fastbook (/github/fastai/fastbook/tree/master)
/ 14_resnet.ipynb (/github/fastai/fastbook/tree/master/14_resnet.ipynb)

In [ ]:

```
#hide
! [ -e /content ] && pip install -Uqq fastbook
import fastbook
fastbook.setup_book()
```

In [ ]:

```
#hide
from fastbook import *
```

[[chapter_resnet]]

ResNets

In this chapter, we will build on top of the CNNs introduced in the previous chapter and explain to you the ResNet (residual network) architecture. It was introduced in 2015 by Kaiming He et al. in the article "Deep Residual Learning for Image Recognition" (https://arxiv.org/abs/1512.03385) and is by far the most used model architecture nowadays. More recent developments in image models almost always use the same trick of residual connections, and most of the time, they are just a tweak of the original ResNet.

We will first show you the basic ResNet as it was first designed, then explain to you what modern tweaks make it more performant. But first, we will need a problem a little bit more difficult than the MNIST dataset, since we are already close to 100% accuracy with a regular CNN on it.

Going Back to Imagenette

It's going to be tough to judge any improvements we make to our models when we are already at an accuracy that is as high as we saw on MNIST in the previous chapter, so we will tackle a tougher image classification problem by going back to Imagenette. We'll stick with small images to keep things reasonably fast.

Let's grab the data—we'll use the already-resized 160 px version to make things faster still, and will random crop to 128 px:

In [ ]:

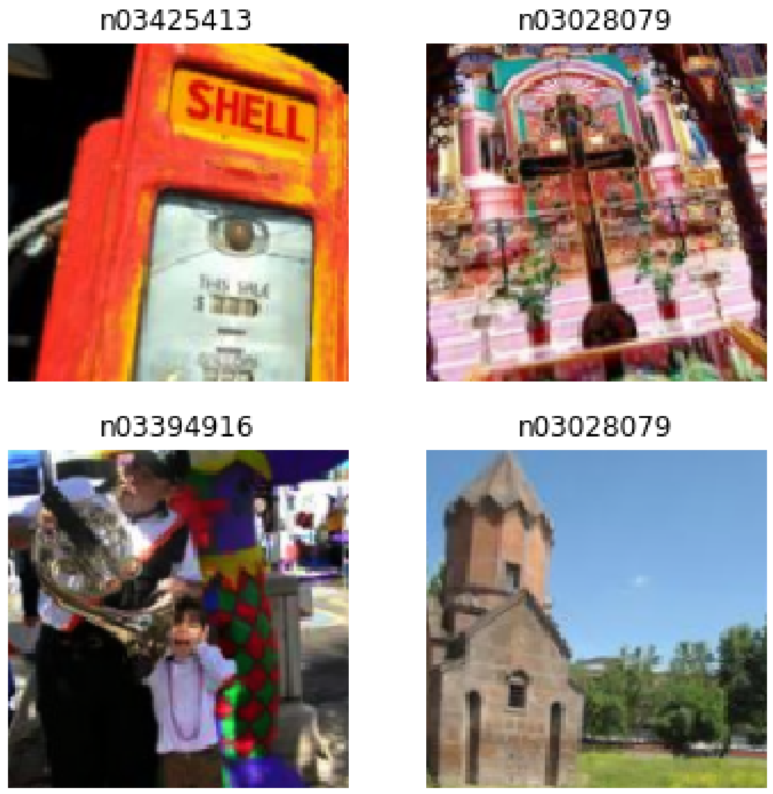
```
def get_data(url, presize, resize):
    path = untar_data(url)
    return DataBlock(
        blocks=(ImageBlock, CategoryBlock), get_items=get_image_files,
        splitter=GrandparentSplitter(valid_name='val'),
        get_y=parent_label, item_tfms=Resize(presize),
        batch_tfms=[*aug_transforms(min_scale=0.5, size=resize),
                    Normalize.from_stats(*imagenet_stats)],
    ).dataloaders(path, bs=128)
```

In [ ]:

```
dls = get_data(URLs.IMAGENETTE_160, 160, 128)
```

In [ ]:

```
dls.show_batch(max_n=4)
```



When we looked at MNIST we were dealing with 28×28-pixel images. For Imagenette we are going to be training with 128×128-pixel images. Later, we would like to be able to use larger images as well—at least as big as 224×224 pixels, the ImageNet standard. Do you recall how we managed to get a single vector of activations for each image out of the MNIST convolutional neural network?

The approach we used was to ensure that there were enough stride-2 convolutions such that the final layer would have a grid size of 1. Then we just flattened out the unit axes that we ended up with, to get a vector for each image (so, a matrix of activations for a mini-batch). We could do the same thing for Imagenette, but that would cause two problems:

- We'd need lots of stride-2 layers to make our grid 1×1 at the end—perhaps more than we would otherwise choose.
- The model would not work on images of any size other than the size we originally trained on.

One approach to dealing with the first of these issues would be to flatten the final convolutional layer in a way that handles a grid size other than 1×1. That is, we could simply flatten a matrix into a vector as we have done before, by laying out each row after the previous row. In fact, this is the approach that convolutional neural networks up until 2013 nearly always took. The most famous example is the 2013 ImageNet winner VGG, still sometimes used today. But there was another problem with this architecture: not only did it not work with images other than those of the same size used in the training set, but it required a lot of memory, because flattening out the convolutional layer resulted in many activations being fed into the final layers. Therefore, the weight matrices of the final layers were enormous.

This problem was solved through the creation of *fully convolutional networks*. The trick in fully convolutional networks is to take the average of activations across a convolutional grid. In other words, we can simply use this function:

In [ ]:

```
def avg_pool(x): return x.mean((2,3))
```

As you see, it is taking the mean over the x- and y-axes. This function will always convert a grid of activations into a single activation per image. PyTorch provides a slightly more versatile module called `nn.AdaptiveAvgPool2d` , which averages a grid of activations into whatever sized destination you require (although we nearly always use a size of 1).

A fully convolutional network, therefore, has a number of convolutional layers, some of which will be stride 2, at the end of which is an adaptive average pooling layer, a flatten layer to remove the unit axes, and finally a linear layer. Here is our first fully convolutional network:

In [ ]:

```
def block(ni, nf): return ConvLayer(ni, nf, stride=2)
def get_model():
    return nn.Sequential(
        block(3, 16),
        block(16, 32),
        block(32, 64),
        block(64, 128),
        block(128, 256),
        nn.AdaptiveAvgPool2d(1),
        Flatten(),
        nn.Linear(256, dls.c))
```

We're going to be replacing the implementation of `block` in the network with other variants in a moment, which is why we're not calling it `conv` any more. We're also saving some time by taking advantage of fastai's `ConvLayer`, which that already provides the functionality of `conv` from the last chapter (plus a lot more!).

> stop: Consider this question: would this approach makes sense for an optical character recognition (OCR) problem such as MNIST? The vast majority of practitioners tackling OCR and similar problems tend to use fully convolutional networks, because that's what nearly everybody learns nowadays. But it really doesn't make any sense! You can't decide, for instance, whether a number is a 3 or an 8 by slicing it into small pieces, jumbling them up, and deciding whether on average each piece looks like a 3 or an 8. But that's what adaptive average pooling effectively does! Fully convolutional networks are only really a good choice for objects that don't have a single correct orientation or size (e.g., like most natural photos).

Once we are done with our convolutional layers, we will get activations of size `bs x ch x h x w` (batch size, a certain number of channels, height, and width). We want to convert this to a tensor of size `bs x ch`, so we take the average over the last two dimensions and flatten the trailing 1×1 dimension like we did in our previous model.

This is different from regular pooling in the sense that those layers will generally take the average (for average pooling) or the maximum (for max pooling) of a window of a given size. For instance, max pooling layers of size 2, which were very popular in older CNNs, reduce the size of our image by half on each dimension by taking the maximum of each 2×2 window (with a stride of 2).

As before, we can define a `Learner` with our custom model and then train it on the data we grabbed earlier:

In [ ]:

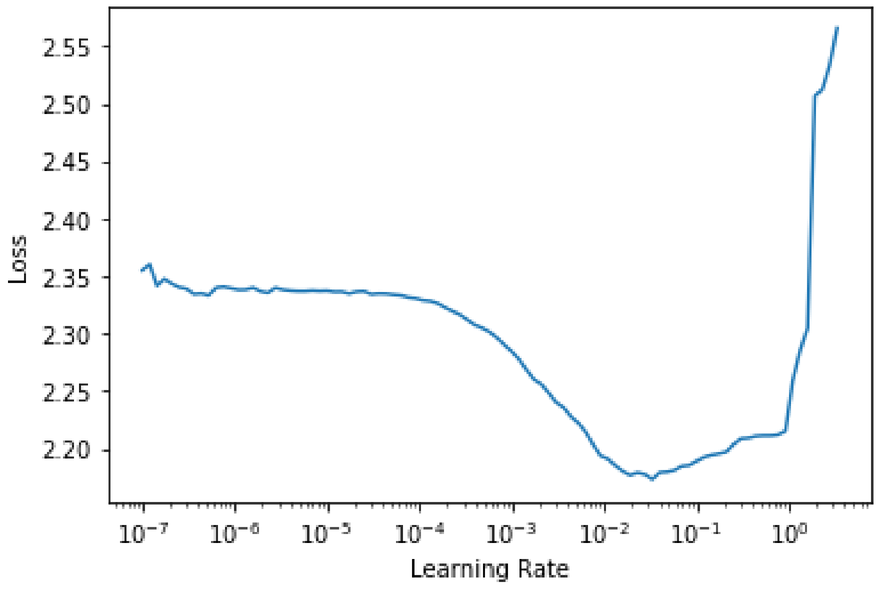
```
def get_learner(m):
    return Learner(dls, m, loss_func=nn.CrossEntropyLoss(), metrics=accuracy
                  ).to_fp16()

learn = get_learner(get_model())
```

In [ ]:

```
learn.lr_find()
```

Out[ ]:

```
(0.47863011360168456, 3.981071710586548)
```



3e-3 is often a good learning rate for CNNs, and that appears to be the case here too, so let's try that:

In [ ]:

```
learn.fit_one_cycle(5, 3e-3)
```

epoch	train_loss	valid_loss	accuracy	time
0	1.901582	2.155090	0.325350	00:07
1	1.559855	1.586795	0.507771	00:07
2	1.296350	1.295499	0.571720	00:07
3	1.144139	1.139257	0.639236	00:07
4	1.049770	1.092619	0.659108	00:07

That's a pretty good start, considering we have to pick the correct one of 10 categories, and we're training from scratch for just 5 epochs! We can do way better than this using a deeper mode, but just stacking new layers won't really improve our results (you can try and see for yourself!). To work around this problem, ResNets introduce the idea of *skip connections*. We'll explore those and other aspects of ResNets in the next section.

Building a Modern CNN: ResNet

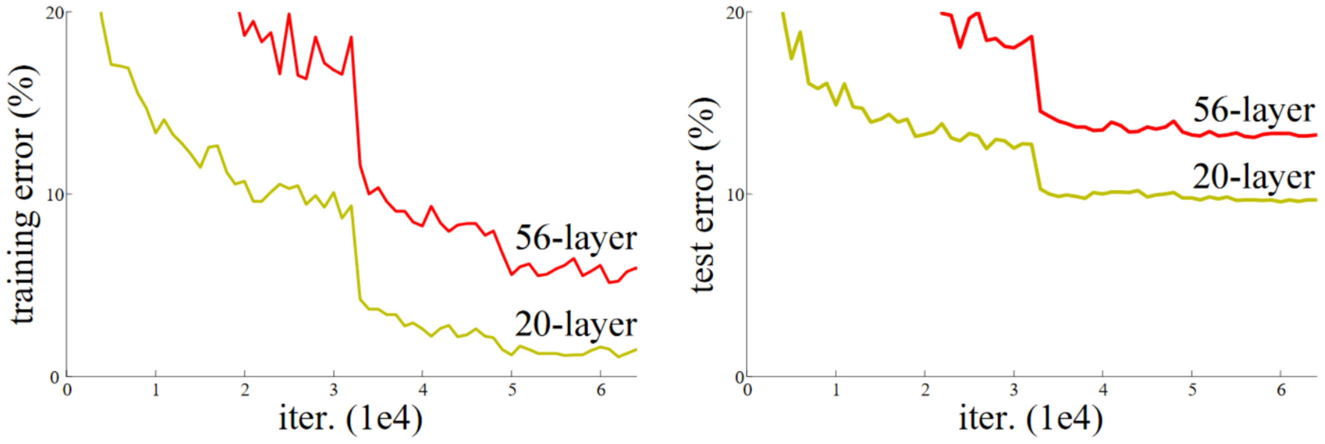
We now have all the pieces we need to build the models we have been using in our computer vision tasks since the beginning of this book: ResNets. We'll introduce the main idea behind them and show how it improves accuracy on Imagenette compared to our previous model, before building a version with all the recent tweaks.

Skip Connections

In 2015, the authors of the ResNet paper noticed something that they found curious. Even after using batchnorm, they saw that a network using more layers was doing less well than a network using fewer layers—and there were no other differences between the models. Most interestingly, the difference was observed not only in the validation set, but also in the training set; so, it wasn't just a generalization issue, but a training issue. As the paper explains:

> : Unexpectedly, such degradation is not caused by overfitting, and adding more layers to a suitably deep model leads to higher training error, as [previously reported] and thoroughly verified by our experiments.

This phenomenon was illustrated by the graph in <>, with training error on the left and test error on the right.



As the authors mention here, they are not the first people to have noticed this curious fact. But they were the first to make a very important leap:

> : Let us consider a shallower architecture and its deeper counterpart that adds more layers onto it. There exists a solution by construction to the deeper model: the added layers are identity mapping, and the other layers are copied from the learned shallower model.

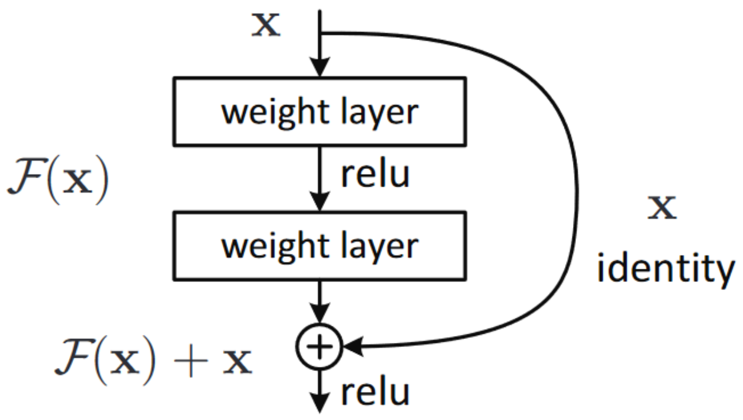
As this is an academic paper this process is described in a rather inaccessible way, but the concept is actually very simple: start with a 20-layer neural network that is trained well, and add another 36 layers that do nothing at all (for instance, they could be linear layers with a single weight equal to 1, and bias equal to 0). The result will be a 56-layer network that does exactly the same thing as the 20-layer network, proving that there are always deep networks that should be *at least as good* as any shallow network. But for some reason, SGD does not seem able to find them.

> jargon: Identity mapping: Returning the input without changing it at all. This process is performed by an *identity function*.

Actually, there is another way to create those extra 36 layers, which is much more interesting. What if we replaced every occurrence of `conv(x)` with `x + conv(x)`, where `conv` is the function from the previous chapter that adds a second convolution, then a batchnorm layer, then a ReLU. Furthermore, recall that batchnorm does `gamma*y + beta`. What if we initialized `gamma` to zero for every one of those final batchnorm layers? Then our `conv(x)` for those extra 36 layers will always be equal to zero, which means `x+conv(x)` will always be equal to `x`.

What has that gained us? The key thing is that those 36 extra layers, as they stand, are an *identity mapping*, but they have *parameters*, which means they are *trainable*. So, we can start with our best 20-layer model, add these 36 extra layers which initially do nothing at all, and then *fine-tune the whole 56-layer model*. Those extra 36 layers can then learn the parameters that make them most useful.

The ResNet paper actually proposed a variant of this, which is to instead "skip over" every second convolution, so effectively we get `x+conv2(conv1(x))`. This is shown by the diagram in <> (from the paper).


x
weight layer
relu
weight layer
$\mathcal{F}(\mathbf{x})$
x
identity
$\mathcal{F}(\mathbf{x}) + \mathbf{x}$
relu

That arrow on the right is just the `x` part of `x+conv2(conv1(x))`, and is known as the *identity branch* or *skip connection*. The path on the left is the `conv2(conv1(x))` part. You can think of the identity path as providing a direct route from the input to the output.

In a ResNet, we don't actually proceed by first training a smaller number of layers, and then adding new layers on the end and fine-tuning. Instead, we use ResNet blocks like the one in <> throughout the CNN, initialized from scratch in the usual way, and trained with SGD in the usual way. We rely on the skip connections to make the network easier to train with SGD.

There's another (largely equivalent) way to think of these ResNet blocks. This is how the paper describes it:

> : Instead of hoping each few stacked layers directly fit a desired underlying mapping, we explicitly let these layers fit a residual mapping. Formally, denoting the desired underlying mapping as H(x), we let the stacked nonlinear layers fit another mapping of F(x) := H(x)−x. The original mapping is recast into F(x)+x. We hypothesize that it is easier to optimize the residual mapping than to optimize the original, unreferenced mapping. To the extreme, if an identity mapping were optimal, it would be easier to push the residual to zero than to fit an identity mapping by a stack of nonlinear layers.

Again, this is rather inaccessible prose—so let's try to restate it in plain English! If the outcome of a given layer is `x`, when using a ResNet block that returns `y = x+block(x)` we're not asking the block to predict `y`, we are asking it to predict the difference between `y` and `x`. So the job of those blocks isn't to predict certain features, but to minimize the error between `x` and the desired `y`. A ResNet is, therefore, good at learning about slight differences between doing nothing and passing though a block of two convolutional layers (with trainable weights). This is how these models got their name: they're predicting residuals (reminder: "residual" is prediction minus target).

One key concept that both of these two ways of thinking about ResNets share is the idea of ease of learning. This is an important theme. Recall the universal approximation theorem, which states that a sufficiently large network can learn anything. This is still true, but there turns out to be a very important difference between what a network *can learn* in principle, and what it is *easy for it to learn* with realistic data and training regimes. Many of the advances in neural networks over the last decade have been like the ResNet block: the result of realizing how to make something that was always possible actually feasible.

> note: True Identity Path: The original paper didn't actually do the trick of using zero for the initial value of `gamma` in the last batchnorm layer of each block; that came a couple of years later. So, the original version of ResNet didn't quite begin training with a truly identity path through the ResNet blocks, but nonetheless having the ability to "navigate through" the skip connections did indeed make it train better. Adding the batchnorm `gamma` init trick made the models train at even higher learning rates.

Here's the definition of a simple ResNet block (where `norm_type=NormType.BatchZero` causes fastai to init the `gamma` weights of the last batchnorm layer to zero):

In [ ]:

```
class ResBlock(Module):
    def __init__(self, ni, nf):
        self.convs = nn.Sequential(
            ConvLayer(ni,nf),
            ConvLayer(nf,nf, norm_type=NormType.BatchZero))

    def forward(self, x): return x + self.convs(x)
```

There are two problems with this, however: it can't handle a stride other than 1, and it requires that `ni==nf`. Stop for a moment to think carefully about why this is.

The issue is that with a stride of, say, 2 on one of the convolutions, the grid size of the output activations will be half the size on each axis of the input. So then we can't add that back to `x` in `forward` because `x` and the output activations have different dimensions. The same basic issue occurs if `ni!=nf`: the shapes of the input and output connections won't allow us to add them together.

To fix this, we need a way to change the shape of `x` to match the result of `self.convs`. Halving the grid size can be done using an average pooling layer with a stride of 2: that is, a layer that takes 2×2 patches from the input and replaces them with their average.

Changing the number of channels can be done by using a convolution. We want this skip connection to be as close to an identity map as possible, however, which means making this convolution as simple as possible. The simplest possible convolution is one where the kernel size is 1. That means that the kernel is size `ni*nf*1*1`, so it's only doing a dot product over the channels of each input pixel—it's not combining across pixels at all. This kind of *1x1 convolution* is very widely used in modern CNNs, so take a moment to think about how it works.

> jargon: 1x1 convolution: A convolution with a kernel size of 1.

Here's a ResBlock using these tricks to handle changing shape in the skip connection:

In [ ]:

```
def _conv_block(ni,nf,stride):
    return nn.Sequential(
        ConvLayer(ni, nf, stride=stride),
        ConvLayer(nf, nf, act_cls=None, norm_type=NormType.BatchZero))
```

In [ ]:

```
class ResBlock(Module):
    def __init__(self, ni, nf, stride=1):
        self.convs = _conv_block(ni,nf,stride)
        self.idconv = noop if ni==nf else ConvLayer(ni, nf, 1, act_cls=None)
        self.pool = noop if stride==1 else nn.AvgPool2d(2, ceil_mode=True)

    def forward(self, x):
        return F.relu(self.convs(x) + self.idconv(self.pool(x)))
```

Note that we're using the `noop` function here, which simply returns its input unchanged (*noop* is a computer science term that stands for "no operation"). In this case, `idconv` does nothing at all if `ni==nf`, and `pool` does nothing if `stride==1`, which is what we wanted in our skip connection.

Also, you'll see that we've removed the ReLU (`act_cls=None`) from the final convolution in `convs` and from `idconv`, and moved it to *after* we add the skip connection. The thinking behind this is that the whole ResNet block is like a layer, and you want your activation to be after your layer.

Let's replace our `block` with `ResBlock`, and try it out:

In [ ]:

```
def block(ni,nf): return ResBlock(ni, nf, stride=2)
learn = get_learner(get_model())
```

In [ ]:

```
learn.fit_one_cycle(5, 3e-3)
```

epoch	train_loss	valid_loss	accuracy	time
0	1.973174	1.845491	0.373248	00:08
1	1.678627	1.778713	0.439236	00:08
2	1.386163	1.596503	0.507261	00:08
3	1.177839	1.102993	0.644841	00:09
4	1.052435	1.038013	0.667771	00:09

It's not much better. But the whole point of this was to allow us to train *deeper* models, and we're not really taking advantage of that yet. To create a model that's, say, twice as deep, all we need to do is replace our `block` with two `ResBlock` s in a row:

In [ ]:

```
def block(ni, nf):
    return nn.Sequential(ResBlock(ni, nf, stride=2), ResBlock(nf, nf))
```

In [ ]:

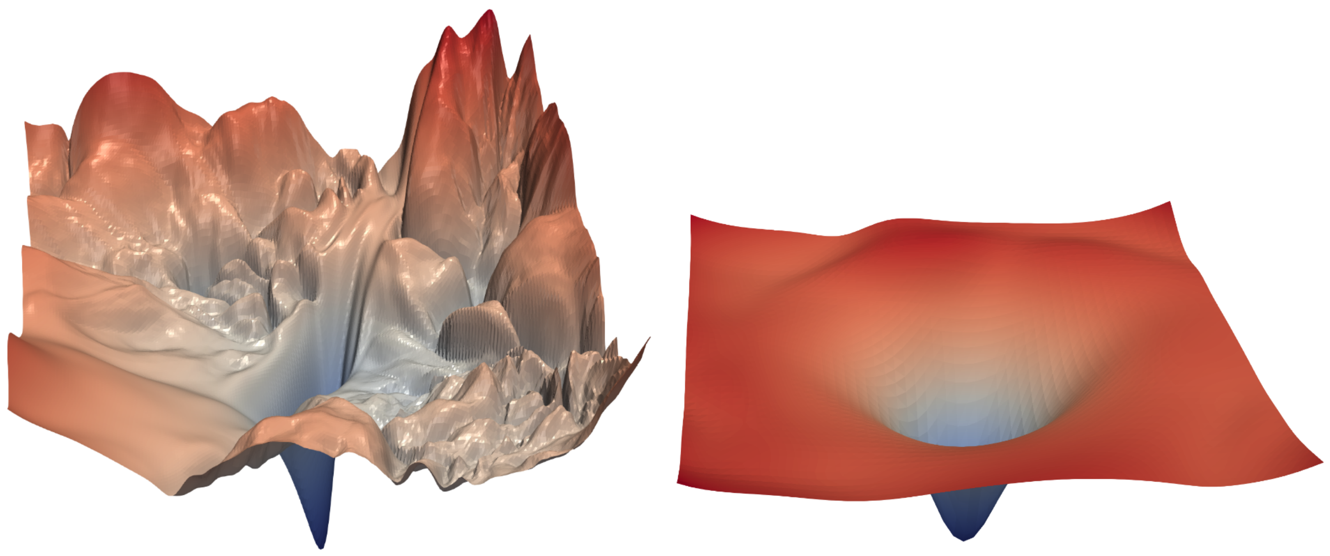
```
learn = get_learner(get_model())
learn.fit_one_cycle(5, 3e-3)
```

epoch	train_loss	valid_loss	accuracy	time
0	1.964076	1.864578	0.355159	00:12
1	1.636880	1.596789	0.502675	00:12
2	1.335378	1.304472	0.588535	00:12
3	1.089160	1.065063	0.663185	00:12
4	0.942904	0.963589	0.692739	00:12

Now we're making good progress!

The authors of the ResNet paper went on to win the 2015 ImageNet challenge. At the time, this was by far the most important annual event in computer vision. We have already seen another ImageNet winner: the 2013 winners, Zeiler and Fergus. It is interesting to note that in both cases the starting points for the breakthroughs were experimental observations: observations about what layers actually learn, in the case of Zeiler and Fergus, and observations about which kinds of networks can be trained, in the case of the ResNet authors. This ability to design and analyze thoughtful experiments, or even just to see an unexpected result, say "Hmmm, that's interesting," and then, most importantly, set about figuring out what on earth is going on, with great tenacity, is at the heart of many scientific discoveries. Deep learning is not like pure mathematics. It is a heavily experimental field, so it's important to be a strong practitioner, not just a theoretician.

Since the ResNet was introduced, it's been widely studied and applied to many domains. One of the most interesting papers, published in 2018, is Hao Li et al.'s "Visualizing the Loss Landscape of Neural Nets" (https://arxiv.org/abs/1712.09913). It shows that using skip connections helps smooth the loss function, which makes training easier as it avoids falling into a very sharp area. <> shows a stunning picture from the paper, illustrating the difference between the bumpy terrain that SGD has to navigate to optimize a regular CNN (left) versus the smooth surface of a ResNet (right).



Our first model is already good, but further research has discovered more tricks we can apply to make it better. We'll look at those next.

A State-of-the-Art ResNet

In "Bag of Tricks for Image Classification with Convolutional Neural Networks" (https://arxiv.org/abs/1812.01187), Tong He et al. study different variations of the ResNet architecture that come at almost no additional cost in terms of number of parameters or computation. By using a tweaked ResNet-50 architecture and Mixup they achieved 94.6% top-5 accuracy on ImageNet, in comparison to 92.2% with a regular ResNet-50 without Mixup. This result is better than that achieved by regular ResNet models that are twice as deep (and twice as slow, and much more likely to overfit).

> jargon: top-5 accuracy: A metric testing how often the label we want is in the top 5 predictions of our model. It was used in the ImageNet competition because many of the images contained multiple objects, or contained objects that could be easily confused or may even have been mislabeled with a similar label. In these situations, looking at top-1 accuracy may be inappropriate. However, recently CNNs have been getting so good that top-5 accuracy is nearly 100%, so some researchers are using top-1 accuracy for ImageNet too now.

We'll use this tweaked version as we scale up to the full ResNet, because it's substantially better. It differs a little bit from our previous implementation, in that instead of just starting with ResNet blocks, it begins with a few convolutional layers followed by a max pooling layer. This is what the first layers, called the *stem* of the network, look like:

In [ ]:

```
def _resnet_stem(*sizes):
    return [
        ConvLayer(sizes[i], sizes[i+1], 3, stride = 2 if i==0 else 1)
            for i in range(len(sizes)-1)
    ] + [nn.MaxPool2d(kernel_size=3, stride=2, padding=1)]
```

In [ ]:

```
#hide_output
_resnet_stem(3,32,32,64)
```

Out[ ]:

```
[ConvLayer(
   (0): Conv2d(3, 32, kernel_size=(3, 3), stride=(2, 2), padding=(1
   (1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track
   (2): ReLU()
 ), ConvLayer(
   (0): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(
   (1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track
   (2): ReLU()
 ), ConvLayer(
   (0): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1), padding=(
   (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track
   (2): ReLU()
 ), MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1, ceil_
```

```
[ConvLayer(
   (0): Conv2d(3, 32, kernel_size=(3, 3), stride=(2, 2), padding=(1,
1))
   (1): BatchNorm2d(32, eps=1e-05, momentum=0.1)
   (2): ReLU()
 ), ConvLayer(
   (0): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1))
   (1): BatchNorm2d(32, eps=1e-05, momentum=0.1)
   (2): ReLU()
 ), ConvLayer(
   (0): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1))
   (1): BatchNorm2d(64, eps=1e-05, momentum=0.1)
   (2): ReLU()
 ), MaxPool2d(kernel_size=3, stride=2, padding=1, ceil_mode=False)]
```

> jargon: Stem: The first few layers of a CNN. Generally, the stem has a different structure than the main body of the CNN.

The reason that we have a stem of plain convolutional layers, instead of ResNet blocks, is based on a very important insight about all deep convolutional neural networks: the vast majority of the computation occurs in the early layers. Therefore, we should keep the early layers as fast and simple as possible.

To see why so much computation occurs in the early layers, consider the very first convolution on a 128-pixel input image. If it is a stride-1 convolution, then it will apply the kernel to every one of the 128×128 pixels. That's a lot of work! In the later layers, however, the grid size could be as small as 4×4 or even 2×2, so there are far fewer kernel applications to do.

On the other hand, the first-layer convolution only has 3 input features and 32 output features. Since it is a 3×3 kernel, this is 3×32×3×3 = 864 parameters in the weights. But the last convolution will have 256 input features and 512 output features, resulting in 1,179,648 weights! So the first layers contain the vast majority of the computation, but the last layers contain the vast majority of the parameters.

A ResNet block takes more computation than a plain convolutional block, since (in the stride-2 case) a ResNet block has three convolutions and a pooling layer. That's why we want to have plain convolutions to start off our ResNet.

We're now ready to show the implementation of a modern ResNet, with the "bag of tricks." It uses four groups of ResNet blocks, with 64, 128, 256, then 512 filters. Each group starts with a stride-2 block, except for the first one, since it's just after a `MaxPooling` layer:

In [ ]:

```
class ResNet(nn.Sequential):
    def __init__(self, n_out, layers, expansion=1):
        stem = _resnet_stem(3,32,32,64)
        self.block_szs = [64, 64, 128, 256, 512]
        for i in range(1,5): self.block_szs[i] *= expansion
        blocks = [self._make_layer(*o) for o in enumerate(layers)]
        super().__init__(*stem, *blocks,
                         nn.AdaptiveAvgPool2d(1), Flatten(),
                         nn.Linear(self.block_szs[-1], n_out))

    def _make_layer(self, idx, n_layers):
        stride = 1 if idx==0 else 2
        ch_in,ch_out = self.block_szs[idx:idx+2]
        return nn.Sequential(*[
            ResBlock(ch_in if i==0 else ch_out, ch_out, stride if i==0 else 1
            for i in range(n_layers)
        ])
```

The `_make_layer` function is just there to create a series of `n_layers` blocks. The first one is going from `ch_in` to `ch_out` with the indicated `stride` and all the others are blocks of stride 1 with `ch_out` to `ch_out` tensors. Once the blocks are defined, our model is purely sequential, which is why we define it as a subclass of `nn.Sequential`. (Ignore the `expansion` parameter for now; we'll discuss it in the next section. For now, it'll be `1`, so it doesn't do anything.)

The various versions of the models (ResNet-18, -34, -50, etc.) just change the number of blocks in each of those groups. This is the definition of a ResNet-18:

In [ ]:

```
rn = ResNet(dls.c, [2,2,2,2])
```

Let's train it for a little bit and see how it fares compared to the previous model:

In [ ]:

```
learn = get_learner(rn)
learn.fit_one_cycle(5, 3e-3)
```

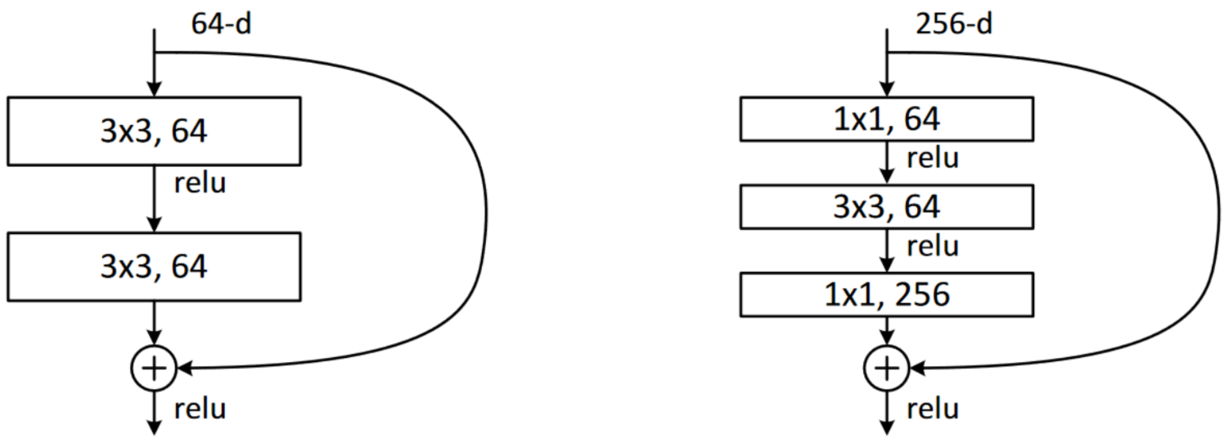
epoch	train_loss	valid_loss	accuracy	time
0	1.673882	1.828394	0.413758	00:13
1	1.331675	1.572685	0.518217	00:13
2	1.087224	1.086102	0.650701	00:13
3	0.900428	0.968219	0.684331	00:12
4	0.760280	0.782558	0.757197	00:12

Even though we have more channels (and our model is therefore even more accurate), our training is just as fast as before, thanks to our optimized stem.

To make our model deeper without taking too much compute or memory, we can use another kind of layer introduced by the ResNet paper for ResNets with a depth of 50 or more: the bottleneck layer.

Bottleneck Layers

Instead of stacking two convolutions with a kernel size of 3, bottleneck layers use three different convolutions: two 1×1 (at the beginning and the end) and one 3×3, as shown on the right in <>.



Why is that useful? 1×1 convolutions are much faster, so even if this seems to be a more complex design, this block executes faster than the first ResNet block we saw. This then lets us use more filters: as we see in the illustration, the number of filters in and out is 4 times higher (256 instead of 64) diminish then restore the number of channels (hence the name bottleneck). The overall impact is that we can use more filters in the same amount of time.

Let's try replacing our `ResBlock` with this bottleneck design:

In [ ]:

```
def _conv_block(ni,nf,stride):
    return nn.Sequential(
        ConvLayer(ni, nf//4, 1),
        ConvLayer(nf//4, nf//4, stride=stride),
        ConvLayer(nf//4, nf, 1, act_cls=None, norm_type=NormType.BatchZero))
```

We'll use this to create a ResNet-50 with group sizes of `(3,4,6,3)`. We now need to pass `4` in to the `expansion` parameter of `ResNet`, since we need to start with four times less channels and we'll end with four times more channels.

Deeper networks like this don't generally show improvements when training for only 5 epochs, so we'll bump it up to 20 epochs this time to make the most of our bigger model. And to really get great results, let's use bigger images too:

In [ ]:

```
dls = get_data(URLs.IMAGENETTE_320, presize=320, resize=224)
```

We don't have to do anything to account for the larger 224-pixel images; thanks to our fully convolutional network, it just works. This is also why we were able to do *progressive resizing* earlier in the book—the models we used were fully convolutional, so we were even able to fine-tune models trained with different sizes. We can now train our model and see the effects:

In [ ]:

```
rn = ResNet(dls.c, [3,4,6,3], 4)
```

In [ ]:

```
learn = get_learner(rn)
learn.fit_one_cycle(20, 3e-3)
```

epoch	train_loss	valid_loss	accuracy	time
0	1.613448	1.473355	0.514140	00:31
1	1.359604	2.050794	0.397452	00:31
2	1.253112	4.511735	0.387006	00:31
3	1.133450	2.575221	0.396178	00:31
4	1.054752	1.264525	0.613758	00:32
5	0.927930	2.670484	0.422675	00:32
6	0.838268	1.724588	0.528662	00:32
7	0.748289	1.180668	0.666497	00:31
8	0.688637	1.245039	0.650446	00:32
9	0.645530	1.053691	0.674904	00:31
10	0.593401	1.180786	0.676433	00:32
11	0.536634	0.879937	0.713885	00:32
12	0.479208	0.798356	0.741656	00:32
13	0.440071	0.600644	0.806879	00:32
14	0.402952	0.450296	0.858599	00:32
15	0.359117	0.486126	0.846369	00:32
16	0.313642	0.442215	0.861911	00:32
17	0.294050	0.485967	0.853503	00:32
18	0.270583	0.408566	0.875924	00:32
19	0.266003	0.411752	0.872611	00:33

We're getting a great result now! Try adding Mixup, and then training this for a hundred epochs while you go get lunch. You'll have yourself a very accurate image classifier, trained from scratch.

The bottleneck design we've shown here is typically only used in ResNet-50, -101, and -152 models. ResNet-18 and -34 models usually use the non-bottleneck design seen in the previous section. However, we've noticed that the bottleneck layer generally works better even for the shallower networks. This just goes to show that the little details in papers tend to stick around for years, even if they're actually not quite the best design! Questioning assumptions and "stuff everyone knows" is always a good idea, because this is still a new field, and there are lots of details that aren't always done well.

Conclusion

You have now seen how the models we have been using for computer vision since the first chapter are built, using skip connections to allow deeper models to be trained. Even if there has been a lot of research into better architectures, they all use one version or another of this trick, to make a direct path from the input to the end of the network. When using transfer learning, the ResNet is the pretrained model. In the next chapter, we will look at the final details of how the models we actually used were built from it.

Questionnaire

1. How did we get to a single vector of activations in the CNNs used for MNIST in previous chapters? Why isn't that suitable for Imagenette?
2. What do we do for Imagenette instead?
3. What is "adaptive pooling"?
4. What is "average pooling"?
5. Why do we need `Flatten` after an adaptive average pooling layer?
6. What is a "skip connection"?
7. Why do skip connections allow us to train deeper models?
8. What does <> show? How did that lead to the idea of skip connections?
9. What is "identity mapping"?
10. What is the basic equation for a ResNet block (ignoring batchnorm and ReLU layers)?
11. What do ResNets have to do with residuals?
12. How do we deal with the skip connection when there is a stride-2 convolution? How about when the number of filters changes?
13. How can we express a 1×1 convolution in terms of a vector dot product?
14. Create a `1x1 convolution` with `F.conv2d` or `nn.Conv2d` and apply it to an image. What happens to the `shape` of the image?
15. What does the `noop` function return?
16. Explain what is shown in <>.

17. When is top-5 accuracy a better metric than top-1 accuracy?
18. What is the "stem" of a CNN?
19. Why do we use plain convolutions in the CNN stem, instead of ResNet blocks?
20. How does a bottleneck block differ from a plain ResNet block?
21. Why is a bottleneck block faster?
22. How do fully convolutional nets (and nets with adaptive pooling in general) allow for progressive resizing?

Further Research

1. Try creating a fully convolutional net with adaptive average pooling for MNIST (note that you'll need fewer stride-2 layers). How does it compare to a network without such a pooling layer?
2. In <> we introduce *Einstein summation notation*. Skip ahead to see how this works, and then write an implementation of the 1×1 convolution operation using `torch.einsum`. Compare it to the same operation using `torch.conv2d`.
3. Write a "top-5 accuracy" function using plain PyTorch or plain Python.
4. Train a model on Imagenette for more epochs, with and without label smoothing. Take a look at the Imagenette leaderboards and see how close you can get to the best results shown. Read the linked pages describing the leading approaches.

In [ ]:

```

```