In [1]:
```
#hide
# from utils import *
from fastai.vision.all import *
```

[[chapter_intro]]

Your deep learning journey

Hello, and thank you for letting us join you on your deep learning journey, however far along that you may be! If you are a complete beginner to deep learning and machine learning, then you are most welcome here. Our only expectation is that you already know how to code, preferably in Python.

> note: If you don't have any experience coding, that's OK too! The first three chapters have been explicitly written in a way that will allow executives, product managers, etc to understand the most important things they'll need to know about deep learning. When you see bits of code in the text, try to look over them to get an intuitive sense of what they're doing. We'll explain them line by line. The details of the syntax are not nearly as important as the high level understanding of what's going on.

If you are already a confident deep learning practitioner, then you will also find a lot here. In this book we will be showing you how to achieve world-class results, including techniques from the latest research. As we will show, this doesn't require advanced mathematical training, or years of study. It just requires a bit of common sense and tenacity.

Deep learning is for everyone

A lot of people assume that you need all kinds of hard-to-find stuff to get great results with deep learning, but, as you'll see in this book, those people are wrong. Here's a list of a few thing you **absolutely don't need** to do world-class deep learning:

```
asciidoc
[[myths]]
.What you don't need to do deep learning
[options="header"]
|======
| Myth (don't need) | Truth
| Lots of math | Just high school math is sufficient
| Lots of data | We've seen record-breaking results with <50 items
of data
| Lots of expensive computers | You can get what you need for state
of the art work for free
|======
```

Deep learning is a computer technique to extract and transform data – with use cases ranging from human speech recognition to animal imagery classification – by using multiple layers of neural networks. Each of these layers takes the inputs from previous layers and progressively refines them. The algorithms involved can train the layers by learning to minimize errors and improve their own accuracy (we will discuss those in details in the next section).

Deep learning has power, flexibility, and simplicity. That's why we believe it should be applied across many disciplines. These include the social and physical sciences, the arts, medicine, finance, scientific research, and much more. To give a personal example, despite having no background in medicine, Jeremy started Enlitic, a company that uses deep learning algorithms to diagnose illness and disease. Within months of starting the company, it was announced that their algorithm could identify malignant tumors more accurately than radiologists.

Here's a list of some of the thousands of tasks where deep learning, or methods heavily using deep learning, is now the best in the world:

- NLP:: answering questions; speech recognition; summarizing documents; classifying documents; finding names, dates, etc. in documents; searching for articles mentioning a concept
- Computer vision:: satellite and drone imagery interpretation (e.g. for disaster resilience); face recognition; image captioning; reading traffic signs; locating pedestrians and vehicles in autonomous vehicles
- Medicine:: Finding anomalies in radiology images, including CT, MRI, and x-ray; counting features in pathology slides; measuring features in ultrasounds; diagnosing diabetic retinopathy
- Biology:: folding proteins; classifying proteins; many genomics tasks, such as tumor-normal sequencing and classifying clinically actionable genetic mutations; cell classification; analyzing protein/protein interactions
- Image generation:: Colorizing images; increasing image resolution; removing noise from images; converting images to art in the style of famous artists
- Recommendation systems:: web search; product recommendations; home page layout
- Playing games:: Better than humans and better than any other computer algorithm at Chess, Go, most Atari videogames, many real-time strategy games
- Robotics:: handling objects that are challenging to locate (e.g. transparent, shiny, lack of texture) or hard to pick up
- Other applications:: financial and logistical forecasting; text to speech; much much more...

What is remarkable is that deep learning has such varied application yet nearly all of deep learning is based on a single type of model, the neural network.

But neural networks are not in fact completely new. In order to have a wider perspective on the field, it is worth it to start with a bit of history.

Neural networks: a brief history

In 1943 Warren McCulloch, a neurophysiologist, and Walter Pitts, a logician, teamed up to develop a mathematical model of an artificial neuron. They declared that:

> : *Because of the "all-or-none" character of nervous activity, neural events and the relations among them can be treated by means of propositional logic. It is found that the behavior of every net can be described in these terms*. (Pitts and McCulloch; A Logical Calculus of the Ideas Immanent in Nervous Activity)

They realised that a simplified model of a real neuron could be represented using simple addition and thresholding as shown in <>. Pitts was self-taught, and, by age 12, had received an offer to study at Cambridge with the great Bertrand Russell. He did not take up this invitation, and indeed throughout his life did not accept any offers of advanced degrees or positions of authority. Most of his famous work was done whilst he was homeless. Despite his lack of an officially recognized position, and increasing social isolation, his work with McCulloch was influential, and was picked up by a psychologist named Frank Rosenblatt.

Natural and artificial neurons

Rosenblatt further developed the artificial neuron to give it the ability to learn. Even more importantly, he worked on building the first device that actually used these principles: The Mark I Perceptron. Rosenblatt wrote about this work: "we are about to witness the birth of such a machine – a machine capable of perceiving, recognizing and identifying its surroundings without any human training or control". The perceptron was built, and was able to successfully recognize simple shapes.

An MIT professor named Marvin Minsky (who was a grade behind Rosenblatt at the same high school!) along with Seymour Papert wrote a book, called "Perceptrons", about Rosenblatt's invention. They showed that a single layer of these devices was unable to learn some simple, critical mathematical functions (such as XOR). In the same book, they also showed that using multiple layers of the devices would allow these limitations to be addressed. Unfortunately, only the first of these insights was widely recognized, as a result of which the global academic community nearly entirely gave up on neural networks for the next two decades.

Perhaps the most pivotal work in neural networks in the last 50 years is the multi-volume *Parallel Distributed Processing* (PDP), released in 1986 by MIT Press. Chapter 1 lays out a similar hope to that shown by Rosenblatt:

> : *…people are smarter than today's computers because the brain employs a basic computational architecture that is more suited to deal with a central aspect of the natural information processing tasks that people are so good at. …we will introduce a computational framework for modeling cognitive processes that seems… closer than other frameworks to the style of computation as it might be done by the brain.* (PDP, chapter 1)

The premise that PDP is using here is that traditional computer programs work very differently to brains, and that might be why computer programs had (at that point) been so bad at doing things that brains find easy (such as recognizing objects in pictures). The authors claim that the PDP approach is "closer than other frameworks" to how the brain works, and therefore it might be better able to handle these kinds of tasks.

In fact, the approach laid out in PDP is very similar to the approach used in today's neural networks. The book defined "Parallel Distributed Processing" as requiring:

1. A set of *processing units*
2. A *state of activation*
3. An *output function* for each unit
4. A *pattern of connectivity* among units
5. A *propagation rule* for propagating patterns of activities through the network of connectivities
6. An *activation rule* for combining the inputs impinging on a unit with the current state of that unit to produce a new level of activation for the unit
7. A *learning rule* whereby patterns of connectivity are modified by experience
8. An *environment* within which the system must operate

We will see in this book that modern neural networks handle each of these requirements.

In the 1980's most models were built with a second layer of neurons, thus avoiding the problem that had been identified by Minsky (this was their "pattern of connectivity among units", to use the framework above). And indeed, neural networks were widely used during the 80s and 90s for real, practical projects. However, again a misunderstanding of the theoretical issues held back the field. In theory, adding just one extra layer of neurons was enough to allow any mathematical model to be approximated with these neural networks, but in practice such networks were often too big and slow to be useful.

Although researchers showed 30 years ago that to get practical good performance you need to use even more layers of neurons, it is only in the last decade that this has been more widely appreciated. Neural networks are now finally living up to their potential, thanks to the understanding to use more layers as well as improved ability to do so thanks to improvements in computer hardware, increases in data availability, and algorithmic tweaks that allow neural networks to be trained faster and more easily. We now have what Rosenblatt had promised: "a machine capable of perceiving, recognizing and identifying its surroundings without any human training or control".

This is what you will learn how to build them in this book.

What you will learn

To be exact, after reading this book you will know:

- How to train models that achieve state of the art results in:
    - Computer vision: Image classification (e.g. classify pet photos by breed), and image localization and detection (e.g. find where the animals in an image are)
    - Natural Language Processing (NLP): Document classification (e.g. movie review sentiment analysis), and language modelling
    - Tabular data (e.g. sales prediction) with categorical data, continuous data, and mixed data, including time series

    - Collaborative filtering (e.g. movie recommendation)
- How to turn your models into web applications
- Why and how deep learning models work, and how to use that knowledge to improve the accuracy, speed, and reliability of your models
- The latest deep learning techniques which really matter in practice
- How to read a deep learning research paper
- How to implement deep learning algorithms from scratch
- How to think about ethical implications of your work, to help ensure that you're making the world a better place, and that your work isn't misused for harm

See the table of contents for a complete list; but to give you a taste, here's some of the techniques covered (don't worry if none of these words mean anything to you yet – you'll learn them all soon): Affine functions and non-linearities; Parameters and activations; Random init and transfer learning; SGD, Momentum, Adam and more optimizers; Convolutions; Batch normalization; Dropout; Data augmentation; Weight decay; Resnet and Densenet architectures; Image classification and regression; Embeddings; Recurrent neural networks (RNNs); Transformers; Segmentation; U-net; Generative Adversarial Networks (GANs), and much more.

> note: If you look at the end of each chapter, you'll find a questionnaire. That's a great place also to see what we cover in each chapter, since (we hope!) by the end of each chapter you'll be able to answer all the questions there. In fact, one of our reviewers (thanks Fred!) said that he likes to read the questionnaire *first*, before reading the chapter, so that way he knows what to look for.

Who we are

Since we are going to be spending a lot of time together, let's get to know each other a bit… We are Sylvain and Jeremy, your guides on this journey. We hope that you will find us well suited for this position.

Jeremy has been using and teaching machine learning for around 30 years. He started using neural networks 25 years ago. During this time, he has led many companies and projects which have machine learning at their core, including founding the first company to focus on deep learning and medicine, Enlitic, and taking on the role of President and Chief Scientist of the world's largest machine learning community, Kaggle. He is the co-founder, along with Dr Rachel Thomas, of fast.ai, the organisation that built the course this book is based on.

From time to time you will hear directly from us, in sidebars like this one from Jeremy:

> J: Hi everybody, I'm Jeremy! You might be interested to know that I do not have any formal technical education. I completed a Bachelor of Arts, with a major in philosophy, and didn't do very well in my university grades. I was much more interested in doing real projects, rather than theoretical studies, so I worked full-time at a management consulting firm called McKinsey and Company throughout my degree. If you're somebody who would rather get their hands dirty building

> stuff rather than spend years learning abstract concepts, then you will understand where I am coming from! Look out for sidebars from me to find information most suited to people with a less mathematical or formal technical background—that is, people like me…

Sylvain, on the other hand, knows a lot about formal technical education. In fact, he has written 10 maths textbooks, covering the entire advanced French maths curriculum!

> S: Unlike Jeremy, I have not spent many years coding and applying machine learning algorithms. Rather, I recently came to the machine learning world, by watching Jeremy's fast.ai course videos. So, if you are somebody who has not opened a terminal and written commands at the command line, then you will understand where I am coming from! Look out for sidebars from me to find information most suited to people with a more mathematical or formal technical background, but less real-world coding—that is, people like me…

The fast.ai course has been studied by hundreds of thousands of students, from all walks of life, from all parts of the world. Sylvain stood out as the most impressive student of the course that Jeremy had ever seen, which led to him joining fast.ai, and then becoming the co-author, along with Jeremy, of the fastai software library.

All this means that you have the best of both worlds: the people who know more about the software than anybody, because they wrote it, an expert on maths, and an expert on coding and machine learning, but also people who understand what it feels like to be a relative outsider in maths, and a relative outsider in coding and machine learning.

Anybody who has watched sports knows that if you have a two-person commentary team then you also need a third person to do "special comments". Our special commentator is Alexis Gallagher. Alexis has a very diverse background: he has been a zoology researcher, screenplay writer, an improv performer, a McKinsey consultant (like Jeremy!), a Swift coder, and a CTO.

> A: I've decided it's time for me to learn about this AI stuff! After all, I've tried pretty much everything else… But I don't really have a background in machine learning, or in Python. Still… how hard can it be? I'm going to be learning throughout this book, just like you are. Look out for my sidebars for learning tips that I found helpful on my journey, and hopefully you will find helpful too.

How to learn deep learning

Harvard professor David Perkins, who wrote Making Learning Whole, has much to say about teaching. The basic idea is to teach the *whole game*. That means that's if you're teaching baseball, you first take people to a baseball game or get them to play it. You don't teach them how to line thread into a ball, the physics of a parabola, or the coefficient of friction of a ball on a bat.

Paul Lockhart, a Columbia math PhD, former Brown professor, and K-12 math teacher, imagines in the influential essay A Mathematician's Lament a nightmare world where music and art are taught the way math is taught. Children would not be allowed to listen to or play music until they have spent over a decade mastering music notation and theory, spending classes transposing sheet music into a different key. In art class, students study colours and applicators, but aren't allowed to actually paint until college. Sound absurd? This is how math is taught–we require students to spend years doing rote memorization, and learning dry, disconnected *fundamentals* that we claim will pay off later, long after most of them quit the subject.

Unfortunately, this is where many teaching resources on deep learning begin–asking learners to follow along with the definition of the Hessian and theorems for the Taylor approximation of your loss functions, without ever giving examples of actual working code. We're not knocking calculus. We love calculus and have even taught it at the college level, but we don't think it's the best place to start when learning deep learning!

In deep learning, it really helps if you have the motivation to fix your model to get it to do better. That's when you start learning the relevant theory. But you need to have the model in the first place. We teach almost everything through real examples. As we build out those examples, we go deeper and deeper, and we'll show you how to make your projects better and better. This means that you'll be gradually learning all the theoretical foundations you need, in context, in a way that you'll see why it matters and how it works.

So, here's our commitment to you. Throughout this book, we will follow these principles:

- Teaching the *whole game* – starting off by showing how to use a complete, working, very usable, state of the art deep learning network to solve real world problems, by using simple, expressive tools. And then gradually digging deeper and deeper into understanding how those tools are made, and how the tools that make those tools are made, and so on…
- Always teaching through examples: ensuring that there is a context and a purpose that you can understand intuitively, rather than starting with algebraic symbol manipulation ;
- Simplifying as much as possible: we've spent years building tools and teaching methods that make previously complex topics very simple ;
- Removing barriers: deep learning has, until now, been a very exclusive game. We're breaking it open, and ensuring that everyone can play.

The hardest part of deep learning is artisanal: how do you know if you've got enough data; whether it is in the right format; if your model is training properly; and if it's not, what should you do about it? That is why we believe in learning by doing. As with basic data science skills, with deep learning you only get better through practical experience. Trying to spend too much time on the theory can be counterproductive. The key is to just code and try to solve problems: the theory can come later, when you have context and motivation.

There will be times when the journey will feel hard. Times where you feel stuck. Don't give up! Rewind through the book to find the last bit where you definitely weren't stuck, and then read slowly through from there to find the first thing that isn't clear. Then try some code experiments yourself, and Google around for more tutorials on whatever the issue you're stuck with is--often you'll find some different angle on the material which might help it to click. Also, it's expected and normal to not understand everything (especially the code) on first reading. Trying to understand the material serially before proceeding can sometimes be hard. Sometimes things click into place after you got more context from parts down the road, from having a bigger picture. So if you do get stuck on a section, try moving on anyway and make a note to come back to it later.

Remember, you don't need any particular academic background to succeed at deep learning. Many important breakthroughs are made in research and industry by folks without a PhD, such as the paper Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks, one of the most influential papers of the last decade, with over 5000 citations, which was written by Alec Radford when he was an under-graduate. Even at Tesla, where they're trying to solve the extremely tough challenge of making a self-driving car, CEO Elon Musk says:

> : "A PhD is definitely not required. All that matters is a deep understanding of AI & ability to implement NNs in a way that is actually useful (latter point is what’s truly hard). Don’t care if you even graduated high school."

What you will need to succeed however is to apply what you learn in this book to a personal project and always persevere.

Your projects and your mindset

Whether you're excited to identify if plants are diseased from pictures of their leaves, auto-generate knitting patterns, diagnose TB from x-rays, or determine when a raccoon is using your cat door, we will get you using deep learning on your own problems (via pre-trained models from others) as quickly as possible, and then will progressively drill into more details. You'll learn how to use deep learning to solve your own problems at state-of-the-art accuracy within the first 30 minutes of the next chapter! (And feel free to skip straight there now if you're dying to get coding right away.) There is a pernicious myth out there that you need to have computing resources and datasets the size of those at Google to be able to do deep learning, and it's not true.

So, what sort of tasks make for good test cases? You could train your model to distinguish between Picasso and Monet paintings or to pick out pictures of your daughter instead of pictures of your son. It helps to focus on your hobbies and passions–setting yourself four of five little projects rather than striving to solve a big, grand problem tends to work better when you're getting started. Since it is easy to get stuck, trying to be too ambitious too early can often backfire. Then, once you've got the basics mastered, aim to complete something you're really proud of!

> J: Deep learning can be set to work on almost any problem. For instance, my first startup was a company called FastMail, which provided enhanced email services when it launched in 1999 (and still does to this day). In 2002 I set it up to use a

> primitive form of deep learning – single-layer neural networks – to help categorise emails and stop customers from receiving spam.

Common character traits in the people that do well at deep learning include playfulness and curiosity. The late physicist Richard Feynman is an example of someone who we'd expect to be great at deep learning: his development of an understanding of the movement of subatomic particles came from his amusement at how plates wobble when they spin in the air.

Let's now focus on what you will learn, starting with the software.

The software: PyTorch, fastai, and Jupyter (and why it doesn't matter)

We've completed hundreds of machine learning projects using dozens of different packages, and many different programming languages. At fast.ai, we have written courses using most of the main deep learning and machine learning packages used today. After PyTorch came out in 2017 we spent over a thousand hours testing it before deciding that we would use it for future courses, software development, and research. Since that time PyTorch has become the world's fastest-growing deep learning library and is already used for most research papers at top conferences. This is generally a leading indicator of usage in industry, because these are the papers that end up getting used in products and services commercially. We have found that PyTorch is the most flexible and expressive library for deep learning. It does not trade off speed for simplicity, but provides both.

PyTorch works best as a low-level foundation library, providing the basic operations for higher level functionality. The fastai library is the most popular library for adding this higher-level functionality on top of PyTorch. It's also particularly well suited for the purposes of this book, because it is unique in providing a deeply layered software architecture (there's even a peer-reviewed academic paper about this layered API). In this book, as we go deeper and deeper into the foundations of deep learning, we will also go deeper and deeper into the layers of fastai. This book covers version 2 of the fastai library, which is a from-scratch rewrite providing many unique features.

However, it doesn't really matter what software you learn, because it takes only a few days to learn to switch from one library to another. What really matters is learning the deep learning foundations and techniques properly. Our focus will be on using code which as clearly as possible expresses the concepts that you need to learn. Where we are teaching high-level concepts, we will use high level fastai code. Where we are teaching low-level concepts, we will use low-level PyTorch, or even pure Python code.

If it feels like new deep learning libraries are appearing at a rapid pace nowadays, then you need to be prepared for a much faster rate of change in the coming months and years. As more people enter the field, they will bring more skills and ideas, and try more things. You should assume that whatever specific libraries and software you learn today will be obsolete in a year or two. Just think about the number of changes of libraries and technology stacks that occur all the time in the world of web programming — and yet this is a much more mature and slowgrowing area than deep learning. We strongly believe that the focus in learning needs to be on understanding the underlying techniques and how to apply them in practice, and how to quickly build expertise in new tools and techniques as they are released.

By the end of the book, you'll understand nearly all the code that's inside fastai (and much of PyTorch too), because each chapter we'll be digging a level deeper to understand exactly what's going on as we build and train our models. This means that you'll have learnt the most important best practices used in modern deep learning—not just how to use them, but how they really work and are implemented. If you want to use those approaches in another framework, you'll have the knowledge you need to develop it if needed.

Since the most important thing for learning deep learning is writing code and experimenting, it's important that you have a great platform for experimenting with code. The most popular programming experimentation platform is called Jupyter. This is what we will be using throughout this book. We will show you how you can use Jupyter to train and experiment with models and introspect every stage of the data pre-processing and model development pipeline. Jupyter is the most popular tool for doing data science in Python, for good reason. It is powerful, flexible, and easy to use. We think you will love it!

Let's see it in practice and train our first model.

Your first model

As we said before, we will teach how to do things before we explain why they work. Following this top-down approach, we will begin by actually training an image classifier to recognize dogs and cats with almost 100% accuracy. To train this model and run our experiments, you will need some initial setup. Don't worry, it's not as hard as it looks like.

> s: Do not skip the setup part even if it looks intimidating at first, especially if you have little or no experience using things like a terminal or the command line. Most of that is actually not necessary and you will find that the easiest servers can be setup with just your usual web browser. It is crucial that you run your own experiments in parallel with this book in order to learn.

Getting a GPU deep learning server

To do nearly everything in this book, you'll need access to a computer with an NVIDIA GPU (unfortunately other brands of GPU are not fully supported by the main deep learning libraries). However, we don't recommend you buy one; in fact, even if you already have one, we don't suggest you use it just yet! Setting up a computer takes time and energy, and you want all your energy to focus on deep learning right now. Therefore, we instead suggest you rent access to a computer that already has everything you need preinstalled and ready to go. Costs can be as little as US$0.25 per hour while you're using it, and some options are even free.

> jargon: (Graphic Processing Unit) GPU: Also known as a *graphics card*. A special kind of processor in your computer than can handle thousands of single tasks at

> the same time, especially designed for displaying 3D environments on a computer for playing games. These same basic tasks are very similar to what neural networks do, such that GPUs can run neural networks hundreds of times faster than regular CPUs. All modern computers contain a GPU, but few contain the right kind of GPU necessary for deep learning.

The best choice for GPU servers for use with this book change over time, as companies come and go, and prices change. We keep a list of our recommended options on the book website. So, go there now, and follow the instructions to get connected to a GPU deep learning server. Don't worry, it only takes about two minutes to get set up on most platforms, and many don't even require any payment, or even a credit card to get started.

> A: My two cents: heed this advice! If you like computers you will be tempted to setup your own box. Beware! It is feasible but surprisingly involved and distracting. There is a good reason this book is not titled, *Everything you ever wanted to know about Ubuntu system administration, NVIDIA driver installation, apt-get, conda, pip, and Jupyter notebook configuration*. That would be a book of its own. Having designed and deployed our production machine learning infrastructure at work, I can testify it has its satisfactions but it is as unrelated to modelling as maintaining an airplane is to flying one.

Each option shown on the book website includes a tutorial; after completing the tutorial, you will end up with a screen looking like <>.

Initial view of Jupyter Notebooks

You are now ready to run your first Jupyter notebook!

> jargon: Jupyter Notebook: A piece of software that allows you to include formatted text, code, images, videos, and much more, all within a single interactive document. Jupyter received the highest honor for software, the ACM Software System Award, thanks to its wide use and enormous impact in many academic fields, and in industry. Jupyter Notebook is the most widely used software by data scientists for developing and interacting with deep learning models.

Running your first notebook

The notebooks are labelled by chapter, and then by notebook number, so that they are in the same order as they are presented in this book. So, the very first notebook you will see listed, is the notebook that we need to use now. You will be using this notebook to train a model that can recognize dog and cat photos. To do this, we'll be downloading a *dataset* of dog and cat photos, and using that to *train a model*. A *dataset* simply refers to a bunch of data—it could be images, emails, financial indicators, sounds, or anything else. There are many datasets made freely available that are suitable for training models. Many of these datasets are created by academics to help advance research, many are made available for competitions (there are competitions where data scientists can compete to see who has the most accurate model!), and some are by-products of other processes (such as financial filings).

> note: There are two folders containing different versions of the notebooks. The **full** folder contains the exact notebooks used to create the book you're reading now, with all the prose and outputs. The **stripped** version has the same headings and code cells, but all outputs and prose have been removed. After reading a section of the book, we recommend working through the stripped notebooks, with the book closed, and see if you can figure out what each cell will show before you execute it. And try to recall what the code is demonstrating.

To open a notebook, just click on it. The notebook will open, and it will look something like <> (note that there may be slight differences in details across different platforms; you can ignore those differences):

An example of notebook

A notebook consists of *cells*. There are two main types of cell:

- Cells containing formatted text, images, and so forth. These use a format called *markdown*, which we will learn about soon
- Cells containing code, which can be executed, and outputs will appear immediately underneath (which could be plain text, tables, images, animations, sounds, or even interactive applications)

Jupyter notebooks can be in one of two modes, edit mode, or command mode. In edit mode typing the keys on your keyboard types the letters into the cell in the usual way. However, in command mode, you will not see any flashing cursor, and the keys on your keyboard will each have a special function.

Let's make sure that you are in command mode before continuing: press "escape" now on your keyboard to switch to command mode (if you are already in command mode, then this does nothing, so press it now just in case). To see a complete list of all of the functions available, press "h"; press "escape" to remove this help screen. Notice that in command mode, unlike most programs, commands do not require you to hold down "control", "alt", or similar — you simply press the required letter key.

You can make a copy of a cell by pressing "c" (it needs to be selected first, indicated with an outline around the cell; if it is not already selected, click on it once). Then press "v" to paste a copy of it.

When you click on a cell it will be selected. Click on the cell now which begins with the line "# CLICK ME". The first character in that line represents a comment in Python, so is ignored when executing the cell. The rest of the cell is, believe it or not, a complete system for creating and training a state-of-the-art model for recognizing cats versus dogs. So, let's train it now! To do so, just press shift-enter on your keyboard, or press the "play" button on the toolbar. Then, wait a few minutes while the following things happen:

1. A dataset containing called the Oxford-IIT Pet Dataset that contains 7,349 images of cats and dogs from 37 different breeds will be downloaded from the fast.ai datasets collection to the GPU server you are using, and will then be extracted
2. A *pretrained model* will be downloaded from the Internet, which has already been trained on 1.3 million images, using a competition winning model
3. The pretrained model will be *fine-tuned* using the latest advances in transfer learning, to create a model that is specially customised for recognising dogs and cats

The first two steps only need to be run once on your GPU server. If you run the cell again, it will use the dataset and model that have already been downloaded, rather than downloading them again.

In [2]:

```
#id first_training
#caption Results from the first training
# CLICK ME
from fastai.vision.all import *
path = untar_data(URLs.PETS)/'images'

def is_cat(x): return x[0].isupper()
dls = ImageDataLoaders.from_name_func(
    path, get_image_files(path), valid_pct=0.2, seed=42,
    label_func=is_cat, item_tfms=Resize(224))

learn = cnn_learner(dls, resnet34, metrics=error_rate)
learn.fine_tune(1)
```

```
100.00% [811712512/811706944 00:18<00:00]
```

epoch	train_loss	valid_loss	error_rate	time
0	0.179891	0.031254	0.006089	00:11

epoch	train_loss	valid_loss	error_rate	time
0	0.058267	0.024366	0.007442	00:12

You will probably not see exactly the same results that are in the book. There are a lot of sources of small random variation involved in training models. We generally see an error rate of well less than 0.02 in this example.

> important: Depending on your network speed, it might take a few minutes to download the pretrained model and dataset. Running `fine_tune` might take a minute or so. Often models in this book take a few minutes to train, as will your own models. So it's a good idea to come up with good techniques to make the most of this time. For instance, keep reading the next section while your model trains, or open up another notebook and use it for some coding experiments.

Sidebar: This book was written in Jupyter Notebooks

We wrote this book using Jupyter Notebooks, so for nearly every chart, table, and calculation in this book, we'll be showing you all the exact code required to replicate it yourself. That's why very often in this book, you will see some code immediately followed by a table, a picture or just some text. If you go on the book website you will find all the code and you can try running and modifying every example yourself.

You just saw how a cell that outputs a table looks inside the book. Here is an example of a cell that outputs text:

In [3]:
```
1+1
```
Out[3]:
```
2
```

Jupyter will always print or show the result of the last line (if there is one). For instance, here is an example of a cell that outputs an image:

In [4]:
```
img = PILImage.create('images/chapter1_cat_example.jpg')
img.to_thumb(192)
```
Out[4]:

End sidebar

So, how do we know if this model is any good? In the last column of the table you can see the error rate, which is the proportion of images that were incorrectly identified. The error rate serves as our metric -- our measure of model quality, chosen to be intuitive and comprehensible. As you can see, the model is nearly perfect, even though the training time was only a few seconds (not including the one-time downloading of the dataset and the pretrained model). In fact, the accuracy you've achieved already is far better than anybody had ever achieved just 10 years ago!

Finally, let's check that this model actually works. Go and get a photo of a dog, or a cat; if you don't have one handy, just search Google images and download an image that you find there. Now execute the cell with `uploader` defined. It will output a button you can click, so you can select the image you want to classify.

In [5]:
```
#hide_output
uploader = widgets.FileUpload()
uploader
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
```

```
/tmp/ipykernel_26284/4087761153.py in <module>
      1 #hide_output
----> 2 uploader = widgets.FileUpload()
      3 uploader

NameError: name 'widgets' is not defined
```

An upload button

Now we can pass the uploaded file to the model. The notebook will tell you whether it thinks it is a dog, or a cat, and how confident it is. Make sure that it is a clear photo of a single dog or a cat, and not a line drawing, cartoon, or similar. Hopefully, you'll find that your model did a great job!

In [6]:

```
#hide
# For the book, we can't actually click an upload button, so we fake it
uploader = SimpleNamespace(data = ['images/chapter1_cat_example.jpg'])
```

In [7]:

```
img = PILImage.create(uploader.data[0])
is_cat,_,probs = learn.predict(img)
print(f"Is this a cat?: {is_cat}.")
print(f"Probability it's a cat: {probs[1].item():.6f}")
```

```
Is this a cat?: True.
Probability it's a cat: 0.980897
```

Congratulations on your first classifier!

But what does this mean? But what did we actually do? In order to explain this, let's zoom out again to take in the big picture.

What is machine learning?

Your classifier is a deep learning model. As was already mentioned, deep learning models use neural networks, which originally date from the 1950s and have become powerful very recently thanks to recent advancements.

Another key piece of context is that deep learning is just a modern area in the more general discipline of *machine learning*. To understand the essence of what you did when you trained your own classification model, you don't need to understand deep learning. It is enough to see how your model and your training process are examples of the concepts that apply to machine learning in general.

So in this section, we will describe what machine learning is. We will introduce the key concepts, and see how they can be traced back to the original essay that introduced the concept.

Machine learning is, like regular programming, a way to get computers to complete a specific task. But how would you use regular programming to do what we just did in the last section: recognize dogs vs cats in photos? We would have to write down for the computer the exact steps necessary to complete the task.

Normally, it's easy enough for us to write down the steps to complete a task when we're writing a program. We just think about the steps we'd take if we had to do the task by hand, and then we translate them into code. For instance, we can write a function that sorts a list. In general, we write a function that looks something like <> (where *inputs* might be an unsorted list, and *results* a sorted list).

In [8]:
```
#hide_input
#caption A traditional program
#id basic_program
#alt Pipeline inputs, program, results
gv('''program[shape=box3d width=1 height=0.7]
inputs->program->results''')
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
/tmp/ipykernel_26284/31908215.py in <module>
      3 #id basic_program
      4 #alt Pipeline inputs, program, results
----> 5 gv('''program[shape=box3d width=1 height=0.7]
      6 inputs->program->results''')

NameError: name 'gv' is not defined
```

But for recognizing objects in a photo that's a bit tricky; what *are* the steps we take exactly when we recognize an object in a picture? We really don't know, since it all happens in our brain without us being consciously aware of it!

Right back at the dawn of computing, in 1949, an IBM researcher named Arthur Samuel started working on a different way to get computers to complete tasks, which he called *machine learning*. In his classic 1962 essay *Artificial Intelligence: A Frontier of Automation*, he wrote:

> : *Programming a computer for such computations is, at best, a difficult task, not primarily because of any inherent complexity in the computer itself but, rather, because of the need to spell out every minute step of the process in the most exasperating detail. Computers, as any programmer will tell you, are giant morons, not giant brains.*

His basic idea was this: instead of telling the computer the exact steps required to solve a problem, instead, show it examples of the problem to solve, and let it figure out how to solve it itself. This turned out to be very effective: by 1961 his checkers playing program had learned so much that it beat the Connecticut state champion! Here's how he described his idea (from the same essay as above):

> : *Suppose we arrange for some automatic means of testing the effectiveness of any current weight assignment in terms of actual performance and provide a mechanism for altering the weight assignment so as to maximize the performance. We need not go into the details of such a procedure to see that it could be made entirely automatic and to see that a machine so programmed would "learn" from its experience.*

There a number of powerful concepts embedded in this short statement:

- the idea of a "weight assignment"
- the fact that every weight assignment has some "actual performance"
- the requirement that there is an "automatic means" of testing that performance,
- and last, that there is a "mechanism" (i.e., another automatic process) for improving the performance by changing the weight assignments.

Let us take these concepts one by one, in order to understand how they fit together in practice. First, we need to understand what Samuel means by a *weight assignment*.

Weights are just variables, and a weight assignment is a particular choice of values for those variables. The program's inputs are values that it processes in order to produce its results -- for instance, taking image pixels as inputs, and returning the classification "dog" as a result. But the program's weight assignments are other values which define how the program will operate.

Since they will affect the program they are in a sense another kind of input, so we will update our basic picture of <> and replace it with <> in order to take this into account:

In [9]:
```
#hide_input
#caption A program using weight assignment
#id weight_assignment
gv('''model[shape=box3d width=1 height=0.7]
inputs->model->results; weights->model''')
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
/tmp/ipykernel_26284/310237658.py in <module>
      2 #caption A program using weight assignment
      3 #id weight_assignment
----> 4 gv('''model[shape=box3d width=1 height=0.7]
      5 inputs->model->results; weights->model''')

NameError: name 'gv' is not defined
```

We've changed the name of our box from *program* to *model*. This is to follow modern terminology and to reflect that the *model* is a special kind of program: it's one that can do *many different things*, depending on the *weights*. It can be implemented in many different ways. For instance, in Samuel's checkers program, different values of the weights would result in different checkers-playing strategies.

(By the way, what Samuel called *weights* are most generally refered to as model *parameters* these days, in case you have encountered that term. The term *weights* is reserved for a particular type of model parameter.)

Next, he said we need an *automatic means of testing the effectiveness of any current weight assignment in terms of actual performance*. In the case of his checkers program, the "actual performance" of a model would be how well it plays. And you could automatically test the performance of two models by setting them to play against each other, and see which one usually wins.

Finally, he says we need *a mechanism for altering the weight assignment so as to maximize the performance*. For instance, we could look at the difference in weights between the winning model and the losing model, and adjust the weights a little further in the winning *direction*.

We can now see why he said that such a procedure *could be made entirely automatic and... a machine so programed would "learn" from its experience*. Learning would become entirely automatic when the adjustment of the weights was also automatic -- when instead of us improving a model by adjusting its weights manually, we relied on an automated mechanism that produced adjustments based on performance.

<> shows the full picture of Samuel's idea of training a machine learning model.

In [10]:

```
#hide_input
#caption Training a machine learning model
#id training_loop
#alt The basic training loop
gv('''ordering=in
model[shape=box3d width=1 height=0.7]
inputs->model->results; weights->model; results->performance
performance->weights[constraint=false label=update]''')
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
/tmp/ipykernel_26284/23477011.py in <module>
      3 #id training_loop
      4 #alt The basic training loop
----> 5 gv('''ordering=in
      6 model[shape=box3d width=1 height=0.7]
      7 inputs->model->results; weights->model; results->performance

NameError: name 'gv' is not defined
```

Notice the distinction between the model's *results* (e.g., the moves in a checkers game) and its *performance* (e.g., whether it wins the game, or how quickly it wins).

Also note that once the model is trained -- that is, once we've chosen our final, best, favorite weight assignment -- then we can think of the weights as being *part of the model*, since we're not varying them any more.

Therefore actually *using* a model after it's trained looks like <>.

In [11]:

```
#hide_input
#caption Using a trained model as a program
#id using_model
gv('''model[shape=box3d width=1 height=0.7]
inputs->model->results''')
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
/tmp/ipykernel_26284/1322908288.py in <module>
      2 #caption Using a trained model as a program
      3 #id using_model
----> 4 gv('''model[shape=box3d width=1 height=0.7]
      5 inputs->model->results''')
```

```
NameError: name 'gv' is not defined
```

This looks identical to our original diagram in <>, just with the word *program* replaced with *model*. This is an important insight: **a trained model can be treated just like a regular computer program**.

> jargon: Machine Learning: The training of programs developed by allowing a computer to learn from its experience, rather than through manually coding the individual steps.

What is a neural network?

It's not too hard to imagine what the model might look like for a checkers program. There might be a range of checkers strategies encoded, and some kind of search mechanism, and then the weights could vary how strategies are selected, what parts of the board are focused on during a search, and so forth. But it's not at all obvious what the model might look like for an image recognition program, or for understanding text, or for many other interesting problems we might imagine.

What we would like is some kind of function that is so flexible that it could be used to solve any given problem, just by varying its weights. Amazingly enough, this function actually exists! It's the neural network, which we already discussed. That is, if you regard a neural network as a mathematical function, it turns out to be a function which is extremely flexible depending on its weights. A mathematical proof called the *universal approximation theorem* shows that this function can solve any problem to any level of accuracy, in theory. The fact that neural networks are so flexible means that, in practice, they are often a suitable kind of model, and you can focus your effort on the process of training them, that is, of finding good weight assignments.

But what about that process? One could imagine that you might need to find a new "mechanism" for automatically updating weight for every problem. This would be laborious. What we'd like here as well is a completely general way to update the weights of a neural network, to make it improve at any given task. Conveniently, this also exists!

This is called *stochastic gradient descent* (SGD). We'll see how neural networks and SGD work in detail in <>, as well as explaining the universal approximation theorem. For now, however, we will instead use Samuel's own words: *We need not go into the details of such a procedure to see that it could be made entirely automatic and to see that a machine so programed would "learn" from its experience.*

> J: Don't worry, neither SGD nor neural nets are mathematically complex. Both SGD and neural nets nearly entirely rely on addition and multiplication to do their work (but they do a *lot* of addition and multiplication!) The main reaction we hear from students when they see the details is: "is that all it is?"

In other words, to recap, a neural network is a particular kind of machine learning model, which fits right in to Samuel's original conception. Neural networks are special because they are highly flexible, which means they can solve an unusually range of problems just by finding the right weights. This is powerful, because stochastic gradient descent provides us a way to find those weight values automatically.

Having zoomed out, let's now zoom back in and revisit our image classification problem into Samuel's framework.

Our inputs, those are the images. Our weights, those are the weights in the neural net. Our model is a neural net. Our results -- those are the values that are calculated by the neural net, like "dog" or "cat".

What about the next piece, an *automatic means of testing the effectiveness of any current weight assignment in terms of actual performance*? Determining "actual performance" is easy enough: we can simply define our model's performance as its accuracy at predicting the correct answers.

Putting this all together, and assuming that SGD is our mechanism for updating the weight assignments, we can see how our image classifier is a machine learning model, much like Samuel envisioned.

A bit of deep learning jargon

Samuel was working in the 1960s but terminology has changed. Here is the modern deep learning terminology for all the pieces we have discussed:

- The functional form of the *model* is called its *architecture* (but be careful--sometimes people use *model* as a synonym of *architecture*, so this can get confusing) ;
- The *weights* are called *parameters* ;
- The *predictions* are calculated from the *independent variables*, which is the *data* not including the *labels* ;
- The *results* of the model are called *predictions* ;
- The measure of *performance* is called the *loss*;
- The loss depends not only on the predictions, but also the correct *labels* (also known as *targets* or *dependent variable*), e.g. "dog" or "cat".

After making these changes, our diagram in <> looks like <>.

In [12]:
```
#hide_input
#caption Detailed training loop
#id detailed_loop
gv('''ordering=in
model[shape=box3d width=1 height=0.7 label=architecture]
inputs->model->predictions; parameters->model; labels->loss; predictions->loss
loss->parameters[constraint=false label=update]''')
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
/tmp/ipykernel_26284/4277915082.py in <module>
      2 #caption Detailed training loop
      3 #id detailed_loop
----> 4 gv('''ordering=in
      5 model[shape=box3d width=1 height=0.7 label=architecture]
      6 inputs->model->predictions; parameters->model; labels->loss; predictions
->loss

NameError: name 'gv' is not defined
```

Limitations inherent to machine learning

From this picture we can now see some fundamental things about training a deep learning model:

- A model cannot be created without data ;
- A model can only learn to operate on the patterns seen in the input data used to train it ;
- This learning approach only creates *predictions*, not recommended *actions* ;
- It's not enough to just have examples of input data; we need *labels* for that data too (e.g. pictures of dogs and cats aren't enough to train a model; we need a label for each one, saying which ones are dogs, and which are cats).

Generally speaking, we've seen that most organizations that think they don't have enough data, actually mean they don't have enough *labeled* data. If any organization is interested in doing something in practice with a model, then presumably they have some inputs they plan to run their model against. And presumably they've been doing that some other way for a while (e.g. manually, or with some heuristic program), so they have data from those processes! For instance, a radiology practice will almost certainly have an archive of medical scans (since they need to be able to check how their patients are progressing over time), but those scans may not have structured labels containing a list of diagnoses or interventions (since radiologists generally create free text natural language reports, not structured data). We'll be discussing labeling approaches a lot in this book, since it's such an important issue in practice.

Since these kinds of machine learning models can only make *predictions* (i.e. attempt to replicate labels), this can result in a significant gap between organizational goals and model capabilities. For instance, in this book you'll learn how to create a *recommendation system* that can predict what products a user might purchase. This is often used in e-commerce, such as to customize products shown on a home page, by showing the highest-ranked items. But such a model is generally created by looking at a user and their buying history (*inputs*) and what they went on to buy or look at (*labels*), which means that the model is likely to tell you about products they already have, or already know about, rather than new products that they are most likely to be interested in hearing about. That's very different to what, say, an expert at your local bookseller might do, where they ask questions to figure out your taste, and then tell you about authors or series that you've never heard of before.

Another critical insight comes from considering how a model interacts with its environment. For instance, this can create feedback loops, such as:

- A *predictive policing* model is created based on where arrests have been made in the past. In practice, this is not actually predicting crime, but rather predicting arrests, and is therefore partially simply reflecting biases in existing policing processes;

- Law enforcement officers then might use that model to decide where to focus their police activity, resulting in increased arrests in those areas;
- These additional arrests would then feed back to re-training future versions of the model;
- This is a *positive feedback loop*, where the more the model is used, the more biased the data becomes, making the model even more biased, and so forth.

This can also create problems in commercial products. For instance, a video recommendation system might be biased towards recommending content consumed by the biggest watchers of video (for instance, conspiracy theorists and extremists tend to watch more online video content than average), resulting in those users increasing their video consumption, resulting in more of those kinds of videos being recommended...

Now that we have seen the base of the theory, let's go back to our code example and see in detail how the code corresponds to the process we just described.

How our image recognizer works

Let's see just how our image recognizer code maps to these ideas. We'll put each line into a separate cell, and look at what each one is doing (we won't explain every detail of every parameter yet, but will give a description of the important bits; full details will come later in the book).

```
from fastai2.vision.all import *
```

The first line imports all of the fastai.vision library. This gives us all of the functions and classes we will need to create a wide variety of computer vision models.

> J: A lot of Python coders recommend avoiding importing a whole library like this (using the `import *` syntax), because in large software projects it can cause problems. However, for interactive work such as in a Jupyter notebook, it works great. The fastai library is specially designed to support this kind of interactive use, and it will only import the necessary pieces into your environment.

```
path = untar_data(URLs.PETS)/'images'
```

The second line downloads a standard dataset from the fast.ai datasets collection (if not previously downloaded) to your server, extracts it (if not previously extracted), and returns a `Path` object with the extracted location.

> S: Throughout my time studying fast.ai, and even still today, I've learned a lot about productive coding practices. The fastai library and fast.ai notebooks are full of great little tips that have helped make me a better programmer. For instance, notice that the fastai library doesn't just return a string containing the path to the dataset, but a Path object. This is a really useful class from the Python 3 standard library that makes accessing files and directories much easier. If you haven't come across it before, be sure to check out its documentation or a tutorial and try it out. Note that the book.fast.ai website contains links to recommended tutorials for

> each chapter. I'll keep letting you know about little coding tips I've found useful as we come across them.

```
def is_cat(x): return x[0].isupper()
dls = ImageDataLoaders.from_name_func(
    path, get_image_files(path), valid_pct=0.2, seed=42,
    label_func=is_cat, item_tfms=Resize(224))
```

The fourth line tells fastai what kind of dataset we have, and how it is structured. There are various different classes for different kinds of deep learning dataset and problem--here we're using `ImageDataLoaders` . The first part of the class name will generally be the type of data you have, such as image, or text. The second part will generally be the type of problem you are solving, such as classification, or regression.

The other important piece of information that we have to tell fastai is how to get the labels from the dataset. Computer vision datasets are normally structured in such a way that the label for an image is part of the file name, or path, most commonly the parent folder name. Fastai comes with a number of standardized labelling methods, and ways to write your own. Here we define a function on the third line: `is_cat` which labels cats based on a filename rule provided by the dataset creators.

Finally, we define the `Transform` s that we need. A `Transform` contains code that is applied automatically during training; fastai includes many pre-defined `Transform` s, and adding new ones is as simple as creating a Python function. There are two kinds: `item_tfms` are applied to each item (in this case, each item is resized to a 224 pixel square); `batch_tfms` are applied to a *batch* of items at a time using the GPU, so they're particularly fast (we'll see many examples of these throughout this book).

Why 224 pixels? This is the standard size for historical reasons (old pretrained models require this size exactly), but you can pass pretty much anything. If you increase the size, you'll often get a model with better results (since it will be able to focus on more details) but at the price of speed and memory consumption; or vice versa if you decrease the size.

> Note: *classification* and *regression* have very specific meanings in machine learning. These are the two main types of model that we will be investigating in this book. A classification model is one which attempts to predict a class, or category. That is, predicting from a number of discrete possibilities, such as "dog" or "cat". A regression model is one which attempts to predict one or more numeric quantities, such as temperature, or a location. Sometimes people use the word *regression* as a shortcut to a particular kind of model called a *linear regression model*; this is a bad practice, and we won't be using that terminology in this book!

The pets dataset contains 7390 pictures of dogs and cats, consisting of 37 different breeds. Each image is labeled using its filename, for instance the file `great_pyrenees_173.jpg` is the 173rd example of an image of a great pyrenees breed dog in the dataset. The filenames start with an uppercase letter if the image is a cat, and a lowercase letter otherwise. We have to tell fastai how to get labels from the filenames, which we do by calling `from_name_func` (which means that filenames can be extracted using a function applied to the file name), and passing `x[0].isupper()`, which evaluates to `True` if the first letter is uppercase (i.e. it's a cat).

The most important parameter to mention here is `valid_pct=0.2`. This tells fastai to hold out 20% of the data and *not use it for training the model at all*. This 20% of the data is called the *validation set*; the remaining 80% is called the *training set*. The validation set is used to measure the accuracy of the model. By default, the 20% that is held out is selected randomly. The parameter `seed=42` sets the *random seed* to the same value every time we run this code, which means we get the same validation set every time we run this code--that way, if you change your model and re-train it, you know that changes are due to your model, not due to having a different random validation set.

fastai will *always* show you your model's accuracy using *only* the validation set, *never* the training set. This is absolutely critical, because if you train a large enough model for a long enough time, it will eventually learn to *memorize* the label of every item in your dataset! This is not actually a useful model, because what we care about is how well our model works on *previously unseen images*. That is always our goal when creating a model: to be useful on data that the model only sees in the future, after it has been trained.

Even when your model has not fully memorized all your data, earlier on in training it may have memorized certain parts of it. As a result, the longer you train for, the better your accuracy will get on the training set; and the validation set accuracy will also improve for a while, but eventually it will start getting worse, as the model starts to memorize the training set, rather than finding generalizable underlying patterns in the data. When this happens, we say that the model is *over-fitting*.

<> shows what happens when you overfit, using a simplified example where we have just one parameter, and some randomly generated data based on the function `x**2`; as you see, although the predictions in the overfit model are accurate for data near the observed data, they are way off when outside of that range.

Example of overfitting

Overfitting is the single most important and challenging issue when training for all machine learning practitioners, and all algorithms. As we will see, it is very easy to create a model that does a great job at making predictions on the exact data which it has been trained on, but it is much harder to make predictions on data that it has never seen before. And of course, this is the data that will actually matter in practice. For instance, if you create a hand-written digit classifier (as we will very soon!) and use it to recognise numbers written on cheques, then you are never going to see any of the numbers that the model was trained on -- every cheque will have slightly different variations of writing to deal with. We will learn many methods to avoid overfitting in this book. However, you should only use those methods after you have confirmed that overfitting is actually occurring (i.e. you have actually observed the validation accuracy getting worse during training). We often see practitioners using over-fitting avoidance techniques even when they have enough data that they didn't need to do so, ending up with a model that could be less accurate than what they could have achieved.

> important: When you train a model, you must **always** have both a training set, and a validation set, and must measure the accuracy of your model only on the validation set. If you train for too long, with not enough data, you will see the accuracy of your model start to get worse; this is called **over-fitting**. fastai defaults `valid_pct to 0.2`, so even if you forget, fastai will create a validation set for you!

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

The fifth line tells fastai to create a *convolutional neural network* (CNN), and selects what *architecture* to use (i.e. what kind of model to create), what data we want to train it on, and what *metric* to use.

Why a CNN? A CNN is the current state of the art approach to creating computer vision models. We'll be learning all about how they work in this book. Their structure is inspired by how the human vision system works.

There are many different architectures in fastai, which we will be learning about in this book, as well as discussing how to create your own. Most of the time, however, picking an architecture isn't a very important part of the deep learning process. It's something that academics love to talk about, but in practice it is unlikely to be something you need to spend much time on. There are some standard architectures that work most of the time, and in this case we're using one called *ResNet* that we'll be learning a lot about during the book; it is both fast and accurate for many datasets and problems. The "34" in `resnet34` refers to the number of layers in this variant of the architecture (other options are "18", "50", "101", and "152"). Models using architectures with more layers take longer to train, and are more prone to overfitting (i.e. you can't train them for as many epochs before the accuracy on the validation set starts getting worse). On the other hand, when using more data, they can be quite a bit more accurate.

What is a metric? A *metric* is a function that measures quality of the model's predictions using the validation set, and will be printed at the end of each *epoch*. In this case, we're using `error_rate`, which is a function provided by fastai which does just what it says: tells you what percentage of images in the validation set are being classified incorrectly. Another common metric for classification is `accuracy` (which is just `1.0 - error_rate`). fastai provides many more, which will be discussed throughout this book.

The concept of a metric may remind you of loss, but there is an important distinction. The entire purpose of loss was to define a "measure of performance" that the training system could use to update weights automatically. In other words, a good choice for loss is a choice that is easy for stochastic gradient descent to use. But a metric is defined for human consumption. So a good metric is one that is easy for you to understand, and that hews as closely as possible to what you want the model to do. At times, you might decide that the loss function is a suitable metric, but that is not necessarily the case.

`cnn_learner` also has a parameter `pretrained`, which defaults to `True` (so it's used in this case), which sets the weights in your model to values that have already been trained by experts to recognize a thousand different categories across 1.3 million photos (using the famous *ImageNet* dataset). A model that has weights that have already been trained on some other dataset is called a *pretrained model*. You should nearly always use a pretrained model, because it means that your model, before you've even shown it any of your data, is already very capable. And, as you'll see, in a deep learning model many of these capabilities are things you'll need, almost regardless of the details of your project. For instance, parts of pretrained models will handle edge-, gradient-, and color-detection, which are needed for many tasks.

When using a pretrained model, `cnn_learner` will remove the last layer, since that is always specifically customized to the original training task (i.e. ImageNet dataset classification), and replace it with one or more new layers with randomized weights, of an appropriate size for the dataset you are working with. This last part of the model is known as the `head`.

Using pretrained models is the *most* important method we have to allow us to train more accurate models, more quickly, with less data, and less time and money. You might think that would mean that using pretrained models would be the most studied area in academic deep learning... but you'd be very, very wrong! The importance of pretrained models is generally not recognized or discussed in most courses, books, or software library features, and is rarely considered in academic papers. As we write this at the start of 2020, things are just starting to change, but it's likely to take a while. So be careful: most people you speak to will probably greatly underestimate what you can do in deep learning with few resources, because they probably won't deeply understand how to use pretrained models.

Using a pretrained model for a task different to what it was originally trained for is known as *transfer learning*. Unfortunately, because transfer learning is so under-studied, few domains have pretrained models available. For instance, there are currently few pretrained models available in medicine, making transfer learning challenging to use in that domain. In addition, it is not yet well understood how to use transfer learning for tasks such as time series analysis.

> jargon: Transfer learning: Using a pretrained model for a task different to what it was originally trained for.

```
learn.fine_tune(1)
```

The sixth line tells fastai how to *fit* the model. As we've discussed, the architecture only describes a *template* for a mathematical function; but it doesn't actually do anything until we provide values for the millions of parameters it contains.

This is the key to deep learning — how to fit the parameters of a model to get it to solve your problem. In order to fit a model, we have to provide at least one piece of information: how many times to look at each image (known as number of *epochs*). The number of epochs you select will largely depend on how much time you have available, and how long you find it takes in practice to fit your model. If you select a number that is too small, you can always train for more epochs later.

But why is the method called `fine_tune`, and not `fit`? fastai actually *does* have a method called `fit`, which does indeed fit a model (i.e. look at images in the training set multiple times, each time updating the *parameters* to make the predictions closer and closer to the *target labels*). But in this case, we've started with a pretrained model, and we don't want to throw away all those capabilities that it already has. As we'll learn in this book, there are some important tricks to adapt a pretrained model for a new dataset -- a process called *fine-tuning*.

> jargon: Fine tuning: A transfer learning technique where the weights of a pretrained model are updated by training for additional epochs using a different task to that used for pretraining.

When you use the `fine_tune` method, fastai will use these tricks for you. There are a few parameters you can set (which we'll discuss later), but in the default form shown here, it does two steps:

1. Use one *epoch* to fit just those parts of the model necessary to get the new random *head* to work correctly with your dataset
2. Use the number of epochs requested when calling the method to fit the entire model, updating the weights of the later layers (especially the head) faster than the earlier layers (which, as we'll see, generally don't require many changes from the pretrained weights)

The *head* of a model is the part that is newly added to be specific to the new dataset. An *epoch* is one complete pass through the dataset. After calling `fit`, the results after each epoch are printed, showing the epoch number, the training and validation set losses (the "measure of performance" used for training the model), and any *metrics* you've requested (error rate, in this case).

So, with all this code our model learned to recognize cats and dogs just from labeled examples. But how did it do it?

What our image recognizer learned

At this stage we have an image recogniser that is working very well, but we have no idea what it is actually doing! Although many people complain that deep learning results in impenetrable "black box" models (that is, something that gives predictions but that no one can understand), this really couldn't be further from the truth. There is a vast body of research showing how to deeply inspect deep learning models, and get rich insights from them. Having said that, all kinds of machine learning models (including deep learning, and traditional statistical models) can be challenging to fully understand, especially when considering how they will behave when coming across data that is very different to the data used to train them. We'll be discussing this issue throughout this book.

In 2013 a PhD student, Matt Zeiler, and his supervisor, Rob Fergus, published the paper Visualizing and Understanding Convolutional Networks, which showed how to visualise the neural network weights learned in each layer of a model. They carefully analysed the model that won the 2012 ImageNet competition, and used this analysis to greatly improve the model, such that they were able to go on to win the 2013 competition! <> is the picture that they published of the first layers' weights.

Activations of early layers of a CNN

This picture requires some explanation. For each layer, the image part with the light grey background shows the reconstructed weights pictures, and the other section shows the parts of the training images which most strongly matched each set of weights. For layer 1, what we can see is that the model has discovered weights which represent diagonal, horizontal, and vertical edges, as well as various different gradients. (Note that for each layer only a subset of the features are shown; in practice there are thousands across all of the layers.) These are the basic building blocks that it has created automatically for computer vision. They have been widely analysed by neuroscientists and computer vision researchers, and it turns out that these learned building blocks are very similar to the basic visual machinery in the human eye, as well as the handcrafted computer vision features that were developed prior to the days of deep learning. The next layer is represented in <>.

Activations of early layers of a CNN

For layer 2, there are nine examples of weight reconstructions for each of the features found by the model. We can see that the model has learned to create feature detectors that look for corners, repeating lines, circles, and other simple patterns. These are built from the basic building blocks developed in the first layer. For each of these, the right-hand side of the picture shows small patches from actual images which these features most closely match. For instance, the particular pattern in row 2 column 1 matches the gradients and textures associated with sunsets.

<> shows the image from the paper showing the results of reconstructing the features of layer 3.

Activations of medium layers of a CNN

As you can see by looking at the right-hand side of this picture, the features are now able to identify and match with higher levels semantic components, such as car wheels, text, and flower petals. Using these components, layers four and five can identify even higher-level concepts, as shown in <>.

Activations of end layers of a CNN

This article was studying an older model called `AlexNet` that only contained five layers. Networks developed since then can have hundreds of layers--so you can imagine how rich the features developed by these models can be!

When we fine-tuned our pretrained model earlier, we adapted what those last layers focus on (flowers, humans, animals) to specialize on the cats versus dogs problem. More generally, we could specialize such a pretrained problem on many different tasks. Let's have a look at some examples.

Image recognizers can tackle non-image tasks

An image recogniser can, as its name suggests, only recognise images. But a lot of things can be represented as images, which means that an image recogniser can learn to complete many tasks.

For instance, a sound can be converted to a spectrogram, which is a chart that shows the amount of each frequency at each time in an audio file. Fast.ai student Ethan Sutin used this approach to easily beat the published accuracy on environmental sound detection using a dataset of 8732 urban sounds. fastai's `show_batch` clearly shows how each different sound has a quite distinctive spectrogram, as you can see in <>.

show_batch with spectrograms of sounds

Time series can be easily converted into an image by simply plotting the time series in a graph. However, it is often a good idea to try to represent your data in a way that makes it as easy as possible to pull out the most important components. In a time-series, things like seasonality and anomalies are most likely to be of interest. There are various transformations available for time series data; for instance, fast.ai student Ignacio Oguiza created images from a time series data set for olive oil classification. He used a technique called Gramian Angular Field (GAF), and you can see the result in <>. He then fed those images to an image classification model just like the one you see in this chapter. His results, despite having only 30 training set images, were well over 90% accurate, and close to the state-of-the-art.

Converting a time series into an image

Another interesting fast.ai student project example comes from Gleb Esman. He was working on fraud detection at Splunk, and was working with a dataset of users' mouse movements and mouse clicks. He turned these into pictures by drawing an image where the position, speed and acceleration of the mouse was displayed using coloured lines, and the clicks were displayed using small coloured circles as shown in <>. He then fed this into an image recognition model just like the one we've shown in this chapter, and it worked so well that had led to a patent for this approach to fraud analytics!

Converting computer mouse behavior to an image

Another examples comes from the paper Malware Classification with Deep Convolutional Neural Networks which explains that "the malware binary file is divided into 8-bit sequences which are then converted to equivalent decimal values. This decimal vector is reshaped and gray-scale image is generated that represent the malware sample", like in <>.

Malware classification process

They then show "pictures" generated through this process of malware in different categories, as shown in <>.

Malware examples

As you can see, the different types of malware look very distinctive to the human eye. The model they train based on this image representation was more accurate at malware classification than any previous approach shown in the academic literature. This suggests a good rule of thumb for converting a dataset into an image representation: if the human eye can recognize categories from the images, then a deep learning model should be able to do so too.

In general, you'll find that a small number of general approaches in deep learning can go a long way, if you're a bit creative in how you represent your data! You shouldn't think of approaches like the above as "hacky workarounds", since actually they often (as here) beat previously state of the art results. These really are the right way to think about these problem domains.

Jargon recap

We just covered a lot of information so let's recap briefly. <> provides a handy list.

```
asciidoc
[[dljargon]]
.Deep learning vocabulary
[options="header"]
|=====
| Term | Meaning
|**label** | The data that we're trying to predict, such as "dog"
or "cat"
|**architecture** | The _template_ of the model that we're trying
to fit; the actual mathematical function that we're passing the
input data and parameters to
|**model** | the combination of the architecture with a particular
set of parameters
|**parameters** | the values in the model that change what task it
can do, and are updated through model training
|**fit** | Update the parameters of the model such that the
predictions of the model using the input data match the target
labels
|**train** | A synonym for _fit_
|**pretrained model** | A model that has already been trained,
generally using a large dataset, and will be fine-tuned
|**fine tune** | Update a pretrained model for a different task
|**epoch** | One complete pass through the input data
|**loss** | A measure of how good the model is, chosen to drive
training via SGD
|**metric** | A measurement of how good the model is, using the
validation set, chosen for human consumption
|**validation set** | A set of data held out from training, used
only for measuring how good the model is
|**training set** | The data used for fitting the model; does not
include any data from the validation set
|**overfitting** | Training a model in such a way that it
_remembers_ specific features of the input data, rather than
generalizing well to data not seen during training
|**CNN** | Convolutional neural network; a type of neural network
that works particularly well for computer vision tasks
|=====
```

With this vocabulary in hand, we are now in a position to bring together all the key concepts so far. Take a moment to review those definitions and read the following summary. If you can follow the explanation, then you have laid down the basic coordinates for understanding many discussions to come.

Machine learning is a discipline where we define a program not by writing it entirely ourselves, but by learning from data. *Deep learning* is a specialty within machine learning which uses *neural networks* using multiple *layers*. *Image classification* is a representative example (also known as *image recognition*). We start with *labeled data*, that is, a set of images where we have assigned a *label* to each image indicating what it represents. Our goal is to produce a program, called a *model*, which, given a new image, will make an accurate *prediction* regarding what that new image represents.

Every model starts with a choice of *architecture*, a general template for how that kind of model works internally. The process of *training* (or *fitting*) the model is the process of finding a set of *parameter values* (or *weights*) which specializes that general architecture into a model that works well for our particular kind of data. In order to define how well a model does on a single prediction, we need to define a *loss function*, which defines how we score a prediction as good or bad, in order to support training.

In order to make the training process go faster, we might start with a *pretrained model*, a model which has already been trained on someone else's data. We then adapt it to our data by training it a bit more on our data, a process called *fine tuning*.

When we train a model, a key concern is to ensure that our model *generalizes* -- that is, that it learns general lessons from our data which also apply to new items it will encounter, so that it can make good predictions on those items. The risk is that if we train our model badly, instead of learning general lessons it effectively memorizes what it has already seen, and then it will make poor predictions about new images. Such a failure is called *overfitting*. In order to avoid this, we always divide our data into two parts, the *training set* and the *validation set*. We train the model by showing it only the *training set* and then we evaluate how well the model is doing by seeing how well it predicts on items from the *validation set* . In this way, we check if the lessons the model learns from the training set are lessons that generalize to the validation set. In order for a person to assess how well the model is doing on the validation set overall, we define a *metric* . During the training process, when the model has seen every item in the training set, we call that an *epoch*.

All these concepts apply to machine learning in general. That is, they apply to all sorts of schemes for defining a model by training it with data. What makes deep learning distinctive is a particular class of architectures, the architectures based on *neural networks*. In particular, tasks like image classification rely heavily on *convolutional neural networks*, which we will discuss shortly.

Deep learning is not just for image classification

Deep learning's effectiveness for classifying images has been widely discussed in recent years, even showing *super-human* results on complex tasks like recognizing malignant tumours in CT scans. But it can do a lot more than this, as we will show here.

For instance, let's talk about something that is critically important for autonomous vehicles: localising objects in a picture. If a self-driving car doesn't know where a pedestrian is, then it doesn't know how to avoid one! Creating a model which can recognize the content of every individual pixel in an image is called *segmentation*. Here is how we can train a segmentation model using fastai, using a subset of the *Camvid* dataset from the paper Semantic Object Classes in Video: A High-Definition Ground Truth Database:

In [13]:
```
path = untar_data(URLs.CAMVID_TINY)
dls = SegmentationDataLoaders.from_label_func(
    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)
)

learn = unet_learner(dls, resnet34)
learn.fine_tune(8)
```

```
100.18% [2318336/2314212 00:00<00:00]
```

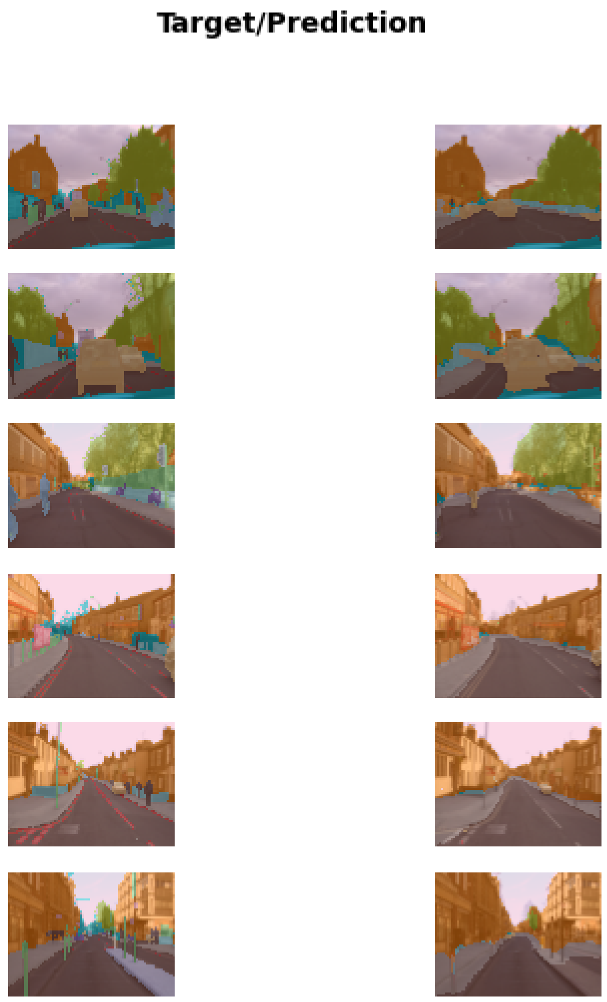
epoch	train_loss	valid_loss	time
0	2.667020	2.518456	00:02

epoch	train_loss	valid_loss	time
0	1.847760	1.492310	00:01
1	1.589073	1.247304	00:01
2	1.407862	1.123045	00:01
3	1.259581	0.932523	00:01
4	1.130517	0.973192	00:01
5	1.027547	0.799841	00:01
6	0.938767	0.771569	00:01
7	0.868429	0.764602	00:01

We are not even going to walk through this code line by line, because it is nearly identical to our previous example! (Although we will, of course, be doing a deep dive into segmentation models in <>, along with all of the other models that we are briefly introducing in this chapter, and many, many more.)

We can visualise how well it achieved its task, by asking the model to color code each pixel of an image. As you can see, it nearly perfectly classifies every pixel in every object; for instance, notice that all of the cars are overlaid with the same colour, and all of the trees are overlaid with the same color (in each pair of images, the left hand image is the ground truth labels, the right hand is the predictions from the model):

In [14]:
```
learn.show_results(max_n=6, figsize=(7,8))
```



One other area where deep learning has dramatically improved in the last couple of years is natural language processing (NLP). Computers can now generate text, translate automatically from one language to another, analyze comments, label words in sentences, and much more. Here is all of the code necessary to train a model which can classify the sentiment of a movie review better than anything that existed in the world just five years ago:

In [15]:
```
from fastai.text.all import *

dls = TextDataLoaders.from_folder(untar_data(URLs.IMDB), valid='test')
learn = text_classifier_learner(dls, AWD_LSTM, drop_mult=0.5, metrics=accuracy)
learn.fine_tune(4, 1e-2)
```

epoch	train_loss	valid_loss	accuracy	time
0	0.459859	0.400997	0.818640	00:43

epoch	train_loss	valid_loss	accuracy	time
0	0.307206	0.240368	0.901800	01:18
1	0.240177	0.205541	0.920160	01:18
2	0.181079	0.199449	0.928280	01:18
3	0.157887	0.196001	0.927520	01:18

This model is using the IMDb dataset from the paper Learning Word Vectors for Sentiment Analysis). It works well with movie reviews of many thousands of words. But let's test it out on a very short one, to see it does its thing:

In [16]:
```
learn.predict("I really liked that movie!")
```

Out[16]:
```
('pos', tensor(1), tensor([5.8537e-05, 9.9994e-01]))
```

Here we can see the model has considered the review to be positive. The second part of the result is the index of "pos" in our data vocabulary and the last part is the probabilities attributed to each class (99.6% for "pos" and 0.4% for "neg").

Now it's your turn! Write your own mini movie review, or copy one from the Internet, and we can see what this model thinks about it.

Sidebar: The order matter

In a Jupyter notebook, the order in which you execute each cell is very important. It's not like Excel, where everything gets updated as soon as you type something anywhere, but it has an inner state that gets updated each time you execute a cell. For instance, when you run the first cell of the notebook (with the CLICK ME comment), you create an object `learn` that contains a model and data for an image classification problem. If we were to run the cell right above (the one that predicts if a review is good or not) straight after, we would get an error as this `learn` object does not contain a text classification model. This cell needs to be run after the one containing

```
from fastai2.text.all import *

dls = TextDataLoaders.from_folder(untar_data(URLs.IMDB), valid='test')
learn = text_classifier_learner(dls, AWD_LSTM, drop_mult=0.5,
                                metrics=accuracy)
learn.fine_tune(4, 1e-2)
```

The outputs themselves can be deceiving: they have the results of the last time the cell was executed, but if you change the code inside a cell without executing it, the old (misleading) results will remain.

Except when we mention it explicitly, the notebooks provided on the book website are meant to be run in order, from top to bottom. In general, when experimenting, you will find yourself executing cells in any order to go fast (which is a super neat feature of Jupyter Notebooks) but once you have explored and arrive at the final version of your code, make sure you can run the cells of your notebooks in order (your future self won't necessarily remember the convoluted path you took otherwise!).

In command mode, pressing `0` twice will restart the *kernel* (which is the engine powering your notebook). This will wipe your state clean and make it as if you had just started in the notebook. Click on the "Cell" menu and then on "Run All Above" to run all cells above the point where you are. We have found this to be very useful when developing the fastai library.

End sidebar

If you ever have any questions about a fastai method, you should use the function `doc` :

```
doc(learn.predict)
```

This will make a small window pop with content like this:

A brief one-line explanation is provided by `doc` . The *show in docs* link is where you'll find all the details in the full documentation, and lots of examples. Also, most of fastai's methods are just a handful of lines, so you can click the *source* link to see exactly what's going on behind the scenes.

Let's move on to something much less sexy, but perhaps significantly more widely commercially useful: building models from plain *tabular* data. It turns out that looks very similar too. Here is the code necessary to train a model which will predict whether a person is a high-income earner, based on their socio-economic background:

> jargon: Tabular: Data that is in the form of a table, such as from a spreadsheet, database, or CSV file. A tabular model is a model which tries to predict one column of a table based on information in other columns of a table.

In [17]:

```
from fastai.tabular.all import *
path = untar_data(URLs.ADULT_SAMPLE)

dls = TabularDataLoaders.from_csv(path/'adult.csv', path, y_names="salary",
    cat_names = ['workclass', 'education', 'marital-status', 'occupation',
                 'relationship', 'race'],
    cont_names = ['age', 'fnlwgt', 'education-num'],
    procs = [Categorify, FillMissing, Normalize])

learn = tabular_learner(dls, metrics=accuracy)
```

```
100.69% [974848/968212 00:00<00:00]
```

As you see, we had to tell fastai which columns are *categorical* (that is, they contain values that are one of a discrete set of choices, such as `occupation`), versus *continuous* (that is, they contain a number that represents a quantity, such as `age`).

There is no pretrained model available for this task (in general, pretrained models are not widely available for any tabular modeling tasks, although some organizations have created them for internal use), so we don't use `fine_tune` in this case, but instead `fit_one_cycle` , the most commonly used method for training fastai models *from scratch* (i.e. without transfer learning):

In [18]:

```
learn.fit_one_cycle(3)
```

epoch	train_loss	valid_loss	accuracy	time

epoch	train_loss	valid_loss	accuracy	time
0	0.381104	0.357087	0.832002	00:04
1	0.358382	0.347734	0.835688	00:04
2	0.347142	0.344169	0.840448	00:04

This model is using the *adult* dataset, from the paper Scaling Up the Accuracy of Naive-Bayes Classifiers: a Decision-Tree Hybrid, which contains some data regarding individuals (like their education, marital status, race, sex, etc.) and whether or not they have an annual income greater than $50k. The model is over 80\% accurate, and took around 30 seconds to train.

Let's look at one more. Recommendation systems are very important, particularly in e-commerce. Companies like Amazon and Netflix try hard to recommend products or movies which you might like. Here's how to train a model which will predict which people might like which movie, based on their previous viewing habits, using the MovieLens dataset:

In [19]:
```
from fastai.collab import *
path = untar_data(URLs.ML_SAMPLE)
dls = CollabDataLoaders.from_csv(path/'ratings.csv')
learn = collab_learner(dls, y_range=(0.5,5.5))
learn.fine_tune(10)
```

```
110.72% [57344/51790 00:00<00:00]
```

epoch	train_loss	valid_loss	time
0	1.525607	1.433031	00:00

epoch	train_loss	valid_loss	time
0	1.365996	1.371801	00:00
1	1.251946	1.184150	00:00
2	1.004050	0.875378	00:00
3	0.789740	0.738321	00:00
4	0.683325	0.705283	00:00
5	0.641180	0.694728	00:00
6	0.624207	0.690513	00:00
7	0.602270	0.689152	00:00
8	0.601474	0.688069	00:00
9	0.599375	0.687988	00:00

This model is predicting movie ratings on a scale of 0.5 to 5.0 to within around 0.6 average error. Since we're predicting a continuous number, rather than a category, we have to tell fastai what range our target has, using the `y_range` parameter.

Although we're not actually using a pretrained model (for the same reason that we didn't for the tabular model), this example shows that fastai lets us use `fine_tune` even in this case (we'll learn how and why this works later in <>). Sometimes it's best to experiment with `fine_tune` versus `fit_one_cycle` to see which works best for your dataset.

We can use the same `show_results` call we saw earlier to view a few examples of user and movie IDs, actual ratings, and predictions:

In [20]:
```
learn.show_results()
```

	userId	movieId	rating	rating_pred
0	45.0	56.0	5.0	3.295857
1	63.0	26.0	3.0	2.997506
2	87.0	60.0	4.0	4.435205
3	98.0	51.0	5.0	4.585992
4	80.0	62.0	3.5	3.864863
5	52.0	55.0	4.0	4.455365
6	90.0	30.0	2.0	3.478552
7	18.0	90.0	3.0	3.713112
8	29.0	46.0	3.0	3.267631

Sidebar: Datasets: food for models

You’ve already seen in this section quite a few models, each one trained using a different dataset, to do a different task. In machine learning and deep learning, we can’t do anything without data. So, the people that create datasets for us to train our models are the (often under-appreciated) heroes. Some of the most useful and important datasets are those that become important *academic baselines*; that is, datasets that are widely studied by researchers and used to compare algorithmic changes. Some of these become household names (at least, among households that train models!), such as MNIST, CIFAR 10, and ImageNet.

The datasets used in this book have been selected because they provide great examples of the kind of data that you are likely to encounter, and the academic literature has many examples of model results using these datasets which you can compare your work to.

Most datasets used in this book took the creators a lot of work to build. For instance, later in the book we’ll be showing you how to create a model that can translate between French and English. The key input to this is a French/English parallel text corpus prepared back in 2009 by Professor Chris Callison-Burch of the University of Pennsylvania. This dataset contains over 20 million sentence pairs in French and English. He built the dataset in a really clever way: by crawling millions of Canadian web pages (which are often multi-lingual) and then using a set of simple heuristics to transform French URLs onto English URLs.

As you look at datasets throughout this book, think about where they might have come from, and how they might have been curated. Then, think about what kinds of interesting dataset you could create for your own projects. (We'll even take you step by step through the process of creating your own image dataset soon.)

fast.ai has spent a lot of time creating cutdown versions of popular datasets that are specially designed to support rapid prototyping and experimentation, and to be easier to learn with. In this book we will often start by using one of the cutdown versions, and we later on scale up to the full-size version (just as we're doing in this chapter!) In fact, this is how the world's top practitioners do their modelling projects in practice; they do most of their experimentation and prototyping with subsets of their data, and only use the full dataset when they have a good understanding of what they have to do.

End sidebar

Each of the models we trained showed a training and validation loss. A good validation set is one of the most important pieces of your training, let's see why and learn how to create one.

Validation sets and test sets

As we've discussed, the goal of a model is to make predictions about data. But the model training process is fundamentally dumb. If we trained a model with all our data, and then evaluated the model using that same data, we would not be able to tell how well our model can perform on data it hasn't seen. Without this very valuable piece of information to guide us in training our model, there is a very good chance it would become good at making predictions about that data but would perform poorly on new data.

It is in order to avoid this that our first step was to split our dataset into two sets, the *training set* (which our model sees in training) and the *validation set*, also known as the *development set* (which is used only for evaluation). This lets us test that the model learns lessons from the training data which generalize to new data, the validation data.

One way to understand this situation is that, in a sense, we don't want our model to get good results by "cheating". If it predicts well on a data item, that should be because it has learned principles that govern that kind of item, and not because the model has been shaped by *actually having seeing that particular item*.

Splitting off our validation data means our model never sees it in training, and so is completely untainted by it, and is not cheating in any way. Right?

In fact, not necessarily. The situation is more subtle. The subtlety is that in realistic scenarios we rarely build a model just by training its weight parameters once. Instead we are likely to explore many versions of a model through various modelling choices regarding network architecture, learning rates, data augmentation strategies, and other factors we will discuss in upcoming chapters. Many of these choices can be described as choices of *hyperparameters*. The word reflects that they are parameters about parameters, since they are the higher-level choices that govern the meaning of the weight parameters.

The problem is that, even though the ordinary training process is only looking at predictions on the training data when it learns values for the weight parameters, the same is not true about us. We, as modellers, are evaluating the model by looking at predictions on the validation data, when we decide to explore new hyperparameter values! So subsequent versions of the model are, indirectly, shaped by having seen the validation data. Just as the automatic training process is in danger of overfitting the training data, we are in danger of overfitting the validation data, by human trial and error and exploration.

The solution to this conundrum is to introduce another level of even more highly reserved data, the "test set". Just as we hold back the validation data from the training process, we must hold back the test set data even from ourselves. It cannot be used to improve the model; it can only be used to evaluate the model at the very end of our efforts. In effect, we define a hierarchy of cuts of our data, based on how fully we want to hide it from training and modelling processes -- training data is fully exposed, the validation data is less exposed, and test data is totally hidden. This hierarchy parallels the different kinds of modelling and evaluation processes themselves -- the automatic training process with back propagation, the more manual process of trying different hyper-parameters between training sessions, and the assessment of our final result.

The test and validation sets should have enough data to ensure that you get a good estimate of your accuracy. If you're creating a cat detector, for instance, you generally want at least 30 cats in your validation set. That means that if you have a dataset with thousands of items, using the default 20% validation set size may be larger than you need. On the other hand, if you have lots of data, using some of it for the validation probably doesn't have any downsides.

Having two levels of "reserved data", a validation set and a test set -- with one level representing data which you are virtually hiding from yourself -- may seem a bit extreme. But the reason it is often necessary is because models tend to gravitate toward the simplest way to do good predictions (memorization), and we as fallible humans tend to gravitate toward fooling ourselves about how well our models are performing. The discipline of the test set helps us keep ourselves intellectually honest. That doesn't mean we *always* need a separate test set--if you have very little data, you may need to just have a validation set--but generally it's best to use one if at all possible.

This same discipline can be critical if you intend to hire a third-party to perform modelling work on your behalf. A third-party might not understand your requirements accurately, or their incentives might even encourage them to misunderstand them. But a good test set can greatly mitigate these risks and let you evaluate if their work solves your actual problem.

To put it bluntly, if you're a senior decision maker in your organization (or you're advising senior decision makers) then the most important takeaway is this: if you ensure that you really understand what test and validation sets are, and why they're important, then you'll avoid the single biggest source of failures we've seen when organizations decide to use AI. For instance, if you're considering bringing in an external vendor or service, make sure that you hold out some test data that the vendor *never gets to see*. Then *you* check their model on your test data, using a metric that *you* choose based on what actually matters to you in practice, and *you* decide what level of performance is adequate. (It's also a good idea for you to try out some simple baseline yourself, so you know what a really simple model can achieve. Often it'll turn out that your simple model can be just as good as an external "expert"!)

Use judgment in defining test sets

To do a good job of defining a validation set (and possibly a test set), you will sometimes want to do more than just randomly grab a fraction of your original dataset. Remember: a key property of the validation and test sets is that they must be representative of the new data you will see in the future. This may sound like an impossible order! By definition, you haven't seen this data yet. But you usually still do know some things.

It's instructive to look at a few example cases. Many of these examples come from predictive modeling competitions on the *Kaggle* platform, which is a good representation of problems and methods you would see in practice.

One case might be if you are looking at time series data. For a time series, choosing a random subset of the data will be both too easy (you can look at the data both before and after the dates your are trying to predict) and not representative of most business use cases (where you are using historical data to build a model for use in the future). If your data includes the date and you are building a model to use in the future, you will want to choose a continuous section with the latest dates as your validation set (for instance, the last two weeks or last month of the available data).

Suppose you want to split the time series data in <> into training and validation sets.

A serie of values

A random subset is a poor choice (too easy to fill in the gaps, and not indicative of what you'll need in production), as we can see in <>.

Random training subset

Use the earlier data as your training set (and the later data for the validation set), as shown in <>.

Training subset using the data up to a certain timestamp

For example, Kaggle had a competition to predict the sales in a chain of Ecuadorian grocery stores. Kaggle's *training data* ran from Jan 1 2013 to Aug 15 2017 and the test data spanned Aug 16 2017 to Aug 31 2017. That way, the competition organizer ensured that entrants were making predictions for a time period that was *in the future*, from the perspective of their model. This is similar to the way quant hedge fund traders do *back-testing* to check whether their models are predictive of future periods, based on past data.

After time series, a second common case is when you can easily anticipate ways the data you will be making predictions for in production may be *qualitatively different* from the data you have to train your model with.

In the Kaggle distracted driver competition, the independent variables are pictures of drivers at the wheel of a car, and the dependent variable is a category such as texting, eating, or safely looking ahead. Lots of pictures were of the same drivers in different positions, as we can see in

<>. If you were the insurance company building a model from this data, note that you would be most interested in how the model performs on drivers you haven't seen before (since you would likely have training data only for a small group of people). This is true of the Kaggle competition as well: the test data consists of people that weren't used in the training set.

Two pictures from the training data, showing the same driver

If you put one of the above images in your training set and one in the validation set, your model will seem to be performing better than it would on new people. Another perspective is that if you used all the people in training your model, your model may be overfitting to particularities of those specific people, and not just learning the states (texting, eating, etc).

A similar dynamic was at work in the Kaggle fisheries competition to identify the species of fish caught by fishing boats in order to reduce illegal fishing of endangered populations. The test set consisted of boats that didn't appear in the training data. This means that you'd want your validation set to include boats that are not in the training set.

Sometimes it may not be clear how your validation data will differ. For instance, for a problem using satellite imagery, you'd need to gather more information on whether the training set just contained certain geographic locations, or if it came from geographically scattered data.

Now that you have got a taste of how to build a model, you can decide what you want to dig into next.

A *Choose Your Own Adventure* moment

If you would like to learn more about how to use deep learning models in practice, including identifying and fixing errors, and creating a real working web application, and how to avoid your model causing unexpected harm to your organization or society more generally, then keep reading the next chapters, *From model to production*, and *Data ethics*. If you would like to start learning the foundations of how deep learning works *under the hood*, skip to <>, *Under the hood: training a digit classifier*. (Did you ever read *Choose Your Own Adventure* books as a kid? Well, this is kind of like that... except with more deep learning than that book series contained.)

Either way, you will need to read all these chapters in order to progress further in the book; but it is totally up to you which order you read them in. They don't depend on each other. If you skip ahead to <>, then we will remind you at the end of that section to come back and read the chapters you skipped over before you go any further.

Questionnaire

It can be hard to know in pages and pages of prose what are the key things you really need to focus on and remember. So we've prepared a list of questions and suggested steps to complete at the end of each chapter. All the answers are in the text of the chapter, so if you're not sure about anything here, re-read that part of the text and make sure you understand it. Answers to all these questions are also available on the book website. You can also visit the forums if you get stuck to get help from other folks studying this material.

1. Do you need these for deep learning?
    - Lots of math T / F
    - Lots of data T / F
    - Lots of expensive computers T / F
    - A PhD T / F
2. Name five areas where deep learning is now the best in the world.
3. What was the name of the first device that was based on the principle of the artificial neuron?
4. Based on the book of the same name, what are the requirements for "Parallel Distributed Processing"?
5. What were the two theoretical misunderstandings that held back the field of neural networks?
6. What is a GPU?
7. Open a notebook and execute a cell containing: `1+1` . What happens?
8. Follow through each cell of the stripped version of the notebook for this chapter. Before executing each cell, guess what will happen.
9. Complete the Jupyter Notebook online appendix.
10. Why is it hard to use a traditional computer program to recognize images in a photo?
11. What did Samuel mean by "Weight Assignment"?
12. What term do we normally use in deep learning for what Samuel called "Weights"?
13. Draw a picture that summarizes Arthur Samuel's view of a machine learning model
14. Why is it hard to understand why a deep learning model makes a particular prediction?
15. What is the name of the theorem that a neural network can solve any mathematical problem to any level of accuracy?
16. What do you need in order to train a model?
17. How could a feedback loop impact the rollout of a predictive policing model?
18. Do we always have to use 224x224 pixel images with the cat recognition model?
19. What is the difference between classification and regression?
20. What is a validation set? What is a test set? Why do we need them?
21. What will fastai do if you don't provide a validation set?
22. Can we always use a random sample for a validation set? Why or why not?
23. What is overfitting? Provide an example.
24. What is a metric? How does it differ to "loss"?
25. How can pretrained models help?
26. What is the "head" of a model?
27. What kinds of features do the early layers of a CNN find? How about the later layers?
28. Are image models only useful for photos?
29. What is an "architecture"?
30. What is segmentation?
31. What is `y_range` used for? When do we need it?
32. What are "hyperparameters"?
33. What's the best way to avoid failures when using AI in an organization?

Further research

Each chapter also has a "further research" with questions that aren't fully answered in the text, or include more advanced assignments. Answers to these questions aren't on the book website--you'll need to do your own research!

1. Why is a GPU useful for deep learning? How is a CPU different, and why is it less effective for deep learning?
2. Try to think of three areas where feedback loops might impact use of machine learning. See if you can find documented examples of that happening in practice.