Table of Contents

Coding a neural network with Keras 3

Feeding images into a neural network 4

Recognizing image contents with a neural network 6

Adding convolution for translational invariance 7

Designing a neural network architecture for image recognition 9

Exploring the CIFAR-10 data set 12

Loading an image data set 13

Dense layers 15

Convolution layers 16

Max pooling 19

Dropout 20

A complete neural network for image recognition 21

Setting up a neural network for training 22

Training a neural network and saving weights 23

Making predictions with the trained neural network 25

Pre-trained neural networks included with Keras 28

Using a pre-trained network for object recognition 30

Transfer learning as an alternative to training a new neural network 32

Extracting features with a pre-trained neural network 35

Training a new neural network with extracted features 37

Making predictions with transfer learning 38

Introduction to the Google Cloud Vision API 42

Setting up Google Cloud Vision account credentials 43

Recognizing objects in photographs with Google Cloud Vision 44

Extracting text from images with Google Cloud Vision 45

Scanned with CamScanner

Deep Learning – Image captioning

What is a neural network?

Image recognition is the ability for computers to look at a photograph and understand what's in the photograph. Here we're passing in a picture to the neural network, and the neural network is generating a label, dog, because that's the main object that appears in the picture. In the past few years, researchers have made huge breakthroughs in image recognition thanks to neural networks. Using neural networks, it's now possible to recognize objects and photographs with high accuracy. Let's take a quick look at the inner workings and structure of a neural network. A neural network is made up of separate nodes called neurons. These neurons are arranged into a series of groups called layers. Nodes in each layer are connected to the nodes in the following layer. Data flows from the input to the output along these connections. Each individual node is trained to perform a simple mathematical calculation and then feed its result to all the nodes it's connected to. The neural network takes in a set of input values in the input layer. Then those values pass through all the following layers. Each node tweaks the value it receives slightly and passes its result onto the next node. For example, we could use this neural network to do addition. To use it to do addition, we pass in two values we want to add, here two and two, into the input nodes. And it will give us the result in the output layer. And it gives us four. But neural networks aren't limited to doing simple operations like addition. When many layers are connected together and data flows through the entire network, neural networks are able to model complex operations. In this course, we'll learn how neural networks with many layers can recognize objects and images.

Scanned with CamScanner

Coding a neural network with Keras

In this course, we'll be using a software framework called Keras to code our neural networks. Keras is a high-level library for building neural networks in Python with only a few lines of code. Let's take a look at an example neural network. This neural network has two input nodes, then a layer with three nodes, a second layer with three nodes, and then finally an output layer with one node. Also know this, that each node is connected to every node in the following layer. These are called densely connected layers. Densely connected layers are the most basic kind of layer in the neural network. Let's see how this example would look in code. First, we'll create a new neural network model with Keras. It's called a sequential model because we're gonna define each layer in order sequentially, or one layer at a time. Next, we'll add a layer with three nodes. But notice that we are also specifying the network has two input nodes by passing in the input dim perimeter. That specifies the number of nodes in the input layer. Next, we add the second layer with three nodes. And then finally, we add the final layer with one node to act as the output layer. That's all that's required to define the basic neural networking, Keras. The Keras also lets us customize how each layer works. One of the most important things to configure are activation functions. Before values flow from the nodes in one layer to the next, they pass through an activation function. Activation functions decide which inputs from the previous layer are important enough to feed to the next layer. Keras lets us choose which activation function is used for each layer by passing in the name of the activation function that we wanna use. Here when we call model.add, I've asked it to use a rectified linear unit, or relu activation function. Keras supports all the standard activation functions in use today. There's also less commonly needed things that we can customize for each layer beyond the activation function. But one of the guiding principles of Keras is that it'll do the best thing that it can if we don't specify

Scanned with CamScanner

extra perimeters. In other words, the default settings are modeled after what are considered best practices. So most of the time, just choosing the number of nodes in a layer and choosing an activation's good enough. The final step of defining a neural network is to compile it by calling model.compile. This tells Keras that we're done defining the model and that we actually wanna build it out of memory. When you compile a model, you have to pass in the optimizer algorithm and the loss function that you wanna use. The optimizer algorithm is used to train the neural network. The loss function is how the training process measures how right or how wrong your neural networks predictions are. This is a complete neural network that we can train to solve very simple classification problems. But to recognize objects and images, we need to create much larger neural networks with much larger input layers and more complex layer types. If you're feeling a little overwhelmed, don't worry. We'll step through each line of code as we build image recognition neural networks. But if you're totally new to the neural networks in Keras, I encourage you to check out my other course in the library called Building Deep Learning Applications with Keras 2.0. It goes into more depth about coding in Keras, and it covers activation functions and optimizers.

Feeding images into a neural network

We've seen the neural networks take the numbers and output numbers. In this example we have a simple neural network that takes the two numbers and outputs the sum of those numbers. But how do we feed in more complex data like images? Let's take a look the how images are stored on the computer. This is a typical image. Let's zoom in on the area around the dog's paw so that we can see the individual pixels that make up the image. Here we can see that the image is actually a series of individual color pixels. But each color pixel is actually made up of a mix of three colors. Red, green, and blue. In the image file, the three different color channels are stored separately. There's actually three separate images in the image file.

Scanned with CamScanner

One that represents the red intensity, one that represents the green intensity, and one that represents the blue intensity of each pixel. The color image that you see on the screen, is just a mix of those three colors. Each pixel is just a number from 0 to 255 that represents how intense the color should be at that point. Here we can see that the bright points are close to 255 and the dark points are close to zero. That means that each color channel is really just a two dimensional ray of integers with one number for each pixel in the image. Inside of the image file, there are three separate rays like this, one for each color. If we layer the three color channels on top of each other, we can think of an image as a three dimensional ray as always three layers deep. So, to be able to feed this image into a neural network, we need a neural network that has one input node for every number in this 3-D array. These numbers add up very quickly. Let's pretend that we want recognize what appears in a 256x256 pixel image. In modern terms, that's a pretty tiny image. Any cell phone takes pictures that are many, many times higher resolution. But even with the small image, we need 256 x 256 x 3 input nodes in our neural network. That comes out to over 196 thousand input nodes. And that's just for the input layer. Each layer of the neural network will use even more nodes. The number of nodes in the entire neural network will quickly grow into the millions. That's why using neural networks for image processing is so computational intensive. Processing an image requires sending it through a neural network of millions of nodes. Because of this, image recognition systems tend to work with images that are very small. It's very common to build image recognition systems that work with images that are between 128 and 512 pixels wide. Larger than that, it gets too slow and requires too much memory. When working with larger images, we usually just scale them down to those smaller sizes before feeding them into the neural network. But even though working with image data is computationally intensive, modern computers have gotten very fast. We have techniques that handle this and we'll talk about that in this course.

Scanned with CamScanner

Recognizing image contents with a neural network

Let's look at how to train the neural network to recognize images. Before we can use the neural network to make classifications we need to train it. Let's pretend that we're training a simple neural network that can recognize the handwritten number eight. To do this we need to collect the dataset of training images. We need lots of images that belong to the class that we wanna recognize. In this case we'll collect 1000 images of the letter eight written by lots of different people. We also need lots of images that represent the other possible kinds of images that we'll see. That way the neural network can learn to tell the two different types or classes of images apart. So we'll also collect 1000 images of other numbers as part of our training dataset. Now that we have training data we can train the neural network by showing it images and telling it what the correct answer will be for each one. So we'll start by showing it some eights and telling it the answer should be one to represent true. We'll give it lots of different examples of what eights could look like. We'll also show it other numbers and tell it that the answer should be zero to represent false. After repeating this process over and over with lots of images the neural network will eventually learn weights for each node that make it possible to separate the images of the number eight from the images of other numbers. Now that the neural network is trained we can pass in a new image and it will tell us its best guess for the correct answer. This is called a prediction. But instead of telling us exactly zero or one it'll give us a floating point number in that range depending on how well it can determine the correct class. So if we pass in an eight it might give us a value close to one like 0.98. And if we pass in another number it might give us a number close to zero like 0.04. But being able to only detect one kind of digit isn't that useful and we could train a separate neural network to recognize each different digit but that's a lot of work. Instead we can build a single neural network that has more than one output. If we give the neural network one output for each possible digit that we want to detect it'll give us a probability score for each

Scanned with CamScanner

one. This works the same way as before except that we train it by telling it to predict the one for the correct answer and a zero for all the other answers. In a modern image recognition model we're often building neural networks that can predict thousands of different classes of objects. It's not unusual to have a thousand or more output nodes in the neural network. When we train the neural network the different layers of the neural network naturally learn to look for different patterns in the input data. You can roughly think of the top layers as looking for simple patterns like lines and sharp edges. And the lower layers use a signal from the higher layers to look for more and more complex shapes and patterns. With all the layers working together the model can identify very complex objects. That means that adding more layers to a neural network tends to give it the capacity to learn more complex patterns and shapes. This is where the term deep learning originally came from. Deep learning is just the idea that making models deeper by adding more capacity to them lets us recognize more complex patterns and data.

Adding convolution for translational invariance

We know that we can train the neural network to recognize an image like this eight, but what happens when the data isn't that clean and simple? If we only train the neural network with pictures of numbers that are perfectly centered, the neural network will get confused if it sees anything else. For example, if we pass in an image like this, where the eight is not centered, the neural network won't be able to make a good prediction. But the eight could appear anywhere in the image. It could just as easily appear at the bottom like this. We need to improve our neural network so that it can recognize objects in any position. This is called translation invariance. This is the idea that a machine learning model should recognize an object no matter where it is moved within the image. Moving the eight around the image doesn't change the fact that it's still just an eight. The solution is to add a new type of layer to our

Scanned with CamScanner

neural network called the convolutional layer. Unlike a normal dense layer, where every node is connected to every other node, this layer breaks apart the image in a special way so that it can recognize the same object in different positions, let's see how it works. The first step is to break the image into small, overlapping tiles. We do this by passing a small window over the image. Each time it lands somewhere, we grab a new image tile. We repeat this until we've covered the entire image. Next, we'll pass each image tile through the same neural network layer. Each tile will be processed the same way and we'll save a value each time. In other words, we're turning the image into an array, where each entry in the array represents whether or not the neural network thinks a certain pattern appears at that part of the image. Next, we'll repeat this exact process again. But this time, we'll use a different set of weights on the nodes in our neural network layer. This will create another feature map that tells us whether or not a certain pattern appears in the image. But because we're using different weights, they'll be looking for a different pattern than the first time. We can repeat this process several times until we have several layers in our new array. This turns our original array into a 3D array. Each element in the array represents where a certain pattern occurs. But because we are checking each tile of the original image, it doesn't matter where in the image a pattern occurs. We can find it anywhere. This 3D array is what we'll feed into the next layer of the neural network. It will use this information to decide which patterns are most important in determining the final output. Adding a convolutional layer makes it possible for our neural network to be able to find the pattern, no matter where it appears in an image. In this example, we have only one convolutional layer. But normally, we'll have several convolutional layers that repeat this process multiple times. The rough idea is that we keep squishing down the image with each convolutional layer while still capturing the most important information from it. By the time we reach the output layer, the neural network will have been able to identify whether or not the object appeared. Convolutional neural networks are a secret weapon for image detection, they make it possible to efficiently detect

Scanned with CamScanner

objects. It's the standard approach for building image recognition systems.

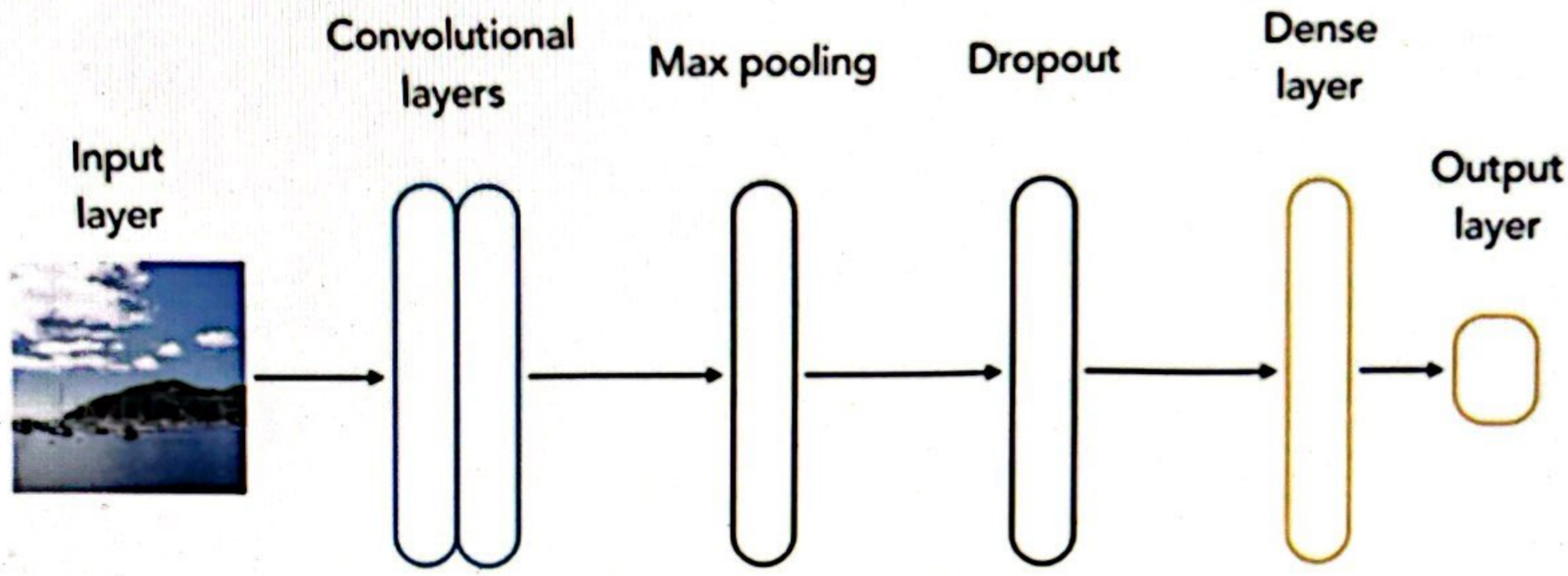
Designing a neural network architecture for image recognition

Before we start coding our image recognition neural network, let's sketch out how it will work. This is the most basic neural network design. We feed it an image, it passes through one or more dense layers, and then it returns an output, but this kind of design doesn't work efficiently for images because objects can appear in lots of different places in an image. The solution is to add one or more convolutional layers to our neural network. These layers will help us detect patterns no matter where they appear in our image. It can be effective to put two or more convolutional layers in a row, so in our neural network, we'll add them in pairs. Our design so far, with two convolutional layers and the dense layer, would work for very simple images, but there are some tricks that we can add to our neural network to make it more efficient. The convolutional layers are looking for patterns in our image and recording whether or not they found those patterns in each part of our image, but we don't usually need to know exactly where in an image a pattern was found down to the specific pixel. It's good enough to know the rough location of where it was found. To solve this problem, we can use a technique called max pooling. Let's look at an example. Imagine that this grid is the output of a convolutional filter that ran over a small part of our image. It's trying to detect a particular pattern and these numbers represent whether or not that pattern was found in the corresponding part of the image. Let's assume that this filter is looking for patterns that look like clouds. A zero in the grid means that the pattern wasn't found at all and a one means the area was a strong match for the pattern. We could pass this information directly to the next layer in our neural network, but if we can reduce the amount of information that we pass to the next layer, it will make

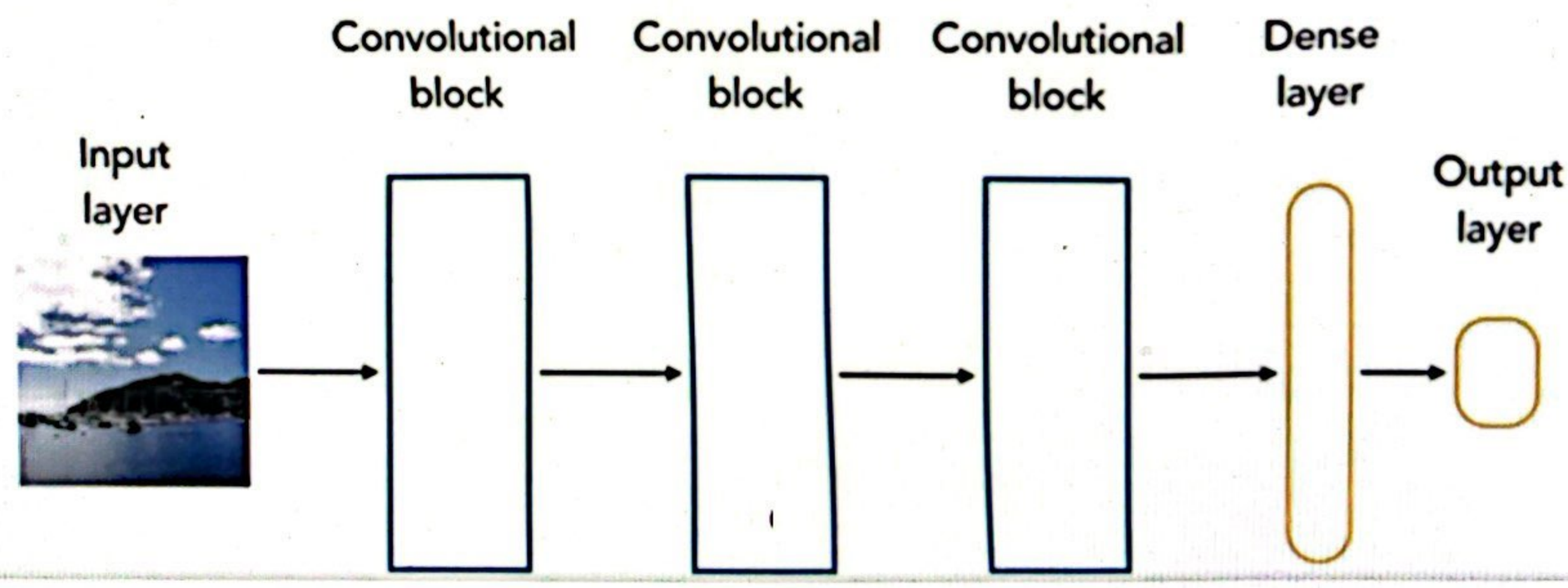
Scanned with CamScanner

the neural network's job easier. The idea of max pooling is to down sample the data by only passing on the most important bits. It works like this. First, we divide this grid into two-by-two squares. Then, within each two-by-two square, we'll find the largest number. If there's a tie, we'll just grab the first one. And then finally, we'll create a new array that only saves the numbers that we selected. The idea is that we're still capturing roughly where each pattern was found in our image, but we're doing it with 1/4 as much data. We'll get nearly the same end result, but they'll be a lot less work for the computer to do in the following layer of the neural network. We have another trick that we can use to make our neural network more robust, it's called dropout. One of the problems with neural networks is that they tend to memorize the input data instead of actually learning how to tell different objects apart. We need a way to prevent that. There's a simple way that we can force the neural network to try really hard to learn within just memorizing its training data. The idea is that we'll add the dropout layer between other layers that will randomly throw away some of the data passing through it by cutting some of the connections in the neural network. It's like going into a computer and just randomly unplugging some cables. By randomly cutting different connections with each training image, the neural network is forced to try harder to learn. It has to learn multiple ways to represent the same ideas because it can't depend on any particular signal always flowing through the neural network. It's called dropout because we're just letting some of the data drop out of the network randomly. Dropout is an idea that might seem counterintuitive, we're actually throwing away data to get a more accurate final result, but in practice it works really well. We have four different kinds of layers in this neural network. The convolutional layers add translational invariance, the max pooling layers down sample the data, and dropout forces the neural network to learn in a more robust way. And then finally, the dense layer maps the output of the previous layers to the output layer so we can predict which class the image belongs to.

Scanned with CamScanner



The first three layers work really well together, so we'll put them together into a block and we'll call the whole thing a convolutional block.



If we wanna make our neural network more powerful and able to recognize more complex images, we can add more layers to it. But instead of just adding layers randomly, we'll add more copies of our convolutional block. When all these layers are working together, we'll be able to detect complex objects like dogs or cars or airplanes. This is a very typical design for an image recognition neural network, but it's also one of the most basic. Researchers are always experimenting

Scanned with CamScanner

with new and increasingly complex ways of chaining together layers to improve the accuracy of their neural networks. The latest designs involve branching pathways, shortcuts between groups of layers and all sorts of other tricks, but they all build on these same basic ideas and this is the approach we'll use in our code.

Exploring the CIFAR-10 data set

To train neural networks to perform accurately, you need large amounts of training data. Since it's difficult to collect thousands of training images, researches build data sets and share them with each other. For our first image recognition project, we'll be using the CIFAR-10 dataset. This dataset includes thousands of pictures of 10 different kinds of objects, like airplanes, automobiles, birds, and so on. Each image in the dataset includes a matching label so we know what kind of image it is. Using this dataset, we can train our neural network to recognize any of these 10 different kinds of object. Before we build an image recognition model, the first step is to look through the training data that we are working with. We wanna check for bad or unexpected training data. Bad training data is a very common source of problems. For example, imagine that you take millions of photographs and ask volunteers to label them for you. This is called crowd sourcing and is a common way to label large data sets. What if one of the labels you ask your volunteers to use is jaguar and you have pictures of both large cats and sports cars? The volunteers might mix up the label and sometimes use it for cats and sometimes use it for sports cars. Because problems like this are common, it's always worth spending some time with your training data and looking for obvious errors or problems. The images in the CIFAR-10 dataset are only 32 pixels by 32 pixels. These are very low resolution images. We're using them here because the lower resolution will make it possible to train the neural network to recognize them relative quickly. With the same code we'll write, we'll also work for larger image sizes. To make it easy for you to look through the

Scanned with CamScanner

CIFAR-10 dataset I've included some code that will display the images from the dataset on the screen. Let's go over the PyCharm. I'm gonna open up 02 view image data set dot py. First on line five, we have a list of the 10 different kinds of images in the dataset. Zero is plane, one is car, and so on. Then on line 19, we'll load the dataset into memory. Keras provides this helper function that makes it easy to access CIFAR-10. Then on line 22, we'll loop through the first 1,000 images with a for-loop. On line 24, we grab an image from the dataset and then on line 26 we grab that image's label. Then on line 28, we'll look up the string name of that label from the list of labels we have at the top of the program. Then, finally, starting on line 31, we'll use Python's Pyplot library to draw the image on the graph and show it. Let's run the program. Right click, chose run. Here's the first image in the dataset. It says it's a picture of a frog and if you squint, you can kinda see that it's a frog. To see the next image in the dataset, just close this window and it will show you another image. Try looking through several of the pictures and seeing if the labels look correct to you. When you've got a good feel for the data, you can go back to PyCharm and then you can click this terminate button twice to stop the program.

Loading an image data set

To train a neural network we need a set of training images. Let's write the code to load and pre process our training images so they're in the right format to feed into a neural network. Let's go ahead and open up 03 loading image dataset.py. For this neural network we'll be using the cifar10 data set. Since the cifar10 data set is used so often, Keras provides a function for easily accessing it. Here on line eight to load the data we'll call cifar10.loaddata. This function returns four different arrays. First it returns an x and y array of training data. So we'll say x_train,y_train=that function call. The x array will contain the actual images from the data set. The y array contains the matching label for each image. The function also returns an x and y

Scanned with CamScanner

array of test data. So we'll add x_test, and y_test. The test data is in the same format as the training data, it's just additional images that we can use to test the neural network to make sure it's performing well. It's always important to test a neural network with data that it didn't see during training to make sure it actually learned how to tell the differences between images and didn't just memorize the training data. Before we can use this data to train a neural network, we need to normalize it. Neural networks work best when the input data are floating point values in between zero and one. Normally images are stored as integer values for each pixel is a number between zero and 255. So to use this data, we need to convert it from integer the floating point and then we need to make sure all the values are between zero and one. So let's go to line 11 and here let's convert the data to floating point values. We can do that by using the as type function and passing in float 32. So first we'll say x train=x train.astype and we'll pass in float32. Then we'll do exactly the same for the test array. We'll say x test = x test astype float32. Now we'll need to scale the data so it's between zero and one. Since we know that our pixel data is between zero and 255, we can just divide all the array values by 255. So we can say x train = x train divide by 255. X test = x text divided by 255. When we divide the NumPy array by a single value like this it will divide every separate array element by 255. It's just a shorter way of writing it without having to loop through every array and divide every element. There's one last bit of cleanup we need to do before we can use our training data. Cifar10 provides the labels for each class as values from zero to nine. But since we are creating a neural network with 10 outputs, we need a separate expected value for each of those outputs. So we need to convert each label from a single number into an array with 10 elements. In that array, one element should be set to one and the rest set to zero. This is something you'll almost always need to do with your training data so keras provides a helper function. It's called keras.utils.to categorical. So let's go to line 19, and we'll say y train = keras.utils.to_categorical. To use that function you just pass in your array with the labels which in our case is y train. And then you pass in the number of classes it has. We know cifar10 has 10 classes. And

Scanned with CamScanner

then we can do exactly the same thing for y test. So we'll say keras.utils.to_categorical pass in y test and 10 classes. And now we've got this data ready to use with the neural network. Let's just run the code and double check all our code work so far. Right click choose run and it looks good. Notice when you run your code you might get these two warning messages. That's okay and that's expected.

Dense layers

Now that we've loaded our data set, we're ready to create a neural network and add the first densely connected layer to it. Let's open open up 04 dense layers.py. The code to load the data set is already here. Starting on line 21 we're ready to add the code to create the neural network itself. The simplest type of neural network has an input, a densely connected layer and then an output. Let's start by creating that. First we need to create a new neural network object in Keras. To do that, we create a new sequential object. So we say model = sequential. The sequential api lets us create a neural network by adding new layers to it one at a time. It's call sequential because you add each layer in sequence and they automatically get connected together in that order. To add a new layer we just call model.add. And then we pass on the type of layer that we want to add. Let's create a dense layer object. This layer class takes on a few parameters. First, we need to tell it how many nodes to include in the layer. Let's add 512 nodes to this layer. So we'll just pass in 512. Next we need to tell it what activation function we want to use for this layer. For a normal layer like this, a common choice is to use a rectified linear unit or relu activation function. It's the standard choice when working with images because it works well and is computationally efficient. So let's use that. We'll say activation=relu. And since this is the first layer in the neural network, we need to tell it the size of the input layer. All the images in our data set are 32 pixels by 32 pixels and have a red green and blue channel. So for the input size we'll use 32 by 32 by 3. So we pass that in there's a

Scanned with CamScanner

parameter called input shape. And then we pass in a list with the values 32, 32 3. And that's everything we need for this layer. Let's go ahead and add the output layer. We'll need one node in the output layer for each kind of object we want to detect. The cifar10 data set has 10 different kinds of objects. Since we're detecting 10 different kinds of objects, we'll create a new dense layer with 10 nodes. So to do that we'll call model.add and we'll create a new dense object and we know it needs 10 nodes. When doing classification with more than one type of object, the output layer will almost always use a softmax activation function. The softmax activation function is a special function that makes sure all the output values from this layer add up to exactly one. The idea is that each output is a value that represents the percent likelihood that a certain type of object was detected. And all 10 values should add up to 100 % or one. So to do that we just say activation = and we pass in the word softmax. When we're building a neural network and adding layers to it, it's helpful to print out a list of the layers in the neural networks so far. Let's go down to line 26 and we can do that by just calling model.summary. Let's run this code and see what the neural network structure looks like so far. Right click choose run and let's expand this area a little bit. Here's the output and we can see that we have two layers so far. Both are dense layers and they're in the right order. Everything looks good so far.

Convolution layers

So far, we've created the neural network with densely connected layers. Now we're ready to add convolutional layers to make it better at finding patterns in images. Let's open up 05_convolutional_layers.py. To be able to recognize images officially, we'll add convolutional layers before our densely connected layers. Convolutional layers are able to look for patterns in an image, no matter where the pattern appears in the image. Let's go down to line 22, this is where we'll insert a convolutional layer. First, to add the layer, we'll call model.add. Now there's two types of convolutional

Scanned with CamScanner

layers: 1D and 2D. Since we're working with images, we'll want to add the two dimensional convolutional layer. For some kinds of data, like sound waves, you can use one dimensional convolutional layers, but typically you'll be working with 2D layers. To create one, we just create a new Conv2D object and then pass in the parameters. The first parameter is how many different filters should be in the layer? Each filter will be capable of detecting one pattern in the image. We'll start with 32. Next, we need to pass in the size of the window that we'll use when creating image tiles from each image. Let's use a window size of three pixels by three pixels. So to do that, we pass in an array of three comma three. This will split up the original image into three by three tiles. When we do that, we have to decide what to do with the edges of the image. If the image size isn't exactly divisible by three, we'll have a few extra pixels left over on the edge. We can either throw that information away, or we can add padding to the image. Padding is just extra zeros added to the edge of the image to make the math work out. The terminology that Keras uses here is a bit confusing. If we want to add extra padding to the image, it's called same padding. There's complex historical reasons why researchers used the term same, but it's easier just to memorize it. For this layer, we do want to have padding, so we'll pass in a parameter padding equals, and the string same, and just like the normal dense layer, convolutional layers also need an activation function. And just like dense layers, we almost always use the relu activation function because of its efficiency. So I'll pass in activation equals relu. And that's it for adding this layer, but there's one more tweak we need to make. Let's look at the next line. This dense layer is no longer the first layer in the neural network, so it shouldn't have an input shape defined anymore. Let's just cut and paste this input shape, and move it up to the convolutional layer because it's now the first layer. To make our neural network more powerful, let's add a few more convolutional layers the same way. First, let's add another one with the same settings, 32 filters and a three by three window size. So we'll say model.add, we'll pass in Conv2D, we'll say 32 filters, and the three by three window size, and we'll also add an activation function, we'll use relu again, activation equals relu. Now in this

Scanned with CamScanner

layer we won't have the image, so we don't need to pass in the padding parameter. Now let's add two more layers with 64 filters each. First we'll add one with padding, so we'll say model.add Conv2D, say 64 filters, I'll use a three by three tile size again. I'll pass in padding equals same, and in activation function we'll use relu. And now we'll do one more without padding, but also with 64 filters. So we can just cut and paste this, paste it here, and just remove the padding. Alright, there's just one thing left to do. Whenever we transition between convolutional layers and dense layers, we need to tell Keras that we're no longer working with 2D data. To do that we need to create a flattened layer and add it to our network. We can do that by calling model.add, and creating a new flattened layer, and there's no parameters required for a flattened layer. Alright, if you look down at line 35, we can see that we're printing out the summary of the neural network structure, so let's run this code and see what it looks like. Right click and choose run. Alright, we can see the neural network now has seven layers. We have four convolutional layers, the flattened layer, and then our two dense layers. Notice that each layer also has a number of parameters listed. This is the total number of weights in that layer. There's also a total number at the bottom for the whole network. As we add more layers that total number will keep increasing. This is the size or complexity of our neural network. The larger the number, the longer it'll take to train and the more data we'll need to train it. It's a good idea to keep an eye on this number as you add layers to your neural network. As you test and refine your neural network, you might find that you can get good results even after you remove some of your layers and reduce this number. When you can do that, that means you'll need less powerful hardware to run your neural network, so it's always a good goal.

Scanned with CamScanner

Max pooling

So far, we've created a basic convolutional neural network. Let's improve its efficiency by adding max pooling. Open up 06_max_pooling.py. Max pooling is where we scale down the output of the convolutional layers by keeping only the largest values and throwing away the smaller ones. This makes the neural network more efficient by throwing away the least useful data and keeping the most useful data. Typically, we'll do max pooling right after a block of convolutional layers. Let's go down to line 25 after we add these two layers, and add our max pooling layer. To do that, we call model.add, and then create a new max pooling 2D layer. The only parameter that we have to pass in is the size of the area that we wanna pool together. We'll use a two pixel by two pixel pool size. So, we'll pass in pool size equals, and then the ray, two comma two. That means that we'll divide their image up into two by two squares and only take the larges value from each two by two region. That will reduce the size of our image while keeping the most important values. Let's add another max pooling layer after the next two convolutional layers exactly the same way. We can just and paste this line. So, we'll copy line 25 down here. Now let's run the code and check the network structure. Right click, choose run. Notice how the max pooling layers have zero parameters. Instead of increasing the size of our neural network, they're actually helping us decrease the size by scaling down the data that passes through them while keeping the most important values. This will help us speed up the training process.

Scanned with CamScanner

Dropout

Selecting transcript lines in this section will navigate to timestamp in the video

- We have one more trick left up our sleeves to make our neural network perform better. Let's add dropout layers to it. Open up 07_dropout.py. One of the problems with neural networks is that they can tend to memorize the input data instead of actually learning how to tell different objects apart. We can force the neural network to try harder to learn without memorizing the input data. The idea is that between certain layers, we'll randomly throw away some of the data by cutting some of the connections between the layers. This is called dropout. Usually we'll add dropout right after max pulling layers, or after a group of dense layers. Let's go down to line 26 right after this max pulling layer. To add a new layer, we'll just call model.ad and we'll create a new dropout layer. The only parameter we need to pass in is the percentage of neural network connections to randomly cut. Usually a value between 25% and 50% works well. We'll use 25%. To do that, we pass in 0.25. Now let's add another dropout layer the same way after the next max pulling layer on line 31. We can just cut and paste this, copy, paste it down here, and that looks good. Let's also go down to line 35. We can add a dropout layer after this dense layer. Here, let's make the neural network work really hard to get the last answer correct. Let's use a 50% dropout. We'll say model.add, create a new dropout layer, and we'll pass in 0.5 for 50%. Now let's run the code to make sure everything looks correct. Right click, choose run. Here's our layers. Notice that the dropout layers have no parameters. Since they're just throwing away data, they aren't adding any extra values to our neural network.

Scanned with CamScanner

A complete neural network for image recognition

Let's finish the code to define our neural network. All that's left to do is tell Keras to compile it. Open up 08_complete_neural_net.py. The final step of defining a neural network is to compile the neural network. When we compile it, we're telling Keras we actually want to create the neural network in memory. We're also telling Keras how we'll be training it and measuring its accuracy. Let's go down to line 39. To compile the neural network, we'll call model.compile. This function takes several parameters. First, we need to tell Keras how to check how right or wrong the guesses from our neural network are. This is called the loss function. So we'll pass in a loss parameter, loss equals. You can write your own loss function, but most of the time there are a few standard functions that you'll choose between. If you're trying to classify an image into different categories, you can use a function called categorical crossentropy. If you're only checking if an image belongs to one category, you'll use binary crossentropy. Since we have 10 different possible categories for our objects with our data set, we'll use categorical crossentropy. So we'll pass that in as a string. Categorical crossentropy. Next, we need to tell Keras which optimization algorithm we'll use to train the neural network. As researchers come up with more efficient training algorithms, Keras adds support for them. For image data like this, a good starting point is to use an optimization algorithm called Adam, or Adaptive Moment Estimation. We'll use that one. So to do that, we'll pass in, optimizer equals adam. Finally, we need to tell Keras what metrics we want it to report during the training process. For this we can just use accuracy. So we'll pass in, metrics equals, and then inside in the array, accuracy, as a string. We pass it in the array for metrics since you can choose to report more than one, but for our purposes, accuracy is all we need. All right, just to make sure we've typed in everything correctly, let's run the code. Right click and choose Run. And everything looks good. Notice that compiling the neural network doesn't actually start the training process. In the next section, we'll

Scanned with CamScanner

run the training process and use the training neural net to make predictions.

Setting up a neural network for training

- In the previous section we wrote all the code for our neural network. Now, we're ready to write the code that starts the training process. Open up 0-one neural network training dot p y. We already have the code here that loads the data set, and we have the code for all the layers in the neural network and here on line 38 we've already compiled the neural network. Now, we just need to add the code to start the training process. Let's do that down here, on line 45. To start the training process in Kerris you call the model dot fit function. This function takes several parameters. The first two parameters to fit are the training data set, and the expected labels for the training data set. We already loaded those up in our code as x training and y training. So, you can pass those in here. So, I'll pass in x training and y training. Next, we need to pass in a batch size. The batch size is how many images we want to feed into the network at once during training. If we set the number too low, training will take a long time and might not ever finish. If we set the number too high, we'll run out of memory on our computer. Typical batch sizes are between 32 and 128 images, but feel free to experiment. For this example let's use a batch size of 32. So, say batch size equals 32, next, we need to So, say batch size equals 32, next, we need to decide how many times we wanna go through our training data set during the process. One full pass through the entire training data set is called an epoch. For this example, let's do 30 passes through the training data set. So, we'll pass in epochs equals 30. So, we'll pass in epochs equals 30. The more passes through the data we do, the more chance the neural network has to learn; but the longer the training process will take. And eventually you'll hit a point where doing additional training doesn't help anymore. So, finding the right number takes some experimentation. In general, the larger your data set, the less training passes you'll do on it. For example, for extremely large

Scanned with CamScanner

data sets with millions of images you might only do five passes. Next, we need to tell Kerris what data we wanna use to validate our training. This is data that the model will never see during training, and it'll only be used to test the accuracy of the training model. When we loaded our data set we created x test and y test, so we can use those. But pass those in as validation data. So, we'll pass in the parameter called validation data and then, in an array we pass in x test and y test. Finally, we need to make sure that Kerris randomizes the order of the training data. It's very important that the neural network sees the training data batches in random order, so that the order of the training data doesn't influence the training. To insure that we'll pass in shuffle equals true. Shuffling is actually the default in Kerris, but I think it's important enough that I would explicitly include it in case of changes in a future version. Not shuffling your data can cause your model to fail to train correctly, and that's it. We're ready to train the model, but notice that we aren't saving the results anywhere. If we run the training process right now we'll be doing all the work and then throwing away the results. In the next video we'll see how to save our training results, we'll wanna do that before we run the lengthy training process.

Training a neural network and saving weights

When we train a neural network, we wanna make sure that we save the results, so that we can reuse the trained model later. Let's learn how to train our neural network and save the results to a file. Open up 02 training and saving weights dot py. Here on line eight, we've already written the code to load our dataset, and then we've coded our neural network. And then on line 39, we've compiled it. And on line 46, we've started the training process. But after training completes, we wanna save the trained neural network to a file so we'll be able to use it to recognize objects and images in other programs. Let's start that on line 56. Saving a neural network is two separate steps. First, we wanna save the structure of the neural network itself. That includes which layers get created and the order

Scanned with CamScanner

that they're hooked together. We could rewrite the neural network code again from scratch each time we use it, but it's a lot easier to save the structure to a file and just load it when we need it. Second, we wanna save the weights of the neural network. As a neural network is trained, the weights of each node are adjusted to control how the signals flow through the network. So by saving the weights, we're saving how the neural network actually works. The reason we save the structure separately from the weights is because often you'll train the same neural network multiple times with different settings or different training datasets. It's convenient to be able to load different sets of weights using the same neural network structure. So first, let's save the neural network structure itself. CARIS can convert the structure of a neural network into JSON by calling the model dot to JSON function. So we'll say 'model structure equals model dot to underscore json' Now, we just need to write this JSON data to a text file. There's lots of ways to do this in Python, but here is one easy way to do it using the path library. First we'll create a new path object, so we'll say, 'f' for file equals path, and then we'll pass in the name of the file we wanna create. So I'm just gonna call it model structure dot JSON. (typing) Then we just need to call the right text function of the path object and pass in the data that we wanna write to the file. So I'll do 'f' dot write text and the data that I want to write is this model structure object, so I'll pass in model structure. Alright, now we wanna save the weights of the neural network. This is even easier, we just need the call model that save weights and pass in the file name. So lets go down to line 61, and I'll write model dot save weights. I'm gonna call the file 'model weights dot h5' The data that gets saved here is in a binary format called HDF5. The HDF5 format is designed for saving and loading large binary files efficiently. So by convention we're using the h5 file extension to indicate the format of the file. Alright, we're ready to train the neural network. To do that, just right click and choose 'run', and we can watch the progress in the console here. During the training process CARIS outputs a progress bar so we can watch what's happening. The first number on the left represents how many samples in our training dataset have been processed. There's 50,000 total images in our

Scanned with CamScanner

training dataset, so we can watch this number increase as training continues. The progress bar itself represents how far along in this pass we are through the training data. Keep in mind, we asked it to do 30 passes through the training data here on line 50. That means that we'll do 30 complete passes through this data. So when the progress bar is complete, that's just the first of 30 total passes. The ETA tells us how much longer this single pass should take. The loss is the numerical representation of how wrong our neural network is right now. The lower this number the better our neural network is performing. We want to see this number go down during the training process. The final number is the current accuracy. This represents how often our neural network is making the correct prediction for the training data. We wanna see this number go up over time as the training process continues. If the loss doesn't go down, and the accuracy doesn't go up over time, that means there's either something wrong with the neural network design, or that there are problems with the training data. In that case you have to go through your code and data step by step and make sure everything looks correct. If that doesn't help, it's possible that your dataset is too small to train your neural network, or that your neural network doesn't have enough layers to capture the patterns in your dataset.

Making predictions with the trained neural network

Now that we've trained our neural network, let's use it to look at new images and make predictions. Let's open up 03_making_predictions.py. When we pass an image through our neural network, it's going to return a likelihood for each type of object it was trained to recognize. In order to decode those numbers into names, we need a list of names that correspond with each number. Here, on line seven, I've already listed the names that were used during the training process. These names correspond to the 10 types of objects that were in the CIFAR10 data set. Now, we're ready to load the neural network. First, we need to load the structure of

Scanned with CamScanner

the network itself. One option is to write out all the code for all the layers of the neural network again, as long as we match what was used during training, but it's a lot easier to load the neural network structure from a file instead. Here in the file list, we already have a file called model_structure.json. This file contains the list of layers in our neural network, and all the details about how they were hooked together. On line 21, we're going to load that text file into memory. We can do that in Python by creating a new path object that represents the file that we want to load. So we'll say f = Path(), and then as a string, in quotes, we'll pass in the name of the file we want to load, which is model_structure.json. Then, to load the file, we can call f.read_text() and save the results to a variable. So I'll say model_structure = f.read_text(). Now that we have the file in memory, we need to tell Keras to rebuild the model using that data. Keras provides a helper function to do this called model_from_json(). So here, on line 25, we'll say model = model_from_json(), and then we'll pass in the model_structure variable we just created. So far, we've only restored the structure of the neural network. To restore its training as well, we need to load the weights file we created when we trained the neural network. Here in the file list, we have a file called model_weights.h5 that we created when we trained the model. To load it, we'll just call model.loadweights() and pass in the filename. So here, on line 28, we'll call model.loadweights(), and we'll pass in the filename, which is model_weights.h5. Great, now the neural network should be ready to use. Let's find an image that we can use to try it out. Here in the file list, I have a file called cat.png. Let's take a look. Yep, it's a picture of a cat. Let's close it and go back to the code. On line 31, let's load this image file. To load an image file, we can use a Keras helper function called image.load_img(). So I'll say img, which is my image, = image.load_img(), and then I just pass in the filename to load, which is cat.png. And finally, we need to tell it to resize the image to the size the neural network expects. We trained this neural network with images that were 32 pixels by 32 pixels, so that's the size we need for any images that we feed into it. To do that automatically, we can pass in the parameter here called target_size. So we'll say target_size

Scanned with CamScanner

= , and then the array (32, 32). Great, now that the image data is in memory, we need to convert it to a 3-D numpy array so that we can feed it to our neural network. There's a helper function for this, too, called image.img_to_array(). All right, so let's go down to line 34, and we'll say image_to_test, which is the one that we'll pass into the neural network, and we'll say this = image.img_to_array(). And then, we'll pass in the img variable we just created. Before we go any further, we also need to normalize the image data. The image we are loading from disk stores each pixel as a red, green, and blue value between zero and 255, but the neural network expects an input value between zero and one. So before we can process this image with our neural network, we need to scale the value for each pixel to a value between zero and one. The easiest way to do this is to divide the whole array by 255, so let's add a / 255 to the end of the line. When we divide a numpy array by a single value, numpy will divide each individual element of the array by that value. So doing this will scale each pixel's red, green, and blue value to a zero-to-one range. So now, this image is ready to be processed by our trained neural network. Right now, we're only testing one image with our neural network, but for efficiency reasons, Keras lets you pass in batches of images at once, so you can run more than one image through the neural network at one time. So we need to create a batch of images to pass in, even though we're only testing this one image. Keras expects these batches as a four-dimensional array. The first dimension is the list of images, and the other three dimensions are the image data itself. Here's a little trick. Since we only have this one image, we can turn it into a 4-D array by adding a new axis to it with numpy. You can do this by calling a function called np.expand_dims() and passing in the name of the array. So on line 37, I'm going to say list_of_images = np.expand_dims(), and I'm going to pass in image_to_test, the variable we just created. We also need to pass in axis = 0 to tell it that the new axis is the first dimension. This is the convention that Keras expects. Now, we have a batch of images that we're ready to feed into the neural network and get a prediction. To do that, we'll just call model.predict(). So on line 40, I'll say results = model.predict(), and then I'll pass in that list_of_images we just

Scanned with CamScanner

created. The results variable will contain a list of results for each image that we passed in. Since we only passed in one image, we can just grab the first array index. So on line 43, I'll say single_result = results[0]. The single_result array is an array with 10 elements. Each element represents how likely the image is to belong to each of the object types we listed at the top of the program. Instead of returning 10 separate numbers, let's just grab the array element with the highest value. That will tell us which single object type was the most likely result. Let's do that on line 46 using numpy's argmax function. So I'll say most_likely_class_index = np.argmax(), and then I'll pass in single_result. We also want to convert this to an integer, so we'll wrap that in an int() function. While we're here, let's also grab the likelihood value of that array index so we can print it out later. So right below that, we'll say, class_likelihood = single_result(), and then we'll pass in as the index the most_likely_class_index. Finally, on line 50, let's look up the name of the object type from our list of class labels. So we'll say class_label = class_labels(), the list we had at the top, and we'll pass in the index that we just created, most_likely_class_index. Now, finally, on line 53, we'll just print out the results. Let's run the program and try it out, and see if it can correctly recognize this picture of a cat. Right-click and choose Run. Great, it predicted that our image is a cat with a likelihood of 99%. We can also go up here to line 31 and try a different picture. So let's go up here, and let's replace cat.png with frog.png. Frog.png is another one of the test images we have in our folder. And let's run it again, right-click, choose Run. And great, it got this one right, too. Feel free to try this out with your own images and see what kinds of images work well, and what kinds of images confuse it.

Pre-trained neural networks included with Keras

Researchers around the world compete with each other to build the most accurate and capable image recognition systems. So instead of them bending their own neural network designs from scratch, it

Scanned with CamScanner

often makes sense to reuse an existing neural network design as a starting point for your own projects. But even better, researchers also trained these neural network designs on large data sets and share the trained versions of the neural networks. So we can take those pre trained neural networks and either reuse them directly, or use them as a starting point for our own training. Keras the library that we're using to build neural networks includes copies of many popular pre trained neural networks that are ready to use. The image recognition models included with Keras are all trained to recognize images from the ImageNet data set. The ImageNet data set is a collection of millions of pictures of objects that have been labeled so that you can use them to train computers to recognize those objects. Each year ImageNet holds a worldwide image recognition contest called the ImageNet Large Scale Visual Recognition Challenge, or ILSVRC. Teams from universities and companies around the world compete to build the most accurate image recognition models. The pre trained models included with Keras are trained on the more limited data set used by this contest. The data set contains images of 1,000 different types of objects, like breeds of animals and types of foods. For example, one of the types of objects in the data set is Granny Smith apple. The date set includes over 1200 pictures of just this specific kind of apple. Let's talk about the neural network designs included with Kares that we can reuse. First is VGG, VGG is a deep neural network with either 16 or 19 layers. It was the state of the art in 2014. It's a very standard convolutional neural network design. It's still used widely today as a basis for other models because it's easy to work with and easy to understand. But newer designs tend to be more efficient. ResNet-50 is a state of the art from 2015, it's a 50 layer neural network that manages to be more accurate, but use less memory than the VGG design. ResNet uses a more complicated design, where higher layers in the neural network are connected not just the layer directly below them, but they also have multiple connections to deeper layers. Inception v3 is another design from 2015 that also performs very well. It has an even more complex design based around layers that branch out into multiple separate paths before rejoining. These networks show the research trends in

Scanned with CamScanner

2014 and 2015 to make neural networks bigger and more complex that try to increase their accuracy. More recent neural network designs tend to be more specialized. For example, Google's MobileNet created in 2017 is designed specifically to be able to run well on low power devices. The idea was to create a neural network that could run quickly on a cell phone without using too much power while still maintaining a decent level of accuracy. Google's NASNet which was created at the end of 2017, explores the idea of having algorithms design neural networks. In a sense they're using machine learning to build and tweak machine learning models on their own. This let them create something that was more accurate than existing models while still using even less computer power. Having access to these pre trained models is useful for two reasons. First you can reuse any of these models directly in your own programs to recognize objects and images. If you need the ability to recognize any of the 1,000 types of object they're already trained on, you're problems already solved. Second if you want to recognize a different type of object that's not in the 1,000 object training set, it's much faster to start with a pre trained neural network and adapt it to your needs, instead of training your own model from scratch. In this course we'll test out both options.

Using a pre-trained network for object recognition

Let's see how to use a pretrained model included with Keras to recognize objects and images. Open O2_image_recognition_pretrained.py. All the pretrained models included with Keras are under the applications package. Here on line three we import the model by saying from Keras applications import vgg16. Next, on line six, we'll create a new instance of the model by creating a new vgg16 object. So, to do that we say, vgg16 dot and then an uppercase vgg16. On line nine, I've already loaded an image file to process. I've included the sample image file called bay.jpg that we can use. Let's take a look at it in the file list. This is a picture of

Scanned with CamScanner

a bay with sailboats and some mountains. Hopefully, the model will detect labels related to those objects. Alright, let's go back to the code. Let's close this. And, on line nine we load the image by calling the built in image.loadimagefunction, and passing in the file name. However, the picture that we're loading is too large for us to process directly with neural network. When you feed images into a neural network, the size of the image need to match the number of input nodes in the neural network. For vgg, images we feed into the network need to be 224 by 224 pixels. So, that's what we set the target size parameter, here, to. Next, on line 12, we'll convert the image data into an array of numbers, so we can feed them into the neural network using the image.img to array function. And then on line 15, we add the fourth dimension to the image. This is because Keras expects us to pass in an array of multiple images at once. This turns our single image into an array of multiple images with just one element. Alright, let's go to line 18. When we feed images into the neural network, they always need to be normalized so that the values for each pixel are between zero and one. The vgg model has a built in normalization function called pre-process input that will do that. We just need to call it and pass in our data. So, we'll say vgg16.preprocessinput, and we'll pass in our data, which is x. Alright, let's go to line 21. Now, we're ready to run the normalized data through the neural network and make a prediction. We can do that by calling model.predict, model.predict, and passing in our data x. The predictions we get back will be the 1,000 element array of floating point numbers. Each element in the array will tell us how likely our picture contains each of the 1,000 objects the model was training to recognize. To make things easier, the vgg model provides a decode predictions function that will tell us the names of the most likely matches. So, let's go to line 24, and here we can call vgg16.decode predictions. And, then we just pass in the predictions object we created above. By default, it will give us the top five most likely matches. I'm gonna add the parameter call top equals, and pass in nine to get the top nine matches. Now, we'll just loop through the predictions and print out each result. Alright, we're ready to run the code. It's worth mentioning that the first time we run this code,

Scanned with CamScanner

Keras will connect to the internet and download the latest version of the vgg16 model. This means that you'll need internet access to run it, and around 100 megabytes of data will be downloaded. Alright, now right click and run. And, we'll get our predictions. Alright, so the top prediction is seashore, with nearly a 40% likelihood. I think that's a pretty good match for our image. Some of the other matches are promontory, lakeside, breakwater, sandbar, and cliff. Those are all reasonable guesses for the picture. If you want, go ahead and try this again, using your own images. It's fun to see what predictions it will make and what kinds of pictures will confuse it.

Transfer learning as an alternative to training a new neural network

If you have a lot of training data, you can train a convolutional neural network to recognize objects and images, but here's a secret, in the real world, you almost never need to train the neural network from scratch. Instead, we can use transfer learning to reuse an existing neural network and adapt it to a new problem. Transfer learning is where you take a model trained on one set of data and then use the knowledge it learned to give it a headstart when solving a new problem. To understand how transfer learning works, let's take a look at how a convolutional neural network processes an image layer by layer. A typical convolutional neural network is structured like this. The network is made up of a series of convolutional layers and the training process teaches each of those layers to be activated when it sees certain patterns in the input image. Those layers learn to tell images apart by looking for those unique patterns. Let's look at the patterns that would activate each layer. These are the kinds of patterns that the first layer is looking for. You can see that it's looking for really basic patterns like splotches of color and lines in an image. Now let's go down to the second convolutional layer. The patterns are starting to look slightly more complex. Here are the images that would activate the third layer. These patterns are

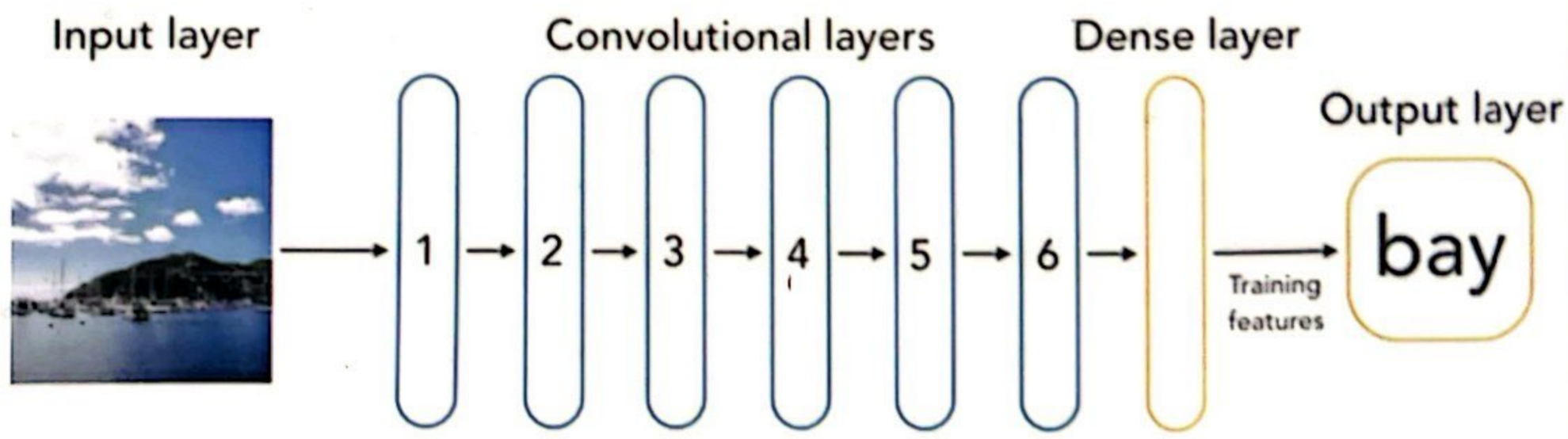
Scanned with CamScanner

starting to resemble objects. The one on the right sort of looks like a hair pattern. Here are some images that would activate the fourth layer. The patterns are even more complex. The middle image almost looks like eyeballs. Let's look at one more. Here are the images that would activate the fifth layer. Now the patterns are getting very complex with curves and shapes. Since neural networks are really just mathematical models, it's impossible to tell exactly what each of these patterns represents, but you can see how each layer is getting more and more complex in what it's looking for. The basic idea is that neural networks learn to detect simple patterns in the top layer, and then the next layer uses that information to look for slightly more complex patterns and so on, down through all the convolutional layers. But the final layer of the neural network is a densely connected layer that uses the information from the convolutional layers to decide which object is in the image. With transfer learning, we're gonna start with a neural network that's already been trained to recognize objects from a large dataset like ImageNet. To reuse this neural network with new data, we can simply slice off the last layer. We'll keep all the layers that detect patterns, but remove the part that maps those patterns to specific objects. We'll call this pre-trained neural network a feature extractor because we're using it to extract training features from images. Next, we'll create a new neural network to replace the last layer in the original network. This is the only part that we'll have to train ourselves. When we build our new image recognition system, we'll pass our new training images through the feature extractor and save the results for each training image to a file. Then, we'll use those extracted features to train the new neural network. Since we're using the feature extractor to recognize shapes and patterns, our new neural network only has to learn to tell which patterns map to which objects. Since this new neural network isn't doing much work, it can learn to do it with a small amount of training data. And here's how we'll do predictions with transfer learning. When we wanna test the new image, we have to first pass it through the same feature extractor. Then we can use those extracted features as input to our newly-trained neural network, which will give us the final prediction.

Scanned with CamScanner

Transfer learning can cut the time required to build an image recognition system from days to minutes. When I'm starting new projects, I almost always try transfer learning first before training a new neural network from scratch. If transfer learning works, there's no need to do the extra work to train a new model. Transfer learning is also really useful when you only have a small training dataset available. Training a neural network from scratch is sort of like teaching a baby to read. The baby has to learn about letters and words and sentences before it can read and understand anything. Transfer learning is more like asking an adult that already knows how to read to learn something new. Since the adult already knows how to read, they need less material to learn a new topic. They don't need alphabet flashcards and spelling tests. The same basic idea applies to neural networks. If you only have a few hundred training images for your image recognition system, you don't have enough data to teach your model from scratch, so it makes sense to start with a model trained for something else and adapt it to your problem.

Transfer Learning



Scanned with CamScanner

Extracting features with a pre-trained neural network

Let's use transfer learning to build an image recognition system that can identify pictures of dogs. The first step is to build a feature extractor that can extract training features from our images. Let's get started. First, we need some training data. I've included some along with the example code. Let's take a look here in the training data folder. First, I have a sub-folder called dogs. These pictures are 64 by 64 pixel images from the image net dataset. If you're building your own image recognition system, you can use your own pictures of whatever kind of objects you wanna recognize instead. Next, we have a folder called "not dogs." These are various pictures of anything that's not a dog. It's important that these pictures are as varied as possible, so that the model can learn the difference between dogs and other types of objects. Alright, let's take a look at the code. Open up "04_feature_extraction.py". We're gonna write the code that will use the pretrained model to extract features from our training images and save those features to a file. Here, starting on line eight, I've already written a code to load the list of images in each folder. Then, on line 11, we'll create an empty array to hold the list of images. When we process the images, we need to remember which images were dogs and which ones were not dogs. So, on line 12, we'll create another array called labels. Each time we load an image and put in the images array, we'll also add either a one or zero to the labels array. If the image is a dog we'll add a one, and if the image is not a dog we'll add a zero. Then, on line 15, we're loop through all the files in the "not dogs" folder and process each one. On line 17 we load the image using Keras' load image helper function. This will load the image file's contents to the memory. Then, on line 20, we convert the image data into an array using the img to array function, and on line 23, we add that to our list of images. On line 26 we add zero to the labels array, since we know that this image is not a dog. Starting on line 29 we'll do exactly the same thing, but this time for the dog images. The only difference is that on line 40 we add the one to the

Scanned with CamScanner

labels array, because you know each image is a picture of a dog. At this point we have one list with all the images and a matching list in the same order with the labels for each image. Now we're ready to create our training data array. On line 43 we'll create an array called x_train that will have all of our training images. Keras expects all of our training images to be a numpy array instead of a normal Python list. To convert the Python to a numpy array, we use the numpy array function. So, we just say, "np.array" then we pass in the images list. And then on line 46 we'll do exactly the same thing for the labels. So we'll say, "np.array" and we pass in the labels. To extract features we'll use the vgg16 model pretrained on the image net dataset. This model's included with Keras. First, we need to normalize our training dataset so all the pixel values are in the zero to one range. I've done that here on line 49 using the vgg16 preprocess_input function. Now, on line 52, we're ready to create the neural network itself. We'll do that by creating a new vgg16 object. So, in lower case we'll say, "vgg16 dot" and then upper case, "VGG16" to create a new object. But we also need to pass in a few options. First, we wanna tell Keras that we wanna load the version of the neural network that was pretrained on the image net dataset. We can do that by passing in "weights=" and then the string, "imagenet". And since we're only using this neural network for feature extraction, we wanna chop off the last layer of the neural network. Since this is such a common thing to do, Keras provides a flag to tell it we wanna do that called "include_top=False", so we'll pass in the perimeter, "include_top=False". In Keras terminology, the top is the last layer of the neural network, so by saying "include_top=False" we're saying we want the neural network without the last layer attached. Finally, we need to tell it what size images we're using as training data. Our training images are 64 pixels by 64 pixels with three color channels, one for red, one for green, and one for blue. So, we'll pass in an input shape, say "input_shape=" and then in the array, we're pass in "64,64,3". We're using small image sizes in this example to keep the training time as quick as possible, but when you're building your own image recognition systems, you can use larger sized images like 224 pixels by 224 pixels. To do that, you can just bump up the size here. Alright,

Scanned with CamScanner

now we wanna feed all of our training images through the neural network and capture the results. To do that on line 55, we just call the predict function on our neural network, and pass in an array with all of our training data. So, we'll say pretrained neural network, nn, dot predict, and we'll pass in the "x_train" variable. The features x array will now contain the set of features that represent each of the training images in our dataset. The last step is to save these features to disk. We can do that with a library called joblib. It has a convenient function called dump for writing an array to disc. I've already done that for the features on line 58 and for the labels on line 61. Alright, let's run the program. Right click and choose run. It will take a few seconds to load all the images and run them through our pretrained feature extractor. When this finishes it will write up two files, x_train.dat and y_train.dat. These files contain the features and labels that represent our training data. We'll use these features to train a new neural network in the next section.

Training a new neural network with extracted features

We've used the pre trained neural network to extract features from our training images. Now we're ready to train a new neural network that uses those extracted features. Let's open up 05 training with extracted features.py. This is the code to train a simple neural network. The code is exactly like training any other neural network but with two small differences. The first difference is how we load our training data. Instead of loading raw images to train with, we're gonna load the features that we extracted with the pre trained VGG 16 neural network. If you look at the file list on the left, you can see that we already have our extracted features stored in a file called x train.dat and our labels stored in a file called y train.dat. Back here on line seven and eight I've already loaded those two files. Next, starting on line 13, we have the code to define our neural network. The second difference is in how we define our layers. Since we use VGG 16 to extract features from our image, this neural network has

Scanned with CamScanner

no convolutional layers. Instead it only has the final dense layers of the neural network. These are the only layers that we'll be retraining. Next, we'll compile the model in line 19 the same way as normal. And then on line 16 we'll call model.fit to train the model. And then, finally, at the bottom, starting on line 34, we'll save the train model and its weights to files. Let's run the code then train the neural network. Right click and choose run. And notice how fast the training completed. That took a tiny fraction of the time it would take to train a neural network from scratch. You can see in the file list on the left that our train model is now saved in two files, model structure.json, and model weights.h5. In the next video, we'll use our transfer learning model to make predictions with real images.

Making predictions with transfer learning

We've used transfer learning to create and train a neural network that can recognize pictures of dogs. Let's see how they use that neural network to make predictions. Open up 06_making_predictions.py. This code is exactly the same as the code we'd use to make predictions with the standard neural network. There's just one key change. We'll need the pre processor image with the vgg16 feature extractor. First, we can see on line eight that we're loading the structure of the neural network. And then on line 15, we're loading the trained weights. Here on line 18, we're loading an image that we want to test. We'll try out an image called dog.png. Let's check it out. Yep it's a picture of a dog. Then on line 21, we're converting the image to an array and on line 24 we're turning it to a four dimensional array so that we can feed it in that karis. So far, all the code is exactly as it would be for any neural network. But here's the key difference. Since our neural network was trained using features extracted from a pre trained neural network, we need to follow the same procedure for extracting features for any image that we want to test. So here on line 30, we need to create an instance of our pre trained neural network. This should be exactly like the one

Scanned with CamScanner

we used to generate our training data. So we'll use the same code here. So we'll say feature_extraction_model = vgg16. Then upper case VGG16 the creepy object. And we'll pass on the same options. We'll say weights = in a stream image net. We'll pass an include_top = false. And finally we need to set the input shape. So we'll say input_shape = and then the ray 64, 64, 30. Now we need to run our image through that pre trained neural network to extract the features that will feed into the second neural network. We can do this by just calling the predict function on the model and saving the result. So we'll just say feature_extraction_model.predict and then we'll pass on the images. Great, now that we have the extracted features we can pass those in to our second neural networks predict function to get it's final prediction for this image. So let's go down to line 34 and we'll say results = model.predict and then we'll pass on those features we just created. The rest of the code is exactly the same as using any other neural network. On line 37, we just grab the first result. And then on line 40, we just print out the results. Let's run the code and see what it predicts for this image. So right click and choose run. And it says a picture of a dog is in fact a picture of a dog with 100 percent confidence. Let's try another picture. Go back up here to line 18 and instead of dog.png we have another picture called notdog.png. Let's check that one out. Yup that's not a dog. Let's close that and let's run the code again and see what prediction we get for this image. Right click and choose run. And it correctly predicted that this image is not a dog. Feel free to try this out with your own images. But keep in mind that our training data is fairly small. So the accuracy may vary. But what we just demonstrated here is really powerful. With only a few training images, we built a program that can tell pictures of dogs apart from pictures that aren't dogs. Only a few years ago this was science fiction. And since we used transfer learning to do it, we're able to train the model in just seconds. Transfer learning is a very powerful technique. Try it out on your own programs and see if you can build a new object detection model yourself.

Scanned with CamScanner

When to use an API instead of building your own solution

Selecting transcript lines in this section will navigate to timestamp in the video

- [Instructor] Depending on the kind of project that you're building, sometimes it makes more sense to use an off-the-shelf image recognition API instead of building your own custom solution. All the major cloud vendors now provide image recognition APIs. So if you're using cloud services from Google, Amazon, or Microsoft, you can also use their image recognition capabilities. In addition to the products offered by the large cloud vendors, there are also many start-ups and smaller companies that offer image recognition APIs. All of these products will let you upload an image and get back a list of objects that appear in that image. And best of all, using these APIs, usually only requires a few lines of code. The downside, is that these systems have a built in list of objects that they recognize, so you're limited to recognizing the kinds of objects that they already understand. So when might you choose to use an API instead of training your own machine learning model? First, if you don't have any training data, you might not have any other choice. The APIs have their own built-in image recognition models that are already pre-trained on many millions of images, so you don't need to do any training yourself. Next, if you need to detect many different kinds of objects in your application, it's often easier to use an API. Google's Cloud Vision API can detect thousands of different objects, because they have access to a nearly unlimited amount of training data. It would be very difficult to train your own model at that scale. Along those same lines, if you only need to detect common types of objects, likes cars or buildings or animals, it might be easier to use an API. These systems are pre-trained to recognize common types of objects. Sometimes they can even give you very granular results, by telling you the specific breed of dog if a dog is detected. Most importantly, these APIs are quick and easy to use. So if you don't have the time or money to build your own solution, you can easily test out an API and see if it's good enough for your project. But there are also times

Scanned with CamScanner

when using an image recognition API just won't work for your project. If you're a position where you have access to specialized training data, that isn't available to a company like Microsoft or Google, it might be worth building your own model. This is also true if you're trying to detect something very specialized, that might only apply to your industry. You're not likely to find an off-the-shelf solution that works in very specialized cases. There are also times when the training data is just too sensitive to share with anyone else. For example, many medical applications train their own models, because they can't share the underlying patient data that's used to train the model. There's also cases where the training data might be a trade secret. But sometimes it makes sense to combine your own model with an off-the-shelf model. In addition to basic image recognition, all of the cloud services off their own special features. For example, Google Cloud Vision can detect the logos of well known companies and they can detect famous landmarks in photographs. In some cases, you might use those features in cooperation with your own custom model to solve a larger problem. For example, you can build you own model to recognize different types of clothing, and then use Google's API to recognize which logos appear on that clothing. Another special case is if you need optical character recognition, or OCR. That's where you wanna pull all the text out of a photograph. It's very difficult to build a high quality OCR system. If you need this capability, I recommend just using Google's API for this. You can use the API to extract text from an image, while still building your own models to do everything else. So which vendor has the best API? There's no simple answer. All of the vendors are constantly improving their systems with more training data and adding new features. Depending on the type of images you're working with, one vendor might work better than another. I recommend trying out a few different vendors and seeing what works best for you. You can also take into consideration what extra features the vendor offers, like logo recognition or OCR. For example, Google's particularly good at OCR. And of course, you can always use APIs from multiple vendors. If no one company offers all the features you need, you can combine them and use more than one.

Scanned with CamScanner

Introduction to the Google Cloud Vision API

In this section we'll be using the Google Cloud Vision API for object recognition and text extraction. Let's take a look at what Google Cloud Vision offers. The best part about the Cloud Vision API is that you don't have to do any training yourself. You upload an image and it gives you back results from it's pre-train model. So it's very quick to get started. Also the pricing model is simple. You pay per 1,000 API requests and the prices are fairly inexpensive. The catch is that each type of detection in an image counts as one API call. So if you ask for a list of objects that appear in an image and the text that appears in the same image that actually counts as two separate API calls. All the processing happens on Google servers in the Cloud so you don't need any specialized hardware. You just upload your images to Google and get back the results. Let's take a look at Google's demo and see what kinds of data the API can extract. Open up your web browser and go to cloud.google.com/vision. If you scroll down this webpage you'll see that Google offers a demo. Here I have an example of a road sign. Let's drag and drop this image on the webpage and see what Google can detect. On the first tab are the labels of the objects that it detected in the image. We can see that the top results are road, infrastructure, and traffic sign, which makes sense. The other results look good too, like sky and signage. The next tab is web entities. One of the neat things Google can do is look for webpages that had similar images and give you back results based on those pages. So here it even guessed that the sign is from the Minnesota Department of Transportation. The next tab, Document, is where it shows all the text that it was able to extract. We can see that it was able to read the word road. It also looks like it detected the other words so it's possible that we'll get more text back when we use the API. You can also get back some document properties like dominant colors. And here on the Safe Search tab it shows if the image contains sensitive content like violence or nudity. There are other things that the API can do too that aren't represented here. For example it can detect faces in the image and tell you the emotion of each person's face.

Scanned with CamScanner

Overall this is a powerful API, but keep in mind that each tab here represents a different call to the API. So if you want all this information about your image this would actually count as six separate calls to the API.

Setting up Google Cloud Vision account credentials

Before you can use the Google Cloud Vision API, you need to set up your Google account. Then, we'll need to download the credentials file that will give our code access to the Cloud Vision API. In your web browser, go to console.cloud.google.com To access this page, you'll need to login with your Google account. If you already have a Gmail account, you can use that email address and password. If you don't have a Google account, you'll be given the option to create a new one. After logging in, you'll see the Google Cloud Platform dashboard. From here, you can access all the different Google Cloud services. If this is your very first time using Google Cloud, you'll see a button to create a new project. If that's the case, just create a project called Image Recognition Class, and wait until it finishes before you continue. Since the Google Cloud Platform has so many features, not all of them are enabled by default. We need to enable the Google Cloud Vision API before we can use it. To do that, go to the menu, and go to API's and Services, and then click on Library. This is a searchable list of all the Google services that you can enable. Type in Cloud Vision and hit Enter. And then here, click on Cloud Vision. If it's not already enabled, choose to enable it for your project, and wait until it completes before you move on. Here I've already got it enabled. Finally, you need to provide billing details for your account before you can use some of the features of Google Cloud. To do that, go to the menu, and go to Billing. Be aware that you're responsible for any charges that you incur on Google Cloud. In order for our code to be able to call the Cloud Vision API, we also need a credentials file. This file keeps our account safe from unauthorized use. To do that, we're gonna go back to the menu, go to API's and Services, and then click on Credentials. On this screen, click on Create Credentials, and

Scanned with CamScanner

then choose Service Account Key. For the Service Account, on the dropdown, choose New Service Account, and then give the name. We can call it Image Recognition Class. For the role, just choose Project and Viewer. That should be the only permission that you need. And then when you're done, click Create, and then we'll create the file and download it. Alright, the file's downloaded, I'm gonna go to my downloads folder, which I've opened here, and I'm gonna rename this file to Credentials.json So I'm gonna right-click, and rename, change it to Credentials.json Now, I'm gonna copy this file into PyCharm. So I'm gonna right-click, choose Copy, and I'm gonna go to PyCharm, and then I'm gonna paste it in 06. So right-click and Paste. And, OK. Alright, that's everything we need, we're ready to get started

Recognizing objects in photographs with Google Cloud Vision

Alright, let's use the Cloud Vision API for image recognition. Before going any further, make sure you've created the Google Cloud account and downloaded the credentials file. If you aren't sure how to do that, you can review the previous video. Alright, let's open up cloud_image_recognition.py. This file uses the Google Cloud API to upload a file and get back a response from Google with a list of objects detected in the image. On line seven, you can put in the name of the image file that you wanna check. I've included the sample image to test with, so you can leave this as road_sign.jpg for now. Let's take a look at the picture. This is a picture of a road sign on the highway. If the API works correctly, it should come up with labels like road and sign. Alright, let's go back to the code. On line eight is the name of the credentials file that we wanna use to access the Cloud Vision API. You should already have a credentials.json file. If not, you can review the previous video. On line 11 we read the credentials file into memory and then, on line 12, we create an instance of the Google API client. Since we wanna use the Vision API service, we pass in the string vision and v1. We also need to pass in the credentials file in the same line. On line 15 we load the image file

Scanned with CamScanner

from disc and convert it to a base 64 encoded version. Google's API requires the images be uploaded in base 64 format. The rest of the code here is the minimum code needed to make requests to the Google Cloud Vision API. First, on line 20, we create an object that represents the batch request that we're making to Google. We're required to pass in the image data to check, and then the features that we want back. In this case, we want a list of labels of what appears in the image so we'll pass in LABEL_DETECTION. Notice that the batch request object is an array. In this case, we're only asking it to annotate one image, but you can pass in more that one image in a single request if you want. Then on line 30 we create a Python request object using the Google API library. Here we're asking it to access the image's API and then annotate the images according to our batch request that we defined above. Then on line 33 we connect to Google and execute the request. The results will be stored in the response object. On line 36 we check for errors, and on line 40 we get back the results. Then finally on lines 42 and 43 we print out the results. Let's run the code and see what happens. Right-click, choose Run, and here's what we got back. So Google says our image is a road, infrastructure, traffic sign, sky, signage, these all look like great labels for our image. But notice that the percentages don't add up to 100%. Unlike the custom model we built earlier in the course, Google's model can detect multiple separate objects in the same image. Since it's not just classifying the entire image as one type of object, you'll get many different labels back representing separate detections. From here, you could save these labels to a database, or use the labels to make decisions with how to process the image. Google's done the hard work and now it's up to you to decide how you wanna use this data in your program.

Extracting text from images with Google Cloud Vision

- Let's try using the cloud vision API to extract texts from an image. Before going any further, make sure you've created the Google cloud

Scanned with CamScanner

account and downloaded the credentials file. If you aren't sure how to do that you can review the previous video. Alright, let's open up cloud text recognition.py. This file uses the Google cloud vision API to upload an image and get back a response with all the texts that Google was able to extract from the image. On line seven, you can put in the name of the image that you want to check. I put in text.pmg. Let's take a look at that image. This is a scan of a page of dialogue from a play by Shakespeare. If the API works correctly, we should get back all this text as plain text. Alright, let's go back to the code. On line eight is the name of the credentials file that we want to use to access the cloud vision API. You should already have this file. If not, you can review the previous video. On line 11, we'll read this credentials file into an object and on line 12 we'll create an instance of the Google API client. Since we want to use the visions service, we'll pass in vision and v1. We'll also pass in the credentials object that we just loaded above. Now on line 15, we'll load the image file from disc and convert it to bay 64. Google's API requires the images are uploaded in bay 64 format. The rest of the code here is the minimum code needed to make requests to the Google cloud vision API. If you've already played with the object detection API, the code here is almost identical. First on line 20, we create an object that represents the batch request that we're making to Google. We're gonna pass in the image date of the check. And then the features that we want back from the API. Since we want Google to extract texts from our image, we'll pass in text detection. On line 30, we create the python request object using the Google API library. Here, we're asking it to access the images API and annotate the images according to the batch request that we defined above. Then on line 33, we connect the API and execute the request. The results will be stored in the response object. And then on line 36, we make sure there weren't any errors. When Google returns the result, it will return a nest of the array with a key called text annotations. So on line 40, we pull out this result. The result from the API, we'll be an array with one entry for each piece of detected text. In this case, our image only has one text block. So on line 43, we'll just grab the first result. And then on line 44, we'll pull out the description key that

Scanned with CamScanner

includes the actual text that was detected. Now, let's go to line 47. There's a bounding poly key that lists the pixel coordinates in the image where the text was found. Let's print this out as well. Alright, let's run the code and see what happens. Right click. Choose run. And great, it looks like it pulled all the text out correctly. It even got the capitals and the lowercase correct. At the bottom, we can also see where in the image it found the text. These pixel locations are really helpful in a lot of applications. For example, we can use this API to scan business cards and the coordinates tell us where each piece of text appeared and how big it was. That can help us figure out which piece of text represented which piece of data on the business card. This image is pretty simple so the accuracy was very good. In more difficult images the accuracy might not be quite as good. But overall, this API is among the most accurate OCR tools I've seen.

Scanned with CamScanner