Article

Fastai: A Layered API for Deep Learning

Jeremy Howard [1,2,*,†] and **Sylvain Gugger** [1,†]

1   Fast.ai, San Francisco, CA 94117, USA; s@fast.ai
2   Data Science Institute, University of San Francisco, San Francisco, CA 94117-1080, USA
*   Correspondence: j@fast.ai
†   These authors contributed equally to this work.

Abstract: fastai is a deep learning library which provides practitioners with high-level components that can quickly and easily provide state-of-the-art results in standard deep learning domains, and provides researchers with low-level components that can be mixed and matched to build new approaches. It aims to do both things without substantial compromises in ease of use, flexibility, or performance. This is possible thanks to a carefully layered architecture, which expresses common underlying patterns of many deep learning and data processing techniques in terms of decoupled abstractions. These abstractions can be expressed concisely and clearly by leveraging the dynamism of the underlying Python language and the flexibility of the PyTorch library. fastai includes: a new type dispatch system for Python along with a semantic type hierarchy for tensors; a GPU-optimized computer vision library which can be extended in pure Python; an optimizer which refactors out the common functionality of modern optimizers into two basic pieces, allowing optimization algorithms to be implemented in 4–5 lines of code; a novel 2-way callback system that can access any part of the data, model, or optimizer and change it at any point during training; a new data block API; and much more. We used this library to successfully create a complete deep learning course, which we were able to write more quickly than using previous approaches, and the code was more clear. The library is already in wide use in research, industry, and teaching.

Keywords: deep learning; data processing pipelines

1. Introduction

fastai is a modern deep learning library, https://github.com/fastai/fastai available from GitHub as open source under the Apache 2 license, which can be installed directly using the conda or pip package managers. It includes https://docs.fast.ai/ complete documentation and tutorials, and is the subject of the book *Deep Learning for Coders with fastai and PyTorch: AI Applications Without a PhD* [1].

fastai is organized around two main design goals: to be approachable and rapidly productive, while also being deeply hackable and configurable. Other libraries have tended to force a choice between conciseness and speed of development, or flexibility and expressivity, but not both. We wanted to get the clarity and development speed of Keras [2] and the customizability of PyTorch. This goal of getting the best of both worlds motivated the design of a layered architecture. A high-level API powers ready-to-use functions to train models in various applications, offering customizable models with sensible defaults. It is built on top of a hierarchy of lower-level APIs which provide composable building blocks (see Figure 1). This way, a user wanting to rewrite part of the high-level API or add particular behavior to suit their needs does not have to learn how to use the lowest level.
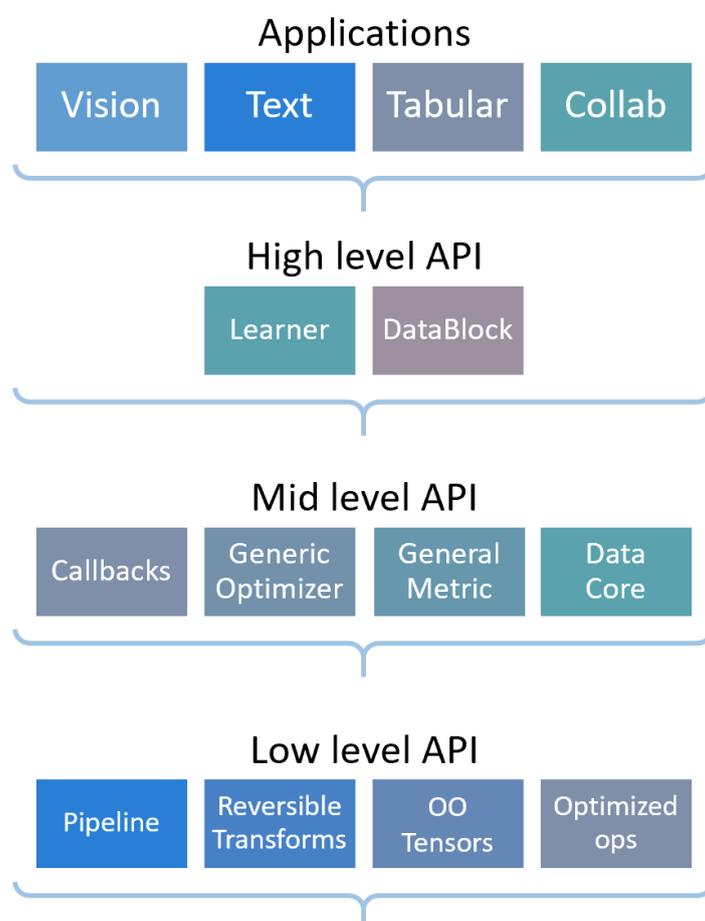
Figure 1. The layered API from fastai.

　　The high-level of the API is most likely to be useful to beginners and to practitioners who are mainly in interested in applying pre-existing deep learning methods. It offers concise APIs over four main application areas: vision, text, tabular and time-series analysis, and collaborative filtering. These APIs choose intelligent default values and behaviors based on all available information. For instance, fastai provides a single `Learner` class which brings together architecture, optimizer, and data, and automatically chooses an appropriate loss function where possible. Integrating these concerns into a single class enables fastai to curate appropriate default choices. To give another example, generally a training set should be shuffled, and a validation does not. So fastai provides a single `DataLoaders` class which automatically constructs validation and training data loaders with these details already handled. This helps practitioners ensure that they do not make mistakes such as failing to include a validation set. In addition, because the training set and validation set are integrated into a single class, fastai is able, by default, always to display metrics during training using the validation set.

　　This use of intelligent defaults–based on our own experience or best practices–extends to incorporating state-of-the-art research wherever possible. For instance, transfer learning is critically important for training models quickly, accurately, and cheaply, but the details matter a great deal. fastai automatically provides transfer learning optimised batch-normalization [3] training, layer freezing, and discriminative learning rates [4]. In general, the library's use of integrated defaults means it requires fewer lines of code from the user to re-specify information or merely to connect components. As a result, every line of user code tends to be more likely to be meaningful, and easier to read.

The mid-level API provides the core deep learning and data-processing methods for each of these applications, and low-level APIs provide a library of optimized primitives and functional and object-oriented foundations, which allows the mid-level to be developed and customised. The library itself is built on top of PyTorch [5], NumPy [6], PIL [7], pandas [8], and various other libraries. In order to achieve its goal of hackability, the library does not aim to supplant or hide these lower levels or this foundation. Within a fastai model, one can interact directly with the underlying PyTorch primitives; and within a PyTorch model, one can incrementally adopt components from the fastai library as conveniences rather than as an integrated package.

We believe fastai meets its design goals. A user can create and train a state-of-the-art vision model using transfer learning with four understandable lines of code. Perhaps more tellingly, we were able to implement recent deep learning research papers with just a couple of hours work, while matching the performance shown in the papers. We have also used the library for our winning entry in the DawnBench competition [9], training a ResNet-50 on ImageNet to accuracy in 18 min.

The following sections describe the main functionality of the various API levels in more detail and review prior related work. We chose to include a lot of code to illustrate the concepts we are presenting. While that code made change slightly as the library or its dependencies evolve (it is running against fastai v2.0.0), the ideas behind stay the same. The next section reviews the high-level APIs "out-of-the-box" applications for some of the most used deep learning domains. The applications provided are vision, text, tabular, and collaborative filtering.

2. Applications

2.1. Vision

Here is an example of how to fine-tune an ImageNet [10] model on the Oxford IIT Pets dataset [11] and achieve close to state-of-the-art accuracy within a couple of minutes of training on a single GPU:

```
from fastai.vision.all import *
path = untar_data(URLs.PETS)
dls = ImageDataLoaders.from_name_re(path=path, bs=64,
    fnames = get_image_files(path/"images"), pat = r'/([^/]+)_\d+.jpg$',
    item_tfms=RandomResizedCrop(450, min_scale=0.75),
    batch_tfms=[*aug_transforms(size=224, max_warp=0.), Normalize.from_stats(*imagenet_stats)])
learn = cnn_learner(dls, resnet34, metrics=error_rate)
learn.fit_one_cycle(4)
```

This is not an excerpt; these are all of the lines of code necessary for this task. Each line of code does one important task, allowing the user to focus on what they need to do, rather than minor details. Let's look at each line in turn:

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

This first line imports all the necessary pieces from the library. fastai is designed to be usable in a read–eval–print loop (REPL) environment as well as in a complex software system. Even if using the "import *" syntax is not generally recommended, REPL programmers generally prefer the symbols they need to be directly available to them, which is why fastai supports the "import *" style. The library is carefully designed to ensure that importing in this way only imports the symbols that are actually likely to be useful to the user and avoids cluttering the namespace or shadowing important symbols.

```
path = untar_data(URLs.PETS)
```

The second line downloads a standard dataset from the https://course.fast.ai/datasets fast.ai datasets collection (if not previously downloaded) to a configurable location (`~/.fastai/data`

by default), extracts it (if not previously extracted), and returns a `pathlib.Path` object with the extracted location.

```
dls = ImageDataLoaders.from_name_re(...)
```

This line sets up the `DataLoaders` object. This is an abstraction that represents a combination of training and validation data and will be described more in a later section. `DataLoaders` can be flexibly defined using the data block API (see Section 3.1), or, as here, can be built for specific predefined applications using specific subclasses. In this case, the `ImageDataLoaders` subclass is created using a regular expression labeller. Many other labellers are provided, particularly focused on labelling based on different kinds of file and folder name patterns, which are very common across a wide range of datasets.

One interesting feature of this API, which is also shared by lower-level fastai data APIs, is the separation of item level and batch level transforms. *Item transforms* are applied, in this case, to individual images on the CPU. *Batch transforms*, on the other hand, are applied to a minibatch, on the GPU if available. While fastai supports data augmentation on the GPU, images need to be of the same size before being batched. `aug_transforms()` selects a set of data augmentations that work well across a variety of vision datasets and problems and can be fully customized by providing parameters to the function. This is a good example of a simple "helper function"; it is not strictly necessary, because the user can list all the augmentations that they require using the individual data augmentation classes. However, by providing a single function which curates best practices and makes the most common types of customization available through a single function, users have fewer pieces to learn in order to get good results.

After defining a `DataLoaders` object the user can easily look at the data with a single line of code (result in Figure 2):
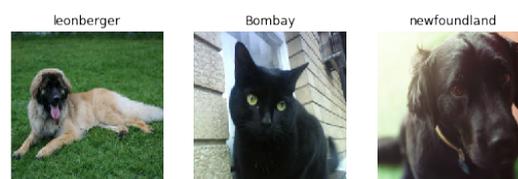
```
dls.show_batch()
```


Figure 2. A DataLoaders object built with the fastai library knows how to show its elements in a meaningful way. Here the result on the Oxford IIT Pets image classification dataset.

```
learn = cnn_learner(dls, resnet34, metrics=error_rate)
```

This fourth line creates a `Learner`, which provides an abstraction combining an optimizer, a model, and the data to train it – this will be described in more detail in Section 4.1. Each application has a customized function that creates a `Learner`, which will automatically handle whatever details it can for the user. For instance, in this case it will download an ImageNet-pretrained model, if not already available, remove the classification head of the model, replace it with a head appropriate for this particular dataset, and set appropriate defaults for the optimizer, weight decay, learning rate, and so forth (except where overridden by the user).

```
learn.fit_one_cycle(4)
```

The fifth line fits the model. In this case, it is using the 1cycle policy [12], which is a recent best practice for training and is not widely available in most deep learning libraries by default. It is annealing both the learning rates, and the momentums, printing metrics on the validation set, displaying results in an HTML table (if run in a Jupyter Notebook, or a console table otherwise), recording losses and metrics after every batch to allow plotting later, and so forth. A GPU will be used if one is available.

After training a model the user can view the results in various ways, including analysing the errors with `show_results()` (result in Figure 3):

```
learn.show_results()
```


Figure 3. A Learner knows from the data and the model type how to represent the results. It can even highlight model errors (here predicted class at bottom and actual at top).

Here is another example of a vision application, this time for segmentation on the CamVid dataset [13]:

```
from fastai.vision.all import *
path = untar_data(URLs.CAMVID)
dls = SegmentationDataLoaders.from_label_func(path=path, bs=8,
    fnames = get_image_files(path/"images"),
    label_func = lambda o: path/'labels'/f'{o.stem}_P{o.suffix}',
    codes = np.loadtxt(path/'codes.txt', dtype=str),
    batch_tfms=[*aug_transforms(size=(360,480)), Normalize.from_stats(*imagenet_stats)])
learn = unet_learner(dls, resnet34, metrics=acc_segment)
learn.fit_one_cycle(8, pct_start=0.9)
```

The lines of code to create and train this model are almost identical to those for a classification model, except for those necessary to tell fastai about the differences in the processing of the input data. The exact same line of code that was used for the image classification example can also be used to display the segmentation data as in Figure 4.

Figure 4. In this case, fastai knows that the data is for a segmentation task, and therefore it color-codes and overlays, with transparency, the segmentation layer on top of the input images.

Furthermore, the user can also view the results of the model, which again are visualized automatically in a way suitable for this task, see Figure 5
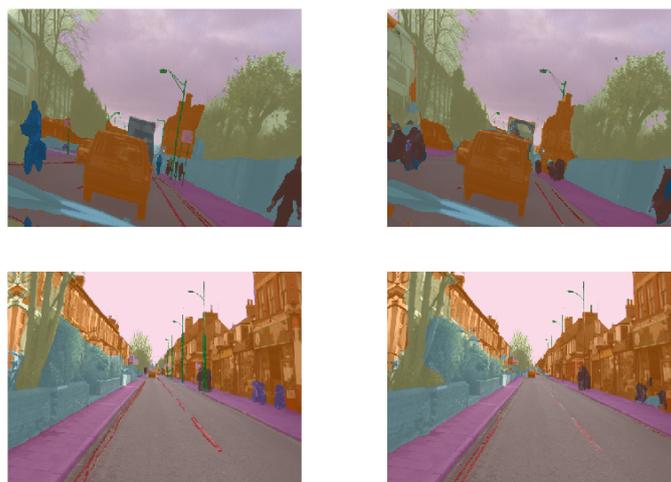
Figure 5. For a segmentation task, the ground-truth mask is laid at the right of the predicted mask.

2.2. Text

In modern natural language processing (NLP), perhaps the most important approach to building models is through fine-tuning pre-trained language models. To train a language model in fastai requires very similar code to the previous examples (here on the IMDb dataset [14]):

```
from fastai.text.all import *
path = untar_data(URLs.IMDB_SAMPLE)
df_tok,count = tokenize_df(pd.read_csv(path/'texts.csv'), ['text'])
dls_lm = TextDataLoaders.from_df(df_tok, path=path,
    vocab=make_vocab(count), text_col='text', is_lm=True)
learn = language_model_learner(dls_lm, AWD_LSTM, metrics=Perplexity()])
learn.fit_one_cycle(1, 2e-2, moms=(0.8,0.7,0.8))
```

Fine-tuning this model for classification requires the same basic steps:

```
dls_clas = TextDataLoaders.from_df(df_tok, path=path
    vocab=make_vocab(count), text_col='text', label_col='label')
learn = text_classifier_learner(dls_clas, AWD_LSTM, metrics=accuracy)
learn.fit_one_cycle(1, 2e-2, moms=(0.8,0.7,0.8))
```

The same API is also used to view the `DataLoaders`, with the result in Figure 6

	text	category
0	xxbos xxmaj raising xxmaj victor xxmaj vargas : a xxmaj review \n\n xxmaj you know , xxmaj raising xxmaj victor xxmaj vargas is like sticking your hands into a big , xxunk bowl of xxunk . xxmaj it 's warm and gooey , but you 're not sure if it feels right . xxmaj try as i might , no	negative
1	xxbos xxup the xxup shop xxup around xxup the xxup corner is one of the xxunk and most feel -good romantic comedies ever made . xxmaj there 's just no getting around that , and it 's hard to actually put one 's feeling for this film into words . xxmaj it 's not one of those films that	positive

Figure 6. In text classification, the batches are shown in a DataFrame with the tokenized texts.

The biggest challenge with creating text applications is often the processing of the input data. fastai provides a flexible processing pipeline with predefined rules for best practices, such as handling capitalization by adding tokens. For instance, there is a compromise between lower-casing all text and losing information, versus keeping the original capitalisation and ending up with too many tokens in your vocabulary. fastai handles this by adding a special single token representing that the next symbol should be treated as uppercase or sentence case and then converts the text itself to lowercase. fastai uses several these special tokens. Another example is that a sequence of more than three repeated

characters is replaced with a special repetition token, along with several repetitions and then the repeated character. These rules largely replicate the approaches discussed in [4] and are not normally made available as defaults in most NLP modelling libraries.

The tokenization is flexible and can support many different organizers. The default used is Spacy. A SentencePiece tokenizer [15] is also provided by the library. Subword tokenization [16,17], such as that provided by SentencePiece, was used in many recent NLP breakthroughs [18,19].

Numericalization and vocabulary creation often requires many lines of code, and careful management here fails and caching. In fastai that is handled transparently and automatically. Input data can be provided in many different forms, including: a separate file on disk for each document, delimited files in various formats, and so forth. The API also allows for complete customisation. SentencePiece is particularly useful for handling multiple languages and was used in MultiFIT [20], along with fastai, for this purpose. This provided models and state-of-the-art results across many different languages using a single code base.

fastai's text models are based on AWD-LSTM [21]. The user community have provided external connectors to the popular HuggingFace Transformers library [22]. The training of the models proceeds in the same way as for the vision examples with defaults appropriate for these models automatically selected. We are not aware of other libraries that provide direct support for transfer learning best practices in NLP, such as those shown in [4]. Because the tokenisation is built on top of a layered architecture, users can replace the base tokeniser with their own choices and will automatically get support for the underlying parallel process model provided by fastai. It will also automatically handle serialization of intermediate outputs so that they can be reused in future processing pipelines.

The results of training the model can be visualised with the same API as used for image models, shown in a way appropriate for NLP (see Figure 7).

	text	category	category_
0	xxbos xxup the xxup shop xxup around xxup the xxup corner is one of the xxunk and most feel - good romantic comedies ever made . xxmaj there 's just no getting around that , and it 's hard to actually put one 's feeling for this film into words . xxmaj it 's not one of those films that	positive	positive
1	xxbos xxmaj now that xxmaj che(2008) has finished its relatively short xxmaj australian cinema run (extremely limited xxunk screen in xxmaj xxunk , after xxunk) , i can xxunk join both xxunk of " at xxmaj the xxmaj movies " in taking xxmaj steven xxmaj soderbergh to task . \n\n xxmaj it 's usually satisfying to watch	negative	negative

Figure 7. In text classification, results are displayed in a DataFrame with the tokenized texts.

2.3. Tabular

Tabular models have not been very widely used in deep learning; Gradient boosting machines and similar methods are more commonly used in industry and research settings. However, there have been examples of competition winning approaches and academic state-of-the-art results using deep learning [23]. Deep learning models are particularly useful for datasets with high cardinality categorical variables because they provide embeddings that can be used even for non-deep learning models [24]. One of the challenges is there has not been examples of libraries which directly support best practices for tabular modelling using deep learning.

The pandas library [8] already provides excellent support for processing tabular data sets, and fastai does not attempt to replace it. Instead, it adds additional functionality to pandas DataFrames through various pre-processing functions, such as automatically adding features that are useful for modelling with date data. fastai also provides features for automatically creating appropriate `DataLoaders` with separated validation and training sets, using a variety of mechanisms, such as randomly splitting rows, or selecting rows based on some column.

The code to create and train a model suitable for this data should look familiar, there is just information specific to tabular data requires when building the `DataLoaders` object.

```
from fastai2.tabular.all import *
path = untar_data(URLs.ADULT_SAMPLE)
df = pd.read_csv(path/'adult.csv')
dls = TabularDataLoaders.from_df(df, path,
    procs=[Categorify, FillMissing, Normalize],
    cat_names=['workclass', 'education', 'marital-status', 'occupation', 'relationship', 'race'],
    cont_names=['age', 'fnlwgt', 'education-num'],
    y_names='salary', valid_idx=list(range(1024,1260)), bs=64)
learn = tabular_learner(dls, layers=[200,100], metrics=accuracy)
learn.fit_one_cycle(3)
```

As for every other application, `dls.show_batch` and `learn.show_results` will display a DataFrame with some samples.

fastai also integrates with NVIDIA's cuDF library, providing end-to-end GPU optimized data processing and model training. fastai is the first deep learning framework to integrate with cuDF in this way.

2.4. Collaborative Filtering

Collaborative filtering is normally modelled using a probabilistic matrix factorisation approach [25]. In practice however, a dataset generally has much more than just (for example) a user ID and a product ID, but instead has many characteristics of the user, product, time period, and so forth. It is quite standard to use those to train a model, therefore, fastai attempts to close the gap between collaborative filtering and tabular modelling. A collaborative filtering model in fastai can be simply seen as a tabular model with high cardinality categorical variables. A classic matrix factorisation model is also provided. Both are trained using the same steps that we've seen in the other applications, as in this example using the popular Movielens dataset [26]:

```
from fastai2.collab import *
ratings = pd.read_csv(untar_data(URLs.ML_SAMPLE)/'ratings.csv')
dls = CollabDataLoaders.from_df(ratings, bs=64, seed=42)
learn = collab_learner(dls, n_factors=50, y_range=[0, 5.5])
learn.fit_one_cycle(3)
```

2.5. Deployment

fastai is mostly focused on model training, but once this is done you can easily export the PyTorch model to serve it in production. The command `Learner.export` will serialize the model as well as the input pipeline (just the transforms, not the training data) to be able to apply the same to new data.

The library provides `Learner.predict` and `Learner.get_preds` to evaluate the model on an item or a new inference DataLoader. Such a DataLoader can easily be built from a set of items with the command `test_dl`.

3. High-Level API Design Considerations

3.1. High-Level API Foundations

The *high-level API* is that which is used by people using these applications. All the fastai applications share some basic components. One such component is the visualisation API, which uses a small number of methods, the main ones being `show_batch` (for showing input data) and `show_results` (for showing model results). Different types of model and datasets are able to use this consistent API because of fastai's type dispatch system, a lower-level component which will be discussed in Section 5.3. The transfer learning capability shared across the applications relies on PyTorch's parameter groups, and fastai's mid-level API then leverages these groups, such as the generic optimizer (see Section 4.3).

In all those applications, the `Learner` obtained gets the same functionality for the model training. The recommended way of training models using a variant of the 1cycle policy [12] which uses a

warm-up and annealing for the learning rate while doing the opposite with the momentum parameter (see Figure 8).
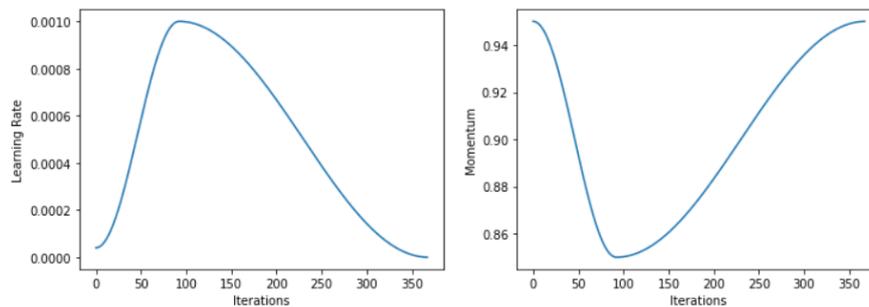

Figure 8. The hyper-parameters schedule in the 1cycle policy.

The learning rate is the most important hyper-parameter to tune (and very often the only one since the library sets proper defaults). Other libraries often provide help for grid search or AutoML to guess the best value, but the fastai library implements the learning rate finder [27] which much more quickly provides the best value for this parameter after a mock training. The command `learn.lr_find()` will return a graph like the one shown in Figure 9.
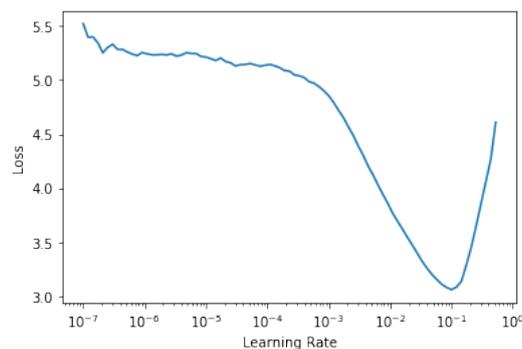

Figure 9. The learning rate finder does a mock training with an exponentially growing learning rate over 100 iterations. A good value is then the minimum value on the graph divided by 10.

Another important high-level API component, which is shared across all of the applications, is the *data block API*. The data block API is an expressive API for data loading. It is the first attempt we are aware of to systematically define all of the steps necessary to prepare data for a deep learning model, and give users a mix and match recipe book for combining these pieces (which we refer to as *data blocks*). The steps that are defined by the data block API are:

- Getting the source items,
- Splitting the items into the training set and one or more validation sets,
- Labelling the items,
- Processing the items (such as normalization), and
- Optionally collating the items into batches.

Here is an example of how to use the data block API to get the MNIST dataset [28] ready for modelling:

```
mnist = DataBlock(
    blocks=(ImageBlock(cls=PILImageBW), CategoryBlock),
    get_items=get_image_files,
    splitter=GrandparentSplitter(),
```

```
    get_y=parent_label)
dls = mnist.dataloaders(untar_data(URLs.MNIST_TINY), batch_tfms=Normalize)
```

In fastai v1 and earlier we used a fluent instead of a functional API for this (meaning the statements to execute those steps were chained one after the other). We discovered that this was a mistake; while fluent APIs are flexible in the order in which the user can define the steps, that order is very important in practice. With this functional `DataBlock` you do not have to remember if you need to split before or after labelling your data, for instance. Also, fluent APIs, at least in Python, tend not to work well with auto completion technologies. The data processing can be defined using Transforms (see Section 5.2). Here is an example of using the data blocks API to complete the same segmentation seen earlier:

```
path = untar_data(URLs.CAMVID_TINY)
camvid = DataBlock(blocks=(ImageBlock, ImageBlock(cls=PILMask)),
    get_items=get_image_files,
    splitter=RandomSplitter(),
    get_y=lambda o: path/'labels'/f'{o.stem}_P{o.suffix}')
dls = camvid.dataloaders(path/"images",
    batch_tfms=[*aug_transforms(), Normalize.from_stats(*imagenet_stats)])
```

Object detection can also be completed using the same functionality (here using the COCO dataset [29]):

```
coco_source = untar_data(URLs.COCO_TINY)
images, lbl_bbox = get_annotations(coco_source/'train.json')
lbl = dict(zip(images, lbl_bbox))

coco = DataBlock(blocks=(ImageBlock, BBoxBlock, BBoxLblBlock),
    get_items=get_image_files,
    splitter=RandomSplitter(),
    getters=[noop, lambda o:lbl[o.name][0], lambda o:lbl[o.name][1]],
    n_inp=1)
dls = coco.dataloaders(coco_source, item_tfms=Resize(128),
    batch_tfms=[*aug_transforms(), Normalize.from_stats(*imagenet_stats)])
```

In this case, the targets are a tuple of two things: a list of bounding boxes and a list of labels. This is why there are three blocks, a list of getters and an extra argument to specify how many of the blocks should be considered the input (the rest forming the target).

The data for language modeling seen earlier can also be built using the data blocks API:

```
df = pd.read_csv(untar_data(URLs.IMDB_SAMPLE)/'texts.csv')
df_tok,count = tokenize_df(df, 'text')
imdb_lm = DataBlock(blocks=TextBlock(make_vocab(count),is_lm=True),
                    get_x=attrgetter('text'),
                    splitter=RandomSplitter())
dls = imdb_lm.dataloaders(df_tok, bs=64, seq_len=72)
```

We have heard from users that they find the data blocks API provides a good balance of conciseness and expressivity. Many libraries have provided various approaches to data processing. In the data science domain the scikit-learn [30] *pipeline* approach is widely used. This API provides a very high level of expressivity, but is not opinionated enough to ensure that a user completes all of the steps necessary to get their data ready for modelling. As another example, TensorFlow [31] provides the tf.data library, which does not as precisely map the steps necessary for users to complete their task to the functionality provided by the API. The Torchvision [32] library is a good example of an API which is highly specialised to a small subset of data processing tasks for a specific subdomain. fastai tries to capture the benefits of both extremes of the spectrum, without compromises; the data blocks API is how most users transform their data for use with the library.

3.2. Incrementally Adapting PyTorch Code

Users often need to use existing pure PyTorch code (i.e., code that does not use fastai), such as their existing code-bases developed without fastai, or using third party code written in pure PyTorch. fastai supports incrementally adding fastai features to this code, without requiring extensive rewrites.

For instance, at the time of writing, the official PyTorch repository includes a MNIST training example (https://github.com/pytorch/examples/blob/master/mnist/main.py). In order to train this example using fastai's `Learner` only two steps are required. First, the 30 lines in the example covering the `test()` and `train()` functions can be removed. Then, the 4 lines of the training loop is replaced with this code:

```
data = DataLoaders(train_loader, test_loader).cuda()
learn = Learner(data, Net(), loss_func=F.nll_loss, opt_func=Adam, metrics=accuracy)
learn.fit_one_cycle(epochs, lr)
```

With no other changes, the user now has the benefit of all fastai's callbacks, progress reporting, integrated schedulers such as 1cycle training, and so forth.

3.3. Consistency Across Domains

As the application examples have shown, the fastai library allows training a variety of kinds of application models, with a variety of kinds of datasets, using a very consistent API. The consistency covers not just the initial training, but also visualising and exploring the input data and model outputs. Such consistency helps students, both through having less to learn, and through showing the unifying concepts across different types of model. It also helps practitioners and researchers focus on their model development rather than learning incidental differences between APIs across domains. It is of particular benefit when, for instance, an NLP expert tries to bring their expertise across to a computer vision application.

There are many libraries that provide high-level APIs to specific applications, such as Facebook's Torchvision [32], Detectron [33], and Fairseq [34]. However, each library has a different API, input representation, and requires different assumptions about training details, all of which a user must learn from scratch each time. This means that there are many deep learning practitioners and researchers who become specialists in specific subfields, partially based on their understanding of the toiling of those subfields. By providing a consistent API fastai users are able to quickly move between different fields and reuse their expertise.

Customizing the behaviour of predefined applications can be challenging, which means that researchers often end up "reinventing the wheel", or, constraining themselves to the specific parts which there tooling allows them to customize. Because fastai provides a layered architecture, users of the software can customize every part, as they need. The layered architecture is also an important foundation in allowing PyTorch users to incrementally add fastai functionality to existing code bases. Furthermore, fastai's layers are reused across all applications, so an investment in learning them can be leveraged across many different projects.

The approach of creating layered APIs has a long history in software engineering. Software engineering best practices involve building up decoupled components which can be tied together in flexible ways, and then creating increasingly less abstract and more customized layers on top of each part.

The layered API design is also important for researchers and practitioners aiming to create best in class results. As the field of deep learning matures, there are more and more architectures, optimizers, data processing pipelines, and other approaches that can be selected from. Trying to bring multiple approaches together into a single project can be extremely challenging, when each one is using a different, incompatible API, and has different expectations about how a model is trained. For instance, in the original mixup article [35], the code provided by the researchers only works on one specific

dataset, with one specific set of metrics, and with one specific optimizer. Attempting to combine the researchers' mixup code with other training best practices, such as mixed precision training [36], requires rewriting it largely from scratch. The next section will look at the mid-level API pieces that fastai provides, which can be mixed and matched together to allow custom approaches to be quickly and reliably created.

4. Mid-Level APIs

Many libraries, including fastai version 1 or earlier, provide a high-level API to users, and a low-level API used internally for that functionality, but nothing in between. This has two problems: the first is that it becomes harder and harder to create additional high-level functionality, as the system becomes more sophisticated, because the low-level API becomes increasingly complicated and cluttered. The second problem is that for users of the system who want to customize and adapt it, they often have to rewrite significant parts of the high-level API, and understand the large surface area of the low-level API in order to do so. This tends to mean that only a small dedicated community of specialists can really customize the software.

These issues are common across nearly all software development, and many software engineers have worked hard to find ways to deal with this complexity and develop layered architectures. The issue in the deep learning community, however, is that these practices have not seemed to be widely understood or adopted. There are, however, exceptions; most relevant to this paper, the PyTorch library [5] has a carefully layered design and is highly customizable.

Much of the innovation in fastai is in its new mid-level APIs. This section will look at the following mid-level APIs: data, callbacks, optimizer, model layers, and metrics. These APIs are what the four fastai applications are built with and are also fully documented and available to users so that they can build their own applications or customize the existing ones.

4.1. Learner

As already noted, a library can provide more appropriate defaults and user-friendly behaviour by ensuring that classes have all the information they need to make appropriate choices. One example of this is the `DataLoaders` class, which brings together all the information necessary for creating the data required for modelling. fastai also provides the `Learner` class, which brings together all the information necessary for training a model based on the data. The information which `Learner` requires, and is stored as state within a learner object, is: a PyTorch model, and optimizer, a loss function, and a `DataLoaders` object. Passing in the optimizer and loss function is optional, and in many situations fastai can automatically select appropriate defaults.

`Learner` is also responsible (along with `Optimizer`) for handling fastai's transfer learning functionality. When creating a `Learner` the user can pass a *splitter*. This is a function that describes how to split the layers of a model into PyTorch parameter groups, which can then be frozen, trained with different learning rates, or more generally handled differently by an optimizer.

One area that we have found particularly sensitive in transfer learning is the handling of batch-normalization layers [3]. We tried a wide variety of approaches to training and updating the moving average statistics of those layers, and different configurations could often change the error rate by as much as 300%. There was only one approach that consistently worked well across all datasets that we tried, which is to never freeze batch-normalization layers, and never turn off the updating of their moving average statistics. Therefore, by default, `Learner` will bypass batch-normalization layers when a user asks to freeze some parameter groups. Users often report that this one minor tweak dramatically improves their model accuracy and is not something that is found in any other libraries that we are aware of.

`DataLoaders` and `Learner` also work together to ensure that model weights and input data are all on the same device. This makes working with GPUs significantly more straightforward and makes it easy to switch from CPU to GPU as needed.

4.2. Two-Way Callbacks

In fastai version 0.7, we repeatedly modified the training loop in Learner to support many different tweaks and customizations. Over time, however, this became unwieldy. We noticed that there was a core subset of functionality that appeared in every one of these tweaks, and that all the other changes that were required could be refactored into a specific set of customization points. In other words, a wide variety of training methods can be represented using a single, universal training system. Once we extracted those common pieces, we were left with the basic fastai training loop, and the customisation points that we call *two-way callbacks*.

The Learner class's novel 2-way callback system allows gradients, data, losses, control flow, and anything else to be read and changed at any point during training. There is a rich history of using callbacks to allow for customisation of numeric software, and today nearly all modern deep learning libraries provide this functionality. However, fastai's callback system is the first that we are aware of that supports the design principles necessary for complete two-way callbacks:

- A callback should be available at every single point that code can be run during training, so that a user can customise every single detail of the training method ;
- Every callback should be able to access every piece of information available at that stage in the training loop, including hyper-parameters, losses, gradients, input and target data, and so forth ;

This is the way callbacks are usually designed, but in addition, there is a key design principle:

- Every callback should be able to modify all these pieces of information, at any time before they are used, and be able to skip a batch, epoch, training or validation section, or cancel the whole training loop.

This is why we call these 2-way callbacks, as the information not only flows from the training loop to the callbacks, but on the other way as well. For instance, here is the code for training a single batch b in fastai:

```
try:
  self._split(b);                              self.cb('begin_batch')
  self.pred = self.model(*self.x);             self.cb('after_pred')
  if len(self.y) == 0: return
  self.loss = self.loss_func(self.pred, *self.y); self.cb('after_loss')
  if not self.training: return
  self.loss.backward();                        self.cb('after_back')
  self.opt.step();                             self.cb('after_step')
  self.opt.zero_grad()
except CancelBatchException:                   self.cb('after_cancel')
finally:                                       self.cb('after_batch')
```

This example clearly shows how every step of the process is associated with a callback (the calls to self.cb() and shows how exceptions are used as a flexible control flow mechanism for them.

In fastai v0.7, we did not follow these three design principles. As a result, we had to frequently change the training loop to support additional functionality, and new research papers. On the other hand, with this new callback system we have not had to change the training loop at all, and have used callbacks to implement mixup augmentation, generative adversarial networks, optimized mixed precision training, PyTorch hooks, the learning rate finder, and many more. Most importantly, we have not yet come across any cases where mixing and matching these callbacks has caused any problems. Therefore, users can use all the training features that they want, and can easily do ablation studies, adding, removing, and modifying techniques as needed.

Case Study: Generative Adversarial Network Training Using Callbacks

A good example of a fastai callback is GANTrainer, which implements training of generative adversarial networks [37]. To do so, it must complete the following tasks:

- Freeze the generator and train the critic for one (or more) step by:

  - getting one batch of "real" images ;
  - generating one batch of "fake" images;
  - have the critic evaluate each batch and compute a loss function from that, which rewards positively the detection of real images and penalizes the fake ones;
  - update the weights of the critic with the gradients of this loss.

- Freeze the critic and train the generator for one (or more) step by:

  - generating one batch of "fake" images;
  - evaluate the critic on it;
  - return a loss that rewards positively the critic thinking those are real images;
  - update the weights of the generator with the gradients of this loss.

To do so, it relies on a `GANModule` that contains the generator and the critic, then delegates the input to the proper model depending on the value of a flag `gen_mode` and on a `GANLoss` that also has a generator or critic behavior and handles the evaluation mentioned earlier. Then, it defines the following callback methods:

- `begin_fit`: Initialises the generator, critic, loss functions, and internal storage
- `begin_epoch`: Sets the critic or generator to training mode
- `begin_validate`: Switches to generator mode for showing results
- `begin_batch`: Sets the appropriate target depending on whether it is in generator or critic mode
- `after_batch`: Records losses to the generator or critic log
- `after_epoch`: Optionally shows a sample image

This callback is then customized with another callback, which defines at what point to switch from critic to generator and vice versa. fastai includes several possibilities for this purpose, such as an `AdaptiveGANSwitcher`, which automatically switches between generator and critic training based on reaching certain thresholds in their respective losses. This approach to training can allow models to be trained significantly faster and more easily than with standard fixed schedule approaches.

4.3. Generic Optimizer

fastai provides a new generic optimizer foundation that allows recent optimization techniques to be implemented in a handful of lines of code, by refactoring out the common functionality of modern optimizers into two basic pieces:

- *stats*, which track and aggregate statistics such as gradient moving averages ;
- *steppers*, which combine stats and hyper-parameters to update the weights using some function.

This has allowed us to implement every optimizer that we have attempted in fastai, without needing to extend or change this foundation. This was very beneficial, both for research and development. As an example of a development improvement, here are the entire changes needed to make to support decoupled weight decay (also known as AdamW [38]):

```
steppers = [weight_decay] if decouple_wd else [l2_reg]
```

On the other hand, the implementation in the PyTorch library required creating an entirely new class, with over 50 lines of code. The benefit for research comes about because it it easy to rapidly implement new papers as they come out, recognise similarities and differences across techniques, and try out variants and combinations of these underlying differences, many of which have not yet

been published. The resulting code tends to look a lot like the maths shown in the paper. For instance, Figure 10 shows the code in fastai, and the algorithm from the paper, for the LAMB optimizer [39].
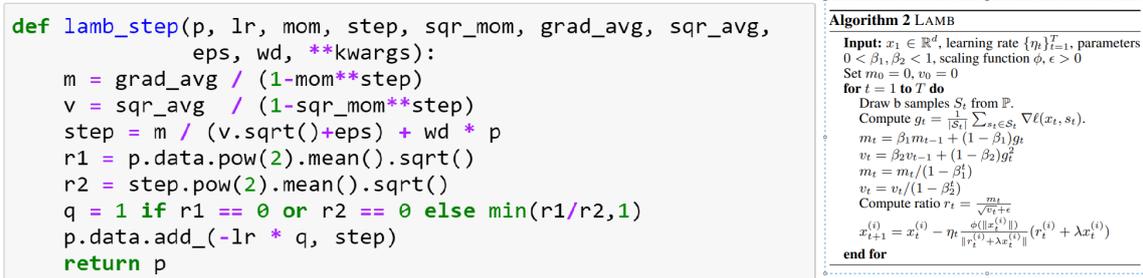
```python
def lamb_step(p, lr, mom, step, sqr_mom, grad_avg, sqr_avg,
              eps, wd, **kwargs):
    m = grad_avg / (1-mom**step)
    v = sqr_avg  / (1-sqr_mom**step)
    step = m / (v.sqrt()+eps) + wd * p
    r1 = p.data.pow(2).mean().sqrt()
    r2 = step.pow(2).mean().sqrt()
    q = 1 if r1 == 0 or r2 == 0 else min(r1/r2,1)
    p.data.add_(-lr * q, step)
    return p
```

Algorithm 2 LAMB

Input: $x_1 \in \mathbb{R}^d$, learning rate $\{\eta_t\}_{t=1}^T$, parameters $0 < \beta_1, \beta_2 < 1$, scaling function ϕ, $\epsilon > 0$
Set $m_0 = 0$, $v_0 = 0$
for $t = 1$ **to** T **do**
    Draw b samples S_t from $\mathbb{P}$.
    Compute $g_t = \frac{1}{|S_t|} \sum_{s_t \in S_t} \nabla \ell(x_t, s_t)$.
    $m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$
    $v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$
    $m_t = m_t/(1 - \beta_1^t)$
    $v_t = v_t/(1 - \beta_2^t)$
    Compute ratio $r_t = \frac{m_t}{\sqrt{v_t}+\epsilon}$
    $x_{t+1}^{(i)} = x_t^{(i)} - \eta_t \frac{\phi(\|x_t^{(i)}\|)}{\|r_t^{(i)} + \lambda x_t^{(i)}\|}(r_t^{(i)} + \lambda x_t^{(i)})$
end for

Figure 10. The LAMB algorithm and implementation.

The only difference between the code and the figure are:

- the means that update m_t and v_t do not appear as this done in a separate function *stat*;
- the authors do not provide the full definition of the ϕ function they use (it depends on undefined parameters), the code below is based on the official TensorFlow implementation.

To support modern optimizers such as LARS fastai allows the user to choose whether to aggregate stats at model, layer, or per activation level.

4.4. Generalized Metric API

Nearly all machine learning and deep learning libraries provide some support for *metrics*. These are generally defined as simple functions which take the mean, or in some cases a custom reduction function, across some measurement which is logged during training. However, some metrics cannot be correctly defined using this framework. For instance, the dice coefficient, which is widely used for measuring segmentation accuracy, cannot be directly expressed using a simple reduction.

To provide a more flexible foundation to support metrics like this fastai provides a `Metric` abstract class which defines three methods: `reset`, `accumulate`, and `value` (which is a property). Reset is called at the start of training, accumulate is called after each batch, and then finally value is called to calculate the final check. Whenever possible, we can thus avoid recording and storing all predictions in memory. For instance, here is the definition of the dice coefficient:

```python
class Dice(Metric):
    def __init__(self, axis=1): self.axis = axis
    def reset(self): self.inter,self.union = 0,0
    def accumulate(self, learn):
        pred,targ = flatten_check(learn.pred.argmax(self.axis), learn.y)
        self.inter += (pred*targ).float().sum().item()
        self.union += (pred+targ).float().sum().item()

    @property
    def value(self): return 2. * self.inter/self.union if self.union>0 else None
```

The Scikit-learn library [30] already provides a wide variety of useful metrics, so instead of reinventing them, fastai provides a simple wrapper function, `skm_to_fastai`, which allows them to be used in fastai, and can automatically add pre-processing steps such as sigmoid, argmax, and thresholding.

4.5. Fastai.Data.External

Many libraries have recently started integrating access to external datasets directly into their APIs. fastai builds on this trend, by curating and collecting several datasets (hosted by the AWS Public Dataset Program ((https://aws.amazon.com/opendata/public-datasets/)) in a single place and

making them available through the `fastai.data.external` module. fastai automatically downloads, extracts, and caches these datasets when they are first used. This is similar to functionality provided by Torchvision, TensorFlow datasets, and similar libraries, with the addition of closer integration into the fastai ecosystem. For instance, fastai provides cut-down "sample" versions of many of its datasets, which are small enough that they can be downloaded and used directly in documentation, continuous integration testing, and so forth. These datasets are also used in the documentation, along with examples showing users what they can expect when training models with his datasets. Because the documentation is written in interactive notebooks (as discussed in a later section) this also means that users can directly experiment with these datasets and models by simply running and modifying the documentation notebooks.

4.6. Funcs_kwargs and DataLoader

Once a user has their data available, they need to get it into a form that can be fed to a PyTorch model. The most common class used to feed models directly is the `DataLoader` class in PyTorch. This class provides fast and reliable multi-threaded data-processing execution, with several points allowing customisation. However, we have found that it is not flexible enough to conveniently do some of the tasks that we have required, such as building a `DataLoader` for an NLP language model. Therefore, fastai includes a new `DataLoader` class on top of the internal classes that PyTorch uses. This combines the benefits of the fast and reliable infrastructure provided by PyTorch with a more flexible and expressive front-end for the user.

`DataLoader` provides 15 extension points via customizable methods, which can be replaced by the user as required. These customizable methods represent the 15 stages of data loading that we have identified, and which fit into three broad stages: sample creation, item creation, and batch creation. In contrast, in the standard PyTorch `DataLoader` class only a small subset of these stages is explicitly made available for customization by the user. Unless a user's requirements are met by this subset, the user is forced to implement their own solution from scratch. The impact of this additional customizability can be quite significant. For instance, the fastai language model `DataLoader` went from 90 lines of code to 30 lines of code after adopting this approach.

What makes this flexibility possible is a Python decorator that is called `funcs_kwargs`. This decorator creates a class in which any method can be replaced by passing a new function to the constructor, or by replacing it through subclassing. This allows users to replace any part of the logic in the `DataLoader` class. In order to maximise the power of this, nearly every part of the fastai `DataLoader` is a method with a single line of code. Therefore, virtually every design choice can be adjusted by users.

fastai also provides a transformed `DataLoader` called `TfmdDL`, which subclasses `DataLoader`. In `TfmdDL` the callbacks and customization points execute *Pipelines* of *Transforms*. Both mechanisms are described in Section 5.2; this section provides a brief overview here. A `Transform` is simply a Python function, which can also include its inverse function—that is, the function which "undoes" the transform. Transforms can be composed using the `Pipeline` class, which then allows the entire function composition to be inverted as well. We refer to these two directions, the forward and inverse directions of the functions, as the `Transforms`' `encodes` and `decodes` methods.

`TfmdDL` provides the foundations for the visualisation support discussed in the application section, with the basic template for showing a batch of data. In order to do this, it needs to *decode* any transforms in the pipeline, which it does automatically. For instance, an integer representing a level of a category will be converted back into the string that the integer represents.

4.7. Fastai.Data.Core

When users who need to create a new kind of block for the data blocks API, or need a level of customization that even the data blocks API does not support, they can use the mid-level components that the data block API is built on. These are a small number of simple classes which combine the transform pipelines functionality of fastai with Python's collections interface.

The most basic class is transformed list, or `TfmdLists`, which lazily applies a transform pipeline to a collection, while providing a standard Python collection interface. This is an important foundational functionality for deep learning, such as the ability to index into a collection of filenames, and on demand read an image file then apply any processing, such as data augmentation and normalization, necessary for a model. `TfmdLists` also provides *subset* functionality, which allows the user to define subsets of the data, such as those representing training and validation sets. Information about what subset an item belongs to is passed down to transforms, so that they can ensure that they do the appropriate processing – for instance, data augmentation processing would be generally skipped for a validation set, unless doing test time augmentation.

Another important data class at this layer of the API is `Datasets`, which applies multiple transform pipelines in parallel to a single collection. Like `TfmdLists`, it provides a standard Python collection interface. Indexing into a `Datasets` object returns a tuple containing the result of each transform pipeline on the input item. This is the class used by the data blocks API to return, for instance, a tuple of an image tensor, and a label for that image, both derived from the same input filename.

4.8. Layers and Architectures

PyTorch (like many other libraries) provides a basic "sequential" layer object, which can be combined in sequence to form a component of a network. This represents simple composition of functions, where each layer's output is the next layer's input. However, many components in modern network architectures cannot be represented in this way. For instance, ResNet blocks [40], and any other block which requires a skip connection, are not compatible with sequential layers. The normal workaround for this in PyTorch is to write a custom forward function, effectively relying on the full flexibility of Python to escape the limits of composing these sequence layers.

However, there is a significant downside: the model is now no longer amenable to easy analysis and modification, such as removing the final few layers in order to do transfer learning. This also makes it harder to support automatic drawing graphs representing a model, printing a model summary, accurately reporting on model computation requirements by layer, and so forth.

Therefore, fastai attempts to provide the basic foundations to allow modern neural network architectures to be built by stacking a small number of predefined building blocks. The first piece of this system is the `SequentialEx` layer. This layer has the same basic API as PyTorch's `nn.Sequential`, with one key difference: the original input value to the function is available to every layer in the block. Therefore, the user can, for instance, include a layer which adds the current value of the sequential block to the input value of the sequential block (such as is done in a ResNet).

To take full advantage of this capability, fastai also provides a `MergeLayer` class. This allows the user to pass any function, which will in turn be provided with the layer block input value, and the current value of the sequential block. For instance, if you pass in a simple add function, then `MergeLayer` provides the functionality of an identity connection in a standard resnet block. Or, if the user passes in a concatenation function, then it provides the basic functionality of a concatenating connection in a Densenet block [41]. In this way, fastai provides primitives which allow representing modern network architecture out of predefined building blocks, without falling back to Python code in the forward function.

fastai also provides a general-purpose class for combining these layers into a wide range of modern convolutional neural network architectures. These are largely based on the underlying foundations from ResNet [40], and therefore this class is called `XResNet`. By providing parameters to this class, the user can customise it to create architectures that include squeeze and excitation blocks [42], grouped convolutions such as in ResNext [43], depth-wise convolutions such as in the Xception architecture [44], widening factors such as in Wide ResNets [45], self-attention and symmetric self-attention functionality , custom activation functions, and more. By using this generic refactoring of these clusters of modern neural network architectures, we were able to design and experiment with

novel combinations very easily. It is also clearer to users exactly what is going on in their models, because the various specific architectures are clearly represented as changes to input parameters.

One set of techniques that is extremely useful in practice are the tweaks to the ResNet architecture described in [46]. These approaches are used by default in `XResNet`. Another architecture tweak which has worked well in many situations is the recently developed Mish activation function [47]. fastai includes an implementation of Mish which is optimised using PyTorch's just-in-time compiler (JIT).

A similar approach was used to refactor the U-Net architecture [48]. Through looking at a range of competition winning and state-of-the-art papers in segmentation, we curated a set of approaches that work well together in practice. These are made available by default in fastai's U-Net implementation, which also dynamically creates the U-Net cross connections for any given input size.

5. Low-Level APIs

The layered approach of the fastai library has a specific meaning at the lower levels of it stack. Rather than treating Python [49] itself as the base layer of the computation, which the middle layer relies on, those layers rely on a set of basic abstractions provided by the lower layer. The middle layer is programmed in that set of abstractions. The low-level of the fastai stack provides a set of abstractions for:

- Pipelines of transforms: Partially reversible composed functions mapped and dispatched over elements of tuples
- Type-dispatch based on the needs of data processing pipelines
- Attaching semantics to tensor objects, and ensuring that these semantics are maintained throughout a `Pipeline`
- GPU-optimized computer vision operations
- Convenience functionality, such as a decorator to make patching existing objects easier, and a general collection class with a NumPy-like API.

The rest of this section will explain how the transform pipeline system is built on top of the foundations provided by PyTorch, type dispatch, and semantic tensors, providing the flexible infrastructure needed for the rest of fastai.

5.1. PyTorch Foundations

The main foundation for fastai is the PyTorch [5] library. PyTorch provides a GPU optimised tensor class, a library of useful model layers, classes for optimizing models, and a flexible programming model which integrates these elements. fastai uses building blocks from all parts of the PyTorch library, including directly patching its tensor class, entirely replacing its library of optimizers, providing simplified mechanisms for using its hooks, and so forth. In earlier prototypes of fastai we used TensorFlow [31] as our platform (and before that used Theano [50]), but switched to PyTorch because we found that it had a fast core, a simple and well curated API, and rapidly growing popularity in the research community. At this point most papers at the top deep learning conferences are implemented using PyTorch.

fastai builds on many other open source libraries. For CPU image processing fastai uses and extends the Python imaging library (PIL) [7], for reading and processing tabular data it uses pandas, for most of its metrics it uses Scikit-Learn [30], and for plotting it uses Matplotlib [51]. These are the most widely used libraries in the Python open source data science community and provide the features necessary for the fastai library.

5.2. Transforms and Pipelines

One key motivation is the need to often be able to undo some subset of transformations that are applied to create the data used to modelling. This strings that represent categories cannot be

used in models directly and are turned into integers using some vocabulary. In addition, pixel values for images are generally normalized. Neither of these can be directly visualized, and therefore at inference time we need to apply the inverse of these functions to get data that is understandable. Therefore, fastai introduces a `Transform` class, which provides callable objects, along with a *decode* method. The decode method is designed to invert the function provided by a transform; it needs to be implemented manually by the user ; it is similar to the `inverse_transform` you can provide in Scikit-Learn [30] pipelines and transformers. By providing both the encode and decode methods in a single place, the user ends up with a single object which they can compose into pipelines, serialize, and so forth.

Another motivation for this part of the API is the insight that PyTorch data loaders provide tuples, and PyTorch models expect tuples as inputs. Sometimes these tuples should behave in a connected and dependent way, such as in a segmentation model, where data augmentation must be applied to both the independent and dependent variables in the same basic way. Sometimes, however, different implementations must be used for different types; for instance, when doing affine transformations to a segmentation mask nearest-neighbor interpolation is needed, but for an image generally a smoother interpolation function would be used.

In addition, sometimes transforms need to be able to opt out of processing altogether, depending on context. For instance, except when doing test time augmentation, data augmentation methods should not be applied to the validation set. Therefore, fastai automatically passes the current subset index to transforms, allowing them to modify their behaviour based on subset (for instance, training versus validation). This is largely hidden from the user, because base classes are provided which automatically do this context-dependent skipping. However, advanced users requiring complete customization can use this functionality directly.

Transforms in deep learning pipelines often require state, which can be dependent on the input data. For example, normalization statistics could be based on a sample of data batches, a categorization transform could get its vocabulary directly from the dependent variable, or an NLP numericalization transform could get its vocabulary from the tokens used in the input corpus. Therefore, fastai transforms and pipelines support a *setup* method, which can be used to create this state when setting up a `Pipeline`. When pipelines are set up, all previous transforms in the pipeline are run first, so that the transform being set up receives the same structure of data that it will when being called.

This is closely connected to the implementation of `TfmdList`. Because a `TfmdList` lazily applies a pipeline to a collection, fastai can automatically call the `Pipeline setup` method as soon as it is connected to the collection in a `TfmdList`.

5.3. Type Dispatch

The fastai type dispatch system is like the `functools.singledispatch` system provided in the Python standard library while supporting multiple dispatch over two parameters. Dispatch over two parameters is necessary for any situation where the user wants to be able to customize behavior based on both the input and target of a model. For instance, fastai uses this for the `show_batch` and `show_results` methods. As shown in the application section, these methods automatically provide an appropriate visualisation of the input, target, and results of a model, which requires responding to the types of both parameters. In one example the input was an image, and the target was a segmentation mask, and the show results method automatically used a colour-coded overlay for the mask. On the other hand, for an image classification problem, the input would be shown as an image, the prediction and target would be shown as text labels, and color-coded based on whether they were correct.

It also provides a more expressive and yet concise syntax for registering additional dispatched functions or methods, taking advantage of Python's recently introduced type annotations syntax. Here is an example of creating two different methods which dispatch based on parameter types:

```
@typedispatch
def f_td_test(x:numbers.Integral, y): return x+1
@typedispatch
def f_td_test(x:int, y:float): return x+y
```

Here `f_td_test` has a generic implementation for `x` of numeric types and all `y`s, then a specialized implementation when `x` is an `int` and `y` is a `float`.

5.4. Object-Oriented Semantic Tensors

By using fastai's transform pipeline functionality, which depends heavily on types, the mid and high-level APIs can provide a lot of power, conciseness, and expressivity for users. However, this does not work well with the types provided by PyTorch, since the basic tensor type does not have any subclasses which can be used for type dispatch. Furthermore, subclassing PyTorch tensors is challenging, because the basic functionality for instantiating the subclasses is not provided and doing any kind of tensor operation will strip away the subclass information.

Therefore, fastai provides a new tensor base class, which can be easily instantiated and subclassed. fastai also patches PyTorch's tensor class to attempt to maintain subclass information through operations wherever possible. Unfortunately, it is not possible to always perfectly maintain this information throughout every possible operation, and therefore all fastai `Transform` automatically maintain subclass types appropriately.

fastai also provides the same functionality for Python imaging library classes, along with some basic type hierarchies for Python built-in collection types, NumPy arrays, and so forth.

5.5. GPU-Accelerated Augmentation

The fastai library provides most data augmentation in computer vision on the GPU at the batch level. Historically, the processing pipeline in computer vision has always been to open the images and apply data augmentation on the CPU, using a dedicated library such as PIL [7] or OpenCV [52], then batch the results before transferring them to the GPU and using them to train the model. On modern GPUs however, architectures, such as a standard ResNet-50, are often CPU-bound. Therefore fastai implements most common functions on the GPU, using PyTorch's implementation of `grid_sample` (which does the interpolation from the coordinate map and the original image).

Most data augmentations are random affine transforms (rotation, zoom, translation, etc), functions on a coordinates map (perspective warping) or easy functions applied to the pixels (contrast or brightness changes), all of which can easily be parallelized and applied to a batch of images. In fastai, we combine all affine and coordinate transforms in one step to only apply one interpolation, which results in a smoother result (see Figure 11). Most other vision libraries do not do this and lose a lot of detail of the original image when applying several transformations in a row.
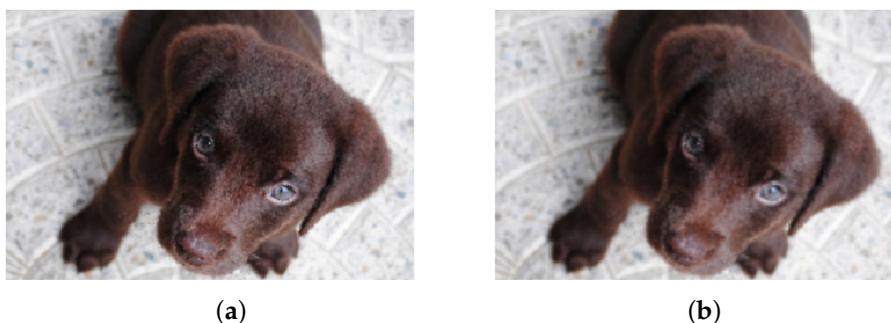

(**a**)　　　　　　　　　　　　　　　　　　　　(**b**)

Figure 11. A rotation and a zoom apply to an image with one interpolation only (**a**) or two interpolations (**b**). The latter results in more texture loss.

The type-dispatch system helps apply appropriate transforms to images, segmentation masks, key-points or bounding boxes (and users can add support for other types by writing their own functions).

5.6. Convenience Functionality

fastai has a few more additions designed to make Python easier to use, including a NumPy-like API for lists called L, and some decorators to make delegation or patching easier.

Delegation is used when one function will call another and send it a bunch of keyword arguments with defaults. To avoid repeating those, they are often grouped into `**kwargs`. The problem is that they then disappear from the signature of the function that delegates, and you can't use the tools from modern IDEs to get tab-completion for those delegated arguments or see them in its signature. To solve this, fastai provides a decorator called `@delegates` that will analyze the signature of the delegated function to change the signature of the original function. For instance the initialization of `Learner` has 11 keyword-arguments, so any function that creates a `Learner` uses this decorator to avoid mentioning them all. As an example, the function `tabular_learner` is defined like this:

```
@delegates(Learner.__init__)
def tabular_learner(dls, layers, emb_szs=None, config=None, **kwargs):
```

but when you look at its signature, you will see the 11 additional arguments of `Learner.__init__` with their defaults.

Monkey-patching is an important functionality of the Python language when you want to add functionality to existing objects. fastai makes it easier and more concise with a `@patch` decorator, using Python's type-annotation system. For instance, here is how fastai adds the `write()` method to the `pathlib.Path` class:

```
@patch
def write(self:Path, txt, encoding='utf8'):
    self.parent.mkdir(parents=True,exist_ok=True)
    with self.open('w', encoding=encoding) as f: f.write(txt)
```

Lastly, inspired by the NumPy [6] library, fastai provides a collection type, called L, that supports fancy indexing and has a lot of methods that allow users to write simple expressive code. For example, the code below takes a list of pairs, selects the second item of each pair, takes its absolute value, filters items greater than 4, and adds them up:

```
d = dict(a=1,b=-5,d=6,e=9).items()
L(d).itemgot(1).map(abs).filter(gt(4)).sum()
```

L uses context-dependent functionality to simplify user code. For instance, the `sorted` method can take any of the following as a key: a callable (sorts based on the value of calling the key with the item), a string (used as an attribute name), or an int (used as an index).

6. Nbdev

To assist in developing this library, we built a programming environment called nbdev, which allows users to create complete Python packages, including tests and a rich documentation system, all in Jupyter Notebooks [53]. nbdev is a system for *exploratory programming*. Exploratory programming is based on the observation that most developers spend most of their time as coders exploring and experimenting. Exploration is easiest developing on the prompt (or REPL), or using a notebook-oriented development system, such as Jupyter Notebooks. However, these systems are not as strong for the "programming" part, since they are missing features provided by IDEs and editors,

such as good documentation lookup, good syntax highlighting, integration with unit tests, and (most importantly) the ability to produce final, distributable source code files.

nbdev is built on top of Jupyter Notebook and adds the following critically important tools for software development:

- Python modules are automatically created, following best practices such as automatically defining `__all__` with exported functions, classes, and variables
- Navigate and edit code in a standard text editor or IDE, and export any changes automatically back into your notebooks
- Automatically create searchable, hyperlinked documentation from your code (as seen in Figure 12; any word surrounded in backticks will by hyperlinked to the appropriate documentation, a sidebar is created in the documentation site with links to each of module, and more
- Pip installers (uploaded to pypi automatically)
- Testing (defined directly in notebooks, and run in parallel)
- Continuous integration
- Version control conflict handling

We plan to provide more information about the features, benefits, and history behind nbdev in a future paper.
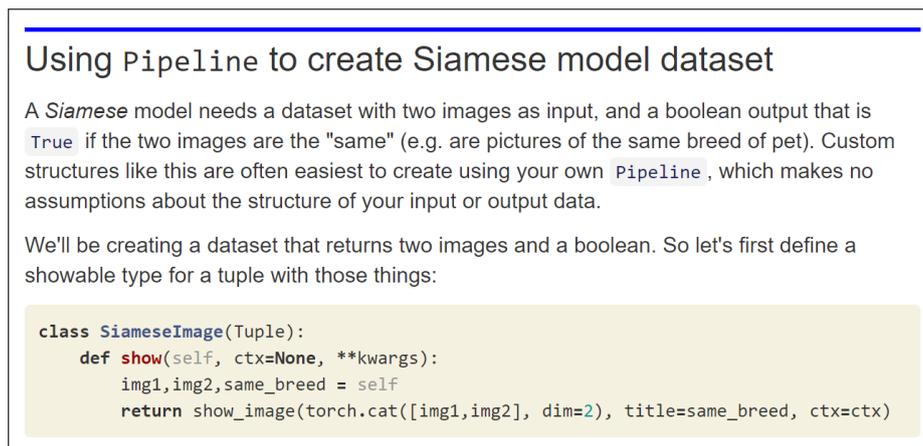
Figure 12. Example of fastai's documentation, automatically generated using nbdev.

7. Related Work

There is a long history of high-level APIs for deep learning in Python, and this history has had a significant influence on the development of fastai. The first example of a Python library for deep learning that we found is Calysto/conx, which implemented back propagation in Python in 2001. Since that time there have been dozens of approaches to high-level APIs with perhaps the most significant, in chronological order, being Lasagne [54] (begun in 2013), Fuel/Blocks (begun in 2014), and Keras [2] (begun in 2015). There have been other directions as well, such as the configuration-based approach popularized by Caffe [55], and lower-level libraries such as Theano [50], TensorFlow [31] and PyTorch [5].

APIs from general machine learning libraries have also been an important influence on fastai. SPSS and SAS provided many utilities for data processing and analysis since the early days of statistical computing. The development of the S language was a very significant advance, which led directly to projects such as R [56], SPLUS, and xlisp-stat [57]. The direction taken by R for both data processing (largely focused on the "Tidyverse" [58]) and model building (built on top of R's rich "formula" system) shows how a very different set of design choices can result in a very different (and effective) user experience. Scikit-learn [30], Torchvision [32], and pandas [8] are examples of libraries which provide

a function composition abstraction that (like fastai's `Pipeline`) are designed to help users process their data into the format they need (Scikit-Learn also being able to perform learning and predictions on that processed data). There are also projects such as MLxtend [59] that provide a variety of utilities building on the functionality of their underlying programming languages (Python, in the case of MLxtend).

The most important influence on fastai is, of course, PyTorch [5]; fastai would not have been possible without it. The PyTorch API is extensible and flexible, and the implementation is efficient. fastai makes heavy use of `torch.Tensor` and `torch.nn` (including `torch.nn.functional`). On the other hand, fastai does not make much use of PyTorch's higher level APIs, such as `nn.optim` and annealing, instead independently creating overlapping functionality based on the design approaches and goals described above.

8. Results and Conclusions

Early results from using fastai are very positive. We used the fastai library to rewrite the entire fast.ai course "Practical Deep Learning for Coders", which contains 14 h of material, across seven modules, and covers all the applications described in this paper (and some more). We found that we were able to replicate or improve on all the results in previous versions of the material and were able to create the data pipelines and models needed much more quickly and easily than we could before. We also heard from early adopters of pre-release versions of the library that they were able to more quickly and easily write deep learning code and build models than with previous versions. fastai has already been selected as part of the official PyTorch Ecosystem (https://pytorch.org/ecosystem/). According to the 2019 Kaggle ML & DS Survey (https://www.kaggle.com/c/kaggle-survey-2019), 10% of data scientists in the Kaggle community are already using fastai. Many researchers are using fastai to support their work (e.g., [60–63]).

Based on our experience with fastai, we believe that using a layered API in deep learning has very significant benefits for researchers, practitioners, and students. Researchers can see links across different areas more easily, rapidly combine and restructure ideas, and run experiments on top of strong baselines. Practitioners can quickly build prototypes, and then build on and optimize those prototypes by leveraging fastai's PyTorch foundations, without rewriting code. Students can experiment with models and try out variations, without being overwhelmed by boilerplate code when first learning ideas.

The basic ideas expressed in fastai are not limited to use in PyTorch, or even Python. There is already a partial port of fastai to Swift, called SwiftAI [64], and we hope to see similar projects for more languages and libraries in the future.

References

1.   Howard, J.; Gugger, S. *Deep Learning for Coders With Fastai and PyTorch: AI Applications Without a PhD*, 1st ed.; O'Reilly Media, Inc.: Sevastopol, CA, USA, 2020.

2. Chollet, F. Keras. 2015. Available online: https://keras.io (accessed on 14 February 2020).

3. Ioffe, S.; Szegedy, C. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. *arXiv* **2015**, arXiv:1502.03167.

4. Howard, J.; Ruder, S. Fine-tuned Language Models for Text Classification. *arXiv* **2018**, arXiv:1801.06146.

5. Paszke, A.; Gross, S.; Chintala, S.; Chanan, G.; Yang, E.; DeVito, Z.; Lin, Z.; Desmaison, A.; Antiga, L.; Lerer, A. Automatic Differentiation in PyTorch. NIPS Autodiff Workshop. 2017. Available online: https://openreview.net/forum?id=BJJsrmfCZ (accessed on 14 February 2020).

6. Oliphant, T. *NumPy: A Guide to NumPy*; Trelgol Publishing: Spanish Fork, UT, USA, 2006.

7. Clark, A. Python Imaging Library (Pillow Fork). Available online: https://github.com/python-pillow/Pillow (accessed on 14 February 2020).

8. McKinney, W. Data Structures for Statistical Computing in Python. In Proceedings of the 9th Python in Science Conference, Austin, TX, USA, 11–17 July 2010; pp. 51–56.

9. Coleman, C.; Narayanan, D.; Kang, D.; Zhao, T.; Zhang, J.; Nardi, L.; Bailis, P.; Olukotun, K.; Zaharia, C. DAWNBench: An End-to-End Deep Learning Benchmark and Competition. *Training* **2017**, *100*, 102.

10. Deng, J.; Dong, W.; Socher, R.; Li, L.J.; Li, K.; Fei-Fei, L. ImageNet: A Large-Scale Hierarchical Image Database. In Proceedings of the 2009 IEEE Conference on Computer Vision and Pattern Recognition, Miami, FL, USA, 22–24 June 2009.

11. Parkhi, O.M.; Vedaldi, A.; Zisserman, A.; Jawahar, C.V. Cats and Dogs. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, Providence, RI, USA, 16–21 June 2012.

12. Smith, L.N. A disciplined approach to neural network hyper-parameters: Part 1—learning rate, batch size, momentum, and weight decay. *arXiv* **2018**, arXiv:1803.09820.

13. Brostow, G.J.; Shotton, J.; Fauqueur, J.; Cipolla, R. Segmentation and Recognition Using Structure from Motion Point Clouds. *ECCV* **2008**, *1*, 44–57.

14. Maas, A.L.; Daly, R.E.; Pham, P.T.; Huang, D.; Ng, A.Y.; Potts, C. Learning Word Vectors for Sentiment Analysis. In *Proceedings of the 49th Annual Meeting of the Association For Computational Linguistics: Human Language Technologies*; Association for Computational Linguistics: Portland, ON, USA, 2011; pp. 142–150.

15. Kudo, T.; Richardson, J. SentencePiece: A simple and language independent subword tokenizer and detokenizer for Neural Text Processing. *arXiv* **2018**, arXiv:1808.06226.

16. Wu, Y.; Schuster, M.; Chen, Z.; Le, Q.V.; Norouzi, M.; Macherey, W.; Krikun, M.; Cao, Y.; Gao, Q.; Macherey, K.; et al. Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation. *arXiv* **2016**, arXiv:1609.08144.

17. Kudo, T. Subword Regularization: Improving Neural Network Translation Models with Multiple Subword Candidates. *arXiv* **2018**, arXiv:1804.10959.

18. Radford, A.; Wu, J.; Child, R.; Luan, D.; Amodei, D.; Sutskever, I. Language Models are Unsupervised Multitask Learners. *OpenAI Blog* **2019**, *1*, 9.

19. Devlin, J.; Chang, M.; Lee, K.; Toutanova, K. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. *arXiv* **2018**, arXiv:1810.04805.

20. Eisenschlos, J.; Ruder, S.; Czapla, P.; Kadras, M.; Gugger, S.; Howard, J. MultiFiT: Efficient Multi-lingual Language Model Fine-tuning. In Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP), Hong Kong, China, 3–7 November 2019. [CrossRef]

21. Merity, S.; Keskar, N.S.; Socher, R. Regularizing and Optimizing LSTM Language Models. *arXiv* **2017**, arXiv:1708.02182.

22. Wolf, T.; Debut, L.; Sanh, V.; Chaumond, J.; Delangue, C.; Moi, A.; Cistac, P.; Rault, T.; Louf, R.; Funtowicz, M.; et al. HuggingFace's Transformers: State-of-the-art Natural Language Processing. *arXiv* **2019**, arXiv:abs/1910.03771.

23. De Brébisson, A.; Simon, É.; Auvolat, A.; Vincent, P.; Bengio, Y. Artificial Neural Networks Applied to Taxi Destination Prediction. *arXiv* **2015**, arXiv:1508.00021.

24. Guo, C.; Berkhahn, F. Entity Embeddings of Categorical Variables. *arXiv* **2016**, arXiv:1604.06737.

25. Mnih, A.; Salakhutdinov, R.R. Probabilistic matrix factorization. *Adv. Neural Inf. Process. Syst.* **2008**, *2008* 1257–1264.

26. Harper, F.M.; Konstan, J.A. The MovieLens Datasets: History and Context. *ACM Trans. Interact. Intell. Syst.* **2015**, *5*, 19:1–19:19. [CrossRef]

27. Smith, L.N. No More Pesky Learning Rate Guessing Games. *arXiv* **2015**, arXiv:1506.01186.

28. LeCun, Y.; Cortes, C.; Burges, C. MNIST Handwritten Digit Database. 2010 Available online: http://yann.Lecun.Com/exdb/mnist (accessed on 14 February 2020).

29. Lin, T.; Maire, M.; Belongie, S.J.; Bourdev, L.D.; Girshick, R.B.; Hays, J.; Perona, P.; Ramanan, D.; Dollár, P.; Zitnick, C.L. *Microsoft COCO: Common Objects in Context*; Springer: Cham, Switzerland, 2014.

30. Pedregosa, F.; Varoquaux, G.; Gramfort, A.; Michel, V.; Thirion, B.; Grisel, O.; Blondel, M.; Prettenhofer, P.; Weiss, R.; Dubourg, V.; et al. Scikit-learn: Machine Learning in Python. *J. Mach. Learn. Res.* **2011**, *12*, 2825–2830.

31. Abadi, M.; Agarwal, A.; Barham, P.; Brevdo, E.; Chen, Z.; Citro, C.; Corrado, G.S.; Davis, A.; Dean, J.; Devin, M.; et al. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. *arXiv* **2015**, arXiv:1603.04467.

32. Massa, F.; Chintala, S. Torchvision. Available online: https://github.com/pytorch/vision/tree/master/torchvision (accessed on 14 February 2020).

33. Girshick, R.; Radosavovic, I.; Gkioxari, G.; Dollár, P.; He, K. Detectron. 2018. Available online: https://github.com/facebookresearch/detectron (accessed on 14 February 2020).

34. Ott, M.; Edunov, S.; Baevski, A.; Fan, A.; Gross, S.; Ng, N.; Grangier, D.; Auli, M. Fairseq: A Fast, Extensible Toolkit for Sequence Modeling. *arXiv* **2019**, arXiv:1904.01038.

35. Zhang, H.; Cissé, M.; Dauphin, Y.N.; Lopez-Paz, D. Mixup: Beyond Empirical Risk Minimization. *arXiv* **2017**, arXiv:1710.09412.

36. Micikevicius, P.; Narang, S.; Alben, J.; Diamos, G.; Elsen, E.; Garcia, D.; Ginsburg, B.; Houston, M.; Kuchaiev, O.; Venkatesh, G.; et al. Mixed Precision Training. 2017. Available online: http://xxx.lanl.gov/abs/1710.03740 (accessed on 14 February 2020).

37. Goodfellow, I.; Pouget-Abadie, J.; Mirza, M.; Xu, B.; Warde-Farley, D.; Ozair, S.; Courville, A.; Bengio, Y. Generative Adversarial Nets. In *Advances in Neural Information Processing Systems 27*; Ghahramani, Z., Welling, M., Cortes, C., Lawrence, N.D., Weinberger, K.Q., Eds.; Curran Associates, Inc.: Red Hook, NY, USA, 2014; pp. 2672–2680.

38. Loshchilov, I.; Hutter, F. Fixing Weight Decay Regularization in Adam. *arXiv* **2017**, arXiv:1711.05101.

39. You, Y.; Li, J.; Hseu, J.; Song, X.; Demmel, J.; Hsieh, C. Reducing BERT Pre-Training Time from 3 Days to 76 Minutes. *arXiv* **2019**, arXiv:1904.00962.

40. He, K.; Zhang, X.; Ren, S.; Sun, J. Deep Residual Learning for Image Recognition. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, Columbus, OH, USA, 17–19 June 1997.

41. Huang, G.; Liu, Z.; Weinberger, K.Q. Densely Connected Convolutional Networks. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, San Juan, PR, USA, 17–19 June 1997.

42. Hu, J.; Shen, L.; Sun, G. Squeeze-and-Excitation Networks. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, Providence, RI, USA, 18–21 June 2018.

43. Xie, S.; Girshick, R.; Dollar, P.; Tu, Z.; He, K. Aggregated Residual Transformations for Deep Neural Networks. In Proceedings of the 2017 IEEE Conf. Comput. Vision Pattern Recog. (CVPR), Honolulu, HI, USA, 21–26 July 2017. [CrossRef]

44. Chollet, F. Xception: Deep Learning with Depthwise Separable Convolutions. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, Honolulu, HI, USA, 28 July 2017.

45. Zagoruyko, S.; Komodakis, N. Wide Residual Networks. *arXiv* **2016**, arXiv:1605.07146.

46. He, T.; Zhang, Z.; Zhang, H.; Zhang, Z.; Xie, J.; Li, M. Bag of Tricks for Image Classification with Convolutional Neural Networks. *CoRR* **2018**. Available online: http://xxx.lanl.gov/abs/1812.01187 (accessed on 14 February 2020).

47. Misra, D. Mish: A Self Regularized Non-Monotonic Neural Activation Function. 2019. Available online: http://xxx.lanl.gov/abs/1908.08681 (accessed on 14 February 2020).

48. Ronneberger, O.; Fischer, P.; Brox, T. U-Net: Convolutional Networks for Biomedical Image Segmentation. In *International Conference on Medical Image Computing and Computer-Assisted Intervention*; Springer: Cham, Switzerland, 2015.

49. Python Core Team. *Python: A Dynamic, Open Source Programming Language*; Python Software Foundation, Python version 3.7; Python Core Team: Cham, Switzerland, 2019.

50. Theano Development Team. Theano: A Python framework for fast computation of mathematical expressions. *arXiv* **2016**, arXiv:1605.02688.

51. Hunter, J.D. Matplotlib: A 2D graphics environment. *Comput. Sci. Eng.* **2007**, *9*, 90–95. [CrossRef]

52. Bradski, G. The OpenCV Library. *Dr. Dobb's J. Softw. Tools* **2000**. Available online: https://www.drdobbs.com/open-source/the-opencv-library/184404319 (accessed on 14 February 2020).

53. Kluyver, T.; Ragan-Kelley, B.; Pérez, F.; Granger, B.; Bussonnier, M.; Frederic, J.; Kelley, K.; Hamrick, J.; Grout, J.; Corlay, S.; et al. Jupyter Notebooks—A publishing format for reproducible computational workflows. In *Positioning and Power in Academic Publishing: Players, Agents and Agendas*; Loizides, F., Schmidt, B., Eds.; IOS Press: Clifton, VA, USA, 2016; pp. 87–90.

54. Dieleman, S.; Schlüter, J.; Raffel, C.; Olson, E.; Sønderby, S.K.; Nouri, D.; Maturana, D.; Thoma, M.; Battenberg, E.; Kelly, J.; et al. *Lasagne: First Release*; Zenodo: Geneva, Switzerland, 2015. [CrossRef]

55. Jia, Y.; Shelhamer, E.; Donahue, J.; Karayev, S.; Long, J.; Girshick, R.; Guadarrama, S.; Darrell, T. Caffe: Convolutional Architecture for Fast Feature Embedding. *arXiv* **2014**, arXiv:1408.5093.

56. R Core Team. *R: A Language and Environment for Statistical Computing*; R Foundation for Statistical Computing: Vienna, Austria, 2017.

57. Luke Tierney. *XLISP-STAT: A Statistical Environment Based on the XLISP Language (Version 2.0)*; Technical Report 28; School of Statistics, University of Minnesota: Minneapolis, MA, USA, 1989.

58. Wickham, H.; Averick, M.; Bryan, J.; Chang, W.; McGowan, L.D.; François, R.; Grolemund, G.; Hayes, A.; Henry, L.; Hester, J.; et al. Welcome to the tidyverse. *J. Open Source Softw.* **2019**, *4*, 1686. [CrossRef]

59. Raschka, S. MLxtend: Providing machine learning and data science utilities and extensions to Python's scientific computing stack. *J. Open Source Softw.* **2018**, *3*. [CrossRef]

60. Revay, S.; Teschke, M. Multiclass Language Identification using Deep Learning on Spectral Images of Audio Signals. *arXiv* **2019**, arXiv:1905.04348,

61. Koné, I.; Boulmane, L. Hierarchical ResNeXt Models for Breast Cancer Histology Image Classification. In *International Conference Image Analysis and Recognition*; Springer: Cham, Switzerland, 2018.

62. Elkins, A.; Freitas, F.F.; Sanz, V. Developing an App to interpret Chest X-rays to support the diagnosis of respiratory pathology with Artificial Intelligence. *arXiv* **2019**, arXiv:1906.11282.

63. Anand, S.; Mahata, D.; Aggarwal, K.; Mehnaz, L.; Shahid, S.; Zhang, H.; Kumar, Y.; Shah, R.R.; Uppal, K. Suggestion Mining from Online Reviews using ULMFiT. *arXiv* **2019**, arXiv:1904.09076.

64. Jeremy, H.; Sylvain, G. *SwiftAI*; Fast.ai, Inc.: Paris, France, 2019.