POLITECNICO
MILANO 1863

Politecnico di Milano
Artificial Neural Network and Deep Learning

# Multiclass Semantic Segmentation

Second Challenge

*Authors*
Enrico Gherardi
Ludovico Righi

December 28, 2020

# Contents

**1 Setting Up the Environment** **1**
1.1 Create the Dataset . . . . 1
1.2 Create the Final JSON . . . . 1

**2 Basic Models** **2**
2.1 Basic Model . . . . 2
2.2 Transfer Learning . . . . 2
2.3 Basic Unet . . . . 3

**3 Advanced Techniques for the Challenge** **4**
3.1 Our Custom Unet . . . . 4
3.2 Weighted Focal Loss . . . . 5
3.3 Multiclass Dice Loss . . . . 6

# Chapter 1

# Setting Up the Environment

## 1.1 Create the Dataset

Firstly, we decided to use only the Bipbip dataset but the same procedure can be applied to the others in the same way. To use the CustomDataset Class, we wrote a python script to move all the Haricot and Mais images in a folder called "Images" and their relative masks in a folder called "Annotations". Then we have also randomly split the dataset in training and validation creating the files "train.txt" and "validation.txt".

## 1.2 Create the Final JSON

We create an external script to generate the final JSON to submit our model. We made this decision to try different checkpoints of the same training. Although we decided to work only on Bipbip dataset, we decided to evaluate all the datasets for future extensions. We created a file "directory_path.txt" to iterate to all the datasets with their crops and weeds. For every folder we:

- load all the images with the method flow_from_directory()
- pass all the images to our model
- resize the output image to the starting resolution
- compute the result, rle-encode it, and update the result dictionary

# Chapter 2

# Basic Models

For every model we used Data Argumentation to prevent data scarcity and we try to train it with the highest resolution and bath-size that our system allow us. In general, we use (1024x1024x3) of input-size and 2-4 of batch-size.

## 2.1 Basic Model

Our first experiment was on the basic model presented in the Binary Segmentation Notebook. We adapt it to solve our Multiclass task. Unfortunately, this model was too simple and was not capable to reach good results.

## 2.2 Transfer Learning

For every model in this section we:

- use as an encoder a famous CNN architecture
- use 256 starting filters
- use 5 layers for up-sampling, convolution and relu
- use pre-trained IMAGENET weights but we train the entire network to get better results.

Our first try was with a VGG16, and with it we have a discrete result: 0.38 of meanIoU on Haricot and 0.51 on Mais.
Then we try to use as an encoder the Xception. In this way, we get a good result: 0.56 of meanIoU on Haricot and 0.66 on Mais.

```
vgg = tf.keras.applications.VGG16(weights='imagenet', include_top=False,
    input_shape=(img_h, img_w, 3))
vgg.summary()
for layer in vgg.layers:
    layer.trainable = True


    def create_model(depth, start_f, num_classes):
```

```
model = tf.keras.Sequential()
# Encoder
model.add(vgg)
# 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'))
    model.add(tf.keras.layers.ReLU())
    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'))

return model
```

## 2.3 Basic Unet

We try to use a basic unet, with 128 starting filters and 4 concatenated layer, but without any specific trick, we reach only a meanIoU of 0.31 on Haricot and 0.43 on Mais.
In the next chapter, we will introduce our unet custom model.

# Chapter 3

# Advanced Techniques for the Challenge

## 3.1 Our Custom Unet

As shown in the code below, we create our custom version of the Unet adding:

- BatchNormalization layer: to make networks faster and more stable through normalization of the input layer by re-centering and re-scaling.
- Try to reach the better combination of layers and filters

With our model we reach our best result: 0.60 of meanIoU on Haricot and 0.74 on Mais.

```
from tensorflow.keras.layers import Input, Conv2D, BatchNormalization,
   Activation, MaxPool2D, UpSampling2D, Concatenate
from tensorflow.keras.models import Model
def conv_block(inputs, filters, pool=True):
    x = Conv2D(filters, 3, padding="same")(inputs)
    x = BatchNormalization()(x)
    x = Activation("relu")(x)

    x = Conv2D(filters, 3, padding="same")(x)
    x = BatchNormalization()(x)
    x = Activation("relu")(x)

    if pool == True:
        p = MaxPool2D((2, 2))(x)
        return x, p
    else:
        return x

def build_unet(shape, num_classes):
    inputs = Input(shape)

    """ Encoder """
    x1, p1 = conv_block(inputs, 16, pool=True)
    x2, p2 = conv_block(p1, 32, pool=True)
    x3, p3 = conv_block(p2, 64, pool=True)
    x4, p4 = conv_block(p3, 128, pool=True)
    x5, p5 = conv_block(p4, 256, pool=True)

    """ Bridge """
    b1 = conv_block(p5, 512, pool=False)

    """ Decoder """
    u1 = UpSampling2D((2, 2), interpolation="bilinear")(b1)
    c1 = Concatenate()([u1, x5])
    x6 = conv_block(c1, 256, pool=False)

    u2 = UpSampling2D((2, 2), interpolation="bilinear")(x6)
    c2 = Concatenate()([u2, x4])
    x7 = conv_block(c2, 128, pool=False)

    u3 = UpSampling2D((2, 2), interpolation="bilinear")(x7)
    c3 = Concatenate()([u3, x3])
    x8 = conv_block(c3, 64, pool=False)

    u4 = UpSampling2D((2, 2), interpolation="bilinear")(x8)
    c4 = Concatenate()([u4, x2])
    x9 = conv_block(c4, 32, pool=False)

    u5 = UpSampling2D((2, 2), interpolation="bilinear")(x9)
    c5 = Concatenate()([u5, x1])
    x10 = conv_block(c5, 16, pool=False)


    """ Output layer """
    output = Conv2D(num_classes, 1, padding="same",
       activation="softmax")(x10)

    return Model(inputs, output)
```

## 3.2 Weighted Focal Loss

We tried different Losses to solve the problem of the unbalanced frequency of background, crop and weed in our dataset images. The Focal loss adds a factor $(1pt)^\gamma$ to the standard cross entropy criterion. Setting $\gamma > 0$ reduces the relative loss for well-classified examples (pt $>$ .5), putting more focus on hard, misclassified examples.
We try to add this loss to our models, but we did not get the expected results.

```
def categorical_focal_loss(alpha, gamma=2.):
    alpha = np.array(alpha, dtype=np.float32)
    def categorical_focal_loss_fixed(y_true, y_pred):
        # Clip the prediction value to prevent NaN's and Inf's
        epsilon = K.epsilon()
        y_pred = K.clip(y_pred, epsilon, 1. - epsilon)
        # Calculate Cross Entropy
        cross_entropy = -y_true * K.log(y_pred)
        # Calculate Focal Loss
        loss = alpha * K.pow(1 - y_pred, gamma) * cross_entropy
        # Compute mean loss in mini_batch
        return K.mean(K.sum(loss, axis=-1))
    return categorical_focal_loss_fixed
```

## 3.3 Multiclass Dice Loss

When using cross entropy loss, the statistical distributions of labels play a big role in training accuracy. The more unbalanced the label distributions are, the more difficult the training will be. Although weighted cross entropy loss can alleviate the difficulty, the improvement is not significant nor the intrinsic issue of cross entropy loss is solved. In cross entropy loss, the loss is calculated as the average of per-pixel loss, and the per-pixel loss is calculated discretely, without knowing whether its adjacent pixels are boundaries or not. As a result, cross entropy loss only considers loss in a micro sense rather than considering it globally, which is not enough for image level prediction.
So we decided to implement our multiclass version of the Dice-Loss: a loss that considers the loss information both locally and globally, which is critical for high accuracy.
With the dice-loss we get a boost in the meanIoU from 1 to 5 per cent in all our models.

```
from keras import backend as K
def dice_coef_multi(y_true, y_pred, smooth=1e-7):
    y_true_f = K.flatten(K.one_hot(K.cast(y_true, 'int32'),
        num_classes=3)[...,1:])
    y_pred_f = K.flatten(y_pred[...,1:])
    intersect = K.sum(y_true_f * y_pred_f, axis=-1)
    denom = K.sum(y_true_f + y_pred_f, axis=-1)
    return K.mean((2. * intersect / (denom + smooth)))

def dice_coef_loss_multi(y_true, y_pred):
    return 1 - dice_coef_multi(y_true, y_pred)
```