметризации (слева без него, справа с ним)

6 Запишите функцию потерь GAN. Как в ней обозначается дискриминатор и генератор? Как она используется при обучении (ищем минимум, максимум или как-то еще)?

$$L(D,G) = -\frac{1}{n}\sum_{x_i \in X} logD(x_i) - \frac{1}{n}\sum_{z_i \in Z} log(1 - D(G(z_i))) \to \min_G \max_D L(D,G)$$

Здесь D – дискриминатор, G – генератор. В этой функции потерь мы отдельно считаем ошибку на реальных объектах (x_i) и сгенерированных объектах ($G(z_i)$). Сначала мы хотим минимизировать этот функционал на дискриминаторе, хотим научить его лучше разделять объекты. Потом мы максимизируем функционал на параметрах генератора, то есть мы хотим сдвинуть ближе реальные и сгенерированные объекты.

При обучении:

- Генерируем n объектов из случайного шума
- Выбираем случайно n реальных объектов
- Берем случайный шум, пропускаем через генератор, передаем это в дискриминатор. Получаем вероятность того, что наши сгенеррированные объекты принадлежат к реальным объектам (и то же самое делаем для самих реальных объектов).
- Используем функцию потерь, чтобы обновить параметры дискриминатора. И так k раз

- Делаем одну итерацию обучения генератора. Еще раз генерим случайный шум и отдаем его дискриминатору, еще раз считаем функцию потерь (при ГС по параметрам генератора нам нужно только второе слагаемое функции потерь, то есть будем смотреть только на то, как хорошо получилось разделить сгенерированные объекты).

Сначала обучаем дискриминатор, он дает нам некую меру сходства (насколько два класса (реальные и сгенерированные изображения) похожи друг на друга). Потом исользуем эту меру сходства, чтобы обучить генератор, чтобы он сдвигался ближе к реальным изображениям.

7 Что такое затухание градиентов в GAN? Покажите это математически.

Затухание градиентов — одна из проблем GAN-ов. Предположим, что мы имеем ситуацию, похожую на изображенную на рисунке 5. В таком случае, как видно, дискриминатор может провести линию так, чтобы идеально (или в целом почти идеально, сути дела не меняет) разделить два класса. Это здорово, но не очень.

Если классы идеально разделились дискриминатором, то $D(x_i) = 1$ для любого x_i в X, т.е. для всех объектов первого класса он вернул 1. Для всех же сгенерированных объектов $D(\hat{x}_j) = D(G(z_j)) = 0$ для любого z_j в Z.

Вспомним, как выглядит функция потерь:

$$L(D, G) = -\frac{1}{n} \sum_{x_i \in X} \log D(x_i) - \frac{1}{n} \sum_{z_j \in Z} \log[1 - D(G(z_j))] \tag{5}$$

Подставим получившиеся значения в функцию потерь 5:

$$L(D, G) = -\frac{1}{n} \sum_{x_i \in X} \log 1 - \frac{1}{n} \sum_{z_j \in Z} \log[1 - 0] = 0 = \text{const} \tag{6}$$

Константа в том смысле, что нет зависимости от весов генератора или дискриминатора. Это значит, что градиент функции потерь оказывается равным нулю, т.е. ни дискриминатор, ни генератор более не обучаются, поскольку веса не обновляются. Ну понятно, что картинка эта далека от того, чего мы на самом деле хотим, так что это проблема.

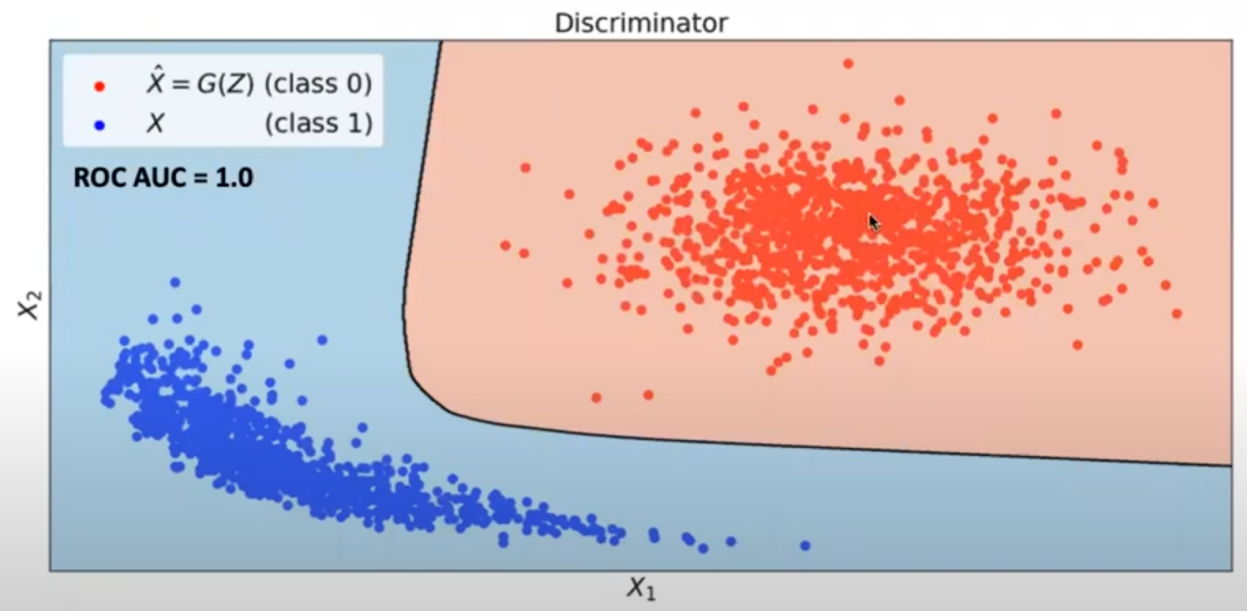


Рис. 5: Ситуация, возникшая на некоторой итерации обучения GAN и ведущая к затуханию градиентов

8 Запишите теорему о замене переменных для функции плотности распределения случайной величины. Что значит каждое обозначение в ней?

Теорема (о замене переменных): *Пусть даны* $p_z(z)$ *и* $z = f(x)$, *тогда* $p_x(x)$ *находится следующим образом:*

$$p_x(x_i) = p_z(f(x_i)) \left| \det \frac{\partial f(x_i)}{\partial x_i} \right| \tag{7}$$

где указан определитель матрицы первых производных, которая записывается как:

$$\frac{\partial f(x_i)}{\partial x_i} = \begin{pmatrix} \frac{\partial f(x_i)_1}{\partial x_{i1}} & \cdots & \frac{\partial f(x_i)_1}{\partial x_{in}} \\ \vdots & \ddots & \vdots \\ \frac{\partial f(x_i)_m}{\partial x_{i1}} & \cdots & \frac{\partial f(x_i)_m}{\partial x_{in}} \end{pmatrix} \tag{8}$$

Здесь $p_z(z)$ — распределение z, $p_x(x)$ — функция распределения x, $z = f(x)$ — функция, переводящая x в z.

Если говорить на пальцах, то этот модуль определителя в 7 — отношение объема ∂z к новому объему ∂x.

Про матрицу первых производных: т.к. x, $z = f(x)$ — некоторые многомерные векторы, то логично, что у них есть различные компоненты, индексы которых и указываются в матрице первых производных (то есть например левая верхняя компонента это производная первой компоненты $f(x_i)$ по первой компоненте x_i).

9 Запишите функцию преобразования для Real-NVP нормализационного потока. Как посчитать ее Якобиан? Как выглядит обратная функция?

Предпосылка: и x, и z имеют размерность D.

9.1 Функция преобразования

$$z = f(x) = \begin{cases} z_{1:d} = x_{1:d} \\ z_{d+1:D} = x_{d+1:D} \odot \exp[s(x_{1:d})] + t(x_{1:d}) \end{cases} \tag{9}$$

Здесь:

- $z_{1:d}$ — первые d компонент вектора z;
- $s(x_{1:d})$ и $t(x_{1:d})$ — нейронные сети с d входами и $(D-d)$ выходами;
- $\odot$ — поэлементное умножение.

9.2 Якобиан

Матрица первых производных нижнетреугольная и выглядит как-то так:

$$\frac{\partial f(x)}{\partial x} = \begin{pmatrix} \mathbb{I}_d & 0 \\ \dfrac{\partial z_{1:d}}{\partial x_{1:d}} & \operatorname{diag}\{\exp[s(x_{1:d})]\} \end{pmatrix} \tag{10}$$

Но нас интересует на сама матрица, а ее определитель, а он для нижнетреугольной матрицы записывается просто как:

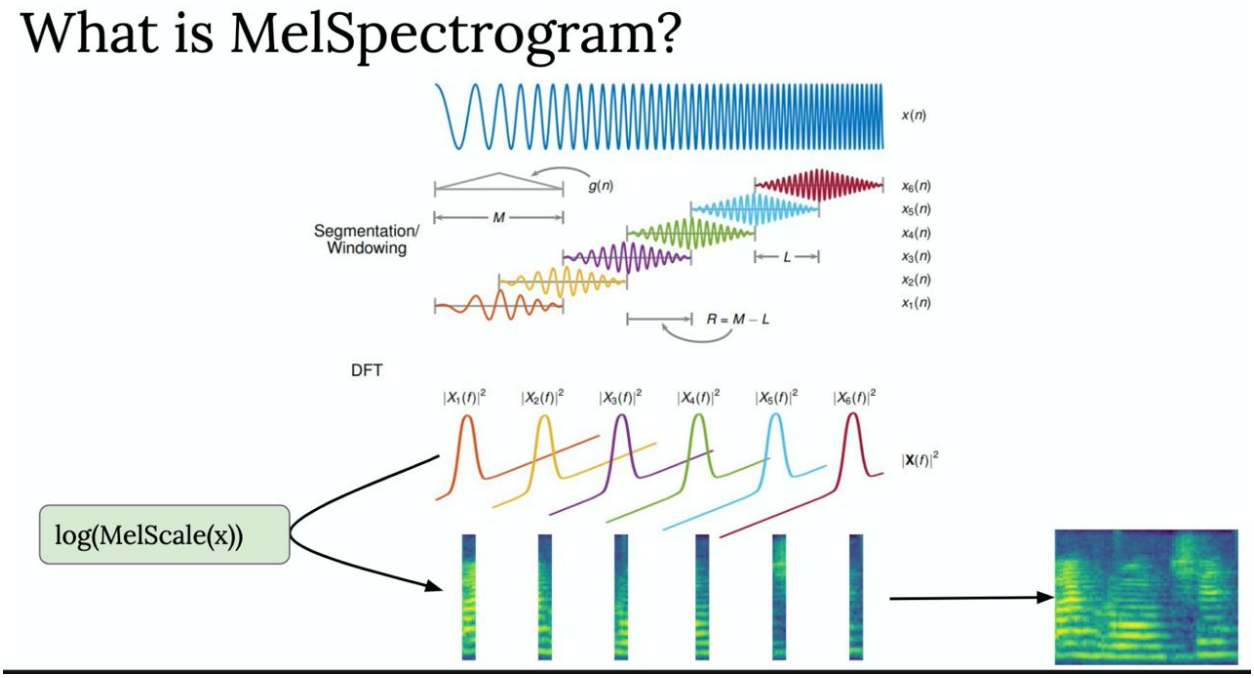
$$\left|\det \frac{\partial f(x)}{\partial x}\right| = \exp\left[\sum_{j=d+1}^{d} s(x_{1:d})\right] \tag{11}$$

9.3 Обратная функция

Тут тривиально:

$$x = f^{-1}(z) = \begin{cases} x_{1:d} = z_{1:d} \\ x_{d+1:D} = [z_{d+1:D} - t(z_{1:d})] \odot \exp[-s(z_{1:d})] \end{cases} \tag{12}$$

10 Что такое мел-спектрограмма? Как она получается из wav-файла?



Зачем нам вообще спектрограммы? Потому что в одном звуке в сыром сигнале содержится 2000-4000 амплитуд, это дорого хранить. Плюс, мы не инвариантны к шумам и трансформациям.

У нас есть исходный сигнал. Мы по нему бежим каким-то окном с каким-то шагом (при этом идем с пересечением – окна могут накладываться друг на друга). К каждому вырезанному окну мы применяем оконную функцию. Зачем она нужна? На следующем шаге мы будем применять преобразование Фурье, а оно работает с сигналом, который периодичен, то есть он начинается и кончается в нуле (поэтому часто такие фукнции имеют форму 'колокольчика', в домашке например мы применяли косинуи-

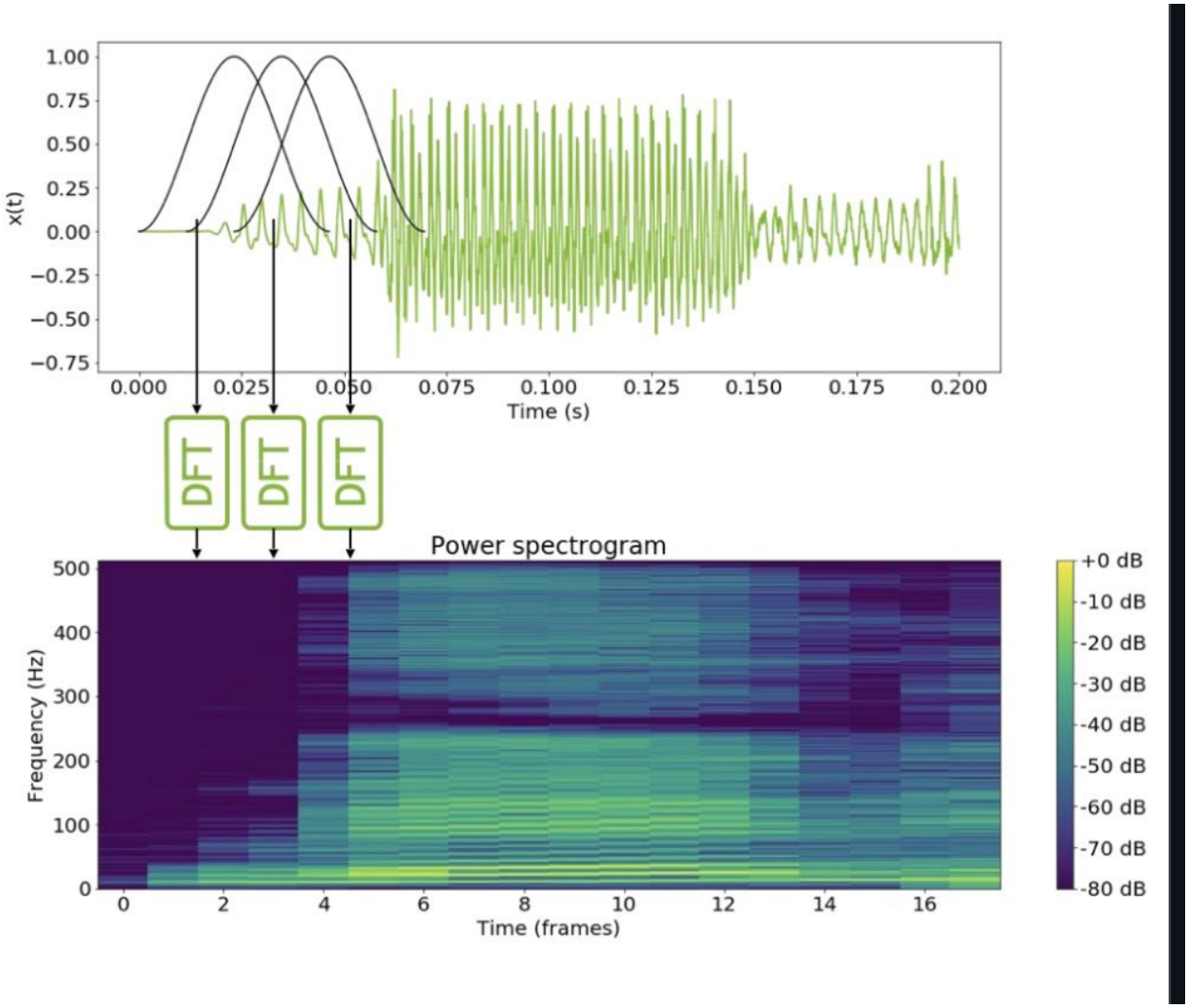
сальный фильтр). По сути мы поэлементно перемножаем значение нашего сигнала на значение оконной функции в данной точке.

Дальше к каждому окошку применяем преобразование Фурье. Суть преобразования Фурье – представляем сырой сигнал как совокупность синусов и косинусов (например, в форме f(t) = 5 + 2sin(2t + 2) - 3cos(0.2t - 1)). У разных косинусов и синусов будут разные 'веса' и константы. В каждый момент времени каждая отдельная (ко-)синусоида вносит разный вклад в формирование сигнала, веса при них меняются в каждый момент времени. Дискретное преобразование Фурье можно представить как умножение матрицы на окно, которое мы получили на прошлом шаге. Элемент этой матрицы $M_{mn} = exp(-2\Pi i\frac{(m-1)(n-1)}{N})$, где N – длина сигнала. На выходе мы получим вектор той же длины, что исходный, но с мнимыми числами (i.e. от размерности (100,) перешли к размерности (100,2), добавилось второе измерение).

Далее считаем квадрат комплексной нормы. То есть поэлементно для каждого комплексного числа считаем корень из (квадрат действительной части + квадрат мнимой части). Снова получаем вектор исходного размера (i.e. (100,)).

Берем первую половину вектора + 1 в силу его симметричности.

На этом этапе мы получили просто спектрограмму.



Один фрейм (столбик) в спектрограмме – один раз мы выполнили всю операцию выше. Чем выше идем по столбику, тем более высокие частоты кодируются. Яркость цвета – насколько в этот момент звучала та или иная частота. Желтый внизу – это были басы, желтый вверху – это был писк.

Как теперь перейти к мелспектрограмме? Для каждого фрейма мы храним все 500 значений частот, хотя часто там наблюдается пустота. Плюс, люди хорошо слышат низкие частоты, а высокие не очень различают. Поэтому мы хотим учесть низкие частоты больше, а высокие – меньше. Для

этого переводим все в мел-шкалу (также реализуемо матричным умножениям). $\mathrm{m} = 2595 \log_{10}(1 + \frac{f}{700})$, где f – значение частоты, а m – уже значение мелчастоты. Потом еще берем логарифм от мелспектрограммы.

Эта шкала лучше отражает, что человеческое ухо слышит. Проблема одна – это преобразование идет не через квадратную матрицу, поэтому восстановить оригинальный сигнал уже не получится.

11 Какие метрики качества ASR вы знаете? Как они вычисляются?

1. Word Error Rate (WER) – доля ошибочных слов.

   $WER = \frac{S+D+I}{N} = \frac{S+D+I}{S+D+C}$, где

   S – кол-во замен, D – удалений, I – вставок, C – совпадений.

   В примере, где текст "Quick brown fox jumped over a lazy dog"

   был распознан как "Quick brow an fox jumped over a lazy dog"

   $WER = \frac{1+1+1}{8} = \frac{3}{8}$. Процент неправильного распознавания слов речи – это только количественный показатель точности распознавания (количество ошибок распознавания на фразу или слово), но не вероятность распознавания (вероятность неправильного распознавания слова во фразе), потому как он не ограничивается интервалом вероятности $[0; 1]$ и не имеет верхнего предела. Например, представим, что диктор произнес фразу, состоящую из 10 слов, но система ее полностью распознала неправильно и предложила гипотезу из 12 других слов. В этом случае, $WER = 120\%$ $(S = 10, I = 2, H = D = 0)$, и это означает, что показатель точности отрицательный (-20%), что не имеет смысла с точки зрения теории вероятностей.

2. CER – то же самое, но посимвольно. Пробелы считаются. Возможна стратификация или придание весов по типу датасета или тематике.

3. В Attend & Spell можно просто посчитать Евклидово расстояние между векторами.

4. В LAS минимизируем кросс-энтропию.

12 Что такое MuLaw-кодирование, для чего оно нужно и как оно вычисляется?

В одном взятом сэмпле обычно находится 16 бит. Это означает, что у нас 2^{16} значений. Нам приходится делать softmax на 2^{16} значений. Очень много получается, хотим сделать поменьше.

Рассматриваемый метод позволяет ужать значения амплитуд, сохранив по максимуму первоначальную информацию.

Интуиция: человеческое ухо слышит в лог шкале: мы хорошо различаем звуки при низких амплитудах и плохо различаем при высоких. Нам надо хранить низкие амплитуды в высоком разрешении, а высокие амплитуды в низком, ибо мы и так их особо не слышим.

Реализация идеи хороша видна на следующем графике 6: здесь показано, как звуки с разными амплитудами преобразуются согласно данному кодированию

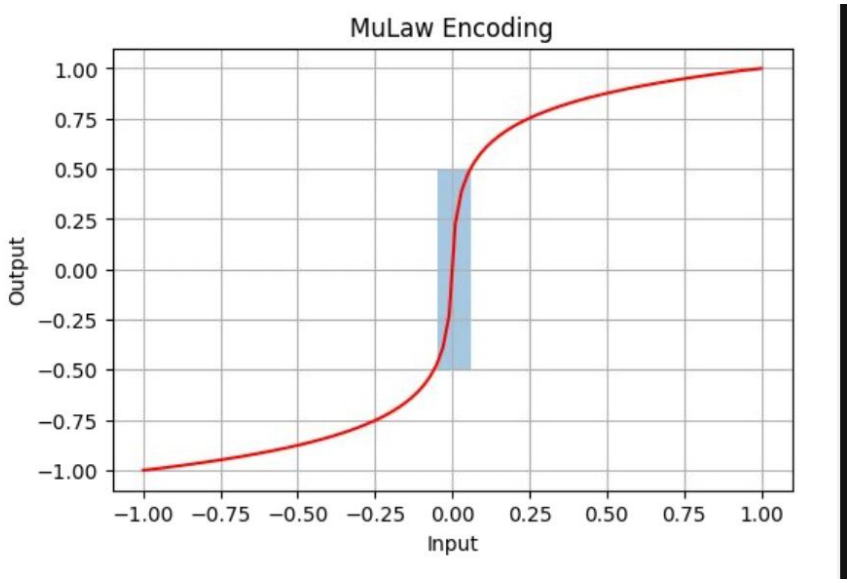


Рис. 6: Визуализация Mu law Encoding

Как можно увидеть, от -0.1 до 0.1 мы очень компактно храним входные частоты, но чем дальше амплитуда, тем больше у нас становится шаг(скорость функции возрастае), и соответственно в тем меньшем разрешении мы храним звуки с указанной амплитудой.

Гиперпараметром считается количество конечных значений, в которые мы хотим перевести все наши амплитуды.

Формула:

$$f(x_t) = \text{sign}(x_t)\frac{\ln(1+\mu|x_t|)}{\ln(1+\mu)} \tag{13}$$

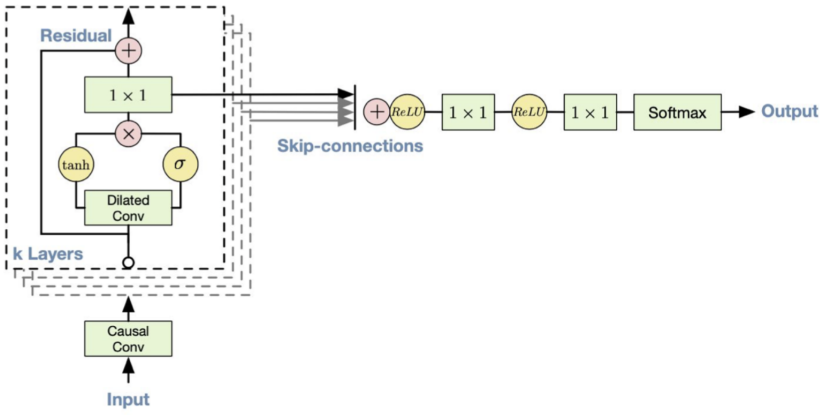
μ - это просто параметр сжатия (обычно он вот такой - 255).

x- нормализованное число, которое подвергается сжатию.

13 Как устроен WaveNet? Почему он медленно работает на этапе применения?

$$p(\mathbf{x}) = \prod_{t=1}^{T} p(x_t \mid x_1, \ldots, x_{t-1})$$

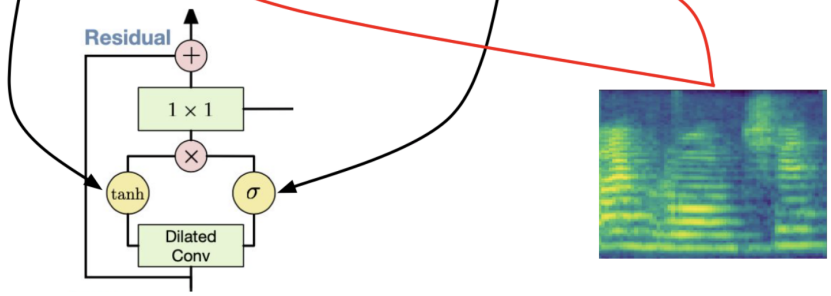
$$p(x_t|x_i, \ldots, x_{t-1}) \sim \mathrm{Cat}(\pi_\theta)$$



Авторегрессионная модель (моделируем условные вероятности по всем предыдущим сэмплам). Вход – амплитуды, в train, например, 100 ground truth сэмплов. Для них у нас, например, 12 мелов. Прогоняем через каузальную свертку мелы, растягивая их. Берем первые 99 сэмплов и загоняем в нашу модель. Для каждого сэмпла предсказываем следующий со сдвигом 1. По такой же схеме в строительном блоке (ниже) обуславливаем сэмплы на мелы. Когда будем считать ошибку, возьмем последние 99 сэмплов. Амплитуды проходят через k одинаковых строительных блоков.

Структура блока:

- $\mathbf{z} = \tanh(W_{f,k} * \mathbf{x}) \odot \sigma(W_{g,k} * \mathbf{x})$
- $\mathbf{z} = \tanh(W_{f,k} * \mathbf{x} + V_{f,k} * \mathbf{y}) \odot \sigma(W_{g,k} * \mathbf{x} + V_{g,k} * \mathbf{y})$



1. Каждый сигнал x (на схеме перед dilated conv) сворачиваем двумя каузальными свертками W_f и W_g. На каждую кладем сигмоиду и поэлементно перемножаем. Наша сигмоида выучивает, на какие куски аудио смотреть, на какие – нет. 1 открывает gate внимания, 0 – закрывает. 1x1 – свертка с ядром kernel size. Яичко. Это не так важно.

   FYI:

   Dilated свертки увеличивают рецептивное поле. Dilation растет экспоненциально – от 1 до 1000 с шагом 2 в каокй-то степени. Рецептивное поле раздувается, что позволяет нам моделировать очень длинные зависимости. Нам это важно, потому что в wav много занимает, но хранит мало информации. Одна секунда – 16 000 сэмплов. Секунда – мало, данных – много.

   Второй момент – каузальные свёртки. Нужны, чтобы моделировать авторегрессию. На стандартной свертке синий – первый уровень, серый – второй уровень. На вход синие данные сворачиваем сверткой размером 3 со страйдом 1. Три шарика свернули в один. На самом

центральном шарике видно, что он сворачивается, при этом берет вклад и слева, и справа – смотрит и в прошлую, и в будущее. Нам это не подходит, так как мы не хотим смотреть в будущее. Нам нужно как-то свертку обрубать. Вклад каждого шарика серого или оранжевого цвета или из настоящего, или из будущего. Как делать? Можно добавлять в саму свёртку паддинг. Левый – правильный, правый – лишний. Вырезаем его. Можно маскировать свертки. С паддингами проще читать код.

2. Предсказываем, исходя из mel-синтезиса. Сворачиваем mel с двумя свертками (в формуле y – mel, V – свертки). Так мы обуславливаем нашу генерацию. Так считаем каждый слой. x и спектрограмма разного размера по времени, и чтобы они совпадали мы должны растянуть спектрограмму до размера сэмпла. Это делается или transpose сверткой, или линейной интерполяцией.

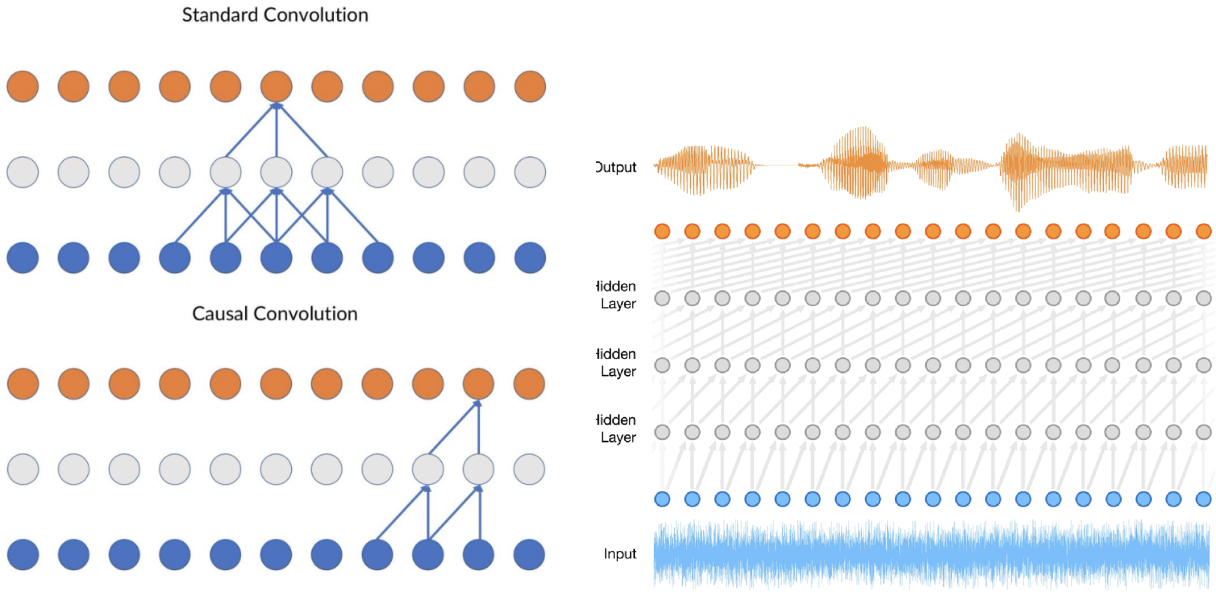
Skip-connections для каждого сэмпла выдает распределение для следующего.

Дальше MuLaw encoding. У каждого представления 16000 сэмплов, и каждый сэмпл – число. Один сэмпл занимает 16 бит, 2^{16} значений. Чтобы софтмакс на таком количестве не ломался, ужимаем амплитуды с сохранением выразительности (см. вопрос 12).

Функция потерь. У нас должно быть какое-то распределение. Чтобы было категориальное, в самом конце на каждый сэмпл навешиваем softmax и предсказываем вероятность той или иной амплитуды. Но в целом нам ничего не мешает использовать нормальное распределение. Или смесь распределений.

Почему медленно: каузальные свертки. Мы делаем что-то для каждого сэмпла отдельно, т. к. следующий зависит от предыдущего. Усовершенство-

ванные сетки работают с несколькими сэмплами одновременно. Например, изначально кормятся белым шумом:



14 Что является обучающей выборкой в задаче рекомендаций? Что такое рейтинг? Приведите примеры, как он может задаваться.

Поступает пара (u, i) – user, item.

По ней считаем какие-то признаки $x_1^{u,i}, ...x_\alpha^{u,i}$. Например, сколько лайков u поставил всем видео этого автора; как много видео с такими же тегами просмотрены; отношение всех кликов ко всем просмотрам это видео.

Обучающая выборка – R, набор признаков для всех известных пар (u, i).

Рейтинг – какая-то оценка результата взаимодействия пользователя u с айтемом i. Это может быть вещественное число, например, рейтинг, поставленный фильму, от 1 до 10. Может быть что-то бинарное – поставил или не поставил лайк. Что-то категориальное – в фейсбуке может быть лайк с разными реакциями. Множество необязательно будет упорядоченным.

15 Какие вы знаете метрики качества рекомендаций? Как можно измерить ошибку, если рейтинги вещественные? Если рейтинги — это классы? Как учесть, что нам важен порядок, в котором задаются объекты? Как устроена метрика nDCG?

Если рейтинги вещественные, то можно свести к задаче регресии. Тогда метрики будут MSE, MAE, RMSE и так далее. Если же мы сводим к задаче классификации, также можно взять знакомые метрики вроде F-меры, AUC ROC и т.д. Но рекомендательная система же выдает какое-то ограниченное число айтемов (и нам важно, что находится именно в топе наших рекомендаций). Тогда предполагаем, что рек.система выдает ранжированный список айтемов, и будем считать, что мы показываем пользователю top-k рекомендаций. Поэтому на этом числе k мы и будем считать качество.

hitrate@k = $[R_u(k) \cap L_u \neq \varnothing]$, где $R_u(k)$ – наши top-k рекомендаций, L_u – айтемы, которые пользователю действительно понравились. То есть это индикатор того, что хотя бы 1 из наших рекомендованных айтемов понравился пользователю.

precision@k = $\dfrac{|R_u(k) \cap L_u|}{k}$ – сколько айтемов из тех, что мы порекомендовали, пользователю действительно понравилось.

recall@k = $\dfrac{|R_u(k) \cap L_u|}{|L_u|}$ – показыыает, насколько мы смогли вытянуть

в топ релевантный контент.

Есть метрики более специфичные для рекомендаций – это метрики качества ранжирования. Неплохо было бы учитывать, в каком порядке мы выдаем рекоемндации (айтемы, которые по идее должны сильно зайти пользователю, должны идти первыми, например).

Пусть $a_{ui} = a(u, i)$, то есть предсказание нашей системы по этому юзеру и по этому айтема. Далее мы сортируем айтемы по возрастанию a_{ui} для конкретного юезра u (и берем первые k рекомендаций). Тогда для этого списка рекомендаций посчитать меру DCG, которая будет накидывать штраф за позицию айтема в ранджировании.

$$\text{DCG@k} = \sum_{p=1}^{k} g(r_{uip}) d(p)$$

p – номер позиции в ранжировании, r_{uip} – оценка, которую пользователь u поставил айтему на позиции p (истинные рейтинги). $g(r_{uip}) = 2^r - 1$ или $g(r) = r_{uip}$. $d(p) = \dfrac{1}{\log(p+1)}$, это как раз штраф за позицию. Например, если у нас взаимодействие может принимать только значения 0 и 1 (например, был клик или нет), то мы просто будем складывать $1/\log(1 + \text{p})$, и эта сумма будет больше, если единички у нас оказываются в начале списка рекомендаций.

Можно посчитать еще normalized DCG@k(u) = $\frac{DCG@k(u)}{maxDCG@k(u)}$.

Как еще можно измерить качество рекомендательных систем?
1. Novelty (число айтемов, которые раньше пользователю не рекомендова-

ли)

2. Разнообразие. Можно посчитать расстояние между эмбеддингами айтемов, которые мы порекомендовали. Или использовать метаинформацию об айтемах (жанры, исполнители) и посчитать ее дисперсию.

3. Serendipity – умение рекомендовать редкие айтемы, которые пользователю заходят. Например, b – новая книга, B – множество книг, которые пользователь уже оценил, C_{Bw} – число книг автора w в множестве B, C_B – максимальное число книг одного автора в B. Тогда мы можем измерить сходство между b и B.

d(b, B) = $\frac{1+C_B-C_{Bw}}{1+C_B}$

То есть рекомендация будет новее, если юзер еще не видел айтемы этого автора/испольнителя/etc.

16 Как устроены user-based рекомендации?

Ссылка на оригинальный конспект: https://github.com/hse-ds/iad-applied-ds/blob/master/2020/lectures/lecture01-recommender.pdf

Как понять, что пользователю может понравиться товар? Первый вариант — поискать похожих на него пользователей и посмотреть, что нравится им; также можно поискать товары, похожие на те, которые этот пользователь уже покупал. Методы коллаборативной фильтрации строят рекомендации для пользователя на основе похожестей между пользователями и товарами, которые похожие пользователи выбирают.

Сначала необходимо определить, что пользователи похожи между собой. Для этого считается корреляция Пирсона.

Далее мы можем посчитать сходство товаров исходя из тех рейтингов, что рассматриваемые два пользователя им поставили.

Эти формулы определения похожести между пользователями приведены здесь:

Два пользователя похожи, если они ставят товарам одинаковые оценки. Рассмотрим двух пользователей u и v Обозначим через I_{uv} множество товаров i, для которых известны оценки обоих пользователей:

$$I_{uv} = \{i \in I \mid \exists r_{ui} \ \& \ \exists r_{vi}\}.$$

Тогда сходство двух данных пользователей можно вычислить через корреляцию Пирсона:

$$w_{uv} = \frac{\sum_{i \in I_{uv}} (r_{ui} - \bar{r}_u)(r_{vi} - \bar{r}_v)}{\sqrt{\sum_{i \in I_{uv}} (r_{ui} - \bar{r}_u)^2} \sqrt{\sum_{i \in I_{uv}} (r_{vi} - \bar{r}_v)^2}},$$

где $\bar{r}_u$ и $\bar{r}_v$ — средние рейтинги пользователей по множеству товаров I_{uv}.

Чтобы вычислять сходства между товарами i и j, введём множество пользователей U_{ij}, для которых известны рейтинги этих товаров:

$$U_{ij} = \{u \in U \mid \exists r_{ui} \ \& \ \exists r_{uj}\}.$$

Тогда сходство двух данных товаров можно вычислить через корреляцию Пирсона:

$$w_{ij} = \frac{\sum_{u \in U_{ij}} (r_{ui} - \bar{r}_i)(r_{uj} - \bar{r}_j)}{\sqrt{\sum_{u \in U_{ij}} (r_{ui} - \bar{r}_i)^2} \sqrt{\sum_{u \in U_{ij}} (r_{uj} - \bar{r}_j)^2}},$$

где $\bar{r}_i$ и $\bar{r}_j$ — средние рейтинги товаров по множеству пользователей U_{ij}. Отметим, что существуют и другие способы вычисления похожестей — например, можно вычислять скалярные произведения между векторами рейтингов двух товаров.

Рис. 7: Формулы для определения похожести пользователей

После того, как мы определили степень похожести товаров между собой и пользователей между собой, нам нужно понять, какие все-таки товары нам рекомендовать пользователю.

Здесь существует два подхода:

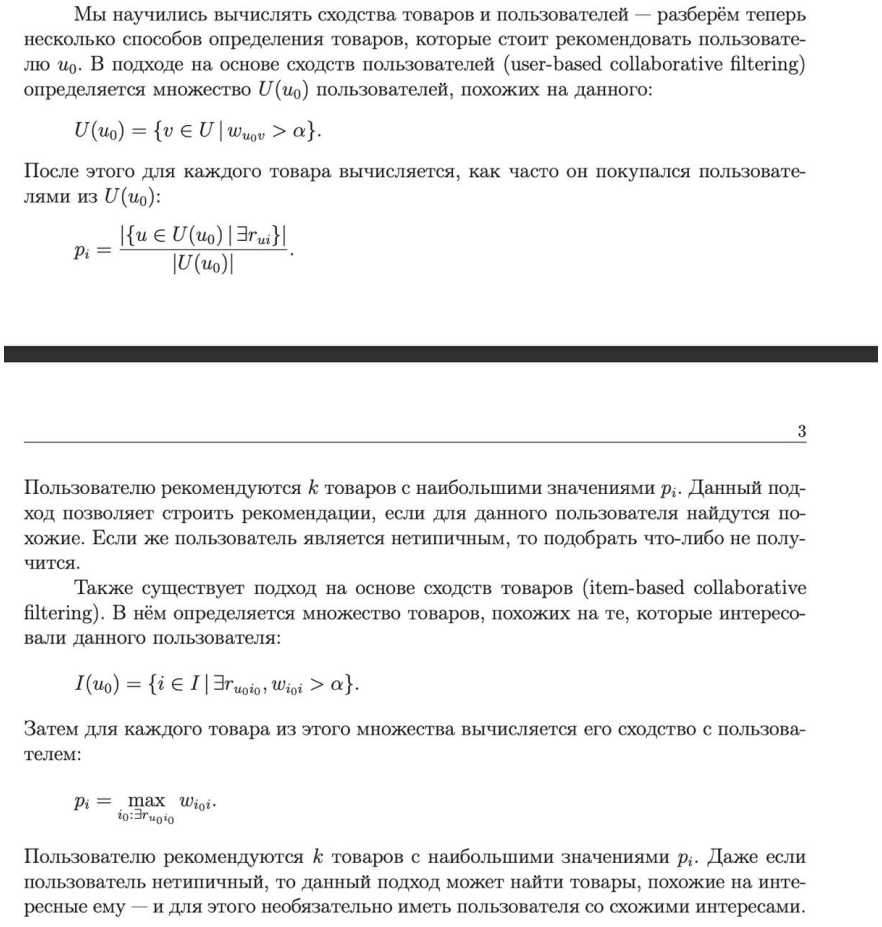
1) user-based collaborative filtering

2) item-based collaborative filtering

В основе первого подхода лежит процедура, согласно которой мы определяем множество похожих пользователей $U(u_0)$: $U(u_0) = \{v \in U | w_{u_0 v} > \alpha\}$. Эта штука называется коллаборацией, то есть множество похожих пользователей, которые (примерно) одинако оценивают продукты.

Дальше рекомендуем пользователи те продукты, которые он сам не оценил, но которые хорошо оценили пользователи из коллаборации.

Для этого мы считаем, как часто товар покупался пользователем из множества $U(u_0)$



Мы научились вычислять сходства товаров и пользователей — разберём теперь несколько способов определения товаров, которые стоит рекомендовать пользователю u_0. В подходе на основе сходств пользователей (user-based collaborative filtering) определяется множество $U(u_0)$ пользователей, похожих на данного:

$$U(u_0) = \{v \in U \mid w_{u_0 v} > \alpha\}.$$

После этого для каждого товара вычисляется, как часто он покупался пользователями из $U(u_0)$:

$$p_i = \frac{|\{u \in U(u_0) \mid \exists r_{ui}\}|}{|U(u_0)|}.$$

Пользователю рекомендуются k товаров с наибольшими значениями p_i. Данный подход позволяет строить рекомендации, если для данного пользователя найдутся похожие. Если же пользователь является нетипичным, то подобрать что-либо не получится.

Также существует подход на основе сходств товаров (item-based collaborative filtering). В нём определяется множество товаров, похожих на те, которые интересовали данного пользователя:

$$I(u_0) = \{i \in I \mid \exists r_{u_0 i_0}, w_{i_0 i} > \alpha\}.$$

Затем для каждого товара из этого множества вычисляется его сходство с пользователем:

$$p_i = \max_{i_0 : \exists r_{u_0 i_0}} w_{i_0 i}.$$

Пользователю рекомендуются k товаров с наибольшими значениями p_i. Даже если пользователь нетипичный, то данный подход может найти товары, похожие на интересные ему — и для этого необязательно иметь пользователя со схожими интересами.

Рис. 8: Формула рассчета частоты покупки товаром пользователями из коллаборации

17 Что такое модель со скрытыми переменными (LFM)? Как в ней предсказывается рейтинг? Как устроен функционал, на который она обучается?

Для каждого пользователя мы имеем d-мерный вектор p_u, для каждого айтема – d-мерный вектор q_i. Мы хотим, чтобы $< p_u, q_i > \approx r_{ui}$, то есть чтобы скалярное произведение этих векторов аппроксимировало истинный рейтинг.

В вольной интерпретации (с лекции) можно представить, что d – это число жанров вообще среди наших айтемов. В p_u мы имеем набор оценок, насколько юзеру нравится каждый из жанров, а в q_i – распределение жанров в конкретном айтеме. Тогда $< p_u, q_i >$ – мера сонаправленности, чем больше метч между юзером и айтемом, тем больше скалярное произведение. По факту конечно внутри векторов не жанры и предпочтения юзеров, а какие латентные неинтерпретируемые факторы, которые рек.система сама выделила.

Функционал: $\sum\limits_{(u,i,r_{ui})\in\mathbb{R}} (r_{ui} - b_u - b_i - < p_u, q_i >)^2 \to min_{b_u,b_i,q_i,p_u}$

То есть мы не просто считаем квадратичную ошибку между истинным рейтингом и нашей аппроксимацией скалярным произведением векторов, но и отнимаем из истинного рейтинга среднюю оценку пользователя на всех айтемах (b_u) и среднюю оценку этого айтема (b_i) (т.к., например, какие-то юзеры всем ставлят только 4-5 звездочек, и для них 4 это скорее плохо,

а кто-то очень сильно придирается, и для них 4 это круто). По сути в этом функционале мы обучаем оба элемента скалярного произведения – и вектор пользователя, и вектор айтема. Еще к этому функционалу можно накинуть регуляризацию (потребовать, чтобы нормы векторов были не очень большими) ($\lambda \sum ||p_u||^2 + \mu \sum ||q_i||^2$).

В чем тут есть проблема? Мы тут перемножаем параметры, которые обучаем, это плохо (функция не выпуклая, локальные минимумы появляются и все такое).

По факту у нас есть матрица P, где лежат все вектора для пользователей, и матрица Q, где лежат все вектора для айтемов, и мы пытаемся осуществить приближение $P^T Q \approx R$.

Как обучаем?
1) SGD. На каждой итерации берем случайные пары u, i, r_{ui}. Но это довольно плохо работает
2) Alternating Least Squares (ALS). Как-то инициализируем матрицы P и Q. Далее фиксируем Q, обучаем на функционал. Тогда вектора q_i оказываются фиксированы, они типа как постоянные признаки, а вектора p_u – это типа как веса при признаках. Получается что-то вроде линейной модели. И так для каждого пользователя мы натрасиваем вектор p. Потом фиксируем матрицу P и подбираем вектора в Q и вот так шаги меняются.
Это работает лучше, так как при фиксированнии части скаларного изображения получается нормальная линейная выпуклая функция (как в обычных линейных моделях).

Практический нюанс (хз надо или нет но пусть будет). Потоянно обновлять матрицы вычислительно тяжко. А айтемов у нас обычно сильно меньше, чем пользователей. Тогда мы можем хранить только сравнительно небольшую матрицу Q. При появлении нового юзера подбираем для него p_u при фиксированном Q, и делаем рекомендацию через $< p_u, q_i >$ (делаем один шаг в ALS). То есть векторы айтемов мы фиксируем, а векторы пользователей постоянно дообучаем на основе последних действий. Раз в сутки/неделю/месяц/хз полностью осуществляем ALS и обновляем матрицу Q тоже.

18 Как устроена модель iALS?

Нужна для учета неявной информации.

Бывают ситуации, когда рейтинги специфичные. Например, для пар u, i мы либо знаем, что взаимодействие было, либо не знаем ничего. Например, мне отгрузили информацию про то, как пользователи ходят по интернету. Я знаю, что он там только был. Не знаю, понравилось ли ему. Или человек как овощ смотрел Netflix – и что с ним делать? Целевая переменная принимает одно значение, и другие модели can't help. Есть iALS – implicit alternating least squares.

Ввожу показатель $S_{u,i} = 1$ (взаимодействие было) или 0 (не было). И показатель $C_{u,i} = \alpha$, если $S_{u,i} = 1$, и 1 в обратном случае. $\alpha \sim 50$

Изначально все i, с которыми пользователь взаимодействовал – положительный пример. Нет – отрицательный. Но это неправильно. Вдруг они понравятся пользователю, когда увидит? Поэтому мы просто учитываем отрицательную информацию с очень маленьким весом. За это отвечает $C_{u,i}$. Так я создаю себе второй класс i, с которыми пользователь не взаимодействовал.

Суммируем $C_{u,i}$ по всем айтемам, аналогично LFM:

$$\sum_{u=1}^{n}\sum_{i=1}^{m} C_{u,i}(S_{u,i} - b_i - b_u - < p_u, q_i >)^2 \to \min b_u, b_i, p_u, q_i \qquad (14)$$

19 Как обычно устроена рекомендательная система? Опишите шаги отбора кандидатов, ранжирования, переранжирования. Приведите примеры, как каждый из них может быть устроен

В рекомендательной системе может участвовать очень большое количество то- варов. При каждом посещении пользователем веб-страницы, где есть блок рекомендаций, необходимо выдать ему k наиболее подходящих товаров, причём достаточно быстро (пользователь не может ждать минуту, пока загрузится страница). В хоро- шей рекомендательной системе участвуют сотни признаков — их вычисление для каждого товара, а затем ещё и применение ко всем товарам градиентного бустинга или графа вычислений вряд ли получится успеть сделать за 1 секунду. Из-за этого рекомендательные системы работают в несколько этапов: обычно всё начинается с отбора кандидатов, где быстрая модель выбирает небольшое количество (тысячи или десятки тысяч) товаров, а затем только для этих товаров вычисляется полный на- бор признаков и применяется полноценная модель. В качестве быстрой модели может выступать линейная модель на нескольких самых важных признаках или, например, простая коллаборативная модель.

19.1 Отбор кандидатов

Отбор кандидатов это про то, как выбрать подмножество предметов, которые пользователю интересны.

- те же предметы, производители, что и в истории пользователя

- самое популярное сейчас
- Матричное разложение: пользователь - исполнитель. $r_{user,singer} \approx < p_{user}, p_{singer} >$
- на основе сходства: p_{user}, q_{item} - эмбеддинги (например, из LFM). Ищем K items с максимальным значением скалярного произведения $< p_{user}, q_{item} >$. Ищу вещи, которые ближе всего к пользователю.

О ранжировании смотрите здесь https://developers.google.com/machine-learning/recommendation/dnn/re-ranking

и здесь https://github.com/hse-ds/iad-applied-ds/blob/master/2020/lectures/lecture03-recommender.pdf

После отбора кандидатов следует ранжирование

Переранжирование - помимо максимизации вероятности клика, мы хотим учесть бизнес-требования. Например, мы не хотим показать видео от одного и того же автора. Мы пересортировываем наш изначальное ранжирование. Так мы учитываем дополнительные требования к нашей рекомендациии.