- Познакомился со стандартныйм аугментором Keras - ImageDataGenerator, но в зачётный код он не вошёл, т.к. его заменил ImageDataAugmentor - более продвинутый аугментор на основе albumentations

А в чем смысл установки ImageDataAugmentor, если в ImageDataGenerator есть параметр preprocessing_function, где тоже можно выполнить аугментацию через albumentations?

- Также имеется страничка, где можно наглядно подобрать аугментации, подходящие для задачи

Не знал о таком, очень полезная вещь.

С другой стороны, сейчас нечасто при файн-тюнинге делают много аугментаций. Например, в рекомендациях по файн-тюнингу модели BiT предлагают делать только random flip, resize и crop (что можно сделать с помощью слоев keras). Я также замечал, что более сложные агументации, например поворот изображения, не всегда дают прирост точности, но могут увеличить время, затрачиваемое на одну эпоху. Кроме того сильные аугментации могут снизить точность на валидации из-за проблемы, которая известна как «batch normalization train-test discrepancy».

```
preprocess_input = tf.keras.applications.densenet.preprocess_input,
```

В качестве альтернативы можно встроить слой нормализации в модель. Тогда изображения из генератора будет легче визуализировать, потому что они не будут нормализованы.

- Исследовательские запуски моделей

Было бы более наглядно привести графики точности на обучении и на валидации для каждой из моделей. Тогда ситуация моглоа бы стать более понятной. Например, некоторые модели могут иметь большой момент в батч-нормализации, из-за чего точность на валидации получается ниже и нестабильнее. Эта проблема исчезает после достаточного количества эпох (например 10-20).

```
Dense(256, activation='elu')(x)
```

А вы уверены, что этот слой нужен? Больше слоев не всегда значит лучше. Можно было проверить и сравнить.

```
remmonitor = RemoteMonitor(
```

Интересная вещь, по идее должна сильно помочь на кагле, где вроде бы нельзя напрямую следить за выполнением ноутбука в бэкграунде.

```
reduce_lr = ReduceLROnPlateau()
# (По итору его не использовал, т.к. использовал оптимизатор Adamax)
```

Как связано уменьшение learning rate с использованием оптимизатора Adamax? Вот алгоритм Adamax:

Algorithm 2: *AdaMax*, a variant of Adam based on the infinity norm. See section 7.1 for details. Good default settings for the tested machine learning problems are $\alpha = 0.002$, $\beta_1 = 0.9$ and $\beta_2 = 0.999$. With β_1^t we denote β_1 to the power t. Here, $(\alpha/(1-\beta_1^t))$ is the learning rate with the bias-correction term for the first moment. All operations on vectors are element-wise.

Require: α: Stepsize
Require: $\beta_1, \beta_2 \in [0, 1)$: Exponential decay rates
Require: $f(\theta)$: Stochastic objective function with parameters θ
Require: θ_0: Initial parameter vector
$m_0 \leftarrow 0$ (Initialize 1st moment vector)
$u_0 \leftarrow 0$ (Initialize the exponentially weighted infinity norm)
$t \leftarrow 0$ (Initialize timestep)
while θ_t not converged **do**
$t \leftarrow t + 1$
$g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$ (Get gradients w.r.t. stochastic objective at timestep t)
$m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$ (Update biased first moment estimate)
$u_t \leftarrow \max(\beta_2 \cdot u_{t-1}, |g_t|)$ (Update the exponentially weighted infinity norm)
$\theta_t \leftarrow \theta_{t-1} - (\alpha/(1-\beta_1^t)) \cdot m_t/u_t$ (Update parameters)
end while
return θ_t (Resulting parameters)

Параметр alpha – это learning rate, то есть поведение оптимизатора Adamax зависит от выбранного learning rate, как и для практически всех остальных оптимизаторов. Исключение составляет лишь Adadelta, авторы которого предлагают заменить фиксированный learning rate адаптивным, но даже в этом случае можно использовать параметр learning rate, как это сделано в Keras.

Поэтому непонятно чем обоснован отказ от затухающего learning rate.

```
for i in range(TTA_STEPS):
    print('TTA step: ', i)
    preds = model.predict(sub_generator, verbose=1)
    predictions.append(preds)
```

Можно было сделать TTA сначала на валидационном датасете и проверить, повышает ли он точность, то есть сравнить точность:

1. Предсказаний без TTA
2. Предсказаний на каждом шаге TTA
3. Усредненных предсказаний

Потому что TTA при неудачно выбранных параметрах может наоборот понижать точность.

- # Для второй модели придётся заново создать Sub-генератора, только уже со требуемым ею (пустым, в отличии от DenseNet121) пре-процессингом

А если встроить нормализацию как первый слой модели, то такой проблемы не возникнет:

keras.layers.Lambda(efficientnet.preprocess_input)

```
densenet121_val_accuracy = 0.9652
efficientnetb6_val_accuracy = 0.9691

pred3 = (densenet121_val_accuracy * pred +  efficientnetb6_val_accuracy *
pred2) / 2
```

По-моему странно брать точность в качестве весов. Например, одна модель имеет точность 99.5%, другая 98%. Очевидно первую надо брать с весом близким к единице, а вторую с весом близким к нулю. А если следовать вашему подходу, то получатся веса 0.995 и 0.98, что очень близко.

Почему выполняется деление на 2? Если мы считаем взвешенное среднее, то надо делить на сумму весов (0.9652 + 0.9691), а не на 2. Но вообще следом выполняется операция argmax, которой не важно на что вы поделите или умножите – результат будет тот же. Поэтому деление на 2 по-моему излишне.

- Для простых изображений (т.е. для которых все 10 аугментаций выдали один и тот же результат) - 10 аугментаций достаточно, если был хотя бы небольшой разнобой, то надо увеличивать кол-во аугментация, пока не наметиться явное преобладание како-то класса (100-200 раз).

Представим ситуацию, где наметилось явное преобладание класса, например класс 0 предсказан 70 раз, класс 1 предсказан 130 раз. Значит более вероятно, что изображение принадлежит классу номер 1. Но я бы не советовал делать на таком изображении псевдо-лейбелинг, потому что видно, что модель все равно сомневается в ответе. Чтобы быть уверенным в том, что псевдо-лейбелинг повысит, а не понизит точность, скорее стоит использовать только те изображения, где мы уверены, что модель выдала верный результат. А это те изображения, где один класс предсказывается почти во всех случаях.

- Аугментация должна убирать зависимость от фона и цвета, потому что модель их тоже будет учитывать и это плохо

Это с одной стороны логично, но с другой стороны на практике обесцвечивание, например, понижает точность. Так что вопрос остается спорным.

Есть кстати библиотека tensorflow-hub с новыми моделями, например

```
import tensorflow_hub as hub
model =
hub.KerasLayer("https://tfhub.dev/google/imagenet/efficientnet_v2_imagenet21k_b0/feature_vector/2")
```