POLITECNICO
MILANO 1863

Report of Homework 2 - Artificial Neural Networks and Deep Learning 2020/2021

Tommaso Fontana Daniele Comi

20/12/2020

1 Introduction

In this competition, ACRE organizers ask you to segment RGB images to distinguish between crop, weeds, and background. ACRE is the Agri-food Competition for Robot Evaluation, part of the METRICS project funded by the European Union's Horizon 2020 research and innovation program under grant agreement No 871252. The image segmentation model will have as training input and target the following cases illustrated in Figure 1.

Input image

Target segmented mask

Figure 1: Example of the segmentation problem the model has to learn.

2 Data set handling

The data set we chose to focus on is the Bipbip Maize data set. It is constituted of 90 images and the corresponding 90 target masks. Unfortunately the classes to be segmented are highly unbalanced due to the most frequent presence of the Background class.
The data set was augmented, example in the below Figure 2, through default pre-processing functions dependent on the model to be used and they were:

- tensorflow.keras.applications.densenet.preprocess_input
- tensorflow.keras.applications.resnet.preprocess_input

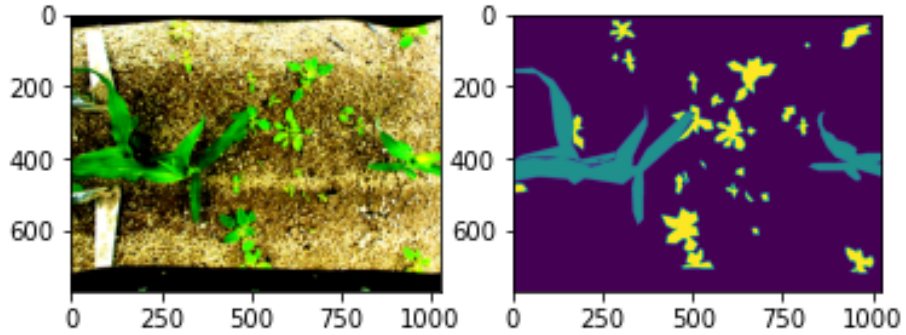

Figure 2: Example of processed training data: augmented resized input and resized target.

While the RGB images were of an original dimension of about 2048x1536, we choose to handle 1024x768 RGB images so that we are able to loose the minimum of information from the original, double-sized, input images.

3 Model Manual Tuning

The data set has been then further divided for validation purposes: 80% of the images were for training (72 out of 90) and the remaining 18 were for validation.

Having better hardware and time we could have used multiple holdouts, K-fold, cross-validation or Monte-Carlo Cross-validation to have a statistically significant evaluation of the performance of the model. Furthermore, having more resources we could adopt Hyper-parameters optimization strategies such as Grid-search and Bayesian Optimization which could improve the performance of the model but requires massive computational power.

Instead of doing these procedures we actually tried different kinds of models and we actually choose to focus our attention on the highly unbalance of the classes to be segmented, it's an important problem. To solve this we began at first to sample the entire data set by simply counting the corresponding encounters of classes pixel by pixel and writing this count in a dictionary composed of three keys, one per class: 0,1 and 2. After having done so we exploit this precise sampling to compute the weights to assign to each class as function of their frequency. The code is shown down here in Figure 3.

```
def compute_segmentation_masks_count(img_path):
        mask_img = Image.open(img_path)
        mask_img = mask_img.resize([img_h, img_w], resample=Image.NEAREST)
        mask_arr = np.array(mask_img)
        new_mask_arr = np.zeros(mask_arr.shape[:2], dtype=mask_arr.dtype)
        new_mask_arr[np.where(np.all(mask_arr == [216, 124, 18], axis=-1))] = 0
        new_mask_arr[np.where(np.all(mask_arr == [255, 255, 255], axis=-1))] = 1
        new_mask_arr[np.where(np.all(mask_arr == [216, 67, 82], axis=-1))] = 2
        unique, counts = np.unique(new_mask_arr, return_counts=True)
        while len(counts) < 3:
            counts = np.append(counts, [1]) #Laplace smoothing like
        return counts

def compute_weight_unbalance_mean(filenames):
        counts = np.array([0, 0, 0], dtype=np.float64)
        for img_path in filenames:
            count = compute_segmentation_masks_count(img_path
                    .replace("jpg", "png").replace("Images", "Masks"))
            count_norm = count / np.linalg.norm(count)
            count_norm[1] = count_norm[0] / count_norm[1]
            count_norm[2] = count_norm[0] / count_norm[2]
            counts += count_norm
        classes = [0, 1, 2]
        return dict(zip(classes, counts/len(filenames)))
```

Figure 3: General weighting

But then we thought that we would have been more precise if we had no more a simple re-weighting of the classes for the entire data set, but if we had a re-weighting for each single sample. So we slightly changed the final computation and we represented these weights to be assigned as a circular iterator so that it could be used without any problems in the continuous training process epoch after epoch, with the constraints of having batches of size 1 (so that there would be an exact correspondence between a sample and its weights) and of having a shuffling only during the initial reading of the samples file paths.
In Figure 4 we show everything explained above.

```
def compute_weight_unbalance_matrix(filenames):
        counts = []
        for img_path in filenames:
            count = compute_segmentation_masks_count(img_path
                    .replace("jpg", "png").replace("Images", "Masks"))
            count_norm = count / np.linalg.norm(count)
            count_norm[1] = count_norm[0] / count_norm[1]
            count_norm[2] = count_norm[0] / count_norm[2]
            counts.append(count_norm)
        return np.asarray(counts)

sample_weights = compute_weight_unbalance_matrix(filenames)
sample_weights_iterator = cycle(sample_weights)
```

Figure 4: Per sample weighting

Now that we had the weight to fix the unbalance for each single sample in training, we needed a loss function to handle this. In Tensorflow/Keras there is no existing loss to handle such specific type of weighting and the existing parameters class_weight and sample_weight, in the Tensorflow/Keras' method Model.fit, do not work at all for our specific dynamic case and type of data (lots of issues present at their GitHub repositories). So, we wrote our new loss function which simply added the notion of weight to the output of any kind of existing loss function passed as argument, as shown below in Figure 5. (In our case we used as base loss the Sparse Categorical Cross Entropy loss).

```
def weightedLoss(originalLossFunc):

    def lossFunc(true, pred):

        classSelectors = true #being sparse categorical cross entropy, no argmax here
        weightsList = next(sample_weights_iterator)

        classSelectors = [K.equal(tf.cast(i, tf.float32), classSelectors)
                          for i in range(len(weightsList))]

        classSelectors = [K.cast(x, K.floatx()) for x in classSelectors]

        weights = [sel * w for sel,w in zip(classSelectors, weightsList)]

        weightMultiplier = weights[0]
        for i in range(1, len(weights)):
            weightMultiplier = weightMultiplier + weights[i]

        loss = originalLossFunc(true,pred)
        loss = loss * weightMultiplier

        return loss

    return lossFunc
```

Figure 5: Custom weighted loss

4 Model Choice

The first architecture used was a VGG16 with 5 layers to to do up sampling and perform segmentation on the produced activation masks. We have been using 3x3 filter. We used the fine-tuning technique to better refine also the weights of the base model other than the ones used to perform the segmentation.

We then moved to try different kinds of models, the one we focused more out attention were ResNet152V2 and DenseNet201 because they gave us the best overall results.

For each various model we always applied fine-tuning and we always added some regularization factor such as L2 regularization and Dropout layers in order to reduce over fitting.

5 Architecture of the final models

These two final models are composed by a top model of 5 decoders (UpSampling and Conv2D) and LeakyReLU as activation function which is known to perform slightly better than ReLU. Batch-Normalization was already present in the base models, which were fine-tuned, to improve the regularization, particularly the robustness to co-variate shift.
Moreover, on top of these layers we added Dropout with 0.2 rates to further regularize the model and prevent over fitting together with the L2 regularization applied also to the base model. These values showed to perform better. The actual model here described can be seen in the below Figure 6.

```
models = [tf.keras.applications.DenseNet201, tf.keras.applications.ResNet152V2]
model = models[i](weights='imagenet', include_top=False, input_shape=(img_h, img_w, 3))

def create_model(model_input, depth, num_classes):

    model = tf.keras.Sequential()

    # Encoder
    # -------
    model.add(model_input)

    start_f = 256

    # Decoder
    # -------
    for i in range(depth):
        model.add(tf.keras.layers.UpSampling2D(2, interpolation='bilinear'))
        model.add(tf.keras.layers.Conv2D(filters=start_f,
                                         kernel_size=(3, 3),
                                         strides=(1, 1),
                                         padding='same',
                                         kernel_initializer=tf.keras.initializers.GlorotNormal()))
        model.add(tf.keras.layers.LeakyReLU()) #using Xavier and LeakyReLU

        start_f = start_f // 2

    # Prediction Layer
    # ----------------
    model.add(tf.keras.layers.Conv2D(filters=num_classes,
                                     kernel_size=(1, 1),
                                     strides=(1, 1),
                                     padding='same',
                                     activation='softmax',
                                     kernel_initializer=tf.keras.initializers.GlorotNormal()))

    return model

model = create_model(model, depth=5, num_classes=3)
#adding dropout and L2 regularization (see notebook)
model = add_regularizations(model, dropout_after="leaky")
```

Figure 6: Architecture of the final models

The final models have respectively:

- 22908835 trainable parameters out of 23137891 with a total of 222 layers for the model with DenseNet201 as base model.
- 63298723 trainable parameters out of 63442467 with a total of 173 layers for the model with ResNet152V2 as base model.

6 Training

The training parameters are summarized in Table 1.

Table 1: Training parameters

Training argument	Parameter	Reason
loss	Custom weighted sparse categorical cross entropy	It's a multi-label classification with unbalanced label proportionality.
steps_per_epochs	72	Using a batch-size of 1 to allow correct per-sample-weight.
validation_steps	18	Using a batch-size of 1 to allow correct per-sample-weight.
Weight initialization	Xavier	To have better starting weights which should speed up the training and let us have a better final result of gradient descent.
optimizer	Nadam	Dozat (the author) says that is should give better results than Adam which is the de-facto standard.
learning_rate	0.0001	Value which allowed the model to learn, the suggested values of 0.001 was too high.
beta_1	0.9	Value suggested in the paper.
beta_2	0.999	Value suggested in the paper.
Callback	Early Stopping	To stop the model when it reaches convergence and it act as weight decay which help preventing over-fit.
delta	0.001	A small enough value, it's $\approx \frac{1}{100}$ of the final loss of the models.
patience	10	A patience big enough that the RLROP can act at least 3 times before stopping.
restoring best weights	True	When having a validation set True is the best option.
Callback	RLROP	Reduce Learning Rate On Plateau, this should help the model achieve better performance in the last part of the training.
RLROP factor	0.1	This value was taken from an example.
RLROP patience	3	We set the patience so that it can trigger 3 times before the early stopping kills the process.
RLROP delta	0.001	The same delta of the early stopping.

For the weight initialization with Xavier Initialization we used GlorotNormal to better initialize weights W and letting back propagation algorithm start in advantage position, given that final result of gradient descent is affected by weights initialization.

$$W \sim \mathcal{N}\left(\mu = 0,\ \sigma^2 = \frac{2}{N_{in} + N_{out}}\right)$$

We also set a checkpoint folder in order to save our model state during the training process after each epoch.
For reproducible result we set a constant seed in order to have the same pseudo-randomization on data by TensorFlow and by Numpy. The chosen seed was "randomly" 1234.

Also, to really speed up the training process at each epoch we decided to load all these training data into RAM without letting it read and elaborate every time from the disk, this really improved speed and reduced latency allowing us to explore more options.

7 Results

With the first architecture using DenseNet201 we were able to reach on CodaLab an IoU on the test data set of 77.57%, while with the second architecture using as base model ResNet152V2, we've reached an IoU on the test set of 77.61%.
They are indeed very similar models, also in the metrics acquired and shown in the Figures 7 and 8 below. One could say that at the cost of slightly worse results it's preferable the model with DenseNet201 as base due to its smaller number of parameters with respect to ResNet152V2 which means reduce prediction time and potential reduce of further small over fitting.

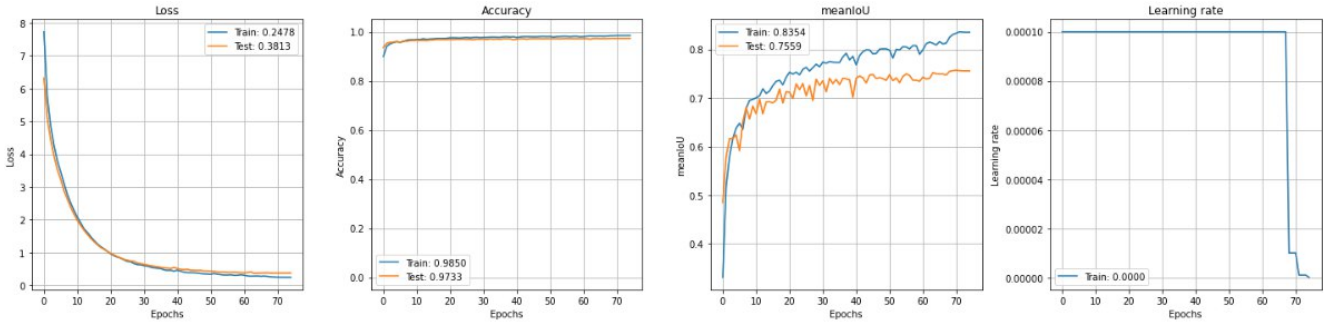

Figure 7: The training results of the run with DenseNet201 as base model.

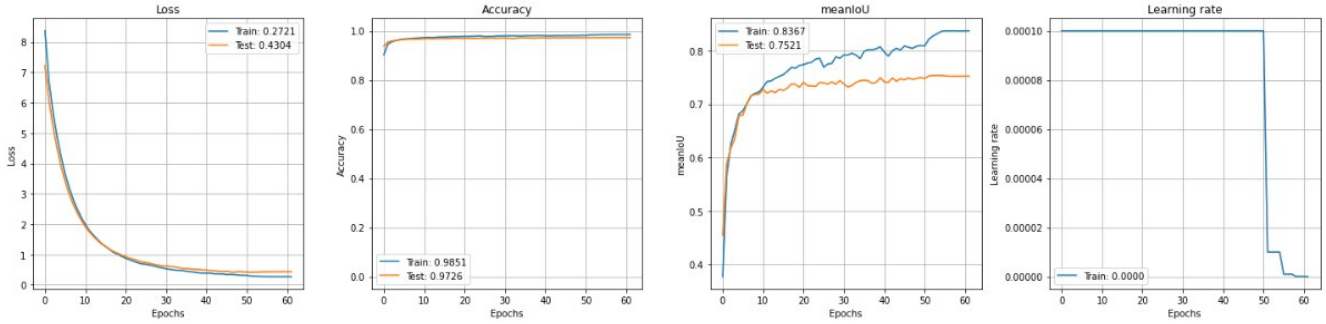

Figure 8: The training results of the run with ResNet152V2 as base model.

In the images it's also possible to appreciate the great work done by the regularization techniques by looking at the losses.

In the below Figures 9 and 10 it's also good to see the difference at prediction on a particular same random image.

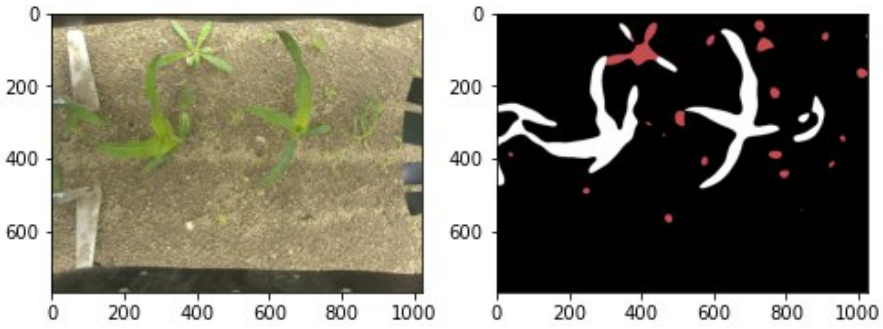

Figure 9: Prediction result with DenseNet201 as base model.

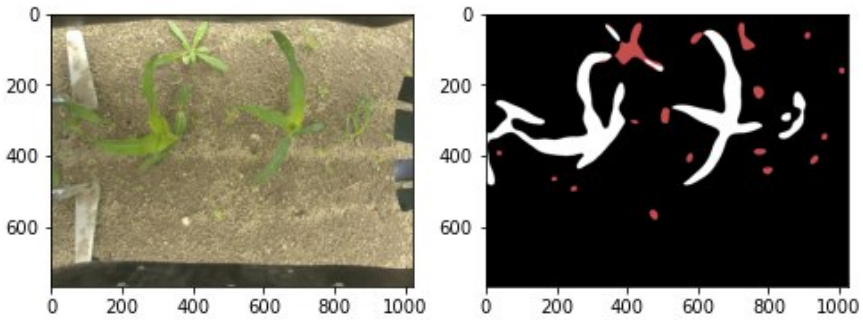

Figure 10: Prediction result with ResNet152V2 as base model.

Given their really high similarity we tried also to apply a Stacked Ensembling model, using a meta learner to be trained in order to have a model assigning the best possible weights to these two models at prediction time. Unfortunately we only improved the global result on the overall data set to 0.0634313027 but not on the one we chose to specifically train onto (it was reduced to about 75%).

8 Conclusions

Again, we've come to the same conclusion that such Deep Learning and Machine Learning problems are much affected by little but important changes, there is much more than, for example, the number of layers in a model to be considered. We've seen the importance of having to regularize a model in presence of little data in order to get the best of it, helping us also with data augmentation and various other techniques.
In this second challenge we've really seen the empirical side of Deep Learning: you think something may work or may be it won't? You have to try it! There is very little theoretical background to guarantee some results on any given model on any kind and type of data sets. For example using the EfficientNet B5 as base model performed actually worse than these two best models presented and the reason is not very clear to us yet.
As last note, as always there are really a lot of potential hyper parameters but it's not easy given our hardware resources available to search for the optimum in the space of these hyper parameters.

9 Possible improvements

We want the network to see most of the possible details, so one possible improvement would be to use the tiling technique so that we would be able to virtually train the network on the full-sized images and so let it being able to spot most of details. Although, this require more work to be consistent with the custom weighted loss during the training process.
An example of tiling applied to an input sample is shown below in the Figure 11.

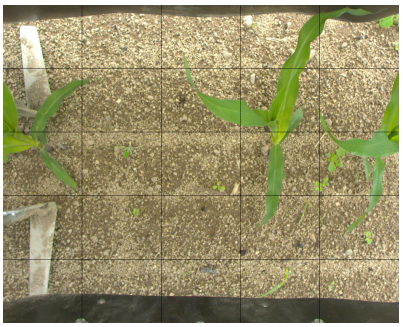

Figure 11: Tiling applied to a sample.

An other possible improvement, which again may or may not be an improvement, is applying as data augmentation a copy-paste technique on input images by randomly selecting different objects to be scaled and pasted in a same image and be further segmented. This technique has shown some good results in the very recent paper Simple Copy-Paste is a Strong Data Augmentation Method for Instance Segmentation (`https://arxiv.org/pdf/2012.07177.pdf`).