Artificial Neural Networks and Deep Learning, Competition 1

Tommaso Capacci, Simone Giampà, Gabriele Ginestroni

Development, failures, and teamwork 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 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 upper bound 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 as our best-handcrafted model. It was clear there were some problems: after some attempts, we realized that we were using the default ADAM learning rate which eventually worked well for the handcrafted model (since its weights were randomly initialized through initializers) but not for the supernets, whose weights only needed to be fine-tuned with a lower learning rate.

Moreover, 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 also 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 has been separately building models using different supernets and augmentation techniques. The development of each model consisted of the following steps:

1) Fine-tune the entire supernet using a **Global Average Pooling** layer 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 froze all its 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 pre-trained supernet allowed us in most cases to get an extra 2-3% over the validation data.

Once we got a satisfying amount of well-performing models (tested on the test set during the first phase of the challenge), we decided to go for an **ensemble** method to improve the generalization capabilities of our models. To get the maximum performance out of the ensemble, we decided to include in it models with an almost balanced mix of supernets of the same type (to avoid giving too much "voting" power to a specific architecture). The final ensemble includes 2 Xception, 2 EfficientNetB2, and 1 InceptionResNetV2 model, and the prediction is performed by maximizing the sum of the class probabilities.

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 strategies helps in capturing most of the variance
- we independently trained models using different train-validation splits, similar to what happens in bootstrapping in the Bagging technique

Model selection

For our models, we decided not to use cross-validation for model selection. This choice has been taken because training would have taken too much time. To overcome this, we used our test accuracy of the "development phase" of the challenge to select our best-performing models.

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. For our models, 32-64 x 2 turned out to be a good choice for the **batch size**. Higher values for batch size seemed to cause higher overfitting in our tests. Mixed precision calculations are theoretically useful for limiting overfitting and work as a regularization mechanism for floating-point computations.

Image pre-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, learned 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 image standardization. We suspect this is 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 longer 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 pre-process input function (to be passed to the image data generators) that slightly altered those two properties, in addition to the specific supernet pre-processing function. The initial models we uploaded exploited this technique with good results, by applying the same pre-process at test time. Later, we dropped this approach since we discovered that caused several CPU bottlenecks during the training. One solution, which we decided not to implement, could have been inserting some pre-processing layer inside the network architecture.

Another technique that turned out to be very useful has been the **resizing** of the images implemented using a resizing layer. Most of our models showed improved performances whenever a resizing in the rage 192-299 (preserving the original aspect ratio) with bicubic interpolation was applied.

Augmentation and imbalanced classes handling

To visualize the effects of augmentation we created an additional notebook that allowed us to plot some examples of the augmented images. In this way, we were able to understand which were plausible intervals for the parameters of the augmentation. To enhance our independent training framework, we tried to diversify these parameters for the models included in the ensemble to reach a greater generalization capability.

For what concerns the imbalanced dataset we tried different approaches: for all models, we used scikit-learn **class-weights** during the training phase to increase the importance of samples from the underrepresented classes; additionally, for some models, we also chose to **oversample** the minority classes in such a way that, in the

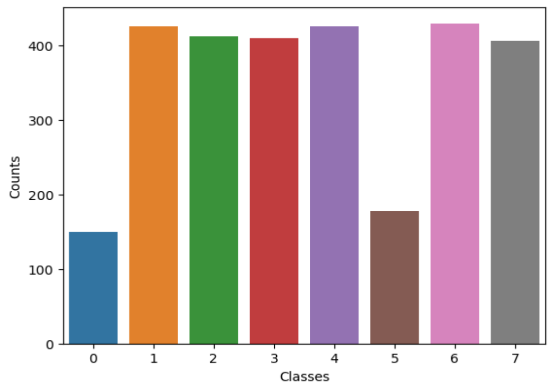

Figure 1: original imbalanced training split

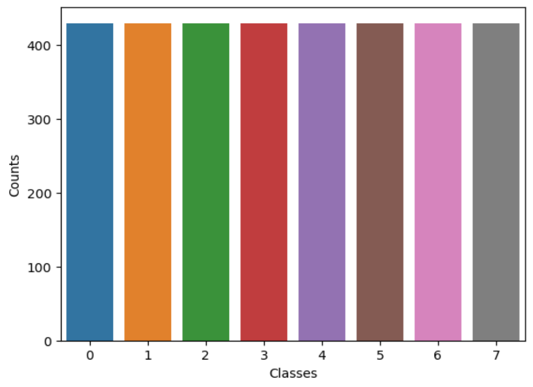

Figure 2: balanced training split

end, all the classes had the same number of samples. Note that the 80-20% train split has been performed before this step, as we wanted the validation to be a real representation of the original dataset distribution. For the same reason, we decided to build the validation split in a stratified way. The oversampling strategy has been implemented by duplicating, without augmentation, the images from each class (trying to replicate each sample from a specific class the same number of times). The augmentation is finally computed online from the training generator, which loads the images from the rebalanced training split folder.

Regularization methods

We used many regularization methods, including **dropout** layers (and their 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 our tests are the dropout layer and the global average pooling one, which showed significant improvements in the validation accuracy while preventing too high overfitting. The gaussian alternative of the dropout layer is known 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 improve 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 a 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 which 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 EfficientNetB2.

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 pre-trained supernets. Indeed, it caused a very fast overfitting of the training but ended 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 of the following steps: train with a fixed initial learning rate, take note of the epoch in which the validation accuracy improvement stalls, define a scheduler that step decreases the learning rate at that epoch and restart from the second step.

We did not try other 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. The Keras SVM implementation turned out to be quite handy since it allows to train end-to-end of the SVM classifier together with the whole network. This approach proved to increase a little bit the generalization capability without showing any significant improvement with respect to the usual dense layer.

CutMix augmentation

We also tested this library, present in the Keras-CV module. However, we didn't get good results. Probably this happened because we performed the training using only the cutmixed images, without including the original samples. Therefore, we decided to stay with the augmentation we were previously using.

Keras tuner

We gave a try to the Keras Hyperband tuner to find a good configuration for the dense layers. In particular, it has been applied to the *lambda* parameter of the L2 regularizers, *dropout rate,* and dense *units* hyperparameters. However, due to time constraints, we just implemented it without fully testing it. The partial results we got seemed to be reasonable and were used as inspiration for our final models.