Generate ndvi images from rgb images to reduce the costs of using new technologies in agriculture

Ivan De Cosmis, Federico Canistro

***Abstract*—We have created and trained a model using DeepLab and EfficientNet to obtain NDVI(Normalized Difference Vegetation Index) images starting from RGB images, using different metrics to evaluate the results. We found that this work could be a solution for the economic problem that arises using multispectral cameras for most farmers.**

***Index Terms*—machine learning, deep learning, computer vision, encoder, decoder, NDVI**

I. Introduction

Over the years the importance of having an efficient and safe food source has increased dramatically due to several factors such as:

1) The growth of the population which, being constantly increasing (it is expected to reach 9.7 billion people by 2050), makes it necessary to have an efficient and effective production of food systems.
2) Environmental safety is a key factor that allows us to avoid food borne epidemics or food pollution
3) The environmental impact that comes from intensive agriculture that leads to deforestation, depletion of water resources and pollution, reasons why efficiency has become very important to reduce environmental impact and promote sustainability

In recent times a technology called UAV (Unmanned Aerial Vehicle) has become very important for monitoring the crop, its quality and condition. This technology uses a multispectral camera to detect most of the useful indicators on the crop and to be able to carry out the necessary evaluations. A multispectrum camera is a camera capable of performing detection with full spectral separation. The issue is that this cameras are very expensive. This experiment aims to find a way to carry out these detections through simple RGB images thus eliminating the need for these expensive cameras. The most important band detected with multispectral cameras is the NIR (Near Infrared). NIR is used to calculate the NDVI index together with the red band:

$$NDVI = \frac{NIR - Red}{NIR - Red} \tag{1}$$

The Normalized Difference Vegetation Index (NDVI) is a widely used tool for assessing and monitoring the health and quantity of vegetation in a particular area. NDVI is based on the analysis of the reflective properties of vegetation and is calculated using measurements of reflectance levels in the near-infrared and red spectral bands. NDVI is used to monitor crop growth status, identify areas with potential plant health issues, and guide decisions on crop management, such as irrigation and fertilizer application. To solve this Image to image transition problem we have used a "encoder-decoder" method. This architecture is made from two principal components:

1) Encoder: processes the input and "encodes" it into a lower-dimensional representation, capturing the salient information of the input.
2) Decoder: receives this encoded representation and "decrypts" it to generate the desired output.

Our encoder is based on EfficentNet, it is a type of neural network architecture that is mainly based on convolutional neural networks (CNNs). We chose EfficientNet because it is a neural architecture designed to obtain a good balance between precision and computational complexity, allowing to obtain high performances with a relatively low number of parameters. We did this experiment limiting the number of parameters to a maximum of 10 millions. The reason for this chose is that we needed to run this model on a limited hardware and in real time (i.e. on a dronve that is taking images). Our decoder is based on DeepLab. It use Semantic segmentation, a technique that assigns a class to each pixel in an image, enabling a better understanding of the image's structure and content. DeepLab is designed to process the features extracted by the encoder and produce a final high-resolution segmentation map. After the image has been processed by the encoder and features have been extracted, the decoder is responsible for restoring the original resolution of the image and generating a detailed segmentation map. This is achieved through the use of transposed convolution layers or upsampling operations, which increase the size of the image without losing significant information. We used RedEdge dataset, that is a subset from WeedMap. We performed a data cleaning operation on the dataset to remove all the image from the set that were all black (images that were outside the crop ground limit). This operation was done to avoid imbalance between classes. The goal of this experiment is therefore to be able to train a model that is capable of obtaining the NDVI value starting from an RGB image that has good quantitative and qualitative values, thus reducing the costs for obtaining vegetation index measurements.

II. Related works

A. NDVI/NDRE prediction from standard RGB aerial imagery using deep learning

We have used this paper to broadly understand the various aspects of the problem, starting from the various concepts such as NDVI and taking inspiration from the different approaches they have used [1]

B. PyTorch

We used PyTorch to build the model and to be able to calculate the different metrics [2]

C. Flops-counter

We used this library to count the number of operation performed by the model and the computational complexity [3]

D. Dataset WeedMap

We used the subset RedEdge from WeedMap to perform training and test operations on out model[4]

E. Wandb

Wandb is a tool for visualizing and tracking your machine learning experiments. It allows you to save everything you need to compare and reproduce models such as architecture, hyperparameters, weights, model predictions, GPU usage, git commits, and even datasets [5]

III. Materials and Methods

Given the goal to find a way to obtain NDVI image outputs from RGB inputs we decided to try an Image to image translation approach using an "encoder-decoder" method.
The encoder is based on EfficentNet, specifically we are using "efficientnet-b2", allowing us to achieve a good balance between accuracy and computational complexity, with 8.642.739 parameters.
The decoder is based on DeepLab.

A. Dataset and pre-processing

We used a subset called RedEdge from WeedMap containing 5 folders:

Fig. 1: RedEdge directory

Each one having the same subfolders structure. The one that we have used are:

1) 000/groundtruth: This are the groundtruth images for the set inside the current folder(000), we have used this images to find the different classes for each images and calculate the metrics for each class
2) 000/tile/NDVI: This are the annotations durin the train and the inference loop
3) 000/tile/RGB: These are the images that the model must start from in order to generate NDVI images

The training set consists of:

1) Folder 000
2) Folder 001
3) Folder 002
4) Folder 004

The test set consist of:

1) Folder 003

We have applied early-stopping method during the train loop, so we needed a validation set. The validation set is composed by 20% of the entire test set

While testing the model we noticed a strong imbalance between the classes, due to a high number of black images in the sets.
This images are the border of the analyzed ground:

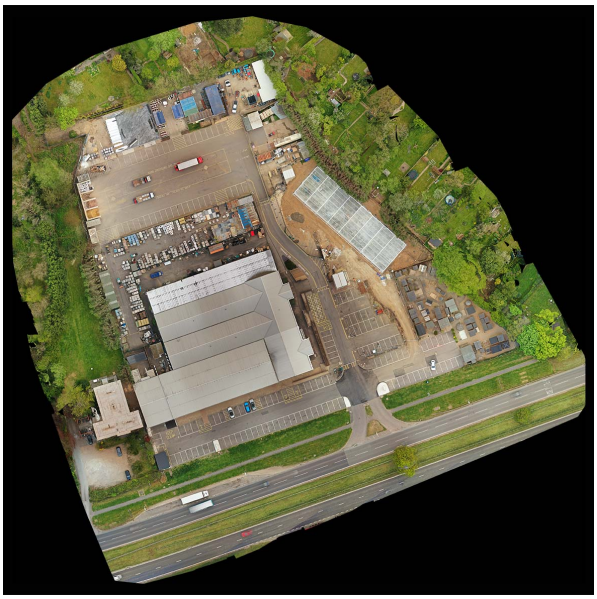

Fig. 2: Image example for black borders

When this image is divided into smaller images we can find completely black images that are part of the areas outside the terrain under analysis. So we used a scripts to remove all the total black images from the RGB folder, and the corresponding images from the NDVI folder and groundtruth folder using the script in the file "image_remover.ipynb"

The number of elements for each set are the following:

1) Number of elements in the train set: 362 (65%)
2) Number of elements in the validation set: 93 (16.7%)
3) Number of elements in the test set: 102 (18.3 %)

B. Model description

The DeepLabV3Plus model is a convolutional neural network (CNN) used for semantic image segmentation. It is an advanced version of the DeepLabV3 model, designed to improve segmentation accuracy.

The model is based on the encoder-decoder architecture, where the encoder extracts features from the input image, and the decoder reconstructs the segmented image from the extracted features.

it is important to emphasize that *encoder_weights='imagenet'* indicates that the model uses pre-trained weights of the encoder from training on ImageNet to extract image features. This can be advantageous because the pre-trained weights from ImageNet have learned to recognize general features of images and can help the model achieve better performance in the specific task it is used for.

The encoder of the model uses EfficientNet-B2 as a pre-trained base on ImageNet data. EfficientNet-B2 is a neural network architecture that combines various techniques, such as uniform scaling of coefficients, to achieve a good trade-off between accuracy and computational complexity.

The decoder of the DeepLabV3Plus model consists of several components. After the encoder, an Atrous Spatial Pyramid Pooling (ASPP) is applied to capture contextual information at different scales. ASPP uses dilated convolutions with different dilation rates to obtain a multi-scale view of the image.

Subsequently, upscaling operations are performed to increase the output resolution. Additionally, a residual connection is added between the encoder's output and the decoder's output to preserve and combine information at different scales.

Finally, the model applies a sigmoid activation function to the final output to obtain a normalized value between 0 and 1 for each pixel of the output image.

During training, the model utilizes the Adam optimization algorithm with a learning rate of 0.0002 to update the network's weights. The Mean Squared Error (MSE) loss function is used, which compares the model's output with the target segmentation labels to measure the prediction error.

The model is initialized and moved to the GPU (if available) using the PyTorch device to leverage hardware acceleration. The total number of model parameters is also calculated to monitor its complexity and size.

C. Training

In training the model on the images we perform the following points:

1) A loss criterion is defined using the *torch.nn.MSELoss()* function, which calculates the mean squared error between the model predictions and the target annotations.
2) An Adam optimizer is created using *torch.optim.Adam()* to update the model weights during training. A learning rate of 0.0002 is set.
3) Other variables are initialized, such as the number of training epochs, the best loss so far (initialized as infinity), patience (the number of epochs where the loss does not improve before terminating training), a delta threshold to determine significant improvement in loss, a counter to monitor patience, and a saved epoch for the best model.
4) The batch size is set to 30.
5) The training loop begins, iterating over the specified number of epochs.
6) Within each epoch, it iterates over a dataloader (train_loader) that provides the training images and their respective annotations.
7) The images and annotations are moved to the GPU (if available) using .to(device) to leverage GPU acceleration during training.
8) The optimizer is zeroed using *optimizer.zero_grad()* to clear the gradients accumulated from previous iterations.
9) The images are passed through the model (model(images)) to obtain the model's predictions.
10) The loss between the model predictions and annotations is calculated using the previously defined loss criterion $(loss = criterion(outputs, annotations))$.

Gradient backpropagation is performed using *loss.backward()* to compute gradients with respect to the model's parameters.

1) The optimizers are updated using *optimizer.step()* to update the model's weights based on the computed gradients.
2) Evaluation metrics such as mean squared error (MSE) and mean absolute error (MAE) are calculated and updated, both for the entire image and specific classes.
3) The training loop continues until all iterations in the dataloader are completed.
4) The average loss on the entire training set is computed using *.compute()* on the evaluation metrics.
5) The current epoch and the average MSE loss on the training set are displayed.
6) The loss values are logged using
7) wandb.log() to monitor the training progress using the Weights and Biases platform.
8) Model validation is performed using the *validate_model()* function on a validation dataloader *(validation_loader)*.
9) If the difference between the previous best loss and the current validation loss is greater than the delta threshold, the current model is saved as the best so far, and the patience counter is reset. Otherwise, the patience counter is incremented. If the patience counter exceeds the specified patience threshold, training is stopped, and the stopping epoch and the epoch of the saved best model are printed.

D. Testing

In testing the model on the images we perform the following points:

1) We load the previously saved best model weights using *torch.load('model_backup')* and sets the model to evaluation mode using *model.eval()*.
2) The code then calculates and prints the computational complexity (MACs) and the number of parameters of the model using the *get_model_complexity_info()* function. The model input size is specified as (3, 224, 224)

beacouse it has to be divisible by x16, else the model will not accept the input.
3) The test loop begins, iterating over the test loader that provides test images and annotations.
4) The images and annotations are moved to the GPU (if available) using *.to(device)* for efficient processing.
5) Forward pass is performed through the model using *model.predict(images)* to obtain the model's predictions.
6) Evaluation metrics such as mean squared error (MSE) and mean absolute error (MAE) are calculated and updated using the *update()* function for the entire image and specific classes.
7) The ground truth images are retrieved using the *get_groundtruth()* function from the groundtruth loader.
8) For each output, annotation, and ground truth image, a subset of selected pixels is obtained using the *get_selected_pixels_tensor()* function for different threshold values, this subsets represents the pixel for the analyzed class, with which we can calculate the MAE and MSE for that class.
9) The evaluation metrics for the subsets of selected pixels are computed and updated.

Then,

1) A wandb table is created to log the input images, ground truth images, and output images for visual comparison.
2) The computed errors are averaged and stored in variables: *mse_test, mae_test, mse_test_one, mse_test_two, mse_test_three, mae_test_one, mae_test_two, mae_test_three, mae_test_average, mse_test_average*.
3) The average errors for different classes and overall errors are logged using *wandb.summary*.
4) The trained model weights are saved locally using *wandb.save('model.pth')*.

Resuming, this test code loads the best saved model, performs inference on the test dataset, calculates evaluation metrics, logs the results using Weights e Biases, and saves the model weights locally.

IV. Experimental results

After testing the model the quantitative results are the following:

	macroaveraged	microaveraged	class_one	class_two	class_three
MAE	0.14632	0.07824	0.07141	0.16348	0.20407
MSE	0.03533	0.01196	0.00963	0.04194	0.05440

The microaveraged row refers to the average error between all the images during the inference. The macroaveraged row refers to the average error calculated between class errors.

As we can see from this values, the MAE microaveraged error and the MSE microaveraged error are both very small, taking into consideration also the limited train dataset and the limit on the complexity of the network, we can state that these results are very good.

We repeated the train and the test multiple times to assure consistency of the results, and the values are basically the same, here another table from another run:

	macroaveraged	microaveraged	class_one	class_two	class_three
MAE	0.14691	0.08390	0.07758	0.16286	0.20028
MSE	0.03535	0.01307	0.01086	0.04209	0.05310

Analyzing the error for each class we can see that the model have the best result with the class one and the worst from the class three probably depending on the distribution of each class in the training set. In particular the background class(class one) is very present in the training images, the weed class(class two)is less present than the background class and the crop class is less present than the weed class.

The qualitative results are the following:

1) Input image:

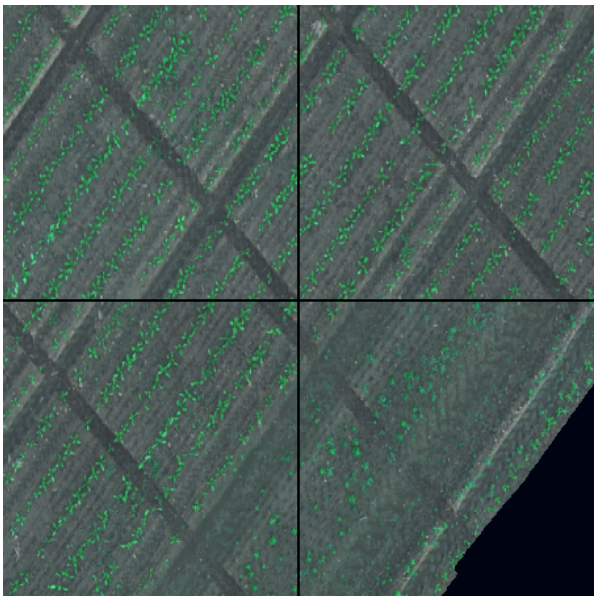

Fig. 3: Input image

2) Groundtruth:

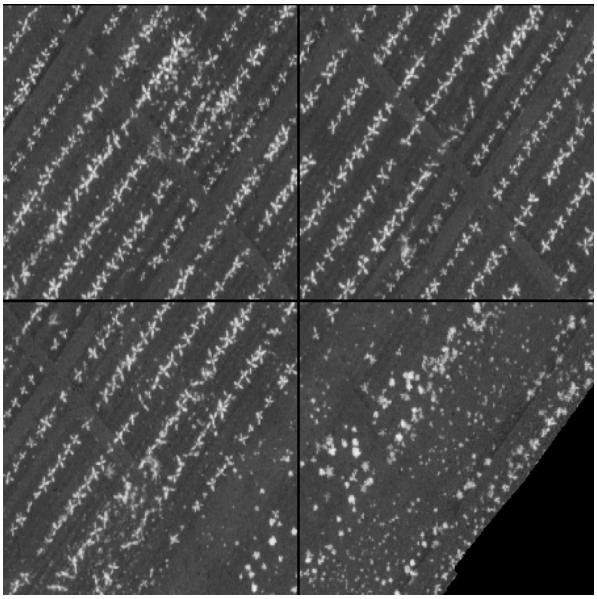

Fig. 4: Groundtruth image

3) Output image:

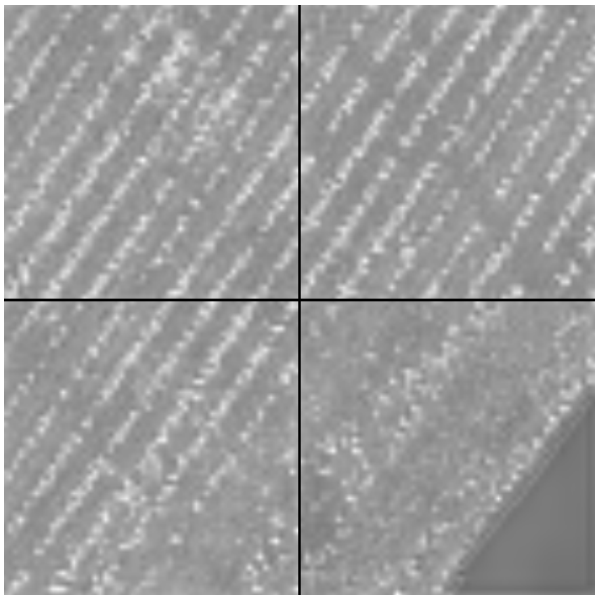

Fig. 5: Output image

We can also see from the graphical results that the model can successfully predict the background class of the images, but the crops and weed are a little blurry and lost some details from the original image.

V. Discussion

From the results we can say that the model, considering the limited dataset and the complexity limit, worked well. The only flaw we can notice is that it produces images with class 3 that are slightly blurry and have less detail, also as indicated by the mae_class_three metric. We think this problem is caused by the fact that for each image the dominant class is the background one, limiting model learning to the details of classes two and tree. A solution can be expanding the dataset with more synthetic data utilizing an offline augmentation or taking more data from the soils under analysis. The encoder EfficentNet, in particular "efficentnet-b2", allowed us to have a small number of parameters but with very good performance.

VI. Conclusion

In conclusion we can say that using this method is possible to obtain NDVI images from RGB ones using a model trained with RGB images and NDVI images.
A possible solution to improve the resulting images is having access to a better machine that can support a more complex model, increasing the complexity of the encoder allowing a more precise output.
This method can be paired with a transmission system from the drone that is taking the frames to the machine that have to compute the operations.
The only problem is that this solution includes spending more money on the machine that has to do the operations when the focus of this study is spending less money to avoid multispectral cameras.
Online computing should be analyzed as a possible solution for the economic problem.

References

[1] C. Davidson, V. Jaganathan, A. N. Sivakumar, J. M. P. Czarnecki, and G. Chowdhary, "Ndvi/ndre prediction from standard rgb aerial imagery using deep learning," *Computers and Electronics in Agriculture*, vol. 203, p. 107396, 2022. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0168169922007049

[2] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems 32*. Curran Associates, Inc., 2019, pp. 8024–8035. [Online]. Available: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

[3] V. Sovrasov. (2018-2023) ptflops: a flops counting tool for neural networks in pytorch framework. [Online]. Available: https://github.com/sovrasov/flops-counter.pytorch

[4] P. L. F. L. J. N. C. S. A. W. I. Sa, M. Popovic; R. Khanna; Z. Chen and R. Siegwart, "Weedmap: A large-scale semantic weed mapping framework using aerial multispectral imaging and deep neural network for precision farming," *MDPI Remote Sensing*, vol. 10, no. 9, Aug 2018.

[5] L. Biewald, "Experiment tracking with weights and biases," 2020, software available from wandb.com. [Online]. Available: https://www.wandb.com/