Отчет о проделанной работе

БДЗ-1

Парамонов Всеволод

Contents

1 Обработка данных

1.1 Обзор выборки

В первую очередь было необходимо подготовить данные для обучения моделей. Для этого был написан кастомный **DataLoader**, внутри которого на вход подается изображение и из csv-файла этому изображению присваивался соответствующий класс. По итогам этого этапа можно было рассмотреть изображения и сделать примерное представления что из себя представляет тот или иной класс:

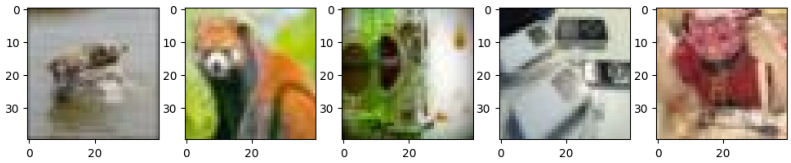

Figure 1: Примеры изображений

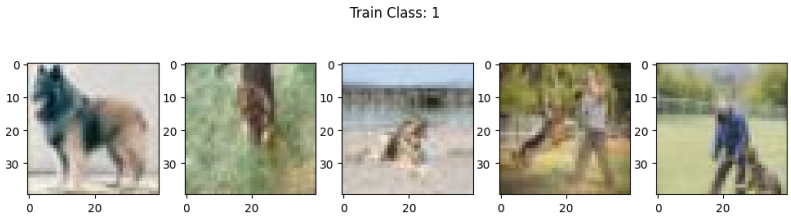

Figure 2: Примеры изображений из класса 1

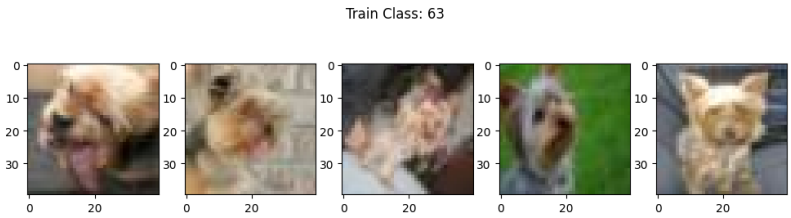

Figure 3: Примеры изображений из класса 63

Также в выборке можно заметить равномерное распределение классов, что наводит на мысль о дальнейшем стратифицированном делении выборки, чтобы сохранить это свойство, т.к. классов много и существует вероятность, что присутствие какого-то класса в батчах будет мало:

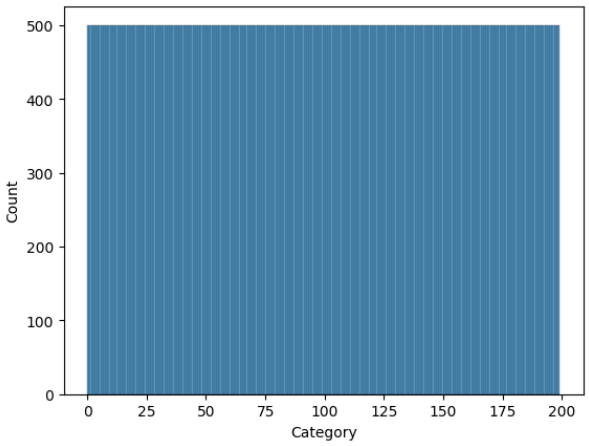

Figure 4: Распределение классов в обучающей выборке

1.2 Разбиение на Train / Val

В итоговом пайплайне 20% выборки отходило на валидацию. Итого в обучающей выборке находится 80 000 изображений, в валидационной - 20 000. Далее было необходимо установить параметры для **DataLoader**'а, а именно размер батча, что играет важную роль. Были опробованы батчи размером 32, 64, 128, но по итогу лучший результат был достигнут при размере батча равным 64

В этом блоке есть момент, который потенциально может улучшить показатели модели: увеличение размера обучающей выборки. На первых итерациях я отделял 30% на валидационную выборку и после уменьшения этой доли показатели модели улучшились. Также я пробовал делить выборки стратифицированно, но это не сыграло сильной роли

1.3 Аугментация данных

Для регуляризации моделей использовались разлчиные комбинации методов аугментации. В финальный сеттинг вошли следующие аугментации

- **RandomResizedCrop**
- **RandomHorizontalFlip**
- **RandAugment**
- **Normalize** (с параметрами для ImageNet)
- **RandomErasing**

Также стоит упомянуть, что также в тестах вместо **RandAugment** использовались методы аугментации **AutoAugment** и **TrivialAugmentWide**, который является SOTA

для аугментаций. Использование последнего действительно улучшало обобщающую способность модели (качество на валидационной выборке было сравнительно выше, чем на обучающей), но начиная с определенной эпохи преимущество данного метода нивелировалось и модели для обоих наборов аугментаций сравнялись в качестве

Дополнительной мерой регуляризации выступали такие аугментации как **CutMix** и **MixUp**, которые рандомно выбирались на каждой эпохе. Включение данных аугментаций значительно улучшило обобщающую способность моделей

По ходу экспериментов с аугментациями проходил контроль, что аугментации не слишком сильные и не "путают" модель. Батчи для обучающей и валидационной выборках после применения финального набора аугментаций выглядят следующим образом:

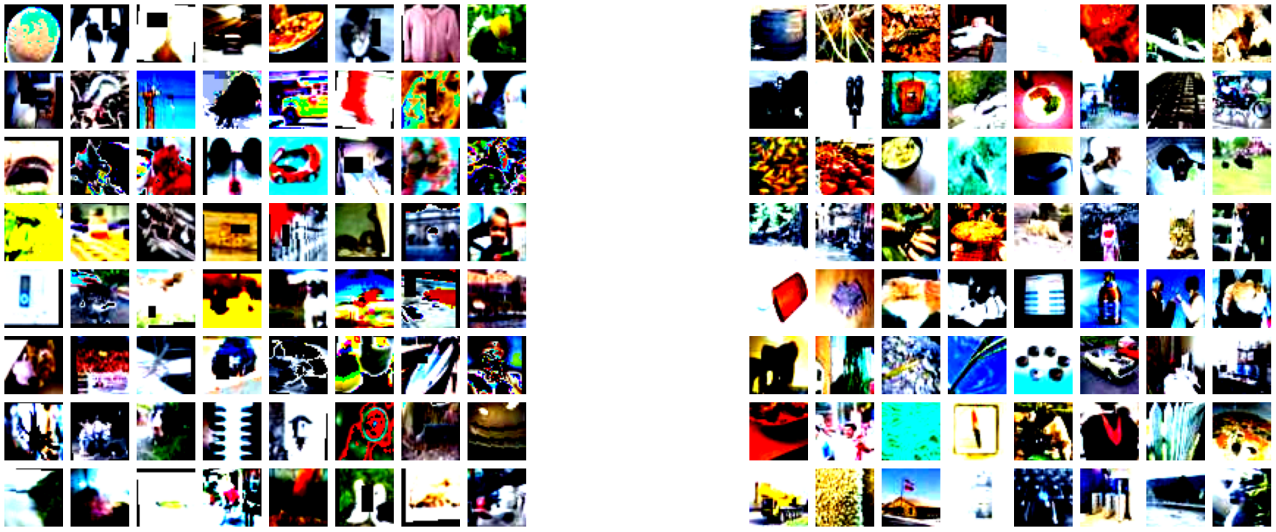

Figure 5: Батч с изображениями для обеих выборок

2 Мониторинг / Обучение

2.1 Сеттинг для обучения

Для обучения всех моделей использовался стандартный код для обучения нейронных сетей, взятый из ДЗ-2. В качестве лосса использовался CrossEntropyLoss с дополнительной регуляризацией в виде LabelSmoothing $= 0.1$. Также по мере обучения в Kaggle сохранялись 2 файла: файл с весами, параметрами оптимайзера эпохи, в которой качество на валидационной выборке было максимальными и похожий файл, но с результатами последней эпохи. Это все было сделано в целях как дальнейшего переиспользования весов в роли инициализации для других моделей, так и дальнейшего инференса

2.2 Мониторинг результатов

Для контроля результатов было необходимо придумать способ для мониторинга результатов, а также сохранения архитектур и параметров моделей, использование которых дало наилучшее значение Accuracy. Поскольку комбинаций архитектур, оптимизаторов, шедулеров и т.д. бессчетное количество, то для того, чтобы была возможность опробовать как можно больше комбинаций, было создано некоторое количество аккаунтов в Kaggle. Для агрегированного мониторинга всех моделей сразу мною был написан телеграм-бот, который уведомлял о промежуточных результатах каждой из моделей, а также на вход перед каждым запуском принимал комментарии, которые описывают проводимый эксперимент:

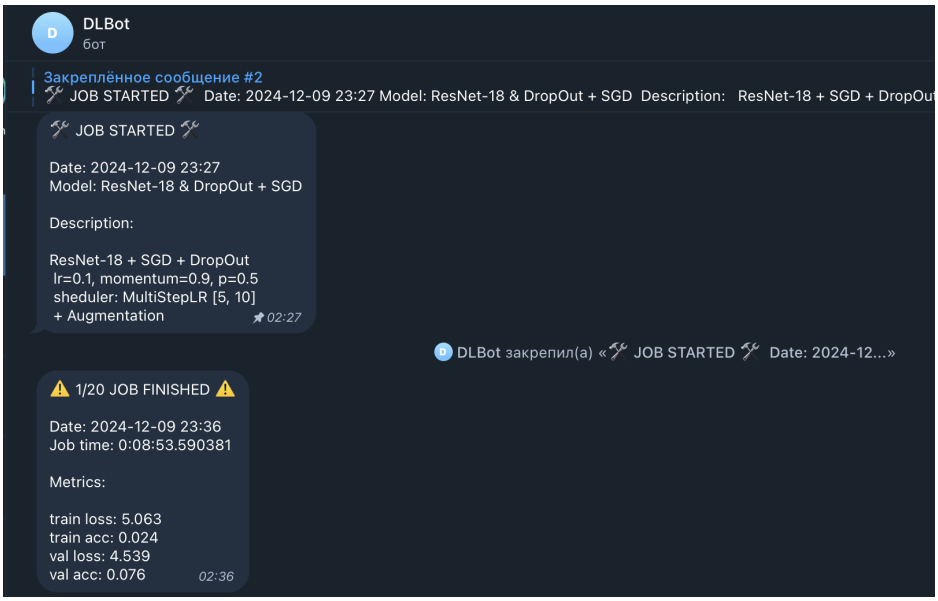

Figure 6: Пример мониторинга эксперимента

Проблема использования WnB была в том, что, я не был уверен в возможности логирования одновременно всех моделей из разных тетрадок при параллельном запуске. Так как длительные запуски были в моменты, когда я был не у компьютера, то мне было важно удобно наблюдать за текущими результатами, таким образом логирование в телеграм боте позволяло быстро узнать актуальные значения метрик. К тому же у меня уже были написаны некоторые заготовки, поэтому их доработка не заняла большого количества времени

3 Эксперименты

Основным шагом является выбор архитектуры модели. Для этого я нашел похожие задачи в интернете (Tiny ImageNet, CIFAR100) и выбрал пул моделей, которые показали наилучшее качество на изображениях небольшого размера По ходу выполнения экспериментов сформировалось требование, что модель должна быть простой, потому что:

- В больших архитектурах достаточно тяжело разобраться в составляющих, что автоматически влечет усложнение адаптации этой архитектуры под небольшие изображения
- Большие модели достаточно долго обучаются, что автоматически приводит к уменьшению числа экспериментов (об этом будет далее)
- Большие модели достаточно быстро переобучаются и требуется большое количество методов регуляризации, чтобы улучшить обобщающую способность $\Rightarrow$ бОльшее количество комбинаций техник для регуляризации, которые надо попробовать. Но как было упомянуто в пункте выше из-за долгого обучения больших моделей подбор этих регуляризационных методов занял бы большое количество времени. Поэтому гораздо проще и быстрее было использовать небольшие архитектуры (как **ResNet18** или **ResNet34**) и в случае если они окажутся слабыми, то усложнить их за счет добавления линейных слоев в выходе модели

3.1 Swin & ViT

Были также опробованы трансформеры, такие как **Swin** и **ViT**, но как было упомянуто в начале главы с экспериментами, обучение одной эпохи занимало огромное количество времени. Например, на обучение одной эпохи **Swin** ушло 1.5 часа и качество было сравнимо с несколькими эпохами **ResNet18** (на них уходит гораздо меньше времени)

К тому же лишь после эксперимента я увидел в документации в PyTorch, что минимальный размер входного изображения 240x240 пикселей, что по идее должно было бы заставить задуматься о рациональности использования этих моделей

3.2 EfficientNet

Одной из рассматриваемых моделей был **EfficientNet-B0**. Поскольку тестирование архитектур происходило от наиболее поздней к наиболее ранней по дате выхода (и соответственно на первых попытках мой опыт в решении таких задач был не очень велик), то результаты получились соответствующие. В данном сеттинге я использовал не обычный transform в DataLoader, а расширял обучающую выборку агументациями

Что касается архитектуры, то тут я ничего не менял кроме выходного слоя. Со временем я понял, что это очень грубая ошибка, поскольку стоило получше узнать архитектуру модели и поэкспериментировать с параметрами входного слоя (например, уменьшить *stride* до 1). Получились следующие результаты:

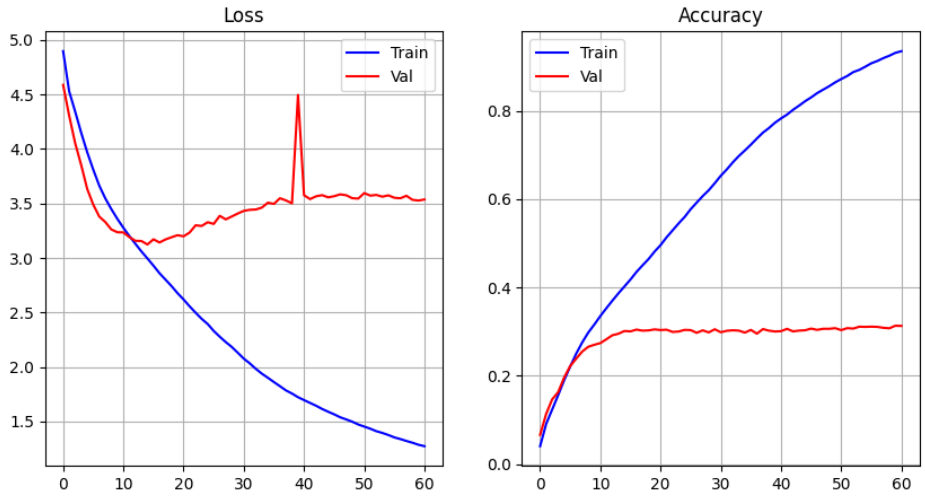

Figure 7: Результаты EfficientNet-B0

По графику видно, что модель быстро переобучилась и на выходе получилось плохое качество на валидационной выборке. Поэтому и было принято решение остановиться на этой модели, поскольку дальнейшее использование более глубоких модификаций скорее всего не привело бы к улучшению качества модели

Что потенциально могло помочь: усиление регуляризации (например, добавить Dropout в выходном слое или увеличить $weight\_decay$), добавление более сильных аугментаций в бОльшем количестве или изменение параметров входного слоя

3.3 DenseNet

Далее были протестированы архитектуры **DenseNet**, а именно их неглубокие версии (**DenseNet-121**). На данном этапе я уже не расширял выборку аугментациями, а использовал transform внутри DataLoader'a. Это в целом ускорило процесс обучения и при этом сохранило ее качество. Также на данном этапе я поменял параметры входного слоя под изображения размером 40x40, чтобы модель была более адаптирована под текущую задачу. Были опробованы оптимизаторы **AdamW**, **SGD** и **Adam**, но по итогам лучше всего оказался **AdamW**. Тут стоит отметить, что во всех экспериментах с разными моделями были опробованы различные оптимизаторы и **AdamW** в большинстве случаев показал сравнительное преимущество относительно других. В итоге получились следующие результаты:

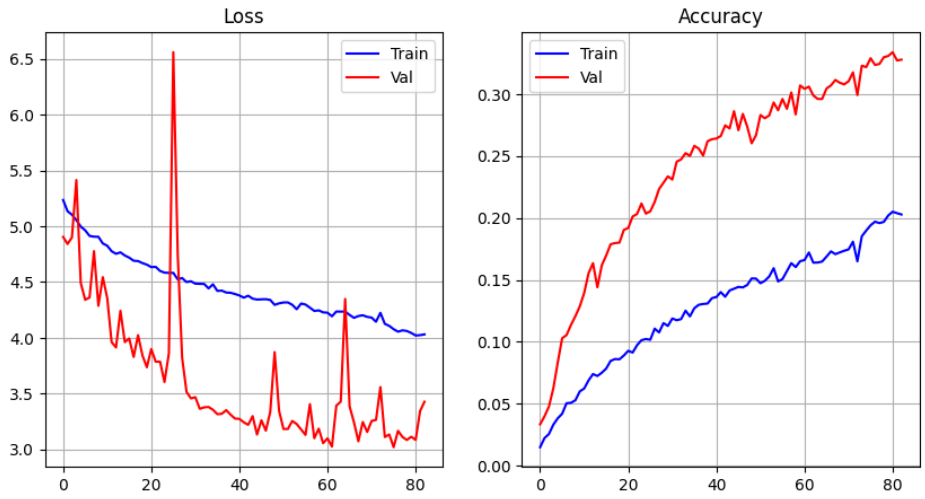

Figure 8: Результаты DenseNet-121

Тут уже видно улучшение относительно предыдущего эксперимента, но все же есть ряд проблем. Во-первых, аномальное поведение лосса на валидационной выборке: по ходу обучения возникает ряд скачков, что может говорить о нестабильных вычислениях. Во-вторых, медленный темп обучения. Это может быть связано с изначально маленьким значением learning rate. В этом также можно убедиться по графику точности: спустя 80 эпох качество на валидационной выборке достаточно низкое, хотя и видно, что модель хорошо описывает данные и на данном этапе неактуальна проблема переобучения

Что потенциально могло помочь: из-за медленного темпа обучения можно было вначале использовать несколько раозгревочных эпох с достаточно высоким learning rate или вначале установить более высокое значение. На графике изображен процесс обучения с learning rate $= 1e^{-3}$, возможно, для данной модели это довольно маленькое значение

3.4 ResNet

Финальный выбор пал на семейство моделей **ResNet** в силу их простоты, большого количества сеттингов в интернете для решения схожих задач и скорости обучения. Также использование **ResNet** дает возможность по-разному усложнять ту или иную конфигурацию достаточно простым способом: как говорилось ранее, если модель недообучается, то можно ее усложнить добавлением выходных слоев

Вначале был опробован **ResNet18**. В целом с **ResNet18** было проведено бОльшая часть экспериментов, поскольку при небольших изменениях архитектуры добивались наилучшие результаты в плане процесса обучения: оно было стабильным и быстрым, что позволяло

совершать большое количество экспериментов с этой моделью. Вначале был использован следующий сеттинг:

- Уменьшение $kernel\_size$ до 3 во входном слое
- Изменение **MaxPooling** на **AdaptiveMaxPooling** с целью уменьшения количества гиперпараметров
- Добавление **Dropout(p=0.5)** в выходном слое
- **SGD** в качестве оптимизатора с начальными значениями $lr = 1e^{-3}$, $momentum = 0.9$, $weight\_decay = 1e^{-3}$
- **ReduceLROnPlateau** в качестве шедулера с параметрами $factor = 0.5$ и $patience = 8$

Использование данного сеттинга позволило достичь следующих результатов:

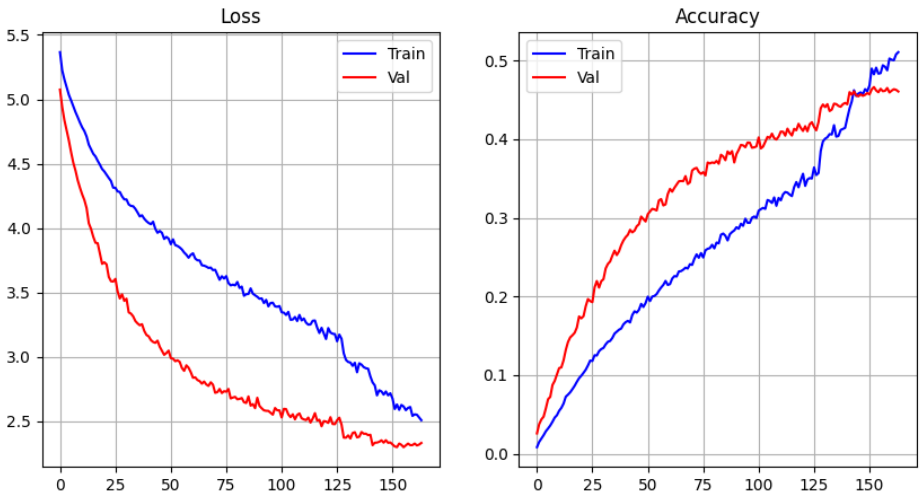

Figure 9: Результаты ResNet18

На графиках виден заметный прирост: удается пробить качество 40%+ на валидации. Однако все же присутствует ряд проблем, а именно: резкое переобучение на 125 эпохе и медленное обучение. Это может быть вызвано недостаточной регуляризации, однако если заглядывать в будущее, то увеличение сложности модели помогло улучшить качество, но переобучение также осталось

Финальный сеттинг, который дал качество 45.9% точности на сабмите выглядит следующим образом:

- За основу была взята модель **ResNet34**

- Во входном слое был изменены параметры свертки, а именно уменьшение $kernel\_size$ до 3
- В выходном слое был добавлен **Dropout(p=0.5)**
- Использовался оптимизатор **AdamW** с параметрами $lr = 1e^{-3}$ и $weight\_decay = 1e^{-3}$
- Использовался шедулер **ReduceLROnPlateau** с $factor = 0.1$ и $patience = 10$

После обучения 100 эпох получился следующий график ошибки и точности:

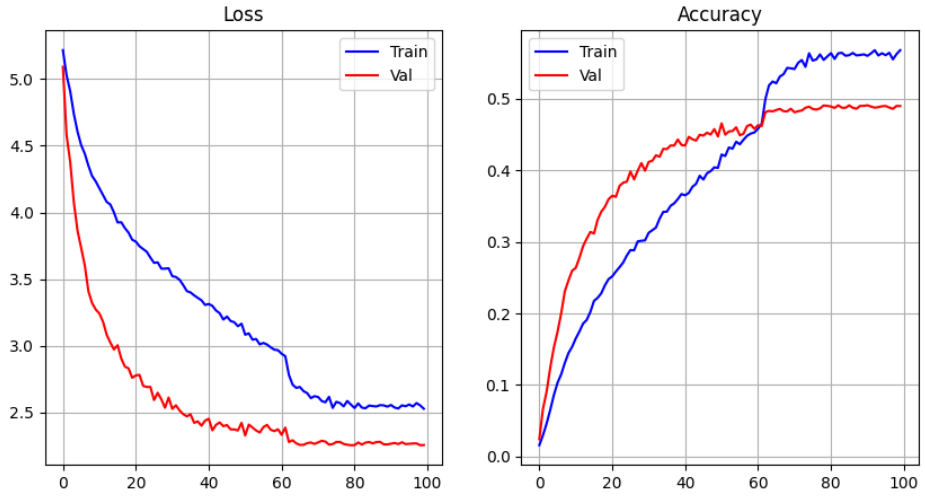

Figure 10: Результаты ResNet34

По графика видно, что до 60 эпохи обучение идет стабильно: точность на валидации превышает точность на обучающей выборке, ошибка на валидации уменьшается быстрее, чем ошибка на обучающей выборке. Но после 60 эпохи видно, что модель начала переобучаться и как ошибка, так и точность вышли на плато на валидационной выборке. Причиной этому может быть недостаточная регуляризация: возможно стоило увеличить кол-во аугментаций в **RandAugment** или же увеличить их магнитуду

Мною была выдвинута гипотеза, что значение $patience = 10$ на поздних итерациях это достаточно много, поэтому было опробовано на более поздних этапах установить значение $patience = 2$ и $factor = 0.6$, чтобы learning rate не так сильно уменьшался и не останавливал процесс обучения. К сожалению, данный способ не дал значительного роста

В итоге данный сеттинг принес качество $\sim$49% на валидационной выборке и $\sim$45.9% на сабмите

4 Выводы

По итогам всех экспериментов были сформированы несколько предложений, которые в теории могли бы улучшить финальный результат:

1. **Усложнение модели:** Мною были опробованы и более сложные архитектуры, например, **ResNet50**, но с этой архитектурой возникли некоторые сложности в настройке. Например, если для **ResNet18** и **ResNet34** набор оптимизатора и шедулера давали одинаково хорошие результаты, то для **ResNet34** требовались дополнительные исследования подходящего сеттинга

2. **Аугментации:** Возможно, мною были использованы недостаточные хорошие аугментации и следовало уделить им бОльшее внимание, поскольку большинство экспериментов показывали, что модель либо недообучалась, либо переобучалась

3. **Смена постановки задачи:** С течением времени возникла идея поменять задачу с классификации изображений на задачу **Image retrieval**. Но тут могли возникнуть трудности с реализацией данного подхода. К тому же интуитивно кажется, что итоговое качество такого подхода не будет превышать качество сверточных сетей