МИНИСТЕРСТВО ОБРАЗАВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ
БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
Факультет прикладной математики и информатики
Кафедра информационных систем управления

ПОИСК ЛИЦА НА ИЗОБРАЖЕНИИ НА ОСНОВЕ СВЁРТОЧНЫХ НЕЙРОННЫХ СЕТЕЙ

Курсовая работа

Гумилевского Кирилла Сергеевича
студента 3 курса
специальность «Информатика»

Научный руководитель:
Заведующий кафедрой ИСУ,
доктор технических наук, профессор
Краснопрошин В.В.

Минск
2019

АННОТАЦИЯ

Гумилевский К.С. Поиск лица на изображении на основе свёрточных нейронных сетей. Курсовая работа / Минск: БГУ, 2019. – 28 с.

Рассматривается задача поиска лица на изображении на основе свёрточных нейронных сетей.

АНАТАЦЫЯ

Гумілеўскі К.С. Пошук асобы на малюнку на аснове свёртачных нейронавых сетак. Курсавая работа / Мінск: БДУ, 2019. – 28 с.

Разглядаецца задача пошуку асобы на малюнку на аснове свёртачных нейронавых сетак.

ANNOTATION

Gumilevski K.S. Face detection on images based on convolutional neural networks. Course work / Minsk: BSU, 2019. – 28 p.

The task of detecting face on image based on convolutional neural neyworks.

ОГЛАВЛЕНИЕ

ПЕРЕЧЕНЬ СОКРАЩЕНИЙ И УСЛОВНЫХ ОБОЗНАЧЕНИЙ

СНС – свёрточные нейронные сети
Lr - learning rate, скорость обучения

ВВЕДЕНИЕ

Искусственные нейронные сути уже давно известны научному сообществу. Однако лишь недавно получили широкое практическое применение. Это стало возможным благодаря технологии CUDA, которая позволяет производить параллельные вычисления на GPU.

Сферы применения искусственных нейронных сетей очень обширны: начиная от уже классических задач классификации и регрессии, заканчивая их применением в компьютерном зрении.

Нейронные сети, в частности СНС, отлично подходят для задач компьютерного зрения. Они способны выделять из изображений объекты очень сложной формы. Также СНС имеют очень хорошую обобщающую способность, что позволяет им распознавать различные вариации одного и того же объекта, обучаясь на ограниченном наборе данных.

В качестве прикладной задачи для исследования нейронных сетей была выбрана задача распознавания лиц и, в частности, одна из важнейших её подзадач – поиск лица на изображении. В данной работе подробно описаны принципы обучения и работы СНС, выделены основные этапы решения задачи поиска лица,построены и сравнены различные модели, основанные на СНС.

ГЛАВА 1

ПОСТАНОВКА ЗАДАЧИ И ПОДГОТОВКА ОБУЧАЮЩЕЙ ВЫБОРКИ

1.1 Постановка задачи

Дано:

RGB-изображение произвольного размера с одним, вариативным по положению, лицом на нём.

Требуется:

Построить функцию $F: \mathbb{R}^{M\text{x}N\text{x}3} \rightarrow \mathbb{R}^4$, которая сможет выделять лица на изображении, а именно строить прямоугольную ограничительную рамку, внутри которой расположено лицо. ($\mathbb{R}^{M\text{x}N\text{x}3}$, по сути, - массив пикселей, который описывает изображение, M, N – размеры изображения, а 3 соответствует трём RGB каналам). Функция F возвращает координаты левой верхней и правой нижней вершины прямоугольной ограничительной рамки.

Рисунок 1.1. – Визуализация исходной постановки задачи

Предлагается подготовить набор размеченных данных и по нему обучить нейронную сеть. В работе рассмотрены и сравнены различные архитектуры нейронных сетей.

1.2 Подготовка обучающей выборки

Существует множество размеченных наборов данных для распознавания лиц. Некоторые платные, некоторые выдаются по запросу. Некоторые содержат большую вариативность по освещению, другие — по положению лица. Некоторые сняты в лабораторных условиях, другие собраны из фотографий, снятых в естественной среде обитания. Чётко сформулировав требования к данным, можно легко выбрать подходящий датасет или собрать его из нескольких.

Для меня в рамках данной курсовой работы требования были таковы: обучающая выборка должна быть легко доступна для скачивания и содержать вариативность по положению лица. Требованиям удовлетворили сразу несколько наборов данных, из них я выбрал один – UMDFaces (Batch 3).

Набор данных UMDFaces содержит 367 888 аннотаций лица для 8 277 субъектов, разделенных на 3 части (я пользовался третьей частью). Помимо ограничительных рамок, которые использовались в моей работе, UMDFaces предоставляет данные о расположении 21 ключевой точки лица, пол, сгенерированный предварительно обученной нейронной сетью, данные о положении лица (отклонение от нормали и др) и многое-многое другое.

Из всего набора данных было выделено 22 тысячи изображений для обучения и тестирования нейронных сетей. Из них 16 тысяч – train set (использовался для обучения нейронных сетей), 4 тысячи – validation set (использовался для подборки гиперпараметров моделей и предварительной оценки точности на данных, которые не видела нейронная сеть), остальные 2 тысячи – test set (в свою очередь использовался для объективной оценки окончательной обученной модели).

Так как использование нейронных сетей подразумевает использование изображений фиксированного размера, все изображения были сжаты (или растянуты в случае если они были меньше размера 256 на 256) до размера 256 на 256 с помощью библиотеки keras_preprocessing. Такое преобразование изображений потребовало изменений в размеченных данных, а именно

$$new\_x \leftarrow \frac{256}{w}\, x, \qquad new\_y \leftarrow \frac{256}{h}\, y$$

$$new\_w \leftarrow \frac{256}{w}\, w, \qquad new\_h \leftarrow \frac{256}{h}\, h$$

где new_x, new_y, new_w, new_h – обновленные значения x-координаты, y-координаты левого верхнего пискеля прямоугольной ограничительной рамки, ширины и высоты прямоугольной ограничительной рамки.

Также, для однородности ответов в размеченных данных (в данном случае под однородностью понимается то, что все компоненты ответов – части координат некоторого пикселя) я перешел от системы (x, y, w, h) к системе (x, y, x + w, y + h). Система (x, y, w, h): x, y-координаты левого верехнего пикселя, ширина и высота прямоугольной ограничительной рамки), а система (x, y, x + w, y + h): x, y-координаты левого верехнего и правого нижнего пикселей.

Реализация подготовки обучающей выборки может быть найдена в приложении А.

ГЛАВА 2

МОДЕЛИ И АЛГОРИТМЫ ДЛЯ ПОИСКА ЛИЦ НА ИЗОБРАЖЕНИИ

2.1 Свёрточные нейронные сети

2.1.1 Что такое CHC. Введение

CHC — одна из самых влиятельных инноваций в области компьютерного зрения. Впервые нейронные сети привлекли всеобщее внимание в 2012 году, когда Алекс Крижевски благодаря им выиграл конкурс ImageNet (грубо говоря, это ежегодная олимпиада по машинному зрению), снизив рекорд ошибок классификации с 26% до 15%, что тогда стало прорывом. Сегодня глубинное обучение лежит в основе многих передовых исследований и услуг компаний: Facebook использует нейронные сети для алгоритмов автоматического проставления тегов, Google — для поиска среди фотографий пользователя, Amazon — для генерации рекомендаций товаров, Pinterest — для персонализации домашней страницы пользователя, а Instagram — для поисковой инфраструктуры.

Но классический и один из самых популярных вариантов использования сетей это обработка изображений. Сегодня CHC используются повсюду: в задачах классификации, поиска, распознавания, сегментации изображений и видео, и это лишь часть задач, которые умеют решать с помощью CHC.

2.1.2 Общая структура CHC

Что конкретно делают CHC? Берётся изображение, пропускается через серию свёрточных, нелинейных слоев, слоев объединения и полносвязных слоёв, и генерируется вывод. Что представляет из себя вывод? Зависит от задачи. Например, если мы решаем задачу классификации, то вывод - класс или вероятность классов, которые лучше всего описывают изображение. В нашем же случае выводом являются координаты прямоугольной ограничительной рамки, в которой находится лицо. Сложный момент — понимание того, что делает каждый из слоев CHC. Так давайте перейдем к нему!

2.1.3 Первый слой CHC

Первый слой в CHC - свёрточный. Для упрощения понимания возьмём небольшую размерность изображения: будем считать, что входное изображение - матрица 32 x 32 x 3 с пиксельными значениями.

Что представляет из себя свёрточный слой?

Для того, чтобы максимально учесть и сохранить структуру изображения, будем рассматривать небольшие “кусочки” изображения, например 5 на 5. Будем двигаться по всем “кусочкам” 5 на 5(а точнее 5x5x3, ведь размер изображения – 32x32x3) входного изображения и применять некоторые преобразования. Для свёрточного слоя это преобразование – поэлементное умножение нашего “кусочка” (матрицы 5x5x3) на некоторую другую матрицу, которую называют

фильтром. Заметим, что фильтр также имеет размер 5x5x3. В терминах CHC эти “ кусочки ” называют *рецептивным полем* или *полем восприятия*, а размеры рецептивного поля (в нашем случае 5 на 5) являются настраиваемыми параметрами архитектуры нейронной сети.

Теперь давайте для примера возьмем позицию, в которой находится фильтр. Пусть это будет левый верхний угол. Фильтр производит свёртку, то есть двигается по входному изображению и, как уже было упомянуто выше, умножает значения фильтра на исходные значения пикселей изображения (поэлементное умножение). Все эти умножения суммируются (всего 5 * 5 * 3 = 75 умножений). И в итоге получается одно число. Это число просто символизирует нахождение фильтра в верхнем левом углу изображения.

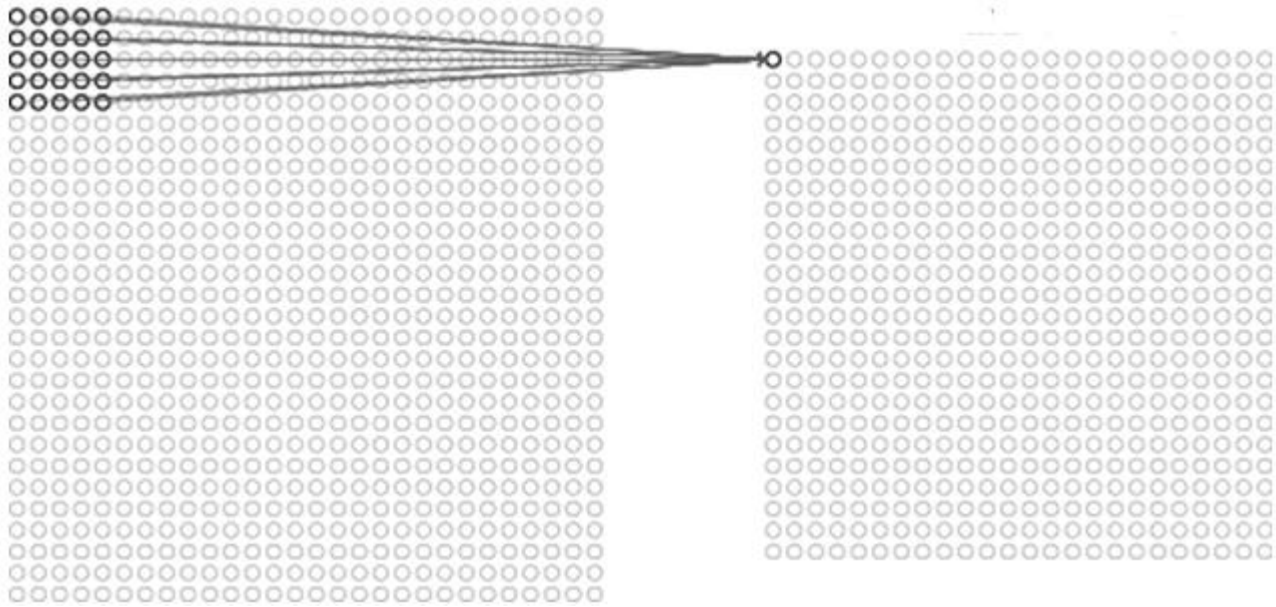

Рисунок 2.1. - Формирование карты признаков

Теперь повторим этот процесс в каждой позиции изображения. Следующий шаг — перемещение фильтра вправо на какой-то шаг (размер шага также является настраиваемым параметром архитектуры нейронной сети, в нашем примере шаг = 1), затем еще на шаг вправо и так далее пока изображение справа не закончится, затем спускаемся на шаг вниз и проделываем аналогичные операции, таким образом мы покрываем всё изображение. Каждая уникальная позиция введённого изображения производит число. После прохождения фильтра по всем позициям получается матрица 28 x 28 x 1, которую называют *функцией активации* или *картой признаков*. Матрица 28 x 28 получается потому, что есть 784 различных позиции, которые могут пройти через фильтр 5 x 5 изображения 32 x 32. Эти 784 числа преобразуются в матрицу 28 x 28. Нетрудно убедиться, что в общем случае (для размера входного изображения NxNx3, размера фильтра F и шага равного S) получаем карту признаков размера MxMx1, где $M = \frac{N-F}{S} + 1$. Также заметим, что если $N - F$ не делится нацело на S, то данные размеры фильтра и шага являются недопустимыми.

Допустим, теперь мы используем два 5 x 5 x 3 фильтра вместо одного. Тогда выходным значением будет 28 x 28 x 2 и в общем случае размер выходной карты признаков – MxMxT, где $M = \frac{N-F}{S} + 1$, а T – количество фильтров.

Теперь давайте посмотрим, что на самом деле свёртка делает на высоком уровне. Каждый фильтр можно рассматривать как идентификатор некоторого свойства (на первом слое под свойствами понимаются простейшие кривые, прямые границы, простые цвета, то есть самые простые характеристики, которые имеют все изображения в общем). Скажем, наш первый фильтр 7 x 7 x 3, и он будет детектором кривых. (для примера давайте игнорировать тот факт, что у фильтра глубина 3, и рассмотрим только верхний слой фильтра и изображения, для простоты).

0	0	0	0	0	30	0
0	0	0	0	30	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	0	0	0	0

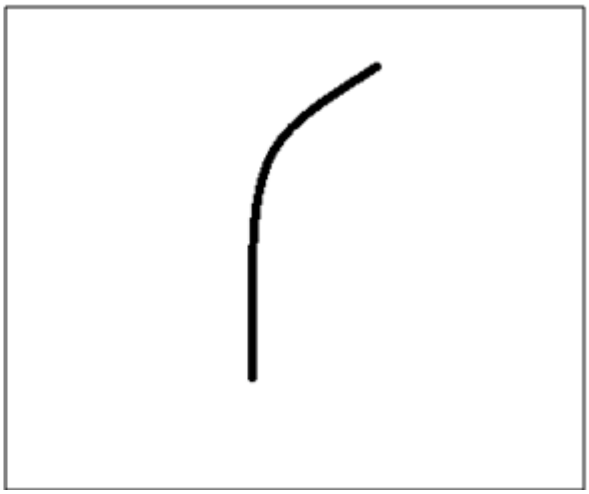

Рисунок 2.2. – отдельный фильтр: матричное представление и визуализация

Когда у нас в левом верхнем углу входного изображения есть фильтр, он производит умножение значений фильтра на значения пикселей этой области. Давайте рассмотрим пример изображения и установим фильтр в верхнем левом углу. (Рисунок 2.3)

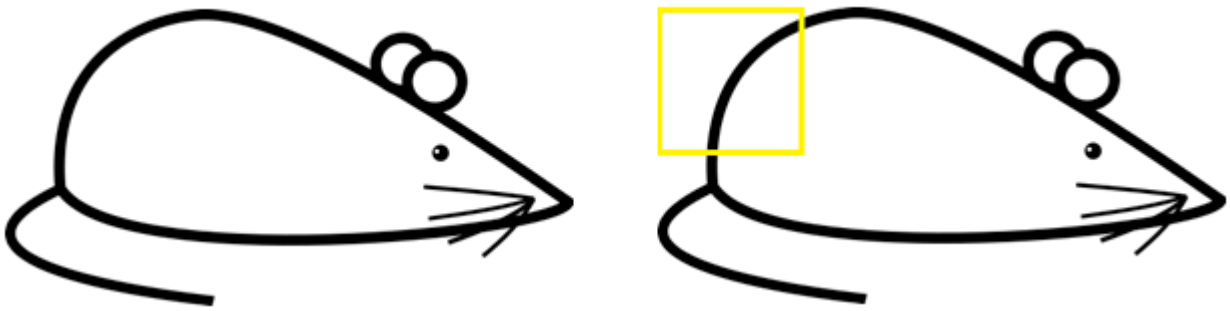

Рисунок 2.3. – исходное изображение / исходное изображение с отмеченным рецептивным полем на нём

Всё, что нам нужно, это умножить значения фильтра на исходные значения пикселей изображения.

0	0	0	0	0	0	30
0	0	0	0	50	50	50
0	0	0	20	50	0	0
0	0	0	50	50	0	0
0	0	0	50	50	0	0
0	0	0	50	50	0	0
0	0	0	50	50	0	0

*

0	0	0	0	0	30	0
0	0	0	0	30	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	0	0	0	0

Рисунок 2.4. – Операция свёртки для заданного рецептивного поля и фильтра

В результате поэлементного умножения и сложения результатов получаем 6600 (большое число!). По сути, если на вводном изображении есть форма, в общих чертах похожая на кривую, которую представляет этот фильтр, и все умноженные значения суммируются, то результатом будет большое значение. Теперь давайте посмотрим, что произойдёт, когда мы переместим фильтр. (Рисунок 2.5)

0	0	0	0	0	0	0
0	40	0	0	0	0	0
40	0	40	0	0	0	0
40	20	0	0	0	0	0
0	50	0	0	0	0	0
0	0	50	0	0	0	0
25	25	0	50	0	0	0

*

0	0	0	0	0	30	0
0	0	0	0	30	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	0	0	0	0

Рисунок 2.5. – Операция свёртки для некоторой другой области изображени

В результате поэлементного умножения и сложения результатов получаем 0, что свидетельсвтует о том, что в новой области изображения нет ничего, что “похоже” на кривую фильтра.

Подытожим пример: вывод данного свёрточного слоя — карта признаков. В самом простом случае, при наличии одного фильтра (на практике же используется большее количество различных фильтров) свертки (и если этот фильтр — детектор кривой), карта свойств покажет области, в которых больше вероятности наличия кривых. В этом примере в левом верхнем углу значение нашей 28 x 28 x 1 карты свойств будет 6600. Это высокое значение показывает, что, возможно, что-то похожее на кривую присутствует на изображении, и такая вероятность активировала фильтр. В правом верхнем углу значение у карты свойств будет 0, потому что на той части изображения не было ничего, что могло активировать фильтр (проще говоря, в этой области не было кривой). И это только для одного фильтра. Это фильтр, который обнаруживает линии с изгибом наружу. Могут быть другие фильтры для линий, изогнутых внутрь или просто прямых. Чем больше фильтров, тем больше глубина карты свойств, и тем больше информации мы имеем о вводной картинке.

Замечание: фильтр, о котором рассказано в этом примере, упрощён для упрощения математики свёртывания. На рисунке 2.6 видны примеры фактических визуализаций фильтров первого свёрточного слоя обученной сети. Но идея здесь та же. Фильтры на первом слое сворачиваются вокруг вводного изображения и

производят большие значения, когда специфическая черта, которую они ищут, есть во входном изображении.

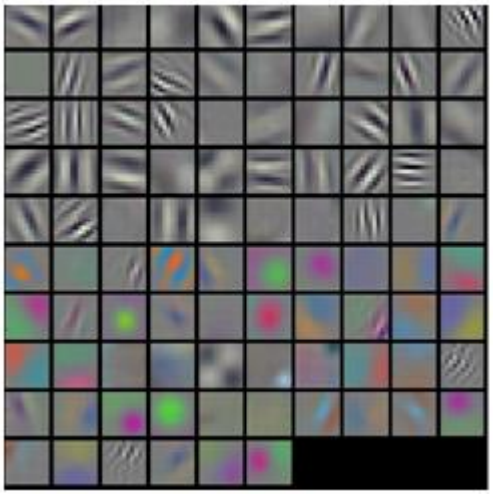

Рисунок 2.6. – Визуализация фактических фильтров

2.1.4 CHC. Идём глубже.

Сегодня классическая архитектура CHC выглядит примерно так:

$$Input \rightarrow Conv \rightarrow ReLU \rightarrow Conv \rightarrow ReLU \rightarrow Pool \rightarrow Conv \rightarrow ReLU \rightarrow \cdots \rightarrow FC$$

Где Input – наше входное изображение, Conv – свёрточный слой, ReLU(Rectifie Linear Unit) – функция активации, FC – полносвязный слой.

Мы уже поговорили о том, что делает первый свёрточный слой. А что же делает функция активации? Функция активации осуществляет нелинейное преобразование, благодаря которым оправдывается глубина CHC, потому что только линейными преобразованиями, которыми являются свёртки, результата не добиться.

ReLU — это преобразование max(x, 0). Вообще, существует множество различных функций активации. Почему же я выбрал именно ReLU для своих моделей? Во-первых, она наиболее вычислительно эффективна и, во-вторых, ReLU не “убивает” градиенты в положительной области, что хорошо сказывается на их вычислении при обратном ходе, а значит и на обучении (подробнее об этом в разделе 2.1.4).

Давайте вернёмся к более глубоким свёрточным слоям. Мы говорили о том, что умеют определять фильтры первого свёрточного слоя. Они обнаруживают свойства базового уровня, такие как, например, границы и кривые. Когда изображение проходит через один свёрточный слой, выход первого слоя становится вводным значением 2-го слоя. Теперь это сложнее визуализировать. Когда мы говорили о первом слое, вводом были только данные исходного изображения. Но когда мы перешли ко 2-му слою, вводным значением для него стали карты признаков — результат обработки предыдущим слоем. В случае со входным слоем для второго свёрточного слоя каждый набор вводных данных описывает позиции, где на исходном изображении встречаются определенные базовые признаки.

Теперь, когда применяются наборы фильтров поверх этого (пропускается картинка через второй свёрточный слой), на выходе будут активированы

фильтры, которые представляют свойства более высокого уровня. Типами этих свойств могут быть полукольца (комбинация прямой границы с изгибом) или квадратов (сочетание нескольких прямых ребер). Чем больше свёрточных слоёв проходит изображение и чем дальше оно движется по сети, тем более сложные характеристики выводятся в картах активации.

В интернете есть отличный проект по визуализации процессов, происходящих в СНС: yosinski.com/deepvis.

Про Pool и некоторые другие, не такие стандартные для СНС, техники, которые использовались в моих моделях, я расскажу в разделе 2.1.5.

А теперь давайте поговорим о том, благодаря чему это всё работает.

2.1.5 Обучение СНС

Способ, которым компьютер способен корректировать значения фильтров (или весов) — это обучающий процесс, который называют методом обратного распространения ошибки.

Но перед тем, как перейти к описанию данного метода, давайте перепишем описанные выше операции в матричном виде.

Как было показано ранее, прямое распространение для свёрточного слоя под номером k состоит из двух этапов:

1) Первым является вычисление промежуточного значения Z_k, которое получается в результате свертки входных данных A_{k-1} из предыдущего слоя с W_k фильтром и добавления смещения b_k(*замечание:* известно, что линейные преобразавания, коим является свёртка, можно представить в виде перемножения матриц):
$$Z_k = W_k * A_{k-1} + b_k$$
2) Второй – применение к Z_k нелинейной функции активации g_k:
$$A_k = g_k(Z_k)$$

Также стоит упомянуть еще одну ключевую вещь при обучении нейронных сетей – *функция потерь*. После прохождения прямого распространения нейронная сеть возвращает некоторый результат. Например, в случае с задачей поиска лица нейронная сеть возвращает 4 числа – координаты ограничительной рамки(обозначим их как xp_1, yp_1, xp_2, yp_2). Надо научить понимать компьютер, насколько это хороший результат. Пусть нам известны правильные ответы для данного изображения – x_1, y_1, x_2, y_2, тогда один из вариантов задания *функции потерь* это mse (среднеквадратичная ошибка):

$$\mathrm{L} = \frac{1}{4}((xp_1 - x_1)^2 + (yp_1 - y_1)^2 + (xp_2 - x_2)^2 + (yp_2 - y_2)^2)$$

Таким образом получаем, что чем меньше значение функции потерь, тем лучше нейронная сеть предсказала координаты. Так вот, по сути, цель метода обратного распространения ошибки – минимизировать функцию потерь путём изменения параметров сети (будем делать это градиентными методами).

Теперь можем переходить к обратному распространению ошибки для k-ого свёрточного слоя!

Замечание: ниже используется сокращённая запись частной производной.

Мы хотим оценить влияние изменения параметров на итоговую карту признаков, а затем и на конечный результат.

$$dA_k = \frac{\partial L}{\partial A_k}, dZ_k = \frac{\partial L}{\partial Z_k}, dW_k = \frac{\partial L}{\partial W_k}, db_k = \frac{\partial L}{\partial b_k}$$

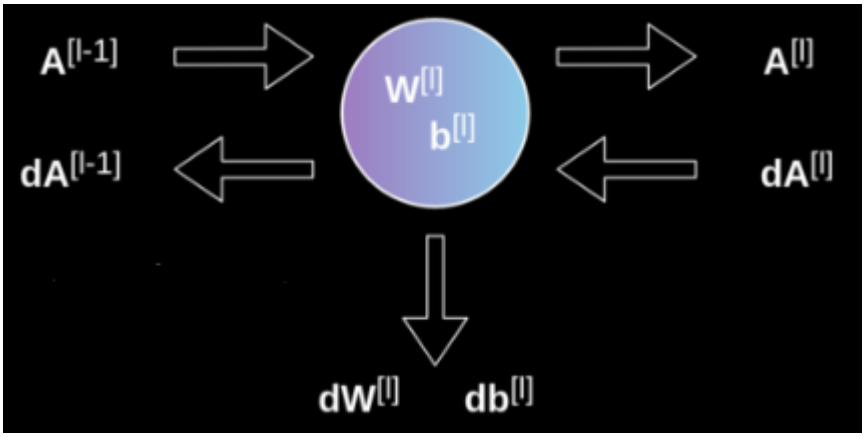

Рисунок 2.7. – Входные и выходные данные для свёрточного слоя при прямом и обратном распространении

Наша задача - рассчитать dW_k и db_k - производные, связанные с параметрами текущего слоя, а также значение dA_{k-1}, которое будет передано на предыдущий уровень. Как показано на рисунке 2.7, мы получаем dA_k в качестве входа. Размеры dW и W, db и b, а также dA и A соответственно одинаковы. Первым шагом является получение промежуточного значения dZ_k:

$$dZ_k = dA_k * g'(Z_k)$$

Теперь мы имеем дело с обратным распространением для самой свёртки, и для её вычисления мы будем использовать матричную операцию, называемую полной свёрткой, которая представлена ниже (на примере 2 на 2). (Рисунок 2.8)

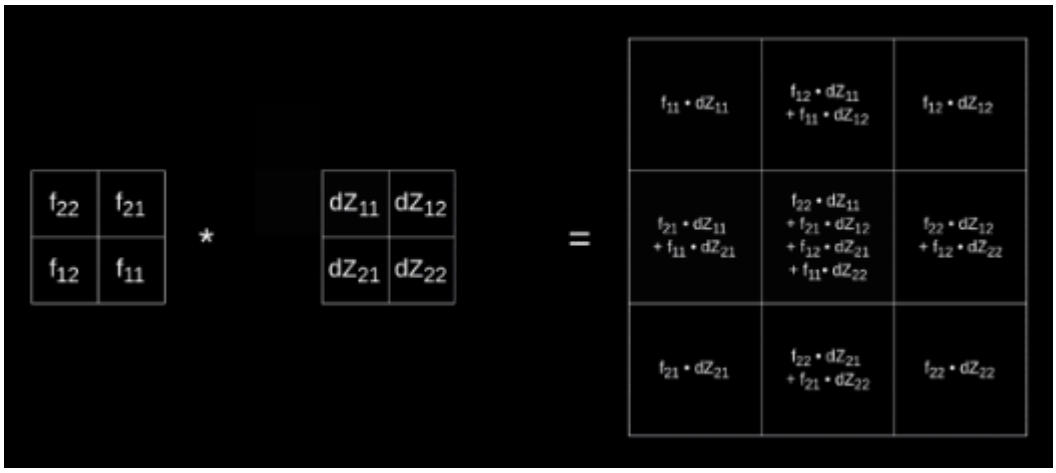

Рисунок 2.8. – Операция полной свёртки

Обратите внимание, что во время этого процесса мы используем фильтр, повёрнутый на 180 градусов. Эта операция может быть описана следующей формулой:

$$dA+= \sum_i \sum_j W * dZ[i][j]$$

где W - фильтр, а $dZ[i][j]$ получен из предыдущего слоя.

Стоит упомянуть, что в настоящее время нам не нужно беспокоиться об обратном распространении – deep learning-фреймфорки делают это за нас, но это очень полезно для понимания процессов, которые происходят внутри СНС.

2.1.6 Некоторые другие инструменты: нормализация изображений, batch нормализация, pool-слои

Нормализация изображений, как один из способов предобработки данных, гарантирует, что каждый входной параметр (в данном случае пиксель) имеет одинаковое распределение данных. Нормализацию можно осуществлять сразу по всему изображению или же по каждому RGB-каналу отдельно. Нормализация изображений достигается вычитанием среднего значения из каждого пикселя, а затем делением результата на среднеквадратичное отклонение. Для того, чтобы понимать, зачем это нужно, давайте вспомним как обучаются нейронные сети. В результате работы метода обратного распространения ошибки веса модифицируются так, чтобы уменьшить значение функции потерь. При обновлении весов мы пользуемся формулой вида: $W = W_i - \alpha \frac{dL}{dW}$, при том для всех весов α(lr, скорость обучения) постоянна. Отсюда и появляется необходимость в нормализации данных: если бы данные не были нормализованы, то диапозоны распределений бы значительно различались, что бы, в свою очередь, вызывало неравномерные изменения весов в каждом из направлений.

В некотором смысле аналогично работает batch нормализация. Сеть обучается методом обратного распространения ошибки, по батчам, то есть ошибка считается по какому-то подмножеству обучающей выборки. Стандартный способ нормировки — для каждого k рассмотрим распределение элементов батча. Вычтем среднее и поделим на дисперсию выборки, получив распределение с центром в 0 и дисперсией 1. Такое распределение позволит сети быстрее обучатся, т.к. все числа получатся одного порядка. Но ещё лучше ввести две переменные для каждого признака, обобщив нормализацию следующим образом:

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β
Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma \widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

Рисунок 2.9. – Операция нормализации

Это и есть batch normalization. Стоит упомянуть, что помимо ускорения скорости обучения batch normalization является одним из способов регуляризации.

И давайте, наконец, упомянем довольно часто используемые в CHC Pool-слои. Они используются в основном для уменьшения размеров карты признаков и ускорения вычислений. Эти слои довольно просто устроены - нам нужно разделить наше изображение на разные области, а затем выполнить некоторые операции для каждой из этих частей. Например, для MaxPooling-слоёв, которые использовались в моих моделях, мы выбираем максимальное значение из каждого региона и помещаем его в соответствующее место в выводе. Как и в случае со слоем свёртки, у нас есть два доступных гиперпараметра - размер региона и шаг. В наиболее стандартной реализации мы берём регионы 2 на 2 с шагом 2 и для каждого из регионов находим максимальное значение – его и записываем в качестве результата. Таким образом высота и ширина карты признаков уменьшается в 2 раза.

Рисунок 2.10. – Пример использования MaxPooling – слоя

Стоит упомянуть, что если Pooling выполняется для многоканальных изображений/карт признаков, то он должен производиться для каждого канала отдельно, таким образом Pool-слои уменьшают размеры карт признаков, но не уменьшают количество каналов.

2.1.7 Описание моей модели, основанной на CHC

Теперь, когда большинство использованных инструментов описаны, рад представить первую из моделей, основанную на CHC.

Её архитектура представлена на рисунке 2.11:

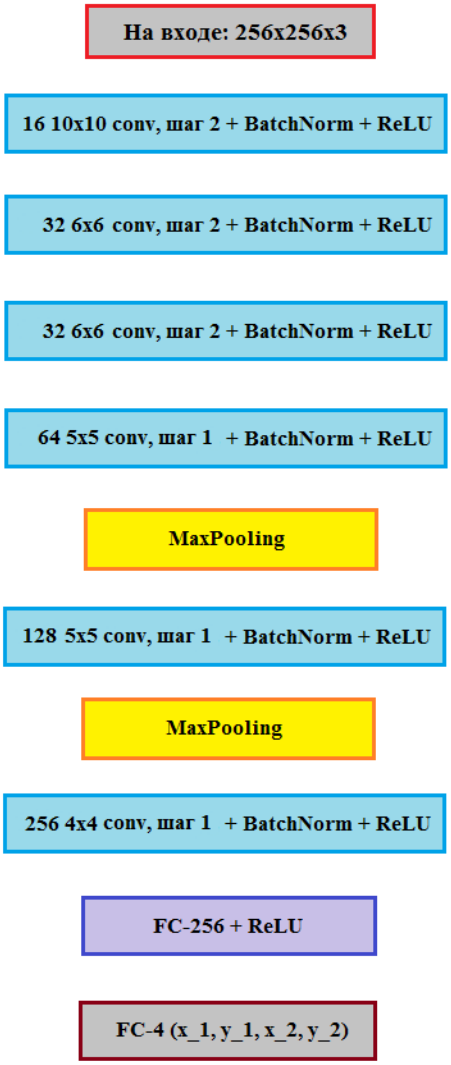

Рисунок 2.11. – Архитектура первой модели

Оптимизатор: Adam (lr = 1e-3)

Функция потерь: mse (средняя квадратичная ошибка)

Количество тренируемых параметров: 908788 (пока что не было цели оптимизировать данный показатель)

Данная сеть обучалась на 60 эпохах, около 6 часов на GPU

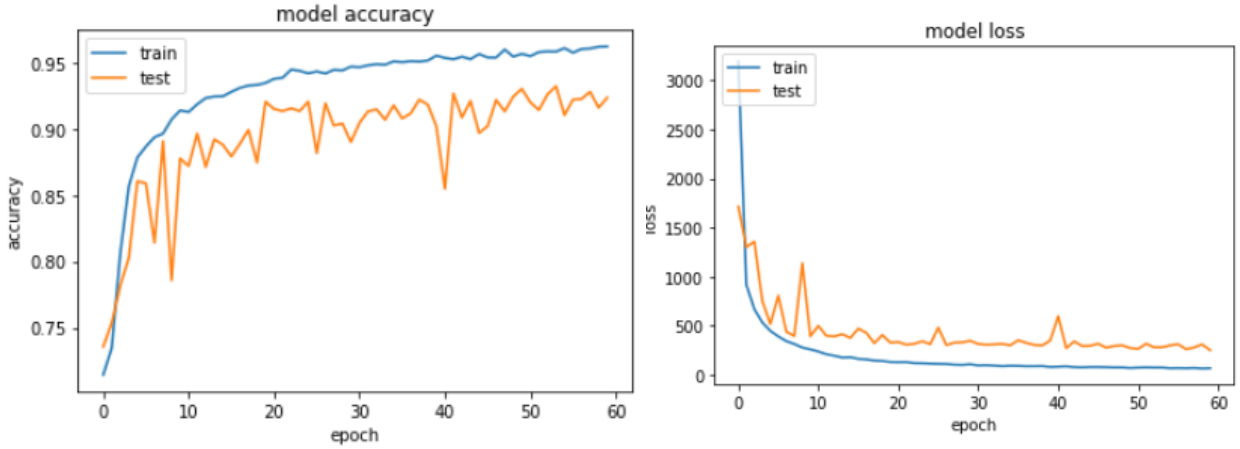

Рис. 2.12. - Графики зависимости точности на первой модели и значения функции потерь от номера эпохи обучения (оранжевой линией – показатели на validation set'e, синей – на train set'e)

Точность на тестовой выборке: 91.8 %
Пример работы модели представлен на рисунке 2.13:

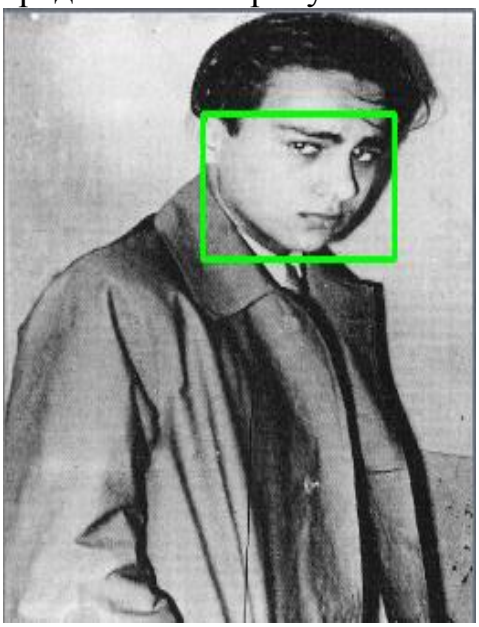

Рисунок 2.13. – Пример результата работы первой модели

2.2 Перенос обучения и тонкая настройка

2.2.1 Что такое перенос обучения и тонкая настройка

На практике люди довольно редко обучают всю свёрточную нейронную сеть “с нуля”, потому что это, во-первых, вычислительно дорого и занимает много времени (даже на GPU обучение может длиться неделями), а во-вторых, достаточно редко можно найти подходящий набор данных достаточно большого размера. Вместо этого люди используют такой инструмент, как *перенос обучения*. Его суть заключается в том, чтобы использовать предварительно обученные на очень большом наборе данных (например, ImageNet, который содержит 1,2 миллиона изображений с 1000 категориями) нейронные сети для решения своей задачи. Предварительно обученные свёрточные нейронные сети уже умеют выделять сложные фичи на изображениях, этим мы и пользуемся. Всё, что надо сделать – заменить последние полносвязные слои из обученной сети на новые, подходящие для нашей задачи и дообучить новую полносвязную часть на своих данных, в свою очередь свёрточная часть от старой сети остаётся неизменной.

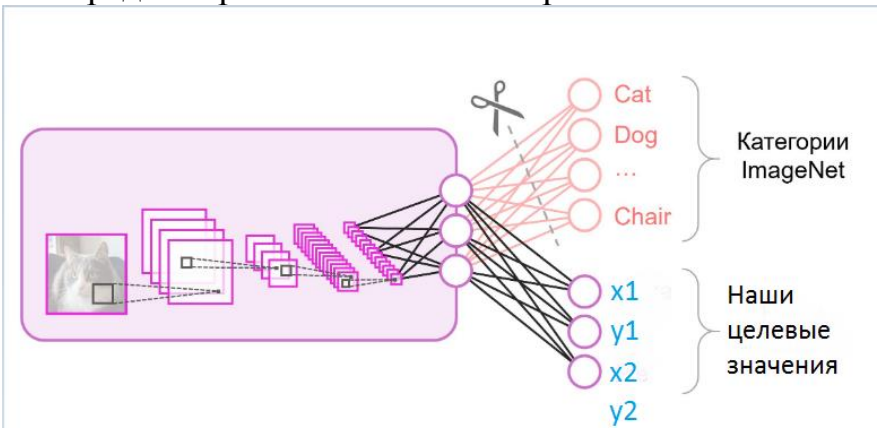

Рисунок 2.14. – Визуализация техники переноса обучения

Тонкая настройка сети (fine tuning) позволяет пойти дальше и еще больше увеличить качество работы предварительно обученной сети на новой задаче. Для этого обучаются не только новые полносвязные слои, который были добавлены в сеть, но и некоторые слои предварительно обученной нейронной сети. Это особенно эффективно, когда новый набор данных достаточно сильно отличается от исходного набора, на котором обучалась сеть.

Алгоритм тонкой настройки предварительно обученной нейронной сети

Для тонкой настройки сети необходимо выполнить следующие действия:

1) Заменить последние полносвязные слои предварительно обученной нейронной сети новыми полносвязными слоями, подходящим под нашу задачу.
2) “Заморозить” сверточные слои предварительно обученной нейронной сети. В результате эти слои не будут обучаться.
3) Провести обучение составной сети с новыми полносвязными слоями на новом наборе данных.
4) “Разморозить” несколько слоев сверточной части предварительно обученной нейронной сети.
5) Дообучить сеть с размороженными сверточными слоями на новом наборе данных. Именно этот этап и называется тонкой настройкой (fine tuning).

Тонкую настройку можно проводить только после того, как обучены новые полносвязные слои. Изначально веса в полносвязных слоях назначаются случайным образом, поэтому на первых этапах обучения сигнал об ошибке будет очень большой. Если этот сигнал распространится в сверточную часть сети, то результат предварительного обучения может быть утерян, т.к. веса нейронов будут сильно меняться. Когда же обучение полносвязных слоёв на новом наборе данных завершено, то таких сильных изменений весов уже не будет, и можно переходить к обучению сверточной части.

2.2.2 Описание моей модели, основанной на переносе обучения и тонкой настройке

Перейдём к описанию модели, построенной на основе переноса обучения и тонкой настройки.

За базовую модель я взял архитектуру VGG-16, обученную на ImageNet’e. Главное преимущество VGG-16 состоит в том, что она довольно просто устроена. Есть архитектуры, которые показывали на ImageNet’e намного лучшие результаты при меньшем количестве обучаемых параметров, но для того, чтобы использовать тонкую настройку надо было разобраться в их архитектурах, которые зачастую довольно сложные и очень глубокие. К примеру Inception-v3 содержит 205 слоёв.

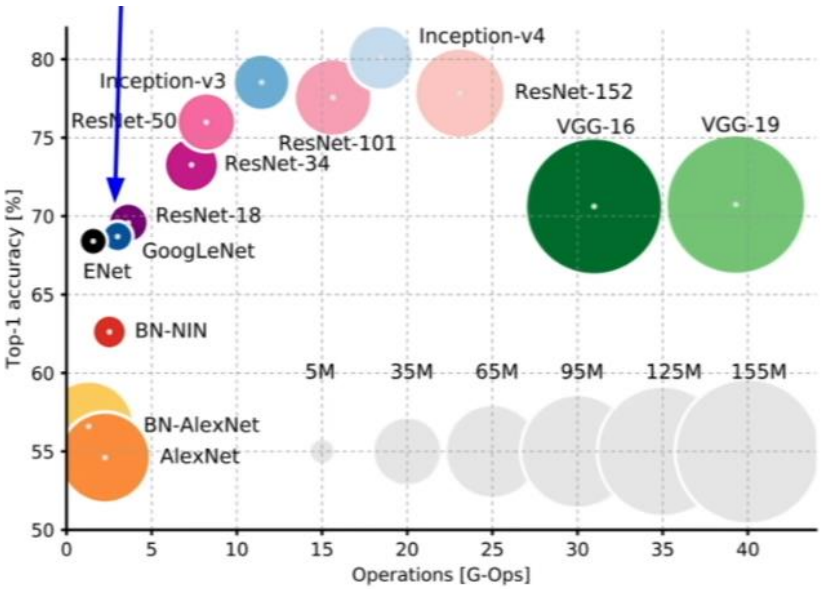

Рисунок 2.15. – Различные базовые модели, обученные на ImageNet'е, представлена зависимость точности от количества операций

Безусловно, один из способов улучшения построенной модели – замена базовой модели на, например, Inception-v3.

Итак, с базовой архитектурой определились, давайте продолжим.

Как описано в 2.2.1, первый шаг - замена последних полносвязных слоёв VGG-16 на новые полносвязные слои, подходящие под нашу задачу:

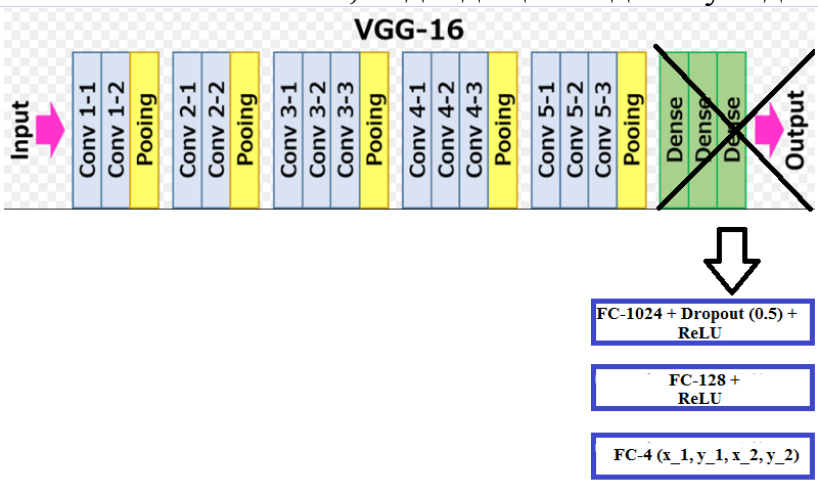

Рисунок 2.15. – Архитектура второй модели

Шаг 2 - обучаем полносвязные слои на 15 эпохах - получаем точность около 82 %.
Оптимизатор: Adam (lr = 5e-5)
Функция потерь: mse (mean squared error)

Шаг 3 - дообучаем последний, пятый свёрточный блок, процесс обучения сходится за 30 эпох.
Оптимизатор: Adam (lr = 1e-5)
Функция потерь: mse

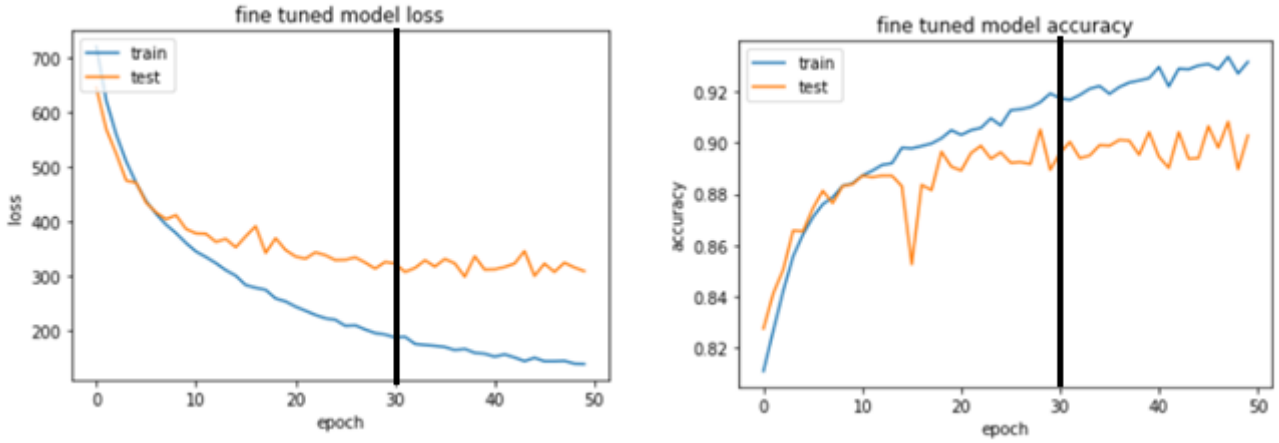

Рис. 2.16. - Графики зависимости значения функции потерь и точности на второй модели от номера эпохи обучения (оранжевой линией – показатели на validation set'е, синей – на train set'е)

В результате получаем точность на тестовой выборке: 89.7 %, а всё обучение длилось около 3 часов на GPU, что значительно меньше, чем в случае с первой моделью (6 часов).

2.3 Сравнение моделей

Результаты сравнения моделей представлены в таблице ниже (Таблица 2.1).

Таблица 2.1. – Сравнение построенных моделей

	Точность на тестовой выборке	Время обучения	Кол-во эпох, за которое сошёлся процесс обучения
Модель 1	91.8 %	6 часов	60
Модель 2	89.7 %	3 часа	15 + 30

Обе модели работают в среднем за 0.7-0.8 секунд на отдельно взятом изображении, что довольно много, данный показатель планируется улучшить за счёт уменьшения количества параметров в сети и некоторых других техник оптимизации нейронных сетей.

2.4 Выводы и предложения по улучшению моделей

В данной главе описаны основные принципы работы СНС. Свёрточные слои вместе с функциями активации позволяют выделять различные формы на изображениях, а в случае с глубокими СНС – весьма сложные формы, чем мы и пользовались при решении задачи поиска лица на изображении. Также СНС имеют очень хорошую обобщающую способность, что позволяет им распознавать различные вариации одного и того же объекта, обучаясь на ограниченном наборе данных. Помимо свёрток, в главе описаны такие инструменты, как нормализация изображений, batch нормализация, pool-слои, а также такие техники, как перенос обучения, тонкая настройка и т.д. Данные инструменты и техники легли в основу построенных моделей.

Первая из моделей (обученная "с нуля") лучше показала себя на тестовой выборке, что, в первую очередь, связано с тем, что первая и вторая модели имеют

примерно одинаковую глубину, но большая часть второй модели была обучена на другом наборе данных (ImageNet’e). Из этого можно сделать вывод, что когда наш набор данных довольно сильно отличается от набора, на котором была обучена базовая модель, надо дообучать большее число свёрточных блоков исходной базовой модели. Тем не менее вторая модель показала довольно хорошие результаты на тестовой выборке (точность – 89.7 % против 91.8 % первой модели). Один из вариантов её улучшения – изменение базовой модели на, например, Inception-v3, которая показывает лучшие результаты на ImageNet’e при меньшем количестве обучаемых параметров. В то же время сеть Inception-v3 намного глубже и в её архитектуре сложнее разобраться, но мы ведь любим сложные эффективные архитектуры! Помимо вышеупомянутого, процесс обучения второй модели сошелся значительно быстрее и в целом обучение заняло значительно меньше времени, что и является одним из основных преимуществ переноса обучения и тонкой настройки.

Также стоит упомянуть, что в основу обучения легли градиентные методы (у обеих моделей оптимизатор – Adam), у которых есть проблема схождения к локальным минимумам. Существуют различные техники для борьбы с данной проблемой, однако в своей работе я не испробовал их. Применение таких техник – также один из вариантов улучшения моделей.

ГЛАВА 3
РАЗРАБОТКА ПРОГРАММНОГО ИНСТРУМЕНТАРИЯ

3.1 Использовавшиеся технологии.

При написании моделей я использовал язык Python. Python отлично подходит для исследовательских проектов в сфере машинного обучения. Для него написано множество различных полезных бибилиотек и фреймворков: тут и фреймфорки для разработки в сфере глубоко обучения, различные библиотеки для работы с данными, для визуализации и т.д. Всё это значительно ускоряет процесс разработки. В то же время Python довольно легко освоить, а код получается очень читабельным, что также немаловажно.

Теперь давайте рассмотрим более детально использование некоторых библиотек.

При подготовке обучающей выборки мне помогли такие библиотеки, как:

1) csv
   Во-первых, различные данные о картинках из набора данных UMDFaces хранились в csv-файлах, и ,во-вторых, нет возможности хранить такое большое количесвто данных(22000 изображений) в оперативной памяти компьютера, поэтому на помощь снова приходят csv-файлы и метод flow_from_dataframe из keras. В csv-файлах обучающая выборка хранилась в виде название изображения (id-строка) и соответсвующие данному изображению ответы.
2) os, glob для работы с операционной системой.
3) cv2 для считывания и выделения дополнительной информации об изображениях.

Набор данных для обучения подготовлен, можем переходить к построению моделей и их обучению.

В качестве основного фреймворка для построениея и обучения моделей использовался keras. Этот фреймворк содержит многочисленные реализации широко применяемых строительных блоков нейронных сетей, таких как слои, целевые и передаточные функции, оптимизаторы, и множество инструментов для упрощения работы с изображениями и текстом.

При использовании keras нам не нужно заниматься высчитыванием градиентов и прочими вещами, которые нужны при обучении, keras делает это за нас. По сути, keras лишь требует задать архитектуру сети, функцию потерь, оптимизатор, lr и прочие гиперпараметры, а остальные тонкости обучения он выполнит за тебя. Помимо вышеупомянутых преимуществ, keras содержит огромное множество других возможностей для работы с данными и нейронными сетями, например ModelCheckpoint и EarlyStopping, которые позволяют сохранять лучшие (по некоторому критерию, например по точности на validation set’e) из весов.

```
Epoch 00002: val_loss improved from 645.20841 to 568.81200, saving model to tl_weights_fine_tuned.h5
```

Или, например, вот так просто можно загрузить данные для обучения (Рисунок 3.1):

```
traindf = pd.read_csv(train_csv, dtype={'id': str, 'x1': np.int32, 'y1': np.int32, 'x2': np.int32, 'y2': np.int32})
testdf = pd.read_csv(test_csv, dtype={'id': str, 'x1': np.int32, 'y1': np.int32, 'x2': np.int32, 'y2': np.int32})

datagen = ImageDataGenerator(rescale=1./255., validation_split=0.2)

train_dir = r'/content/drive/My Drive/Colab Notebooks/face_detection/data/data_UMD/train_test/train'
test_dir = r'/content/drive/My Drive/Colab Notebooks/face_detection/data/data_UMD/train_test/test'

train_generator = datagen.flow_from_dataframe(dataframe=traindf, directory=train_dir, x_col="id",
                                              y_col = ["x1", "y1", "x2", "y2"],
                                              subset='training', batch_size=64, seed=42,
                                              shuffle=True, class_mode="other",
                                              target_size=(256, 256))
```

Рисунок 3.1. – Загрузка данных в keras

Еще один большой плюс keras состоит в том, что обучение можно запускать на GPU, что значительно ускоряет процесс. Также нельзя не отметить простоту и читабельность кода на keras, убедитесь сами на примере задания архитектуры сети (Рисунок 3.2)

```
model = Sequential()

model.add(Conv2D(32, (12, 12), strides=4, input_shape=(256, 256, 3)))
model.add(layers.BatchNormalization())
model.add(Activation('relu'))
model.add(Conv2D(64, (4, 4), strides=2))
model.add(layers.BatchNormalization())
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Conv2D(64, (5, 5), strides=2))
model.add(layers.BatchNormalization())
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Flatten())
model.add(Dense(100, activation='relu'))
model.add(Dense(4, activation='linear'))

model.compile(optimizers.Adam(lr=5e-4), loss='mean_squared_error', metrics=['accuracy'])
```

Рисунок 3.2. – Задание архитектуры сети в keras

Также я использовал библиотеку pandas для загрузки данных из csv, а для визуализации различных графиков зависимостей использовался matplotlib. Tensorflow использовался в качестве вычислительного бэкенда keras.

3.2 Выводы

При написании практической части курсовой работы использовался язык Python. Python отлично подходит для исследовательских проектов в сфере машинного обучения. Для него написано множество различных полезных бибилиотек и фреймворков: тут и фреймфорки для разработки в сфере глубоко обучения, различные библиотеки для работы с данными, для визуализации и т.д. В то же время Python довольно легко освоить.

При написании моделей я использовал фреймворк keras с tensorflow в качестве вычислительного бэкенда. Keras – один из самых простых и удобных фреймворков для построения и обучения нейронных сетей. Одно из главных

преимуществ Keras – возможность запускать обучение на GPU, что значительно ускоряет процесс. Также в keras есть множество других полезных инструментов, некоторые из которых описаны в пункте 3.1.

ЗАКЛЮЧЕНИЕ

Для решения задачи поиска лица на изображении были изучены основные принципы работы свёрточных нейронных сетей. В работе описаны процесс подготовки обучающей выборки и основные принципы работы СНС. Также было построено 2 модели на основе СНС. Первая из моделей была обучена “с нуля” на предварительно подготовленном наборе данных. В основу второй модели легли техники переноса обучения и тонкой настройки. Первая из моделей показала лучшие результаты на тестовой выборке (точность 91.8%, точность у второй модели – 89.7%). В то же время обучение второй модели заняло значительно меньше времени, 3 часа, у первой модели обучение длилось 6 часов. В пункте 2.4 были предложены варианты улучшения второй модели.

Таким образом для решения прикладной задачи поиска лица на изображении был подготовлен набор размеченных данных и по нему было обучено 2, основанные на СНС, модели.

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. В.А. Головко, В.В. Краснопрошин Нейросетевые технологии обработки данных: учебное пособие – Минск: БГУ, 2017. – 264 с.

2. М. Нильсен Нейронные сети и глубинное обучение

Интернет ресурсы:

1. Курс CS231n от Stanford University: https://youtube.com/watch?v=vT1JzLTH4G4&list=PL3FW7Lu3i5JvHM8ljYj-zLfQRF3EO8sYv

2. https://towardsdatascience.com/transfer-learning-from-pre-trained-models-f2393f124751

3. https://habr.com/ru/post/309508/

4. https://towardsdatascience.com/gentle-dive-into-math-behind-convolutional-neural-networks-79a07dd44cf9

ПРИЛОЖЕНИЕ А

Некоторые части реализации подготовки обучающей выборки

```
def make_dir_if_not_exist(directory):
    if not os.path.exists(directory):
        os.makedirs(directory)
```

```
def parse_UMD(image_dir, anno_dir, train_size, test_size):
    anno_file = os.path.join(anno_dir, 'umdfaces_batch3_ultraface.csv')
    dataset = []

    num_train_test_size = train_size + test_size

    count = 0
    with open(anno_file) as csvfile:
        readCSV = csv.reader(csvfile, delimiter=',')
        next(readCSV)
        for row in readCSV:
            per_data_info = []

            img_name = row[1]
            face_x = row[4]
            face_y = row[5]
            face_width = row[6]
            face_height = row[7]

            per_data_info.append(img_name)
            per_data_info.append(face_x)
            per_data_info.append(face_y)
            per_data_info.append(face_width)
            per_data_info.append(face_height)

            if count % 500 == 0:
                print(count)

            count += 1
            dataset.append(per_data_info)
```

```
def generate_images_and_csv (image_dir, save_images_dir, save_csv, dataset, start_idx, data_size):
    csv_data = [['id', 'x', 'y', 'w', 'h']]

    for i in range(start_idx, start_idx + data_size):
        per_data_info = dataset[i]
        full_name = per_data_info[0]
        index_of_slash = full_name.find('/')
        img_name = full_name[index_of_slash+1:]
        folder_name = img_name[:index_of_slash]
        x = int(float(per_data_info[1]))
        y = int(float(per_data_info[2]))
        w = int(float(per_data_info[3]))
        h = int(float(per_data_info[4]))

        folder_path = os.path.join(image_dir, folder_name)

        for filename in glob.glob(folder_path + '\*.jpg'):
            length = len(img_name)
            filename_short=filename[len(filename)-length:]

            if(filename_short == img_name):
                img = cv2.imread(folder_path + '\\' + img_name)
                img_height, img_width = img.shape[:2]

                x = int(x * 256 / img_width)
                y = int(y * 256 / img_height)
                w = int(w * 256 / img_width)
                h = int(h * 256 / img_height)

                cv2.imwrite(save_images_dir + '\\' + img_name, img)

                row = [img_name, x, y, w, h]
                csv_data.append(row)

    with open(save_csv, 'w', newline='') as csvFile:
        writer = csv.writer(csvFile)
        writer.writerows(csv_data)
```

```
def new_csv(csv_dir, new_csv_dir):
    csvfile = open(csv_dir)
    new_csv = open(new_csv_dir, 'w', newline='')

    reader = csv.reader(csvfile, delimiter=',')
    writer = csv.writer(new_csv)

    next(reader)
    writer.writerow(['id', 'x1', 'y1', 'x2', 'y2'])
    for row in reader:
        img_name = row[0]
        x = int(row[1])
        y = int(row[2])
        w = int(row[3])
        h = int(row[4])
        writer.writerow([img_name, str(x), str(y), str(x + w), str(y + h)])

    csvfile.close()
    new_csv.close()

new_csv(r'D:\jupyter_projects\Face Detection\data\umdfaces_batch3\data_UMD\train_test\testLabels.csv',
        r'D:\jupyter_projects\Face Detection\data\umdfaces_batch3\data_UMD\train_test\test.csv')
```