fastai V1 for PyTorch

document created Thu Oct 4 10:44:26 DST 2018

Contents

Introduction

This PDF has been generated by C. Klukas on the base of the fastai documentation, hosted at https://github.com/fastai/fastai_docs. Content authors are listed in the section **Credits**. The fastai documentation has been published under the Apache License, Version 2.0 (Apache License Version 2.0, January 2004 http://www.apache.org/licenses/). The terms of the license are listed under the section **License for Documentation**. This documents sections may be incomplete or appear in the wrong order. Go to http://docs.fast.ai/ to view the official online documentation of fastai.

Credits

Development Lead

- Jeremy Howard

Core Team

- Sylvain Gugger

Main Contributors

- Stas Beckman
- Fransesco Ingham
- Fred Monroe
- Andrew Shaw
- Rachel Thomas

And a big thanks to all of our GitHub contributors!

Individual Contributor License Agreement

Adapted from http://www.apache.org/licenses/icla.txt © The Apache Software Foundation

Thank you for your interest in fast.ai, Inc (the "Company"). In order to clarify the intellectual property license granted with Contributions from any person or entity, the Company must have a Contributor License Agreement ("CLA") on file that has been signed by each Contributor, indicating agreement to the license terms below. This license is for your protection as a Contributor as well as the protection of the Company and its users; it does not change your rights to use your own Contributions for any other purpose.

You accept and agree to the following terms and conditions for Your present and future Contributions submitted to the Company. In return, the Company shall not use Your Contributions in a way that is contrary to the public benefit or inconsistent with its bylaws in effect at the time of the Contribution. Except for the license granted herein to the Company and recipients of software distributed by the Company, You reserve all right, title, and interest in and to Your Contributions.

1. Definitions. "**You**" (or "**Your**"): "You" (or "Your") shall mean the copyright owner or legal entity authorized by the copyright owner that is making this Agreement with the Company. For legal entities, the entity making a Contribution and all other entities that control, are controlled by, or are under common control with that entity are considered to be a single Contributor. For the purposes of this definition, "control" means (i) the power, direct or indirect, to cause the direction or management of such entity, whether by contract or otherwise, or (ii) ownership of fifty percent (50%) or more of the outstanding shares, or (iii) beneficial ownership of such entity. "**Contribution**": "Contribution" shall mean any original work of authorship, including any modifications or additions to an existing work, that is intentionally submitted by You to the Company for inclusion in, or documentation of, any of the products owned or managed by the Company (the "Work"). For the purposes of this definition, "submitted" means any form of electronic, verbal, or written communication sent to

the Company or its representatives, including but not limited to communication on electronic mailing lists, source code control systems, and issue tracking systems that are managed by, or on behalf of, the Company for the purpose of discussing and improving the Work, but excluding communication that is conspicuously marked or otherwise designated in writing by You as "Not a Contribution."

2. Grant of Copyright License. Subject to the terms and conditions of this Agreement, You hereby grant to the Company and to recipients of software distributed by the Company a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable copyright license to reproduce, prepare derivative works of, publicly display, publicly perform, sublicense, and distribute Your Contributions and such derivative works.
3. Grant of Patent License. Subject to the terms and conditions of this Agreement, You hereby grant to the Company and to recipients of software distributed by the Company a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable (except as stated in this section) patent license to make, have made, use, offer to sell, sell, import, and otherwise transfer the Work, where such license applies only to those patent claims licensable by You that are necessarily infringed by Your Contribution(s) alone or by combination of Your Contribution(s) with the Work to which such Contribution(s) was submitted. If any entity institutes patent litigation against You or any other entity (including a cross-claim or counterclaim in a lawsuit) alleging that your Contribution, or the Work to which you have contributed, constitutes direct or contributory patent infringement, then any patent licenses granted to that entity under this Agreement for that Contribution or Work shall terminate as of the date such litigation is filed.
4. You represent that you are legally entitled to grant the above license. If your employer(s) has rights to intellectual property that you create that includes your Contributions, you represent that you have received permission to make Contributions on behalf of that employer, that your employer has waived such rights for your Contributions to the Company, or that your employer has executed a separate Corporate CLA with the Company.
5. You represent that each of Your Contributions is Your original creation (see section 7 for submissions on behalf of others). You represent that Your Contribution submissions include complete details of any third-party license or other restriction (including, but not limited to, related patents and trademarks) of which you are personally aware and which are associated with any part of Your Contributions.
6. You are not expected to provide support for Your Contributions, except to the extent You desire to provide support. You may provide support for free, for a fee, or not at all. Unless required by applicable law or agreed to in writing, You provide Your Contributions on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied, including, without limitation, any warranties or conditions of

TITLE, NON- INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A PARTICULAR PURPOSE.

7. Should You wish to submit work that is not Your original creation, You may submit it to the Company separately from any Contribution, identifying the complete details of its source and of any license or other restriction (including, but not limited to, related patents, trademarks, and license agreements) of which you are personally aware, and conspicuously marking the work as "Submitted on behalf of a third-party: [named here]".
8. You agree to notify the Company of any facts or circumstances of which you become aware that would make these representations inaccurate in any respect.

Contributor Covenant Code of Conduct

Our Pledge

In the interest of fostering an open and welcoming environment, we as contributors and maintainers pledge to making participation in our project and our community a harassment-free experience for everyone, regardless of age, body size, disability, ethnicity, gender identity and expression, level of experience, education, socio-economic status, nationality, personal appearance, race, religion, or sexual identity and orientation.

Our Standards

Examples of behavior that contributes to creating a positive environment include:

- Using welcoming and inclusive language
- Being respectful of differing viewpoints and experiences
- Gracefully accepting constructive criticism
- Focusing on what is best for the community
- Showing empathy towards other community members

Examples of unacceptable behavior by participants include:

- The use of sexualized language or imagery and unwelcome sexual attention or advances
- Trolling, insulting/derogatory comments, and personal or political attacks
- Public or private harassment
- Publishing others' private information, such as a physical or electronic address, without explicit permission
- Other conduct which could reasonably be considered inappropriate in a professional setting

Our Responsibilities

Project maintainers are responsible for clarifying the standards of acceptable behavior and are expected to take appropriate and fair corrective action in response to any instances of unacceptable behavior.

Project maintainers have the right and responsibility to remove, edit, or reject comments, commits, code, wiki edits, issues, and other contributions that are not aligned to this Code of Conduct, or to ban temporarily or permanently any contributor for other behaviors that they deem inappropriate, threatening, offensive, or harmful.

Scope

This Code of Conduct applies both within project spaces and in public spaces when an individual is representing the project or its community. Examples of representing a project or community include using an official project e-mail address, posting via an official social media account, or acting as an appointed representative at an online or offline event. Representation of a project may be further defined and clarified by project maintainers.

Enforcement

Instances of abusive, harassing, or otherwise unacceptable behavior may be reported by contacting the project team at info@fast.ai. All complaints will be reviewed and investigated and will result in a response that is deemed necessary and appropriate to the circumstances. The project team is obligated to maintain confidentiality with regard to the reporter of an incident. Further details of specific enforcement policies may be posted separately.

Project maintainers who do not follow or enforce the Code of Conduct in good faith may face temporary or permanent repercussions as determined by other members of the project's leadership.

Attribution

This Code of Conduct is adapted from the Contributor Covenant, version 1.4, available at https://www.contributor-covenant.org/version/1/4/code-of-conduct.html

How to contribute to fastai

First, thanks a lot for wanting to help! Make sure you have read the doc on code style first. (Note that we don't follow PEP8, but instead follow a coding style designed specifically for numerical and interactive programming.)

Did you find a bug?

- Nobody is perfect, especially not us. But first, please double-check the bug doesn't come from something on your side. The forum is a tremendous source for help, and we'd advise to use it as a first step. Be sure to include as much code as you can so that other people can easily help you.
- Then, ensure the bug was not already reported by searching on GitHub under Issues.
- If you're unable to find an open issue addressing the problem, open a new one. Be sure to include a title and clear description, as much relevant information as possible, and a code sample or an executable test case demonstrating the expected behavior that is not occurring.
- Be sure to add the complete error messages.

Do you want to contribute to the documentation?

- Sign the Contributor License Agreement.
- Please read Contributing to the documentation

License for Documentation

to cause the direction or management of such entity, whether by contract or otherwise, or (ii) ownership of fifty percent (50%) or more of the outstanding shares, or (iii) beneficial ownership of such entity.

"You" (or "Your") shall mean an individual or Legal Entity exercising permissions granted by this License.

"Source" form shall mean the preferred form for making modifications, including but not limited to software source code, documentation source, and configuration files.

"Object" form shall mean any form resulting from mechanical transformation or translation of a Source form, including but not limited to compiled object code, generated documentation, and conversions to other media types.

"Work" shall mean the work of authorship, whether in Source or Object form, made available under the License, as indicated by a copyright notice that is included in or attached to the work (an example is provided in the Appendix below).

"Derivative Works" shall mean any work, whether in Source or Object form, that is based on (or derived from) the Work and for which the editorial revisions, annotations, elaborations, or other modifications represent, as a whole, an original work of authorship. For the purposes of this License, Derivative Works shall not include works that remain separable from, or merely link (or bind by name) to the interfaces of, the Work and Derivative Works thereof.

"Contribution" shall mean any work of authorship, including the original version of the Work and any modifications or additions to that Work or Derivative Works thereof, that is intentionally submitted to Licensor for inclusion in the Work by the copyright owner or by an individual or Legal Entity authorized to submit on behalf of the copyright owner. For the purposes of this definition, "submitted" means any form of electronic, verbal, or written communication sent to the Licensor or its representatives, including but not limited to communication on electronic mailing lists, source code control systems, and issue tracking systems that are managed by, or on behalf of, the Licensor for the purpose of discussing and improving the Work, but excluding communication that is conspicuously marked or otherwise designated in writing by the copyright owner as "Not a Contribution."

"Contributor" shall mean Licensor and any individual or Legal Entity on behalf of whom a Contribution has been received by Licensor and subsequently incorporated within the Work.

2. Grant of Copyright License.

Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable copyright license to reproduce, prepare Derivative Works of, publicly display, publicly perform, sublicense, and distribute the Work and such

Derivative Works in Source or Object form.

3. Grant of Patent License.

Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable (except as stated in this section) patent license to make, have made, use, offer to sell, sell, import, and otherwise transfer the Work, where such license applies only to those patent claims licensable by such Contributor that are necessarily infringed by their Contribution(s) alone or by combination of their Contribution(s) with the Work to which such Contribution(s) was submitted. If You institute patent litigation against any entity (including a cross-claim or counterclaim in a lawsuit) alleging that the Work or a Contribution incorporated within the Work constitutes direct or contributory patent infringement, then any patent licenses granted to You under this License for that Work shall terminate as of the date such litigation is filed.

4. Redistribution.

You may reproduce and distribute copies of the Work or Derivative Works thereof in any medium, with or without modifications, and in Source or Object form, provided that You meet the following conditions:

You must give any other recipients of the Work or Derivative Works a copy of this License; and You must cause any modified files to carry prominent notices stating that You changed the files; and You must retain, in the Source form of any Derivative Works that You distribute, all copyright, patent, trademark, and attribution notices from the Source form of the Work, excluding those notices that do not pertain to any part of the Derivative Works; and If the Work includes a "NOTICE" text file as part of its distribution, then any Derivative Works that You distribute must include a readable copy of the attribution notices contained within such NOTICE file, excluding those notices that do not pertain to any part of the Derivative Works, in at least one of the following places: within a NOTICE text file distributed as part of the Derivative Works; within the Source form or documentation, if provided along with the Derivative Works; or, within a display generated by the Derivative Works, if and wherever such third-party notices normally appear. The contents of the NOTICE file are for informational purposes only and do not modify the License. You may add Your own attribution notices within Derivative Works that You distribute, alongside or as an addendum to the NOTICE text from the Work, provided that such additional attribution notices cannot be construed as modifying the License. You may add Your own copyright statement to Your modifications and may provide additional or different license terms and conditions for use, reproduction, or distribution of Your modifications, or for any such Derivative Works as a whole, provided Your use, reproduction, and distribution of the Work otherwise complies with the conditions stated in this License.

5. Submission of Contributions.

Unless You explicitly state otherwise, any Contribution intentionally submitted for inclusion in the Work by You to the Licensor shall be under the terms and conditions of this License, without any additional terms or conditions. Notwithstanding the above, nothing herein shall supersede or modify the terms of any separate license agreement you may have executed with Licensor regarding such Contributions.

6. Trademarks.

This License does not grant permission to use the trade names, trademarks, service marks, or product names of the Licensor, except as required for reasonable and customary use in describing the origin of the Work and reproducing the content of the NOTICE file.

7. Disclaimer of Warranty.

Unless required by applicable law or agreed to in writing, Licensor provides the Work (and each Contributor provides its Contributions) on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied, including, without limitation, any warranties or conditions of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A PARTICULAR PURPOSE. You are solely responsible for determining the appropriateness of using or redistributing the Work and assume any risks associated with Your exercise of permissions under this License.

8. Limitation of Liability.

In no event and under no legal theory, whether in tort (including negligence), contract, or otherwise, unless required by applicable law (such as deliberate and grossly negligent acts) or agreed to in writing, shall any Contributor be liable to You for damages, including any direct, indirect, special, incidental, or consequential damages of any character arising as a result of this License or out of the use or inability to use the Work (including but not limited to damages for loss of goodwill, work stoppage, computer failure or malfunction, or any and all other commercial damages or losses), even if such Contributor has been advised of the possibility of such damages.

9. Accepting Warranty or Additional Liability.

While redistributing the Work or Derivative Works thereof, You may choose to offer, and charge a fee for, acceptance of support, warranty, indemnity, or other liability obligations and/or rights consistent with this License. However, in accepting such obligations, You may act only on Your own behalf and on Your sole responsibility, not on behalf of any other Contributor, and only if You agree to indemnify, defend, and hold each Contributor harmless for any liability incurred by, or claims asserted against, such Contributor by reason of your accepting any such warranty or additional liability.

Welcome to fastai

```
from fastai import *
from fastai.vision import *
from fastai.docs import *
from fastai.gen_doc.nbdoc import *
from fastai.core import *
from fastai.basic_train import *
from fastai.collab import *
from fastai.tabular import *
```

The fastai library simplifies training fast and accurate neural nets using modern best practices. It's based on research in to deep learning best practices undertaken at fast.ai, including "out of the box" support for `vision`, `text`, `tabular`, and `collab` (collaborative filtering) models. If you're looking for the source code, head over to the fastai repo on GitHub. For brief examples, see the examples folder; detailed examples are provided in the full documentation (see the sidebar). For example, here's how to train an MNIST model using resnet18 (from the vision example):

```
untar_data(MNIST_PATH)
data = image_data_from_folder(MNIST_PATH)
learn = ConvLearner(data, tvm.resnet18, metrics=accuracy)
learn.fit(1)
```

```
Total time: 00:09
epoch  train loss  valid loss  accuracy
0      0.089204    0.038701    0.986752  (00:09)
```

```
jekyll_note("""This documentation is all built from notebooks;
that means that you can try any of the code you see in any notebook yourself!
You'll find the notebooks in the <a href="https://github.com/fastai/fastai_docs/tree/master/do
<a href="https://github.com/fastai/fastai_docs">fastai_docs</a> repo. For instance,
<a href="https://github.com/fastai/fastai_docs/blob/master/docs_src/index.ipynb">here</a>
is the notebook source of what you're reading now.""")
```

Note: This documentation is all built from notebooks; that means that you can try any of the code you see in any notebook yourself! You'll find the notebooks in the docs_src folder of the fastai_docs repo. For instance, here is the notebook source of what you're reading now.

Installation

To install fastai, we recommend `conda` (replace `cuda92` with your CUDA toolkit version):

```
conda install -c pytorch -c fastai fastai pytorch-nightly cuda92
```

For alternative installations, including pip and CPU-only options, see the fastai readme.

Reading the docs

To get started quickly, click *Applications* on the sidebar, and then choose the application you're interested in. That will take you to a walk-through of training a model of that type. You can then either explore the various links from there, or dive more deeply into the various fastai modules.

We've provided below a quick summary of the key modules in this library. For details on each one, use the sidebar to find the module you're interested in. Each module includes an overview and example of how to use it, along with documentation for every class, function, and method. API documentation looks, for example, like this:

An example function

```
show_doc(rotate)
```

`rotate`

> `rotate(degrees:uniform) -> Image :: TfmAffine`

Rotate image by `degrees`. [source]

Types for each parameter, and the return type, are displayed following standard Python type hint syntax. Sometimes for compound types we use type variables. Types that are defined by fastai or Pytorch link directly to more information about that type; try clicking *Image* in the function above for an example. The docstring for the symbol is shown immediately after the signature, along with a link to the source code for the symbol in GitHub. After the basic signature and docstring you'll find examples and additional details (not shown in this example). As you'll see at the top of the page, all symbols documented like this also appear in the table of contents.

For inherited classes and some types of decorated function, the base class or decorator type will also be shown at the end of the signature, delimited by `::`. For `vision.transforms`, the random number generator used for data augmentation is shown instead of the type, for randomly generated parameters.

Module structure

Imports

fastai is designed to support both interactive computing as well as traditional software development. For interactive computing, where convenience and speed of experimentation is a priority, data scientists often prefer to grab all the symbols they need, with `import *`. Therefore, fastai is designed to support this approach, without compromising on maintainability and understanding.

In order to do so, the module dependencies are carefully managed (see next section), with each exporting a carefully chosen set of symbols when using `import *`. In general, for interactive computing, you'll want to import from both `fastai`, and from one of the *applications*, such as:

```
from fastai import *
from fastai.vision import *
```

That will give you all the standard external modules you'll need, in their customary namespaces (e.g. `pandas as pd`, `numpy as np`, `matplotlib.pyplot as plt`), plus the core fastai libraries. In addition, the main classes and functions for your application (`fastai.vision`, in this case), e.g. creating a `DataBunch` from an image folder and training a convolutional neural network (with `ConvLearner`), are also imported.

If you wish to see where a symbol is imported from, either just type the symbol name (in a REPL such as Jupyter Notebook or IPython), or (in most editors) wave your mouse over the symbol to see the definition. For instance:

```
ConvLearner
```

```
fastai.vision.learner.ConvLearner
```

Dependencies

At the base of everything are the two modules `core` and `torch_core` (we're not including the `fastai.` prefix when naming modules in these docs). They define the basic functions we use in the library; `core` only relies on general modules, whereas `torch_core` requires pytorch. Most type-hinting shortcuts are defined there too (at least the one that don't depend on fastai classes defined later). Nearly all modules below import `torch_core`.

Then, there are three modules directly on top of `torch_core`: - `data`, which contains the class that will take a `Dataset` or pytorch `DataLoader` to wrap it in a `DeviceDataLoader` (a class that sits on top of a `DataLoader` and is in charge of putting the data on the right device as well as applying transforms such as normalization) and regroup then in a `DataBunch`. - `layers`, which contains

basic functions to define custom layers or groups of layers - `metrics`, which contains all the metrics

From `layers`, we have all the modules in the models folder that are defined. Then from `data` we can split on one of the four main *applications*, which each has their own module: `vision`, `text` `collab`, or `tabular`. Each of those submodules is built in the same way with: - a submodule named `transform` that handles the transformations of our data (data augmentation for computer vision, numericalizing and tokenizing for text and preprocessing for tabular) - a submodule named `data` that contains the class that will create datasets and the helper functions to create `DataBunch` objects.

This takes care of building your model and handling the data. We regroup those in a `Learner` object to take care of training. More specifically: - `callback` (depends on `data`) defines the basis of callbacks and the `CallbackHandler`. Those are functions that will be called every step of the way of the training loop and can allow us to customize what is happening there; - `basic_train` (depends on `callback`) defines `Learner` and `Recorder` (which is a callback that records training stats) and has the training loop; - `callbacks` (depends on `basic_train`) is a submodule defining various callbacks, such as for mixed precision training or 1cycle annealing; - `learn` (depends on `callbacks`) defines helper functions to invoke the callbacks more easily.

The module `tta` (for Test Time Augmentation) depends on `basic_train`, the module `collab` (for collaborative filtering) depends on `basic_train` and `layers`, so does the module `vision.train` (for our models with a skeleton trained on imagenet and a custom head for classification) and the module `text.train` (to automatically get learner objects for NLP) depends on `callbacks` (specifically the `rnn` callback) and `models` (specifically the rnn models).

Here is a graph of the key module dependencies:

Training modules overview

```
from fastai.basic_train import *
from fastai.gen_doc.nbdoc import *
from fastai import *
```

The fastai library is structured training around a `Learner` object that binds together a pytorch model, some data with an optimizer and a loss function, which then will allow us to launch training.

`basic_train` contains the definition of this `Learner` class along with the wrapper around pytorch optimizer that the library uses. It defines the basic training loop that is used each time you call the `fit` function in fastai (or one of its

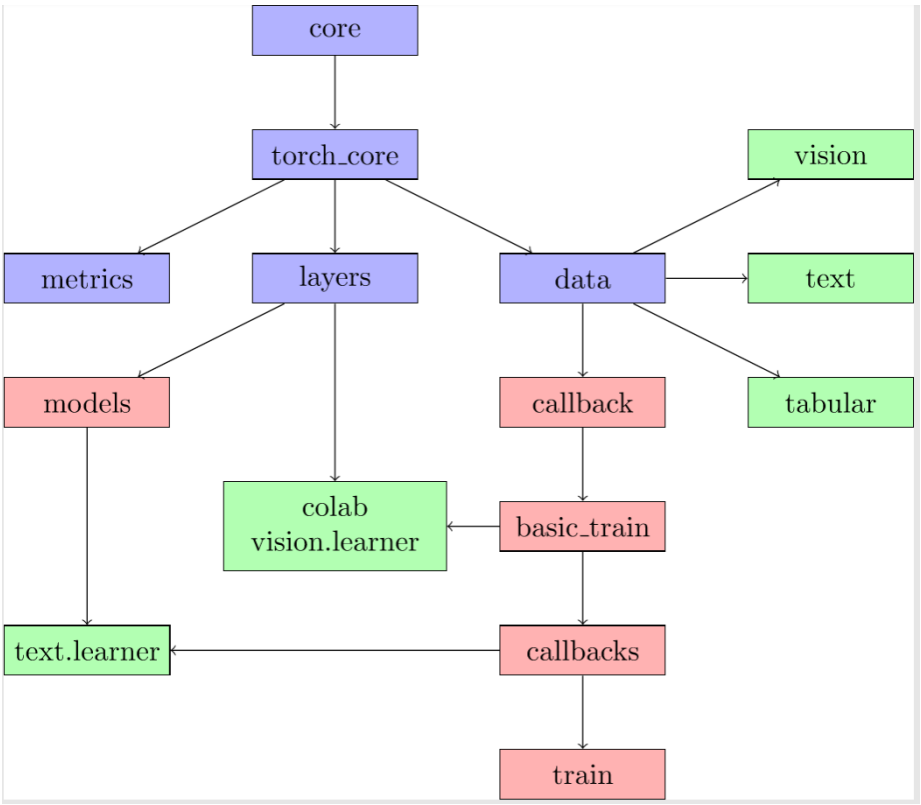

Figure 1: Modules overview

variants). This training loop is kept to the minimum number of instructions, and most of its customization happens in `Callback` objects.

`callback` contains the definition of those, as well as the `CallbackHandler` that is responsible for the communication between the training loop and the `Callback` functions. It maintains a state dictionary to be able to provide to each `Callback` all the informations of the training loop, easily allowing any tweaks you could think of.

In `callbacks`, each `Callback` is then implemented in separate modules. Some deal with scheduling the hyperparameters, like `callbacks.one_cycle`, `callbacks.lr_finder` or `callback.general_sched`. Others allow special kind of trainings like `callbacks.fp16` (mixed precision) or `callbacks.rnn`. The `Recorder` or `callbacks.hooks` are useful to save some internal data.

`train` then implements those callbacks with useful helper functions. Lastly `metrics` contains all the functions you might want to call to evaluate your results.

Walk-through of key functionality

We'll do a quick overview of the key pieces of fastai's training modules. See the separate module docs for details on each. We'll use the classic MNIST dataset for the training documentation, cut down to just 3's and 7's. To minimize the boilerplate in our docs we've defined the basic imports and paths we need in `fastai.docs`. It also has a `get_mnist` function to grab a `DataBunch` of the data for us, which will automatically download and unzip the data if not already done.

```
from fastai.docs import *
data = get_mnist()
```

Basic training with `Learner`

We can create minimal simple CNNs using `simple_cnn` (see `models` for details on creating models):

```
model = simple_cnn((3,16,16,2))
```

The most important object for training models is `Learner`, which needs to know, at minimum, what data to train with and what model to train.

```
learn = Learner(data, model)
```

That's enough to train a model, which is done using `fit`. If you have a CUDA-capable GPU it will be used automatically. You have to say how many epochs to train for.

```
learn.fit(1)
```

```
Total time: 00:02
epoch  train loss  valid loss
0      0.130048    0.102490    (00:02)
```

Viewing metrics

To see how our training is going, we can request that it reports various `metrics` after each epoch. You can pass it to the constructor, or set it later. Note that metrics are always calculated on the validation set.

```
learn.metrics=[accuracy]
learn.fit(1)
```

```
Total time: 00:02
epoch  train loss  valid loss  accuracy
0      0.094618    0.073190    0.973013  (00:02)
```

Extending training with callbacks

You can use `callback`s to modify training in almost any way you can imagine. For instance, we've provided a callback to implement Leslie Smith's 1cycle training method.

```
cb = OneCycleScheduler(learn, lr_max=0.01)
learn.fit(1, callbacks=cb)
```

```
Total time: 00:02
epoch  train loss  valid loss  accuracy
0      0.068852    0.048199    0.981354  (00:02)
```

The `Recorder` callback is automatically added for you, and you can use it to see what happened in your training, e.g.:

```
learn.recorder.plot_lr(show_moms=True)
```

Extending `Learner` with `train`

Many of the callbacks can be used more easily by taking advantage of the `Learner` extensions in `train`. For instance, instead of creating OneCycleScheduler manually as above, you can simply call `Learner.fit_one_cycle`:

```
learn.fit_one_cycle(1)
```

```
Total time: 00:02
epoch  train loss  valid loss  accuracy
0      0.047730    0.037270    0.986752  (00:02)
```

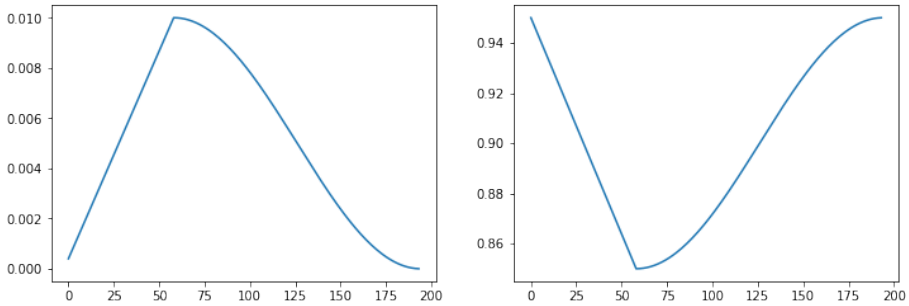

Figure 2: png

Applications

Note that if you're training a model for one of our supported *applications*, there's a lot of help available to you in the application modules:

- `vision`
- `text`
- `tabular`
- `collab`

For instance, let's use `ConvLearner` (from `vision`) to quickly fine-tune a pre-trained Imagenet model for MNIST (not a very practical approach, of course, since MNIST is handwriting and our model is pre-trained on photos!) Note that `tvm` is the namespace we use for `torchvision.models`.

```
learn = ConvLearner(data, tvm.resnet18, metrics=accuracy)
learn.fit_one_cycle(1)
```

```
Total time: 00:05
epoch  train loss  valid loss  accuracy
0      0.114983    0.074512    0.973503  (00:05)
```

Basic training functionality

```
from fastai.basic_train import *
from fastai.gen_doc.nbdoc import *
from fastai import *
from fastai.vision import *
from fastai.docs import *
```

`basic_train` wraps together the data (in a `DataBunch` object) with a pytorch model to define a `Learner` object. This is where the basic training loop is defined for the `fit` function. The `Learner` object is the entry point of most of

the `Callback` functions that will customize this training loop in different ways (and made available through the `train` module), notably:

- `Learner.lr_find` will launch an LR range test that will help you select a good learning rate
- `Learner.fit_one_cycle` will launch a training using the 1cycle policy, to help you train your model fast.
- `Learner.to_fp16` will convert your model in half precision and halp you launch a training in mixed precision.

```
show_doc(Learner, title_level=2)
```

class Learner

> `Learner(data:DataBunch, model:Module, opt_fn:Callable='Adam', loss_fn:Callable='cross_entropy', metrics:Collection[Callable]=None, true_wd:bool=True, bn_wd:bool=True, wd:Floats=0.01, train_bn:bool=True, path:str=None, model_dir:str='models', callback_fns:Collection[Callable]=None, callbacks:Collection[Callback]=<factory>, layer_groups:ModuleList=None)`

Train `model` using `data` to minimize `loss_fn` with optimizer `opt_fn`. [source]

The main purpose of `Learner` is to train `model` using `Learner.fit`. After every epoch, all *metrics* will be printed, and will also be available to callbacks.

The default weight decay will be `wd`, which will be handled using the method from Fixing Weight Decay Regularization in Adam if `true_wd` is set (otherwise it's L2 regularization). If `bn_wd` is False then weight decay will be removed from batchnorm layers, as recommended in Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour. You can also turn off batchnorm layer training entirely (i.e. freeze all batchnorm learnable parameters) by disabling `train_bn`.

To use discriminative layer training pass an `nn.Module` for each layer group to be optimized with different settings.

Any model files created will be saved in `path/model_dir`.

You can pass a list of `callbacks` that you have already created, or (more commonly) simply pass a list of callback functions to `callback_fns` and each function will be called (passing `self`) on object initialization, with the results stored as callback objects. For a walk-through, see the training overview page. You may also want to use an `application` to fit your model, e.g. using the `ConvLearner` subclass:

```
data = get_mnist()
learn = ConvLearner(data, tvm.resnet18, metrics=accuracy)
learn.fit(1)
```

```
Total time: 00:05
epoch  train loss  valid loss  accuracy
0      0.086652    0.038705    0.986261  (00:05)
```

Model fitting methods

```
show_doc(Learner.fit)
```

`fit`

> `fit`(`epochs`:`int`, `lr`:`Union`[`float`, `Collection`[`float`], `slice`]=`slice(None, 0.003, None)`, `wd`:`Floats`=`None`, `callbacks`:`Collection`[`Callback`]=`None`)

Fit the model on this learner with `lr` learning rate, `wd` weight decay for `epochs` with `callbacks`. [source]

Uses discriminative layer training if multiple learning rates or weight decay values are passed. To control training behaviour, use the `callback` system or one or more of the pre-defined `callbacks`.

```
show_doc(Learner.fit_one_cycle)
```

`fit_one_cycle`

> `fit_one_cycle`(`learn`:`Learner`, `cyc_len`:`int`, `max_lr`:`Union`[`float`, `Collection`[`float`], `slice`]=`slice(None, 0.003, None)`, `moms`:`Point`=`(0.95, 0.85)`, `div_factor`:`float`=`25.0`, `pct_start`:`float`=`0.3`, `wd`:`float`=`None`, `kwargs`)

Fit a model following the 1cycle policy. [source]

Uses the `OneCycleScheduler` callback.

```
show_doc(Learner.lr_find)
```

`lr_find`

> `lr_find`(`learn`:`Learner`, `start_lr`:`float`=`1e-05`, `end_lr`:`float`=`10`, `num_it`:`int`=`100`, `kwargs`:`Any`)

Explore lr from `start_lr` to `end_lr` over `num_it` iterations in `learn`. [source]

Runs the learning rate finder defined in `LRFinder`, as discussed in Cyclical Learning Rates for Training Neural Networks.

Mixed precision training

```
show_doc(Learner.to_fp16)
```

`to_fp16`

> `to_fp16`(`learn`:`Learner`, `loss_scale`:`float`=`512.0`, `flat_master`:`bool`=`False`) -> `Learner`

Transform `learn` in FP16 precision. [source]

Uses the `MixedPrecision` callback to train in mixed precision (i.e. forward and backward passes using fp16, with weight updates using fp32), using all NVIDIA recommendations for ensuring speed and accuracy.

Discriminative layer training

When fitting a model you can pass a list of learning rates (and/or weight decay amounts), which will apply a different rate to each *layer group* (i.e. the parameters of each module in `self.layer_groups`). See the Universal Language Model Fine-tuning for Text Classification paper for details and experimental results in NLP (we also frequently use them successfully in computer vision, but have not published a paper on this topic yet). When working with a `Learner` on which you've called `split`, you can set hyperparameters in four ways:

1. `param = [val1, val2 ..., valn]` (n = number of layer groups)
2. `param = val`
3. `param = slice(start,end)`
4. `param = slice(end)`

If we chose to set it in way 1, we must specify a number of values exactly equal to the number of layer groups. If we chose to set it in way 2, the chosen value will be repeated for all layer groups. See `Learner.lr_range` for an explanation of the `slice` syntax).

Here's an example of how to use discriminative learning rates (note that you don't actually need to manually call `Learner.split` in this case, since fastai uses this exact function as the default split for `resnet18`; this is just to show how to customize it):

```
# creates 3 layer groups
learn.split(lambda m: (m[0][6], m[1]))
# only randomly initialized head now trainable
learn.freeze()
```

```
learn.fit_one_cycle(1)
```

```
Total time: 00:04
epoch  train loss  valid loss  accuracy
0      0.040739    0.029042    0.992640  (00:04)
```

```
# all layers now trainable
learn.unfreeze()
# optionally, separate LR and WD for each group
```

```
learn.fit_one_cycle(1, max_lr=(1e-4, 1e-3, 1e-2), wd=(1e-4,1e-4,1e-1))
```

```
Total time: 00:05
epoch  train loss  valid loss  accuracy
0      0.023351    0.017301    0.995093  (00:05)
```

```
show_doc(Learner.lr_range)
```

lr_range

> `lr_range`(`lr`:`Union`[`float`, `slice`]) -> `ndarray`

Build differential learning rates. [source]

Rather than manually setting an LR for every group, it's often easier to use `Learner.lr_range`. This is a convenience method that returns one learning rate for each layer group. If you pass `slice(start,end)` then the first group's learning rate is `start`, the last is `end`, and the remaining are evenly geometrically spaced.

If you pass just `slice(end)` then the last group's learning rate is `end`, and all the other groups are `end/3`. For instance (for our learner that has 3 layer groups):

```
learn.lr_range(slice(1e-5,1e-3)), learn.lr_range(slice(3e-4))
```

```
(array([1.e-05, 1.e-04, 1.e-03]), array([1.e-04, 1.e-04, 3.e-04]))
```

```
show_doc(Learner.unfreeze)
```

unfreeze

> `unfreeze`()

Unfreeze entire model. [source]

Sets every layer group to *trainable* (i.e. `requires_grad=True`).

```
show_doc(Learner.freeze)
```

freeze

> `freeze`()

Freeze up to last layer. [source]

Sets every layer group except the last to *untrainable* (i.e. `requires_grad=False`).

```
show_doc(Learner.freeze_to)
```

freeze_to

> freeze_to(n:int)

Freeze layers up to layer `n`. [source]

```
show_doc(Learner.split)
```

split

> split(split_on:SplitFuncOrIdxList)

Split the model at `split_on`. [source]

A convenience method that sets `layer_groups` based on the result of `split_model`. If `split_on` is a function, it calls that function and passes the result to `split_model` (see above for example).

Saving and loading models

Simply call `Learner.save` and `Learner.load` to save and load models. Only the parameters are saved, not the actual architecture (so you'll need to create your model in the same way before loading weights back in). Models are saved to the `path/model_dir` directory.

```
show_doc(Learner.load)
```

load

> load(name:PathOrStr)

Load model `name` from `self.model_dir`. [source]

```
show_doc(Learner.save)
```

save

> save(name:PathOrStr)

Save model with `name` to `self.model_dir`. [source]

Other methods

```
show_doc(Learner.init)
```

`init`

> `init(init)` [source]

Initializes all weights (except batchnorm) using function `init`, which will often be from PyTorch's `nn.init` module.

```
show_doc(Learner.mixup)
```

`mixup`

> `mixup(learn:Learner, alpha:float=0.4, stack_x:bool=False, stack_y:bool=True) -> Learner`

Add mixup https://arxiv.org/abs/1710.09412 to `learn`. [source]

Uses `MixUpCallback`.

```
show_doc(Learner.pred_batch)
```

`pred_batch`

> `pred_batch(learn:Learner, is_valid:bool=True) -> Tuple[Tensors, Tensors, Tensors]`

Returns input, target and output of the model on a batch [source]

Get the first batch of predictions. Mainly useful for debugging and quick tests.

```
show_doc(Learner.create_opt)
```

`create_opt`

> `create_opt(lr:Floats, wd:Floats=0.0)`

Create optimizer with `lr` learning rate and `wd` weight decay. [source]

You generally won't need to call this yourself - it's used to create the `nn.optim` optimizer before fitting the model.

```
show_doc(Recorder, title_level=2)
```

class Recorder

> `Recorder(learn:Learner) :: LearnerCallback`

A `LearnerCallback` that records epoch, loss, opt and metric data during training. [source]

A `Learner` creates a `Recorder` object automatically - you do not need to explicitly pass to `callback_fns` - because other callbacks rely on it being available. It

stores the smoothed loss, hyperparameter values, and metrics each batch, and provides plotting methods for each. Note that `Learner` automatically sets an attribute with the snake-cased name of each callback, so you can access this through `Learner.recorder`, as shown below.

Plotting methods

```
show_doc(Recorder.plot)
```

`plot`

> `plot(skip_start:int=10, skip_end:int=5)`

Plot learning rate and losses, trimmed between `skip_start` and `skip_end`. [source]

This is mainly used with the learning rate finder, since it shows a scatterplot of loss vs learning rate.

```
learn = ConvLearner(data, tvm.resnet18, metrics=accuracy)
learn.lr_find()
learn.recorder.plot()
```

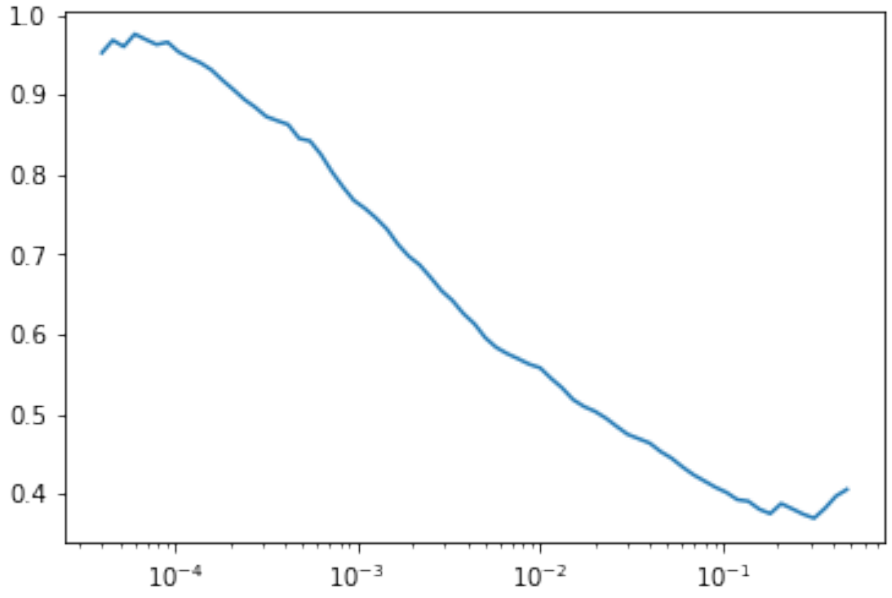

Figure 3: png

```
show_doc(Recorder.plot_losses)
```

plot_losses

```
plot_losses()
```

Plot training and validation losses. [source]

Note that validation losses are only calculated once per epoch, whereas training losses are calculated after every batch.

```
learn.fit_one_cycle(2)
learn.recorder.plot_losses()
```

```
Total time: 00:09
epoch  train loss  valid loss  accuracy
0      0.113814    0.065617    0.980373  (00:04)
1      0.046937    0.040896    0.988224  (00:04)
```

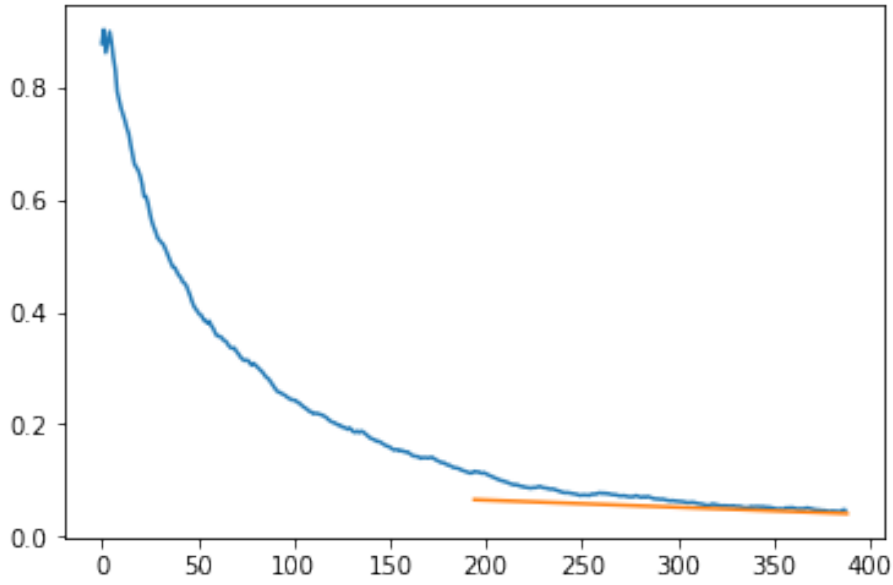

Figure 4: png

```
show_doc(Recorder.plot_lr)
```

plot_lr

```
plot_lr(show_moms=False)
```

Plot learning rate, `show_moms` to include momentum. [source]

```
learn.recorder.plot_lr(show_moms=True)
```

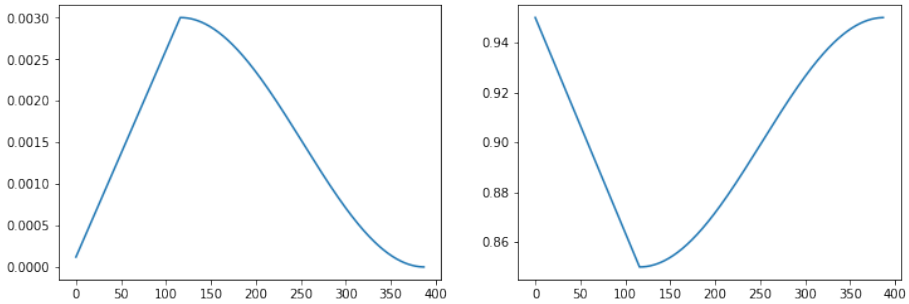

Figure 5: png

```
show_doc(Recorder.plot_metrics)
```

plot_metrics

> plot_metrics()

Plot metrics collected during training. [source]

Note that metrics are only collected at the end of each epoch, so you'll need to train at least two epochs to have anything to show here.

```
learn.recorder.plot_metrics()
```

Callback methods

You don't call these yourself - they're called by fastai's `callback` system automatically to enable the class's functionality.

```
show_doc(Recorder.on_backward_begin)
```

on_backward_begin

> on_backward_begin(smooth_loss:Tensor, kwargs:Any)

Record the loss before any other callback has a chance to modify it. [source]

```
show_doc(Recorder.on_batch_begin)
```

on_batch_begin

> on_batch_begin(kwargs:Any)

Record learning rate and momentum at beginning of batch. [source]

```
show_doc(Recorder.on_epoch_end)
```

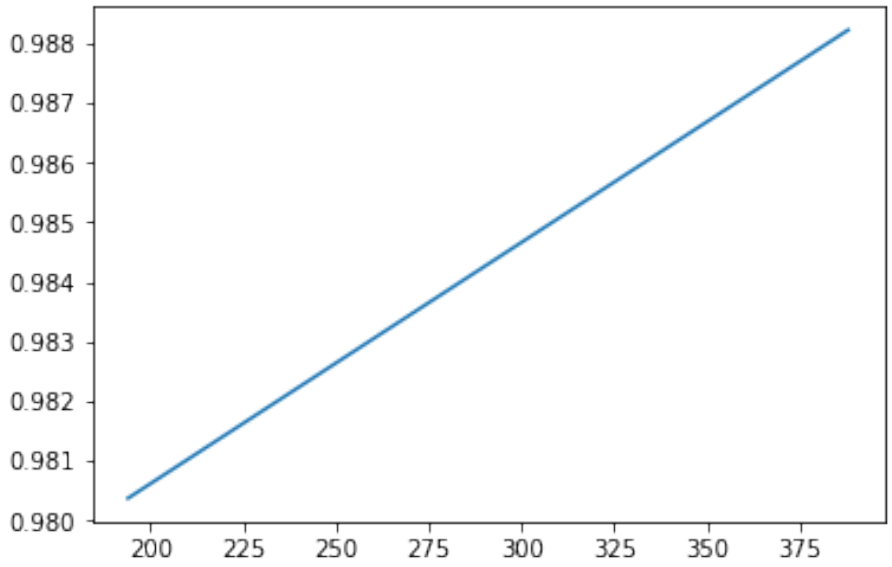

Figure 6: png

`on_epoch_end`

> `on_epoch_end(epoch:int, num_batch:int, smooth_loss:Tensor, last_metrics='Collection', kwargs:Any) -> bool`

Save epoch info: num_batch, smooth_loss, metrics. [source]

`show_doc(Recorder.on_train_begin)`

`on_train_begin`

> `on_train_begin(pbar:PBar, metrics:MetricFuncList, kwargs:Any)`

Initialize recording status at beginning of training. [source]

Module functions

Generally you'll want to use a `Learner` to train your model, since they provide a lot of functionality and make things easier. However, for ultimate flexibility, you can call the same underlying functions that `Learner` calls behind the scenes.

`show_doc(fit)`

fit

> fit(epochs:int, model:Module, loss_fn:LossFunction, opt:Optimizer, data:DataBunch, callbacks:Optional[Collection[Callback]]=None, metrics:OptMetrics=None)

Fit the `model` on `data` and learn using `loss` and `opt`. [source]

Note that you have to create the `Optimizer` yourself if you call this function, whereas `Learn.fit` creates it for you automatically.

```
show_doc(train_epoch)
```

train_epoch

> train_epoch(model:Module, dl:DataLoader, opt:Optimizer, loss_func:LossFunction)

Simple training of `model` for 1 epoch of `dl` using optim `opt` and loss function `loss_func`. [source]

You won't generally need to call this yourself - it's what `fit` calls for each epoch.

```
show_doc(validate)
```

validate

> validate(model:Module, dl:DataLoader, loss_fn:OptLossFunc=None, metrics:OptMetrics=None, cb_handler:Optional[CallbackHandler]=None, pbar:Union[MasterBar, ProgressBar, NoneType]=None) -> Iterator[Tuple[IntOrTensor, Ellipsis]]

Calculate loss and metrics for the validation set. [source]

This is what `fit` calls after each epoch. You can call it if you want to run inference on a `DataLoader` manually.

```
show_doc(loss_batch)
```

loss_batch

> loss_batch(model:Module, xb:Tensor, yb:Tensor, loss_fn:OptLossFunc=None, opt:OptOptimizer=None, cb_handler:Optional[CallbackHandler]=None, metrics:OptMetrics=None) -> Tuple[Union[Tensor, int, float, str]]

Calculate loss and metrics for a batch, call out to callbacks as necessary. [source]

You won't generally need to call this yourself - it's what `fit` and `validate` call for each batch. It only does a backward pass if you set `opt`.

Other classes

```
show_doc(LearnerCallback, title_level=3)
```

class LearnerCallback

> LearnerCallback(learn:Learner) :: Callback

Base class for creating callbacks for a `Learner`. [source]

Undocumented Methods - Methods moved below this line will intentionally be hidden

```
show_doc(Learner.tta_only)
```

_tta_only

> _tta_only(learn:Learner, is_test:bool=False, scale:float=1.35) -> Iterator[List[Tensor]]

Computes the outputs for several augmented inputs for TTA [source]

```
show_doc(Learner.get_preds)
```

_learn_get_preds

> _learn_get_preds(learn:Learner, is_test:bool=False) -> List[Tensor]

Wrapper of get_preds for learner [source]

```
show_doc(Learner.TTA)
```

_TTA

> _TTA(learn:Learner, beta:float=0.4, scale:float=1.35, is_test:bool=False) -> Tensors [source]

```
show_doc(Recorder.format_stats)
```

format_stats

> format_stats(stats:MetricsList)

Format stats before printing. [source]

```
show_doc(Learner.loss_fn)
```

cross_entropy

cross_entropy(input, target, weight=None, size_average=None, ignore_index=-100, reduce=None, reduction='elementwise_mean')

function.

See :class:~torch.nn.CrossEntropyLoss for details.

Args: input (Tensor) : :math:(N, C) where C = number of classes or :math:(N, C, H, W) in case of 2D Loss, or :math:(N, C, d_1, d_2, ..., d_K) where :math:K > 1 in the case of K-dimensional loss. target (Tensor) : :math:(N) where each value is :math:0 \leq \text{targets}[i] \leq C-1, or :math:(N, d_1, d_2, ..., d_K) where :math:K \geq 1 for K-dimensional loss. weight (Tensor, optional): a manual rescaling weight given to each class. If given, has to be a Tensor of size C size_average (bool, optional): Deprecated (see :attr:reduction). By default, the losses are averaged over each loss element in the batch. Note that for some losses, there multiple elements per sample. If the field :attr:size_average is set to False, the losses are instead summed for each minibatch. Ignored when reduce is False. Default: True ignore_index (int, optional): Specifies a target value that is ignored and does not contribute to the input gradient. When :attr:size_average is True, the loss is averaged over non-ignored targets. Default: -100 reduce (bool, optional): Deprecated (see :attr:reduction). By default, the losses are averaged or summed over observations for each minibatch depending on :attr:size_average. When :attr:reduce is False, returns a loss per batch element instead and ignores :attr:size_average. Default: True reduction (string, optional): Specifies the reduction to apply to the output: 'none' | 'elementwise_mean' | 'sum'. 'none': no reduction will be applied, 'elementwise_mean': the sum of the output will be divided by the number of elements in the output, 'sum': the output will be summed. Note: :attr:size_average and :attr:reduce are in the process of being deprecated, and in the meantime, specifying either of those two args will override :attr:reduction. Default: 'elementwise_mean'

Examples::

```
>>> input = torch.randn(3, 5, requires_grad=True)
>>> target = torch.randint(5, (3,), dtype=torch.int64)
>>> loss = F.cross_entropy(input, target)
>>> loss.backward()
```

Additional training functions

train provides a number of extension methods that are added to Learner (see below for a list and details), along with three simple callbacks:

- ShowGraph

- GradientClipping
- BnFreeze

```
from fastai.gen_doc.nbdoc import *
from fastai.train import *
from fastai.vision import *
from fastai.docs import *
from fastai import *
```

Learner extension methods

These methods are automatically added to all Learner objects created after importing this module. They provide convenient access to a number of callbacks, without requiring them to be manually created.

```
show_doc(fit_one_cycle)
```

fit_one_cycle

> fit_one_cycle(learn:Learner, cyc_len:int, max_lr:Union[float, Collection[float], slice]=slice(None, 0.003, None), moms:Point=(0.95, 0.85), div_factor:float=25.0, pct_start:float=0.3, wd:float=None, kwargs)

Fit a model following the 1cycle policy. [source]

Fit a model with 1cycle training. See OneCycleScheduler for details.

```
show_doc(lr_find)
```

lr_find

> lr_find(learn:Learner, start_lr:float=1e-05, end_lr:float=10, num_it:int=100, kwargs:Any)

Explore lr from start_lr to end_lr over num_it iterations in learn. [source]

See LRFinder for details.

```
show_doc(to_fp16)
```

to_fp16

> to_fp16(learn:Learner, loss_scale:float=512.0, flat_master:bool=False) -> Learner

Transform learn in FP16 precision. [source]

See MixedPrecision for details.

```
show_doc(mixup)
```

mixup

> mixup(learn:Learner, alpha:float=0.4, stack_x:bool=False, stack_y:bool=True) -> Learner

Add mixup https://arxiv.org/abs/1710.09412 to `learn`. [source]

See `MixUpCallback` for more details.

A last extension method comes from the module tta.

```
show_doc(Learner.TTA, full_name='TTA')
```

TTA

> TTA(learn:Learner, beta:float=0.4, scale:float=1.35, is_test:bool=False) -> Tensors [source]

Applies Test Time Augmentation to `learn` on the validation set or the test test (depending on `is_test`). We take the average of our regular predictions (with a weight `beta`) with the average of predictions obtained thourh augmented versions of the training set (with a weight `1-beta`). The transforms decided for the training set are applied with a few changes `scale` controls the scale for zoom (which isn't random), the cropping isn't random but we make sure to get the four corners of the image. Flipping isn't random but applied once on each of those corner images (so that makes 8 augmented versions total).

We'll show examples below using our MNIST sample.

```
data = get_mnist()
```

```
show_doc(ShowGraph)
```

class ShowGraph

> ShowGraph(learn:Learner) :: LearnerCallback

Update a graph of learner stats and metrics after each epoch. [source]

```
learn = ConvLearner(data, tvm.resnet18, metrics=accuracy, callback_fns=ShowGraph)
learn.fit(3)
```

```
show_doc(ShowGraph.on_epoch_end, doc_string=False)
```

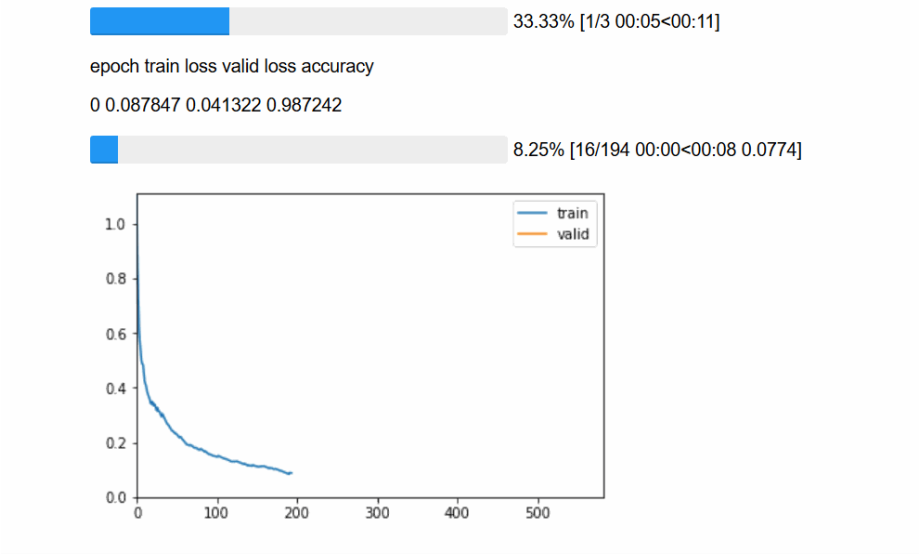

Figure 7: Training graph

`on_epoch_end`

> `on_epoch_end(n_epochs:int, last_metrics:MetricsList, kwargs) -> bool` [source]

If we have `last_metrics`, plot them in `self.pbar`. Set the size of the graph with `n_epochs`.

`show_doc(GradientClipping)`

`class GradientClipping`

> `GradientClipping(learn:Learner, clip:float) :: LearnerCallback`

To do gradient clipping during training. [source]

Clips gradient at a maximum absolute value of `clip` during training. For instance:

```
learn = ConvLearner(data, tvm.resnet18, metrics=accuracy,
    callback_fns=partial(GradientClipping, clip=0.1))
learn.fit(1)
```

```
Total time: 00:05
epoch  train loss  valid loss  accuracy
0      0.090287    0.034165    0.989696  (00:05)
```

```
show_doc(GradientClipping.on_backward_end, doc_string=False)
```

on_backward_end

on_backward_end(kwargs) [source]

Clip the gradients after they are computed but before the optimizer step.

```
show_doc(BnFreeze)
```

class BnFreeze

BnFreeze(learn:Learner) :: LearnerCallback

Freeze moving average statistics in all non-trainable batchnorm layers. [source]

For batchnorm layers where `requires_grad==False`, you generally don't want to update their moving average statistics, in order to avoid the model's statistics getting out of sync with its pre-trained weights. You can add this callback to automate this freezing of statistics (internally, it calls `eval` on these layers).

```
learn = ConvLearner(data, tvm.resnet18, metrics=accuracy, callback_fns=BnFreeze)
learn.fit(1)
```

```
Total time: 00:04
epoch  train loss  valid loss  accuracy
0      0.074059    0.038646    0.986261  (00:04)
```

```
show_doc(BnFreeze.on_epoch_begin, doc_string=False)
```

on_epoch_begin

on_epoch_begin(kwargs:Any) [source]

Set back the batchnorm layers on `eval` mode after the model has been set to `train`.

Undocumented Methods - Methods moved below this line will intentionally be hidden

```
show_doc(one_cycle_scheduler)
```

one_cycle_scheduler

one_cycle_scheduler(lr_max:float, kwargs:Any) -> OneCycleScheduler [source]

Training metrics

Metrics for training fastai models are simply functions that take `input` and `target` tensors, and return some metric of interest for training. You can write your own metrics by defining a function of that type, and passing it to `Learner` in the [code]metrics[/code] parameter, or use one of the following pre-defined functions.

```
from fastai.gen_doc.nbdoc import *
from fastai.metrics import *
```

Predefined metrics:

```
show_doc(accuracy)
```

`accuracy`

> `accuracy`(`input`:`Tensor`, `targs`:`Tensor`) -> `Rank0Tensor`

Compute accuracy with `targs` when `input` is bs * n_classes. [source]

```
show_doc(accuracy_thresh, doc_string=False)
```

`accuracy_thresh`

> `accuracy_thresh`(`y_pred`:`Tensor`, `y_true`:`Tensor`, `thresh`:`float`=`0.5`, `sigmoid`:`bool`=`True`) -> `Rank0Tensor` [source]

Compute accuracy when `y_pred` and `y_true` for multi-label models, based on comparing predictions to `thresh`, `sigmoid` will be applied to `y_pred` if the corresponding flag is True.

```
show_doc(dice)
```

`dice`

> `dice`(`input`:`Tensor`, `targs`:`Tensor`) -> `Rank0Tensor`

Dice coefficient metric for binary target. [source]

```
show_doc(fbeta)
```

`fbeta`

> `fbeta`(`y_pred`:`Tensor`, `y_true`:`Tensor`, `thresh`:`float`=`0.5`, `beta`:`float`=`2`, `eps`:`float`=`1e-09`, `sigmoid`:`bool`=`True`) -> `Rank0Tensor`

Compute the f_beta between preds and targets. [source]

See the F1 score wikipedia page for details.

```
show_doc(exp_rmspe)
```

exp_rmspe

> exp_rmspe(pred:Tensor, targ:Tensor) -> Rank0Tensor

Exp RMSE between `pred` and `targ`. [source]

Classes for callback implementors

```
from fastai.gen_doc.nbdoc import *
from fastai.callback import *
from fastai import *
```

fastai provides a powerful *callback* system, which is documented on the `callbacks` page; look on that page if you're just looking for how to use existing callbacks. If you want to create your own, you'll need to use the classes discussed below.

A key motivation for the callback system is that additional functionality can be entirely implemented in a single callback, so that it's easily read. By using this trick, we will have different methods categorized in different callbacks where we will find clearly stated all the interventions the method makes in training. For instance in the `LRFinder` callback, on top of running the fit function with exponentially growing LRs, it needs to handle some preparation and clean-up, and all this code can be in the same callback so we know exactly what it is doing and where to look if we need to change something.

In addition, it allows our `fit` function to be very clean and simple, yet still easily extended. So far in implementing a number of recent papers, we haven't yet come across any situation where we had to modify our training loop source code - we've been able to use callbacks every time.

```
show_doc(Callback)
```

class Callback

> Callback()

Base class for callbacks that want to record values, dynamically change learner params, etc. [source]

To create a new type of callback, you'll need to inherit from this class, and implement one or more methods as required for your purposes. Perhaps the

easiest way to get started is to look at the source code for some of the predefined fastai callbacks. You might be surprised at how simple they are! For instance, here is the **entire** source code for `GradientClipping`:

```
@dataclass
class GradientClipping(LearnerCallback):
    clip:float
    def on_backward_end(self, **kwargs):
        if self.clip:
            nn.utils.clip_grad_norm_(self.learn.model.parameters(), self.clip)
```

You generally want your custom callback constructor to take a `Learner` parameter, e.g.:

```
@dataclass
class MyCallback(Callback):
    learn:Learner
```

Note that this allows the callback user to just pass your callback name to `callback_fns` when constructing their `Learner`, since that always passes `self` when constructing callbacks from `callback_fns`. In addition, by passing the learner, this callback will have access to everything: e.g all the inputs/outputs as they are calculated, the losses, and also the data loaders, the optimizer, etc. At any time: - Changing self.learn.data.train_dl or self.data.valid_dl will change them inside the fit function (we just need to pass the `DataBunch` object to the fit function and not data.train_dl/data.valid_dl) - Changing self.learn.opt.opt (We have an `OptimWrapper` on top of the actual optimizer) will change it inside the fit function. - Changing self.learn.data or self.learn.opt directly WILL NOT change the data or the optimizer inside the fit function.

In any of the callbacks you can unpack in the kwargs: - `n_epochs`, contains the number of epochs the training will take in total - `epoch`, contains the number of the current - `iteration`, contains the number of iterations done since the beginning of training - `num_batch`, contains the number of the batch we're at in the dataloader - `last_input`, contains the last input that got through the model (eventually updated by a callback) - `last_target`, contains the last target that gor through the model (eventually updated by a callback) - `last_output`, contains the last output spitted by the model (eventually updated by a callback) - `last_loss`, contains the last loss computed (eventually updated by a callback) - `smooth_loss`, contains the smoothed version of the loss - `last_metrics`, contains the last validation loss and emtrics computed - `pbar`, the progress bar

Methods your subclass can implement

All of these methods are optional; your subclass can handle as many or as few as you require.

```
show_doc(Callback.on_train_begin)
```

on_train_begin

on_train_begin(kwargs:Any)

To initialize constants in the callback. [source]

Here we can initiliaze anything we need. The optimizer has now been initialized. We can change any hyper-parameters by typing, for instance:

```
self.opt.lr = new_lr
self.opt.mom = new_mom
self.opt.wd = new_wd
self.opt.beta = new_beta
```

```
show_doc(Callback.on_epoch_begin)
```

on_epoch_begin

on_epoch_begin(kwargs:Any)

At the beginning of each epoch. [source]

This is not technically required since we have `on_train_begin` for epoch 0 and `on_epoch_end` for all the other epochs, yet it makes writing code that needs to be done at the beginning of every epoch easy and more readable.

```
show_doc(Callback.on_batch_begin)
```

on_batch_begin

on_batch_begin(kwargs:Any)

Set HP before the step is done. Returns xb, yb (which can allow us to modify the input at that step if needed). [source]

Here is the perfect place to prepare everything before the model is called. Example: change the values of the hyperparameters (if we don't do it on_batch_end instead)

If we return something, that will be the new value for `xb`,`yb`.

```
show_doc(Callback.on_loss_begin)
```

on_loss_begin

on_loss_begin(kwargs:Any)

Called after forward pass but before loss has been computed. Returns the output (which can allow us to modify it). [source]

Here is the place to run some code that needs to be executed after the output has been computed but before the loss computation. Example: putting the output back in FP32 when training in mixed precision.

If we return something, that will be the new value for the output.

```
show_doc(Callback.on_backward_begin)
```

on_backward_begin

> on_backward_begin(kwargs:Any)

Returns the loss (which can allow us to modify it, for instance for reg functions) [source]

Here is the place to run some code that needs to be executed after the loss has been computed but before the gradient computation. Example: `reg_fn` in RNNs.

If we return something, that will be the new value for loss. Since the recorder is always called first, it will have the raw loss.

```
show_doc(Callback.on_backward_end)
```

on_backward_end

> on_backward_end(kwargs:Any)

Called after backprop but before optimizer step. Useful for true weight decay in AdamW. [source]

Here is the place to run some code that needs to be executed after the gradients have been computed but before the optimizer is called.

```
show_doc(Callback.on_step_end)
```

on_step_end

> on_step_end(kwargs:Any)

Called after the step of the optimizer but before the gradients are zeroed. [source]

Here is the place to run some code that needs to be executed after the optimizer step but before the gradients are zeroed

```
show_doc(Callback.on_batch_end)
```

`on_batch_end`

> `on_batch_end(kwargs:Any)`

Called at the end of the batch. [source]

Here is the place to run some code that needs to be executed after a batch is fully done. Example: change the values of the hyperparameters (if we don't do it on_batch_begin instead)

If we return true, the current epoch is interrupted (example: lr_finder stops the training when the loss explodes)

```
show_doc(Callback.on_epoch_end)
```

`on_epoch_end`

> `on_epoch_end(kwargs:Any) -> bool`

Called at the end of an epoch. [source]

Here is the place to run some code that needs to be executed at the end of an epoch. Example: Save the model if we have a new best validation loss/metric.

If we return true, the training stops (example: early stopping)

```
show_doc(Callback.on_train_end)
```

`on_train_end`

> `on_train_end(kwargs:Any)`

Useful for cleaning up things and saving files/models. [source]

Here is the place to tidy everything. It's always executed even if there was an error during the training loop, and has an extra kwarg named exception to check if there was an exception or not. Examples: save log_files, load best model found during training

Annealing functions

The following functions provide different annealing schedules. You probably won't need to call them directly, but would instead use them as part of a callback. Here's what each one looks like:

```
annealings = "NO LINEAR COS EXP POLY".split()
fns = [annealing_no, annealing_linear, annealing_cos, annealing_exp, annealing_poly(0.8)]
for fn, t in zip(fns, annealings):
    plt.plot(np.arange(0, 100), [fn(2, 1e-2, o)
        for o in np.linspace(0.01,1,100)], label=t)
```

```
plt.legend();
```

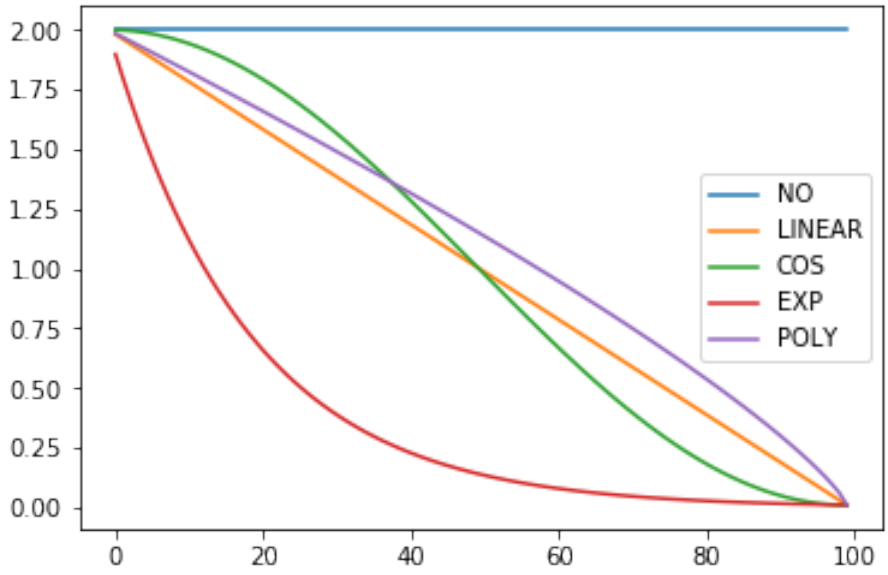

Figure 8: png

```
show_doc(annealing_cos)
```

annealing_cos

`annealing_cos(start:Number, end:Number, pct:float) -> Number`

Cosine anneal from **start** to **end** as pct goes from 0.0 to 1.0. [source]

```
show_doc(annealing_exp)
```

annealing_exp

`annealing_exp(start:Number, end:Number, pct:float) -> Number`

Exponentially anneal from **start** to **end** as pct goes from 0.0 to 1.0. [source]

```
show_doc(annealing_linear)
```

annealing_linear

`annealing_linear(start:Number, end:Number, pct:float) -> Number`

Linearly anneal from **start** to **end** as pct goes from 0.0 to 1.0. [source]

```
show_doc(annealing_no)
```

annealing_no

> annealing_no(start:Number, end:Number, pct:float) -> Number

No annealing, always return `start`. [source]

```
show_doc(annealing_poly)
```

annealing_poly

> annealing_poly(degree:Number) -> Number

Anneal polynomically from `start` to `end` as pct goes from 0.0 to 1.0. [source]

```
show_doc(CallbackHandler)
```

class CallbackHandler

> CallbackHandler(callbacks:Collection[Callback], beta:float=0.98)

Manage all of the registered callback objects, smoothing loss by momentum `beta`. [source]

You probably won't need to use this class yourself. It's used by fastai to combine all the callbacks together and call any relevant callback functions for each training stage. The methods below simply call the equivalent method in each callback function in `self.callbacks`.

```
show_doc(CallbackHandler.on_backward_begin)
```

on_backward_begin

> on_backward_begin(loss:Tensor)

Handle gradient calculation on `loss`. [source]

```
show_doc(CallbackHandler.on_backward_end)
```

on_backward_end

> on_backward_end()

Handle end of gradient calculation. [source]

```
show_doc(CallbackHandler.on_batch_begin)
```

on_batch_begin

`on_batch_begin(xb:Tensor, yb:Tensor)`

Handle new batch `xb`,`yb`. [source]

`show_doc(CallbackHandler.on_batch_end)`

on_batch_end

`on_batch_end(loss:Tensor)`

Handle end of processing one batch with `loss`. [source]

`show_doc(CallbackHandler.on_epoch_begin)`

on_epoch_begin

`on_epoch_begin()`

Handle new epoch. [source]

`show_doc(CallbackHandler.on_epoch_end)`

on_epoch_end

`on_epoch_end(val_metrics:MetricsList) -> bool`

Epoch is done, process `val_metrics`. [source]

`show_doc(CallbackHandler.on_loss_begin)`

on_loss_begin

`on_loss_begin(out:Tensor)`

Handle start of loss calculation with model output `out`. [source]

`show_doc(CallbackHandler.on_step_end)`

on_step_end

`on_step_end()`

Handle end of optimization step. [source]

`show_doc(CallbackHandler.on_train_begin)`

on_train_begin

> on_train_begin(epochs:int, pbar:PBar, metrics:MetricFuncList)

About to start learning. [source]

```
show_doc(CallbackHandler.on_train_end)
```

on_train_end

> on_train_end(exception:Union[bool, Exception])

Handle end of training, `exception` is an `Exception` or False if no exceptions during training. [source]

```
show_doc(OptimWrapper)
```

class OptimWrapper

> OptimWrapper(opt:Optimizer, wd:Floats=0.0, true_wd:bool=False, bn_wd:bool=True)

Basic wrapper around an optimizer to simplify HP changes. [source]

This is a convenience class that provides a consistent API for getting and setting optimizer hyperparameters. For instance, for `optim.Adam` the momentum parameter is actually `betas[0]`, whereas for `optim.SGD` it's simply `momentum`. As another example, the details of handling weight decay depend on whether you are using `true_wd` or the traditional L2 regularization approach.

This class also handles setting different WD and LR for each layer group, for discriminative layer training.

```
show_doc(OptimWrapper.create)
```

create

> create(opt_fn:Union[type, Callable], lr:Union[float, Tuple, List[T]], layer_groups:ModuleList, kwargs:Any) -> Optimizer

Create an optim.Optimizer from `opt_fn` with `lr`. Set lr on `layer_groups`. [source]

```
show_doc(OptimWrapper.read_defaults)
```

read_defaults

> read_defaults()

Read the values inside the optimizer for the hyper-parameters. [source]

```
show_doc(OptimWrapper.read_val)
```

read_val

> read_val(key:str) -> Union[List[float], Tuple[List[float], List[float]]]

Read a hyperparameter key in the optimizer dictionary. [source]

```
show_doc(OptimWrapper.set_val)
```

set_val

> set_val(key:str, val:Any, bn_groups:bool=True) -> Any

Set the values inside the optimizer dictionary at the key. [source]

```
show_doc(OptimWrapper.step)
```

step

> step()

Set weight decay and step optimizer. [source]

```
show_doc(OptimWrapper.zero_grad)
```

zero_grad

> zero_grad()

Clear optimizer gradients. [source]

```
show_doc(SmoothenValue)
```

class SmoothenValue

> SmoothenValue(beta:float)

Create a smooth moving average for a value (loss, etc). [source]

Used for smoothing loss in `Recorder`.

```
show_doc(SmoothenValue.add_value)
```

`add_value`

> `add_value`(`val`:`float`)

Add current value to calculate updated smoothed value. [source]

```
show_doc(Stepper)
```

`class Stepper`

> `Stepper`(`vals`:`StartOptEnd`, `n_iter`:`int`, `func`:`Optional`[`AnnealFunc`]=`None`)

Used to "step" from start,end (`vals`) over `n_iter` iterations on a schedule defined by `func` [source]

Used for creating annealing schedules, mainly for `OneCycleScheduler`.

```
show_doc(Stepper.step)
```

`step`

> `step`() -> `Number`

Return next value along annealed schedule. [source]

Undocumented Methods - Methods moved below this line will intentionally be hidden

```
show_doc(do_annealing_poly)
```

`do_annealing_poly`

> `do_annealing_poly`(`start`:`Number`, `end`:`Number`, `pct`:`float`, `degree`:`Number`) -> `Number`

Helper function for `anneal_poly`. [source]

List of callbacks

```
from fastai.gen_doc.nbdoc import *
from fastai.callbacks import *
from fastai.basic_train import *
from fastai.train import *
from fastai import callbacks
```

fastai's training loop is highly extensible, with a rich *callback* system. See the `callback` docs if you're interested in writing your own callback. See below for a list of callbacks that are provided with fastai, grouped by the module they're defined in.

Every callback that is passed to `Learner` with the `callback_fns` parameter will be automatically stored as an attribute. The attribute name is snake-cased, so for instance `ActivationStats` will appear as `learn.activation_stats` (assuming your object is named `learn`).

`callbacks`

This sub-package contains more sophisticated callbacks that each are in their own module. They are (click the link for more details):

`OneCycleScheduler`

Train with Leslie Smith's 1cycle annealing method.

`MixedPrecision`

Use fp16 to take advantage of tensor cores on recent NVIDIA GPUs for a 200% or more speedup.

`GeneralScheduler`

Create your own multi-stage annealing schemes with a convenient API.

`MixUpCallback`

Data augmentation using the method from mixup: Beyond Empirical Risk Minimization

`LRFinder`

Use Leslie Smith's learning rate finder to find a good learning rate for training your model.

`HookCallback`

Convenient wrapper for registering and automatically deregistering PyTorch hooks. Also contains pre-defined hook callback: `ActivationStats`.

`train` and `basic_train`

`Recorder`

Track per-batch and per-epoch smoothed losses and metrics.

`ShowGraph`

Dynamically display a learning chart during training.

`BnFreeze`

Freeze batchnorm layer moving average statistics for non-trainable layers.

Hook callbacks

This provides both a standalone class and a callback for registering and automatically deregistering PyTorch hooks, along with some pre-defined hooks. Hooks can be attached to any `nn.Module`, for either the forward or the backward pass.

We'll start by looking at the pre-defined hook `ActivationStats`, then we'll see how to create our own.

```
from fastai.gen_doc.nbdoc import *
from fastai.callbacks.hooks import *
from fastai.docs import *
from fastai import *
from fastai.train import *
from fastai.vision import *
```

```
show_doc(ActivationStats)
```

class ActivationStats

> `ActivationStats(learn:Learner, modules:Sequence[Module]=None, do_remove:bool=True) :: HookCallback`

Callback that record the activations. [source]

`ActivationStats` saves the layer activations in `self.stats` for all `modules` passed to it. By default it will save activations for *all* modules. For instance:

```
learn = ConvLearner(get_mnist(), tvm.resnet18, callback_fns=ActivationStats)
learn.fit(1)
```

```
Total time: 00:14
epoch  train loss  valid loss
0      0.073261    0.035140    (00:14)
```

The saved `stats` is a `FloatTensor` of shape `(2,num_batches,num_modules)`. The first axis is `(mean,stdev)`.

```
len(learn.data.train_dl),len(learn.activation_stats.modules)
```

```
(194, 44)
```

```
learn.activation_stats.stats.shape
```

```
torch.Size([2, 44, 194])
```

So this shows the standard deviation (`axis0==1`) of 5th last layer (`axis1==-5`) for each batch (`axis2`):

```
plt.plot(learn.activation_stats.stats[1][-5].numpy());
```

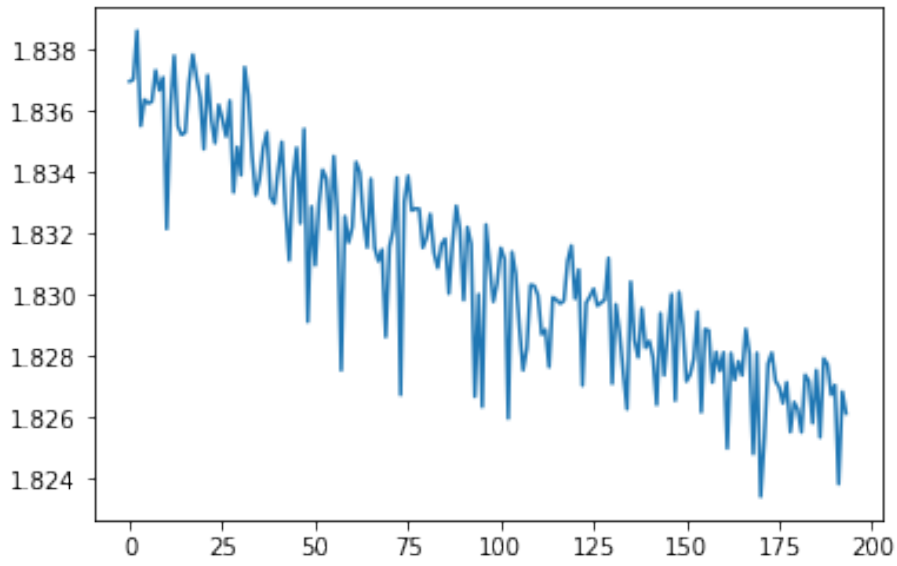

Figure 9: png

```
show_doc(Hook)
```

class Hook

> Hook(m:Module, hook_func:HookFunc, is_forward:bool=True)

Create a hook. [source]

Registers and manually deregisters a PyTorch hook. Your `hook_func` will be called automatically when forward/backward (depending on `is_forward`) for your module `m` is run, and the result of that function is placed in `self.stored`.

`show_doc(Hook.remove)`

remove

> remove() [source]

Deregister the hook, if not called already.

`show_doc(Hooks)`

class Hooks

> Hooks(ms:ModuleList, hook_func:HookFunc, is_forward:bool=True)

Create several hooks. [source]

Acts as a `Collection` (i.e. `len(hooks)` and `hooks[i]`) and an `Iterator` (i.e. `for hook in hooks`) of a group of hooks, one for each module in `ms`, with the ability to remove all as a group. Use `stored` to get all hook results. `hook_func` and `is_forward` behavior is the same as `Hook`. See the source code for `HookCallback` for a simple example.

`show_doc(Hooks.remove)`

remove

> remove() [source]

Deregister all hooks created by this class, if not previously called.

Convenience functions for hooks

`show_doc(hook_output)`

hook_output

> hook_output(module:Module) -> Hook [source]

Function that creates a Hook for module that simply stores the output of the layer.

```
show_doc(hook_outputs)
```

hook_outputs

> hook_outputs(modules:ModuleList) -> Hooks [source]

Function that creates a Hook for all passed modules that simply stores the output of the layers. For example, the (slightly simplified) source code of model_sizes is:

```
def model_sizes(m, size):
    x = m(torch.zeros(1, in_channels(m), *size))
    return [o.stored.shape for o in hook_outputs(m)]
```

```
show_doc(model_sizes)
```

model_sizes

> model_sizes(m:Module, size:tuple=(256, 256), full:bool=True) -> Tuple[Sizes, Tensor, Hooks]

Pass a dummy input through the model to get the various sizes. [source]

It can be useful to get the size of each layer of a model (e.g. for printing a summary, or for generating cross-connections for a DynamicUnet), however they depend on the size of the input. This function calculates the layer sizes by passing in a minimal tensor of size.

```
show_doc(HookCallback)
```

class HookCallback

> HookCallback(learn:Learner, modules:Sequence[Module]=None, do_remove:bool=True) :: LearnerCallback

Callback that registers given hooks. [source]

For all modules, uses a callback to automatically register a method self.hook (that you must define in an inherited class) as a hook. This method must have the signature:

```
def hook(self, m:Model, input:Tensors, output:Tensors)
```

If `do_remove` then the hook is automatically deregistered at the end of training. See `ActivationStats` for a simple example of inheriting from this class.

Undocumented Methods - Methods moved below this line will intentionally be hidden

`show_doc(HookCallback.remove)`

`remove`

> `remove`() [source]

`show_doc(HookCallback.on_train_begin)`

`on_train_begin`

> `on_train_begin`(`kwargs`)

To initialize constants in the callback. [source]

`show_doc(HookCallback.on_train_end)`

`on_train_end`

> `on_train_end`(`kwargs`)

Useful for cleaning up things and saving files/models. [source]

`show_doc(ActivationStats.hook)`

`hook`

> `hook`(`m:Module`, `i:Tensors`, `o:Tensors`) -> `Tuple`[`Rank0Tensor`, `Rank0Tensor`] [source]

`show_doc(ActivationStats.on_batch_end)`

`on_batch_end`

> `on_batch_end`(`kwargs`)

Called at the end of the batch. [source]

`show_doc(ActivationStats.on_train_begin)`

on_train_begin

on_train_begin(kwargs)

To initialize constants in the callback. [source]

```
show_doc(ActivationStats.on_train_end)
```

on_train_end

on_train_end(kwargs)

Useful for cleaning up things and saving files/models. [source]

Mixed precision training

This module allows the forward and backward passes of your neural net to be done in fp16 (also known as *half precision*). This is particularly important if you have an NVIDIA GPU with tensor cores, since it can speed up your training by 200% or more.

```
from fastai.gen_doc.nbdoc import *
from fastai.callbacks.fp16 import *
from fastai.docs import *
from fastai import *
from fastai.vision import *
```

Overview

To train your model in mixed precision you just have to call `Learner.to_fp16`, which converts the model and modifies the existing `Learner` to add `MixedPrecision`.

```
show_doc(Learner.to_fp16)
```

to_fp16

to_fp16(learn:Learner, loss_scale:float=512.0, flat_master:bool=False) -> Learner

Transform `learn` in FP16 precision. [source]

For example:

```
learn = ConvLearner(get_mnist(), tvm.resnet18, metrics=accuracy).to_fp16()
learn.fit_one_cycle(1)
```

```
Total time: 00:10
epoch  train loss  valid loss  accuracy
0      0.113659    0.068874    0.978901  (00:10)
```

Details about mixed precision training are available in NVIDIA's documentation. We will just summarize the basics here.

The only parameter you may want to tweak is `loss_scale`. This is used to scale the loss up, so that it doesn't underflow fp16, leading to loss of accuracy (this is reversed for the final gradient calculation after converting back to fp32). Generally the default `512` works well, however. You can also enable or disable the flattening of the master parameter tensor with `flat_master=True`, however in our testing the different is negligible.

Internally, the callback ensures that all model parameters (except batchnorm layers, which require fp32) are converted to fp16, and an fp32 copy is also saved. The fp32 copy (the `master` parameters) is what is used for actually updating with the optimizer; the fp16 parameters are used for calculating gradients. This helps avoid underflow with small learning rates.

All of this is implemented by the following Callback.

```
show_doc(MixedPrecision)
```

class MixedPrecision

> `MixedPrecision(learn:Learner, loss_scale:float=512.0, flat_master:bool=False) :: Callback`

Callback that handles mixed-precision training. [source]

You don't have to call the following functions yourself - they're called by the callback framework automatically. They're just documented here so you can see exactly what the callback is doing.

```
show_doc(MixedPrecision.on_backward_begin)
```

on_backward_begin

> `on_backward_begin(last_loss:Rank0Tensor, kwargs:Any) -> Rank0Tensor`

Scale gradients up by `loss_scale` to prevent underflow. [source]

```
show_doc(MixedPrecision.on_backward_end)
```

on_backward_end

> `on_backward_end(kwargs:Any)`

Convert the gradients back to FP32 and divide them by the scale. [source]

```
show_doc(MixedPrecision.on_loss_begin)
```

on_loss_begin

> on_loss_begin(last_output:Tensor, kwargs:Any) -> Tensor

Convert half precision output to FP32 to avoid reduction overflow. [source]

```
show_doc(MixedPrecision.on_step_end)
```

on_step_end

> on_step_end(kwargs:Any)

Update the params from master to model and zero grad. [source]

```
show_doc(MixedPrecision.on_train_begin)
```

on_train_begin

> on_train_begin(kwargs:Any)

Ensure everything is in half precision mode. [source]

```
show_doc(MixedPrecision.on_train_end)
```

on_train_end

> on_train_end(kwargs:Any)

Remove half precision transforms added at `on_train_begin`. [source]

The 1cycle policy

```
from fastai.gen_doc.nbdoc import *
from fastai import *
from fastai.vision import *
from fastai.docs import *
```

What is 1cycle?

This Callback allows us to easily train a network using Leslie Smith's 1cycle policy. To learn more about the 1cycle technique for training neural networks check out Leslie Smith's paper and for a more graphical and intuitive explanation check out Sylvain Gugger's post.

To use our 1cycle policy we will need an optimum learning rate. We can find this learning rate by using a learning rate finder which can be called by using `lr_finder`. It will do a mock training by going over a large range of learning rates, then plot them against the losses. We will pick a value a bit before the minimum, where the loss still improves. Our graph would look something like this:

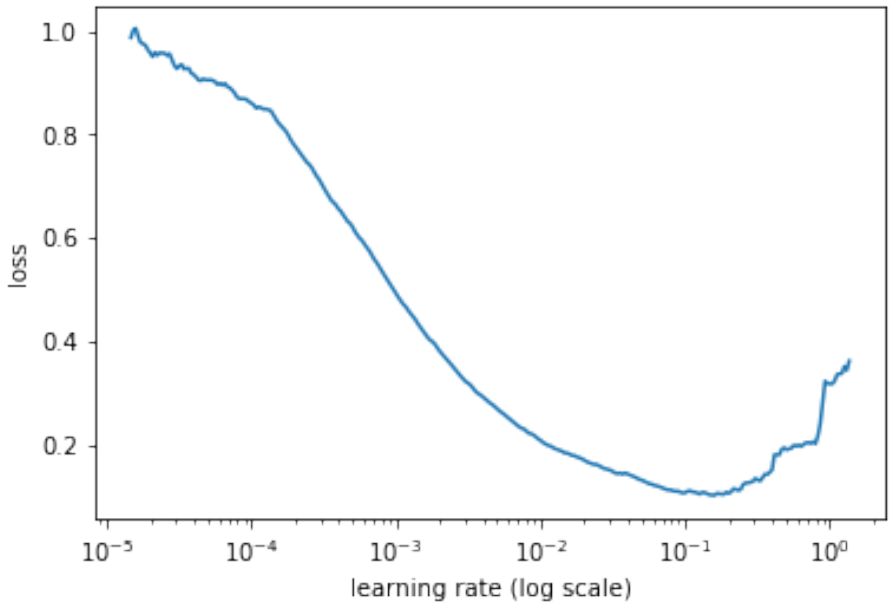

Figure 10: onecycle_finder

Here anything between `3x10^-2` and `10^-2` is a good idea.

Next we will apply the 1cycle policy with the chosen learning rate as the maximum learning rate. The original 1cycle policy has three steps:

```
1. We progressively increase our learning rate from lr_max/div_factor to lr_max and at the same t
2. We do the exact opposite: we progressively decrease our learning rate from lr_max to lr_max/di
3. We further decrease our learning rate from lr_max/div_factor to lr_max/(div_factor x 100) and
```

This gives the following form:

Unpublished work has shown even better results by using only two phases: the same phase 1, followed by a second phase where we do a cosine annealing from lr_max to 0. The momentum goes from mom_min to mom_max by following the symmetric cosine (see graph a bit below).

Basic Training

The one cycle policy allows to train very quickly, a phenomenon termed *superconvergence*. To see this in practice, we will first train a `ConvLearner` and see how our results compare when we use the `OneCycleScheduler` with `fit_one_cycle`.

```
data = get_mnist()
learn = ConvLearner(data, tvm.resnet18, metrics=accuracy, pretrained=False)
```

First lets find the optimum learning rate for our comparison by doing an LR range test.

```
learn.lr_find()
```

```
learn.recorder.plot()
```

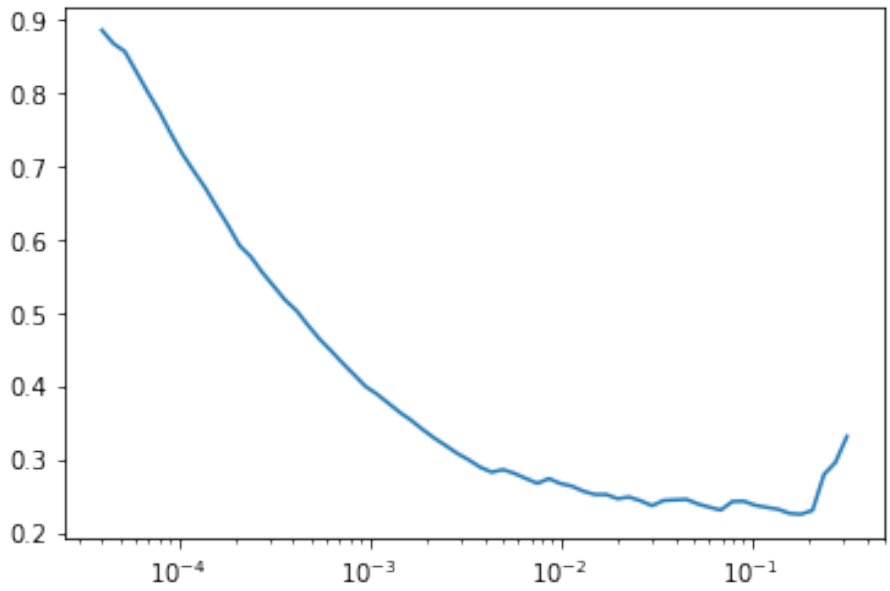

Figure 11: png

Here 2e-2 looks like a good value, a tenth of the minimum of the curve. That's going to be the highest learning rate in 1cycle so let's try a constant training at that value.

```
learn.fit(3, 2e-2)
```

```
Total time: 00:33
epoch  train loss  valid loss  accuracy
0      0.054232    0.013850    0.996075  (00:11)
```

```
1       0.029648    0.015441    0.994112  (00:10)
2       0.029239    0.035542    0.992149  (00:10)
```

We can also see what happens when we train at a lower learning rate

```
learn = ConvLearner(data, tvm.resnet18, metrics=accuracy, pretrained=False)
learn.fit(3, 2e-3)
```

```
Total time: 00:32
epoch  train loss  valid loss  accuracy
0       0.041529    0.014266    0.996075  (00:10)
1       0.041001    0.014097    0.995093  (00:10)
2       0.018079    0.005545    0.998528  (00:10)
```

Training with the 1cycle policy

Now to do the same thing with 1cycle, we use `fit_one_cycle`.

```
learn = ConvLearner(data, tvm.resnet18, metrics=accuracy, pretrained=False)
learn.fit_one_cycle(3, 2e-2)
```

```
Total time: 00:32
epoch  train loss  valid loss  accuracy
0       0.062914    0.596522    0.837095  (00:10)
1       0.024758    0.015712    0.994112  (00:10)
2       0.005879    0.008754    0.998037  (00:10)
```

This gets the best of both world and we can see how we get a far better accuracy and a far lower loss in the same number of epochs. It's possible to get to the same amazing results with training at constant learning rates, that we progressively diminish, but it will take a far longer time.

Here is the schedule of the lrs (left) and momentum (right) that the new 1cycle policy uses.

```
learn.recorder.plot_lr(show_moms=True)
```

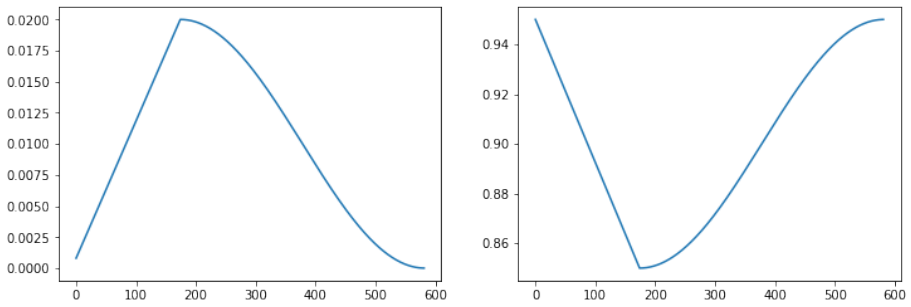

Figure 12: png

```
show_doc(OneCycleScheduler, doc_string=False)
```

class OneCycleScheduler

> `OneCycleScheduler`(`learn`:`Learner`, `lr_max`:`float`, `moms`:`Floats`=`(0.95, 0.85)`, `div_factor`:`float`=`25.0`, `pct_start`:`float`=`0.3`) :: `Callback` [source]

class OneCycleScheduler

> `OneCycleScheduler`(`learn`:`Learner`, `lr_max`:`float`, `moms`:`Floats`=`(0.95, 0.85)`, `div_factor`:`float`=`25.0`, `pct_start`:`float`=`0.3`) :: `Callback` [source]

Create a `Callback` that handles the hyperparameters settings following the 1cycle policy for `learn`. `lr_max` should be picked with the `lr_find` test. In phase 1, the learning rates goes from `lr_max/div_factor` to `lr_max` linearly while the momentum goes from `moms[0]` to `moms[1]` linearly. In phase 2, the learning rates follows a cosine annealing from `lr_max` to 0, as the momentum goes from `moms[1]` to `moms[0]` with the same annealing.

```
show_doc(OneCycleScheduler.steps, doc_string=False)
```

steps

> `steps`(`steps_cfg`:`StartOptEnd`) [source]

steps

> `steps`(`steps_cfg`:`StartOptEnd`) [source]

Build the `Stepper` for the `Callback` according to `steps_cfg`.

```
show_doc(OneCycleScheduler.on_train_begin, doc_string=False)
```

on_train_begin

> `on_train_begin`(`n_epochs`:`int`, `kwargs`:`Any`) [source]

on_train_begin

> `on_train_begin`(`n_epochs`:`int`, `kwargs`:`Any`) [source]

Initiate the parameters of a training for `n_epochs`.

```
show_doc(OneCycleScheduler.on_batch_end, doc_string=False)
```

`on_batch_end`

`on_batch_end`(`kwargs:Any`) [source]

`on_batch_end`

`on_batch_end`(`kwargs:Any`) [source]

Prepares the hyperparameters for the nex batch.

Learning Rate Finder

```
from fastai.gen_doc.nbdoc import *
from fastai import *
from fastai.docs import *
```

Learning rate finder plots lr vs loss relationship for a `Learner`. The idea is to reduce the amount of guesswork on picking a good starting learning rate.

Overview:
1. First run lr_find `learn.lr_find()` 2. Plot the learning rate vs loss `learn.recorder.plot()` 3. Pick a learning rate before it diverges then start training

Technical Details: (first described by Leslie Smith)
>Train `Learner` over a few iterations. Start with a very low `start_lr` and change it at each mini-batch until it reaches a very high `end_lr`. `Recorder` will record the loss at each iteration. Plot those losses against the learning rate to find the optimal value before it diverges.

Choosing a good learning rate

For a more intuitive explanation, please check out Sylvain Gugger's post

```
data = get_mnist()
def simple_learner(): return Learner(data, simple_cnn((3,16,16,2)))
learn = simple_learner()
```

First we run this command to launch the search.

```
learn.lr_find()
```

Then we plot the loss versus the learning rates. We're interested in finding a good order of magnitude of learning rate, so we plot with a log scale.

```
learn.recorder.plot()
```

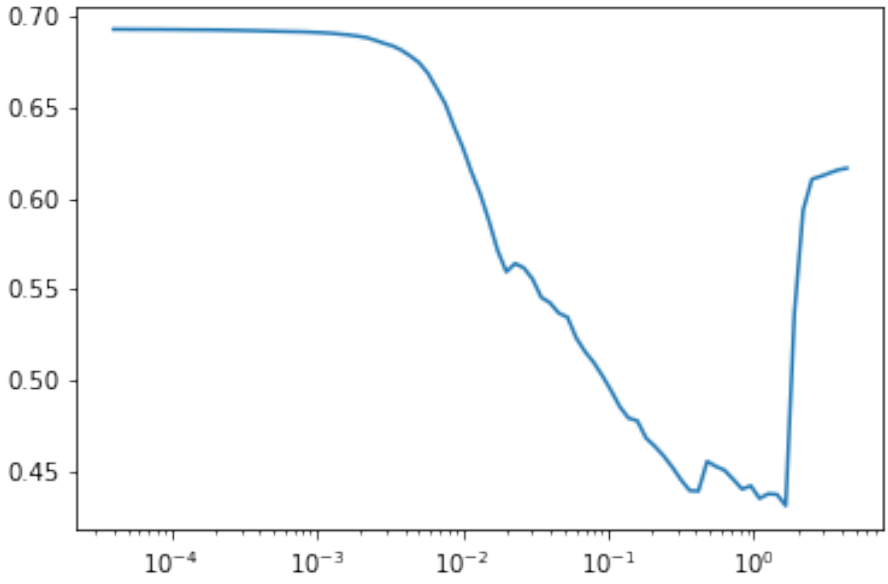

Figure 13: png

Then, we choose a value that is an order of magnitude before the mimum: the minimum value is on the edge diverging so it is too high. An order of magnitude before, a value that's still aggressive (for quicker training) but still safer from exploding. (In this example case 1e-1 is a good choice).

Let's start training with this optimal value:

```
simple_learner().fit_one_cycle(2, 1e-1)
```

```
Total time: 00:05
epoch  train loss  valid loss
0      0.088468    0.040429    (00:02)
1      0.036378    0.032030    (00:02)
```

Picking the minimum isn't a good idea because training will diverge.

```
simple_learner().fit_one_cycle(2, 1e-0)
```

```
Total time: 00:05
epoch  train loss  valid loss
0      0.399384    0.449538    (00:02)
1      0.202556    0.185107    (00:02)
```

Picking a value to far below the minimum isn't optimal because training is too slow.

```
simple_learner().fit_one_cycle(2, 1e-2)
```

```
Total time: 00:05
epoch  train loss  valid loss
0      0.131312    0.081172    (00:02)
1      0.072979    0.058261    (00:02)
```

```
show_doc(LRFinder, doc_string=False)
```

class LRFinder

> LRFinder(learn:Learner, start_lr:float=1e-05, end_lr:float=10, num_it:int=100) :: LearnerCallback [source]

Creates a `LRFinder` `Callback` for `learn` to go on a mock training from `start_lr` to `end_lr` for `num_it` iterations. Training is interrupted before the end of `num_it` if the losses diverge too early.

Undocumented Methods - Methods moved below this line will intentionally be hidden

```
show_doc(LRFinder.on_train_end)
```

on_train_end

> on_train_end(kwargs:Any)

Cleanup learn model weights disturbed during LRFind exploration. [source]

```
show_doc(LRFinder.on_batch_end)
```

on_batch_end

> on_batch_end(iteration:int, smooth_loss:TensorOrNumber, kwargs:Any)

Determine if loss has runaway and we should stop. [source]

```
show_doc(LRFinder.on_train_begin)
```

on_train_begin

> on_train_begin(kwargs:Any)

Initialize optimizer and learner hyperparameters. [source]

```
show_doc(LRFinder.on_epoch_end)
```

`on_epoch_end`

`on_epoch_end(kwargs:Any)`

Tell Learner if we need to stop. [source]

Mixup data augmentation

```
from fastai.gen_doc.nbdoc import *
from fastai.callbacks.mixup import *
from fastai.docs import *
from fastai import *
```

What is Mixup?

This module contains the implementation of a data augmentation technique called Mixup. It is extremely efficient at regularizing models in computer vision (we used it to get our time to train CIFAR10 to 94% on one GPU to 6 minutes).

As the name kind of suggests, the authors of the mixup article propose to train the model on a mix of the pictures of the training set. Let's say we're on CIFAR10 for instance, then instead of feeding the model the raw images, we take two (which could be in the same class or not) and do a linear combination of them: in terms of tensor it's

```
new_image = t * image1 + (1-t) * image2
```

where t is a float between 0 and 1. Then the target we assign to that image is the same combination of the original targets:

```
new_target = t * target1 + (1-t) * target2
```

assuming your targets are one-hot encoded (which isn't the case in pytorch usually). And that's as simple as this.

Dog or cat? The right answer here is 70% dog and 30% cat!

As the picture above shows, it's a bit hard for a human eye to comprehend the pictures obtained (although we do see the shapes of a dog and a cat) but somehow, it makes a lot of sense to the model which trains more efficiently. The final loss (training or validation) will be higher than when training without mixup even if the accuracy is far better, which means that a model trained like this will make predictions that are a bit less confident.

Figure 14: mixup

Basic Training

To test this method, we will first build a `simple_cnn` and train it like we did with `basic_train` so we can compare its results with a network trained with Mixup.

```
data = get_mnist()
model = simple_cnn((3,16,16,2))
learn = Learner(data, model, metrics=[accuracy])
```

```
learn.fit(10)
```

```
Total time: 00:26
epoch  train loss  valid loss  accuracy
0      0.133347    0.105171    0.962709  (00:02)
1      0.092242    0.095590    0.970559  (00:02)
2      0.080206    0.065994    0.979392  (00:02)
3      0.062501    0.052990    0.981845  (00:02)
4      0.052910    0.047916    0.984298  (00:02)
5      0.049038    0.063314    0.976938  (00:02)
6      0.039721    0.044997    0.983808  (00:02)
7      0.041155    0.034764    0.986752  (00:02)
8      0.030892    0.035803    0.987242  (00:02)
9      0.026575    0.028857    0.990186  (00:02)
```

Mixup implementation in the library

In the original article, the authors suggested four things:

```
1. Create two separate dataloaders and draw a batch from each at every iteration to mix them up
2. Draw a t value following a beta distribution with a parameter alpha (0.4 is suggested in their
3. Mix up the two batches with the same value t.
4. Use one-hot encoded targets
```

The implementation of this module is based on these suggestions but was modified when experiments suggested modifications with positive impact in performance.

The authors suggest to use the beta distribution with the same parameters alpha. Why do they suggest this? Well it looks like this:

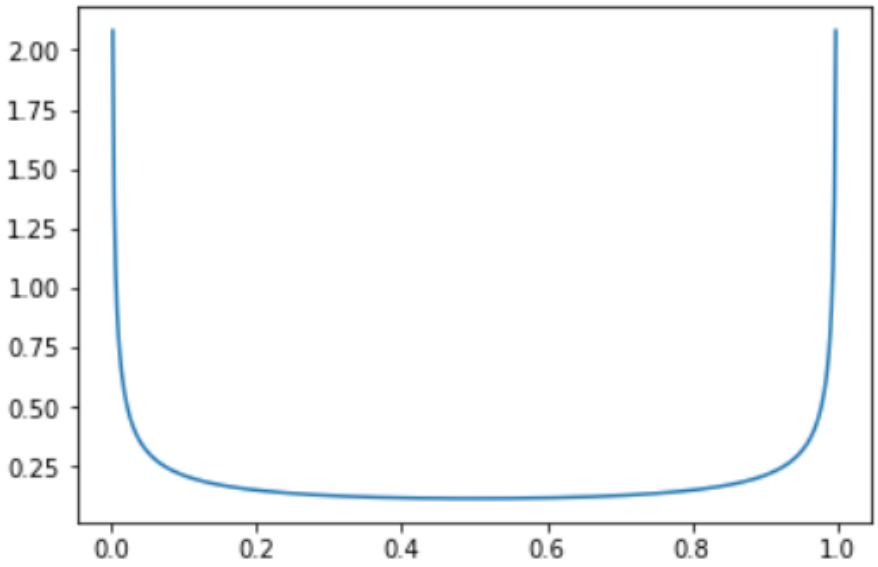

Figure 15: betadist

so it means there is a very high probability of picking values close to 0 or 1 (in which case the image is almost from 1 category) and then a somewhat constant probability of picking something in the middle (0.33 as likely as 0.5 for instance).

While this works very well, it's not the fastest way we can do this and this is the first suggestion we will adjust. The main point that slows down this process is wanting two different batches at every iteration (which means loading twice the amount of images and applying to them the other data augmentation function). To avoid this slow down, ou be a little smarter and mixup a batch with a shuffled version of itself (this way the images mixed up are still different).

Using the same parameter t for the whole batch is another suggestion we will modify. In our experiments, we noticed that the model can train faster if we draw a different `t` for every image in the batch (both options get to the same result in terms of accuracy, it's just that one arrives there more slowly). The last trick we have to apply with this is that there can be some duplicates with this strategy: let's say we decide to mix `image0` with `image1` then `image1` with `image0`, and that we draw `t=0.1` for the first, and `t=0.9` for the second. Then

```
image0 * 0.1 + shuffle0 * (1-0.1) = image0 * 0.1 + image1 * 0.9
```

and

```
image1 * 0.9 + shuffle1 * (1-0.9) = image1 * 0.9 + image0 * 0.1
```

will be the sames. Of course we have to be a bit unlucky but in practice, we saw there was a drop in accuracy by using this without removing those duplicates. To avoid them, the tricks is to replace the vector of parameters `t` we drew by:

```
t = max(t, 1-t)
```

The beta distribution with the two parameters equal is symmetric in any case, and this way we insure that the biggest coefficient is always near the first image (the non-shuffled batch).

Adding Mixup to the Mix

Now we will add `MixUpCallback` to our Learner so that it modifies our input and target accordingly. The `mixup` function does that for us behind the scene, with a few other tweaks detailed below.

```
model = simple_cnn((3,16,16,2))
learner = Learner(data, model, metrics=[accuracy]).mixup()
learner.fit(10)
```

```
Total time: 00:28
epoch  train loss  valid loss  accuracy
0      0.375882    0.172907    0.955348  (00:03)
1      0.352643    0.145850    0.970559  (00:02)
2      0.325987    0.108551    0.978901  (00:02)
3      0.313872    0.107560    0.985280  (00:02)
4      0.314499    0.095656    0.987733  (00:02)
5      0.317359    0.089879    0.988714  (00:02)
6      0.312434    0.092151    0.989696  (00:02)
7      0.307406    0.088207    0.990677  (00:02)
8      0.301020    0.081922    0.991659  (00:02)
9      0.308276    0.089387    0.992149  (00:02)
```

Training the net with Mixup improves the best accuracy. Note that the validation loss is higher than without MixUp, because the model makes less confident

predictions: without mixup, most precisions are very close to 0. or 1. (in terms of probability) whereas the model with MixUp will give predictions that are more nuanced. Be sure to know what is the thing you want to optimize (lower loss or better accuracy) before using it.

```
show_doc(MixUpCallback, doc_string=False)
```

Create a `Callback` for mixup on `learn` with a parameter `alpha` for the beta distribution. `stack_x` and `stack_y` determines if we stack our inputs/targets with the vector lambda drawn or do the linear combination (in general, we stack the inputs or ouputs when they correspond to categories or classes and do the linear combination otherwise).

```
show_doc(MixUpCallback.on_batch_begin, doc_string=False)
```

Draws a vector of lambda following a beta distribution with `self.alpha` and operates the mixup on `last_input` and `last_target` according to `self.stack_x` and `self.stack_y`.

Dealing with the loss

We often have to modify the loss so that it is compatible with Mixup: pytorch was very careful to avoid one-hot encoding targets when it could, so it seems a bit of a drag to undo this. Fortunately for us, if the loss is a classic cross-entropy, we have

```
loss(output, new_target) = t * loss(output, target1) + (1-t) *
loss(output, target2)
```

so we won't one-hot encode anything and just compute those two losses then do the linear combination.

The following class is used to adapt the loss to mixup. Note that the `mixup` function will use it to change the `Learner.loss_fn` if necessary.

```
show_doc(MixUpLoss, doc_string=False, title_level=3)
```

Create a loss function from `crit` that is compatible with MixUp.

Undocumented Methods - Methods moved below this line will intentionally be hidden

```
show_doc(MixUpLoss.forward)
```

Training tweaks for an RNN

```
from fastai.gen_doc.nbdoc import *
```

```
from fastai.callbacks.rnn import *
```

```
from fastai.gen_doc.nbdoc import *
from fastai.callbacks.rnn import *
```

This callback regroups a few tweaks to properly train RNNs. They all come from this article by Stephen Merity et al.

Adjusting the learning rate to sequence length: since we're modifying the bptt at each batch, sometimes by a lot (we divide it by 2 randomly), the learning rate has to be adjusted to take this into account, mainly being multiplied by the ratio `seq_len/bptt`.

Activation Regularization: on top of weight decay, we apply another form of regularization that is pretty similar and consists in adding to the loss a scaled factor of the sum of all the squares of the ouputs (with dropout applied) of the various layers of the RNN. Intuitively, weight decay tries to get the network to learn small weights, this is to get the model to learn to produce smaller activations.

Temporal Activation Regularization: lastly, we add to the loss a scaled factor of the sum of the squares of the `h_(t+1) - h_t`, where `h_i` is the output (before dropout is applied) of one layer of the RNN at the time step i (word i of the sentence). This will encourage the model to produce activations that don't vary too fast between two consecutive words of the sentence.

```
show_doc(RNNTrainer, doc_string=False)
```

class RNNTrainer

> `RNNTrainer(learn:Learner, bptt:int, alpha:float=0.0, beta:float=0.0, adjust:bool=True) :: Callback` [source]

Create a `Callback` that adds to learner the RNN tweaks for training on data with `bptt`. `alpha` is the scale for AR, `beta` is the scale for TAR. If `adjust` is False, the learning rate isn't adjusted to the sequence length.

```
show_doc(RNNTrainer.on_loss_begin, doc_string=False)
```

on_loss_begin

> `on_loss_begin(last_output:Tuple[Tensor, Tensor, Tensor], kwargs)` [source]

The fastai RNNs return `last_ouput` that are tuples of three elements, the true output (that is returned) and the hidden states before and after dropout (which are saved internally for the next function).

```
show_doc(RNNTrainer.on_backward_begin, doc_string=False)
```

`on_backward_begin`

> `on_backward_begin`(`last_loss`:`Rank0Tensor`, `last_input`:`Tensor`, `kwargs`) [source]

Adjusts the learning rate to the size of `last_input`. Adds to `last_loss` the AR and TAR.

TrainingPhase and General scheduler

Creates a scheduler that lets you train a model with following different `TrainingPhase`.

```
from fastai.gen_doc.nbdoc import *
from fastai.callbacks.general_sched import *
from fastai import *
from fastai.docs import *
from fastai.vision import *
```

```
show_doc(TrainingPhase, doc_string=False)
```

`class TrainingPhase`

> `TrainingPhase`(`length`:`int`, `lrs`:`Floats`, `moms`:`Floats`, `lr_anneal`:`AnnealFunc`=`None`, `mom_anneal`:`AnnealFunc`=`None`) [source]

Create a phase for training a model during `length` iterations, following a schedule given by `lrs` and `lr_anneal`, `moms` and `mom_anneal`. More specifically, the phase will make the learning rate (or momentum) vary from the first value of `lrs` (or `moms`) to the second, following `lr_anneal` (or `mom_anneal`). If an annealing function is speficied but `lrs` or `moms` is a float, it will decay to 0. If no annealing function is specified, the default is a linear annealing if `lrs` (or `moms`) is a tuple, a constant parameter if it's a float.

```
show_doc(GeneralScheduler)
```

`class GeneralScheduler`

> `GeneralScheduler`(`learn`:`Learner`, `phases`:`Collection`[`TrainingPhase`]) :: `Callback`

Schedule multiple `TrainingPhase` for a `Learner`. [source]

```
show_doc(GeneralScheduler.on_batch_end, doc_string=False)
```

on_batch_end

on_batch_end(kwargs:Any) [source]

Takes a step in the current phase and prepare the hyperparameters for the next batch.

```
show_doc(GeneralScheduler.on_train_begin, doc_string=False)
```

on_train_begin

on_train_begin(n_epochs:int, kwargs:Any) [source]

Initiates the hyperparameters to the start values of the first phase.

Let's make an example by using this to code SGD with warm restarts.

```
def fit_sgd_warm(learn, n_cycles, lr, mom, cycle_len, cycle_mult):
    n = len(learn.data.train_dl)
   phases = [TrainingPhase(n * (cycle_len * cycle_mult**i), lr, mom, lr_anneal=annealing_cos) f
    sched = GeneralScheduler(learn, phases)
    learn.callbacks.append(sched)
    if cycle_mult != 1:
     total_epochs = int(cycle_len * (1 - (cycle_mult)**n_cycles)/(1-cycle_mult))
    else: total_epochs = n_cycles * cycle_len
    learn.fit(total_epochs)
```

```
data = get_mnist()
learn = ConvLearner(data, tvm.resnet18)
fit_sgd_warm(learn, 3, 1e-3, 0.9, 1, 2)
```

```
Total time: 00:53
epoch  train loss  valid loss
0      0.095063    0.066866    (00:07)
1      0.056481    0.031150    (00:07)
2      0.024891    0.030582    (00:07)
3      0.020713    0.020097    (00:07)
4      0.017127    0.018523    (00:07)
5      0.013179    0.019353    (00:07)
6      0.006822    0.017379    (00:07)
```

```
learn.recorder.plot_lr()
```

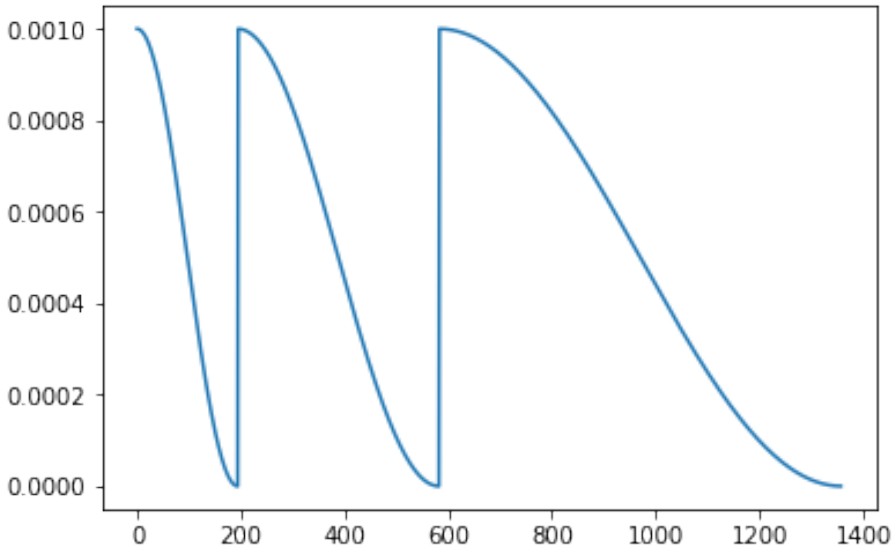

Figure 16: png

Undocumented Methods - Methods moved below this line will intentionally be hidden

Application fields

`from fastai.gen_doc.nbdoc import *`

The fastai library allows you to train a `Model` on a certain `DataBunch` very easily by binding them together inside a `Learner` object. This module regroups the tools the library provides to help you preprocess and group your data in this format.

`collab`

This submodule handles the collaborative filtering problems.

`tabular`

This sub-package deals with tabular (or structured) data.

`text`

This sub-package contains everything you need for Natural Language Processing.

`vision`

This sub-package contains the classes that deal with Computer Vision.

Module structure

In each case (except for `collab`), the module is organized this way:

`transform`

This sub-module deals with the pre-processing (data augmentation for images, cleaning for tabular data, tokenizing and numericalizing for text).

`data`

This sub-module defines the dataset class(es) to deal with this kind of data.

`models`

This sub-module defines the specific models used for this kind of data.

`learner`

When it exists, this sub-module contains functions will directly bind this data with a suitable model and add the necessary callbacks.

Computer vision

```
from fastai.gen_doc.nbdoc import *
from fastai.vision import *
from fastai import *
from fastai.docs import *
```

The `vision` module of the fastai library contains all the necessary functions to define a Dataset and train a model for computer vision tasks. It contains four different submodules to reach that goal: - `vision.image` contains the basic definition of an `Image` object and all the functions that are used behind the scenes to apply transformations to such an object, - `vision.transform` contains all the transforms we can use for data augmentation, - `vision.data` contains the definition of `ImageClassificationDataset` as well as the utility function to easily build a `DataBunch` for Computer Vision problems. - `vision.learner` lets you build and fine-tune models with a pretrained CNN backbone or train a randomly initialized model from scratch.

Each of the four module links above includes a quick overview and examples of the functionality of that module, as well as complete API documentation. Below, we'll provide a walk-thru of end to end computer vision model training with the most commonly used functionality.

Minimal training example

First, create a data folder containing a MNIST subset in `data/mnist_sample` (which we're calling `MNIST_PATH`), using this little documentation helper that will download it for you:

```
untar_data(MNIST_PATH)
MNIST_PATH
```

```
PosixPath('../data/mnist_sample')
```

Since this contains standard `train` and `valid` folders, and each contains one folder per class, you can create a `DataBunch` in a single line:

```
data = image_data_from_folder(MNIST_PATH)
```

You load a pretrained model ready for fine tuning (`tvm` is the namespace for `torchvision.models`):

```
learn = ConvLearner(data, tvm.resnet18, metrics=accuracy)
```

And now you're ready to train!

```
learn.fit(1)
```

```
Total time: 00:05
epoch  train loss  valid loss  accuracy
0      0.073646    0.048234    0.987242  (00:05)
```

Let's look briefly at each of the `vision` submodules.

Getting the data

The most important piece of `vision.data` for classification is the `ImageClassificationDataset`. If you've got labels as subfolders, then you can just say:

```
ds = ImageClassificationDataset.from_folder(MNIST_PATH/'train')
```

Images

That brings us to `vision.image`, which defines the `Image` class. Our dataset will return `Image` objects when we index it. Images automatically display in notebooks:

```
img,label = ds[0]
img
```

Figure 17: png

You can change the way they're displayed:

```
img.show(figsize=(2,2), title='MNIST digit')
```

Figure 18: png

And you can transform them in various ways:

```
img.rotate(35)
```

Figure 19: png

Data augmentation

`vision.transforms` lets us do data augmentation. Simplest is to choose from a standard set of transforms, where the defaults are designed for photos:

```
help(get_transforms)
```

```
Help on function get_transforms in module fastai.vision.transform:

get_transforms(do_flip: bool = True, flip_vert: bool = False, max_rotate: float = 10.0, max_zoom
   Utility func to easily create a list of flip, rotate, `zoom`, warp, lighting transforms.
```

...or create the exact list you want:

```
tfms = [rotate(degrees=(-20,20)), symmetric_warp(magnitude=(-0.3,0.3))]
```

You can apply these to an existing dataset:

```
tds = DatasetTfm(ds, tfms)
```

```
fig,axes = plt.subplots(1,4,figsize=(8,2))
for ax in axes: apply_tfms(tfms, ds[0][0]).show(ax=ax)
```

Figure 20: png

You can create a `DataBunch` with your transformed training and validation data loaders in a single step, passing in a tuple of *(train_tfms, valid_tfms)*:

```
data = image_data_from_folder(MNIST_PATH, ds_tfms=(tfms, []))
```

Training and interpretation

Now you're ready to train a model. To create a model, simply pass your `DataBunch` and a model creation function (such as one provided by `tvm`:

torchvision.models) to ConvLearner, and call fit:

```
learn = ConvLearner(data, tvm.resnet18, metrics=accuracy)
learn.fit(1)
```

```
Total time: 00:09
epoch  train loss  valid loss  accuracy
0      0.127079    0.064693    0.977920  (00:09)
```

Now we can take a look at the most incorrect images, and also the classification matrix.

```
interp = ClassificationInterpretation.from_learner(learn)
```

```
HBox(children=(IntProgress(value=0, max=16), HTML(value='')))
```

```
HBox(children=(IntProgress(value=0, max=16), HTML(value='0.00% [0/16 00:00<00:00]')))
```

```
interp.plot_top_losses(9, figsize=(6,6))
```

```
interp.plot_confusion_matrix()
```

Computer Vision Learner

vision.learner is the module that defines the Conv_Learner class, to easily get a model suitable for transfer learning.

```
from fastai.gen_doc.nbdoc import *
from fastai.vision import *
from fastai import *
from fastai.docs import *
```

Transfer learning

Transfer learning is a technique where you use a model trained on a very large dataset (usually ImageNet in computer vision) and then adapt it to your own dataset. The idea is that it has learned to recognize many features on all of this data, and that you will benefit from this knowledge, especially if your dataset is small, compared to starting from a randomly initiliazed model. It has been proved in this article on a wide range of tasks that transfer learning nearly always give better results.

In practice, you need to change the last part of your model to be adapted to your own number of classes. Most convolutional models end with a few linear layers (a part will call head). The last convolutional layer will have analyzed features in the image that went through the model, and the job of the head is

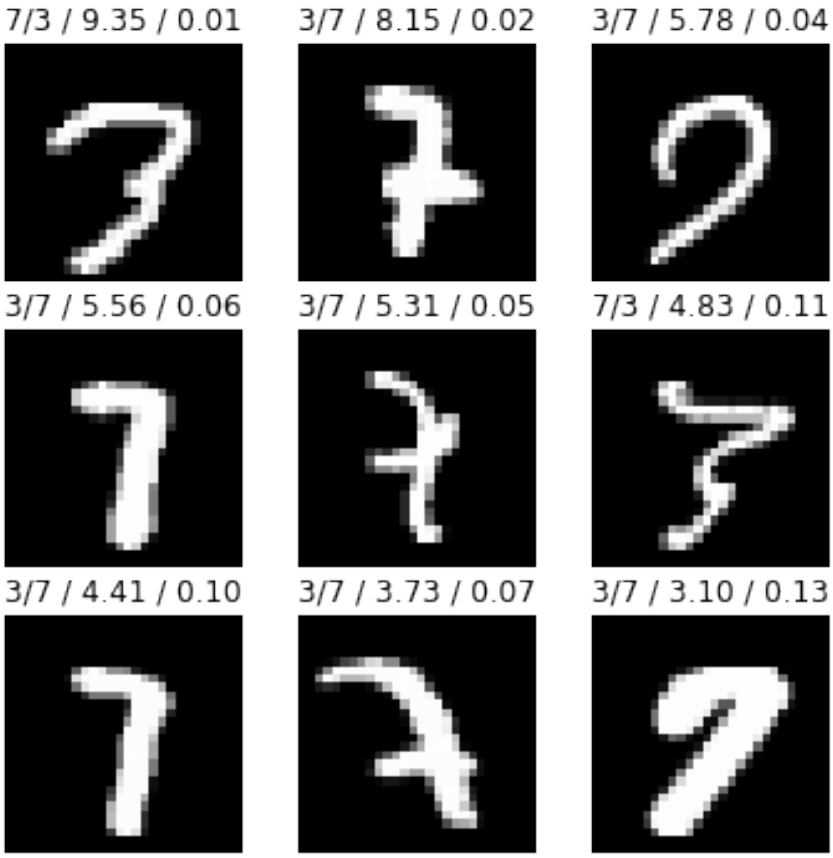

Figure 21: png

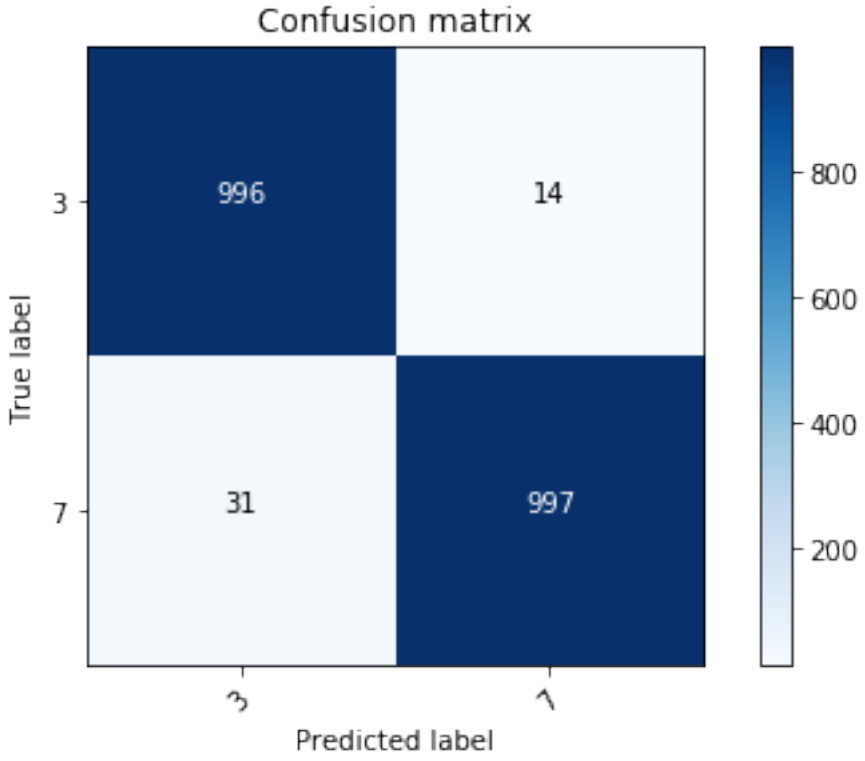

Figure 22: png

to convert those in predictions for each of our classes. In transfer learning we will keep all the convolutional layers (called the body or the backbone of the model) with their weights pretrained on ImageNet but will define a new head initiliazed randomly.

Then we will train the model we obtain in two phases: first we freeze the body weights and only train the head (to convert those analyzed features into predictions for our own data), then we unfreeze the layers of the backbone (gradually if necessary) and fine-tune the whole model (possily using differential learning rates).

The `ConvLearner` class helps you to automatically get a pretrained model from a given architecture with a custom head that is suitable for your data.

```
show_doc(ConvLearner, doc_string=False)
```

class ConvLearner

> `ConvLearner(data:DataBunch, arch:Callable, cut:Union[int, Callable]=None, pretrained:bool=True, lin_ftrs:Optional[Collection[int]]=None, ps:Floats=0.5, custom_head:Optional[Module]=None, split_on:Union[Callable, Collection[ModuleList], NoneType]=None, kwargs:Any) :: Learner` [source]

This class creates a `Learner` object from the `data` object and model inferred from it with the backbone given in `arch`. Specifically, it will cut the model defined by `arch` (randomly initialized if `pretrained` is False) at the last convolutional layer by default (or as defined in `cut`, see below) and add: - an `AdaptiveConcatPool2d` layer, - a `Flatten` layer, - blocks of [`nn.BatchNorm1d`, `nn.Dropout`, `nn.Linear`, `nn.ReLU`] layers.

The blocks are defined by the `lin_ftrs` and `ps` arguments. Specifically, the first block will have a number of inputs inferred from the backbone `arch` and the last one will have a number of outputs equal to `data.c` (which contains the number of classes of the data) and the intermediate blocks have a number of inputs/outputs determined by `lin_frs` (of course a block has a number of inputs equal to the number of outputs of the previous block). The default is to have an intermediate hidden size of 512 (which makes two blocks `model_activation` -> 512 -> `n_classes`). If you pass a float then the final dropout layer will have the value `ps`, and the remaining will be `ps/2`. If you pass a list then the values are used for dropout probabilities directly.

Note that the very last block doesn't have a `nn.ReLU` activation, to allow you to use any final activation you want (generally included in the loss function in pytorch). Also, the backbone will be frozen if you choose `pretrained=True` (so only the head will train if you call `fit`) so that you can immediately start phase one of training as described above.

Alternatively, you can define your own `custom_head` to put on top of the backbone. If you want to specify where to split `arch` you should so in the argument `cut` which can either be the index of a specific layer (the result will not include that layer) or a function that, when passed the model, will return the backbone you want.

The final model obtained by stacking the backbone and the head (custom or defined as we saw) is then separated in groups for gradual unfreezeing or differential learning rates. You can specify of to split the backbone in groups with the optional argument `split_on` (should be a function that returns those groups when given the backbone).

The `kwargs` will be passed on to `Learner`, so you can put here anything that `Learner` will accept (`metrics`, `loss_fn`, `opt_fn`…)

```
untar_data(MNIST_PATH)
data = image_data_from_folder(MNIST_PATH, ds_tfms=get_transforms(do_flip=False, max_warp=0),
```

```
learner = ConvLearner(data, tvm.resnet18, metrics=[accuracy])
learner.fit_one_cycle(1,1e-3)
```

```
Total time: 00:09
epoch  train loss  valid loss  accuracy
0      0.131621    0.065316    0.976448  (00:09)
```

Customize your model

You can customize `ConvLearner` for your own models default `cut` and `split_on` functions by adding it them the dictionary `model_meta`. The key should be your model and the value should be a dictionary with the keys `cut` and `split_on` (see the source code for examples). The constructor will call `create_body` and `create_head` for you based on `cut`; you can also call them yourself, which is particularly useful for testing.

```
show_doc(create_body)
```

`create_body`

> `create_body`(`model`:`Module`, `cut`:`Optional`[`int`]=`None`, `body_fn`:`Callable`[`Module`, `Module`]=`None`)

Cut off the body of a typically pretrained `model` at `cut` or as specified by `body_fn`. [source]

```
show_doc(create_head, doc_string=False)
```

`create_head`

`create_head`(`nf`:`int`, `nc`:`int`, `lin_ftrs`:`Optional`[`Collection`[`int`]]=`None`, `ps`:`Floats`=`0.5`) [source]

Model head that takes `nf` features, runs through `lin_ftrs`, and ends with `nc` classes. `ps` is the probability of the dropouts, as documented above in `ConvLearner`.

Utility methods

```
show_doc(num_features)
```

`num_features`

`num_features`(`m`:`Module`) -> `int`

Return the number of output features for a `model`. [source]

```
show_doc(ClassificationInterpretation)
```

`class ClassificationInterpretation`

`ClassificationInterpretation`(`data`:`DataBunch`, `y_pred`:`Tensor`, `y_true`:`Tensor`, `loss_class`:`type`=`'CrossEntropyLoss'`, `sigmoid`:`bool`=`True`)

Interpretation methods for classification models. [source]

This provides a confusion matrix and visualization of the most incorrect images. Pass in your `data`, calculated `preds`, actual `y`, and the class of your loss function, and then use the methods below to view the model interpretation results. For instance:

```
learn = ConvLearner(get_mnist(), tvm.resnet18)
learn.fit(1)
preds,y = learn.get_preds()
interp = ClassificationInterpretation(data, preds, y, loss_class=nn.CrossEntropyLoss)
```

```
Total time: 00:07
epoch  train loss  valid loss
0      0.093727    0.043710    (00:07)
```

```
HBox(children=(IntProgress(value=0, max=16), HTML(value='')))
```

```
HBox(children=(IntProgress(value=0, max=16), HTML(value='0.00% [0/16 00:00<00:00]')))
```

```
show_doc(ClassificationInterpretation.plot_top_losses)
```

plot_top_losses

> `plot_top_losses`(`k`, `largest`=`True`, `figsize`=`(12, 12)`)

Show images in `top_losses` along with their loss, label, and prediction. [source]

The `k` items are arranged as a square, so it will look best if `k` is a square number (4, 9, 16, etc). The title of each image shows: prediction, actual, loss, probability of actual class.

```
interp.plot_top_losses(9, figsize=(7,7))
```

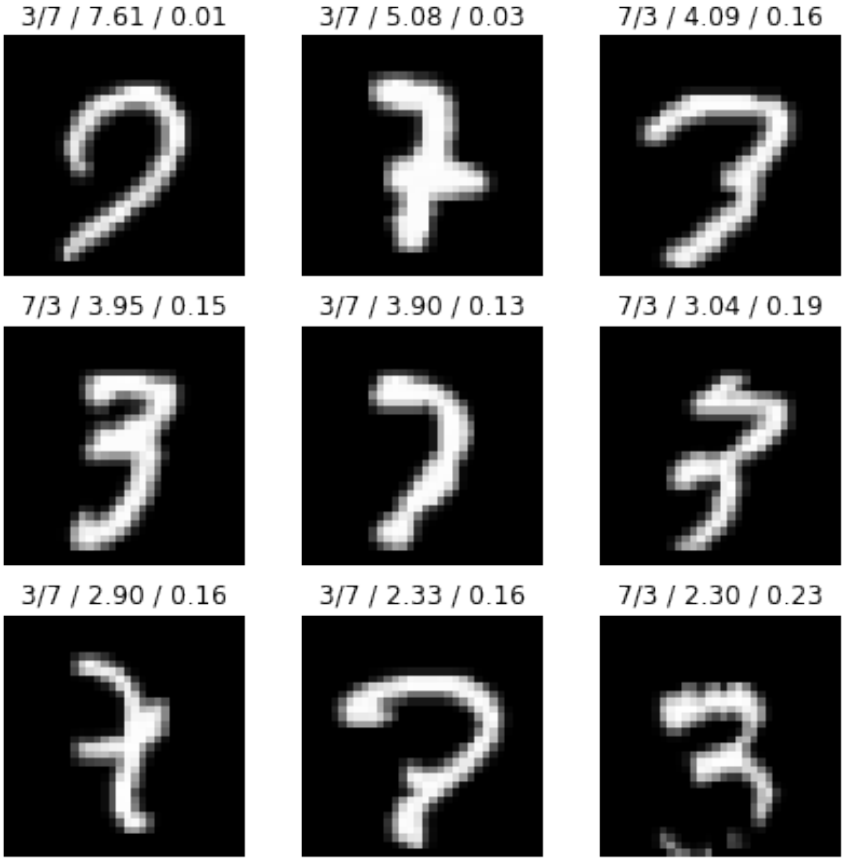

Figure 23: png

```
show_doc(ClassificationInterpretation.top_losses)
```

top_losses

> top_losses(k, largest=True)

k largest(/smallest) losses. [source]

Returns tuple of *(losses,indices)*.

```
interp.top_losses(9)
```

```
(tensor([7.6106, 5.0751, 4.0854, 3.9486, 3.8968, 3.0431, 2.8968, 2.3341, 2.2959]),
 tensor([1086, 1368,  875,  986, 1631,  539, 1709, 1921,  795]))
```

```
show_doc(ClassificationInterpretation.plot_confusion_matrix)
```

plot_confusion_matrix

> plot_confusion_matrix(normalize:bool=False, title:str='Confusion matrix', cmap:Any='Blues', figsize:tuple=None)

Plot the confusion matrix. [source]

```
interp.plot_confusion_matrix()
```

```
show_doc(ClassificationInterpretation.confusion_matrix)
```

confusion_matrix

> confusion_matrix()

Confusion matrix as an `np.ndarray`. [source]

```
interp.confusion_matrix()
```

```
array([[ 997,   13],
       [  15, 1013]])
```

Undocumented Methods - Methods moved below this line will intentionally be hidden

```
show_doc(ClassificationInterpretation.from_learner)
```

from_learner

> from_learner(learn:Learner, loss_class:type='CrossEntropyLoss', sigmoid:bool=True)

Factory method to create from a Learner. [source]

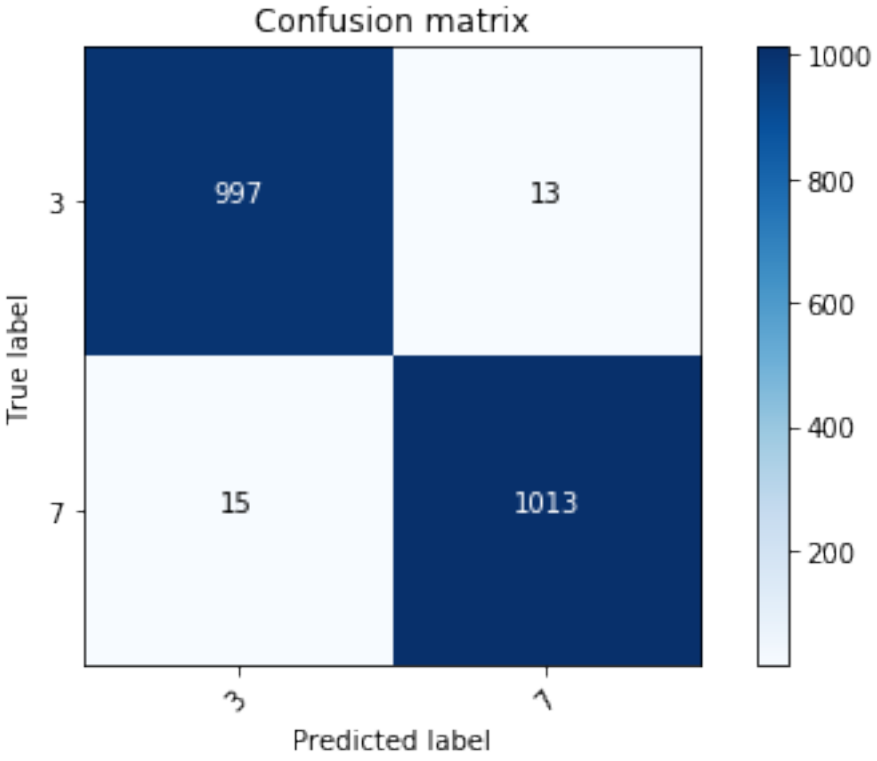

Figure 24: png

Image transforms

```
from fastai.gen_doc.nbdoc import *
from fastai.vision import *
from fastai import *
```

fastai provides a complete image transformation library written from scratch in PyTorch. Although the main purpose of the library is for data augmentation when training computer vision models, you can also use it for more general image transformation purposes. Before we get in to the detail of the full API, we'll look at a quick overview of the data augmentation pieces that you'll almost certainly need to use.

Data augmentation

Data augmentation is perhaps the most important regularization technique when training a model for Computer Vision: instead of feeding the model with the same pictures every time, we do small random transformations (a bit of rotation, zoom, translation, etc…) that don't change what's inside the image (for the human eye) but change its pixel values. Models trained with data augmentation will then generalize better.

To get a set of transforms with default values that work pretty well in a wide range of tasks, it's often easiest to use `get_transforms`. Depending on the nature of the images in your data, you may want to adjust a few arguments, the most important being:

- `do_flip`: if True the image is randomly flipped (default beheavior)
- `flip_vert`: limit the flips to horizontal flips (when False) or to horizontal and vertical flips as well as 90-degrees rotations (when True)

`get_transforms` returns a tuple of two list of transforms: one for the training set and one for the validation set (we don't want to modify the pictures in the validation set, so the second list of transforms is limited to resizing the pictures). This can be then passed directly to define a `DataBunch` object (see below) which is then associated with a model to begin training.

Note that the defaults got `get_transforms` are generally pretty good for regular photos - although here we'll add a bit of extra rotation so it's easier to see the differences.

```
tfms = get_transforms(max_rotate=25)
len(tfms)
```

```
2
```

We first define here a function to return a new image, since transformation functions modify their inputs. We also define a little helper function `plots_f`

to let us output a grid of transformed images based on a function - the details of this function aren't important here.

```
def get_ex(): return open_image('imgs/cat_example.jpg')

def plots_f(rows, cols, width, height, **kwargs):
   [apply_tfms(tfms[0], get_ex(), **kwargs).show(ax=ax) for i,ax in enumerate(plt.subplots(
        rows,cols,figsize=(width,height))[1].flatten())]
```

If we want to have a look at what this transforms actually do, we need to use the `apply_tfms` function. It will be in charge of picking the values of the random parameters and doing the transformation to the `Image` object. This function has multiple arguments you can customize (see its documentation for details), we will highlight here the most useful. The first one we'll need to set, especially if our images are of different shapes, is the target `size`. It will ensure all the images are cropped or padded to the same size so we can then collate them into batches.

```
plots_f(2, 4, 12, 6, size=224)
```

Figure 25: png

Note that the target `size` can be a rectangle if you specify a tuple of int (height by width).

```
plots_f(2, 4, 12, 8, size=(300,200))
```

The second argument that can be customized is how we treat missing pixels: when applying transforms (like a rotation), some of the pixels inside the square won't have values from the image. We can decide to have them: - black (`padding_mode`='zeros') - the value of the one at the nearest border (`padding_mode`='border') - the value of the one symmetric according to the nearest borer (`padding_mode`='reflect')

Figure 26: png

The last one is the default behavior, here is what the two other look like:

```
plots_f(2, 4, 12, 6, size=224, padding_mode='zeros')
```

```
plots_f(2, 4, 12, 6, size=224, padding_mode='border')
```

The third argument that might be useful to change is `do_crop`. Images are often rectangles of different ratios, so to get them to the target `size`, we can either take a random crop from the result of our transforms (which is the default behavior), or add padding on the side that needs to get bigger.

```
plots_f(2, 4, 12, 6, size=224, do_crop=False, padding_mode='zeros')
```

Data augmentation details

If you want to quickly get a set of random transforms that have proved to work well in a wide range of tasks, you should use the `get_transforms` function. The most important parameters to adjust are *do_flip* and *flip_vert*, depending on the type of images you have.

```
show_doc(get_transforms,  arg_comments={
   'do_flip': 'if True, a random flip is applied with probability 0.5',
   'flip_vert': 'requires do_flip=True. If True, the image can be flipped vertically or rotated
   'max_rotate': 'if not None, a random rotation between -max\_rotate and max\_rotate degrees is
```

Figure 27: png

Figure 28: png

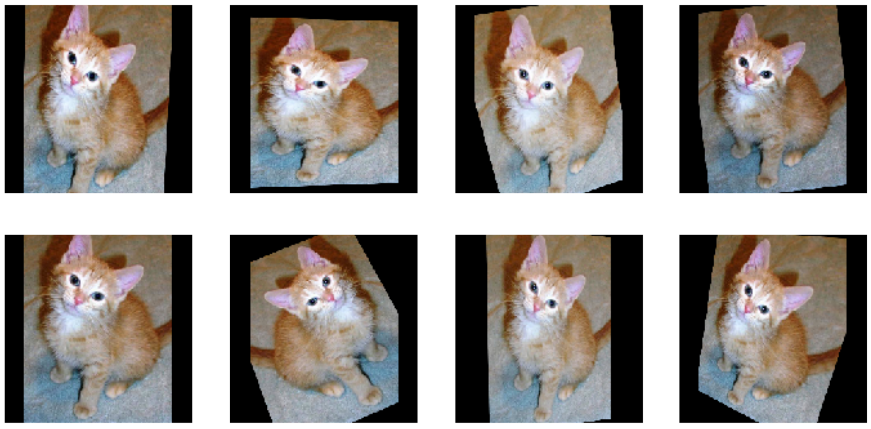

Figure 29: png

```
    'max_zoom': 'if not 1. or less, a random zoom betweem 1. and max\_zoom is applied with probabi
    'max_lighting': 'if not None, a random lightning and contrast change controlled by max\_light
    'max_warp': 'if not None, a random symmetric warp of magnitude between -max\_warp and maw\_wa
    'p_affine': 'the probability that each affine transform and symmetric warp is applied',
    'p_lighting': 'the probability that each lighting transform is applied',
    'xtra_tfms': 'a list of additional transforms you would like to be applied'
})
```

`get_transforms`

> `get_transforms(do_flip:bool=True, flip_vert:bool=False, max_rotate:float=10.0, max_zoom:float=1.1, max_lighting:float=0.2, max_warp:float=0.2, p_affine:float=0.75, p_lighting:float=0.75, xtra_tfms:float=None) -> Collection[Transform]`

Utility func to easily create a list of flip, rotate, `zoom`, warp, lighting transforms.

- *do_flip*: if True, a random flip is applied with probability 0.5
- *flip_vert*: requires do_flip=True. If True, the image can be flipped vertically or rotated of 90 degrees, otherwise only an horizontal flip is applied
- *max_rotate*: if not None, a random rotation between -max_rotate and max_rotate degrees is applied with probability p_affine
- *max_zoom*: if not 1. or less, a random zoom betweem 1. and max_zoom is applied with probability p_affine
- *max_lighting*: if not None, a random lightning and contrast change controlled by max_lighting is applied with probability p_lighting
- *max_warp*: if not None, a random symmetric warp of magnitude between -max_warp and maw_warp is applied with probability p_affine

- *p_affine*: the probability that each affine transform and symmetric warp is applied
- *p_lighting*: the probability that each lighting transform is applied
- *xtra_tfms*: a list of additional transforms you would like to be applied [source]

This function returns a tuple of two list of transforms, one for the training set and the other for the validation set (which is limited to a center crop by default.

```
tfms = get_transforms(max_rotate=25); len(tfms)
```

```
2
```

Let's see how `get_transforms` changes this little kitten now.

```
plots_f(2, 4, 12, 6, size=224)
```

Figure 30: png

Another useful function that gives basic transforms is:

```
show_doc(zoom_crop, arg_comments={
    'scale': 'Ratio to which zoom the image',
   'do_rand': "If true, transform is randomized, otherwise it's a `zoom` of `scale` and a center
    'p': 'Probability to apply the zoom'
})
```

zoom_crop

zoom_crop(scale:float, do_rand:bool=False, p:float=1.0)

Randomly zoom and/or crop.

- *scale*: Ratio to which zoom the image

- *do_rand*: If true, transform is randomized, otherwise it's a `zoom` of `scale` and a center crop
- *p*: Probability to apply the zoom [source]

`scale` should be a given float if `do_rand` is false, otherwise it can be a range of floats (and the zoom will have a random value inbetween). Again, here is a sense of what this can give us.

```
tfms = zoom_crop(scale=(0.75,2), do_rand=True)
plots_f(2, 4, 12, 6, size=224)
```

Figure 31: png

```
show_doc(rand_resize_crop, ignore_warn=True, arg_comments={
    'size': 'Final size of the image',
    'max_scale': 'Zooms the image to a random scale up to this',
   'ratios': 'Range of ratios in which a new one will be randomly picked'
})
```

`rand_resize_crop`

> `rand_resize_crop`(`size`:`int`, `max_scale`:`float`=`2.0`, `ratios`:`Point`=`(0.75, 1.33)`)

Randomly resize and crop the image to a ratio in `ratios` after a zoom of `max_scale`.

- *size*: Final size of the image
- *max_scale*: Zooms the image to a random scale up to this
- *ratios*: Range of ratios in which a new one will be randomly picked [source]

This transform is an implementation of the main approach used for nearly all winning Imagenet entries since 2013, based on Andrew Howard's Some Improvements on Deep Convolutional Neural Network Based Image Classification. It determines a new width and height of the image after the random scale and squish to the new ratio are applied. Those are switched with probability 0.5, then we return the part of the image with the width and height computed centered in `row_pct`, `col_pct` if width and height are both less than the corresponding size of the image, otherwise we try again with new random parameters.

```
tfms = [rand_resize_crop(224)]
plots_f(2, 4, 12, 6, size=224)
```

Figure 32: png

Randomness

The functions that define each transform, like `rotate`or `flip_lr` are deterministic. The fastai library will then randomize them in two different ways: - each transform can be defined with an argument named `p` representing the probability for it to be applied - each argument that is type-annoted with a random function (like `uniform` or rand_int) can be replaced by a tuple of arguments accepted by this function, and on each call of the transform, the argument that is passed inside the function will be picked randomly using that random function.

If we look at the function `rotate` for instance, we see it had an argument `degrees` that is type-annotated as uniform.

First level of randomness: We can define a transform using `rotate` with `degrees` fixed to a value, but by passing an argument `p`. The rotation will then be executed with a probability of `p` but always with the same value of `degrees`.

```
tfm = [rotate(degrees=30, p=0.5)]
fig, axs = plt.subplots(1,5,figsize=(12,4))
for ax in axs:
    img = apply_tfms(tfm, get_ex())
    title = 'Done' if tfm[0].do_run else 'Not done'
    img.show(ax=ax, title=title)
```

Figure 33: png

Second level of randomness: We can define a transform using `rotate` with `degrees` defined as a range, without an argument `p`. The rotation will then always be executed with a random value picked uniformly between the two floats we put in `degrees`.

```
tfm = [rotate(degrees=(-30,30))]
fig, axs = plt.subplots(1,5,figsize=(12,4))
for ax in axs:
    img = apply_tfms(tfm, get_ex())
    title = f"deg={tfm[0].resolved['degrees']:.1f}"
    img.show(ax=ax, title=title)
```

Figure 34: png

All combined: We can define a transform using `rotate` with `degrees` defined as a range, and an argument `p`. The rotation will then always be executed with a probability `p` and a random value picked uniformly between the two floats we put in `degrees`.

```
tfm = [rotate(degrees=(-30,30), p=0.75)]
```

```
fig, axs = plt.subplots(1,5,figsize=(12,4))
for ax in axs:
    img = apply_tfms(tfm, get_ex())
   title = f"Done, deg={tfm[0].resolved['degrees']:.1f}" if tfm[0].do_run else f'Not done'
    img.show(ax=ax, title=title)
```

Figure 35: png

List of transforms

Here is the list of all the deterministic functions on which the transforms are built. As explained before, each of those can have a probability `p` of being executed, and any time an argument is type-annotated with a random function, it's possible to randomize it via that function.

```
show_doc(brightness)
```

`brightness`

> `brightness(x, change:uniform) -> Image :: TfmLighting`

Apply `change` in brightness of image `x`. [source]

This transform adjusts the brightness of the image depending of the value in `change`. A `change` of 0 will transform the image in black and a `change` of 1 will transform the image to white. 0.5 doesn't do anything.

```
fig, axs = plt.subplots(1,5,figsize=(12,4))
for change, ax in zip(np.linspace(0.1,0.9,5), axs):
   brightness(get_ex(), change).show(ax=ax, title=f'change={change:.1f}')
```

```
show_doc(contrast)
```

`contrast`

> `contrast(x, scale:log_uniform) -> Image :: TfmLighting`

Figure 36: png

Apply `scale` to contrast of image `x`. [source]

This adjusts the contrast depending of the value in `scale`. A `scale` of 0 will transform the image in grey and a very high `scale` will transform the picture in super-contrast. 1. doesn't do anything.

```
fig, axs = plt.subplots(1,5,figsize=(12,4))
for scale, ax in zip(np.exp(np.linspace(log(0.5),log(2),5)), axs):
   contrast(get_ex(), scale).show(ax=ax, title=f'scale={scale:.2f}')
```

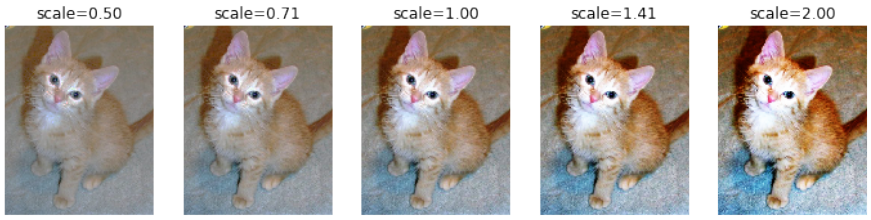

Figure 37: png

```
show_doc(crop)
```

`crop`

> `crop(x, size, row_pct:uniform=0.5, col_pct:uniform=0.5) -> Image :: TfmPixel`

Crop `x` to `size` pixels. `row_pct`,`col_pct` select focal point of crop. [source]

This transform takes a crop of the image to return one of the given size. The position is given by `(col_pct, row_pct)`, with `col_pct` and `row_pct` being normalized between 0. and 1.

```
fig, axs = plt.subplots(1,5,figsize=(12,4))
for center, ax in zip([[0.,0.], [0.,1.],[0.5,0.5],[1.,0.], [1.,1.]], axs):
   crop(get_ex(), 300, *center).show(ax=ax, title=f'center=({center[0]}, {center[1]})')
```

Figure 38: png

```
show_doc(crop_pad, ignore_warn=True, arg_comments={
    'x': 'Image to transform',
   'size': "Size of the crop, if it's an int, the crop will be square",
   'padding_mode': "How to pad the output image ('zeros', 'border' or 'reflection')",
   'row_pct': 'Between 0. and 1., position of the center on the y axis (0. is top, 1. is bottom, 0
   'col_pct': 'Between 0. and 1., position of the center on the x axis (0. is left, 1. is right, 0
})
```

crop_pad

> `crop_pad`(x, `size`, `padding_mode`='`reflection`', `row_pct`:`uniform`=0.5, `col_pct`:`uniform`=0.5) -> `Image` :: `TfmCrop`

Crop and pad tfm - `row_pct`,`col_pct` sets focal point.

- *x*: Image to transform
- *size*: Size of the crop, if it's an int, the crop will be square
- *padding_mode*: How to pad the output image ('zeros', 'border' or 'reflection')
- *row_pct*: Between 0. and 1., position of the center on the y axis (0. is top, 1. is bottom, 0.5 is center)
- *col_pct*: Between 0. and 1., position of the center on the x axis (0. is left, 1. is right, 0.5 is center) [source]

This works like `crop` but if the target size is bigger than the size of the image (on one or the other dimension), padding is applied according to `padding_mode` (see `pad` for an example of all the options) and the position of center is ignored on that dimension.

```
fig, axs = plt.subplots(1,5,figsize=(12,4))
for size, ax in zip(np.linspace(200,600,5), axs):
   crop_pad(get_ex(), int(size), 'zeros', 0.,0.).show(ax=ax, title=f'size = {int(size)}')
```

```
show_doc(dihedral)
```

dihedral

Figure 39: png

`dihedral`(`x`, `k`:`partial(<function uniform_int at 0x7fb18cba7378>, 0, 8)`) -> `Image` :: `TfmPixel`

Randomly flip `x` image based on `k`. [source]

This transform applies one of all the transformations possible of the image by combining a flip (horizontal or vertical) and a rotation of a multiple of 90 degrees.

```
fig, axs = plt.subplots(2,4,figsize=(12,8))
for k, ax in enumerate(axs.flatten()):
    dihedral(get_ex(), k).show(ax=ax, title=f'k={k}')
plt.tight_layout()
```

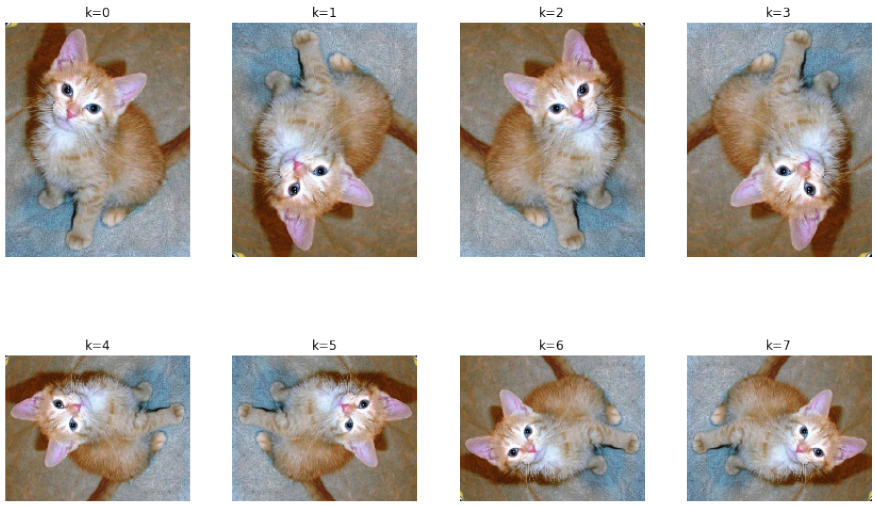

Figure 40: png

```
show_doc(flip_lr)
```

`flip_lr`

`flip_lr`(`x`) -> `Image` :: `TfmPixel` [source]

This transform horizontally flips the image.

```
fig, axs = plt.subplots(1,2,figsize=(6,4))
get_ex().show(ax=axs[0], title=f'no flip')
flip_lr(get_ex()).show(ax=axs[1], title=f'flip')
```

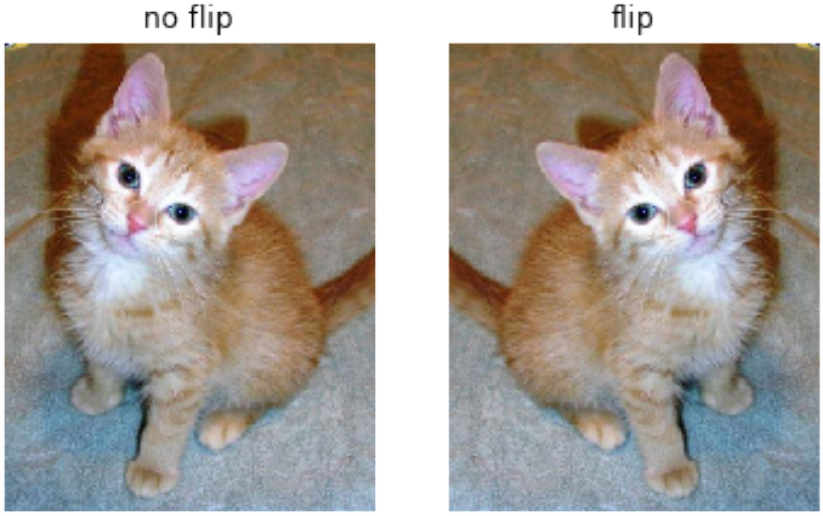

Figure 41: png

```
show_doc(jitter, doc_string=False)
```

jitter

> `jitter(c, img_size, magnitude:uniform) -> Image :: TfmCoord` [source]

This transform changes the pixels of the image by randomly replacing them with pixels from the neighborhood (how far we go is controlled by the value of `magnitude`).

```
fig, axs = plt.subplots(1,5,figsize=(12,4))
for magnitude, ax in zip(np.linspace(-0.05,0.05,5), axs):
    tfm = jitter(magnitude=magnitude)
   get_ex().jitter(magnitude).show(ax=ax, title=f'magnitude={magnitude:.2f}')
```

```
show_doc(pad)
```

pad

Figure 42: png

`pad(x, padding, mode='reflection') -> Image :: TfmPixel`

Pad `x` with `padding` pixels. `mode` fills in space ('zeros','reflection','border'). [source]

Pads the image by adding `padding` pixel on each side of the picture accordin to `mode`: - `mode` = zeros pads with zeros, - `mode` = border repeats the pixels at the border. - `mode` = reflection pads by taking the pixels symmetric to the border.

```
fig, axs = plt.subplots(1,3,figsize=(12,4))
for mode, ax in zip(['zeros', 'border', 'reflection'], axs):
    pad(get_ex(), 50, mode).show(ax=ax, title=f'mode={mode}')
```

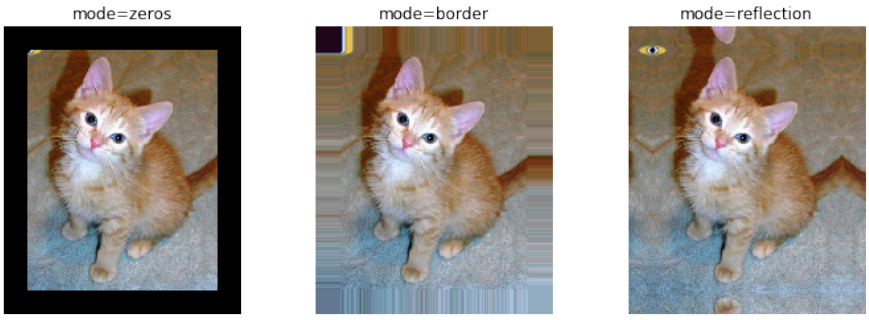

Figure 43: png

```
show_doc(perspective_warp)
```

`perspective_warp`

`perspective_warp(c, img_size, magnitude:partial(<function uniform at 0x7fb18cba71e0>, size=8)=0) -> Image :: TfmCoord`

Apply warp of `magnitude` to `c`. [source]

Perspective wrapping is a deformation of the image as it was seen in a different plane of the 3D-plane. The new plane is determined by telling where we want each of the four corners of the image (from -1 to 1, -1 being left/top, 1 being right/bottom).

```
fig, axs = plt.subplots(2,4,figsize=(12,8))
for i, ax in enumerate(axs.flatten()):
    magnitudes = torch.tensor(np.zeros(8))
    magnitudes[i] = 0.5
   perspective_warp(get_ex(), magnitudes).show(ax=ax, title=f'coord {i}')
```

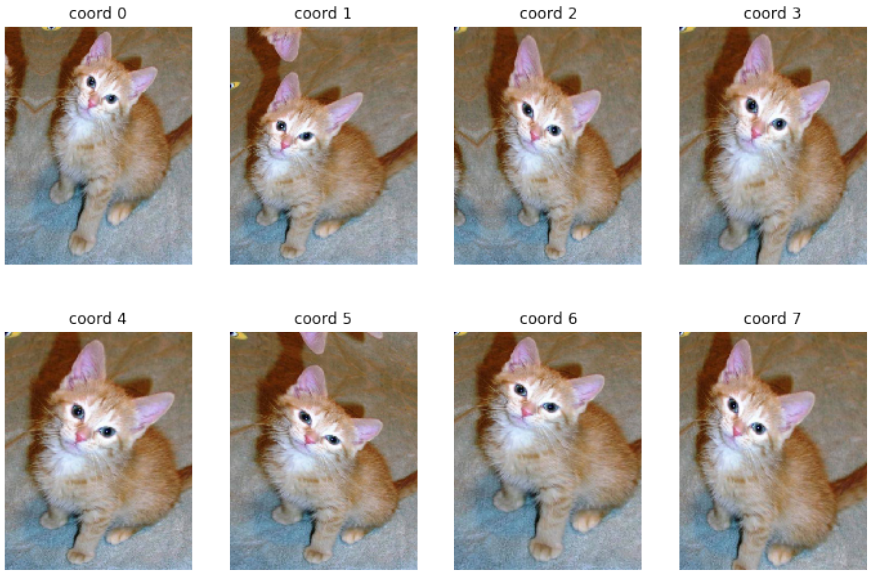

Figure 44: png

```
show_doc(rotate)
```

rotate

`rotate(degrees:uniform) -> Image :: TfmAffine`

Rotate image by `degrees`. [source]

```
fig, axs = plt.subplots(1,5,figsize=(12,4))
for deg, ax in zip(np.linspace(-60,60,5), axs):
   get_ex().rotate(degrees=deg).show(ax=ax, title=f'degrees={deg}')
```

```
show_doc(skew)
```

Figure 45: png

skew

> `skew(c, img_size, direction:uniform_int, magnitude:uniform=0)`
> `-> Image :: TfmCoord`

Skew `c` field with random `direction` and `magnitude`. [source]

```
fig, axs = plt.subplots(2,4,figsize=(12,8))
for i, ax in enumerate(axs.flatten()):
    get_ex().skew(i, 0.2).show(ax=ax, title=f'direction={i}')
```

Figure 46: png

```
show_doc(squish)
```

squish

> squish(scale:uniform=1.0, row_pct:uniform=0.5, col_pct:uniform=0.5) -> Image :: TfmAffine

Squish image by `scale`. `row_pct`,`col_pct` select focal point of zoom. [source]

```
fig, axs = plt.subplots(1,5,figsize=(12,4))
for scale, ax in zip(np.linspace(0.66,1.33,5), axs):
   get_ex().squish(scale=scale).show(ax=ax, title=f'scale={scale:.2f}')
```

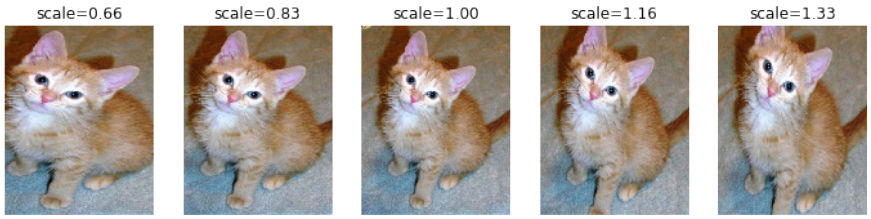

Figure 47: png

```
show_doc(symmetric_warp, doc_string=False)
```

symmetric_warp

> symmetric_warp(c, img_size, magnitude:partial(<function uniform at 0x7fb18cba71e0>, size=4)=0) -> Image :: TfmCoord [source]

Apply the four tilts at the same time, each with a strength given in the vector `magnitude`. See `tilt` just below for the effect of each individual tilt.

```
tfm = symmetric_warp(magnitude=(-0.2,0.2))
_, axs = plt.subplots(2,4,figsize=(12,6))
for ax in axs.flatten():
    img = apply_tfms(tfm, get_ex(), padding_mode='zeros')
    img.show(ax=ax)
```

```
show_doc(tilt, doc_string=False)
```

tilt

> tilt(c, img_size, direction:uniform_int, magnitude:uniform=0) -> Image :: TfmCoord [source]

Tilts `c` in the `direction` given (0: left, 1: right, 2: top, 3: bottom) with a certain `magnitude`. A positive number is a tilt forward (toward the person looking at the picture), a negative number a tilt backward.

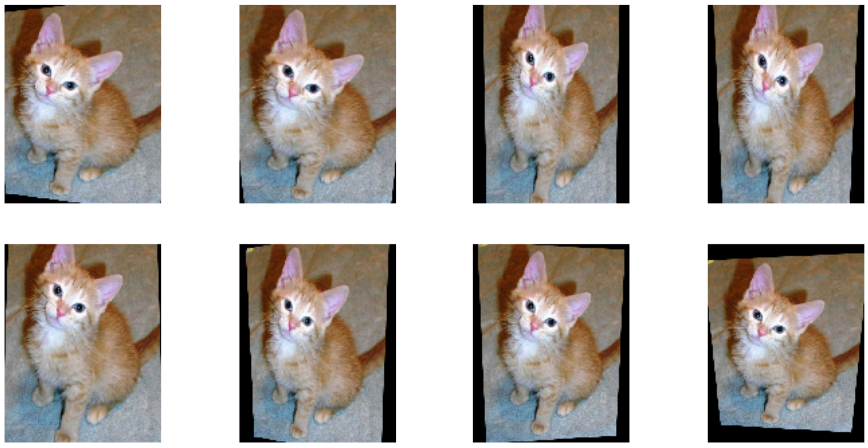

Figure 48: png

```
fig, axs = plt.subplots(2,4,figsize=(12,8))
for i in range(4):
   get_ex().tilt(i, 0.4).show(ax=axs[0,i], title=f'direction={i}, fwd')
   get_ex().tilt(i, -0.4).show(ax=axs[1,i], title=f'direction={i}, bwd')
```

```
show_doc(zoom)
```

zoom

> zoom(scale:uniform=1.0, row_pct:uniform=0.5, col_pct:uniform=0.5)
> -> Image :: TfmAffine

Zoom image by `scale`. `row_pct`,`col_pct` select focal point of zoom. [source]

```
fig, axs = plt.subplots(1,5,figsize=(12,4))
for scale, ax in zip(np.linspace(1., 1.5,5), axs):
   get_ex().squish(scale=scale).show(ax=ax, title=f'scale={scale:.2f}')
```

Convenience functions

These functions simplify creating random versions of `crop_pad` and `zoom`.

```
show_doc(rand_crop)
```

rand_crop

> rand_crop(args, kwargs)

Figure 49: png

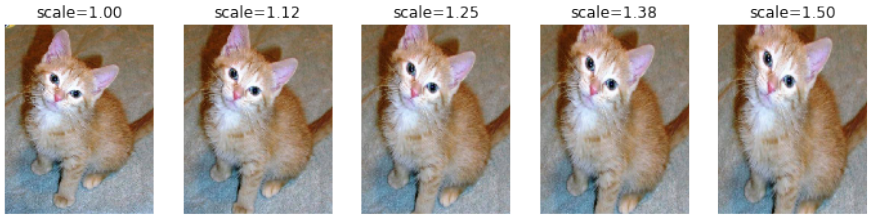

Figure 50: png

Randomized version of `crop_pad`. [source]

```
tfm = rand_crop()
_, axs = plt.subplots(2,4,figsize=(12,6))
for ax in axs.flatten():
    img = apply_tfms(tfm, get_ex(), size=224)
    img.show(ax=ax)
```

Figure 51: png

```
show_doc(rand_zoom)
```

rand_zoom

rand_zoom(args, kwargs)

Randomized version of `zoom`. [source]

```
tfm = rand_zoom(scale=(1.,1.5))
_, axs = plt.subplots(2,4,figsize=(12,6))
for ax in axs.flatten():
    img = apply_tfms(tfm, get_ex())
    img.show(ax=ax)
```

The fastai Image classes

```
from fastai.gen_doc.nbdoc import *
from fastai.vision import *
```

The fastai library is built such that the pictures loaded are wrapped in an `Image`. This `Image` contains the array of pixels associated to the picture, but

Figure 52: png

also has a lot of built-in functions that will help the fastai library to process transformations applied to the corresponding image. There are also sub-classes for special types of image-like objects:

- `ImageMask` for segmentation masks
- `ImageBBox` for bounding boxes

See the following sections for documentation of all the details of these classes. But first, let's have a quick look at the main functionality you'll need to know about.

Opening an image and converting to an `Image` object is easily done by using the `open_image` function:

```
img = open_image('imgs/cat_example.jpg')
img
```

To look at the picture that this `Image` contains, you can also use its `show` method. It will show a resized version and has more options to customize the display.

```
img.show()
```

This `show` method can take a few arguments (see the documentation of `show_image` for details) but the two we will use the most in this documentation are: - `ax` which is the matplolib.pyplot axes on which we want to show the image - `title` which is an optional title we can give to the image.

```
_,axs = plt.subplots(1,4,figsize=(12,4))
for i,ax in enumerate(axs): img.show(ax=ax, title=f'Copy {i+1}')
```

If you're interested in the tensor of pixels, it's stored in the fastai.vision.data

Figure 53: png

Figure 54: png

Figure 55: png

attribute of an `Image`.

```
img.data.shape
```

```
torch.Size([3, 500, 394])
```

The Image classes

`Image` is the class that wraps every picture in the fastai library. It is subclassed to create `ImageMask` and `ImageBBox` when dealing with segmentation and object detection tasks.

```
show_doc(Image, arg_comments={
    'px': 'pixel tensor of the underlying image'
})
```

class Image

> `Image(px:Tensor) :: ImageBase`

Support applying transforms to image data.

- *px*: pixel tensor of the underlying image [source]

Most of the functions of the `Image` class deal with the internal pipeline of transforms, so they are only shown at the end of this page. The easiest way to create one is through the function `open_image`.

```
show_doc(open_image)
```

open_image

> `open_image(fn:PathOrStr) -> Image`

Return `Image` object created from image in file `fn`. [source]

```
img = open_image('imgs/cat_example.jpg')
img
```

In a Jupyter Notebook, the representation of an `Image` is its underlying picture (shown to its full size). On top of containing the tensor of pixels of the Image (and automatically doing the conversion after decoding the image), this class contains various methods for the implementation of transforms. The `Image.show` method also allows to pass more arguments, under the hood, it just calls:

```
show_doc(show_image, arg_comments ={
    'x': '`Image` to show',
   'y': 'Potential target to be superposed on the same graph (mask, bounding box, points)',
    'ax': 'matplotlib.pyplot axes on which show the image',
```

Figure 56: png

```
    'figsize': 'Size of the figure',
    'alpha': 'Transparency to apply to y (if applicable)',
    'title': 'Title to display on top of the graph',
    'hide_axis': 'If True, the axis of the graph are hidden',
    'cmap': 'Color map to use'
})
```

show_image

> show_image(x:Image, y:Image=None, ax:Axes=None, figsize:tuple=(3, 3), alpha:float=0.5, title:Optional[str]=None, hide_axis:bool=True, cmap:str='viridis')

Plot tensor `x` using matplotlib axis `ax`. `figsize`,`axis`,`title`,`cmap` and `alpha` pass to `ax.imshow`.

- *x*: `Image` to show
- *y*: Potential target to be superposed on the same graph (mask, bounding box, points)
- *ax*: matplotlib.pyplot axes on which show the image
- *figsize*: Size of the figure
- *alpha*: Transparency to apply to y (if applicable)
- *title*: Title to display on top of the graph
- *hide_axis*: If True, the axis of the graph are hidden
- *cmap*: Color map to use [source]

This allows us to completely customize the display of an `Image`. We'll see examples of the `y` functionality below with segmentation and bounding boxes tasks, for now here is an example using the other features. Note that the beahvior of `show_image` and `Image.show` are the same.

```
show_image(img, figsize=(2,1), title='Small kitten')
```

Figure 57: png

```
img.show(figsize=(10,5), title='Big kitten')
```

An `Image` object also has a few attributes that can be useful: - `Image.data` gives you the underlying tensor of pixel - `Image.shape` gives you the size of that

Figure 58: png

tensor (channels x height x width) - `Image.size` gives you the size of the image (height x width)

```
img.data, img.shape, img.size
```

```
(tensor([[[0.1294, 0.0863, 0.0392,  ..., 0.4706, 0.4941, 0.4863],
          [0.0745, 0.0471, 0.0392,  ..., 0.4706, 0.4863, 0.4863],
          [0.0706, 0.0510, 0.0627,  ..., 0.4784, 0.4784, 0.4784],
          ...,
          [0.3059, 0.3647, 0.3686,  ..., 0.5412, 0.5725, 0.5725],
          [0.3294, 0.4000, 0.4039,  ..., 0.5882, 0.5765, 0.5765],
          [0.3843, 0.4627, 0.4667,  ..., 0.6471, 0.5725, 0.5725]],

         [[0.0235, 0.0000, 0.0000,  ..., 0.3490, 0.3686, 0.3725],
          [0.0000, 0.0000, 0.0000,  ..., 0.3569, 0.3725, 0.3725],
          [0.0000, 0.0000, 0.0157,  ..., 0.3647, 0.3686, 0.3686],
          ...,
          [0.3882, 0.4588, 0.4627,  ..., 0.6471, 0.6784, 0.6784],
          [0.4118, 0.4941, 0.4980,  ..., 0.6941, 0.6824, 0.6824],
          [0.4667, 0.5569, 0.5608,  ..., 0.7529, 0.6784, 0.6784]],

         [[0.0980, 0.0863, 0.1059,  ..., 0.1765, 0.2078, 0.2078],
          [0.0706, 0.0745, 0.1137,  ..., 0.1922, 0.2078, 0.2157],
          [0.1020, 0.1176, 0.1647,  ..., 0.2078, 0.2118, 0.2157],
          ...,
          [0.4941, 0.5608, 0.5647,  ..., 0.7294, 0.7608, 0.7529],
          [0.5176, 0.5961, 0.6000,  ..., 0.7765, 0.7647, 0.7569],
          [0.5725, 0.6588, 0.6627,  ..., 0.8353, 0.7608, 0.7529]]]),
 torch.Size([3, 500, 394]),
 torch.Size([500, 394]))
```

For a segmentation task, the target is usually a mask. The fastai library represents it as an `ImageMask` object.

```
show_doc(ImageMask, arg_comments={
    'px': 'pixel tensor of the underlying mask'
})
```

class ImageMask

> `ImageMask(px:Tensor) :: Image`

Class for image segmentation target.

- *px*: pixel tensor of the underlying mask [source]

To easily open a mask, the function `open_mask` plays the same role as `open_image`:

```
show_doc(open_mask)
```

open_mask

> open_mask(fn:PathOrStr) -> ImageMask

Return `ImageMask` object create from mask in file `fn`. [source]

```
open_mask('imgs/mask_example.png')
```

Figure 59: png

An `ImageMask` object has the same properties as an `Image`. The only difference is that when applying the transformations to an `ImageMask`, it will ignore the functions that deal with lighting and keep values of 0 and 1. As explained earlier, it's easy to show the segmentation mask over the associated `Image` by using the `y` argument of `show_image`.

```
img = open_image('imgs/car_example.jpg')
mask = open_mask('imgs/mask_example.png')
_,axs = plt.subplots(1,3, figsize=(8,4))
img.show(ax=axs[0], title='no mask')
img.show(ax=axs[1], y=mask, title='masked')
mask.show(ax=axs[2], title='mask only')
```

For an objection detection task, the target is a bounding box containg the picture.

```
show_doc(ImageBBox, arg_comments={
    'px': 'pixel tensor of the underlying mask'
})
```

class ImageBBox

> ImageBBox(px:Tensor) :: ImageMask

Figure 60: png

Image class for bbox-style annotations.

- *px*: pixel tensor of the underlying mask [source]

Internally, the `ImageBBox` is just an `ImageMask` with a square mask representing the bounding box (or bonding boxes in several channels). This is to deal with data augmentation and might be removed in future developments. To create an `ImageBBox`, we have this helper function that will take a list of bounding boxes, each representing by a list of four numbers representing the coordinates of two corners of the box with the following convention: top, left, bottom, right.

```
show_doc(ImageBBox.create, arg_comments={
   'bboxes': 'list of bboxes (each of those being four integers with the top, left, bottom, right
    'h': 'height of the input image',
    'w': 'width of the input image'
})
```

`create`

> `create(bboxes:Collection[Collection[int]], h:int, w:int) -> ImageBBox`

Create an ImageBBox object from `bboxes`.

- *bboxes*: list of bboxes (each of those being four integers with the top, left, bottom, right convention)
- *h*: height of the input image
- *w*: width of the input image [source]

We need to pass the dimensions of the input image so that `ImageBBox` can internally create an `ImageMask` of the size of the `Image`. Again, the `Image.show` method will display the bouding box on the same image if it's passed as a `y` argument.

```
img = open_image('imgs/car_bbox.jpg')
```

```
bbox = ImageBBox.create([[96, 155, 270, 351]], *img.size)
_,axs = plt.subplots(1,2)
img.show(y=bbox, ax=axs[0], title='Bounding Box')
img.show(y=ImageMask(bbox.px), ax=axs[1], title='Internal mask')
```

Figure 61: png

To help with the conversion of images or to show them, we use these helper functions:

`show_doc(pil2tensor)`

`pil2tensor`

> `pil2tensor(image:ndarray) -> Tensor`

Convert PIL style `image` array to torch style image tensor. [source]

`show_doc(image2np)`

`image2np`

> `image2np(image:Tensor) -> ndarray`

Convert from torch style `image` to numpy/matplotlib style. [source]

`show_doc(bb2hw)`

`bb2hw`

> `bb2hw(a:Collection[int]) -> ndarray`

Convert bounding box points from (width,height,center) to (height,width,top,left). [source]

Applying transforms

All the transforms available for data augmentation in computer vision are defined in the vision.transform module. When we want to apply them to an `Image`, we use this function:

```
show_doc(apply_tfms, arg_comments={
    'tfms': '`Transform` or list of `Transform`',
    'x': '`Image` to apply the `tfms` to',
   'do_resolve': 'if False, the values of random parameters are kept from the last draw',
    'xtra': 'extra arguments to pass to the transforms',
    'size': 'desired target size',
    'mult': 'makes sure the final size is a multiple of mult',
   'do_crop': 'if True, crops the image to the final size, otherwise pads it using `padding_mode`
   'padding_mode': "how to pad the image ('zeros', 'border', 'reflection')"
})
```

`apply_tfms`

> `apply_tfms(tfms:Union[Transform, Collection[Transform]], x:Tensor, do_resolve:bool=True, xtra:Optional[Dict[Transform, dict]]=None, size:Union[int, TensorImageSize, NoneType]=None, mult:int=32, do_crop:bool=True, padding_mode:str='reflection', kwargs:Any) -> Tensor`

Apply all `tfms` to `x` - `do_resolve`: bind random args - `size`, `mult` used to crop/pad.

- *tfms*: `Transform` or list of `Transform`
- *x*: `Image` to apply the `tfms` to
- *do_resolve*: if False, the values of random parameters are kept from the last draw
- *xtra*: extra arguments to pass to the transforms
- *size*: desired target size
- *mult*: makes sure the final size is a multiple of mult
- *do_crop*: if True, crops the image to the final size, otherwise pads it using `padding_mode`
- *padding_mode*: how to pad the image ('zeros', 'border', 'reflection') [source]

Before showing examples, let's take a few moments to comment those arguments a bit more: - `do_resolve` decides if we resolve the random arguments by drawing new numbers or not. The intended use is to have the `tfms` applied to the input `x` with `do_resolve`=True, then, if the target `y` needs to be applied data augmentation (if it's a segmentation mask or bounding box), apply the `tfms` to `y` with `do_resolve`=False. - `mult` default value is very important to make sure your image can pass through most recent CNNs: they divide the size of the

input image by 2 a certain amount of time so both dimensions of your picture you should be mutliples of at least 32. Only change the value of this parameter if you know it will be accepted by your model.

Here are a few helper functions to help us load the examples we saw before.

```
def get_class_ex(): return open_image('imgs/cat_example.jpg')
def get_seg_ex(): return open_image('imgs/car_example.jpg'), open_mask('imgs/mask_example.png
def get_bb_ex():
    img = open_image('imgs/car_bbox.jpg')
    return img, ImageBBox.create([[96, 155, 270, 351]], *img.size)
```

Now lets grab our usual bunch of transforms and see what they do.

```
tfms = get_transforms()
_, axs = plt.subplots(2,4,figsize=(12,6))
for ax in axs.flatten():
    img = apply_tfms(tfms[0], get_class_ex(), size=224)
    img.show(ax=ax)
```

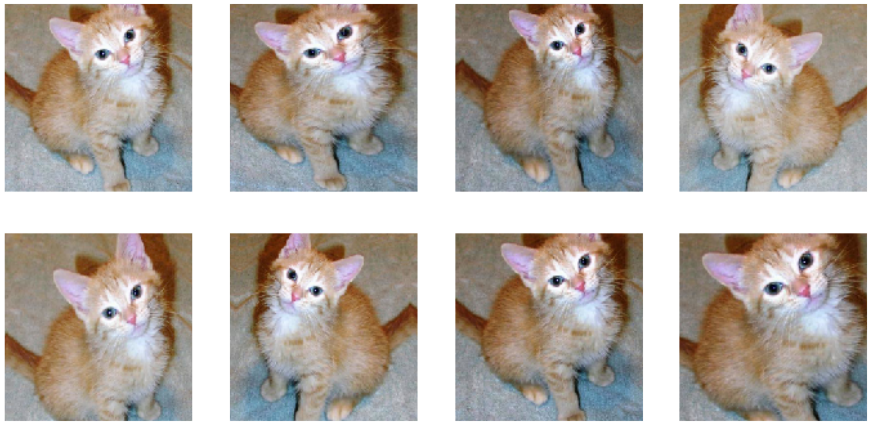

Figure 62: png

Now let's check what it gives for a segmentation task. Note that, as instructed by the documentation of `apply_tfms` we first apply the transforms to the input, then apply them to the target while adding `do_resolve`=False.

```
tfms = get_transforms()
_, axs = plt.subplots(2,4,figsize=(12,6))
for ax in axs.flatten():
    img,mask = get_seg_ex()
    img = apply_tfms(tfms[0], img, size=224)
    mask = apply_tfms(tfms[0], mask, do_resolve=False, size=224)
    img.show(ax=ax, y=mask)
```

Figure 63: png

Internally, each transforms save the values it picked randomly in a dictionary called resolved, which is how they can reuse those numbers for the target.

```
tfms[0][4]
```

```
RandTransform(tfm=TfmAffine (zoom), kwargs={'row_pct': (0, 1), 'col_pct': (0, 1), 'scale': (1.0
```

Now for the bounding box, the `ImageBBox` object will automatically update the coordinates of the two opposite corners in its data attribute.

```
tfms = get_transforms()
_, axs = plt.subplots(2,4,figsize=(12,6))
for ax in axs.flatten():
    img,bbox = get_bb_ex()
    img = apply_tfms(tfms[0], img, size=224)
    bbox = apply_tfms(tfms[0], bbox, do_resolve=False, size=224)
    img.show(ax=ax, y=bbox)
```

Randomness

As explained in the transform module, to indicate to a `Transform` how to randomize an argument, we use a type annotation by a random function. Here is the list of the available functions.

```
show_doc(rand_bool)
```

rand_bool

> `rand_bool(p:float, size:Optional[List[int]]=None) -> BoolOrTensor`

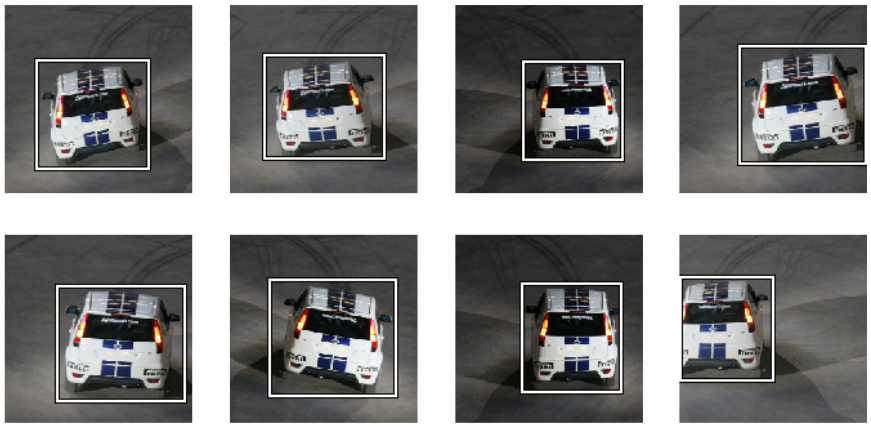

Figure 64: png

Draw 1 or shape=`size` random booleans (True occuring probability `p`). [source]

```
rand_bool(0.5, 8)
```

```
tensor([1, 1, 1, 1, 0, 1, 1, 1], dtype=torch.uint8)
```

```
show_doc(uniform)
```

`uniform`

> `uniform(low:Number, high:Number=None, size:Optional[List[int]]=None) -> FloatOrTensor`

Draw 1 or shape=`size` random floats from uniform dist: min=`low`, max=`high`. [source]

```
uniform(0,1,(8,))
```

```
tensor([0.3415, 0.6459, 0.8519, 0.7157, 0.0412, 0.4517, 0.2181, 0.6656])
```

```
show_doc(uniform_int)
```

`uniform_int`

> `uniform_int(low:int, high:int, size:Optional[List[int]]=None) -> IntOrTensor`

Generate int or tensor `size` of ints between `low` and `high` (included). [source]

```
uniform_int(0,2,(8,))
```

```
tensor([0, 0, 1, 1, 1, 2, 1, 2])
```

```
show_doc(log_uniform, doc_string=False)
```

log_uniform

> `log_uniform(low, high, size:Optional[List[int]]=None) -> FloatOrTensor` [source]

Picks a random number (or tensor `size`) between log(`low`) and log(`high`), then returns its exponential (so that it's between `low` and `high` in the end).

```
log_uniform(0.5,2,(8,))
```

```
tensor([0.5447, 0.8798, 1.3271, 0.9458, 1.2901, 0.7366, 0.9897, 1.4100])
```

Fastai internal pipeline

What does a transform do?

Typically, a data augmentation operation will randomly modify an image input. This operation can apply to pixels (when we modify the contrast or brightness for instance) or to coordinates (when we do a rotation, a zoom or a resize). The operations that apply to pixels can easily be coded in numpy/pytorch, directly on an array/tensor but the ones that modify the coordinates are a bit more tricky.

They usually come in three steps: first we create a grid of coordinates for our picture: this is an array of size `h * w * 2` (`h` for height, `w` for width in the rest of this post) that contains in position i,j two floats representing the position of the pixel (i,j) in the picture. They could simply be the integers i and j, but since most transformations are centered with the center of the picture as origin, they're usually rescaled to go from -1 to 1, (-1,-1) being the top left corner of the picture, (1,1) the bottom right corner (and (0,0) the center), and this can be seen as a regular grid of size h * w. Here is a grid what our grid would look like for a 5px by 5px image.

Then, we apply the transformation to modify this grid of coordinates. For instance, if we want to apply an affine transformation (like a rotation) we will transform each of those vectors `x` of size 2 by `A @ x + b` at every position in the grid. This will give us the new coordinates, as seen here in the case of our previous grid.

There are two problems that arise after the transformation: the first one is that the pixel values won't fall exactly on the grid, and the other is that we can get values that get out of the grid (one of the coordinates is greater than 1 or lower than -1).

To solve the first problem, we use an interpolation. If we forget the rescale for a minute and go back to coordinates being integers, the result of our transfor-

mation gives us float coordinates, and we need to decide, for each (i,j), which pixel value in the original picture we need to take. The most basic interpolation called nearest neighbor would just round the floats and take the nearest integers. If we think in terms of the grid of coordinates (going from -1 to 1), the result of our transformation gives a point that isn't in the grid, and we replace it by its nearest neighbor in the grid.

To be smarter, we can perform a bilinear interpolation. This takes an average of the values of the pixels corresponding to the four points in the grid surrounding the result of our transformation, with weights depending on how close we are to each of those points. This comes at a computational cost though, so this is where we have to be careful.

As for the values that go out of the picture, we treat them by padding it either: - by adding zeros on the side, so the pixels that fall out will be black (zero padding) - by replacing them by the value at the border (border padding) - by mirroring the content of the picture on the other side (reflection padding).

Be smart and efficient

Usually, data augmentation libraries have separated the different operations. So for a resize, we'll go through the three steps above, then if we do a random rotation, we'll go again to do those steps, then for a zoom etc… The fastai library works differently in thsense that it will do all the transformations on the coordinates at the same time, so that we only do those three steps once, especially the last one (the interpolation) that is the most heavy in computation.

The first thing is that we can regroup all affine transforms in just one (since an affine transform composed by an affine transform is another affine transform). This is already done in some other libraries but we pushed the thing one step further though to integrate the resize, the crop and any non-affine transformation of the coordinates in the process. Let's dig in!

- In step 1, when we create the grid, we use the new size we want for our image, so `new_h, new_w` (and not `h, w`). This takes care of the resize operation.
- In step 2, we do only one affine transformation, by multiplying all the affine matrices of the transforms we want to do beforehand (those are 3 by 3 matrices, so it's super fast), then we apply to the coords any non-affine transformation we might want (jitter, perspective wrappin, etc) before…
- Step 2.5: we crop (either center or randomly) the coordinates we want to keep. Crop is easy to do whenever we want, but by doing it just before the interpolation, we don't compute pixel values that won't be used at the end, gaining again a bit of efficiency
- Finally step 3: the final interpolation. Afterward, we can apply on the picture all the tranforms that operate pixel-wise (as said before brightness, contrast for instance) and we're done with data augmentation.

Note that the transforms operating on pixels are applied in two phases: - first the transforms that deal with lighting properties are applied to the logits of the pixels. To only do once the conversion pixels -> logits -> pixels, we group the together, - then we apply the transforms that modify the pixel.

This is why all transforms have an attribute (like `TfmAffine`, `TfmCoord`, `TfmCrop` or `TfmPixel`) so that the fastai library can regroup them and apply them all together at the right step. In terms of implementation:

- _affine_grid is reponsible for creating the grid of coordinates
- _affine_mult is in charge of doing the affine multiplication on that grid
- _grid_sampleis the function that is responsible for the interpolation step

Final result

TODO: add a comparison of speeds.

Also, adding a new transformation almost doesn't hurt performance (since the costly steps are done only once) when with classic data aug implementations, it usually result in a longer training time.

Even in terms of final result, doing only one interpolation gives a better result: if we stack several transforms and do an interpolation on each one, we approximate the true value of our coordinates in some way, which tends to blur a bit the image. By regrouping all the transformations together and only doing this step at the end, we can get something nicer.

Look at how the same rotation then zoom done separately (so with two interpolations)

is blurrier than regrouping the transforms and doing just one interpolation

Transform classes

The basic class that defines transformation in the fastai library is `Transform`.

```
show_doc(Transform, title_level=3,
      alt_doc_string="Create a `Transform` for `func` and assign it a priority `order`.")
```

class Transform

`Transform`(`func`:`Callable`, `order`:`Optional`[`int`]=`None`)

Create a `Transform` for `func` and assign it a priority `order`. [source]

```
show_doc(RandTransform, title_level=3, doc_string=False)
```

class RandTransform

> RandTransform(tfm:Transform, kwargs:dict, p:int=1.0, resolved:dict=<factory>, do_run:bool=True, is_random:bool=True) [source]

Create a `Transform` from func that can be randomized. Each argument of `func` in kwargs is analyzed and if it has a type annotaiton that is a random function, this function will be called to pick a value for it. This value will be stored in the `resolved` dictionary. Following the same idea, `p` is the probability for func to be called and `do_run` will be set to True if it was the cause, False otherwise. Lastly, setting `is_random` to False allows to send specific values for each parameter.

```
show_doc(RandTransform.resolve)
```

resolve

> resolve()

Binds any random variables in the transform. [source]

To handle internally the data augmentation as explained earlier, each `Transform` as a type, so that the fastai library can regoup them together efficiently. There are five types of `Transform` which all work as decorators for a deterministic function.

```
show_doc(TfmAffine, title_level=3, doc_string=False)
```

class TfmAffine

> TfmAffine(func:Callable, order:Optional[int]=None) :: Transform [source]

Decorate `func` to make it an affine transform; `func` should return the 3 by 3 matrix representing the transform. The default `order` is 5 for such transforms.

```
show_doc(TfmCoord, title_level=3, doc_string=False)
```

class TfmCoord

> TfmCoord(func:Callable, order:Optional[int]=None) :: Transform [source]

Decorate `func` to make it a coord transform; `func` should take two mandatory arguments: `c` (the flow of coordinate) and `img_size` (the size of the corresponding image) and return the modified flow of coordinates. The default `order` is 4 for such transforms.

```
show_doc(TfmLighting, title_level=3, doc_string=False)
```

class TfmLighting

> TfmLighting(func:Callable, order:Optional[int]=None) :: Transform [source]

Decorate `func` to make it a lighting transform; `func` takes the logits of the pixel tensor and changes them. The default `order` is 8 for such transforms.

```
show_doc(TfmPixel, title_level=3, doc_string=False)
```

class TfmPixel

> TfmPixel(func:Callable, order:Optional[int]=None) :: Transform [source]

Decorate `func` to make it a pixel transform; `func` takes the pixel tensor and modifies it. The default `order` is 10 for such transforms.

```
show_doc(TfmCrop, title_level=3, doc_string=False)
```

class TfmCrop

> TfmCrop(func:Callable, order:Optional[int]=None) :: TfmPixel [source]

Decorate `func` to make it a crop transform; This is a special case of `TfmPixel` with `order` set to 99.

To help with the conversion to logits for the `TfmLighting`, we use these helper functions:

```
show_doc(logit)
```

logit

> logit(x:Tensor) -> Tensor [source]

Take the element-wise logit of `x`. Logit is the invert function of the sigmoid, defined by log(x/(1-x)).

```
show_doc(logit_)
```

logit_

`logit_(x:Tensor) -> Tensor` [source]

In-place version of `logit`.

Internal Image class

Image is a subclass of ImageBase, which is a shell containing the basic methods necessary for applying data augmentation. Creating an Image object is done by passing a tensor of pixels representing a picture.

```
show_doc(Image, title_level=3, arg_comments={
    'px': 'Array of pixels'
})
```

class Image

`Image(px:Tensor) :: ImageBase`

Support applying transforms to image data.

- *px*: Array of pixels [source]

```
show_doc(Image.affine)
```

affine

`affine(func:AffineFunc, args, kwargs) -> Image`

Equivalent to `image.affine_mat = image.affine_mat @ func()`. [source]

```
show_doc(Image.clone)
```

clone

`clone()`

Mimic the behavior of torch.clone for `Image` objects. [source]

```
show_doc(Image.coord)
```

coord

`coord(func:CoordFunc, args, kwargs) -> Image`

Equivalent to `image.flow = func(image.flow, image.size)`. [source]

```
show_doc(Image.lighting)
```

lighting

`lighting(func:LightingFunc, args:Any, kwargs:Any)`

Equivalent to `image = sigmoid(func(logit(image)))`. [source]

```
show_doc(Image.refresh)
```

refresh

`refresh()`

Apply any logit, flow, or affine transfers that have been sent to the `Image`. [source]

```
show_doc(Image.resize)
```

resize

`resize(size:Union[int, TensorImageSize]) -> Image`

Resize the image to `size`, size can be a single int. [source]

```
show_doc(Image.show, full_name='show')
```

show

`show(ax:Axes=None, y:Image=None, kwargs)` [source]

Send the `Image` to `show_image` with `ax`, `y` and the `kwargs`.

Undocumented Methods - Methods moved below this line will intentionally be hidden

```
show_doc(ImageBase.clone)
```

clone

`clone() -> ImageBase`

Clone this item and its `data`. [source]

```
show_doc(Image.crop_pad)
```

<a id=><lambda>

`<lambda>(x, args, kwargs)` [source]

```
show_doc(Image.contrast)
```

<a id=><lambda>

<lambda>(x, args, kwargs) [source]

```
show_doc(ImageBase.lighting)
```

lighting

lighting(func:LightingFunc, args, kwargs) -> ImageBase [source]

```
show_doc(ImageBase)
```

class ImageBase

ImageBase() :: ItemBase

Image based Dataset items derive from this. Subclass to handle lighting, pixel, etc… [source]

```
show_doc(ImageBase.coord)
```

coord

coord(func:CoordFunc, args, kwargs) -> ImageBase [source]

```
show_doc(Image.brightness)
```

<a id=><lambda>

<lambda>(x, args, kwargs) [source]

```
show_doc(Image.flip_lr)
```

<a id=><lambda>

<lambda>(x, args, kwargs) [source]

```
show_doc(Image.pad)
```

<a id=><lambda>

<lambda>(x, args, kwargs) [source]

```
show_doc(Image.pixel)
```

pixel

pixel(func:LightingFunc, args, kwargs) -> Image

Equivalent to `image.px = func(image.px)`. [source]

```
show_doc(Image.zoom)
```

<a id=><lambda>

<lambda>(x, args, kwargs) [source]

```
show_doc(Image.dihedral)
```

<a id=><lambda>

<lambda>(x, args, kwargs) [source]

```
show_doc(ImageMask.refresh)
```

refresh

refresh()

Apply any logit, flow, or affine transfers that have been sent to the `Image`. [source]

```
show_doc(ImageBase.affine)
```

affine

affine(func:AffineFunc, args, kwargs) -> ImageBase [source]

```
show_doc(Image.jitter)
```

<a id=><lambda>

<lambda>(x, args, kwargs) [source]

```
show_doc(Image.squish)
```

<a id=><lambda>

<lambda>(x, args, kwargs) [source]

```
show_doc(ImageBase.set_sample)
```

set_sample

set_sample(kwargs) -> ImageBase

Set parameters that control how we **grid_sample** the image after transforms are applied. [source]

```
show_doc(ImageBase.pixel)
```

pixel

pixel(func:LightingFunc, args, kwargs) -> ImageBase [source]

```
show_doc(Image.skew)
```

<a id=><lambda>

<lambda>(x, args, kwargs) [source]

```
show_doc(Image.perspective_warp)
```

<a id=><lambda>

<lambda>(x, args, kwargs) [source]

```
show_doc(Image.zoom_squish)
```

<a id=><lambda>

<lambda>(x, args, kwargs) [source]

```
show_doc(Image.crop)
```

<a id=><lambda>

<lambda>(x, args, kwargs) [source]

```
show_doc(Image.tilt)
```

<a id=><lambda>

<lambda>(x, args, kwargs) [source]

```
show_doc(Image.rotate)
```

<a id=><lambda>

<lambda>(x, args, kwargs) [source]

```
show_doc(ImageMask.lighting)
```

`lighting`

> `lighting`(`func`:`LightingFunc`, `args`:`Any`, `kwargs`:`Any`) -> `Image`

Equivalent to `image = sigmoid(func(logit(image)))`. [source]

```
show_doc(Image.symmetric_warp)
```

<a id=><lambda>

> `<lambda>`(`x`, `args`, `kwargs`) [source]

Computer vision data

```
from fastai.gen_doc.nbdoc import *
from fastai.vision import *
from fastai.docs import *
```

This module contains the classes that define datasets handling `Image` objects and their tranformations. As usual, we'll start with a quick overview, before we get in to the detailed API docs.

Quickly get your data ready for training

To get you started as easily as possible, the fastai provides two helper functions to create a `DataBunch` object that you can directly use for training a classifier. To demonstrate them you'll first need to download and untar the file by executing the following cell. This will create a data folder containing a MNIST subset in `data/mnist_sample`.

```
untar_data(MNIST_PATH)
MNIST_PATH
```

```
PosixPath('../data/mnist_sample')
```

The first way to define a `DataBunch` object requires you to put your data in folders this way (the `test` folder is optional):

```
path\
  train\
    clas1\
    clas2\
    ...
    clasn\
  valid\
    clas1\
    clas2\
```

```
    ...
    clasn\
  test\
```

You can then use `image_data_from_folder`:

```
data = image_data_from_folder(MNIST_PATH, ds_tfms=get_transforms(do_flip=False), size=24)
```

Here the datasets will be automatically created from the structure above, and we precise: - the transforms to apply to the images in `ds_tfms` (here with `do_flip`=False because we don't want to flip numbers), - the target `size` of our pictures (here 28).

As all `DataBunch`, data then as a `train_dl` and a `valid_dl` that are `DataLoader`. If you want to have a look at a few images in a batch inside a batch, you can use `show_image_batch`, the `rows` argument being the number of rows and columns in the array.

```
show_image_batch(data.train_dl, data.train_ds.classes, rows=3, figsize=(5,5))
```

The second way to define the data for a classifier requires a structure like this:

```
path\
  train\
  test\
  labels.csv
```

where the labels.csv file defines what the label(s) of each image in the training set. This is the format you will need to use when each image can have multiple labels. It also works with single labels:

```
pd.read_csv(MNIST_PATH/'labels.csv').head()
```

```
<tr style="text-align: right;">
  <th></th>
  <th>name</th>
  <th>label</th>
</tr>
```

```
<tr>
  <th>0</th>
  <td>train/3/7463.png</td>
  <td>0</td>
</tr>
<tr>
  <th>1</th>
  <td>train/3/21102.png</td>
  <td>0</td>
</tr>
<tr>
  <th>2</th>
```

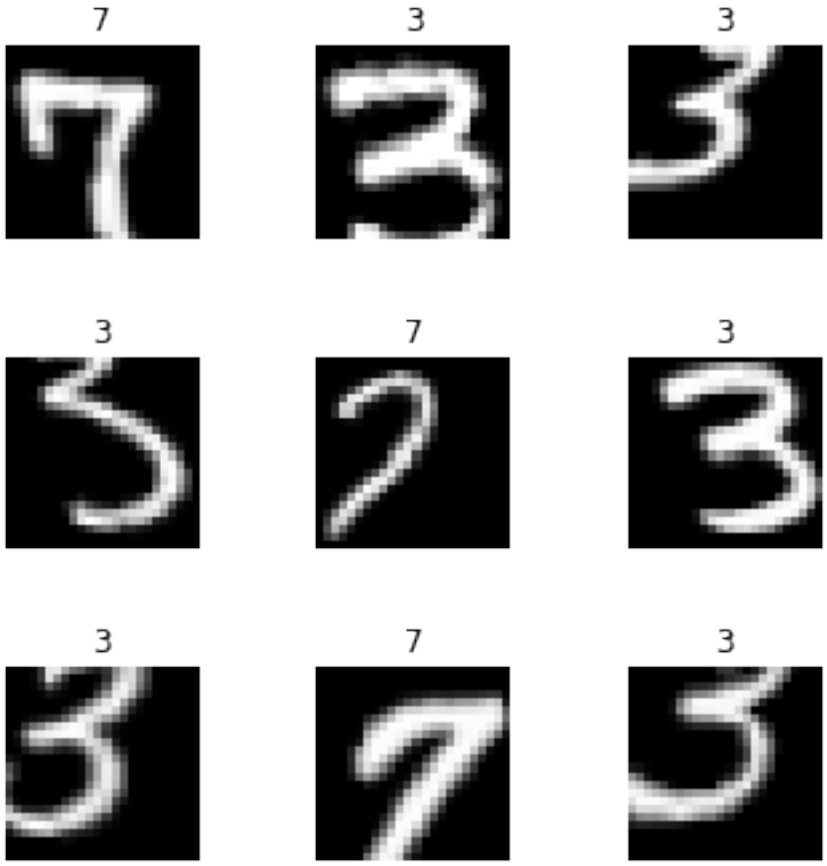

Figure 65: png

```
  <td>train/3/31559.png</td>
  <td>0</td>
</tr>
<tr>
  <th>3</th>
  <td>train/3/46882.png</td>
  <td>0</td>
</tr>
<tr>
  <th>4</th>
  <td>train/3/26209.png</td>
  <td>0</td>
</tr>
```

You can then use `image_data_from_csv`:

```
data = image_data_from_csv(MNIST_PATH, ds_tfms=get_transforms(do_flip=False), size=28)
```

```
show_image_batch(data.train_dl, classes=[3,7], rows=3, figsize=(5,5))
```

An example of multiclassification can be downloaded with the following cell. It's a sample of the planet dataset.

```
untar_data(PLANET_PATH)
```

If we open the labels files, we seach that each image has one or more tags, separated by a space.

```
df =pd.read_csv(PLANET_PATH/'labels.csv')
df.head()
```

```
<tr style="text-align: right;">
  <th></th>
  <th>image_name</th>
  <th>tags</th>
</tr>
```

```
<tr>
  <th>0</th>
  <td>train_21983</td>
  <td>partly_cloudy primary</td>
</tr>
<tr>
  <th>1</th>
  <td>train_9516</td>
  <td>clear cultivation primary water</td>
</tr>
<tr>
  <th>2</th>
  <td>train_12664</td>
```

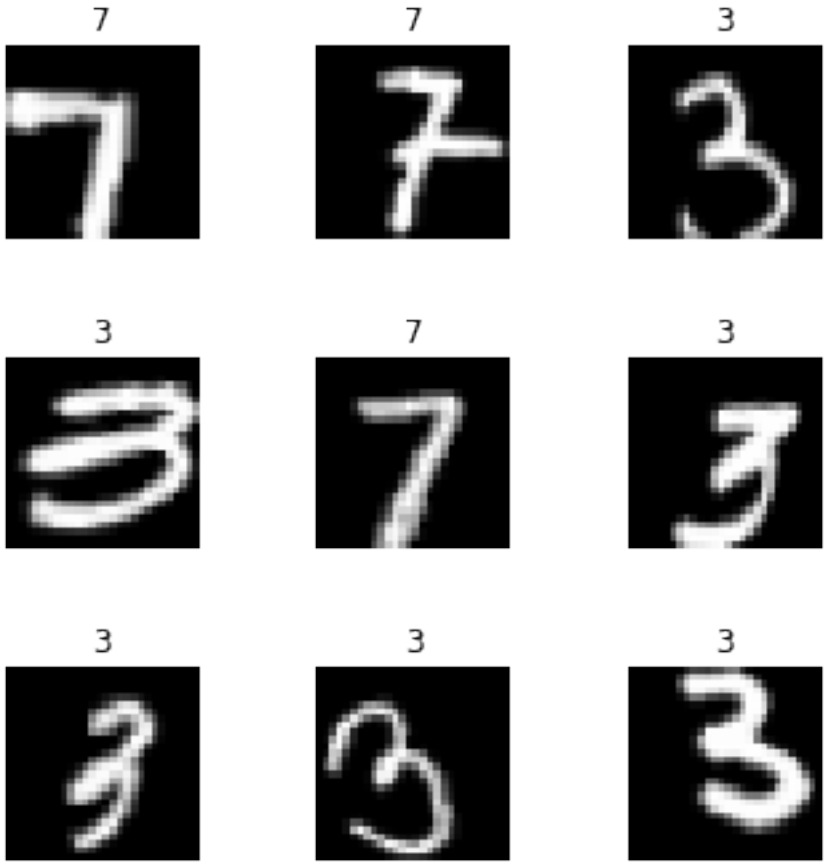

Figure 66: png

```
  <td>haze primary</td>
</tr>
<tr>
  <th>3</th>
  <td>train_36960</td>
  <td>clear primary</td>
</tr>
<tr>
  <th>4</th>
  <td>train_5302</td>
  <td>haze primary road</td>
</tr>
```

```
tfms = get_transforms(flip_vert=True, max_lighting=0.1, max_zoom=1.05, max_warp=0.)
data = image_data_from_csv(PLANET_PATH, folder='train', ds_tfms=tfms, size=128, suffix='.jpg')
```

The `show_image_batch` function wil then print all the labels that correspond to each image.

```
show_image_batch(data.train_dl, data.train_ds.classes, rows=3)
```

Defining a DataBunch

If you quickly want to get a `DataBunch` and train a model, you should process your data to have it in one of the formats the following functions handle.

```
show_doc(image_data_from_folder)
```

`image_data_from_folder`

> `image_data_from_folder(path:PathOrStr, train:PathOrStr='train', valid:PathOrStr='valid', test:Union[Path, str, NoneType]=None, kwargs:Any) -> DataBunch`

Create `DataBunch` from imagenet style dataset in `path` with `train`,`valid`,`test` subfolders. [source]

```
path\
  train\
    clas1\
    clas2\
    ...
    clasn\
  valid\
    clas1\
    clas2\
    ...
    clasn\
```

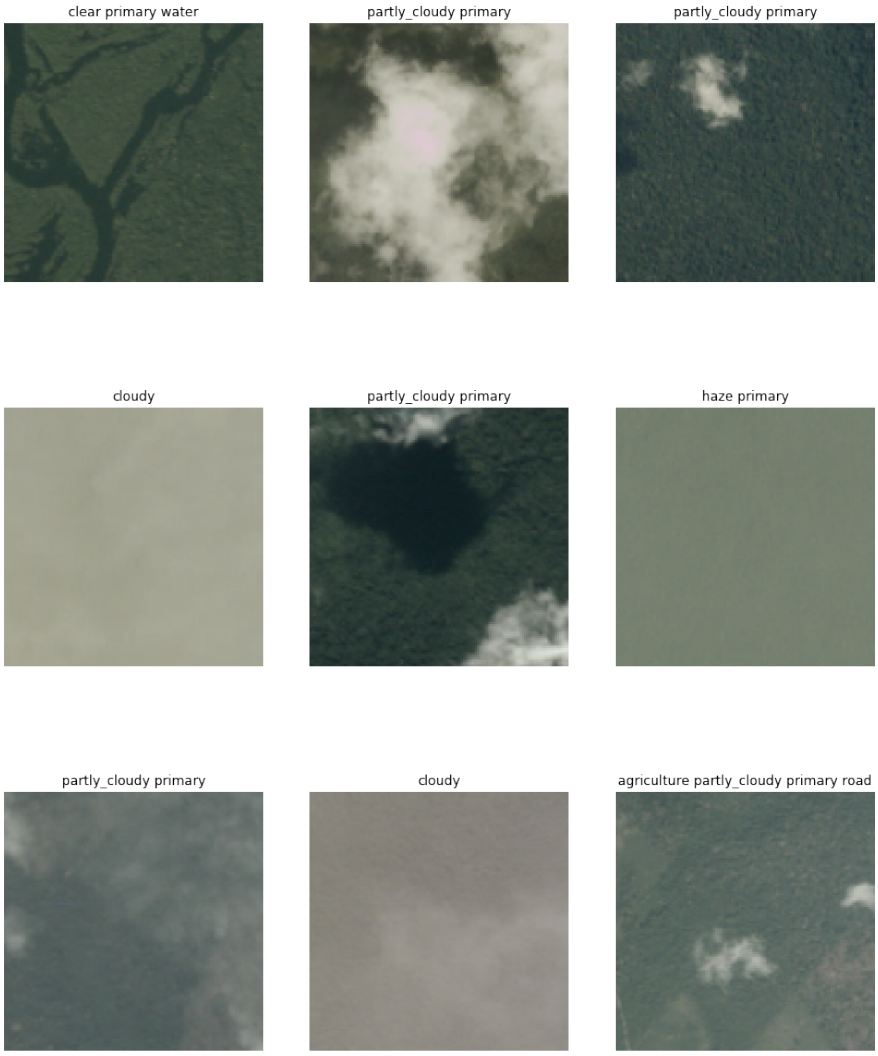

Figure 67: png

```
  test\
```

Note that the test folder is optional. A few of the kwargs you can pass are: - `bs`: Desired batchsize for the dataloaders - `num_workers`: The number of process to launch for data collection - `ds_tfms`: Tuple of two lists of transforms (first for training and second for validation and test set) - `size`: Target size for those transforms - `tfms`: List of transforms to be applied at a batch level (like normalization) - `device`: The device on which to put the batches

```
data = image_data_from_folder(MNIST_PATH, ds_tfms=get_transforms(do_flip=False), size=24)
```

```
show_doc(image_data_from_csv)
```

image_data_from_csv

> image_data_from_csv(path:PathOrStr, folder:PathOrStr='.', sep=None, csv_labels:PathOrStr='labels.csv', valid_pct:float=0.2, test:Union[Path, str, NoneType]=None, suffix:str=None, kwargs:Any) -> DataBunch

Create a `DataBunch` from a csv file. [source]

Create `DataBunch` from `path` by splitting the data in `folder` and labelled in a file `csv_labels` between a training and valid set, putting aside `valid_pct` for the validation. An optional `test` folder contains unlabelled data and `suff` contains an optional suffix to add to the filenames in `csv_labels` (like '.jpg')

The same list of kwargs as in `image_data_from_folder` applies here.

```
data = image_data_from_folder(MNIST_PATH, 'train', ds_tfms=get_transforms(do_flip=False), siz
```

In both cases, you can have a quick look at your data by using the following function:

```
show_doc(show_image_batch, arg_comments={
    'dl': 'A dataloader from which to show a sample',
    'classes': 'List of classes (for the labels)',
    'rows': 'Will make a square of `rows` by `rows` images',
    'figsize': 'Size of the graph shown',
   'denorm': 'Optional function to denormalize the data if it was normalized'
})
```

show_image_batch

> show_image_batch(dl:DataLoader, classes:StrList, rows:int=None, figsize:Tuple[int, int]=(12, 15), denorm:Callable=None)

Show a few images from a batch.

- *dl*: A dataloader from which to show a sample
- *classes*: List of classes (for the labels)

- *rows*: Will make a square of `rows` by `rows` images
- *figsize*: Size of the graph shown
- *denorm*: Optional function to denormalize the data if it was normalized [source]

```
show_image_batch(data.train_dl, data.train_ds.classes, 3, figsize=(6,6))
```

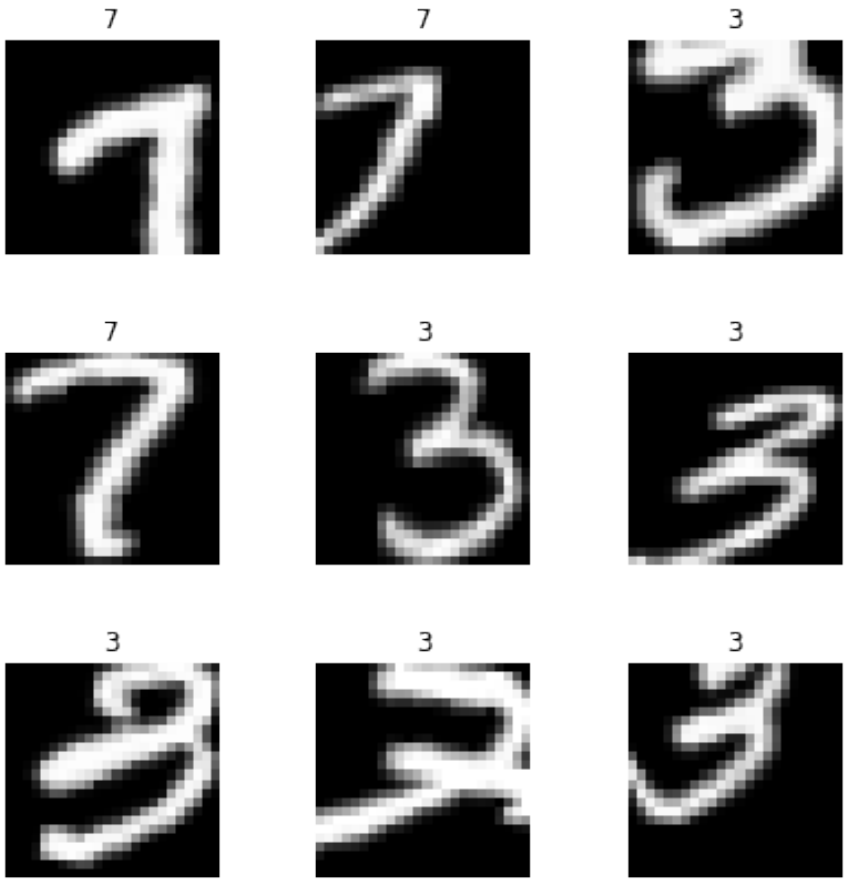

Figure 68: png

Another function that will help visualize a batch is:

```
show_doc(show_images, doc_string=False)
```

`show_images`

`show_images`(`x`:`Collection`[`Image`], `y`:`int`, `rows`:`int`, `classes`:`StrList`,

`figsize`:`Tuple`[`int`, `int`]=`(9, 9)`) [source]

Plot a square of `rows` by `rows` images in `x` titled according to `classes[y]` with a total size of `figsize`.

```
x,y = next(iter(data.train_dl))
x,y = x.cpu(),y.cpu()
show_images(x, y, 3, data.train_ds.classes, figsize=(6,6))
```

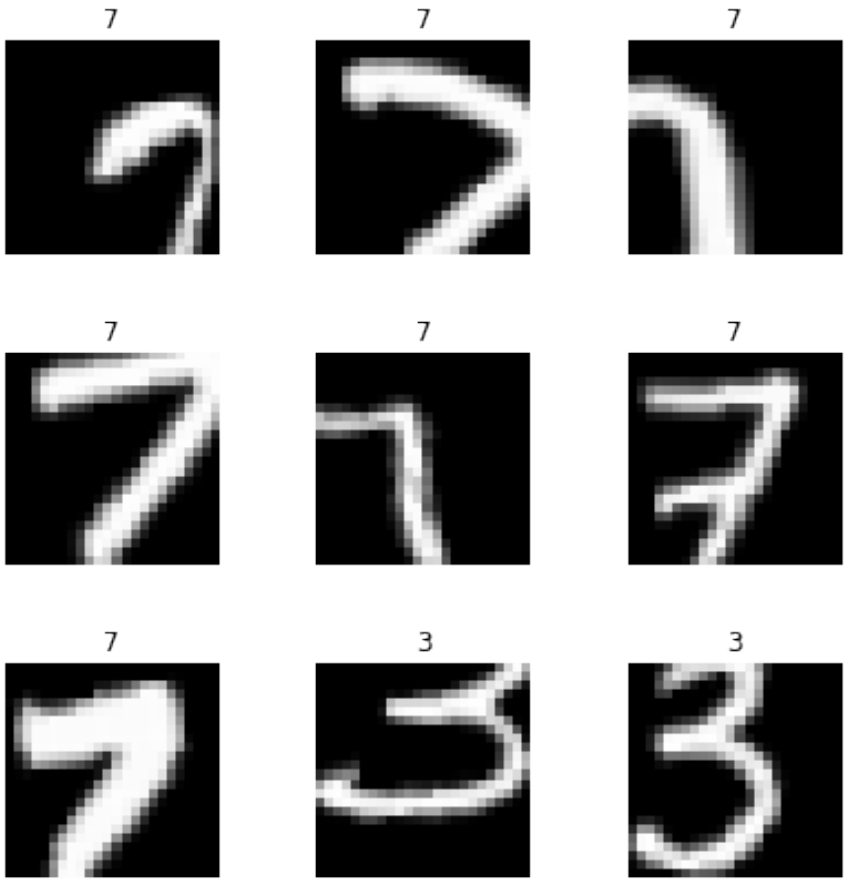

Figure 69: png

Both those functions will handle data with single or multi labels.

A last function that is useful if you're in a segmentation task or an object detection task is

```
show_doc(show_xy_images, doc_string=False)
```

show_xy_images

> show_xy_images(x:Tensor, y:Tensor, rows:int, figsize:tuple=(9, 9)) [source]

Plot a square of `rows` by `rows` images in `x` with the corresponding `y` and a total size of `figsize`.

To show an example, we're going to create a fake batch by transforming the same `Image` several times. See the documentation of `apply_tfms` for more details abut what is happening in this code.

```
def get_img_mask(): return open_image('imgs/car_example.jpg'), open_mask('imgs/mask_example.p

img,mask = get_img_mask()
bs,size = 32,128
tfms = get_transforms()

x,y = torch.zeros(bs,3,size,size), torch.zeros(bs,1,size,size).long()
for i in range(bs):
    img, mask = get_img_mask()
    img = apply_tfms(tfms[0], img, size=size)
    mask = apply_tfms(tfms[0], mask, do_resolve=False, size=size)
    x[i],y[i] = img.data,mask.data

show_xy_images(x, y, 3)
```

Data normalization

You may also want to normalize your data, which can be done by using the following functions.

```
show_doc(normalize)
```

normalize

> normalize(x:Tensor, mean:FloatTensor, std:FloatTensor) -> Tensor

Normalize `x` with `mean` and `std`. [source]

```
show_doc(denormalize)
```

denormalize

> denormalize(x:Tensor, mean:FloatTensor, std:FloatTensor) -> Tensor

Figure 70: png

Denormalize `x` with `mean` and `std`. [source]

```
show_doc(normalize_funcs, doc_string=False)
```

`normalize_funcs`

> `normalize_funcs`(`mean`:`FloatTensor`, `std`:`FloatTensor`, `do_y`=`False`, `device`=`None`) -> `Tuple`[`Callable`, `Callable`] [source]

Create `normalize` and `denormalize` functions using `mean` and `std`. `device` will store them on the device specified. `do_y` determines if the target should also be normaized or not.

On MNIST the mean and std are 0.1307 and 0.3081 respectively (looked on Google). If you're using a pretrained model, you'll need to use the normalization that used to train the model. The imagenet norm and denorm functions are stored as constants inside the library named 2250])) and 2250])). If you're training a model on CIFAR-10, you can also use 2610])) and 2610])).

Note that if you normalize your input, you'll need to denormalize it when using functions like `show_images` or `show_image_batch`. The correct way to look at those pictures is to denormalize them first:

```
data = image_data_from_folder('../data/mnist_sample/', ds_tfms=get_transforms(do_flip=False),
                              tfms=imagenet_norm, size=24)
show_image_batch(data.train_dl, data.train_ds.classes, 3, figsize=(6,6), denorm=imagenet_deno
```

Datasets

Depending on the task you are tackling, you'll need one of the following fastai datasets.

```
show_doc(ImageClassificationDataset)
```

`class ImageClassificationDataset`

> `ImageClassificationDataset`(`fns`:`FilePathList`, `labels`:`StrList`, `classes`:`Optional`[`Classes`]=`None`) :: `ImageDataset`

`Dataset` for folders of images in style {folder}/{class}/{images}. [source]

This is the basic dataset for image classification: `fns` are the filenames of the images and `labels` the corresponding labels. Optionally, `classes` contains a name for each possible label.

```
show_doc(ImageClassificationDataset.from_folder)
```

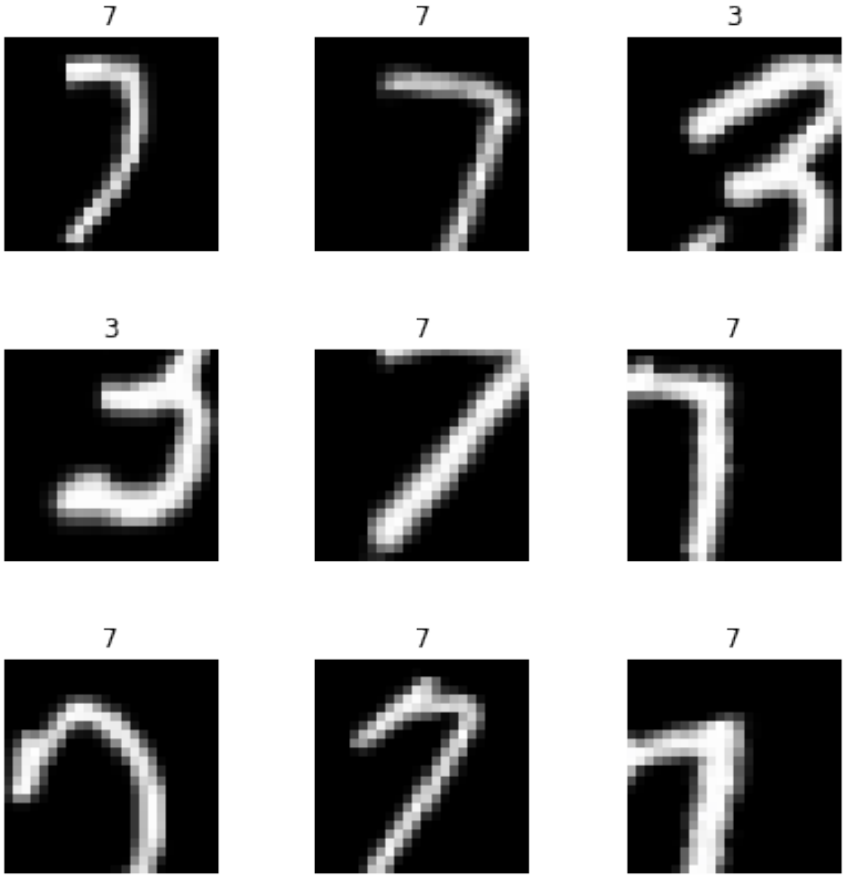

Figure 71: png

from_folder

> from_folder(folder:Path, classes:Optional[Classes]=None, valid_pct:float=0.0, check_ext:bool=True) -> Union[ImageClassificationDataset, List[ImageClassificationDataset]]

Dataset of `classes` labeled images in `folder`. Optional `valid_pct` split validation set. [source]

Create an `ImageClassificationDataset` automatically from a `folder`. If `classes` is None, it will be set to the names of the directories in `folder`. `check_ext` forces the function to only keep filenames with image extensions.

```
show_doc(ImageClassificationDataset.from_single_folder, doc_string=False)
```

from_single_folder

> from_single_folder(folder:PathOrStr, classes:Classes, check_ext=True) [source]

Typically used for define a test set. Label all images in `folder` with `classes[0]`. `check_ext` forces the function to only keep filenems with image extensions.

```
show_doc(ImageMultiDataset, doc_string=False)
```

class ImageMultiDataset

> ImageMultiDataset(fns:FilePathList, labels:StrList, classes:Optional[Classes]=None) :: LabelDataset [source]

This is the basic dataset for image classification with multiple labels: `fns` are the filenames of the images and `labels` the corresponding labels (may be more than one for each image). Optionally, `classes` contains a name for each possible label.

```
show_doc(ImageMultiDataset.from_folder, doc_string=False)
```

from_folder

> from_folder(path:PathOrStr, folder:PathOrStr, fns:Series, labels:StrList, valid_pct:float=0.2, classes:Optional[Classes]=None) [source]

To create an `ImageMultiDataset` automatically in `path` from a `folder` and `fns`. If `classes` is None, it will be set to the names of the different `labels` seen. You can split the images in this `folder` in a train/valid dataset if `valid_pct` is non-zero. `check_ext` forces the function to only keep filenems with image extensions.

```
show_doc(ImageMultiDataset.from_single_folder, doc_string=False)
```

from_single_folder

> from_single_folder(folder:PathOrStr, classes:Classes, check_ext=True) [source]

Typically used for define a test set. Label all images in `folder` with `classes[0]`. `check_ext` forces the function to only keep filenems with image extensions.

To help scan a folder for these `Dataset`, we use the following helper function:

```
show_doc(get_image_files, doc_string=False)
```

get_image_files

> get_image_files(c:Path, check_ext:bool=True) -> FilePathList [source]

Return list of files in `c` that are images. `check_ext` will filter to keep only the files with image extensions.

```
show_doc(SegmentationDataset, doc_string=False)
```

class SegmentationDataset

> SegmentationDataset(x:Collection[PathOrStr], y:Collection[PathOrStr]) :: DatasetBase [source]

This is the basic dataset for image sementation: `x` contains the filenames of the images and `y` the ones of the masks.

```
show_doc(ObjectDetectDataset, doc_string=False)
```

class ObjectDetectDataset

> ObjectDetectDataset(x_fns:FilePathList, bbs:Collection[Collection[int]]) :: Dataset [source]

This is the basic dataset for object detection: `x` contains the filenames of the images and `bbs` the corresponding bounding boxes.

Finally, to apply transformations to `Image` in a `Dataset`, we use this last class.

```
show_doc(ImageDataset)
```

class ImageDataset

> ImageDataset(fns:FilePathList, y:ndarray) :: LabelDataset

Abstract Dataset containing images [source]

```
show_doc(DatasetTfm, doc_string=False)
```

class DatasetTfm

> DatasetTfm(ds:Dataset, tfms:Collection[Transform]=None, tfm_y:bool=False, kwargs:Any) :: Dataset [source]

Dataset that applies the list of transforms tfms to every item drawn. If tfms should be applied to the targets as well, tfm_y should be True. kwargs will be passed to apply_tfms internally.

Then this last function automatizes the process of creating DatasetTfm:

```
show_doc(transform_datasets, doc_string=False)
```

transform_datasets

> transform_datasets(train_ds:Dataset, valid_ds:Dataset, test_ds:Optional[Dataset]=None, tfms:Optional[Tuple[Collection[Transform], Collection[Transform]]]=None, kwargs:Any) [source]

Create train, valid and maybe test DatasetTfm from train_ds, valid_ds and maybe test_ds using tfms. It should be a tuple containing the transforms for the training set, then for the validation and test set.

Undocumented Methods - Methods moved below this line will intentionally be hidden

```
show_doc(ImageMultiDataset.get_labels)
```

get_labels

> get_labels(idx:int) -> StrList [source]

```
show_doc(ImageMultiDataset.encode)
```

encode

> encode(x:Collection[int])

One-hot encode the target. [source]

Computer Vision models zoo

```
from fastai.gen_doc.nbdoc import *
from fastai.vision.models.darknet import Darknet
from fastai.vision.models.wrn import wrn_22, WideResNet
```

On top of the models offered by torchivision, the fastai library has implementations for the following models:

- Darknet architecture, which is the base of Yolo v3
- Unet architecture based on a pretrained model. The original unet is described here, the model implementation is detailed in `models.unet`
- Wide resnets architectures, as introduced in this article.

```
show_doc(Darknet, doc_string=False)
```

class Darknet

> `Darknet(num_blocks:Collection[int], num_classes:int, nf=32)` :: `Module` [source]

Create a Darknet with blocks of sizes given in `num_blocks`, ending with `num_classes` and using `nf` initial features. Darknet53 uses `num_blocks = [1,2,8,8,4]`.

```
show_doc(WideResNet, doc_string=False)
```

class WideResNet

> `WideResNet(num_groups:int, N:int, num_classes:int, k:int=1, drop_p:float=0.0, start_nf:int=16)` :: `Module` [source]

Create a wide resnet with blocks `num_groups` groups, each containing blocks of size `N`. `k` is the width of the resnet, `start_nf` the initial number of features. Dropout of `drop_p` is applied at the end of each block.

```
show_doc(wrn_22)
```

wrn_22

> `wrn_22()` [source]

Creates a wide resnet for CIFAR-10 with `num_groups=3`, `N=3`, `k=6` and `drop_p=0.`.

Dynamic U-Net

This module builds a dynamic U-Net from any backbone pretrained on ImageNet, automatically inferring the intermediate sizes.

```
from fastai.gen_doc.nbdoc import *
from fastai.vision.models.unet import *
```

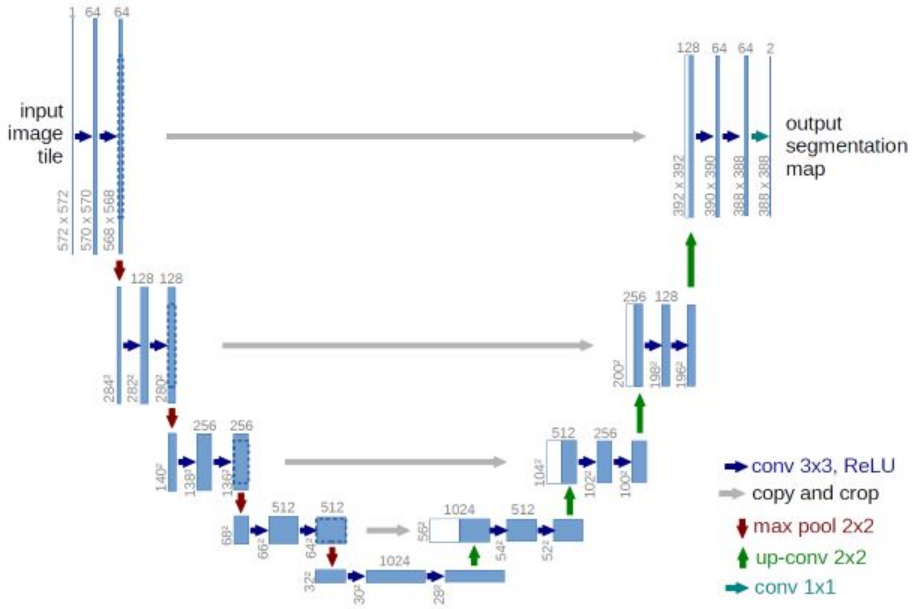

Figure 72: U-Net architecure

This is the original U-Net. The difference here is that the left part is a pretrained model.

```
show_doc(DynamicUnet, doc_string=False)
```

`class DynamicUnet`

> `DynamicUnet(encoder:Module, n_classes:int)` :: `Sequential` [source]

Builds a U-Net from a given `encoder` (that can be a pretrained model) and with a final output of `n_classes`. During the initialization, it uses `Hooks` to determine the intermediate features sizes by passing a dummy input throught the model.

```
show_doc(UnetBlock, doc_string=False)
```

class UnetBlock

UnetBlock(up_in_c:int, x_in_c:int, hook:Hook) :: Module [source]

Builds a U-Net block that receives the output of the last block to be upsampled (size up_in_c) and the activations features from an intermediate layer of the encoder (size x_in_c, this is the lateral connection). The hook is set to this intermediate layer to store the output needed for this block.

Text models, data, and training

```
from fastai.gen_doc.nbdoc import *
from fastai.text import *
from fastai import *
from fastai.docs import *
```

The text module of the fastai library contains all the necessary functions to define a Dataset suitable for the various NLP (Natural Language Processing) tasks and quickly generate models you can use for them. Specifically: - text.transform contains all the scripts to preprocess your data, from raw text to token ids, - text.data contains the definition of TextDataset, which the main class you'll need in NLP, - text.learner contains helper functions to quickly create a language model or an RNN classifier.

Have a look at the links above for full details of the API of each module, of read on for a quick overview.

Quick Start: Training an IMDb sentiment model with *ULMFiT*

Let's start with a quick end-to-end example of training a model. We'll train a sentiment classifer on a sample of the popular IMDb data, showing 4 steps:

1. Reading and viewing the IMDb data
2. Getting your data ready for modeling
3. Fine-tuning a language model
4. Building a classifier

Reading and viewing the IMDb data

Contrary to images in Computer Vision, text can't directly be transformed into numbers to be fed into a model. The first thing we need to do is to preprocess our data so that we change the raw texts to lists of words, or tokens (a step that is called tokenization) then transform these tokens into numbers (a step that is

called numericalization). These numbers are then passed to embedding layers that wil convert them in arrays of floats before passing them through a model.

You can find on the web plenty of Word Embeddings to directly convert your tokens into floats. Those word embeddings have generally be trained on a large corpus such as wikipedia. Following the work of ULMFiT, the fastai library is more focused on using pre-trained Language Models and fine-tuning them. Word embeddings are just vectors of 300 or 400 floats that represent different words, but a pretrained language model not only has those, but has also been trained to get a representation of full sentences and documents.

That's why the library is structured around three steps:

1. Get your data preprocessed and ready to use in a minimum amount of code,
2. Create a language model with pretrained weights that you can fine-tune to your dataset,
3. Create other models such as classifiers on top of the encoder of the language model.

To show examples, we have provided a small sample of the IMDB dataset which contains 1,000 reviews of movies with labels (positive or negative).

```
untar_data(IMDB_PATH)
IMDB_PATH
```

```
PosixPath('../data/imdb_sample')
```

Creating a dataset from your raw texts is very simple if you have it in one of those ways - organized it in folders in an ImageNet style - organized in a csv file with labels columns and a text columns

Here, the sample from imdb is in a train and valid csv files that looks like this:

```
df = pd.read_csv(IMDB_PATH/'train.csv', header=None)
df.head()
```

```
<tr style="text-align: right;">
  <th></th>
  <th>0</th>
  <th>1</th>
</tr>
```

```
<tr>
  <th>0</th>
  <td>0</td>
  <td>Un-bleeping-believable! Meg Ryan doesn't even ...</td>
</tr>
<tr>
  <th>1</th>
  <td>1</td>
```

```
  <td>This is a extremely well-made film. The acting...</td>
</tr>
<tr>
  <th>2</th>
  <td>0</td>
  <td>Every once in a long while a movie will come a...</td>
</tr>
<tr>
  <th>3</th>
  <td>1</td>
  <td>Name just says it all. I watched this movie wi...</td>
</tr>
<tr>
  <th>4</th>
  <td>0</td>
  <td>This movie succeeds at being one of the most u...</td>
</tr>
```

Labels are encoded (though we can tell 1 seems to be positive from the bits of comments we see), and the file classes.txt contain the correspondance between index and names.

```
classes = read_classes(IMDB_PATH/'classes.txt')
classes[0], classes[1]
```

```
('negative', 'positive')
```

Getting your data ready for modeling

To create a dataset, we can just use the `TextDataset.from_csv` method. It will automatically do the two step of preprocessing for us. There is more information about those two steps in `text.transform` where you'll also find how to customize the tokenizer from the fastai defaults. Note that to execute this line, you need to download the english model of spacy, which can be done by typing in your terminal:

```
python -m spacy download en
```

```
for file in ['train_tok.npy', 'valid_tok.npy']:
   if os.path.exists(IMDB_PATH/'tmp'/file): os.remove(IMDB_PATH/'tmp'/file)
```

```
train_ds = TextDataset.from_csv(IMDB_PATH, name='train', classes=classes)
```

```
Tokenizing train.
```

```
HBox(children=(IntProgress(value=0, max=1), HTML(value='')))
```

```
HBox(children=(IntProgress(value=0, max=1), HTML(value='0.00% [0/1 00:00<00:00]')))
```

```
Numericalizing train.
```

Note that to avoid redoing this step (which can be quite time-consuming if you have a very large dataset), the fastai library has created a `tmp` directory in the folder given, to store the tokens (in `_tok.npy` files), the labels (in `_lbl.npy` files), the ids of the tokens (in `_id.npy` files) and the dictionary (`itos.pkl`). When you reexecute the line in the future, it will directly load that data.

To get a `DataBunch` quickly, there are also several factory methods depending on how our data is structured. They are all detailed in `transform.data`, here we'll use `text_data_from_csv`.

```
# Language model data
data_lm = text_data_from_csv(Path(IMDB_PATH), data_func=lm_data)
# Classifier model data
data_clas = text_data_from_csv(Path(IMDB_PATH), data_func=classifier_data, vocab=data_lm.trai
```

```
Tokenizing valid.
```

```
HBox(children=(IntProgress(value=0, max=1), HTML(value='')))
```

```
HBox(children=(IntProgress(value=0, max=1), HTML(value='0.00% [0/1 00:00<00:00]')))
```

```
Numericalizing valid.
```

Note that `TextDataset` was called behind the scenes for `train.csv` and `valid.csv` but, as explained earlier, it only preprocessed the validation set, since it had stored the result for the training set. The `data_func` argument here tells the `text_data_from_csv` function how to organize the data. In the first one, we prepare it for a language model, and in the second one, for a classifier (see the differences in `text.data`).

For the classifier, we also pass the vocabulary (mapping from ids to words) that we want to use: this is to ensure that `data_clas` will use the same dictionary as `data_lm`.

Fine-tuning a language model

We can use the `data_lm` object we created earlier to fine-tune a pretrained language model. fast.ai has an English model available that we can download.

```
download_wt103_model()
```

Now we can create a learner object that will directly create a model, load the pretrained weights and be ready for fine-tuning.

```
learn = RNNLearner.language_model(data_lm, pretrained_fnames=['lstm_wt103', 'itos_wt103'], dr
learn.fit_one_cycle(1, 1e-2)
```

```
Total time: 00:04
epoch  train loss  valid loss  accuracy
0      4.699907    4.201264    0.247246  (00:04)
```

Like a computer vision model, we can then unfreeze the model and fine-tune it.

```
learn.unfreeze()
learn.fit_one_cycle(5, 1e-3)
```

```
Total time: 00:27
epoch  train loss  valid loss  accuracy
0      4.464529    4.121362    0.255822  (00:05)
1      4.350247    4.043319    0.261671  (00:05)
2      4.229458    4.010417    0.264702  (00:05)
3      4.141184    3.991524    0.266229  (00:05)
4      4.082627    3.987816    0.265016  (00:05)
```

And finally we save the encoder to be able to use it for classification in the next section.

```
learn.save_encoder('ft_enc')
```

Building a classifier

We now use the `data_clas` object we created earlier to build a classifier with our fine-tuned encoder. The learner object can be done in a single line.

```
learn = RNNLearner.classifier(data_clas, drop_mult=0.5)
learn.load_encoder('ft_enc')
learn.fit_one_cycle(1, 1e-2)
```

```
Total time: 00:05
epoch  train loss  valid loss  accuracy
0      0.591999    0.632481    0.620000  (00:05)
```

Again, we can unfreeze the model and fine-tune it.

```
learn.freeze_to(-2)
learn.fit_one_cycle(1, slice(5e-3/2., 5e-3))
```

```
Total time: 00:05
epoch  train loss  valid loss  accuracy
0      0.537493    0.570067    0.685000  (00:05)
```

```
learn.unfreeze()
learn.fit_one_cycle(1, slice(2e-3/100, 2e-3))
```

```
Total time: 00:09
epoch  train loss  valid loss  accuracy
0      0.436120    0.544485    0.735000  (00:09)
```

NLP model creation and training

```
from fastai.gen_doc.nbdoc import *
from fastai.text import *
from fastai.docs import *
```

The main thing here is `RNNLearner`. There are also some utility functions to help create and update text models.

```
show_doc(RNNLearner, doc_string=False)
```

`class RNNLearner`

> `RNNLearner`(`data`:`DataBunch`, `model`:`Module`, `bptt`:`int`=`70`, `split_func`:`OptSplitFunc`=`None`, `clip`:`float`=`None`, `adjust`:`bool`=`False`, `alpha`:`float`=`2.0`, `beta`:`float`=`1.0`, `kwargs`) :: `Learner` [source]

Handles the whole creation of a `Learner` from data and a `model` with a text data using a certain `bptt`. The `split_func` is used to properly split the model in different groups for gradual unfreezing and differential learning rates. Gradient clipping of `clip` is optionally applied. `adjust`, `alpha` and `beta` are all passed to create an instance of `RNNTrainer`. Can be used for a language model or an RNN classifier. It also handles the conversion of weights from a pretrained model as well as saving or loading the encoder.

Factory methods

```
show_doc(RNNLearner.classifier, doc_string=False)
```

`classifier`

> `classifier(data:DataBunch, bptt:int=70, max_len:int=1400, emb_sz:int=400, nh:int=1150, nl:int=3, lin_ftrs:Collection[int]=None, ps:Collection[float]=None, pad_token:int=1, drop_mult:float=1.0, qrnn:bool=False, kwargs) -> RNNLearner` [source]

Create an RNNLearner with a classifier model from `data`. The model used is the encoder of an AWD-LSTM that is built with embeddings of size `emb_sz`, a hidden size of `nh`, and `nl` layers (the `vocab_size` is inferred from the `data`). All the dropouts are put to values that we found worked pretty well and you can control their strength by adjusting `drop_mult`. If `qrnn` is True, the model uses QRNN cells instead of LSTMs.

The input texts are fed into that model by bunch of `bptt` and only the last `max_len` activations are considerated. This gives us the backbone of our model. The head then consists of: - a layer that concatenates the final outputs of the RNN with the maximum and average of all the intermediate outputs (on the sequence length dimension), - blocks of (`nn.BatchNorm1d`, `nn.Dropout`, `nn.Linear`, `nn.ReLU`) layers.

The blocks are defined by the `lin_ftrs` and `drops` arguments. Specifically, the first block will have a number of inputs inferred from the backbone arch and the last one will have a number of outputs equal to data.c (which contains the number of classes of the data) and the intermediate blocks have a number of inputs/outputs determined by `lin_ftrs` (of course a block has a number of inputs equal to the number of outputs of the previous block). The dropouts all have a the same value ps if you pass a float, or the corresponding values if you pass a list. Default is to have an intermediate hidden size of 50 (which makes two blocks model_activation -> 50 -> n_classes) with a dropout of 0.1.

```
data = get_imdb(classifier=True)
learn = RNNLearner.classifier(data, drop_mult=0.5)
```

```
show_doc(RNNLearner.language_model, doc_string=False)
```

`language_model`

> `language_model(data:DataBunch, bptt:int=70, emb_sz:int=400, nh:int=1150, nl:int=3, pad_token:int=1, drop_mult:float=1.0, tie_weights:bool=True, bias:bool=True, qrnn:bool=False, pretrained_fnames:OptStrTuple=None, kwargs) -> RNNLearner` [source]

Create an RNNLearner with a language model from `data` of a certain `bptt`. The model used is an AWD-LSTM that is built with embeddings of size `emb_sz`, a hidden size of `nh`, and `nl` layers (the `vocab_size` is inferred from the `data`). All the dropouts are put to values that we found worked pretty well and you can control their strength by adjusting `drop_mult`. If `qrnn` is True, the model uses

QRNN cells instead of LSTMs. The flag `tied_weights` control if we should use the same weights for the encoder and the decoder, the flag `bias` controls if the last linear layer (the decoder) has bias or not.

You can specify `pretrained_fnames` if you want to use the weights of a pretrained model. This should be a list of the name of the weight file and the name of the corresponding dictionary. The dictionary is needed because the function will internally convert the embeddings of the pretrained models to match the dictionary of the `data` passed (a word may have a different id for the pretrained model). Those two files should be in the models directory of `data.path`.

```
data = get_imdb()
learn = RNNLearner.language_model(data, pretrained_fnames=['lstm_wt103', 'itos_wt103'], drop_
```

Loading and saving

```
show_doc(RNNLearner.load_encoder)
```

`load_encoder`

> `load_encoder`(`name`:`str`)

Load the encoder `name` from the model directory. [source]

```
show_doc(RNNLearner.save_encoder)
```

`save_encoder`

> `save_encoder`(`name`:`str`)

Save the encoder to `name` inside the model directory. [source]

```
show_doc(RNNLearner.load_pretrained, doc_string=False)
```

`load_pretrained`

> `load_pretrained`(`wgts_fname`:`str`, `itos_fname`:`str`) [source]

Opens the weights in the `wgts_fname` of `self.model_dir` and the dictionary in `itos_fname` then adapts the pretrained weights to the vocabulary of the data. The two files should be in the models directory of the `learner.path`.

Utility functions

```
show_doc(lm_split)
```

`lm_split`

> `lm_split(model:Module) -> List[Module]`

Split a RNN `model` in groups for differential learning rates. [source]

```
show_doc(rnn_classifier_split)
```

`rnn_classifier_split`

> `rnn_classifier_split(model:Module) -> List[Module]`

Split a RNN `model` in groups for differential learning rates. [source]

```
show_doc(convert_weights, doc_string=False)
```

`convert_weights`

> `convert_weights(wgts:Weights, stoi_wgts:Dict[str, int], itos_new:StrList) -> Weights` [source]

Convert the `wgts` from an dictionary `stoi_wgts` (mapping of word to id) to a new dictionary `itos_new` (correspondans id to word).

NLP Preprocessing

```
from fastai.gen_doc.nbdoc import *
from fastai.text import *
from fastai import *
from fastai.docs import *
```

`text.tranform` contains the functions that deal behind the scenes with the two main tasks when preparing texts for modelling: *tokenization* and *numericalization*.

Tokenization splits the raw texts into tokens (wich can be words, or punctuation signs…). The most basic way to do this would be to separate according to spaces, but it's possible to be more subtle; for instance, the contractions like "isn't" or "don't" should be split in ["is","n't"] or ["do","n't"]. By default fastai will use the powerful spacy tokenizer.

Numericalization is easier as it just consists in attributing a unique id to each token and mapping each of those tokens to their respective ids.

Tokenization

Introduction

This step is actually divided in two phases: first, we apply a certain list of `rules` to the raw texts as preprocessing, then we use the tokenizer to split them in lists of tokens. Combining together those `rules`, the `tok_func`and the `lang` to process the texts is the role of the `Tokenizer` class.

```
show_doc(Tokenizer, doc_string=False)
```

class Tokenizer

> `Tokenizer`(`tok_func`:`Callable`=`'SpacyTokenizer'`, `lang`:`str`=`'en'`, `rules`:`ListRules`=`None`, `special_cases`:`StrList`=`None`, `n_cpus`:`int`=`None`) [source]

This class will process texts by appling them the `rules` then tokenizing them with `tok_func(lang)`. `special_cases` are a list of tokens passed as special to the tokenizer and `n_cpus` is the number of cpus to use for multi-processing (by default, half the cpus available). We don't directly pass a tokenizer for multi-processing purposes: each process needs to initiate a tokenizer of its own. The rules and special_cases default to

```
default_rules = [fix_html, replace_rep, replace_wrep, deal_caps,
spec_add_spaces, rm_useless_spaces]
```

[source]

and

```
default_spec_tok = [BOS, FLD, UNK, PAD]
```

[source]

```
show_doc(Tokenizer.process_text)
```

process_text

> `process_text`(`t`:`str`, `tok`:`BaseTokenizer`) -> `List`[`str`]

Processe one text `t` with tokenizer `tok`. [source]

```
show_doc(Tokenizer.process_all)
```

process_all

> `process_all`(`texts`:`StrList`) -> `List`[`List`[`str`]]

Process a list of `texts`. [source]

For an example, we're going to grab some IMDB reviews.

```
untar_data(IMDB_PATH)
IMDB_PATH
```

```
HBox(children=(IntProgress(value=0, max=570696), HTML(value='')))
```

```
PosixPath('../data/imdb_sample')
```

```
df = pd.read_csv(IMDB_PATH/'train.csv', header=None)
example_text = df.iloc[2][1]; example_text
```

```
'Every once in a long while a movie will come along that will be so awful that I feel compelled to
```

```
tokenizer = Tokenizer()
tok = SpacyTokenizer('en')
' '.join(tokenizer.process_text(example_text, tok))
```

```
'every once in a long while a movie will come along that will be so awful that i feel compelled to
```

As explained before, the tokenizer split the text according to words/punctuations signs but in a smart manner. The rules (see below) also have modified the text a little bit. We can tokenize a list of texts directly at the same time:

```
df = pd.read_csv(IMDB_PATH/'train.csv', header=None)
texts = df[1].values
tokenizer = Tokenizer()
tokens = tokenizer.process_all(texts)
' '.join(tokens[2])
```

```
'every once in a long while a movie will come along that will be so awful that i feel compelled to
```

Customize the tokenizer

The `tok_func` must return an instance of `BaseTokenizer`:

```
show_doc(BaseTokenizer)
```

class BaseTokenizer

> `BaseTokenizer(lang:str)`

Basic class for a tokenizer function. [source]

```
show_doc(BaseTokenizer.tokenizer)
```

tokenizer

> tokenizer(t:str) -> List[str] [source]

Take a text `t` and returns the list of its tokens.

```
show_doc(BaseTokenizer.add_special_cases)
```

add_special_cases

> add_special_cases(toks:StrList) [source]

Record a list of special tokens `toks`.

The fastai library uses spacy tokenizers as its default. The following class wraps it as `BaseTokenizer`.

```
show_doc(SpacyTokenizer)
```

class SpacyTokenizer

> SpacyTokenizer(lang:str) :: BaseTokenizer

Wrapper around a spacy tokenizer to make it a `BaseTokenizer`. [source]

If you want to use your custom tokenizer, just subclass the `BaseTokenizer` and override its `tokenizer` and `add_spec_cases` functions.

Rules

Rules are just functions that take a string and return the modified string. This allows you to customize the list of `default_rules` as you please. Those `default_rules` are:

```
show_doc(deal_caps, doc_string=False)
```

deal_caps

> deal_caps(t:str) -> str [source]

In `t`, if a word is written in all caps, we put it in a lower case and add a special token before. A model will more easily learn this way the meaning of the sentence. The rest of the capitals are removed.

```
deal_caps("I'm suddenly SHOUTING FOR NO REASON!")
```

```
"i'm suddenly  xxup shouting  xxup for no  xxup reason!"
```

```
show_doc(fix_html, doc_string=False)
```

fix_html

fix_html(x:str) -> str [source]

This rules replaces a bunch of HTML characters or norms in plain text ones. For instance
 are replaced by \n,   by spaces etc…

```
fix_html("Some HTML text<br />")
```

```
'Some HTML& text\n'
```

```
show_doc(replace_rep, doc_string=False)
```

replace_rep

replace_rep(t:str) -> str [source]

Whenever a character is repeated more than three times in t, we replace the whole thing by 'TK_REP n char' where n is the number of occurences and char the character.

```
replace_rep("I'm so excited!!!!!!!!")
```

```
"I'm so excited xxrep 8 ! "
```

```
show_doc(replace_wrep, doc_string=False)
```

replace_wrep

replace_wrep(t:str) -> str [source]

Whenever a word is repeated more than four times in t, we replace the whole thing by 'TK_WREP n w' where n is the number of occurences and w the word repeated.

```
replace_wrep("I've never ever ever ever ever ever ever ever done this.")
```

```
"I've never  xxwrep 7 ever  done this."
```

```
show_doc(rm_useless_spaces)
```

rm_useless_spaces

rm_useless_spaces(t:str) -> str

Remove multiple spaces in t. [source]

```
rm_useless_spaces("Inconsistent   use  of     spaces.")
```

```
'Inconsistent use of spaces.'
```

```
show_doc(spec_add_spaces)
```

spec_add_spaces

> spec_add_spaces(t:str) -> str

Add spaces around / and # in `t`. [source]

```
spec_add_spaces('I #like to #put #hashtags #everywhere!')
```

```
'I  # like to  # put  # hashtags  # everywhere!'
```

Numericalization

To convert our set of tokens to unique ids (and be able to have them go through embeddings), we use the following class:

```
show_doc(Vocab, doc_string=False)
```

class Vocab

> Vocab(path:PathOrStr) [source]

Contain the correspondance between numbers and tokens and numericalize. `path` should point to the 'tmp' directory with the token and id files.

```
show_doc(Vocab.create, doc_string=False)
```

create

> create(path:PathOrStr, tokens:Tokens, max_vocab:int, min_freq:int) -> Vocab [source]

Create a `Vocab` dictionary from a set of `tokens` in `path`. Only keeps `max_vocab` tokens, and only if they appear at least `min_freq` times, set the rest to `UNK`.

```
show_doc(Vocab.numericalize)
```

numericalize

> numericalize(t:StrList) -> List[int]

Convert a list of tokens `t` to their ids. [source]

```
show_doc(Vocab.textify)
```

textify

`textify(nums:Collection[int]) -> List[str]`

Convert a list of `nums` to their tokens. [source]

```
vocab = Vocab.create(IMDB_PATH, tokens, max_vocab=1000, min_freq=2)
vocab.numericalize(tokens[2])[:10]
```

```
[207, 321, 11, 6, 246, 144, 6, 22, 88, 240]
```

Undocumented Methods - Methods moved below this line will intentionally be hidden

```
show_doc(SpacyTokenizer.tokenizer)
```

tokenizer

`tokenizer(t:str) -> List[str]` [source]

```
show_doc(SpacyTokenizer.add_special_cases)
```

add_special_cases

`add_special_cases(toks:StrList)` [source]

NLP datasets

```
from fastai.gen_doc.nbdoc import *
from fastai.text import *
from fastai.docs import *
from fastai import *
```

This module contains the `TextDataset` class, which is the main dataset you should use for your NLP tasks. It automatically does the preprocessing steps described in `text.transform`. It also defines a few helper function to quickly get a `DataBunch` ready.

Quickly assemble your data

You should get your data in one of the following formats to make the most of the fastai library and use one of the `text_data` function: - raw text files in folders train, valid, test in an ImageNet style, - a csv (with no index or Header) where the first column(s) gives the label(s) and the folowwing one the associated text,

- tokens and labels arrays already saved, - ids, vocabulary (correspondance id to word) and labels already saved.

If you are assembling the data for a language model, you should define your labels as always 0 to respect those formats. The first time you create a `DataBunch` with one of those functions, your data will be preprocessed automatically and saved, so that the next time you call it is almost instantaneous.

text_data functions

All those functions will require a `data_func` argument that explains how to assemble the datasets into a `DataBunch`. It can be one of the following: - `standard_data`: the datasets are directly used to create a `DataBunch`, - `lm_data`: the datasets are assembled to create a `DataBunch` suitable for a language model, - `classifier_data`: the datasets are assembled to create a `DataBunch` suitable for an NLP classifier.

```
show_doc(text_data_from_folder, doc_string=False)
```

`text_data_from_folder`

> `text_data_from_folder(path:PathOrStr, tokenizer:Tokenizer=None, train:str='train', valid:str='valid', test:Optional[str]=None, shuffle:bool=True, data_func:Callable[Collection[DatasetBase], PathOrStr, KWArgs, DataBunch]='standard_data', vocab:Vocab=None, kwargs)` [source]

This function will create a `DataBunch` from texts placed in `path` in a `train`, `valid` and maybe `test` folders. Text files in the `train` and `valid` folders should be places in subdirectories according to their classes (always the same for a language model) and the ones for the `test` folder should all be placed there directly. `tokenizer` will be used to parse those texts into tokens. The `shuffle` flag will optionally shuffle the texts found.

You can pass a specific `vocab` for the numericalization step (if you are building a classifier from a language model you fine-tuned for instance). kwargs will be split between the `TextDataset` function and to the `get_data` function, you can precise there parameters such as `max_vocab`, `chunksize`, `min_freq`, `n_labels` (see the `TextDataset` documentation) or `bs`, `bptt` and `pad_idx` (see the sections LM data and classifier data).

```
show_doc(text_data_from_csv, doc_string=False)
```

`text_data_from_csv`

> `text_data_from_csv(path:PathOrStr, tokenizer:Tokenizer=None, train:str='train', valid:str='valid', test:Optional[str]=None,`

data_func:Callable[Collection[DatasetBase], PathOrStr, KWArgs, DataBunch]='standard_data', vocab:Vocab=None, kwargs) -> DataBunch [source]

This function will create a DataBunch from texts placed in path in a train.csv, valid.csv and maybe test.csv files. These csv files should have no header or index, and the label(s) should be the first column(s) (be sure to adjust the parameter n_labels if you have more than one). tokenizer will be used to parse those texts into tokens.

You can pass a specific vocab for the numericalization step (if you are building a classifier from a language model you fine-tuned for instance). kwargs will be split between the TextDataset function and to the get_data function, you can precise there parameters such as max_vocab, chunksize, min_freq, n_labels (see the TextDataset documentation) or bs, bptt and pad_idx (see the sections LM data and classifier data).

```
show_doc(text_data_from_tokens, doc_string=False)
```

text_data_from_tokens

text_data_from_tokens(path:PathOrStr, train:str='train', valid:str='valid', test:Optional[str]=None, data_func:Callable[Collection[DatasetBase], PathOrStr, KWArgs, DataBunch]='standard_data', vocab:Vocab=None, kwargs) -> DataBunch [source]

This function will create a DataBunch from texts already tokenized placed in path in files named f{train}{tok_suff}.npy, f{train}{lbl_suff}.npy, f{valid}{tok_suff}.npy, f{valid}{lbl_suff}.npy and maybe f{test}{tok_suff}.npy. If no label file exists, labels will default to all zeros. tok_suff and lbl_suff are '_tok' and '_lbl' respectively.

You can pass a specific vocab for the numericalization step (if you are building a classifier from a language model you fine-tuned for instance). kwargs will be split between the TextDataset function and to the get_data function, you can precise there parameters such as max_vocab, chunksize, min_freq, n_labels, tok_suff and lbl_suff (see the TextDataset documentation) or bs, bptt and pad_idx (see the sections LM data and classifier data).

```
show_doc(text_data_from_ids, doc_string=False)
```

text_data_from_ids

text_data_from_ids(path:PathOrStr, train:str='train', valid:str='valid', test:Optional[str]=None, data_func:Callable[Collection[DatasetBase], PathOrStr, KWArgs, DataBunch]='standard_data', itos:str='itos.pkl', kwargs) -> DataBunch [source]

This function will create a `DataBunch` from texts already tokenized placed in `path` in files named `f{train}{id_suff}`.npy, `f{train}{lbl_suff}`.npy, `f{valid}{id_suff}`.npy, `f{valid}{lbl_suff}`.npy and maybe `f{test}{id_suff}`.npy. If no label file exists, labels will default to all zeros. `id_suff` and `lbl_suff` are '_ids' and '_lbl' respectively. The `itos` file should contain the correspondance from ids to words.

kwargs will be split between the `TextDataset` function and to the `get_data` function, you can precise there parameters such as `max_vocab`, `chunksize`, `min_freq`, `n_labels`, `tok_suff` and `lbl_suff` (see the `TextDataset` documentation) or `bs`, `bptt` and `pad_idx` (see the sections LM data and classifier data).

Example

Untar the IMDB sample dataset if not already done:

```
untar_data(IMDB_PATH)
IMDB_PATH
```

```
PosixPath('../data/imdb_sample')
```

Since it comes in the form of csv files, we will use the corresponding `text_data` method. Here is an overview of what your file you should look like:

```
pd.read_csv(IMDB_PATH/'train.csv', header=None).head()
```

```
<tr style="text-align: right;">
  <th></th>
  <th>0</th>
  <th>1</th>
</tr>
```

```
<tr>
  <th>0</th>
  <td>0</td>
  <td>Un-bleeping-believable! Meg Ryan doesn't even ...</td>
</tr>
<tr>
  <th>1</th>
  <td>1</td>
  <td>This is a extremely well-made film. The acting...</td>
</tr>
<tr>
  <th>2</th>
  <td>0</td>
  <td>Every once in a long while a movie will come a...</td>
</tr>
<tr>
```

```
  <th>3</th>
  <td>1</td>
  <td>Name just says it all. I watched this movie wi...</td>
</tr>
<tr>
  <th>4</th>
  <td>0</td>
  <td>This movie succeeds at being one of the most u...</td>
</tr>
```

And here is a simple way of creating your `DataBunch`.

```
data_lm = text_data_from_csv(Path(IMDB_PATH), data_func=lm_data)
```

```
Tokenizing valid.
```

```
HBox(children=(IntProgress(value=0, max=1), HTML(value='')))
```

```
HBox(children=(IntProgress(value=0, max=1), HTML(value='0.00% [0/1 00:00<00:00]')))
```

```
Numericalizing valid.
```

The TextDataset class

Behind the scenes, the previous functions will create a training, validation and maybe test `TextDataset` which is the class responsible for collecting and pre-processing the data.

```
show_doc(TextDataset, doc_string=False)
```

class TextDataset

> `TextDataset`(`path`:`PathOrStr`, `tokenizer`:`Tokenizer`=`None`, `vocab`:`Vocab`=`None`, `max_vocab`:`int`=`60000`, `chunksize`:`int`=`10000`, `name`:`str`=`'train'`, `min_freq`:`int`=`2`, `n_labels`:`int`=`1`, `create_mtd`:`TextMtd`=`<TextMtd.CSV: 1>`, `classes`:`Classes`=`None`) [source]

This class shouldn't be initialized directly as it will rely on internal files being put in an 'tmp' folder of `path`. `tokenizer` and `vocab` will be used to tokenize and numericalize the texts (if needed). `max_vocab` and `min_freq` are passed at the create of the vocabulary (if needed). `chunksize` is the size of chunks

preprocessed when loading the data from csv or folders. `name` is the name of the set that will be used to name the temporary files. `n_labels` is the number of labels if creating the data from a csv file. `classes` is the correspondance between label and classe. `create_mtd` is an internal flag that tells the `TextDataset` how it was created. It can be: - `CSV` if it was created from texts or csv - `TOK` if it was created from tokens (which means the `TextDataset` will always skip the tokenization) - `IDS` if it was created from tokens (which means the `TextDataset` will always skip the tokenization and the numericalization)

Factory methods

Instead of using the `TextDataset` init method, one of the following factory functions should be used instead:

```
show_doc(TextDataset.from_folder, doc_string=False)
```

`from_folder`

> `from_folder(folder:PathOrStr, tokenizer:Tokenizer=None, name:str='train', classes:Classes=None, shuffle:bool=True, kwargs) -> TextDataset` [source]

Creates a `TextDataset` named `name` by scanning the subfolders in `folder` and using `tokenizer`. If `classes` are passed, only the subfolders named accordingly are checked. If `shuffle` is True, the data will be shuffled. Any additional `kwargs` are passed to the init method of `TextDataset`.

```
show_doc(TextDataset.from_one_folder, doc_string=False)
```

`from_one_folder`

> `from_one_folder(folder:PathOrStr, classes:Classes, tokenizer:Tokenizer=None, name:str='train', shuffle:bool=True, kwargs) -> TextDataset` [source]

Creates a `TextDataset` named `name` by scanning the text files in `folder` and using `tokenizer`. All files are labelled `classes[0]` so this is typically used for the test set. If `shuffle` is True, the data will be shuffled. Any additional `kwargs` are passed to the init method of `TextDataset`.

```
show_doc(TextDataset.from_csv, doc_string=False)
```

`from_csv`

> `from_csv(folder:PathOrStr, tokenizer:Tokenizer=None, name:str='train', kwargs) -> TextDataset` [source]

Creates a `TextDataset` named `name` with the texts in `name`.csv using `tokenizer`. Any additional `kwargs` are passed to the init method of `TextDataset`.

```
show_doc(TextDataset.from_tokens, doc_string=False)
```

`from_tokens`

> `from_tokens`(`folder`:`PathOrStr`, `name`:`str`=`'train'`, `tok_suff`:`str`=`'_tok'`, `lbl_suff`:`str`=`'_lbl'`, `kwargs`) -> `TextDataset` [source]

Creates a `TextDataset` named `name` from tokens and labels saved in `f{name}{tok_suff}.npy` and `f{name}{lbl_suff}.npy` respectively. Any additional `kwargs` are passed to the init method of `TextDataset`.

```
show_doc(TextDataset.from_ids, doc_string=False)
```

`from_ids`

> `from_ids`(`folder`:`PathOrStr`, `name`:`str`=`'train'`, `id_suff`:`str`=`'_ids'`, `lbl_suff`:`str`=`'_lbl'`, `itos`:`str`=`'itos.pkl'`, `kwargs`) -> `TextDataset` [source]

Creates a `TextDataset` named `name` from ids, labels and dictionary saved in `f{name}{id_suff}.npy`, `f{name}{lbl_suff}.npy` and `itos` respectively. Any additional `kwargs` are passed to the init method of `TextDataset`.

Preprocessing

The internal preprocessing is done by the two following methods:

```
show_doc(TextDataset.tokenize)
```

`tokenize`

> `tokenize`()

Tokenize the texts in the csv file. [source]

```
show_doc(TextDataset.numericalize)
```

`numericalize`

> `numericalize`()

Numericalize the tokens in the token file. [source]

Internally, the `TextDataset` will create a 'tmp' folder in which he will copy or save the following files: - `name`.csv (if created from folders or csv) - `name`_tok.npy and `name`_lbl.npy (created by `TextDataset.tokenize` from the last step or

copied if created from tokens) - `name_ids.npy`, `name_lbl.npy` and `itos` (created by `TextDataset.numericalize` from the last step or copied if created from ids)

Then, when you invoke the `TextDataset` again, it will look for those temporary files and check their consistency to use them, in order to avoid doing again the numericalization or the tokenization. If you feel those files have been corrupted in any way, the following method will clear the 'tmp' subfolder of those files:

```
show_doc(TextDataset.clear)
```

clear

> `clear`()

Remove all temporary files. [source]

Internal methods

```
show_doc(TextDataset.check_ids)
```

check_ids

> `check_ids`() -> `bool`

Check if a new numericalization is needed. [source]

```
show_doc(TextDataset.check_toks)
```

check_toks

> `check_toks`() -> `bool`

Check if a new tokenization is needed. [source]

```
show_doc(TextDataset.general_check)
```

general_check

> `general_check`(`pre_files`:`Collection`[`PathOrStr`], `post_files`:`Collection`[`PathOrStr`])

Check that post_files exist and were modified after all the prefiles. [source]

Language Model data

A language model is trained to guess what the next word is inside a flow of words. We don't feed it the different texts separately but concatenate them all together in a big array. To create the batches, we split this array into `bs`

chuncks of continuous texts. Note that in all NLP tasks, we use the pytoch convention of sequence length being the first dimension (and batch size being the second one) so we transpose that array so that we can read the chunks of texts in columns. Here is an example of batch from our imdb sample dataset.

```
data = get_imdb()
x,y = next(iter(data.train_dl))
example = x[:20,:10].cpu()
texts = pd.DataFrame([data.train_ds.vocab.textify(l).split(' ') for l in example])
texts
```

```
<tr style="text-align: right;">
  <th></th>
  <th>0</th>
  <th>1</th>
  <th>2</th>
  <th>3</th>
  <th>4</th>
  <th>5</th>
  <th>6</th>
  <th>7</th>
  <th>8</th>
  <th>9</th>
</tr>
```

```
<tr>
  <th>0</th>
  <td>\n</td>
  <td>for</td>
  <td>the</td>
  <td>could</td>
  <td>wells</td>
  <td>are</td>
  <td>big</td>
  <td>.</td>
  <td>for</td>
  <td>my</td>
</tr>
<tr>
  <th>1</th>
  <td>xxbos</td>
  <td>all</td>
  <td>xxunk</td>
  <td>have</td>
  <td>'</td>
  <td>:</td>
  <td>an</td>
```

```
  <td>after</td>
  <td>him</td>
  <td>xxunk</td>
</tr>
<tr>
  <th>2</th>
  <td>xxfld</td>
  <td>of</td>
  <td>beer</td>
  <td>ran</td>
  <td>original</td>
  <td>stop</td>
  <td>incompetent</td>
  <td>shooting</td>
  <td>in</td>
  <td>shot</td>
</tr>
<tr>
  <th>3</th>
  <td>1</td>
  <td>five</td>
  <td>xxunk</td>
  <td>away</td>
  <td>narrative</td>
  <td>and</td>
  <td>as</td>
  <td>carradine</td>
  <td>future</td>
  <td>so</td>
</tr>
<tr>
  <th>4</th>
  <td>un</td>
  <td>minutes</td>
  <td>could</td>
  <td>from</td>
  <td> </td>
  <td>xxunk</td>
  <td>he</td>
  <td>she</td>
  <td>.</td>
  <td>far</td>
</tr>
<tr>
  <th>5</th>
  <td>-</td>
```

```
  <td>while</td>
  <td>have</td>
  <td>that</td>
  <td>as</td>
  <td>what</td>
  <td>seemed</td>
  <td>beats</td>
  <td>\n</td>
  <td>up</td>
</tr>
<tr>
  <th>6</th>
  <td>xxunk</td>
  <td>the</td>
  <td>been</td>
  <td>thing</td>
  <td>has</td>
  <td>you</td>
  <td>in</td>
  <td>a</td>
  <td>xxbos</td>
  <td>my</td>
</tr>
<tr>
  <th>7</th>
  <td>-</td>
  <td>protagonist</td>
  <td>the</td>
  <td>,</td>
  <td>been</td>
  <td>want</td>
  <td>the</td>
  <td>xxunk</td>
  <td>xxfld</td>
  <td>xxunk</td>
</tr>
<tr>
  <th>8</th>
  <td>believable</td>
  <td>is</td>
  <td>xxunk</td>
  <td>but</td>
  <td>noted</td>
  <td>.</td>
  <td>dream</td>
  <td>xxunk</td>
```

```
  <td>1</td>
  <td>,</td>
</tr>
<tr>
  <th>9</th>
  <td>!</td>
  <td>xxunk</td>
  <td>for</td>
  <td>anyway</td>
  <td>.</td>
  <td>use</td>
  <td>.</td>
  <td>before</td>
  <td>after</td>
  <td>i</td>
</tr>
<tr>
  <th>10</th>
  <td>meg</td>
  <td>her</td>
  <td>a</td>
  <td>.</td>
  <td>viewers</td>
  <td>your</td>
  <td>\n\n</td>
  <td>johnny</td>
  <td>the</td>
  <td>thought</td>
</tr>
<tr>
  <th>11</th>
  <td>ryan</td>
  <td>early</td>
  <td>massive</td>
  <td>i</td>
  <td>may</td>
  <td>time</td>
  <td>a</td>
  <td>can</td>
  <td>failure</td>
  <td>it</td>
</tr>
<tr>
  <th>12</th>
  <td>does</td>
  <td>life</td>
```

```
  <td>series</td>
  <td>actually</td>
  <td>find</td>
  <td>to</td>
  <td>lack</td>
  <td>get</td>
  <td>of</td>
  <td>would</td>
</tr>
<tr>
  <th>13</th>
  <td>n't</td>
  <td>as</td>
  <td>of</td>
  <td>was</td>
  <td>it</td>
  <td>start</td>
  <td>of</td>
  <td>his</td>
  <td>"</td>
  <td>xxunk</td>
</tr>
<tr>
  <th>14</th>
  <td>even</td>
  <td>a</td>
  <td>gags</td>
  <td>getting</td>
  <td>informative</td>
  <td>planning</td>
  <td>humor</td>
  <td>revenge</td>
  <td>the</td>
  <td>in</td>
</tr>
<tr>
  <th>15</th>
  <td>look</td>
  <td>butcher</td>
  <td>built</td>
  <td>into</td>
  <td>to</td>
  <td>and</td>
  <td>is</td>
  <td>.</td>
  <td>xxunk</td>
```

```
  <td>my</td>
</tr>
<tr>
  <th>16</th>
  <td>her</td>
  <td>.</td>
  <td>upon</td>
  <td>this</td>
  <td>note</td>
  <td>not</td>
  <td>the</td>
  <td>still</td>
  <td>"</td>
  <td>xxunk</td>
</tr>
<tr>
  <th>17</th>
  <td>usual</td>
  <td>weird</td>
  <td>gags</td>
  <td>film</td>
  <td>plot</td>
  <td>filming</td>
  <td>biggest</td>
  <td>alive</td>
  <td>at</td>
  <td>.</td>
</tr>
<tr>
  <th>18</th>
  <td>xxunk</td>
  <td>stuff</td>
  <td>,</td>
  <td>,</td>
  <td>details</td>
  <td>until</td>
  <td>problem</td>
  <td>,</td>
  <td>the</td>
  <td>first</td>
</tr>
<tr>
  <th>19</th>
  <td>lovable</td>
  <td>.</td>
  <td>but</td>
```

```
  <td>although</td>
  <td>that</td>
  <td>everything</td>
  <td>with</td>
  <td>it</td>
  <td>box</td>
  <td>,</td>
</tr>
```

Then, as suggested in this article from Stephen Merity et al., we don't use a fixed `bptt` through the different batches but slightly change it from batch to batch.

```
iter_dl = iter(data.train_dl)
for _ in range(5):
    x,y = next(iter_dl)
    print(x.size())
```

```
torch.Size([73, 64])
torch.Size([68, 64])
torch.Size([76, 64])
torch.Size([76, 64])
torch.Size([66, 64])
```

This is all done internally when we use the following inside of the `text_data` functions.

```
show_doc(lm_data, doc_string=False)
```

`lm_data`

> `lm_data`(`datasets`:`Collection`[`TextDataset`], `path`:`PathOrStr`, `kwargs`) -> `DataBunch` [source]

Create a `DataBunch` from the `datasets` for language modelling. Internally calls the next class, that takes the kwargs to define the dataloaders.

```
show_doc(LanguageModelLoader, doc_string=False)
```

`class LanguageModelLoader`

> `LanguageModelLoader`(`dataset`:`TextDataset`, `bs`:`int`=`64`, `bptt`:`int`=`70`, `backwards`:`bool`=`False`) [source]

Takes the texts from `dataset` and concatenate them all, then create a big array with `bs` columns (transposed from the data source so that we read the texts in the columns). Spits batches with a size approximately equal to `bptt` but changing at every batch. If `backwards` is True, reverses the original text.

```
show_doc(LanguageModelLoader.batchify, doc_string=False)
```

batchify

batchify(data:ndarray) -> LongTensor [source]

Called at the inialization to create the big array of text ids from the `data` array.

```
show_doc(LanguageModelLoader.get_batch)
```

get_batch

get_batch(i:int, seq_len:int) -> Tuple[LongTensor, LongTensor]

Create a batch at `i` of a given `seq_len`. [source]

Classifier data

When preparing the data for a classifier, we keep the different texts separate, which poses another challenge for the creation of batches: since they don't all have the same length, we can't easily collate them together in batches. To help with this we use two different techniques: - padding: each text is padded with the `PAD` token to get all the ones we picked to the same size - sorting the texts (ish): to avoid having together a very long text with a very short one (which would then have a lot of `PAD` tokens), we regroup the texts by order of length. For the training set, we still add some randomness to avoid showing the same batches at every step of the training.

Here is an example of batch with padding (the padding index is 1, and the padding is applied before the sentences start).

```
data = get_imdb(classifier=True)
iter_dl = iter(data.train_dl)
_ = next(iter_dl)
x,y = next(iter_dl)
x[:20,-10:]
```

```
tensor([[   1,    1,    1,    1,    1,    1,    1,    1,    1,    1],
        [   1,    1,    1,    1,    1,    1,    1,    1,    1,    1],
        [   1,    1,    1,    1,    1,    1,    1,    1,    1,    1],
        [   1,    1,    1,    1,    1,    1,    1,    1,    1,    1],
        [  42,   42,   42,   42,    1,    1,    1,    1,    1,    1],
        [  43,   43,   43,   43,   42,   42,   42,    1,    1,    1],
        [  44,   44,   44,   44,   43,   43,   43,   42,    1,    1],
        [  39,   39,   39,   39,   44,   44,   44,   43,   42,   42],
        [  19,    0,   12,  224,   39,   39,   39,   44,   43,   43],
        [  15,  772,  307, 2118,   12,  111, 5128,   39,   44,   44],
```

```
[2753,    5, 1809, 1119,  661, 1224,   53,   14,   39,   39],
[5972,    0,  891,    3,  792,    4,  315,   25,   14,   14],
[  15,    0,    4,   19,    6,   24,  344,    9,   22,   17],
[   9,  249,   12,   12,  218,    2,  303,   53,   17,    6],
[   6,   64,  143,  156,   53,   25,   11,    6,    2,  111],
[2284,    0,   13,   13,    2,  261, 1838,  212,  106,   22],
[   7, 3797,    0,    4,  129,  186,   52,  271,  722,   13],
[3284,   46,  160,   24,   11, 1704,  153,    8,   22,   78],
[  18,    6,    9,  298,    2,   48,    8,  244,  163,    6],
[   0,    0,  484,    0,  103,   54,  657,    8,  654,  209]],
device='cuda:0')
```

All of this is done behind the scenes when calling the following in a `text_data` function

```
show_doc(classifier_data, doc_string=False)
```

`classifier_data`

> `classifier_data`(`datasets`:`Collection`[`TextDataset`], `path`:`PathOrStr`, `kwargs`) -> `DataBunch` [source]

Create a `DataBunch` in `path` from the `datasets` for language modelling. The kwargs are passed to the next classes and can contain: the batchsize `bs`, the `bptt`, the padding index `pad_idx` and wether or not to apply the `pad_first`.

```
show_doc(SortSampler, doc_string=False)
```

class SortSampler

> `SortSampler`(`data_source`:`NPArrayList`, `key`:`KeyFunc`) :: `Sampler` [source]

pytorch `Sampler` to batchify the `data_source` by order of length of the texts. Used for the validation and (if applicable) the test set.

```
show_doc(SortishSampler, doc_string=False)
```

class SortishSampler

> `SortishSampler`(`data_source`:`NPArrayList`, `key`:`KeyFunc`, `bs`:`int`) :: `Sampler` [source]

pytorch `Sampler` to batchify with size `bs` the `data_source` by order of length of the texts with a bit of randomness. Used for the training set.

```
show_doc(pad_collate, doc_string=False)
```

pad_collate

> pad_collate(samples:BatchSamples, pad_idx:int=1, pad_first:bool=True) -> Tuple[LongTensor, LongTensor] [source]

Function used by the pytorch DataLoader to collate the samples in batches while adding padding with pad_idx. If pad_first is True, padding is applied at the beginning (before the sentence starts) otherwise it's applied at the end.

Undocumented Methods - Methods moved below this line will intentionally be hidden

```
TextMtd = IntEnum('TextMtd', 'CSV TOK IDS')
```

[source]

```
show_doc(standard_data)
```

standard_data

> standard_data(datasets:Collection[DatasetBase], path:PathOrStr, kwargs) -> DataBunch

Simply create a DataBunch from the datasets. [source]

New Methods - Please document or move to the undocumented section

```
show_doc(read_classes)
```

read_classes

> read_classes(fname) [source]

Implementation of the language models

```
from fastai.gen_doc.nbdoc import *
from fastai.text.models import *
from fastai import *
```

This module fully implements the AWD-LSTM from Stephen Merity et al. The main idea of the article is to use a RNN with dropout everywhere, but in an intelligent way. There is a difference with the usual dropout, which is why you'll see a RNNDropout module: we zero things, as is usual in dropout, but we always zero the same thing according to the sequence dimension (which is the

first dimension in pytorch). This ensures consistency when updating the hidden state through the whole sentences/articles.

This being given, there are five different dropouts in the AWD-LSTM: - the first one, embedding dropout, is applied when we look the ids of our tokens inside the embedding matrix (to transform them from numbers to a vector of float). We zero some lines of it, so random ids are sent to a vector of zeros instead of being sent to their embedding vector. - the second one, input dropout, is applied to the result of the embedding with dropout. We forget random pieces of the embedding matrix (but as stated in the last paragraph, the same ones in the sequence dimension). - the third one is the weight dropout. It's the trickiest to implement as we randomly replace by 0s some weights of the hidden-to-hidden matrix inside the RNN: this needs to be done in a way that ensure the gradients are still computed and the initial weights still updated. - the fourth one is the hidden dropout. It's applied to the output of one of the layers of the RNN before it's used as input of the next layer (again same coordinates are zeroed in the sequence dimension). This one isn't applied to the last output, but rather… - the fifth one is the output dropout, it's applied to the last output of the model (and like the others, it's applied the same way through the first dimension).

Basic functions to get a model

```
show_doc(get_language_model, doc_string=False)
```

get_language_model

> `get_language_model(vocab_sz:int, emb_sz:int, n_hid:int, n_layers:int, pad_token:int, tie_weights:bool=True, qrnn:bool=False, bias:bool=True, bidir:bool=False, output_p:float=0.4, hidden_p:float=0.2, input_p:float=0.6, embed_p:float=0.1, weight_p:float=0.5) -> Module` [source]

Creates an AWD-LSTM with a first embedding of `vocab_sz` by `emb_sz`, a hidden size of `n_hid`, RNNs with `n_layers` that can be bidirectional if `bidir` is True. The last RNN as an output size of `emb_sz` so that we can use the same decoder as the encoder if `tie_weights` is True. The decoder is a `Linear` layer with or without `bias`. If `qrnn` is set to True, we use [QRNN cells] instead of LSTMS. `pad_token` is the token used for padding.

`embed_p` is used for the embedding dropout, `input_p` is used for the input dropout, `weight_p` is used for the weight dropout, `hidden_p` is used for the hidden dropout and `output_p` is used for the output dropout.

Note that the model returns a list of three things, the actual output being the first, the two others being the intermediate hidden states before and after dropout (used by the `RNNTrainer`). Most loss functions expect one out-

put, so you should use a Callback to remove the other two if you're not using `RNNTrainer`.

```
show_doc(get_rnn_classifier, doc_string=False)
```

`get_rnn_classifier`

> `get_rnn_classifier(bptt:int, max_seq:int, n_class:int, vocab_sz:int, emb_sz:int, n_hid:int, n_layers:int, pad_token:int, layers:Collection[int], drops:Collection[float], bidir:bool=False, qrnn:bool=False, hidden_p:float=0.2, input_p:float=0.6, embed_p:float=0.1, weight_p:float=0.5) -> Module` [source]

Creates a RNN classifier with a encoder taken from an AWD-LSTM with arguments `vocab_sz`, `emb_sz`, `n_hid`, `n_layers`, `bias`, `bidir`, `qrnn`, `pad_token` and the dropouts parameters. This encoder is fed the sequence by successive bits of size `bptt` and we only keep the last `max_seq` outputs for the pooling layers.

The decoder use a concatenation of the last outputs, a `MaxPooling` of all the ouputs and an `AveragePooling` of all the outputs. It then uses a list of `BatchNorm`, `Dropout`, `Linear`, `ReLU` blocks (with no `ReLU` in the last one), using a first layer size of `3*emb_sz` then follwoing the numbers in `n_layers` to stop at `n_class`. The dropouts probabilities are read in `drops`.

Note that the model returns a list of three things, the actual output being the first, the two others being the intermediate hidden states before and after dropout (used by the `RNNTrainer`). Most loss functions expect one output, so you should use a Callback to remove the other two if you're not using `RNNTrainer`.

Basic NLP modules

On top of the pytorch or the fastai `layers`, the language models use some custom layers specific to NLP.

```
show_doc(EmbeddingDropout, doc_string=False, title_level=3)
```

`class EmbeddingDropout`

> `EmbeddingDropout(emb:Module, embed_p:float) :: Module` [source]

Applies a dropout with probability `embed_p` to an embedding layer `emb` in training mode. Each row of the embedding matrix has a probability `embed_p` of being replaced by zeros while the others are rescaled accordingly.

```
enc = nn.Embedding(100, 7, padding_idx=1)
enc_dp = EmbeddingDropout(enc, 0.5)
```

```
tst_input = torch.randint(0,100,(8,))
enc_dp(tst_input)
```

```
tensor([[-0.0000,  0.0000,  0.0000,  0.0000,  0.0000,  0.0000, -0.0000],
        [-0.0000, -0.0000,  0.0000,  0.0000,  0.0000, -0.0000, -0.0000],
        [-1.2322, -2.9648, -2.4237, -1.8529,  1.6633, -1.8501,  1.1157],
        [ 0.3169,  0.0026,  2.9233, -2.6387, -1.9049,  0.1821, -2.8376],
        [ 0.0000,  0.0000, -0.0000,  0.0000, -0.0000, -0.0000, -0.0000],
        [-0.5815, -0.0366,  0.5165,  0.3036, -0.2163,  3.2384,  3.1961],
        [-1.2322, -2.9648, -2.4237, -1.8529,  1.6633, -1.8501,  1.1157],
        [-0.0000, -0.0000, -0.0000, -0.0000, -0.0000,  0.0000, -0.0000]],
       grad_fn=<EmbeddingBackward>)
```

```
show_doc(RNNDropout, doc_string=False, title_level=3)
```

class RNNDropout

> RNNDropout(p:float=0.5) :: Module [source]

Applies a dropout with probability `p` consistently over the first dimension in training mode.

```
dp = RNNDropout(0.3)
tst_input = torch.randn(3,3,7)
tst_input, dp(tst_input)
```

```
(tensor([[[-0.1608, -0.4899, -1.1367, -0.6127, -0.7680, -2.0614, -0.4157],
          [-0.0191,  0.7016, -1.0082, -1.3032,  0.4533, -0.4507,  2.0596],
          [-0.6750,  0.2974, -0.1997,  0.3986, -0.3976,  1.8443,  0.9806]],

         [[ 0.0124, -1.0716,  1.4654, -0.0578,  0.8262, -0.8311,  0.1116],
          [-0.1480, -0.0049, -0.3177,  0.6667,  1.0542,  0.5257,  0.7902],
          [ 0.4894,  0.1988, -1.4117,  0.6704,  1.2137,  0.3384, -0.2210]],

         [[ 1.3769,  0.4947, -0.1429, -0.2220, -1.9291, -0.9767, -1.5125],
          [ 0.8084,  0.9601, -0.7518, -0.8468, -1.9733, -1.0515, -0.4144],
          [ 1.1245,  1.3381,  1.2535,  0.1040,  0.2260,  0.2125, -1.7326]]]),
 tensor([[[-0.2297, -0.6999, -1.6239, -0.8753, -0.0000, -2.9449, -0.5939],
          [-0.0272,  1.0023, -0.0000, -1.8618,  0.6476, -0.6438,  0.0000],
          [-0.9642,  0.4248, -0.0000,  0.0000, -0.0000,  2.6348,  0.0000]],

         [[ 0.0178, -1.5308,  2.0934, -0.0825,  0.0000, -1.1873,  0.1594],
          [-0.2115, -0.0071, -0.0000,  0.9525,  1.5061,  0.7510,  0.0000],
          [ 0.6992,  0.2840, -0.0000,  0.0000,  0.0000,  0.4834, -0.0000]],

         [[ 1.9670,  0.7066, -0.2041, -0.3171, -0.0000, -1.3953, -2.1607],
          [ 1.1548,  1.3716, -0.0000, -1.2097, -2.8190, -1.5022, -0.0000],
```

```
       [ 1.6065,  1.9116,  0.0000,  0.0000,  0.0000,  0.3036, -0.0000]]]))
```

```
show_doc(WeightDropout, doc_string=False, title_level=3)
```

class WeightDropout

> WeightDropout(module:Module, weight_p:float, layer_names:StrList=['weight_hh_l0']) :: Module [source]

Applies dropout of probability `weight_p` to the layers in `layer_names` of `module` in training mode. A copy of those weights is kept so that the dropout mask can change at every batch.

```
module = nn.LSTM(5, 2)
dp_module = WeightDropout(module, 0.4)
getattr(dp_module.module, 'weight_hh_l0')
```

```
Parameter containing:
tensor([[-0.2749,  0.2294],
        [ 0.6515,  0.6442],
        [-0.6894,  0.1142],
        [ 0.3626,  0.3325],
        [-0.6264,  0.0868],
        [-0.5230,  0.5899],
        [ 0.2907, -0.4919],
        [-0.0321,  0.3461]], requires_grad=True)
```

It's at the beginning of a forward pass that the dropout is applied to the weights.

```
tst_input = torch.randn(4,20,5)
h = (torch.zeros(1,20,2), torch.zeros(1,20,2))
x,h = dp_module(tst_input,h)
getattr(dp_module.module, 'weight_hh_l0')
```

```
tensor([[-0.4582,  0.3823],
        [ 0.0000,  0.0000],
        [-1.1490,  0.1904],
        [ 0.0000,  0.5542],
        [-0.0000,  0.0000],
        [-0.8717,  0.0000],
        [ 0.4845, -0.8198],
        [-0.0535,  0.0000]], grad_fn=<MulBackward0>)
```

```
show_doc(SequentialRNN, doc_string=False, title_level=3)
```

class SequentialRNN

> SequentialRNN(args) :: Sequential [source]

Create a `Sequentiall` module with `args` that has a `reset` function.

```
show_doc(SequentialRNN.reset)
```

`reset`

> `reset`() [source]

Call the `reset` function of `self.children` (if they have one).

```
show_doc(dropout_mask, doc_string=False)
```

`dropout_mask`

> `dropout_mask`(`x`:`Tensor`, `sz`:`Collection`[`int`], `p`:`float`) [source]

Create a dropout mask of size `sz`, the same type as `x` and probability `p`.

```
tst_input = torch.randn(3,3,7)
dropout_mask(tst_input, (3,7), 0.3)
```

```
tensor([[1.4286, 1.4286, 0.0000, 0.0000, 0.0000, 1.4286, 0.0000],
        [1.4286, 0.0000, 1.4286, 0.0000, 1.4286, 0.0000, 0.0000],
        [1.4286, 1.4286, 1.4286, 1.4286, 1.4286, 1.4286, 0.0000]])
```

Such a mask is then expanded in the sequence length dimension and multiplied by the input to do an `RNNDropout`.

Language model modules

```
show_doc(RNNCore, doc_string=False, title_level=3)
```

`class RNNCore`

> `RNNCore`(`vocab_sz`:`int`, `emb_sz`:`int`, `n_hid`:`int`, `n_layers`:`int`, `pad_token`:`int`, `bidir`:`bool`=`False`, `hidden_p`:`float`=`0.2`, `input_p`:`float`=`0.6`, `embed_p`:`float`=`0.1`, `weight_p`:`float`=`0.5`, `qrnn`:`bool`=`False`) :: `Module` [source]

Create an AWD-LSTM encoder with an embedding layer of `vocab_sz` by `emb_sz`, a hidden size of `n_hid`, `n_layers` layers. `pad_token` is passed to the `Embedding`, if `bidir` is True, the model is bidirectional. If `qrnn` is True, we use QRNN cells instead of LSTMs. Dropouts are `embed_p`, `input_p`, `weight_p` and `hidden_p`.

```
show_doc(RNNCore.reset)
```

reset

`reset`()

Reset the hidden states. [source]

```
show_doc(LinearDecoder, doc_string=False, title_level=3)
```

class LinearDecoder

`LinearDecoder`(`n_out`:`int`, `n_hid`:`int`, `output_p`:`float`, `tie_encoder`:`Module`=`None`, `bias`:`bool`=`True`) :: `Module` [source]

Create a the decoder to go on top of an `RNNCore` encoder and create a language model. `n_hid` is the dimension of the last hidden state of the encoder, `n_out` the size of the output. Dropout of `output_p` is applied. If a `tie_encoder` is passed, it will be used for the weights of the linear layer, that will have `bias` or not.

Classifier modules

```
show_doc(MultiBatchRNNCore, doc_string=False, title_level=3)
```

class MultiBatchRNNCore

`MultiBatchRNNCore`(`bptt`:`int`, `max_seq`:`int`, `args`, `kwargs`) :: `RNNCore` [source]

Wrap an `RNNCore` to make it process full sentences: text is passed by chunks of sequence length `bptt` and only the last `max_seq` outputs are kept for the next layer. `args` and `kwargs` are passed to the `RNNCore`.

```
show_doc(MultiBatchRNNCore.concat)
```

concat

`concat`(`arrs`:`Collection`[`Tensor`]) -> `Tensor`

Concatenate the `arrs` along the batch dimension. [source]

```
show_doc(PoolingLinearClassifier, doc_string=False, title_level=3)
```

class PoolingLinearClassifier

`PoolingLinearClassifier`(`layers`:`Collection`[`int`], `drops`:`Collection`[`float`]) :: `Module` [source]

Create a linear classifier that sits on an `RNNCore` encoder. The last output, `MaxPooling` of all the outputs and `AvgPooling` of all the outputs are concatenated, then blocks of `bn_drop_lin` are stacked, according to the values in `layers` and `drops`.

```
show_doc(PoolingLinearClassifier.pool, doc_string=False)
```

`pool`

> `pool`(`x`:`Tensor`, `bs`:`int`, `is_max`:`bool`) [source]

Pool `x` (of batch size `bs`) along the batch dimension. `is_max` decides if we do an `AvgPooling` or a `MaxPooling`.

Undocumented Methods - Methods moved below this line will intentionally be hidden

```
show_doc(WeightDropout.forward)
```

`forward`

> `forward`(`args`:`Classes`)

Should be overridden by all subclasses.

.. note:: Although the recipe for forward pass needs to be defined within this function, one should call the :class:`Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them. [source]

```
show_doc(RNNCore.forward)
```

`forward`

> `forward`(`input`:`LongTensor`) -> `Tuple`[`Tensor`, `Tensor`]

Should be overridden by all subclasses.

.. note:: Although the recipe for forward pass needs to be defined within this function, one should call the :class:`Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them. [source]

```
show_doc(EmbeddingDropout.forward)
```

`forward`

> `forward(words:LongTensor, scale:Optional[float]=None) -> Tensor`

Should be overridden by all subclasses.

.. note:: Although the recipe for forward pass needs to be defined within this function, one should call the :class:`Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them. [source]

```
show_doc(RNNDropout.forward)
```

`forward`

> `forward(x:Tensor) -> Tensor`

Should be overridden by all subclasses.

.. note:: Although the recipe for forward pass needs to be defined within this function, one should call the :class:`Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them. [source]

New Methods - Please document or move to the undocumented section

Tabular data

```
%reload_ext autoreload
%autoreload 2
```

```
from fastai.gen_doc.nbdoc import *
from fastai.tabular import *
from fastai import *
from fastai.docs import *
from fastai.tabular.models import *
```

`tabular` contains all the necessary classes to deal with tabular data, across two modules: - `tabular.transform`: defines the `TabularTransform` class to help with preprocessing; - `tabular.data`: defines the `TabularDataset` that handles that data, as well as the methods to quickly get a `DataBunch`.

To create a model, you'll need to use `models.tabular`. See below for and end-to-end example using all these modules.

Preprocessing tabular data

Tabular data usually comes in the form of a delimited file (such as CSV) containing variables of different kinds: text/category, numbers, and perhaps some missing values. The example we'll work with in this section is a sample of the adult dataset which has some census information on individuals. We'll use it to train a model to predict whether salary is greater than $50k or not.

```
untar_data(ADULT_PATH)
ADULT_PATH
```

```
PosixPath('../data/adult_sample')
```

```
df = pd.read_csv(ADULT_PATH/'adult.csv')
df.head()
```

```
<tr style="text-align: right;">
  <th></th>
  <th>age</th>
  <th>workclass</th>
  <th>fnlwgt</th>
  <th>education</th>
  <th>education-num</th>
  <th>marital-status</th>
  <th>occupation</th>
  <th>relationship</th>
  <th>race</th>
  <th>sex</th>
  <th>capital-gain</th>
  <th>capital-loss</th>
  <th>hours-per-week</th>
  <th>native-country</th>
  <th>&gt;=50k</th>
</tr>
```

```
<tr>
  <th>0</th>
  <td>49</td>
  <td>Private</td>
  <td>101320</td>
  <td>Assoc-acdm</td>
  <td>12.0</td>
  <td>Married-civ-spouse</td>
  <td>NaN</td>
  <td>Wife</td>
  <td>White</td>
  <td>Female</td>
  <td>0</td>
```

```
  <td>1902</td>
  <td>40</td>
  <td>United-States</td>
  <td>1</td>
</tr>
<tr>
  <th>1</th>
  <td>44</td>
  <td>Private</td>
  <td>236746</td>
  <td>Masters</td>
  <td>14.0</td>
  <td>Divorced</td>
  <td>Exec-managerial</td>
  <td>Not-in-family</td>
  <td>White</td>
  <td>Male</td>
  <td>10520</td>
  <td>0</td>
  <td>45</td>
  <td>United-States</td>
  <td>1</td>
</tr>
<tr>
  <th>2</th>
  <td>38</td>
  <td>Private</td>
  <td>96185</td>
  <td>HS-grad</td>
  <td>NaN</td>
  <td>Divorced</td>
  <td>NaN</td>
  <td>Unmarried</td>
  <td>Black</td>
  <td>Female</td>
  <td>0</td>
  <td>0</td>
  <td>32</td>
  <td>United-States</td>
  <td>0</td>
</tr>
<tr>
  <th>3</th>
  <td>38</td>
  <td>Self-emp-inc</td>
  <td>112847</td>
```

```
  <td>Prof-school</td>
  <td>15.0</td>
  <td>Married-civ-spouse</td>
  <td>Prof-specialty</td>
  <td>Husband</td>
  <td>Asian-Pac-Islander</td>
  <td>Male</td>
  <td>0</td>
  <td>0</td>
  <td>40</td>
  <td>United-States</td>
  <td>1</td>
</tr>
<tr>
  <th>4</th>
  <td>42</td>
  <td>Self-emp-not-inc</td>
  <td>82297</td>
  <td>7th-8th</td>
  <td>NaN</td>
  <td>Married-civ-spouse</td>
  <td>Other-service</td>
  <td>Wife</td>
  <td>Black</td>
  <td>Female</td>
  <td>0</td>
  <td>0</td>
  <td>50</td>
  <td>United-States</td>
  <td>0</td>
</tr>
```

Here all the information that will form our input is in the 14 first columns, and the dependant variable is the last column. We will split our input between two types of variables: categoricals and continuous. - Categorical variables will be replaced by a category, a unique id that identifies them, before passing through an embedding layer. - Continuous variables will be normalized then directly fed to the model.

Another thing we need to handle are the missing values: our model isn't going to like receiving NaNs so we should remove them in a smart way. All of this preprocessing is done by `TabularTransform` objects and `TabularDataset`.

We can define a bunch of Transforms that will be applied to our variables. Here we transform all categorical variables into categories, and we replace missing values for continuous variables by the median of the corresponding column.

```
tfms = [FillMissing, Categorify]
```

We split our data into training and validation sets.

```
train_df, valid_df = df[:-2000].copy(),df[-2000:].copy()
```

First let's split our variables between categoricals and continuous (we can ignore the dependant variable at this stage). fastai will assume all variables that aren't dependent or categorical are continuous, unless we explicitly pass a list to the `cont_names` parameter when constructing our `DataBunch`.

```
dep_var = '>=50k'
cat_names = ['workclass', 'education', 'marital-status', 'occupation', 'relationship', 'race',
```

Now we're ready to pass these details to `tabular_data_from_df` to create the `DataBunch` that we'll use for training.

```
data = tabular_data_from_df(ADULT_PATH, train_df, valid_df, dep_var, tfms=tfms, cat_names=cat_
print(data.train_ds.cont_names)  # `cont_names` defaults to: set(df)-set(cat_names)-{dep_var}
```

```
['hours-per-week', 'age', 'capital-gain', 'fnlwgt', 'education-num', 'capital-loss']
```

We can grab a mini-batch of data and take a look (note that `to_np` here converts from pytorch tensor to numpy):

```
(cat_x,cont_x),y = next(iter(data.train_dl))
for o in (cat_x, cont_x, y): print(to_np(o[:5]))
```

```
[[ 1 16  5  1  4  5  2 40  1]
 [ 5 12  3  2  1  5  2 40  1]
 [ 5 16  3  4  1  1  2 40  1]
 [ 8 10  3  5  1  5  2 40  1]
 [ 3 10  3  2  1  5  2 40  1]]
[[-0.03578891 -1.5102453  -0.14592563  1.0260849  -0.02974687 -0.21676035]
 [ 0.36816576  0.24721362 -0.14592563 -1.0722153  -0.42159945 -0.21676035]
 [-0.03578891  1.5653079  -0.14592563 -0.39026728 -0.02974687 -0.21676035]
 [-0.03578891  1.272398   -0.14592563 -0.13906312  1.1458108  -0.21676035]
 [-0.03578891 -0.11892366  0.83429855 -0.27299133  1.1458108  -0.21676035]]
[0 0 0 0 1]
```

After being processed in `TabularDataset`, the categorical variables are replaced by ids and the continuous variables are normalized. The codes corresponding to categorical variables are all put together, as are all the continuous variables.

Defining a model

Once we have our data ready in a `DataBunch`, we just need to create a model to then define a `Learner` and start training. The fastai library has a flexible and powerful `TabularModel` in `models.tabular`. To use that function, we just need to specify the embedding sizes for each of our categorical variables.

```
learn = get_tabular_learner(data, layers=[200,100], emb_szs={'native-country': 10}, metrics=ac
```

```
learn.fit_one_cycle(1, 1e-2)
```

```
Total time: 00:04
epoch  train loss  valid loss  accuracy
0      0.349120    0.349844    0.839500  (00:04)
```

Tabular data preprocessing

```
from fastai.gen_doc.nbdoc import *
from fastai.tabular import *
from fastai.docs import *
```

Overview

This package contains the basic class to define a transformation for preprocessing dataframes of tabular data, as well as basic `TabularTransform`. Preprocessing includes things like - replacing non-numerical variables by categories, then their ids, - filling missing values, - normalizing continuous variables.

In all those steps we have to be careful to use the correspondance we decide on our training set (which id we give to each category, what is the value we put for missing data, or how the mean/std we use to normalize) on our validation or test set. To deal with this, we use a speciall class called `TabularTransform`.

The data used in this document page is a subset of the adult dataset. It gives a certain amount of data on individuals to train a model to predict wether their salary is greater than $50k or not.

```
df = get_adult()
train_df, valid_df = df[:800].copy(),df[800:].copy()
train_df.head()
```

```
<tr style="text-align: right;">
  <th></th>
  <th>age</th>
  <th>workclass</th>
  <th>fnlwgt</th>
  <th>education</th>
  <th>education-num</th>
  <th>marital-status</th>
  <th>occupation</th>
  <th>relationship</th>
  <th>race</th>
  <th>sex</th>
  <th>capital-gain</th>
  <th>capital-loss</th>
```

```
  <th>hours-per-week</th>
  <th>native-country</th>
  <th>&gt;=50k</th>
</tr>

<tr>
  <th>0</th>
  <td>49</td>
  <td>Private</td>
  <td>101320</td>
  <td>Assoc-acdm</td>
  <td>12.0</td>
  <td>Married-civ-spouse</td>
  <td>NaN</td>
  <td>Wife</td>
  <td>White</td>
  <td>Female</td>
  <td>0</td>
  <td>1902</td>
  <td>40</td>
  <td>United-States</td>
  <td>1</td>
</tr>
<tr>
  <th>1</th>
  <td>44</td>
  <td>Private</td>
  <td>236746</td>
  <td>Masters</td>
  <td>14.0</td>
  <td>Divorced</td>
  <td>Exec-managerial</td>
  <td>Not-in-family</td>
  <td>White</td>
  <td>Male</td>
  <td>10520</td>
  <td>0</td>
  <td>45</td>
  <td>United-States</td>
  <td>1</td>
</tr>
<tr>
  <th>2</th>
  <td>38</td>
  <td>Private</td>
  <td>96185</td>
```

```
  <td>HS-grad</td>
  <td>NaN</td>
  <td>Divorced</td>
  <td>NaN</td>
  <td>Unmarried</td>
  <td>Black</td>
  <td>Female</td>
  <td>0</td>
  <td>0</td>
  <td>32</td>
  <td>United-States</td>
  <td>0</td>
</tr>
<tr>
  <th>3</th>
  <td>38</td>
  <td>Self-emp-inc</td>
  <td>112847</td>
  <td>Prof-school</td>
  <td>15.0</td>
  <td>Married-civ-spouse</td>
  <td>Prof-specialty</td>
  <td>Husband</td>
  <td>Asian-Pac-Islander</td>
  <td>Male</td>
  <td>0</td>
  <td>0</td>
  <td>40</td>
  <td>United-States</td>
  <td>1</td>
</tr>
<tr>
  <th>4</th>
  <td>42</td>
  <td>Self-emp-not-inc</td>
  <td>82297</td>
  <td>7th-8th</td>
  <td>NaN</td>
  <td>Married-civ-spouse</td>
  <td>Other-service</td>
  <td>Wife</td>
  <td>Black</td>
  <td>Female</td>
  <td>0</td>
  <td>0</td>
  <td>50</td>
```

```
  <td>United-States</td>
  <td>0</td>
</tr>
```

We see it contains numerical variables (like `age` or `education-num`) as well as categorical ones (like `workclass` or `relationship`). The original dataset is clean, but we removed a few values to give examples of dealing with missing variables.

```
cat_names = ['workclass', 'education', 'marital-status', 'occupation', 'relationship', 'race',
cont_names = ['age', 'fnlwgt', 'education-num', 'capital-gain', 'capital-loss', 'hours-per-wee
```

Transforms for tabular data

```
show_doc(TabularTransform, doc_string=False)
```

class TabularTransform

> `TabularTransform`(`cat_names`:`StrList`, `cont_names`:`StrList`) [source]

Base class for creating transforms for dataframes with categorical variables `cat_names` and continuous variables `cont_names`. Note that any column not in one of those lists won't be touched.

```
show_doc(TabularTransform.__call__)
```

__call__

> `__call__`(`df`:`DataFrame`, `test`:`bool`=`False`)

Apply the correct function to `df` depending on `test`. [source]

This simply calls `apply_test` if `test` or `apply_train` otherwise. Those functions apply the changes in place.

```
show_doc(Categorify.apply_train, doc_string=False)
```

apply_train

> `apply_train`(`df`:`DataFrame`) [source]

Must be implemented by an inherited class with the desired transformation logic.

```
show_doc(Categorify.apply_test, doc_string=False)
```

apply_test

apply_test(df:DataFrame) [source]

If not implemented by an inherited class, defaults to calling apply_train.

The following TabularTransform are implemented in the fastai library. Note that the replacement from categories to codes as well as the normalization of continuous variables are automatically done in a TabularDataset.

```
show_doc(Categorify, doc_string=False)
```

class Categorify

Categorify(cat_names:StrList, cont_names:StrList) :: TabularTransform [source]

Changes the categorical variables in cat_names in categories. Variables in cont_names aren't affected.

```
show_doc(Categorify.apply_train, doc_string=False)
```

apply_train

apply_train(df:DataFrame) [source]

Transforms the variable in the cat_names columns in categories. The category codes are the unique values in these columns.

```
show_doc(Categorify.apply_test, doc_string=False)
```

apply_test

apply_test(df:DataFrame) [source]

Transforms the variable in the cat_names columns in categories. The category codes are the ones used for the training set, new categories are replaced by NaN.

```
tfm = Categorify(cat_names, cont_names)
tfm(train_df)
tfm(valid_df, test=True)
```

Since we haven't changed the categories by their codes, nothing visible has changed in the dataframe yet, but we can check that the variables are now categorical and view their corresponding codes.

```
train_df['workclass'].cat.categories
```

```
Index([' ?', ' Federal-gov', ' Local-gov', ' Private', ' Self-emp-inc',
       ' Self-emp-not-inc', ' State-gov', ' Without-pay'],
      dtype='object')
```

The test set will be given the same category codes as the training set.

```
valid_df['workclass'].cat.categories
```

```
Index([' ?', ' Federal-gov', ' Local-gov', ' Private', ' Self-emp-inc',
       ' Self-emp-not-inc', ' State-gov', ' Without-pay'],
      dtype='object')
```

```
show_doc(FillMissing, doc_string=False)
```

class FillMissing

> FillMissing(cat_names:StrList, cont_names:StrList, fill_strategy:FillStrategy=<FillStrateg
> 1>, add_col:bool=True, fill_val:float=0.0) :: TabularTransform
> [source]

Transform that fills the missing values in `cont_names`. `cat_names` variables are left untouched (their missing value will be raplced by code 0 in the `TabularDataset`). `fill_strategy` is adopted to replace those nans and if `add_col` is True, whenever a column `c` has missing values, a column named `c_nan` is added and flags the line where the value was missing. The `fill_strategy` can be: - `FillStrategy.MEDIAN`: nans are replaced by the median value of the column, - `FillStrategy.COMMON`: nans are replaced by the most common value of the column, - `FillStrategy.CONSTANT`: nans are replaced by `fill_val`.

```
show_doc(FillMissing.apply_train, doc_string=False)
```

apply_train

> apply_train(df:DataFrame) [source]

Fills the missing values in the `cont_names` columns.

```
show_doc(FillMissing.apply_test, doc_string=False)
```

apply_test

> apply_test(df:DataFrame) [source]

Fills the missing values in the `cont_names` columns with the ones picked during train.

```
train_df[cont_names].head()
```

```
<tr style="text-align: right;">
  <th></th>
  <th>age</th>
  <th>fnlwgt</th>
```

```
  <th>education-num</th>
  <th>capital-gain</th>
  <th>capital-loss</th>
  <th>hours-per-week</th>
</tr>

<tr>
  <th>0</th>
  <td>49</td>
  <td>101320</td>
  <td>12.0</td>
  <td>0</td>
  <td>1902</td>
  <td>40</td>
</tr>
<tr>
  <th>1</th>
  <td>44</td>
  <td>236746</td>
  <td>14.0</td>
  <td>10520</td>
  <td>0</td>
  <td>45</td>
</tr>
<tr>
  <th>2</th>
  <td>38</td>
  <td>96185</td>
  <td>NaN</td>
  <td>0</td>
  <td>0</td>
  <td>32</td>
</tr>
<tr>
  <th>3</th>
  <td>38</td>
  <td>112847</td>
  <td>15.0</td>
  <td>0</td>
  <td>0</td>
  <td>40</td>
</tr>
<tr>
  <th>4</th>
  <td>42</td>
  <td>82297</td>
```

```
  <td>NaN</td>
  <td>0</td>
  <td>0</td>
  <td>50</td>
</tr>
```

```
tfm = FillMissing(cat_names, cont_names)
tfm(train_df)
tfm(valid_df, test=True)
train_df[cont_names].head()
```

```
<tr style="text-align: right;">
  <th></th>
  <th>age</th>
  <th>fnlwgt</th>
  <th>education-num</th>
  <th>capital-gain</th>
  <th>capital-loss</th>
  <th>hours-per-week</th>
</tr>
```

```
<tr>
  <th>0</th>
  <td>49</td>
  <td>101320</td>
  <td>12.0</td>
  <td>0</td>
  <td>1902</td>
  <td>40</td>
</tr>
<tr>
  <th>1</th>
  <td>44</td>
  <td>236746</td>
  <td>14.0</td>
  <td>10520</td>
  <td>0</td>
  <td>45</td>
</tr>
<tr>
  <th>2</th>
  <td>38</td>
  <td>96185</td>
  <td>10.0</td>
  <td>0</td>
  <td>0</td>
  <td>32</td>
```

```
</tr>
<tr>
  <th>3</th>
  <td>38</td>
  <td>112847</td>
  <td>15.0</td>
  <td>0</td>
  <td>0</td>
  <td>40</td>
</tr>
<tr>
  <th>4</th>
  <td>42</td>
  <td>82297</td>
  <td>10.0</td>
  <td>0</td>
  <td>0</td>
  <td>50</td>
</tr>
```

Values issing in the `education-num` column are replaced by 10, which is the median of the column in `train_df`. Categorical variables are not changed, since `nan` is simply used as another category.

```
valid_df[cont_names].head()
```

```
<tr style="text-align: right;">
  <th></th>
  <th>age</th>
  <th>fnlwgt</th>
  <th>education-num</th>
  <th>capital-gain</th>
  <th>capital-loss</th>
  <th>hours-per-week</th>
</tr>
```

```
<tr>
  <th>800</th>
  <td>45</td>
  <td>96975</td>
  <td>10.0</td>
  <td>0</td>
  <td>0</td>
  <td>40</td>
</tr>
<tr>
  <th>801</th>
  <td>46</td>
```

```
  <td>192779</td>
  <td>10.0</td>
  <td>15024</td>
  <td>0</td>
  <td>60</td>
</tr>
<tr>
  <th>802</th>
  <td>36</td>
  <td>376455</td>
  <td>10.0</td>
  <td>0</td>
  <td>0</td>
  <td>38</td>
</tr>
<tr>
  <th>803</th>
  <td>25</td>
  <td>50053</td>
  <td>10.0</td>
  <td>0</td>
  <td>0</td>
  <td>45</td>
</tr>
<tr>
  <th>804</th>
  <td>37</td>
  <td>164526</td>
  <td>10.0</td>
  <td>0</td>
  <td>0</td>
  <td>40</td>
</tr>
```

Undocumented Methods - Methods moved below this line will intentionally be hidden

New Methods - Please document or move to the undocumented section

```
show_doc(TabularTransform.apply_train)
```

apply_train

`apply_train`(df:DataFrame)

Function applied to `df` if it's the train set. [source]

```
show_doc(TabularTransform.apply_test)
```

`apply_test`

`apply_test`(df:DataFrame)

Function applied to `df` if it's the test set. [source]

Tabular data handling

This module defines the main class to handle tabular data in the fastai library: `TabularDataset`. As always, there is also a helper function to quickly get your data.

To allow you to easily create a `Learner` for your data, it provides `get_tabular_learner`.

```
from fastai.gen_doc.nbdoc import *
from fastai.tabular import *
from fastai.docs import *
```

Quickly get the data in a `DataBunch`

The best way to quickly get your data in a DataBunch is to organize it in two (or three) dataframes. One for training, one for validation, and if you have it, one for test. Here we are interested in a subsample of the adult dataset.

```
df = get_adult()
train_df, valid_df = df[:800].copy(),df[800:].copy()
train_df.head()
```

```
<tr style="text-align: right;">
  <th></th>
  <th>age</th>
  <th>workclass</th>
  <th>fnlwgt</th>
  <th>education</th>
  <th>education-num</th>
  <th>marital-status</th>
  <th>occupation</th>
  <th>relationship</th>
  <th>race</th>
```

```
  <th>sex</th>
  <th>capital-gain</th>
  <th>capital-loss</th>
  <th>hours-per-week</th>
  <th>native-country</th>
  <th>&gt;=50k</th>
</tr>
<tr>
  <th>0</th>
  <td>49</td>
  <td>Private</td>
  <td>101320</td>
  <td>Assoc-acdm</td>
  <td>12.0</td>
  <td>Married-civ-spouse</td>
  <td>NaN</td>
  <td>Wife</td>
  <td>White</td>
  <td>Female</td>
  <td>0</td>
  <td>1902</td>
  <td>40</td>
  <td>United-States</td>
  <td>1</td>
</tr>
<tr>
  <th>1</th>
  <td>44</td>
  <td>Private</td>
  <td>236746</td>
  <td>Masters</td>
  <td>14.0</td>
  <td>Divorced</td>
  <td>Exec-managerial</td>
  <td>Not-in-family</td>
  <td>White</td>
  <td>Male</td>
  <td>10520</td>
  <td>0</td>
  <td>45</td>
  <td>United-States</td>
  <td>1</td>
</tr>
<tr>
  <th>2</th>
```

```
  <td>38</td>
  <td>Private</td>
  <td>96185</td>
  <td>HS-grad</td>
  <td>NaN</td>
  <td>Divorced</td>
  <td>NaN</td>
  <td>Unmarried</td>
  <td>Black</td>
  <td>Female</td>
  <td>0</td>
  <td>0</td>
  <td>32</td>
  <td>United-States</td>
  <td>0</td>
</tr>
<tr>
  <th>3</th>
  <td>38</td>
  <td>Self-emp-inc</td>
  <td>112847</td>
  <td>Prof-school</td>
  <td>15.0</td>
  <td>Married-civ-spouse</td>
  <td>Prof-specialty</td>
  <td>Husband</td>
  <td>Asian-Pac-Islander</td>
  <td>Male</td>
  <td>0</td>
  <td>0</td>
  <td>40</td>
  <td>United-States</td>
  <td>1</td>
</tr>
<tr>
  <th>4</th>
  <td>42</td>
  <td>Self-emp-not-inc</td>
  <td>82297</td>
  <td>7th-8th</td>
  <td>NaN</td>
  <td>Married-civ-spouse</td>
  <td>Other-service</td>
  <td>Wife</td>
  <td>Black</td>
  <td>Female</td>
```

```
  <td>0</td>
  <td>0</td>
  <td>50</td>
  <td>United-States</td>
  <td>0</td>
</tr>
```

```
cat_names = ['workclass', 'education', 'marital-status', 'occupation', 'relationship', 'race',
dep_var = '>=50k'
```

```
show_doc(tabular_data_from_df, doc_string=False)
```

tabular_data_from_df

> tabular_data_from_df(path, train_df:DataFrame, valid_df:DataFrame, dep_var:str, test_df:OptDataFrame=None, tfms:Optional[Collection[TabularTransform]]=None, cat_names:OptStrList=None, cont_names:OptStrList=None, stats:OptStats=None, log_output:bool=False, kwargs) -> DataBunch [source]

Creates a `DataBunch` in `path` from `train_df`, `valid_df` and optionally `test_df`. The dependent variable is `dep_var`, while the categorical and continuous variables are in the `cat_names` columns and `cont_names` columns respectively. If `cont_names` is None then we assume all variables that aren't dependent or categorical are continuous. The `TabularTransform` in `tfms` are applied to the dataframes as preprocessing, then the categories are replaced by their codes+1 (leaving 0 for `nan`) and the continuous variables are normalized. You can pass the `stats` to use for that step. If `log_output` is True, the dependant variable is replaced by its log.

Note that the transforms should be passed as `Callable`: the actual initialization with `cat_names` and `cont_names` is done inside.

```
tfms = [FillMissing, Categorify]
data = tabular_data_from_df(ADULT_PATH, train_df, valid_df, dep_var=dep_var, tfms=tfms, cat_na
```

You can then easily create a `Learner` for this data with `get_tabular_learner`.

```
show_doc(get_tabular_learner)
```

get_tabular_learner

> get_tabular_learner(data:DataBunch, layers:Collection[int], emb_szs:Dict[str, int]=None, metrics=None, kwargs)

Get a `Learner` using `data`, with `metrics`, including a `TabularModel` created using the remaining params. [source]

`emb_szs` is a `dict` mapping categorical column names to embedding sizes; you only need to pass sizes for columns where you want to override the default behaviour of the model.

The TabularDataset class

```
show_doc(TabularDataset, doc_string=False)
```

`class TabularDataset`

> `TabularDataset`(`df`:`DataFrame`, `dep_var`:`str`, `cat_names`:`OptStrList`=`None`, `cont_names`:`OptStrList`=`None`, `stats`:`OptStats`=`None`, `log_output`:`bool`=`False`) :: `DatasetBase` [source]

A dataset from `DataFrame` `df` with the dependent being the `dep_var` column, while the categorical and continuous variables are in the `cat_names` columns and `cont_names` columns respectively. Categories are replaced by their codes+1 (leaving 0 for `nan`) and the continuous variables are normalized. You can pass the `stats` to use for normalization; if none, then will be calculated from your data. If the flag `log_output` is True, the dependant variable is replaced by its log.

```
show_doc(TabularDataset.from_dataframe, doc_string=False)
```

`from_dataframe`

> `from_dataframe`(`df`:`DataFrame`, `dep_var`:`str`, `tfms`:`Optional`[`Collection`[`TabularTransform`]]=`None`, `cat_names`:`OptStrList`=`None`, `cont_names`:`OptStrList`=`None`, `stats`:`OptStats`=`None`, `log_output`:`bool`=`False`) -> `TabularDataset` [source]

Factory method to create a `TabularDataset` from `df`. The only difference from the constructor is that it gets a list `tfms` of `TabularTfm` that it applied before passing the dataframe to the class initialization.

Simple model for tabular data

```
from fastai.gen_doc.nbdoc import *
from fastai.tabular.models import TabularModel
```

```
show_doc(TabularModel, doc_string=False)
```

class TabularModel

> TabularModel(emb_szs:ListSizes, n_cont:int, out_sz:int, layers:Collection[int], ps:Collection[float]=None, emb_drop:float=0.0, y_range:OptRange=None, use_bn:bool=True) :: Module [source]

Create a model suitable for tabular data. `emb_szs` match each categorical variable size with an embedding size, `n_cont` is the number of continuous variables. The model consists of `Embedding` layers for the categorical variables, followed by a `Dropout` of `emb_drop`, and a `BatchNorm` for the continuous variables. The results are concatenated and followed by blocks of `BatchNorm`, `Dropout`, `Linear` and `ReLU` (the first block skip `BatchNorm` and `Dropout`, the last block skips the `ReLU`).

The sizes of the blocks are given in `layers` and the probabilities of the `Dropout` in `drops`. The last size is `out_sz`, and we add a last activation that is a sigmoid rescaled to cover `y_range` (if it's not `None`). Lastly, if `use_bn` is set to False, all `BatchNorm` layers are skipped except the one applied to the continuous variables.

Generally it's easiest to just create a learner with `get_tabular_learner`, which will automatically create a `TabularModel` for you.

Collaborative filtering

```
from fastai.gen_doc.nbdoc import *
from fastai.collab import *
from fastai.docs import *
from fastai import *
```

This package contains all the necessary functions to quickly train a model for a collaborative filtering task.

Overview

Collaborative filtering is when you're tasked to predict how much a user is going to like a certain item. The fastai library contains a `CollabFilteringDataset` class that will help you create datasets suitable for training, and a function `get_colab_learner` to build a simple model directly from a ratings table. Let's first see how we can get started before devling in the documentation.

For our example, we'll use a small subset of the MovieLens dataset. In there, we have to predict the rating a user gave a given movie (from 0 to 5). It comes in the form of a csv file where each line is the rating of a movie by a given person.

```
ratings = get_movie_lens()
ratings.head()
```

```
<tr style="text-align: right;">
  <th></th>
  <th>userId</th>
  <th>movieId</th>
  <th>rating</th>
  <th>timestamp</th>
</tr>
```

```
<tr>
  <th>0</th>
  <td>73</td>
  <td>1097</td>
  <td>4.0</td>
  <td>1255504951</td>
</tr>
<tr>
  <th>1</th>
  <td>561</td>
  <td>924</td>
  <td>3.5</td>
  <td>1172695223</td>
</tr>
<tr>
  <th>2</th>
  <td>157</td>
  <td>260</td>
  <td>3.5</td>
  <td>1291598691</td>
</tr>
<tr>
  <th>3</th>
  <td>358</td>
  <td>1210</td>
  <td>5.0</td>
  <td>957481884</td>
</tr>
<tr>
  <th>4</th>
  <td>130</td>
  <td>316</td>
  <td>2.0</td>
  <td>1138999234</td>
</tr>
```

We'll first turn the `userId` and `movieId` columns in category codes, so that we can replace them with their codes when it's time to feed them to an `Embedding` layer. This step would be even more important if our csv had names of users,

or names of items in it.

```
series2cat(ratings, 'userId','movieId')
```

Now that this step is done, we can directly create a `Learner` object:

```
learn = get_collab_learner(ratings, n_factors=50, pct_val=0.2, min_score=0., max_score=5.)
```

And the immediately begin training

```
learn.fit_one_cycle(5, 5e-3, wd=0.1)
```

```
Total time: 00:03
epoch  train loss  valid loss
0      2.371715    1.861580    (00:00)
1      1.082379    0.712514    (00:00)
2      0.731760    0.698491    (00:00)
3      0.606913    0.692382    (00:00)
4      0.569178    0.690507    (00:00)
```

```
show_doc(CollabFilteringDataset, doc_string=False)
```

class CollabFilteringDataset

> `CollabFilteringDataset`(`user`:`Series`, `item`:`Series`, `ratings`:`ndarray`) :: `DatasetBase` [source]

This is the basic class to buil a `Dataset` suitable for colaborative filtering. `user` and `item` should be categorical series that will be replaced with their codes internally and have the corresponding `ratings`. One of the factory methods will prepare the data in this format.

```
show_doc(CollabFilteringDataset.from_df, doc_string=False)
```

from_df

> `from_df`(`rating_df`:`DataFrame`, `pct_val`:`float`=`0.2`, `user_name`:`Optional`[`str`]=`None`, `item_name`:`Optional`[`str`]=`None`, `rating_name`:`Optional`[`str`]=`None`) -> `Tuple`[`ColabFilteringDataset`, `ColabFilteringDataset`] [source]

Takes a `rating_df` ans splits it randomly for train and test following `pct_val` (unless it's None). `user_name`, `item_name` and `rating_name` give the names of the corresponding columns (defaults to the first, the second and the third column).

```
show_doc(CollabFilteringDataset.from_csv, doc_string=False)
```

from_csv

> from_csv(csv_name:str, kwargs) -> Tuple[ColabFilteringDataset, ColabFilteringDataset] [source]

Opens the file in `csv_name` as a `DataFrame` and feeds it to `show_doc.from_df` with the `kwargs`.

Model and `Learner`

```
show_doc(EmbeddingDotBias, doc_string=False, title_level=3)
```

class EmbeddingDotBias

> EmbeddingDotBias(n_factors:int, n_users:int, n_items:int, min_score:float=None, max_score:float=None) :: Module [source]

Creates a simple model with `Embedding` weights and biases for `n_users` and `n_items`, with `n_factors` latent factors. Takes the dot product of the embeddings and adds the bias, then feed the result to a sigmoid rescaled to go from `min_score` to `max_score`.

```
show_doc(get_collab_learner, doc_string=False)
```

get_collab_learner

> get_collab_learner(ratings:DataFrame, n_factors:int, pct_val:float=0.2, user_name:Optional[str]=None, item_name:Optional[str]=None, rating_name:Optional[str]=None, test:DataFrame=None, metrics=None, min_score:float=None, max_score:float=None, loss_fn:LossFunction='mse_loss', kwargs) -> Learner [source]

Creates a `Learner` object built from the data in `ratings`, `pct_val`, `user_name`, `item_name`, `rating_name` to `CollabFilteringDataset`. Optionally, creates another `CollabFilteringDataset` for `test`. `kwargs` are fed to `DataBunch.create` with these datasets. The model is given by `EmbeddingDotBias` with `n_factors`, `min_score` and `max_score` (the numbers of users and items will be inferred from the data).

Undocumented Methods - Methods moved below this line will intentionally be hidden

```
show_doc(EmbeddingDotBias.forward)
```

```
jekyll_note('To get started with fastai, have a look at the <a href="/training">training overvie
```

Note: To get started with fastai, have a look at the training overview. The documentation below covers some lower-level details.

```
from fastai import *
from fastai.docs import *
from fastai.gen_doc.nbdoc import *
```

Core modules of fastai

The basic foundations needed in several parts of the library are provided by these modules:

`data`

This module defines the basic `DataBunch` class which is what will be needed to create a `Learner` object with a model. It also defines the `DeviceDataLoader`, a class that wraps a pytorch `DataLoader` to put batches on the right device.

`layers`

This module contains the definitions of basic custom layers we need in most of our models, as well as a few helper functions to create simple blocks.

Most of the documentation of the following two modules can be skipped at a first read, unless you specifically want to know what a certain function is doing.

`core`

This module contains the most basic functions and imports, notably: - pandas as pd - numpy as np - matplotlib.pyplot as plt

`torch_core`

This module contains the most basic functions and imports that use pytorch. We follow pytorch naming conventions, mainly: - torch.nn as nn - torch.optim as optim - torch.nn.functional as F

Get your data ready for training

This module defines the basic `DataBunch` object that is used inside `Learner` to train a model. This is the generic class, that can take any kind of fastai `Dataset` or `DataLoader`. You'll find helpful functions in the data module of every application to directly create this `DataBunch` for you.

```
from fastai.gen_doc.nbdoc import *
from fastai.data import *
```

```
show_doc(DataBunch, doc_string=False)
```

class DataBunch

> `DataBunch`(`train_dl`:`DataLoader`, `valid_dl`:`DataLoader`, `test_dl`:`Optional`[`DataLoader`]=`None`, `device`:`device`=`None`, `tfms`:`Optional`[`Collection`[`Callable`]]=`None`, `path`:`PathOrStr`='.', `collate_fn`:`Callable`='data_collate') [source]

Bind together a `train_dl`, a `valid_dl` and optionally a `test_dl`, ensures they are on `device` and apply to them `tfms` as batch are drawn. `path` is used internally to store temporary files, `collate_fn` is passed to the pytorch `Dataloader` (replacing the one there) to explain how to collate the samples picked for a batch. By default, it applies data to the object sent (see in `vision.image` why this can be important).

An example of `tfms` to pass is normalization. `train_dl`, `valid_dl` and optionally `test_dl` will be wrapped in `DeviceDataLoader`.

```
show_doc(DataBunch.create, full_name='create', doc_string=False)
```

create

> `create`(`train_ds`, `valid_ds`, `test_ds`=`None`, `path`:`PathOrStr`='.', `bs`:`int`=`64`, `ds_tfms`:`Union`[`Transform`, `Collection`[`Transform`], `NoneType`]=`None`, `num_workers`:`int`=`4`, `tfms`:`Optional`[`Collection`[`Callable`]]=`None`, `device`:`device`=`None`, `collate_fn`:`Callable`='data_collate', `size`:`int`=`None`, `kwargs`) -> `DataBunch` [source]

Create a `DataBunch` from `train_ds`, `valid_ds` and optionally `test_ds`, with batch size `bs` and by using `num_workers`. `tfms` and `device` are passed to the init method.

`ds_tfms`, `size` and `kwargs` are added in the `vision` application. `ds_tfms` is then a tuple of transforms to be applied to the datasets (the first one for `train_ds` and the second one for `valid_ds` and maybe `test_ds`), `size` and the `kwargs` are passed along to `transform_datasets`.

```
show_doc(DataBunch.holdout, doc_string=False)
```

holdout

holdout(is_test:bool=False) -> DeviceDataLoader [source]

Return the `self.test_dl` if `is_test` is True, else `self.valid_dl`.

```
show_doc(DataBunch.labels_to_csv, full_name='labels_to_csv', doc_string=False)
```

labels_to_csv

labels_to_csv(dest:str) [source]

This method is only available in the `vision` application. Save `self.labels` in a csv at `dest`.

```
show_doc(DeviceDataLoader, doc_string=False)
```

class DeviceDataLoader

DeviceDataLoader(dl:DataLoader, device:device, tfms:List[Callable]=None, collate_fn:Callable='data_collate') [source]

Put the batches of `dl` on `device` after applying an optional list of `tfms`. `collate_fn` will replace the one of `dl`. All dataloaders of a `DataBunch` are of this type.

Factory method

```
show_doc(DeviceDataLoader.create, doc_string=False)
```

create

create(dataset:Dataset, bs:int=1, shuffle:bool=False, device:device=device(type='cuda'), tfms:Collection[Callable]=None, num_workers:int=4, collate_fn:Callable='data_collate', kwargs:Any) [source]

Create a `DeviceDataLoader` on `device` from a `dataset` with batch size `bs`, `num_workers`processes and a given `collate_fn`. The dataloader will `shuffle` the data if that flag is set to True, and `tfms` are passed to the init method. All `kwargs` are passed to the pytorch `DataLoader` class initialization.

Internal methods

`show_doc(DeviceDataLoader.proc_batch)`

`proc_batch`

> `proc_batch`(b:`Tensor`) -> `Tensor`

Proces batch `b` of `TensorImage`. [source]

`show_doc(DeviceDataLoader.add_tfm)`

`add_tfm`

> `add_tfm`(`tfm`:`Callable`) [source]

Add `tfm` to `self.tfms`.

`show_doc(DeviceDataLoader.remove_tfm)`

`remove_tfm`

> `remove_tfm`(`tfm`:`Callable`) [source]

Remove `tfm` from `self.tfms`.

Generic classes

Those two last classes are just empty shell to be subclassed by one of the applications.

`show_doc(DatasetBase, title_level=3)`

`class DatasetBase`

> `DatasetBase`() :: `Dataset`

Base class for all fastai datasets. [source]

`show_doc(LabelDataset, title_level=3)`

`class LabelDataset`

> `LabelDataset`() :: `DatasetBase`

Base class for fastai datasets that do classification. [source]

layers

This module contains many layer classes that we might be interested in using in our models. These layers complement the default Pytorch layers which we can also use as predefined layers.

```
from fastai import *
from fastai.docs import *
from fastai.gen_doc.nbdoc import *
```

```
show_doc(AdaptiveConcatPool2d, doc_string=False)
```

class AdaptiveConcatPool2d

`AdaptiveConcatPool2d(sz:Optional[int]=None)` :: `Module` [source]

```
from fastai.gen_doc.nbdoc import *
from fastai.layers import *
```

Layer that concats `AdaptiveAvgPool2d` and `AdaptiveMaxPool2d`. Output will be `2*sz` or 2 if `sz` is None.

The `AdaptiveConcatPool2d` object uses adaptive average pooling and adaptive max pooling and concatenates them both. We use this because it provides the model with the information of both methods and improves performance. This technique is called `adaptive` because it allows us to decide on what output dimensions we want, instead of choosing the input's dimensions to fit a desired output size.

Let's try training with Adaptive Average Pooling first, then with Adaptive Max Pooling and finally with the concatenation of them both to see how they fare in performance.

We will first define a `simple_cnn` using Adapative Max Pooling by changing the source code a bit.

```
data = get_mnist()
```

```
def simple_cnn_max(actns:Collection[int], kernel_szs:Collection[int]=None,
                   strides:Collection[int]=None) -> nn.Sequential:
    "CNN with `conv2d_relu` layers defined by `actns`, `kernel_szs` and `strides`"
    nl = len(actns)-1
    kernel_szs = ifnone(kernel_szs, [3]*nl)
    strides    = ifnone(strides   , [2]*nl)
    layers = [conv2d_relu(actns[i], actns[i+1], kernel_szs[i], stride=strides[i])
          for i in range(len(strides))]
    layers.append(nn.Sequential(nn.AdaptiveMaxPool2d(1), Flatten()))
    return nn.Sequential(*layers)
```

```
model = simple_cnn_max((3,16,16,2))
learner = Learner(data, model, metrics=[accuracy])
learner.fit(5)
```

```
Total time: 00:11
epoch  train loss  valid loss  accuracy
0      0.096509    0.087127    0.971050  (00:02)
1      0.049341    0.057675    0.978410  (00:02)
2      0.039522    0.039750    0.986752  (00:02)
3      0.032399    0.037580    0.986752  (00:02)
4      0.023001    0.031079    0.986752  (00:02)
```

Now let's try with Adapative Average Pooling now.

```
def simple_cnn_avg(actns:Collection[int], kernel_szs:Collection[int]=None,
               strides:Collection[int]=None) -> nn.Sequential:
   "CNN with `conv2d_relu` layers defined by `actns`, `kernel_szs` and `strides`"
    nl = len(actns)-1
    kernel_szs = ifnone(kernel_szs, [3]*nl)
    strides    = ifnone(strides   , [2]*nl)
   layers = [conv2d_relu(actns[i], actns[i+1], kernel_szs[i], stride=strides[i])
        for i in range(len(strides))]
   layers.append(nn.Sequential(nn.AdaptiveAvgPool2d(1), Flatten()))
    return nn.Sequential(*layers)
```

```
model = simple_cnn_avg((3,16,16,2))
learner = Learner(data, model, metrics=[accuracy])
learner.fit(5)
```

```
Total time: 00:11
epoch  train loss  valid loss  accuracy
0      0.148990    0.120217    0.955348  (00:02)
1      0.095889    0.079068    0.972031  (00:02)
2      0.078218    0.057952    0.979392  (00:02)
3      0.051828    0.043309    0.984789  (00:02)
4      0.042995    0.038250    0.987242  (00:02)
```

Finally we will try with the concatenation of them both `AdaptiveConcatPool2d`. We will see that, in fact, it increases our accuracy and decreases our loss considerably!

```
def simple_cnn(actns:Collection[int], kernel_szs:Collection[int]=None,
               strides:Collection[int]=None) -> nn.Sequential:
   "CNN with `conv2d_relu` layers defined by `actns`, `kernel_szs` and `strides`"
    nl = len(actns)-1
    kernel_szs = ifnone(kernel_szs, [3]*nl)
    strides    = ifnone(strides   , [2]*nl)
   layers = [conv2d_relu(actns[i], actns[i+1], kernel_szs[i], stride=strides[i])
        for i in range(len(strides))]
```

```
   layers.append(nn.Sequential(AdaptiveConcatPool2d(1), Flatten()))
    return nn.Sequential(*layers)

model = simple_cnn((3,16,16,2))
learner = Learner(data, model, metrics=[accuracy])
learner.fit(5)
```

```
Total time: 00:11
epoch  train loss  valid loss  accuracy
0      0.067155    0.053196    0.981354  (00:02)
1      0.031042    0.027170    0.991659  (00:02)
2      0.025594    0.026606    0.990677  (00:02)
3      0.017740    0.016498    0.994112  (00:02)
4      0.018267    0.018616    0.993621  (00:02)
```

```
show_doc(Lambda, doc_string=False)
```

class Lambda

Lambda(func:LambdaFunc) :: Module [source]

Lambda allows us to define functions and use them as layers in our networks inside a Sequential object.

So, for example, say we want to apply a log_softmax loss and we need to change the shape of our output batches to be able to use this loss. We can add a layer that applies the necessary change in shape by calling:

```
Lambda(lambda x: x.view(x.size(0),-1))
```

Let's see an example of how the shape of our output can change when we add this layer.

```
model = nn.Sequential(
   nn.Conv2d(3,  16, kernel_size=3, stride=2, padding=1), nn.ReLU(),
   nn.Conv2d(16, 16, kernel_size=3, stride=2, padding=1), nn.ReLU(),
   nn.Conv2d(16, 10, kernel_size=3, stride=2, padding=1), nn.ReLU(),
    nn.AdaptiveAvgPool2d(1),
)

model.cuda()

for xb, yb in data.train_dl:
    out = (model(*[xb]))
    print(out.size())
    break
```

```
torch.Size([64, 10, 1, 1])
```

```
model = nn.Sequential(
   nn.Conv2d(3,  16, kernel_size=3, stride=2, padding=1), nn.ReLU(),
   nn.Conv2d(16, 16, kernel_size=3, stride=2, padding=1), nn.ReLU(),
   nn.Conv2d(16, 10, kernel_size=3, stride=2, padding=1), nn.ReLU(),
    nn.AdaptiveAvgPool2d(1),
    Lambda(lambda x: x.view(x.size(0),-1))
)

model.cuda()

for xb, yb in data.train_dl:
    out = (model(*[xb]))
    print(out.size())
    break
```

```
torch.Size([64, 10])
```

```
show_doc(Flatten)
```

Flatten

Flatten() -> Tensor

Flattens `x` to a single dimension, often used at the end of a model. [source]

The function we build above is actually implemented in our library as `Flatten`. We can see that it returns the same size when we run it.

```
model = nn.Sequential(
   nn.Conv2d(3,  16, kernel_size=3, stride=2, padding=1), nn.ReLU(),
   nn.Conv2d(16, 16, kernel_size=3, stride=2, padding=1), nn.ReLU(),
   nn.Conv2d(16, 10, kernel_size=3, stride=2, padding=1), nn.ReLU(),
    nn.AdaptiveAvgPool2d(1),
    Flatten(),
)

model.cuda()

for xb, yb in data.train_dl:
    out = (model(*[xb]))
    print(out.size())
    break
```

```
torch.Size([64, 10])
```

```
show_doc(PoolFlatten)
```

PoolFlatten

```
PoolFlatten() -> Sequential
```

Apply `nn.AdaptiveAvgPool2d` to `x` and then flatten the result. [source]

We can combine these two final layers (AdaptiveAvgPool2d and `Flatten`) by using `Pool Flatten`.

```
model = nn.Sequential(
   nn.Conv2d(3,  16, kernel_size=3, stride=2, padding=1), nn.ReLU(),
   nn.Conv2d(16, 16, kernel_size=3, stride=2, padding=1), nn.ReLU(),
   nn.Conv2d(16, 10, kernel_size=3, stride=2, padding=1), nn.ReLU(),
    PoolFlatten()
)

model.cuda()

for xb, yb in data.train_dl:
    out = (model(*[xb]))
    print(out.size())
    break
```

```
torch.Size([64, 10])
```

```
show_doc(ResizeBatch)
```

ResizeBatch

```
ResizeBatch(size:int) -> Tensor
```

Layer that resizes x to `size`, good for connecting mismatched layers. [source]

Another use we give to the Lambda function is to resize batches with `ResizeBatch` when we have a layer that expects a different input than what comes from the previous one. Let's see an example:

```
a = torch.tensor([[1., -1.], [1., -1.]])
print(a)
```

```
tensor([[ 1., -1.],
        [ 1., -1.]])
```

```
out = ResizeBatch(4)
print(out(a))
```

```
tensor([[ 1., -1.,  1., -1.]])
```

```
show_doc(StdUpsample, doc_string=False)
```

class StdUpsample

```
StdUpsample(n_in:int, n_out:int) :: Module
```
[source]

Increases the dimensionality of our data from `n_in` to `n_out` by applying a transposed convolution layer to the input and with batchnorm and a RELU activation.

```
show_doc(CrossEntropyFlat, doc_string=False)
```

class CrossEntropyFlat

> `CrossEntropyFlat`(`weight`=`None`, `size_average`=`None`, `ignore_index`=`-100`, `reduce`=`None`, `reduction`=`'elementwise_mean'`) :: `CrossEntropyLoss` [source]

Same as nn.CrossEntropyLoss, but flattens input and target. Is used to calculate cross entropy on arrays (which Pytorch will not let us do with their nn.CrossEntropyLoss function). An example of a use case is image segmentation models where the output in an image (or an array of pixels).

The parameters are the same as nn.CrossEntropyLoss: `weight` to rescale each class, `size_average` whether we want to sum the losses across elements in a batch or we want to add them up, `ignore_index` what targets do we want to ignore, `reduce` on whether we want to return a loss per batch element and `reduction` specifies which type of reduction (if any) we want to apply to our input.

```
show_doc(Debugger)
```

class Debugger

> `Debugger`() :: `Module`

A module to debug inside a model. [source]

The debugger module allows us to peek inside a network while its training and see in detail what is going on. We can see inputs, ouputs and sizes at any point in the network.

For instance, if you run the following:

```
model = nn.Sequential(
    nn.Conv2d(3,  16, kernel_size=3, stride=2, padding=1), nn.ReLU(),
    Debugger(),
    nn.Conv2d(16, 16, kernel_size=3, stride=2, padding=1), nn.ReLU(),
    nn.Conv2d(16, 10, kernel_size=3, stride=2, padding=1), nn.ReLU(),
)

model.cuda()
```

```
learner = Learner(data, model, metrics=[accuracy])
learner.fit(5)
```

... you'll see something like this:

```
/home/ubuntu/fastai/fastai/layers.py(74)forward()
     72     def forward(self,x:Tensor) -> Tensor:
     73         set_trace()
---> 74         return x
     75
     76 class StdUpsample(nn.Module):

ipdb>
```

```
show_doc(bn_drop_lin, doc_string=False)
```

bn_drop_lin

> bn_drop_lin(n_in:int, n_out:int, bn:bool=True, p:float=0.0, actn:Optional[Module]=None) [source]

The `bn_drop_lin` function returns a sequence of batch normalization, dropout and a linear layer. This custom layer is usually used at the end of a model.

`n_in` represents the number of size of the input `n_out` the size of the output, `bn` whether we want batch norm or not, `p` is how much dropout and `actn` is an optional parameter to add an activation function at the end.

```
show_doc(conv2d)
```

conv2d

> conv2d(ni:int, nf:int, ks:int=3, stride:int=1, padding:int=None, bias=False) -> Conv2d

Create `nn.Conv2d` layer: `ni` inputs, `nf` outputs, `ks` kernel size. `padding` defaults to `k//2`. [source]

```
show_doc(conv2d_relu, doc_string=False)
```

conv2d_relu

> conv2d_relu(ni:int, nf:int, ks:int=3, stride:int=1, padding:int=None, bn:bool=False) -> Sequential [source]

Create a `conv2d` layer with `nn.ReLU` activation and optional(`bn`) `nn.BatchNorm2d`: `ni` input, `nf` out filters, `ks` kernel, `stride`:stride, `padding`:padding, `bn`: batch normalization.

```
show_doc(conv2d_trans)
```

conv2d_trans

> conv2d_trans(ni:int, nf:int, ks:int=2, stride:int=2, padding:int=0) -> ConvTranspose2d

Create nn.ConvTranspose2d layer: ni inputs, nf outputs, ks kernel size, stride: stride. padding defaults to 0. [source]

```
show_doc(conv_layer, doc_string=False)
```

conv_layer

> conv_layer(ni:int, nf:int, ks:int=3, stride:int=1) -> Sequential [source]

The conv_layer function returns a sequence of nn.Conv2D, BatchNorm2d and a leaky RELU activation function.

n_in represents the number of size of the input n_out the size of the output, ks kernel size, stride the stride with which we want to apply the convolutions.

```
show_doc(get_embedding, doc_string=False)
```

get_embedding

> get_embedding(ni:int, nf:int) -> Module [source]

Create an embedding layer with input size ni and output size nf.

```
show_doc(simple_cnn)
```

simple_cnn

> simple_cnn(actns:Collection[int], kernel_szs:Collection[int]=None, strides:Collection[int]=None) -> Sequential

CNN with conv2d_relu layers defined by actns, kernel_szs and strides. [source]

```
show_doc(std_upsample_head, doc_string=False)
```

std_upsample_head

> std_upsample_head(c, nfs:Collection[int]) -> Module [source]

Create a sequence of upsample layers with a RELU at the beggining and a nn.ConvTranspose2d.

nfs is a list with the input and output sizes of each upsample layer and c is the output size of the final 2D Transpose Convolutional layer.

```
show_doc(trunc_normal_)
```

trunc_normal_

> `trunc_normal_(x:Tensor, mean:float=0.0, std:float=1.0) -> Tensor`

Truncated normal initialization. [source]

Basic core

This module contains all the basic functions we need in other modules of the fastai library (split with `torch_core` that contains the ones requiring pytorch). Its documentation can easily be skipped at a first read, unless you want to know what a given fuction does.

```
from fastai.gen_doc.nbdoc import *
from fastai.core import *
```

Global constants

```
default_cpus = min(16, num_cpus())
```

[source]

Check functions

```
show_doc(ifnone)
```

ifnone

> `ifnone(a:Any, b:Any) -> Any`

`a` if `a` is not None, otherwise `b`. [source]

```
show_doc(is_listy)
```

is_listy

> `is_listy(x:Any) -> bool` [source]

Check if `x` is a `Collection`.

```
show_doc(is_tuple)
```

`is_tuple`

> `is_tuple(x:Any) -> bool` [source]

Check if `x` is a `tuple`.

Collection related functions

```
show_doc(arrays_split)
```

`arrays_split`

> `arrays_split(mask:ndarray, arrs:NPArrayableList) -> SplitArrayList`

Given `arrs` is [a,b,...] and `mask`index - return[(a[mask],a[~mask]),(b[mask],b[~mask]),...]. [source]

```
show_doc(extract_kwargs)
```

`extract_kwargs`

> `extract_kwargs(names:StrList, kwargs:KWArgs)`

Extracs the keys in `names` from the `kwargs`. [source]

```
show_doc(get_chunk_length, doc_string=False)
```

`get_chunk_length`

> `get_chunk_length(csv_name:PathOrStr, chunksize:int) -> int` [source]

Return the number of chunks we will have when opening the `DataFrame` in `csv_name` with `chunksize`.

```
show_doc(get_total_length, doc_string=False)
```

`get_total_length`

> `get_total_length(csv_name:PathOrStr, chunksize:int) -> int` [source]

Return the total length we will have when opening the `DataFrame` in `csv_name` with `chunksize`.

```
show_doc(idx_dict)
```

idx_dict

idx_dict(a) [source]

Create a dictionary value to index from a.

```
idx_dict(['a','b','c'])
```

```
{'a': 0, 'b': 1, 'c': 2}
```

```
show_doc(listify)
```

listify

listify(p:OptListOrItem=None, q:OptListOrItem=None)

Make p same length as q [source]

```
show_doc(random_split)
```

random_split

random_split(valid_pct:float, arrs:NPArrayableList) -> SplitArrayList

Randomly split arrs with valid_pct ratio. good for creating validation set. [source]

```
show_doc(series2cat)
```

series2cat

series2cat(df:DataFrame, col_names)

Categorifies the columns col_names in df. [source]

```
show_doc(uniqueify)
```

uniqueify

uniqueify(x:Series) -> List[Any] [source]

Return the unique elements in x.

Files management and downloads

```
show_doc(download_url)
```

download_url

download_url(url:str, dest:str, overwrite:bool=False)

Download url to dest unless is exists and not overwrite. [source]

```
show_doc(find_classes)
```

find_classes

find_classes(folder:Path) -> FilePathList

List of label subdirectories in imagenet-style folder. [source]

```
show_doc(maybe_copy)
```

maybe_copy

maybe_copy(old_fnames:Collection[PathOrStr], new_fnames:Collection[PathOrStr])

Copy the old_fnames to new_fnames location if new_fnames don't exist or are less recent. [source]

Others

```
show_doc(ItemBase, title_level=3)
```

class ItemBase

ItemBase()

All transformable dataset items use this type. [source]

```
show_doc(camel2snake)
```

camel2snake

camel2snake(name:str) -> str [source]

Format name by removing capital letters from a class-style name and separates the subwords with underscores.

```
camel2snake('DeviceDataLoader')
```

```
'device_data_loader'
```

```
show_doc(even_mults)
```

even_mults

> even_mults(start:float, stop:float, n:int) -> ndarray

Build evenly stepped schedule from `start` to `stop` in `n` steps. [source]

```
show_doc(noop)
```

noop

> noop(x) [source]

Return `x`.

```
show_doc(num_cpus)
```

num_cpus

> num_cpus() -> int

Get number of cpus [source]

```
show_doc(partition)
```

partition

> partition(a:Collection[T_co], sz:int) -> List[Collection[T_co]]

Split iterables `a` in equal parts of size `sz` [source]

```
show_doc(partition_by_cores)
```

partition_by_cores

> partition_by_cores(a:Collection[T_co], n_cpus:int) -> List[Collection[T_co]]

Split data in `a` equally among `n_cpus` cores [source]

torch_core

This module contains all the basic functions we need in other modules of the fastai library (split with `core` that contains the ones not requiring pytorch). Its documentation can easily be skipped at a first read, unless you want to know what a given fuction does.

```
from fastai.gen_doc.nbdoc import *
from fastai.torch_core import *
```

Global constants

`AdamW = partial(optim.Adam, betas=(0.9,0.99))`

[source]

`bn_types = (nn.BatchNorm1d, nn.BatchNorm2d, nn.BatchNorm3d)`

[source]

```
default_device = torch.device('cuda') if torch.cuda.is_available()
else torch.device('cpu')
```

[source]

Functions that operate conversions

`show_doc(flatten_model, full_name='flatten')`

`flatten`

> `flatten`(`m`) [source]

Flattens all the layers of `m`.

`show_doc(model2half)`

`model2half`

> `model2half`(`model`:`Module`) -> `Module`

Convert `model` to half precision except the batchnorm layers. [source]

`show_doc(requires_grad, doc_string=False)`

`requires_grad`

> `requires_grad`(`m`:`Module`, `b`:`Optional`[`bool`]=`None`) -> `Optional`[`bool`] [source]

If `b` is None, returns the `requires_grad` state of the first layer of `m`. Otherwise, sets `requires_grad=b` in all children of `m`.

`show_doc(tensor)`

tensor

`tensor(x)` [source]

Ensures `x` is a torch `Tensor`.

```
show_doc(to_data)
```

to_data

`to_data(b:ItemsList)`

Recursively map lists of items in `b` to their wrapped data [source]

```
show_doc(to_device)
```

to_device

`to_device(b:Tensors, device:device)`

Ensure `b` is on `device`. [source]

```
show_doc(to_half, doc_string=False)
```

to_half

`to_half(b:Collection[Tensor]) -> Collection[Tensor]` [source]

Put the input of the batch `b` in half precision.

```
show_doc(to_np)
```

to_np

`to_np(x)` [source]

Convert `x` to a numpy array.

Functions to deal with model initialization

```
show_doc(apply_init)
```

apply_init

`apply_init(m, init_func:LayerFunc)`

Initialize all non-batchnorm layers of `m` with `init_func`. [source]

```
show_doc(apply_leaf)
```

apply_leaf

`apply_leaf`(`m`:`Module`, `f`:`LayerFunc`)

Apply `f` to children of `m`. [source]

`show_doc(cond_init)`

cond_init

`cond_init`(`m`:`Module`, `init_func`:`LayerFunc`)

Initialize the non-batchnorm layers of `m` with `init_func` [source]

`show_doc(in_channels)`

in_channels

`in_channels`(`m`:`Module`) -> `List`[`int`]

Return the shape of the first weight layer in `m`. [source]

Function that deal get informations on a Model

`show_doc(children)`

children

`children`(`m`:`Module`) -> `ModuleList`

Get children of module `m`. [source]

`show_doc(first_layer)`

first_layer

`first_layer`(`m`:`Module`) -> `Module`

Retrieve first layer in a module `m`. [source]

`show_doc(num_children)`

num_children

`num_children`(`m`:`Module`) -> `int`

Get number of children modules in module `m`. [source]

`show_doc(range_children)`

range_children

> range_children(m:Module) -> Iterator[int]

Return iterator of len of children of m. [source]

```
show_doc(trainable_params)
```

trainable_params

> trainable_params(m:Module) -> ParamList

Return list of trainable params in m. [source]

Functions to deal with BatchNorm layers

```
show_doc(bn2float)
```

bn2float

> bn2float(module:Module) -> Module

If module is batchnorm don't use half precision. [source]

```
show_doc(set_bn_eval)
```

set_bn_eval

> set_bn_eval(m:Module)

Set bn layers in eval mode for all recursive children of m. [source]

```
show_doc(split_bn_bias)
```

split_bn_bias

> split_bn_bias(layer_groups:ModuleList) -> ModuleList

Sort each layer in layer_groups into batchnorm (bn_types) and non-batchnorm groups. [source]

Other functions

```
show_doc(calc_loss)
```

calc_loss

> calc_loss(y_pred:Tensor, y_true:Tensor, loss_class:type='CrossEntropyLoss', bs=64)

Calculate loss between `y_pred` and `y_true` using `loss_class` and `bs`. [source]

```
show_doc(data_collate)
```

data_collate

> data_collate(batch:ItemsList) -> Tensor

Convert `batch` items to tensor data. [source]

```
show_doc(split_model, doc_string=False)
```

split_model

> split_model(model:Module, splits:Collection[Union[Module, ModuleList]], want_idxs:bool=False) [source]

Splits the `model` according to the layer in `splits`. If `splits` are layers, the model is split at those (not included) sequentially. If `want_idxs` is True, the corresponding indexes are returned. If `splits` are lists of layers, the model is split according to those.

```
show_doc(split_model_idx)
```

split_model_idx

> split_model_idx(model:Module, idxs:Collection[int]) -> ModuleList

Split `model` according to the indices in `idxs`. [source]

Undocumented Methods - Methods moved below this line will intentionally be hidden

New Methods - Please document or move to the undocumented section

How to contribute to jupyter notebooks

```
from fastai.gen_doc.nbdoc import *
from fastai.gen_doc import *
```

The documentation is built from notebooks in `docs_src/`. Follow the steps below to build documentation. For more information about generating and authoring notebooks, see fastai.gen_doc.gen_notebooks.

Modules

fastai.gen_doc.gen_notebooks

Generate and update notebook skeletons automatically from modules. Includes an overview of the whole authoring process.

fastai.gen_doc.convert2html

Create HTML (jekyll) docs from notebooks.

fastai.gen_doc.nbdoc

Underlying documentation functions; most important is `show_doc`.

Process for contributing to the docs

If you want to help us and contribute to the docs, you jsut have to make modifications to the source notebooks, our scripts will then automatically convert them to HTML. There is just one script to run after cloning the fastai_docs repo, to ensure that everything works properly. The rest of this page goes more in depth about all the functionalities this module offers, but if you jsut want to add a sentence or correct a typo, make a PR with the notebook changed and we'll take care of the rest.

Thing to run after git clone

Make sure you follow this recipe:

```
git clone https://github.com/fastai/fastai_docs
cd fastai_docs
tools/run-after-git-clone
```

This will take care of everything that is explained in the following two sections. We'll tell you what they do, but you need to execute just this one script.

Note: windows users, not using bash emulation, will need to invoke the command as:

```
python tools\run-after-git-clone
```

after-git-clone #1: a mandatory notebook strip out

Currently we only store `source` code cells under git (and a few extra fields for documentation notebooks). If you would like to commit or submit a PR, you need to confirm to that standard.

This is done automatically during `diff`/`commit` git operations, but you need to configure your local repository once to activate that instrumentation.

Therefore, your developing process will always start with:

```
tools/trust-origin-git-config
```

The last command tells git to invoke configuration stored in `fastai/.gitconfig`, so your `git diff` and `git commit` invocations for this particular repository will now go via `tools/fastai-nbstripout` which will do all the work for you.

You don't need to run it if you run:

```
tools/run-after-git-clone
```

If you skip this configuration your commit/PR involving notebooks will not be accepted, since it'll carry in it many JSON bits which we don't want in the git repository. Those unwanted bits create collisions and lead to unnecessarily complicated and time wasting merge activities. So please do not skip this step.

Note: we can't make this happen automatically, since git will ignore a repository-stored `.gitconfig` for security reasons, unless a user will tell git to use it (and thus trust it).

If you'd like to check whether you already trusted git with using `fastai/.gitconfig` please look inside `fastai/.git/config`, which should have this entry:

```
[include]
        path = ../.gitconfig
```

or alternatively run:

```
tools/trust-origin-git-config -t
```

after-git-clone #2: automatically updating doc notebooks to be trusted on git pull

We want the doc notebooks to be already trusted when you load them in `jupyter notebook`, so this script which should be run once upon `git clone`, will install a `git post-merge` hook into your local check out.

The installed hook will be executed by git automatically at the end of `git pull` only if it triggered an actual merge event and that the latter was successful.

To trust run:

```
tools/trust-doc-nbs-install-hook
```

You don't need to run it if you run:

```
tools/run-after-git-clone
```

To distrust run:

```
rm .git/hooks/post-merge
```

Validate any notebooks you're contributing to

If you were using a text editor to make changes, when you are done working on a notebook improvement, please, make sure to validate that notebook's format, by simply loading it in the jupyter notebook.

Alternatively, you could use a CLI JSON validation tool, e.g. jsonlint:

```
jsonlint-php example.ipynb
```

but it's second best, since you may have a valid JSON, but invalid notebook format, as the latter has extra requirements on which fields are valid and which are not.

Update the doc

To update the documentation notebooks to reflect changes in the library, you should use `update_notebooks` or the `sgen_notebooks` script. Only use this when you have added a new function that you want to document.

Updating docs from within notebook:

update_notebooks can be run from the notebook.

Default updates all notebooks in the fastai folder:

```
update_notebooks()
```

To update specific python file only + html also:

```
update_notebooks('../fastai/gen_doc/nbdoc.py', update_html=True, create_missing=True)
```

Updating notebooks from script:

update_notebooks can be run as a script for convenience.

Default updates all notebooks in the fastai folder:

```
python fastai/gen_doc/sgen_notebooks.py
```

Or from the tools folder

```
python tools/sgen_notebooks.py
```

To update specific python file only and html:

```
python fastai/gen_doc/sgen_notebooks.py fastai/vision/transform.py --update_html=True --crea
```

To link docs with no accompanying modules:

```
python tools/sgen_notebooks.py docs_src/index.ipynb --update_html=True --update_nb_links=Tru
```

Notebook generation

This module contains the scripts and API to auto-generate or update a documentation notebook skeleton from a given .py file or a full package. It is not expected you'd use this skeleton as your final docs - you should add markdown, examples, etc to it. The skeleton just has a minimal list of exported symbols.

fastai.gen_doc.sgen_notebooks is a script that transforms a given module into a notebook skeleton. The usage is

```
python -m sgen_notebooks package path_to_result [--update]
```

- **package** is the package you want to write the documentation of. Note that if the package isn't installed in your environment, you need to execute to execute the script in a place where package is a directory (or make a simlink to it). The script will search thourgh all the subdirectories to create all the relevant notebooks.
- **path_to_result** is a directory where you want those notebooks. The script will auto-execute them, so this directory should contain the file nbdoc.py from this package. If the module you are documenting isn't installed, you will also need to have a simlink to it in your path_to_result folder.
- if the flag –**update** is added, the script will update the notebooks (to reflect the addition of new functions or new arguments).

Alternatively, you can access the same functionality through the module API, documented below.

Important note: The notebooks automatically generated or updated need to be trusted before you can see the results in the output cells. To trust a notebook, click on File, then Trust notebook.

This module also contains the scripts and API to convert the documentation notebooks into HTML, which is the format used for the final documentation site.

```
from fastai import gen_doc
from fastai.gen_doc import nbdoc
from fastai.gen_doc.nbdoc import *
```

```
from fastai.gen_doc.gen_notebooks import *
```

Installation

This package requires: - nbconvert: conda install nbconvert - nb_extensions: conda install -c conda-forge jupyter_contrib_nbextensions

Once nbextensions is installed, your home page of jupyter notebook will look like this:

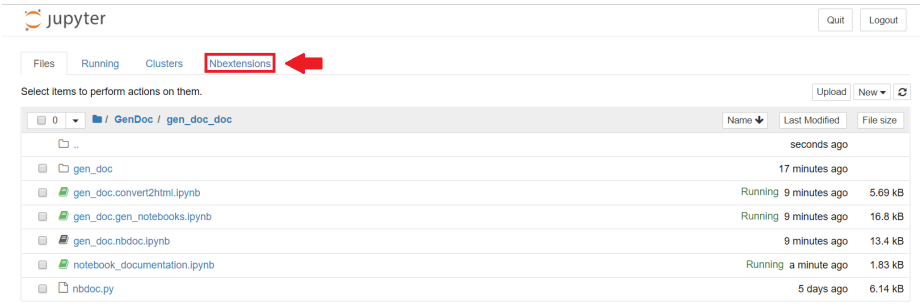

Figure 73: Homepage with nbextension

Click on the Nbextensions tab then make sure the hide inputs extension is activated:

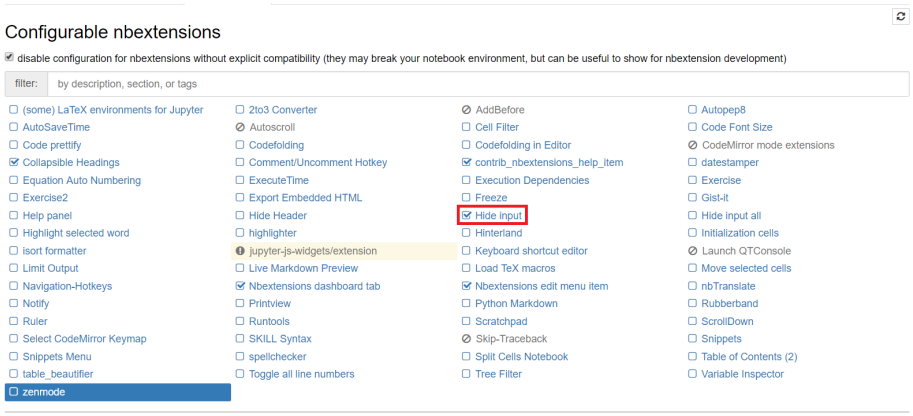

Figure 74: Activate hidden input

As its name suggests, this will allow you to hide input cells and only show their results.

Convert modules into notebook skeleton

The first (optional) step is to create a notebook "skeleton" - i.e. a notebook containing all the classes, methods, functions, and other symbols you wish to document. You can create this manually if you prefer, however using the automatic approach can save you some time and ensure you don't miss anything. For the initial skelton, use `create_module_page`, which creates a new module from scratch. To update it later with any newly-added symbols, use `update_module_page`.

```
show_doc(create_module_page, arg_comments={
    'mod': 'the module',
    'dest_path': 'the folder in which to generate the notebook',
   'force': 'if False, will raise an exception if the notebook is already present'})
```

`create_module_page`

> `create_module_page`(`mod`, `dest_path`, `force`=`False`)

Create the documentation notebook for module `mod_name` in path `dest_path`

- *mod*: the module
- *dest_path*: the folder in which to generate the notebook
- *force*: if False, will raise an exception if the notebook is already present [source]

```
show_doc(link_nb)
```

`link_nb`

> `link_nb`(`nb_path`) [source]

```
show_doc(update_module_page, arg_comments={
    'mod': 'the module',
    'dest_path': 'the folder in which to generate the notebook'})
```

`update_module_page`

> `update_module_page`(`mod`, `dest_path`='.')

Update the documentation notebook of a given module.

- *mod*: the module
- *dest_path*: the folder in which to generate the notebook [source]

All the cells added by a user are conserved, only the cells of new symbols (aka that weren't documented before) will be inserted at the end. You can then move them to wherever you like in the notebook. For instance, to update this module's documentation, simply run:

```
update_module_page(gen_doc.gen_notebooks, '.')
```

You can also generate and update *all* modules in a package using `update_notebooks`.

Updating module metadata

Jekyll pulls the documentation title, summary, and keywords from the metadata of each notebook.
Notebook metadata structure looks like this: `'metadata': { 'jekyll': {...} }`

To update metadata of these notebooks, run `generate_missing_metadata('.')`. Then open the notebook `jekyll_metadata.ipynb` to change the metadata.

```
show_doc(generate_missing_metadata)
```

`generate_missing_metadata`

> `generate_missing_metadata(dest_file)` [source]

```
show_doc(update_nb_metadata)
```

`update_nb_metadata`

> `update_nb_metadata(nb_path=None, title=None, summary=None, keywords='fastai', overwrite=True, kwargs)`

Creates jekyll metadata for given notebook path. [source]

Updating all module docs

```
show_doc(update_notebooks)
```

`update_notebooks`

> `update_notebooks(source_path, dest_path=None, update_html=True, update_nb=False, update_nb_links=True, do_execute=False, html_path=None)`

`source_path` can be a directory or a file. Assume all modules reside in the fastai directory. [source]

As a convenience method, there's `update_notebooks` to update all notebooks. This snippet does the whole lot for you:

```
update_notebooks('.', update_html=False, update_nb=True, update_nb_links=True):
```

Add documentation

The automatically generated module will only contain the table of contents and the doc string of the functions and classes in your module (or the ones you picked with ___all___). You should add more prose to them in markdown cells, or examples of uses inside the notebook.

At any time, if you don't want the input of a code cell to figure in the final result, you can use the little button in your tool bar to hide it.

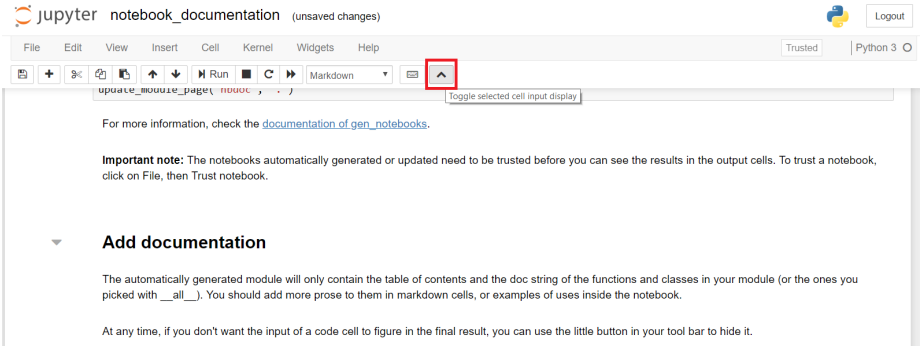

Figure 75: Button to hide an input

The same button can show you the hidden input from a cell. This used in conjunction with the helper functions from nbdoc should allow you to easily add any content you need.

Convert notebook to html

Once you're finished, don't forget to properly save your notebook, then you can either convert all the notebooks together with the script:

```
python -m convert2html dir
```

- **dir** is the directory where all your notebooks are stored.

If you prefer to do this in a notebook, you can simply type:

```
from fastai.gen_doc.convert2html import convert_nb
convert_nb('gen_doc.gen_notebooks.ipynb', '../docs')
```

For more information see the documentation of convert2html.

Undocumented Methods - Methods moved below this line will intentionally be hidden

Documentation notebook functions

```
from fastai.gen_doc.nbdoc import *
```

`nbdoc` contains the functions for documentation notebooks. The most important is `show_doc`:

Show the documentation of a function

```
show_doc(show_doc, doc_string=False)
```

`show_doc`

> `show_doc`(`elt`, `doc_string`:`bool`=`True`, `full_name`:`str`=`None`, `arg_comments`:`dict`=`None`, `title_level`=`None`, `alt_doc_string`:`str`=`''`, `ignore_warn`:`bool`=`False`, `markdown`=`True`) [source]

Show the documentation of an `elt` (function, class or enum). `doc_string` decices if we show the doc string of the element or not, `full_name` will override the name shown, `arg_comments` is a dictionary that will then show list the arguments with comments. `title_level` is the level of the corresponding cell in the TOC, `alt_doc_string` is a text that can replace the `doc_string`. `ignore_warn` will ignore warnings if you pass arguments in `arg_comments` that don't appear to belong to this function and `markdown` decides if the return is a Markdown cell or plain text.

Plenty of examples of uses of this cell can been seen through the documentation, and you will want to *hide input* those cells for a clean final result.

Convenience functions

```
show_doc(get_source_link)
```

`get_source_link`

> `get_source_link`(`mod`, `lineno`) -> `str`

Returns link to `lineno` in source code of `mod`. [source]

```
show_doc(show_video)
```

show_video

show_video(url)

Display video in url. [source]

show_doc(show_video_from_youtube)

show_video_from_youtube

show_video_from_youtube(code, start=0)

Display video from Youtube with a code and a start time. [source]

Functions for internal fastai library use

show_doc(get_exports)

get_exports

get_exports(mod) [source]

Get the exports of mod.

show_doc(get_fn_link)

get_fn_link

get_fn_link(ft) -> str

Return function link to notebook documentation of ft. [source]

show_doc(get_ft_names)

get_ft_names

get_ft_names(mod, include_inner=False) -> List[str]

Return all the functions of module mod. [source]

show_doc(is_enum)

is_enum

is_enum() [source]

Check if something is an enumerator.

show_doc(import_mod)

import_mod

> import_mod(mod_name:str)

Return module from `mod_name`. [source]

```
show_doc(link_docstring)
```

link_docstring

> link_docstring(modules, docstring:str, overwrite:bool=False)
> -> str

Search `docstring` for backticks and attempt to link those functions to respective documentation. [source]

Conversion notebook to HTML

This is the module to convert a jupyter notebook to an html page. It will normally render everything the way it was in the notebooks, including suppressing the input of the code cells with input cells masked, and converting the links between notebooks in links between the html pages.

```
from fastai.gen_doc.nbdoc import *
from fastai.gen_doc.convert2html import *
```

Functions

```
show_doc(convert_nb)
```

convert_nb

> convert_nb(fname, dest_path='.')

Converts a notebook `fname` to html file in `dest_path` [source]

```
show_doc(convert_all)
```

convert_all

> convert_all(folder, dest_path='.')

Converts all notebooks `folder` to html pages in `dest_path`. [source]

Here's an example to convert all the docs in this folder:

```
convert_all('.', '../docs')
```

Undocumented Methods - Methods moved below this line will intentionally be hidden

New Methods - Please document or move to the undocumented section

```
show_doc(read_nb)
```