ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ «САМАРСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ
ИМЕНИ АКАДЕМИКА С.П. КОРОЛЕВА

Лекции по курсу «Интеллектуальные системы»

Для студентов направления 090401

Очное обучение 2023-2024 учебный год

Лектор: Солдатова Ольга Петровна, к.т.н., доцент

Современные свёрточные сети.

Лекции 7-8

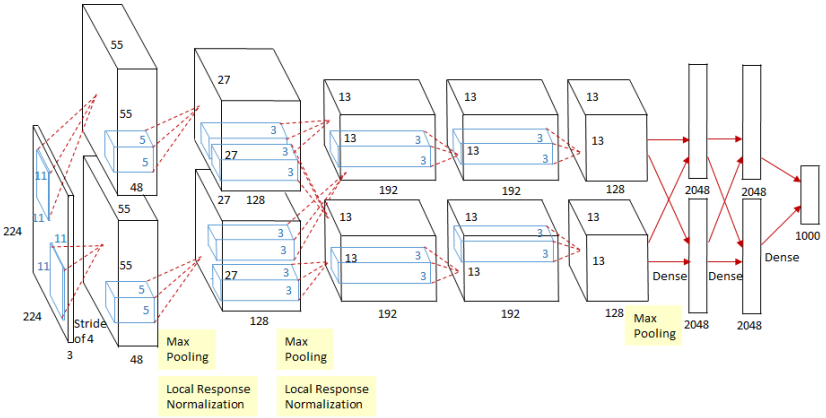
Современные свёрточные архитектуры.
Архитектура сети AlexNet.
Архитектура сети VGG.
Архитектура сети Inception.
Архитектура сети ResNet.
Рекуррентные нейронные сети.
Архитектура рекуррентных сетей.
Вентильные нейроны и вентильные сети.
Сети LSTM.
Архитектура сетей LSTM.
Порождающие сети.
Порождающие состязательные сети.
Схема работы генеративно-состязательной сети GAN.
Архитектуры, основанные на GAN.

Литература по курсу

1. Куприянов А.В. «Искусственный интеллект и машинное обучение» https://do.ssau.ru/moodle/course/view.php?id=1459
2. Столбов В.Ю. Интеллектуальные информационные системы управления предприятием / В.Ю. Столбов, А.В. Вожаков, С.А. Федосеев. – Москва : Издательство Литрес, 2021. – 470 с. – Режим доступа: по подписке. – URL: https://www.litres.ru/book/artem-viktorovich-vo/intellektualnye-informacionnye-sistemy-upravleniya-pr-66403444/.
3. Осовский С. Нейронные сети для обработки информации / Пер. с пол. И.Д. Рудинского. – М.: Финансы и статистика, 2002. – 344 с.: ил.
4. Горбаченко, В. И. Интеллектуальные системы: нечеткие системы и сети : учебное пособие для вузов / В. И. Горбаченко, Б. С. Ахметов, О. Ю. Кузнецова. — 2-е изд., испр. и доп. — Москва : Издательство Юрайт, 2018. — 105 с. — (Университеты России). — ISBN 978-5-534-08359-0. — Текст : электронный // ЭБС Юрайт [сайт]. — URL: https://urait.ru/bcode/424887 – Режим доступа: https://urait.ru/bcode/424887
5. 4. Гафаров Ф.М. Искусственные нейронные сети и приложения: учеб. пособие / Ф.М. Гафаров, А.Ф. Галимянов. – Казань: Изд-во Казан. ун-та, 2018. – 121 с. – Текст : электронный. – Режим доступа: https://repository.kpfu.ru/?p_id=187099
6. Хайкин С. Нейронные сети: Полный курс: Пер. с англ. - 2-е изд. – М.: Вильямс, 2006. – 1104 с.: ил.
7. Борисов В.В., Круглов В.В., Федулов А.С. Нечёткие модели и сети. – М.: Горячая линия–Телеком, 2007. -284 с.: ил.
8. Рутковская Д., Пилиньский М., Рутковский Л. Нейронные сети, генетические алгоритмы и нечёткие системы: Пер. с польск. И.Д.Рудинского, - М.: Горячая линия – Телеком, 2007. – 452 с. ил.
9. Николенко С., Кадурин А., Архангельская Е. Глубокое обучение. — СПб.: Питер, 2018. — 480 с.: ил. — (Серия ≪Библиотека программиста≫).
10. Гудфеллоу Я., Бенджио И., Курвилль А. Глубокое обучение / пер. с анг. А. А. Слинкина. – 2-е изд., испр. – М.: ДМК Пресс, 2018. – 652 с.: цв. ил.

База *ImageNet*. Эволюция СНС *AlexNet*

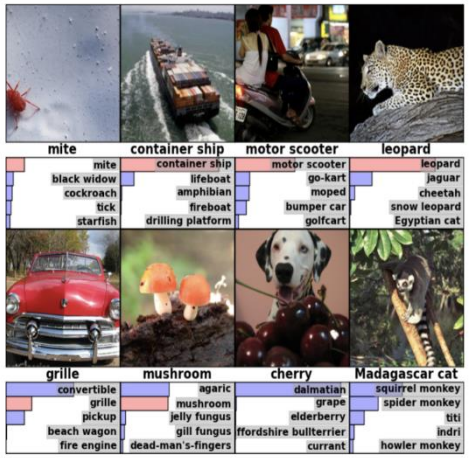
База данных **ImageNet** — проект по созданию и сопровождению массивной базы данных аннотированных изображений (4,197,122 images, 21841 synsets).
В аннотациях перечислены объекты, попавшие на изображение, и прямоугольники с их координатами.
AlexNet — свёрточная нейронная сеть, которая оказала большое влияние на развитие машинного обучения, в особенности — на алгоритмы компьютерного зрения. Сеть с большим отрывом выиграла конкурс по распознаванию изображений ImageNet LSVRC-2012.

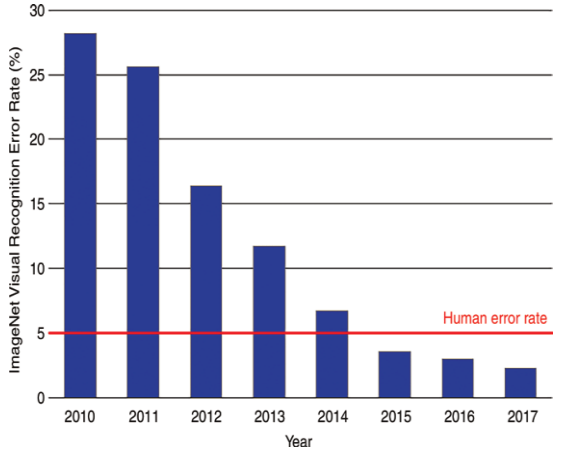


Особенности AlexNet

1. Как функция активации используется **Relu** вместо арктангенса для добавления в модель нелинейности. За счет этого при одинаковой точности метода скорость становится в 6 раз быстрее.

2.Использование дропаута вместо регуляризации решает проблему переобучения. Однако время обучения удваивается с показателем дропаута 0,5.



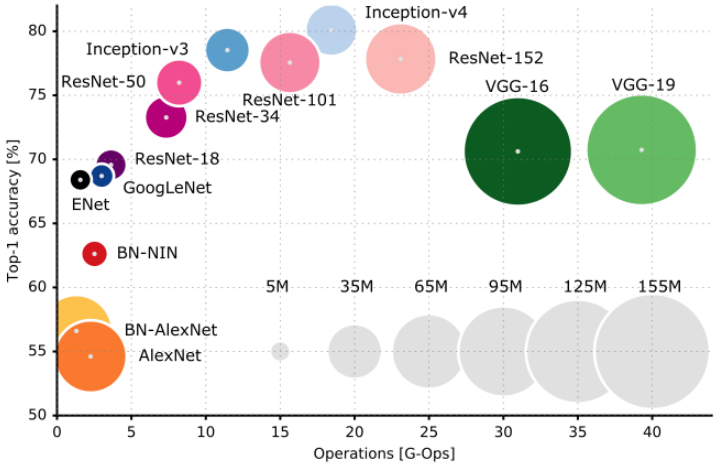


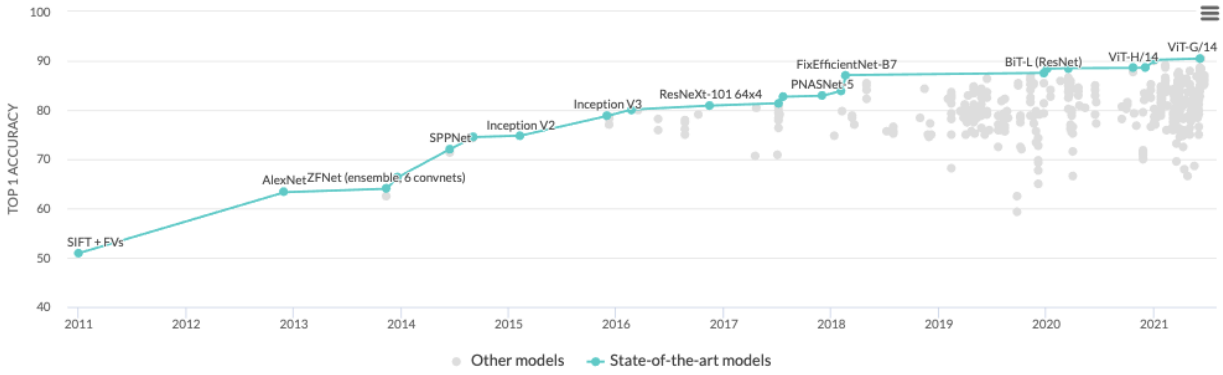
Эволюция СНС

ImageNet Large Scale Visual Recognition Challenge

С 2010 года ведётся проект ILSVRC (англ. *ImageNet Large Scale Visual Recognition Challenge* — Кампания по широкомасштабному распознаванию образов в ImageNet), в рамках которого различные программные продукты ежегодно соревнуются в классификации и распознавании объектов и сцен в базе данных ImageNet.

В 2014 г., через год после публикации AlexNet все участники конкурса ImageNet стали использовать сверточные нейронные сети для решения задачи классификации. AlexNet была **первой** реализацией сверточных нейронных сетей и открыла новую эру исследований.





Современные свёрточные архитектуры

Классификация изображений это базовая задача обработки изображений, наиболее изученная, с наибольшим числом разнообразных нейронных сетей, которые ее решают. Многие другие задачи обработки изображений используют для своего решения части нейронных сетей полученные именно для классификации.

Число различных архитектур для классификации изображений составляет десятки, а возможно и сотни. Познакомимся только с базовыми подходами к созданию таких сетей.

Простые, но достаточно эффективные сети - это слоистые сети с последовательным подключением слоев. Первые слои таких сетей это сверточные слои, последние - полносвязные, причем последний слой имеет столько нейронов, сколько классов распознается и, как правило, сопровождается функцией активации softmax. Представители: **LeNet, AlexNet**.

Более сложные сети, придуманные для упрощения построения глубоких сетей, это блочные сети, в которых используются последовательно подключенные блоки слоев. Сами блоки регулярны, строятся по одному принципу, например чередованием сверточных слоев, активации и субдискретизации. Блочность упрощает разработку сетей и их описание. Представитель: **VGG.**

Также разработали разветвленные архитектуры, в которых к одному входу применяется одновременно и параллельно несколько разных сверток, результаты которых затем объединяются. Так получились блочно-последовательные сети с разветвленными блоками. Представители **NiN, GoogleNet**.

Современные свёрточные архитектуры

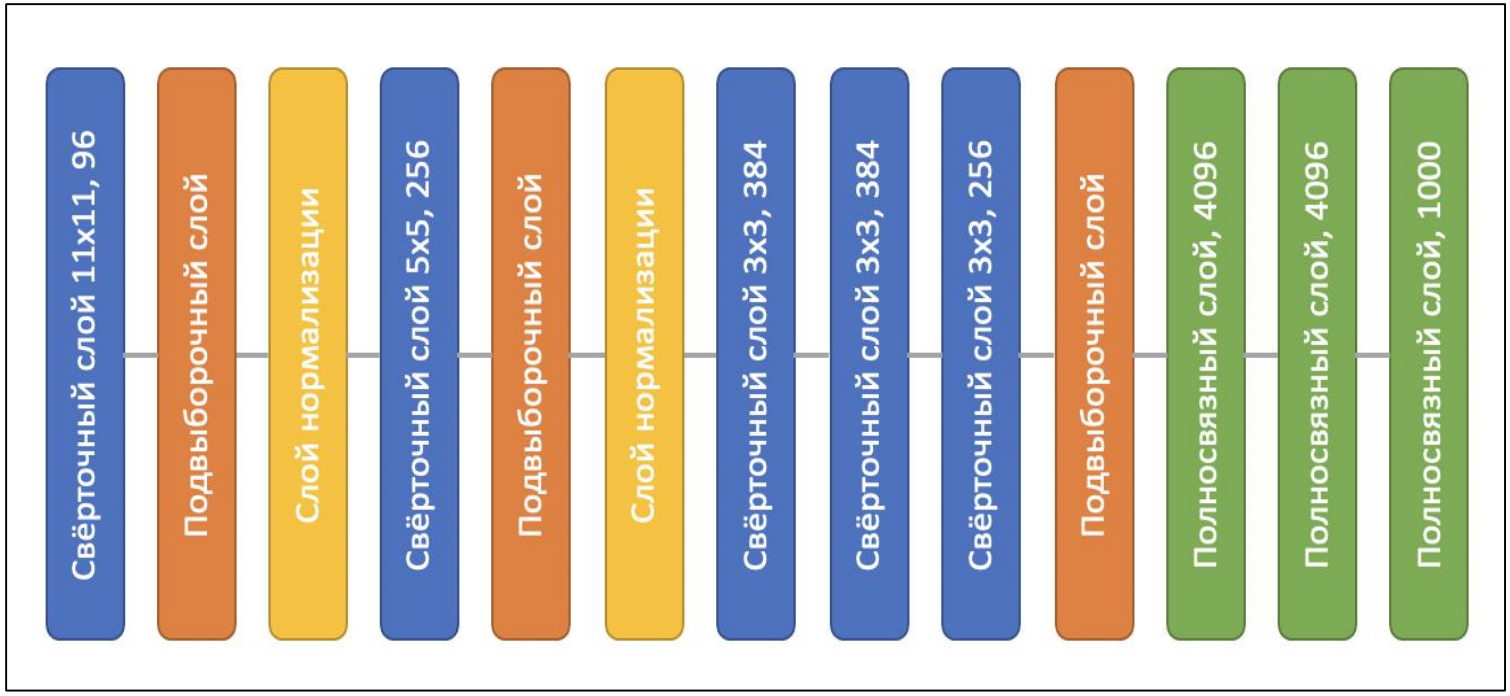
В сетях с последовательным подключением слоев для классификации изображений наблюдается иерархия слоев. Слои с небольшим номером реагируют (фильтруют) простые формы - кусочки линий, цветные пятна, текстуры. Слои же с большим номером реагируют уже на более абстрактные формы, например что-то похожее на лицо человека (если обучалось на изображениях лиц).

Часто бывает так, что для классификации важны не только абстрактные признаки, но и детальные. В этом случае в последовательные сети добавляют связи, которые минуют некоторые слои, передавая детальную информацию на более дальние слои. Представители: **ResNet, DenseNet**.

Сеть AlexNet в свое время победила в соревновании по классификации изображений "ImageNet Large Scale Visual Recognition Challenge 2012" и состоит из 8 сверточных слоев нейронов, двух полносвязных скрытых слоев по 4096 нейрона и полносвязного слоя для классификации с 1000 нейронами (по числу классов изображений на соревновании), а также слоев субдискретизации. Использовались функции активации ReLU, как более простые.

На рисунке 32 показана архитектура сети AlexNet.

Архитектура сети AlexNet (рисунок 32)



Современные свёрточные архитектуры

Одной из самых популярных глубоких свёрточных архитектур является модель, которую принято называть VGG. Название происходит от того, что эта модель была разработана в Оксфордском университете в группе визуальной геометрии (Visual Geometry Group), и их модели, представленные на ряд конкурсов по компьютерному зрению, выступали там под кодовым названием VGG. Соревнования проходили в 2014 году, что делает VGG самой «старой» из представленных моделей.

VGG — это на самом деле сразу две конфигурации свёрточных сетей, на 16 и 19 слоев. Основным нововведением, стала идея использовать фильтры размером 3x3 с единичным шагом свертки вместо использовавшихся в лучших моделях предыдущих лет свёрток с фильтрами 7x7 с шагом 2 и 11x11 с шагом 4. Причем это хорошо аргументированное предложение:

Во-первых, рецептивное поле трех подряд идущих свёрточных слоев размером 3x3 имеет размер 7x7, в то время как весов у них будет всего 27, против 49 в фильтре 7x7. Аналогично обстоит дело и с фильтрами 11x11. Это значит, что VGG может стать более глубокой, то есть содержать больше слоев, при этом одновременно *уменьшая* общее число весов. Для того, чтобы это было правдой, между соответствующими свёрточными слоями не должно быть слоев субдискретизации.

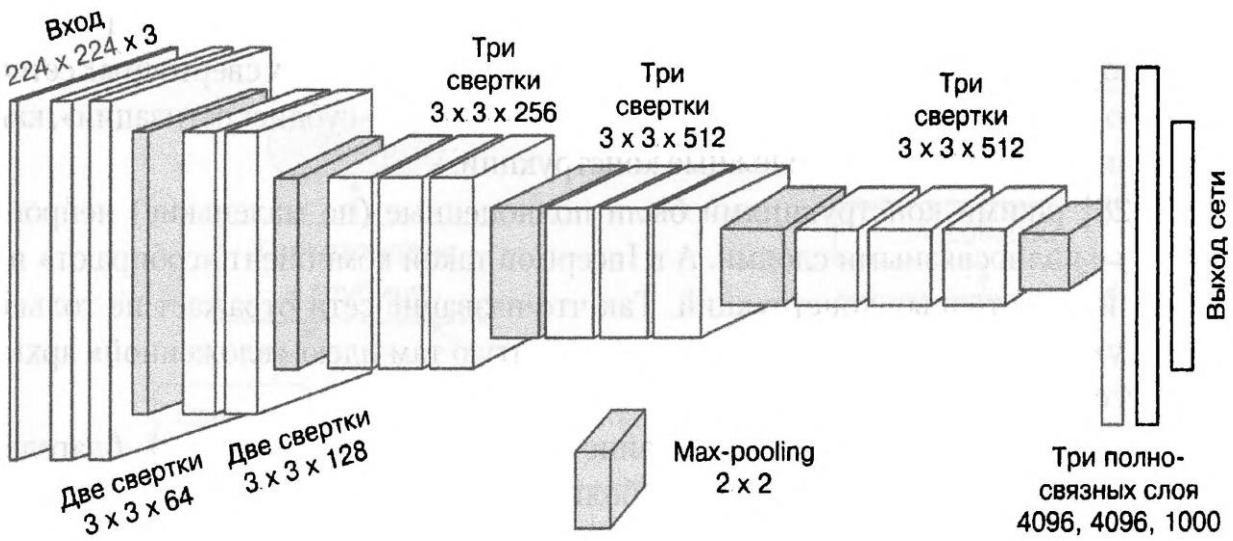
Современные свёрточные архитектуры

Во-вторых, наличие дополнительной нелинейности между слоями позволяет увеличить ≪разрешающую способность≫ по сравнению с единственным слоем с большей свёрткой. Этот аргумент можно использовать для того, чтобы ввести в сеть свёртки размером 1x1; такие слои тоже позволяют добавить дополнительную нелинейность в сеть, не меняя размер рецептивного поля. Полученная в итоге модель в 2014 году одержала победу в одной из номинаций известного соревнования по компьютерному зрению, ImageNet Large Scale Visual Recognition Competition (ILSVRC). А популяризация свёрточных слоев 3x3 привела, в частности, к тому, что NVIDIA в очередном релизе библиотеки cuDNN специально оптимизировала работу с такими свёртками.

Схема одной из VGG-сетей показана на рисунке 33. Пример другой архитектуры сети VGG16 представлен на рисунке 34.

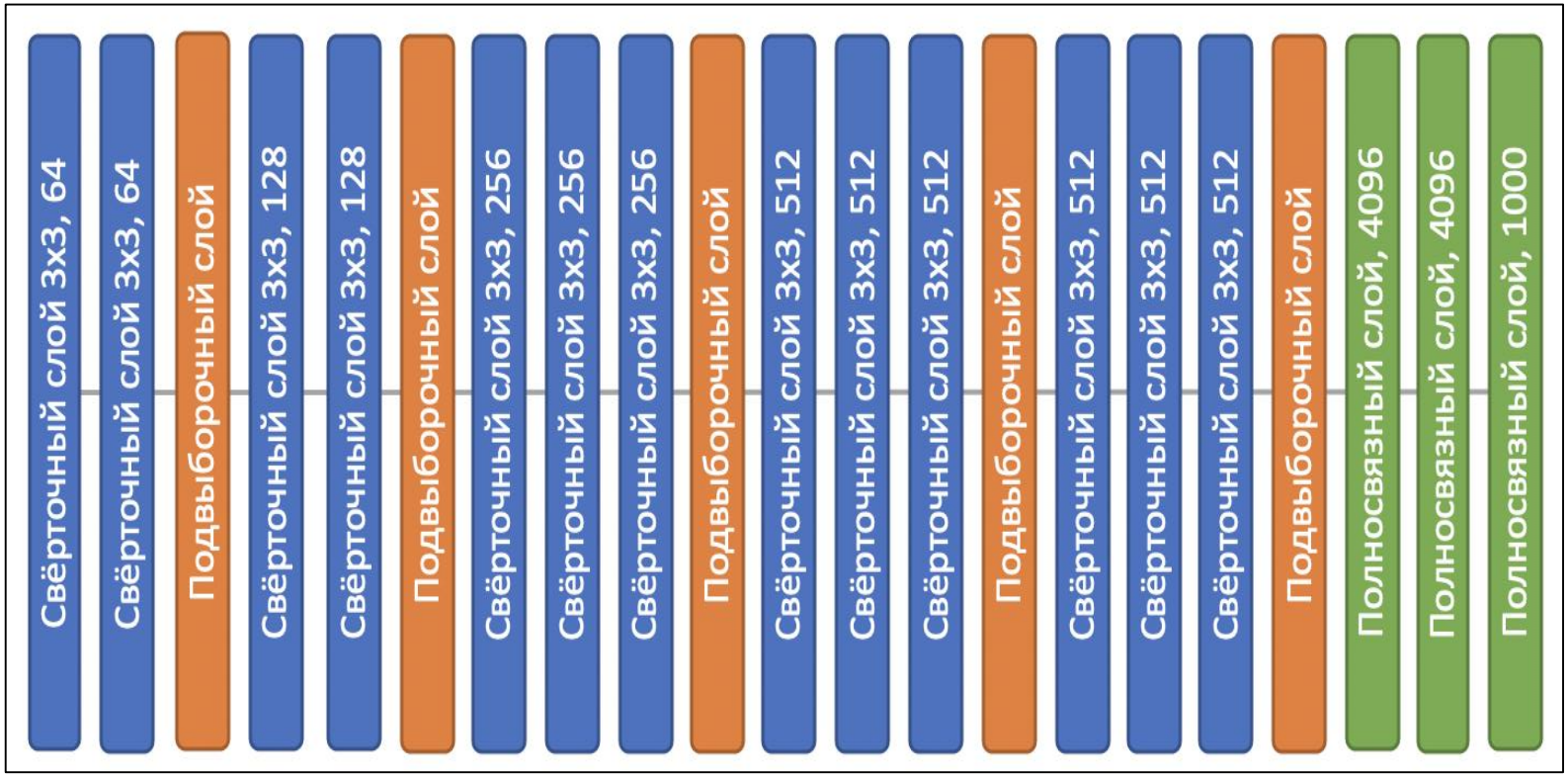
Следует обратить внимание на три вещи: во-первых, две-три свёртки 3x3 следуют друг за другом без субдискретизации; во-вторых, число карт признаков постепенно растет на более глубоких уровнях сети; в-третьих, полученные признаки окончательно ≪сплющиваются≫ в одномерный вектор и на нём работают последние, уже полносвязные слои. Всё это стандартные методы создания архитектур глубоких свёрточных сетей, и стоит следовать этим общим принципам.

Сеть VGG (рисунок 33)



Веса уже готовых моделей, обученных на больших наборах данных, например ImageNet, можно найти в Интернете. Это общее место для многих современных моделей компьютерного зрения: дело в том, что даже с современными мощностями датасеты настолько большие, а обучение настолько долгое и сложное (например, каждый вариант VGG в 2014 году обучали две-три недели на четырех лучших на тот момент видеокартах), что проще скачать готовые веса и, немного дообучить их для конкретной задачи.

Сеть VGG16 (рисунок 34)



Современные свёрточные архитектуры

Следующая важная свёрточная архитектура — это архитектура Inception.

Она была разработана в Google и появилась практически одновременно с VGG, в сентябре 2014 года; команда GoogLeNet победила с этой сетью в нескольких номинациях все того же конкурса ILSVRC-2014. Авторы предложли использовать в качестве строительных блоков для глубоких свёрточных сетей не просто последовательность ≪свертка — нелинейность — субдискретизация≫, как это обычно делается, а более сложные конструкции.

В Inception такую конструкцию ≪собирают≫ из небольших свёрточных блоков, что позволило реализовать большую глубину — 22 слоя без учета субдискретизации, при этом у Inception меньше параметров, чем у VGG. Блоками Inception являются модули, комбинирующие свёртки размером 1x1, 3x3 и 5x5, а также max-pooling субдискретизацию. Каждый блок представляет собой объединение четырех ≪маленьких≫ сетей, выходы которых объединяются в выходные каналы и передаются на следующий слой. Выбор набора свёрток и субдискретизации обусловлен скорее удобством, чем необходимостью. Команда GoogLeNet, разработавшая Inception, во второй версии своей модели тоже пересмотрела архитектуру этих модулей. Одно из ключевых нововведений — использование свёрточных слоев 1x1 не столько в качестве дополнительной нелинейности, сколько для понижения размерности между слоями. Свертки 3x3 и тем более 5x5 между слоями с большим числом каналов (а в Inception-модулях каналов может быть вплоть до 1024), оказываются крайне ресурсоемкими, несмотря на малые размеры отдельно взятых фильтров. А фильтры 1x1 могут помочь сократить число каналов, прежде чем подавать их на фильтры большего размера.

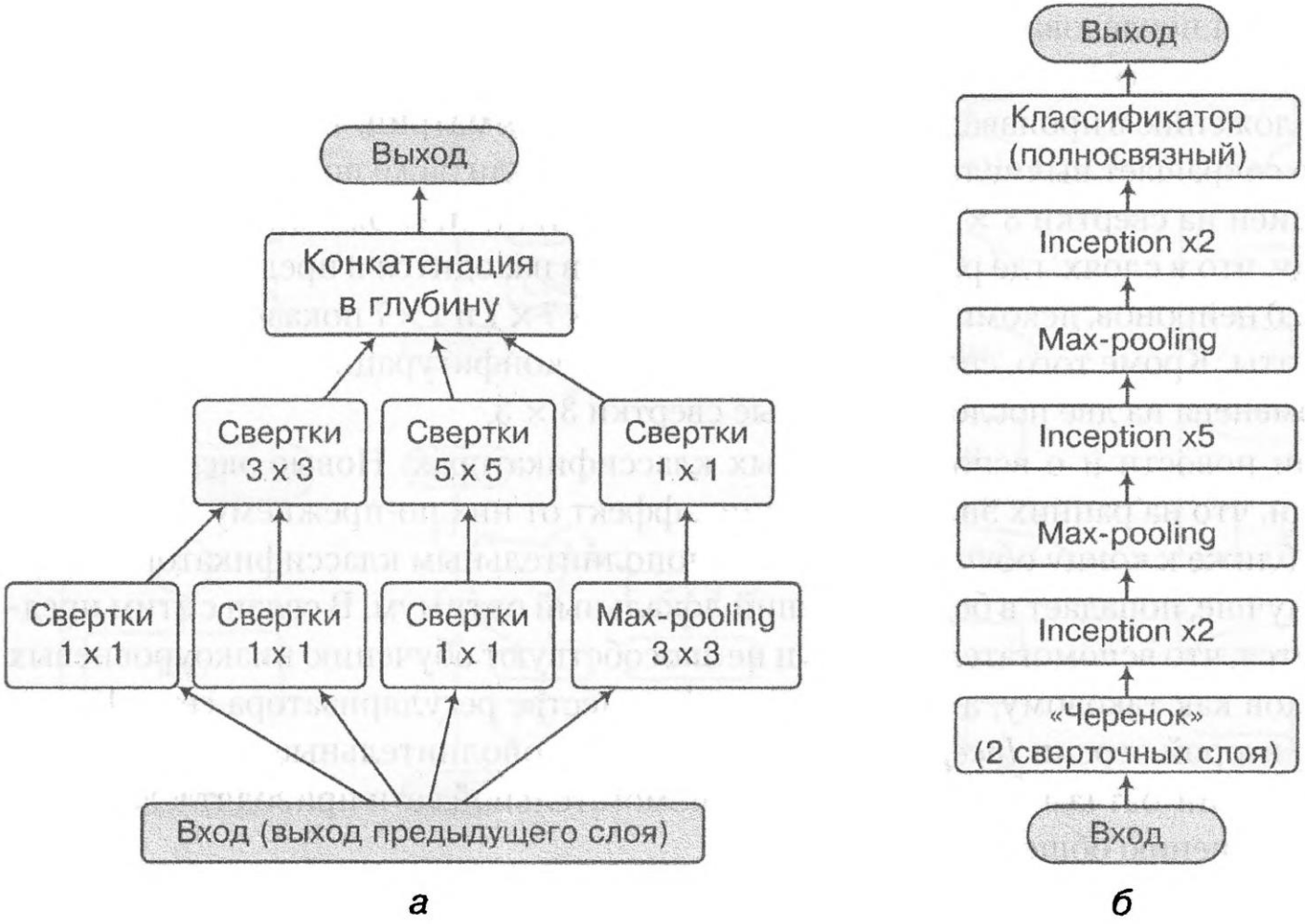
Современные свёрточные архитектуры

Эта идея отражена на рис. 35 *а*, на котором показана структура одного блока. А на рис. 35 *б* представлена общая и высокоуровневая схема всей сети: она начинается с двух «обычных» свёрточных фрагментов, а затем идут 11 Inception-модулей, дважды перемежаемых субдискретизацией, которая понижает размерность; после этого сеть завершается традиционными полносвязными слоями, дающими уже собственно выход классификатора.

Помимо конфигурации Inception-модулей и понижения размерности с помощью сверток 1x1, представлена еще одна важная идея. С учетом достаточно глубокой архитектуры сети — а в общей сложности *GoogLeNet* содержит порядка 100 различных слоев с общей глубиной в 22 параметризованных слоя, или 27 слоев с учетом субдискретизаций — эффективное распространение градиентов по ней вызывает сомнения. Чтобы решить эту проблему, авторы предложили добавить вспомогательные классифицирующие сети поверх некоторых промежуточных слоев.

Иначе говоря, они добавили две новые небольшие полносвязные сети, делающие предсказания на основе промежуточных признаков, вывели из них ту же функцию ошибки классификации и обучили на той же задаче не только всю сеть, но и отдельно первую ее часть, а также первую и вторую. В архитектуре присутствуют две такие дополнительные сети, состоящие из субдискретизации усреднением, свертки 1x1, полносвязного слоя, дропаута и линейного слоя с softmax в качестве функции ошибки классификатора. При обучении модели ошибка от этих подсетей добавляется к общей функции ошибки с понижающим коэффициентом 0,3. Потенциально этот приём позволил улучшить окончательное решение.

Сеть Inception (рисунок 35)



Современные свёрточные архитектуры

Через год в общую структуру Inception-модулей внесли новое изменение: заменили практически все ≪большие≫ свёртки на композиции сверток размерности 3x3 и 1xn. Свёртку размера 5x5 можно заменить двумя последовательными свёрточными слоями, каждый размера 3x3, при этом не потеряв в выразительной силе и сократив общее число весов. Однако авторы второй версии Inception пошли дальше и предложили заменить свёртки произвольного размера nxn на два последовательных слоя размером nx1 и 1xn. Такой подход существенно сокращает вычислительную сложность модели даже по сравнению с факторизацией на свертки 3x3 . Экспериментируя с такими слоями, авторы пришли к выводу, что в слоях, где размер карты признаков находится в пределах от 12x12 до 20x20 нейронов, декомпозиция в пары свёрток 7x1 и 1x7 показывает хорошие результаты. Кроме того, свёртка 5x5 из базовой конфигурации Inception-модуля была заменена на две последовательные свертки 3x3.

Кроме того, во второй версии Inception выяснилось, что дополнительные слои нормализации по мини-батчам или дропаута во вспомогательной сети приводят к дальнейшему улучшению общего решения.

Как показала практика, глубокие архитектуры стало возможно обучать эффективно, однако те решения, к которым сходились нейронные сети большой глубины, часто оказывались хуже, чем у менее глубоких моделей. И эта ≪деградация≫ не была связана с переобучением, как можно было бы предположить.

Современные свёрточные архитектуры

Оказалось, что это более фундаментальная проблема: с добавлением новых слоев ошибка растет не только на тестовом, но и на обучающем множестве. Хотя, казалось бы, более «мелкая» сеть — это частный случай более глубокой: мы ведь могли бы обучить менее глубокую сеть, а затем ее веса использовать для инициализации более глубокой; дополнительные слои при этом можно просто инициализировать так, чтобы они копировали вход в выход.

Тогда ошибка более глубокой сети по определению не будет выше, чем ошибка ее подсети, а после обучения ожидается улучшение. Но эксперименты показывают, что этого не происходит.

Для решения проблемы деградации команда из Microsoft Research разработала новую идею: *глубокое остаточное обучение* (deep residual learning), которое легло в основу сети *ResNet*. В базовой структуре новой модели нет ничего нового: это слои, идущие последовательно друг за другом. Отдельные уровни - это просто свёрточные слои, обычно с дополнительной нормализацией по мини-батчам. Разница в том, что в остаточном блоке слой из нейронов можно «обойти»: есть специальная связь между выходом предыдущего слоя $x^{(k)}$ и входом следующего слоя $x^{(k+1)}$ которая идет напрямую.

Современные свёрточные архитектуры

Базовый слой нейронной сети на рисунке 36 *а* превращается в остаточный блок с обходным путем на рисунке 36 *б*.

Математически это реализуется достаточно просто: когда два пути, ≪сложный≫ и ≪обходной≫, сливаются обратно, их результаты складываются друг с другом. Остаточный блок выражает такую функцию:

$$y^{(k)} = F\left(x^{k}\right) + x^{(k)}$$

где $x^{(k)}$ — входной вектор слоя k, $F\left(x^{k}\right)$ — функция, которую вычисляет слой нейронов, $y^{(k)}$ — выход остаточного блока, который потом станет входом следующего слоя $x^{(k+1)}$.

Получается, что если блок в целом должен аппроксимировать функцию *H(x),* то это достигается тогда, когда *F(x)* аппроксимирует остаток (*residue*) *H(x)–x,* отсюда и название *остаточные сети* (*residual networks*). В остаточном блоке слой нейронов обучается воспроизводить *изменения* входных значений, необходимые для получения итоговой функции.

Во-первых, часто получается, что обучить ≪остаточную≫ функцию проще, чем исходную; например, тождественную функцию *h(x)=x*. Главная же причина состоит в том, что градиент во время обратного распространения может проходить через этот блок беспрепятственно, градиенты не будут затухать, ведь всегда есть возможность пропустить градиент напрямую:

Современные свёрточные архитектуры

$$\frac{\partial y^{(k)}}{\partial x^{(k)}} = 1 + \frac{\partial F(x^k)}{\partial x^{(k)}}.$$

Это значит, что даже насыщенный и полностью обученный слой F, производные которого близки к нулю, не помешает обучению. Примеры таких ≪обходных путей≫, которые использовались в разных вариантах сети *ResNet* и других исследованиях этой группы авторов, показаны на рисунке 36.

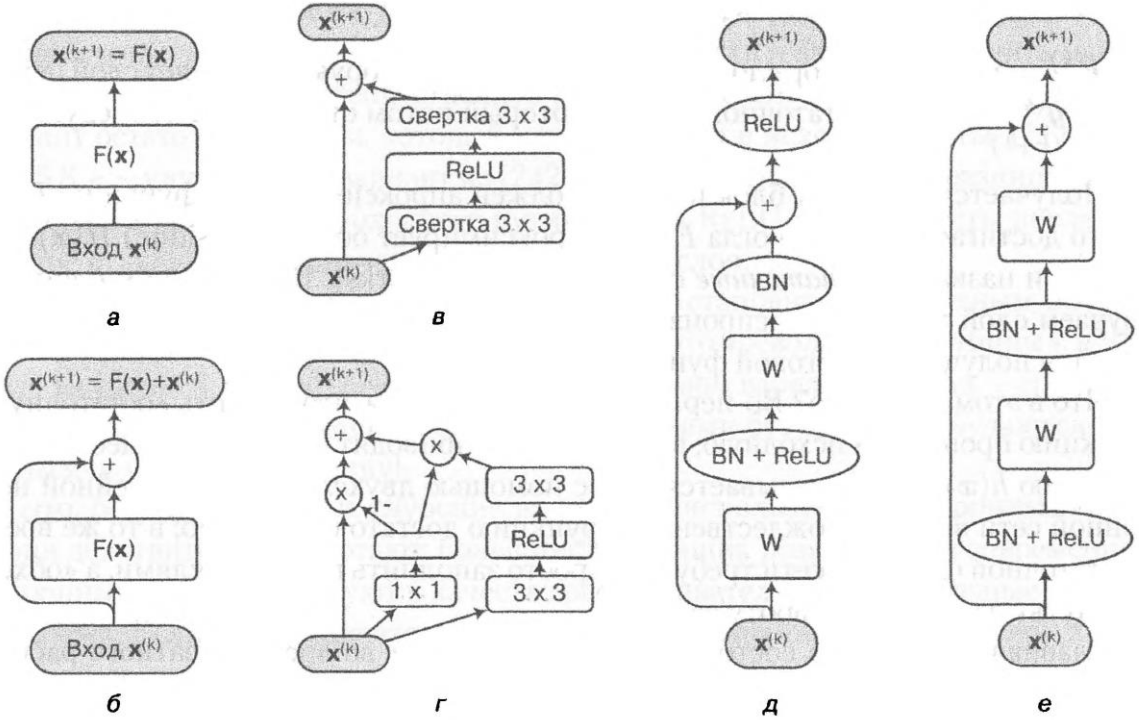
Для примера на рисунке 36 *в* изображён простой вариант остаточного блока, в котором $y^{(k)} = F(x^k) + x^{(k)}$, а на рисунке 36 *г* — вариант посложнее:

$$y^{(k)} = \left(1 - \sigma\left(f\left(x^{(k)}\right)\right)\right) x^{(k)} + \sigma\left(f\left(x^{(k)}\right)\right) F(x^k),$$

где f — другая функция входа, реализованная через свертки 1x1.

Это значит, что $F(x^k)$ и $x^{(k)}$ суммируются не с равными весами, а с весами, управляемыми дополнительным ≪гейтом≫. Достаточно просто обучить гейты так, чтобы веса были равными, то есть чтобы всегда выполнялось $f\left(x^{(k)}\right) = 0$. Простота в данном случае важнее выразительности и важно обеспечить максимально свободное и беспрепятственное течение градиентов.

Сеть ResNet (рисунок 36)



Блоки сетей для остаточного обучения: *а* — базовый блок; *б* — такой же блок с остаточной связью; *в* — простой блок с остаточной связью; *г* — блок с остаточной связью, контролирующийся гейтом; *д* — блок с остаточной связью исходной сети ResNet; *е* — более поздняя модификация: копирование входа в выход.

Современные свёрточные архитектуры

На рисунке 36 *д* показан вариант остаточного блока, который использовался в первоначальном варианте, а на рисунке 36 *е* — улучшенный вариант. Разница между ними в том, что из «обходного пути» убрана ReLU-нелинейность, последнее «препятствие» на пути значений с предыдущего слоя.

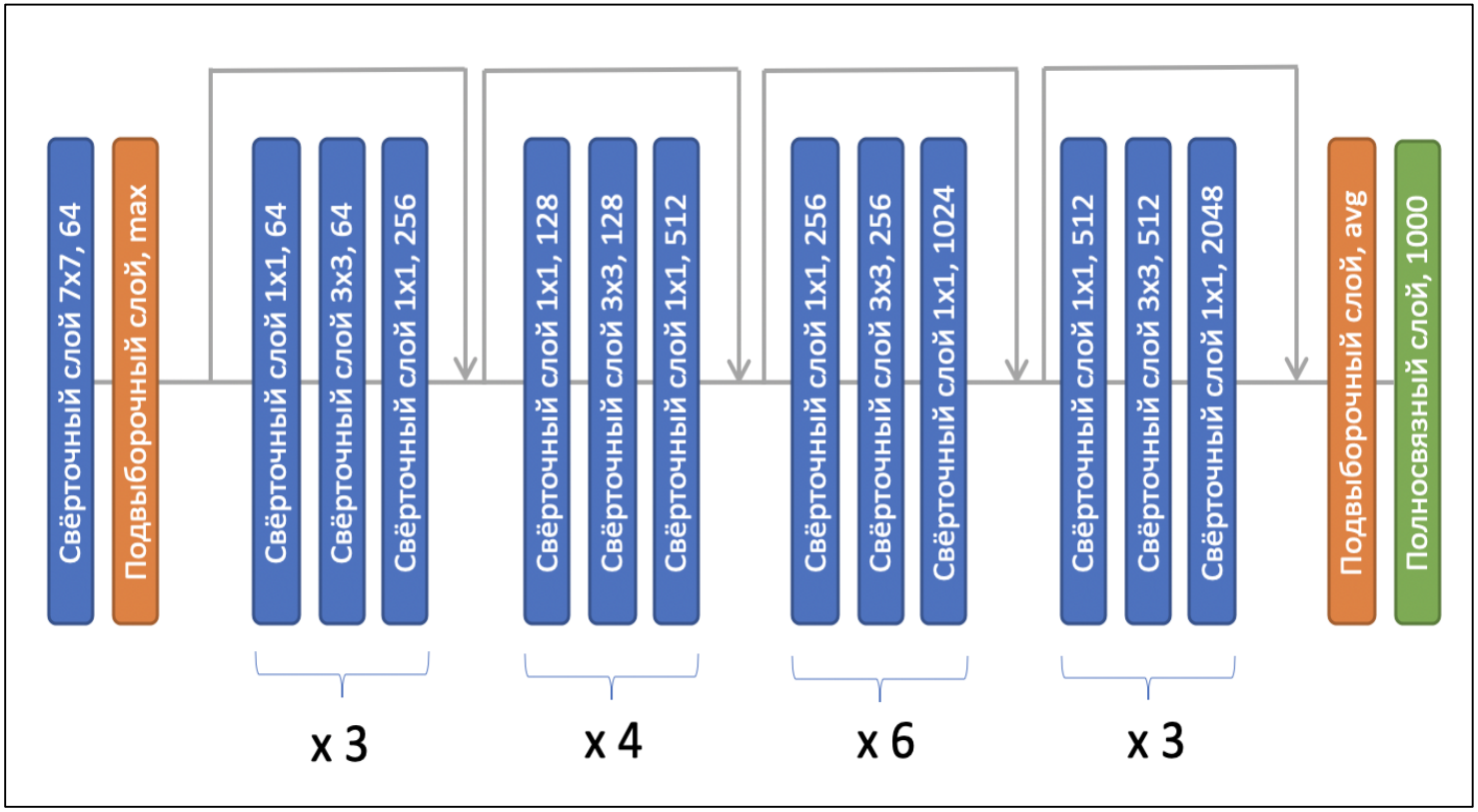
На рисунке 37 представлена архитектура сети ResNet50.

Архитектурно это приводит к тому, что становится возможным обучать очень глубокие сети. Каймин Хе называет это «революцией глубины»: в VGG было 19 уровней, в GoogLeNet — 22 уровня, в первом варианте ResNet — сразу 152, а в последних версиях сетей с остаточными связями без проблем обучаются сети до тысячи уровней в глубину.

Это самые глубокие из реально используемых нейронных сетей. И они действительно работают: большинство лучших результатов в современных нейронных сетях используют в качестве распознавателя объектов разные варианты ResNet. Если нужно что-то распознавать на изображениях, скорее всего, следует пользовать одну из этих архитектур.

Правда, в некоторых приложениях от них отказываются ради скорости и экономии ресурсов: большую свёрточную сеть с остаточными связями нельзя поместить в смартфон. Если ресурсы важны, стоит выбирать модели, которые показывают немного более слабые результаты в распознавании, но имеют при этом на порядок меньше весов; в частности, *MobileNets* и *SqueezeNet*.

Архитектура сети ResNet50 (рисунок 37)



Рекуррентные нейронные сети

Часто исходными данными для решения задачи являются ***последовательности***. Это могут быть временные ряды (изменения цен акций, значения температуры), естественно возникающие последовательности с зависимыми элементами (предложения естественного языка, человеческая речь при распознавании) —любые данные, где соседние точки зависят друг от друга, и эту зависимость нельзя игнорировать.

Можно пытаться моделировать последовательности с внутренними зависимостями и обычными нейронными сетями. Например, одна из первых приходящих в голову идей — зафиксировать некоторую длину истории l и подавать на вход нейронной сети предыдущие l значений ряда: будем пытаться предсказать значение x_n из ряда $x_1, x_2, \dots, x_{n-2}, x_{n-1}$ как функцию $x_n = f(x_{n-1}, \dots, x_{n-l})$. Пример такой архитектуры показан на рис. 38.

Здесь выходы получаются из трех предыдущих входов: $y_4 = f(x_1, x_2, x_3)$, $y_5 = f(x_2, x_3, x_4)$, и $y_6 = f(x_3, x_4, x_5)$. Если задачей модели является предсказание следующего элемента (такая задача стоит, например, при порождении текстов или предсказании временных рядов), в качестве функции ошибки можно взять разницу между y_j и соответствующим x_j (например, процент угаданных слов).

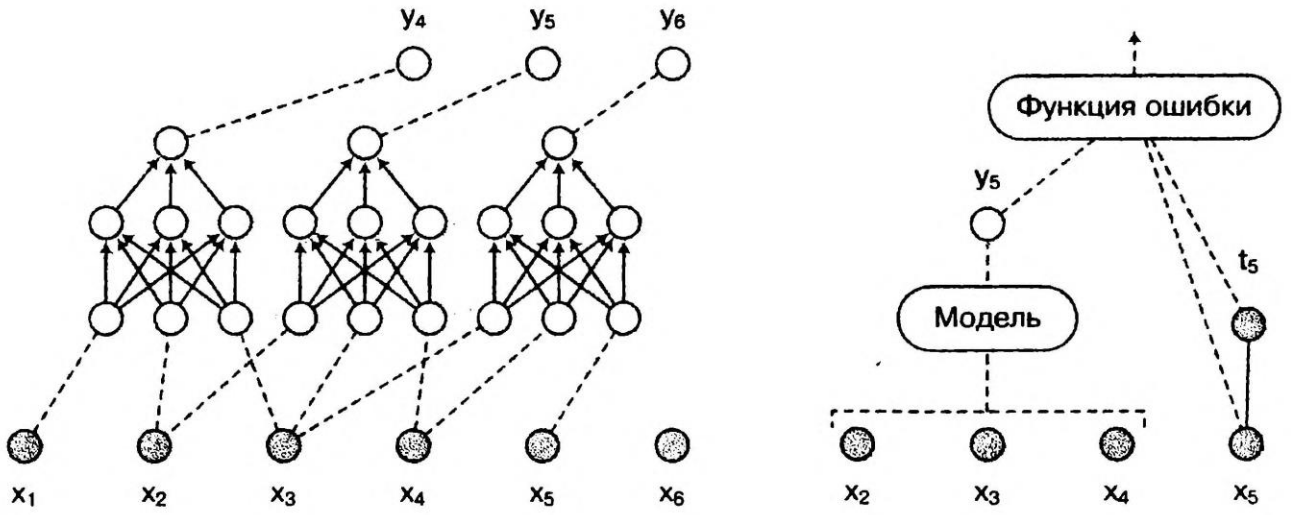
Рекуррентные нейронные сети

А если задача состоит в том, чтобы предсказать одну последовательность по другой (как, например, в распознавании речи: получив на вход последовательность звуков, выдать последовательность слов), потребуются правильные ответы, последовательность будет выглядеть как $\{x_i, t_i\}_{i=1}^{n}$, а функция ошибки будет оценивать, насколько точно наш результат y_j предсказывает правильный ответ t_i из обучающей выборки.

Веса у всех изображенных сетей общие, то есть такая сеть будет рассматривать последовательность из обучающих данных как много независимых тестовых примеров. Фактически это одномерные свёртки.

Для некоторых задач такая идея может сработать, но часто длина зависимости тоже не известна заранее, а предсказание получить, тем не менее, нужно в любой момент времени. В такой ситуации было бы хорошо, если бы нейронная сеть могла ***запоминать*** что-то из истории приходящих на вход данных, сохранять некое внутреннее состояние, которое можно было бы потом использовать для предсказания будущих элементов последовательности.

Архитектура рекуррентной нейронной сети (рисунок 38)



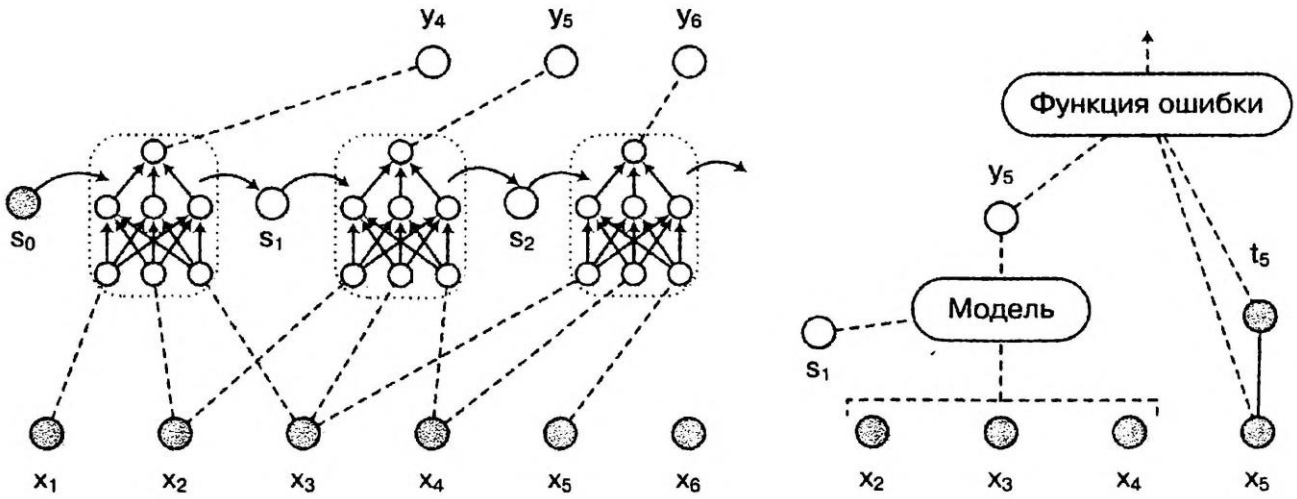
Архитектура обычной нейронной сети с фиксированным размером истории. Слева: одна и та же нейронная сеть применяется к последовательным окнам входа. Справа, результат нейронной сети сравнивается с очередным элементом последовательности

Рекуррентные нейронные сети

Для того чтобы отразить такую временную зависимость в данных, часто используются так называемые ***рекуррентные нейронные сети.*** Обычные нейронные сети имеют фиксированное число входов и воспринимают каждый из них как независимый. В рекуррентных же сетях связи между нейронами могут идти не только от нижнего слоя к верхнему, но и от нейрона к предыдущему значению самого этого нейрона или других нейронов того же слоя. Именно это позволяет отразить зависимость переменной от своих значений в разные моменты времени: нейрон обучается использовать не только текущий вход и то, что с ним сделали нейроны предыдущих уровней, но и то, что происходило с ним самим и, возможно, другими нейронами на предыдущих входах.

Эта идея проиллюстрирована на рис. 39. Он отличается от рис. 38 только наличием связей между последовательными элементами; эти связи отражают присутствие некоторого ***скрытого состояния*** s_t, которое во время t формально зависит от всего, что раньше происходило во входной последовательности.

Архитектура рекуррентной нейронной сети (рисунок 39)



Архитектура рекуррентной нейронной сети в тех же обозначениях, что на рис. 38. Слева: рекуррентная нейронная сеть получает на вход свое предыдущее состояние и последовательные окна входа. Справа: результат рекуррентной нейронной сети точно так же сравнивается с очередным элементом последовательности.

Рекуррентные нейронные сети

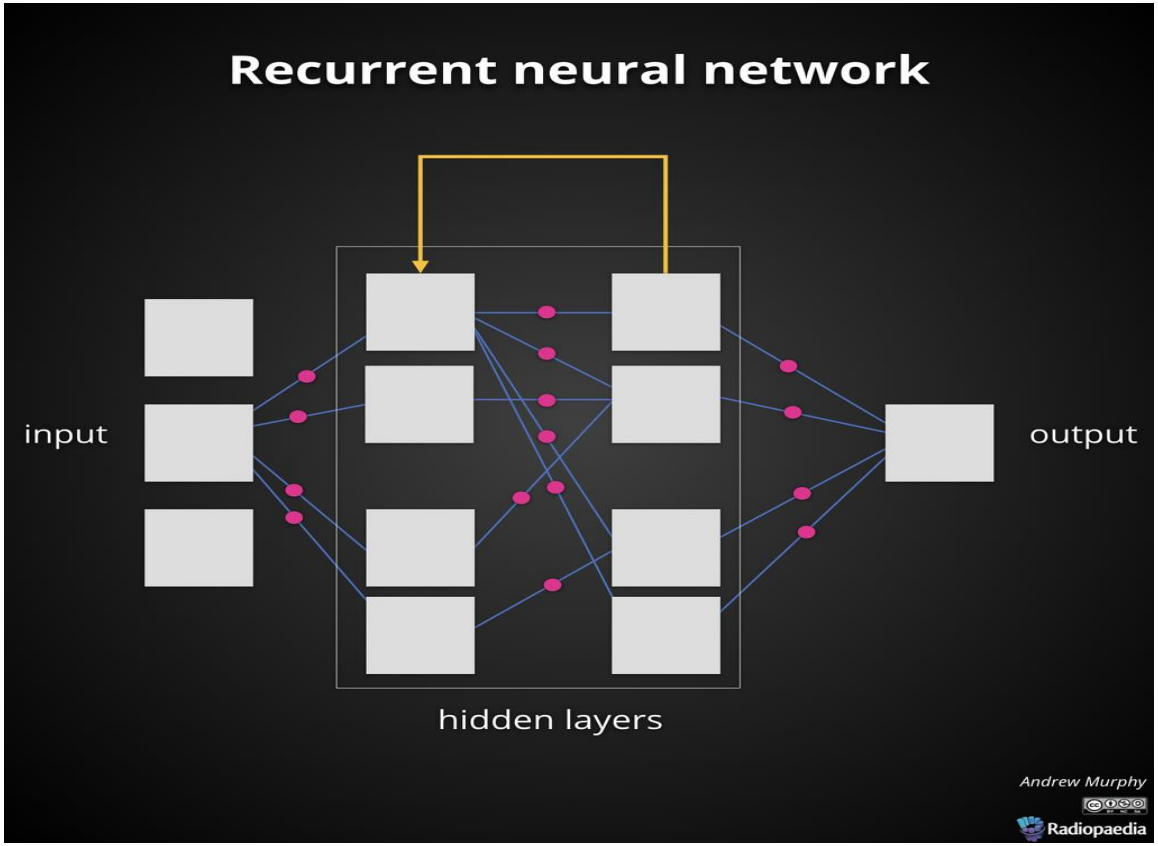
Фактически, в структуру нейронной сети вводятся блоки запоминания, которые запоминают несколько значений с предыдущих шагов по времени. Используя такие блоки, можно создавать любую нейронную сеть, персептрон или сверточную, которая будет понимать время.

Можно подать запомненный сигнал (выход нейрона) на слой с меньшим или тем же номером, как на рисунке 40 (желтая линия). Нельзя сделать это с текущим значением сигнала (выхода), иначе, чтобы посчитать его значение, надо это значение знать. А с запомненным с прошлого такта времени - можно, ведь оно уже посчитано.

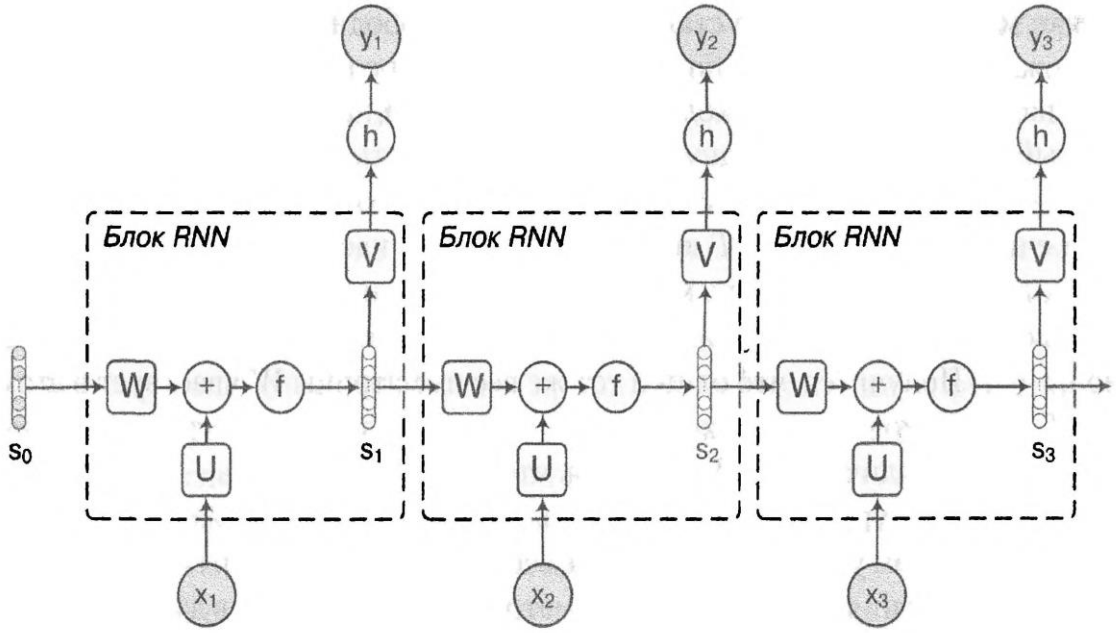
Такие сети, в которых добавленная связь идет в обратном направлении, чем все остальные, назвали сетями с обратными связями, или рекуррентные сети (recurrent neural networks, RNN).

На рисунке 41 представлен пример архитектуры рекуррентной сети.

Пример рекуррентной нейронной сети (рисунок 40)



Архитектура RNN (рисунок 41)



Слой RNN — это блок, развернутый во времени на входную последовательность.

Вентильные нейроны и вентильные сети

Вычислительный блок с множителем назвали **"вентиль" (gate).**

Пусть у нейрона будет некоторое ***состояние*** C_t - число, описывающее его текущее состояние в момент времени t. Текущее состояние C_t определяется предыдущим с некоторым множителем f_t.

$$C_t = f_t * C_{t-1}$$

Такой множитель, от 0 до 1, определяет какую часть состояния нужно **"забыть",** а соответствующий вентиль называют **вентилем забывания** (кружочек с крестиком на рисунке 42).

Надо также добавлять новую информацию, полученную на текущем такте времени, поэтому надо ввести еще один вентиль (**вентиль добавления**, это сумматор, кружочек с плюсиком на рисунке 42), который будет добавлять информацию и тогда текущее состояние определяется как

$$C_t = f_t * C_{t-1} + C_{new}$$

Если научиться регулировать коэффициент забывания и величину, добавляемую в состояние, то можно регулировать и состояние нейрона. Необязательно, но можно, также различать между собой состояние и выход нейрона. Для общности можно считать, что **выход нейрона** h_t есть некоторая функция от состояния, например, **гипертангенс** с некоторым множителем (рисунок 43):

$$h_t = o_t * tanh(C_t)$$

Вентильные нейроны и вентильные сети

Коэффициент забывания f_t должен быть в диапазоне от 0 до 1 и должен регулироваться в процессе обучения. Если в качестве такого коэффициента взять выход обычного нейрона с функцией активации **"сигмоида"** (сигма). Диапазон изменения будет нужный, а изменять выход можем изменением входов в нейрон и его весов. Такой коэффициент будет зависеть от текущего входа x_t, предыдущего выхода h_{t-1} и весов этого нейрона:

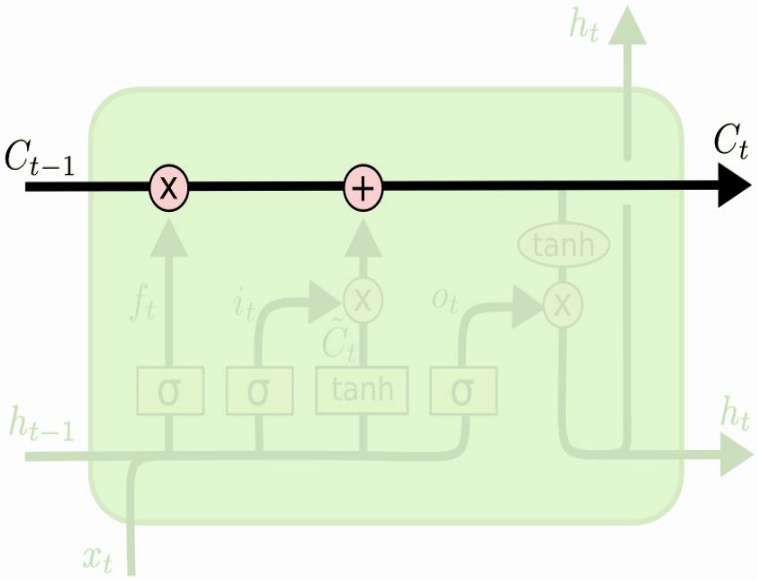
$$f_t = \sigma\left(W_{fh} * h_{t-1} + W_{fx} * x_t + b_f\right)$$

Нейронные сети, состоящие из вентильных нейронов называют **вентильными нейронными сетями**. Сегодня их придумано десятки тысяч. Пожалуй самой известной среди них является **LSTM-сеть** (Long Short Term Memory, долгая краткосрочная память). Архитектура вентильного нейрона сети представлена на рисунке 44.

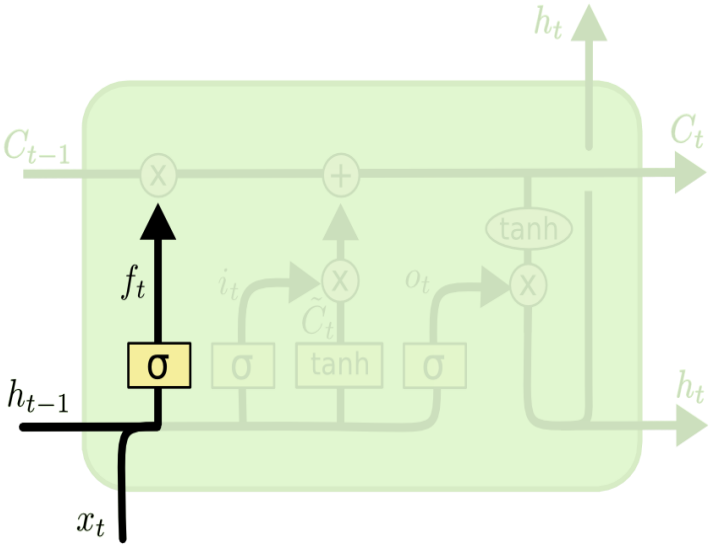
В этой сети в одном вентильном нейроне используется 4 обычных, которые определяют коэффициенты для вентилей забывания f_t, добавления i_t, $\widetilde{C}_t$ и выхода o_t.

Одна из самых широко известных и часто применяющихся конструкций из вентильных нейронов — это LSTM (от слов *Long Short-Term Memory*; на русский язык это можно перевести как «долгая краткосрочная память»). Стандартная архитектура LSTM-ячейки показана на рисунке 45.

Вентильные нейроны (рисунок 42)

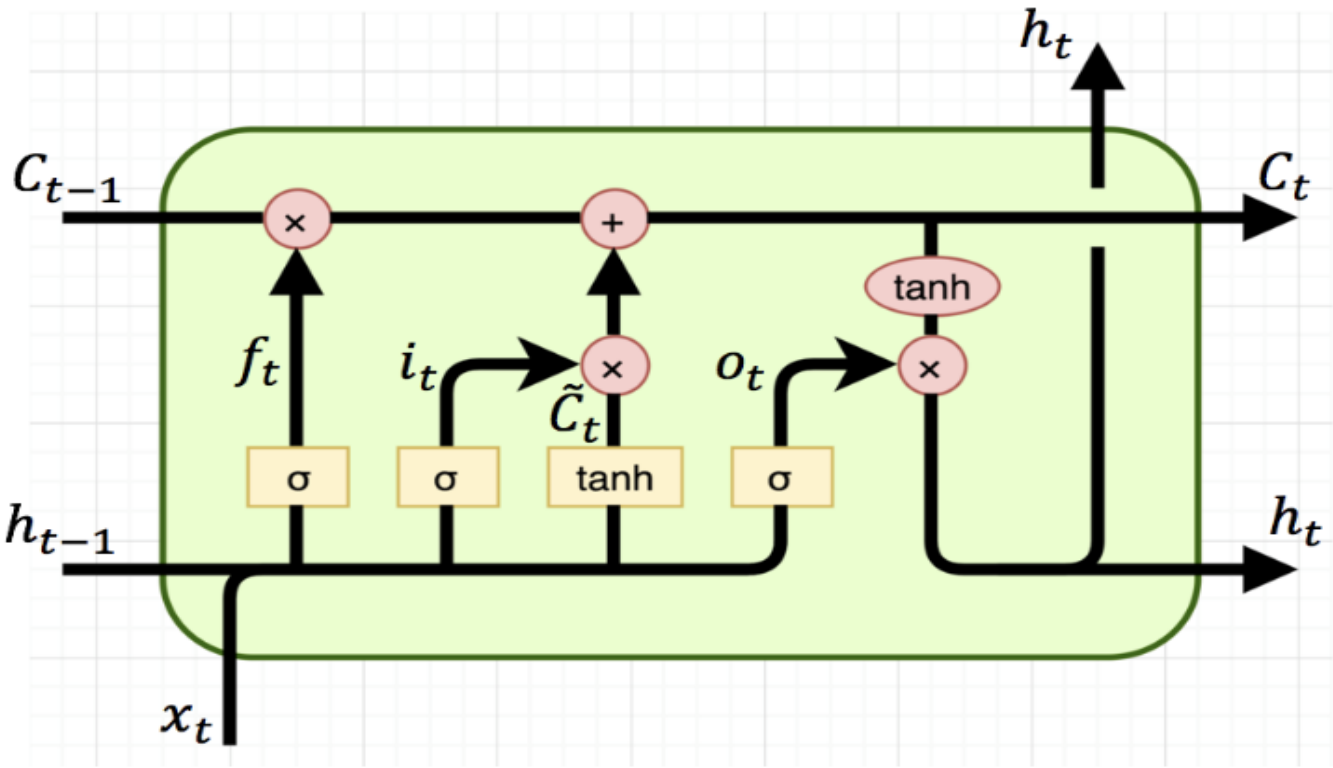


Вентильные нейроны (рисунок 43)



$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \; + \; b_f\right)$$

Вентильные нейроны сети LSTM (рисунок 44)



Сети LSTM

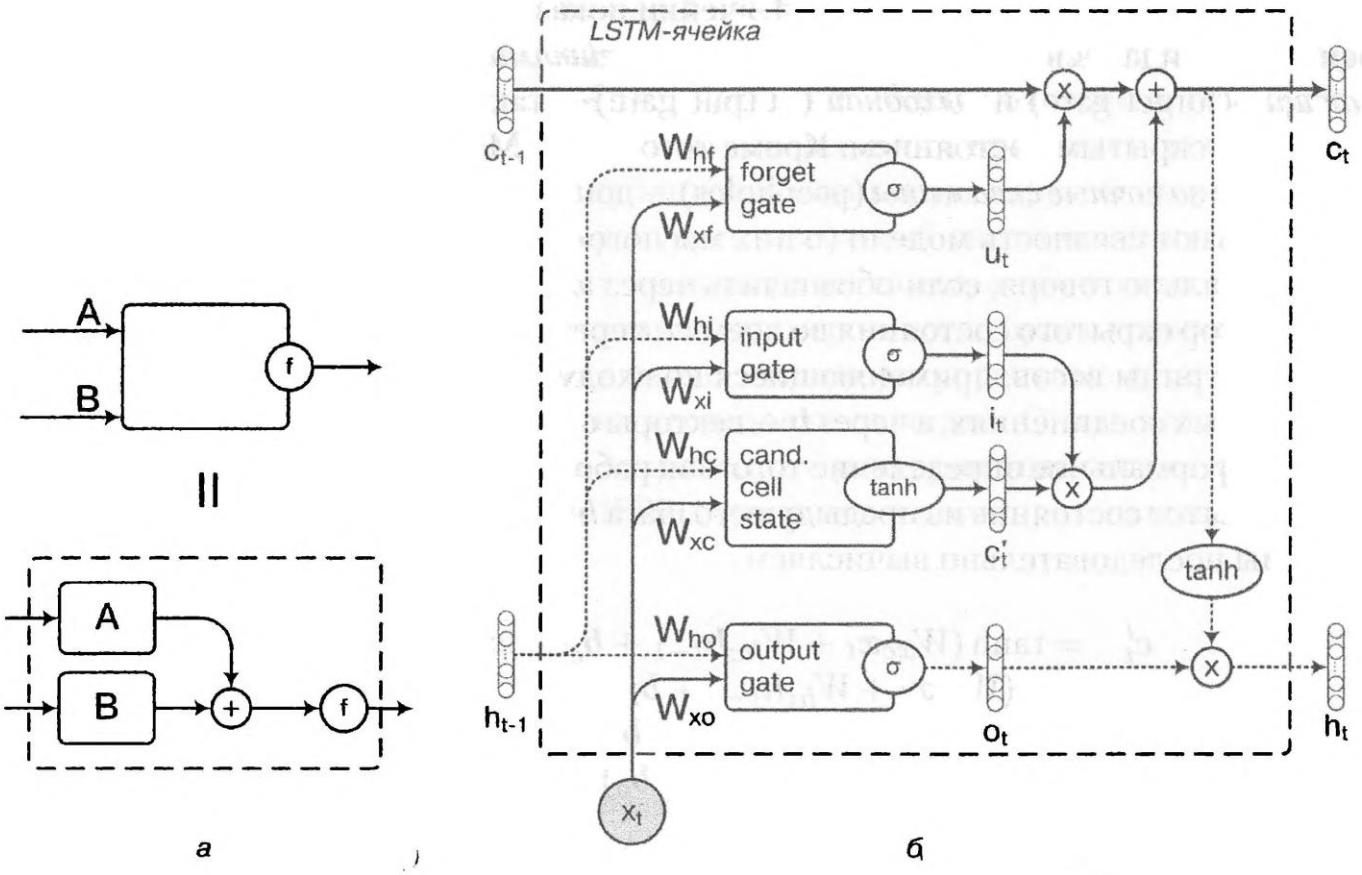
В LSTM есть три основных вида узлов: **входной гейт (*input gate*), забывающий гейт (*forget gate*) и выходной гейт (*output gate*), а также рекуррентная ячейка со скрытым состоянием**. Кроме того, в LSTM часто добавляют еще так называемые **замочные скважины (*peepholes*)** — дополнительные соединения, которые увеличивают связность модели.

Формально говоря, если обозначить через x_t входной вектор во время t, через h_t — вектор скрытого состояния во время t, через W_i (с разными вторыми индексами) — матрицы весов, применяющиеся ко входу, через W_h — матрицы весов в рекуррентных соединениях, а через b — векторы свободных членов, мы получим следующее формальное определение того, как работает LSTM: на очередном входе x_t, имея скрытое состояние из предыдущего шага h_{t-1} и собственно состояние ячейки c_{t-1}, мы последовательно вычисляем:

$$\acute{c}_t = tanh(W_{xc}x_t + W_{hc}h_{t-1} + b_{\acute{c}}) \quad \text{candidate cell state}$$
$$i_t = \sigma(W_{xi}x_t + W_{hi}h_{t-1} + b_i) \quad \text{input gate}$$
$$f_t = \sigma\left(W_{xf}x_t + W_{hf}h_{t-1} + b_f\right) \quad \text{forget gate}$$
$$o_t = \sigma(W_{xo}x_t + W_{ho}h_{t-1} + b_0) \quad \text{output gate}$$
$$c_t = f_t \odot c_{t-1} + i_t \odot \acute{c}_t \quad \text{cell state}$$
$$h_t = o_t \odot tanh(c_t) \quad \text{block output ,}$$

где $\odot$ - векторное произведение.

Сети LSTM (рисунок 45)



а *б*

Сети LSTM

На рисунке 45 а вводится обозначение **гейта**, на рисунке 45 б показана **структура LSTM**; пунктирные линии введены исключительно для удобства и показывают связи, относящиеся к скрытому состоянию h_t (а не c_t). По существу это означает следующее. На вход LSTM, как и в обычной RNN, подаются два вектора: новый вектор из входных данных x_t и вектор скрытого состояния h_{t-1}, который получен из скрытого состояния этой ячейки на предыдущем шаге.

Кроме того, внутри у каждого LSTM-блока есть **«ячейка памяти»** (cell) — вектор, который выполняет функцию памяти. Вектор ячейки на шаге t обозначен через c_t, а $\acute{c}_t$, который получается в первом уравнении, — это вектор, полученный из входа и предыдущего скрытого состояния, который становится кандидатом на новое значение памяти. Получается он из x_t и h_{t-1} обычным для нейронных сетей преобразованием: сначала вычисляется линейная функция, потом гиперболический тангенс. Но $\acute{c}_t$, — это всего лишь кандидат в новое значение памяти. Прежде чем его запишут на место c_{t-1} значение-кандидат и старое значение проходят через еще два гейта: входной гейт i_t и забывающий гейт f_t.

В соответствии с выше приведённой формулой новое значение получается как линейная комбинация из старого с коэффициентами из забывающего гейта f_t и нового кандидата $\acute{c}_t$ с коэффициентами из входного гейта i_t. Там, где значения вектора забывающего гейта f_t будут близки к нулю, старое значение c_{t-1} «забудется», а там, где значения i_t будут велики, новый входной вектор прибавится к тому, что было в памяти.

Сети LSTM

Кроме того, LSTM благодаря своей архитектуре решают проблему исчезающих градиентов, которая мешала рекуррентным сетям обучать долгосрочные зависимости. Пусть в LSTM нет забывающего гейта; тогда $f_t = 1$ во всех компонентах и для всех t. Тогда вектор «памяти» ячейки будет вычисляться как: $c_t = c_{t-1} + i_t \odot \acute{c}_t$. А это значит, что:

$$\frac{\partial c_t}{\partial c_{t-1}} = 1 .$$

Этот эффект, при котором в рекурсивном вычислении состояния ячейки нет никакой нелинейности, в литературе называется **«каруселью константной ошибки»** (constant error carousel): ошибки в сети из LSTM пропагируются без изменений, и скрытые состояния LSTM могут сохранять свои значения неограниченно долго. Это решает **проблему «исчезающих градиентов»:** независимо от матрицы рекуррентных весов ошибка сама собой затухать не будет.

С другой стороны LSTM никак не защищает от **«взрывающихся градиентов»:** если градиент начнет неограниченно расти, линейная зависимость от c_{t-1} никак этому не помешает. Поэтому при реализации LSTM, как и RNN, обычно используют отсечение градиентов (*gradient clipping*), искусственно запрещая им расти далее определенных значений.

Сети LSTM

Обычная инициализация весов нейронной сети маленькими случайными числами хорошо работает для почти всех весов LSTM-ячеек, особым случаем является только свободный член забывающего гейта b_f. Дело в том, что если этот свободный член инициализировать около нуля, это фактически будет значить, что все LSTM-ячейки изначально будут иметь значение f_t около 1/2. Это значит, что карусель константной ошибки перестает работать: фактически во все ячейки вводится фактор «забывания» в 1/2, и в результате ошибки и память будут затухать экспоненциально.

Поэтому свободный член b_f нужно инициализировать большими значениями, от 1 до 2: тогда значения забывающих гейтов f_t в начале обучения будут близки к нулю и градиенты не будут быстро затухать.

Обычную рекуррентную сеть можно рассматривать как частный случай LSTM с гейтами, где некоторые значения зафиксированы в виде констант. Если установить забывающий гейт f_t всегда равным нулю во всех компонентах, а во входном гейте i_t и выходном гейте o_t установить все компоненты в единицы, получится, что предыдущее значение ячейки c_{t-1} всегда забывается, а запоминается значение-кандидат $\acute{c}_t$. на выход оно переходит целиком. Единственным отличием от стандартных рекуррентных сетей будет в таком случае «лишнее» применение *tanh*.

Порождающие сети

Задача классификации это не единственный класс задач, который хотелось бы научиться решать. Как научиться порождать новые объекты, похожие на объекты из данных? Например, как на основе датасета MNIST научиться порождать рукописные цифры? Как генерировать более сложные изображения, например, фото человеческих лиц?

С точки зрения математики существует разница между дискриминативными, разделяющими моделями и генеративными, порождающими:

- • дискриминативные модели обучают функцию, которая отображает вход x в некоторую метку класса y; в вероятностных терминах это значит, что они обучают условное распределение $p(y|x)$;
- • порождающие (генеративные) модели обучают совместное распределение данных $p(x,y)$; это можно использовать для того, чтобы получить $p(y|x)$, ведь для фиксированного x получается $p(y|x) = \frac{p(x,y)}{p(x)} \propto p(x,y)$, но совместное распределение дает больше информации, его можно использовать для порождения новых данных.

Еще одно важное различие состоит в том, что дискриминативные модели решают только задачи обучения с учителем, а порождающие модели могут пытаться решать и задачи обучения без учителя, когда меток никаких нет, и нужно просто смоделировать распределение данных p(x); поэтому во многих практических задачах часто бывают нужны именно порождающие модели.

Порождающие состязательные сети

Если моделировать эти условные вероятности последовательно глубокими нейронными сетями, получится модель, которая сможет последовательно породить x компонент за компонентом, каждый раз для порождения x_i опираясь на уже порожденные $x_1, \dots, x_{i-1}$.

Другой подход к порождению сложных распределений, идею которого используется и в состязательных сетях, состоит в том, чтобы начать с простого распределения $p(z)$ на скрытые факторы z и затем применить к нему сложное преобразование, которое и будет содержать в себе всю сложность требующихся многообразий.

Обычно в качестве $p(z)$ можно взять обычное нормальное распределение, а задача состоит в том, чтобы обучить биективную функцию $f: X \to Z$ так, чтобы распределение данных на X превращалось бы в простое распределение на Z. Тогда, чтобы сгенерировать новую точку из распределения, похожего на $\hat{p}_{data}$, достаточно будет породить точку нормальным законом распределения, а затем применить обратную функцию $f^{-1}: Z \to X$.

Обучить такую функцию можно просто максимизируя правдоподобие, через непрерывные функции f градиент прекрасно «протаскивается» с помощью формулы замены переменных:

Порождающие состязательные сети

$$p_X(x) = p_Z(f(x)) \left| det \left(\frac{\partial f(x)}{\partial x} \right) \right|^{-1}$$

где $\frac{\partial \mathrm{f(x)}}{\partial \mathrm{x}}$ — это матрица частных производных (якобиан) функции f.

Именно эта идея лежит в основе одной из лучших в настоящий момент моделей для работы со звуком, WaveNet, разработанной в компании Google DeepMind. Идея состоит в том, чтобы моделировать условное распределение с помощью нейронных сетей.

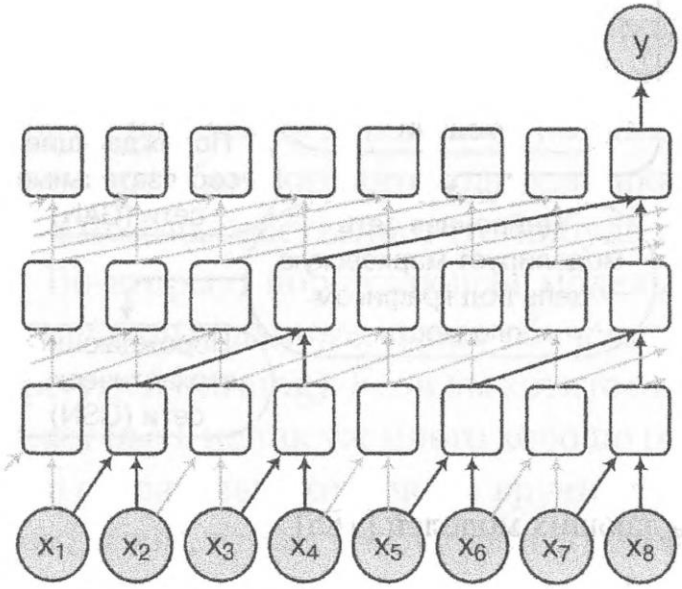
$$p(x|h) = \prod_{t=1}^{T} p(x_t|x_1, \dots, x_{t-1}, h)$$

На звуковых данных полезно делать одномерные свёртки, но свёртки не должны «забегать вперед» по времени, поэтому в WaveNet используются так называемые каузальные свертки (causal convolutions), которые смотрят только назад. Кроме того, их разумно прореживать со временем, чтобы получался «обобщенный» взгляд на более далекую историю.

Порождающие состязательные сети

Все это изображено на рисунке 46 а, где черные стрелочки показывают связи, участвующие в порождении очередного выхода: обратите внимание, что все входы $x_1, \ldots, x_8$ оказывают влияние на выход третьего свёрточного слоя, но при этом каждый вход за счет разреженной структуры более поздних слоев участвует только один раз. А дальше с помощью таких сверток строится архитектура, изображенная на рисунке 46 б. Эта архитектура состоит из нескольких последовательных слоев разреженных сверток, управляемых похожей на гейты структурой, и в ней снова встречаются трюки, которые мы уже не раз видели: остаточные связи, связи «через уровень» и так далее. В результате порождать речь получается довольно хорошо, да и в порождении музыки это, наверное, одна из лучших существующих моделей.

Порождающие сети (рисунок 46)



а

б

Порождающие сети

Для чего нужны порождающие модели?

Во-первых, они позволяют проверить, насколько хорошо использовано распределение данных.

Во-вторых, порождающая модель иногда позволяет использовать не только размеченные данные, делая обучение с частичным привлечением учителя.

В-третьих, порождающие модели хорошо справляются с задачами, в которых может быть несколько правильных ответов.

В-четвертых: иногда просто нужно именно порождать ответы. Например, предположим, необходимо сгенерировать новую фотографию.

Основной критерий, по которому различаются порождающие модели — это то, представляют ли они плотность $p(x,y)$ в явном виде; иначе говоря, можем ли мы подсчитать $p(x,y)$ как функцию от x и y или модель представляет собой черный ящик, который может генерировать новые примеры (x,y), но считать плотность не умеет.

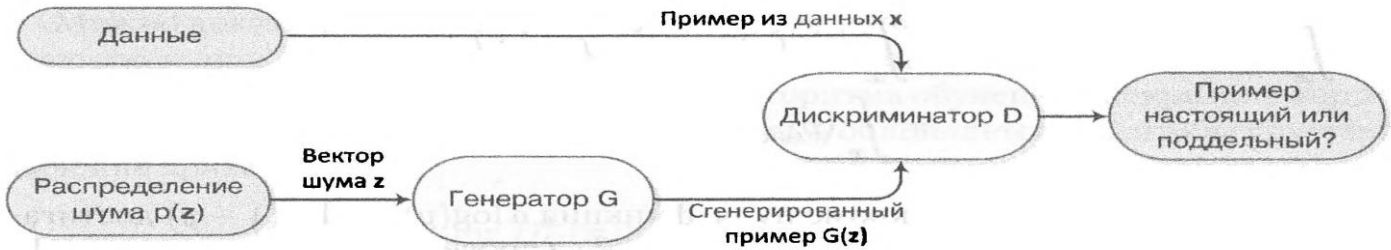
Рассмотрим *порождающие состязательные сети* (Generative Adversarial Networks, GANs). Основная идея появилась в работе 2014 года, главным автором которой был Иэн Гудфеллоу. Сейчас порождающие состязательные сети развиваются очень быстро, и активно используются для обработки изображений и видеороликов.

Порождающие состязательные сети

В базовом варианте модель порождающих состязательных сетей состоит из двух искусственных нейронных сетей, которые соперничают друг с другом. Одна из них, генератор (все связанное с генератором далее будет обозначаться буквой g или G), порождает объекты в пространстве данных, а вторая, дискриминатор (про него будем говорить с индексами d или D), учится отличать порожденные генератором объекты от настоящих примеров из обучающей выборки. Таким образом, получается, что модель GAN состоит из двух частей с противоположными целями (рисунок 47):

- цель дискриминатора — решать самую обычную задачу бинарной классификации: по заданному примеру, похожему на элемент пространства данных, решить, был ли он «настоящим» или был порожден генератором;
- а цель генератора — сделать так, чтобы дискриминатор не смог различить распределение данных p_{data} и распределение p_{gen}, которое порождает генератор; если бы дискриминатор работал идеально, то эта цель совпадала бы с целью научиться порождать данные из в точности такого распределения, как во входной выборке; на практике получается не так идеально, но все равно неплохо.

Схема работы генеративно-состязательной сети GAN (рисунок 47)



Порождающие состязательные сети

Одна сеть пытается научиться порождать правильные примеры, обманывая вторую, а вторая пытается отличить порожденные первой примеры от настоящих. По мере обучения они постепенно делают друг друга лучше. Практическая цель всего этого состоит в том, чтобы генератор в конце концов победил и научился делать p_{gen} настолько похожим на p_{data} что никто не отличит — но чтобы побеждал он обязательно в сложной честной борьбе GAN не генерирует новые примеры из ничего. Нейронная сеть преобразовывает случайные входы, сгенерированные из некоторого «посевного» распределения, которое во всех наших примерах будет просто стандартным нормальным распределением *N(0,1)* (многомерным). То есть вместо того, чтобы порождать примеры, сеть будет преобразовать простую функцию (стандартное нормальное распределение) в сложную (распределение данных) — а это как раз задача, для которой нейронные сети отлично подходят.

Поэтому формально генератор можно записать так:

$G\{z; \theta_g\}: Z \to X$,

где Z — некоторое пространство скрытых (латентных) факторов, на котором задано априорное распределение $p_z(z)$. А дискриминатор, в свою очередь, выглядит так:

$D = D\{x; \theta_d\}: X \to [0,1]$.

Порождающие состязательные сети

Он отображает объекты из пространства данных в отрезок [0,1], который интерпретируется как вероятность того, что пример был действительно «настоящий», из p_{data} а не сгенерированный из p_{gen}. Цель дискриминатора состоит в том, чтобы на обучающей выборке выдавать максимальный результат, а на порожденных генератором примерах — минимальный.

Целевая функция для дискриминатора: на примерах из p_{data} ожидаемый ответ дискриминатора должен быть как можно больше, а на примерах из p_{gen} — как можно меньше, то есть дискриминатор должен максимизировать следующую величину:

$$\mathbb{E}_{x\sim p_{data}(x)} log[D(x)] + \mathbb{E}_{x\sim p_{gen}(x)} log\big[\big(1 - D(x)\big)\big],$$

где p_{gen}. — это порождаемое генератором распределение, $p_{gen}(x) = G_{x\sim p_z(z)}$.

С другой стороны, генератор должен научиться обманывать дискриминатор, то есть минимизировать по p_{gen} следующую величину:

$$\mathbb{E}_{x\sim p_{gen}(x)} log\big[\big(1 - D(x)\big)\big] = \mathbb{E}_{z\sim p_z(xz)} log\Big[\Big(1 - D\big(G(z)\big)\Big)\Big].$$

Если теперь объединить эти функции в одну, то увидим, что фактически дискриминатор и генератор играют между собой в игру, которую в теории игр называют минимаксной игрой, решая следующую задачу оптимизации:

Порождающие состязательные сети

$\min_G \max_D V(D,G)$, где

$V(G,D) = \mathbb{E}_{x\sim p_{data}(x)} log[D(x)] + \mathbb{E}_{z\sim p_z(xz)} log\left[\left(1 - D\left(G(z)\right)\right)\right]$.

Отсюда и происходит название модели.

Схематически это можно изобразить так, как показано на рисунке 47. Алгоритм обучения GAN тоже очень простой: надо поочередно обновлять то веса генератора, то веса дискриминатора, каждый раз считая «противника» фиксированным. Оказывается, несложно и формально доказать, что идея работает, и при вышеописаннjм процессе обучения p_{gen} действительно постепенно сходится к p_{data}.

Можно доказать и то, что p_{gen} сходится к p_{data} при попеременном обучении генератора и дискриминатора. Если модели, представляющие G и D достаточно выразительны, и на каждом шаге алгоритма обучения дискриминатор достигает оптимума при условии текущего G, а p_{gen} обновляется так, чтобы улучшать значение критерия: $\mathbb{E}_{x\sim p_{data}(x)} log[D_G^*(x)] + \mathbb{E}_{x\sim p_{gen}(x)} log\left[\left(1 - D_G^*(x)\right)\right]$, то p_{gen} сходится к p_{data}.

Порождающие состязательные сети

Понятно, что на практике D и G представляют собой нейронные сети, и оптимизация происходит по параметрам θ_g а не по самому распределению p_{gen} в общем виде. Заметим, что для того, чтобы формально следовать доказанному результату, нам нужно было бы после каждого шага обновления генератора полностью дообучать дискриминатор до сходимости, ведь задача классификации, которую он решает, изменилась, и генератору нужно научиться обманывать версию дискриминатора, соответствующую своей текущей версии. Но на практике нельзя полностью обучать сложную модель на каждом шаге, поэтому используется, так сказать, стохастическая версия этого обучения: делается один шаг обучения G, потом несколько шагов обучения D потом еще один шаг обучения G и т. д. Несмотря на то, что использование глубоких нейронных сетей приводит к появлению множества критических точек в пространстве параметров, удачные практические результаты их использования позволяют считать нейросети подходящими моделями и не обращать внимания на отсутствие теоретических гарантий.

Архитектуры, основанные на GAN

Первые порождающие состязательные сети связаны с архитектурой DCGAN (Deep Convolutional Generative Adversarial Networks). Это обычная GAN, в которой генератор и дискриминатор представляют собой глубокие свёрточные сети, что логично для обработки изображений. Однако есть несколько важных особенностей, которые стоит иметь в виду, если применять GAN к тем или иным изображениям:

- вместо субдискретизации используются дополнительные свёрточные уровни, в которых свёртки участвуют с пробелами (strided convolutions); эта идея известна как «полностью свёрточные» сети (all convolutional nets);
- полносвязные скрытые слои тоже не используются; фактически от свёрточной сети остаются только сами свёртки;
- нормализация по мини-батчам используется как в генераторе, так и в дискриминаторе;
- в генераторе везде используется ReLU, а в дискриминаторе — протекающий ReLU (Leaky ReLU).

Модель была обучена, на датасете интерьеров спален LSUN (Large-Scale Scene Understanding Challenge) и на собранном авторами датасете из трех миллионов человеческих лиц, В результате не просто получился генератор, который производит разумные интерьеры и человеческие лица, но и у пространства скрытых факторов нашлись очень интересные свойства. Можно, например, вычесть из фотографий мужчин в очках фотографии мужчин без очков, прибавить женщин без очков и получить женщин в очках.

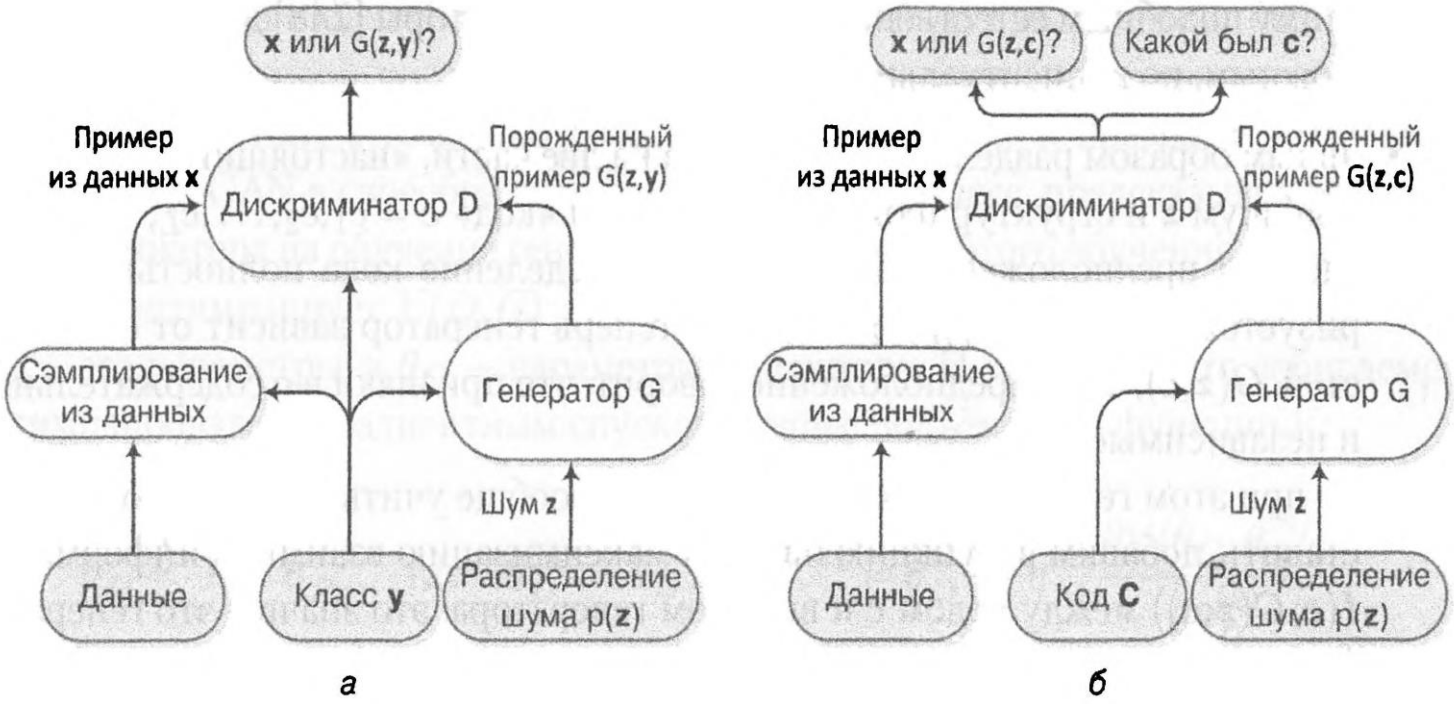
Архитектуры, основанные на GAN

Появились и практически важные задачи, которые такие архитектуры решают лучше всех. Например, решается задача *повышения разрешения* изображения (image super-resolution): можно разумным образом увеличить разрешение фотографии, добавить деталей так, чтобы они казались правдоподобными на человеческий взгляд (понятно, что речь не идет о том, чтобы восстановить «настоящие» детали, такой информации в фотографии низкого качества просто нет).

Архитектура SRGAN очень похожа на DCGAN, и результаты получаются очень хорошие. Похожа на это и задача восстановления МРТ-изображений в медицине (MRI reconstruction), которую решают с помощью GAN.

Следующее очень естественное расширение архитектуры состязательных сетей — *условный GAN* (conditional GAN). Эта модель использует дополнительную информацию y , которая часто доступна вместе с примерами x ; например, это может быть метка класса изображения (кошка или собака, номер цифры в MNIST и так далее). Подаётся y вместе со случайным вектором на вход генератору, тот же y на подаётся на вход дискриминатора для различения (рисунок 38 *a);* в результате генератор пытается строить модель условного распределения $p_{gen}(x|y) = G(z, y)$, а дискриминатор $D(x|y)$ строит распределение вероятности того, что вход «настоящий», тоже при условии y.

Архитектуры, основанные на GAN (рисунок 38)



Архитектуры, основанные на GAN

На рисунке 38 приведены две архитектуры модификаций GAN: *а* — условный GAN; *б* — InfoGAN. Например, в GAN в качестве объектов можно использовать фотографии человеческих лиц, а в качестве дополнительной информации — возраст человека на фотографии. Такой условный GAN способен порождать лица заданного возраста; более того, по фотографии лица можно сначала отобразить его в пространство признаков, а потом изменить признак возраста и породить новую фотографию, таким образом «состарив» или «омолодив» человека на фотографии. Для этого нужна чуть более сложная архитектура.

Еще одна интересная модификация условных GAN, *InfoGAN*, добавляет к обычной задаче оптимизации, решаемой в GAN, дополнительные теоретико информационные ограничения. Задача состоит в том, чтобы обучить на скрытом слое не просто «хорошее» представление, из которого можно сэмплировать, а «распутанное» представление (disentangled representation), в котором отдельные признаки имеют естественную интерпретацию. Например, хотелось бы, чтобы признаки у GAN, генерирующей человеческое лицо, соответствовали цвету глаз, форме носа и тому подобным естественным особенностям. Авторы данной архитектуры добиваются этого несложным, но концептуальным способом:

Архитектуры, основанные на GAN

- явным образом разделим вектор шума на две части, «настоящий» несжимаемый шум z и структурированный шум, или «код» $c = c_1, c_2 \dots, c_L$;
- сделаем предположение о том, что распределение кода полностью факторизуется, $p(c) = p(c_1), p(c_2), \dots, p(c_L)$ теперь генератор зависит от обеих частей, $G(z, c)$, а это предположение говорит, что признаки в с содержательные и независимые;
- но при этом генератор пока что не обязан вообще учитывать c; чтобы его заставить, добавим в функцию ошибки максимизацию взаимной информации $I\big(c; G(z, c)\big)$ между кодом c и выходом генератора; это значит, что генератор будет пытаться как можно сильнее учитывать c в своем выходе, максимизировать влияние c на $G(z, c)$, то есть по сути будет пытаться сделать так, чтобы по $G(z, c))$ можно было восстановить c (рисунок 38 б).

В результате структура модели и обучение усложняются не сильно — по сути, добавляется один полносвязный слой, который считает и максимизирует нижнюю оценку $I\big(c; G(z, c)\big)$. Но скрытые представления получаются действительно «распутанными»: например, на MNIST выделяются признаки, соответствующие самой цифре, ее ширине, углу поворота и т. д.

Аппаратная поддержка CHC

Высокопроизводительная платформа Nvidia DGX Station

Пиковая производительность:
480 Tflops (FP16)

GPU: **4X Tesla V100**

Общий объём памяти GPU: **128 ГБ**

ЦП: **CPU Intel Xeon E5-2698 v4 2.2 GHz**

Общий объём системной памяти: **256 ГБ**

Хранилище данных: **3X 1.92 ТБ RAID 0**

Объём диска для ОС: **1.92 ТБ SSD**