ANNDL Homework 1 Report - De Novellis De Luca Pettorruso (Team AnotherDeep)

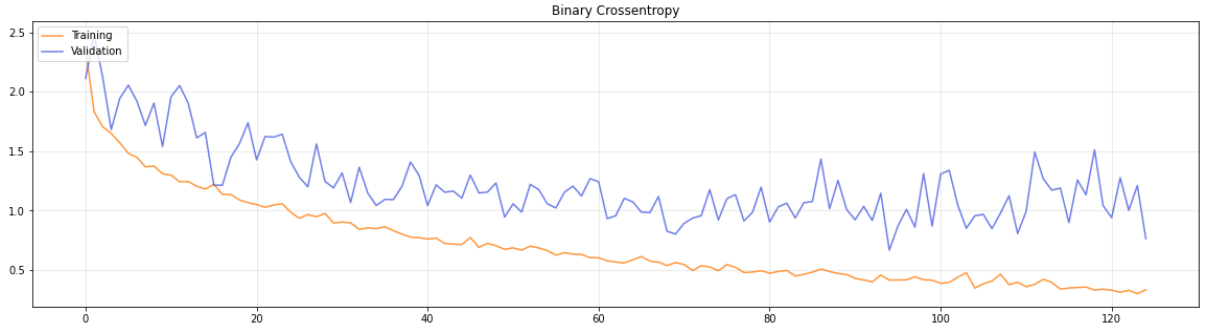
Introduction

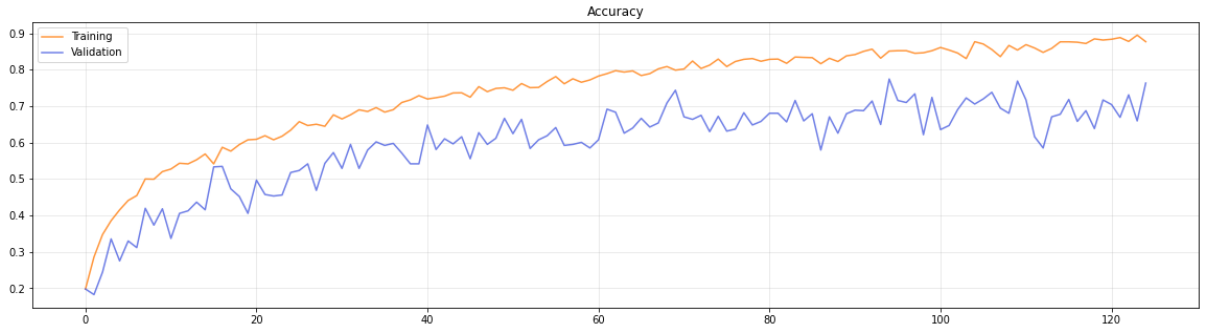
The goal of the challenge was to implement and train in Python a convolutional neural network and use it to classify images of plants to the correct species in a set of 8 possible classes.
To implement the network we worked using Python Notebooks on the Google Colab environment, using the free gpu available for the training part.
The first task was to split the dataset given into training and validation, and to do this we used the split-folders library and tried different combinations during the course of the challenge ranging from 80%-20% train-validation to 90%-10%.

Custom Model (Scratch)

Our first approach was to build the convolutional neural network with the structure shown in the laboratory session, to get familiar with Google Colab and with the challenge's platform. The first structure for our network was the one shown in the laboratory session, that consisted in a sequence of filter layers of size 32-64-128-256-512 interleaved by max pooling layers for the feature extraction part, and a flattening layer, a single dense layer of 128 neurons and the output layer for the classifier part. With it we reached an accuracy of about 43% on the test set of the platform, which we managed to increase by about 6% by increasing the batch size from 40 to 90. In addition, we performed augmentation on the training data by applying rotations, height and width shifts, zoom and horizontal and vertical flips.
We tried tuning hyperparameters such as the rates of the augmentation transformations or different values for the batch size but couldn't improve the accuracy of the model, so we decided to experiment with the structure of the network, both in the feature extractor part and in the classifier.
First of all we introduced two dropout layers, before the dense and the output layers of the classifier because, as seen in the lecture, it could decrease the overfitting on the data given to us. We tried values ranging from 0.1 to 0.5, and also to give different dropout rates to the two layers, but we observed that the best performances were with 0.2 in both layers.
Regarding the convolutional layer we thought that, since the accuracy and loss results were bad and the model would stop learning after few epochs, the model was too simple for the task given, therefore we increased the number of convolutional layers and the number of filters, and we built an architecture with 64-128-256-512-1024 as the size of the filter layers and left the rest unchanged.
Those two changes greatly improved the performance of our model, and we increased the accuracy to around 70% on our validation test, a result that remained similar on the test set of the codalab platform, achieving around 68%.
Another improvement we did with the model built from scratch was changing the flatten layer with a global average pooling 2D layer, because we found out that this layer improves the representation of the vector, while the flattening layer just re-arranges the elements. This substitution was also followed by a change of the activation function of the convolutional layer from 'relu' to 'leaky relu' with a parameter alpha equal to 0.01. These two changes allowed us to increase the score on validation set up to 78%, which resulted in a final score of around 76% on the test.

After this we made a few other trials such as changing again the structure of the convolution or introducing regularization but nothing improved the model, so we switched to transfer learning to try to get better results.





Transfer learning

Again, we first started the transfer learning phase by trying the model shown in the laboratory sessions, which consisted of the vgg16 supernet as the feature extractor, and the same classifier we used for the previous network. We trained it once with the whole feature extractor frozen and then fine tuned it by unfreezing everything but the first 14 layers, and with this model we got an accuracy of around 85% on the validation set and 83% on the Codalab platform.

As we saw that the training was a bit slower than the one of the model made from scratch and considering also that the original classifier structure of the vgg16 network was simpler than ours, we decided to change our classifier and make it very simple with a dense layer of 4096 units and the output layer. This let us reach 86% accuracy on Codalab platform, and we noticed that adding other layers, dense or dropout, or adding or removing neurons from the already present ones did not increase the accuracy.

This result was already better than what we had achieved with the network from scratch, but looking around on the keras documentation we found that there were some more recent network structures that had better performances on both top-1 and top-5 accuracy on the imagenet dataset, so we decided to first try many supernets by fine tuning them with the same percentage of layers (50%) frozen once, and then start tuning and experimenting more with the most “promising” one found.

With this reasoning in mind, beside the vgg16, we tried vgg19, MobileNet, EfficientNet and ConvNext and found that the Large version of the latter was the best one, reaching around 90% accuracy on our validation set, while the EfficientNet was the one reaching the best accuracy - 88% on test set - compared to its very fast training time (around 10s per epoch). We started making more experiments with the ConvNextLarge supernet to improve again the result and the considerations that improved the performances and lead us to the final model were the following:

1) By looking into tutorials and documentation, we found that 224x224 was the “suggested” input shape for the net, and that by passing it the original shape (96x96) the supernet was basically too “deep” for the size of the image and downsampled it too much, so we added a resizing layer to our model before the supernet layer that resize the image to 224x224.

2) Another thing we saw was that our model was noticeably weaker on the first and the sixth class, because the given dataset was imbalanced and had less images from those classes, so our model was “learning” to be stronger in the most represented classes, basically modeling the bias of the dataset. To counter this we introduced an algorithm to calculate the size of each class in

the dataset and to give a weight to each class proportional to how unrepresented it was. So in the end we had an array with a higher weight for the first and sixth class and therefore, during training, the loss in those classes had a higher value.

3) We made different trials to adjust the number of layers to keep frozen during the fine tuning, and found that 25% was the percentage of layers that lead to the best performances, so we concluded that this was the part of layers responsible for the computing of the “high level” features and that the training on imagenet already found the best weights for this part.

4) We tried to change the rate of training and validation from the 80%-20% that we used until that time and found that with 90%-10% the result on the Codalab test set was pretty inaccurate (losing around 4-5% from the validation results) but that with a split of 85%-15% the results were basically as accurate as the 80%-20% split so we used that one for the final model’s training.

5) We realized that for the fine tuning, since the weights are already initialized to the ones trained with the imagenet dataset, we just needed to slightly tune them and not to distort them, so we reduced the learning rate to 1/10 of the default one used to train the classifier and thus set it to 1e-4. Also, we noticed that both the training of the classifier and the fine tuning converged very fast, so to avoid overfitting on the training data and to reduce the utilization of the Colab GPU we set a low value for the patience (10).

Other changes in the classifier part such as introducing the dropout as with the model from scratch, changing the size or adding more layers didn’t lead to any improvement, so we just kept the simple classifier, applied the changes listed above and at the end obtained an accuracy of 0.9242 on the Codalab platform in the final phase.

Final Model summary and Final Considerations

At the end, the structure of our final model was this:

```
def build_model(input_shape):

    # Build the neural network layer by layer
    input_layer = tfkl.Input(shape=input_shape, name='input_layer')

    #resizing of each image from (96,96,3) to (224,224,3)
    x = tfkl.Resizing(224, 224, interpolation="bicubic", name='resizing')(input_layer)

    x = supernet(x)

    #use of a classifier layer composed of 4096 units
    classifier_layer = tfkl.Dense(units=4096, name='Classifier', kernel_initializer=tfk.initializers.HeUniform(seed), activation='relu')(x)

    #output layer classifies the 8 species
    output_layer = tfkl.Dense(units=8, activation='softmax', kernel_initializer=tfk.initializers.GlorotUniform(seed), name='output_layer')(classifier_layer)

    # Connect input and output through the Model class
    model = tfk.Model(inputs=input_layer, outputs=output_layer, name='model')

    # Compile the model
    model.compile(loss=tfk.losses.CategoricalCrossentropy(), optimizer=tfk.optimizers.Adam(), metrics='accuracy')

    # Return the model
    return model
```

The results on the Codalab platform were pretty satisfactory, although we had a drop of around 1% of accuracy between the first phase and the second one, which meant that maybe we were slightly overfitting with regards to the Codalab set of the first phase.