Package ‘imageseg’

May 30, 2022

Type Package

Title Deep Learning Models for Image Segmentation

Version 0.5.0

Maintainer Juergen Niedballa `<niedballa@izw-berlin.de>`

Description A general-purpose workflow for image segmentation using TensorFlow models based on the U-Net architecture by Ronneberger et al. (2015) <arXiv:1505.04597> and the U-Net++ architecture by Zhou et al. (2018) <arXiv:1807.10165>. We provide pre-trained models for assessing canopy density and understory vegetation density from vegetation photos. In addition, the package provides a workflow for easily creating model input and model architectures for general-purpose image segmentation based on grayscale or color images, both for binary and multi-class image segmentation.

License MIT + file LICENSE

BugReports `https://github.com/EcoDynIZW/imageseg/issues`

Encoding UTF-8

Imports grDevices, keras, magick, magrittr, methods, purrr, stats, tibble, foreach, parallel, doParallel, dplyr

Suggests R.rsp, testthat

VignetteBuilder R.rsp

RoxygenNote 7.1.2

NeedsCompilation no

Author Juergen Niedballa [aut, cre] (<`https://orcid.org/0000-0002-9187-2116`>), Jan Axtner [aut] (<`https://orcid.org/0000-0003-1269-5586`>), Leibniz Institute for Zoo and Wildlife Research [cph]

Repository CRAN

Date/Publication 2022-05-29 22:40:12 UTC

R topics documented:

`imageseg-package` *Overview of the imageseg package*

Description

This package provides a streamlined workflow for image segmentation using deep learning models based on the U-Net architecture by Ronneberger (2015) and the U-Net++ architecture by Zhou et al. (2018). Image segmentation is the labelling of each pixel in a images with class labels. Models are convolutional neural networks implemented in **keras** using a TensorFlow backend. The workflow supports grayscale and color images as input, and binary or multi-class output.

We provide pre-trained models for two forest structural metrics: canopy density and understory vegetation density. These trained models were trained with large and diverse training data sets, allowing for robust inferences. The package workflow is implemented in a few function, allowing for simple predictions on your own images without specialist knowledge of convolutional neural networks.

If you have training data available, you can also create and train your own models, or continue model training on the pre-trained models.

The workflow implemented here can also be used for other image segmentation tasks, e.g. in the cell biology or for medical images. We provide two examples in the package vignette (bacteria detection in darkfield microscopy from color images, breast cancer detection in grayscale ultrasound images).

Functions for model predictions

The following functions are used to perform image segmentation on your images. They resize images, load them into R, convert them to model input, load the model and perform predictions. The functions are given in the order they would typically be run. See the vignette for complete examples.

`findValidRegion`	Subset image to valid (informative) region (optional)
`resizeImages`	Resize and save images
`loadImages`	Load image files with magick
`imagesToKerasInput`	Convert magick images to array for keras
`loadModel`	Load TensorFlow model from hdf5 file
`imageSegmentation`	Model predictions from images based on TensorFlow model

Functions for model training

This function assist in creating models in keras based on the U-Net architecture. See the vignette for complete examples.

`dataAugmentation`	Rotating and mirroring images, and modulating colors
`u_net`	Create a U-Net architecture
`u_net_plusplus`	Create a U-Net++ architecture

Download pre-trained models for forest structural metrics

Links to both pre-trained models (canopy and understory), example classifications and all training data used can be found in the GitHub readme under:

`https://github.com/EcoDynIZW/imageseg`

Vignette

The package contains a pdf vignette demonstrating the workflow for predictions and model training using various examples. It covers installation and setup, model predictions and training the forest structural models, and two more general applications of image segmentation (multi-class image segmentation of RGB microscopy images, and single-class image segmentation of grayscale ultrasound breast scan images). See `browseVignettes(package = "imageseg")`.

Author(s)

Juergen Niedballa, Jan Axtner

Maintainer: Juergen Niedballa <niedballa@izw-berlin.de>

References

Ronneberger O., Fischer P., Brox T. (2015) U-Net: Convolutional Networks for Biomedical Image Segmentation. In: Navab N., Hornegger J., Wells W., Frangi A. (eds) Medical Image Computing and Computer-Assisted Intervention – MICCAI 2015. MICCAI 2015. Lecture Notes in Computer Science, vol 9351. Springer, Cham. doi: 10.1007/9783319245744_28

Zhou, Z., Rahman Siddiquee, M. M., Tajbakhsh, N., & Liang, J. (2018). Unet++: A nested u-net architecture for medical image segmentation. In Deep learning in medical image analysis and multimodal learning for clinical decision support (pp. 3-11). Springer, Cham. doi: 10.48550/arXiv.1807.10165

See Also

keras tensorflow magick

dataAugmentation — *Data augmentation: rotating and mirroring images, and adjusting colors*

Description

Rotate and/or mirror images to create augmented training data. Optionally, apply a random shift in brightness, saturation and hue to a certain percentage of the images

Usage

```
dataAugmentation(
  images,
  subset = NULL,
  rotation_angles = 0,
  flip = FALSE,
  flop = FALSE,
  brightness_shift_lim = c(90, 110),
  saturation_shift_lim = c(95, 105),
  hue_shift_lim = c(80, 120),
  fraction_random_BSH = 0
)
```

Arguments

`images`	list. Output of `loadImages`. List with two items ($info: data frame with information about images, $img: tibble containing magick images)
`subset`	integer. Indices of images to process. Can be useful for only processing subsets of images (e.g. training images, not test/validation images).
`rotation_angles`	integer. Angles in which to rotate images using `image_rotate`)?
`flip`	logical. mirror along horizontal axis (turn images upside-down using `image_flip`)?
`flop`	mirror along vertical axis (switch left and right) using `image_flop`)?
`brightness_shift_lim`	numeric. Lower and upper limits for argument `brightness` in `image_modulate`
`saturation_shift_lim`	numeric. Lower and upper limits for argument `saturation` in `image_modulate`
`hue_shift_lim`	numeric. Lower and upper limits for argument hue in `image_modulate`
`fraction_random_BSH`	numeric. Fraction of images to apply random brightness / saturation / hue shifts to (between 0 and 1)

Details

For creating training data for canopy, rotation and mirroring in both axes is appropriate. For understory vegetation density, only flop images (don't flip), and don't apply a hue shift since recognition of the orange flysheet is color-critical.

Value

A list with 2 elements: $info, a data frame with information about the images, and $img, a tibble with magick images

Examples

```
# Example 1: Canopy
wd_images_can <- system.file("images/canopy/resized",
                             package = "imageseg")

images_can <- loadImages(imageDir = wd_images_can)

images_can_aug <- dataAugmentation(images = images_can,
                                   rotation_angles = c(0, 90, 180, 270),
                                   flip = TRUE,
                                   flop = TRUE)
images_can_aug
```

`findValidRegion` *Subset image to valid (informative) region*

Description

Load images, find and crop valid (informative) region. This function removes black borders from images, and is suitable for restricting hemispherical (fisheye) images to the actual informative region in the image center.

Usage

```
findValidRegion(image, fileName, threshold = 0.1)
```

Arguments

`image`	magick image
`fileName`	file name of image to load
`threshold`	numeric. Minimum range (max - min) of grayscale values for rows/columns to be included in the output image.

Details

Images are converted to grayscale according to the formula in Li et al. (2020). L = 0.30R + 0.59G + 0.11B We use a default threshold of 10, but it can be adjusted freely.

This function can optionally be called inside `resizeImages` to crop each image to the informative region before resizing to the dimensions expected by the models. It is not recommended though since it may return different crop masks for images and their masks.

Value

A list with 3 items:

* img: the magick image. * geometry_area: a geometry string that can be used as argument `geometry` in `image_crop`. * geometry_area_df: a data frame containing the information from geometry_area (can be useful for finding consensus are to crop from many images)

Warning

Depending on the quality of the photographic equipment used, supposedly dark regions of images may be affected by stray light and light diffraction. This will be especially prevalend when using fisheye adapters, e.g. for smartphones. In such cases, the function will not provide reliable output. Adjusting 'threshold' may help to a degree, but needs to be set on a case-to-case basis for individual images. In such cases it might be easier to instead use e.g. GIMP to measure out the valid image region.

References

Li, Kexin, et al. "A New Method for Forest Canopy Hemispherical Photography Segmentation Based on Deep Learning." Forests 11.12 (2020): 1366.

Examples

```
wd_images_can <- system.file("images/canopy/raw",
                              package = "imageseg")
lf <- list.files(wd_images_can, full.names = TRUE)
img <- findValidRegion(fileName = lf[1])
img

# finding consensus among multiple images
## Not run:
wd_with_many_images <- "..."
lf <- list.files(wd_with_many_images)
test <- lapply(lf, findValidRegion)
# combine geometry_area_df from many images
geometry_areas_df <- do.call(rbind, lapply(test, FUN = function(x) x$geometry_area_df))
# summary to decide on suitable values
summary(geometry_areas_df)

## End(Not run)
```

`imageSegmentation` *Model predictions from images based on TensorFlow model*

Description

This function uses a pre-trained TensorFlow model to create predictions from input data. It was mainly designed to predict canopy cover and understory vegetation density from forest habitat photographs using the pre-trained models we provide.

Usage

```
imageSegmentation(
  model,
  x,
  dirOutput,
  dirExamples,
  subsetArea,
  threshold = 0.5,
  returnInput = FALSE
)
```

Arguments

`model`	trained model to use in predictions
`x`	array of images as model input (can be created with `imagesToKerasInput`)
`dirOutput`	character. Directory to save output images to (optional)
`dirExamples`	character. Directory to save example classification to (optional)
`subsetArea`	If "circle", vegetation density will be calculated for a circular area in the center of the predicted images. Can also be a number between 0 and 1 (to scale the circle relative to the image dimensions), or a matrix of 0 and 1 in the same dimensions as images in x.
`threshold`	numeric value at which to split binary predictions. Can be useful to only return high-confidence pixels in predictions. It is not relevant for multi-class predictions.
`returnInput`	logical. If `dirOutput` is defined, save input images alongside output?

Details

By default, vegetation density will be calculated across the entire input images. If canopy images are hemispherical and have black areas in the corner that should be ignored, set `subsetArea` to "circle". If the relevant section of the images is smaller than the image frame, give a number between 0 and 1 (indicating how big the circle is, relative to the image dimensions). Alternatively, provide a custom matrix of 0 and 1 in the same dimensions as the input images in x. 0 indicates areas to ignore in the vegetation calculations, 1 is included. `subsetArea = "circle"` only works if input images in x are square.

The canopy density models predicts sky and the understory vegetation density model predicts the red flysheet The percentage of these is equivalent to openness (canopy openness or understory openness). This value is in the column "predicted".

The interpretation of openness depends on context:

- Canopy Cover images: openness = Gap Fraction and Fraction Soil
- Hemispherical canopy images: openness = Canopy openness and site openness (in flat terrain)

See e.g. Gonsamo et al. (2013) for more details.

Generally speaking, "predicted" is the percentage of the image that is 1 in the binary prediction.

The column "not_predicted" is the opposite (1-predicted). It is thus equivalent to vegetation density in the two vegetation models.

Depending on the context, "not_predicted" can for example mean: canopy cover, canopy closure, understory vegetation density. In canopy cover images, the vegetation density corresponds to canopy cover. In hemispherical images, vegetation density corresponds to canopy closure.

Value

A list. The type and number of list items depends on the classification. For binary classifications (1 prediction class), the following list items are returned:

- image (input images)
- prediction (model prediction)
- prediction_binary (binary prediction, only 0 or 1)
- examples (images with their image segmentation results)
- summary (data frame with fraction of image predicted)
- mask (an image showing the area for which summary statistics were calculated (in white, only if `subsetArea` is defined)

in multi-class models:

- image (input images)
- prediction_most_likely (the class with the highest probability, coded in grayscale)
- class1 - classX: for each class, the predicted probabilities
- examples (images with their image segmentation results)
- summary (data frame with fraction of image covered by vegetation (black)).
- mask (an image showing the area for which summary statistics were calculated (in white, only if `subsetArea` is defined)

Examples

```
## Not run:

# Example 1: Canopy
wd_images <- system.file("images/canopy/resized",
                         package = "imageseg")
images <- loadImages(imageDir = wd_images)
x <- imagesToKerasInput(images)

wd_model_can <- "C:/Path/To/Model"      # change this
filename_model_can <- "imageseg_canopy_model.hdf5"
model_can <- loadModel(file.path(wd_model_can, filename_model_can))


results_can <- imageSegmentation(model = model_can,
                                 x = x)
```

```
results_can$image
results_can$prediction
results_can$prediction_binary
results_can$vegetation


# Example 2: Understory
wd_images_us <- system.file("images/understory/resized",
                             package = "imageseg")
images_us <- loadImages(imageDir = wd_images_us)
x <- imagesToKerasInput(images_us)

# note, here we just specify the complete path, not separate by directory and file name as above
model_file_us <- "C:/Path/To/Model/imageseg_understory_model.hdf5"
model_us <- loadModel(model_file_us)

results_us <- imageSegmentation(model = model_us,
                                x = x)
results_us$image
results_us$prediction
results_us$prediction_binary
results_us$vegetation


## End(Not run)
```

imagesToKerasInput *Convert magick images in tibble to array for keras*

Description

This function converts a tibble of images into input for TensorFlow models in keras. Specifically, images are converted to 4D arrays (image, height, width, channels). It can process color images and masks (for model training).

Usage

```
imagesToKerasInput(
  images,
  subset = NULL,
  type = NULL,
  grayscale = NULL,
  n_class = 1,
  max = 1
)
```

Arguments

`images`	list. Output of `loadImages` or `dataAugmentation`. List with two items ($info: data frame with information about images, $img: tibble containing magick images)
`subset`	integer. Indices of images to process. Can be useful for only processing subsets of images (e.g. training images, not test/validation images).
`type`	character. Can be "image" or "mask" and will set color channels of array accordingly (optional).
`grayscale`	logical. Defines color channels of images: 1 if codeTRUE, 3 if `FALSE`.
`n_class`	For mask images, how many classes do they contain? (note that binary classifications like the canopy model have one class only)
`max`	integer. Maximum value of output color values range. Can be 1 or 255.

Details

The function will try to infer the colorspace from images, but if the colorspaces are inconsistent one has to define 'colorspace'. `type = "image"` can have either colorspace "sRGB" or "Gray", masks are always "Gray". color images have three color channels in the arrays, grayscale images have one color channel. `n_class` is only relevant for masks. It determines the dimensions of the output. The default 1 is the (binary case). Higher values are for multi-class cases. If n_class is 2 or larger, keras::to_categorical() will be applied, and the `u_net` model will use softmax instead of sigmoid activation in the final layer.

By default, color value range will be 0-1. Alternatively, set `max` to 255 to create color value range 0-255 (e.g. to create input for Habitat-Net models).

Value

An array with the following dimensions: image, height, width, channels

Examples

```
# Example 1: Canopy

# images
wd_images_can <- system.file("images/canopy/resized",
                             package = "imageseg")
images_can <- loadImages(imageDir = wd_images_can)
x <- imagesToKerasInput(images_can)
str(x)   # a 4D array with an attribute data frame

# masks

wd_mask_can <- system.file("images/canopy/masks",
                             package = "imageseg")
masks_can <- loadImages(imageDir = wd_mask_can)
y <- imagesToKerasInput(masks_can, type = "mask", grayscale = TRUE)
str(y)   # a 4D array with an attribute data frame
```

```
# Example 2: Understory
wd_images_us <- system.file("images/understory/resized",
                             package = "imageseg")
images_us <- loadImages(imageDir = wd_images_us)
x <- imagesToKerasInput(images_us)
str(x)   # a 4D array, with an attribute data frame
```

`loadImages` *Load image files with magick*

Description

This function loads images from disk to R, where one can inspect them and then pass them on to `imagesToKerasInput`, which converts them to input for keras (TensorFlow) models.

Usage

```
loadImages(
  imageDir,
  fileNames,
  pattern,
  patternInclude = TRUE,
  imageFormats = c("JPG|TIF|PNG|JPEG|TIFF")
)
```

Arguments

`imageDir`	character. Directory containing the images to load
`fileNames`	character. File names to load (they will still be filtered by `pattern`, if defined)
`pattern`	character. Pattern to search in file names
`patternInclude`	logical. Include images with pattern in file names (TRUE) or exclude (FALSE)
`imageFormats`	character. Image file formats to read.

Value

A list with 2 slots: "img" contains images as a tibble, "info" contains basic information about the images.

Examples

```
# Example 1: Canopy
wd_images_can <- system.file("images/canopy/resized",
                             package = "imageseg")

images_can <- loadImages(imageDir = wd_images_can)
images_can
```

```
# Example 2: Understory
wd_images_us <- system.file("images/understory/resized",
                             package = "imageseg")
images_us <- loadImages(imageDir = wd_images_us)
images_us
```

---

loadModel *Load TensorFlow model from hdf5 file*

Description

Load TensorFlow model from hdf5 file

Usage

```
loadModel(modelFile, restoreCustomObjects = TRUE)
```

Arguments

`modelFile`	character. File name of the .hdf5 model file to load
`restoreCustomObjects`	logical. Restore custom objects (loss function & dice coefficient) used in training of habitat models

Details

Loads a trained TensorFlow model from a hdf5 file, and (optionally) restores custom objects.

Value

keras model

Examples

```
## Not run:
# Canopy model
wd_model_can <- "C:/Path/To/Model"     # change this
filename_model_can <- "imageseg_canopy_model.hdf5"
model_can <- loadModel(file.path(wd_model_can, filename_model_can))

# Understory model
# note, here we just specify the complete path, not separate by directory and file name as above
model_file_us <- "C:/Path/To/Model/imageseg_understory_model.hdf5"
model_us <- loadModel(model_file_us)

## End(Not run)
```

resizeImages *Resize and save images*

Description

Resize and save images

Usage

```
resizeImages(
  imageDir,
  fileNames,
  pattern,
  patternInclude = TRUE,
  type,
  dimensions,
  validRegion,
  preserveAspect = TRUE,
  filter = NULL,
  colorspace,
  binary,
  gravity = "Center",
  imageFormats = c("JPG|TIF|PNG|JPEG|TIFF"),
  outDir,
  cores = 1,
  compression = "Lossless"
)
```

Arguments

`imageDir`	Character. Directory containing raw images
`fileNames`	character. File names to load (they will still be filtered by `pattern`, if defined)
`pattern`	character. Pattern to search in file names
`patternInclude`	logical. Include images with pattern in file names (TRUE) or exclude (FALSE)
`type`	character. "canopy" or "understory". Will set image dimensions accordingly to predefined c(256, 256) or c(160, 256), respectively (optional). Alternatively, use `dimensions`.
`dimensions`	integer. image dimensions provides as c(width, height) in pixels. If specified, overrides `type`
`validRegion`	character. If defined, use string as argument `geometry` in `image_crop` (output of `geometry_area`), which will crop all images to the same region before resizing (optional). If undefined, don't crop.
`preserveAspect`	logical. If TRUE, images will be cropped to aspect ratio of output before resizing (thus preserving original aspect ratio, but losing parts of the image). If FALSE, images will be simply resized from their input size to the desired output (not preserving aspect ratio).

`filter`	character. Resampling filter. Passed to argument `filter` in `image_resize`. See `magick::filter_types()` for available options. Default is LanczosFilter.
`colorspace`	character. If defined, image will be converted to the requested colorspace. If undefined, colorspace will remain unchanged. Must be a valid argument to `magick::colorspace_types()`. In practice, only "sRGB" and "Gray" will be relevant.
`binary`	logical. If colorspace is "Gray", make the output binary?
`gravity`	if preserveAspect = TRUE and images need to be cropped, the `gravity` argument to use in `image_crop`.
`imageFormats`	character. Image file formats to read.
`outDir`	character. Directory to save resized images in.
`cores`	integer. Number of cores to use for parallel processing
`compression`	character. Compression type to use in `image_write`. See `compress_types`. By default, "Lossless" for grayscale images, "Undefined" for color images.

Details

Resizing is done by `image_resize` and will ensure that the resized images have exactly the desired dimensions.

If `preserveAspect = TRUE`, input images will first be cropped to the maximum area with the aspect ratio of the desired output (1:1 (square) for `type = "canopy"`, 5:8 for `type = "understory"`), by default in the center of the input image (argument `gravity`). This will usually lead to the loss of parts of the image, but the remaining part of the image is not deformed compared to the original. Alternatively, if `preserveAspect = FALSE`, input images will be resized to the requested dimensions without cropping (thus no loss of part of the image), but the aspect ratio changes. If aspect ratio changes too strongly it may negatively affect model performance.

Resizing is done using "!" in the geometry syntax. See `geometry` for details.

compression = "Lossless" is used to ensure no compression artefacts in saved images (which would for example introduce grayscale values in black/white images). If small file sizes are important, you can change it to save compressed images.

Value

No R output, only resized images are saved on disk

Examples

```
# Example 1: Canopy
wd_can <- system.file("images/canopy/raw",
                      package = "imageseg")

wd_out_can <- file.path(tempdir(), "canopy", "resized")
resizeImages(imageDir = wd_can,
             type = "canopy",
             outDir = wd_out_can)

filename_resized <- list.files(wd_out_can, full.names = TRUE)
```

```
# check output
img_can <- magick::image_read(filename_resized)
img_can

# Example 2: Understory
wd_us <- system.file("images/understory/raw",
                      package = "imageseg")
wd_out_us <- file.path(tempdir(), "understory", "resized")

# note, these are png images
resizeImages(imageDir = wd_us,
             type = "understory",
             outDir = wd_out_us)

filename_resized <- list.files(wd_out_us, full.names = TRUE)

# check output
img_us <- magick::image_read(filename_resized)
img_us
```

u_net *Create a U-Net architecture*

Description

Create a U-Net architecture

Usage

```
u_net(
  net_h,
  net_w,
  grayscale = FALSE,
  layers_per_block = 2,
  blocks = 4,
  n_class = 1,
  filters = 16,
  dropout = 0,
  batch_normalization = TRUE,
  kernel_initializer = "he_normal"
)
```

Arguments

net_h	Input layer height.
net_w	Input layer width.

`grayscale`	Defines input layer color channels - 1 if 'TRUE', 3 if 'FALSE'.
`layers_per_block`	Number of convolutional layers per block (can be 2 or 3)
`blocks`	Number of blocks in the model.
`n_class`	Number of classes.
`filters`	Integer, the dimensionality of the output space (i.e. the number of output filters in the convolution).
`dropout`	Dropout rate (between 0 and 1).
`batch_normalization`	Should batch normalization be used in the block?
`kernel_initializer`	Initializer for the kernel weights matrix.

Details

This function creates a U-Net model architecture according to user input. It allows flexibility regarding input, output and the hidden layers. See the package vignette for examples.

The function was adapted and slightly modified from the `u_net()` function in the platypus package (`https://github.com/maju116/platypus/blob/master/R/u_net.R`).

Differences compared to platypus implementation:

- added argument: layers_per_block (can be 2 or 3)
- kernel size in layer_conv_2d_transpose is 2, not 3.
- dropout layers are only included if user specifies dropout > 0
- n_class = 1 by default (sufficient for binary classification used for vegetation model, e.g. sky or not sky)
- automatic choice of activation of output layer: "sigmoid" if n_class = 1, otherwise "softmax"
- allows non-square input images (e.g. 160x256 used in understory vegetation density model)

Value

U-Net model.

A keras model as returned by `keras_model`

Examples

```
## Not run:
# U-Net model for 256x256 pixel RGB input images with a single output class
# this model was used for canopy density

model <- u_net(net_h = 256,
net_w = 256,
grayscale = FALSE,
filters = 32,
blocks = 4,
layers_per_block = 2
)

# several arguments above were not necessary because they were kept at their default.
# Below is the same model, but shorter:

model <- u_net(net_h = 256,
net_w = 256,
filters = 32
)

model


## End(Not run)
```

u_net_plusplus *Create a U-Net++ architecture*

Description

Create a U-Net++ architecture.

Usage

```
u_net_plusplus(
  net_h,
  net_w,
  grayscale = FALSE,
  blocks = 4,
  n_class = 1,
  filters = 16,
  kernel_initializer = "he_normal"
)
```

Arguments

`net_h`	Input layer height.
`net_w`	Input layer width.
`grayscale`	Defines input layer color channels - 1 if ‘TRUE‘, 3 if ‘FALSE‘.
`blocks`	Number of blocks in the model.
`n_class`	Number of classes.
`filters`	Integer, the dimensionality of the output space (i.e. the number of output filters in the convolution).
`kernel_initializer`	Initializer for the kernel weights matrix.

Details

The function was ported to R from Python code in https://github.com/albertsokol/pneumothorax-detection-unet/blob/master/models.py. For more details, see https://github.com/MrGiovanni/UNetPlusPlus.

Value

U-Net++ model.

Examples

```
## Not run:
# U-Net++ model for 256x256 pixel RGB input images with a single output class

model <- u_net_plusplus(net_h = 256,
net_w = 256,
filters = 32,
blocks = 3
)

model


## End(Not run)
```

Index