Report Homework 1 - Augmented Tensorflop

Daniele Civati, Matteo Palazzoli, Francesco Panebianco | Prof. Matteucci

Work Process & Data Handling

We started out separately, working on our own model, then halfway through phase one we selected the best performing and tuned it in different ways. The data loading pipeline was also something we all did differently to see what worked best. Initially, we loaded the images as numpy arrays, iterating over the directory. This method lets us split the dataset into three parts (training, validation, test) as opposed to ImageDataGenerator that offers only one single split into training and validation. We tried some basic splits like 80-10-10 and 70-15-15 but in the end we ended up making 77-8-15. To perform the augmentation we also tried to use an `ImageDataGenerator` on the splits, calling the `flow()` method, but in the end we decided to use the `tf.data.Dataset` utility calling the method `from_tensor_slides()`[1], because of some constraints that are explained below.

Model From Scratch

The first model we tried was a simple CNN from scratch, made of 5 convolutional layers separated by 2x2 MaxPooling layers and a 2-layer dense block with Dropout of 0.3. This model had no data augmentation, no pretrained parts, *ReLU* activation for both convolutional and dense layers, filters with a kernel_size = 3, 'valid' padding, resulting in a receptive field of 94 (over 96x96 images). It was trained with *Adam* (learning rate 1e-3), returning an accuracy of 0.5853 on the local test set.

The first change we made was increasing the size of the convolutional filters to 5x5, which of course required the padding to be switched to 'same'. This was not particularly useful, since the test accuracy dropped to 0.51. The third model went back to 3x3 filters, but had the activation function changed to the *Swish*, which is said to improve learning and was left in almost all subsequent models from scratch. As for the optimizer, it was at this point that we started experimenting with something different from *Adam*. In fact, we tested *Adadelta* (learning rate 1e-3), then *RMSProps* and *Adagrad*, which however resulted in weak convergence compared to *Adam*, giving out around 0.55 of accuracy after a long number of epochs. The next model had Dropout replaced with *Weight Decay*: specifically, the dense layer was regularized with *Elastic Net* (l1=1e-5, l2=1e-4) for the weights and Ridge (1e-4) for the bias, yielding similar performance.

Basic Data Augmentation

We experimented with various *Data Augmentation* techniques. Initially, we took the Scratch Model that loaded the data in numpy arrays. We instantiated three ImageDataGenerator objects, one for each section of the data (training, validation, test), in which we could give the augmentation parameters for the training split. Then we created our data augmentation pipeline with the method *flow()*. We tried some basic operations like random flipping, rotating, zooming and shifting on the training set. Flipping was made both horizontally and vertically; the rotation range was about 0.1, the zoom had a range of about ±0.2 and the shift was about 25 pixels. Still, we knew things could be improved, so we decided to try removing pieces like the random rotation and zoom. We did that and our performance grew by about 0.3 on the validation data. We also tried a mixed approach in

[1] https://www.tensorflow.org/api_docs/python/tf/data/Dataset#from_tensor_slices

which we decrease the effect of rotation and zoom but we concluded that it was better to just remove them from the Generator. In some of the models, techniques were implemented as Keras layers (RandomFlip, RandomTranslation), that operate only during the training phase.[2]

Best Model From Scratch

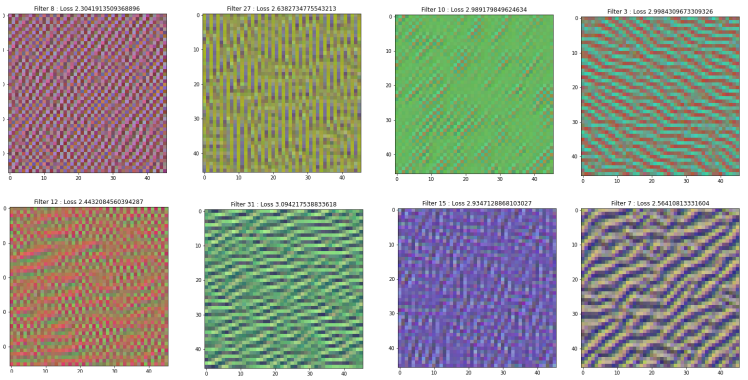
After the first experiments in *Data Augmentation* the basic model from scratch reached a performance of 0.6107, which was quickly increased by tuning the augmentation parameters, reaching 0.6889. It was at that point that we went back to changing the structure of the network and the number of filters, iteratively tuning it until we built our best model from scratch: *NomadEve*.

NomadEve has the structure that can be seen in the figure on the right. The Dense layer is replaced with a *GlobalAveragePooling* layer which also contributed to improving performance. The activation function is *Swish* for all layers and no regularization is performed.

Layer (type)	Output Shape	Param #
conv2d_59 (Conv2D)	(None, 96, 96, 32)	896
max_pooling2d_50 (MaxPooling	(None, 48, 48, 32)	0
conv2d_60 (Conv2D)	(None, 48, 48, 64)	18496
max_pooling2d_51 (MaxPooling	(None, 24, 24, 64)	0
conv2d_61 (Conv2D)	(None, 24, 24, 128)	73856
max_pooling2d_52 (MaxPooling	(None, 12, 12, 128)	0
conv2d_62 (Conv2D)	(None, 12, 12, 256)	295168
max_pooling2d_53 (MaxPooling	(None, 6, 6, 256)	0
conv2d_63 (Conv2D)	(None, 6, 6, 512)	1180160
conv2d_64 (Conv2D)	(None, 6, 6, 512)	2359808
max_pooling2d_54 (MaxPooling	(None, 3, 3, 512)	0
global_average_pooling2d_5 (	(None, 512)	0
output_layer (Dense)	(None, 8)	4104

Total params: 3,932,488
Trainable params: 3,932,488
Non-trainable params: 0

It's called *NomadEve* because the optimizer used was not *Adam*, instead it was changed to *Nadam*, which was capable of escaping local minima more easily by exploring more, moving through the hypothesis space like a nomad and finally giving 0.7630 accuracy in the local test set after a few re-runs, which resulted in around 0.72 on the hidden test set in *Codalab*. We wanted at this point to visualize the learned filters on the uppermost layer, which were derived by 30 steps of gradient ascent[3]. The figure on the right shows 8 of the most interesting ones.



Transfer Learning

We have tried different pretrained models to see the performances we could achieve. In all the models we used two dense layers and two dropout layers with rate 0.3. All trials had similar dense blocks, all having *ReLU* as activation function and only varying the number of neurons depending on the model used. The first model that we tried was *VGG16* without data augmentation and we observed, in local, an accuracy of 70%, then we tried with data augmentation and observed a worse accuracy, in fact it had dropped to 47%. At this point we decided to test *EfficientNet B5*, *VGG19*, Xception and *InceptionV3*. *VGG19* and *EfficientNet B5* were similar with approximately 65-66% of accuracy while *Inception V3* and *Xception* about 51-56%. It was now time to unfreeze some pretrained weights.

[2] https://www.tensorflow.org/api_docs/python/tf/keras/layers/RandomFlip
[3] https://keras.io/examples/vision/visualizing_what_convnets_learn/

Fine Tuning

Since *VGG-19* and *VGG-16* were the best performing on transfer learning, we decided to mainly focus on these two to get started on fine tuning. It is important to note that until this point the loss function was always *Categorical Crossentropy*, and from here on we started experimenting with *KLDivergence* and especially *Sigmoid Focal Crossentropy*[4] to handle the class imbalance, which we kept for a while since it seemed to allow us to gain a 5% accuracy on the local test set. Another model which performed rather well locally but very poorly on the hidden test set was *Xception*: with a local accuracy of 0.9069 and even an F1 score of 0.8997. Many models were tested, and for each of them many configurations of frozen layers and dense / GlobalAveragePooling. In the end though, the model which was more consistent in its result was *FocalLeaf*, which was also perfected by introducing proper class weighting, based on the scarcity of data points for different classes; it used *Sigmoid Focal CrossEntropy* and it was trained using the *Yogi* optimizer[5] (learning rate 1e-4), which really made a difference in training all subsequent versions of the fine tuning effort. The penultimate milestone was *EntropicLeafV3*, which went back to the *Categorical Crossentropy* loss function but had better class weighting, resulting in a performance on the hidden test set of 0.8595.

Advanced Data Augmentation

We kept the basic augmentation for a while, until our tutors suggested *CutOut* and *CutMix*. These utilities were included in *KerasCV,* which unfortunately was not supported in our environment (Kaggle Code, with Tensorflow 2.6.4). So initially we tried to implement *CutOut* from scratch, replacing a random square window with gray pixels or gaussian noise in a fraction of training images. It didn't change much, likely because we didn't entirely adhere to the paper's suggestions (e.g. we used uniform distribution instead of beta). Discarding *CutOut*, we then implemented *MixUp*[6] and *CutMix*[7] exactly following the guides, migrating our dataset to a `tf.data.Dataset` for better compatibility with the code. Each utility halves the dataset, so it has to be duplicated before applying anyone of the two. Then, we tried running the model with both enabled, and obtained a validation accuracy of 91.20%, and F1 = 79.73% on our test set. We removed the *MixUp* function and experimented with *CutMix* only, which gave us better results.

Final Model

Finally, the last milestone was created, by freezing less pretrained layers, changing the augmentation pipeline with the inclusion of *CutMix*, more tuning on class weights and less dropout regularization. This model was later called *BlendedLeaf*, and was our best performing model on the hidden test set, giving an accuracy of 0.8753 in phase 1 and 0.8647 in phase 2.

Further tests on improving this model were using ensemble techniques, namely bagging, which however did not surpass its performance.

---

[4] https://www.tensorflow.org/addons/api_docs/python/tfa/losses/SigmoidFocalCrossEntropy
[5] https://www.tensorflow.org/addons/api_docs/python/tfa/optimizers/Yogi
[6] https://keras.io/examples/vision/mixup/
[7] https://keras.io/examples/vision/cutmix/