Homework 02

ELE 2346 - DEEP LEARNING

Pedro Henrique Cardoso Paulo
pedrorjpaulo.phcp@gmail.com

Professor: Raul Queiroz Feitosa

Departamento de Engenharia Mecânica
PUC-RJ Pontifícia Universidade Católica do Rio de Janeiro
May, 2023

Homework 02

ELE 2346 - DEEP LEARNING

Pedro Henrique Cardoso Paulo

May, 2023

1 Introduction

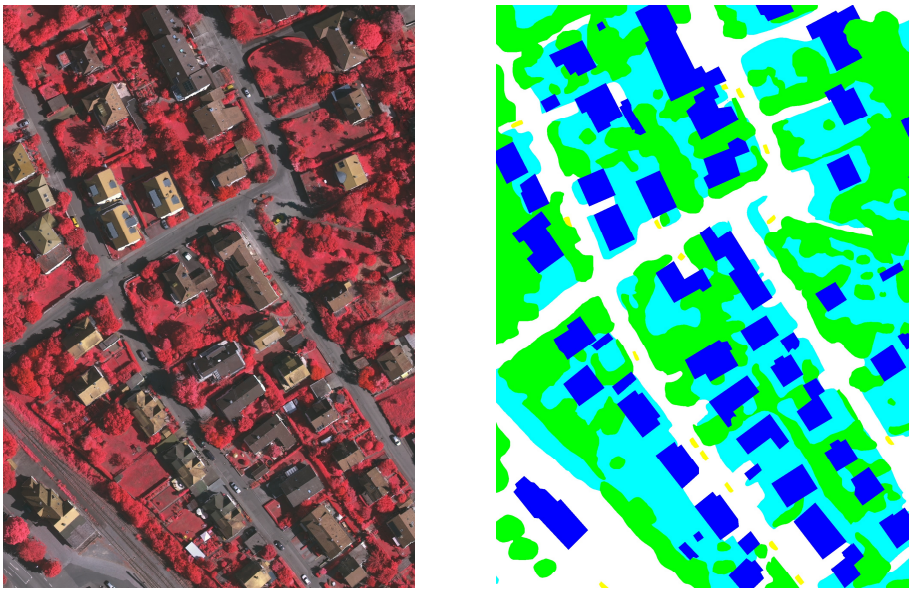
1.1 Objectives

The main objectives of this exercice is to provide the students some experience with:

- The PyTorch library
- Segmentation models
- The patch strategy for dealing with big images

1.2 Dataset

For this exercice, the dataset will be comprised of two images that will be splitted into train, validation and test sets by creating patches of the image. Figure 1 shows the images belonging to the train and test dataset.



(a) Train set **(b)** Train mask

Figure 1: Train image to be used in the exercice

It is important to notice that the datasets are deeply unbalanced, with the class Car (4) having considerably less samples than the others. The table 1 summarizes the class distribution of the pixels in the train and test dataset.

1.3 Exercice

The main objective of this notebook is to implement a DeepLabV3+ Network for semantic segmentation using Segmentation Models of PyTorch. The following steps should be followed:

1. Split the train image for training (80%) and validation (20%)

Class	train image	test image
0 (Buildings)	17.62%	18.07%
1 (Tree)	33.59%	29.75%
2 (Low vegetation)	25.50%	35.36%
3 (Car)	00.27%	00.38%
4 (Impervious surfaces)	23.01%	16.44%

Table 1: Classes distribution in train and test datasets

2. Generate patches from the training image (128x128, 64x64, 32x32)
3. Train a DeepLabV3+ network using the Segmentation Models of pytorch using the ecoder `'mobilenet_v2'` from scratch and with pretrained weights of `'imagenet'`, `encoder_depth = 4` and `decoder_channels = [256, 128, 64, 32]`
4. For training, use the `\weighted_categorical_crossentropy"` as a loss function. To compute the weights you must count the number of pixels of each class
5. Evaluate the model on the test image using patches and make the mosaic to visualize the complete image
6. Compute metrics for each class
7. Compare and analize the results

Use the following mean and standard deviation values according to the adopted weights initialization:

```
#Normalization for ImageNet
mean_ = [0.485, 0.456, 0.406]
std_  = [0.229, 0.224, 0.225]

#Normalization for training from scratch
mean_ = 0.0
std_  = 1.0
```

1.4 Cases of study

From the proposed exercice, the following tests will be performed and compared regardign general error metrics and confusion matrices:

1. Using imagenet weights as starting values
   a Creating patches of 32 x 32 pixels
   b Creating patches of 64 x 64 pixels
   c Creating patches of 128 x 128 pixels
2. Using randomly initiated weights
   a Creating patches of 32 x 32 pixels
   b Creating patches of 64 x 64 pixels
   c Creating patches of 128 x 128 pixels

2 Methodology

2.1 Dataset definition

In order to perform the tasks demanded by the exercice, the class `PatchDataset` was created. This class was based on the `DroneDataset` used on the in class exercice and has the function of reading the train image adn breaking it into patches for training the network. The python code of this class is exemplified below:

```
class PatchDataset(Dataset):

    def __init__(self, img_path, mask_path, mean, std, transform=None, n_patches=32, stride=1)
                                                                                    :
        self.img_path = img_path
        self.mask_path = mask_path
```

```
        self.transform = transform
        self.n_patches = n_patches
        self.stride = stride
        self.mean = mean
        self.std = std

        #Reading the image
        img = cv2.imread(self.img_path)
        img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
        mask = cv2.imread(self.mask_path, cv2.IMREAD_GRAYSCALE)
        self.img = img
        self.mask = mask
        orig_gray_lbls = np.sort(np.unique(mask))
        self.label_dict = {}
        self.classes_weights = []
        class_i = 0
        for orig_lbl in orig_gray_lbls:
          mask[mask == orig_lbl] = class_i
          self.label_dict[class_i] = orig_lbl
          print(f'Total number of elements in class {class_i}: {np.sum(mask == class_i)} ({np.
                                                                sum(mask == class_i)/mask.size})')
          self.classes_weights.append(mask.size/np.sum(mask == class_i))
          class_i += 1

        img = torch.from_numpy(img).long()
        mask = torch.from_numpy(mask).long()

        self.img_patches = img.unfold(0,self.n_patches,self.stride).unfold(1,self.n_patches,
                                                                self.stride)
        self.mask_patches = mask.unfold(0,self.n_patches,self.stride).unfold(1,self.n_patches,
                                                                self.stride)
        self.img_patches_shape = self.img_patches.shape

    def __len__(self):
        self.img_patches_shape = self.img_patches.shape
        return self.img_patches_shape[0]*self.img_patches_shape[1]

    def __getitem__(self, idx):

        self.img_patches_shape = self.img_patches.shape

        idx_x = idx % self.img_patches_shape[0]
        idx_y = idx // self.img_patches_shape[0]

        img = self.img_patches[idx_x, idx_y, :, :, :]
        img = img.transpose(0,2).transpose(0,1).numpy().astype(np.uint8)

        mask = self.mask_patches[idx_x, idx_y, :,  :].numpy()

        if self.transform is not None:
            aug = self.transform(image=img, mask=mask)
            img = Image.fromarray(aug['image'])
            mask = aug['mask']

        if self.transform is None:
            img = Image.fromarray(img)

        t = T.Compose([T.ToTensor(), T.Normalize(self.mean, self.std)])
        img = t(img)
        mask = torch.from_numpy(mask).long()

        return img, mask
```

This class also estimates weights to be used by the `torch.nn.CrossEntropyLoss` metric in order to train the network. The weights for each class are calculated by the expression displayed in equation 1, where n_i is the number of pixels belonging to the class i in the image. It is worth noticing that this weight corresponds to the inverse of the class distribution in the training dataset, as displayed in table 1.

$$w_i = \frac{\sum n_i}{n_i} \tag{1}$$

Another implemented class was the `PatchTestDataset`, that has the objective of generating patches for executing predictions. This class is similar to the `PatchDataset` class, but also has a method that allows, given a trained model, excecute patch predictions for the input image and consolidate the results as a segmentation for the whole image area. This method can be explored in the code below.

```
    def predict_full(self, model, device):

      n_classes = len(self.label_dict)
      full_mask = torch.zeros((n_classes, self.img.shape[0], self.img.shape[1]))
      mean_mask = torch.zeros((n_classes, self.img.shape[0], self.img.shape[1]))
      t = T.Compose([T.ToTensor(), T.Normalize(mean, std)])
      model.eval()
      model.to(device)
      for i in range(len(self)):

        image, mask = self[i]
        idx_x = i % self.img_patches_shape[0]
        idx_y = i // self.img_patches_shape[0]

        coord_x = idx_x * self.stride
        coord_y = idx_y * self.stride

        image = t(image)
        image=image.to(device)
        mask = mask.to(device)
        with torch.no_grad():

          image = image.unsqueeze(0)
          mask = mask.unsqueeze(0)

          masked = model(image)
          masked = masked.cpu().squeeze(0)

        full_mask[:,coord_x:coord_x+self.n_patches, coord_y:coord_y+self.n_patches] += masked
        mean_mask[:,coord_x:coord_x+self.n_patches, coord_y:coord_y+self.n_patches] = torch.
                                                        ones(masked.shape)

      img = torch.from_numpy(self.img).long()
      img = img.numpy().astype(np.uint8)

      img = Image.fromarray(img)
      real_mask = torch.from_numpy(self.mask).long()
      full_mask /= mean_mask
      pred_mask = full_mask.argmax(dim=0)

      return img, real_mask, pred_mask
```

It is worth mentioning that, since different patches can have intersections, the convention used for these cases is to consider the result for the intersection as the simple average of all the patches that cover that area.

2.2 Train and validation split

In order to perform validation, the dataset created by dividing the train image in patches as splitted using the torch functionality `SubsetRandomSampler`. After that, two `DataLoaders` were created for train and validation. The code for this particular step is presented below.

```
# Creating data indices for training and validation splits:
dataset_size = len(dataset)
indices = list(range(dataset_size))
split = int(np.floor(validation_split * dataset_size))
if shuffle_dataset :
    np.random.seed(random_seed)
    np.random.shuffle(indices)
train_indices, val_indices = indices[split:], indices[:split]

# Creating PT data samplers and loaders:
train_sampler = SubsetRandomSampler(train_indices)
valid_sampler = SubsetRandomSampler(val_indices)

train_loader = torch.utils.data.DataLoader(dataset, batch_size=batch_size,
                                           sampler=train_sampler)
validation_loader = torch.utils.data.DataLoader(dataset, batch_size=batch_size,
                                                sampler=valid_sampler)
```

2.3 Hyperparameter selection

In this exercice, the only hyperparameters that allowed some tunning where the stride of the patches, the parameters related to the optimization epochs and the maximum number of epochs. The selected stride was

16 in order to allow at least 50% overlapping in the 32 x 32 filter case. Also, the number of epochs was raised to 20 based on some preliminary runs made with the network, that showed in the history plots that there was still some room for improvement of the model by allowing more training epochs. The parameters regarding the optimization process (learning rate, optimizer) were kept unchanged.

Other relevant comment regarding hyperparameters is that, due to limitations regarding the `segmentation-models-pytorch` package API, the proposed hyperparameters to be used in the DeepLabV3+ model and listed in the step 3 of the exercice were not possible. In order to comply with the limitations of the API, the following configuration were used:

```
model = smp.DeepLabV3Plus('mobilenet_v2', encoder_weights='imagenet', classes=5,
                                                    decoder_channels=256)
```

It is worth mentioning that, while the limitation of the `decoder_channels` is documented on the package's documentation, the limitatin on the `classes` parameter seems to be an implementation error and only the default value 5 worked on this case. For more information regarding the DeepLabV3+ model API, refer to the package documentation.

3 Results and discussions

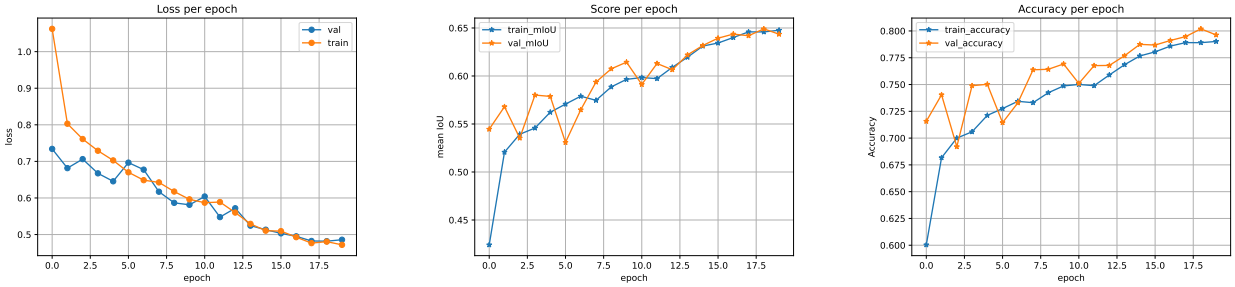
3.1 Colab Notebook

All the code, examples and tests made are documented on the following Colab Notebooks.

- Link to the Colab Notebook
- Link to the Colab Notebook used for metrics calculation

3.2 Item 1a

Figure 2 shows the convergency of the 20 epochs using the ImageNet weights as initialization and dividing the train image into 32 x 32 patches with a stride of 16 pixels. It is possible to see that the value of 20 epochs seems to be a reasonable value for the problem with no overfitting (train and validation with very different performances) and apparently a final value very close to the minimum achievable with more epochs. More epochs could perhaps improve the final result, but given the computational cost for the problem, the result obtained for this case is considered satisfactory.



(a) Loss evolution **(b)** Score evolution **(c)** Accuracy evolution

Figure 2: Metrics evolution during training for item 1a

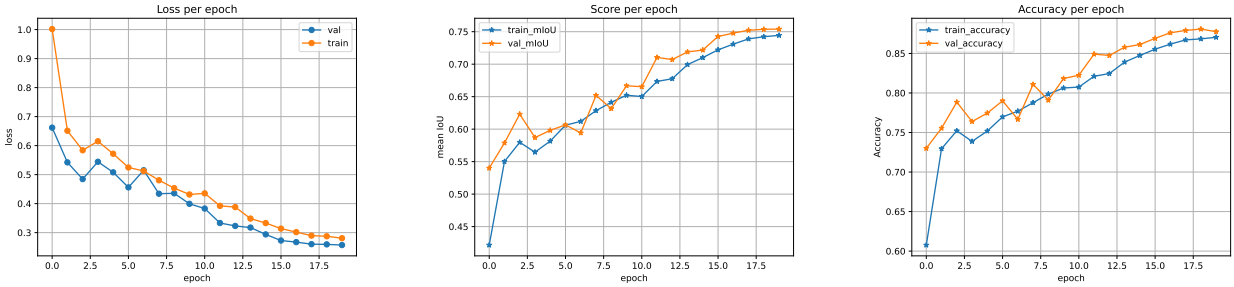
In figure 3 we see a comparison of the results obtained by running a prediction in the train and test datasets. The error metric indicated in the image is the mean IoU between the classes. For this first test it is worth noticing that good results were obtained for both the train and test results, but there is still room for improvement. The model seems to present great confusion between the classes 1 (Tree) and 2 (Low vegetation), which is justifiable for their similarities regarding color and aspect when seen from above. It is also worth mentioning that the class 3 (Car) i considerably well predicted considering that it is a subsampled class, ti all thanks to the weights selected. Also worth mentioning is that the improvement in the class 3 prediction also comes with a cost, since it is possible to see in the results that two near class 3 regions tend to be merged, supressing class 4 (Impervious surfaces) between them.

(a) Train results **(b)** test results

Figure 3: Results of predictions for item 1a

3.3 Item 1b

Figure 4 shows the convergency of the 20 epochs using the ImageNet weights as initialization and dividing the train image into 64 x 64 patches with a stride of 16 pixels. As compared to case 1a, the same conclusions can be taken from this test, together with a general improvement in the metrics results.



(a) Loss evolution **(b)** Score evolution **(c)** Accuracy evolution

Figure 4: Metrics evolution during training for item 1b

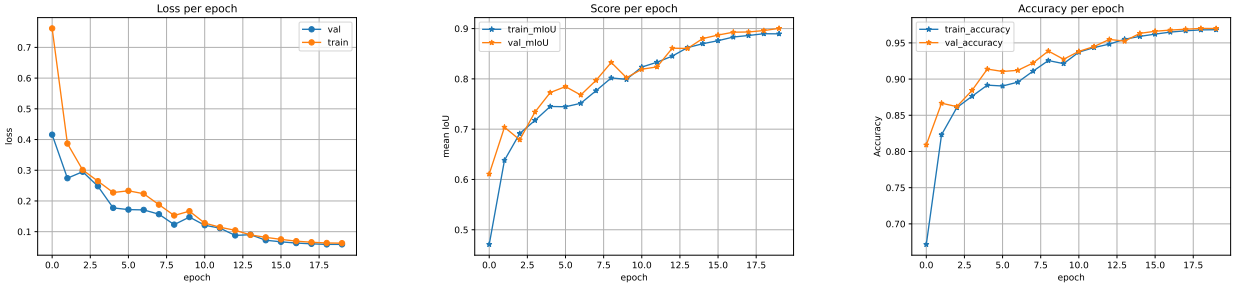
In figure 5 we see a comparison of the results obtained by running a prediction in the train and test datasets. The error metric indicated in the image is the mean IoU between the classes. Once again, the conclusions from the test 1a are still valid, with the comment on the improvement in performance and specially better separation between classes 1 and 2.

(a) Train results **(b)** test results

Figure 5: Results of predictions for item 1b

3.4 Item 1c

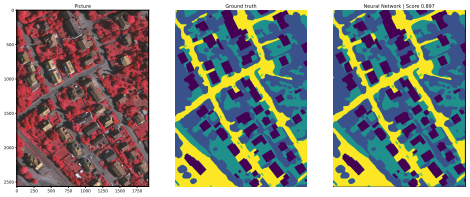
Figure 6 shows the convergency of the 20 epochs using the ImageNet weights as initialization and dividing the train image into 128 x 128 patches with a stride of 16 pixels. Once again, as compared to cases 1a and 1b, the same conclusions can be taken from this test, together with a general improvement in the metrics results.



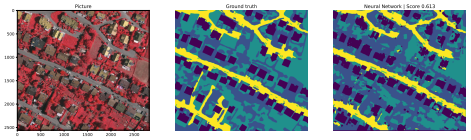
(a) Loss evolution **(b)** Score evolution **(c)** Accuracy evolution

Figure 6: Metrics evolution during training for item 1c

In figure 5 we see a comparison of the results obtained by running a prediction in the train and test datasets. The error metric indicated in the image is the mean IoU between the classes. Same conclusions from the previous cases can be drawn from this test.



(a) Train results

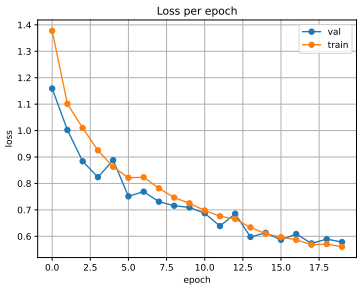


(b) test results

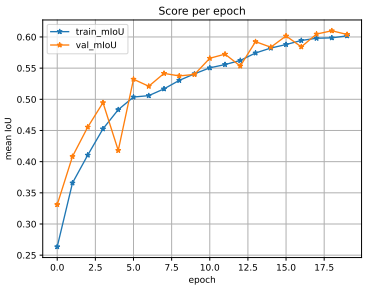
Figure 7: Results of predictions for item 1c

3.5 Item 2a

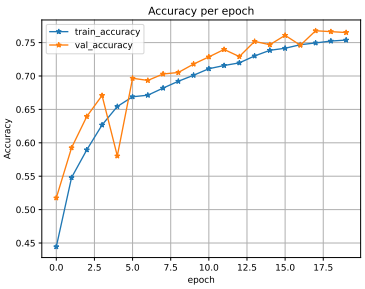
Figure 8 shows the convergency of the 20 epochs using the random weights as initialization and dividing the train image into 32 x 32 patches with a stride of 16 pixels. Overall, the results are similar to the case 1a, but is noticeable a worse performence of this initial weights. Perhaps this could be mitigated by allowing more training epochs, but it does not seem to be the case since the aspect of the convergency is similar to the one displayed in Figure 2



(a) Loss evolution



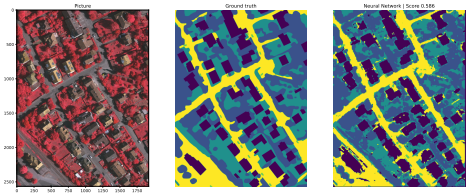
(b) Score evolution



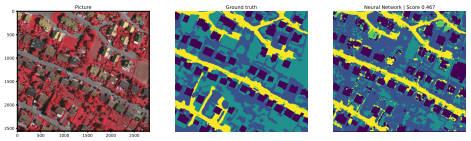
(c) Accuracy evolution

Figure 8: Metrics evolution during training for item 2a

The results obtained by running predicitions on the images nd displayed in Figure 9 also show a worse version of the prediction performed in case 1a.



(a) Train results

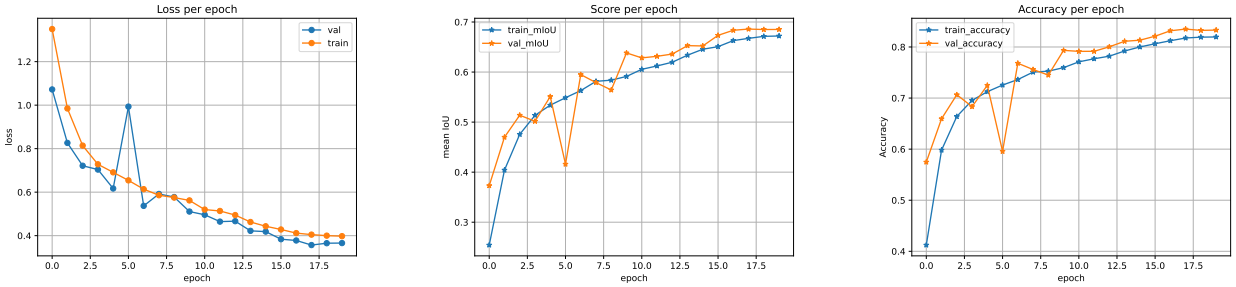


(b) test results

Figure 9: Results of predictions for item 2a

3.6 Item 2b

Figures 10 and 11 show the convergency of the 20 epochs and test results obtained by using the random weights as initialization and dividing the train image into 64 x 64 patches with a stride of 16 pixels. Once again, slightly worse results are obtained when compared to case 1b.



(a) Loss evolution **(b)** Score evolution **(c)** Accuracy evolution

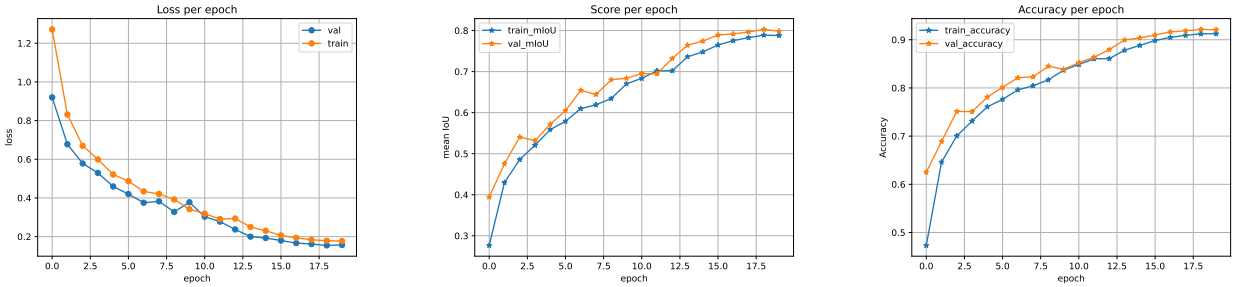
Figure 10: Metrics evolution during training for item 2b

(a) Train results **(b)** test results

Figure 11: Results of predictions for item 2b

3.7 Item 2c

Figures 12 and 13 show the convergency of the 20 epochs and test results obtained by using the random weights as initialization and dividing the train image into 128 x 128 patches with a stride of 16 pixels. Once again, slightly worse results are obtained when compared to case 1c.



(a) Loss evolution **(b)** Score evolution **(c)** Accuracy evolution

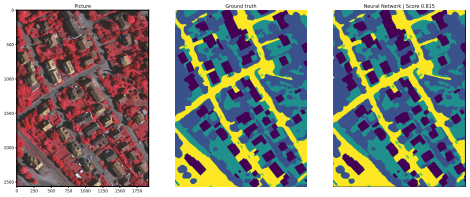
Figure 12: Metrics evolution during training for item 2c

3.8 Results Summary and Conclusions

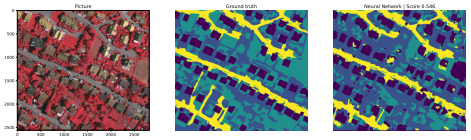
The patch results at the end of the training are summarized in table 2. From this results it is possible to draw three main conclusions:

- Bigger patches, such as 128 x 128 tend to present better values than smaller patches. This is probably due to the greater information arera they provide to the model when compared to their smaller, more localized, couterparts
- Starting from the ImageNet weights allow better performances at the end of the 20 epochs
- The change in initialization of the weights does not change considerably the training time for 20 epochs

The same conclusions can be achieved by analizing the results summary for the whole train and test images regarding precision, recall and F1 score. It should be kept in mind that this scores were calculated using the



(a) Train results



(b) test results

Figure 13: Results of predictions for item 2c

Exp	train acc	train loss	train IoU	val acc	val loss	val IoU	training time
1a	79.0%	0.472	0.648	79.6%	0.486	0.644	18.20 min
1b	86.8%	0.281	0.744	87.8%	0.257	0.754	20.03 min
1c	96.8%	0.063	0.890	97.0%	0.059	0.900	30.69 min
2a	75.4%	0.561	0.602	76.5%	0.578	0.604	19.11 min
2b	82.0%	0.398	0.672	83.3%	0.366	0.685	21.00 min
2c	91.2%	0.177	0.788	92.1%	0.157	0.798	29.82 min

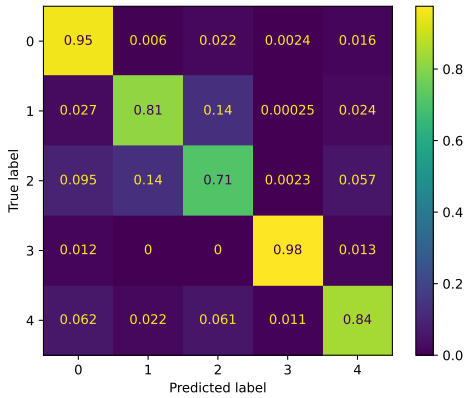
Table 2: Results summary for batch training

`scikit-learn` functions having the average by classes taken using the `average='weighted'` option. It is also important to notice that, while the train and validation differences at the end of the training are very small, the differences in performance between the train and test images are not neglegible.

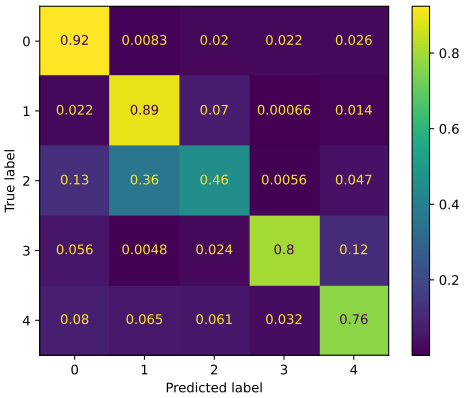
Exp	train precision	train recall	train F1 score	test precision	test recall	test F1 score
1a	82.31%	81.91%	81.87%	75.32%	72.28%	71.07%
1b	90.58%	90.28%	90.32%	76.95%	75.00%	74.30%
1c	97.51%	97.42%	97.44%	78.34%	77.08%	76.73%
2a	78.19%	77.75%	77.70%	72.68%	69.50%	68.36%
2b	85.36%	84.87%	84.89%	74.66%	71.62%	70.12%
2c	93.93%	93.77%	93.79%	75.25%	73.11%	71.92%

Table 3: Results summary for train and test image

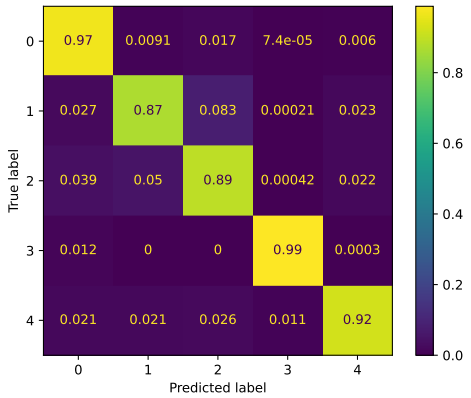
Figures 14 and 15 represent the prediction results for train and test images in terms of confusion matrices, with each class being normalized by its real labels. It is interesting to notice that the greatest source of confusion between predictions lays between classes 1 (Tree) and 2 (Low vegetation), that are similar in color and aspect when observed in the images. Once again, it is made evident that the performance in the train image is far superior from the one in the test image.



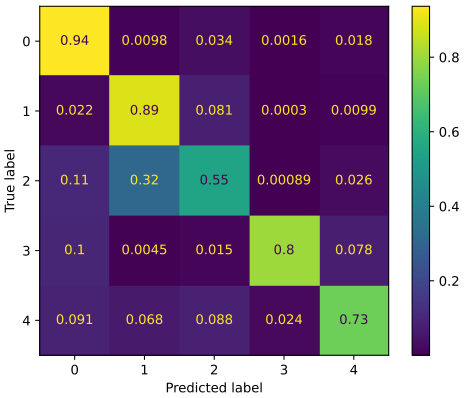
(a) Item 1a train results



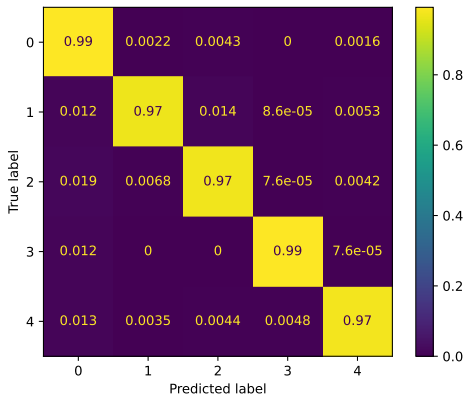
(b) Item 1a test results



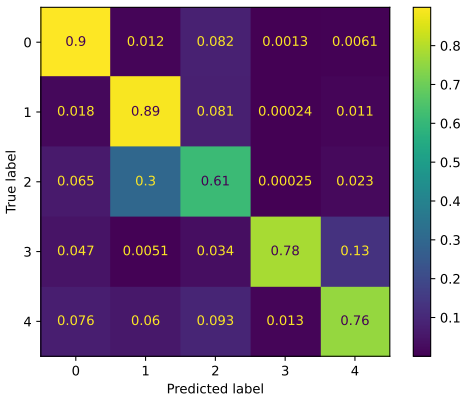
(c) Item 1b train results



(d) Item 1b test results

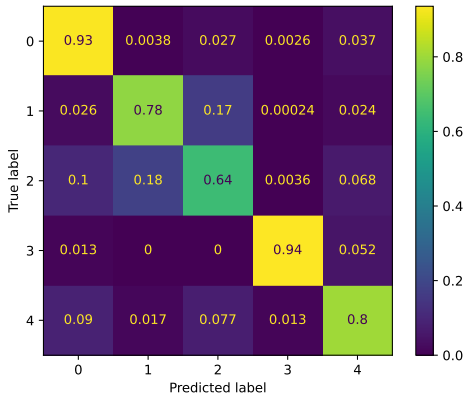


(e) Item 1c train results

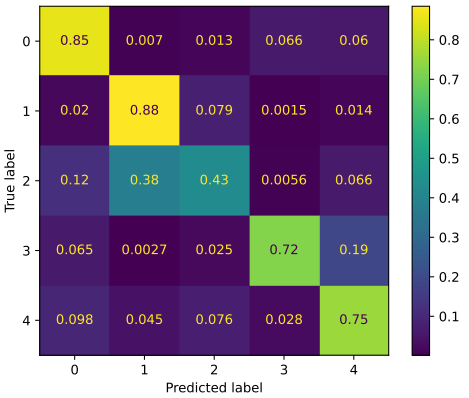


(f) Item 1c test results

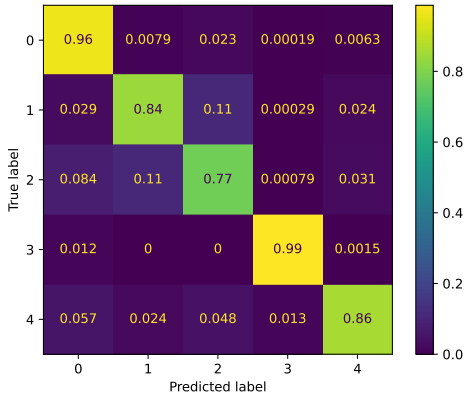
Figure 14: Confusion matrices for performed tests item 1



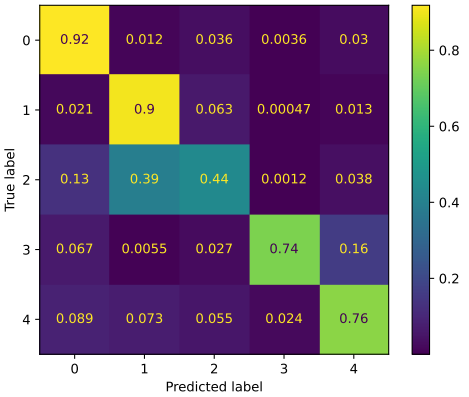
(a) Item 2a train results



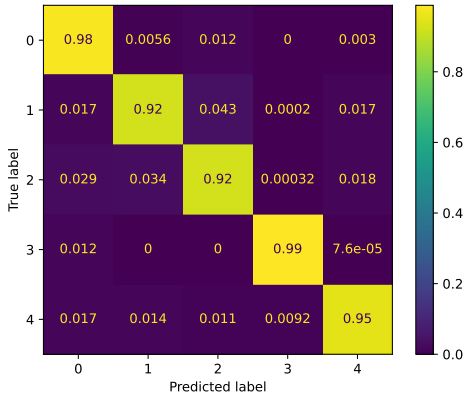
(b) Item 2a test results



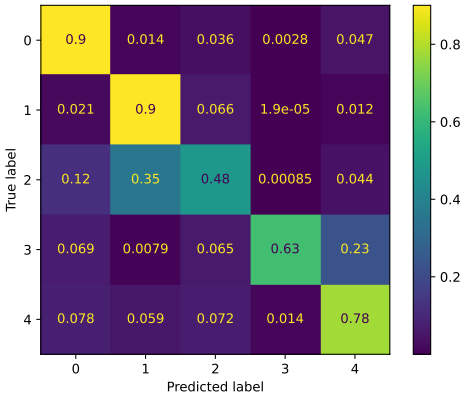
(c) Item 2b train results



(d) Item 2b test results



(e) Item 2c train results



(f) Item 2c test results

Figure 15: Confusion matrices for performed tests item 2