Package: rplatypus (via r-universe)

November 5, 2026

Type Package

Title Computer Vision for Medical Imaging

Version 0.2.1

Description Semantic segmentation and object detection for medical images, from a folder of pictures to a trained model and a figure, in a handful of lines or one configuration file. Architectures, losses and training run on 'PyTorch' through 'pyplatypus', a 'Python' package which is installed and isolated automatically; nothing has to be set up by hand. R keeps the parts that make an analysis fast - reading and combining masks, drawing boxes, plotting, comparing models.

License MIT + file LICENSE

URL https://github.com/maju116/rplatypus, https://maju116.github.io/rplatypus/

BugReports https://github.com/maju116/rplatypus/issues

SystemRequirements Python (>= 3.10). Not a prerequisite the user installs: the package declares what it needs through 'reticulate' and 'uv' provisions an isolated environment on first use, leaving any system Python untouched.

Encoding UTF-8

Depends R (>= 4.1)

Imports ggplot2, grDevices, parallel, rlang, reticulate (>= 1.41), tools, utils

Suggests callr, jsonlite, knitr, pkgload, png, RNifti, rmarkdown, testthat (>= 3.0.0), withr, yaml

Config/testthat/edition 3

VignetteBuilder knitr

Config/roxygen2/version 8.1.0

NeedsCompilation no

Author Michal Maj [aut, cre]

Maintainer Michal Maj <michalmaj116@gmail.com>

Config/pak/sysreqs python3

Repository https://cran.r-universe.dev

Date/Publication 2026-11-04 22:00:02 UTC

RemoteUrl https://github.com/cran/rplatypus

RemoteRef HEAD

RemoteSha 3bd95253d551ac3f849d3f4fba10126a719aca0f

Contents

as.list.platypus_spec . . . 3
augmentation_step . . . 4
available_components . . . 4
available_transforms . . . 6
available_weights . . . 8
callback_swa . . . 9
callbacks . . . 10
classes_to_colours . . . 12
colormaps . . . 12
colours_to_classes . . . 13
ct_windows . . . 14
decodings . . . 14
detection_anchors . . . 15
detection_crops . . . 16
detection_data . . . 18
drawing_style . . . 20
encoders . . . 21
evaluate . . . 23
evaluate_cases . . . 24
evaluate_classes . . . 25
evaluate_images . . . 26
export_weights . . . 27
inspect_masks . . . 28
instance_data . . . 29
loss_boundary . . . 31
losses . . . 32
mask_coverage . . . 34
mask_volume . . . 35
metrics . . . 36
models . . . 37
optimizers . . . 43
overlay_agreement . . . 46
overlay_mask . . . 47
platypus_conditions . . . 48
platypus_device . . . 50
platypus_fit . . . 50

platypus_spec . . . 51
platypus_status . . . 54
platypus_use_torch . . . 54
plot.platypus_fit . . . 55
plot_anchors . . . 56
plot_boxes . . . 57
plot_masks . . . 58
predict.platypus_fit . . . 60
read_images . . . 62
read_masks . . . 62
read_record . . . 63
save_masks . . . 65
save_spec . . . 66
save_volumes . . . 67
segmentation_data . . . 68
separations . . . 71
series_report . . . 72
split_dataset . . . 73
split_path . . . 75
training_history . . . 76
unite_masks . . . 77
volume_info . . . 77
yolo3 . . . 78

Index **82**

`as.list.platypus_spec` *Turn a specification back into plain R data*

Description

Useful for looking at what the engine made of your arguments, and for writing it out.

Usage

```
## S3 method for class 'platypus_spec'
as.list(x, ...)
```

Arguments

`x`	A `platypus_spec`.
`...`	Unused.

Value

A nested list.

Examples

```
## Not run:
str(as.list(spec))

# The same thing a YAML file would hold, which is the point: a configuration file
# and this code describe the same run. `yaml::write_yaml(as.list(spec), "run.yaml")`
# produces a file the Python engine reads directly.

## End(Not run)
```

`augmentation_step` *An augmentation step*

Description

Names come from the installed 'albumentations', so whatever it offers is available and a misspelling is caught when the specification is built, with a suggestion.

Usage

```
augmentation_step(name, ...)
```

Arguments

`name`	The transform, for instance `"HorizontalFlip"`.
`...`	Its parameters, for instance `p = 0.5`.

Value

An augmentation specification.

Examples

```
augmentation_step("HorizontalFlip", p = 0.5)
```

`available_components` *What this package can do, asked of the package*

Description

Six listings beside `available_transforms()` and `available_weights()`: the losses, the metrics, the architectures, the optimisers, the callbacks and the tasks. Three columns - the name you write in a specification, the arguments you may tune, and what it is for.

Usage

```
available_losses(task = "semantic_segmentation")

available_metrics(task = "semantic_segmentation")

available_models(task = "semantic_segmentation")

available_optimizers()

available_callbacks()

available_tasks()
```

Arguments

task	`"semantic_segmentation"`, `"instance_segmentation"` or `"object_detection"`. Which components a specification may name depends on what it is asking for, and for losses it genuinely differs: the three regression losses are listed only at the instance task, which is the only one with a target for them. No count is written here on purpose - the counts are asserted in the test suite against the pinned engine, which is where a number nobody can re-measure from a help page belongs. This parameter listed two tasks until the pin moved to `0.8.0a22`; the third arrived in the engine's `0.8.0a20`.

Details

Worth having because the alternative is leaving R for a website. The number of named components grows with every release, and these are read from the engine's own schema, so a loss added there appears here with nothing in this package edited. Nothing is maintained by hand; a list maintained by hand is a list that will be wrong.

`description` is the component's summary line. The rest of what the engine knows about it - including the measurements, where there are any - is on the configuration reference of the engine's own site, which is generated from the same text.

Value

A data frame with `name`, `arguments` and `description`. `arguments` is a single string per row, comma-separated, because a column of vectors prints as something nobody can read; `strsplit(x, ", ")` takes it apart.

Detection chooses fewer things, and says so

A detection specification chooses no loss component and no metrics at all. Its objective is fixed, with `box_loss` on the model picking the box term, and every detection run reports the same set - `map_50`, `map_50_95`, `mean_matched_iou` - which comes out of the run rather than going into the specification.

So `available_losses("object_detection")` and `available_metrics("object_detection")` raise a condition whose message says that, rather than returning a table with no rows. An empty table reads as "there are none", which is a different and wrong answer.

See Also

`available_transforms()` and `available_weights()`, the two that came first, and `platypus_spec()`, where these names are written.

Examples

```
## Not run:
available_losses()
#>            name                    arguments
#> 1      boundary               region, alpha
#> 2           cce             label_smoothing
#> 3      cce_dice          cce_weight, smooth
#> ...

# Which metrics take an argument at all, and which do not.
available_metrics()[, c("name", "arguments")]

# The entry point, for somebody who has read nothing yet.
available_tasks()

# Detection says what it has instead of a loss to choose.
try(available_losses("object_detection"))

## End(Not run)
```

`available_transforms` *Which augmentations are available*

Description

The transforms the installed 'albumentations' offers, and - for volumes - the ones it can actually apply to them.

Usage

```
available_transforms(rank = 2, pattern = NULL)
```

Arguments

`rank`	2 for images, 3 for volumes.
`pattern`	Optional regular expression to filter the names, for browsing: `"Flip"`, `"3D$"`, `"Elastic|Grid"`.

Details

Worth asking rather than guessing. Support for volumes is uneven: most geometric transforms work and several intensity ones raise from inside the library, so `rank = 3` returns a shorter list than `rank = 2`. A transform outside it is refused when the specification is built, by name, rather than failing an hour into training.

The number of names at `rank = 2` is a property of the installed 'albumentations'. **The number at `rank = 3` is not, and is deliberately not written down here.** The engine finds that list by offering each transform a probe volume, and some of what it refuses fails for the probe's reasons rather than the library's: a group of transforms want three channels where the probe has one, and another group want a target - reference image, bounding box, metadata of their own - that a probe cannot supply and a specification cannot name either.

A third group used to fail on the probe's *size*, which engine 0.8.0a10 fixed by retrying a refusal at a larger probe before believing it. That is also where a long-standing superstition went: the count used to vary between machines and was attributed to them, when the cause was one transform randomising its own parameters, so the same machine gave two answers. Escalating removed it. The count is still not promised, because it belongs to the installed library rather than to this package.

The check that decides your run is made when the specification is built, against your own shape and parameters.

Value

A character vector of transform names.

See Also

`augmentation_step()`, which uses one.

Examples

```
## Not run:
available_transforms()                       # everything, for images
#> 118 names, with albumentations 2.0.8

available_transforms(rank = 3)               # what volumes can take: fewer

available_transforms(rank = 3, pattern = "Flip|Crop")
#> [1] "AtLeastOneBBoxRandomCrop" "CenterCrop"   "CenterCrop3D"
#> [4] "CropAndPad"   "CropNonEmptyMaskIfExists"  "HorizontalFlip"
#> [7] "RandomCrop"   "RandomCrop3D"              "RandomCropFromBorders"
#> [10] "RandomResizedCrop"  "RandomSizedBBoxSafeCrop"  "RandomSizedCrop"
#> [13] "VerticalFlip"

# Which ones a volume cannot take, by name rather than by count.
head(setdiff(available_transforms(), available_transforms(rank = 3)), 3)
#> [1] "AdditiveNoise"      "BBoxSafeRandomCrop" "ChannelDropout"

## End(Not run)
```

`available_weights` *The published weights, and what they are*

Description

Weights are named rather than downloaded by hand, and a name is pinned to one commit of the repository holding it - so `"dsbowl-unet"` means one set of numbers today and the same set next year. This lists what is published, with the commit each name resolves to.

Usage

```
available_weights()
```

Details

Read the descriptions rather than only the names. A model's limitations decide whether it answers your question: the nuclei weights are semantic, so touching nuclei come back as one region and a count taken from them would be wrong.

Value

A data frame with `name`, `repo`, `filename`, `revision` and `description`.

See Also

`u_net()` and the other architectures, whose `weights` argument takes a name from here, a `hf://owner/repo/file.safetens` reference, or a path to a local file.

Examples

```
## Not run:
available_weights()

spec <- platypus_spec(
  data = segmentation_data("images/", "images/", colormap = binary_colormap),
  models = list(u_net("nuclei", input_shape = c(256, 256), blocks = 4, filters = 16,
                      weights = "dsbowl-unet", fit = FALSE))
)
masks <- predict(platypus_fit(spec), split = "validation")

## End(Not run)
```

`callback_swa` *Average the weights over the last part of the run*

Description

Stochastic weight averaging. Past `start`, every epoch's weights are folded into a running average and at the end that average becomes the model - the claim being that a point in the middle of a flat region generalises better than whichever corner the last epoch stopped in.

Usage

```
callback_swa(start = NULL, learning_rate = NULL)
```

Arguments

`start`	The fraction of the run after which averaging begins - 0.75 folds the last quarter. Unset uses the engine's default of 0.75. A fraction rather than an epoch so it survives a change to `epochs`, and the final epoch is always folded, so 1 averages one set of weights rather than none.
`learning_rate`	Hold the rate at this value once averaging begins. Unset leaves whatever the run was doing. Worth setting, and the reason is the whole mechanism: averaging is only worth something while the weights are still moving, and a rate that has decayed towards zero produces a set of nearly identical snapshots whose average is the last one. For the same reason a run cannot ask for this and `callback_cosine_annealing()` at once - the two undo each other and the specification refuses the pair.

Details

What it is worth is not what a table of scores shows. Three seeds, 60 epochs, lesions with ambiguous edges:

run	Dice	volume bias	volume |error|
plain	0.9639 ±0.0100	+1.71% ±12.18%	12.60% ±4.19%
`callback_swa()`	0.9667 ±0.0087	+2.08% ±2.15%	12.38% ±4.31%

Dice does not move and neither does the per-case error - both are inside the seed noise. What moves is **reproducibility**: the plain runs' volume bias was +4.5%, +12.2% and -11.6% across three seeds, and with averaging -0.1%, +4.2% and +2.1%. A spread 5.7 times tighter, improving on all three seeds when paired, by 7.34% at 4.9 standard errors.

So read it as *"the same model twice"* rather than *"a better model"* - which is what averaging is for, and the reason it belongs beside `mask_volume()` in any argument about whether a measured volume can be compared across runs.

Batch-normalisation statistics are recomputed afterwards, by a pass over the training data. They have to be: an averaged weight tensor inherits the statistics of whichever epoch was last rather than

averaging them, so without the pass the model is evaluated under the wrong normalisation and scores far worse than it should with nothing to say why. Models here have batch normalisation on by default, so this is nearly every run.

Value

A callback specification, to be passed to a model constructor.

See Also

callbacks for the others, `mask_volume()` for reading a volume in millilitres.

Examples

```
callback_swa()
callback_swa(start = 0.5, learning_rate = 1e-3)
```

`callbacks` *Callbacks*

Description

Things that happen between epochs. `monitor` is checked against the metrics the model actually asks for, so a callback watching a number that will never arrive is refused when the specification is built rather than waited on forever.

Usage

```
callback_early_stopping(
  monitor = "val_loss",
  patience = 10,
  min_delta = 0,
  restore_best = TRUE
)

callback_model_checkpoint(path, monitor = "val_loss", save_best_only = TRUE)

callback_reduce_lr_on_plateau(
  monitor = "val_loss",
  factor = 0.1,
  patience = 5,
  min_lr = 0
)

callback_cosine_annealing(min_lr = 0, epochs = NULL)

callback_csv_logger(path)

callback_terminate_on_nan()
```

Arguments

monitor	What to watch: "val_loss", "train_loss", or "val_" followed by a metric name, such as "val_dice".
patience	Epochs to wait before acting.
min_delta	Improvement smaller than this does not count.
restore_best	Put the best weights back when training stops, rather than keeping the last - which are by definition the ones that were not improving. On by default. Note that callback_swa() replaces nothing if a run stops before averaging began.
path	Where to write.
save_best_only	Only write when the watched quantity improves, which is the default. FALSE writes every epoch, which is what to use when a run may be interrupted rather than when the best epoch is what you want to keep.
factor, min_lr	For the learning rate schedule. min_lr is the floor for both callback_reduce_lr_on_plateau() and callback_cosine_annealing().
epochs	For callback_cosine_annealing(), how many epochs to spread the decay over. Unset means the model's epochs.

Details

Whether the watched quantity should rise or fall is worked out from its name: anything ending in loss is minimised, everything else maximised.

There is a seventh, documented separately because it needs a page of its own: callback_swa() averages the weights over the last part of the run. It watches nothing and changes how reproducible a run is rather than how well it scores.

Value

A callback specification.

Cosine annealing

callback_cosine_annealing() decays the learning rate from its initial value to min_lr along a cosine. It watches nothing: it is a function of how far through the run you are rather than of how the run is going, which is the difference from callback_reduce_lr_on_plateau() - and a run may legitimately want both.

Each parameter group decays from its **own** initial rate, so an encoder_learning_rate set on a model is not flattened at the first epoch. That would have been invisible: the history records one learning_rate, the first group's.

epochs unset means the model's own epochs, which is what you want - a cosine that ends where the run ends.

Examples

```
callback_early_stopping(monitor = "val_dice", patience = 10)
```

classes_to_colours *Paint a mask with its colormap*

Description

Paint a mask with its colormap

Usage

```
classes_to_colours(mask, colormap)
```

Arguments

mask	Class indices, counted from 1: a `height x width` matrix, or an `image x height x width` array for several at once.
colormap	A list of RGB triples, one per class; see binary_colormap.

Value

The same shape with a trailing RGB axis, values 0-255.

Examples

```
mask <- matrix(c(1, 1, 2, 2), nrow = 2)
dim(classes_to_colours(mask, binary_colormap))
```

colormaps *Colormaps and their labels*

Description

`binary_colormap` is black background, white foreground: the ordinary two-class problem. `voc_colormap` is the twenty-one class palette used by Pascal VOC, kept because datasets and other people's masks still arrive in it.

Usage

```
binary_colormap

binary_labels

voc_colormap

voc_labels
```

Format

A list of RGB triples, or a character vector of names.

Details

The labels are the matching class names, for figure legends and `mask_coverage()`. Position in the list is the class index throughout, counting from 1.

Value

Four exported objects rather than functions. `binary_colormap` and `voc_colormap` are lists of integer vectors of length three - one RGB triple per class, 2 entries and 21 entries respectively - where position in the list is the class index, counting from 1. `binary_labels` and `voc_labels` are character vectors of the matching class names, the same length as their colormap and in the same order, so the nth label names the class the nth triple draws.

Examples

```
mask_coverage(matrix(c(1, 2, 2, 1), 2), labels = binary_labels)
```

`colours_to_classes` *Read a colour mask back to class indices*

Description

The inverse of `classes_to_colours()`. Anything matching no colour becomes background, and `coverage` in the result says how much of the mask that was - a number near 1 means the colormap does not describe this dataset, which is the quietest way to train on nothing.

Usage

```
colours_to_classes(mask, colormap, tolerance = 0)
```

Arguments

`mask`	An RGB array: `height x width x 3`, or with a leading image axis.
`colormap`	A list of RGB triples, one per class.
`tolerance`	Allow this much difference per channel. Useful after lossy resizing, though masks should be resized with `nearest = TRUE` so it is not needed.

Value

A list with `classes` (indices counted from 1) and `coverage` (the fraction of pixels that matched a colour).

Examples

```
coloured <- classes_to_colours(matrix(c(1, 2, 2, 1), nrow = 2), binary_colormap)
colours_to_classes(coloured, binary_colormap)$classes
```

ct_windows *The named CT windows*

Description

The windows radiologists use, as `c(centre, width)` in Hounsfield units. Pass a name to `segmentation_data()` rather than the numbers; this is here for looking them up.

Usage

```
ct_windows()
```

Details

Held here rather than fetched from the engine so that looking up a constant does not require starting Python. A test checks the two lists against each other whenever the engine is available, so the copy cannot quietly drift.

Value

A named list of `c(centre, width)` pairs.

Examples

```
ct_windows()$lung
ct_windows()[["soft_tissue"]]
```

decodings *How predicted channels become objects*

Description

A model's `decode` is a **list** of these, not one: several decodings share one forward pass, so a second one costs seconds where a second training run costs an hour. `evaluate()` then returns a row per model and decoding.

Usage

```
decode_connected_components(minimum_area = NULL)

decode_nearest_seed(minimum_area = NULL)

decode_watershed(minimum_area = NULL)

decode_follow_flows(steps = NULL, step_size = NULL, minimum_area = NULL)
```

Arguments

`minimum_area`	Drop an object smaller than this many pixels. Unset keeps everything, and that is the engine's default on purpose: the right floor is a property of the data rather than of the method. A nucleus at 256x256 in the Data Science Bowl runs from 4 pixels up, so a floor picked here would silently delete the smallest real objects in one dataset while letting speckle through in another. It is applied after the objects are formed, so it removes answers rather than changing how they were reached.
`steps`	For `decode_follow_flows()`, how many Euler steps each pixel takes. **Measured as not mattering**: one step scores 0.9914 against 0.9923 converged, a span below the 0.0013 two runs differ by, because the field is continuous within an object and diverges at a seam - so the gap between two objects opens on the first step rather than after many. Adjustable because that argument is about this data's object sizes.
`step_size`	For `decode_follow_flows()`, how far one step moves, in pixels, on a unit-length field. **A cliff rather than a speed knob**: 0.9923 at 1, 0.4715 at 2, 0.0303 at 10. A step longer than a pixel jumps over the sink it was converging on and lands in a neighbour's basin.

Details

Each declares which separation it can read, and asking for one that cannot read the model's signal is refused when the specification is built rather than at the end of a run. `decode_connected_components()` reads none of them - it ignores the separation signal entirely - which is exactly why it belongs in every model's list: it is the row the signal's own row has to be read against. Left unset, a model decodes with its own signal's decoding plus that yardstick.

Value

A decoding block, to be passed inside a model's `decode` list.

See Also

separations, `instance_data()`

Examples

```
list(decode_nearest_seed(), decode_connected_components())
decode_follow_flows(step_size = 0.5)
```

`detection_anchors` *The anchors a detector used, and whether they were fitted*

Description

A detector cannot be reloaded without its anchors, and when they were fitted rather than named in the specification this is the only record. `export_weights()` writes them into the sidecar for the same reason.

Usage

```
detection_anchors(object, model = NULL)
```

Arguments

object	A fit from platypus_fit() on a detection specification.
model	Which model, when the specification trained several.

Value

A list: anchors as three groups of pairs, fitted saying whether they were fitted to your data, and when they were, the mean overlap they achieve and a data frame with one row per anchor.

Examples

```
## Not run:
anchors <- detection_anchors(fit)
anchors$mean_iou        # ~0.88 fitted on BCCD, against 0.65 for COCO's nine
anchors$per_anchor      # one row per anchor

# `fitted` says whether these were fitted to the data or taken from the
# specification. A detector cannot be reloaded without them, so they travel
# with the weights.

## End(Not run)
```

---

detection_crops *Cut every detection out of the image it was found in*

Description

A detector that knows three classes, followed by a classifier that knows eighty: find the objects, cut them out, hand the pieces to something else. predict() returns boxes in each image's own pixels precisely so they can be used on the photograph they came from, and this is what uses them.

Usage

```
detection_crops(
  object,
  model = NULL,
  split = "test",
  score_threshold = NULL,
  context = 0,
  size = NULL,
  fit = "letterbox",
  ...
)
```

Arguments

`object`	A `platypus_fit()` from a detection specification.
`model`	Which model, if the fit holds several.
`split`	Which split's images to read.
`score_threshold`	Crop at this confidence instead of the specification's `operating_point`.
`context`	Expand every box by this fraction of its own width and height, per side.
`size`	`c(height, width)` every crop is brought to, or `NULL` to keep each as cut. Crops of one image differ in size otherwise, which is why this returns a list either way.
`fit`	`"letterbox"` to preserve the aspect by padding, `"stretch"` to fill the frame.
`...`	Unused.

Details

Cut from the image at its **native size**, not from the square the network sees. Cropping the letterboxed frame would cut a shrunken object and then scale it back up: on a 1024-wide photograph fed to a 416 model, every crop would be two fifths of the size the data allows.

Four more decisions, each of which taken the other way changes what a downstream model sees without changing anything you would notice:

- **Rounded outward**, so the crop contains the whole box. Rounding to nearest loses up to a pixel per side, which on a 26-pixel platelet is a twelfth of the object.
- `context` **grows with the box**, not by a number of pixels, so one value suits a platelet and a white cell. Zero by default: a detector's box is tight by construction, and a classifier trained on whole objects does worse on something cut exactly at the edge - but expanding silently would put things nobody asked for into every crop.
- `size` **letterboxes rather than stretches.** Resizing a tall box to a square changes the aspect of every non-square object, and a classifier then sees a shape that does not occur in nature. `fit = "stretch"` is there for models trained that way, and has to be asked for.
- **Read at** `operating_point`**, not at the specification's** `score_threshold`**.** That one sits near zero so that average precision integrates the whole ranking - hundreds of boxes an image, almost all of them the tail it exists to measure over. Cropping those would hand a classifier mostly noise.

`dropped` is the number of boxes that clipped to nothing. A model may place a box off the frame and `predict()` reports what it said, so at a low threshold some arrive with no extent at all; those are dropped and counted rather than the arrays quietly coming back shorter than the detections.

Value

A list with one entry per image, each a list of `key`, `crops` (a list of `height x width x channels` arrays), `boxes`, `scores`, `labels`, `names` and `dropped`.

Examples

```
## Not run:
found <- detection_crops(fit, size = c(224, 224))
length(found[[1]]$crops)
plot_masks(found[[1]]$crops[[1]])        # look at one

# Everything above the operating point, as one flat list ready for another model.
pieces <- unlist(lapply(found, `[[`, "crops"), recursive = FALSE)

## End(Not run)
```

`detection_data` *Where the images and their annotation files are*

Description

The detection counterpart of `segmentation_data()`. Two layouts, the same two as segmentation: `nested_dirs` expects one directory per sample holding an `images` and an `annotations` subdirectory, and `config_file` expects a CSV with `images` and `annotations` columns. A detection dataset in Pascal VOC's own layout - one flat `JPEGImages` beside one flat `Annotations` - is neither, and the way in is to write three CSV files listing the pairs; `examples/detect_blood_cells.py` in the engine does exactly that for BCCD in a dozen lines.

Usage

```
detection_data(
  train,
  validation = NULL,
  classes,
  test = NULL,
  split = NULL,
  mode = c("nested_dirs", "config_file"),
  annotation_format = c("pascal_voc", "labelme"),
  coordinates = NULL,
  strict_labels = TRUE,
  window = NULL,
  subdirs = c("images", "annotations"),
  column_sep = ";",
  shuffle = TRUE
)
```

Arguments

`train, validation`
: Paths to the training and validation data.

`classes` Class names, **in class order**. Position is the class index the model learns, and that is why it is written down rather than read from the files: sorted, BCCD's three come out `Platelets, RBC, WBC`, which is reproducible and anatomically meaningless. A model trained with `RBC` first and used where the first class is `WBC` reports plausible boxes with plausible scores and nothing downstream can tell.

There is no background entry, unlike a segmentation colormap. A pixel always belongs to some class, so segmentation needs a name for "none of them"; a region of an image holding no object is simply not a box. One class is therefore a legitimate thing to ask for.

`test` Optional test data. Annotations are read if they are there, and the split is then scoreable; without them it can be predicted and `evaluate()` refuses it by name, because scoring against empty truth would report zero and a zero in a table reads as a result.

`split` Divide `train` instead of naming a `validation` set: a list with `fractions` (two numbers, or three to cut a test set too) and `group_by`, which is required and may be `NULL`. Exactly one of `split` and `validation`.

This is the only way to divide one folder for detection. `split_dataset()` writes a `masks` column and so cannot be used here; `split` divides the samples themselves, so column names never come into it. A test set cut this way carries its annotations, so unlike a separate `test` folder it can be scored and not only predicted on.

`mode` `"nested_dirs"` or `"config_file"`.

`annotation_format`

`"pascal_voc"` for the XML most detection datasets ship, or `"labelme"` for its JSON.

`coordinates` How to read a Pascal VOC box. The format's own convention is 1-based and inclusive of both ends, so a ten-pixel-wide box runs `xmin=1 xmax=10` and the width is `xmax - xmin + 1`. Plenty of tools write the same files 0-based and exclusive, where that box is `xmin=0 xmax=10`. Reading one as the other shifts every box by a pixel and shrinks it by one, which is invisible at a glance and costs real overlap on a small object - at 24 pixels a side, one pixel is 4%.

`NULL`, the default, means the VOC convention. Only meaningful for `pascal_voc`: LabelMe stores continuous pixel coordinates and has no convention to pick, so giving both is an error rather than a setting that does nothing.

`strict_labels` Refuse an object whose class is not in `classes`. `FALSE` skips it instead, which is how you train on two of a dataset's three classes. `TRUE` by default, because an unexpected label is more often a typo in `classes` than an intention.

`window` How values in real units reach the model, for DICOM and NIfTI. See `segmentation_data()`; ignored for ordinary pictures, which is what most detection data is.

`subdirs` For `nested_dirs`, the names of the image and annotation subdirectories.

`column_sep` For `config_file`, what separates several paths in one cell.

`shuffle` Shuffle the training data between epochs.

Value

A data specification, to be passed to `platypus_spec()`.

See Also

yolo3(), plot_boxes()

Examples

```
detection_data("train/", "valid/", classes = c("RBC", "WBC", "Platelets"))
```

---

drawing_style *The decisions a drawing makes*

Description

Every value plot_masks(), plot_boxes(), overlay_mask() and overlay_agreement() use when you do not give them one: which colours mean found, missed and invented, one colour per class for plot_anchors(), how much of the image an overlay lets through, how a box is labelled, and which boxes are worth drawing at all.

Usage

```
drawing_style()
```

Details

They belong to the engine - pyplatypus.style - and are mirrored here so that drawing needs no Python. A test holds the two together, so the same call draws the same picture from either package.

Reach for this when you want to match a figure rather than restate it: passing colours = drawing_style()$box_colours to something else keeps one source, and typing the hex codes starts a second.

Value

A named list: agreement_colours, box_colours, class_colours, overlay_alpha, box_label_format and box_min_score.

See Also

plot_masks(), plot_boxes(), overlay_agreement()

Examples

```
drawing_style()$agreement_colours[["missed"]]

# The same list the engine returns, which is what keeps the two drawings the same.
str(drawing_style())
```

encoders *Pretrained encoders, and what they are worth*

Description

`u_net()` and the other model constructors take an `encoder`, which replaces the built-in contracting path with a backbone architecture, and a separate `pretrained` flag that loads that backbone's ImageNet weights. This page is the measurement behind them, because the honest summary is narrower than the idea.

How to ask

```
u_net("polyps", input_shape = c(256, 256),
      encoder = "resnet34",      # the architecture
      pretrained = TRUE,         # and its ImageNet weights
      freeze_encoder = 5)        # held still while the decoder settles
```

Two flags rather than one, so naming an architecture never reaches the network on a machine that has none - a hospital network often has none.

What it is worth

Measured on three datasets, one per modality, three seeds each, every configuration compared at its best validation Dice with the best weights restored. Training sets of 40 to 100 images on purpose, because that is the regime transfer learning is for.

configuration	DS Bowl, microscopy	SIIM-ACR, chest X-ray	Kvasir-SEG, endoscopy
built-in encoder, 1.9M parameters	**0.8460**	0.0531	0.5830
`encoder = "resnet34"`, 9.0M	0.7891	0.1073	0.5670
the same, `pretrained`, `freeze_encoder = 5`	0.8333	**0.1203**	**0.5932**
the same, `encoder_learning_rate = lr/10`	0.8326	0.1011	0.5329

Four things come out of it, and each repeats on all three:

Use `freeze_encoder`, not `encoder_learning_rate` alone Freezing was the better of the two everywhere. On Kvasir a tenth of the rate is the *worst* row in the table - below the built-in encoder and below the same backbone trained from nothing. Giving one rate to the whole network is worse still: a randomly initialised decoder's gradients undo the transferred weights before the decoder has learned anything worth having.

Do not expect a better score The best pretrained configuration beats the built-in encoder on one dataset of three, by 0.0102, against a seed-to-seed spread of 0.0170 - inside the noise. And endoscopy is the fairest test available within medical imaging: camera photographs of tissue, three channels, natural lighting. With the backbone frozen for a whole run on DS Bowl the score is 0.6068 against 0.8460, so the features genuinely do not transfer to these images.

Expect to reach the same score sooner On Kvasir the built-in encoder needed 150, 110 and 83 epochs across its three seeds; pretrained and frozen needed 64, 82 and 83. That halving is the one benefit that showed up consistently, and on a dataset that takes a day to train it is the difference that matters.

A bigger encoder earns its place where the target is thin On SIIM-ACR, where a pneumothorax covers a median 0.59% of the frame, every backbone variant roughly doubles the built-in encoder's score - and the built-in encoder's spread between seeds there (0.0586) is larger than its own mean, with two of three seeds never learning at all. This is a reason to be able to swap the encoder that has nothing to do with ImageNet.

Measure on your own images before trusting any of this: the regime where transfer helps is narrow, and three public datasets are three public datasets. The engine ships `examples/compare_pretrained_encoders.py`, which is this measurement parameterised.

Which backbones

Any convolutional family timm names, as timm names it: `"resnet18"`, `"resnet34"`, `"resnet50"`, `"resnext50_32x4d"`, `"efficientnet_b0"` through `"b3"`, `"mobilenetv3_large_100"`, `"densenet121"`, `"regnety_040"`, `"vgg16"`, `"inception_v3"` and others. `blocks` then says how many of the backbone's stages to use.

Two kinds are refused rather than approximated:

- **A 3D `input_shape`.** ImageNet is images, so there is nothing to transfer to a volume. Refused while the specification is read, before anything is downloaded.
- **The patch-based families** (`"convnext_*"`, `"swin_*"`). Their features begin at a quarter of the input, so the half-resolution level a U-shaped decoder recovers fine detail from does not exist. Inventing it would put a made-up level exactly where the detail comes from.

The finest level is always platypus's own: every ImageNet stem halves the input, so no backbone has a full-resolution feature, and the decoder's output head sits on one. That stage is trained from scratch, which is why `freeze_encoder` and `encoder_learning_rate` leave it alone - it is the one part with everything to learn.

Installing

Nothing to do. The environment this package builds always includes the backbones, which cost 10 MB against PyTorch's own several hundred - not a trade worth making anyone opt into. On a GTX 10-series card or older see `platypus_use_torch()`.

See Also

models for the arguments, `platypus_spec()` for the specification.

`evaluate` *Compare the trained models*

Description

One row per model: what it is, how big it is, how long it trained, and how it scored.

Usage

```
evaluate(object, ...)

## S3 method for class 'platypus_fit'
evaluate(object, split = "validation", ...)
```

Arguments

`object`	A `platypus_fit()`.
`...`	Unused.
`split`	Which data to score on: `"validation"`, `"train"` or `"test"`.

Details

The `loss` column is only comparable between models trained on the same loss, which is why `loss_function` is there beside it. A Focal-Tversky of 0.05 is not better than a CCE-Dice of 0.14; it is not even the same question. Metrics stay comparable, because they measure the mask rather than the objective.

Value

A data frame, one row per model.

A note on the name

`evaluate` is also exported by the **evaluate** package, which knitr depends on, so the two mask each other depending on which was attached last. It is a generic here, so `evaluate(fit)` dispatches on the fit's class whichever one you reach - and `rplatypus::evaluate(fit)` says so outright. The name is kept because it is the word for what this does; `platypus_fit()` carries a prefix only because `fit` is a generic in the tidymodels packages and a plain `fit` would have been theirs to own.

Examples

```
## Not run:
evaluate(fit)                       # one row per model, on the validation split
evaluate(fit, split = "test")

# A split with no masks is refused by name rather than scored against nothing;
# `predict()` still works on it.

## End(Not run)
```

`evaluate_cases` *Score every case separately*

Description

`evaluate()` gives one row per model: one number for a whole set. This gives one row per case - each image, or each patient when `group_by` is set - because one number hides the distribution, and the distribution is usually the finding. A model averaging Dice 0.86 that scores 0.1 on three patients has a failure mode, and the mean is where it hides.

Usage

```
evaluate_cases(object, ...)

## S3 method for class 'platypus_fit'
evaluate_cases(
  object,
  model = NULL,
  split = "validation",
  group_by = NULL,
  ...
)

## S3 method for class 'platypus_cases'
summary(object, ...)
```

Arguments

`object`	A `platypus_cases` from `evaluate_cases()`.
`...`	Unused.
`model`	Which model, by the name in the specification. Defaults to the first.
`split`	Which data to score on.
`group_by`	A pattern picking a group out of each case's name, as in `split_dataset()`. With it, the rows are patients rather than slices: a patient's slices are pooled into one score, the way a volume would be.

Details

`summary()` on the result gives the distribution: mean, standard deviation, median, range, and which case scored worst.

Tiled models are scored on the whole image, not on tiles. Overlaps are accumulated across a case's tiles and the metric applied once, which is that image's score exactly; averaging the tiles' scores would be a different number, and unkind to any case whose object happens to straddle a boundary.

Value

A data frame with one row per case, of class `platypus_cases`.

See Also

split_dataset() for keeping a patient out of two sets in the first place.

Examples

```
## Not run:
cases <- evaluate_cases(fit)
summary(cases)
head(cases[order(cases$dice), ])          # the ones worth looking at

evaluate_cases(fit, group_by = "^(patient\\\\d+)_")

## End(Not run)
```

---

evaluate_classes *Average precision per class*

Description

Detection only. On a segmentation fit this refuses by name and points at evaluate_cases(). The name carries no detection_ prefix on purpose: this is an S3 generic, so a future task gains a method under the same name rather than a second spelling of one idea.

Usage

```
evaluate_classes(object, ...)

## S3 method for class 'platypus_fit'
evaluate_classes(object, model = NULL, split = "validation", ...)
```

Arguments

object	A fit from platypus_fit() on a detection specification.
...	Unused.
model	Which model, when the specification trained several.
split	Which split to score.

Details

The row that matters on unbalanced data, which is most detection data: BCCD has 4,155 red cells against 372 white and 361 platelets, so a single number is a number about red cells. evaluate() deliberately carries no overall precision or recall for the same reason - averaging them over classes needs a weighting, and every choice of weighting is a different claim.

Value

A data frame, one row per class: average precision at IoU 0.5, the mean overlap of the boxes that matched, the number of true boxes, and precision and recall at the model's operating_point.

Examples

```
## Not run:
evaluate_classes(fit, "bccd")

# The row that matters on unbalanced data, which is most data: BCCD has 4155 red
# cells against 372 white and 361 platelets, so a single number is a number about
# red cells.

## End(Not run)
```

`evaluate_images` *Score every image separately*

Description

Detection only. On a segmentation fit this refuses by name and points at `evaluate_cases()`. The name carries no `detection_` prefix on purpose: this is an S3 generic, so a future task gains a method under the same name rather than a second spelling of one idea.

Usage

```
evaluate_images(object, ...)

## S3 method for class 'platypus_fit'
evaluate_images(
  object,
  model = NULL,
  split = "validation",
  score_threshold = NULL,
  ...
)
```

Arguments

`object`	A `platypus_fit()` on a detection specification.
`...`	Unused.
`model`	Which model, by the name in the specification. Defaults to the first.
`split`	Which data to score on.
`score_threshold`	The confidence at which the counts are read. Unset, the specification's `operating_point` is used, which is what it is for: "how many were missed" is undefined over the whole ranking, where every box the model dimly considered counts as a prediction.

Details

`evaluate()` gives one row per model and `evaluate_classes()` one row per class. This gives one row per image, which is the question asked next: not how well on average, but **which** pictures it fails on. A mean over a split says 0.86; it does not say the misses are four frames where the stain is dark, and that difference is usually something about the data rather than the model.

Sort by `missed` for the frames it cannot see, and by `mean_matched_iou` for the ones where it finds everything and places it badly. Those are different problems - the first is usually the data, the second usually the anchors.

There is no average precision here, deliberately. Average precision is the area under a precision-recall curve, so it is a property of a ranking over a whole dataset; computed on one image with three boxes it moves on a single box's rank and means nothing. What is meaningful for one picture is counting and overlap, which is what the columns carry.

`mean_matched_iou` is `NA` rather than 0 when nothing matched. An image where the model found nothing and one where it found badly-placed boxes are different failures, and a zero would merge them.

Value

A data frame of class `platypus_images`, one row per image: `key`, `n_truth`, `n_predicted`, `matched`, `missed`, `spurious` and `mean_matched_iou`.

See Also

`evaluate_classes()` for one row per class, `evaluate_cases()` for the segmentation counterpart.

Examples

```
## Not run:
images <- evaluate_images(fit)
head(images[order(-images$missed), ])          # the frames it cannot see
head(images[order(images$mean_matched_iou), ]) # the ones it places badly

## End(Not run)
```

---

export_weights *Save a trained model's weights*

Description

Writes safetensors plus a `.json` sidecar recording what the weights are for: architecture, input shape, channels, classes, and whatever else you pass. Loading refuses a model they do not belong to, which is what the sidecar is for - weights trained on a different colormap with the same number of classes load cleanly and predict nonsense.

Usage

```
export_weights(object, path, model = NULL, ...)
```

Arguments

`object`	A `platypus_fit()`.
`path`	Where to write, with or without the `.safetensors` extension.
`model`	Which model, by the name in the specification. Defaults to the first.
`...`	Extra fields for the sidecar, for instance `data = "BBBC038v1"`, `licence = "CC0 1.0"`.

Details

Anything in `...` joins the sidecar. The data they were trained on and its licence belong there: weights whose provenance lives in somebody's memory cannot be used by anybody else, including you in a year.

Value

The path written, invisibly, with the sidecar's contents attached as the `recorded` attribute - so you can see what was put beside the weights without opening the file.

Examples

```
## Not run:
export_weights(fit, "weights/my-run",
             data = "our 2026 cohort", licence = "internal use only")

## End(Not run)
```

`inspect_masks` *What the colormap or labels match in your masks*

Description

Reads a sample of one split's masks and reports what the specification's colours, or list of labels, actually matched. Trains nothing and loads no model.

Usage

```
inspect_masks(spec, split = c("train", "validation", "test"), limit = 500)
```

Arguments

`spec`	A `platypus_spec()`.
`split`	`"train"`, `"validation"` or `"test"`.
`limit`	How many masks to read. They are spread across the split rather than taken from its start, because datasets arrive sorted and a prefix would answer about the beginning.

Details

platypus_fit() does a small version of this before every run and refuses when a declared class appears in none of the masks it read. This is the same question asked deliberately, over as much of the data as you want - which is what to reach for when that refusal looks wrong.

Value

A one-row data frame: unmatched, samples_checked, total_samples, and present_classes and missing_classes as comma-separated strings so the frame stays printable. The classes are also attached as integer vectors in the attributes present and missing.

What the numbers mean

missing_classes is the one to act on. A class that appears in no mask cannot be learned: its channel of the target is zero everywhere, so there is no gradient towards it and the model is never shown the thing it is being asked to find. Either the colours do not describe these masks, or they name a class the data does not contain.

unmatched is the fraction of mask pixels matching no entry, which fall back to the background class. **Read it knowing that it scales with the size of the thing being segmented**, so it is loud for a large structure and almost silent for a small lesion. Measured on masks that are white, with a colormap asking for a colour that is not there: a foreground covering 20% of the image gives 19.8%, and one covering 0.6% gives 0.61% - which is less than the 0.72% that JPEG compression leaves around the edge of a perfectly correct mask. That is why the refusal is built on class presence instead.

Examples

```
## Not run:
inspect_masks(spec)
inspect_masks(spec, split = "validation", limit = 2000)

## End(Not run)
```

---

instance_data *Data for instance segmentation: tell the objects apart, not just find them*

Description

Masks stored one file per object, read as a label map that keeps each object's identity instead of uniting them into one foreground. The question this answers is *how many nuclei, and how big is each one*, which a semantic mask cannot answer however good it is: two touching nuclei are one region in it.

Usage

```
instance_data(
  train,
  validation = NULL,
  test = NULL,
  split = NULL,
  mode = c("nested_dirs", "config_file"),
  window = NULL,
  subdirs = c("images", "masks"),
  column_sep = ";",
  shuffle = TRUE
)
```

Arguments

`train`	Training data: a directory, a configuration file, or a `platypus_split` from `split_dataset()`, which carries its own validation set and division.
`validation`	A validation set, or `FALSE` for none. Exactly one of `validation` and `split`, and neither when `train` is already a `platypus_split`.
`test`	A test set to predict on. Scored only if it carries masks.
`split`	Divide `train` instead of naming a `validation` set: a list with `fractions` (two numbers, or three to cut a test set too) and `group_by`, which is required and may be `NULL`. Exactly one of `split` and `validation`.
`mode`	`"nested_dirs"` or `"config_file"`.
`window`	How values in real units reach the model, for DICOM and NIfTI. See `segmentation_data()`.
`subdirs`	For `nested_dirs`, the names of the image and mask subdirectories. The second one holds **one file per object** rather than one mask per image, which is the layout this task is named for.
`column_sep`	For `config_file`, what separates several paths in one cell.
`shuffle`	Shuffle the training data between epochs.

Details

Why this exists rather than an argument on `segmentation_data()`. Every field `instance_data()` takes, that one takes too - what makes it a different constructor is the five it **refuses**: `colormap`, `labels`, `regions`, `channels_from` and `target_spacing` do not apply to a directory of per-object masks, and the engine rejects them by name. A constructor whose contract is "these five do not apply" is a contract rather than a label, and it gets that refusal from its own argument list rather than from a table somebody has to keep in step.

The consequence worth knowing before you choose: `task` is still read off the constructors rather than typed. Calling this one *is* how a specification says it is about instances, so there is no `task = "instance_segmentation"` to write and nothing to keep in agreement with it. See `platypus_spec()`.

What the model does with this is the other half, and it is one block: a model's `separation` says what the network predicts - `rim` or `flows` - and `decode` says how those channels become objects. Neither belongs here, because they describe the run and not the data.

Value

A data specification, to be passed to `platypus_spec()`.

See Also

`segmentation_data()` for one mask per image, `platypus_spec()`

Examples

```
instance_data("stage1_train/", "stage1_validation/")
```

`loss_boundary` *A loss that knows how far a wrong voxel is from the truth*

Description

Dice and IoU count a voxel the same wherever it sits, which is why a model can reach 0.88 on Dice while its volumes run a fifth too large: the overshoot is all at the boundary, and for a small lesion the boundary is most of the object. This adds `mean(phi * p)`, where `phi` is the signed distance to the truth's boundary - negative inside, positive outside - so a voxel predicted far outside the truth costs in proportion to how far.

Usage

```
loss_boundary(region = NULL, alpha = NULL)
```

Arguments

`region`	The overlap term this is added to - any other loss. Unset uses the engine's default, which is Dice. `loss_focal()` here is what this feature was originally asked for: focal combined with a distance metric.
`alpha`	How much of the objective is the region term, `1 - alpha` the surface term. Unset uses the engine's default of 0.9. **Not 0.5, and that was measured.** At 0.5 the surface term owns half the objective while the prediction is still random, which is where it is unstable: three seeds gave a volume error spread of ±28.28% against the baseline's ±3.11%, with every number worse. Left unset here on purpose - the number is a measurement and belongs in the one place the measurement lives, so that an engine that learns better cannot be contradicted from this side.

Details

What it buys and what it costs, measured. Three seeds, 60 epochs, lesions with ambiguous edges, against Dice alone:

loss	Dice	volume bias	volume |error|
`loss_dice()`	0.9525 ±0.0027	-5.18% ±3.11%	16.90% ±0.90%

`loss_boundary()`	0.9411 ±0.0093	-0.06% ±4.15%	22.17% ±2.88%

It trades **accuracy per case** for **being unbiased over a series**. "Is there a lesion" wants the overlap; "has it grown since March" wants a volume that is not systematically wrong, and no single number answers both - which is the same point `mask_volume()` makes about reporting millilitres beside an overlap score. The bias improvement is suggestive on three seeds, since the baseline's own bias wanders ±3.11%; the costs are established.

It costs time too. The signed distance transform runs once per sample, in the loader's workers: 5.3 ms for a 256x256 image, 23.4 ms for a 64x64x32 volume and 709.8 ms at 128^3. On large volumes give `platypus_fit()` workers, or the transform becomes the training.

Value

A loss specification, to be passed to a model constructor.

See Also

losses for the nine region losses, `mask_volume()` for reading a volume in millilitres rather than voxels.

Examples

```
loss_boundary()
loss_boundary(region = loss_focal(gamma = 2))
loss_boundary(region = loss_dice(), alpha = 0.95)
```

---

losses *Loss functions*

Description

The objective the model is trained against. Every one of these works unchanged on volumes as well as images.

Usage

```
loss_iou(smooth = 1)

loss_dice(smooth = 1)

loss_cce(label_smoothing = 0)

loss_cce_dice(cce_weight = 0.5, smooth = 1)

loss_focal(gamma = 2, alpha = NULL)
```

```
loss_tversky(alpha = 0.5, smooth = 1)

loss_focal_tversky(alpha = 0.5, gamma = 1, smooth = 1)

loss_combo(alpha = 0.5, ce_ratio = 0.5)

loss_mse(where = NULL)

loss_l1(where = NULL)

loss_smooth_l1(beta = NULL, where = NULL)

loss_lovasz(per_image = FALSE)
```

Arguments

`smooth`	Added to numerator and denominator to keep an empty class finite.
`label_smoothing`	For cross-entropy, train towards `1 - label_smoothing` instead of towards 1, so the model is not pushed to be certain. Worth having where the labels themselves are uncertain, which on a hand-drawn boundary they are.
`cce_weight`	For `loss_cce_dice()`, how much of the total the cross-entropy half provides. At 0 or 1 it is one of the two alone, which is worth knowing before tuning it as though it were a third thing.
`gamma`	For focal losses, how hard to discount the pixels already classified well.
`alpha`	For Tversky, the weight on false negatives; raising it buys recall. Note that `alpha = 0.5` is Dice exactly only when `smooth = 0`.
`ce_ratio`	For `loss_combo()`, how its cross-entropy half is tilted between punishing missed foreground (towards 1) and false alarms (towards 0), while `alpha` sets how much of the total each half provides.
`where`	For the three regression losses, where the target means anything and so where the error is averaged. A flow field is defined on the objects and undefined off them - there is no direction towards a centre for a pixel in no object - so the builder writes zero on the background, and `"everywhere"` would spend most of the objective learning that zero. The foreground is 13.97% of a Data Science Bowl image, so `"everywhere"` gives 86% of the gradient to a constant; `"foreground"` is the default for that reason rather than by taste.
`beta`	For `loss_smooth_l1()`, where the quadratic part ends and the linear one begins, **in the target's own units**. A flow field is scaled to roughly `[-1, 1]`, so the default of 1 makes the whole range quadratic - which is `loss_mse()` with extra steps. Lower it to get any of the robustness this loss is for.
`per_image`	For `loss_lovasz()`, sort the errors of each image separately rather than over the whole batch. The harsher reading, and on mostly empty images the noisier one.

Details

There is one documented separately because it needs a page of its own: `loss_boundary()` adds the signed distance to the truth's boundary on top of any of these, which removes the systematic volume bias at some cost in per-case accuracy.

Three of these score numbers rather than classes - `loss_mse()`, `loss_l1()` and `loss_smooth_l1()` - and they are not alternatives to the others. A class target has nothing for them to measure and a continuous one has nothing for an overlap to measure, so they belong in exactly one place: the `loss` of `separation_flows()`, where the target is a vector field. Measured against the pinned engine rather than assumed: giving one as **any** segmentation model's own `loss` is refused by name - semantic or instance - and giving a class loss to `separation_flows()` is refused by name the other way round.

This page said *an instance model's own* until the pin moved to `0.8.0a22`, and it was true rather than wrong: the refusal lived on the instance class alone, and a semantic model accepted `loss_mse()` and trained with it, reporting a validation loss of 0.0014 for a mask that was half missing where `loss_cce()` reaches a Dice of 1.0000. The default `where = "foreground"` is what made it look as though it had worked - it evaluates the objective only inside the labelled objects and leaves the background unconstrained. The engine refuses it at either task now (`pyplatypus#223`).

Value

A loss specification, to be passed to a model constructor.

See Also

`separation_flows()`, whose `loss` is the one place the regression losses belong

Examples

```
loss_focal(gamma = 2)
loss_tversky(alpha = 0.7)
loss_smooth_l1(beta = 0.1, where = "foreground")
```

---

mask_coverage *How much of each class a mask covers*

Description

A quick sanity check before training: a foreground of half a percent is a problem the loss has to be chosen for, and it is better to know before the run than after.

Usage

```
mask_coverage(mask, labels = NULL)
```

Arguments

`mask`	Class indices, counted from 1.
`labels`	Optional names for the classes.

Value

A data frame with one row per class.

Examples

```
mask <- matrix(1L, 8, 8); mask[2:4, 2:4] <- 2L
mask_coverage(mask, labels = c("background", "nucleus"))

# A class with 0 pixels is the thing to look for: its target channel is zero
# everywhere, so there is no gradient towards it and the model is never shown
# what it is being asked to find.
```

---

mask_volume *How much of a volume a class occupies, in millilitres*

Description

A voxel count answers nothing on its own: the same tumour is 4000 voxels on one scanner and 12000 on another, and neither number goes in a report. Multiplied by the volume of a voxel it becomes millilitres, which is comparable between scans, between patients and against whatever the radiologist measured by hand.

Usage

```
mask_volume(mask, spacing, class = 2L)
```

Arguments

mask	A mask of class indices, as predict() returns - one volume, not a stack.
spacing	Voxel size in millimetres, length 3. volume_info() reports it; so does the spacing_1/2/3 columns of its output.
class	Which class to measure, numbered from 1 as the colormap and labels are. Defaults to 2, which is the foreground in a binary segmentation.

Value

The volume in millilitres.

Examples

```
mask <- array(1L, dim = c(10, 10, 4))
mask[3:6, 3:6, 2:3] <- 2L
# 32 voxels of 2 x 2 x 5 mm: 32 * 20 mm^3 = 640 mm^3 = 0.64 ml
mask_volume(mask, spacing = c(2, 2, 5))
```

`metrics` *Metrics*

Description

What gets reported. Unlike the losses these are computed on the hard prediction - the mask you will actually be handed - because a metric taken on probabilities reads higher than the model deserves.

Usage

```
metric_iou(smooth = 1, include_background = TRUE)

metric_dice(smooth = 1, include_background = TRUE)

metric_tversky(alpha = 0.5, smooth = 1, include_background = TRUE)

metric_cldice(iterations = 5, smooth = 1, include_background = FALSE)
```

Arguments

`smooth`	Smoothing. Zero is allowed and is the honest choice for a reported number: with smoothing, a class absent from an image scores a perfect 1.
`include_background`	Average over the background class as well. In medical images the background is usually most of the picture, so leaving it in can turn a model that found nothing into one that looks respectable. Papers report foreground only.
`alpha`	For Tversky, the weight on false negatives.
`iterations`	For clDice, how many times the skeleton is peeled. It has to reach the half-width of the thickest structure, or its core is never reduced to a centreline and the score is refused rather than reported - an empty skeleton otherwise reads as a perfect 1.

Details

Three of these ask the same question - how much of the mask coincides - and differ in how they weigh the ways of being wrong. `metric_cldice()` asks a different one: whether the *structure* coincides. For something thin and connected that is the question worth asking, and the two disagree in a way worth knowing. Measured on a nine-pixel vessel: one drawn two pixels thin scores Dice 0.875, and the same vessel severed in the middle scores 0.947 - **Dice prefers the severed one**. clDice gives them 1.000 and 0.943.

Its limit, also measured: at one pixel wide a displacement of one pixel leaves no overlap at all, and every overlap-based metric reads 0, this one included.

`include_background` defaults to `FALSE` for clDice alone. The background's skeleton lies inside the background by construction, so that class scores 1 whatever the model did - averaging it in moved 0.7865 to 0.8933 on a three-pixel vessel.

Value

A metric specification.

Examples

```
metric_dice(include_background = FALSE)

# Vessels, airways, neurons, cracks: thin and connected, where Dice asks the wrong thing.
metric_cldice()
```

`models` *Segmentation models*

Description

Four members of the U-shaped family. They take the same arguments and differ only in what happens between the contracting and expanding paths: `u_net` concatenates its skip connections, `linknet` adds them, `res_u_net` makes each block residual, and `u_net_plus_plus` fills in the nested skip pathways.

Usage

```
u_net(
  name,
  input_shape,
  channels = NULL,
  n_class = NULL,
  blocks = NULL,
  filters = NULL,
  block_width = NULL,
  dropout = NULL,
  batch_normalization = NULL,
  separable_conv = NULL,
  spatial_dropout = NULL,
  upsample = NULL,
  deep_supervision = NULL,
  activation = NULL,
  initialiser = NULL,
  loss = NULL,
  metrics = NULL,
  optimizer = NULL,
  callbacks = NULL,
  augmentation = NULL,
  epochs = NULL,
  batch_size = NULL,
  splits = NULL,
  weights = NULL,
```

```
  fit = NULL,
  encoder = NULL,
  pretrained = NULL,
  freeze_encoder = NULL,
  encoder_learning_rate = NULL,
  separation = NULL,
  decode = NULL
)

u_net_plus_plus(
  name,
  input_shape,
  channels = NULL,
  n_class = NULL,
  blocks = NULL,
  filters = NULL,
  block_width = NULL,
  dropout = NULL,
  batch_normalization = NULL,
  separable_conv = NULL,
  spatial_dropout = NULL,
  upsample = NULL,
  deep_supervision = NULL,
  activation = NULL,
  initialiser = NULL,
  loss = NULL,
  metrics = NULL,
  optimizer = NULL,
  callbacks = NULL,
  augmentation = NULL,
  epochs = NULL,
  batch_size = NULL,
  splits = NULL,
  weights = NULL,
  fit = NULL,
  encoder = NULL,
  pretrained = NULL,
  freeze_encoder = NULL,
  encoder_learning_rate = NULL,
  separation = NULL,
  decode = NULL
)

res_u_net(
  name,
  input_shape,
  channels = NULL,
  n_class = NULL,
```

```
  blocks = NULL,
  filters = NULL,
  block_width = NULL,
  dropout = NULL,
  batch_normalization = NULL,
  separable_conv = NULL,
  spatial_dropout = NULL,
  upsample = NULL,
  deep_supervision = NULL,
  activation = NULL,
  initialiser = NULL,
  loss = NULL,
  metrics = NULL,
  optimizer = NULL,
  callbacks = NULL,
  augmentation = NULL,
  epochs = NULL,
  batch_size = NULL,
  splits = NULL,
  weights = NULL,
  fit = NULL,
  encoder = NULL,
  pretrained = NULL,
  freeze_encoder = NULL,
  encoder_learning_rate = NULL,
  separation = NULL,
  decode = NULL
)

linknet(
  name,
  input_shape,
  channels = NULL,
  n_class = NULL,
  blocks = NULL,
  filters = NULL,
  block_width = NULL,
  dropout = NULL,
  batch_normalization = NULL,
  separable_conv = NULL,
  spatial_dropout = NULL,
  upsample = NULL,
  deep_supervision = NULL,
  activation = NULL,
  initialiser = NULL,
  loss = NULL,
  metrics = NULL,
  optimizer = NULL,
```

```
  callbacks = NULL,
  augmentation = NULL,
  epochs = NULL,
  batch_size = NULL,
  splits = NULL,
  weights = NULL,
  fit = NULL,
  encoder = NULL,
  pretrained = NULL,
  freeze_encoder = NULL,
  encoder_learning_rate = NULL,
  separation = NULL,
  decode = NULL
)
```

Arguments

`name`	Unique within a specification. It names this model's outputs, its rows in every score table, and the directory its run record is written to - so two models called the same thing would overwrite one record and report it as two, which is why the engine refuses a repeat rather than disambiguating it.
`input_shape`	Tile size: `c(height, width)`, or `c(depth, height, width)`.
`channels`	Input channels. 3 for colour, 1 for greyscale. Derived from `channels_from` when `segmentation_data()` names one file per channel.
`n_class`	**Removed.** The number of classes comes from the data - `colormap` or `labels` in `segmentation_data()` - and a model that also declared it could only ever disagree with its own data. Passing it is an error that says so.
`blocks`	How many times the resolution is halved. Taken from the weights file when `weights` is given and this is not.
`filters`	Filters in the first block; doubled at each level. Taken from the weights file when `weights` is given and this is not.
`block_width`	Convolutions per block. Two is the usual U-Net block; one trains faster and holds less, three rarely pays for itself. It composes with every architecture, including the residual blocks of `res_u_net()`, where it is the number of convolutions inside the residual rather than beside it.
`dropout`	Drop this fraction of activations in every block during training, 0 to switch it off. The first thing to reach for when the training loss keeps falling and the validation loss does not.
`batch_normalization`	Normalise each batch's activations between convolutions, which is what lets a deeper network train at all at a useful learning rate. On by default; the reason to turn it off is a batch so small that its statistics are noise, which is the case a 3D run with `batch_size = 1` lands in.
`separable_conv`	Depthwise-separable convolutions. Far fewer parameters, a little slower to converge.

`spatial_dropout`
: Drop whole feature maps rather than individual activations. On an image the neighbouring pixels of a dropped activation carry almost the same information, so ordinary dropout removes less than it appears to; dropping a channel removes a feature. Does nothing unless `dropout` is above 0, which sets the rate.

`upsample`
: Interpolate and convolve instead of using a transposed convolution.

`deep_supervision`
: Train every decoder depth, not only the last. `u_net_plus_plus` only, since it is the one with intermediate outputs to supervise.

`activation`
: One of `"relu"`, `"leaky_relu"`, `"elu"`, `"selu"`, `"gelu"`, `"silu"`, `"tanh"`.

`initialiser`
: One of `"he_normal"`, `"he_uniform"`, `"glorot_normal"`, `"glorot_uniform"`.

`loss`
: The objective trained against, see losses and `available_losses()`. It is what a score table's `loss` column is comparable within and not across: a Focal-Tversky of 0.05 is not better than a cross-entropy of 0.14, it is a different question, which is why the table carries the loss's name beside the number. Unset uses the engine's default.

`metrics`
: A list of metrics, see metrics.

`optimizer`
: An optimiser, see optimizers.

`callbacks`
: A list of callbacks, see callbacks.

`augmentation`
: A list of `augmentation_step()` steps, applied to training data only.

`epochs`, `batch_size`
: Training length and batch size.

`splits`
: Cut each image into a grid of tiles instead of shrinking it: one number per spatial dimension, so `c(2, 3)` gives six tiles. Use it to segment a large image without throwing away resolution; predictions are reassembled to the original size.

`weights`
: A checkpoint to start from, or to use as-is with `fit = FALSE`.

`fit`
: Set `FALSE` to load `weights` and skip training.

`encoder`
: A pretrained-architecture backbone to use as the contracting path instead of the built-in encoder, named as timm names it: `"resnet34"`, `"efficientnet_b0"`, `"mobilenetv3_large_100"`, `"densenet121"`, `"vgg16"` and the other convolutional families. Naming one builds that architecture; it does not load anything. **2D only** - ImageNet is images, so a backbone with a 3D `input_shape` is refused while the specification is read. The patch-based families (`convnext_*`, `swin_*`) are refused too: their features start at a quarter of the input, and the half-resolution level a U-shaped decoder recovers fine detail from does not exist. `blocks` then says how many of the backbone's stages to use.

`pretrained`
: Load the backbone's ImageNet weights. Separate from `encoder` on purpose, so naming an architecture never reaches the network on a machine that has none. Measured on three datasets, transfer does not beat the built-in encoder here - see encoders for the table and what it is actually good for.

`freeze_encoder`
: Hold the transferred layers still for this many epochs, then train them. **Use this rather than** `encoder_learning_rate` **alone**: it was the better of the two on all three datasets measured. A number at or above `epochs` keeps them fixed for the whole run.

`encoder_learning_rate`
: A separate, smaller rate for the layers that arrived pretrained. The full-resolution stage in front of them is platypus's own and starts random, so it keeps the optimiser's rate. Giving this instead of `freeze_encoder` measured worse everywhere, and on one dataset worse than no backbone at all.

`separation`
: What the network predicts so that touching objects can be told apart - `separation_rim()` or `separation_flows()`. Giving it is what makes this an **instance** model: it has no counterpart on a semantic one, so a model carrying it can only mean instances, which is how a specification knows its task without anybody typing it. The data has to be `instance_data()` to match. The head's width follows from the signal and the number of classes and is derived rather than stated.

`decode`
: A **list** of decodings turning those channels into objects, e.g. `list(decode_nearest_seed(), decode_connected_components())`. A list rather than one, because several decodings share a single forward pass: a second costs seconds where a second training run costs an hour, so `evaluate()` returns a row per model *and* decoding. Unset, the engine uses the signal's own decoding plus `decode_connected_components()`, which ignores the signal and is therefore the row the others have to be read against. Only meaningful with `separation`.

Details

`input_shape` is the tile the model sees: two numbers for images, three for volumes. Its length is what makes a model 2D or 3D - there is no separate switch. Every spatial dimension must divide by `2^blocks`, or the decoder cannot line up with the encoder, and that is checked when the specification is built rather than part-way through training.

Value

A model specification, to be passed to `platypus_spec()`.

Where the defaults live

Every argument below except `name` and `input_shape` is `NULL` unless you set it, and a `NULL` is left out of the request entirely - so the engine's own default applies. The defaults are therefore written down once, in `pyplatypus`, rather than here as well:

```
channels 3          blocks 4           filters 16       block_width 2
dropout 0           batch_normalization TRUE            separable_conv FALSE
spatial_dropout TRUE                   upsample FALSE   deep_supervision FALSE
activation "relu"                      initialiser "he_normal"
loss cce            metrics iou        optimizer adam   callbacks none
epochs 10           batch_size 8       fit TRUE
```

The second reason matters more than tidiness. A value sent is indistinguishable from a value chosen, so defaults travelling from here would silence the engine's own knowledge: `weights` naming a published file records its architecture, `blocks` and `filters`, and those are adopted **only** where the specification stayed silent. Sending `blocks = 4` because that is what R happened to say would override a file that knows better.

Examples

```
u_net("unet", input_shape = c(256, 256), blocks = 4, filters = 16)

# A large image, tiled rather than shrunk
u_net("hd", input_shape = c(256, 256), splits = c(4, 6))
```

`optimizers` *Optimisers*

Description

How the weights are updated. `optimizer_adam()` unless there is a reason not to: it adapts the step per parameter, so it usually trains without anything tuned except `learning_rate`. `optimizer_sgd()` with `momentum` is the recipe most of the classification literature's best numbers were produced with, in exchange for depending on a tuned rate and usually a schedule - see callbacks for those.

Usage

```
optimizer_adam(
  learning_rate = 0.001,
  beta_1 = 0.9,
  beta_2 = 0.999,
  eps = 1e-08,
  weight_decay = 0,
  amsgrad = FALSE
)

optimizer_adamw(
  learning_rate = 0.001,
  beta_1 = 0.9,
  beta_2 = 0.999,
  eps = 1e-08,
  weight_decay = 0.01
)

optimizer_sgd(
  learning_rate = 0.001,
  momentum = 0,
  nesterov = FALSE,
  weight_decay = 0
)

optimizer_rmsprop(
  learning_rate = 0.001,
  alpha = 0.99,
  momentum = 0,
  eps = 1e-08,
```

```
  weight_decay = 0
)

optimizer_adagrad(
  learning_rate = 0.001,
  lr_decay = 0,
  eps = 1e-10,
  weight_decay = 0
)

optimizer_adadelta(
  learning_rate = 0.001,
  rho = 0.9,
  eps = 1e-06,
  weight_decay = 0
)

optimizer_adamax(
  learning_rate = 0.001,
  beta_1 = 0.9,
  beta_2 = 0.999,
  eps = 1e-08,
  weight_decay = 0
)

optimizer_nadam(
  learning_rate = 0.001,
  beta_1 = 0.9,
  beta_2 = 0.999,
  eps = 1e-08,
  weight_decay = 0
)
```

Arguments

`learning_rate`	How big a step to take. The one setting worth trying first when a run will not learn: too high and the loss moves without improving, too low and it improves too slowly to tell from not improving at all. Each parameter group decays from its own rate, so an `encoder_learning_rate` is not flattened by a schedule.
`beta_1`	Decay for the running average of the gradient. Lower reacts to the last few batches, higher smooths over more of them; 0.9 is the usual value and rarely worth moving.
`beta_2`	Decay for the running average of the squared gradient, which is what scales each parameter's step. Below about 0.99 training gets noticeably noisier. In `optimizer_adamax()` it decays the largest gradient seen instead.
`eps`	Added to the denominator so a parameter whose gradient has been zero for a while does not take an enormous step. A guard, not a knob.

`weight_decay` Pull the weights towards zero at every step, which is L2 regularisation. Off by default because it interacts with the loss and is not free: on a small dataset it can cost more than the overfitting it prevents. In `optimizer_adamw()` it is applied to the weights directly rather than through the gradient, which is the whole difference between the two.

`amsgrad` Keep the largest squared-gradient average seen so far rather than the current one, so the step size never grows back. It fixes a convergence case in Adam's proof and usually changes nothing in practice; worth trying if a run plateaus and then gets worse.

`momentum` Carry a fraction of the previous step into this one. 0 is plain gradient descent; 0.9 is the usual choice and is what makes SGD competitive with the adaptive methods.

`nesterov` Look ahead to where momentum is about to carry the weights before measuring the gradient. Needs `momentum` above 0, and giving it without momentum is refused rather than ignored.

`alpha` For `optimizer_rmsprop()`, the decay for its running average of the squared gradient - the same work `beta_2` does in the Adam family, under that family's name.

`lr_decay` For `optimizer_adagrad()`, shrink the rate further as training goes on, on top of the shrinking Adagrad already does by accumulating every squared gradient it has seen.

`rho` For `optimizer_adadelta()`, how long a window of squared gradients to remember. Adadelta exists to avoid Adagrad's ever-shrinking step and this is the parameter that does it.

Details

No run in this project has compared them on segmentation, so this page says what each one does differently rather than which is better. `available_optimizers()` lists them with the arguments each accepts.

The rest are here because a configuration should be able to name what torch offers. `optimizer_adamw()` is Adam with weight decay applied to the weights rather than through the gradient, which is why its decay defaults to 0.01 rather than 0 - choosing it and leaving the decay at zero leaves nothing to choose. `optimizer_adagrad()` accumulates every squared gradient it has seen, so its step only ever shrinks and a long run can stop learning; `optimizer_adadelta()` is the same idea with a window instead of a sum. `optimizer_adamax()` tracks the largest gradient rather than the average of their squares, which is steadier where a few batches carry very large ones. `optimizer_rmsprop()` is Adam's denominator without its momentum, and `optimizer_nadam()` is Adam with the look-ahead that SGD calls `nesterov`.

Every argument left unset is decided by the engine, which is where the defaults live.

Value

An optimiser specification.

See Also

available_optimizers() for the same list with its arguments, and callbacks() for moving the rate during a run.

Examples

```
optimizer_adam(learning_rate = 1e-4)
optimizer_sgd(learning_rate = 0.01, momentum = 0.9, nesterov = TRUE)
```

---

overlay_agreement *Where a prediction and the truth disagree*

Description

Three colours rather than one: what was found, what was missed, and what was invented. A Dice of 0.85 says nothing about which of the three is costing you, and for most clinical questions they are not interchangeable - a missed lesion and a false alarm are different kinds of wrong.

Usage

```
overlay_agreement(
  image,
  prediction,
  truth,
  alpha = drawing_style()$overlay_alpha,
  colours = drawing_style()$agreement_colours
)
```

Arguments

image	A height x width x channels array.
prediction, truth	Class indices, counted from 1.
alpha	How strongly to tint.
colours	Named colours for hit, missed and false_alarm.

Value

A height x width x 3 array of 0-255 integers.

Examples

```
image <- array(runif(8 * 8 * 3, 0.2, 0.6), dim = c(8, 8, 3))
truth <- matrix(1L, 8, 8); truth[2:4, 2:4] <- 2L
predicted <- matrix(1L, 8, 8); predicted[3:5, 2:4] <- 2L
shown <- overlay_agreement(image, prediction = predicted, truth = truth)
dim(shown)

# Three colours, not one: a missed lesion and a false alarm cost different things,
# and a single overlap score hides which one you have.
```

---

overlay_mask *Lay a mask over the image it belongs to*

Description

The picture that answers "where is it looking". Background - class 1 - is left alone, so only the classes that were found are tinted.

Usage

```
overlay_mask(image, mask, colormap, alpha = drawing_style()$overlay_alpha)
```

Arguments

image	A `height x width x channels` array. Values in 0-1 or 0-255; greyscale is repeated across three channels.
mask	Class indices, counted from 1, matching the image's height and width.
colormap	A list of RGB triples, one per class.
alpha	How strongly to tint, 0 to 1.

Value

A `height x width x 3` array of 0-255 integers.

Examples

```
image <- array(runif(8 * 8 * 3, 0.2, 0.6), dim = c(8, 8, 3))
mask <- matrix(1L, 8, 8); mask[2:4, 2:4] <- 2L
tinted <- overlay_mask(image, mask, colormap = list(c(0, 0, 0), c(255, 0, 0)))
dim(tinted)
```

`platypus_conditions` *When something is wrong: the conditions this package raises*

Description

Every failure that comes back from the engine is raised as a **classed condition**, so a script can tell one kind of problem from another instead of matching on the message. Until now that was true and undocumented, which made it true and useless.

What the class vector looks like

Three or four entries, outermost first:

```
platypus_config_error    the specific kind
platypus_error         every failure from this package, which is the one to catch
error                    an ordinary R error
condition
```

So `tryCatch(..., platypus_error = )` catches everything this package raises and nothing else, and a specific class narrows it. `platypus_error` appears once even when the engine sends no kind at all.

What to catch

`platypus_error`, unless there is a reason not to. The specific kinds come from the engine, so the list below belongs to the version this package pins, and a later engine may add one. Catching the base class is the thing that keeps working.

Two are worth naming on purpose:

`platypus_config_error` A specification that does not describe a runnable experiment. This one carries **data**: `conditionCall()` aside, the condition has a `problems` data frame with one row per problem - `where`, `problem` and, when it is short enough to be useful, `got` - and a `source` naming the YAML file if it came from one. The engine reports every problem it found rather than the first, so the frame is what makes "three things are wrong" readable.

`platypus_training_stopped` Not a defect. A callback ended the run early, which is what it was asked to do, and anything scoring the result afterwards should treat it as a finished run rather than a failure.

The kinds the pinned engine can send

One class per kind, prefixed. `tests/testthat/test-conditions.R` holds this list to what the engine actually defines, so a kind added upstream fails here rather than arriving undocumented.

```
platypus_config_error            a specification that cannot be run
platypus_engine_error          the engine could not do what the specification asked
platypus_model_error             the architecture could not be built as described
```

platypus_data_error	the dataset on disk is not what the specification says
platypus_image_error	an image could not be read, or is not the shape claimed
platypus_mask_error	a mask does not match its colormap
platypus_channel_error	one channel per file, and the files do not agree
platypus_volume_error	a volume, its orientation or its spacing
platypus_dicom_error	a DICOM file or series
platypus_augmentation_error	a transform that cannot do what was asked of it
platypus_detection_error	detection metrics
platypus_detection_data_error	annotations, boxes or classes
platypus_instance_error	a separation target, or decoding one into objects
platypus_instance_data_error	the per-object mask files a case has, or has not
platypus_weights_error	weights that do not belong to this model
platypus_training_stopped	a callback ended the run early
platypus_unknown_case	a case or model name that is not in this run

platypus_unknown_case is the one with no class behind it in the engine: the R bridge raises it itself, because asking for a case that does not exist is a question the bridge can answer without troubling Python.

Why R has more of these than Python does

The engine exports three exception classes - PlatypusError, ConfigError, WeightsError - and raises the other twelve internally, so a Python caller catches PlatypusError and inspects it. R derives its class from the kind the engine sends with every failure, so all of them arrive as catchable names. That is a wider surface on this side, and deliberately so: R has no exception hierarchy to lean on, and inherits() is cheaper than parsing a message.

See Also

platypus_status(), for when the problem is that the engine never started.

Examples

```
# A specification with two problems, caught and read as data rather than as text.
## Not run:
spec <- platypus_spec(
  data = segmentation_data("train", "valid", colormap = binary_colormap),
  models = list(u_net("m", input_shape = c(64, 64), loss = "focaal"))
)

tryCatch(
  platypus_fit(spec),
  platypus_config_error = function(e) {
    print(e$problems)        # one row per problem: where, problem, got
    invisible(NULL)
  },
  platypus_error = function(e) {
    message("something else went wrong: ", conditionMessage(e))
  }
)
```

```
## End(Not run)
```

`platypus_device` *Where the work will happen*

Description

Reports the PyTorch build in use and the device it can see. Starts the engine if it is not running.

Usage

```
platypus_device()
```

Value

A list, printed readably.

Why this page shows no output

What this prints is a property of the machine it runs on, so an example output here would describe whichever machine built this page rather than yours. Run it and read your own.

Examples

```
## Not run:
platypus_device()

## End(Not run)
```

`platypus_fit` *Train the models in a specification*

Description

Trains every model the specification asks for, in order, each with its own data pipeline - so two models may differ in input size or tiling and still be compared fairly, on the same data.

Usage

```
platypus_fit(
  spec,
  device = NULL,
  num_workers = "auto",
  strict_data = TRUE,
  check_masks = TRUE,
  verbose = FALSE
)
```

Arguments

spec	A platypus_spec().
device	"cuda", "cpu", or NULL to use a GPU when one is present.
num_workers	Processes used to read and decode images. "auto" picks a small number based on the machine. This matters more than it sounds: reading in the main process alone leaves the GPU waiting on the disk, and on the Data Science Bowl at 128x128 that is the difference between 35 seconds an epoch and under 10. Set 0 to avoid multiprocessing entirely if it causes trouble.
strict_data	Treat an incomplete sample as an error. Set FALSE to train on the rest and be told what was skipped.
check_masks	Read a sample of the training masks before training, and refuse when a class the specification declares appears in none of them. A colormap or set of labels that matches none of the labelled tissue is the quietest way a run wastes a day: every mask reads as background, the loss falls because background is most of a medical image, and the model learns to answer "nothing here". Set FALSE only when the missing class is real but rarer than the sample - and see inspect_masks() first, which asks the same question over as much of the data as you like.
verbose	Report each epoch as it finishes.

Details

Validation data is never augmented. Measuring a model on distorted images measures the distortion.

This blocks until training finishes. With verbose = TRUE each epoch is reported as it completes, which on anything longer than a few minutes is the difference between waiting and wondering.

Value

A platypus_fit.

Examples

```
## Not run:
fit <- platypus_fit(spec, verbose = TRUE)
evaluate(fit)
masks <- predict(fit, "unet")

## End(Not run)
```

---

platypus_spec *Build an experiment specification*

Description

Either from arguments, or from a YAML file by passing its path as the only argument. Both produce the same object, and everything downstream treats them identically.

Usage

```
platypus_spec(
  data,
  models = NULL,
  task = NULL,
  seed = NULL,
  deterministic = NULL,
  output_dir = NULL,
  check_paths = TRUE
)
```

Arguments

data	A segmentation_data(), detection_data() or instance_data() specification, or the path to a YAML file. Which one it is decides task.
models	A list of model specifications, see models and available_models(). More than one is the ordinary case rather than the exception: they are trained in turn against the same data and the same split, which is what makes their numbers comparable, and a comparison is written by listing several entries of the same architecture with different settings.
task	Which task this specification describes, one of "semantic_segmentation", "object_detection" or "instance_segmentation". Normally left unset: the data and model constructors already decide it, and segmentation_data() with u_net() can only mean one thing. Given, it is **checked against them rather than trusted**, so it can only agree or refuse. The third task sharpened what this argument is for rather than making it load-bearing. An instance *data* block declares nothing a segmentation one lacks, so instance_data() is what says the word - but a *model* can still say it on its own, because separation is required on an instance model and has no counterpart on a semantic one.
seed	Makes a run repeatable as far as the arithmetic is deterministic, which goes further on some hardware than others - so it is worth knowing where it stops. **Repeatable:** the split (split_dataset() gives the same partition on any machine), a detector's fitted anchors, the order the data arrives in and its augmentation - including inside the loader's worker processes - and, on the processor, every number to twelve decimal places. **Not repeatable:** a run on a CUDA device. Measured on a GTX 1070, two runs of one configuration at one seed agreed to twelve decimals on the processor and differed in the fourth after a single epoch on the card, because cuDNN chooses algorithms neither package constrains. Nor across versions of the engine: anything that changes the order data is drawn in changes what one seed draws, and 0.8.0a3 did exactly that. So it is the argument for comparing two of your own runs, and not for publishing a number somebody else can land on. platypus_device() reports which device yours will use.
deterministic	Ask the engine for deterministic algorithms as well, which is the only thing that makes a run on a card repeatable - seed on its own does not, as above.

The cost is real and arrives later than any other refusal here. Some convolutions become much slower, and an operation with no deterministic implementation raises *when it is reached*, so a run can stop ten minutes in rather than when the specification is built. Everything else this package refuses, it refuses before reading a single image; this cannot, because whether an operation has a deterministic algorithm is a fact about the installed engine and the device rather than about the specification.

Needs `seed`, and is refused without one: constraining the algorithms while the draw still varies leaves the run exactly as unrepeatable and only slower.

`output_dir` Where to write a record of the run: the specification, the history, and anything the run worked out that the specification does not already say - a detector's fitted anchors, which it cannot be reloaded without. `NULL`, the default, writes nothing.

Unset means *nothing is written*, not "written to a default place". A field with a default cannot be told apart from one somebody set to that default, so sending the default from here would have put files in every user's working directory the moment the engine learned to write them - which it just did.

`check_paths` Verify that the data paths exist. Turn it off to build a specification on a machine that does not hold the data.

Details

The engine validates it, so a mistake is caught here rather than part-way through a training run - and reported against the model that has it, by the name you gave it.

Value

A `platypus_spec`.

Examples

```
## Not run:
spec <- platypus_spec(
  data = segmentation_data("train/", "valid/", colormap = binary_colormap),
  models = list(
    u_net("unet", input_shape = c(256, 256)),
    linknet("linknet", input_shape = c(256, 256), loss = loss_focal_tversky(alpha = 0.7))
  )
)

# The same thing, from a file
spec <- platypus_spec("experiment.yaml")

## End(Not run)
```

platypus_status *Is the engine ready?*

Description

Reports whether the Python side has been started, and what it is. Calling this does not start it - use it to check before a long run, or when something is wrong.

Usage

```
platypus_status()
```

Value

A list with the engine version, the interpreter in use and where its environment lives, or NULL fields when Python has not been started yet.

Why this page shows no output

What this prints is a property of the machine it runs on, so an example output here would describe whichever machine built this page rather than yours. Run it and read your own.

Examples

```
## Not run:
platypus_status()

## End(Not run)
```

platypus_use_torch *Ask for a particular PyTorch build*

Description

Call this before anything starts the engine - straight after `library(rplatypus)` - and before the first call that needs Python.

Usage

```
platypus_use_torch(build = c("pascal", "default"))
```

Arguments

build	"pascal" for the last PyTorch built against CUDA 12, or NULL for the default.

Details

There is one reason to: **a GeForce GTX 10-series card, or anything older**. PyTorch 2.8 and later ship CUDA 13 builds, and CUDA 13 dropped the Maxwell, Pascal and Volta generations outright. No driver update brings them back. Without this, such a card is simply not used and everything runs on the processor, perhaps ten times slower, with nothing obviously wrong.

On anything from Turing - the RTX 20-series onwards - the default is correct and this is unnecessary.

Value

The requirement string now in force, invisibly.

Why this page shows no output

What this prints is a property of the machine it runs on, so an example output here would describe whichever machine built this page rather than yours. Run it and read your own.

Examples

```
## Not run:
library(rplatypus)
platypus_use_torch("pascal")   # GTX 10-series and older

## End(Not run)
```

---

plot.platypus_fit *Plot what happened during training*

Description

One panel per quantity, one line per model. Useful for the thing a final number cannot tell you: whether it had stopped improving, or was still going when it ran out of epochs.

Usage

```
## S3 method for class 'platypus_fit'
plot(x, metrics = NULL, ...)
```

Arguments

x	A platypus_fit().
metrics	Which columns to plot; defaults to the losses and metrics.
...	Unused.

Value

A ggplot.

Examples

```
## Not run:
plot(fit)                              # every metric that was recorded
plot(fit, metrics = "dice")

## End(Not run)
```

`plot_anchors` *Draw the anchors against the boxes they were fitted to*

Description

The picture the 2020 package drew, and the one no summary statistic replaces. Every annotated box is a point - its width against its height, both as fractions of the model's input - coloured by class, with the anchors on top.

Usage

```
plot_anchors(object, model = NULL, split = "train", log = FALSE, size = 1.1)
```

Arguments

`object`	A fit from `platypus_fit()` on a detection specification.
`model`	Which model, when the specification trained several.
`split`	Which split's boxes to draw.
`log`	Draw both axes on a log scale. Detection datasets span a wide range of sizes • a BCCD platelet is about two fifths the side of a red cell and a fifth of a white one - and on linear axes the smallest class collapses into the corner.
`size`	Point size for the boxes.

Details

What it answers that a mean overlap does not: **whether a class has any anchor near it at all**. A mean of 0.65 can be one class covered well and another not covered at all, and those two situations want different fixes. On blood cells the three classes occupy three distinct regions - median sides of 133, 69 and 26 pixels at a 416 input - and anchors fitted to all three together cover them at 0.88, 0.85 and 0.83, so fitting to the mixture does not abandon the smallest class. COCO's nine cover the same three at 0.64, 0.70 and 0.70: *evenly* worse rather than blind to one, which is the useful thing to know. They are not aimed elsewhere - they span 10 to 373 pixels a side because COCO holds objects of every size, and most of that range describes nothing here.

Worth drawing for a split the anchors were **not** fitted on. Anchors that sit among the training boxes and away from the validation ones say the two halves hold different objects, and no training curve shows that.

Value

A ggplot object.

See Also

`detection_anchors()` for the numbers, `yolo3()` for choosing them.

Examples

```
## Not run:
fit <- platypus_fit(spec)
plot_anchors(fit)                      # the split they were fitted to
plot_anchors(fit, split = "validation")  # and one they were not

## End(Not run)
```

---

plot_boxes *Draw boxes on images*

Description

The detection counterpart of `plot_masks()`, and the function the 2020 package had under this name. Boxes arrive from `predict()` in each image's own pixels, so they land on the picture without anything being undone first.

Usage

```
plot_boxes(
  images,
  boxes,
  truth = NULL,
  which = NULL,
  min_score = drawing_style()$box_min_score,
  labels = NULL,
  colours = drawing_style()$box_colours,
  size = 0.6,
  text_size = 3
)
```

Arguments

`images`	An image stack from `read_images()`, or a single image.
`boxes`	Boxes to draw: one data frame as `predict()` returns per image, or a list of them - one per image in `images`. Columns `xmin`, `ymin`, `xmax`, `ymax`, and optionally `name` and `score`.
`truth`	Boxes to draw as ground truth, the same shape as `boxes`.
`which`	Which images to draw. Defaults to the first four, like `plot_masks()`.

`min_score`	Drop predicted boxes below this confidence before drawing. The default follows a detector's own `score_threshold` of 0.01, which is kept low so average precision can be computed over the whole ranking - and which puts far more boxes on a picture than anyone wants to look at. `0.5` is the usual choice for a figure.
`labels`	Row labels. Defaults to the names of `boxes`, which `predict()` sets to the sample keys.
`colours`	Named vector giving the colour for `prediction` and for `truth`.
`size`	Line width of a box.
`text_size`	Size of the class label, or `NULL` to draw no labels.

Details

Predictions and truth can be drawn together, and when they are, the point is the comparison: truth in one colour, predictions in another, both on the same image. A detector that found the right number of objects in the wrong places and one that found the wrong number in the right places score similarly and look nothing alike.

Value

A ggplot object.

See Also

`predict.platypus_fit()`, `plot_masks()`

Examples

```
## Not run:
found <- predict(fit, split = "test")
plot_boxes(read_images(files, size = NULL), found, min_score = 0.5)

## End(Not run)
```

---

plot_masks *Look at masks beside the images they came from*

Description

Builds a grid: one row per image, one column per panel. Which panels appear depends on what you pass - the image alone, the truth, the prediction, and where they disagree.

Usage

```
plot_masks(
  images,
  prediction = NULL,
  truth = NULL,
  colormap = binary_colormap,
  which = NULL,
  alpha = drawing_style()$overlay_alpha,
  labels = NULL,
  slice = NULL
)
```

Arguments

images	An image x height x width x channels array, or one height x width x channels image.
prediction	Predicted class indices, from predict.platypus_fit().
truth	True class indices, if you have them.
colormap	A list of RGB triples, one per class.
which	Which images to show.
alpha	How strongly to tint the overlays.
labels	Row labels; defaults to the image number.
slice	For volumes, which slice to draw. A volume cannot honestly be shown as one picture, so one plane is chosen and said out loud rather than a projection being invented. Counted along the last axis, which after the canonical reorientation is the axial direction - the view a radiologist scrolls through. "middle" takes the middle slice, which is the sensible first look.

Details

The disagreement panel is usually the one worth looking at. It separates what was found from what was missed and what was invented, and those are not interchangeable: for most clinical questions a missed lesion and a false alarm cost different things, and a single overlap score hides which one you have.

Value

A ggplot.

Examples

```
## Not run:
plot_masks(images, prediction = masks, truth = truth, colormap = binary_colormap)

## End(Not run)
```

`predict.platypus_fit` *Predict masks*

Description

Predict masks

Usage

```
## S3 method for class 'platypus_fit'
predict(
  object,
  model = NULL,
  split = "test",
  type = c("class", "probability"),
  space = c("model", "source"),
  cases = NULL,
  each = NULL,
  ...
)
```

Arguments

`object`	A `platypus_fit()`.
`model`	Which model, by the name given in the specification. Defaults to the first one trained.
`split`	Which data to predict on.
`type`	`"class"` gives the class index per pixel, which is the mask to look at; `"probability"` keeps the per-class channel.
`space`	`"model"` or `"source"`; see above. Note that the two return different shapes of thing, an array and a list, because they are different things.
`cases`	Case names to predict, as `evaluate_cases()` reports them. With this the result is a **named list** of just those, in the order asked for. A name that is not in the split is an error rather than a missing entry.
`each`	A function called as `each(case, mask)` once per image, instead of the predictions being returned together. Whatever it returns is collected under the case's name; return `NULL` from it and nothing accumulates. This is the form that does not need the split to fit in memory.
`...`	Unused.

Value

With `space = "model"`: for `"class"`, an integer array of `image x height x width`, classes numbered from 1 to match the colormap; for `"probability"`, the same with a trailing class axis. Tiled models return images at their original size, reassembled.

With `space = "source"`: a list of such arrays, one per scan, each on that scan's grid.

Which grid the answer is on

`space = "model"` returns everything on the grid the model worked at, stacked into one array. That is what training saw, and it is the only form that can be a single array, because a stack needs one shape.

`space = "source"` maps each prediction back onto the grid of the scan it was computed from - undoing the resampling and the crop - and therefore returns a **list**, one mask per scan. This is what you want for anything that has to meet the original data: laying a mask over the scan in a viewer, writing it beside the scan with `save_volumes()`, or measuring it in millilitres of the scan's own voxels. Scans differ in size, so the list is not a limitation being worked around - resizing them to a common shape is exactly how a mask ends up describing anatomy it was not computed from.

Two things are lost mapping back, and neither can be recovered. Interpolation softens a boundary, so the mask returns slightly smoother than the model drew it. And where reading cropped anatomy away, the mask comes back padded with background - which means *not examined*, not *nothing there*. Give the model an `input_shape` that covers the anatomy if that distinction matters.

One at a time, when the split will not fit

The default returns everything together, which for a tiled run is the whole split at full resolution: 200 fundus photographs at 2048 x 2048 is about 6.7 GB before anything is drawn, and the engine's own `predict` was killed by exactly that. Two arguments avoid it.

`cases` names the ones wanted. `each` hands them over one at a time and keeps only what the function gives back. They compose: `cases` says which, `each` says what to do with them.

The saving is in memory and not in time. The engine's stream is sequential, so asking for two cases out of two hundred still runs the model over two hundred - it simply never holds more than one answer at once.

Examples

```
## Not run:
masks <- predict(fit, "unet")                     # an array, on the model's grid
masks <- predict(fit, "unet", space = "source")   # a list, on each scan's own grid

# The two worst cases, without holding the other 198.
cases <- evaluate_cases(fit, split = "validation")
worst <- cases$case[order(cases$dice)][1:2]
two <- predict(fit, split = "validation", cases = worst)

# Or walk the split, keeping one number per image and none of the masks.
coverage <- predict(fit, split = "validation", each = function(case, mask) mean(mask > 1))

# `space = "source"` returns a list and not an array, which is the API being
# honest: scans differ in size, a stacked array needs one shape, and resizing
# them to match is how a mask ends up describing anatomy it was not computed from.

## End(Not run)
```

read_images *Read images as the model sees them*

Description

Reads through the engine's own pipeline rather than a separate decoder, so what you plot is what the model was given. A picture drawn from a differently resized copy is a picture of something the model never saw, and the disagreements it appears to show may be nothing but the resizing.

Usage

```
read_images(paths, size = NULL, channels = 3, nearest = FALSE)
```

Arguments

paths	Image files.
size	`c(height, width)` to read at, or `NULL` to keep each image's own size - which only works if they all agree.
channels	3 for colour, 1 for greyscale.
nearest	Use nearest-neighbour resizing. Required for masks: interpolating one invents colours belonging to no class, which then quietly become background.

Value

An `image x height x width x channels` array of 0-255 values.

Examples

```
## Not run:
images <- read_images(list.files("test/", full.names = TRUE), size = c(256, 256))

## End(Not run)
```

read_masks *Read mask files as class indices*

Description

Masks come two ways, and this reads both. `colormap` is for masks stored as pictures, one colour per class. `labels` is for masks stored as label maps - a vector of the voxel values, in class order - which is how volume formats store them.

Usage

```
read_masks(paths, colormap = NULL, labels = NULL, size = NULL, tolerance = 0)
```

Arguments

paths	Image files.
colormap	A list of RGB triples, one per class. Give this or labels.
labels	The voxel value of each class, in class order: c(0, 1) for a binary segmentation. For NIfTI and other label maps. Give this or colormap.
size	c(height, width) to read at, or NULL to keep each image's own size - which only works if they all agree.
tolerance	Passed to colours_to_classes(). Ignored for label maps, which are compared to within half a unit - a label map that has been resampled or merely passed through a float will not satisfy an exact comparison, and a label that fails to match becomes background.

Value

An array of class indices, counted from 1.

Examples

```
## Not run:
# Masks stored as pictures: the colormap says which colour is which class.
truth <- read_masks(list.files("masks/", full.names = TRUE),
                    colormap = list(c(0, 0, 0), c(255, 255, 255)))

# Masks stored as label maps, which is how volumes do it. Exactly one of the two.
truth <- read_masks("case_01/seg.nii.gz", labels = c(0, 1))

# `tolerance` is for masks that have been through a lossy resize, where a colour
# that should be (255, 0, 0) arrives as (254, 1, 0) and matches nothing.
truth <- read_masks(paths, colormap = voc_colormap, tolerance = 2)

## End(Not run)
```

---

read_record *Read what a run wrote down about itself*

Description

platypus_fit() leaves a record beside the weights when the specification named an output_dir: the whole specification, whatever the run worked out for itself, the epoch-by-epoch history, and the version of the engine that produced it. This reads one back.

Usage

```
read_record(path)
```

Arguments

path	The run.json to read, or the directory holding it - so both read_record("out/my_model") and read_record("out/my_model/run.json") work.

Details

Named to match read_record() in the engine, and read_images() and read_masks() here.

Value

A platypus_record: a list with written, pyplatypus, model, specification, derived and history. Everything is plain R data, because a record describes what happened rather than being a live thing - except history, which comes back as a data frame with one row per epoch.

That frame is **not** quite what training_history() returns, and the difference is worth knowing before you rbind() them: a record belongs to one model, whose name is the top-level model field, so the frame has no model column. training_history() spans every model in the fit and needs one. Every other column is the same.

Why you would want it

The specification in the record is a thing you can run again, not a description of a run. And for a detector whose anchors were **fitted** rather than named, derived$anchors is the only copy outside a weights sidecar - so a record is how a result stays reproducible after the session that produced it has gone.

It does not start Python

Deliberately. Everything else in this package asks the engine; this reads a text file, and a first call to the engine provisions an environment of several gigabytes. A record you cannot read without the thing that wrote it is worth less than one you can, so this needs only jsonlite, and works on a machine that has never had Python.

See Also

training_history() for the same numbers from a fit in the current session, and export_weights(), whose sidecar answers a different question - what the weights are for, rather than what the run was.

Examples

```
## Not run:
spec <- platypus_spec(
  data = segmentation_data("train", "valid", colormap = binary_colormap),
  models = list(u_net("vessels", input_shape = c(256, 256))),
  output_dir = "runs"
)
fit <- platypus_fit(spec)

record <- read_record("runs/vessels")
record$pyplatypus          # the engine that produced it
head(record$history)       # one row per epoch
```

```
record$specification       # enough to run it again

## End(Not run)
```

`save_masks` *Write masks to files*

Description

Predictions are only half useful while they are an array in a session. This writes them as ordinary PNGs that open in anything - ImageJ, QuPath, a viewer, a colleague's machine - and returns the paths, so it can sit at the end of a pipeline.

Usage

```
save_masks(masks, dir, colormap, names = NULL, suffix = "")
```

Arguments

`masks`	Class indices from `predict.platypus_fit()` - an `image x height x width` array, or one `height x width` matrix. Probabilities are accepted too and reduced to the most likely class. For **volumes** use `save_volumes()`, which writes NIfTI carrying the geometry of the scan each mask came from. This function cannot reliably tell a stack of volumes from a stack of images with a class axis - the ranks are identical - so it does not try; it refuses only the case a shape can prove.
`dir`	Directory to write into; created if it does not exist.
`colormap`	A list of RGB triples, one per class.
`names`	File names, or the source image paths to take names from. Defaults to `mask_0001.png` and so on.
`suffix`	Added before the extension, so predictions from different models can sit in one directory without overwriting each other.

Details

Files go where you say, not beside the images they came from. Writing into somebody's dataset directory is a surprising thing for a function to do, and a second run against a different model would quietly mix its results in with the first.

Value

The paths written, invisibly.

Examples

```
## Not run:
masks <- predict(fit, "unet", split = "test")
save_masks(masks, "predictions", binary_colormap, suffix = "_unet")

## End(Not run)
```

`save_spec` *Write a specification back out as YAML*

Description

`platypus_spec()` reads a YAML file and builds a specification from arguments. This is the way back, so the two are one pipeline rather than one and a half: a specification assembled in R becomes a file somebody else can run, or that a script can keep beside its results.

Usage

```
save_spec(spec, path, defaults = FALSE)
```

Arguments

`spec`	A `platypus_spec`, from `platypus_spec()`.
`path`	Where to write it. The parent directory must exist.
`defaults`	Write every field at its resolved value, including the ones you never set.

Details

It writes what you set, not what the specification holds. A model built from two arguments holds 29 fields once the engine has filled in its defaults, and writing all of them is not merely verbose - it changes what the file does. A model given `weights` and no `architecture` takes the architecture from the weights sidecar, and that adoption happens precisely because the field was never set. Freeze the defaults into the file and the adoption can never happen again: the file looks like the original and quietly stops doing what the original did.

`defaults = TRUE` is the opposite want, and a legitimate one: freeze this run exactly as it resolved so it reproduces years from now. The cost is the paragraph above, by design.

Value

The path, invisibly.

See Also

`platypus_spec()`, which reads one back, and `read_record()` for what a run *did* as opposed to what it was asked to do.

Examples

```
## Not run:
spec <- platypus_spec(
  data = segmentation_data("train/", "valid/", colormap = binary_colormap),
  models = list(u_net("unet", input_shape = c(256, 256)))
)

save_spec(spec, "experiment.yaml")
identical(as.list(platypus_spec("experiment.yaml")), as.list(spec))

# Everything the engine resolved, for a run that has to reproduce in five years.
save_spec(spec, "frozen.yaml", defaults = TRUE)

## End(Not run)
```

---

save_volumes *Save predicted volumes as NIfTI*

Description

Writes each mask as a label-map NIfTI carrying the geometry of the scan it was predicted from. That is the whole reason this is not `save_masks()` with a different extension: a mask array without an affine cannot be laid over its scan by any viewer, any registration tool or any volume calculation - it either lands in the wrong place or is refused.

Usage

```
save_volumes(masks, dir, reference, names = NULL, suffix = "")
```

Arguments

`masks`	Masks of class indices, as `predict()` returns. A **list** of volumes is what `predict(space = "source")` gives, and the form to prefer: each mask is already on the grid of the scan it was computed from, so the geometry matches by construction and the scans need not be the same size as each other. An array is also accepted: one volume as `depth x height x width`, a stack as `volume x depth x height x width`, probabilities as `volume x depth x height x width x class` which are collapsed to the most likely class. A single volume's probabilities have the same rank as a stack of masks and are not guessed at - collapse them yourself, or keep the volume axis.
`dir`	Where to write. Created if it does not exist.
`reference`	The scan each mask was predicted from, in the same order. Its affine is what the mask is written with, so this is required rather than optional.

names	File names, without extension. Taken from reference when unset, which works when the scans have distinct filenames. They often do not - one directory per case, the same ct.nii.gz inside each - and writing every mask to one name would silently keep only the last, so that case is an error asking for this argument. split_path(split, "validation")$key is usually the answer.
suffix	Appended to each name, for telling two models' output apart.

Details

Carrying the affine means the mask has to be on the same grid as the scan, which is what predict(space = "source") is for: it maps each prediction back out of the model's grid onto the scan's. Without it, a resampled or resized prediction will be refused here rather than written somewhere wrong.

Value

The paths written, invisibly.

See Also

save_masks() for 2D masks as pictures.

Examples

```
## Not run:
validation <- split_path(split, "validation")
masks <- predict(fit, split = "validation", space = "source")
save_volumes(masks, "predictions", reference = validation$images)

## End(Not run)
```

---

segmentation_data *Where the images and masks are*

Description

Two layouts. nested_dirs expects one directory per sample containing an images and a masks subdirectory - the Data Science Bowl arrangement, where a sample may hold dozens of separate mask files that are combined into one. config_file expects a CSV with images and masks columns, several paths per cell if needed, resolved relative to the file itself so a configuration travels with its data.

Usage

```
segmentation_data(
  train,
  validation = NULL,
  colormap = NULL,
  labels = NULL,
```

```
  test = NULL,
  split = NULL,
  mode = c("nested_dirs", "config_file"),
  window = NULL,
  dicom_window = NULL,
  target_spacing = NULL,
  channels_from = NULL,
  subdirs = c("images", "masks"),
  column_sep = ";",
  shuffle = TRUE
)
```

Arguments

`train`, `validation`
: Paths to the training and validation data. A `split_dataset()` may be given as `train` on its own: it carries all three paths and selects `config_file` mode, so a split needs no unpacking.

  `validation = FALSE` **says this run has no validation set**, which is the third and last answer to the question `validation` and `split` both answer - a final fit on every case you have, once the hyperparameters are settled. It goes in this argument rather than a new one because it is the same question, and a second argument could contradict the first.

  Leaving `validation` and `split` *both* unset is still an error, and that is deliberate: "I have no validation set" and "I forgot" look identical, and only one of them is a decision.

`colormap`
: A list of RGB triples, one per class, background first. For masks stored as pictures. Give this or `labels`, not both.

`labels`
: The voxel value of each class, in class order, for masks stored as label maps - `c(0, 1)` for a binary segmentation, `c(0, 1, 2)` where 1 is liver and 2 is tumour. This is how NIfTI and every other volume format label anything. Give this or `colormap`, not both.

  Include 0 unless the masks really do fill every voxel. A value matching no entry falls back to the first class, so a list without 0 trains the background as whatever that first class is. The engine refuses that combination rather than let it look plausible, which means the error arrives before the run and not in the scores.

`test`
: Optional test data. Only images are read from it.

`split`
: Divide `train` instead of naming a `validation` set: a list with `fractions` (two numbers, or three to cut a test set as well) and `group_by`. Exactly one of `split`, `validation`, and `validation = FALSE`.

  `group_by` **has to be given, even as** `NULL`**.** It is a regular expression read against each sample's name, and everything sharing a group lands in a single split - usually a patient, sometimes a study or a scanner. `NULL` divides by file instead, and is a perfectly good answer when the images are independent.

  It is required rather than optional because the mistake it prevents leaves no trace: slices of one patient in training and validation at once make validation measure

memory rather than generalisation, and the score comes out several points too high with nothing in the output to say so.

Nothing is written. split_dataset() is still the way when the three CSV files are the point - to keep, to hand to a colleague, to cite - and its result can be passed straight to train.

mode — "nested_dirs" or "config_file".

window — How values in real units reach the model. A named window - "lung", "soft_tissue", "bone", "brain", "abdomen", "liver", "mediastinum", "subdural", "stroke" - or an explicit c(centre, width) in Hounsfield units, or "auto" for the window the file recorded, which only DICOM has, or "full" for the whole range present. Applies to DICOM and to NIfTI; ignored for ordinary pictures, which are already 0-255.

Worth choosing rather than leaving: a fixed window is what makes two scans comparable. Scaling each scan by its own darkest and brightest voxel lets a single metal implant or marker rescale everything else in it.

NULL, the default, sends nothing and lets the engine decide - which also keeps ordinary image work running against an engine too old to know the setting exists.

dicom_window — The former name of window, still accepted. It was accurate while DICOM was the only format that needed it.

target_spacing — For volumes: resample every scan to this many millimetres per voxel, then centre-crop or pad to the model's input_shape. Without it a volume is simply resized into that shape, which is only harmless when every scan covers the same amount of the patient - and clinical scans do not. Forty slices of 1 mm is 40 mm of patient and forty of 2.5 mm is 100 mm, so resizing alone leaves the same organ a different size in each, and nothing in the data says so.

c(1, 1, 1) is the usual starting point. Ignored for 2D.

channels_from — For datasets that keep one channel per file: one pattern per channel, in channel order. BraTS ships four MRI sequences per patient - T1, T1 after contrast, T2 and FLAIR - because different tumour structures are visible in different sequences; satellite sets keep their bands apart the same way.

The order is stated rather than inferred, and that is the point. Sorted, the BraTS names come out flair, t1, t1ce, t2 - reproducible and anatomically meaningless. A model trained with FLAIR in channel one and used on data whose channel one is T1 returns a plausible answer with the right shape and the right range, and nothing further down can notice.

Patterns are Python regular expressions, like group_by in split_dataset(), and R's own quoting applies: c("_t1\\.nii", "_t1ce\\.nii", "_t2\\.nii", "_flair\\.nii"). Each must match exactly one of a sample's files - "_t1" would match both _t1.nii.gz and _t1ce.nii.gz, which is an error rather than a race - and every file must be claimed by some pattern, because one left out is data the model never sees.

subdirs — For nested_dirs, the names of the image and mask subdirectories.

column_sep — For config_file, what separates several paths in one cell.

shuffle — Shuffle the training data between epochs.

Details

Classes are named one of two ways, and exactly one of them. `colormap` is for masks stored as pictures: a list of RGB triples where position is the class index. `labels` is for masks stored as label maps - a vector of the voxel values, in class order - which is how every volume format stores them, and how a single-channel PNG can be read as well.

A sample that is missing its masks is an error, not a warning - a warning in a loop over five hundred directories is a warning nobody reads.

Value

A data specification, to be passed to `platypus_spec()`.

Examples

```
segmentation_data("train/", "valid/", colormap = binary_colormap)
```

`separations` *What the network predicts, so objects can be told apart*

Description

One per model, and it decides the head's width and which losses are admissible - both derived by the engine rather than stated here, so there is no channel count to get wrong. Pass the result as a model's `separation`; see `u_net()`.

Usage

```
separation_rim(width = NULL)

separation_flows(mode = NULL, loss = NULL, weight = NULL)
```

Arguments

`width`	For `rim`, how many pixels wide the rim is, **in the model's input resolution**. Unset uses the engine's default, which is 2 and stands on an argument rather than on a measurement: the published sweep peaked at 3 pixels of *source* resolution, which on a 520-pixel case read at 256 is about 1.5 pixels of model input, and a sweep in model pixels has not been run.
`mode`	For `flows`, `"diffusion"` or `"centroid"`. Unset uses the engine's default of `"diffusion"`, which stands on a measurement: decoded against the truth it reaches 0.9924 where `"centroid"` reaches 0.9478, because a straight line to a centroid leaves a crescent-shaped object.
`loss`	For `flows`, the objective for the vector channels, which has to be one defined over numbers rather than over classes. Unset uses the engine's default.
`weight`	For `flows`, how much of the total objective the vector half provides.

Details

`separation_rim()` trains three classes - background, object interior, object rim - and the rim is what separates two touching objects. `separation_flows()` trains a vector field pointing at each object's own sink, so every foreground pixel carries a value and none of them is spent on being a boundary.

What this is worth was measured and it is not flattering. On the Data Science Bowl's held-out cases the flow route is worth +0.0680 of instance average precision over `decode_connected_components()`, which ignores the signal entirely, and the rim route +0.0640 - while substituting a *perfect mask* is worth +0.2398. So the mask is the limit by a factor of 3.6, and a run that beats the published weights' 0.4888 without reaching 0.7655 has improved its mask and said nothing about separation.

There is no `hv`. Its markers lose to a one-pixel erosion on perfect inputs, 0.9382 against 0.9807, so naming it would promise a mechanism that was measured as absent.

Value

A separation block, to be passed as a model's `separation`.

See Also

decodings, `instance_data()`

Examples

```
separation_rim(width = 2)
separation_flows(mode = "diffusion")
```

---

series_report *Check folders of DICOM slices before training on them*

Description

One row per directory, saying whether it is a usable series and what is wrong with it if it is not. Problems are reported as a column rather than raised, because a hundred cases out of an archive will contain a few with a missing slice, two series in one folder, or duplicated files - and stopping at the first turns an afternoon's work into a week's.

Usage

```
series_report(paths)
```

Arguments

`paths`	Directories, each holding the slices of one series.

Details

Reads no pixels. Every check runs on the DICOM headers, so this is usable on a hundred gigabytes of data and is the first thing worth running on a new dataset.

Value

A data frame with one row per directory: `path`, `ok`, `slices`, `sorted_by`, `spacing_1/2/3`, `series_uid` and `problem`.

What it catches

A **missing slice**, which does not make a volume shorter - it puts everything past the gap in the wrong place, and no metric would show it. **Two series in one folder**, which is the normal state of an archive export: a scout, a reconstruction and a phase sitting together, which stacked would interleave two anatomies. **Duplicated slices**, from a copy or from several echoes mixed together. Slices of different **size**, **angle** or **spacing**.

It also reports how the slices were ordered. `"position"` means the geometry decided, which is what you want. `"instance_number"` means the files carry no positions and the order came from a number that is only required to be unique - worth knowing before trusting the result.

See Also

`volume_info()` for NIfTI files, `split_dataset()` for dividing the cases up.

Examples

```
## Not run:
cases <- list.dirs("dicom_export", recursive = FALSE)
report <- series_report(cases)

report[!report$ok, c("path", "problem")]     # the ones to deal with first
table(report$sorted_by)

## End(Not run)
```

`split_dataset` *Split a dataset into training, validation and test sets*

Description

Divides one directory of samples into three sets and writes each as a CSV the specification can point at. Nothing is copied: the files stay where they are and the CSVs list them, which matters when the data is large or read-only.

Usage

```
split_dataset(
  root,
  out_dir,
  group_by = NULL,
  fractions = c(0.7, 0.15, 0.15),
  seed = 0,
```

```
  mode = c("nested_dirs", "config_file"),
  subdirs = c("images", "masks"),
  column_sep = ";",
  strict = TRUE,
  relative = TRUE
)
```

Arguments

root	The directory holding the samples, laid out as mode describes.
out_dir	Where to write train.csv, validation.csv and test.csv.
group_by	A pattern picking the group out of a sample's name; see above. NULL treats every sample as its own group, which is right when one sample is one patient.
fractions	Two or three shares that add up to 1: train, validation and optionally test. Measured in samples, not in groups, so patients with very different numbers of slices do not skew them.
seed	Makes the split reproducible. The same inputs and seed give the same split on any machine and in any file order - without that, two runs are not comparable and the reason for a difference is impossible to find.
mode	"nested_dirs" for one directory per sample, "config_file" for a CSV.
subdirs	The image and mask subdirectory names, for nested_dirs.
column_sep	Separator between several paths in one CSV cell.
strict	Treat an incomplete sample as an error rather than skipping it.
relative	Write paths relative to the CSV where possible, so the dataset and its split can be moved together.

Value

A platypus_split: the three paths, plus how many samples and groups each set received. Pass it straight to segmentation_data().

Why group_by exists

Splitting per file is the obvious thing and, for medical images, the wrong one. A scan is many slices of one patient, so dividing slices at random puts the same anatomy in training and in validation at once. The model then meets the neighbouring slice of what it is being tested on, validation measures memory rather than generalisation, and Dice comes out several points too high with nothing in the result to say so.

group_by names the unit that must not be split - usually a patient, sometimes a study, a scanner or a site. Every sample sharing a group lands in exactly one set.

Patterns are Python regular expressions

group_by is passed to the engine exactly as written, and read there as a Python regular expression. In practice the dialects agree on everything you are likely to need

- ^, $, \\d, [A-Z], +, *, () - and R's own quoting rules still apply, so a backslash is doubled: "^(patient\\\\d+)_". It is not translated on the way through, deliberately: a translation that quietly got it wrong would produce a working split with the wrong groups in it, which is the failure this argument exists to prevent.

A pattern that matches no sample is an error, never a fallback to one-group-per-sample.

See Also

evaluate_cases(), which reports a score per case rather than one per set.

Examples

```
## Not run:
split <- split_dataset(
  "scans", "splits",
  group_by = "^(patient\\\\d+)_",
  fractions = c(0.7, 0.15, 0.15)
)
split

spec <- platypus_spec(
  data = segmentation_data(split, colormap = binary_colormap),
  models = list(u_net("unet", input_shape = c(256, 256)))
)

## End(Not run)
```

---

split_path *The files in one part of a split*

Description

Reads the CSV that split_dataset() wrote and hands back the paths ready to use.

Usage

```
split_path(split, which = c("train", "validation", "test"))
```

Arguments

split	A split_dataset().
which	"train", "validation" or "test".

Details

The reason this exists rather than read.csv(): the CSVs store paths **relative to themselves**, so that a dataset and its split can be moved or mounted elsewhere together. Read directly, those paths do not open from wherever your session happens to be - which is a trap worth removing rather than documenting.

Value

A data frame with `key`, `group`, `images` and `masks`, the paths made absolute. `images` and `masks` hold one path per sample, or several separated by the split's separator when a sample has several files.

Examples

```
## Not run:
split <- split_dataset("scans", "splits", group_by = "^(patient\\\\d+)_")
validation <- split_path(split, "validation")
images <- read_images(validation$images, size = c(32, 32, 32), channels = 1)

## End(Not run)
```

`training_history` *The epoch-by-epoch record*

Description

The epoch-by-epoch record

Usage

```
training_history(object)
```

Arguments

`object`	A `platypus_fit()`.

Value

A data frame with one row per model per epoch.

Examples

```
## Not run:
history <- training_history(fit)
head(history)

# For a detector the loss is broken into four terms on purpose: a run whose
# coordinate loss falls while its objectness loss does not is finding the right
# places and refusing to commit, and the reverse is confident nonsense.

## End(Not run)
```

unite_masks *Combine several binary masks into one*

Description

The Data Science Bowl stores one file per nucleus; datasets like it are common. Later masks win where they overlap, so an explicit class beats the background it sits on.

Usage

```
unite_masks(masks)
```

Arguments

masks	A list of `height x width` matrices of class indices.

Value

One `height x width` matrix.

Examples

```
# Data Science Bowl stores one file per nucleus, so a sample's mask arrives as several
# matrices that have to become one. Later masks win where they overlap.
one <- matrix(1L, 8, 8); one[2:4, 2:4] <- 2L
two <- matrix(1L, 8, 8); two[6:7, 5:7] <- 2L
united <- unite_masks(list(one, two))
table(united)
```

volume_info *What is in a volume file*

Description

Shape and voxel spacing, read through the engine so the numbers are the ones training would use - including the canonical reorientation, which changes what "the first axis" means.

Usage

```
volume_info(paths)
```

Arguments

paths	One or more paths to NIfTI files.

Value

A data frame with one row per file: `path`, `spacing_1/2/3` in millimetres, `shape_1/2/3` in voxels, and `voxel_ml`, the volume of one voxel in millilitres.

See Also

`mask_volume()`, which turns a segmentation into millilitres.

Examples

```
## Not run:
volume_info("scans/case_01/images/ct.nii.gz")

## End(Not run)
```

`yolo3` *YOLOv3*

Description

One detector, as an entry in `platypus_spec()`'s `models`.

Usage

```
yolo3(
  name,
  input_shape = c(416, 416),
  channels = NULL,
  anchors = NULL,
  anchors_per_grid = NULL,
  ignore_threshold = NULL,
  score_threshold = NULL,
  nms_threshold = NULL,
  operating_point = NULL,
  min_visibility = NULL,
  box_loss = NULL,
  optimizer = NULL,
  callbacks = NULL,
  augmentation = NULL,
  epochs = NULL,
  batch_size = NULL,
  weights = NULL,
  fit = NULL
)
```

Arguments

`name`	Unique within a spec; names the outputs.
`input_shape`	`c(height, width)`, each a multiple of 32. The three grids are the input divided by 32, 16 and 8, so a side that does not divide gives a coarsest grid that does not tile the image. Bigger is not obviously better: on BCCD, 608 bought 0.014 of mAP@0.5 for twice the training time, and the grid does not quantise a box's centre - the strides are fixed and the offset is a continuous output, so a larger input buys more cells rather than finer ones.
`channels`	Channels the model takes.
`anchors`	Box shapes to predict offsets from, **as fractions of the input size**: a list of three groups, coarsest grid first, each group a list of `c(width, height)` pairs. COCO publishes its nine in pixels at a 416 input, so those need dividing by 416. `NULL` fits them to your data.
`anchors_per_grid`	How many anchors to fit per grid when `anchors` is unset. It also sets the head's width, which is `anchors_per_grid * (n_class + 5)`.
`ignore_threshold`	A cell that is not responsible for an object but whose box overlaps one this much is left unsupervised rather than trained towards "no object". Without it, a nearly-correct guess beside the responsible cell is punished, which is the one thing a detector should not learn.
`score_threshold`	Discard a predicted box below this confidence. Low on purpose: average precision is computed over the whole ranking, so a cut here is a ceiling on recall that no threshold chosen later can lift.
`nms_threshold`	Two boxes of the same class overlapping this much are one object and the lower-scoring one is dropped. Per class, so a platelet on a red cell survives.
`operating_point`	The confidence at which precision and recall are reported. Separate from `score_threshold` because they answer different questions: that one decides what is computed at all, this one decides where on the curve to stand.
`min_visibility`	How much of a box must survive an augmentation that removes part of the frame, as a fraction of its original area. A convention rather than a measurement; what is defensible is the direction. A box keeping two pixels of a cell teaches the model that a two-pixel fragment is a whole cell, which produces false positives everywhere, while dropping a heavily truncated object only fails to teach it about that object. Irrelevant unless a transform can lose part of the frame, which flips and rotations never do.
`box_loss`	How the box coordinates are scored: `"offsets"`, YOLOv3's own, or `"giou"`. Unset uses the engine's default, which is `"offsets"` - what the published `bccd-yolo3` weights were trained with. `"giou"` **is not the better objective, and that is measured.** Blood cells, three seeds each, 150 epochs, on the dataset's own test split:

box_loss	mAP@0.5	mAP@[.50:.95]	matched IoU

`giou`	0.8666 ±0.0147	0.5078 ±0.0083	0.7929 ±0.0069
`offsets`	0.8583 ±0.0167	0.5172 ±0.0195	**0.8038 ±0.0001**

Average precision separates them in neither direction - both gaps are smaller than the seed spread. The overlap of the boxes it matched does separate them, and `"giou"` is worse by 0.011, which is 2.7 standard errors of the difference. It is also sixty times less repeatable on exactly that quantity.

The reason is visible in `detection_anchors()`: GIoU exists because plain IoU is a flat zero for boxes that do not touch, so it has no gradient where a detector is most wrong - and anchors fitted to blood cells already cover them at a mean IoU of 0.877, so a prediction never starts disjoint and the advantage never arrives. On data whose fitted anchors cover poorly it should be a different story, which is untested and so not claimed.

Choose it to read the loss, not to raise the score. The offsets term cannot reach zero - cross-entropy against a soft target bottoms out at that target's entropy - so a converged run and a stalled one print the same number, near 5.9. Under `"giou"` the coordinate term in `training_history()` reads about 0.05 on training against 0.47 on validation: a localisation gap you can see, because zero means the boxes are right.

`optimizer`, `callbacks`, `augmentation`, `epochs`, `batch_size`

As for `u_net()`. There is no `loss` and no `metrics`: YOLOv3's objective is part of its architecture, and mean average precision is not one option among several. A field that accepts a value and ignores it is worse than no field.

`weights` A registry name such as `"bccd-yolo3"`, an `hf://owner/repo/file@commit` reference, or a path to a local file. The anchors come with it.

`fit` Set `FALSE` to load weights and skip training, which is how you get boxes on an image without a GPU and an afternoon.

Details

Anchors. Unset, they are fitted to your training boxes with k-means under an IoU distance, which is what you want: COCO's nine anchors borrowed for blood cells cover their boxes at a mean overlap of 0.65 against 0.88 for anchors fitted to them. Fitted anchors are recorded with the run and in the sidecar when weights are exported, because **a detector cannot be reloaded without them** - the same weights read with other anchors decode every box scaled by a fixed factor, with plausible boxes, plausible scores, wrong places, and nothing in the output to say so.

For the same reason, naming `weights` and `anchors` together is an error: loading takes the anchors recorded beside the file, which would leave the ones here describing nothing.

Value

One model specification.

See Also

`detection_data()`, `plot_boxes()`, `available_weights()`

Examples

```
yolo3("cells", input_shape = c(416, 416))
yolo3("cells", input_shape = c(416, 416), weights = "bccd-yolo3", fit = FALSE)
```

Index

as.list.platypus_spec, 3
augmentation_step, 4
augmentation_step(), *7*, *41*
available_callbacks (available_components), 4
available_components, 4
available_losses (available_components), 4
available_losses(), *41*
available_metrics (available_components), 4
available_models (available_components), 4
available_models(), *52*
available_optimizers (available_components), 4
available_optimizers(), *45*, *46*
available_tasks (available_components), 4
available_transforms, 6
available_transforms(), *4*, *6*
available_weights, 8
available_weights(), *4*, *6*, *80*

binary_colormap, *12*
binary_colormap (colormaps), 12
binary_labels (colormaps), 12

callback_cosine_annealing (callbacks), 10
callback_cosine_annealing(), *9*
callback_csv_logger (callbacks), 10
callback_early_stopping (callbacks), 10
callback_model_checkpoint (callbacks), 10
callback_reduce_lr_on_plateau (callbacks), 10
callback_swa, 9
callback_swa(), *11*
callback_terminate_on_nan (callbacks), 10
callbacks, 10, *10*, *41*, *43*
callbacks(), *46*
classes_to_colours, 12
classes_to_colours(), *13*
colormaps, 12
colours_to_classes, 13
colours_to_classes(), *63*
ct_windows, 14

decode_connected_components (decodings), 14
decode_follow_flows (decodings), 14
decode_nearest_seed (decodings), 14
decode_watershed (decodings), 14
decodings, 14, *42*, *72*
detection_anchors, 15
detection_anchors(), *57*
detection_crops, 16
detection_data, 18
detection_data(), *52*, *80*
drawing_style, 20

encoders, 21, *41*
evaluate, 23
evaluate(), *14*, *19*, *25*, *42*
evaluate_cases, 24
evaluate_cases(), *24–27*, *60*, *75*
evaluate_classes, 25
evaluate_classes(), *27*
evaluate_images, 26
export_weights, 27
export_weights(), *15*, *64*

inspect_masks, 28
inspect_masks(), *51*
instance_data, 29
instance_data(), *15*, *42*, *52*, *72*

linknet (models), 37

loss_boundary, 31
loss_boundary(), *34*
loss_cce *(*losses*)*, 32
loss_cce_dice *(*losses*)*, 32
loss_combo *(*losses*)*, 32
loss_dice *(*losses*)*, 32
loss_focal *(*losses*)*, 32
loss_focal_tversky *(*losses*)*, 32
loss_iou *(*losses*)*, 32
loss_l1 *(*losses*)*, 32
loss_lovasz *(*losses*)*, 32
loss_mse *(*losses*)*, 32
loss_smooth_l1 *(*losses*)*, 32
loss_tversky *(*losses*)*, 32
losses, 32, *32*, *41*

mask_coverage, 34
mask_coverage(), *13*
mask_volume, 35
mask_volume(), *9, 10*, *32*, *78*
metric_cldice *(*metrics*)*, 36
metric_dice *(*metrics*)*, 36
metric_iou *(*metrics*)*, 36
metric_tversky *(*metrics*)*, 36
metrics, 36, *41*
models, *22*, 37, *52*

optimizer_adadelta *(*optimizers*)*, 43
optimizer_adagrad *(*optimizers*)*, 43
optimizer_adam *(*optimizers*)*, 43
optimizer_adamax *(*optimizers*)*, 43
optimizer_adamw *(*optimizers*)*, 43
optimizer_nadam *(*optimizers*)*, 43
optimizer_rmsprop *(*optimizers*)*, 43
optimizer_sgd *(*optimizers*)*, 43
optimizers, *41*, 43
overlay_agreement, 46
overlay_agreement(), *20*
overlay_mask, 47

platypus_conditions, 48
platypus_device, 50
platypus_device(), *52*
platypus_fit, 50
platypus_fit(), *16, 17*, *23*, *25, 26*, *28, 29*, *32*, *55, 56*, *60*, *76*
platypus_spec, 51
platypus_spec(), *6*, *19*, *22*, *28*, *30, 31*, *42*, *51*, *66*, *71*, *78*
platypus_status, 54
platypus_status(), *49*
platypus_use_torch, 54
platypus_use_torch(), *22*
plot.platypus_fit, 55
plot_anchors, 56
plot_anchors(), *20*
plot_boxes, 57
plot_boxes(), *20*, *80*
plot_masks, 58
plot_masks(), *20*, *57, 58*
predict(), *16, 17*, *35*, *57, 58*, *67*
predict.platypus_fit, 60
predict.platypus_fit(), *58, 59*, *65*

read_images, 62
read_images(), *57*
read_masks, 62
read_record, 63
read_record(), *66*
res_u_net *(*models*)*, 37

save_masks, 65
save_masks(), *68*
save_spec, 66
save_volumes, 67
save_volumes(), *61*, *65*
segmentation_data, 68
segmentation_data(), *14*, *18, 19*, *30, 31*, *40*, *52*, *74*
separation_flows *(*separations*)*, 71
separation_flows(), *34*, *42*
separation_rim *(*separations*)*, 71
separation_rim(), *42*
separations, *15*, 71
series_report, 72
split_dataset, 73
split_dataset(), *19*, *24, 25*, *30*, *52*, *69, 70*, *73*, *75*
split_path, 75
summary.platypus_cases *(*evaluate_cases*)*, 24

training_history, 76
training_history(), *64*, *80*

u_net *(*models*)*, 37
u_net(), *8*, *71*, *80*
u_net_plus_plus *(*models*)*, 37

unite_masks, 77

voc_colormap *(*colormaps*)*, 12
voc_labels *(*colormaps*)*, 12
volume_info, 77
volume_info(), *35*, *73*

yolo3, 78
yolo3(), *20*, *57*