Module 22: Deep Neural Networks (Part 2)

Quick Reference Guide

Learning Outcomes:

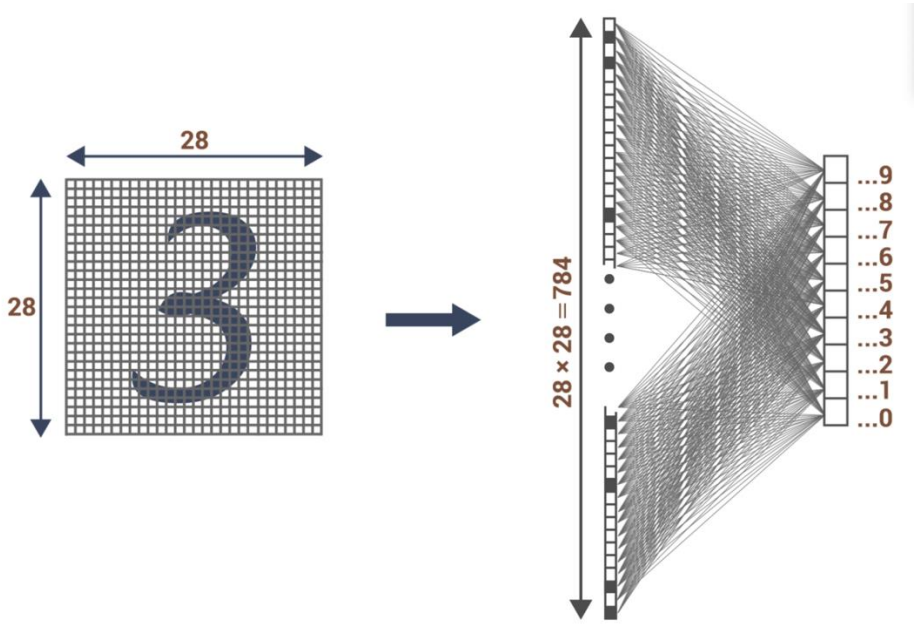
1. Articulate the basics of a CNN and its different layers.
2. Process images in preparation for training a neural network.
3. Train a neural network to recognize items and images.
4. Use a pretrained network to enhance model performance.
5. Interpret a neural network model.
6. Compare an LSTM network to time series analysis.
7. Train an LSTM network for time series analysis.
8. Apply neural network techniques to a linear regression problem.
9. Apply AI/ML techniques to a specific field you are interested in.

Video 1: Introduction to Convolutional Neural Networks (CNNs)

Convolutional neural networks, or CNNs, took the machine learning world by storm when they became dominant in image classification competitions, starting in 2012. But before diving into CNNs, explore how you can approach image classification using only **dense layers**. This actually works pretty well for small and simple images, such as pictures of handwritten digits. Here is how you would proceed.

1. First, you would flatten the image, meaning that you would break it up into individual pixels and arrange those pixels into an **array**.
2. Then you would put the array through one or more dense layers that would create connections between all of the pixels.

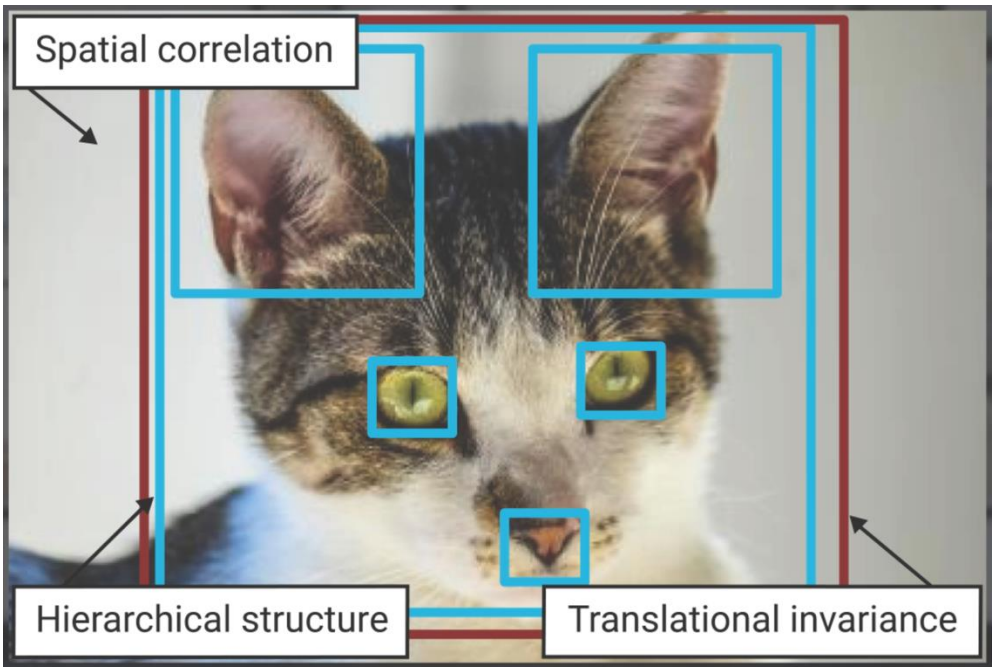
3. Finally, a **softmax** layer would summarize a result into probabilities for the various output classes.



The main problem with this approach is that it begins by destroying all of the spatial information that is key to recognizing objects and images. That information must then be relearned in the training process.

Natural images have certain regularities. First, they are **spatially correlated**, meaning that if all of the neighbors of a particular pixel are white, then it is likely that the pixel itself is also white. Objects in natural images are built up in a **hierarchical structure**. The face of a cat, for example, is characterized as having ears, eyes, and a nose of a certain shape, and in a particular arrangement. The eyes themselves are made up of smaller parts, such as circles and lines, and these parts can be reused in other features, like the nose and the ears. The objects in natural images are also **translationally invariant.** This means that the photo can be classified as containing a cat,

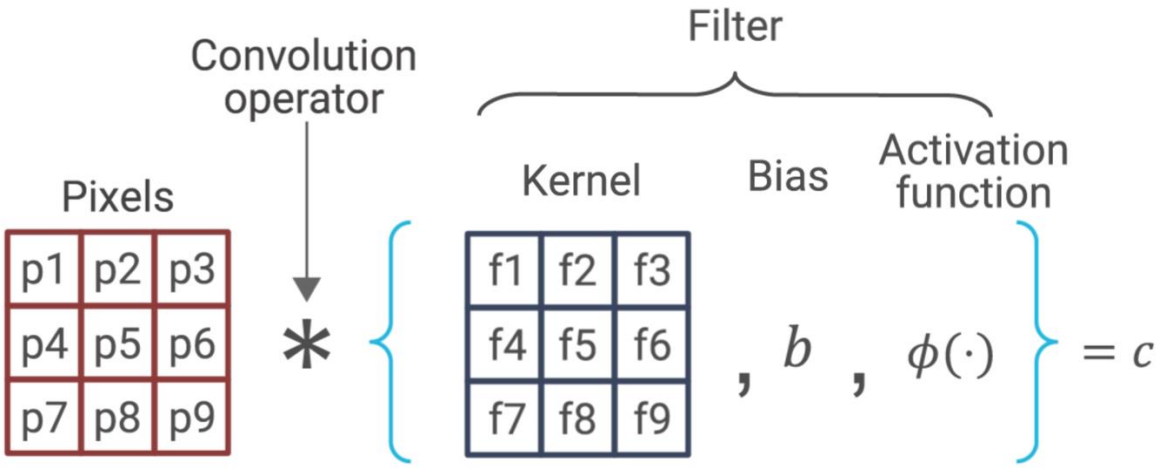
regardless of whether the cat is at the bottom, the middle, or the top of the image.



Dense layers do not capture any of these regularities. One could do much better by designing a neural network in a way that leverages these special characteristics of natural images. This is exactly what convolutional networks do.

The core operation of a CNN is the **convolution**. You have already encountered 1D convolutions in trended time series data. The convolutions here are a little more complicated, as they involve several components: a **kernel**, a **bias**, and an **activation function**, which together constitute a **filter**.

The kernel is a small grid of numbers. In this example, the kernel is a 2D matrix.

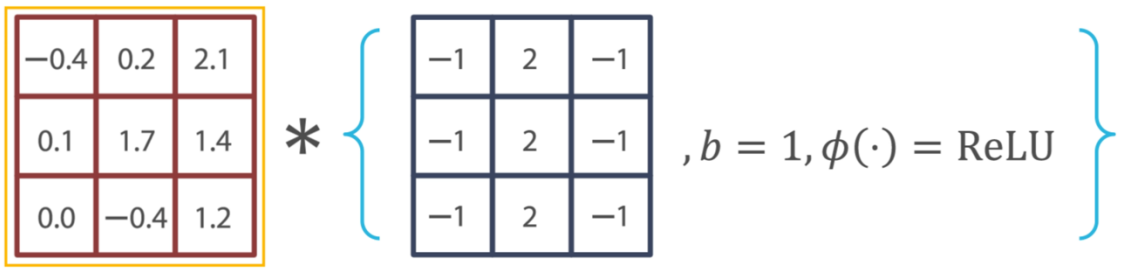


A 2D convolution can be done on a patch of pixels that is of the same size as the kernel. Here you have a 3 × 3 patch of pixels. And taking the convolution on that patch with the filter produces a single number, c.

$$c = \phi\left(b + \sum_{i=1}^{9} p_i f_i\right)$$

This number is computed by first multiplying each pixel with its corresponding weight in the kernel, then adding up all of those products, then adding in the bias term, and finally applying the activation function to the result.

Now try an example.

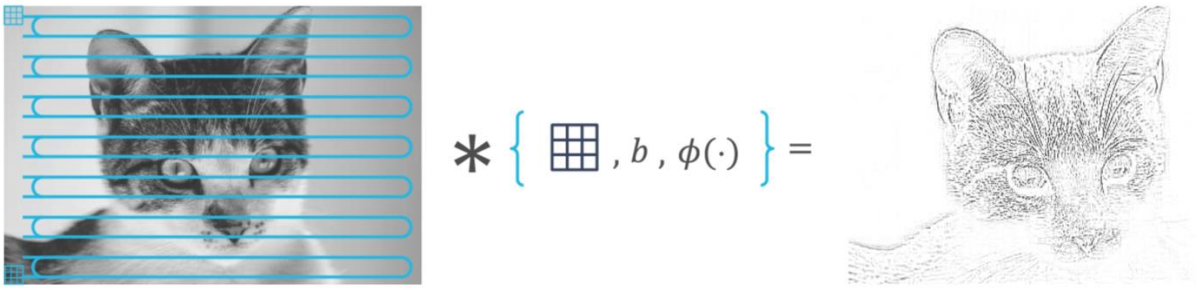


You will convolve a 3 × 3 pixel matrix with the filter consisting of a 3 × 3 kernel, a bias value of 1, and a ReLU activation function. This equals the activation function evaluated on the bias, plus the sum of the pairwise products of the two matrices.

$$
\begin{aligned}
&= \phi\left(b + \sum_i p_i f_i\right) \\
&= \mathrm{ReLU}(1 + (-0.4)(-1) + (0.2)(2) + \cdots) \\
&= \mathrm{ReLU}(-2.2) \\
&= 0
\end{aligned}
$$

That all adds up to −2.2, a negative number. So the ReLU function discards it and the result is 0. That is the convolution for a small part of an image.

To process an entire picture, you perform the convolution on every patch by sweeping the filter over the whole image. The output of this is a new image with certain attributes highlighted, depending on the weights in the filter.



Image

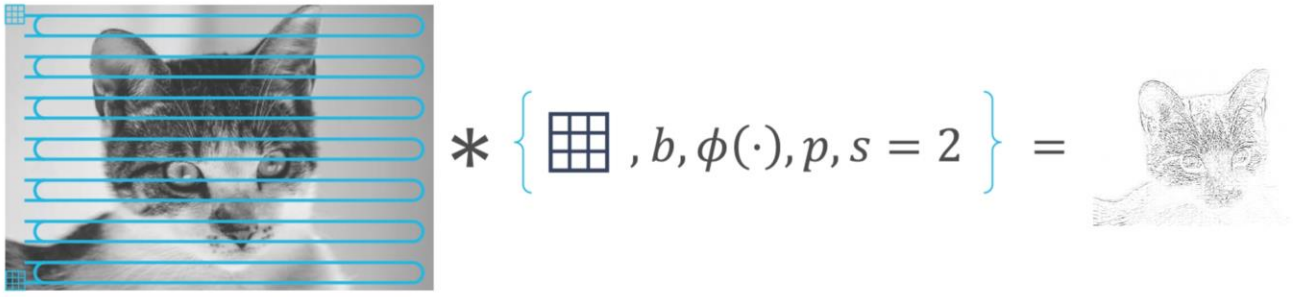
For example, this filter highlights vertical lines. From now on, you will stop calling the output an image and instead call it a feature map. That is because it indicates the locations in the image where the feature represented by the filter can be found. There are two additional parameters related to the sweeping step.

First, as you can see here, the kernels are not allowed to leave the boundaries of the image. As a result, the output will have fewer columns and rows than the input. To prevent this, one can add an outer padding of zeroes that is precisely thick enough, so that the size of the output equals the size of the input.

$$\{ \boxplus, b, \phi(\cdot), p \}$$

In Keras, this is controlled with the **padding** parameter, which can be set to valid if no padding is desired, or same if you want to include the padding.

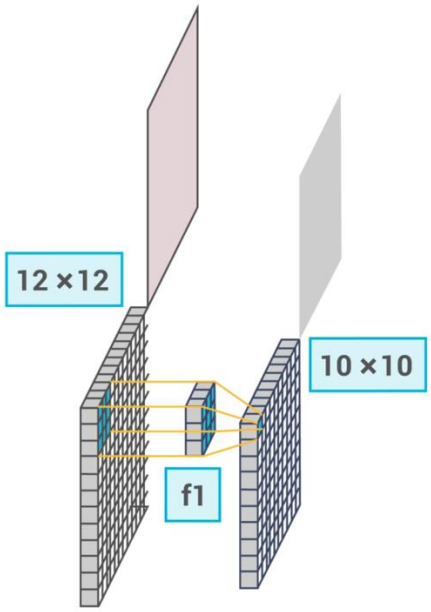
Another important parameter of the filter is the **stride**.



You can also have the convolution advance in strides that are larger than a single pixel. So a stride value of 2, for example, will only calculate every

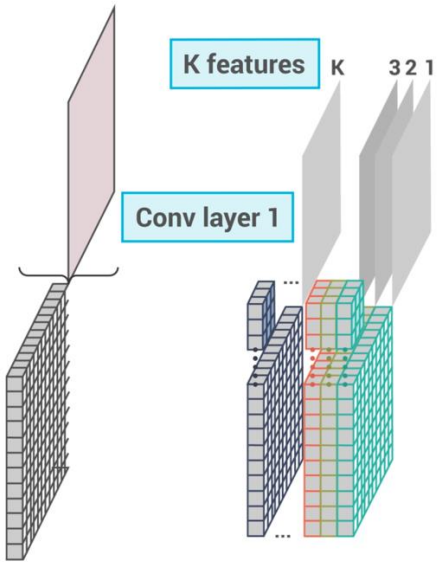
other convolution. This will reduce the height and the width of the output by a factor of 2, which means that the number of pixels in the output will be reduced by a factor of 4. The stride, padding, and activation function are set by the modeler when creating the filter. The kernel weights and bias value are learned through training.

Here you see the convolution of an image with a filter in a 3D perspective.



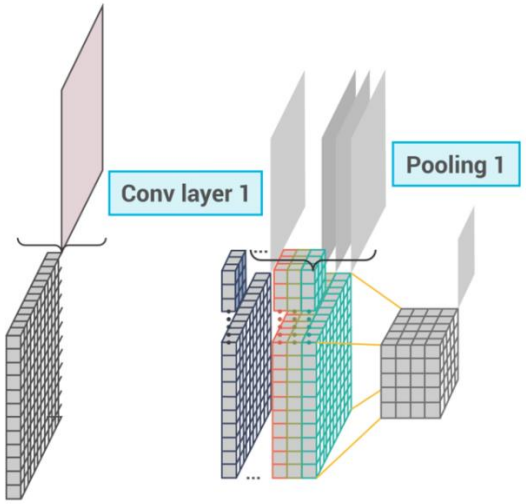
The wall of blocks on the left represents the image. Again, you are assuming that it is in grayscale, so each block in the wall is a single pixel and contains one number. To the right of the image, you see filter number 1. The convolution of the input image with filter 1 produces the feature map on the right. This is a 10 × 10 matrix because the filter has been swept with a stride length of 1 and no padding.

You can repeat this procedure K times for K different filters, and thus generate K different feature maps. Together these constitute a single **convolution layer**.

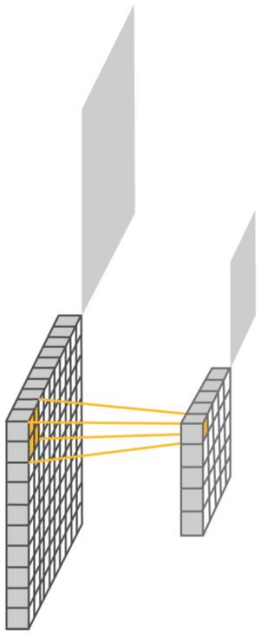


All of the filters in a convolution layer will share the same kernel size, activation function, stride, and padding parameters. This is useful for computational reasons and it also greatly simplifies the design of the convolution layer. In Keras, you define a convolution layer by specifying the number of filters, the size of the kernel, the stride length, whether or not to pad the border, and the activation function.

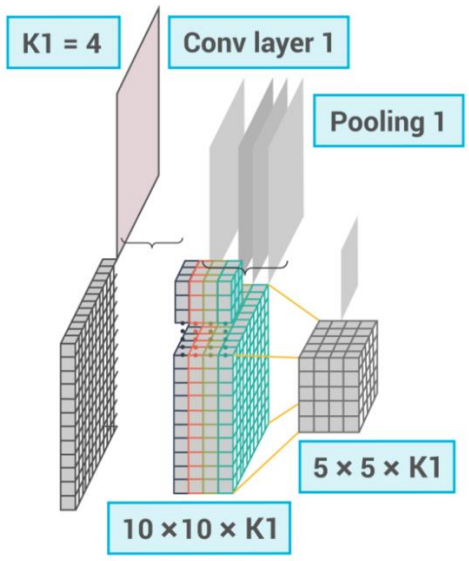
Convolution layers are often followed by a **pooling layer**, which compresses the information down to a smaller size.



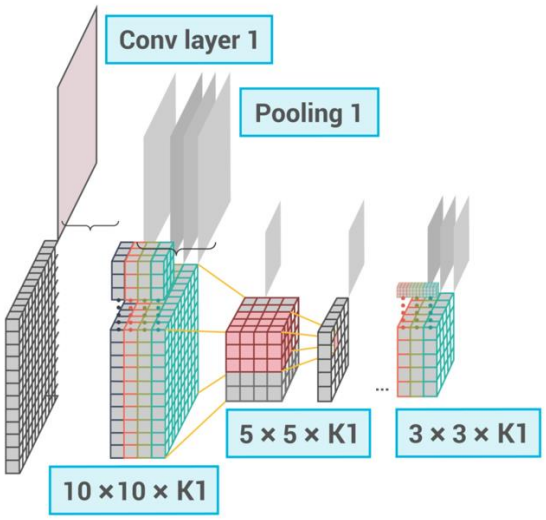
And the most common way of pooling is with **max pooling**. The max pooling operation takes a feature map as input and produces another feature map of reduced size. It does this by tiling the input and outputting the largest value from each tile.



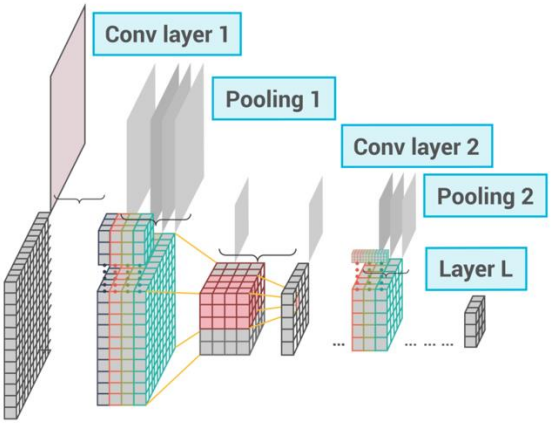
You can think of this operation as zooming out of the picture. The convolution layers are meant to identify features at one zoom level, then the max pooling layer zooms out to a new level, where the features from the previous level combine to find new composite features. This is how neural networks capture the hierarchical structure of natural images. Having pooled the first layer, you are left with a reduced block of features.



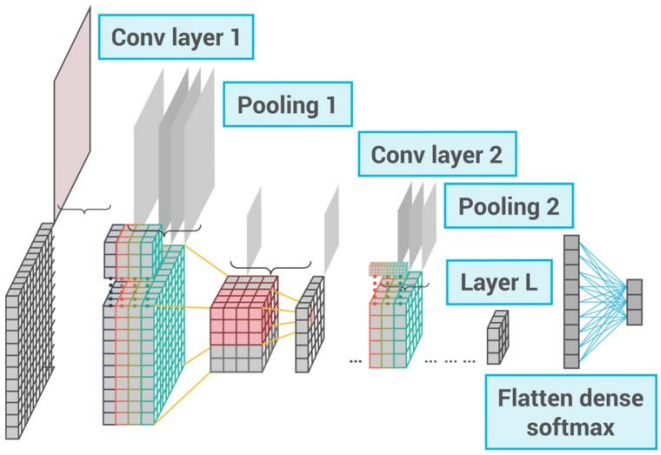
The next step is to apply another convolution layer to this block. But since this is now a 3D block of data, you will apply a 3D kernel.



In this picture, you see a 3 × 3 × 5 filter with a total of 45 tunable weights. These combine to aggregate features from the previous layers and produce a single feature map in the next downstream layer. You repeat this for several different feature blocks to form a new convolution layer.



You can pool this layer and continue in this way, building features and zooming out as many times as necessary. And after doing this a few times, you are left with a very small set of numbers that capture the highest-level features that are needed to identify the objects in the image. And then the final step is to flatten and softmax this final set of numbers.

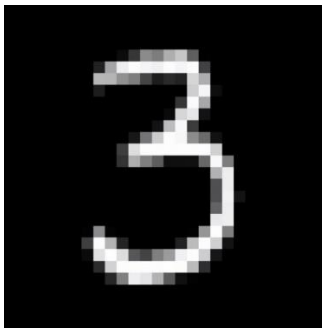


You can think of the sequence prior to the final dense layer as a process for learning highly nonlinear features that you then use as inputs to a multinomial logistic regression at the end. At no point did you explicitly have to tell the neural network what features to look for. The kernels were

molded into useful features by the training process itself. However, doing this requires lots of data and a lot of computation.

Video 2: Neural Networks for Vision (Part 1)

Now you can explore how to create **convolutional neural networks** in Keras. For this demo, you will be using the MNIST dataset, which contains 28 × 28 images of handwritten Arabic numerals. For example, here is one image from this dataset, a handwritten number three.



You are going to try to build a convolutional neural network that can recognize 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9. So a dense neural network can achieve on the order of 97% test set accuracy on the raw pixel data. But you will see that a convolutional neural network can do better.

First review how you can create a **dense network** to recognize MNIST digits. There are two different ways to create this model. The first is using the simpler **sequential API** from Keras.

```
model = keras.Sequential([
        layers.Dense(64, activation="relu"),
        layers.Dense(10, activation="softmax")
])
```

The second is using the more complex **functional API**.

```
inputs = keras.Input()
features = layers.Dense(64, activation="relu")(inputs)
outputs = layers.Dense(10, activation="softmax")(features)
model = keras.Model(inputs=inputs, outputs=outputs)
```

Both models do the exact same thing, but it is good to see both. Now after creating this model, you compile and fit it as usual.

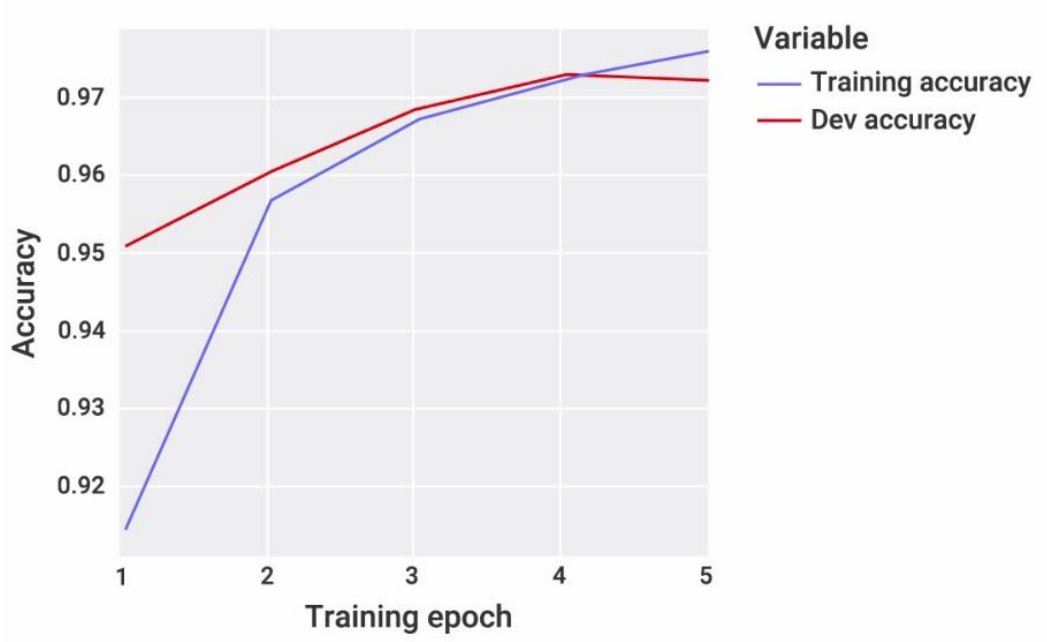
```
model.compile(optimizer="rmsprop",
              loss="sparse_categorical_crossentropy",
              metrics=["accuracy"])


model.fit(train_images_1D,
          train_labels,
          epochs=5,
          validation_data = (dev_images_1D, dev_labels))
```

And when you look at the summary it is apparent that this dense network has 50,890 parameters.

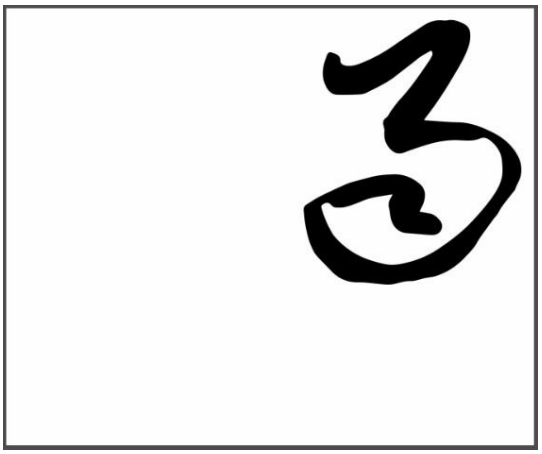
Layer (type)	Output shape	Params #
input_1 (InputLayer)	[(None, 784)]	0
dense_10 (Dense)	(None, 64)	50,240
dense_11 (Dense)	(None, 10)	650

Total params: 50,890
Trainable params: 50,980
Non-trainable params: 0

Now after five epochs, these parameters are fairly good, yielding an unseen development set accuracy of just above 97% after five training epochs.



But you can do a much better job with a convolutional neural network. A dense network like this has to relearn every feature and every location. So, for example, imagine you have a 3 in a position of the image that has not appeared before, maybe on the top right.

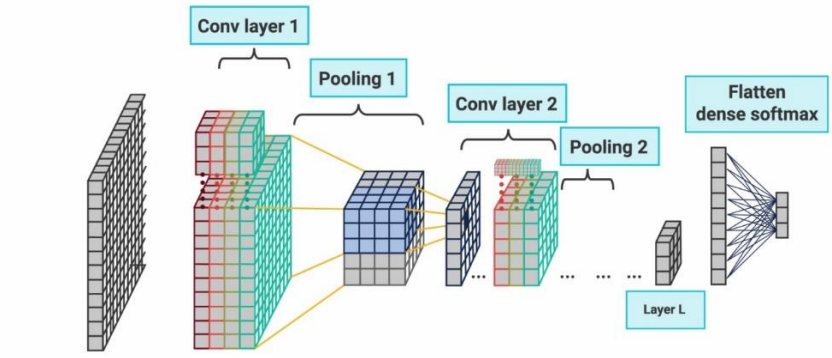


Since the training dataset has no such observations, the classifier would do a pretty poor job on that kind of image.

Now, by contrast, convolutional networks are **translation invariant**. But what did they look like in code in Keras? Well, here is an example of some code that creates a convolutional neural network.

```
model = keras.Sequential([
    layers.Conv2D(filters=32, kernel_size=3, activation="relu",
                  input_shape = (28, 28, 1)),
    layers.MaxPooling2D(pool_size=2),
    layers.Conv2D(filters=64, kernel_size=3, activation="relu"),
    layers.MaxPooling2D(pool_size=2),
    layers.Conv2D(filters=128, kernel_size=3, activation="relu"),
    layers.Flatten(),
    layers.Dense(10, activation="softmax")
])
```

The **input_shape** is optional, but it is specified in this case to emphasize that your model will be working on two-dimensional data. Now, that is in contrast to your dense model from before, which used just a one-dimensional version of its inputs. The resulting model has three convolutional layers and two max pooling layers.



Each of the pooling layers **downsamples** the features by a factor of two, and it selects the largest value from among each block. In your case, you selected k1 equal to 32, k2 equal to 64, and k3 equal to 128.

```
model = keras.Sequential([
        layers.Conv2D(filters=32, kernel_size=3, activation="relu",
                      input_shape = (28, 28, 1)),
        layers.MaxPooling2D(pool_size=2),
        layers.Conv2D(filters=64, kernel_size=3, activation="relu"),
        layers.MaxPooling2D(pool_size=2),
        layers.Conv2D(filters=128, kernel_size=3, activation="relu"),
        layers.Flatten(),
        layers.Dense(10, activation="softmax")
])
```

In other words, as you go deeper, each convolutional layer is learning a larger number of features. These choices are entirely arbitrary. And selecting the best network is hard.

You will have to rely on the wisdom of the broader community, who have conducted millions of painstaking experiments to arrive at the current best practices. In this case, these specific features were picked based on the book *Deep Learning with Python* by François Chollet.

You can use the **model summary** feature to understand each of the layers in this convolutional neural network. You see that the first convolution layer takes as input a 28 × 28 × 1 image, and it produces an output of size 26 × 26 × 32. To do this, it requires 320 parameters.

Layer (type)	Output shape	Params #
conv2d_15 (Conv2D)	(None, 26, 26, 32)	320
max_pooling2d_10 (MaxPooling2D)	(None, 13, 13, 32)	0
conv2d_16 (Conv2D)	(None, 11, 11, 64)	18,496
max_pooling2d_11 (MaxPooling2D)	(None, 5, 5, 64)	0
conv2d_17 (Conv2D)	(None, 3, 3, 128)	73,856
flatten_5 (Flatten)	(None, 1,152)	0
dense_13 (Dense)	(None, 10)	11,530

Total params: 104,202
Trainable params: 104,202
Non-trainable params: 0

Why? Since each of your 32 filters in your convolutional layer is a 3 × 3 grid of numbers, you have 3 times 3 plus 1 times 32 parameters. And the extra plus 1 is from the bias term for each of the filters. Note that there are no parameters for your max pooling and your flattened layer. Both of these layer types, they do a specific predefined task, which requires no training and thus, no parameters. Now in total, from the summary, you see that your model has 104,202 parameters, which you will learn by doing a **gradient descent** on the **cross-entropy loss** of your training set.

You have an alternate way to specify this exact same network, but now using the functional API.

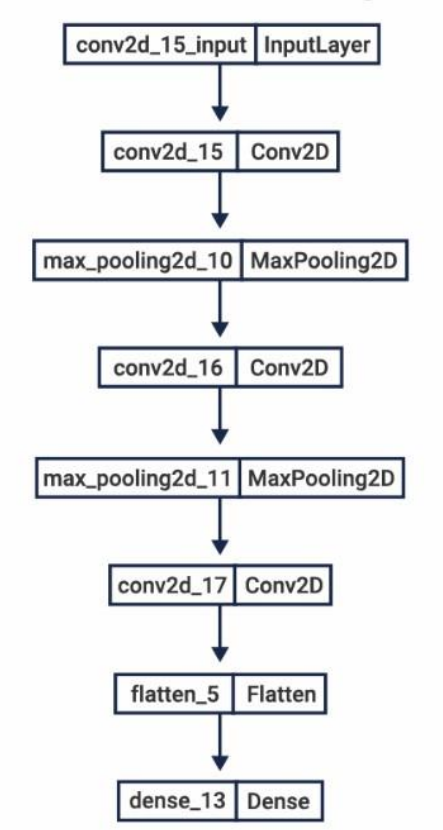
```
inputs = keras.Input(shape=(28, 28, 1))
x = layers.Conv2D(filters=32, kernel_size=3, activation="relu")(inputs)
x = layers.MaxPooling2D(pool_size=2)(x)
x = layers.Conv2D(filters=64, kernel_size=3, activation="relu")(x)
x = layers.MaxPooling2D(pool_size=2)(x)
```

```
x = layers.Conv2D(filters=128, kernel_size=3, activation="relu")(x)
x = layers.Flatten()(x)
outputs = layers.Dense(10, activation="softmax")(x)
model = keras.Model(inputs=inputs, outputs=outputs)
```

Now as before, this model does the exact same thing as its sequential API counterpart. You can also visualize your neural network using the **keras.utils.vis_utils.plot_model** function.

```
from keras.utils.vis_utils import plot_model
plot_model(model)
```

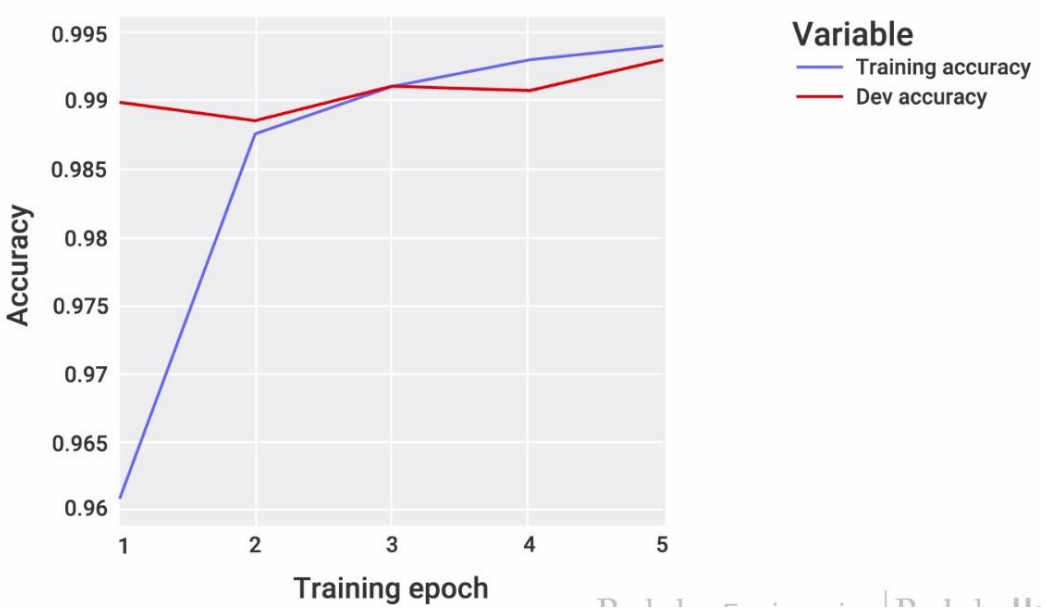
The resulting visualization can be helpful for understanding a network, especially complex networks that are graphs rather than sequential pipelines.



Training a convolutional neural network is exactly like training a dense neural network. You run the code shown.

```
model.compile(optimizer="rmsprop",
              loss="sparse_categorical_crossentropy",
              metrics=["accuracy"])

history = model.fit(train_images, train_labels, epochs=5,
            validation_data = (test_images, test_labels))
```

On this Google Colab instance, this takes a little over a minute. If you try to run this code and it is significantly slower, make sure that you have selected GPU as a runtime type. This time, plotting the training and validation set accuracy, you see that you get much better performance than your dense network.



You are getting 99.3% accuracy, or an error rate of only 0.7%.

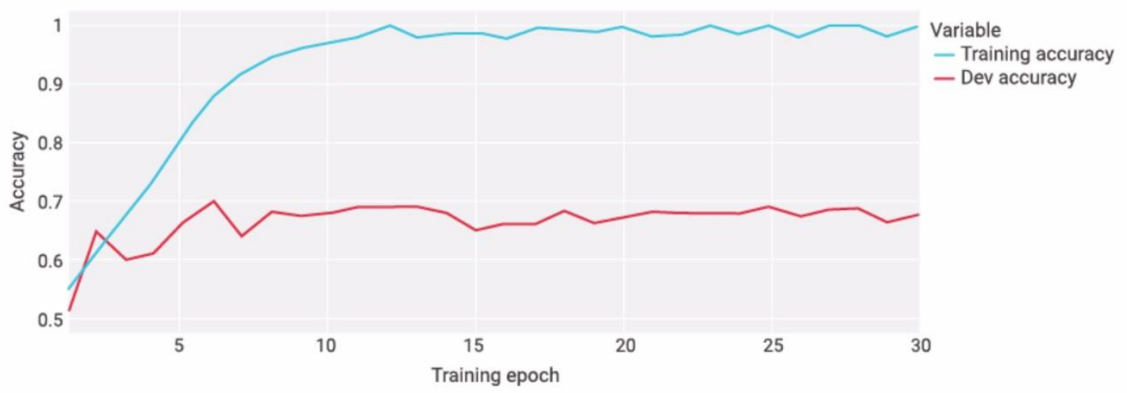
Video 3: Neural Networks for Vision (Part 2)

On the Kaggle platform, Microsoft Research has provided a large dataset of dog and cat images at https://www.kaggle.com/c/dogs-vs-cats. Next, you are going to try to build a **binary classifier** that takes an image and tells you whether it is a dog or a cat. You are going to assume that every image includes either a dog or a cat. In other words, your model does not have to worry about the possibility that the image contains neither or both.

You can first try out the exact model that you used on the MNIST dataset.

```
model = keras.Sequential([
    layers.Rescaling(1./255),
    layers.Conv2D(filters=32, kernel_size=3, activation="relu"),
    layers.MaxPooling2D(pool_size=2),
    layers.Conv2D(filters=64, kernel_size=3, activation="relu"),
    layers.MaxPooling2D(pool_size=2),
    layers.Conv2D(filters=128, kernel_size=3, activation="relu"),
    layers.Flatten(),
    layers.Dense(1, activation="sigmoid")
])
```

The difference here is that, rather than having a ten-way softmax that outputs ten probabilities, one for each written digit, you will instead just have a **single sigmoidal neuron** that gives a probability that the image contains a dog.



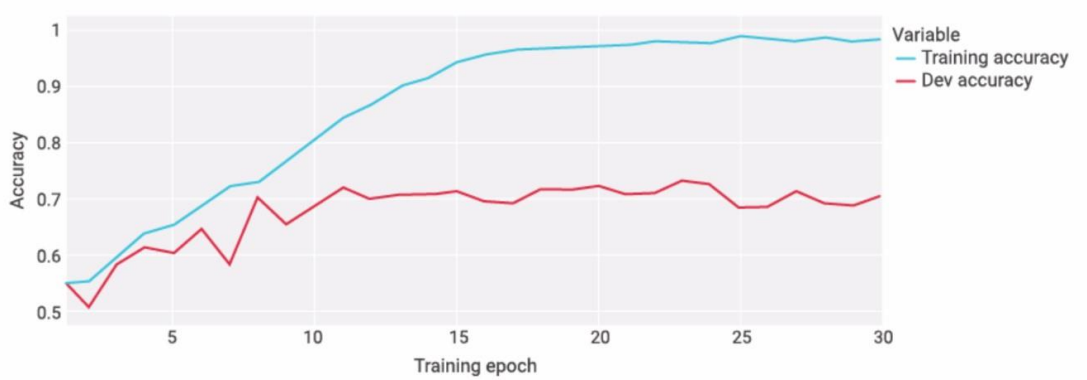
After training your network, you see that it is able to get great accuracy on the training set, in fact near 100%, with that same model architecture. But if you look at your development set, you see that it is unfortunately overfitting, getting only a bit worse than 70% accuracy on the validation set.

Now see what happens if you add a couple of additional convolutional layers and max pooling layers.

```
model = keras.Sequential([
    layers.Rescaling(1./255),
    layers.Conv2D(filters=32, kernel_size=3, activation="relu"),
    layers.MaxPooling2D(pool_size=2),
    layers.Conv2D(filters=64, kernel_size=3, activation="relu"),
    layers.MaxPooling2D(pool_size=2),
    layers.Conv2D(filters=128, kernel_size=3, activation="relu"),
    layers.MaxPooling2D(pool_size=2),
    layers.Conv2D(filters=256, kernel_size=3, activation="relu"),
    layers.MaxPooling2D(pool_size=2),
    layers.Conv2D(filters=512, kernel_size=3, activation="relu"),
    layers.Flatten(),
    layers.Dense(1, activation="sigmoid")
```

])

After training this deeper model, you see that the deeper model gets slightly better validation set accuracy.



So what do you do next? Well, there are so many tricks out there that can be applied. For example, one technique is **data augmentation**, where you will take your original dataset and you will add a bunch of distorted, rescaled, rotated versions of each image that is in the training set. Another trick you can use is **dropout**, where you randomly disable connections between some neurons.

However, here you are going to focus on one of the most powerful techniques you can use to create a useful deep learning model. This first approach is to use a **pretrained network**. Remember how all those different researchers trained neural network models to solve ImageNet? Well, it turns out that you can use those networks to solve new image recognition problems other than ImageNet.

To understand how that is possible, start by considering one network that does pretty well on ImageNet, called **VGG16**. This network originally appeared in a paper called *Very Deep Convolutional Networks for Large-*

Scale Image Recognition by Karen Simonyan and Andrew Zisserman, in 2014. As part of the Keras library, there are a bunch of pretrained libraries, including a copy of that VGG16 model. So the code shown loads the VGG16 model.

```
conv_base = keras.applications.vgg16.VGG16(
    weights="imagenet",
    include_top=True,
    input_shape=(224, 224, 3))
```

After loading their model, you can look at a summary of it. It is just a model like yours but their model is actually huge. It is 138,350,544 parameters.

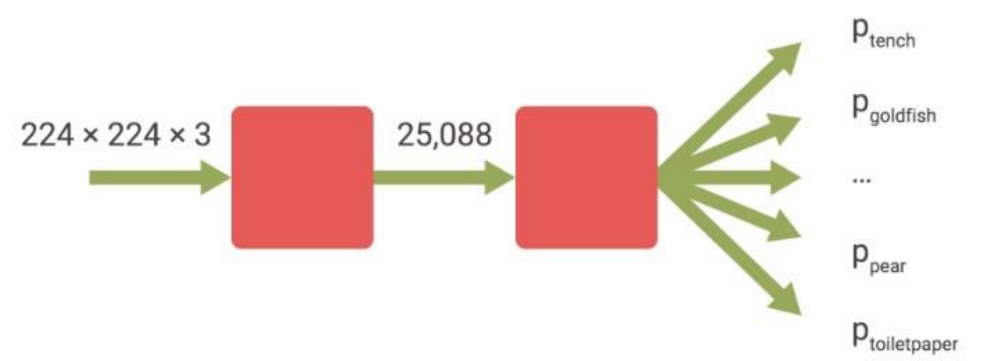
```
Model: "vgg16"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_7 (InputLayer)        [(None, 224, 224, 3)]     0
 block1_conv1 (Conv2D)       (None, 224, 224, 64)      1792
 block1_conv2 (Conv2D)       (None, 224, 224, 64)      36928
 block1_pool (MaxPooling2D)  (None, 112, 112, 64)      0
 block2_conv1 (Conv2D)       (None, 112, 112, 128)     73856
 block2_conv2 (Conv2D)       (None, 112, 112, 128)     147584
 block2_pool (MaxPooling2D)  (None, 56, 56, 128)       0
 block3_conv1 (Conv2D)       (None, 56, 56, 256)       295168
 block3_conv2 (Conv2D)       (None, 56, 56, 256)       590080
 block3_conv3 (Conv2D)       (None, 56, 56, 256)       590080
 block3_pool (MaxPooling2D)  (None, 28, 28, 256)       0
 block4_conv1 (Conv2D)       (None, 28, 28, 512)       1180160
 block4_conv2 (Conv2D)       (None, 28, 28, 512)       2359808
 block4_conv3 (Conv2D)       (None, 28, 28, 512)       2359808
 block4_pool (MaxPooling2D)  (None, 14, 14, 512)       0
 block5_conv1 (Conv2D)       (None, 14, 14, 512)       2359808
 block5_conv2 (Conv2D)       (None, 14, 14, 512)       2359808
 block5_conv3 (Conv2D)       (None, 14, 14, 512)       2359808
 block5_pool (MaxPooling2D)  (None, 7, 7, 512)         0
 flatten (Flatten)           (None, 25088)             0
 fc1 (Dense)                 (None, 4096)              102764544
 fc2 (Dense)                 (None, 4096)              16781312
 predictions (Dense)         (None, 1000)              4097000

=================================================================
Total params: 138,357,544
Trainable params: 138,357,544
Non-trainable params: 0
```

It consists of a large number of things you have seen, convolutional and max pooling layers. Now, as with the models that you built today, you see that the convolutional layers in VGG16 have an increasing number of filters as you move toward the top of the network.

Note that the **input** to a neural network is considered the **bottom** and the **output** of a neural network is considered the **top**. That terminology can be a little confusing in the context of looking at the summary, because the first line of the model in the summary, the input layer, is actually the bottom. And then the last layer, called block5_pool, is the top.

Now observe that, as you move toward the top, there are max pooling layers that downsample the features, just like your network. But unlike your network, the max pooling layers only appear every few convolutional layers. Well, at the very end of the network at the top, you see that the output of the convolution and the max pooling layers gets flattened into 25,088 features. That is similar to your model, where the convolutional part of your network got eventually flattened into 1,152 features. Now, unlike your model built on MNIST, which fed your 1,152 convolutionarily-generated features straight into a ten-way softmax, their VGG16 network actually has two additional dense layers. And then, finally, a 1,000-way softmax.

How did VGG16's designers know to use this specific set of layers or sequence of layers? Well, they did not. By experimenting with different neural networks, they found the one that would perform best on ImageNet. The team's choices were guided by theoretical concerns but the only way they knew that their architecture worked, was to try a bunch of experiments. So how does that help you? Well, I like to think of VGG16's model as having two primary stages.



The **first stage** takes as input a 224 × 224 × 3 image, where 3 represents the number of color channels. And then that first stage outputs a list of 25,088 features. Those features represent the presence of 512 different features in each of the 49 regions of the image.

So then the next three layers are the **second stage**. This is fc1, fc2, and predictions, where predictions is that 1,000-way softmax that outputs the probability that the image belongs to each of the ImageNet classes. In other words, it gives the probability that the image contains a tench, a goldfish, a pear, toilet paper, and so forth. So this second chunk of the network, consisting of two dense layers and a softmax, does not get to see the original images directly. Instead, it only has access to the 25,088 features provided by that first big convolutional base. Now since these features were generated by a sequence of convolution and max pooling layers, you expect that these features will have nice properties, like being able to recognize objects, even if they are in somewhat different regions of the image, where that object never appears in the training set. Another way of putting that is **translational invariance**.

Now with this idea of the network having two primary stages in mind, the VGG16 model is useful for your dog–cat classification task. Now that is true, even though “dog” and “cat” are not labels in the original ImageNet

dataset. In other words, the softmax layer of VGG16, it does not produce a p_{dog} or p_{cat}. In fact, VGG16 would be useful for identifying any number of objects that are not in the ImageNet dataset. Like horses, for example. Those were not in ImageNet, but VGG16 would be useful to consider trying to identify a horse. So the ImageNet dataset does not contain pictures of horses, but it does have images of zebras, and bison, and gazelle, and so forth. Now, since many of these animals have similar components to horses, some of the features learned in the early layers to distinguish these animals, could also, in principle, be used to detect an image of a horse.

So what you are going to do is change the parameter, **include_top**, to **False**.

```
conv_base = keras.applications.vgg16.VGG16(
    weights="imagenet",
    include_top=False,
    input_shape=(224, 224, 3))
```

When you do this, the parts of the network starting with the flattened step get chopped off. The top of the network is gone. So the flattened layer, the last two dense layers, and the 1,000-way softmax are removed.

```
Model: "vgg16"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_7 (InputLayer)         [(None, 224, 224, 3)]     0
block1_conv1 (Conv2D)        (None, 224, 224, 64)      1792
block1_conv2 (Conv2D)        (None, 224, 224, 64)      36928
block1_pool (MaxPooling2D)   (None, 112, 112, 64)      0
block2_conv1 (Conv2D)        (None, 112, 112, 128)     73856
block2_conv2 (Conv2D)        (None, 112, 112, 128)     147584
block2_pool (MaxPooling2D)   (None, 56, 56, 128)       0
block3_conv1 (Conv2D)        (None, 56, 56, 256)       295168
block3_conv2 (Conv2D)        (None, 56, 56, 256)       590080
block3_conv3 (Conv2D)        (None, 56, 56, 256)       590080
block3_pool (MaxPooling2D)   (None, 28, 28, 256)       0
block4_conv1 (Conv2D)        (None, 28, 28, 512)       1180160
block4_conv2 (Conv2D)        (None, 28, 28, 512)       2359808
block4_conv3 (Conv2D)        (None, 28, 28, 512)       2359808
block4_pool (MaxPooling2D)   (None, 14, 14, 512)       0
block5_conv1 (Conv2D)        (None, 14, 14, 512)       2359808
block5_conv2 (Conv2D)        (None, 14, 14, 512)       2359808
block5_conv3 (Conv2D)        (None, 14, 14, 512)       2359808
block5_pool (MaxPooling2D)   (None, 7, 7, 512)         0

=================================================================
Total params: 14,714,688
Trainable params: 14,714,688
Non-trainable params: 0
```

So now if you look at the model summary, you see the output of the network is of size 7 × 7 × 512, rather than being a list of 1,000 probabilities, one for each class.

Now if you feed an image into this network that is been truncated, you get back these 25,088 features as a summary of the input image. So how do

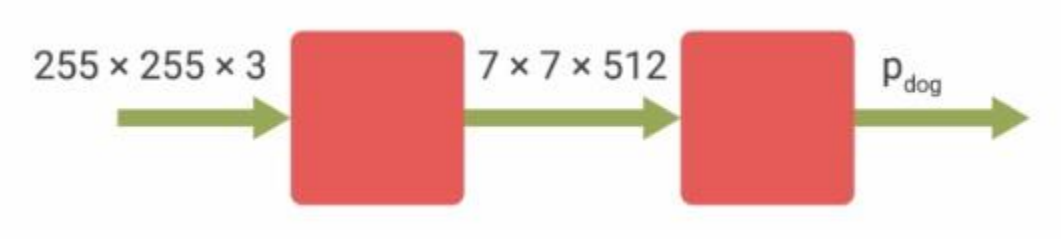
you actually use this to train a dog–cat model? You simply stick your own second primary stage under the end of their existing network to replace what you just chopped off. Now since you only need to generate a True False value, the simplest choice is just to feed those features into a sigmoidal neuron. In other words, here you are doing just standard **logistic regression,** albeit on the 25,088 generated features, rather than on the original image data. The code shown here, using the sequential API, creates such a network.

```
model_with_vgg16_bottom = keras.Sequential([
    conv_base,
    layers.Flatten(),
    layers.Dense(1, activation="sigmoid")
])
```

Now the resulting model is as shown.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
vgg16 (Functional)           (None, 7, 7, 512)         14714688
flatten_18 (Flatten)         (None, 25088)             0
dense_30 (Dense)             (None, 1)                 25089

=================================================================
Total params: 14,739,777
```

The first stage computes this 7 × 7 × 512 featurization of the image, using the 14,714,688 features of the convolutional base that Karen Simonyan and Andrew Zisserman generated back in 2014. Then there are an additional 25,089 parameters, for your part of the model, that you will generate yourself using the dog and cat dataset.



So those parameters correspond to the weights of the 25,088 features that were generated by VGG16, as well as an intercept term. You are now very close.

But before you train your network on your dog–cat dataset, there are two important things you should do. The first is that you should freeze this VGG16 network. That is, when you perform your gradient descent, you should not mess with the weights of the network that generate the 25,888 features. Now to do this, you set **conv_base.trainable** equal to **False**.

```
model_with_vgg16_bottom = keras.Sequential([
    conv_base,
    layers.Flatten(),
    layers.Dense(1, activation="sigmoid")
])
conv_base.trainable = False
```

After doing this, if you look at your summary, your model now has only 25,089 trainable parameters, and the others will be left alone during the gradient descent process. They are frozen.

```
Layer (type)                 Output Shape              Param #
=================================================================
vgg16 (Functional)           (None, 7, 7, 512)         14714688
flatten_23 (Flatten)         (None, 25088)             0
dense_36 (Dense)             (None, 1)                 25089

=================================================================
Total params: 14,739,777
Trainable params: 25,089
Non-trainable params: 14,714,688
_________________________________________________________________
```

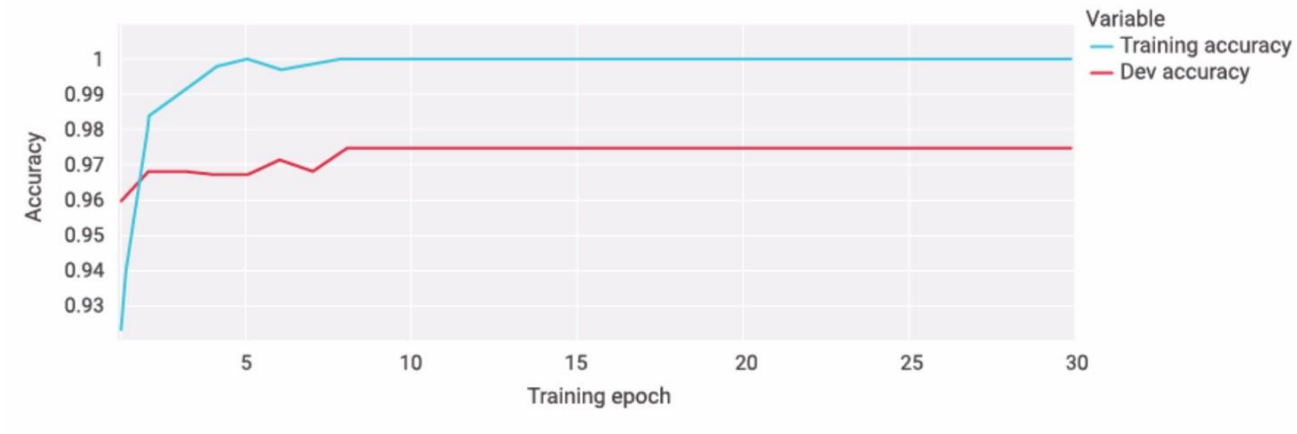
There is a second thing you should do. It turns out that the VGG16 network was trained with the data in a very specific, somewhat nonstandard format. For example, the blue channel came before the red channel in that dataset. However, in your setup, the red channel is first. So luckily a function exists in Keras that can manipulate your images, so that they also have the quirks of the dataset that was originally fed to VGG16. This helper function is called **keras.applications.vgg16.preprocess_input**. So this function is much more natural to add to your neural network by using the functional API.

```
inputs = keras.Input(shape=(224, 224, 3))
x = keras.applications.vgg16.preprocess_input(inputs)
x = conv_base(x)
x = layers.Flatten()(x)
outputs = layers.Dense(1, activation="sigmoid")(x)
model_with_vgg16_bottom = keras.Model(inputs, outputs)
conv_base.trainable = False
```

As always, the functional and the sequentional API implementations, they result in the exact same network. So now if you look at the resulting model summary, the first three layers represent the pre-processing.

```
Layer (type)                 Output Shape              Param #
=================================================================
input_22 (InputLayer)        [(None, 224, 224, 3)]     0
tf.__operators__.getitem_6    (None, 224, 224, 3)       0
(SlicingOpLambda)
tf.nn.bias_add_6 (TFOpLambda)  (None, 224, 224, 3)      0
vgg16 (Functional)            (None, 7, 7, 512)         14714688
flatten_24 (Flatten)          (None, 25088)             0
dense_37 (Dense)              (None, 1)                 25089
=================================================================
Total params: 14,739,777
Trainable params: 25,089
Non-trainable params: 14,714,688
```

Now that you have done those two important things, you simply train the model as usual. Recall that the convolutional neural network that you trained, totally from scratch, was only able to get 70% accuracy on the validation set. By contrast, your new model, which leverages the existing VGG16 model to featurize each image, is able to get 97% accuracy on the validation set.



This idea of reusing the bottom of existing neural networks is incredibly important in modern machine learning. In major companies and research institutes—like, say, Google—large neural networks have been trained at enormous expense, which are reused all across these organizations. That is because the bottoms of these networks can **featurize** data in a way that is incredibly useful for all kinds of tasks.

In many contexts, it is actually pretty rare to train your own gigantic model entirely from scratch. Consider the GPT-3 text generation neural network. This model has hundreds of billions of parameters. It is simply a better foundation for the purpose of text generation than anything that you could build yourself from raw data.

Video 4: Fine-Tuning Your Model

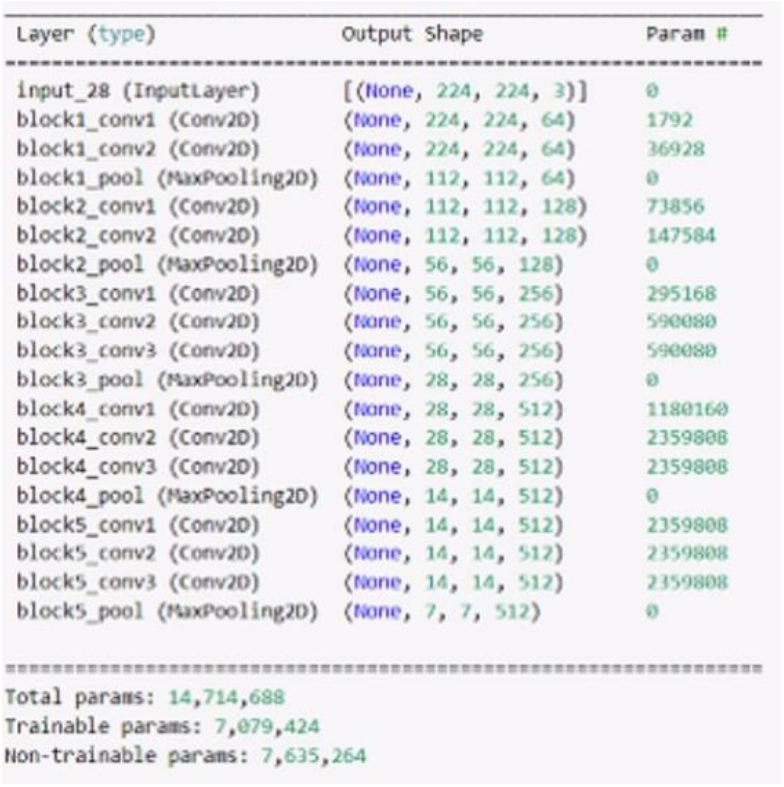
Often you can squeeze out just a bit more performance by **fine-tuning** your model. For your specific problem here, fine-tuning is only going to make a tiny difference, but in some cases it can help a lot more. Fine-tuning a model is a relatively minor tweak to your workflow. First you train your combined model.

```
history_with_vgg16_bottom = model_with_vgg16_bottom.fit(
    train_dataset, epochs=30,
    validation_data=validation_dataset)
```

You will then unfreeze the last few layers of the huge model that you are using as a base. So for this example, you will unfreeze the last four layers of VGG16. You will then train again, jointly train the parameters of these last few layers of their network, along with 25,089 parameters of your sigmoidal neuron.

```
conv_base.trainable = True
for layer in conv_base.layers[:-4]:
  layer.trainable = False
```

Note that this model summary shown, it only lets you see the trainable parameters and the convolutional base. And it does not show the 25,089 parameters of your sigmoidal neuron, which will also be trained.



Layer (type)	Output Shape	Param #
input_28 (InputLayer)	[(None, 224, 224, 3)]	0
block1_conv1 (Conv2D)	(None, 224, 224, 64)	1792
block1_conv2 (Conv2D)	(None, 224, 224, 64)	36928
block1_pool (MaxPooling2D)	(None, 112, 112, 64)	0
block2_conv1 (Conv2D)	(None, 112, 112, 128)	73856
block2_conv2 (Conv2D)	(None, 112, 112, 128)	147584
block2_pool (MaxPooling2D)	(None, 56, 56, 128)	0
block3_conv1 (Conv2D)	(None, 56, 56, 256)	295168
block3_conv2 (Conv2D)	(None, 56, 56, 256)	590080
block3_conv3 (Conv2D)	(None, 56, 56, 256)	590080
block3_pool (MaxPooling2D)	(None, 28, 28, 256)	0
block4_conv1 (Conv2D)	(None, 28, 28, 512)	1180160
block4_conv2 (Conv2D)	(None, 28, 28, 512)	2359808
block4_conv3 (Conv2D)	(None, 28, 28, 512)	2359808
block4_pool (MaxPooling2D)	(None, 14, 14, 512)	0
block5_conv1 (Conv2D)	(None, 14, 14, 512)	2359808
block5_conv2 (Conv2D)	(None, 14, 14, 512)	2359808
block5_conv3 (Conv2D)	(None, 14, 14, 512)	2359808
block5_pool (MaxPooling2D)	(None, 7, 7, 512)	0

Total params: 14,714,688
Trainable params: 7,079,424
Non-trainable params: 7,635,264

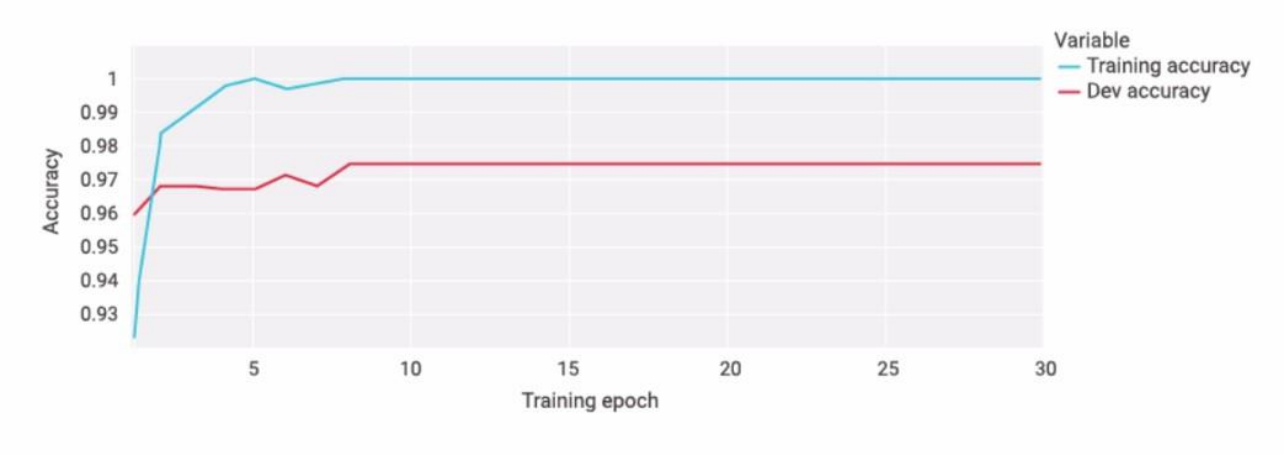
In other words, after you unfreeze the last four layers, the last 7,079,424 parameters that are part of the last three convolutional 2D stages, they are going to be part of your gradient descent process. So why were these three layers picked specifically? Conceptually, the reason why you only unfreeze the topmost layers is that the bottom layers capture something more universal about all image classification. Whereas the top few layers, they are conceptually higher-level features that are more likely to be usefully informed by your dataset. Note that the syntax first tells the neural network that it can be trained, but then tells it that the first several layers, up until the fifth to last layer, that they are not trainable.

So now the third and final step in your fine-tuning process is to call **fit**, but on your existing already trained model.

model_with_vgg16_bottom.compile(loss="binary_crossentropy",

```
        optimizer="rmsprop",
        metrics=["accuracy"])

fine_tuning_history = model_with_vgg16_bottom.fit(
    train_dataset, epochs=30, validation_data=validation_dataset)
```

In this example, you set the learning rate to a smaller value. You do not want to make large gradient descent steps, because you are just trying to eke out a bit more performance. So the first figure here shows the accuracy of your VGG16 base model, when you train it from scratch.



Now the second figure you see gives you the evolution of the accuracy after you fine-tune.

In this case, fine-tuning has only a tiny impact, bringing you from 97.4% to 97.5% validation set accuracy. Note that if you run these exact same experiments, you might get different results, again, because of the fundamentally random nature of training neural networks. However, you will find that fine-tuning, where you unfreeze the base you are using, can give you a small, but sometimes really significant, performance boost in some real-world problems.

Mini-Lesson: Tricks of the Trade

Researchers and practitioners acquire knowledge through experience and word-of-mouth, which helps them successfully apply neural networks to complex real-world problems. Unfortunately, the 'tricks' they learn, although often theoretically sound, can take years for newcomers to assimilate. As a result, newcomers to the field spend a lot of time wondering why their networks train slowly and perform poorly. Now you'll take a look at some techniques to assist you in preparing high-performance neural networks.

Shuffling the Examples

Neural networks learn the fastest from the most random samples. Therefore, selecting a sample that is the most unfamiliar to the system at each iteration is advisable. However, this applies only to stochastic learning (since the batch does not care about the order of the presentation of inputs). Although there is no simple way to determine which inputs are information rich, one straightforward technique that crudely implements this idea is to choose consecutive examples from different classes (since training examples belonging to the same class will most likely contain similar information).

Early Stopping

Regularization techniques are essential to improve the generalization ability of neural networks. One of the most commonly used strategies is early stopping. This technique can be explained in its simplest form as follows:

Take a validation set that is independent of the training set and monitor the errors on this set during training. In the training set, the error will decrease, whereas, in the validation set, the error will decrease and then increase. The early stopping point occurs when the validation set errors are at their lowest. The network weights are at their most general at this point.

Neural Networks Classification and Prior Class Probabilities

A common problem in multilayer perceptron (MLP) classification is related to the prior probabilities of the individual classes. If the number of training examples that correspond to each class differs significantly, then sometimes, the network may have difficulty learning the rarer classes. When prior class probabilities are unequal, a simple method of alleviating difficulty is to adjust (e.g., equalize) the number of patterns in each class. You can do this either through subsampling (removing patterns from higher frequency classes), or by duplication (of patterns in lower frequency classes). In subsampling, patterns may be removed randomly, or heuristics may be used to remove patterns in regions with low ambiguity. However, subsampling can cause lost information.

Applying Divide and Conquer to Large-Scale Pattern Recognition Tasks

Traditionally, large-scale problems have always been solved using the divide-and-conquer paradigm, which is a powerful approach. Hence, a hierarchical approach can be used to modularize classification tasks in

large-scale application domains, such as speech recognition, where there are thousands of classes and millions of training samples. Divide and conquer proves to be an effective tool for breaking down a complex problem into many smaller tasks. Furthermore, agglomerative clustering can automatically impose a suitable hierarchical structure on a set of classes, even if it contains tens of thousands of classes. In contrast to the relatively small standard benchmarks for learning machines, factors such as training method, model selection, and generalization ability take on a new meaning when dealing with large-scale probability estimation problems.

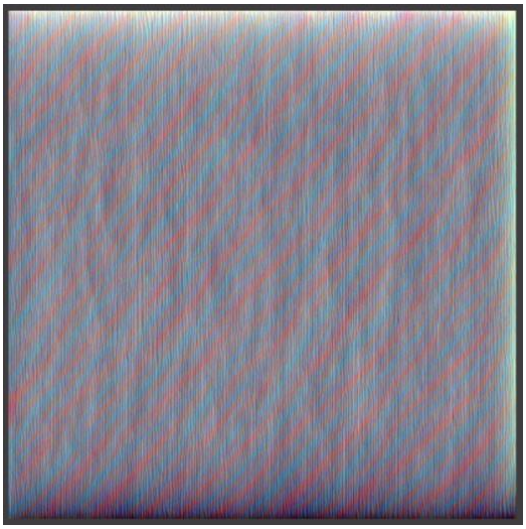
Video 5: Interpreting Neural Networks for Vision

Models like VGG16 learn some sort of general features that are useful for image recognition. That is, each convolutional layer implements some set of features, each of which represents some interesting feature. But what are these filters? And how can you describe them in a way that makes sense?

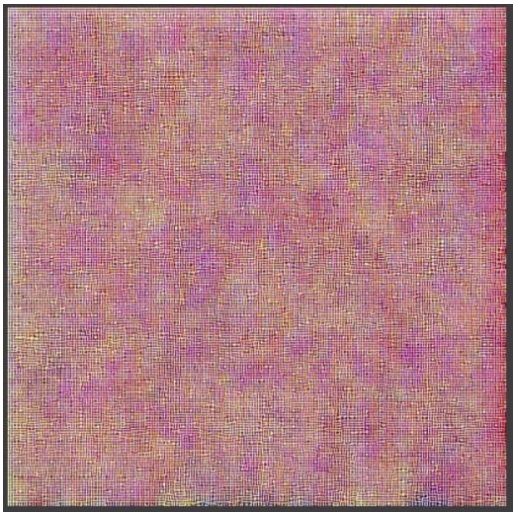
In *How to Visualize Convolutional Features in 40 Lines of Code* by astrophysicist Fabio Graetz, Fabio provides code that allows you to visualize the features that VGG16 computes. So the basic idea behind this approach is that Fabio picks a feature he would like to maximize. He then searches the space of all possible images to find an image that highly activates that feature. He searches the space of images by conducting a gradient descent on the **input**, instead of on the model. So what do you do after identifying an image which highly activates a target feature? Well, since the image highly activates that feature, you can think of your network as trying to identify images that look like the image you have generated.

Note that it is possible that there maybe are several completely different-looking images, which might activate a particular feature of a neuron. And if that is the case, Fabio's procedure will identify only one such exemplar image.

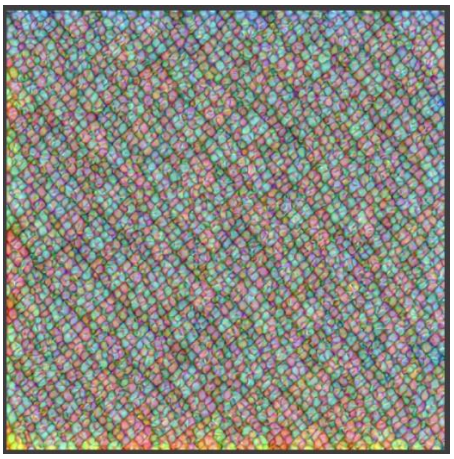
Now look at an example. This image highly activates the 12th filter of one of the early layers of VGG16.



Whereas this next image shown, these strange patterns, these highly activate the 110th filter of this same layer.

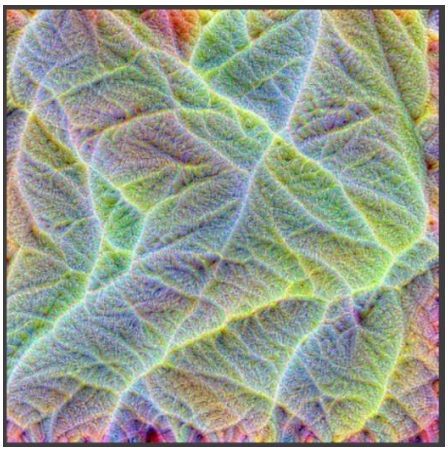


In some sense then, the 12th filter of this layer is looking for images with diagonal banding. Whereas the 110th filter, it looks for what I might describe as 90° edges. Now as you go deeper into the network, the textures that activate the filters look increasingly complex. For example, the image shown highly activates the 190th filter of a later layer of VGG16.

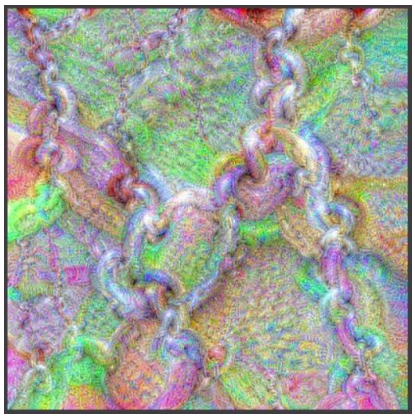


As an example, you can imagine that images of explosions or bubbles, or things like that, they might more highly activate this feature than images of, say, a desert.

Going even deeper into the model, you see more interesting exemplar images. For example, this image highly activates filter 123 in one of the later stages of the network.



It seems plausible that this feature tends to be large for leafy plants. And then once you get to the very top convolutional layer, the exemplar images become really recognizable to humans, many of them at least. For example, this image shown, which highly activates filter 265 in the final convolutional layer, looks like chains.



Now keep in mind, that is just one example of a potentially huge number of images that highly activate this filter.

As an experiment, after identifying an exemplar image, you can go grab a real-world image you think resembles that generated exemplar image. If you want to test the hypothesis that a given filter is a chain detector, for example. Fabio does an experiment where he feeds this image of a chain and he finds that filter 265 in the final layer is actually highly activated.

Now that is true, even though the exemplar image that he generated is a more complex linkage of chains. It is some kind of chain-recognizing neuron. While generating such exemplar images may seem like a purely aesthetic pursuit that you are doing just for fun, this sort of thinking is actually useful. Notably, it gives you a way of interpreting your neural networks. So even though VGG16 has tens of millions of parameters in its convolutional base, you can get some sense of what it is doing.

So in a world where neural network models are increasingly making these really important decisions, often with fidelity that exceeds human capability, the ability to audit and understand the reasoning is vital. As an example of how interpretability can be very hard, consider the world of **adversarial machine learning inputs**. There is a paper called *One Pixel Attack for Fooling Deep Neural Networks* by Jiawei Su, Danilo Vargas, and Kouichi Sakurai, from 2017. In that paper, the authors take a trained and fairly accurate model on the ImageNet dataset and they look for weaknesses in the model. So usually changing just a single pixel in an image will have no impact on the predictions of a neural network. However, in some limited cases, changing a single pixel can dramatically change the model's opinion.

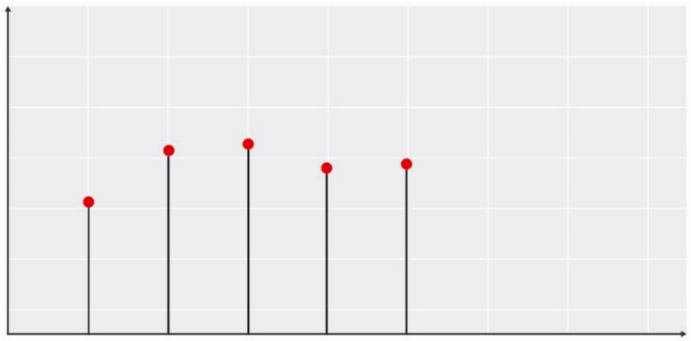
Models like VGG16 perform well on the natural distribution of images out there, but they often fail on images that are outside that natural distribution by only one pixel. So it is not clear that this fact has any practical consequence. But it does suggest that maybe these networks are not as smart as they might appear.

Video 6: Neural Networks for Text or Time Series

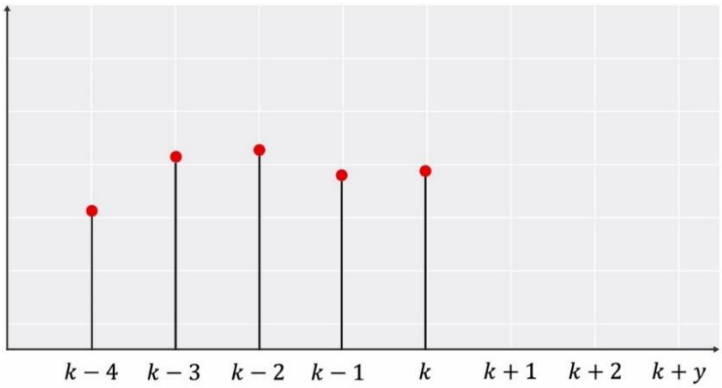
So you have seen how, with CNNs, you adapted the architecture of neural networks to work with data of a particular type, namely images. Now you

will learn how to create neural networks for processing **sequential data**. You have encountered sequential data before when you studied time series. However, the applications of this type of data for neural networks go way beyond that. They include things like machine translation, speech recognition, automatic image captioning, and even machine-generated music and art. The type of neural networks that work best for sequential data are called **recurrent neural networks**, or RNNs. Next you will explore the fundamental ideas, in order to give a sense of how they work.

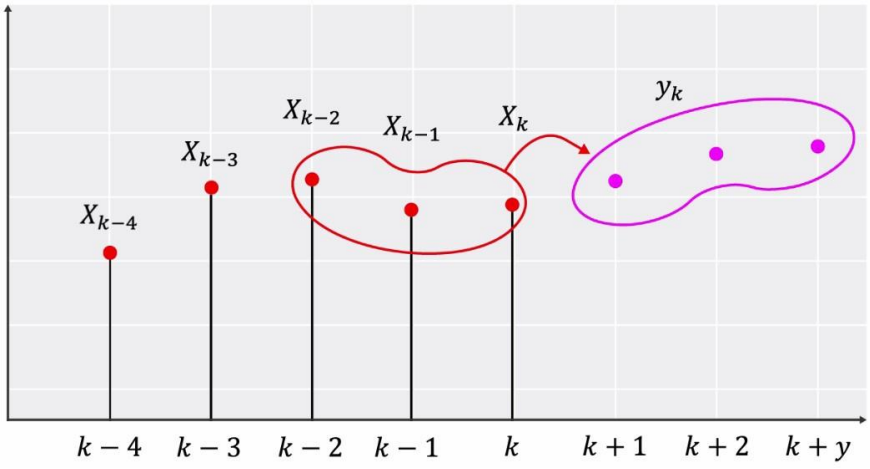
Say you have a sequential dataset consisting of a long list of numbers or words.



Your goal is to guess the next few items in the list. You will need some notation. Use k to denote your current position in the list. Then $k+1$ is the upcoming position and $k-1$, $k-2$, et cetera, are points in the past.

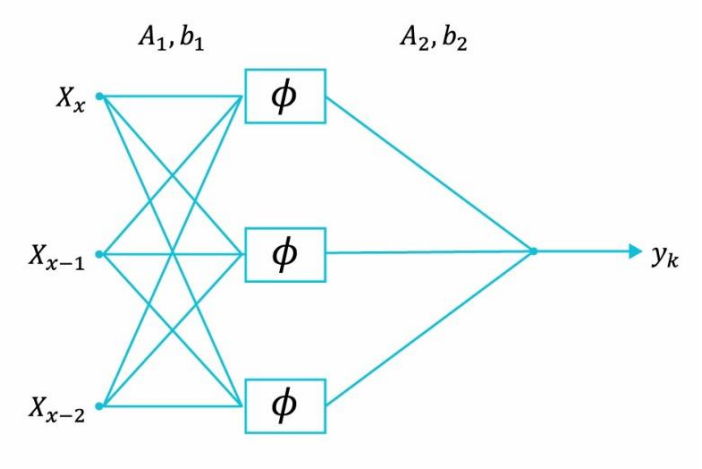


You will base your prediction on a finite history; in this case, the most recent three data points.

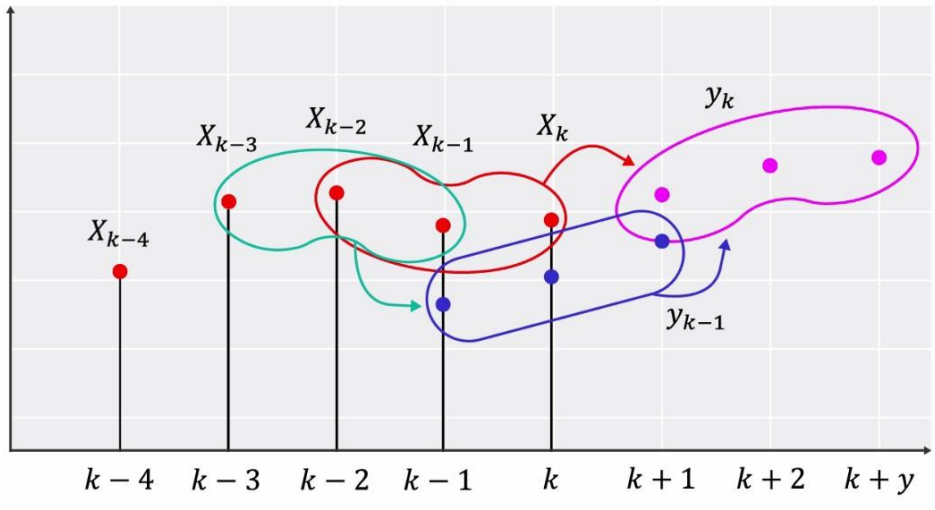


You will assume that the list is a time series, and so k represents a moment in time. But keep in mind that the list could also be a document, in which case k points to the word that you are currently focused on. You will denote the prediction that you make at time, k, with y_k, which may also be a list. The goal for your prediction, y_k, will be to match the ground truth as closely as possible.

First consider a naive approach in which you feed the historical data into a plain vanilla dense neural network with y_k as the output.



This is similar to what you tried with CNNs. And the problem with this approach is the same. It begins by discarding all of the sequential information, which it must then relearn in the training process. But it also ignores another important aspect of the problem, which is that the predictions are made continuously.

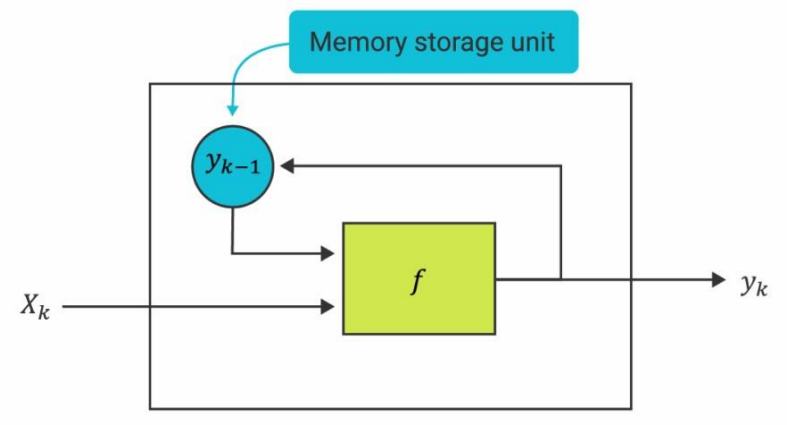


$$y_k = f(X_k, y_{k-1})$$

And so at time k you also have available the previous prediction from y_{k-1}. That prediction was made on the basis of data from $k-3$ to $k-1$. So if you include y_{k-1} in your model, then you implicitly gain access to X_{k-3}. And thus, you extend your historical information by one data point. Simply by including the previous prediction in your model, you automatically gain implicit access to all of the historical data, without having to store it in memory.

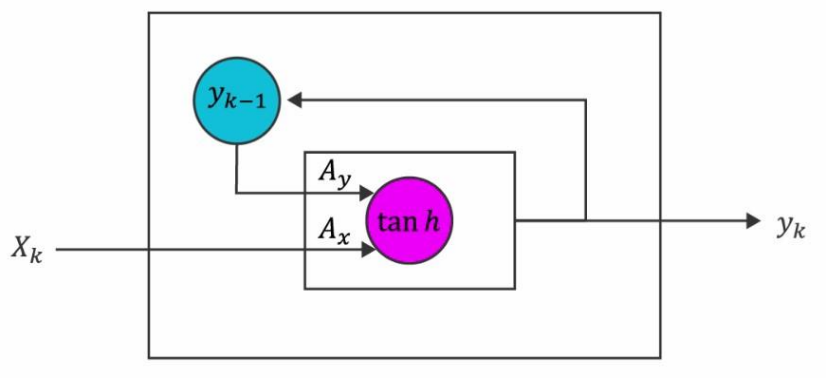
This is not a new idea. You call it **autoregression** in the context of ARMA models. In the neural network world, it is the fundamental idea behind recurrent neural networks. Both of these terms, autoregressive and recurrent, refer to the idea of feeding the output of the model back upon

itself, as a way of preserving information from the past. Here is a graphical representation of the idea.



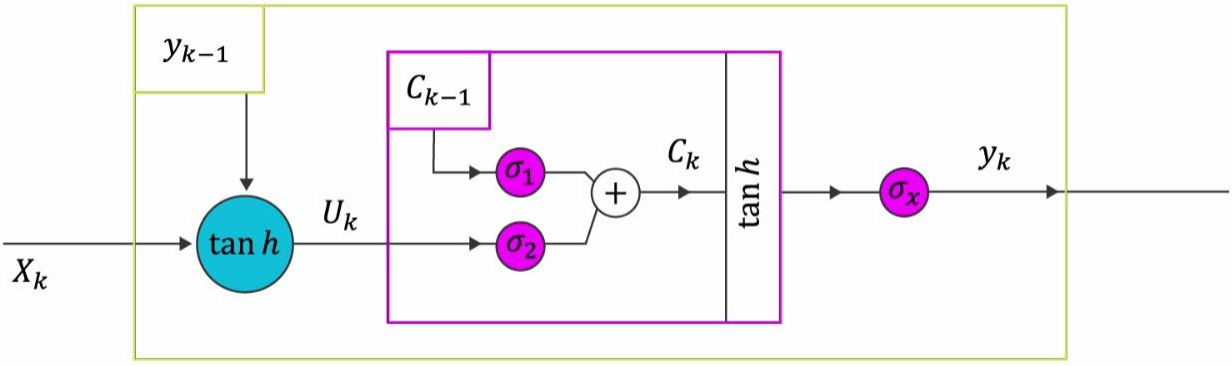
$$y_k = f(X_k, y_{k-1})$$

The blue circle is a **memory storage unit** that keeps a copy of the previous output. At each time step, the model, f, takes the value of the memory unit, along with the new input, X_k, and produces a new output, y_k. This output is exported and also fed back to the memory unit, where it replaces the previous value. In neural networks, a component like this is called a **memory cell**. And a network with memory cells is called a **recurrent neural network,** or RNN. Each cell in an RNN can hold multiple values in memory. And the total memory content of an RNN, at any given point in time, is called the **state** of the network. When you initialize an RNN, you must populate its memory with some initial state. And you will typically set it all to zeros or to some random noise. Then as you feed information through the network, such as text or music, it will continually update its state, much as you do when you read a book. You hold in your memory a representation of the data, that is based on what was most recently read. In artificial intelligence, as in human intelligence, the ability to hold information in memory is essential to your comprehension of the text.



$$y_k = \tan h\left(A_y y_{k-1} + A_x X_k + b\right)$$

Here you see the simplest type of memory cell in a recurrent neural network. The f-function is just a dense layer that takes X_k and y_{k-1} as inputs, transforms them with weight matrices, A_y and A_x, adds an offset vector, b, and applies an activation function. Bear in mind that both the input and the output of the cell can be vectors. So this model is naturally capable of integrating many sources of information as inputs. This type of memory cell is implemented in scikit-learn in the simple RNN class. One of the limitations of such simple memory cells is that they tend to work well only for short predictions. Researchers looking to improve its performance came up with a more complex design that nests two memory cells within one. This design is called the **long short-term memory cell**, or LSTM. The internal cell, shown here in purple, keeps its own state, which you call C.



$$C_k = \sigma_1 C_{k-1} + \sigma_2 U_k$$

This state is not exported from the cell. It is a so-called hidden state, but it serves as a repository of long-term information. In this diagram, you will notice three purple circles. Each of these are dense layers applied to the input, X_k, and the previous output, y_{k-1}, with a sigmoid activation function. You can think of these as dials or dimmer switches that are controlled by X and y. The two dimmers, σ_1 and σ_2, regulate the rate at which information is discarded from or added to the state, C. You can see from this formula that if σ_1 is less than 1, then C_k will retain only a fraction of its previous value.

$$\sigma_1 = \sigma\left(A_{y1} y_{k-1} + A_{x1} X_k + b_1\right)$$

Similarly, σ_2 regulates the amount of the input, U, that is added to C at every time step.

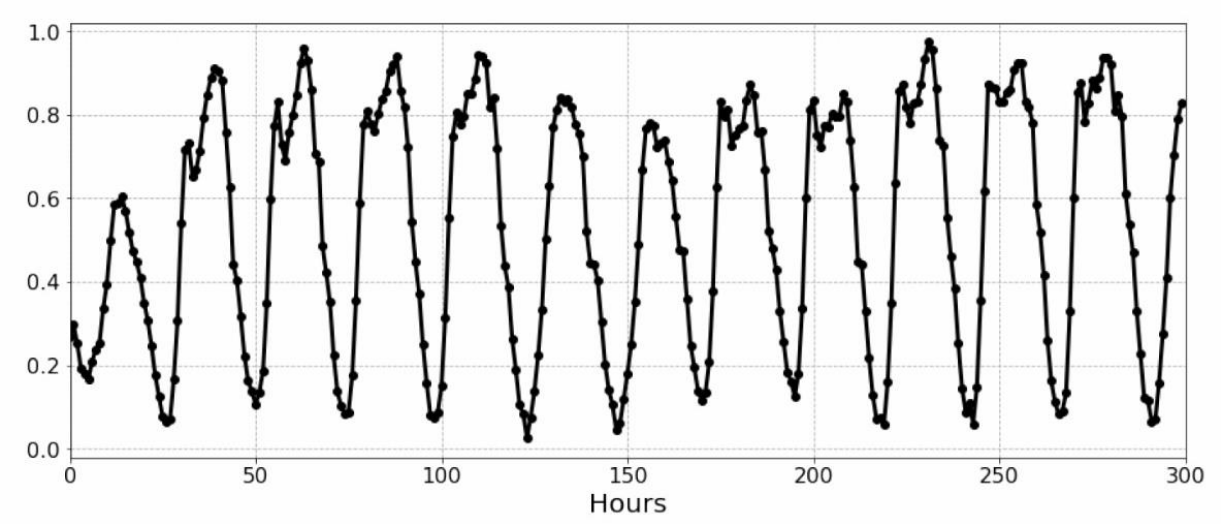
$$\sigma_2 = \sigma\left(A_{y2} y_{k-1} + A_{x2} X_k + b_2\right)$$

The input to the internal cell, U, is also controlled by a dense layer, this one with a tan h activation function. And the output of the entire LSTM cell is a third dimmer, σ_3, multiplied by the $tan\ h$ of C_k.

$$y_k = \sigma_3 \tan h\,(C_k)$$

This model is much more complicated than the simple RNN you saw previously. Each LSTM cell contains within it four dense layers and two memory states.

Next, you will test these three models on the task of predicting vehicular flow on a freeway. The data for this problem are two years of hourly flow from a particular sensor on the 405 freeway near Irvine.



Here you see the first 300 hours of data, from January 1st to January 12th of 2018. The measurements have been normalized to fall within a range from 0 to 1. In total, there are 13,980 hours of data. Your goal will be to make a 24-hour forecast, based on the most recently observed 6 hours of flow. The main difference between the training of RNNs and other models you have seen, has to do with how the data is prepared.

```
def build window(data, h, f):
dataX, dataY = [1, []
        for i in range(len(data)-h-f):
        dataX.append(data[i:(i+h)])
        dataY.append(data[(i+h):(i+h+f)])
dataX = np.array(dataX) dataX = np.reshape(dataX, (dataX.shape[0], 1,
dataX.shape[1]))
dataY = np.array(dataY)
return dataX, dataY
```

The single long array must be split into a series of historical and forecasting windows.

```
dataX, dataY = build_window(data, history_length, forecast_length)
data.shape, dataX.shape, dataY.shape
((13980,), (13950, 1, 6), (13950, 24))

num train samples = int(0.8 * len(raw data))
trainX, trainY = dataX[:num_train_samples,:,:], dataY[:num_train_samples,:]
valX, valY = dataX[num train samples:,:,:], dataY[num train samples:,:]
trainX.shape, trainY.shape, valX.shape, valY.shape
((11184, 1, 6), (11184, 24), (2766, 1, 6), (2766, 24))
```

The function shown here takes the data along with the lengths of the two **windows**—h, in your case, equals 6 and f is 24. The function creates 13,950 samples. In each sample, the input, stored in **dataX**, is an array of length 6. And the output, stored in **dataY**, is an array of length 24.

The three models that you will test are, first, a plain vanilla dense neural network with a 16-unit hidden layer and a ReLU activation function.

```
model dense = Sequential()
model dense.add(Flatten())
model dense.add(Dense(16, activation="relu"))
model dense.add(Dense(24))
```

The output layer uses 24 units to produce a 24-hour prediction. The second and third models are a simple RNN and an LSTM, each with 16 internal units.

```
model simprnn = Sequential()
model simprnn.add(SimpleRNN(16, input shape=(1,6)))
model simprnn.add(Dense(24))
```

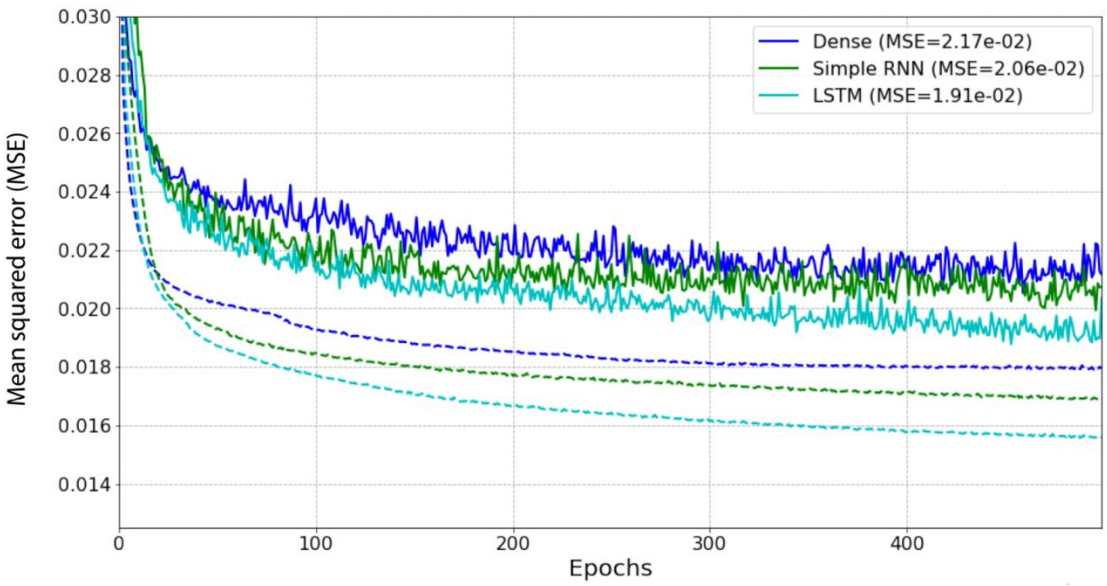
```
model lstm = Sequential()
model lstm.add(LSTM(16, input_shape=(1, 6)))
model lstm.add(Dense(24))
```

In every case, the model was trained on the mean squared error loss function.

```
model.compile(loss='mse', metrics=["mae"])
```

```
history = model.fit(trainX, trainY,
                    epochs=500,
                    batch size=32,
                    validation data = (valX, valY) )
```

You trained for 500 epochs with a batch size of 32. And is the result.



The smooth dashed lines are the training loss for the three models, and the noisier solid lines are their respective validation scores. In this case, the

dense network actually did quite well, almost matching the simple RNN in performance. It ended up with the validation loss of 2.17 times 10 to the minus 2, compared to the simple RNN's 2.06 times 10 to the minus 2. But the LSTM beat them both, with a score of 1.91 times 10 to the minus 2.

Clearly these predictions can be improved by incorporating additional information and neural networks offer a means for doing this. Things like the flows from adjacent stations, traffic speeds, and also time of day and weather conditions, can all be put into an extended input vector in order to improve the model.

Video 7: Neural Networks for Regression

As you have seen time and time again, a good machine learning idea for classification can also be used for regression and vice versa. That means you can also do regression using neural networks. The only difference is that at your output layer, rather than having a sigmoidal or a softmax activation, you instead have a '**None' activation**. That is, the final neuron simply returns the **weighted sum** of its inputs, plus an **offset**. In other words, your final stage is simply a linear regression on the features generated by the bottom of the neural network.

So, as a practical example, recall the housing dataset.

LotArea	TotalBsmtSF	GrLivArea	WoodDeckSF	SalePrice
2,500	910	1,826	0	155,000
6,000	789	789	0	115,000
11,198	1,122	2,504	144	325,000

3,182	600	1,200	0	151,000
17,104	654	1,496	100	179,665

A selection of some rows and columns from that dataset is as shown. Suppose you want to predict the sale price from the other 79 columns, of which you are only showing four here. After dropping all of the non-numeric columns, you are left with 37 features.

df = df._get_numeric_data().dropna()

And after dropping all rows with missing numeric values, you are left with just over 1,000 rows.

So next, you split your data into a training and a validation set using the code shown.

```
from sklearn.utils import shuffle
df = shuffle(df)
training_set, dev_set = np.split(df, [121])

training_features = training set.iloc[:, :-1]
training_observations = training_set["SalePrice”]

dev_features = dev_set.iloc[:, :-1]
dev_observations = dev_set[“SalePrice"]
```

This gives you a training set of 1,000 samples, and a **dev_set** of 121 samples. Since the **SalePrice** is the last column in the dataset, this code shown puts all the training and **dev_set** features into separate variables,

and it also puts the observed **training_set** and **dev_set** prices into their own respective variables.

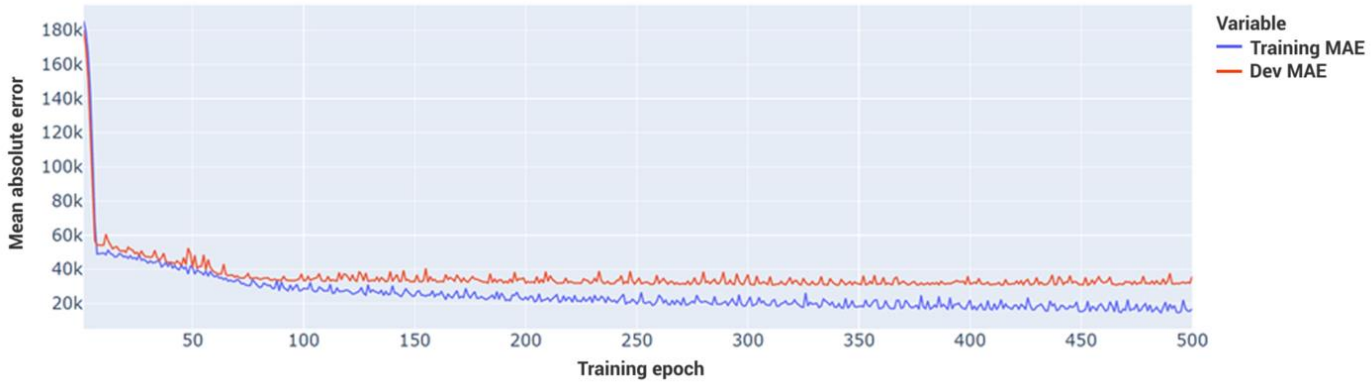
So now that you have nice clean data, training a neural network on this is just as simple as creating a model with the sequential API, where the final layer now has an activation of None.

Note that you can also simply not provide an activation and the default of None will be used.

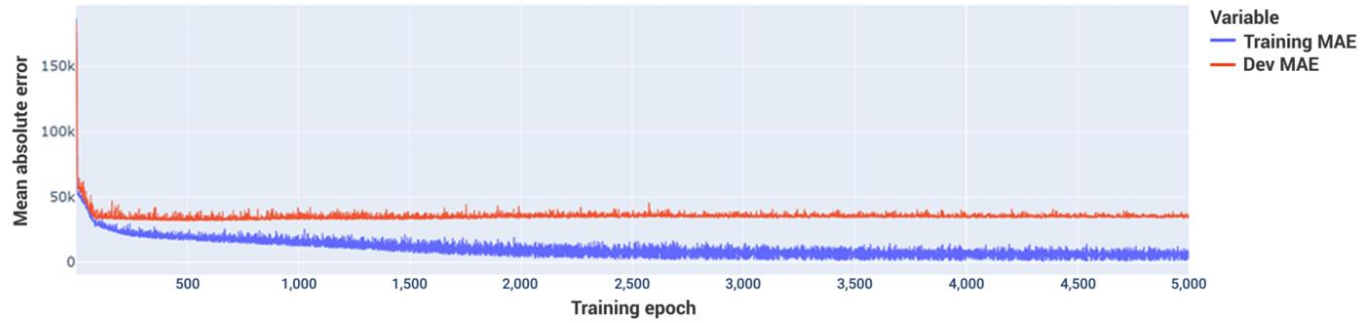
```
house_model = keras.Sequential([
layers.Dense(64, activation="relu"),
layers.Dense(64, activation="relu"),
layers.Dense(64, activation="relu"),
layers.Dense(64, activation="relu"),
layers.Dense(1, activation=None),
])

house_model.compile(optimizer="rmsprop”,
                    loss="mse", metrics=["mae"])

house_model_history = house_model.fit(training_features,
training_observations,
    epochs = 500, verbose = 0,
    validation data = (dev features dev observations))
```

So here the architecture is entirely arbitrary.

The resulting mean absolute error (MAE) is as shown.



You see that, in other words, your model—after a lot of training epochs—is able to get within $30,000 of the price on average for unseen houses in your validation set. After training out to 5,000 epochs, you see that the model does not appear overfit, even given a pretty long time to train.



It is possible that if you increase the number of training epochs, you might actually begin to see a rise in the dev MAE. And actually, it is possible that even beyond that, you would start to see the dev MAE drop again, because of the double descent phenomenon. However, this arbitrary model architecture is not complex enough to start memorizing the training set. So that would tend to yield this mean absolute error very close to 0 on the training set. It just does not look like it is getting there.