In [ ]:
```
#hide
#skip
! [ -e /content ] && pip install -Uqq fastai  # upgrade fastai on colab
```

In [ ]:
```
#default_exp vision.data
```

In [ ]:
```
#export
from fastai.torch_basics import *
from fastai.data.all import *
from fastai.vision.core import *
import types
```

In [ ]:
```
#hide
from nbdev.showdoc import *
# from fastai.vision.augment import *
```

Vision data

> Helper functions to get data in a `DataLoaders` in the vision application and higher class `ImageDataLoaders`

The main classes defined in this module are `ImageDataLoaders` and `SegmentationDataLoaders`, so you probably want to jump to their definitions. They provide factory methods that are a great way to quickly get your data ready for training, see the vision tutorial for examples.

Helper functions

In [ ]:
```
#export
@delegates(subplots)
def get_grid(
    n:int, # Number of axes in the returned grid
    nrows:int=None, # Number of rows in the returned grid, defaulting to `int(math.
    ncols:int=None, # Number of columns in the returned grid, defaulting to `ceil(n
    add_vert=0,
    figsize:tuple=None, # Width, height in inches of the returned figure
    double:bool=False, # Whether to double the number of columns and `n`
    title:str=None, # If passed, title set to the figure
    return_fig:bool=False, # Whether to return the figure created by `subplots`
    flatten:bool=True, # Whether to flatten the matplot axes such that they can be
    **kwargs,
) -> (plt.Figure, plt.Axes): # Returns just `axs` by default, and (`fig`, `axs`) if
    "Return a grid of `n` axes, `rows` by `cols`"
    if nrows:
        ncols = ncols or int(np.ceil(n/nrows))
    elif ncols:
        nrows = nrows or int(np.ceil(n/ncols))
```

```
    else:
        nrows = int(math.sqrt(n))
        ncols = int(np.ceil(n/nrows))
    if double: ncols*=2 ; n*=2
    fig,axs = subplots(nrows, ncols, figsize=figsize, **kwargs)
    if flatten: axs = [ax if i<n else ax.set_axis_off() for i, ax in enumerate(axs.
    if title is not None: fig.suptitle(title, weight='bold', size=14)
    return (fig,axs) if return_fig else axs
```

This is used by the type-dispatched versions of `show_batch` and `show_results` for the vision application. The default `figsize` is `(cols*imsize, rows*imsize+0.6)`. `imsize` is passed down to `subplots`. `suptitle`, `sharex`, `sharey`, `squeeze`, `subplot_kw` and `gridspec_kw` are all passed down to plt.subplots. If `return_fig` is `True`, returns `fig,axs`, otherwise just `axs`.

In [ ]:
```
# export
def clip_remove_empty(
    bbox:TensorBBox, # Coordinates of bounding boxes
    label:TensorMultiCategory # Labels of the bounding boxes
):
    "Clip bounding boxes with image border and remove empty boxes along with corres
    bbox = torch.clamp(bbox, -1, 1)
    empty = ((bbox[...,2] - bbox[...,0])*(bbox[...,3] - bbox[...,1]) <= 0.)
    return (bbox[~empty], label[TensorBase(~empty)])
```

This is used in `bb_pad`

In [ ]:
```
bb = TensorBBox([[-2,-0.5,0.5,1.5], [-0.5,-0.5,0.5,0.5], [1,0.5,0.5,0.75], [-0.5,-0
bb,lbl = clip_remove_empty(bb, TensorMultiCategory([1,2,3,2,5]))
test_eq(bb, TensorBBox([[-1,-0.5,0.5,1.], [-0.5,-0.5,0.5,0.5], [-0.5,-0.5,0.5,0.5]]
test_eq(lbl, TensorMultiCategory([1,2,2]))
```

In [ ]:
```
#export
def bb_pad(
    samples:list, # List of 3-tuples like (image, bounding_boxes, labels)
    pad_idx:int=0 # Label that will be used to pad each list of labels
):
    "Function that collects `samples` of labelled bboxes and adds padding with `pad
    samples = [(s[0], *clip_remove_empty(*s[1:])) for s in samples]
    max_len = max([len(s[2]) for s in samples])
    def _f(img,bbox,lbl):
        bbox = torch.cat([bbox,bbox.new_zeros(max_len-bbox.shape[0], 4)])
        lbl  = torch.cat([lbl, lbl .new_zeros(max_len-lbl .shape[0])+pad_idx])
        return img,bbox,lbl
    return [_f(*s) for s in samples]
```

This is used in `BBoxBlock`

In [ ]:
```
img1,img2 = TensorImage(torch.randn(16,16,3)),TensorImage(torch.randn(16,16,3))
bb1 = tensor([[-2,-0.5,0.5,1.5], [-0.5,-0.5,0.5,0.5], [1,0.5,0.5,0.75], [-0.5,-0.5,
lbl1 = tensor([1, 2, 3, 2])
bb2 = tensor([[-0.5,-0.5,0.5,0.5], [-0.5,-0.5,0.5,0.5]])
lbl2 = tensor([2, 2])
```

```
samples = [(img1, bb1, lbl1), (img2, bb2, lbl2)]
res = bb_pad(samples)
non_empty = tensor([True,True,False,True])
test_eq(res[0][0], img1)
test_eq(res[0][1], tensor([[-1,-0.5,0.5,1.], [-0.5,-0.5,0.5,0.5], [-0.5,-0.5,0.5,0.
test_eq(res[0][2], tensor([1,2,2]))
test_eq(res[1][0], img2)
test_eq(res[1][1], tensor([[-0.5,-0.5,0.5,0.5], [-0.5,-0.5,0.5,0.5], [0,0,0,0]]))
test_eq(res[1][2], tensor([2,2,0]))
```

Show methods -

In [ ]:
```
#export
@typedispatch
def show_batch(x:TensorImage, y, samples, ctxs=None, max_n=10, nrows=None, ncols=No
    if ctxs is None: ctxs = get_grid(min(len(samples), max_n), nrows=nrows, ncols=n
    ctxs = show_batch[object](x, y, samples, ctxs=ctxs, max_n=max_n, **kwargs)
    return ctxs
```

In [ ]:
```
#export
@typedispatch
def show_batch(x:TensorImage, y:TensorImage, samples, ctxs=None, max_n=10, nrows=No
    if ctxs is None: ctxs = get_grid(min(len(samples), max_n), nrows=nrows, ncols=n
    for i in range(2):
        ctxs[i::2] = [b.show(ctx=c, **kwargs) for b,c,_ in zip(samples.itemgot(i),c
    return ctxs
```

`TransformBlock`s for vision

These are the blocks the vision application provide for the data block API.

In [ ]:
```
#export
def ImageBlock(cls=PILImage):
    "A `TransformBlock` for images of `cls`"
    return TransformBlock(type_tfms=cls.create, batch_tfms=IntToFloatTensor)
```

In [ ]:
```
#export
def MaskBlock(codes=None):
    "A `TransformBlock` for segmentation masks, potentially with `codes`"
    return TransformBlock(type_tfms=PILMask.create, item_tfms=AddMaskCodes(codes=co
```

In [ ]:
```
#export
PointBlock = TransformBlock(type_tfms=TensorPoint.create, item_tfms=PointScaler)
BBoxBlock = TransformBlock(type_tfms=TensorBBox.create, item_tfms=PointScaler, dls_

PointBlock.__doc__ = "A `TransformBlock` for points in an image"
BBoxBlock.__doc__  = "A `TransformBlock` for bounding boxes in an image"
```

In [ ]:
```
show_doc(PointBlock, name='PointBlock')
```

`PointBlock` [source]

A `TransformBlock` for points in an image

In [ ]:
```
show_doc(BBoxBlock, name='BBoxBlock')
```

`BBoxBlock` [source]

A `TransformBlock` for bounding boxes in an image

In [ ]:
```
#export
def BBoxLblBlock(vocab=None, add_na=True):
    "A `TransformBlock` for labeled bounding boxes, potentially with `vocab`"
    return TransformBlock(type_tfms=MultiCategorize(vocab=vocab, add_na=add_na), it
```

If `add_na` is `True`, a new category is added for NaN (that will represent the background class).

ImageDataLoaders -

In [ ]:
```
#export
class ImageDataLoaders(DataLoaders):
    "Basic wrapper around several `DataLoader`s with factory methods for computer v
    @classmethod
    @delegates(DataLoaders.from_dblock)
    def from_folder(cls, path, train='train', valid='valid', valid_pct=None, seed=N
                    batch_tfms=None, **kwargs):
        "Create from imagenet style dataset in `path` with `train` and `valid` subf
        splitter = GrandparentSplitter(train_name=train, valid_name=valid) if valid
        get_items = get_image_files if valid_pct else partial(get_image_files, fold
        dblock = DataBlock(blocks=(ImageBlock, CategoryBlock(vocab=vocab)),
                           get_items=get_items,
                           splitter=splitter,
                           get_y=parent_label,
                           item_tfms=item_tfms,
                           batch_tfms=batch_tfms)
        return cls.from_dblock(dblock, path, path=path, **kwargs)

    @classmethod
    @delegates(DataLoaders.from_dblock)
    def from_path_func(cls, path, fnames, label_func, valid_pct=0.2, seed=None, ite
        "Create from list of `fnames` in `path`s with `label_func`"
        dblock = DataBlock(blocks=(ImageBlock, CategoryBlock),
                           splitter=RandomSplitter(valid_pct, seed=seed),
                           get_y=label_func,
                           item_tfms=item_tfms,
                           batch_tfms=batch_tfms)
        return cls.from_dblock(dblock, fnames, path=path, **kwargs)

    @classmethod
    def from_name_func(cls,
```

```
    path:(str, Path), # Set the default path to a directory that a `Learner` ca
    fnames:list, # A list of `os.Pathlike`'s to individual image files
    label_func:callable, # A function that receives a string (the file name) an
    **kwargs
) -> DataLoaders:
    "Create from the name attrs of `fnames` in `path`s with `label_func`"
    if sys.platform == 'win32' and isinstance(label_func, types.LambdaType) and
        # https://medium.com/@jwnx/multiprocessing-serialization-in-python-with
        raise ValueError("label_func couldn't be lambda function on Windows")
    f = using_attr(label_func, 'name')
    return cls.from_path_func(path, fnames, f, **kwargs)

@classmethod
def from_path_re(cls, path, fnames, pat, **kwargs):
    "Create from list of `fnames` in `path`s with re expression `pat`"
    return cls.from_path_func(path, fnames, RegexLabeller(pat), **kwargs)

@classmethod
@delegates(DataLoaders.from_dblock)
def from_name_re(cls, path, fnames, pat, **kwargs):
    "Create from the name attrs of `fnames` in `path`s with re expression `pat`
    return cls.from_name_func(path, fnames, RegexLabeller(pat), **kwargs)

@classmethod
@delegates(DataLoaders.from_dblock)
def from_df(cls, df, path='.', valid_pct=0.2, seed=None, fn_col=0, folder=None,
            y_block=None, valid_col=None, item_tfms=None, batch_tfms=None, **kw
    "Create from `df` using `fn_col` and `label_col`"
    pref = f'{Path(path) if folder is None else Path(path)/folder}{os.path.sep}
    if y_block is None:
        is_multi = (is_listy(label_col) and len(label_col) > 1) or label_delim
        y_block = MultiCategoryBlock if is_multi else CategoryBlock
    splitter = RandomSplitter(valid_pct, seed=seed) if valid_col is None else C
    dblock = DataBlock(blocks=(ImageBlock, y_block),
                       get_x=ColReader(fn_col, pref=pref, suff=suff),
                       get_y=ColReader(label_col, label_delim=label_delim),
                       splitter=splitter,
                       item_tfms=item_tfms,
                       batch_tfms=batch_tfms)
    return cls.from_dblock(dblock, df, path=path, **kwargs)

@classmethod
def from_csv(cls, path, csv_fname='labels.csv', header='infer', delimiter=None,
    "Create from `path/csv_fname` using `fn_col` and `label_col`"
    df = pd.read_csv(Path(path)/csv_fname, header=header, delimiter=delimiter)
    return cls.from_df(df, path=path, **kwargs)

@classmethod
@delegates(DataLoaders.from_dblock)
def from_lists(cls, path, fnames, labels, valid_pct=0.2, seed:int=None, y_block
               **kwargs):
    "Create from list of `fnames` and `labels` in `path`"
    if y_block is None:
        y_block = MultiCategoryBlock if is_listy(labels[0]) and len(labels[0])
            RegressionBlock if isinstance(labels[0], float) else CategoryBlock)
    dblock = DataBlock.from_columns(blocks=(ImageBlock, y_block),
```

```
                           splitter=RandomSplitter(valid_pct, seed=seed),
                           item_tfms=item_tfms,
                           batch_tfms=batch_tfms)
        return cls.from_dblock(dblock, (fnames, labels), path=path, **kwargs)

ImageDataLoaders.from_csv = delegates(to=ImageDataLoaders.from_df)(ImageDataLoaders
ImageDataLoaders.from_name_func = delegates(to=ImageDataLoaders.from_path_func)(Ima
ImageDataLoaders.from_path_re = delegates(to=ImageDataLoaders.from_path_func)(Image
ImageDataLoaders.from_name_re = delegates(to=ImageDataLoaders.from_name_func)(Image
```

This class should not be used directly, one of the factory methods should be preferred instead. All those factory methods accept as arguments:

- `item_tfms` : one or several transforms applied to the items before batching them
- `batch_tfms` : one or several transforms applied to the batches once they are formed
- `bs` : the batch size
- `val_bs` : the batch size for the validation `DataLoader` (defaults to `bs`)
- `shuffle_train` : if we shuffle the training `DataLoader` or not
- `device` : the PyTorch device to use (defaults to `default_device()`)

In [ ]:
```
show_doc(ImageDataLoaders.from_folder)
```

`ImageDataLoaders.from_folder` [source]

> `ImageDataLoaders.from_folder` (**`path`** , **`train`** = *`'train'`* , **`valid`** = *`'valid'`* , **`valid_pct`** = *`None`* , **`seed`** = *`None`* , **`vocab`** = *`None`* , **`item_tfms`** = *`None`* , **`batch_tfms`** = *`None`* , **`bs`** = *`64`* , **`val_bs`** = *`None`* , **`shuffle`** = *`True`* , **`device`** = *`None`*)

Create from imagenet style dataset in `path` with `train` and `valid` subfolders (or provide `valid_pct`)

If `valid_pct` is provided, a random split is performed (with an optional `seed`) by setting aside that percentage of the data for the validation set (instead of looking at the grandparents folder). If a `vocab` is passed, only the folders with names in `vocab` are kept.

Here is an example loading a subsample of MNIST:

In [ ]:
```
path = untar_data(URLs.MNIST_TINY)
dls = ImageDataLoaders.from_folder(path)
```

Passing `valid_pct` will ignore the valid/train folders and do a new random split:

In [ ]:
```
dls = ImageDataLoaders.from_folder(path, valid_pct=0.2)
dls.valid_ds.items[:3]
```

Out[ ]:
```
[Path('/home/hamel/.fastai/data/mnist_tiny/valid/3/9742.png'),
 Path('/home/hamel/.fastai/data/mnist_tiny/valid/3/807.png'),
 Path('/home/hamel/.fastai/data/mnist_tiny/valid/3/8308.png')]
```

In [ ]:
```
show_doc(ImageDataLoaders.from_path_func)
```

`ImageDataLoaders.from_path_func` [source]

> `ImageDataLoaders.from_path_func(path, fnames, label_func, valid_pct=0.2, seed=None, item_tfms=None, batch_tfms=None, bs=64, val_bs=None, shuffle=True, device=None)`

Create from list of `fnames` in `path`s with `label_func`

The validation set is a random `subset` of `valid_pct`, optionally created with `seed` for reproducibility.

Here is how to create the same `DataLoaders` on the MNIST dataset as the previous example with a `label_func`:

In [ ]:
```
fnames = get_image_files(path)
def label_func(x): return x.parent.name
dls = ImageDataLoaders.from_path_func(path, fnames, label_func)
```

Here is another example on the pets dataset. Here filenames are all in an "images" folder and their names have the form `class_name_123.jpg`. One way to properly label them is thus to throw away everything after the last `_`:

In [ ]:
```
show_doc(ImageDataLoaders.from_path_re)
```

`ImageDataLoaders.from_path_re` [source]

> `ImageDataLoaders.from_path_re(path, fnames, pat, valid_pct=0.2, seed=None, item_tfms=None, batch_tfms=None, bs=64, val_bs=None, shuffle=True, device=None)`

Create from list of `fnames` in `path`s with re expression `pat`

The validation set is a random subset of `valid_pct`, optionally created with `seed` for reproducibility.

Here is how to create the same `DataLoaders` on the MNIST dataset as the previous example (you will need to change the initial two / by a \ on Windows):

In [ ]:
```
pat = r'/([^/]*)/\d+.png$'
dls = ImageDataLoaders.from_path_re(path, fnames, pat)
```

In [ ]:
```
show_doc(ImageDataLoaders.from_name_func)
```

`ImageDataLoaders.from_name_func` [source]

> `ImageDataLoaders.from_name_func(path:Path'>), fnames, label_func, valid_pct=0.2, seed=None, item_tfms=None, batch_tfms=None, bs=64, val_bs=None, shuffle=True, device=None)`

Create from the name attrs of `fnames` in `path`s with `label_func`

	Type	Default	Details
`path`	`(str, Path)`		The path to a directory that a downstream learner will use to save files like models.
`fnames`			A list of `pathlib.Path` to individual image files.
`label_func`			A function that receives a string (the file name) and outputs a label.
`valid_pct`	`float`	`0.2`	*No Content*
`seed`	`NoneType`	`None`	*No Content*
`item_tfms`	`NoneType`	`None`	*No Content*
`batch_tfms`	`NoneType`	`None`	*No Content*
`bs`	`int`	`64`	*No Content*
`val_bs`	`NoneType`	`None`	*No Content*
`shuffle`	`bool`	`True`	*No Content*
`device`	`NoneType`	`None`	*No Content*
Returns	`DataLoaders`		

The validation set is a random subset of `valid_pct`, optionally created with `seed` for reproducibility. This method does the same as `ImageDataLoaders.from_path_func` except `label_func` is applied to the name of each filenames, and not the full path.

In [ ]:
```
show_doc(ImageDataLoaders.from_name_re)
```

`ImageDataLoaders.from_name_re` [source]

> `ImageDataLoaders.from_name_re(path, fnames, pat, bs=64, val_bs=None, shuffle=True, device=None)`

Create from the name attrs of `fnames` in `path`s with re expression `pat`

The validation set is a random subset of `valid_pct`, optionally created with `seed` for reproducibility. This method does the same as `ImageDataLoaders.from_path_re` except

`pat` is applied to the name of each filenames, and not the full path.

In [ ]:
```
show_doc(ImageDataLoaders.from_df)
```

`ImageDataLoaders.from_df` [source]

> `ImageDataLoaders.from_df(df, path='.', valid_pct=0.2, seed=None, fn_col=0, folder=None, suff='', label_col=1, label_delim=None, y_block=None, valid_col=None, item_tfms=None, batch_tfms=None, bs=64, val_bs=None, shuffle=True, device=None)`

Create from `df` using `fn_col` and `label_col`

The validation set is a random subset of `valid_pct`, optionally created with `seed` for reproducibility. Alternatively, if your `df` contains a `valid_col`, give its name or its index to that argument (the column should have `True` for the elements going to the validation set).

You can add an additional `folder` to the filenames in `df` if they should not be concatenated directly to `path`. If they do not contain the proper extensions, you can add `suff`. If your label column contains multiple labels on each row, you can use `label_delim` to warn the library you have a multi-label problem.

`y_block` should be passed when the task automatically picked by the library is wrong, you should then give `CategoryBlock`, `MultiCategoryBlock` or `RegressionBlock`. For more advanced uses, you should use the data block API.

The tiny mnist example from before also contains a version in a dataframe:

In [ ]:
```
path = untar_data(URLs.MNIST_TINY)
df = pd.read_csv(path/'labels.csv')
df.head()
```

Out[ ]:

	name	label
0	train/3/7463.png	3
1	train/3/9829.png	3
2	train/3/7881.png	3
3	train/3/8065.png	3
4	train/3/7046.png	3

Here is how to load it using `ImageDataLoaders.from_df`:

In [ ]:
```
dls = ImageDataLoaders.from_df(df, path)
```

Here is another example with a multi-label problem:

In [ ]:
```
path = untar_data(URLs.PASCAL_2007)
df = pd.read_csv(path/'train.csv')
df.head()
```


100.00% [1637801984/1637796771 02:06<00:00]

Out[ ]:

	fname	labels	is_valid
0	000005.jpg	chair	True
1	000007.jpg	car	True
2	000009.jpg	horse person	True
3	000012.jpg	car	False
4	000016.jpg	bicycle	True

In [ ]:
```
dls = ImageDataLoaders.from_df(df, path, folder='train', valid_col='is_valid')
```

Note that can also pass `2` to valid_col (the index, starting with 0).

In [ ]:
```
show_doc(ImageDataLoaders.from_csv)
```

`ImageDataLoaders.from_csv` [source]

> `ImageDataLoaders.from_csv(` **`path`**, **`csv_fname`** = *`'labels.csv'`*, **`header`** = *`'infer'`*, **`delimiter`** = *`None`*, **`valid_pct`** = *`0.2`*, **`seed`** = *`None`*, **`fn_col`** = *`0`*, **`folder`** = *`None`*, **`suff`** = *`''`*, **`label_col`** = *`1`*, **`label_delim`** = *`None`*, **`y_block`** = *`None`*, **`valid_col`** = *`None`*, **`item_tfms`** = *`None`*, **`batch_tfms`** = *`None`*, **`bs`** = *`64`*, **`val_bs`** = *`None`*, **`shuffle`** = *`True`*, **`device`** = *`None`*)

Create from `path/csv_fname` using `fn_col` and `label_col`

Same as `ImageDataLoaders.from_df` after loading the file with `header` and `delimiter`.

Here is how to load the same dataset as before with this method:

In [ ]:
```
dls = ImageDataLoaders.from_csv(path, 'train.csv', folder='train', valid_col='is_va
```

In [ ]:
```
show_doc(ImageDataLoaders.from_lists)
```

`ImageDataLoaders.from_lists` [source]

> `ImageDataLoaders.from_lists(path, fnames, labels, valid_pct=0.2, seed:int=None, y_block=None, item_tfms=None, batch_tfms=None, bs=64, val_bs=None, shuffle=True, device=None)`

Create from list of `fnames` and `labels` in `path`

The validation set is a random subset of `valid_pct`, optionally created with `seed` for reproducibility. `y_block` can be passed to specify the type of the targets.

In [ ]:
```
path = untar_data(URLs.PETS)
fnames = get_image_files(path/"images")
labels = ['_'.join(x.name.split('_')[:-1]) for x in fnames]
dls = ImageDataLoaders.from_lists(path, fnames, labels)
```

In [ ]:
```
#export
class SegmentationDataLoaders(DataLoaders):
    "Basic wrapper around several `DataLoader`s with factory methods for segmentati
    @classmethod
    @delegates(DataLoaders.from_dblock)
    def from_label_func(cls, path, fnames, label_func, valid_pct=0.2, seed=None, co
        "Create from list of `fnames` in `path`s with `label_func`."
        dblock = DataBlock(blocks=(ImageBlock, MaskBlock(codes=codes)),
                           splitter=RandomSplitter(valid_pct, seed=seed),
                           get_y=label_func,
                           item_tfms=item_tfms,
                           batch_tfms=batch_tfms)
        res = cls.from_dblock(dblock, fnames, path=path, **kwargs)
        return res
```

In [ ]:
```
show_doc(SegmentationDataLoaders.from_label_func)
```

`SegmentationDataLoaders.from_label_func` [source]

> `SegmentationDataLoaders.from_label_func(path, fnames, label_func, valid_pct=0.2, seed=None, codes=None, item_tfms=None, batch_tfms=None, bs=64, val_bs=None, shuffle=True, device=None)`

Create from list of `fnames` in `path`s with `label_func`.

The validation set is a random subset of `valid_pct`, optionally created with `seed` for reproducibility. `codes` contain the mapping index to label.

In [ ]:
```
path = untar_data(URLs.CAMVID_TINY)
fnames = get_image_files(path/'images')
def label_func(x): return path/'labels'/f'{x.stem}_P{x.suffix}'
codes = np.loadtxt(path/'codes.txt', dtype=str)
```

```
dls = SegmentationDataLoaders.from_label_func(path, fnames, label_func, codes=codes
```

```
/home/hamel/anaconda3/lib/python3.9/site-packages/torch/_tensor.py:1051: UserWarnin
g: __floordiv__ is deprecated, and its behavior will change in a future version of p
ytorch. It currently rounds toward 0 (like the 'trunc' function NOT 'floor'). This r
esults in incorrect rounding for negative values. To keep the current behavior, use
torch.div(a, b, rounding_mode='trunc'), or for actual floor division, use torch.div
(a, b, rounding_mode='floor').
  ret = func(*args, **kwargs)
```

Export -

In [ ]:
```
#hide
from nbdev.export import notebook2script
notebook2script()
```

```
Converted 00_torch_core.ipynb.
Converted 01_layers.ipynb.
Converted 01a_losses.ipynb.
Converted 02_data.load.ipynb.
Converted 03_data.core.ipynb.
Converted 04_data.external.ipynb.
Converted 05_data.transforms.ipynb.
Converted 06_data.block.ipynb.
Converted 07_vision.core.ipynb.
Converted 08_vision.data.ipynb.
Converted 09_vision.augment.ipynb.
Converted 09b_vision.utils.ipynb.
Converted 09c_vision.widgets.ipynb.
Converted 10_tutorial.pets.ipynb.
Converted 10b_tutorial.albumentations.ipynb.
Converted 11_vision.models.xresnet.ipynb.
Converted 12_optimizer.ipynb.
Converted 13_callback.core.ipynb.
Converted 13a_learner.ipynb.
Converted 13b_metrics.ipynb.
Converted 14_callback.schedule.ipynb.
Converted 14a_callback.data.ipynb.
Converted 15_callback.hook.ipynb.
Converted 15a_vision.models.unet.ipynb.
Converted 16_callback.progress.ipynb.
Converted 17_callback.tracker.ipynb.
Converted 18_callback.fp16.ipynb.
Converted 18a_callback.training.ipynb.
Converted 18b_callback.preds.ipynb.
Converted 19_callback.mixup.ipynb.
Converted 20_interpret.ipynb.
Converted 20a_distributed.ipynb.
Converted 21_vision.learner.ipynb.
Converted 22_tutorial.imagenette.ipynb.
Converted 23_tutorial.vision.ipynb.
Converted 24_tutorial.image_sequence.ipynb.
Converted 24_tutorial.siamese.ipynb.
Converted 24_vision.gan.ipynb.
Converted 30_text.core.ipynb.
Converted 31_text.data.ipynb.
Converted 32_text.models.awdlstm.ipynb.
Converted 33_text.models.core.ipynb.
Converted 34_callback.rnn.ipynb.
Converted 35_tutorial.wikitext.ipynb.
Converted 37_text.learner.ipynb.
Converted 38_tutorial.text.ipynb.
Converted 39_tutorial.transformers.ipynb.
Converted 40_tabular.core.ipynb.
Converted 41_tabular.data.ipynb.
Converted 42_tabular.model.ipynb.
Converted 43_tabular.learner.ipynb.
Converted 44_tutorial.tabular.ipynb.
Converted 45_collab.ipynb.
Converted 46_tutorial.collab.ipynb.
Converted 50_tutorial.datablock.ipynb.
Converted 60_medical.imaging.ipynb.
```

```
Converted 61_tutorial.medical_imaging.ipynb.
Converted 65_medical.text.ipynb.
Converted 70_callback.wandb.ipynb.
Converted 71_callback.tensorboard.ipynb.
Converted 72_callback.neptune.ipynb.
Converted 73_callback.captum.ipynb.
Converted 74_callback.azureml.ipynb.
Converted 97_test_utils.ipynb.
Converted 99_pytorch_doc.ipynb.
Converted dev-setup.ipynb.
Converted app_examples.ipynb.
Converted camvid.ipynb.
Converted migrating_catalyst.ipynb.
Converted migrating_ignite.ipynb.
Converted migrating_lightning.ipynb.
Converted migrating_pytorch.ipynb.
Converted migrating_pytorch_verbose.ipynb.
Converted ulmfit.ipynb.
Converted index.ipynb.
Converted index_original.ipynb.
Converted quick_start.ipynb.
Converted tutorial.ipynb.
```

In [ ]: