Был предоставлен готовый датасет и baseline в формате Kaggle Kernel Notebook.

Это устаревший бейзлайн, много в библиотеках с тех пор поменялось в библиотеках, поэтому был сделан новый бейзлайн. Проблема только в том, что на него пока нет ссылки на платформе.

Далее шли эксперименты с самой легковесной моделью EfficientNetB0 и подбор гиперпараметров

Хорошая идея, мало кто так делает, обычно пытаются подбирать параметры сразу на тяжелой модели и большом размере изображения. Авторы этой работы пошли еще дальше: они подбирают архитектуру на очень легковесном датасете CIFAR-10, а затем используют ее для классификации ImageNet.

После подбора оптимальных настроек подгружена более совершенная модель EfficientNetB3

А из чего следует что она более совершенная? Единственное отличие этой модели в том, что она больше (по ширине и глубине) и обучалась на большем разрешении изображений. Но более глубокая сеть не означает более совершенная. Иначе сейчас использовались бы сети в тысячи слоев глубиной. Предполагается, что более глубокая сеть потенциально может достичь большей генерализации, однако для этого они требуют намного больше обучающих примеров. На небольшом количестве примеров большей генерализации могут достичь менее глубокие сети.

Saving kaggle.json to kaggle.json

Так любой, кто зайдет в этот репозиторий, получит ваш код доступа к Kaggle (key в Kaggle.json показан в выходе ячейки кода)

```
for data_zip in ['train.zip', 'test.zip']:
    with zipfile.ZipFile(DATA_PATH + data_zip, 'r') as z:
        z.extractall(WORK_PATH)
```

Это можно заменить командой !unzip {DATA_PATH}train.zip -d {WORK_PATH}

```
# Словарь с некорректными картинками в обучающей выборке - эти изображения удаляем из набора
```

Отлично что сделали очистку данных, обычно этот шаг пропускают.

Выполним аугментацию данных, что особенно полезно, если сэмплов не очень много.

Вообще аугментация, насколько я знаю, делается всегда, даже если данных очень много.

```
validation_generator = train_datagen.flow_from_directory(
```

У вас валидация делается с аугментациями. Так не должно быть. Представьте, что одну модель вы обучили на слабых аугментациях, другую на сильных. Если валидация делается с аугментациями, то модель с сильными аугментациями покажет более низкую точность на валидации. Если же делать валидацию нормально, без аугментаций, то все может быть наоборот. Поскольку валидация влияет на выбор моделей и гиперпараметров, важно делать ее без аугментаций.

Кроме того вы почему-то не делаете визуализацию изображений с аугментациями. Это полезно делать, так можно сразу выявить и исправить какие-то явные ошибки.

```
# Создадим callback для фиксации длительности каждой эпохи class TimingCallback(Callback):
```

Длительность эпохи и так печатается в консоль, если в fit использовать параметр verbose=1. Но я так понял, вы хотите сохранять длительность эпохи в логи.

```
# останавливать процесс обучения, если целевая
            # метрика не улучшается `patience` эпох подряд:
            EarlyStopping(monitor='val_accuracy',
                          patience=5,
                          restore_best_weights=True),
```

Остановка роста точности не всегда означает, что сеть больше не сможет обучиться. Может быть стоит подождать еще, увеличив параметр patience. Мне нравится следующая цитата: «leave it training. I've often seen people tempted to stop the model training when the validation loss seems to be leveling off. In my experience networks keep training for unintuitively long time. One time I accidentally left a model training during the winter break and when I got back in January it was SOTA ("state of the art").»

```
head = S([
    GlobalAveragePooling2D(),
    Dense(128, use_bias=False, kernel_regularizer='l2'),
    BatchNormalization(axis=1),
    Activation('relu'),
    Dropout(DROPOUT_RATE),
    Dense(10, activation='softmax')
])

# Собираем модель
model = model_assemble(base_model, head)
```

Можно просто подставить base_model первым слоем в Sequential, вместо использования функции model_assemble.

```
# при fine-tuning BatchNorm-слои нужно оставлять замороженными
        # https://keras.io/guides/transfer_learning/#build-a-model
        if not isinstance(layer, BatchNormalization):
            layer.trainable = True
```

Действительно, в гайде написано следующее:

When you unfreeze a model that contains BatchNormalization layers in order to do fine-tuning, you should keep the BatchNormalization layers in inference mode by passing training=False when calling the base model. Otherwise the updates applied to the non-trainable weights will suddenly destroy what the model has learned.

Но я этого не могу понять. Возможно речь идет о старой версии Keras, в которой при установке trainable=True размораживались все переменные слоя, в том числе moving_mean и moving_variance, которые размораживать не нужно никогда (ни при обучении, ни при инференсе). Но можно легко проверить, что в текущей версии Keras разморозка слоя BN работает корректно:

```
from tensorflow.keras.layers import BatchNormalization
layer = BatchNormalization()
layer.build(input_shape=100)
print('Trainble:', [v.name for v in layer.trainable_weights])
print('Non trainble:', [v.name for v in layer.non_trainable_weights])

layer.trainable=False
```

```
print('FREEZING')
print('Trainble:', [v.name for v in layer.trainable_weights])
print('Non trainble:', [v.name for v in layer.non_trainable_weights])

layer.trainable=True
print('UNFREEZING')
print('Trainble:', [v.name for v in layer.trainable_weights])
print('Non trainble:', [v.name for v in layer.non_trainable_weights])
```

Результат:

```
Trainble: ['gamma:0', 'beta:0']
Non trainble: ['moving_mean:0', 'moving_variance:0']
FREEZING
Trainble: []
Non trainble: ['gamma:0', 'beta:0', 'moving_mean:0', 'moving_variance:0']
UNFREEZING
Trainble: ['gamma:0', 'beta:0']
Non trainble: ['moving_mean:0', 'moving_variance:0']
```

Как видим, слой размораживается корректно. На практике разморозка BN не приводит ни к каким проблемам, поэтому непонятно зачем оставлять слои BN замороженными. Наоборот, эти слои размораживать важнее всего, потому что они отвечают за статистики по входным данным, которые сильно меняются при адаптации сети под новую задачу. Можно почитать подробнее в многочисленных статьях на эту тему, например здесь. Фактически можно размораживать только BN слои и получать почти аналогичное качество файн-тюнинга.

Берем более эффективную SOTA-модель EfficientNetB3 в качестве базовой модели

Непонятно из чего следует, что EfficientNet является SOTA моделью. Есть более новые и эффективные модели, например эта модель, либо другие модели 2020-2021 годов.

```
for i in range(tta_steps):
    preds = model.predict(test_subgenerator, verbose=1)
    predictions.append(preds)
```

Тут можно еще делать так: сделать сначала 10 попыток на каждом изображении, а затем делать дополнительные попытки на тех изображениях, в которых результат остался спорным.

Основное тестирование для подбора гиперпараметров было решено проводить на сети EfficientNetB0. Испытывались предобученные сети: EfficientNetE0, EfficientNetE3, EfficientNetE4, EfficientNetE6, EfficientNetE7;

Но увы ни кода, ни результатов этого тестирования (в виде таблицы сравнения) в репозитории нигде нет.

Заметный прирост точности показало добавление в архитектуру "головы" BatchNormalization

Опять-таки нет таблицы сравнения, где бы ясно описывались архитектуры, процесс обучения и достигнутые результаты. Это необходимо, чтобы убедиться, что эксперименты поставлены корректно. Например, сравнение сетей с головами Dense-BN-Dense и Dense-Dense не будет показательным, даже если BN в данном случае увеличивает точность. Поскольку возможно, что если убрать скрытый полносвязный слой и BN, то результат будет еще лучше.

И самое главное: у вас в обучении в ноутбуке используется постепенная разморозка слоев. Вы пробовали обучать всю сеть целиком изначально (не замораживая) и сравнивать результаты?

В целом проделана очень хорошая работа. Было бы здорово, если бы все работы были такими.