Московский физико-технический институт

ФАКУЛЬТЕТ ИННОВАЦИЙ И ВЫСОКИХ ТЕХНОЛОГИЙ

ПРИКЛАДНОЕ МАШИННОЕ ОБУЧЕНИЕ

Лектор: Радослав Нейчев

КОНСПЕКТ ЛЕКЦИЙ

автор: Наталья Лунёва

1 декабря 2020 г.

Оглавление

1 Natural Language Processing

1.1 Embeddings

В этом параграфе также будут рассмотрены следующие методы векторного представления текстов: *bag-of-words, bag-of-ngramms, tf-idf.*

Text preprocessing. Поскольку многие слова в тексте являются копиями других слов, просто находясь в различных формах (например, *кот* и *коты* пишутся по-разному, но имеют один и тот же смысл), то для упрощения модели применятся предобработка текста. Вот два основных способа:

- **Stemming** - процесс нахождения основы слова (не обязательно корня) по заданным правилам. Например, если у слова есть какой-либо суффикс, то он обрезается: котик → кот. Однако иногда случаются *overstemming* (если два различных слова имеют одну и ту же основу: university, universal → univers) и *understemming* (здесь, наоборот, близкие слова имеют разное представление: data → dat, datum → datu).
- **Lemmatization** - процесс приведения слова к его нормальной форме: бежал → бежать, меня → я, первого → первый и т.д.

Также в процессе предобработки стоит обратить внимание на заглавные буквы, пунктуацию, числа, стоп-слова (*да*, *ага*, *и*, *но* и др.). Методы обработки выбираются в зависимости от решаемой задачи.

Bag-of-words. Для начала составляется словарь всех или наиболее часто употребляемых слов исходного датасета. Затем каждому тексту ставится в соответствие вектор длины словаря, где на i-ой позиции записывается количество вхождений i-ого слова. Такой подход уже позволяет сравнивать тексты, например при помощи косинусной меры. Однако у него есть множество недостатков:

- теряется информация о порядке слов;
- вектора представлений слишком большие и разреженные;
- вектора представлений не нормализованы.

Bag-of-ngramms. Чтобы улучшить BoW, будем рассматривать n-граммы, вместо слов. Для фиксированного n разобъём предложения на n-граммы и уже их примем за единицу токена. Далее будем

работать так же как и в предыдущем подходе, т.е. считать количество вхождений токенов в текст и т.д. Такой способ позволяет учитывать порядок слов, но тут же возникает проблема разросшегося словаря. Решение - выкидываем все n-граммы, которые не очень информативны. Неинформативными можно считать словосочетания с высокой частотой встречаемости (скорее всего это какие-то общеупотребимые фразы) и с низкой частотой встречаемости (слишком специфичные фразы либо опечатки).

Term frequency - inverse document frequency. Идея tf-idf состоит в том, чтобы представить некоторую меру, позволяющую оценить значимость слова в данном контексте. Так, например, слова с высокой частотой употребления в пределах одного документа и с низкой частотой употребления в остальных получат большой вес tf-idf. И напротив, слова, встречающиеся почти в каждом документе исходного корпуса текстов, будут иметь маленькую меру значимости.

TF есть отношение числа вхождений некоторого слова к общему числу слов документа:

$$tf(t,d) = \frac{n_t}{\sum_k n_k},$$

где n_t - число вхождений слова t в документ d.

IDF - инверсия частоты, с которой некоторое слово встречается в документах коллекции:

$$idf(t,D) = \log \frac{|D|}{|\{d_i \in D | t \in d_i\}|},$$

где $|D|$ - число документов в коллекции, а $|\{d_i \in D | t \in d_i\}|$ - число документов из коллекции D, в которых встречается слово t. Таким образом, учёт IDF уменьшает вес широкоупотребимых слов.

Сама мера TF-IDF представляется в виде произведения двух сомножителей:

$$tf\text{-}idf(t,d,D) = tf(t,d) \cdot idf(t,D).$$

Теперь перейдём к **word embeddings.** Под эмбеддингом будем подразумевать *маломерное информативное векторное представление* слов (к примеру, one-hot encoding таковым не является).
Одним из способов получения эмбеддингов является построение матрицы словосочетаний. Для двух

слов u и v из словаря посчитаем n_{uv} - количество вхождений данной пары в текстовые последовательности фиксированного размера. Но лучше считать *pointwise mutual information*:

$$PMI = \log \frac{p(u,v)}{p(u)p(v)} = \log \frac{n_{uv}n}{n_u n_v},$$

где $p(u,v)$ - вероятность встретить слова u и v вместе, а $p(u), p(v)$ - вероятность встретить каждое слово по отдельности. А ещё лучше - *positive PMI*:

$$pPMI = \max(0, PMI).$$

Итак, PMI позволяет оценить, насколько слова u и v взаимосвязаны и важны друг для друга. Теперь построим матрицу X, где на позиции ij стоит значение $PMI(i,j)$ или какая-либо другая мера. Используя матричное разложение $X = \Phi\Theta$ для понижения размерности, получаем, что Φ как раз таки является матрицей векторных представлений, обусловленных на контекст. Однако матричное разложение не позволяет учитывать, какие слова для нас важны, а какие нет, используя, например, функцию потерь. Тем самым встаёт задача обучить эмбеддинги. О подходах к решению этой проблемы будет рассказано в следующем параграфе.

1.2 word2vec: linearity, skip-gram, negative sampling

Используя предположение, что слово определяется контекстом, можно обучать эмбеддинги. Для построения обучающей выборки возьмём слово и n-граммы, содержащие его и его соседей. Таким образом, поставлена задача классификации, решить которую можно при помощи простой нейронной сети. На вход подадим one-hot encoded вектор, соответствующий некоторому слову из словаря, и на выходе получим его эмбеддинг. Самое замечательное - это то, что матрица весов сети есть ни что иное, как матрица эмбеддингов, т.е. в i-ой строке находится векторное представление i-ого слова.

В word2vec реализованы две архитектуры: *continuous BOW* (предсказываем слово исходя из его контекста: $p(w_i|w_{i-h}, ..., w_{i+h})$) и *skip-gram* (здесь наоборот, используя текущее слово, предугадываем окружающие его слова: $p(w_{i-h}, ..., w_{i+h}|w_i)$).

Далее рассмотрим подходы, позволяющие улучшить процесс обучения:

- **Subsampling.** Чтобы учесть несбалансированность между редкими и часто встречающимися словами, будем брать слово для обучения с вероятностью $P(w_i) = 1 - \sqrt{\frac{t}{f(w_i)}}$, где $f(w_i)$ - частота слова w_i, а t - фиксированная константа (обычно 10^{-5}).

- **Negative sampling.** При обучении нейронной сети на каждом шаге корректируются все её веса. Но так как размер обучающей выборки и количество параметров сети огромны, весь процесс может занять слишком много времени. Поскольку входной вектор является one-hot encoded, то на выходе по сути предсказываются веса только для одного конкретного слова. Тогда для оптимизации будем вычислять ошибку лишь на некоторых словах: на "правильном" и на небольшом количестве "неправильных" (в оригинальной статье от гугла приводятся следующие значения: достаточно 5-20 слов для маленьких датасетов и 2-5 слов для больших). Для остальных же слов положим значение ошибки равное нулю, т.е. обновления весов происходить не будет. Вероятность выбора слова можно определить как $P(w_i) = \frac{f(w_i)^{3/4}}{\sum_j f(w_j)^{3/4}}$, т.е. веса для более частых слов будут обновляться чаще.

1.3 Unsupervised translation approach

Как уже говорилось ранее, мы считаем, что слово определяется своим контекстом. Тогда можно заметить, что для двух похожих языков одно и то же слово скорее всего будет употреблено в одинаковом контексте. И, следовательно, в этих языках расстояния между словами сохранятся. Эту идею можно использовать для машинного перевода. Задача заключается в том, чтобы найти ортогональное отображение из одного пространства эмбеддингов в другое.

Будем минимизировать следующую величину:

$$W^* = \underset{W \in O_d(\mathbb{R})}{\arg\min} ||WX - Y||_F = UV^T,$$

где X и Y - матрицы эмбеддингов заданных языков, а матрицы U и V взяты из сингулярного разложения: $U\Sigma V^T = SVD(YX^T)$.

Тогда переводом слова s является слово $t = \arg\max_t \cos(Wx_s, y_t)$. Такой выбор связан с тем, что при отображении во второе пространство мы не обязательно попадём прямо в эмбеддинг, а можем оказаться лишь рядом с ним. Поэтому выбирается слово, эмбеддинг которого находится наиболее близко к отображённому.

Как считать расстояния между словами? Для нахождения расстояний между векторами эмбеддингов обычная евклидова метрика не подходит: во время обучения эмбеддингов слова, встречающиеся часто, будут иметь большую евклидову норму, т.к. они будут выталкиваться в ту часть пространства, где сосредоточены слова по их теме. Интересно заметить, что норма общеупотребимых слов (например, артиклей) всё равно не станет слишком большой, т.к. нет определённого контекста, в котором они встречаются, и поэтому их эмбеддинги будет бросать во все стороны одинаково. Возвращаясь к нахождению расстояний, нас волнует не то, насколько слова отличаются по норме, а то, насколько они близки по смыслу. И поскольку векторы похожих слов будут почти сонаправлены, можно использовать косинусную меру близости:

$$similarity = \cos(\theta) = \frac{\mathbf{A} \cdot \mathbf{B}}{||\mathbf{A}||\ ||\mathbf{B}||} = \frac{\sum_{i=1}^{n} A_i B_i}{\sqrt{\sum_{i=1}^{n} A_i^2}\sqrt{\sum_{i=1}^{n} B_i^2}}$$

1.4 Ways to work with text data

В прошлом семестре мы изучали архитектуры рекуррентных и свёрточныйх нейросетей. В этом разделе будут рассмотрены методы их использования для текстовых данных.

RNN for texts. Суть использования рекурентных сетей заключается в последовательной обработке текста. То есть на каждом шаге мы подаём следующий вектор информации, обрабатываем его и получаем новый вектор контекста. Если вспомнить задание, в котором нужно было сгенерировать поэзию Пушкина или Шекспира, то по сути здесь мы делаем то же самое, но входным вектором уже является не закодированный символ, а эмбеддинг токенов.

CNN for texts. Как мы уже знаем, текстовые подпоследовательности несут больше информации, чем отдельный набор слов. Тогда попробуем использовать операцию свёртки к n-граммам. То есть для каждой последовательности $\overrightarrow{c_i} = (c_i, ..., c_{i+n-1})^T$ из текста посчитаем $p_i = f(W\overrightarrow{c_i} + \overrightarrow{b})$, где f -

некоторая функция активации. Тем самым получили новое признаковое представление, которое можно сворачивать и дальше. Но есть небольшая проблема: длина входящих последовательностей различна. Решение - будем пуллить векторы до фиксированного размера (обычно берут *max-over-time-pooling*). Достоинство свёрточных сетей в том, что они хорошо параллелизуются.

1.5 Machine translation metrics, quality functions

Допустим, мы уже научились переводить текст с одного языка на другой. Теперь нужно оценить, насколько качественным получился перевод. Для этого существуют различные метрики, описание которых будет рассмотрено в данном параграфе.

Bilingual Evaluation Understudy. BLEU позволяет оценить, как много n-грамм из исходного текста угадала модель, а также штрафует за слишком короткие переводы:

$$BLEU = brevity\ penalty \cdot \left(\prod_{i=1}^{n} precision_i \right)^{1/n} \cdot 100\%$$

$$brevity\ penalty = \min\left(1, \frac{output\ length}{reference\ length}\right)$$

Недостатки:

- существует несколько вариантов перевода предложения (например, используя синонимы), но за них мы будем получать $BLEU = 0$.
- хороший перевод может иметь низкое значение BLEU из-за маленького количества схожих n-грамм по сравнению с человеческим переводом.

Существуют и другие способы оценки качества, например:

- **METEOR** в отличие от BLEU, учитывает синонимы, сопоставляя их с точным значением слов.
- **NIST** работает как BLEU, но присваивает больший вес редким n-граммам и меньше штрафует за короткие тексты.
- **TER** измеряет объём изменений, которые нужно сделать, чтобы текст совпал с исходным.

1.6 Attention, self-attention approaches

Предпосылки к появлению *attention*. Пусть поставлена seq2seq задача (т.е. получение одной тектовой последовательности из другой). Если брать многослойный энкодер на основе RNN или CNN, то для генерации последовательности B используется только последнее состояние, в котором хранится вся информация об исходной последовательности A. Но в предыдущих слоях также содержится важная информация о подпоследовательностях из A. Как тогда учесть её?

Основная идея - пусть декодер сам будет выбирать, какое состояние енкодера ему лучше использовать. Б.о.о. считаем, что векторы енкодера и декодера одной размерности. Тогда возьмём первое состояние декодера и посчитаем скалярное произведение со всеми состояниями енкодера. Полученные величины будем называть *вниманием*. Полагаем, что чем важее состояние, тем больше результат. Поэтому будем учитывать каждое состояние енкодера с весом, соответствующим его *attention score* (перед этим применив softmax к произведениям для получения распределения вниманий). Сумма взвешенных состояний есть *attention output*. И так будем делать для всех состояний декодера.

Формально:

$h_1, ..., h_n \in \mathbb{R}^k$ - скрытые состояния энкодера.

$s_t \in \mathbb{R}^k$ - скрытое состояние декодера на шаге t.

$e^t = [s^T h_1, ..., s^T h_n]$ - attention score для s_t.

$a_t = \sum_{i=1}^n \alpha_i^t h_i \in \mathbb{R}^k$, где $\alpha^t = softmax(e^t)$, - attention output.

На самом деле можно считать не только скалярное произведение:

- $e_i = s^T W h_i$ - мультипликативное внимание,
  - $W \in \mathbb{R}^{d_2 \times d_1}$ - весовая матрица.
- $e_i = v^T \tanh(W_1 h_i + W_2 s)$ - аддитивное внимание,
  - $W_1 \in \mathbb{R}^{d_3 \times d_1}, W_2 \in \mathbb{R}^{d_2 \times d_1}$ - весовые матрицы,
  - $v \in \mathbb{R}^{d_3}$ - вектор весов.

Self-attention. Механизм внутреннего внимания выявляет взаимосвязи между словами внутри одного текста. Основная суть здесь похожа на простой attention, но операции скалярного произведения считаются не для двух скрытых состояний, а для двух слов. Вообще, на эту тему и на тему трансформеров

написано много хороших статей (например, вот и вот), так что не вижу смысла их здесь переписывать. Я лишь приведу несколько различий между *attention* и *self-attention* для лучшего понимания:

1. AT, находясь в декодере, смотрит на состояния предыдущих слоёв энкодера. SA используется по отдельности в каждом из блоков и обращает внимание на вход того же слоя, где он и применяется.

2. SA выявляет зависимости между частями одной последовательности. AT устанавливает зависимости между двумя последовательностями, позволяя декодеру лучше улавливать исходный контекст.

3. AT применяется один раз при связывании энкодера и декодера. SA может быть применён несколько раз последовательно внутри одного блока и даже параллельно для одного слоя (в последнем случае получится multi-head attention).

1.7 Context based models overview

В этом параграфе мы рассмотрим *Bert*, построенный на базе архитектуры Transformers, и модели, которые предшествовали его появлению: *OpenAI* и *ELMo*. Советую также прочитать статью на хабре.

OpenAI Transformers. Обычный трансформер решал задачу машинного перевода, т.е. перевода данных из одного представления в другое. OpenAI Transformer решает задачу языкового моделирования, что позволяет предсказывать следующее слово на основании предыдущего контекста, используя только декодер. Основная задача - научиться порождать информативное представление для всего текста, которое потом может быть использовано для других задач, например text classification. Фитча модели в том, что меняется не столько архитектура трансформеров, сколько представление входных данных, благодаря чему происходит извлечение новой информации.

ELMo (Embeddings from Language Models). Проблема обычных word2vec подходов состроит в том, что полученные эмбеддинги хранят не всю возможную информацию о контексте (как пример, у омонимичных слов эмбеддинги будут совпадать, хотя очевидно они имеют разное значение). ELMo позволяет создать более глубокое контекстуальное представление слов. Для обучения возьмём обычную языковую модель, например LSTM; в качестве новых эмбеддингов можно использовать скрытые состояния. Но в таком случае будет учитываться только левый контекст. Решение - сделать модель двунаправленной,

т.е. прогонять текст не только слева направо, но и справа налево. Более того, модель можно сделать многоступенчатой и обучаться на новых эмбеддингах для получения более качественного результата.

BERT (Bidirectional Encoder Representation from Transformers). Широкой задачей Берта является получение информативных представлений входных данных. Более узкие задачи - построение информативных эмбеддингов, обусловленных на контекст, как в ELMo, и обучение для классификации, как в OpenAI Transformers. Берт состоит только из энкодера, который по своей структуре похож на обычный энкодер из Трансформера. Нововведения Берта заключаются в следующем:

1. В начало добавляется специальный токен [CLS], который во время обучения собирает в себя всю необходимую информацию для решения задачи классификации.
2. При наложении маски на исходный текст Берт обучается предсказывать слова, которые были пропущены (замаскированы).

Таким образом, Берт добился выдающихся результатов, поскольку он обучался решать сразу несколько задач. Из-за этого модель была вынуждена извлекать как можно больше информации из данных.

1.8 Even more architectures!

ULMFiT (Universal Language Model Fine-tuning for Text Classification) - это LSTM с dropout'ом, добавленным почти во все возможные места. Но в чём была проблема добавить дропаут в рекуррентную модель? Ответ заклчается в том, что в обычных feedforward сетях преобразование на каждом шаге имеет свою матрицу, а в рекуррентных сетях эта матрица одна для всех шагов. И, таким образом, зануляя некоторые веса, модель просто перестаёт обучаться на них. Какие подходы используются в ULMFiT:

- *Encoder Dropout.* Маскируются случайно выбранные слова входной последовательности.
- *Input Dropout.* Происходит зануление определонного столбца (или корректнее сказать элемента) эмбеддинга у всех слов из входящей последовательности.
- *Weight Dropout.* Применяется к некоторым весам матрицы преобразования, причём маска обновляется каждый раз при новом последовательном проходе.
- *Hidden Dropout.* Применяется к скрытым слоям многослойной реккурентной сети.
- *Output Dropout.* Зануляется часть выходной последовательности перед её декодированием.

Transformer-XL. Хотя трансформеры, которые были рассмотрены ранее, хорошо справляются с большим рядом задач, они всё же имеют некоторые недостатки:

- Модель работает с окном фиксированной длины. Это ведёт к тому, что последовательность, которая больше ранее фиксированной длины, разбивается на сегменты, внутри которых происходит независимое обучение, и так теряется возможность улавливать контекст больших текстов. В Transformer-XL добавлена рекуррентность в процедуру обучения: каждый следующий сегмент при подсчёте self-attention использует также скрытые состояния предыдущих сегментов.
- Позиция слов в тексте учитывается наивно. В Transformer-XL positional encoding заменяется на relative positional encoding. Полученные эмбеддинги позволяют оценить, насколько далеко друг от друга находятся слова в тексте.

1.9 Question answering

Здесь будут описаны подходы к получению ответов на вопросы.

Наивное решение. Сначала берём эмбединги текста и вопроса (это мы умеем делать). Затем считаем аттеншн между закодированным вопросом и скрытым состоянием каждого токена в тексте. Результат есть вероятности того, что слово является началом/концом ответа. То есть полагаем, что ответ является целой подпоследовательностью, а не отдельными кусками, разнесёнными по тексту. Формально:

$\alpha_i = softmax_i\ q^T W_s p_i$

$start\ token = \arg\max_i\ \alpha_i$

Аналогично с *end token*, только используем W_e.

Что можно сделать, чтобы улучшить результат? Во-первых, можно добавить в эмбединг текста информацию о словах (например, часть речи). Тогда модель выявит закономерности типа "на вопрос *кто?* ответом скорее всего является существительное". Во-вторых, вместо того чтобы сжимать весь вопрос до одного эмбеддинга, его можно кодировать, как и текст, по токенам. Эта идея реализована в модели BiDAF, или Bidirectional Attention Flow. Не углубляясь в детали, опишем механизм работы. Сначала берутся *context* и *query*, затем вычисляются два внимания: от вопроса к контексту и от контекста к вопросу. Результат конкатенируется, и на основании нового представления решается задача.

2 Reinforcement Learning

2.1 RL problem statement

Основные понятия обучения с подкреплением:

- *Agent* - сущность, которая принимает решения.
- *Environment* - среда, которую наблюдает агент.
- *Observation*, или *state*, - текущее состояние агента.
- *Policy* - некоторое отображение из наблюдений в действия.
- *Action* - действие, совершаемое агентом согласно политике.
- *Feedback*, или *reward*, - награда за действия (может быть отрицательной).

Замечание: по сути, агент - это та же обучаемая модель, просто в rl её принято называть иначе.

Ниже представлено сравнение различных видов обучения: с учителем, без учителя, с подкреплением.

supervised	**unsupervised**	**reinforced**
учимся предсказывать релевантные ответы	пытаемся выявить базовую структуру данных	ищем оптимальную стратегию методом проб и ошибок
нужны правильные ответы	не нужны ни ответы, ни фидбек	агенту нужен фидбек на его собственные действия
модель не влияет на входные данные	модель не влияет на входные данные	агент влияет на среду, а значит и на наблюдения

Теперь перейдём к формальному описанию задачи:

- на каждом шаге имеем состояние $s \in S$, действие $a \in A$ и награду $r \in \mathbb{R}$;
- хотим максимизировать суммарную награду $R = \sum_t r_t$;
- введём политику $\pi(a|s) = P(\text{take action } a \text{ in state } s)$;
- тогда задача сводится к нахождению $\pi^* = \arg\max_\pi \mathbb{E}_\pi[R]$.

И как это максимизировать?

1. играем несколько игр (сессий) по существующей политике;
2. обновляем политику согласно полученному фидбеку;
3. повторяем.

Таким образом, задача обучения с подкреплением состоит в том, чтобы найти оптимальную стратегию агента, максимизируя награду (ср. в обучении с учителем мы минимизировали функцию потерь).

TODO: вставить куда-нибудь замечание, почему плохо, что агент меняет среду. Ответ - в погоне за высокой наградой мы можем сами загнать себя в локальный максимум

2.2 Crossentropy method

Рассмотрим для начала случай так называемых табличных данных, когда имеется конечное число состояний и конечное число действий:

1. инициализируем политику (в нашем случае ею будет матрица размера n_states x $n\_actions$);
2. семплируем N произвольных сессий, на которых будет играть агент;
3. из них выбираем M элитных сессий (т.е. таких, на которых был получен наилучший результат);
4. обновляем политику на основе этих элитных сессий;
5. повторяем.

Если $\pi(a|s) = P$(совершаем действие a в состоянии s), тогда

$$\pi_{new}(a|s) = \frac{\text{сколько раз выбрали действие } a \text{ в состоянии } s}{\text{сколько раз были в состоянии } s}$$

К сожалению табличный подход не работает при большом числе состояний или действий, т.к. количество элементов матрицы станет слишком большим и оптимизировать политику будет невыгодно. В приближенном методе кросс-энтропии алгоритм нахождения оптимальной политики примерно такой же, как и вышеописанный, только вместо матрицы вероятностей используется любая подходящая модель классификации, например *random forest classifier* или *logistic regression*. То есть $\pi_{new}(a|s) = f_\theta(a, s)$. Если пространство действий не конечно, а континуально, тогда можно либо решать задачу регрессии, либо дискретизировать пространство ответов.

2.3 State-value function, Q-function

Недостатком метода кросс-энтропии является то, что при выборе действий мы используем только распределения вероятностей, не учитывая информацию о внешней среде. И затем смотрим на получившийся результат, выбирая лучшие сессии (условно говоря, агент действует наобум в надежде найти оптимальную стратегию). Но как объяснить агенту, чего от него хотят? В этом нам поможет гипотеза вознаграждения Саттона: *наши цели и задачи можно понимать как максимизацию математического ожидания кумулятивной суммы получаемых скалярных величин, которые мы называем наградой.*

Положим

$$G_t = R_t + R_{t+1} + ... + R_T$$

- та самая кумулятивная сумма наград вплоть до конца сессии. В случае, если сессия продолжается до бесконечности, суммарная награда также будет неограничена. Поэтому будем считать её с некоторым дисконтирующим коэффициентом γ (к тому же зачастую лучше сделать что-то хорошее сегодня, чем оставить это на потом). Тогда:

$$G_t = R_t + \gamma R_{t+1} + \gamma^2 R_{t+2} + ... = R_t + \gamma(R_{t+1} + \gamma R_{t+2} + ...) = R_t + \gamma G_{t+1}$$

На самом деле *discounting reward* позволяет учитывать также влияние произошедшего действия в долгосрочной перспективе (т.е. некоторый эффект домино). Тогда формула интерпретируется как:

$$G_0 = R_0 + \gamma R_1 + \gamma^2 R_2 + ... + \gamma^T R_T =$$
$$= (1-\gamma)R_0 + (1-\gamma)\gamma(R_0 + R_1) + (1-\gamma)\gamma^2(R_0 + R_1 + R_2) + ... + \gamma^T \sum_{t=0}^{T} R_t$$

где коэффициент γ^t означает вероятность, что произошедшие действия повлияют на событие шага t, а коэффициент $(1-\gamma)$ - вероятность, что эффект закончится на текущем шаге.

Напомним, что теперь оптимальная политика максимизирует следующую величину:

$$\mathbb{E}_{\pi_\theta}[G_0] = \mathbb{E}_{\pi_\theta}[R_0 + \gamma R_1 + ... + \gamma^T R_T]$$

Введём функцию полезности состояния:

$$v_\pi(s) \triangleq \mathbb{E}_\pi[G_t|S_t = s] = \mathbb{E}_\pi[R_t + \gamma G_{t+1}|S_t = s] =$$

$$= \sum_a \pi(a|s) \sum_{r,s'} p(r, s'|s, a)\big[r + \gamma \mathbb{E}_\pi[G_{t+1}|S_{t+1} = s']\big] =$$

$$= \sum_a \pi(a|s) \sum_{r,s'} p(r, s'|s, a)[r + \gamma v_\pi(s')]$$

Мат. ожидание берётся по тем величинам, которые вносят весь вклад в случайность: по политике π (определяет совершаемое действие) и по среде p (определяет награду за шаг и следующее состояние).

Благодаря функции $v(s)$ мы знаем, какие состояния хорошие, а какие нет. Но эта информация не поможет нам определить стратегию. Поэтому введём функцию полезности действия:

$$q_\pi(s, a) \triangleq \mathbb{E}_\pi[G_t|S_t = s,\ A_t = a] = \mathbb{E}_\pi[R_t + \gamma G_{t+1}|S_t = s, A_t = a] =$$

$$= \sum_{r,s'} p(r, s'|s, a)\big[r + \gamma \mathbb{E}_\pi[G_{t+1}|S_{t+1} = s']\big] = \sum_{r,s'} p(r, s'|s, a)[r + \gamma v_\pi(s')]$$

Неформально, q-функция показывает, сколько мы будем в среднем получать кумулятивной награды, находясь в состоянии s и совершая действие a.

В итоге мы выразили q-функцию через *state-value* функцию, которая, в свою очередь, выражается через q-функцию:

$$v_\pi(s) = \sum_a \pi(a|s) q_\pi(s, a)$$

Теперь можно сравнивать политики:

$$\pi \geq \pi' \ \Leftrightarrow\ v_\pi(s) \geq v_{\pi'}(s)\ \forall s$$

Оптимальная политика π_* - это такая политика, которая лучше или равна любой другой. Если мы используем оптимальную политику из состояния s, то $v_*(s) = \max_\pi v_\pi(s)$ и $q_*(s, a) = \max_\pi q_\pi(s, a)$.

Теперь для функции $v_*(s)$ можно определить рекурсивное соотношение через $v_*(s')$:

$$v_*(s) = \max_a \sum_{r,s'} p(r, s'|s, a)[r + \gamma v_*(s')] \ \left(= \max_a q_*(s, a) \right)$$

Пояснение: поскольку нам известно, в какие состояния s' можно перейти из s, то в определённой выше функции $v(s)$ максимум берётся по второй сумме (условно говоря, вероятность перехода в наилучшее s' будет равна единице).

Используя динамическое программирование, можно найти v-функцию для всех состояний (по теореме о неподвижной точке процесс обновления значений функции должен сойтись). И затем оптимальная политика выражается при помощи q-функции следующим образом:

$$\pi_*(s) = \arg\max_a q_*(a, s)$$

Заметьте, что здесь политика, в отличие от метода кросс-энтропии, однозначно определяет действие для конкретного состояния s. Для некоторой стохастичности можно ввести порог ϵ, при достижении которого совершаемое действие будет выбрано случайно, либо использовать не максимум, а температурный softmax для получения распределения вероятностей.

2.4 Q-learning, approximate Q-learning, DQN

Предыдущий подход содержит несколько проблем. Для начала что делать, если нам не известен параметры среды, а именно $p(r, s'|s, a)$? Как вариант можно насемплировать много сессий и использовать аппроксимацию выборкой. Но в таком случае много времени уйдёт на то, чтобы сыграть целую сессию, причём агент будет делать это с одним и тем же опытом, который обновляется только в конце игры. Тогда сделаем так: будем обновлять q-функцию на каждом шаге агента!

$$Q(s_t, a_t) = \mathop{\mathbb{E}}_{r_t, s_{t+1}} \left[r_t + \gamma \cdot \max_{a'} Q(s_{t+1}, a') \right] \approx \frac{1}{N} \sum_i \left[r_i + \gamma \cdot \max_{a'} Q(s_i^{next}, a') \right]$$

Чтобы не потерять информацию о предыдущем опыте будем использовать скользящее среднее:

$$Q(s,a) = \alpha \cdot \hat{Q}(s,a) + (1-\alpha) \cdot Q(s,a)$$

$$\hat{Q}(s,a) = r(s,a) + \gamma \cdot \max_{a'} Q(s',a')$$

Способ обучения, где в обновлении $\hat{q}$-функции используется взятие максимума, называется Q-learning. В алгоритме обучения EV-SARSA берётся мат.ож.:

$$\hat{Q}(s,a) = r(s,a) + \gamma \cdot \mathop{\mathbb{E}}_{a' \sim \pi(a'|s')} Q(s',a')$$

Experience replay. Ещё одно улучшение заключается в использовании буфера с опытом. Идея состоит в том, чтобы обновлять q-функцию агента не только по текущему состоянию, но и по предыдущей истории. Для этого сохраним взаимодействие агента со средой (т.е. четвёрку s, a, r, s'). Затем возьмём часть истории из буфера и обновим на ней q-функцию. Такой подход позволяет нам учитывать опыт, не вычисляя при этом новый результат взаимодействия, что помогает быстрее сходится.

Approximate Q-learning. Перегруппируем слагаемые в формуле обновления q-функции:

$$Q(s_t,a_t) = Q(s_t,a_t) + \alpha(r_t + \gamma \max_{a'} Q(s_{t+1},a') - Q(s_t,a_t))$$

Полученная формула напоминает градиентный спуск (вернее подъём, так как стоит задача максимизации). Тогда можно посчитать среднеквадратичную ошибку между старым значением $Q(s_t,a_t)$ и новым:

$$L = \big(r_t + \gamma \max_{a'} Q(s_{t+1},a') - Q(s_t,a_t)\big)^2$$

$$\frac{\partial L}{\partial Q} = 2 \cdot \big(r_t + \gamma \max_{a'} Q(s_{t+1},a') - Q(s_t,a_t)\big)$$

Заметим, что в данном случае мы не берём производную по $\max_{a'} Q(s_{t+1},a')$ и у этого есть объяснение: по сути $Q(s_{t+1},a')$ отвечает за следующие состояния и если брать по ней производную, то мы будем подгонять q-функцию под значения себя самой в будущем.

Если принять, что наша q-функция - это некоторая параметрическая модель с матрицей весов W, то на каждом шаге будем обновлять веса по правилу:

$$w_{t+1} = w_t + \alpha \frac{\partial L}{\partial w}$$

Deep Q-network. Пусть нам нужно решить задачу обучения с подкреплением в некоторой среде, содержащей пространственную информацию, например перемещения объекта. Тогда полезно сохранять последовательность состояний, из которой мы сможем извлечь изменения. С этим справится любая свёрточная сеть: на вход её подаётся последовательность, а на выходе будут значения q-функции.

Autocorrelation problem. На самом деле описанный выше подход не работает, и самая главная проблема - корреляция между соседними состояниями. Из-за того что соседние состояния очень похожи, их q-функции также будут отличаться несильно. Поэтому вместо корректного обновления функции, мы будем заменять её на максимум самой себя, что приведёт к неограниченному росту значений.

Чтобы исправить это, предлагается завести вторую сеть, которая будет предсказывать $v(s')$ по состоянию s' (это и есть тот самый $\max_{a'} Q(s', a')$). А затем по текущему значению $Q(s, a)$ и по предсказанному считается лосс, чтобы затем обновить параметры основной сети. Веса дополнительной сети обновляются раз в n шагов, потому что состояния s_t и s_{t+n} уже не так скоррелированы.
Итого, что нужно делать, чтобы достичь хорошего результата в deep q-learning:

- использовать прошлую историю с experience replay (работает не только в DQN);
- брать последовательность состояний для извлечения пространственной информации;
- обновлять параметры сети раз в несколько шагов, чтобы избежать проблему корреляции.

Double DQN. Ещё одна проблема, но не такая значительная, - смещение мат. ожидания ошибки. Мы хотим, чтобы $\mathop{\mathbb{E}}_{s,a} L(s, a) = 0$. Но если брать каждый раз максимум $Q(s', a')$, то среднее значение сместится вправо, поскольку накапливается шум:

пусть $Q^*(s', a') = x$ - истинное значение q-функции для всех a'

так как мы имеем только оценку, то $Q(s', a_i) = x + \epsilon_i$, где $\epsilon_i \sim \mathcal{N}(0, \sigma)$

получаем, $\mathop{\mathbb{E}}_{s'} \max_{a'} Q(s', a') \geq \max_{a'} \mathop{\mathbb{E}}_{s'} Q(s', a') \geq \mathop{\mathbb{E}}_{s', a'} Q(s', a')$

Решение - использовать две оценки q-значений Q^A и Q^B, которые должны компенсировать ошибки друг друга, поскольку эти ошибки независимы:

$$y = r + \gamma \max_{a'} Q(s', a') = r + \gamma Q(s', argmax_{a'} Q(s', a'))$$

$$y = r + \gamma Q^A(s', argmax_{a'} Q^B(s', a'))$$

По одной q-функции будем выбирать действие, а по второй обновляться, причём выбор функции для обновления каждый раз должен происходить случайно.

2.5 Policy gradient and REINFORCE algorithm

Напомню, почему нам пришлось ввести value based подход. Мы хотели обучаться на среде, а для этого нужна некоторая дифференцируемая функция (обычный reward не подходил, т.к. это просто число, которое не несёт никакой информации). Поэтому мы ввели q-функцию и максимизировали её разными методами: проще - дп или q-learning, сложнее - approximate q-learning или DQN. Теперь же предлагается новый подход. Запараметризуем политику π таким образом, чтобы при обучении найти оптимальную оценку её параметра (да, параметризация политики уже была в методе кросс-энтропии, но тогда мы обновлялись лишь раз в сессию, а не на каждом шаге, и не использовали информацию среды).

value based: обучаем функции $Q_\theta(s, a)$ или $V_\theta(s)$, затем вычисляем политику $\pi(s) = \arg\max_a Q_\theta(s, a)$.

policy based: обучаться будет сама политика, а не какая-то вспомогательная функция и $a = \pi_\theta(s)$.

Опять же, в одношаговой модели будем максимизировать среднюю награду $R(s, a)$ за шаг:

$$J = \mathop{\mathbb{E}}_{\substack{s\sim p(s)\\ a\sim\pi_\theta(s|a)}} R(s,a) = \int_s p(s) \int_a \pi_\theta(a|s) R(s,a)\ da\ ds \approx \frac{1}{N} \sum_{i=0}^{N} \sum_{s,a} R(s,a)$$

Возникает следующий вопрос: как вычислять $\frac{dJ}{d\theta}$, если приближённое значение J не зависит от θ. Т.е. мы насемплировали сессии, используя политику π_θ, получили награду, а теперь хотим дифференцировать по какому-то усреднённому набору чисел. Вариант решения - считать численную производную:

$$\nabla J = \frac{J_{\theta+\epsilon} - J_\theta}{\epsilon}$$

Но это плохая идея, которая требует много семплов и вычислений. Второй способ - дифференцировать по θ выражение с интегралом. Но тогда результат перестанет быть мат. ожиданием, и мы не сможем приблизить его средним. Чтобы исправить это, сделаем несколько простых преобразований:

$$\nabla \log \pi(z) = \frac{1}{\pi(z)} \nabla \pi(z)$$

$$\pi(z) \cdot \nabla \log \pi(z) = \nabla \pi(z)$$

$$\nabla J = \int_s p(s) \int_a \nabla \pi_\theta(a|s) R(s,a) \; da \; ds$$

$$\nabla J = \int_s p(s) \int_a \pi_\theta(a|s) \nabla \log \pi_\theta(a|s) R(s,a) \; da \; ds$$

$$\nabla J = \mathop{\mathbb{E}}_{\substack{s \sim p(s) \\ a \sim \pi_\theta(s|a)}} \left[\nabla \log \pi_\theta(a|s) \cdot R(s,a) \right]$$

Поскольку политика π - это некоторая модель, предсказывающая действие (например, нейронная сеть), то мы можем дифференцировать её логарифм по параметрам, а для вычисления ∇J использовать усреднение по семплам:

$$\nabla J \approx \frac{1}{N} \sum_{i=0}^{N} \sum_{s,a} \nabla \log \pi_\theta(a|s) \cdot R(s,a)$$

И модель будет обновляться следующим образом:

$$\theta_{i+1} = \theta_i + \alpha \cdot \nabla J$$

Выше мы рассматривали одношаговую модель (т.е. стояла задача максимизации награды на каждом шаге, а не в долгосрочной перспективе). В целом вместо $R(s,a)$ можно подставить Q-функцию и использовать для её оценки кумулятивную награду: $Q_\pi(s_t, a_t) = \mathbb{E}_s G(s_t, a_t)$, где $G_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + ...$ Внимательный читатель заметит, что в таком случае нарушается дифференцирование под знаком интеграла. Действительно, $\nabla[\pi_\theta(a|s) \cdot Q(s,a)] = \nabla \pi_\theta(a|s) \cdot Q(s,a) + \pi_\theta(a|s) \cdot \nabla Q(s,a) \neq \nabla \pi_\theta(a|s) \cdot Q(s,a)$, поскольку Q-функция зависит от политики π, а значит и от параметра θ. На самом деле это всё работает, только на доказательство нужно очень много выкладок, которые при желании можно прочитать здесь :)

Baseline idea. Если оставить формулу с J в том виде, в котором она есть сейчас, то тогда мы будем максимизировать награду Q, поощряя агента выполнять только хорошие действия. На деле в процессе обучения нам важнее изучать среду, в том числе совершая плохие действия (зато агент приобретёт опыт, как вести себя в таких ситуациях). Чтобы исправить это, вычтем из Q-функции бейзлайн $b(s)$, который показывает полезность состояния s в целом:

$$\nabla J = \mathop{\mathbb{E}}_{\substack{s\sim p(s)\\ a\sim\pi_\theta(a|s)}} \nabla \log \pi_\theta(a|s)\big(Q(s,a) - b(s)\big) =$$

$$= \mathop{\mathbb{E}}_{\substack{s\sim p(s)\\ a\sim\pi_\theta(a|s)}} \nabla \log \pi_\theta(a|s)\ Q(s,a) - \mathop{\mathbb{E}}_{\substack{s\sim p(s)\\ a\sim\pi_\theta(a|s)}} \nabla \log \pi_\theta(a|s)\ b(s)$$

Вообще говоря, $\mathop{\mathbb{E}}_{s,a} \nabla \log \pi_\theta(a|s)\ b(s) = \mathop{\mathbb{E}}_{s} b(s)\ \mathop{\mathbb{E}}_{a} \nabla \log \pi_\theta(a|s) = 0$, поскольку:

$$\mathop{\mathbb{E}}_{a\sim\pi_\theta(a|s)} \nabla \log \pi_\theta(a|s) = \int_a \pi_\theta(a|s) \nabla \log \pi_\theta(a|s)\ da =$$
$$= \int_a \nabla \pi_\theta(a|s)\ da = \frac{\partial}{\partial\theta} \int_a \pi_\theta(a|s) da = \frac{\partial}{\partial\theta} 1 = 0$$

Тем самым вычитание $b(s)$ из Q-функции не меняет ∇J. При этом разброс по s и a становится меньше:

$$Var\big[Q(s,a) - b(s)\big] = Var\big[Q(s,a)\big] - 2 \cdot Cov\big[Q(s,a), b(s)\big] + Var\big[b(s)\big] < Var\big[Q(s,a)\big]$$

т.к. $Var\big[b(s)\big] = 0$, а $Cov\big[Q(s,a), b(s)\big] > 0$, в случае если $b(s)$ и $Q(s,a)$ скоррелированы.

Advantage actor-critic. Обычно в качестве бейзлайна используется среднее по значениям Q. Более продвинутый вариант - $b(s) = V(s)$. Величина $A(s,a) = Q(s,a) - V(s)$ называется *преимуществом*:

$$Q(s,a) = r + \gamma \cdot V(s')$$

$$A(s,a) = r + \gamma \cdot V(s') - V(s)$$

$$\nabla J_{actor} \approx \frac{1}{N} \sum_{i=0}^{N} \sum_{s,a} \nabla \log \pi_\theta(a|s) \cdot A(s,a)$$

Value-функцию V_θ можно предсказывать вместе с политикой π_θ (т.к. нам больше неоткуда её взять). Тогда помимо максимизации ∇J_{actor} будем минимизировать $L_{critic} \approx \frac{1}{N} \sum_{i=0}^{N} \sum_{s,a} \big(V_\theta(s) - [r + \gamma V(s')]\big)^2$.

2.6 Policy gradient applications outside RL

В этом параграфе мы рассмотрим прикладные способы использования RL, а именно на seq2seq задачах.

В части, посвящённой NLP, мы занимались машинным переводом, в качестве оптимизационной величины выступала кросс-энтропия. На деле же бывает так, что хорошо сгенерированный перевод имеет большой лосс, так как он сильно отличается от оригинала. Разберём, какие ещё проблемы встречаются в задаче перевода:

- Обычно мы используем датасеты, в которых train и test принадлежат одному распределению. На практике случается так, что данные, которые приходят на рабочу модель имеют совершенно иное распределение. Из-за этого модель может порождать нерелевантные ответы. Т.е. для seq2seq задач, если однажды был сгенерирован некорректный токен, то дальшейшая генерация текста осложняется тем, что модель будет обуславливаться на предыдущий неправильный контекст.
- Ещё одна проблема seq2seq задач - необходимость большого количества чистых данных. Если обучаемая модель имеет 400 миллионов параметров, а размер датасета - 40 тысяч, то скорее всего произойдёт переобучение (а обучать диалоговую систему на постах из твиттера - не лучшая идея).

Эти проблемы можно решить, используя reinforcement learning:

- Модель учиться максимизировать награду, и поэтому предыдущие токены уже не так сильно влияют на следующие.
- Нам не нужен большой чистый датасет, нужна лишь функция награды (для задач перевода можно использовать метрики качества: bleu, meteor и другие).

Основной недостаток rl - проблема холодного старта. Агенту тяжело обучаться с нуля, когда пространство ответов огромно и непонятно, в каком направлении нужно двигаться, чтобы получить большую награду. Поэтому предлагается взять лучшее из supervised и reinforcement learning: изначально модель учится минимизировать лосс, а затем она становится агентом и дообучается максимизировать награду.

Если вспомнить policy gradient метод, то там мы использовали value-функцию в качестве бейзлайна, чтобы обучаться быстрее. В задаче перевода можно взять для $V(s)$ текст, сгенерированный в жадном режиме, а для $R(s, a)$ - в обычном: $A(s, a) = R(s, a) - R(s, a_{inference}(s))$.

3 Computer Vision

3.1 CV problem statements, metrics in CV

Небольшой обзор задач компьютерного зрения:

1. Классификация: на вход подаётся изображение, необходимо определить его класс.
2. Локализация: хотим уточнить расположение объекта при помощи некоторого *bounding box*, т.е. предсказать дополнительно координаты ограничивающего прямоугольника.
3. Детекция: более сложная задача локализации, когда объектов на изображении несколько.
4. Сегментация: для каждого пикселя на картинке требуется указать его класс.

Intersection over Union. Перейдём к рассмотрению метрик качества. Для двух областей, предсказанной и истинной, IoU есть отношение пересечения к их объединению:

$$\text{IoU} = \frac{\text{area of overlap}}{\text{area of union}}$$

Mean Average Precision. Чтобы оценить качество не только по одному предсказанию, но и в целом по всем найденным областям, используется mAP. Для начала разберёмся, что есть Average Precision. Допустим, у нас есть некоторый объект и по нему дано десять предсказаний. Отсортируем предсказания по уровню уверенности классификатора:

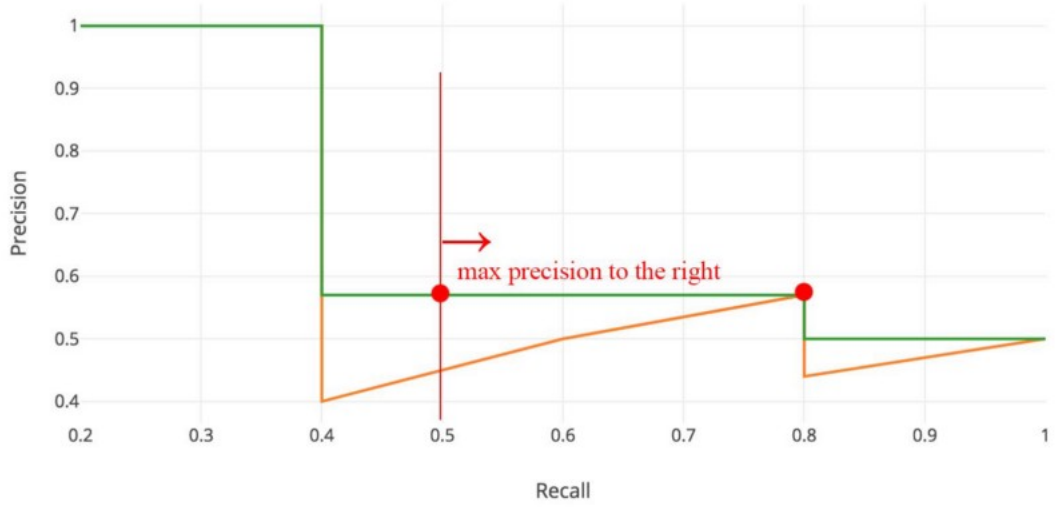
Rank	Correct?	Precision	Recall
1	True	1.0	0.2
2	True	1.0	0.4
3	False	0.67	0.4
4	False	0.5	0.4
5	False	0.4	0.4
6	True	0.5	0.6
7	True	0.57	0.8
8	False	0.5	0.8
9	False	0.44	0.8
10	True	0.5	1.0

Далее для каждого предсказания проверяется его корректность, т.е. рассматривается IoU и если это значение выше некоторого порогового, то предсказание верно. В конце считаем precision и recall, идя от наиболее вероятной детекции к наименее вероятной.

Теперь нужно оценить среднюю точность. Основное определение AP - это площадь под precision-recall графиком (добавить построенному по двум крайним столбцам?):

$$\text{AP} = \int_0^1 p(r)dr$$

Однако на деле, чтобы небольшие изменения в рейтинге несильно меняли AP, кривая сглаживается и для каждого значения recall соответствующее значение precision заменяется на максимум справа:



Average precision считается отдельно по каждому классу, затем результат усредняется и тем самым мы получаем mean Average Precision (можно ещё написать про недостатки этой метрики).

3.2 R-CNN, Fast and Faster structure, main ideas

Мы уже умеем классифицировать изображения при помощи свёрточных сетей. Однако проблема детекции в том, что количество интересующих нас объектов на картинке не фиксировано и мы не знаем, где они находятся. Наиболее очевидное решение - взять произвольный bounding box и решить для него задачу классификации. Но тогда придётся перебрать огромное количество ограничивающих блоков,

чтобы попасть хотя бы в какой-то объект. К счастью, существуют методы, выделяющие содержательные области, и, таким образом, задача классификации упрощается. В этом и есть суть R-CNN.

Region-CNN. Используя выборочный поиск, из изображения извлекается примерно 2000 регионов, которые затем передаются свёрточной сети. Данная модель обладает следующими недостатками:

- алгоритм поиска фиксирован, что может привести к появлению плохих регионов-кандидатов;
- долгое время обучения, так как все блоки прогоняются отдельно (даже если они пересекаются);
- обучается только SVM, на котором построена классификация, без обновления весов CNN.

Fast R-CNN. Недостатки предыдущей модели привели к созданию улучшенной версии. Сначала всё изображение прогоняется через свёрточную сеть для извлечения признаков, на основе которых будет происходить определение регионов с объектами. Затем полносвязные слои, а не SVM, предсказывают класс и поправки к bounding box. Это намного быстрее (мы прогоняем только одну картинку через CNN, а не тысячи блоков), и весь процесс обучения построен на нейронных сетях.

Faster R-CNN. Вышеописанные методы используют выборочный поиск для определения регионов с объектами. Но это довольно медленный процесс, который нужно оптимизировать. Добавим заранее фиксированные bounding box'ы с различыми размерами и соотношением сторон и новую сеть, предсказывающую для каждого пикселя на картинке, является ли он центром какой-либо области, ограниченной одним из блоков, или нет. С этим улучшением модель быстрее справляется с локализацией.

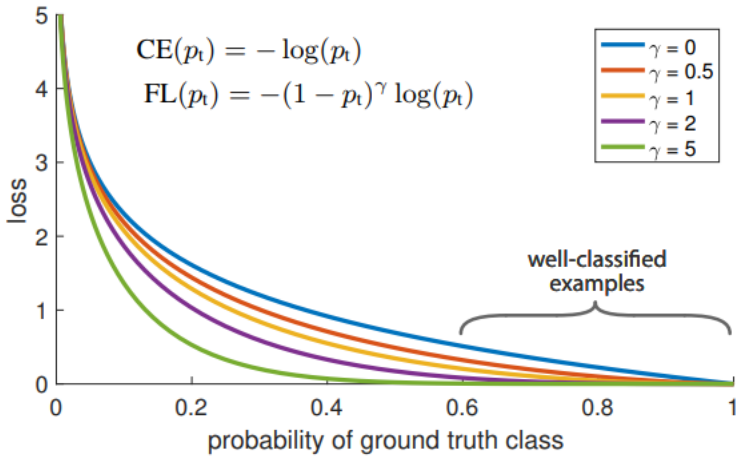
3.3 Focal Loss, Non-Maximum Suppression

Focal Loss. Обычно для классификации используется кросс-энтропия. В таком случае хорошие предсказания (с вероятностью, к примеру, выше 0.6) будут всё ещё иметь большой вклад в ошибку модели. Но хотелось бы исправлять модель на плохих ответах. Так была предложена новая функция потерь:

$$\mathrm{CE}(p_t) = -\log(p_t)$$

$$\mathrm{FL}(p_t) = -(1-p_t)^{\gamma}\log(p_t)$$

т.е. при $\gamma = 0$ имеем обычную кросс-энтропию (на картинке снизу CE - это верхний график).



Non-Maximum Suppression algorithm. В случае, когда претендентов на роль bounding box много, необходимо выбрать, какое именно окно станет искомой детекцией. Отсортируем кандидатов по уровню уверенности в предсказании. Затем по очереди будем смотреть каждого следующего кандидата с вышестоящим, и если они имеют высокое значение IoU, то окно с меньшим уровнем уверенности выкидывается. Такой алгоритм позволяет нам избавиться от плохих детекций.

3.4 YOLO v1, v2, v3 structure, main ideas

You Only Look Once. Вместо того, чтобы выделять bounding box, затем использовать свёртки над ними, а потом предсказывать класс объекта, попробуем сразу заниматься детекцией. В YOLO единственная сверточная сеть предсказывает ограничивающие прямоугольники и вероятности классов для них. Алгоритм работы следующий:

1. Изображение делится на части по сетке (в йоло это 7x7).
2. Для каждой ячейки генерируются bounding box'ы с уровнями доверия.
3. Другая часть йоло предсказывает для каждой ячейки вероятность встретить некоторый класс при условии, что здесь будет bounding box, причём делается это по изначальному делению картинки.
4. Каждому bounding box присваивается метка класса в соответствие с классом ячейки.
5. Применяется Non-Maximum Suppression, и так мы получаем итоговые детекции.

К сожалению мне не хватило сил дописать эту лекцию и лекцию по сегментации :((

3.5 KL divergence. Relation to cross entropy

Пусть имеются распределения $Q(z)$ и $P(z)$ и мы хотим найти меру удалённости между ними. Определим расстояние Кульбака-Лейблера как:

$$D[Q(z), P(z)] = \mathop{\mathbb{E}}_{z\sim Q}[\log Q(z) - \log P(z)] = \int_q q \log q \ dq - \int_q q \log p \ dq$$

Здесь первое слагаемое - это энтропия распределения Q, а второе - его кросс-энтропия с P. Заметим, что KL divergence не является стандартной метрикой, поскольку не выполняется свойство симметричности. Преобразуем расстояние между $Q(z)$ и $P(z|X)$, используя теорему Байеса:

$$D[Q(z), P(z|X)] = \mathop{\mathbb{E}}_{z\sim Q}[\log Q(z) - \log P(X|z) - \log P(z)] + \log P(X)$$

$$\log P(X) - D[Q(z), P(z|X)] = \mathop{\mathbb{E}}_{z\sim Q}[\log P(X|z)] - D[Q(z), P(z)]$$

$$\log P(X) - D[Q(z|X), P(z|X)] = \mathop{\mathbb{E}}_{z\sim Q}[\log P(X|z)] - D[Q(z|X), P(z)]$$

Это выражение пригодится нам далее в разделе про VAE.

3.6 Variational Autoencoders: structure, loss function, training process

Перейдём к рассмотрению автоэнкодеров. Краткая суть их работы состоит в том, что берутся некоторые данные, которые сначала сжимаются в латентное пространство при помощи энкодера, а затем опять разжимаются декодером. Это нужно, чтобы:

1. Получить признаковое пространство меньшей размерности.
2. При помощи свёрточных слоёв сгенерировать информативное описание данных.

Поскольку на входе и на выходе модели стоит один и тот же вектор X, то для обучения не нужна размеченная выборка, т.е. формально:

$z = E(x, \theta_E)$ - результат применения энкодера E к x.

$\hat{x} = D(z, \theta_D)$ - результат применения декодера D к z.

$[\theta_E, \theta_D] = \arg\min L(\hat{x}, x)$ - оптимальные параметры преобразований.

В идеале хочется, чтобы данные из одного класса находились внутри свяного множества, что позволит нам получать похожие объекты при непрерывном изменении параметров в признаковом пространстве. Для этого можно дополнительно подавать энкодеру и декодеру лейблы, которые станут источником разрывности между объектами различных классов.

Далее пусть есть некоторая точка x в латентном пространстве. Мы знаем, какому классу она принадлежит, но хотелось бы ещё понимать, в какой окресности x лежат объекты того же класса. Предполагая, что распределение объектов в пространстве нормальное, нужно предсказать дополнительно среднее и дисперсию. Тем самым мы подобрались к VAE, где вместо позиции объекта в латентном пространстве энкодер выдаёт распределение, согласно которому затем генерируется выборка для декодера:

$X \to \text{encoder } Q \to \mu(X), \Sigma(X) \to \text{sample } z \text{ from } N(\mu(X), \Sigma(X)) \to \text{decoder } P \to f(z) \to ||X - f(z)||^2$

Однако при пробрасывании градиентов назад мы сломаемся на шаге генерации выборки из нормального распределения, поскольку оптимизируются числовые параметры, а не случайные величины. Тогда решение - будем генерировать выборку из $N(0, 1)$, а затем домнажать её на дисперсию и прибавлять среднее. Именно эти параметры и будут обновляться.

Как уже отмечалось выше, мы предположили, что распределение объектов в латентом пространстве является нормальным. Но это неочевидное условие, которое само по себе выполняться не будет.

Вспомним расстояние Кульбака-Лейблера: если взять $Q(z|X)$ в качестве распределения, которое имеют случайные величины в пространстве признаков на данным момент, а $P(z)$ в качестве желаемого распределения, получим, что нужно уменьшать $D[Q(z|X), P(z)]$. Теперь взглянем на последнее выражение в предыдущем параграфе. $P(X)$ - это по сути вероятность конкретного изображения быть нарисованным. То есть мы хоти максимизировать эту вероятность, а значит увеличивать левую часть (здесь второе слагаемое - это некоторая ошибка, которая будет почти нулевой). Правая часть - это то, что мы можем максимизировать. Более того, в ней содержутся KL дивергенция и кросс-энтропия, и то и другое со знаком минус. Тогда уменьшая их, мы будем максимизировать вероятность хорошего изображения.

3.7 Generative Adversarial Networks: structure, loss function, training process

Недостаток автоэнкодеров заключается в том, что мы оптимизируем некоторую метрику, а это не всегда хороший способ оценить похожесть между реальными данными и сгенерированными. Рассмотрим теперь GAN'ы, в которых дополнительная сеть пытается отличить настоящие объекты от искусственных. Структура модели состоит в следующем:

1. Сэмплируется выборка $Z \sim P_z$, которую затем генератор преобразует в объекты $X_p \sim P_g$.
2. Дискриминатор получает настоящие данные X_s и объекты X_p, созданные генератором, и учится отличать одно от другого, то есть решает задачу бинарной классификации.

При этом обучение происходит в два этапа, так как дискриминатор хочет уменьшить ошибку классификации, а генератор - увеличить правдоподобие:

$$\text{d\_loss} = \log(D(X_s)) + \log(1 - D(G(Z)))$$

$$\text{g\_loss} = \log(1 - D(G(Z)))$$

Таким образом, задачу, которую решает GAN, можно записать как:

$$\min_G \max_D \mathbb{E}_{X \sim P}[\log(D(X))] + \mathbb{E}_{Z \sim P_z}[\log(1 - D(G(Z)))]$$

Спасибо замечательной серии статей на хабре, которая легла в основу последних двух параграфов!