

```
In [ ]: #hide
!pip install -Uqq fastbook
import fastbook
fastbook.setup_book()
```

```
In [ ]: #hide
from fastbook import *
```

[[chapter_pet_breeds]]

Image Classification

Now that you understand what deep learning is, what it's for, and how to create and deploy a model, it's time for us to go deeper! In an ideal world deep learning practitioners wouldn't have to know every detail of how things work under the hood... But as yet, we don't live in an ideal world. The truth is, to make your model really work, and work reliably, there are a lot of details you have to get right, and a lot of details that you have to check. This process requires being able to look inside your neural network as it trains, and as it makes predictions, find possible problems, and know how to fix them.

So, from here on in the book we are going to do a deep dive into the mechanics of deep learning. What is the architecture of a computer vision model, an NLP model, a tabular model, and so on? How do you create an architecture that matches the needs of your particular domain? How do you get the best possible results from the training process? How do you make things faster? What do you have to change as your datasets change?

We will start by repeating the same basic applications that we looked at in the first chapter, but we are going to do two things:

- Make them better.
- Apply them to a wider variety of types of data.

In order to do these two things, we will have to learn all of the pieces of the deep learning puzzle. This includes different types of layers, regularization methods, optimizers, how to put layers together into architectures, labeling techniques, and much more. We are not just going to dump all of these things on you, though; we will introduce them progressively as needed, to solve actual problems related to the projects we are working on.

From Dogs and Cats to Pet Breeds

In our very first model we learned how to classify dogs versus cats. Just a few years ago this was considered a very challenging task—but today, it's far too easy! We will not be able to show you the nuances of training models with this problem, because we get a nearly perfect result without worrying about any of the details. But it turns out that the same dataset also allows us to work on a much more challenging problem: figuring out what breed of pet is shown in each image.

In <> we presented the applications as already-solved problems. But this is not how things work in real life. We start with some dataset that we know nothing about. We then have to figure out how it is put together, how to extract the data we need from it, and what that data looks like. For the rest of this book we will be showing you how to solve these problems in practice, including all of the intermediate steps necessary to understand the data that you are working with and test your modeling as you go.

We already downloaded the Pet dataset, and we can get a path to this dataset using the same code as in <>:

```
In [ ]: from fastai.vision.all import *
        path = untar_data(URLs.PETS)
```

Now if we are going to understand how to extract the breed of each pet from each image we're going to need to understand how this data is laid out. Such details of data layout are a vital piece of the deep learning puzzle. Data is usually provided in one of these two ways:

- Individual files representing items of data, such as text documents or images, possibly organized into folders or with filenames representing information about those items
- A table of data, such as in CSV format, where each row is an item which may include filenames providing a connection between the data in the table and data in other formats, such as text documents and images

There are exceptions to these rules—particularly in domains such as genomics, where there can be binary database formats or even network streams—but overall the vast majority of the datasets you'll work with will use some combination of these two formats.

To see what is in our dataset we can use the `ls` method:

```
In [ ]: #hide
        Path.BASE_PATH = path
```

```
In [ ]: path.ls()
```

```
Out[ ]: (#3) [Path('annotations'), Path('images'), Path('models')]
```

We can see that this dataset provides us with *images* and *annotations* directories. The [website \(https://www.robots.ox.ac.uk/~vgg/data/pets/\)](https://www.robots.ox.ac.uk/~vgg/data/pets/) for the dataset tells us that the *annotations* directory contains information about where the pets are rather than what they are. In this chapter, we will be doing classification, not localization, which is to say that we care about what the pets are, not where they are. Therefore, we will ignore the *annotations* directory for now. So, let's have a look inside the *images* directory:

```
In [ ]: (path/"images").ls()
```

```
Out[ ]: (#7394) [Path('images/great_pyrenees_173.jpg'),Path('images/wheaten_terrier_46.jpg'),Path('images/Ragdoll_262.jpg'),Path('images/german_shorthaired_3.jpg'),Path('images/american_bulldog_196.jpg'),Path('images/boxer_188.jpg'),Path('images/staffordshire_bull_terrier_173.jpg'),Path('images/basset_hound_71.jpg'),Path('images/staffordshire_bull_terrier_37.jpg'),Path('images/yorkshire_terrier_18.jpg')...]
```

Most functions and methods in `fastai` that return a collection use a class called `L`. `L` can be thought of as an enhanced version of the ordinary Python `list` type, with added conveniences for common operations. For instance, when we display an object of this class in a notebook it appears in the format shown there. The first thing that is shown is the number of items in the collection, prefixed with a `#`. You'll also see in the preceding output that the list is suffixed with an ellipsis. This means that only the first few items are displayed—which is a good thing, because we would not want more than 7,000 filenames on our screen!

By examining these filenames, we can see how they appear to be structured. Each filename contains the pet breed, and then an underscore (`_`), a number, and finally the file extension. We need to create a piece of code that extracts the breed from a single `Path`. Jupyter notebooks make this easy, because we can gradually build up something that works, and then use it for the entire dataset. We do have to be careful to not make too many assumptions at this point. For instance, if you look carefully you may notice that some of the pet breeds contain multiple words, so we cannot simply break at the first `_` character that we find. To allow us to test our code, let's pick out one of these filenames:

```
In [ ]: fname = (path/"images").ls()[0]
```

The most powerful and flexible way to extract information from strings like this is to use a *regular expression*, also known as a *regex*. A regular expression is a special string, written in the regular expression language, which specifies a general rule for deciding if another string passes a test (i.e., "matches" the regular expression), and also possibly for plucking a particular part or parts out of that other string.

In this case, we need a regular expression that extracts the pet breed from the filename.

We do not have the space to give you a complete regular expression tutorial here, but there are many excellent ones online and we know that many of you will already be familiar with this wonderful tool. If you're not, that is totally fine—this is a great opportunity for you to rectify that! We find that regular expressions are one of the most useful tools in our programming toolkit, and many of our students tell us that this is one of the things they are most excited to learn about. So head over to Google and search for "regular expressions tutorial" now, and then come back here after you've had a good look around. The [book's website \(https://book.fast.ai/\)](https://book.fast.ai/) also provides a list of our favorites.

a: Not only are regular expressions dead handy, but they also have interesting roots. They are "regular" because they were originally examples of a "regular" language, the lowest rung within the Chomsky hierarchy, a grammar classification developed by linguist Noam Chomsky, who also wrote *Syntactic Structures*, the pioneering work searching for the formal grammar underlying human language. This is one of the charms of computing: it may be that the hammer you reach for every day in fact came from a spaceship.

When you are writing a regular expression, the best way to start is just to try it against one example at first. Let's use the `findall` method to try a regular expression against the filename of the `fname` object:

```
In [ ]: re.findall(r'(.+)\d+.jpg$', fname.name)
```

```
Out[ ]: ['great_pyrenees']
```

This regular expression plucks out all the characters leading up to the last underscore character, as long as the subsequent characters are numerical digits and then the JPEG file extension.

Now that we confirmed the regular expression works for the example, let's use it to label the whole dataset. `fastai` comes with many classes to help with labeling. For labeling with regular expressions, we can use the `RegexLabeller` class. In this example we use the data block API we saw in <> (in fact, we nearly always use the data block API—it's so much more flexible than the simple factory methods we saw in <>):

```
In [ ]: pets = DataBlock(blocks = (ImageBlock, CategoryBlock),
                        get_items=get_image_files,
                        splitter=RandomSplitter(seed=42),
                        get_y=using_attr(RegexLabeller(r'(.+)\d+.jpg$'), 'name'),
                        item_tfms=Resize(460),
                        batch_tfms=aug_transforms(size=224, min_scale=0.75))
dls = pets.dataloaders(path/"images")
```

One important piece of this `DataBlock` call that we haven't seen before is in these two lines:

```
item_tfms=Resize(460),  
batch_tfms=aug_transforms(size=224, min_scale=0.75)
```

These lines implement a fastai data augmentation strategy which we call *presizing*. Presizing is a particular way to do image augmentation that is designed to minimize data destruction while maintaining good performance.

Presizing

We need our images to have the same dimensions, so that they can collate into tensors to be passed to the GPU. We also want to minimize the number of distinct augmentation computations we perform. The performance requirement suggests that we should, where possible, compose our augmentation transforms into fewer transforms (to reduce the number of computations and the number of lossy operations) and transform the images into uniform sizes (for more efficient processing on the GPU).

The challenge is that, if performed after resizing down to the augmented size, various common data augmentation transforms might introduce spurious empty zones, degrade data, or both. For instance, rotating an image by 45 degrees fills corner regions of the new bounds with emptiness, which will not teach the model anything. Many rotation and zooming operations will require interpolating to create pixels. These interpolated pixels are derived from the original image data but are still of lower quality.

To work around these challenges, presizing adopts two strategies that are shown in <>:

1. Resize images to relatively "large" dimensions—that is, dimensions significantly larger than the target training dimensions.
2. Compose all of the common augmentation operations (including a resize to the final target size) into one, and perform the combined operation on the GPU only once at the end of processing, rather than performing the operations individually and interpolating multiple times.

The first step, the resize, creates images large enough that they have spare margin to allow further augmentation transforms on their inner regions without creating empty zones. This transformation works by resizing to a square, using a large crop size. On the training set, the crop area is chosen randomly, and the size of the crop is selected to cover the entire width or height of the image, whichever is smaller.

In the second step, the GPU is used for all data augmentation, and all of the potentially destructive operations are done together, with a single interpolation at the end.



Presizing on the training set

This picture shows the two steps:

1. *Crop full width or height*: This is in `item_tfms`, so it's applied to each individual image before it is copied to the GPU. It's used to ensure all images are the same size. On the training set, the crop area is chosen randomly. On the validation set, the center square of the image is always chosen.
2. *Random crop and augment*: This is in `batch_tfms`, so it's applied to a batch all at once on the GPU, which means it's fast. On the validation set, only the resize to the final size needed for the model is done here. On the training set, the random crop and any other augmentations are done first.

To implement this process in fastai you use `Resize` as an item transform with a large size, and `RandomResizedCrop` as a batch transform with a smaller size. `RandomResizedCrop` will be added for you if you include the `min_scale` parameter in your `aug_transforms` function, as was done in the `DataBlock` call in the previous section. Alternatively, you can use `pad` or `squish` instead of `crop` (the default) for the initial `Resize`.

<> shows the difference between an image that has been zoomed, interpolated, rotated, and then interpolated again (which is the approach used by all other deep learning libraries), shown here on the right, and an image that has been zoomed and rotated as one operation and then interpolated just once on the left (the fastai approach), shown here on the left.

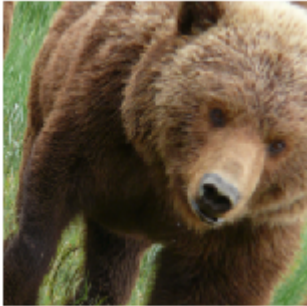
```

In [ ]: #hide_input
        #id interpolations
        #caption A comparison of fastai's data augmentation strategy (left) and
        the traditional approach (right).
        dblock1 = DataBlock(blocks=(ImageBlock(), CategoryBlock()),
                             get_y=parent_label,
                             item_tfms=Resize(460))
        dls1 = dblock1.dataloaders([(Path.cwd()/'images'/'grizzly.jpg')]*100, bs
        =8)
        dls1.train.get_idxs = lambda: Inf.ones
        x,y = dls1.valid.one_batch()
        _,axs = subplots(1, 2)

        x1 = TensorImage(x.clone())
        x1 = x1.affine_coord(sz=224)
        x1 = x1.rotate(draw=30, p=1.)
        x1 = x1.zoom(draw=1.2, p=1.)
        x1 = x1.warp(draw_x=-0.2, draw_y=0.2, p=1.)

        tfms = setup_aug_tfms([Rotate(draw=30, p=1, size=224), Zoom(draw=1.2, p=
        1., size=224),
                               Warp(draw_x=-0.2, draw_y=0.2, p=1., size=224)])
        x = Pipeline(tfms)(x)
        #x.affine_coord(coord_tfm=coord_tfm, sz=size, mode=mode, pad_mode=pad_mo
        de)
        TensorImage(x[0]).show(ctx=axs[0])
        TensorImage(x1[0]).show(ctx=axs[1]);

```



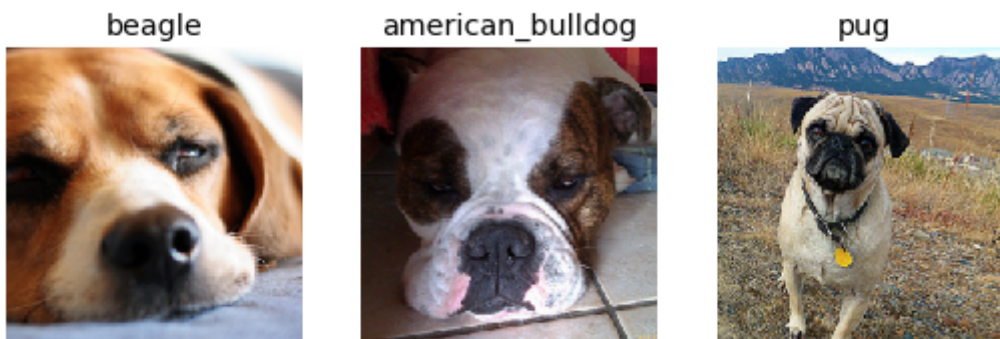
You can see that the image on the right is less well defined and has reflection padding artifacts in the bottom-left corner; also, the grass at the top left has disappeared entirely. We find that in practice using presizing significantly improves the accuracy of models, and often results in speedups too.

The fastai library also provides simple ways to check your data looks right before training a model, which is an extremely important step. We'll look at those next.

Checking and Debugging a DataBlock

We can never just assume that our code is working perfectly. Writing a `DataBlock` is just like writing a blueprint. You will get an error message if you have a syntax error somewhere in your code, but you have no guarantee that your template is going to work on your data source as you intend. So, before training a model you should always check your data. You can do this using the `show_batch` method:

```
In [ ]: dls.show_batch(nrows=1, ncols=3)
```



Take a look at each image, and check that each one seems to have the correct label for that breed of pet. Often, data scientists work with data with which they are not as familiar as domain experts may be: for instance, I actually don't know what a lot of these pet breeds are. Since I am not an expert on pet breeds, I would use Google images at this point to search for a few of these breeds, and make sure the images look similar to what I see in this output.

If you made a mistake while building your `DataBlock`, it is very likely you won't see it before this step. To debug this, we encourage you to use the `summary` method. It will attempt to create a batch from the source you give it, with a lot of details. Also, if it fails, you will see exactly at which point the error happens, and the library will try to give you some help. For instance, one common mistake is to forget to use a `Resize` transform, so you end up with pictures of different sizes and are not able to batch them. Here is what the summary would look like in that case (note that the exact text may have changed since the time of writing, but it will give you an idea):


```
In [ ]: #hide_output
pets1 = DataBlock(blocks = (ImageBlock, CategoryBlock),
                  get_items=get_image_files,
                  splitter=RandomSplitter(seed=42),
                  get_y=using_attr(RegexLabeller(r'(.+)\d+.jpg$'), 'name'))
pets1.summary(path/"images")
```

```

Setting-up type transforms pipelines
Collecting items from /home/jhoward/.fastai/data/oxford-iiit-pet/images
Found 7390 items
2 datasets of sizes 5912,1478
Setting up Pipeline: PILBase.create
Setting up Pipeline: partial -> Categorize

```

```

Building one sample
  Pipeline: PILBase.create
    starting from
      /home/jhoward/.fastai/data/oxford-iiit-pet/images/american_pit_bu
ll_terrier_31.jpg
    applying PILBase.create gives
      PILImage mode=RGB size=500x414
  Pipeline: partial -> Categorize
    starting from
      /home/jhoward/.fastai/data/oxford-iiit-pet/images/american_pit_bu
ll_terrier_31.jpg
    applying partial gives
      american_pit_bull_terrier
    applying Categorize gives
      TensorCategory(13)

```

```

Final sample: (PILImage mode=RGB size=500x414, TensorCategory(13))

```

```

Setting up after_item: Pipeline: ToTensor
Setting up before_batch: Pipeline:
Setting up after_batch: Pipeline: IntToFloatTensor

```

```

Building one batch
Applying item_tfms to the first sample:
  Pipeline: ToTensor
    starting from
      (PILImage mode=RGB size=500x414, TensorCategory(13))
    applying ToTensor gives
      (TensorImage of size 3x414x500, TensorCategory(13))

```

```

Adding the next 3 samples

```

```

No before_batch transform to apply

```

```

Collating items in a batch
Error! It's not possible to collate your items in a batch
Could not collate the 0-th members of your tuples because got the follo
wing shapes
torch.Size([3, 414, 500]),torch.Size([3, 375, 500]),torch.Size([3, 500,
281]),torch.Size([3, 203, 300])

```

```

-----
RuntimeError                                Traceback (most recent call last)
<ipython-input-11-8c0a3d421ca2> in <module>
      4         splitter=RandomSplitter(seed=42),
      5         get_y=using_attr(RegexLabeller(r'(.+)\d+.jpg
$'), 'name'))
----> 6 pets1.summary(path/"images")

~/git/fastai/fastai/data/block.py in summary(self, source, bs, show_batch, **kwargs)
    182         why = _find_fail_collate(s)
    183         print("Make sure all parts of your samples are tensors
of the same size" if why is None else why)
--> 184         raise e
    185
    186     if len([f for f in dls.train.after_batch.fs if f.name != 'noop'])!=0:

~/git/fastai/fastai/data/block.py in summary(self, source, bs, show_batch, **kwargs)
    176     print("\nCollating items in a batch")
    177     try:
--> 178         b = dls.train.create_batch(s)
    179         b = retain_types(b, s[0] if is_listy(s) else s)
    180     except Exception as e:

~/git/fastai/fastai/data/load.py in create_batch(self, b)
    125     def retain(self, res, b): return retain_types(res, b[0] if
is_listy(b) else b)
    126     def create_item(self, s): return next(self.it) if s is None
else self.dataset[s]
--> 127     def create_batch(self, b): return (fa_collate,fa_convert)[s
elf.prebatched](b)
    128     def do_batch(self, b): return self.retain(self.create_batch
(self.before_batch(b)), b)
    129     def to(self, device): self.device = device

~/git/fastai/fastai/data/load.py in fa_collate(t)
    44     b = t[0]
    45     return (default_collate(t) if isinstance(b, _collate_types)
--> 46             else type(t[0])([fa_collate(s) for s in zip(*t)]) i
f isinstance(b, Sequence)
    47             else default_collate(t))
    48

~/git/fastai/fastai/data/load.py in <listcomp>(.0)
    44     b = t[0]
    45     return (default_collate(t) if isinstance(b, _collate_types)
--> 46             else type(t[0])([fa_collate(s) for s in zip(*t)]) i
f isinstance(b, Sequence)
    47             else default_collate(t))
    48

~/git/fastai/fastai/data/load.py in fa_collate(t)
    43 def fa_collate(t):

```

```

44     b = t[0]
--> 45     return (default_collate(t) if isinstance(b, _collate_types)
46             else type(t[0])([fa_collate(s) for s in zip(*t)])) i
f isinstance(b, Sequence)
47             else default_collate(t))

~/anaconda3/lib/python3.7/site-packages/torch/utils/data/_utils/collat
e.py in default_collate(batch)
53         storage = elem.storage()._new_shared(numel)
54         out = elem.new(storage)
--> 55         return torch.stack(batch, 0, out=out)
56     elif elem_type.__module__ == 'numpy' and elem_type.__name__
!= 'str_' \
57         and elem_type.__name__ != 'string':

```

RuntimeError: invalid argument 0: Sizes of tensors must match except in dimension 0. Got 414 and 375 in dimension 2 at /opt/conda/conda-bld/pytorch_1579022060824/work/aten/src/TH/generic/THTensor.cpp:612

Setting-up type transforms pipelines

Collecting items from /home/sgugger/.fastai/data/oxford-iiit-pet/images

Found 7390 items

2 datasets of sizes 5912,1478

Setting up Pipeline: PILBase.create

Setting up Pipeline: partial -> Categorize

Building one sample

Pipeline: PILBase.create

starting from

/home/sgugger/.fastai/data/oxford-iiit-pet/images/american_bulldog_83.

jpg

applying PILBase.create gives

PILImage mode=RGB size=375x500

Pipeline: partial -> Categorize

starting from

/home/sgugger/.fastai/data/oxford-iiit-pet/images/american_bulldog_83.

jpg

applying partial gives

american_bulldog

applying Categorize gives

TensorCategory(12)

Final sample: (PILImage mode=RGB size=375x500, TensorCategory(12))

Setting up after_item: Pipeline: ToTensor

Setting up before_batch: Pipeline:

Setting up after_batch: Pipeline: IntToFloatTensor

Building one batch

Applying item_tfms to the first sample:

Pipeline: ToTensor

starting from

(PILImage mode=RGB size=375x500, TensorCategory(12))

applying ToTensor gives

(TensorImage of size 3x500x375, TensorCategory(12))

Adding the next 3 samples

No before_batch transform to apply

Collating items in a batch

Error! It's not possible to collate your items in a batch

Could not collate the 0-th members of your tuples because got the following shapes:

torch.Size([3, 500, 375]),torch.Size([3, 375, 500]),torch.Size([3, 333, 50

```
0]),
torch.Size([3, 375, 500])
```

You can see exactly how we gathered the data and split it, how we went from a filename to a *sample* (the tuple (image, category)), then what item transforms were applied and how it failed to collate those samples in a batch (because of the different shapes).

Once you think your data looks right, we generally recommend the next step should be using to train a simple model. We often see people put off the training of an actual model for far too long. As a result, they don't actually find out what their baseline results look like. Perhaps your problem doesn't need lots of fancy domain-specific engineering. Or perhaps the data doesn't seem to train the model all. These are things that you want to know as soon as possible. For this initial test, we'll use the same simple model that we used in <>:

```
In [ ]: learn = cnn_learner(dls, resnet34, metrics=error_rate)
learn.fine_tune(2)
```

epoch	train_loss	valid_loss	error_rate	time
0	1.551305	0.322132	0.106225	00:19

epoch	train_loss	valid_loss	error_rate	time
0	0.529473	0.312148	0.095399	00:23
1	0.330207	0.245883	0.080514	00:24

As we've briefly discussed before, the table shown when we fit a model shows us the results after each epoch of training. Remember, an epoch is one complete pass through all of the images in the data. The columns shown are the average loss over the items of the training set, the loss on the validation set, and any metrics that we requested—in this case, the error rate.

Remember that *loss* is whatever function we've decided to use to optimize the parameters of our model. But we haven't actually told fastai what loss function we want to use. So what is it doing? fastai will generally try to select an appropriate loss function based on what kind of data and model you are using. In this case we have image data and a categorical outcome, so fastai will default to using *cross-entropy loss*.

Cross-Entropy Loss

Cross-entropy loss is a loss function that is similar to the one we used in the previous chapter, but (as we'll see) has two benefits:

- It works even when our dependent variable has more than two categories.
- It results in faster and more reliable training.

In order to understand how cross-entropy loss works for dependent variables with more than two categories, we first have to understand what the actual data and activations that are seen by the loss function look like.

Viewing Activations and Labels

Let's take a look at the activations of our model. To actually get a batch of real data from our `DataLoaders`, we can use the `one_batch` method:

```
In [ ]: x,y = dls.one_batch()
```

As you see, this returns the dependent and independent variables, as a mini-batch. Let's see what is actually contained in our dependent variable:

```
In [ ]: y
```

```
Out[ ]: TensorCategory([ 0,  5, 23, 36,  5, 20, 29, 34, 33, 32, 31, 24, 12, 36,
 8, 26, 30,  2, 12, 17,  7, 23, 12, 29, 21,  4, 35, 33,  0, 20, 26, 30,
 3,  6, 36,  2, 17, 32, 11,  6,  3, 30,  5, 26, 26, 29,  7, 36,
                    31, 26, 26,  8, 13, 30, 11, 12, 36, 31, 34, 20, 15,  8,  8, 2
 3], device='cuda:5')
```

Our batch size is 64, so we have 64 rows in this tensor. Each row is a single integer between 0 and 36, representing our 37 possible pet breeds. We can view the predictions (that is, the activations of the final layer of our neural network) using `Learner.get_preds`. This function either takes a dataset index (0 for train and 1 for valid) or an iterator of batches. Thus, we can pass it a simple list with our batch to get our predictions. It returns predictions and targets by default, but since we already have the targets, we can effectively ignore them by assigning to the special variable `_`:

```
In [ ]: preds,_ = learn.get_preds(dl=[(x,y)])
preds[0]
```

```
Out[ ]: tensor([9.9911e-01, 5.0433e-05, 3.7515e-07, 8.8590e-07, 8.1794e-05, 1.8
991e-05, 9.9280e-06, 5.4656e-07, 6.7920e-06, 2.3486e-04, 3.7872e-04, 2.
0796e-05, 4.0443e-07, 1.6933e-07, 2.0502e-07, 3.1354e-08,
                    9.4115e-08, 2.9782e-06, 2.0243e-07, 8.5262e-08, 1.0900e-07, 1.0
175e-07, 4.4780e-09, 1.4285e-07, 1.0718e-07, 8.1411e-07, 3.6618e-07, 4.
0950e-07, 3.8525e-08, 2.3660e-07, 5.3747e-08, 2.5448e-07,
                    6.5860e-08, 8.0937e-05, 2.7464e-07, 5.6760e-07, 1.5462e-08])
```

The actual predictions are 37 probabilities between 0 and 1, which add up to 1 in total:

```
In [ ]: len(preds[0]),preds[0].sum()
```

```
Out[ ]: (37, tensor(1.0000))
```

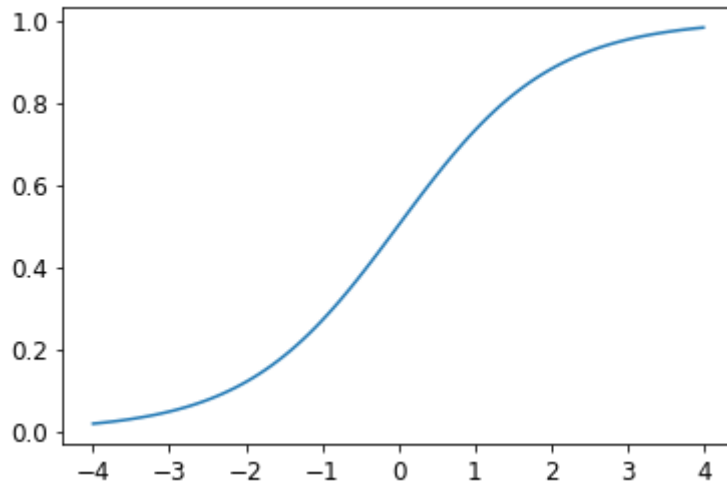
To transform the activations of our model into predictions like this, we used something called the *softmax* activation function.

Softmax

In our classification model, we use the softmax activation function in the final layer to ensure that the activations are all between 0 and 1, and that they sum to 1.

Softmax is similar to the sigmoid function, which we saw earlier. As a reminder sigmoid looks like this:

```
In [ ]: plot_function(torch.sigmoid, min=-4, max=4)
```



We can apply this function to a single column of activations from a neural network, and get back a column of numbers between 0 and 1, so it's a very useful activation function for our final layer.

Now think about what happens if we want to have more categories in our target (such as our 37 pet breeds). That means we'll need more activations than just a single column: we need an activation *per category*. We can create, for instance, a neural net that predicts 3s and 7s that returns two activations, one for each class—this will be a good first step toward creating the more general approach. Let's just use some random numbers with a standard deviation of 2 (so we multiply `randn` by 2) for this example, assuming we have 6 images and 2 possible categories (where the first column represents 3s and the second is 7s):

```
In [ ]: #hide
torch.random.manual_seed(42);
```

```
In [ ]: acts = torch.randn((6,2))*2
acts
```

```
Out[ ]: tensor([[ 0.6734,  0.2576],
                [ 0.4689,  0.4607],
                [-2.2457, -0.3727],
                [ 4.4164, -1.2760],
                [ 0.9233,  0.5347],
                [ 1.0698,  1.6187]])
```


We can't just take the sigmoid of this directly, since we don't get rows that add to 1 (i.e., we want the probability of being a 3 plus the probability of being a 7 to add up to 1):

```
In [ ]: acts.sigmoid()

Out[ ]: tensor([[0.6623, 0.5641],
               [0.6151, 0.6132],
               [0.0957, 0.4079],
               [0.9881, 0.2182],
               [0.7157, 0.6306],
               [0.7446, 0.8346]])
```

In `<>`, our neural net created a single activation per image, which we passed through the `sigmoid` function. That single activation represented the model's confidence that the input was a 3. Binary problems are a special case of classification problems, because the target can be treated as a single boolean value, as we did in `mnist_loss`. But binary problems can also be thought of in the context of the more general group of classifiers with any number of categories: in this case, we happen to have two categories. As we saw in the bear classifier, our neural net will return one activation per category.

So in the binary case, what do those activations really indicate? A single pair of activations simply indicates the *relative* confidence of the input being a 3 versus being a 7. The overall values, whether they are both high, or both low, don't matter—all that matters is which is higher, and by how much.

We would expect that since this is just another way of representing the same problem, that we would be able to use `sigmoid` directly on the two-activation version of our neural net. And indeed we can! We can just take the *difference* between the neural net activations, because that reflects how much more sure we are of the input being a 3 than a 7, and then take the sigmoid of that:

```
In [ ]: (acts[:,0]-acts[:,1]).sigmoid()

Out[ ]: tensor([0.6025, 0.5021, 0.1332, 0.9966, 0.5959, 0.3661])
```

The second column (the probability of it being a 7) will then just be that value subtracted from 1. Now, we need a way to do all this that also works for more than two columns. It turns out that this function, called `softmax`, is exactly that:

```
def softmax(x): return exp(x) / exp(x).sum(dim=1, keepdim=True)
```

jargon: Exponential function (`exp`): Literally defined as e^x , where e is a special number approximately equal to 2.718. It is the inverse of the natural logarithm function. Note that `exp` is always positive, and it increases very rapidly!

Let's check that `softmax` returns the same values as `sigmoid` for the first column, and those values subtracted from 1 for the second column:

```
In [ ]: sm_acts = torch.softmax(acts, dim=1)
        sm_acts
```

```
Out[ ]: tensor([[0.6025, 0.3975],
                [0.5021, 0.4979],
                [0.1332, 0.8668],
                [0.9966, 0.0034],
                [0.5959, 0.4041],
                [0.3661, 0.6339]])
```

`softmax` is the multi-category equivalent of `sigmoid` —we have to use it any time we have more than two categories and the probabilities of the categories must add to 1, and we often use it even when there are just two categories, just to make things a bit more consistent. We could create other functions that have the properties that all activations are between 0 and 1, and sum to 1; however, no other function has the same relationship to the sigmoid function, which we've seen is smooth and symmetric. Also, we'll see shortly that the softmax function works well hand-in-hand with the loss function we will look at in the next section.

If we have three output activations, such as in our bear classifier, calculating softmax for a single bear image would then look like `<>`.



What does this function do in practice? Taking the exponential ensures all our numbers are positive, and then dividing by the sum ensures we are going to have a bunch of numbers that add up to 1. The exponential also has a nice property: if one of the numbers in our activations x is slightly bigger than the others, the exponential will amplify this (since it grows, well... exponentially), which means that in the softmax, that number will be closer to 1.

Intuitively, the softmax function *really* wants to pick one class among the others, so it's ideal for training a classifier when we know each picture has a definite label. (Note that it may be less ideal during inference, as you might want your model to sometimes tell you it doesn't recognize any of the classes that it has seen during training, and not pick a class because it has a slightly bigger activation score. In this case, it might be better to train a model using multiple binary output columns, each using a sigmoid activation.)

Softmax is the first part of the cross-entropy loss—the second part is log likelihood.

Log Likelihood

When we calculated the loss for our MNIST example in the last chapter we used:

```
def mnist_loss(inputs, targets):
    inputs = inputs.sigmoid()
    return torch.where(targets==1, 1-inputs, inputs).mean()
```

Just as we moved from sigmoid to softmax, we need to extend the loss function to work with more than just binary classification—it needs to be able to classify any number of categories (in this case, we have 37 categories). Our activations, after softmax, are between 0 and 1, and sum to 1 for each row in the batch of predictions. Our targets are integers between 0 and 36.

In the binary case, we used `torch.where` to select between `inputs` and `1-inputs`. When we treat a binary classification as a general classification problem with two categories, it actually becomes even easier, because (as we saw in the previous section) we now have two columns, containing the equivalent of `inputs` and `1-inputs`. So, all we need to do is select from the appropriate column. Let's try to implement this in PyTorch. For our synthetic 3s and 7s example, let's say these are our labels:

```
In [ ]: targ = tensor([0,1,0,1,1,0])
```

and these are the softmax activations:

```
In [ ]: sm_acts
```

```
Out[ ]: tensor([[0.6025, 0.3975],
                [0.5021, 0.4979],
                [0.1332, 0.8668],
                [0.9966, 0.0034],
                [0.5959, 0.4041],
                [0.3661, 0.6339]])
```

Then for each item of `targ` we can use that to select the appropriate column of `sm_acts` using tensor indexing, like so:

```
In [ ]: idx = range(6)
        sm_acts[idx, targ]
```

```
Out[ ]: tensor([0.6025, 0.4979, 0.1332, 0.0034, 0.4041, 0.3661])
```

To see exactly what's happening here, let's put all the columns together in a table. Here, the first two columns are our activations, then we have the targets, and finally the result shown immediately above:

```
In [ ]: #hide_input
from IPython.display import HTML
df = pd.DataFrame(sm_acts, columns=["3", "7"])
df['targ'] = targ
df['idx'] = idx
df['loss'] = sm_acts[range(6), targ]
t = df.style.hide_index()
#To have html code compatible with our script
html = t._repr_html_().split('</style>')[1]
html = re.sub(r'<table id="([^\"]+)\s*>', r'<table >', html)
display(HTML(html))
```

3	7	targ	idx	loss
0.602469	0.397531	0	0	0.602469
0.502065	0.497935	1	1	0.497935
0.133188	0.866811	0	2	0.133188
0.99664	0.00336017	1	3	0.00336017
0.595949	0.404051	1	4	0.404051
0.366118	0.633882	0	5	0.366118

Looking at this table, you can see that the final column can be calculated by taking the `targ` and `idx` columns as indices into the two-column matrix containing the `3` and `7` columns. That's what `sm_acts[idx, targ]` is actually doing.

The really interesting thing here is that this actually works just as well with more than two columns. To see this, consider what would happen if we added an activation column for every digit (0 through 9), and then `targ` contained a number from 0 to 9. As long as the activation columns sum to 1 (as they will, if we use softmax), then we'll have a loss function that shows how well we're predicting each digit.

We're only picking the loss from the column containing the correct label. We don't need to consider the other columns, because by the definition of softmax, they add up to 1 minus the activation corresponding to the correct label. Therefore, making the activation for the correct label as high as possible must mean we're also decreasing the activations of the remaining columns.

PyTorch provides a function that does exactly the same thing as `sm_acts[range(n), targ]` (except it takes the negative, because when applying the log afterward, we will have negative numbers), called `nll_loss` (*NLL* stands for *negative log likelihood*):

```
In [ ]: -sm_acts[idx, targ]
```

```
Out[ ]: tensor([-0.6025, -0.4979, -0.1332, -0.0034, -0.4041, -0.3661])
```

```
In [ ]: F.nll_loss(sm_acts, targ, reduction='none')
```

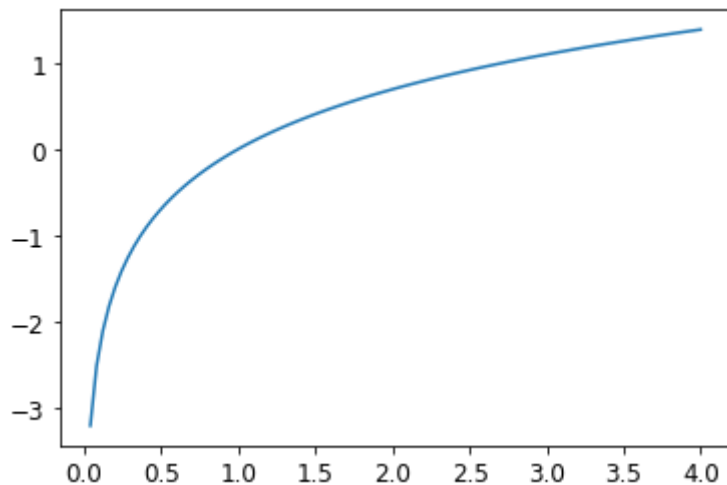
```
Out[ ]: tensor([-0.6025, -0.4979, -0.1332, -0.0034, -0.4041, -0.3661])
```

Despite its name, this PyTorch function does not take the log. We'll see why in the next section, but first, let's see why taking the logarithm can be useful.

Taking the Log

The function we saw in the previous section works quite well as a loss function, but we can make it a bit better. The problem is that we are using probabilities, and probabilities cannot be smaller than 0 or greater than 1. That means that our model will not care whether it predicts 0.99 or 0.999. Indeed, those numbers are so close together—but in another sense, 0.999 is 10 times more confident than 0.99. So, we want to transform our numbers between 0 and 1 to instead be between negative infinity and infinity. There is a mathematical function that does exactly this: the *logarithm* (available as `torch.log`). It is not defined for numbers less than 0, and looks like this:

```
In [ ]: plot_function(torch.log, min=0, max=4)
```



Does "logarithm" ring a bell? The logarithm function has this identity:

$$y = b^a$$
$$a = \log(y, b)$$

In this case, we're assuming that $\log(y, b)$ returns *log y base b*. However, PyTorch actually doesn't define \log this way: \log in Python uses the special number e (2.718...) as the base.

Perhaps a logarithm is something that you have not thought about for the last 20 years or so. But it's a mathematical idea that is going to be really critical for many things in deep learning, so now would be a great time to refresh your memory. The key thing to know about logarithms is this relationship:

$$\log(a*b) = \log(a) + \log(b)$$

When we see it in that format, it looks a bit boring; but think about what this really means. It means that logarithms increase linearly when the underlying signal increases exponentially or multiplicatively. This is used, for instance, in the Richter scale of earthquake severity, and the dB scale of noise levels. It's also often used on financial charts, where we want to show compound growth rates more clearly. Computer scientists love using logarithms, because it means that multiplication, which can create really really large and really really small numbers, can be replaced by addition, which is much less likely to result in scales that are difficult for our computers to handle.

s: It's not just computer scientists that love logs! Until computers came along, engineers and scientists used a special ruler called a "slide rule" that did multiplication by adding logarithms. Logarithms are widely used in physics, for multiplying very big or very small numbers, and many other fields.

Taking the mean of the positive or negative log of our probabilities (depending on whether it's the correct or incorrect class) gives us the *negative log likelihood* loss. In PyTorch, `nll_loss` assumes that you already took the log of the softmax, so it doesn't actually do the logarithm for you.

warning: Confusing Name, Beware: The `nll` in `nll_loss` stands for "negative log likelihood," but it doesn't actually take the log at all! It assumes you have *already* taken the log. PyTorch has a function called `log_softmax` that combines `log` and `softmax` in a fast and accurate way. `nll_loss` is designed to be used after `log_softmax`.

When we first take the softmax, and then the log likelihood of that, that combination is called *cross-entropy loss*. In PyTorch, this is available as `nn.CrossEntropyLoss` (which, in practice, actually does `log_softmax` and then `nll_loss`):

```
In [ ]: loss_func = nn.CrossEntropyLoss()
```

As you see, this is a class. Instantiating it gives you an object which behaves like a function:

```
In [ ]: loss_func(acts, targ)
```

```
Out[ ]: tensor(1.8045)
```

All PyTorch loss functions are provided in two forms, the class just shown above, and also a plain functional form, available in the `F` namespace:

```
In [ ]: F.cross_entropy(acts, targ)
```

```
Out[ ]: tensor(1.8045)
```

Either one works fine and can be used in any situation. We've noticed that most people tend to use the class version, and that's more often used in PyTorch's official docs and examples, so we'll tend to use that too.

By default PyTorch loss functions take the mean of the loss of all items. You can use `reduction='none'` to disable that:

```
In [ ]: nn.CrossEntropyLoss(reduction='none')(acts, targ)
```

```
Out[ ]: tensor([0.5067, 0.6973, 2.0160, 5.6958, 0.9062, 1.0048])
```

s: An interesting feature about cross-entropy loss appears when we consider its gradient. The gradient of `cross_entropy(a,b)` is just `softmax(a)-b`. Since `softmax(a)` is just the final activation of the model, that means that the gradient is proportional to the difference between the prediction and the target. This is the same as mean squared error in regression (assuming there's no final activation function such as that added by `y_range`), since the gradient of `(a-b)**2` is `2*(a-b)`. Because the gradient is linear, that means we won't see sudden jumps or exponential increases in gradients, which should lead to smoother training of models.

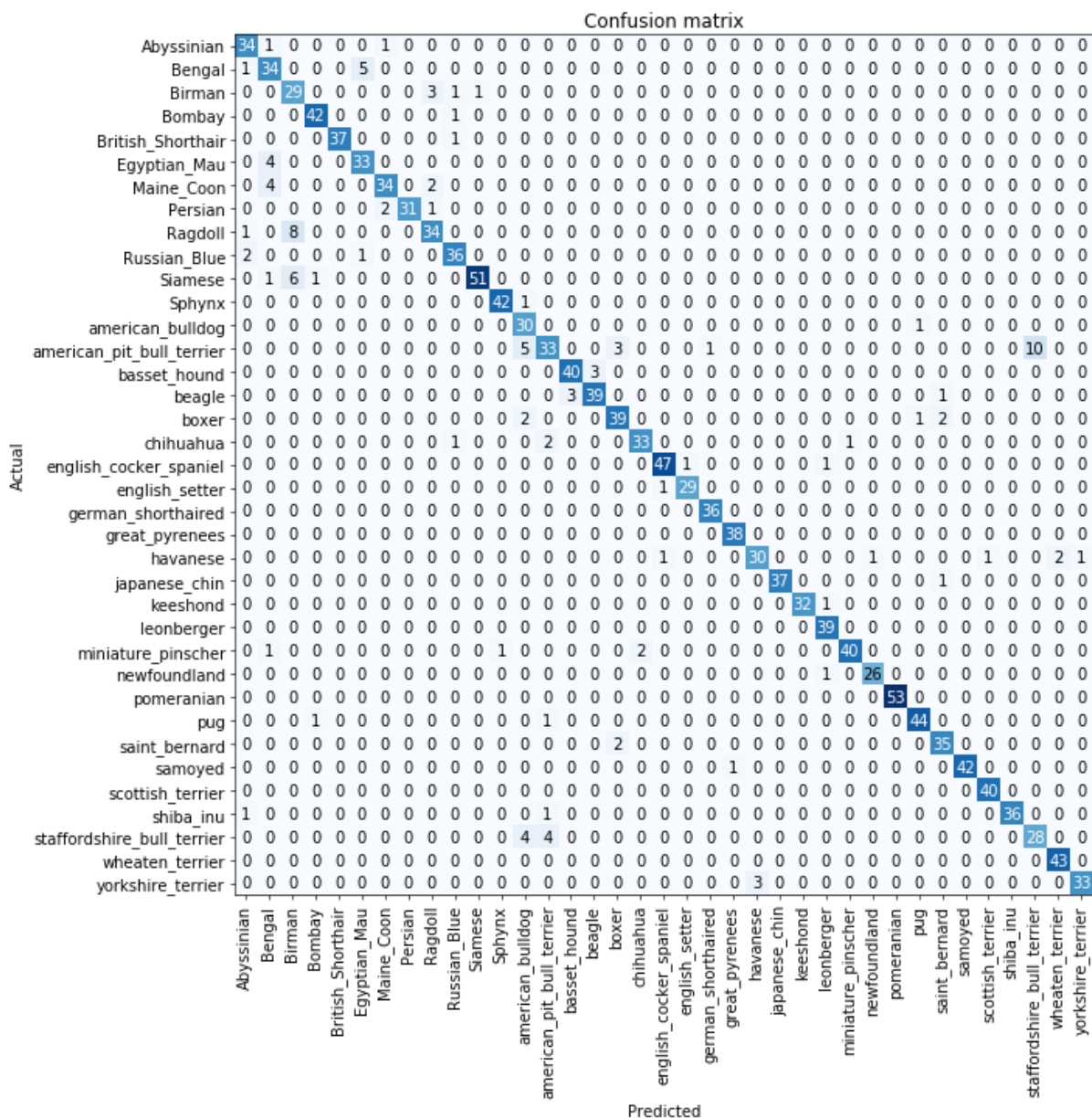
We have now seen all the pieces hidden behind our loss function. But while this puts a number on how well (or badly) our model is doing, it does nothing to help us know if it's actually any good. Let's now see some ways to interpret our model's predictions.

Model Interpretation

It's very hard to interpret loss functions directly, because they are designed to be things computers can differentiate and optimize, not things that people can understand. That's why we have metrics. These are not used in the optimization process, but just to help us poor humans understand what's going on. In this case, our accuracy is looking pretty good already! So where are we making mistakes?

We saw in <> that we can use a confusion matrix to see where our model is doing well, and where it's doing badly:

```
In [ ]: #width 600
interp = ClassificationInterpretation.from_learner(learn)
interp.plot_confusion_matrix(figsize=(12,12), dpi=60)
```



Oh dear—in this case, a confusion matrix is very hard to read. We have 37 different breeds of pet, which means we have 37×37 entries in this giant matrix! Instead, we can use the `most_confused` method, which just shows us the cells of the confusion matrix with the most incorrect predictions (here, with at least 5 or more):

```
In [ ]: interp.most_confused(min_val=5)

Out[ ]: [('american_pit_bull_terrier', 'staffordshire_bull_terrier', 10),
        ('Ragdoll', 'Birman', 8),
        ('Siamese', 'Birman', 6),
        ('Bengal', 'Egyptian_Mau', 5),
        ('american_pit_bull_terrier', 'american_bulldog', 5)]
```

Since we are not pet breed experts, it is hard for us to know whether these category errors reflect actual difficulties in recognizing breeds. So again, we turn to Google. A little bit of Googling tells us that the most common category errors shown here are actually breed differences that even expert breeders sometimes disagree about. So this gives us some comfort that we are on the right track.

We seem to have a good baseline. What can we do now to make it even better?

Improving Our Model

We will now look at a range of techniques to improve the training of our model and make it better. While doing so, we will explain a little bit more about transfer learning and how to fine-tune our pretrained model as best as possible, without breaking the pretrained weights.

The first thing we need to set when training a model is the learning rate. We saw in the previous chapter that it needs to be just right to train as efficiently as possible, so how do we pick a good one? `fastai` provides a tool for this.

The Learning Rate Finder

One of the most important things we can do when training a model is to make sure that we have the right learning rate. If our learning rate is too low, it can take many, many epochs to train our model. Not only does this waste time, but it also means that we may have problems with overfitting, because every time we do a complete pass through the data, we give our model a chance to memorize it.

So let's just make our learning rate really high, right? Sure, let's try that and see what happens:

```
In [ ]: learn = cnn_learner(dls, resnet34, metrics=error_rate)
learn.fine_tune(1, base_lr=0.1)
```

epoch	train_loss	valid_loss	error_rate	time
0	2.778816	5.150732	0.504060	00:20

epoch	train_loss	valid_loss	error_rate	time
0	4.354680	3.003533	0.834235	00:24

That doesn't look good. Here's what happened. The optimizer stepped in the correct direction, but it stepped so far that it totally overshot the minimum loss. Repeating that multiple times makes it get further and further away, not closer and closer!

What do we do to find the perfect learning rate—not too high, and not too low? In 2015 the researcher Leslie Smith came up with a brilliant idea, called the *learning rate finder*. His idea was to start with a very, very small learning rate, something so small that we would never expect it to be too big to handle. We use that for one mini-batch, find what the losses are afterwards, and then increase the learning rate by some percentage (e.g., doubling it each time). Then we do another mini-batch, track the loss, and double the learning rate again. We keep doing this until the loss gets worse, instead of better. This is the point where we know we have gone too far. We then select a learning rate a bit lower than this point. Our advice is to pick either:

- One order of magnitude less than where the minimum loss was achieved (i.e., the minimum divided by 10)
- The last point where the loss was clearly decreasing

The learning rate finder computes those points on the curve to help you. Both these rules usually give around the same value. In the first chapter, we didn't specify a learning rate, using the default value from the fastai library (which is 1e-3):

```
In [ ]: learn = cnn_learner(dls, resnet34, metrics=error_rate)
lr_min, lr_steep = learn.lr_find()
```