Package: rplatypus (via r-universe)

November 6, 2026

Type Package

Title Computer Vision for Medical Imaging

Version 0.3.0

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 (>= 3.11). 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, codetools, jsonlite, knitr, pkgload, png, RNifti, rmarkdown, testthat (>= 3.0.0), withr, yaml

Config/testthat/edition 3

VignetteBuilder knitr

Config/roxygen2/version 8.1.0

Roxygen list(markdown = TRUE)

Config/pak/sysreqs libpng-dev python3

Repository https://maju116.r-universe.dev

Date/Publication 2026-11-05 22:02:59 UTC

RemoteUrl https://github.com/maju116/rplatypus

RemoteRef HEAD

RemoteSha 52534ef353595f36017fc7d852bdfb119ee189bb

Contents

as.list.platypus_spec . . . 3
augmentation_step . . . 4
available_components . . . 5
available_transforms . . . 7
available_weights . . . 8
callback_swa . . . 9
callbacks . . . 11
class_balance . . . 16
classes_to_colours . . . 17
classification_data . . . 17
classification_metrics . . . 20
classifier . . . 21
colormaps . . . 24
colours_to_classes . . . 25
ct_windows . . . 25
decodings . . . 26
detection_anchors . . . 27
detection_crops . . . 28
detection_data . . . 30
drawing_style . . . 32
ellipse_measure . . . 32
encoders . . . 34
evaluate . . . 36
evaluate_cases . . . 38
evaluate_classes . . . 39
evaluate_images . . . 40
export_weights . . . 42
flow_to_colours . . . 43
inspect_masks . . . 44
instance_data . . . 45
instance_shapes . . . 47
instance_sizes . . . 48
loss_boundary . . . 49
losses . . . 50
mask_coverage . . . 53
mask_volume . . . 54
metrics . . . 55
models . . . 57

optimizers . . . 64
overlay_agreement . . . 67
overlay_instance_agreement . . . 68
overlay_instances . . . 69
overlay_mask . . . 70
platypus_conditions . . . 71
platypus_device . . . 73
platypus_fact . . . 74
platypus_fit . . . 75
platypus_spec . . . 76
platypus_status . . . 78
platypus_use_torch . . . 79
plot.platypus_fit . . . 81
plot_anchors . . . 81
plot_boxes . . . 83
plot_masks . . . 84
predict.platypus_fit . . . 85
read_images . . . 89
read_masks . . . 90
read_record . . . 91
save_class_masks . . . 92
save_class_volumes . . . 94
save_masks . . . 95
save_spec . . . 96
save_volumes . . . 97
segmentation_data . . . 99
separations . . . 102
series_report . . . 104
split_dataset . . . 105
split_path . . . 107
training_history . . . 108
unite_masks . . . 109
volume_info . . . 109
yolo3 . . . 110

Index **114**

`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()

available_separations()

available_decodings()
```

Arguments

task	`"semantic_segmentation"`, `"instance_segmentation"`, `"object_detection"` or `"classification"`. 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, and classification lists two of the eleven - the ones that score each answer on its own rather than scoring an overlap. 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`, three until it moved to `0.8.0a49`; the third arrived in the engine's `0.8.0a20` and the fourth in `0.8.0a49`.

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.

Two listings carry one column more, and it is the column they are asked for: `available_decodings()` has `needs`, the separation a decoding requires - empty where it requires none, which is `connected_components` and is why that one is always available - and `available_separations()` has `head_channels`, how wide a head that target makes. Taken from the engine's own rows rather than named on this side, so a listing that grows a field arrives here without R changing.

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.

`available_models("classification")` **raises for the same kind of reason and a different fact.** A classifier chooses no architecture: the network is a contracting path, pooled, with one dense layer on top, and `encoder` decides whose contracting path it is - ours when it is unset, any of timm's when it names a backbone. So there is no `architecture` field to write and a row here would invite one. encoders is the listing that question actually has.

See Also

`available_transforms()` and `available_weights()`, the two that came first, separations and decodings for the constructors these two listings name, and `platypus_spec()`, where all of them 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()

# The two instance axes: what a model can predict, and how those channels become objects.
available_separations()[, c("name", "head_channels")]
available_decodings()[, c("name", "needs")]

# 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, start_epoch = NULL)
```

Arguments

`start`, `start_epoch`

When averaging begins - **one of the two and never both**. `start` is the fraction of the run after which it begins, 0.75 folding the last quarter, and unset uses the engine's default of 0.75; `start_epoch` says the same thing as an epoch number. A fraction survives a change to `epochs` and an epoch count does not, which is why the fraction is the default; an epoch is the one to use when the point is a property of the optimisation rather than of the run's length. The final epoch is always folded, so `start = 1` averages one set of weights rather than none. `callback_early_stopping()` names the same pair with the same two words.

`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,
  start = NULL,
  start_epoch = NULL
)

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

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_linear_annealing(min_lr = 0, epochs = NULL)

callback_step_decay(factor = 0.1, every = NULL, milestones = NULL, min_lr = 0)

callback_csv_logger(path)

callback_ema(decay = 0.999)

callback_stop_after(hours)

callback_terminate_on_collapse(
```

```
  floor = 1e-04,
  patience = 3,
  monitor = "val_foreground"
)

callback_terminate_on_nan()
```

Arguments

monitor	What to watch: `"val_loss"`, `"train_loss"`, `"val_"` followed by a metric name such as `"val_dice"`, `"val_foreground"` and `"train_foreground"` - the share of the prediction that is not background - or `"val_dice_worst"` **and its kind, the lowest any single case scored.** That last one is the number `summary()` on `evaluate_cases()` tells you to read, and until the engine's `0.8.0a36` nothing could act on it: a model averaging 0.86 that scores 0.11 on some patient has a failure mode, and the mean is where it hides. Stopping, checkpointing and cutting the learning rate can all watch it now. Validation only. The training data is shuffled between epochs, so a case has no identity in that pass and there is nothing honest to pool by - `"train_dice_worst"` is refused rather than quietly reported as something else.
patience	Epochs to wait before acting. For `callback_terminate_on_collapse()` that is epochs in a row below the floor, and it is not 1 because an early epoch legitimately predicts almost nothing.
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. **An averager beside this one wins.** `callback_ema()` and `callback_swa()` also replace the weights when training ends, and since engine `0.8.0a57` the order is fixed rather than left to wherever the callbacks sat in the list: the average is what the run hands back, because somebody who configured averaging asked for the averaged model and this argument is a statement about which *epoch* was best. Before that release the answer was whichever of the two this package happened to send last, and the run record reported the EMA steps either way. Re-read here because the engine rewrote its own sentence for this field, which is what the field-prose ledger is for.
start, start_epoch	When the callback may begin to act - **one of the two and never both**. `start` is a fraction of the run (`0.1` leaves the first tenth alone) and `start_epoch` is an epoch number. Two arguments rather than one taking either, because a fraction survives a change to `epochs` and an epoch count does not, and in a YAML specification `1` and `1.0` are the same value once parsed - so one field would make "epoch one" and "the whole run" indistinguishable. For `callback_early_stopping()` this is a **warm-up**: a model that needs fifteen epochs before its validation curve means anything would otherwise need `patience` > `15` for the whole run, paying that slack at the end as well. It delays

the stopping and nothing else - an epoch inside the warm-up can still be the best one and `restore_best` still returns to it, because those are the best weights the run has if it ends for another reason.

For `callback_swa()` it is the point averaging begins, which is the same concept and therefore the same two words.

`path` Where to write. For `callback_model_checkpoint()` that is the weights file; for `callback_csv_logger()` it is one row per epoch appended as the epoch ends - the same numbers `training_history()` returns, so a run that is interrupted still leaves its history behind. Re-read against the engine's own text when it changed in `0.8.0a23`, which is what the field-prose ledger is for: the engine's sentence had named this package's `training_history()` to a Python reader, and the fix there left R's `@param` saying three words about two different files.

The checkpoint's extension decides its format, since the engine's `0.8.0a31`: `.safetensors` is written as safetensors and anything else as a torch archive, which is what `.pt` has always meant. Before that the file was a torch archive whatever it was called, so a checkpoint named after the files this package publishes was written in one format and read as another - and reading now goes by the file's contents rather than by its name, so a checkpoint written by an older engine still loads. A checkpoint is the run's safety net; `export_weights()` is what writes a file for somebody else to load, with the sidecar that says what it is for.

`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.

`when` For `callback_model_checkpoint()`, a named list of thresholds that must **all** hold before anything is written: `list(val_dice = 0.9, val_loss = 0.2)`. **The direction is derived, not configured** - anything ending in `loss` has to be below its threshold and everything else above it, the same rule `monitor` uses - so there is no argument to get backwards. The names are checked against what the model reports when the specification is built, not at the first epoch.

It is an **extra gate beside** `save_best_only`, not a replacement: with both, a file is written when the watched number improves *and* every threshold holds. A threshold is a number about your data and does not transfer between datasets, and a run whose conditions never hold writes nothing and says so when it ends.

`factor`, `min_lr` What the rate is multiplied by, and the floor it stops at. `factor` is the cut on a plateau for `callback_reduce_lr_on_plateau()` and the drop at each step for `callback_step_decay()`, where it must be below 1 - a step schedule that raises the rate is a different idea and accepting it here would hide a typo. `min_lr` is the floor for all four.

`epochs` For `callback_cosine_annealing()` and `callback_linear_annealing()`, how many epochs to spread the decay over. Unset means the model's `epochs`, which is what you want: a curve that ends where the run ends.

`every`, `milestones`

For `callback_step_decay()`, **one of the two and never both**. `every = 30` drops every thirty epochs; `milestones = c(30, 60)` drops at those epochs, 1-based and strictly increasing, which is how a published recipe states it. Two

arguments rather than one that takes either, because `30` and `c(30)` are different schedules and a single argument holding either would be read as whichever was typed. A period survives a change to `epochs`; a list of milestones does not.

`decay` For `callback_ema()`, how much of the average survives each optimiser **step** - not each epoch. 0.999 is the usual value and gives a half-life of about 700 steps; it is the published number because the average is fed per step, which is what the engine's fourth callback hook was added for. Unlike `callback_swa()` this needs no pass over the training data at the end: the batch-normalisation statistics are averaged alongside the weights rather than recomputed.

`hours` For `callback_stop_after()`, a wall-clock budget. An epoch cap cannot express it, because how long an epoch takes is a property of the data and the shape. Checked at the end of an epoch, so the real limit is this plus one epoch; the run's stop reason names the clock, so a run that ran out of time does not read as one that converged.

`floor` For `callback_terminate_on_collapse()`, the share of foreground below which a prediction counts as collapsed. A floor rather than a test for zero, because 0.01% foreground against a target that is 14% foreground has collapsed in every sense that matters. The default sits far below any real target on purpose: this guards against a failure, not against a model being merely bad, and a floor near a real foreground share would end runs that were only disappointing.

Details

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

Value

A callback specification.

The one number that is not about quality

Every epoch reports `foreground` as well - for both passes, `train_foreground` and `val_foreground` - which is the share of the prediction that is **not background**. Nothing is compared against the truth, so it is not a measure of how good the model is; it answers whether the model is still producing a segmentation at all.

It is reported whether or not you ask for it, and that is the point. A model that collapses to predicting background everywhere is invisible in the numbers it already produced: with `include_background = TRUE` the Dice is a mean over class channels and the background channel is nearly perfect, so a collapsed model reports around 0.5 while its loss falls contentedly. A collapse you have to opt into noticing is one you will not notice.

`callback_terminate_on_collapse()` acts on it, `evaluate()` carries it as a column, and `callback_model_checkpoint()`'s `when` can take a condition on it.

There is a twelfth, 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.

One averager per run

callback_swa() and callback_ema() both replace the model's weights when training ends, so a run takes **one of them**; the engine refuses both. They answer the same observation - the last epoch's corner is not the middle of the basin - with different weightings. swa folds the tail uniformly from a declared start; ema weights recent steps more and needs no start, because an average with a decay forgets the beginning by itself.

ema is updated **after every optimiser step**, which is what makes its decay mean what the literature means by it: 0.999 is 0.999 per step, so over 60 epochs of 200 batches that is 12,000 updates and a half-life of roughly 700 steps. A run with few steps wants a smaller number, because an average that forgets more slowly than the run is long never leaves where it started.

One hand on the learning rate

Three of these compute the rate from the epoch rather than from how the run is going - callback_cosine_annealing(), callback_linear_annealing() and callback_step_decay() - and **at most one of them may be used**, because each recomputes the rate every epoch and two of them would simply overwrite each other.

For the same reason a schedule and callback_reduce_lr_on_plateau() **cannot be combined**, and the engine refuses the pair. That refusal replaced a sentence claiming the opposite: a schedule recomputes the rate from the optimiser's initial value at the end of every epoch, so it erases whatever the plateau callback cut. Measured on six epochs of a loss that never improves, the rate with both configured follows the cosine exactly - 0.0976, 0.0905, 0.0794 from 0.1 - where the plateau alone reaches 1e-6.

A schedule beside callback_swa() is fine and the order is no longer luck: swa pins its constant rate after averaging begins, whatever position the callbacks sit in.

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)
```

`class_balance` *How many images each class has, before anything is trained*

Description

The number every decision about this task's metrics turned on, and the person who most needs it is the one who has just written a specification and not yet run anything. A prevalence a reader does not know is one they will discover from a score that looks fine.

Usage

```
class_balance(spec, split = "train")
```

Arguments

`spec`	A `platypus_spec()` describing a classification task.
`split`	Which split to count: `"train"`, `"validation"` or `"test"`. A split that carries no labelling is **refused by name** rather than counted - which is the ordinary shape of a test set, since `0.8.0a52` reads a CSV of `key, images` as a prediction split. Counted anyway it would report every class at zero and read as a finding about the dataset rather than about there being nothing to count.

Details

The sentence under the table is the **engine's own**, printed rather than reproduced: the threshold it fires below is the engine's number, and a copy of it here would be a second place for one fact to live.

Value

A data frame of one row per class, in class order, with columns `class`, `n` and `share`, of class `platypus_class_balance` so that printing it adds the engine's own remark about how lopsided the split is. The engine's sentence is carried on the `remark` attribute.

See Also

`classification_data()`, `split()`

Examples

```
## Not run:
spec <- platypus_spec(
  data = classification_data("train/", "valid/", classes = c("benign", "malignant")),
  models = list(classifier("cnn", input_shape = c(128, 128)))
)
class_balance(spec)

## End(Not run)
```

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))
```

classification_data *Images and the label each of them carries, which may be a set*

Description

The data block for `task = "classification"`. This title said *the one label each of them carries* until `labels_from` arrived, and the field-prose ledger is what asked: the engine rewrote its own one-line description of this block to name all three labellings, and re-reading R's prose for it turned up a heading that had become false rather than a sentence that had.

Usage

```
classification_data(
  train,
  validation = NULL,
  classes,
  labels_from = NULL,
  test = NULL,
  split = NULL,
  mode = c("nested_dirs", "config_file"),
```

```
  strict_labels = TRUE,
  window = NULL,
  subdirs = NULL,
  column_sep = ";",
  shuffle = TRUE
)
```

Arguments

`train`	Path to the training data: a directory of class directories in `"nested_dirs"` mode, or a CSV in `"config_file"` mode.
`validation`	Path to the validation data, or `FALSE` to train without one.
`classes`	The class names, in class order. **Position is the class index the model learns**, which is why it is written rather than read off the directory listing: taken from the filesystem the order would be alphabetical - reproducible, and unrelated to anything - so a model trained where index 1 meant `"benign"` and used where it means `"malignant"` reports plausible probabilities and nothing downstream can tell. In `"nested_dirs"` mode these are the directory names. At least two, where `detection_data()` allows one: a detector with one class answers *where are the cells*, which is a question, and a classifier with one class answers *is this a cell* with the only answer it has. There is no background entry. An image is one of these, and a picture of nothing in particular is a class somebody has to name.
`labels_from`	One CSV column per class, in class order, each cell `0` or `1` - the layout for a label that is a **set**. Position in this list is paired with `classes`, so the headers may be called whatever the dataset calls them and the model still learns the classes in the order you named; stated rather than read off the header row for `channels_from`'s reason, that sorting is reproducible and clinically meaningless. `"config_file"` mode only, and that is a refusal rather than a gap: an image cannot sit in two directories without being two files, so `nested_dirs` cannot express a set of labels at all. `strict_labels` has nothing to choose between here - the columns *are* the classes - and is refused rather than ignored. A cell that is not `0`, `1`, `0.0` or `1.0` is refused by name, and the case that is written for is `-1`: the datasets shaped this way write `-1` for *uncertain* and leave the cell empty for *not mentioned*, both of which have a defensible mapping to 0 or to 1, which is exactly why the reader must not pick one for you. An image with none of the findings is an ordinary labelled row, written as a line of zeros. **There is no threshold, and `metric_accuracy()` is refused.** Each class decides for itself at a probability of 0.5, and a threshold is a number about your data rather than one this format should invent. Accuracy is refused outright rather than redefined: measured on 4000 images with findings at 10%, 5%, 3% and 2%, a model answering no to everything scores 0.8137 on exact match and 0.9507 per class, **above a model holding half the available signal** - so neither reading can be what a `monitor` watches. Use `metric_balanced_accuracy()`, `metric_f1()` or `metric_auc()`.

`test`	Path to a held-out set, scored but never trained on.
`split`	A `split()` block, to cut a validation and test set out of the training data. For this task the cut is **stratified** by the engine, so every class reaches every side - a random 70/15/15 of 100 images at a 3% prevalence leaves some part with none of the rare class 91% of the time, and a per-class metric cannot average over a class its pass never held.
`mode`	`"nested_dirs"` or `"config_file"`.
`strict_labels`	Refuse an image whose label is not in `classes`. `FALSE` skips it instead, which is how to train on two of a dataset's three classes without moving any files. It matters more here than it does for a box: a skipped object leaves the rest of its image in the training set, while a skipped image leaves nothing, so the same setting quietly changes how much data there is. Refused beside `labels_from`, where the state it chooses between cannot arise: the columns are the classes, so there is no cell that could hold a class nobody named.
`window`	How values in real units reach the model, for DICOM and NIfTI - see `segmentation_data()`, which documents the named windows. Ignored for ordinary pictures.
`subdirs`	Refused, and named rather than dropped because every sibling data constructor has it. For segmentation and detection a sample is a directory with the image in one subdirectory and what labels it in the other; a classification sample is one image whose label is the directory it sits in, so there is no second subdirectory.
`column_sep`	For `config_file`, what separates several paths in one cell.
`shuffle`	Shuffle the training data between epochs.

Details

What this constructor is *for* is as much the fields it has not got as the two it adds: `colormap`, `labels`, `regions`, `classes_from`, `channels_from` and `target_spacing` all describe a mask or a grid, and a label is neither.

Where the labelling is written follows from `mode` rather than being said twice. In `"nested_dirs"` mode the directories under `train` are the classes, with the images inside them - the layout nearly every image dataset ships in. In `"config_file"` mode the CSV's `label` column holds the class name.

And `labels_from` is the third way, for a label that is a set rather than one of a list of alternatives - a chest film with four findings is four independent questions. It is a naming rather than a switch, for the reason `regions` and `classes_from` are on the mask side: the *target* changes and not only the activation, so there is nothing a `multilabel = TRUE` could have meant on a `label` column that structurally cannot say an image has two findings.

Value

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

See Also

`classifier()`, `class_balance()`, `available_metrics()`

Examples

```
classification_data("train/", "valid/", classes = c("benign", "malignant"))
```

`classification_metrics` *Metrics for one label per image*

Description

What to report beside the loss for `task = "classification"`. These are a different family from metrics rather than more of it: there is no overlap to measure, no background class to leave out, and no `smooth` - a label is right or it is not.

Usage

```
metric_accuracy()

metric_balanced_accuracy()

metric_f1()

metric_auc()
```

Details

They are also scored differently, and that is measured rather than stylistic. An overlap metric is a formula on counts, so averaging a batch's scores is the score; three of these four are not means over images at all. On 4000 images at a prevalence of 3%, against a model whose own probabilities are the truth, pooling the whole pass against averaging batches of 8 moves balanced accuracy by 0.0035, macro F1 by 0.0734, and the AUC by up to 0.0371 in a direction that does not settle with a bigger batch - because at that prevalence 398 of 500 batches hold no positive at all, so a per-class recall is an average over whichever batches happened to contain the rare class. The engine therefore accumulates these over the pass and computes them once at the end.

`metric_auc()` **is the engine's default and the other three are offered rather than recommended.** At a 3% prevalence a calibrated model never crosses the threshold, so everything read off the hard prediction sees the same all-negative answer: accuracy spans 0.0100 across models of genuinely different quality and is not even monotone - a model with no signal scores above two that have one - while the AUC spans 0.4631. Balanced accuracy and F1 do not rescue it; they move by 0.0000 and -0.0005 between a real signal and a collapse. Being threshold-free is the property that separates them.

So `metric_accuracy()` is here because it is the number people ask for, and its page says what it costs.

Value

A metric specification.

See Also

classifier(), metrics for masks, class_balance() for the prevalence these measurements are all about

Examples

```
metric_auc()

# The number people ask for, beside the one that can tell a signal from a collapse.
classifier("cnn", input_shape = c(128, 128),
           metrics = list(metric_auc(), metric_accuracy()))
```

---

classifier *A contracting path, pooled, with one dense layer on top*

Description

The model block for task = "classification", and **the one model constructor in this package that is not named after an architecture.** u_net(), linknet(), res_u_net() and u_net_plus_plus() choose between four decoders that read the same features; a classifier has nothing after the pooling but a dense layer, so the contracting path *is* the architecture and encoder is the whole answer. The engine has no architecture field here for that reason, and the name follows its own build_classifier.

Usage

```
classifier(
  name,
  input_shape,
  channels = NULL,
  blocks = NULL,
  filters = NULL,
  block_width = NULL,
  pooling = NULL,
  head_dropout = NULL,
  dropout = NULL,
  batch_normalization = NULL,
  separable_conv = NULL,
  spatial_dropout = NULL,
  activation = NULL,
  initialiser = NULL,
  loss = NULL,
  metrics = NULL,
  optimizer = NULL,
  callbacks = NULL,
  augmentation = NULL,
  epochs = NULL,
```

```
  batch_size = NULL,
  weights = NULL,
  fit = NULL,
  encoder = NULL,
  pretrained = NULL,
  normalise = NULL,
  freeze_encoder = NULL,
  encoder_learning_rate = NULL
)
```

Arguments

`name`	A name for the model, used in the score table and in file names.
`input_shape`	The size every image is read at, `c(height, width)`. Unlike a segmentation model nothing has to divide by `2^blocks` - there is no decoder to line up with - but every side must still be at least `2^blocks`, or the deepest level has nothing left to pool.
`channels`	Input channels: 3 for colour, 1 for greyscale.
`blocks`	How many times the resolution is halved before the features are pooled, so how much context the answer is made from. Ignored, and refused, when `encoder` names a backbone - the backbone's own depth is then the only one there is.
`filters`	Filters in the first block, doubled at every level, so this and `blocks` set how wide the pooled feature vector is: `filters * 2^blocks`. Ignored, and refused, with `encoder`.
`block_width`	Convolutions per block, at every level. Raising it adds depth without adding levels, which is the lever to reach for when the objects are small enough that halving the resolution again would lose them - a crop of one cell is exactly that case. Ignored, and refused, with `encoder`.
`pooling`	How the feature map becomes one vector per image. `"avg"` averages evidence over the whole frame, which is what every classification head since the networks that replaced the flattened dense layer does; `"max"` takes the strongest response anywhere, which is the one to reach for when the thing being looked for is small and somewhere in the picture rather than the picture itself.
`head_dropout`	Drop this fraction of the pooled vector before the dense layer. Separate from `dropout`, which sits between convolutions and reaches nothing at all when the contracting path is a pretrained backbone with its own regularisation - this is then the only one left.
`dropout`, `batch_normalization`, `separable_conv`, `spatial_dropout`	Regularisation and shape of the contracting path, as in `u_net()`.
`activation`, `initialiser`	The non-linearity between convolutions, and how the weights start.
`loss`	The objective. Fewer are admissible here than anywhere else in this package, and the two that are - `loss_cce()` and `loss_focal()` - are the ones that score each answer on its own rather than scoring an overlap. `available_losses("classification")` lists them and the engine's refusal names them.

`metrics` What to report beside the loss. `metric_auc()` is the engine's default and that is a measurement rather than a preference: at a 3% prevalence a calibrated model never crosses the threshold, so accuracy, balanced accuracy and F1 all report a real signal and a collapse as the same number. See `metric_auc()`.

`optimizer`, `callbacks`, `augmentation`, `epochs`, `batch_size`, `weights`, `fit`
As in `u_net()`. A callback watching `val_<metric>` is refused when nothing in `metrics` will produce it, and there is no `foreground` to watch - a classification epoch has no background class and reports no such share.

`encoder` A pretrained backbone instead of our own contracting path, by name - see encoders for the table and what it is actually good for. With this, `blocks`, `filters` and `block_width` are refused.

`pretrained`, `freeze_encoder`, `encoder_learning_rate`
How the backbone arrives and how it is trained, as in `u_net()`.

`normalise` Normalise the input with the backbone's own ImageNet statistics, defaulting to `pretrained`. Set it `TRUE` with `pretrained = FALSE` to reload weights you exported from a pretrained encoder without a network: the statistics come from the backbone's configuration rather than from its weights. See u_net for the whole of it; it is recorded in the weights' fingerprint either way, and setting it `TRUE` needs `encoder`, because the statistics belong to a backbone.

Details

`n_class` is not an argument, for the reason it is not one on `u_net()` either: the number of classes comes from `classes` in `classification_data()`, and a model that also declared it could only ever disagree with its own data.

Value

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

See Also

`classification_data()`, `metric_auc()`

Examples

```
classifier("cnn", input_shape = c(128, 128), blocks = 3, filters = 16)

# A pretrained backbone instead of ours, with its early levels held still
classifier("resnet", input_shape = c(224, 224), encoder = "resnet34",
           pretrained = TRUE, freeze_encoder = 2)
```

`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())
```

ellipse_measure *The ellipse a segmented structure implies, in millimetres*

Description

What a sonographer writes down. A fetal head circumference is the perimeter of the ellipse fitted to the skull, and no overlap score stands in for it: two masks with the same Dice can be eight times apart in millimetres, depending on whether they are wrong in a ring all the way round or wrong in a ragged edge.

Usage

```
ellipse_measure(mask, spacing = NULL, class = 2L)
```

Arguments

mask	A mask of class indices, as predict() returns - one image, not a stack.
spacing	Pixel size in millimetres, length 2, in the array's own order: rows first, then columns. Unset measures in pixels. **Measure on the grid the spacing belongs to**: a prediction resized to the model's input shape has pixels of a size nobody recorded, so this wants predict(space = "source").
class	Which class is the structure, numbered from 1 as the colormap and labels are. Defaults to 2, the foreground of a binary segmentation. It is an argument rather than an assumption because this package writes 1 for background where the engine's own masks write 0, so treating anything above zero as foreground would measure the whole image.

Details

The fit is not iterative. For a uniform filled ellipse the covariance of the member pixels has eigenvalues a^2 / 4 and b^2 / 4, so the axes come out of a 2 x 2 eigendecomposition, and the circumference is Ramanujan's second approximation - right to about 1e-7 relative at the eccentricities a fetal skull has, where pi * (a + b), which is what one reaches for instead, is wrong by 1.5% at an axis ratio of 0.7.

The largest connected piece is taken and its holes filled before anything is measured. A prediction is not an ellipse: it can arrive in two pieces or with a hole in the middle, and the second moments of either belong to neither - a hole pushes the variance outwards and a stray island drags the centre.

Value

A one-row data frame with major, minor and circumference - the two semi-axes and the perimeter, in millimetres when spacing was given and in pixels otherwise. All three are 0 for fewer than three pixels of that class, because a region that small has no second moment; that is an ordinary outcome for a model that found nothing, not an error.

See Also

mask_volume() for the same argument in three dimensions, and instance_sizes() for it per object.

Examples

```
# A filled ellipse of 60 x 40 pixels, on half-millimetre pixels: semi-axes of 30 mm and
# 20 mm, whose perimeter is 158.8 mm.
rows <- matrix(rep(seq_len(201), 301), nrow = 201)
columns <- matrix(rep(seq_len(301), each = 201), nrow = 201)
mask <- ifelse(((columns - 150) / 60)^2 + ((rows - 100) / 40)^2 <= 1, 2L, 1L)
## Not run:
# Wrapped because this one does cross to the engine - unlike [mask_volume()], which is a
# count times a product and stays in R so that a reader needs no Python to run its page.
```

```
ellipse_measure(mask, spacing = c(0.5, 0.5))

## End(Not run)
```

`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", at = NULL, ...)
```

Arguments

`object`	A `platypus_fit()`.
`...`	Unused.
`split`	Which data to score on: `"validation"`, `"train"` or `"test"`.
`at`	Which grid to score on. `"model"` is each model's own `load_shape`, and leaving this unset is that. `"source"` puts every prediction back on the grid of the file it came from and scores it against the truth as stored, which is the form in which rows from models of different shapes answer one question. Segmentation only: a detection table has no grid to choose and says so rather than ignoring the argument.

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.

The metrics have a condition of their own and `load_shape` is it. Each model is scored on the grid it reads at, so a model at `input_shape = c(512, 512)` is measured against a truth reduced to 512 while one that tiles 2048 is measured against the truth as stored. Both numbers are a real Dice of a real prediction against a real mask, and they are not answers to the same question: measured on retinal vessels, that difference reported a resized model as **+0.0187 better** where the honest figure was **+0.0033**, and the bias runs the other way on clDice because upsampling smooths a prediction. When the rows disagree about the grid this warns, once; the column is there either way, because a table outlives the session that printed it.

`at = "source"` is the answer to that warning rather than a workaround for it: every prediction goes back onto the grid of the file it came from and is scored against the truth as stored, so the rows answer one question and `load_shape` reads `source`. The grid is the files' own and you cannot name another, which is deliberate - a grid you name resamples the **truth**, so the thing both models are judged against would itself become an interpolation somebody chose. Here only predictions move, and what that costs is in `predict()`: interpolation softens a boundary, and where reading cropped anatomy away the mask comes back padded with background, which means *not examined*.

For a comparison that is correct rather than unfair - a model whose field of view missed the organ did not segment it - but a 0 arriving that way is a result and not a fault. The loss is not reported there: a loss belongs to the objective a model trained with, on the grid it trained on.

Value

A data frame, one row per model.

What the one row pools

The **mean over cases**, since the engine's `0.8.0a40`. A case's own score is a ratio of sums - where a case arrived in tiles their overlap counts are added and the formula applied once, which is reassembly rather than a choice of average - and the split's figure is the mean of those. So this row is exactly `mean(evaluate_cases(fit)$dice)`, which is what anybody comparing the two expected and did not previously get: the engine divided an epoch's totals by the number of batches, so a short last batch weighed more than the others and the figure moved with `batch_size`. On a tiled run the difference was larger still, because tiles were averaged rather than reassembled - worth +0.033 to +0.049 of Dice on the retinal-vessel runs.

`evaluate_cases()` pools a **group** differently again, by adding the counts of the images in it, which is the better statistic for a patient rather than a slice. `maju116/rplatypus#196`.

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")

# Two models of different shapes compared on one grid - the files' own.
evaluate(fit, at = "source")

# 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,
  at = 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.
`at`	Which grid to score on, as in `evaluate()`. Unset is the model's own `load_shape`; `"source"` reads each truth as stored and scores the prediction there, which is the only form in which rows from two models - or from two runs with different `target_spacing` - mean the same thing.

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+)_")

# On the grid the files are on, which is what makes two models comparable.
evaluate_cases(fit, at = "source")

## 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, ignored, spurious and mean_matched_iou.

ignored counts predictions attributed to a truth marked difficult in a Pascal VOC annotation, which Pascal VOC's own evaluation sets aside: calling one matched would claim the model found something the benchmark left out, and calling it spurious would charge it for the same thing. So the counts are n_predicted == matched + ignored + spurious, and ignored is zero for a dataset that marks nothing difficult.

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)
```

`flow_to_colours` *A vector field as a picture: hue is the direction, value is the length*

Description

The convention optical flow has used for twenty years, and a convention rather than a palette decision - which is why it is not in `drawing_style()`. Nothing about it is a choice this project gets to make differently from everyone else's figures, and a reader who has seen one flow visualisation can read this one.

Usage

```
flow_to_colours(flow)
```

Arguments

`flow`	A two-channel vector field: `2 x height x width`, channel 1 the row component and channel 2 the column component, which is the layout the engine's targets use.

Details

What it is for: debugging a `separation_flows()` model. A field that is right everywhere except reversed looks obviously reversed here and identical in any loss. `predict()` with `type = "channels"` is where the field comes from.

A zero vector is black, which is also what the background is, and that is honest - the background has no direction.

Value

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

See Also

`separation_flows()` for the model that predicts one.

Examples

```
field <- array(0, dim = c(2, 2, 3))
field[1, 1, 1] <- 1   # pointing down
field[2, 1, 2] <- 1   # pointing right
shown <- flow_to_colours(field)
# Two directions, two hues; and no vector, no colour.
!identical(shown[1, 1, ], shown[1, 2, ]) && all(shown[2, 3, ] == 0L)
```

`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 - **region names where the specification gave** `regions`, and class numbers otherwise, which is what the engine's own refusal does. The positions are attached as integer vectors in the attributes `present` and `missing` either way, so anything reading them keeps working.

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.

With `regions` **the classes are named rather than numbered**, because a position is the one thing a reader cannot resolve: `missing_classes: 1` means the second region, and counting entries in a configuration file to find out which is the step nobody takes. The positions are still there, as the `missing` and `present` attributes.

That naming matters most for the one failure `regions` invites: `of = 3` against a label map holding `4` is a channel of zeros, so the model trains to predict nothing and scores a Dice of 0 for that region

while every other number in the run looks ordinary. The engine refuses it before training and names the region; this is how to ask the same question over more of the data than the refusal reads.

`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/")
```

`instance_shapes` *What shape each object is, not just how big*

Description

One row per object, beside the rows `instance_sizes()` gives. `voxels` and `area_mm2` answer *how big*; these answer *how stretched* - the semi-axes of the ellipse an object's second moments imply, largest first, and for an image the `eccentricity` that shape analysis actually reports.

Usage

```
instance_shapes(labels, spacing = NULL)
```

Arguments

`labels`	An instance label map from `predict()` - one integer per object, 0 for background, the same numbering `instance_sizes()` reads. Two dimensions for an image, three for a volume.
`spacing`	The size of one voxel in millimetres: two numbers for an image, three for a volume, in the array's own order. Unset measures the sampling grid, and the column names say so.

Details

The fit is not iterative, and it is the same arithmetic `ellipse_measure()` applies to a whole mask: for a uniform solid the variance along an axis is `semi_axis^2 / (rank + 2)`, so the axes come out of one eigendecomposition per object. What this adds is that it is per object **in one pass**: the alternative a caller writes - `ellipse_measure()` on `labels == k`, once per object - rescans the whole array every time, and on a 512 x 512 map of 1939 objects that is 1989 ms against 143.

A volume gets three axes and no eccentricity, because an ellipsoid has no single one; the ratio of two columns is your division rather than a name this package invents.

Value

A data frame with `object` and the semi-axes largest first - `major_mm` and `minor_mm` for an image, with `middle_mm` between them for a volume, named `_px` instead when `spacing` was not given. An image also gets `eccentricity`. One row per object present, in increasing object number; an object with fewer than `rank + 1` voxels has no second moment and gets zeros, which is an ordinary outcome rather than an error.

`eccentricity` **is unitless and it is not spacing-free.** It describes whichever space it was computed in, so on anisotropic voxels the two disagree completely and in both directions: a circle of 80 pixels at `spacing = c(2, 1)` reads 0.0000 on the grid and 0.8660 in millimetres, while an 80 by 40 pixel ellipse at `spacing = c(1, 2)` reads 0.8667 on the grid and 0.0685 physically - physically almost a circle, which the grid calls strongly elongated. Pass your voxel size and read the `_mm` columns unless you mean the grid; the suffix on the axes is what says which of the two a row is.

See Also

`instance_sizes()` for how big each object is, `ellipse_measure()` for the same fit over a whole mask, and `overlay_instances()` to look at them.

Examples

```
labels <- matrix(0L, nrow = 41, ncol = 81)
rows <- matrix(rep(seq_len(41), 81), nrow = 41)
columns <- matrix(rep(seq_len(81), each = 41), nrow = 41)
labels[((rows - 21) / 10)^2 + ((columns - 21) / 18)^2 <= 1] <- 1L
labels[(rows - 21)^2 + (columns - 61)^2 <= 81] <- 2L
## Not run:
# Wrapped because this one crosses to the engine - unlike `instance_sizes()`, which is a
# count times a product and stays in R so that a reader needs no Python to run its page.
# An eigendecomposition per object is not arithmetic a reader verifies by eye, which is
# the line `maju116/pyplatypus#270` drew between the two.
instance_shapes(labels, spacing = c(0.5, 0.5))

## End(Not run)
```

---

instance_sizes *How big each object is, in units somebody can use*

Description

One row per object. `voxels` is the count, which means nothing outside one scanner, and `spacing` turns it into a number a reader can compare - the same argument `mask_volume()` makes for one class, applied per object. That is the question this whole feature is for: *how many nuclei, and how big is each one*.

Usage

```
instance_sizes(labels, spacing = NULL)
```

Arguments

`labels`	An instance label map from `predict()` - one integer per object, 0 for background. Two dimensions for an image, three for a volume.
`spacing`	The size of one voxel in millimetres: two numbers for an image, three for a volume. Unset means the result counts voxels and says nothing about size.

Details

The unit is decided by the rank, and the column is named after it rather than being called `size` with a note somewhere: `area_mm2` for an image, `volume_ml` for a volume. A column whose unit lives in the documentation is a column that ends up in a paper as a bare number.

Without `spacing` there is no such column at all, rather than a count pretending to be a measurement.

Value

A data frame with `object`, `voxels` and, when `spacing` is given, `area_mm2` or `volume_ml`. One row per object present, in increasing object number; an object number absent from the map gets no row, because a model that found objects 1, 2 and 4 found three objects.

See Also

`overlay_instances()` to look at them, `mask_volume()` for one class of a semantic mask.

Examples

```
labels <- matrix(0L, nrow = 8, ncol = 8)
labels[2:4, 2:4] <- 1L
labels[6:7, 6:8] <- 2L
instance_sizes(labels)

# 0.5 mm pixels: object 1 is 9 pixels, so 9 * 0.25 = 2.25 mm^2.
instance_sizes(labels, spacing = c(0.5, 0.5))
```

---

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. `loss_generalized_dice()` here is the combination worth knowing about, because the two features are motivated by the same picture: this loss exists for a lesion in a thousandth of a frame, and a surface term exists for an object whose boundary is most of it.
`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 losses that can be the region term - every one of them except the three that score numbers and this one, held to the engine's own union by a test rather than counted here - and `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_generalized_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 what is already classified well - a pixel or voxel for a mask, a whole image for a label - so the gradient comes from what is not. 0 is plain cross-entropy and 2 is the usual value.
`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.

One of these weights the classes rather than treating them alike - `loss_generalized_dice()`, by the inverse square of how much of the picture each class occupies - and the argument usually made for it is not the argument for it here. It is introduced in the literature as a fix for Dice being dominated by the background, and `loss_dice()` in this package is a mean over the class channels: a lesion in 1% of the frame already carries the same weight as the 99% around it. Measured on a 1% target, with one prediction that finds the lesion and leaks a false positive of equal size and one that misses it and gets the background perfect:

loss	finds it	misses it	prefers finding by
`loss_dice()`	0.1658	0.4731	0.3073
`loss_cce()`	0.1600	0.1600	nothing at all
`loss_cce_dice()`	0.1629	0.3166	0.1536
`loss_generalized_dice()`	0.3278	0.9511	0.6233

So what it adds is weight **beyond** an equal share - on that target the classes come out weighted 1 to 9801 - and the objective that is genuinely dominated is the per-pixel half of `loss_cce_dice()`: to cross-entropy a false positive and a missed lesion of the same size are the same number of wrong pixels, which is the indifference in that row. Two things follow. With every class the same size this is **exactly** `loss_dice()` **pooled over the classes**, so on a balanced problem it costs nothing and buys nothing. And it is not a third `loss_tversky()`: Tversky moves the balance between false positives and false negatives by a number you choose, `loss_focal_tversky()` reweights the hard

pixels, and this reweights classes by how much of the image they occupy, computed from each batch rather than set by hand - so a batch holding no lesion at all is scored on flat weights, which is an argument for a larger `batch_size` on a sparse dataset rather than against the loss.

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, n_class = NULL)
```

Arguments

`mask`	Class indices, counted from 1.
`labels`	Optional names for the classes. **Also what says how many classes there are**, when the mask does not hold them all - see the note below.
`n_class`	How many classes the mask is meant to have, when `labels` was not given. Unset means the largest value present, which is only right when every class occurs.

Value

A data frame with one row per class, including the classes with no pixels.

A class above the largest value present used to have no row at all, which was the one case this function is for: `tabulate()` sizes its table by the largest value it sees, so a three-class mask holding only 1 and 2 reported two classes and said nothing about the third. The highest-numbered class is the usual position of a rare foreground, so the row that went missing was the one worth reading. `maju116/rplatypus#255`.

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)

metric_surface_dice(tau, 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. It defaults to `TRUE` for the three overlap metrics and `FALSE` for `metric_cldice()` and `metric_surface_dice()`, where the background class does not merely flatter a model but measures something that is not the question - its skeleton and its boundary are both artefacts of the frame rather than of what was found. The paragraphs above give both measurements.
`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.

tau — For `metric_surface_dice()`, the tolerance a boundary is allowed to be wrong by, and it is **required because no default would be honest**. A tolerance belongs to the structure and the dataset rather than to a package: conventionally it is the inter-observer variability for that structure - the distance two people drawing the same boundary differ by - and published values exist per organ and per challenge. Take one from the protocol you are comparing against. In millimetres where `target_spacing` puts every case on a common grid, and in voxels where it does not, which is the same pair of answers the signed distance map in `loss_boundary()` gives and for the same reason: a 0.5 mm and a 2.5 mm acquisition are not comparable in voxels at all.

Details

Three of these ask the same question and two ask different ones. `metric_iou()`, `metric_dice()` and `metric_tversky()` all ask how much of the mask coincides and differ in how they weigh the ways of being wrong. `metric_cldice()` asks whether the *structure* coincides, and `metric_surface_dice()` asks **how far wrong the boundary is** - which is the question a contour somebody has to approve is actually judged by, and the one nothing here reported until the pin moved to `0.8.0a26`.

For something thin and connected clDice is the question worth asking, and it and Dice 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.

`metric_surface_dice()` is the other kind of disagreement. `loss_boundary()`'s own page reports a model at Dice 0.9525 whose volumes run 5% small and 17% wrong per case, and no overlap column says that: a displaced boundary and a correct one of the same area score alike. This is normalised surface Dice (Nikolov et al., 2018), symmetric on purpose - the truth's surface near the prediction is the boundary's recall, the prediction's surface near the truth is its precision - and it is the one metric here that reads the voxel spacing, so `tau` is in millimetres when `target_spacing` has put the data on a common grid and in voxels when it has not. It is also what `loss_boundary()` optimises, so until now that loss was being judged by the metric it exists to improve on.

Like clDice it is a property of a whole mask rather than of a batch of counts, so on a tiled run (`splits`) it is reported **per case** by `evaluate_cases()` and is absent from the epoch history - a per-batch average of per-tile boundaries would be a number about nothing. A callback asking to watch `val_surface_dice` on such a run is refused when the configuration is read, the way one watching `val_cldice` is - so the choice to tile and the choice of what to stop on are settled together, before anything is trained.

`include_background` defaults to `FALSE` for those two and `TRUE` for the three overlap metrics, and each has its own measured reason. The background's skeleton lies inside the background by construction, so that class scores 1 for clDice whatever the model did - averaging it in moved 0.7865 to 0.8933 on a three-pixel vessel. The background's *boundary* is the foreground's boundary plus the edge of the picture, and that edge agrees between any two masks whatever happened in them: measured on four random pairs at `tau = 2`, the background scored 0.6110 to 0.7030 against the foreground's 0.5583 to 0.6266 - higher every time. That is agreement on the field of view rather than on the object.

Metrics for one label per image are a different family and have their own page. `metric_accuracy()`, `metric_balanced_accuracy()`, `metric_f1()` and `metric_auc()` score a classifier, where there is no overlap to measure and no background class to leave out - see classification_metrics. None of the names on this page appears there and none of theirs appears here, which is the honest shape: a Dice over one label would be the label itself.

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()

# How far wrong the boundary is, rather than how much of the area coincides. 1.5 mm
# because that is this structure's inter-observer variability, not because of anything
# here.
metric_surface_dice(tau = 1.5)
```

`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,
  normalise = 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,
  normalise = 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,
  normalise = 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,
  normalise = 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. These are reported beside the loss and are **computed on the mask rather than on the objective**, which is what makes them comparable between models: a Focal-Tversky of 0.05 and a CCE-Dice of 0.14 are not the same question, and their Dice is.

A callback watching `val_<metric>` is refused when nothing in this list will produce it - when the configuration is read, rather than at the end of the first epoch with the epoch already paid for. Which is also why a tiled run refuses a callback watching `val_cldice` or `val_surface_dice`: those are properties of a whole mask, reported per case rather than per epoch, so the choice to tile and the choice of what to stop on are settled together.

For a classifier the list is a different family - see classification_metrics, where none of these names appears and the default is measured rather than chosen.

`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.

normalise
: Normalise the input with the backbone's own ImageNet statistics. Defaults to `pretrained`, which is what it meant before it was a setting, so leaving it alone changes nothing. It exists to be set `TRUE` **with** `pretrained = FALSE`, which is the only way to reload weights you exported from a pretrained encoder on a machine with no network: the statistics come from the backbone's own configuration and need no download, while the weights themselves are yours. It is recorded in the weights' fingerprint either way, so a mismatch is refused by name rather than silently predicting something else. Setting it `TRUE` needs `encoder`: the statistics belong to a backbone and the built-in encoder has none, so without one it is refused rather than quietly ignored.

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.

  It may be left out when `weights` **names a file that records it**, since pyplatypus `0.8.0a28`: a published sidecar says what the channels were trained

to predict, and "dsbowl-instances" is a rim of width 2 - a training decision rather than a shape, and nothing a reader can guess from a model card. So u_net("objects", input_shape = c(256, 256), weights = "dsbowl-instances") beside instance_data() is an instance model, and the engine prints what it adopted.

Left out with **no** weights it is still semantic, and still refused beside instance_data() as two tasks in one specification - which is the pair that would otherwise be silent: the same layout and the same architecture, differing only in whether a signal is predicted at all. maju116/rplatypus#181.

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_instance_agreement`

Found, missed and invented objects

Description

`overlay_agreement()` answers the same three-way question about **pixels**, and on an instance result that is the wrong unit: a merged pair and a split object are both nearly perfect pixelwise and both wrong, which is exactly what an average precision will not tell anybody. This matches whole **objects** at one IoU threshold, the way the metric matches them, so the picture cannot disagree with the number it illustrates.

Usage

```
overlay_instance_agreement(
  image,
  prediction,
  truth,
  threshold = 0.5,
  alpha = drawing_style()$overlay_alpha,
  colours = drawing_style()$agreement_colours
)
```

Arguments

`image`	One image, as `overlay_instances()` takes it.
`prediction`	An instance label map from a model.
`truth`	An instance label map from the annotation.
`threshold`	The IoU above which a predicted object counts as a true one, strictly greater. 0.5 is the loosest of the challenge's ten thresholds, so it is the kindest reading, and **below 0.5 is refused** for the reason the metric refuses it: there one prediction can satisfy two true objects and the match stops being one to one.
`alpha`	How much of the tint shows.
`colours`	Named `hit`, `missed` and `false_alarm`, as `overlay_agreement()` takes them.

Value

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

See Also

`overlay_instances()` for the objects themselves.

Examples

```
image <- array(0.5, dim = c(5, 11, 3))
truth <- matrix(0L, nrow = 5, ncol = 11)
truth[2:4, 2:4] <- 1L
truth[2:4, 6:8] <- 2L
predicted <- matrix(0L, nrow = 5, ncol = 11)
predicted[2:4, 2:4] <- 1L    # one found
predicted[2:4, 9:11] <- 2L   # one invented, and object 2 missed
shown <- overlay_instance_agreement(image, predicted, truth)
# Three different colours: hit, missed, invented.
nrow(unique(rbind(shown[3, 2, ], shown[3, 6, ], shown[3, 10, ])))
```

`overlay_instances` *Draw an instance map over its image*

Description

One colour per object, chosen so that **no two touching objects share one**. The rule is adjacency rather than identity: a colour repeats across the picture and never between neighbours, because nobody reads object 214 by its colour and everybody needs to see where one object ends.

Usage

```
overlay_instances(
  image,
  labels,
  alpha = drawing_style()$overlay_alpha,
  outline = FALSE,
  colours = drawing_style()$class_colours
)
```

Arguments

`image`	One image: `height x width x 3`, `height x width x 1` or a matrix. Values may be 0-1 or 0-255.
`labels`	An instance label map - one integer per object, 0 for background. Not a one-hot array: an object identity cannot be carried by a channel.
`alpha`	How much of the tint shows. Ignored when `outline` is true, where the stroke is drawn at full strength, because a translucent outline is a smudge.

`outline` Draw one stroke per object instead of filling it, which leaves the tissue entirely visible and is what a pathologist is used to.

`colours` The palette to take slots from. Eight by default, mirrored from the engine's own. A picture needing more colours than this has **warns**, naming how many it needs and how many objects lost the guarantee: the slot wraps onto a colour a neighbour already holds - provably, since it is only past the end once every colour before it was taken by a neighbour - so two touching objects are drawn as one and this figure answers its own question wrongly. The engine warns in the same words, which is checked rather than claimed. Unreachable on anything measured: the hardest of 670 Data Science Bowl cases needs three of the eight.

Details

That is an exact graph colouring, and it is affordable rather than clever. Measured over all 670 Data Science Bowl cases: the hardest needs three colours and `class_colours` has eight, while the cheap alternative - colour by object number, modulo the palette - draws one touching pair in ten as a single object.

Eight-connected, so two objects meeting at a corner get different colours. A label-maker goes the other way and joins only along edges, because there the risk is merging two objects into one; here the risk is *drawing* two as one, which is the opposite direction.

Value

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

See Also

`overlay_instance_agreement()` for whether the objects are the right ones, `overlay_mask()` for classes rather than objects.

Examples

```
image <- array(0.5, dim = c(5, 9, 3))
labels <- matrix(0L, nrow = 5, ncol = 9)
labels[2:4, 2:4] <- 1L
labels[2:4, 5:7] <- 2L   # touching object 1 along a seam
shown <- overlay_instances(image, labels)
# Two touching objects, two colours.
!identical(shown[3, 2, ], shown[3, 6, ])
```

`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_measure_error         a measurement whose spacing does not fit the array
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_label_error          an image's label, or the class directories holding it
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 fifteen 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()
```

Details

The answer comes from the engine's own `device_report()` since `0.8.0a38` - this package used to compute it, in a file of Python it ships, where no user of the engine could reach it. What stays here is the advice: the line telling you to call `platypus_use_torch()` is about R's session model and would mean nothing to somebody who installed the engine with pip.

An accelerator that is not CUDA is named, because the published weights were all trained on CUDA cards and a score from another backend is not directly comparable with the registry's. Until

`0.8.0a38` the engine asked about CUDA alone, so an Apple Silicon Mac reported `cpu` while a working `mps` backend sat in the same installation.

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_fact` *One fact about the platypus*

Description

The package is named after the animal, and this prints something true about it. It has nothing to do with segmentation.

Usage

```
platypus_fact(number = NULL)
```

Arguments

`number`	Which fact, from 1 to the number there are. `NULL`, the default, picks one at random.

Details

A fact is also printed once when the package is attached, in an interactive session only. To stop that, set `options(rplatypus.facts = FALSE)` in `~/.Rprofile`, or the environment variable `PLATYPUS_FACTS` to `0` or `false` in `~/.Renviron`.

Each fact carries the source it was read from, because several of the best-known claims about platypuses are wrong and four of these exist to say which. The numbering is stable across versions: facts are added to the end and never reordered, so a number can be quoted.

Picking one at random does **not** disturb the random number stream, which is the only part of this worth any care: `library()` must not change what `set.seed(1)` then produces, and `suppressPackageStartupMessages()` hides the message without preventing the side effect, so avoiding it is the only fix that works.

Value

An object of class `platypus_fact`, which prints as the fact and its source.

Examples

```
platypus_fact(1)
platypus_fact()
```

`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()`, `instance_data()` or `classification_data()` specification, or the path to a YAML file. Which one it is decides `task`. **A plain list of the same fields also works**, which is what a caller assembling a configuration programmatically reaches for, and it is the shape a YAML file already proves the engine accepts. A plain list names no task, so `task` is then taken from whatever else does - the models, if they were built by a constructor - and from the `task` argument when nothing else says anything. With neither, it is

semantic segmentation. A constructor is still the better route: it refuses a wrong field here rather than across the bridge, and the fields it refuses are most of what it is for.

`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. For a classifier, which has no architecture to vary, the comparison the task makes easiest is several entries differing only in `encoder` - ours against a pretrained backbone, on the same split.

`task` Which task this specification describes, one of `"semantic_segmentation"`, `"object_detection"`, `"instance_segmentation"` or `"classification"`. 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.

Updating this package

Nothing to do. Each release of this package pins one exact engine version, that pin is what gets declared when the package loads, and the environment is resolved from it on first use. Install a newer rplatypus and the next session uses the engine it asks for; the first call after the upgrade fetches the new engine - a few hundred kilobytes, with PyTorch already cached - and the environment the old version used is left alone, so going back to it costs nothing either.

That is the opposite of the arrangement most R packages wrapping Python have, where a persistent environment is installed once by hand and updating the R package does not touch it. Here there is no environment anybody maintains: there is a pin, and the environment always matches it. The cost is that the two cannot be moved separately - which is the point, since a release that could drift from its engine is the failure the pin exists to prevent.

One case is worth knowing: updating **on the day the engine is published**, uv may still be serving a cached index listing and report that the pinned version does not exist. It is a stale read rather than a missing release, it mends itself, and the failure message says so.

If RETICULATE_PYTHON is set, none of this applies - that Python is used whatever the pin says, and a mismatch is reported as a warning naming both versions.

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.

Saying it once instead of every session

`py_require()` is session state, so this does not survive a restart - and forgetting it costs about ten times the training time without costing an error, which is the kind of bill that goes unnoticed. Set it where a session starts instead:

```
options(rplatypus.torch = "pascal")   # ~/.Rprofile
```

or `PLATYPUS_TORCH=pascal` in `~/.Renviron`, which also reaches runs that never load an `.Rprofile` - `Rscript`, a scheduler, CI. The option wins over the variable, so a session can override what a machine was configured with. `"default"` asks for the ordinary build, which is how to turn the setting off for one session without editing the file it came from.

Both are read when the package is loaded, which is the last moment anything can change what the environment will be. A value neither of them recognises is reported at load and ignored: a bad line in a startup file should not make the package unloadable.

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, boxes, objects or labels*

Description

One function per fit rather than one per task, because the question - *what does this model say about this split* - is the same one. What comes back is not: a mask for segmentation, a table of boxes for detection, a label map of objects for instances, and a row per image for classification. The arguments that describe a grid are refused where there is no grid, by name, rather than accepted and ignored.

Usage

```
## S3 method for class 'platypus_fit'
predict(
  object,
  model = NULL,
  split = "test",
  type = c("class", "probability", "channels"),
  space = c("model", "source"),
  cases = NULL,
  each = NULL,
  decode = 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. "channels" is for an **instance** model only and means the model's own output before any decoding: the class channels are probabilities and a flow field's components are **raw**, which is why they are not called probabilities here - a softmax across a flow component and a class returns a number that means nothing. Read them with flow_to_colours(). On an instance fit, "class" is a label map of **objects**, numbered from 1, and "probability" is refused by name for the reason above. With regions **or** classes_from - the two namings that make a pixel's classes independent - "class" is **refused** rather than computed. The channels are independent binary questions and do not compete, so the most likely one is not an answer: a pixel may be in three classes and naming one of them drops the rest with nothing in the result to say so. Ask for "probability" and write it with save_class_masks(). This page named regions alone until maju116/rplatypus#252, and so did the refusal: a classes_from fit got the most likely channel, which gives every background pixel a class, because there is no background channel for it to lose to. The question is asked of the engine now rather than of either field. On a **classification** fit the two readings are a table and a matrix rather than two arrays: "class" gives one row per image with the answer and how sure the model was, and "probability" gives one row per image and one column per class. "channels" is refused - a classifier has no channels, only one number per class, and those numbers are the probabilities.
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. Refused on a detection, instance or classification fit, each for its own reason - boxes are in

the source image's coordinates already, an object identity cannot survive resampling, and a label is not on a grid at all.

`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. Both this and `each` exist so that a split of full-resolution masks never has to be in memory at once, so both are refused on a classification fit, where the whole answer is one row per image.

`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.

`decode` Which decoding turns the predicted channels into objects, by the name used in the model's `decode` list. Unset takes the model's first, which is the engine's own default rather than one restated here. **Instance models only**; for anything else it is refused, because a semantic or detection fit decodes nothing.

`...` 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, **named by case**. The name and the mask come out of one step of the engine's own stream, so they cannot be misaligned - which matters because the usual next move is to join them to something else per case, and a list paired by position is one dropped batch away from naming the wrong picture.

On an **instance** fit: a list of integer label maps **named by case**, objects numbered from 1 and 0 for background, one entry per image. A list rather than an array for the reason the detection one is a list, and named rather than ordered because a list paired against case names by position is one dropped batch away from naming the wrong picture. Draw them with `overlay_instances()` and measure them with `instance_sizes()`.

On a **classification** fit: for `"class"`, a data frame of one row per image with `key` (the name the split calls it, which is what joins this to anything else), `label` (what it was labelled, or `NA` on a split that carries no labelling), `predicted` and `probability` - the probability of the class that was chosen, not of the right one. For `"probability"`, a numeric matrix of images by classes, with the keys as row names and the class names as columns, in the order `classes` was written.

With `labels_from`, where a label is a set, `"class"` is a longer frame instead: one row per image *per class*, with `key`, `class`, and `label` and `predicted` as logicals. It is a different answer rather than the same answer with a longer string, and this is the shape that says so - an image with none of the findings is a row of `FALSE` per class rather than an empty cell, which could also mean unread. Summarise per class with one `tapply`, or reshape it if you want it wide.

There is no `probability` column on that route, and it is missing on purpose: that column is the chosen class's own number, and with independent questions nothing was chosen. `"probability"` is the whole of the answer there and comes back as the same matrix, one column per class.

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,
  label_map = 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.
`label_map`	The file holds class identities rather than brightness, so read it at full integer width. A 16-bit PNG or a 32-bit TIFF label map is otherwise read through 8-bit greyscale, where every value above 255 arrives as 255 - two classes then merge into one with nothing unmatched to say so. One channel only, because a label map has one, and never for an image: a picture is divided by 255 later. **For a volume this argument does nothing and `nearest` is what matters**, which is the engine's own rule rather than this package's: a NIfTI or a DICOM series never went through an 8-bit conversion, so there is no width to widen, and the flag that says *these numbers are classes* on that route is `nearest`. Passing `label_map = TRUE` alone on a volume returns the values divided by the largest one. Pass both.

Value

An `image x height x width x channels` array of 0-255 values - or of the file's own integer values with `label_map = TRUE` **and**, for a volume, `nearest = TRUE`. A volume read with `label_map = TRUE` and `nearest = FALSE` comes back rescaled to 0-1, and this function warns when it is asked for that combination.

Examples

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

# A label map stored wide, whose values are classes rather than brightness.
values <- read_images("seg.png", channels = 1, nearest = TRUE, label_map = TRUE)

## 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_class_masks *Write overlapping classes, one file per class*

Description

`regions` gives a channel per named class and a sigmoid per channel, so a pixel may be in several classes at once - that is the point of it. A colour map cannot hold that: one pixel gets one colour. `save_masks()` therefore refuses such a prediction, and this is what it refuses it in favour of - one binary picture per class, named after the class.

Usage

```
save_class_masks(
  masks,
  dir,
  classes,
  names = NULL,
  suffix = "",
  threshold = 0.5
)
```

Arguments

masks	Channel probabilities from `predict(fit, type = "probability")` - an `image x height x width x cla` array, or one `height x width x class` array. Class indices are the wrong input and are refused where a shape or a value can prove it: any value above 1 is not a probability. What cannot be proved is a stack of class-index masks whose values are all 1 - `image x height x width` has the same rank as one image's `height x width x class`, exactly as `save_masks()` cannot tell a stack of volumes from a stack of images - so that one is read as one image with eight channels and caught by `classes` not having eight names.
dir	Directory to write into; created if it does not exist.
classes	One name per channel, in channel order, which is the order the regions were written in and the order every per-class score is reported in. `inspect_masks()` names them the same way. Called `classes` and not `names` because `names` in this family is the files - see below.
names	File name stems, one per image, or the source image paths to take them from. Defaults to `mask_0001` and so on. The engine's own function calls this `cases`; in R it is `names`, matching `save_masks()` and `save_volumes()`, and `classes` carries what the engine calls `names`.
suffix	Added before the class, so predictions from different models can sit in one directory without overwriting each other.
threshold	A class is written where its channel reaches this. 0.5 is what every metric in the engine thresholds a channel at, so a different number here writes pictures that disagree with the scores printed beside them.

Details

Each file is `<name><suffix>_<class>.png`, black and white, so it opens in anything and thresholds back to exactly what was written. A class name becomes a filename by replacing anything outside letters, digits, dot, dash and underscore with `_`, so `whole tumour` is `whole_tumour` on disk; two class names that become the same filename are refused rather than written, because one file would hold the second class with the first silently gone.

A class nothing reaches is written as an empty picture rather than skipped. A missing file reads as *this run did not produce that class* and an empty one as *it did and the answer was nothing*, which are different findings and only one of them is true.

Value

The paths written, invisibly, image by image and within an image class by class.

See Also

`save_masks()` for exclusive classes as one colour picture, `save_volumes()` for volumes as NIfTI.

Examples

```
## Not run:
spec <- platypus_spec(... )
```

```
channels <- predict(fit, "brats", split = "test", type = "probability")
save_class_masks(channels, "predictions", classes = c("whole tumour", "core", "enhancing"))

## End(Not run)
```

`save_class_volumes` *Save overlapping classes as one NIfTI per class*

Description

The volume counterpart of `save_class_masks()`, and the one `regions` was motivated by: BraTS names three overlapping regions and stores them as volumes. A label-map NIfTI holds one integer per voxel, so `save_volumes()` refuses an overlapping prediction rather than reducing it, and this is what it refuses it in favour of - one binary volume per class, each carrying the geometry of the scan it was predicted from.

Usage

```
save_class_volumes(
  masks,
  dir,
  reference,
  classes,
  names = NULL,
  suffix = "",
  threshold = 0.5
)
```

Arguments

`masks`	Channel probabilities - a list of `depth x height x width x class` arrays, which is what `predict(type = "probability", space = "source")` gives, or one stacked `volume x depth x height x width x class` array.
`dir`	Directory to write into; created if it does not exist.
`reference`	The scan each mask was predicted from, as paths. Every class of one scan is written with that scan's geometry, which is the whole reason this is not `save_class_masks()` with a different extension.
`classes`	One name per channel, in channel order, which is the order the regions were written in. `inspect_masks()` names them the same way.
`names`	File name stems, one per volume. Defaults to the reference file names with their extensions removed - and names that collide are refused, because one directory per case usually means the same filename inside each.
`suffix`	Added before the class, so predictions from different models can sit in one directory.
`threshold`	A class is written where its channel reaches this. 0.5 is what every metric in the engine thresholds a channel at.

Details

Each file is <name><suffix>_<class>.nii.gz. The class name becomes part of a filename the same way save_class_masks() does, and two class names that become the same filename are refused rather than written.

Why one file per class rather than one file with several values: a voxel in the tumour core is also in the whole tumour, and no single integer says both. Any viewer can load three binary masks over one scan; none can unpack an overlap out of one label map.

Value

The paths written, invisibly, volume by volume and within a volume class by class.

See Also

save_class_masks() for 2D, save_volumes() for exclusive classes.

Examples

```
## Not run:
channels <- predict(fit, "brats", split = "test", type = "probability", space = "source")
save_class_volumes(channels, "predictions",
                   reference = split_path(split, "test")$image,
                   classes = c("whole tumour", "tumour core", "enhancing"))

## 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.

	For **overlapping classes** use `save_class_masks()`. A prediction where some pixel is in two classes at once is refused here rather than reduced: one pixel gets one colour, so writing it as a colour map would mean choosing one of the classes it is in and leaving nothing in the file to say that happened.
`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,
  regions = NULL,
  classes_from = 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.

`regions` Overlapping classes, named as unions of a label-map value set: a list of `list(name = , of = )`, where `of` holds the voxel values belonging to that region. `list(list(name = "whole tumour", of = c(1, 2, 4)), list(name = "enhancing", of = 4))` is the BraTS shape, and the second entry shows that a single value is ordinary rather than an edge case. Give this, `colormap` or `labels`, not more than one.

Two things this field changes that its name does not say. There is **no background entry**, so `n_class` here is not the same number it would be for that many `labels` - with `labels` the background is a class and with `regions` it is not. And each channel is an **independent binary question**, a sigmoid per channel rather than one softmax across them, so a voxel may be in several regions at once - which is the whole point, and the reason `save_masks()` cannot write such a prediction as one picture.

Two regions are the minimum: one region is ordinary binary segmentation, which `labels = c(0, 1)` already says. Disjoint regions are allowed and mean something - grouping several values into one class and dropping the background channel - so they are not refused.

`classes_from` For masks kept as **one binary file per class**: one pattern per class, in class order. The fourth way of saying what a mask's classes are, and the only one that can express an overlap that was drawn independently - a vessel crossing an organ, or two annotators' masks of the same structure. Give this, `colormap`, `labels` or `regions`, not more than one.

It is `regions`' sibling, not its replacement, and the difference is the file on disk. `regions` derives overlapping channels from the values of one label map, which is how BraTS stores its ground truth; this reads channels that were separate files to begin with, because no single voxel value can hold two things drawn independently. Everything the two share is everything after the target: no background entry, a sigmoid per channel rather than one softmax across them, a voxel that may be in several classes at once, and a prediction `save_masks()` cannot write as one picture - use `save_class_masks()` or `save_class_volumes()`.

Patterns are Python regular expressions matched against a sample's **mask** files, with the same two rules `channels_from` has for images and for the same reason: each must match exactly one file, and every mask file must be claimed by some pattern. Both halves are refused, because each is invisible from the other side - a pattern matching nothing is a class the model is never shown, and a file matching no pattern is a class the data has and nobody asked for. R's own quoting applies: `c("_vessel\\.png$", "_organ\\.png$")`.

The order is written rather than inferred, and here it matters more than for channels: a model trained with the vessel in the first channel and used where the first channel is the organ answers plausibly, in range, and nothing further down can notice.

`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

One row per model per epoch, with the loss, whatever metrics were asked for, how long the epoch took and the learning rate it ran at. Two columns arrive whether or not anybody asked:

Usage

```
training_history(object)
```

Arguments

`object`	A `platypus_fit()`.

Details

- `foreground` - the share of the prediction that is not background. Not a measure of quality, because nothing is compared against the truth: it answers whether the model is still producing a segmentation at all. A model collapsing to background reports a plausible loss and, with `include_background = TRUE`, a Dice around 0.5.
- `val_<metric>_worst` - the lowest any single case scored. The distribution is the finding and the mean is where a failure hides, so the number worth reading is in the table rather than only in `summary()` after the run.

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(), *8*, *61*
available_callbacks (available_components), 5
available_components, 5
available_decodings (available_components), 5
available_losses (available_components), 5
available_losses(), *61*
available_metrics (available_components), 5
available_metrics(), *19*
available_models (available_components), 5
available_models(), *77*
available_optimizers (available_components), 5
available_optimizers(), *66, 67*
available_separations (available_components), 5
available_tasks (available_components), 5
available_transforms, 7
available_transforms(), *5, 6*
available_weights, 8
available_weights(), *5, 6*, *112*

binary_colormap, *17*
binary_colormap (colormaps), 24
binary_labels (colormaps), 24

callback_cosine_annealing (callbacks), 11
callback_cosine_annealing(), *10*
callback_csv_logger (callbacks), 11
callback_early_stopping (callbacks), 11
callback_ema (callbacks), 11
callback_linear_annealing (callbacks), 11
callback_model_checkpoint (callbacks), 11
callback_reduce_lr_on_plateau (callbacks), 11
callback_step_decay (callbacks), 11
callback_stop_after (callbacks), 11
callback_swa, 9
callback_swa(), *14, 15*
callback_terminate_on_collapse (callbacks), 11
callback_terminate_on_nan (callbacks), 11
callbacks, *10*, 11, *61*, *64*
callbacks(), *67*
class_balance, 16
class_balance(), *19*, *21*
classes_to_colours, 17
classes_to_colours(), *25*
classification_data, 17
classification_data(), *16*, *23*, *76*
classification_metrics, 20, *57*, *61*
classifier, 21
classifier(), *19*, *21*
colormaps, 24
colours_to_classes, 25
colours_to_classes(), *90*
ct_windows, 25

decode_connected_components (decodings), 26
decode_follow_flows (decodings), 26
decode_nearest_seed (decodings), 26
decode_watershed (decodings), 26
decodings, *6*, 26, *63*, *103*
detection_anchors, 27
detection_anchors(), *82*
detection_crops, 28
detection_data, 30

detection_data(), *18*, *76*, *112*
drawing_style, 32
drawing_style(), *43*

ellipse_measure, 32
ellipse_measure(), *47, 48*
encoders, *6*, *23*, 34, *62*
evaluate, 36
evaluate(), *26*, *30*, *38*, *40*, *63*
evaluate_cases, 38
evaluate_cases(), *12*, *37–41*, *56*, *87*, *106*
evaluate_classes, 39
evaluate_classes(), *41*
evaluate_images, 40
export_weights, 42
export_weights(), *13*, *27*, *92*

flow_to_colours, 43
flow_to_colours(), *86*

inspect_masks, 44
inspect_masks(), *75*
instance_data, 45
instance_data(), *27*, *62, 63*, *76, 77*, *103*
instance_shapes, 47
instance_sizes, 48
instance_sizes(), *33*, *47, 48*, *87*

linknet *(*models*)*, 57
loss_boundary, 49
loss_boundary(), *52*, *56*
loss_cce *(*losses*)*, 50
loss_cce(), *22*
loss_cce_dice *(*losses*)*, 50
loss_combo *(*losses*)*, 50
loss_dice *(*losses*)*, 50
loss_focal *(*losses*)*, 50
loss_focal(), *22*
loss_focal_tversky *(*losses*)*, 50
loss_generalized_dice *(*losses*)*, 50
loss_iou *(*losses*)*, 50
loss_l1 *(*losses*)*, 50
loss_lovasz *(*losses*)*, 50
loss_mse *(*losses*)*, 50
loss_smooth_l1 *(*losses*)*, 50
loss_tversky *(*losses*)*, 50
losses, 50, *50*, *61*

mask_coverage, 53
mask_coverage(), *24*
mask_volume, 54
mask_volume(), *10*, *33*, *48–50*, *110*
metric_accuracy
*(*classification_metrics*)*, 20
metric_auc *(*classification_metrics*)*, 20
metric_auc(), *18*, *23*
metric_balanced_accuracy
*(*classification_metrics*)*, 20
metric_balanced_accuracy(), *18*
metric_cldice *(*metrics*)*, 55
metric_dice *(*metrics*)*, 55
metric_f1 *(*classification_metrics*)*, 20
metric_f1(), *18*
metric_iou *(*metrics*)*, 55
metric_surface_dice *(*metrics*)*, 55
metric_tversky *(*metrics*)*, 55
metrics, *20, 21*, 55, *61*
models, *35*, 57, *77*

optimizer_adadelta *(*optimizers*)*, 64
optimizer_adagrad *(*optimizers*)*, 64
optimizer_adam *(*optimizers*)*, 64
optimizer_adamax *(*optimizers*)*, 64
optimizer_adamw *(*optimizers*)*, 64
optimizer_nadam *(*optimizers*)*, 64
optimizer_rmsprop *(*optimizers*)*, 64
optimizer_sgd *(*optimizers*)*, 64
optimizers, *61*, 64
overlay_agreement, 67
overlay_agreement(), *32*, *68*
overlay_instance_agreement, 68
overlay_instance_agreement(), *70*
overlay_instances, 69
overlay_instances(), *48, 49*, *68, 69*, *87*
overlay_mask, 70
overlay_mask(), *70*

platypus_conditions, 71
platypus_device, 73
platypus_device(), *77*
platypus_fact, 74
platypus_fit, 75
platypus_fit(), *27, 28*, *36*, *39*, *41, 42*, *44*, *50*, *81, 82*, *86*, *108*
platypus_spec, 76
platypus_spec(), *6*, *16*, *19*, *23*, *31*, *35*, *44*, *46*, *63*, *75*, *96, 97*, *102*, *110*
platypus_status, 78

platypus_status(), *73*
platypus_use_torch, 79
platypus_use_torch(), *35*, *73*
plot.platypus_fit, 81
plot_anchors, 81
plot_anchors(), *32*
plot_boxes, 83
plot_boxes(), *31, 32*, *112*
plot_masks, 84
plot_masks(), *32*, *83, 84*
predict(), *28, 29*, *33*, *54*, *83*, *98*
predict.platypus_fit, 85
predict.platypus_fit(), *84, 85*, *95*

read_images, 89
read_images(), *83*
read_masks, 90
read_record, 91
read_record(), *97*
res_u_net *(*models*)*, 57

save_class_masks, 92
save_class_masks(), *86*, *94–96*, *100*
save_class_volumes, 94
save_class_volumes(), *100*
save_masks, 95
save_masks(), *92, 93*, *98*
save_spec, 96
save_volumes, 97
save_volumes(), *88*, *93–95*
segmentation_data, 99
segmentation_data(), *19*, *25*, *30, 31*, *46*, *60*, *76*, *106*
separation_flows *(*separations*)*, 102
separation_flows(), *43*, *52, 53*, *62*
separation_rim *(*separations*)*, 102
separation_rim(), *62*
separations, *6*, *27*, 102
series_report, 104
split(), *16*, *19*
split_dataset, 105
split_dataset(), *31*, *38, 39*, *45*, *77*, *99*, *101, 102*, *104*, *107*
split_path, 107
summary(), *12*, *108*
summary.platypus_cases *(*evaluate_cases*)*, 38

training_history, 108
training_history(), *13*, *91, 92*, *112*

u_net, *23*
u_net *(*models*)*, 57
u_net(), *9*, *22, 23*, *102*, *112*
u_net_plus_plus *(*models*)*, 57
unite_masks, 109

voc_colormap *(*colormaps*)*, 24
voc_labels *(*colormaps*)*, 24
volume_info, 109
volume_info(), *54*, *104*

yolo3, 110
yolo3(), *31*, *82*