Transfer Learning for Computer Vision

1) Задача состоит из двух частей:

Применение нейросетей типа AlexNet и VGG16, предобученных на ImageNet, для бинарной классификации изображений животных (муравьи и пчелы)

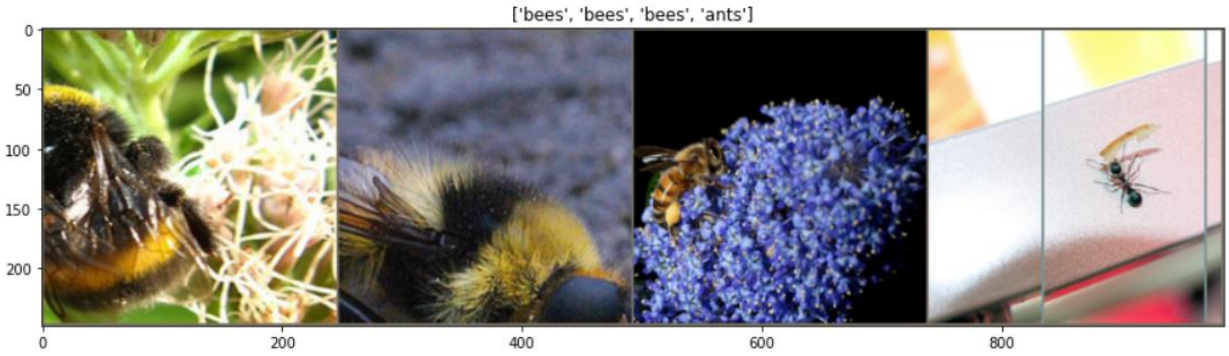
Сравнение нескольких способов применения метода Transfer Learning'a к каждой из предобученных сетей.

Основано на https://pytorch.org/tutorials/beginner/transfer_learning_tutorial.html

Критерий качества – ROC AUC score

2) Стоит задача бинарной классификации.

Датасет состоит из изображений, на каждом из которых находится 1 объект: пчела или муравей.



Данных мало, даже после аугментации изображений (половина изображений из обучающего множества была горизонтально отзеркалена и добавлена в датасет) у нас всего 244 изображения для обучающего множества и 153 для тестового

Однако видим равномерное распределение по классам

```
Images for  train  total: 244
ants: 123  bees: 121

Images for  val  total: 153
ants: 70  bees: 83
```

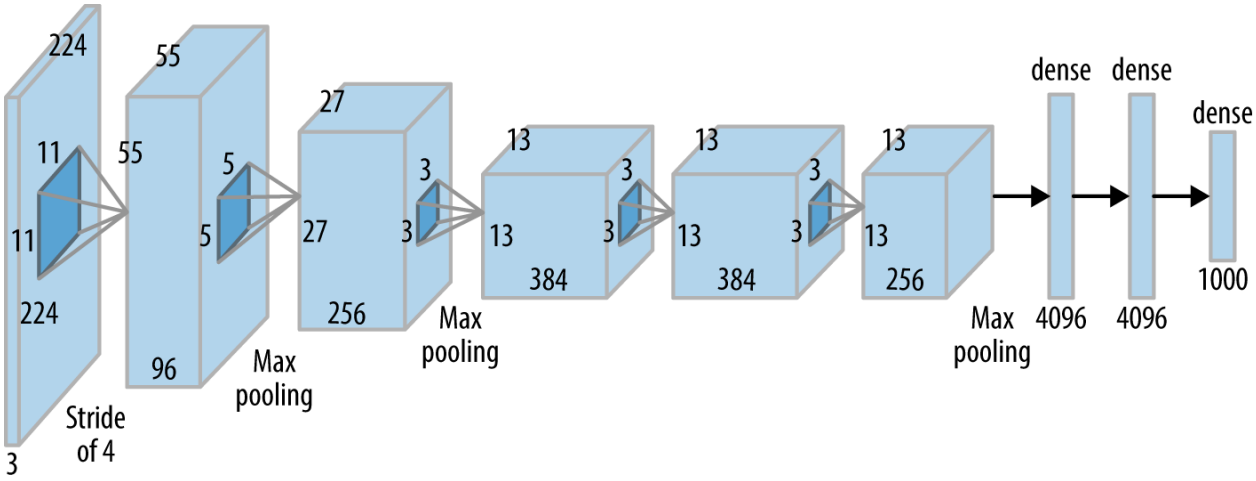
Производимые действия с изображениями:

1) На обучающей выборке проводим аугментацию изображений (о ней выше) – только на обучающей, потому что на тестовой выборке датасет должен быть один и тот же (данные, на которых мы валидируемся должны быть детерминированы)
2) Нормализация (вектор из средних по трем каналам и вектор из стандартных отклонений по трем каналам, нормализация проводится именно теми же числами, которые получились и на ImageNet’e)
3) На обучающей выборке случайно вырезаем изображение 244x244 пикселя (чтобы сеть не зацикливалась, например, на том, что большинство муравьев всегда находится в центре изображения)

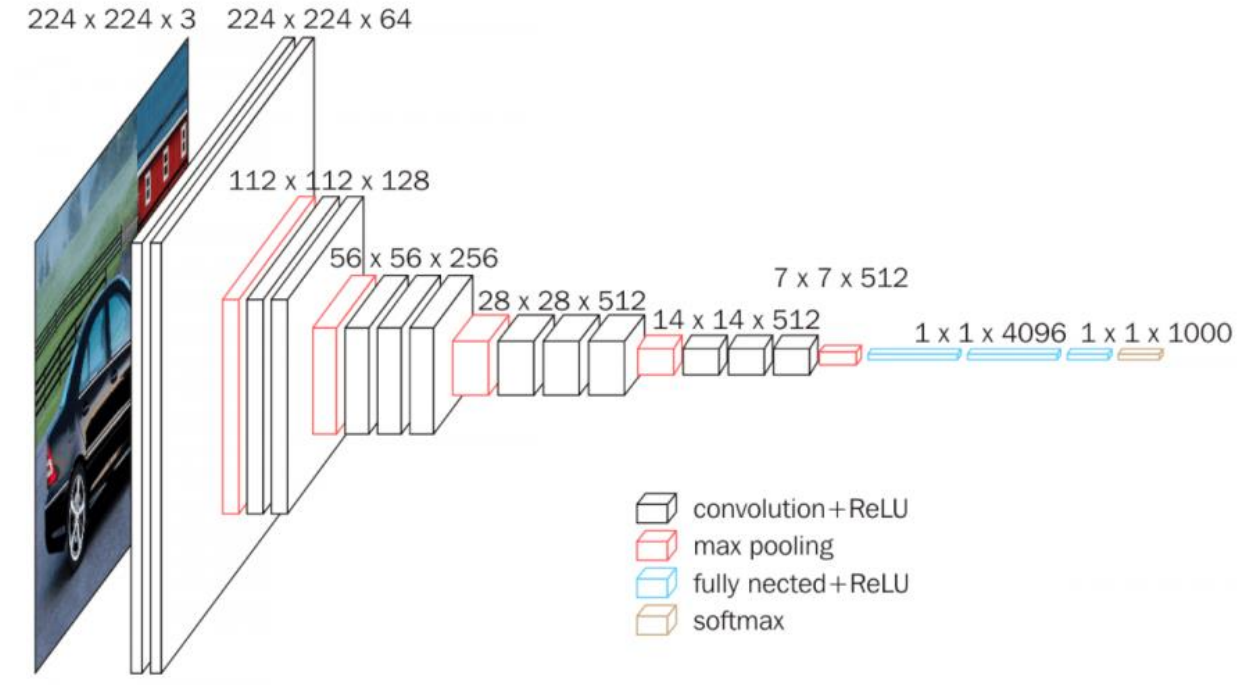
3) Изображений очень мало, скорее всего если обучать сверточную нейронную сеть на этом датасете с нуля, то она переобучится. Именно ради того, чтобы научиться решать задачи, когда у нас имеется мало объектов (медицинские датасеты изображений опухолей, машинный перевод с малораспространненых языков (татарский), etc) я хочу разобраться с тем, как лучше всего использовать метод Transfer Learning'а

«Дообучивать» будем две предобученные на датасете ImageNet сверточные нейронные сети – AlexNet и VGG-16 (обе эти сети победили в конкурсе в 2012 и 2014 годах соответственно).

Архитектура AlexNet



Архитектура VGG-16



Как можно увидеть, обе сети используют свертки (правда разного количества), макс пулинги, одинаковые функции активации (ReLU). У каждой из них по три полносвязных слоя (которые нас не интересуют). Да и на самом деле нас не очень интересует сама «начинка» этих сетей. Нам лишь важно то, как работают свертки:

Из такой очень грубой репрезентации работы сверточных слоев мы можем понять, что первые слои свертки определяют прежде всего какие-либо низкоуровневые паттерны (вертикальные, горизонтальные границы и т.д), чем дальше – тем наша сеть более подстраивается под какую-то конкретную задачу. Это небольшое отступление необходимо будет нам для обоснования метода «размораживания последних сверточных слоев»

Так в чем же идея? Давайте возьмем и загрузим к себе эти две нейросети (не просто их структуру, а уже обученные сети - в них уже «заложены очень хорошие веса»). Но теперь мы удалим всю часть так называемого классификатора (classifier) – то есть все наши полносвязные слои. То, что у нас осталось – сверточные слои (features):

Для каждой из моделей правда предварительно запоминаем, сколько in_features приходило на вход для первого линейного слоя (это логично, так как и нашему слою будет приходить столько же). Для AlexNet это число равно - `9216`, для VGG16 - `25088`

Теперь создадим лишь 1 FC слой `nn.Linear(num_features, 2)` у которого на выходе будет теперь лишь 2 значения – один из наших классов (муравей или пчела)

Функция для обучения – train_model (model, criterion, optimizer, scheduler, num_epochs=25)

Давайте обсудим каждый параметр, который в неё передается:

- model − нейросеть
- loss − оптимизируемая функция ошибки (criterion, cost function, objective) – здесь для каждой модели использую Кросс-Энтропию

- optimizer − оптимизационный алгоритм – использую Adam – по совету (если не можете выбрать, какой оптимизационный алгоритм выбрать – выбирайте Adam и не ошибетесь)
- scheduler − политика изменения learning_rate (на первых этапах обучения мы очень быстро «учимся», лосс падает очень резко, однако далее мы не хотим, чтобы возникла ситуация, когда мы «перепрыгнем» через какой-либо минимум из-за большого learning rate'a, именно поэтому в данном решении я умножаю каждые 7 эпох learning rate на 0.1)
- num_epochs − количество итераций обучения

Функция помимо весов лучшей модели возвращает также список из losses на обучающей и тестовой выборке, список из accuracy для каждой эпохи (это понадобится нас для визуализации процесса обучения). Размер батча – 4 картинки

4) Язык программирования – Python. Основные используемые библиотеки:

1) Pytorch – основная библиотека для обучения

2) torchvision – визуализация данных с датасета

3) matplotlib, seaborn – рисование графиков

4) numpy – иногда используем numpy array (с ними работать привычнее)

5) tqdm.autonotebook – библиотека для удобной визуализации процесса обучения

6) sklearn.metrics – нужна для импортирования roc_auc_score

Для каждой из моделей вызываем функцию обучения (про неё в 3 пункте), которая возвращает лучшие веса модели и списки с лоссом (на трейне и тесте) и accuracy.

5) https://colab.research.google.com/drive/1FEhE-KXrg0jLVaaR8FSTAAtVQfS-309Z?usp=sharing

6 и 7) Переходим непосредственно к самому исследованию, что мы хотим узнать? Как лучше всего применять Transfer Learning к такого рода задачам? Тогда возникает еще один вопрос, как его вообще можно применять? Я решил выделить для себя 4 основных пути (о которых конечно же почитал в интернете):

1) запускаем полный цикл обучения на предобученной сети

2) замораживаем все сверточные слои (feature extractor)

3) заморозим какое-то количество первых сверточных слои (которые отвечают за распознавание низкоуровневых паттернов), так как последние слои представляют собой всё более абстрактные свертки, которые могут не подойти для нашей задачи

4) уменьшим градиент на первых слоях (веса на них будут все меньше меняться, зато для последних слоев (как раз этих абстрактных) обучение будет происходить полностью

То есть всего будет 8 моделей, 4 для AlexNet, 4 для VGG16, в конце сравним результаты. Для каждого обучения будем использовать число эпох = 25.

Начнем с AlexNet’a

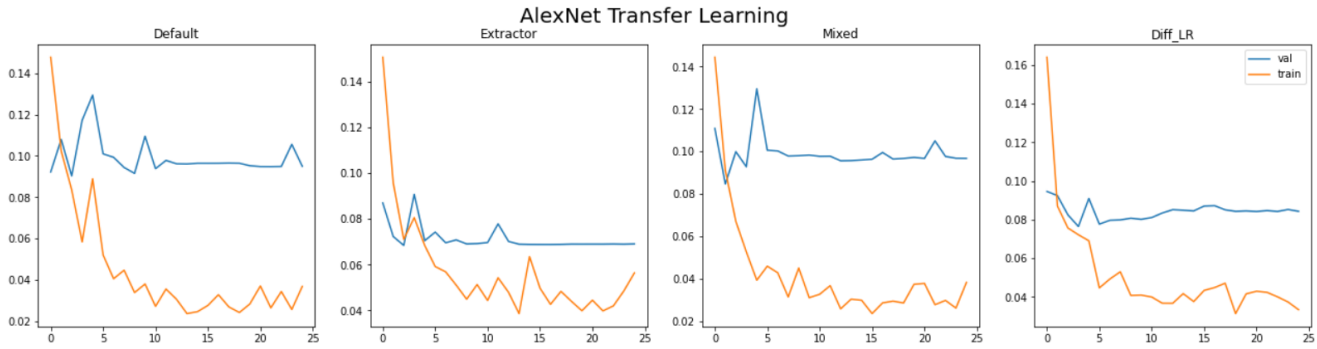


График Функции ошибок для тренировочных и тестовых данных

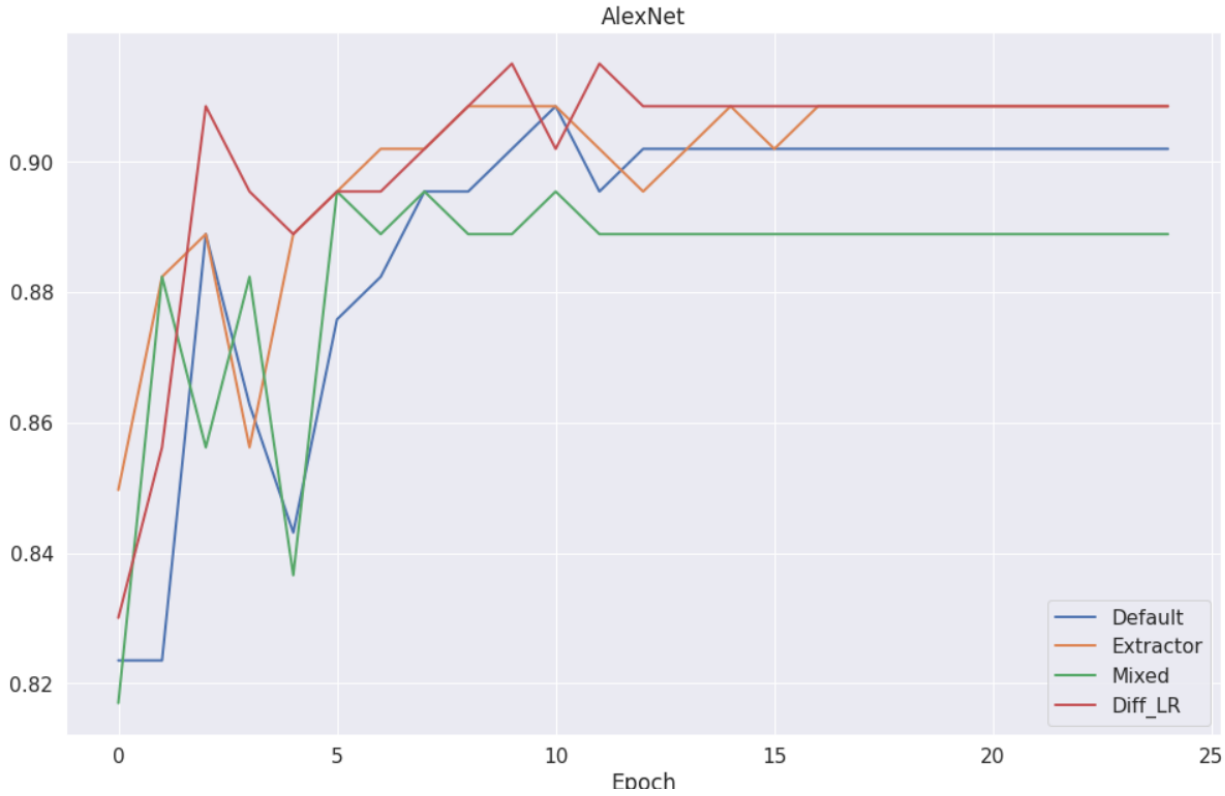


График метрики качества для тестовых данных

Какие выводы можно делать по этим графикам?

1) Начнем с первого и третьего случаев. Почему я их объединил? Потому что вероятнее всего на этих способах модель переобучилась. Почему так? Для первого все очевидно – у нас очень мало данных и поэтому она склонна к переобучению, мы ведь дали возможность обучаться на всех слоях (ничего не замораживали). Для третьего уже не все так просто – мы ведь заморозили часть слоев, откуда здесь переобучение – если посмотреть на устройство AlexNet – у нас есть 12 слоев, из которых 5 обновляют градиенты. Мое предположение: разморозил слишком много слоев для такой маленькой сети (оно во многом основывалось на том, что в случае с 30 слоями VGG всё получилось)
2) Для Feature Extractor'а и уменьшения градиентов первых слоев все прошло удачно! Модели успешно обучились, критичного переобучения заметно не было
3) По времени обучения все случаи примерно равны 2 минутам, разница не критична

Перейдем к VGG16

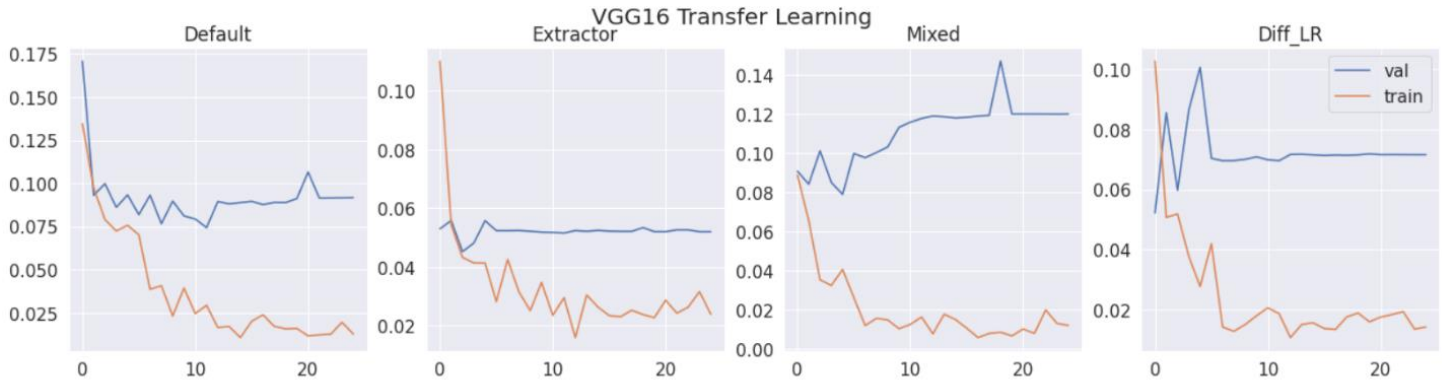


График Функции ошибок для тренировочных и тестовых данных

График метрики качества для тестовых данных

Какие выводы можно делать по этим графикам?

1) Для первого случая все идентично предыдущему – модель переобучилась
2) Для третьего уже все не так просто – модель показывает прекрасные результаты на оценке качества (то есть теперь по моему предположению из-за того, что слоев у нас больше, она менее склонна к переобучению)
3) Feature Extractor и уменьшения градиента первых слоев опять показывают отличный результаты!
4) По времени обучения модели показывают уже существенные различия:
    1) 10 минут 0 секунд
    2) 5 минут 36 секунд
    3) 6 минут 0 секунд
    4) 10 минут 17 секунд

    По которым можно смело сказать: первый способ один из самых затратных по времени и не приносит хороших результатов. Второй способ – самый лучший в

рамках данной задачи, по времени он самый оптимальный и результаты показывает всегда отличные. Четвертый способ – затратный по времени, однако результаты стабильно хорошие

Про третий способ, как и в целом про Transfer Learning хочется сказать (подводя итог):

Для каждой задачи (распознавание животных, распознавание медицинских изображений и т.д) стоит подбирать свой способ решения и экспериментировать!

Где-то разморозка последних слоев просто необходима, так как данные, на которых сеть обучалась очень сильно отличаются от тех, на которых мы хотим обучить сейчас. Если же они примерно совпадают (как у нас сейчас), то второй способ является оптимальным.

Чем я пользовался:

1) https://pytorch.org/tutorials/beginner/transfer_learning_tutorial.html
2) https://cs231n.github.io/transfer-learning/
3) https://habr.com/ru/company/binarydistrict/blog/428255/
4) https://python-school.ru/wiki/transfer-learning/