**Development, fails and team work organization**

We decided to follow an incremental approach: start with simple models, increase their complexity to reach overfitting and then apply regularization techniques to improve the validation accuracy. Starting from handcrafted networks, we tried to maximize the training accuracy in order to understand the generalization capabilities of our simple networks. The best result we achieved was around 84-85% validation accuracy over the local split, reaching overfitting of the training set. This seemed to be an upperbound for every network we tried.

After many trials, we decided to move to transfer learning. The first supernets we tried were InceptionV3 and Xception. Surprisingly, we were getting the same result of our best handcrafted model. It was clear there were some problems in the augmentation we used until that moment. Indeed, all the trainings were performed over an offline augmented training set of approximately 12k images. It became obvious that our networks were not able to reach good generalization because the augmented images were exactly the same at each new epoch. So, once we moved to transfer learning with online augmentation, we finally got, with good consistency, 88-90% validation accuracy. From that moment on, our strategy consisted in separately building models using different supernets and augmentation techniques. The development of each model consisted in the following steps:

1. Fine tune the entire supernet using a Global Average Pooling layer directly connected to the softmax dense layer
2. Train a final classifier network that includes the supernet trained at the previous step and append some dense layers. To do so,we loaded the supernet weights trained at step 1 and freezed all the supernet layers.

Using this approach, we understood that the supernets are very good in generalization, reaching 89-91% validation accuracy even without any dense layer. Hence, the extracted features were very good. Training the final classifier model using our pretrained supernet allowed us in most the cases to get an extra 2-3% over the validation data.

Once we got a satisfying amount of good performing models (tested on the test set during the 1st phase of the challenge), we decided to go for an ensemble method to improve the generalization capabilities of our models. In order to get the maximum performance out of the ensemble, we decided to include in it models with a balanced amount of types of supernets. The final ensemble includes 2 Xception, 2 EfficientNetB2 and 2 InceptionResNetV2 models and the prediction is performed by summing up the class odds.

We are strongly convinced that this approach helped us in improving the test accuracy for the following reasons:

* different supernets learn different patterns
* training models with different augmentation stratgeies helps in capturing the most of the variance
* we independently trained models using different train-validation splits, similarly to what happens in bootstrapping in the Bagging technique

**Kaggle**

We mainly used Kaggle for the most computationally intensive tests. We improved the execution speed by using mixed precision (floats with 16 bits) and Nvidia SLI parallel execution on two gpus (2x Tesla T4 provided by the cloud server). The parallel execution showed faster execution because the batch size is split between the two GPUs, which eventually unify the calculations and apply gradient descent to compute the new weights. The mixed precision calculations are theoretically useful for limiting the overfitting and work as a regularization mechanism for the floating point computations that Tensorflow does.

**Image processing**

The images are pre-processed according to the function needed by the specific supernet. We also tried applying standardization to the images with respect to the mean and standard deviation of the *offline* and *non augmented* training set, learnt by fitting the image data generators (both training and validation) to the training split only. It didn’t show any improvements with respect to the non-standardized inputs, and this was part of the reason we didn’t eventually stick to the images standardization. We suspect this being caused by the fact that the online augmentation changes the mean and standard deviation of the online generated training samples, thus offline computed parameters are no more representative of the new distribution. Other types of pre-processing we tried are adjusting saturation and contrast. This has been performed by defining a custom preprocess input fuction (to be passed to the image data generators) that slightly increases those two properties, in addition to the specific supernet preprocessing function. The initial models we uploaded exploited this technique with good results, by applying the same preprocess at test time. Later on, we dropped this approach since we discovered that caused several cpu bottleneck during the training. One solution, which we decided to not implement, could have been inserting some preprocessing layer inside the network architecture.

2 methods were used for trasversing the images dataset during training, starting from the image data generator class. The first method is the flow method, which used images matrices stored in memory, which was convenient and fast to use, but created a memory bottleneck for resized images. The second method (the one used in the best models) is the flow from directory, which didn’t bottleneck the memory, and accessed directly the images in the directories. This method proved to be also useful for creating a quick switch between the balanced dataset and the oversampled dataset creation.

**Regularization methods**

We used many regularization methods, including dropout layers (and its variation gaussian dropout), L2 regularization in the loss function, global average and max pooling layers. The most convenient and practical methods that we employed in all of our tests are the dropout layer and the global average pooling, which showed significant improvements in the validation accuracy, while preventing too much overfitting. The gaussian alternative of the dropout layer proved to provide more regularization than a standard dropout layer, so its rate parameter was tuned in a few attempts in order not to reduce the training accuracy too much. L2 regularization did not improved the learning using the default lambda parameter. However, it seemed to reduce the overfitting using a lambda in the range (0.001;0.01). This was also true for the dropout, which appeared to be beneficial with an uniform rate among the layers in the range of (0.45;0.55). For all our models, we decided to use a GAP instead of flattening layers, why helped in reducing drastically the number of parameters, allowing faster trainings and better generalization capabilities.

**Supernets**

We tried multiple convolutional neural networks pre-trained on the imagenet dataset. We experimented with Xception, InceptionResNetV2, InceptionV3, EfficientNetB5 and B2.

**Learning rate**

We initially failed to apply transfer learning to our model because the learning rate was the default one (at 0.001) and was way too high for fine-tuning the pretrained supernets. Indeed, it caused a very fast overfitting of the training, but ending up with a very low validation accuracy. We managed to tweak the learning rate correctly by playing with a learning rate scheduler. We created an exponential decay, with a low initial value (about 10^-4). This way it starts learning fast, and it gradually decreases the learning rate so that by the end of the training it is very low, to achieve as many improvements as possible. The exponential decay technique proved to be successful in the training of the supernets and especially for the fine-tuning, where more precise control of the learning rate is needed to maximize the validation accuracy. Another technique used for adjusting the learning rate manually consists in the following steps:

1. Train with a fixed initial learning rate
2. Take note of the epoch in which the validation accuracy improvement stalls
3. Define a scheduler that step decreases the learning rate at that epoch
4. Restart from step number 2

We did not try others optimizers since ADAM already combines both Momentum and RMSProp heuristics.

**Experimental approaches**

**Quasi SVM**

We tried implementing a quasi-SVM approach, by using a modified dense layer. This behaves like a random Fourier feature extractor with a gaussian kernel initializer. This approach proved to increase regularization capability without showing any significant improvement concerning the usual dense layer.

**CutMix augmentation**

CutMix is a new augmentation method that promises to improve the accuracy of the model, by generating images that are a mix of 2 different images of different classes. This library aims to create a random bounding box where to place a different image. The goal is to train a model which can learn to classify multiple classes at the same time with a single image. The labels are modified accordingly, by putting a higher weight on the label corresponding to the class of the image that occupies most of the area. The loss function of the model is still the categorical cross-entropy, but now the encoding of the label is not one-hot but it’s a continuous function.

We managed to successfully use this library, present in the Keras-CV module, but its application didn’t provide our model the learning capability. The model was not able to generalize, and its validation accuracy is equivalent to the one of a random classifier. This is way we discarded this approach for a better augmentation of the training dataset.

**Keras tuner**

Draft report anndl

Outline of the topics to be covered (highlighted = already done)

1. Baseline model composition and accuracy
2. Regularization with dropout, gaussian dropout and l2 regularization in output dense layer
3. Online vs offline augmentation
4. Transfer learning with xception and the other supernets
5. Technique for fine tuning
6. Oversampling technique for under-represented classes and class\_weights approach
7. Exponential decay and adaptive learning rate schedulers
8. Flow and flow from directory
9. Standardization of images
10. Kaggle execution improvement with mixed precision and 2x gpu execution
11. Final models and some parameters used in the best performing model, image resizing
12. Cutmix augmentation not learning
13. Quasi svm dense layer (fourier transform with gaussian kernel)
14. Ensemble method of the best models and different supernets
15. Failure of the learning rate too high

TODO: the other points; I will do document formatting and proof reading