МИНОБРНАУКИ РОССИИ
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«МИРЭА - Российский технологический университет»
РТУ МИРЭА

Институт кибернетики
Кафедра высшей математики

ОТЧЕТ ПО КУРСОВОЙ РАБОТЕ

по дисциплине «Алгоритмы и теория сложности»

Тема курсовой работы: Проверка корректности и наличия медицинских масок в режиме реального времени.

Студент группы КМБО-04-19 Пашшоев Б.

_____
(подпись студента)

Руководитель курсовой работы доцент Бескин А.Л.

_____
(подпись руководителя)

Работа представлена к защите «21» декабря 2021 г.

Допущен к защите «21» декабря 2021 г.

Москва 2021

МИНОБРНАУКИ РОССИИ
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«МИРЭА - Российский технологический университет»
РТУ МИРЭА

Институт кибернетики
Кафедра высшей математики

Утверждаю
Заведующий кафедрой ВМ

_____ Худак Ю.И.
«___» _____ 2021 г.

ЗАДАНИЕ

на выполнение курсовой работы по дисциплине

«Алгоритмы и теория сложности»

Студент Пашшоев Бахтиёржон Группа КМБО-04-19

Тема работы: Проверка корректности и наличия медицинских масок в режиме реального времени.
Исходные данные: Иследовать и реализовать алгоритм глубоко обучение на наборе данных MaskedFace-Net.
Перечень вопросов, подлежащих разработке, и обязательного графического материала:

1. Проанализировать возможности алгоритмов Компьютерного зрения.
2. Исследовать и изучить особенности свёрточных нейронных сетей.
3. Создание и обучение нейросети на наборе данных.
4. Применение алгоритмов. Примеры.
5. Отчет по курсовой работе в виде расчетно-пояснительной записки.

Срок представления к защите курсовой работы: до «21» декабря 2021 г.

Задание на курсовую работу выдал _____ (Бескин А.Л.)
«14» сентября 2021 г.

Задание на курсовую работу получил _____ (Пашшоев Б.)

Оглавление

1. Основные задачи глубокого обучения

Первым шагом к пониманию того, как работает глубокое обучение, является понимание различий между несколькими важными терминами.

Нейронная сеть (искусственная нейронная сеть) — это попытка воспроизведения работы человеческого мозга на компьютере при помощи слоев нейронов.

Искусственный интеллект — способность машины или программы находить решения при помощи вычислений.

Во время первых исследований в области ИИ ученые пытались воспроизвести человеческий интеллект для решения конкретных задач — например, игры с человеком. Было введено большое количество правил, которым должен следовать компьютер. На основе этих правил компьютер принимал решения в согласии с конкретным списком возможных действий.

Машинное обучение — это попытка научить компьютеры самостоятельно обучаться на большом количестве данных вместо жестко постулированных правил. Машинное обучение позволяет компьютерам самостоятельно обучаться. Это возможно благодаря вычислительной мощности современных компьютеров, которые могут легко обрабатывать большие наборы данных.

Глубокое обучение — это метод машинного обучения. Глубокое обучение позволяет обучать модель предсказывать результат по набору входных данных. Для обучения сети можно использовать как контролируемое, так и неконтролируемое обучение.

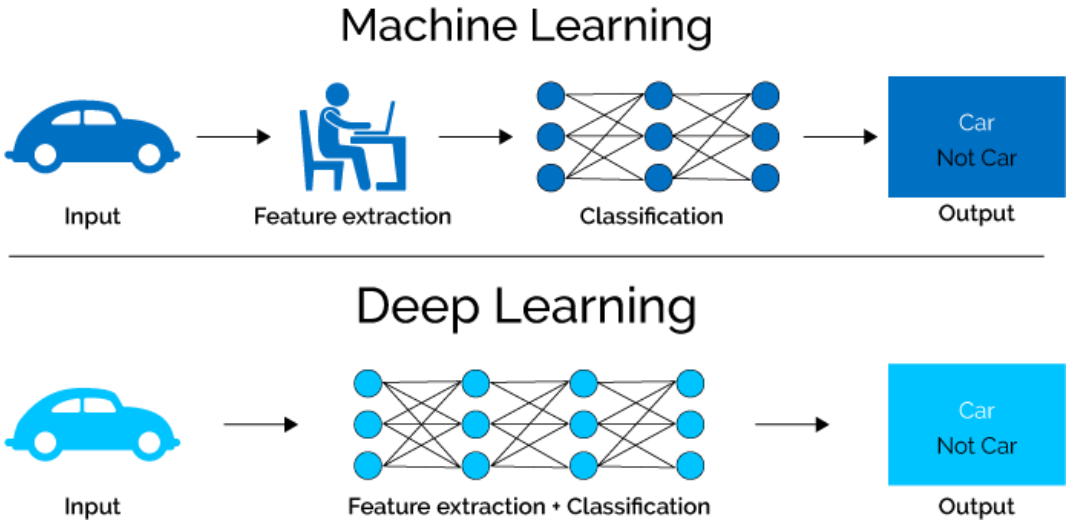


Рисунок 1 - Разница между машинным и глубоким обучением.

Давайте заглянем внутрь нашей модели. Как и у животных, искусственная нейронная сеть содержит взаимосвязанные нейроны. На диаграмме они представлены кругами:

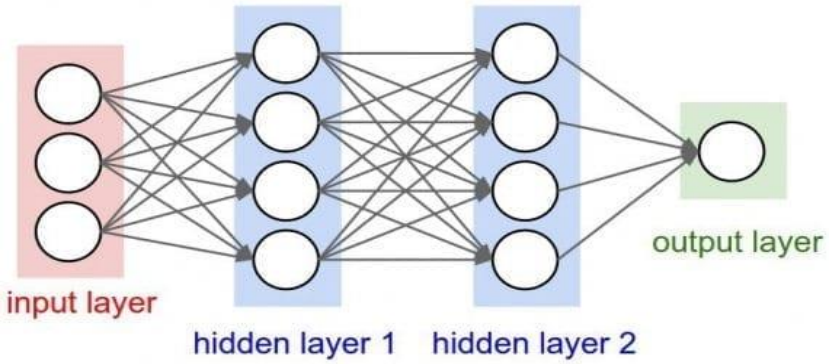


Рисунок 2 - Глубокая нейронная сеть.

Нейроны сгруппированы в три различных типа слоев:

- входной слой;
- скрытый слой (слои);
- выходной слой.

Входной слой принимает входные данные. В нашем примере рис.2 имеется четыре нейрона на входном слое

Скрытые слои выполняют математические вычисления со входными данными. Одна из задач при создании нейронных сетей — определение количества скрытых слоев и нейронов на каждом слое.

Слово «**глубина**» в термине «глубокое обучение» означает наличие более чем одного скрытого слоя.

Выходной слой выдает результат: на нашем рисунке содержит только 1 нейрон.

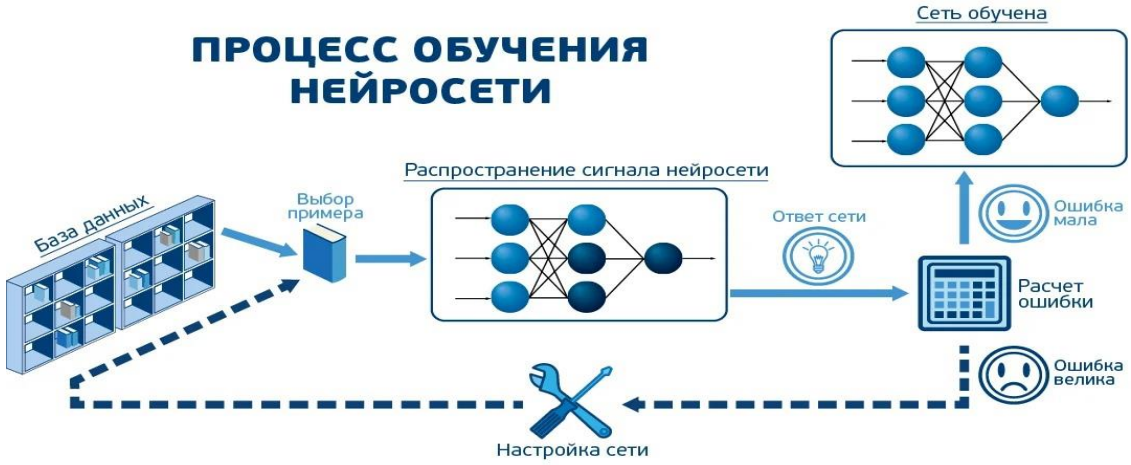


Рисунок 2.1 - Процесс обучения нейронной сети.

Сфера применения ИИ достаточно широка. Она охватывает как известные технологии, так и новые направления, о которых многие даже не слышали. Это широкий спектр решений, от пылесосов до космических станций. ИИ не стоит воспринимать как нечто монолитное и неделимое. Некоторые области, в которых применяется ИИ, становятся новыми подсекторами экономики и способствуют ее дальнейшему развитию. ИИ используется в здравоохранении, розничной торговле и многих других сферах повседневной жизни.

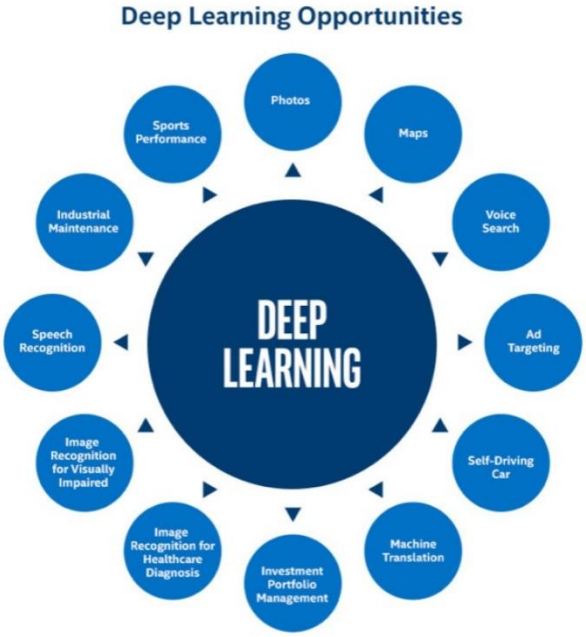


Рисунок 3 - Сферы использования глубокого обучения.

В ближайшее время людей будет окружать еще больше «умных вещей». Вероятно, они станут меньше и эффективнее. Вопреки распространенному мнению, искусственный интеллект, скорее всего, создаст больше рабочих мест, а не сократит их, хотя это могут быть профессии, ранее не существовавшие. Чем большее количество людей используют ИИ, тем большее доверие он вызывает. Он уже стал неотъемлемой частью нашей жизни. Как и в случае с любой новой технологией, ИИ требует тщательного тестирования перед внедрением. Тестеры испытывают технологию на прочность, чтобы убедиться в том, что она работает бесперебойно.

2. Знакомство с работой нейронных сетей

2.1 Знакомство с работой нейронных сетей

Искусственная нейронная сеть обычно обучается с учителем. Это означает наличие обучающего набора (датасета), который содержит примеры с истинными значениями: тегами, классами, показателями.

Замечание. Неразмеченные наборы также используют для обучения нейронных сетей, но мы не будем здесь это рассматривать.

Например, если вы хотите создать нейросеть для оценки тональности текста, **датасетом** будет список предложений с соответствующими каждому эмоциональными оценками. Тональность текста определяют **признаки** (слова, фразы, структура предложения), которые придают негативную или позитивную окраску. **Веса** признаков в итоговой оценке тональности текста (позитивный, негативный, нейтральный) зависят от математической функции, которая вычисляется во время обучения нейронной сети.

Раньше люди генерировали признаки вручную. Чем больше признаков и точнее подобраны веса, тем точнее ответ. Нейронная сеть автоматизировала этот процесс.

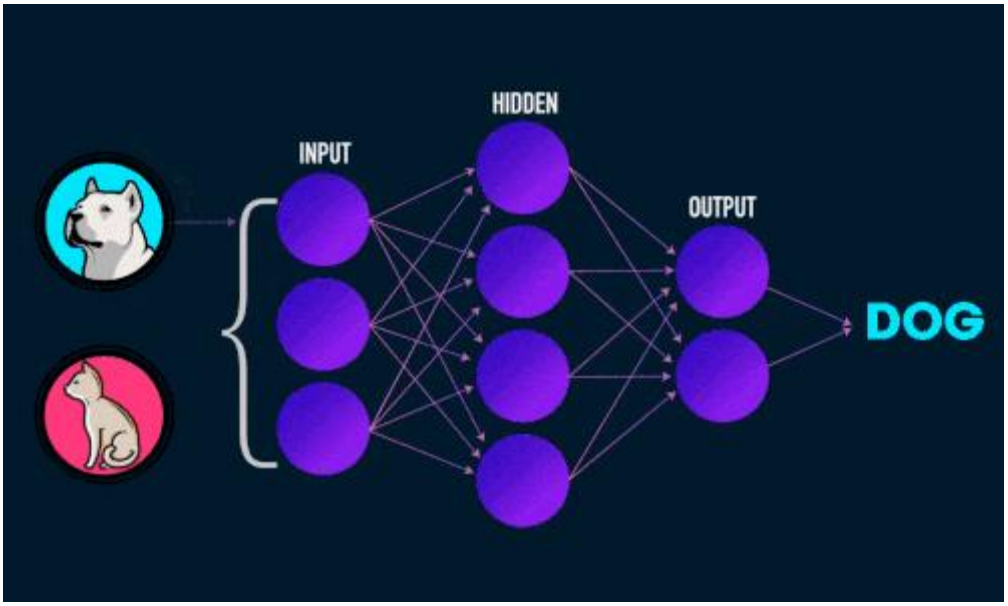


Рисунок 4 - Демонстрация работы нейронной сети.

Искусственная нейронная сеть состоит из трех компонентов:

- Входной слой;
- Скрытые (вычислительные) слои;
- Выходной слой.

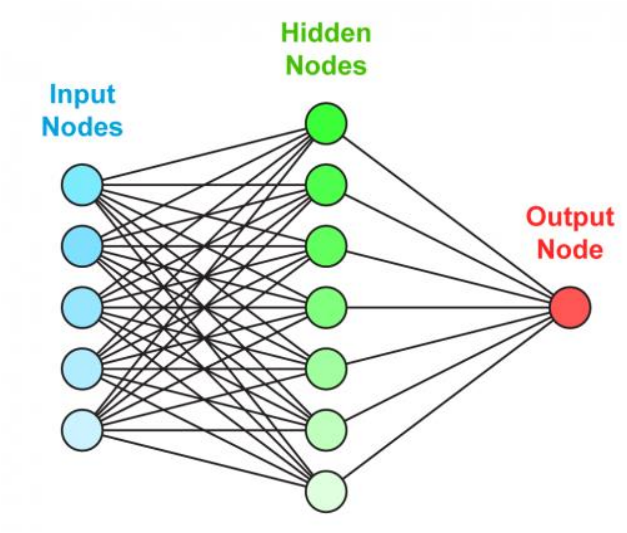


Рисунок 5 - Строение нейронной сети.

2.2 Обучение нейронной сети

Обучение нейронной сети – это процесс, в котором параметры нейронной сети настраиваются посредством моделирования среды, в которую эта сеть встроена. Тип обучения определяется способом подстройки параметров.

Обучение нейросетей происходит в два этапа:

- Прямое распространение ошибки;
- Обратное распространение ошибки.

2.2.1 Прямое распространение ошибки

Во время прямого распространения ошибки делается предсказание ответа. При обратном распространении ошибка между фактическим ответом и предсказанным минимизируется.

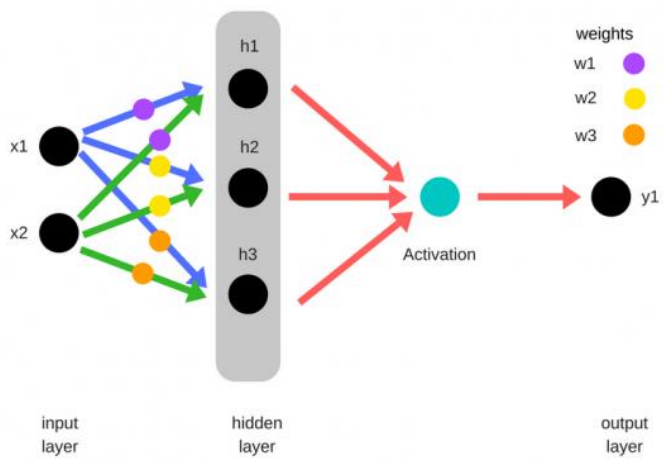


Рисунок 6 - Прямое распространение.

Зададим начальные веса случайным образом:

- w1
- w2

Умножим входные данные на веса для формирования скрытого слоя:

- h1 = (x1 * w1) + (x2 * w1)
- h2 = (x1 * w2) + (x2 * w2)
- h3 = (x1 * w3) + (x2 * w3)

Выходные данные из скрытого слоя передается через нелинейную функцию (функцию активации), для получения выхода сети:

- y_ = f(h1 , h2, h3)

2.2.2 Обратное распространение

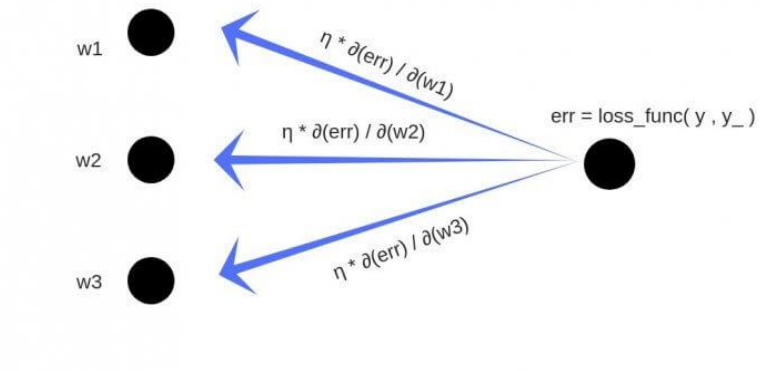


Рисунок 7 - Обратное распространение.

- Суммарная ошибка (total_error) вычисляется как разность между ожидаемым значением «y» (из обучающего набора) и полученным

значением «y_» (посчитанное на этапе прямого распространения ошибки), проходящих через функцию потерь (cost function).
- Частная производная ошибки вычисляется по каждому весу (эти частные дифференциалы отражают вклад каждого веса в общую ошибку (total_loss)).
- Затем эти дифференциалы умножаются на число, называемое скоростью обучения или learning rate (η).

Полученный результат затем вычитается из соответствующих весов.

В результате получатся следующие обновленные веса:

- w1 = w1 — (η * ∂(err) / ∂(w1))
- w2 = w2 — (η * ∂(err) / ∂(w2))
- w3 = w3 — (η * ∂(err) / ∂(w3))

То, что мы предполагаем и инициализируем веса случайным образом, и они будут давать точные ответы, звучит не вполне обоснованно, тем не менее, работает хорошо.

Смещения – это веса, добавленные к скрытым слоям. Они тоже случайным образом инициализируются и обновляются так же, как скрытый слой. Роль скрытого слоя заключается в том, чтобы определить форму базовой функции в данных, в то время как роль смещения – сдвинуть найденную функцию в сторону так, чтобы она частично совпала с исходной функцией.

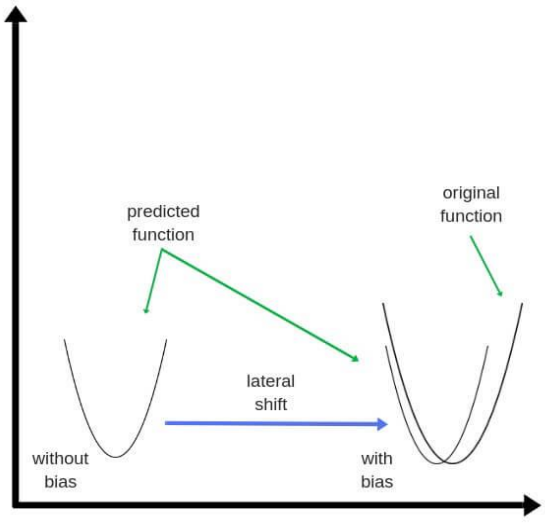


Рисунок 8 - Смещение.

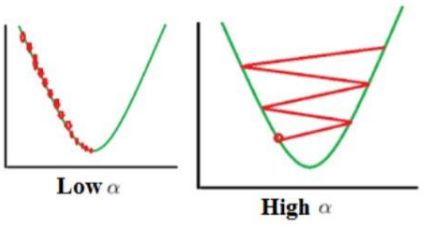
2.2.3 Скорость обучения

В машинном обучении скорость обучения является параметром настройки в алгоритме оптимизации, который определяет размер шага на каждой итерации, приближаясь к минимуму функции потерь . Поскольку он влияет на то, в какой степени вновь полученная информация перекрывает старую информацию, он метафорически представляет скорость, с которой модель машинного обучения «учится».

При установке скорости обучения существует компромисс между скоростью сходимости и превышением. Хотя направление спуска обычно определяется из градиента функции потерь, скорость обучения определяет, насколько большой шаг будет сделан в этом направлении. Слишком высокая скорость обучения заставит обучение перескочить через минимумы, но слишком низкая скорость обучения либо займет слишком много времени, чтобы сойтись, либо застрянет в нежелательном локальном минимуме.

Чтобы достичь более быстрой сходимости, предотвратить колебания и застревания в нежелательных локальных минимумах, скорость обучения часто изменяется во время обучения либо в соответствии с графиком скорости обучения, либо с использованием адаптивной скорости обучения.

Learning rate (скорость обучения) – один из важнейших гиперпараметров обучения, по сути, определяет, как быстро мы будем шагать по функции потерь. Слишком быстро – пролетим мимо минимума, слишком медленно – застрянем в плохом локальном минимуме:



\

Рисунок 9 - Скорость обучения.

2.2.4 Функция активации

Функция активации — это один из самых мощных инструментов, который влияет на силу, приписываемую нейронным сетям. Отчасти, она определяет, какие нейроны будут активированы, другими словами и какая информация будет передаваться последующим слоям.

Без функций активации глубокие сети теряют значительную часть своей способности к обучению. Нелинейность этих функций позволяет моделью приближать более сложные функции чем просто прямую линию.

Ниже приведены примеры распространенных функций активации:

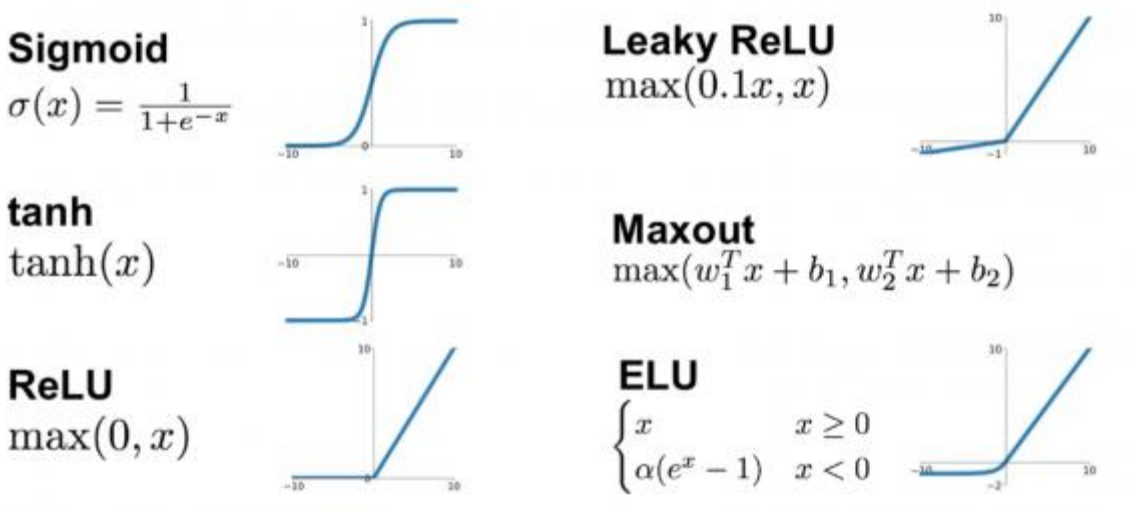


Рисунок 10 - Функции активации.

2.2.5 Функция потерь

Функция потерь находится в центре нейронной сети. Она используется для расчета ошибки между реальными и полученными ответами. Наша глобальная цель — минимизировать эту ошибку. Таким образом, функция потерь эффективно приближает обучение нейронной сети к этой цели.

Функция потерь измеряет «насколько хороша» нейронная сеть в отношении данной обучающей выборки и ожидаемых ответов.

Некоторые примеры функции потерь:

- Квадратичная (среднеквадратичное отклонение) MSE;
- Кросс-энтропия;
- Экспоненциальная (AdaBoost);
- Расстояние Кульбака — Лейблера.

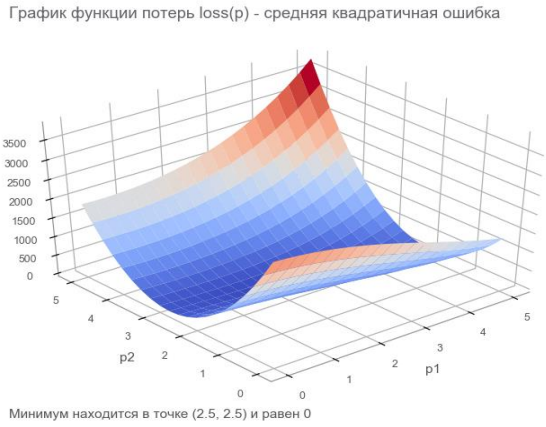


Рисунок 11 - График функции потерь MSE.

2.3 Глубокие нейронные сети

Глубокое обучение (deep learning) – это класс алгоритмов машинного обучения, которые учатся глубже (более абстрактно) понимать данные. Популярные алгоритмы нейронных сетей глубокого обучения представлены на схеме ниже.

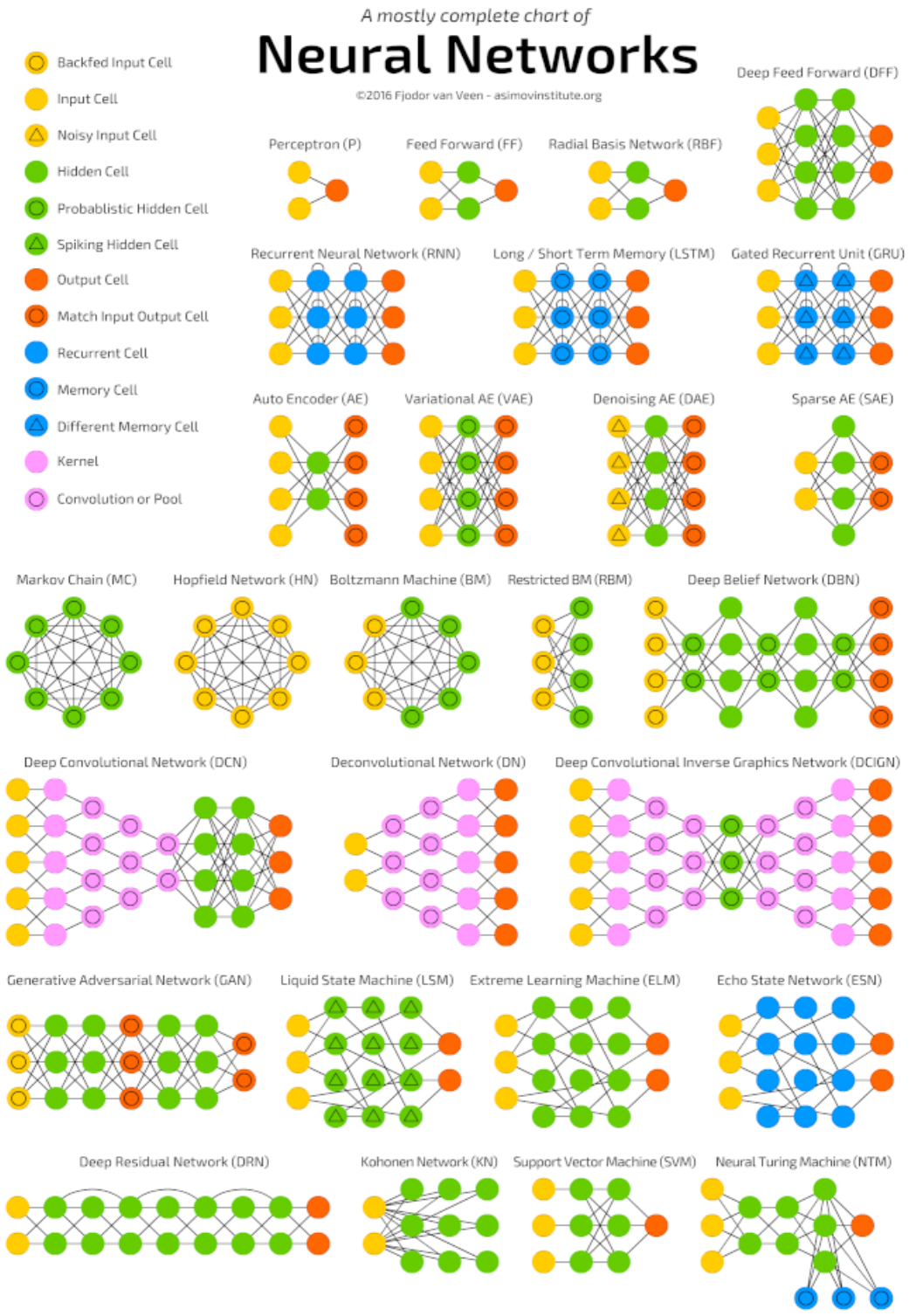


Рисунок 12 - Популярные алгоритмы нейронных сетей.

Более формально в deep learning:

- Используется каскад (пайплайн, как последовательно передаваемый поток) из множества обрабатывающих слоев (нелинейных) для извлечения и преобразования признаков;
- Основывается на изучении признаков (представлении информации) в данных без и с учителем. Функции более высокого уровня (которые находятся в последних слоях) получаются из функций нижнего уровня (которые находятся в слоях начальных слоях);
- Изучает многоуровневые представления, которые соответствуют разным уровням абстракции; уровни образуют иерархию представления.

2.4 Пример обучения нейронной сети

Рассмотрим однослойную нейронную сеть:

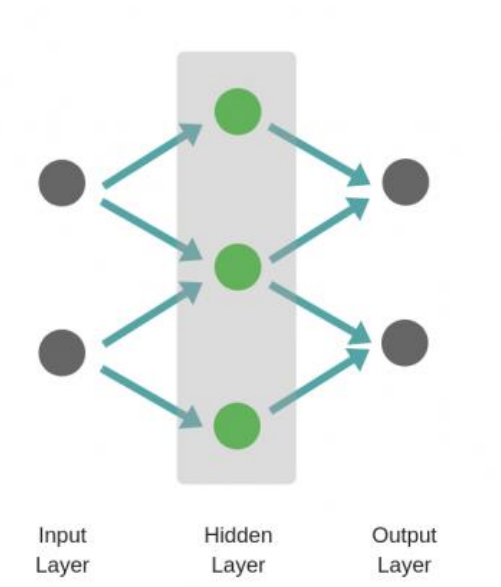


Рисунок 13 - Однослойная нейронная сеть.

Здесь, обучается первый слой (зеленые нейроны), он просто передается на выход.

В то время как в случае двухслойной нейронной сети, независимо от того, как обучается зеленый скрытый слой, он затем передается на синий скрытый слой, где продолжает обучаться:

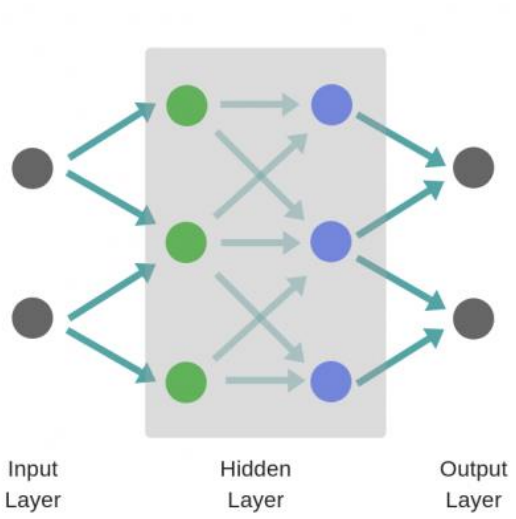


Рисунок 14 - Двуслойная нейронная сеть.

Следовательно, чем больше число скрытых слоев, тем больше возможности обучения сети.

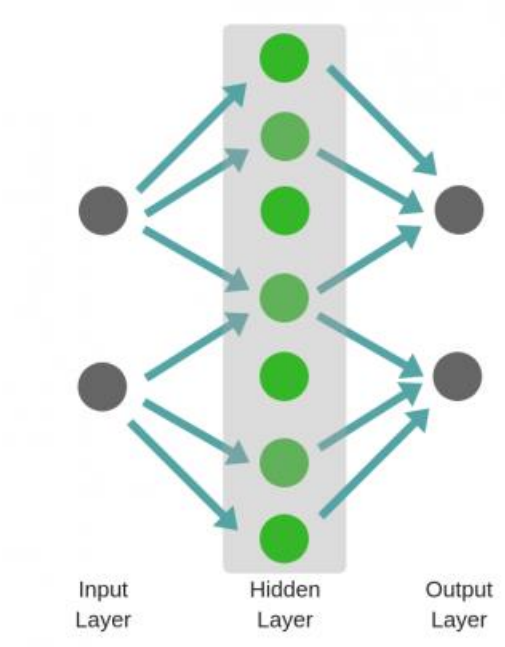


Рисунок 15 - «Широкая» сеть.

Не следует путать глубокую сеть с широкой нейронной сетью.

В случае широкой сети большое число нейронов в одном слое не приводит к глубокому пониманию данных. Но это приводит к изучению большего числа признаков.

Очень заманчиво использовать глубокие и широкие нейронные сети для каждой задачи. Но это может быть плохой идеей, потому что:

- Обе требуют значительно большего количества данных для обучения, чтобы достичь минимальной желаемой точности;
- Обе имеют экспоненциальную сложность;
- Слишком глубокая нейронная сеть попытается сломать фундаментальные представления, но при этом она будет делать ошибочные предположения и пытаться найти псевдо-зависимости, которые не существуют;
- Слишком широкая нейронная сеть будет пытаться найти больше признаков, чем есть. Таким образом, подобно предыдущей, она начнет делать неправильные предположения о данных.

2.5 Проблемы и компромиссы

Основные проблемы которые встречаются в обучении ИНС.

1. Недообучение
2. Переобучение
3. Необходимость большого количества обучающих данных
4. Необходимость мощного аппаратного обеспечение

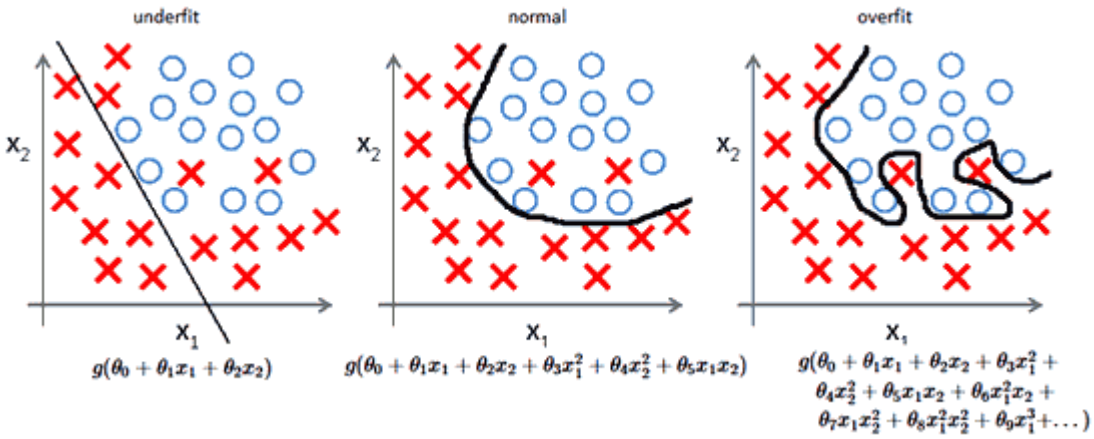


Рисунок 16 - Демонстрация недообучения, нормального обучения и переобучения.

Недообучение – это когда модель делает плохие прогнозы на обучающем и валидационном датасете.

Проблему недообучения можно решить усложнив нашу архитектуру если модель проста относительно задачи, если же архитектура подходящая то собрать еще больше обучающих данных.

Переобучение – это когда модель хорошо предсказывает на обучающем наборе, но на данных которых она раньше не видела даёт плохой результат.

Решением переобучения является **Регуляризация** и увеличение набора данных.

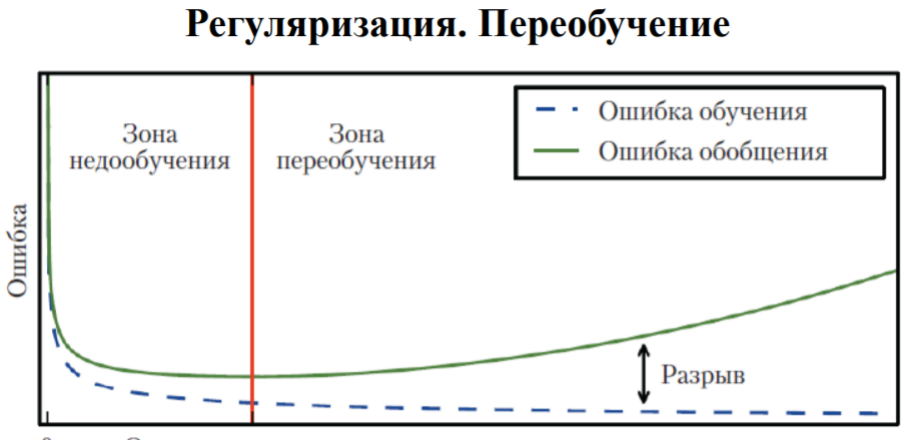


Рисунок 17 - Иллюстрация проблемы

Чтобы получить хороший результат от модели нужно очень много размеченных данных, десятки, сотни тысяч или даже миллионы экземпляров. И соответственно чтобы их обработать нужны очень большие вычисления.

К счастью сообщество тех, кто занимается Глубоким обучением очень «дружное». В интернете в свободном доступе очень много созданных IT-компаниями и людьми, готовых наборов данных, которых можно скачать бесплатно и обучить свою собственную модель. Также сообщество предоставляет в свободный доступ предварительно обученные модели, которые обучались на миллионах данных и много-много часов. Их можно скачать и легко, даже с маленьким набором данных и с не очень мощным аппаратным обеспечением дообучить для собственных целей и получить хорошие результаты. Это частично решает последние 2 проблемы.

Так же большие компании как Google, Amazon, Яндекс и другие предоставляют возможность облачных вычислений бесплатно или же за незначительную сумму для таких больших вычислений.

2.6 Обзор архитектур нейронных сетей

Существует множество различных архитектур нейронных сетей, представим обзор популярнейших архитектур далее.

Многослойный перцептрон/Полносвязная нейросеть состоит из 3 или более слоев. Он использует нелинейную функцию активации , часто тангенциальную или логистическую, которая позволяет классифицировать линейно неразделимые данные. Каждый узел в слое соединен с каждый узлом в последующем слое, что делает сеть полностью связанной. Такая архитектура находит применение в задачах распознавания речи и машинном переводе.

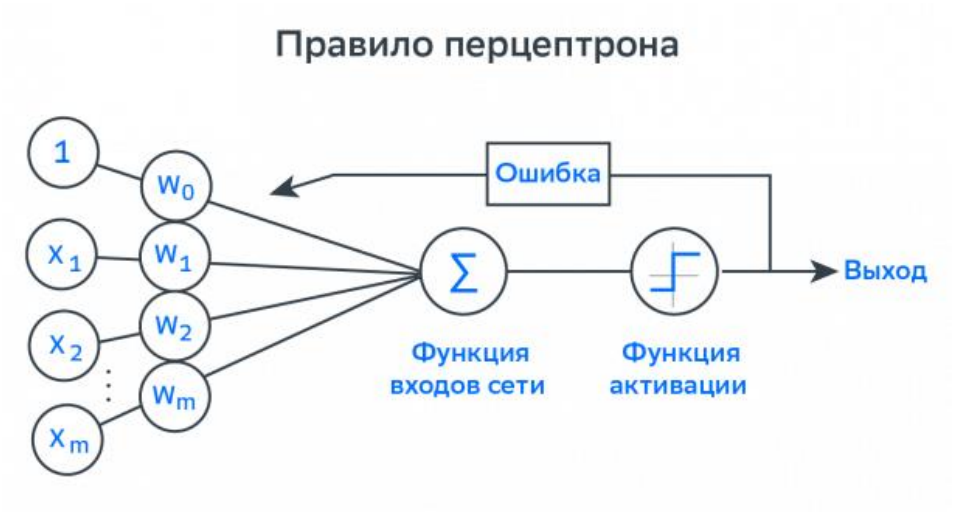


Рисунок 18 - Многослойный перцептрон.

Сверточная нейронная сеть (Convolutional neural network, CNN) содержит один или более объединенных или соединенных сверточных слоев. CNN использует вариацию многослойного перцептрона, рассмотренного выше. Сверточные слои используют операцию свертки для входных данных и передают результат в следующий слой. Эта операция позволяет сети быть глубже с меньшим количеством параметров.

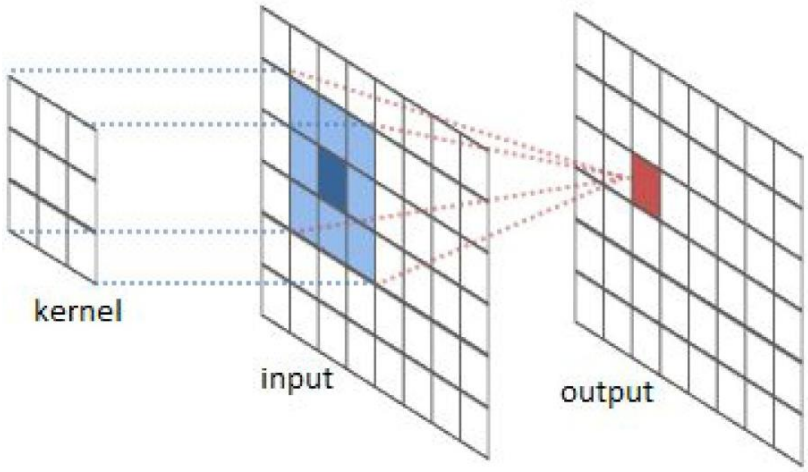


Рисунок 19 - Свёрточная нейронная сеть.

Рекурсивная нейронная сеть — тип глубокой нейронной сети, сформированный при применении одних и тех же наборов весов рекурсивно над структурой, чтобы сделать скалярное или структурированное предсказание над входной структурой переменного размера через активацию структуры в топологическом порядке. В простейшей архитектуре нелинейность, такая как тангенциальная функция активации, и матрица весов, разделяемая всей сетью.

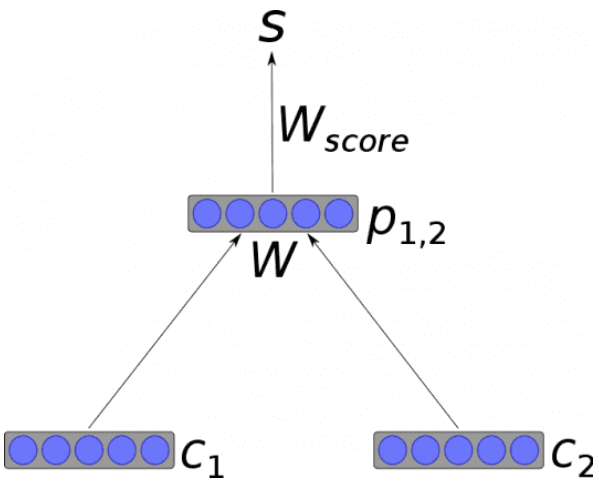


Рисунок 20 - Рекурсивная нейронная сеть.

Рекуррентная нейронная сеть, в отличие от прямой нейронной сети, является вариантом рекурсивной ИНС, в которой связи между нейронами — направленные циклы. Последнее означает, что выходная информация зависит не только от текущего входа, но также от состояний нейрона на предыдущем шаге. Такая память позволяет пользователям решать задачи распознания рукописного текста или речи.

Сеть долгой краткосрочной памяти (Long Short-Term Memory, LSTM) — разновидность архитектуры рекуррентной нейросети, созданная для более точного моделирования временных последовательностей и их долгосрочных зависимостей, чем традиционная рекуррентная сеть. LSTM-сеть не использует функцию активации в рекуррентных компонентах, сохраненные значения не модифицируются, а градиент не стремится исчезнуть во время тренировки. Часто LSTM применяется в блоках по несколько элементов. Эти блоки состоят из 3 или 4 затворов (например, входного, выходного и гейта забывания), которые контролируют построение информационного потока по логистической функции.

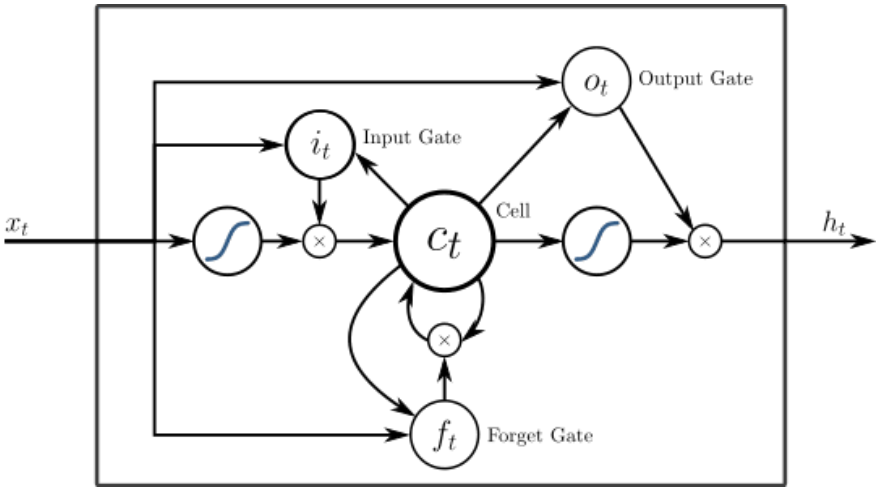


Рисунок 21 - LSTM.

2.7 Свёрточные нейронные сети

Сверточная нейронная сеть основана на удивительно мощной и универсальной математической операции. В этой части мы шаг за шагом рассмотрим механизм их работы на примере стандартной полностью рабочей сети, и изучим то, как они строят качественные визуальные иерархии.

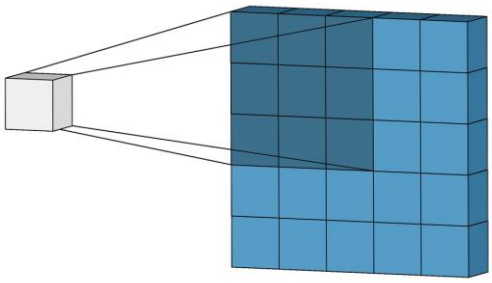


Рисунок 22 - Двумерная свертка.

Двумерная свертка (2D convolution) — это довольно простая операция: начинаем с ядра, представляющего из себя матрицу весов (weight matrix). Ядро “скользит” над двумерным изображением, поэлементно выполняя операцию умножения с той частью входных данных, над которой оно сейчас находится, и затем суммирует все полученные значения в один выходной пиксель.

Ядро повторяет эту процедуру с каждой локацией, над которой оно “скользит”, преобразуя двумерную матрицу в другую все еще двумерную матрицу признаков. Признаки на выходе являются взвешенными суммами (где веса являются значениями самого ядра) признаков на входе, расположенных примерно в том же месте, что и выходной пиксель на входном слое.

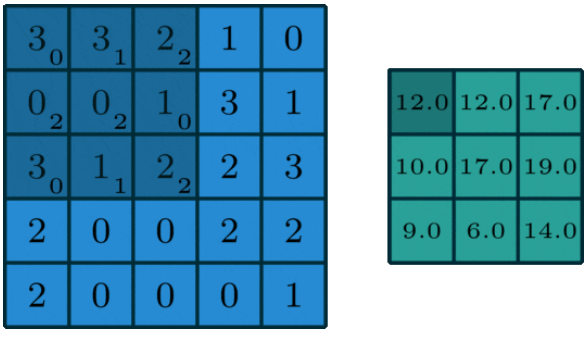


Рисунок 23 - Операция свертки.

Независимо от того, попадает ли входной признак в “примерно то же место”, он определяется в зависимости от того, находится он в зоне ядра, создающего выходные данные, или нет. Это значит, что размер ядра

сверточной нейронной сети определяет количество признаков, которые будут объединены для получения нового признака на выходе.

В примере, приведенном выше, мы имеем 5*5=25 признаков на входе и 3*3=9 признаков на выходе. Для стандартного слоя (standard fully connected layer) мы бы имели весовую матрицу 25*9 = 225 параметров, а каждый выходной признак являлся бы взвешенной суммой всех признаков на входе. Свертка позволяет произвести такую операцию с всего 9-ю параметрами, ведь каждый признак на выходе получается анализом не каждого признака на входе, а только одного входного, находящегося в “примерно том же месте”. Обратите на это внимание, так как это будет иметь важное значение для дальнейшего обсуждения.

Перед тем как мы двинемся дальше, безусловно стоит взглянуть на две техники, которые часто применяются в сверточных нейронных сетях: Padding и Striding.

Padding. В анимации, обратим внимание на то, что в процессе скольжения края по существу обрезаются, преобразуя матрицу признаков размером 5*5 в матрицу 3*3. Крайние пиксели никогда не оказываются в центре ядра, потому что тогда ядру не над чем будет скользить за краем. Это совсем не идеальный вариант, так как мы хотим, чтобы размер на выходе равнялся входному.

Padding добавляет к краям поддельные (fake) пиксели (обычно нулевого значения, вследствие этого к ним применяется термин “нулевое дополнение” — “zero padding”). Таким образом, ядро при проскальзывании позволяет неподдельным пикселям оказываться в своем центре, а затем распространяется на поддельные пиксели за пределами края, создавая выходную матрицу того же размера, что и входная.

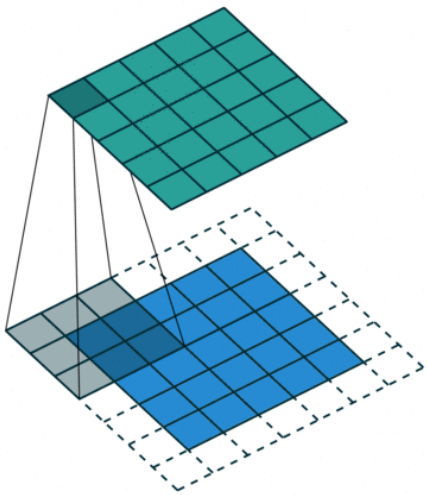


Рисунок 24 - Padding.

Striding. Часто бывает, что при работе со сверточным слоем, нужно получить выходные данные меньшего размера, чем входные. Это обычно необходимо в сверточных нейронных сетях, где размер пространственных размеров уменьшается при увеличении количества каналов. Один из способов достижения этого — использование субдискритизирующих слоев (pooling layer), например, принимать среднее/максимальное значение каждой ветки размером 2*2, чтобы уменьшить все пространственные размеры в два раза. Еще один способ добиться этого — использовать stride (шаг).

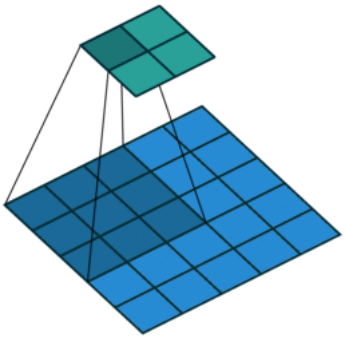


Рисунок 25 - Свертка с шагом 2.

Идея stride заключается в том, чтобы пропустить некоторые области, над которыми скользит ядро. Шаг 1 означает, что берутся пролеты через пиксель, то есть по факту каждый пролет является стандартной сверткой. Шаг 2 означает, что пролеты совершаются через каждые два пикселя, пропуская все другие пролеты в процессе и уменьшая их количество примерно в 2 раза, шаг 3 означает пропуск 3-х пикселей, сокращая количество в 3 раза и т.д.

Приведенные выше диаграммы касаются только случая, когда изображение имеет один входной канал. На практике большинство входных изображений имеют 3 канала, и чем глубже вы в сети, тем больше это число.

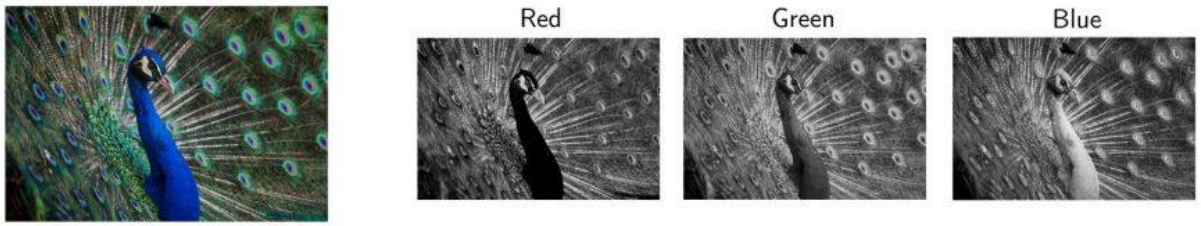


Рисунок 26 - RGB изображение.

Вот где ключевые различия между терминами становятся нужными: тогда как в случае с 1 каналом термины «фильтр» и «ядро» взаимозаменяемы, в общем случае они разные.

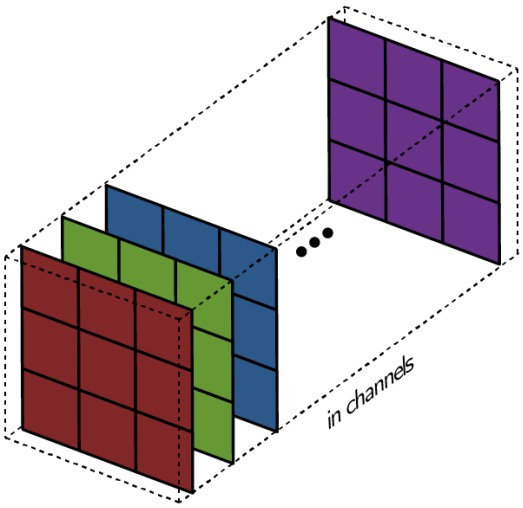


Рисунок 27. Представление изображения в 3 каналах.

Каждый фильтр на самом деле представляет собой коллекцию ядер, причем для каждого отдельного входного канала этого слоя есть одно ядро, и каждое ядро уникально.

Каждый фильтр в сверточном слое создает только один выходной канал и делают они это так: каждое из ядер фильтра «скользит» по их соответствующим входным каналам, создавая обработанную версию каждого из них. Некоторые ядра могут иметь больший вес, чем другие, для того чтобы уделять больше внимания определенным входным каналам (например, фильтр может задать красному каналу ядра больший вес, чем другим каналам, и, следовательно, больше реагировать на различия в образах из красного канала).

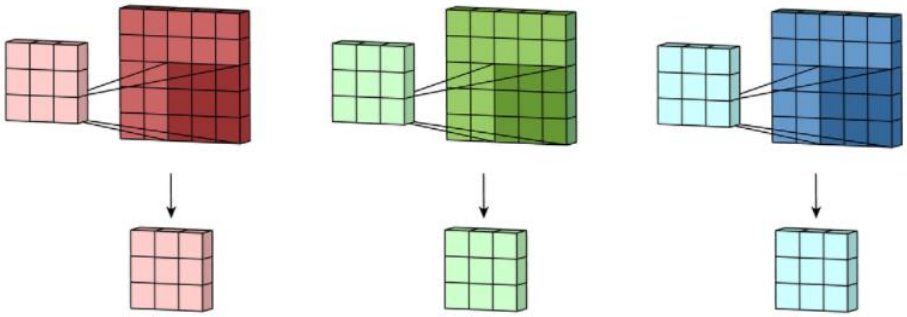


Рисунок 28 - Пример свертки RGB изображения.

Затем каждая из обработанных в канале версий суммируется вместе для формирования одного канала. Ядра каждого фильтра генерируют одну версию каждого канала, а фильтр в целом создает один общий выходной канал:

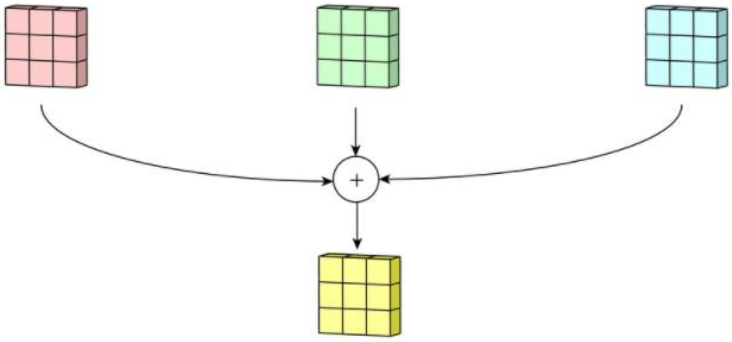


Рисунок 29 - Выходной канал после сверки RGB изображения.

Наконец, каждый выходной файл имеет свое смещение. Смещение добавляется к выходному каналу для создания конечного выходного канала:

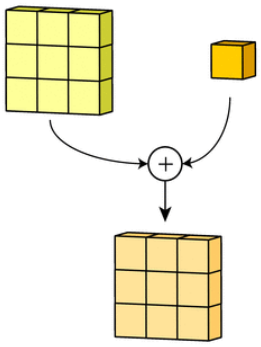


Рисунок 30 - Присваивание смещения к выходному каналу.

Результат для любого количества фильтров идентичен: каждый фильтр обрабатывает вход со своим отличающимся от других набором ядер и скалярным смещением по описанному выше процессу, создавая один выходной канал. Затем они объединяются вместе для получения общего выхода, причем количество выходных каналов равно числу фильтров. При этом обычно применяется нелинейность перед передачей входа другому слою свертки, который затем повторяет этот процесс.

Теперь мы определили все необходимые понятия. Познакомились с основами Глубокого обучения. Дальше основываясь на этих понятиях приступим к основной работе.

3. Постановка задачи

Целью представляемой курсовой работы является обучение и внедрение моделей Глубокого обучение для классификации правильности надевания масок в режиме реального времени, исследование и применение transfer learning’a на конкретно поставленной задаче.

Достижение цели курсовой работы предполагает необходимость решения следующих задач:

1. Изучить и исследовать методы Глубокого обучения для Компьютерного зрения.
2. Собрать достаточно набора данных для обучения модели
3. Выбрать подходящую модель для Переноса обучения
4. С помощью transfer learning’a обучить модель на своих данных
5. Проанализировать обученную модель и улучшить результаты
6. Применить обученную модель для распознавания масок
7. Проанализировать конечный результат

4. Обучение модели

Предварительно обученная модель — это сеть, которая уже была обучена на большом наборе данных и сохранена, что позволяет нам использовать ее для эффективного дообучения нашей собственной модели. Та, которую мы будем использовать, MobileNetV2, была разработана для обеспечения быстрой и вычислительно эффективной производительности. Она была предварительно обучена в ImageNet, наборе данных, содержащем более 14 миллионов изображений и 1000 классов. Мы возьмем эту модель и будем использовать в наших целях. Для быстрых вычислений мы также будем использовать Google Colab. Google Colab - это бесплатный облачный сервис на основе Jupyter Notebook. Google Colab предоставляет всё необходимое для машинного обучения прямо в браузере, даёт бесплатный доступ к очень быстрым GPU и TPU.

4.1 Набор данных

MaskedFace-Net – набор данных для классификации правильно/неправильно надетых медицинских масок состоящий из более чем по 65 000 изображений для каждого класса. Мы будем использовать не весь набор, а только часть, примерно по 900 изображений для каждого класса, а также примерно 900 изображений лиц без маски собранный из различных Kaggle соревнований. В итоге у нас будут три класса:

- **correct,** маска надета правильно
- **incorrect,** маска надета неправильно
- **no_mask ,** нет маски на лице

Количество изображений не так много, но как мы увидим далее, в учебных целях для хорошего результата этого достаточно и нет необходимости обучать модель на больших данных так как это вычислительно дорого.

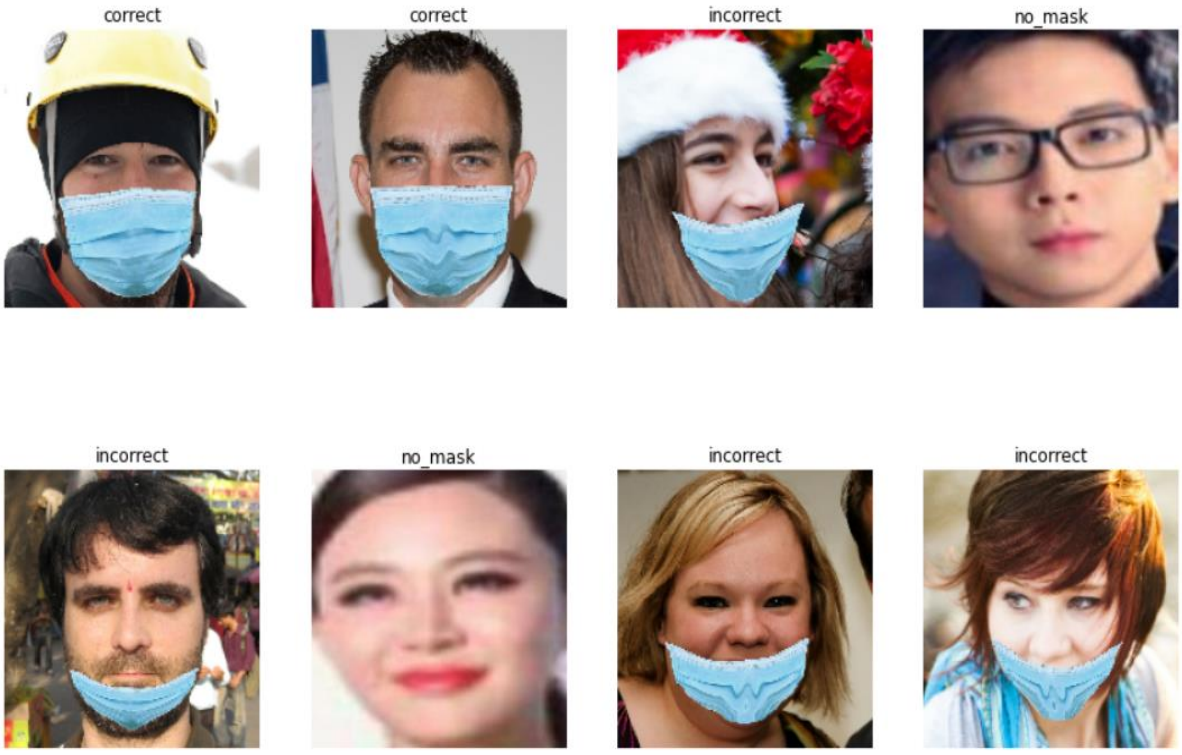


Рисунок 21 - Пример из набора данных

Импорт необходимых пакетов

```
import matplotlib.pyplot as plt
import numpy as np
import os
import tensorflow as tf
import tensorflow.keras.layers as tfl
from tensorflow.keras.preprocessing import
image_dataset_from_directory
from tensorflow.keras.layers.experimental.preprocessing import
RandomFlip, RandomRotation
```

4.2 Создание датасета и деление его на части

При обучении и оценке моделей глубокого обучения в Keras создание набора данных из изображений, хранящихся на памяти, является простым и быстрым. У нас есть 3 папки: correct, incorrect, no_mask. Следующий метод создаст датасет с тремя классами из этих каталогов. Вызовем метод "image_data_set_from_directory()" для чтения из каталога и создания как обучающих, так и валидационных наборов данных. Размер батча у нас будет 64, а размер входного изображения (160, 160).

```
BATCH_SIZE = 64
IMG_SIZE = (160, 160)
directory = "/content/drive/My Drive/CourseWork/ProjectMask/dataset"
train_dataset = image_dataset_from_directory(directory,
                                             shuffle=True,
                                             batch_size=BATCH_SIZE,
                                             image_size=IMG_SIZE,
                                             color_mode='rgb',
                                             validation_split=0.2,
                                             subset='training',
                                             seed=42)
validation_dataset = image_dataset_from_directory(directory,
                                             shuffle=True,
                                             batch_size=BATCH_SIZE,
                                             image_size=IMG_SIZE,
                                             color_mode='rgb',
                                             validation_split=0.2,
                                             subset='validation',
                                             seed=42)
```

```
Found 2835 files belonging to 3 classes.
Using 2268 files for training.
Found 2835 files belonging to 3 classes.
Using 567 files for validation.
```

Рисунок 32 - Вывод предыдущей команды

Как мы видим, у нас всего 2835 файлов, принадлежащих трем классам. Мы разделили их на обучающие и валидационные датасеты. В итоге у нас 2268 изображений в обучающем датасете и 567 изображений в валидационном датасете.

4.3 Аугментация

Далее продемонстрируем как с помощью простых шагов можно увеличить наш набор с помощью аугментации, отражение по горизонтали и поворотов.

```
def data_augmenter():
    data_augmentation = tf.keras.Sequential()
    data_augmentation.add(tf.keras.layers.experimental
                          .preprocessing.RandomFlip('horizontal'))
    data_augmentation.add(tf.keras.layers.experimental
                         .preprocessing.RandomRotation(0.2))
    return data_augmentation
```

Взглянем на то, как изображение из обучающего набора было дополнено простыми преобразованиями. От одного фото до 8 вариаций этого фото всего лишь за три строки кода. Теперь нашей модели есть чему поучиться.

```
data_augmentation = data_augmenter()
for image, _ in train_dataset.take(1):
    plt.figure(figsize=(15, 8))
    first_image = image[0]
    for i in range(8):
        ax = plt.subplot(2, 4, i + 1)
        augmented_image =  data_augmentation(
                                        tf.expand_dims(first_image, 0))
        plt.imshow(augmented_image[0] / 255)
        plt.axis('off')
```

Рисунок 33 - Пример результата аугментации

4.3 Использование MobileNetV2 для Transfer Learning

Модель MobileNetV2 была обучена в наборе данных ImageNet и оптимизирована для работы в мобильных и других устройствах с невысокими мощностями . Она имеет глубину 155 слоев (на случай, если вам захочется построить модель самостоятельно, приготовьтесь к долгому путешествию) и очень эффективна для задач обнаружения объектов и сегментации изображений, а также для задач классификации, подобных этой.

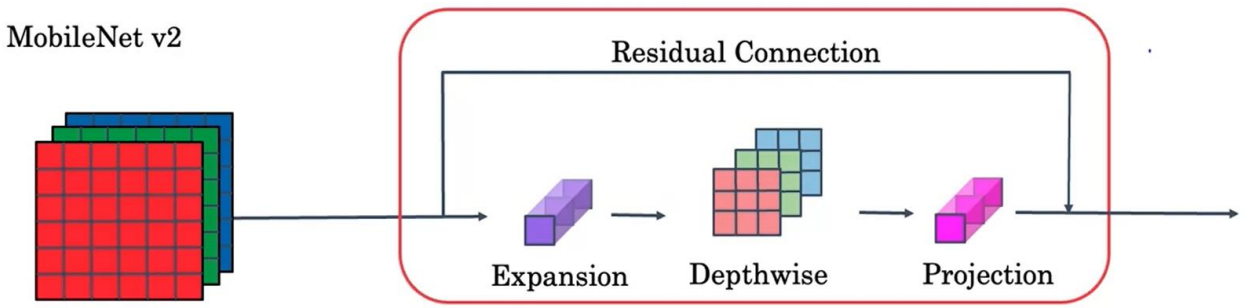


Рисунок 34 - Базовый блок архитектуры MobileNetV2. Модель состоит из 16-ти таких блоков

Загружаем предобученную модель MobileNetV2 вместе с весами, ImageNet weights='imagenet'.

```
IMG_SHAPE = IMG_SIZE + (3,)
base_model = tf.keras.applications.MobileNetV2(input_shape=IMG_SHAPE,
                                               include_top=True,
                                               weights='imagenet')
```

Выведем сведение модели ниже, чтобы увидеть слои модели, размеры выходных данных и общее количество параметров, обучаемых и не обучаемых.

```
base_model.summary()
```

```
block_16_project_BN (BatchNorm  (None, 5, 5, 320)   1280        ['block_16_project[0][0]']
alization)

Conv_1 (Conv2D)                 (None, 5, 5, 1280)  409600      ['block_16_project_BN[0][0]']

Conv_1_bn (BatchNormalization)  (None, 5, 5, 1280)  5120        ['Conv_1[0][0]']

out_relu (ReLU)                 (None, 5, 5, 1280)  0           ['Conv_1_bn[0][0]']

global_average_pooling2d (Glob  (None, 1280)        0           ['out_relu[0][0]']
alAveragePooling2D)

predictions (Dense)             (None, 1000)        1281000     ['global_average_pooling2d[0][0]'
                                                                ]

==================================================================================================
Total params: 3,538,984
Trainable params: 3,504,872
Non-trainable params: 34,112
```

Рисунок 35 - Часть вывода model.summary()

Как мы видим в модели всего 3 538 984 параметров, из них 3 504 872 обучаемых и 34 112 необучаемых параметров. Также обратим внимание на последние 2 слоя здесь. Это так называемые верхние слои, и они отвечают за классификацию в модели: global_average_pooling2d, predictions.

4.4 Заморозка весов базовой модели для подготовки к обучению

Теперь рассмотрим, как можно использовать предварительно обученную модель для наших целей, чтобы она могла распознавать маски. Мы сделаем это в три этапа:

1. Удаляем последний слой (слой классификации)
   - Установить include_top в base_model как False
2. Добавляем новый слой классификации
   - Для трех классов добавим три новых нейрона вместо 1000
3. Заморозим базовую модель и обучим вновь созданный слой классификации.
   - Установим `base_model.trainable=False", чтобы избежать изменение весов базовой модели

```
def Mask_model(image_shape=IMG_SIZE, data_augmentation=data_augmenter()):
    input_shape = image_shape + (3,)
    base_model = tf.keras.applications.MobileNetV2(input_shape=input_shap,
                                                    include_top=False,
                                                    weights='imagenet')
    base_model.trainable = False
    #создадим входной слой (такой же, как входной размер MobileNetV2)
    inputs = tf.keras.Input(shape=input_shape)
    #применим Аугментацию данных к входным данным
    data_augmentation = data_augmenter()
    x = data_augmentation(inputs)
    # предварительная обработка данных с использованием тех же весов,
    # на которых была обучена модель
    x = preprocess_input(x)
    # установим training=False, чтобы избежать отслеживания статистики
    # в слоях Batch нормализации
    x = base_model(x, training=False)
    # добавим новый слой классификации
    x = tf.keras.layers.GlobalAveragePooling2D()(x)
    x =tf.keras.layers.Dropout(0.2)(x)
    # добавим три новых нейрона, каждый будет отвечать за
    # вероятность своего класса, используем функцию активации Сигмоида
    outputs = tf.keras.layers.Dense(3, activation='sigmoid')(x)
    model = tf.keras.Model(inputs, outputs)
    return model
```

4.5 Этап обучения

Теперь у нас всё готово чтобы начать обучение. Создаём свою новую модель, используя функцию data_augmentation(), определенную ранее.

```
model2 = Mask_model(IMG_SIZE, data_augmentation)
```

Установим скорость обучения как 0.001, используем оптимизатор Adam, функцию потерь SparseCategoricalCrossentropy(), и метрику sparse_categorical_accuracy().

```
base_learning_rate = 0.001
model2.compile(optimizer=tf.keras.optimizers.Adam(lr=base_learning_rate),
               loss=tf.keras.losses.SparseCategoricalCrossentropy(),
               metrics=[tf.keras.metrics.sparse_categorical_accuracy])
```

Будем обучать 10 эпох, и историю обучения храним в переменной history для дальнейшего анализа. Передаем все эти данные в метод .fit(). Дальше модель начнет процесс обучения.

```
initial_epochs = 10
history = model2.fit(train_dataset, validation_data=validation_dataset, ep
ochs=initial_epochs)
```

Так как мы использовали мощные GPU от Google Colab и количество данных не было так велико, обучение не заняло много времени, в среднем 30 секунд на каждую эпоху. Это очень быстро. Далее посмотрим, как шло обучение, посмотрим на графики метрики и функции потерь.

```
loss: 0.5458 - sparse_categorical_accuracy: 0.7504 - val_loss: 0.3555 - val_sparse_categorical_accuracy: 0.8624
loss: 0.2949 - sparse_categorical_accuracy: 0.8862 - val_loss: 0.2447 - val_sparse_categorical_accuracy: 0.9171
loss: 0.2315 - sparse_categorical_accuracy: 0.9118 - val_loss: 0.2083 - val_sparse_categorical_accuracy: 0.9383
loss: 0.2006 - sparse_categorical_accuracy: 0.9246 - val_loss: 0.1850 - val_sparse_categorical_accuracy: 0.9365
loss: 0.1914 - sparse_categorical_accuracy: 0.9233 - val_loss: 0.1668 - val_sparse_categorical_accuracy: 0.9506
loss: 0.1907 - sparse_categorical_accuracy: 0.9277 - val_loss: 0.1688 - val_sparse_categorical_accuracy: 0.9489
loss: 0.1661 - sparse_categorical_accuracy: 0.9400 - val_loss: 0.1522 - val_sparse_categorical_accuracy: 0.9524
loss: 0.1564 - sparse_categorical_accuracy: 0.9409 - val_loss: 0.1420 - val_sparse_categorical_accuracy: 0.9559
loss: 0.1440 - sparse_categorical_accuracy: 0.9462 - val_loss: 0.1427 - val_sparse_categorical_accuracy: 0.9506
loss: 0.1446 - sparse_categorical_accuracy: 0.9436 - val_loss: 0.1324 - val_sparse_categorical_accuracy: 0.9524
```

Рисунок 36 - Вывод после каждой эпохи обучения.

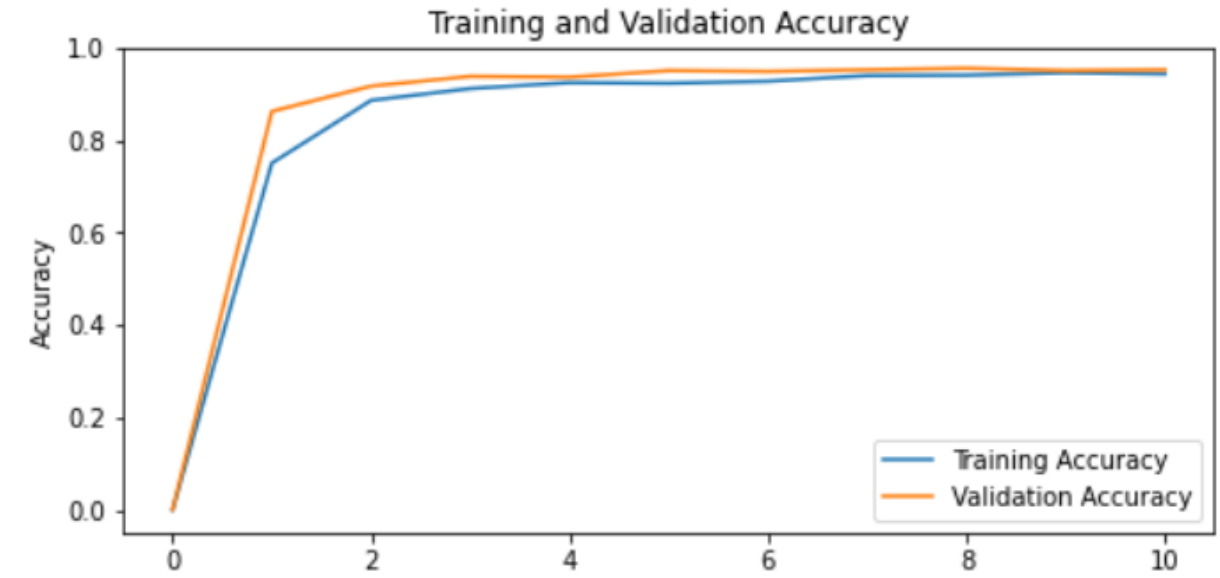


Рисунок 37 - Точность модели в различных эпохах

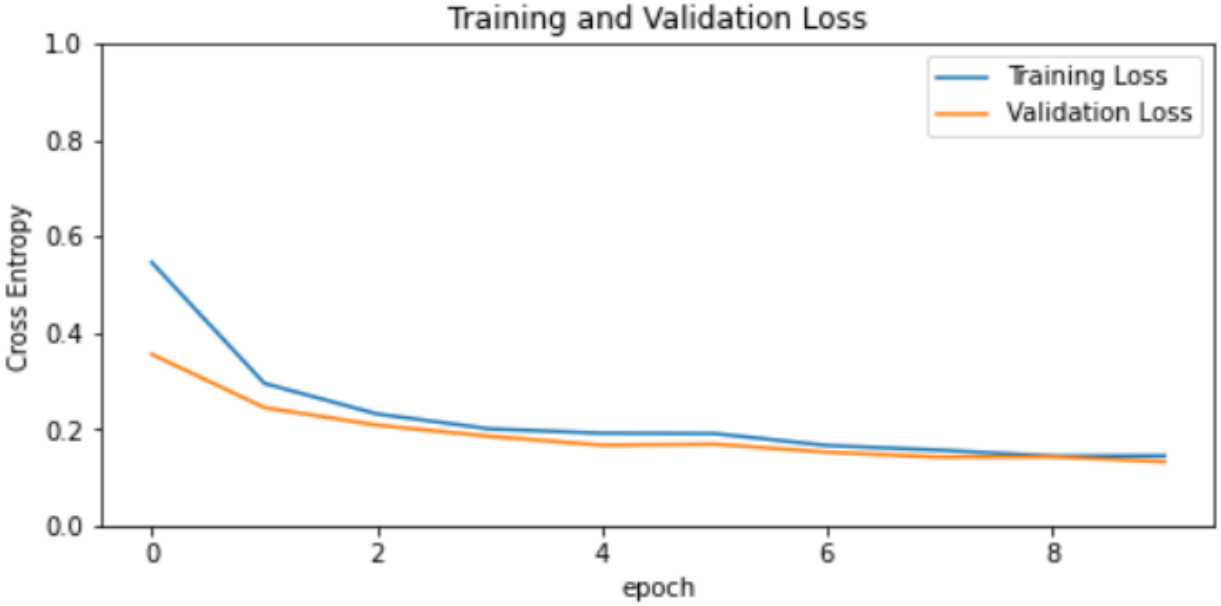


Рисунок 38 - График функции потерь

Как мы видим наша модель очень быстро набрала хорошую точность, уже после первой же эпохи, близкое к единице. В последней эпохе на обучающем наборе точность равна: 0.9436, а на валидационном наборе 0.9524. Значение функции потерь довольно маленькое 0.1446 - на обучающем наборе данных и 0.1324 - на валидационном. Это все благодаря тому, что мы использовали transfer learning, и наша базовая модель MobileNetV2 была очень хорошо обучена на миллионах данных и хорошо умеет извлекать детали из фотографий.

4.6 Fine-tuning

Мы можем попробовать Fine-Tuning, повторно запустив оптимизатор на последних слоях, чтобы повысить точность. Когда мы используем меньшую скорость обучения, мы делаем меньшие шаги, чтобы модель лучше адаптировался к новым данным. Интуиция того, что происходит: ранние слои распознают лишь общие детали такие как горизонтальные, вертикальные линии и т.д. чем глубже слой, тем лучше он распознает более значительные, отличительные и уникальные черты классов. Идея Fine-Tuning'a состоит в том, чтобы разморозить последние несколько слоев базовой модели и дообучить с меньшей скоростью обучения, чтобы и они немножко адаптировались к нашим новым данным и лучше распознавали нужные нам черты.

Сначала разморозим все слои базовой модели, установив "base_model.trainable=True", потом итерируя по слоям, в нашем случае, до 120 - го слоя, установим layer.trainable = False, Затем запустим модель снова еще на несколько эпох, в данном случае на 5, и посмотрим, улучшилась ли наша точность.

```
base_model = model2.layers[4]
base_model.trainable = True
fine_tune_at = 120
for layer in base_model.layers[:fine_tune_at]:
    layer.trainable = False
loss_function=tf.keras.losses.SparseCategoricalCrossentropy()
optimizer =tf.keras.optimizers.Adam(base_learning_rate*0.1)
metrics=[tf.keras.metrics.sparse_categorical_accuracy]
model2.compile(loss=loss_function,
               optimizer = optimizer,
               metrics=metrics)

fine_tune_epochs = 5
total_epochs =  initial_epochs + fine_tune_epochs

history_fine = model2.fit(train_dataset,
                          epochs=total_epochs,
                          initial_epoch=history.epoch[-1],
                          validation_data=validation_dataset)
```

```
loss: 0.2372 - sparse_categorical_accuracy: 0.9123 - val_loss: 0.0867 - val_sparse_categorical_accuracy: 0.9630
loss: 0.0665 - sparse_categorical_accuracy: 0.9753 - val_loss: 0.0672 - val_sparse_categorical_accuracy: 0.9771
loss: 0.0599 - sparse_categorical_accuracy: 0.9793 - val_loss: 0.0298 - val_sparse_categorical_accuracy: 0.9859
loss: 0.0456 - sparse_categorical_accuracy: 0.9850 - val_loss: 0.0367 - val_sparse_categorical_accuracy: 0.9841
loss: 0.0251 - sparse_categorical_accuracy: 0.9925 - val_loss: 0.1949 - val_sparse_categorical_accuracy: 0.9418
loss: 0.0688 - sparse_categorical_accuracy: 0.9780 - val_loss: 0.0335 - val_sparse_categorical_accuracy: 0.9877
```

Рисунок 39 - Вывод после каждой эпохи.

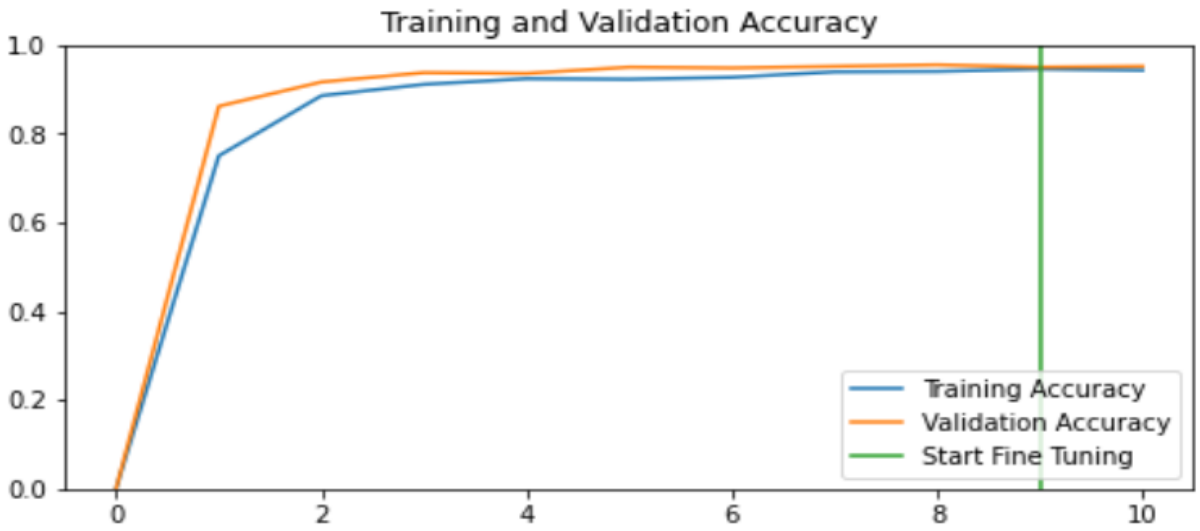


Рисунок 40 - График точности нашей модели

Видно, что результаты улучшились, и достигли почти 0.99 точности, вместо 0.94 что было раньше. Как мы убедились Fine-tuning является хорошей практикой и помогает улучшить модель значительно.

В итоге наша модель достигла точности 99% на валидационном датасете, что является отличным результатом. Далее сохраним модель и приступим ко второй части. Ниже приведён результат классификации модели.

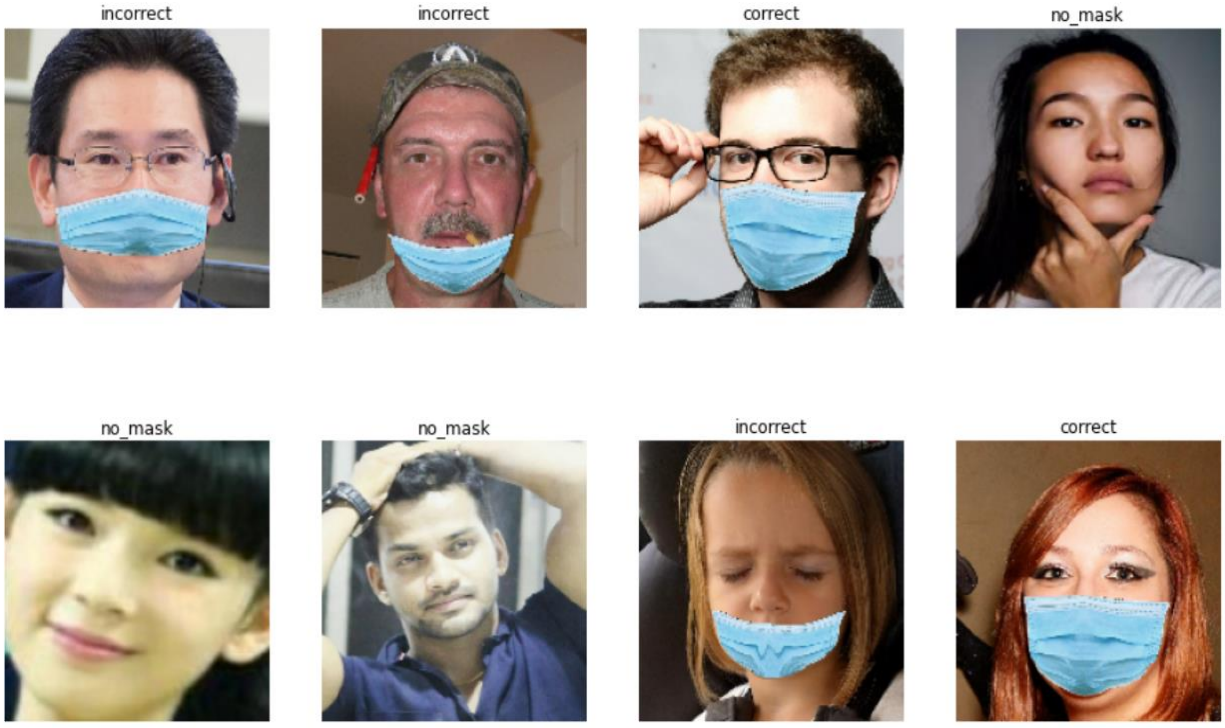


Рисунок 41 - Классификация нашей модели.

5.Распознавание масок в реальном времени

Теперь посмотрим, как мы можем использовать нашу обученную модель на практике в реальном времени. Для этого используем python и библиотеку OpenCV. Но прежде, чем приступить к действию, сразу скажем возможную проблему и возможное его решение. Так как наш набор данных содержал только лица вблизи, наша модель умеет хорошо классифицировать маски только вблизи, и если ему на вход дать изображение человека в полный рост или с далека, то она будет выдавать плохой результат и в этом, конечно же, нет её вины, мы получили то что хотели. Теперь надо ей дать правильный ввод и тогда всё будет отлично. Ниже пример работы вывода модели, где проиллюстрирована проблема: с далека неправильный ответ, но тот же изображение лица вблизи даёт уже правильный результат.

Рисунок 42 – Иллюстрация проблемы.

Ниже рассмотрим одно из решений суть которого состоит в том, чтобы сначала обнаружить лица и потом дать на вход нашей модели те области изображения, где есть лица и потом запустить классификацию на этих участках.

5.1 Обнаружение лиц с помощью mediapipe.

Mediapipe — фреймворк для запуска пайплайнов (предобработка данных, запуск (inference) модели, а также постобработка результатов модели) машинного обучения, позволяющий упростить написание кроссплатформенного кода для запуска моделей. Мы будем использовать модуль Face Detection этой библиотеки. С помощью этого алгоритма выделим изображение лиц и передадим в нашу обученную модель. Мы будем считать изображение с камеры по кадрам, обработаем и выводим результат.

И так шаги нашего алгоритма, следующие:

1. Считать изображение из камеры
2. Найти лица с помощью mediapipe
3. Если есть лица передать в модель для классификации
4. Отобразить результат на изображении
5. Вывод обработанного кадра на экран
6. Вернуться к первому пункту

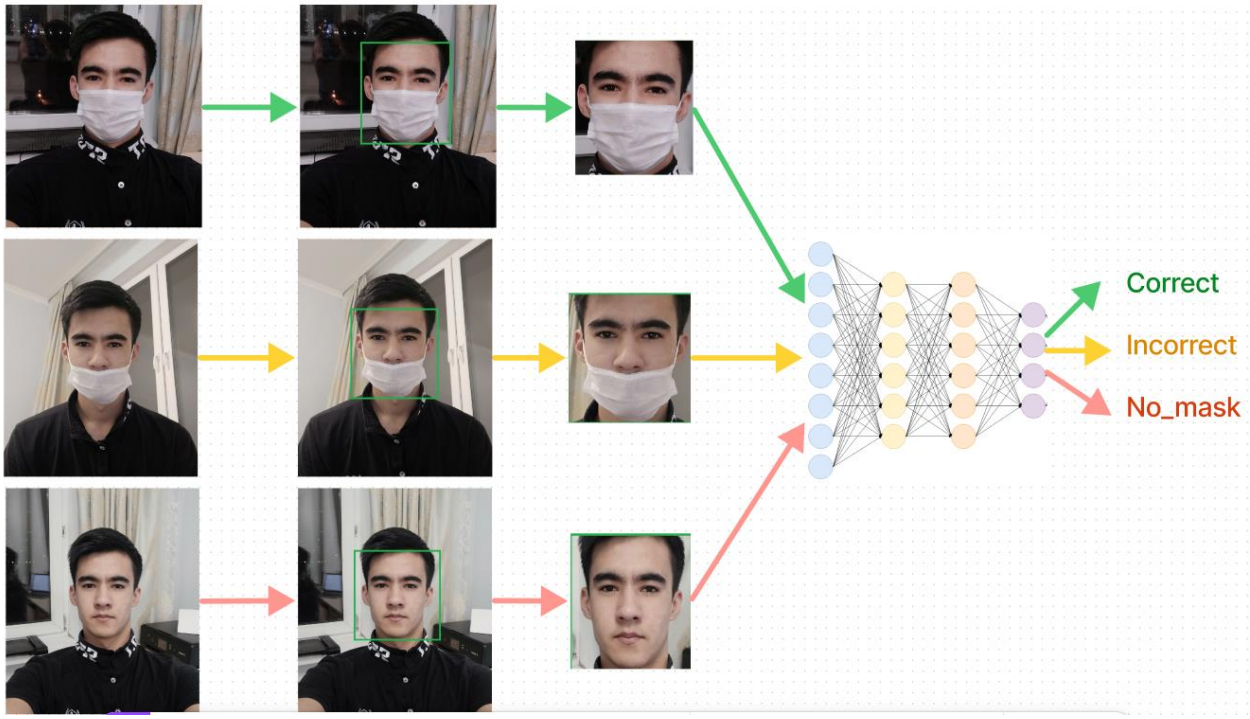


Рисунок 43 - Иллюстрация работы программы

5.2 Построение алгоритма

В первую очередь импортируем необходимые пакеты.

```
import cv2
import numpy as np
import tensorflow as tf
import time
import matplotlib.pyplot as plt
import mediapipe as mp
```

Для еще более быстрой работы конвертируем нашу модель в tensorflow lite. Создадим интерпретатор для работы с tflite моделью, загружая нашу модель. Берем сведения о том какими должны быть входные и выходные данные нашей модели.

```
interpreter = tf.lite.Interpreter(model_path='MaskNet.tflite')
input_details = interpreter.get_input_details()
output_details = interpreter.get_output_details()
interpreter.allocate_tensors()
```

Берем модель для обнаружения лиц.

```
mpFaceDetection = mp.solutions.face_detection
faceDetection = mpFaceDetection.FaceDetection(0.75)
```

Подключимся к камере с помощью метода OpenCV.

```
cap=cv2.VideoCapture(0)
```

Далее приводим код нашего базового цикла с подробными комментариями.

```
while cap.isOpened(): #Пока камера работает
    ret, img= cap.read() # Считаем очередной кадр из камеры
    start_time=time.time() #Счетчик для FPS
    img=cv2.flip(img,1) #Отразим кадр горизонтально, так более привычно для нас
    imgRGB=cv2.cvtColor(img, cv2.COLOR_BGR2RGB) # Конвертируем BGR изображение на RGB
```

Далее с помощью библиотеки mediapipe находим координаты, высоту и ширину лиц в кадре: x,y,w,h.

```
    results = faceDetection.process(imgRGB)
    if results.detections:
        for id, detection in enumerate(results.detections):
            bboxC = detection.location_data.relative_bounding_box
            ih, iw, ic = img.shape
            x, y = int(bboxC.xmin * iw), int(bboxC.ymin * ih)
            w,  h =int(bboxC.width * iw),int(bboxC.height * ih)
```

Далее обрезаем область лиц по полученным данным из общего изображения и передаём в нашу модель для распознавания, декодируем результат и изобразим результат на кадре. И так для каждого найденного лица по циклу.

```
            face_img= cv2.resize(imgRGB[y:y+h+20, x:x+w], (160,160))
            prediction=predict(face_img) # Далее передаем в функцию  для предсказания
            className, color=decode_pred(prediction[0], class_names,img)
            cv2.putText(img, className , (x, y - 10), cv2.FONT_HERSHEY_SIMPLEX, 0.8, color, 2)
            cv2.rectangle(img, (x, y), (x + w, y + h), color, 3)
```

Далее посмотрим какой у нас FPS и это тоже отобразим на верхнем углу изображения. И показываем окончательный результат.

```
                fps=1.0/(time.time() - start_time)
                cv2.putText(img, "FPS:" + str(int(fps)),(10,30),
                                    cv2.FONT_HERSHEY_SIMPLEX, 1.3, (255,0,255), 2)
        # Выводим обработанное изображение на экран
        cv2.imshow('img', img)
```

Функция **predict(img)** запустит нашу модель с заданным изображением и возвращает результат, numpy() массив размера [1 x 3] где в i-ом индексе вероятность i-го класса [P(correct), P(incorrect), P(no_mask)].

```
def predict(img):
  img = tf.expand_dims(np.float32(img), 0)
  interpreter.set_tensor(input_details[0]['index'],img)
  interpreter.invoke()
  prediction = interpreter.get_tensor(output_details[0]['index'])
  return prediction
```

Функция **decode_pred(pred,class_names)** на вход получает вектор, предсказанный моделью и имена классов. Максимальное значение вектора предсказаний и будет являться выводом нашей модели, поэтому находит максимум и в соответствии с этим возвращает класс и цвет, с которым будем выводить результат на экран:

- Зеленый, если ответ **correct**
- Желтый, если ответ **incorrect**
- Красный, если ответ **no_mask**

```
def decode_pred(pred,class_names ):
  mx=-1
  mx_ind=0
  for i in range(pred.shape[0]):
   if pred[i] > mx:
     mx=pred[i]
     mx_ind=i
  className = class_names[mx_ind] + " " + str(round(mx,3))
  color=(0,0,0)
  if(mx_ind==0):
     color=(0,255,0)
  elif(mx_ind==1):
     color=(0,255,255)
  else:
     color=(0,0,255)
  return className, color
```

5.3 Конечный результат

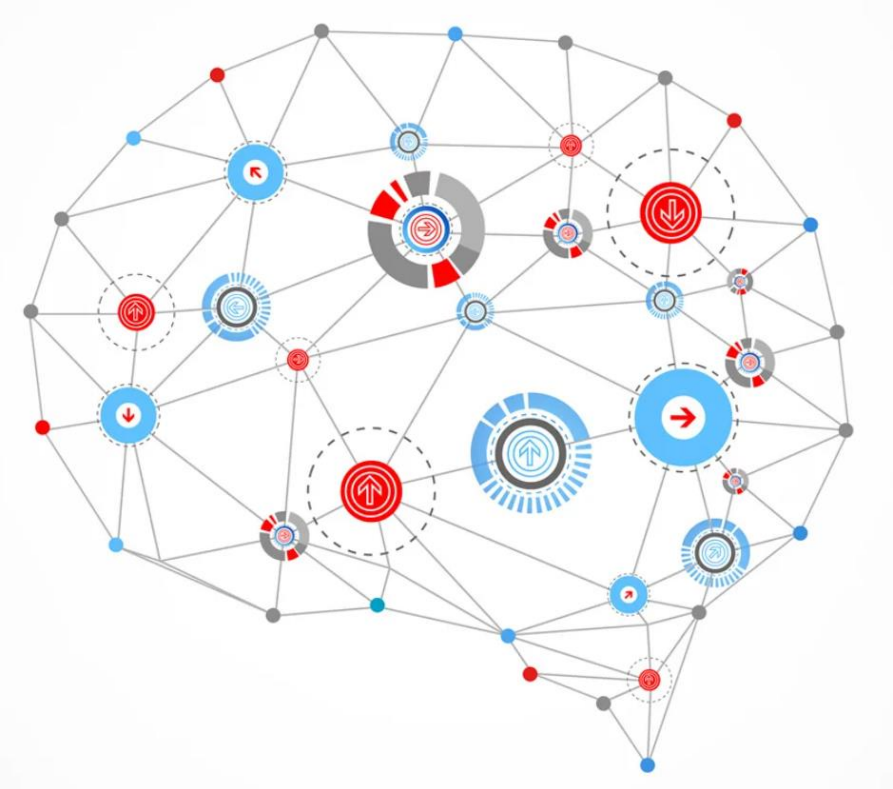
Посмотрим на результат всех проделанных работ. На левом верхнем углу отображается частота обработанных кадров, в среднем 30 кадров в секунду (30 FPS). Сверху прямоугольника – вероятность, предсказанная моделью. Также меняется цвет прямоугольник и надписи в зависимости от класса: зеленый, желтый, красный.

Рисунок 44 – Примеры выполнения программы

Заключение

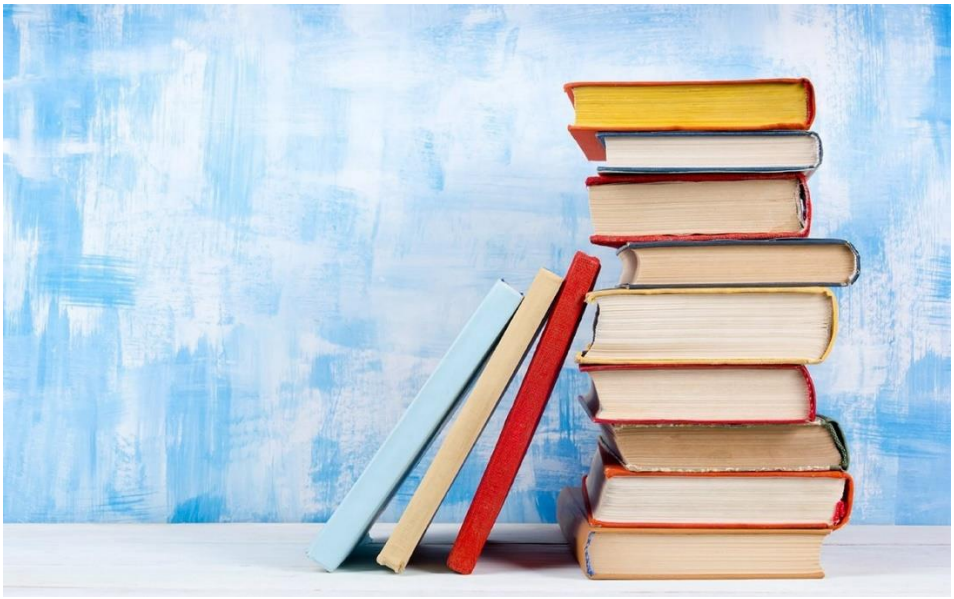
В данной курсовой работе была исследована и рассмотрена задача классификации Компьютерного зрения, а также применение моделей обнаружения лиц на практике. Материалами для обучения модели были наборы данных, собранные с открытых источников, а также предварительно обученная модель Глубокого обучения MobileNetV2. В ходе работы мы обучили модель на своих данных и использовали для классификации в режиме реального времени. Так как мы использовали предварительно обученную модель, наша собственная модель показала очень хорошие точности в классификации. Проанализировали результаты обучения, и еще немного улучшили точность. В режиме реального времени наш алгоритм работает в среднем с 30 FPS, с довольно высокой точностью.

Цель, поставленная в начале работы была достигнута, задачи выполнены.



Список литературы

1. Гудфеллоу Я. Глубокое обучение / Я. Гудфеллоу, И. Бенджио, А. Курвилль. – М.: ДМК Пресс, 2017. – 652 с.
2. Жерон О. Прикладное машинное обучение с помощью Scikit-Learn, Keras и TensorFlow / О. Жерон. М.: Вильямс, 2020. – 1040 с.
3. Нейронные сети и компьютерное зрение / Samsung Research Russia Open Education. https://stepik.org/course/50352/info (15.11.2021 г.)
4. Николенко С. Глубокое обучение. Погружение в мир нейронных сетей / С. Николенко, А. Кадурин, Е. Архангельская. – СПб.: Питер, 2020, – 480 с.
5. Специализация Глубокое обучение / Andrew Ng. https://www.coursera.org/specializations/deep-learning (15. 11. 2021 г.)
6. Шолле Ф. Глубокое обучение на Python / Ф. Шолле. – СПб.: Питер, 2012. – 400 с.
7. Deep Learning and Computer Vision A-Z™: OpenCV, SSD & GANs / Eremenko K. https://www.udemy.com/course/computer-vision-a-z (15.11. 2021 г.)



Приложение

```
import cv2
import numpy as np
import tensorflow as tf
import time
import matplotlib.pyplot as plt
import mediapipe as mp
def predict(img):
   img = tf.expand_dims(np.float32(img), 0)
   interpreter.set_tensor(input_details[0]['index'],img)
   interpreter.invoke()
   prediction = interpreter.get_tensor(output_details[0]['index'])
   return prediction
def decode_pred(pred,class_names ):
   mx=-1
   mx_ind=0
   for i in range(pred.shape[0]):
    if pred[i] > mx:
      mx=pred[i]
      mx_ind=i
   className = class_names[mx_ind] + " " + str(round(mx,3))
   color=(0,0,0)
   if(mx_ind==0):
      color=(0,255,0)
   elif(mx_ind==1):
      color=(0,255,255)
   else:
      color=(0,0,255)
   return className, color
interpreter = tf.lite.Interpreter(model_path='MaskNet.tflite')
input_details = interpreter.get_input_details()
output_details = interpreter.get_output_details()
interpreter.allocate_tensors()
mpFaceDetection = mp.solutions.face_detection
faceDetection = mpFaceDetection.FaceDetection(0.75)
while cap.isOpened(): #Пока камера работает
      ret, img= cap.read() # Считаем очередной кадр из камеры
      start_time=time.time() #Счетчик для FPS
      img=cv2.flip(img,1) #Отразим кадр горизонтально, так более привычно для нас
      imgRGB=cv2.cvtColor(img, cv2.COLOR_BGR2RGB) # Конвертируем BGR изображение на RGB
      results = faceDetection.process(imgRGB)
      if results.detections:
            for id, detection in enumerate(results.detections):
                  bboxC = detection.location_data.relative_bounding_box
                  ih, iw, ic = img.shape
                  x, y = int(bboxC.xmin * iw), int(bboxC.ymin * ih)
                  w,  h =int(bboxC.width * iw),int(bboxC.height * ih)
                  face_img= cv2.resize(imgRGB[y:y+h+20, x:x+w], (160,160))
                  prediction=predict(face_img) # Далее передаем в функцию  для предсказания
                  className, color=decode_pred(prediction[0], class_names,img)
                  cv2.putText(img, className , (x, y - 10), cv2.FONT_HERSHEY_SIMPLEX,
                                                                        0.8, color, 2)
                  cv2.rectangle(img, (x, y), (x + w, y + h), color, 3)
                  fps=1.0/(time.time() - start_time)
                  cv2.putText(img, "FPS:" + str(int(fps)),(10,30),
                                          cv2.FONT_HERSHEY_SIMPLEX, 1.3, (255,0,255), 2)
            cv2.imshow('img', img)
```