**Artificial Neural Networks and Deep Learning, Project 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 results** we achieved were 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 training was 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 the same in every 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 of 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 number of the same type of supernets. The **final ensemble** includes 2 Xception, 2 EfficientNetB2, and 1 InceptionResNetV2 model, and the prediction is performed by summing up 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, like 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 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, therefore 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-processing 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 to not implement because of the very long training time required at each epoch iteration, was the insertion of some pre-processing and data augmentation layers inside the network architecture.

**Augmentation and unbalanced 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.

Chart, bar chart

Description automatically generatedChart, bar chart

Description automatically generatedFor what concerns the unbalanced dataset we tried different approaches: for all models, we used scikit-learn class weights during the training phase; additionally, for some models, we also chose to oversample the minority classes in such a way that, in the end, all the classes had the same number of samples.

**Regularization methods**

We used many regularization methods, including dropout layers (and their variation gaussian dropout), L2 regularization in the loss function, and 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 much 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 training 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:

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 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. 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. We understood that this method requires a training set composed of images processed with cutmix and normal images as well. It eventually didn’t turn useful for our dataset since we didn’t see any improvement in the new models with respect to the non-cutmixed augmentation of the training set in the previous models. This is why we discarded this approach for a better augmentation without using cutmix.

**Keras tuner**