Module 23: Deep Neural Networks 2

Video Transcripts

Video 1: Introduction to Convolutional Neural Networks (CNNs)

In this and the next module, we will learn about convolutional neural networks. As you heard in the previous module, CNNs took the machine learning world by storm when they became dominant in image classification competitions, starting in 2012. But before diving into CNNs, let's see how you might approach image classification using only dense layers. This actually works pretty well for small and simple images, such as the pictures of handwritten digits. Here's how you would proceed. First, you'd flatten the image, meaning that you would break it up into individual pixels and arrange those pixels into an array. Then you would put the array through one or more dense layers that would create connections between all of the pixels. 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. This doesn't seem like a good approach. If someone showed you a photograph and asked you what number it shows, your first instinct would not be to cut up the photograph into tiny little squares and arrange them into a long line.

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 fashion. 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. The “everything affects everything” nature of dense layers does 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. And this is exactly what convolutional networks do.

So let's see how they work. The core operation of a CNN is the convolution. We've already encountered 1D convolutions when we did trended time series data. The convolutions we will do 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. Here, a 2D matrix. But later we will encounter 3D kernels. A 2D convolution can be done on a patch of pixels that is of the same size as the kernel. Here we have a 3 × 3 patch of pixels. And taking the convolution on that patch with the filter produces a single number, c. 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.

Let's do an example. We 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. That is, 1 plus negative 0.4 times negative 1 plus 0.2 times 2, and so on. That all adds up to negative 2.2, which is a negative number. So the ReLU function discards it and the result is 0. That's the convolution for a small bit of an image.

To process an entire picture, we 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. For example, this filter highlights vertical lines. From now on, we 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. And 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. In Keras, this is controlled with the padding parameter, which can be set to valid if no padding is desired, or same if we want to include the padding.

Another important parameter of the filter is the stride. Previously, I had said that we would compute the convolution on all possible patches, but this is actually not necessary. You can also have the convolution advance in strides that are larger than a single pixel. So, for example, a stride value of 2 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. And the kernel weights and bias value are learned through training.

Here we see the convolution of an image with a filter in a 3D perspective. The wall of blocks on the left represents the image. Again, we 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, we 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. We 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, we 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, we will apply a 3D kernel.

In this picture, we 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. We again repeat this for several different feature blocks to form a new convolution layer. And then, again, we 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, we 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 we then use as inputs to a multinomial logistic regression at the end. And what I find amazing about all of this, is that at no point did we 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.

In the next video, Josh will explain how we can create neural networks that do amazing things, using relatively little data, by leveraging a previously trained network. And I'll see you later.

Video 2: Neural Networks for Vision (Part 1)

Now let's look at how we can create convolutional neural networks in Keras. Due to the lengthy training time for these networks, I'm not going to do this demo live. However, the notebooks will be provided, so you can run them on your own and experiment.

For this demo, we'll be using the MNIST dataset, which contains 28 × 28 images of handwritten Arabic numerals. For example, here we see one image from this dataset, which represents somebody somewhere who wrote the number three. We're 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 we'll see that a convolutional neural network can do better. Let's first review how we 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. The second is using the more complex functional API. Both models do the exact same thing, but it's good to see both. Now after creating this model, we compile, and we fit it as usual. And we see, when we look at the summary, that this dense network has 50,890 parameters. Now after five epochs, these parameters are fairly good, yielding an unseen development set accuracy of just above 97% after five training epochs. But we can do a much better job with a convolutional neural network. As Gabriel mentioned, a dense network like this has to relearn every feature and every location. So, for example, imagine we have a 3, off in a position of the image that hasn't 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 for reasons that Gabriel described. But what did they look like in code in Keras? Well, here's an example of some code that creates a convolutional neural network. Note that the input_shape is optional to provide, but I've done so here to emphasize that our model will be working on two-dimensional data. Now, that is in contrast to our dense model from before, which used just a one-dimensional version of its inputs. The resulting model has the same architecture that Gabriel showed in this figure. So, in my case, I have three convolutional layers and two max pooling layers. Each of my pooling layers downsamples the features by a factor of two, and it selects the largest value from among each block. In my case, I selected k1 equal to 32, k2 equal to 64, and k3 equal to 128. In other words, as we go deeper, each convolutional layer is learning a larger number of features.

These choices, I should note, are entirely arbitrary. And selecting the best network is hard. You'll 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 my case, I picked these specific features based on the book, *Deep Learning with Python*, by Francois Chollet. And this is an amazing book, by the way, and I very strongly recommend it as a follow-up or a companion to our course, if you'd like to learn more about modern deep learning practices.

We can use the model summary feature to understand each of the layers in this convolutional neural network. We 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 300 parameters. Why 320? Well, since each of our 32 filters in our convolutional layer is a 3 × 3 grid of numbers, we have 3 times 3 plus 1 times 32 parameters. And the extra plus 1 is from the bias term for each of the filters. To test your understanding, you might try to work out why the second convolutional layer has 18,496 parameters, as well as why the final convolutional layer has 11,530. Note that there are no parameters for our max pooling and our 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, we see that our model has 104,202 parameters, which we'll learn by doing a gradient descent on the cross-entropy loss of our training set.

Before we train our network, I'd like to show you an alternate way to specify this exact same network, but now using the functional API. Now as before, this model does the exact same thing as its sequential API counterpart. I'm showing both because you may see both of these in the real world. We can also visualize our neural network using the keras.utils.vis_utils.plot_model function. The resulting visualization is not as beautiful as Gabriel's, but it can be helpful for understanding a network, especially complex networks that are actually graphs, rather than sequential pipelines.

Training a convolutional neural network is exactly like training a dense neural network. We run the code shown. On the Google Colab instance that I use, this takes a little over a minute. If you try to run this code and it is significantly slower, make sure that you're using a GPU. That is, that we've selected GPU as a runtime type. This time, plotting the training and validation set accuracy, we see that we get much better performance than our dense network. I'm getting 99.3% accuracy, or an error rate of only 0.7%. That's not bad for just a few lines of code. Now it is possible to do a little bit better, but we're going to move on to a more difficult image processing problem next.

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 kaggle.com/c/dogs-vs-cats. Let's try to build a binary classifier that takes an image and tells us whether it is a dog or a cat. We're going to assume that every image includes either a dog or a cat. In other words, our model doesn't have to worry about the possibility that the image contains neither or both.

We can first try out the exact model that we used on the MNIST dataset. The difference here is that, rather than having a ten-way softmax that outputs ten probabilities, one for each written digit, we'll instead just have a single sigmoidal neuron that gives a probability that the image contains a dog. After training our network, we see that it is able to get great accuracy on the training set, in fact near 100%, with that same model architecture. But if we look at our development set, we see that it's unfortunately overfitting, getting only a bit worse than 70% accuracy on the validation set. That's pretty bad. I mean, it's doing better than chance, but it clearly has a long way to go.

So let's see what happens if we add a couple of additional convolutional layers and max pooling layers. After training this deeper model, we see that the deeper model gets slightly better validation set accuracy, but it's not that much. So what do we do next? Well, there are so many tricks out there that can be applied. For example, one technique is data augmentation, where we'll take our original dataset and we'll add a bunch of distorted, rescaled, rotated versions of each image that's in the training set. And another trick you can use is dropout, where we randomly disable connections between some neurons. And in fact, there are dozens of such tweaks that you can apply and you'll take a brief look at a couple of these tricks in the homework.

For our purposes in this video, however, I'm 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 pre-trained network. So to set the stage for this next very, very important big idea, remember how all those different researchers trained neural network models to solve ImageNet? Well, it turns out that we can use those networks to solve new image recognition problems other than ImageNet.

Now to understand how that's possible, let's 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. And it was one of the strongest entrants in the ImageNet contest that year. As part of the Keras library, there are a bunch of pre-trained libraries, including a copy of that VGG16 model. So the code shown, just this line of code, it loads the VGG16 model. And after loading their model, we can look at a summary of it. It's just a model like ours. Now we see that their model is actually huge. It's 138,350,544 parameters. And it consists of a large number of things we've seen, convolutional and max pooling layers. Now, as with the models that we built ourselves today, we see that the convolutional layers in VGG16 have an increasing number of filters as we move towards the top of the network. Now, just a quick note that the input to a neural network is considered the bottom and the output of a neural network is considered the top.

Now 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, that's actually the bottom. And then the last layer, called block5_pool, that's the top. Now observe that, as we move towards the top, we see these max pooling layers that downsample the features, just like our network. But unlike our network, the max pooling layers, they only appear every few convolutional layers. Interesting. Well, at the very end of the network at the top, we see that the output of the convolution and the max pooling layers gets flattened into 25,088 features. And that's similar to our model, where the convolutional part of our network got eventually flattened into 1,152 features. Now, unlike our model built on MNIST, which fed our 1,152 convolutionarily-generated features straight into a ten-way softmax, their VGG16 network, it actually has two additional dense layers. And then, finally, a 1,000-way softmax. Note that this final layer is a 1,000-way softmax because, in ImageNet, there's a 1,000 different things to be classified.

So this raises a really interesting question. How did VGG16's designers know to use this specific set of layers or sequence of layers? Well, they didn't. Keep in mind that at the time this model was generated, back in 2014, there were hundreds or maybe even thousands of research teams that were all experimenting with different neural networks to find the ones that would perform best on ImageNet. And this one just happened to be one of the best. Now certainly that team's choices were guided by theoretical concerns but, in the end, the only way they knew that their architecture worked, was to try a bunch of experiments. So how does that help us? 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. So what are those? Well, those features represent the presence of 512 different features in each of the 49 regions of the image. And we'll come back later to what these features might mean.

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. Now, by the way, I did not know the word 'tench' either. That's apparently just some kind of obscure fish that was used as the zeroeth class for ImageNet. So this second chunk of the network, consisting of two dense layers and a softmax, it doesn't 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, we expect that these features will have nice properties, like being able to recognize objects, even if they're in somewhat different regions of the image, where that object never appears in the training set. It's also...another way of putting that is translational invariance. Now with this idea of the network having two primary stages in mind, I claim that the VGG16 model is useful for our dog cat classification task. Now that's 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 which case my claim would be trivial. Now in fact, I'll go further than my claim about seeing dogs and cats. I also claim that VGG16 would be useful for identifying any number of objects that are not in the ImageNet dataset. Like, for example, waterfalls, or fire, or horses. Those weren't in

ImageNet, but VGG16 would be useful. For example, let's 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 here's where the big, very clever idea comes in. The thing I'm really trying to emphasize in this video. So what we're going to do is change the parameter, include_top, to False. So when we do this, the parts of the network starting with the flattened step, they get chopped off. The top of the network's gone. So in other words, the flattened layer, the last two dense layers, and the 1,000-way softmax are removed. So now if we look at the model summary, we 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 we feed an image into this network that's been truncated, we get back these 25,088 features as a summary of the input image. So how do we actually use this to train a dog cat model? Aha, this is the really cool part. We simply stick our own second primary stage under the end of their existing network to replace what we just chopped off. Now since we 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 we're 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.

Now the resulting model is as shown. 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's an additional 25,089 parameters, for our part of the model, that we'll generate ourselves 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. We're now very close.

But before we train our network on our dog cat dataset, there are two important things we should do. The first thing is that we should freeze this VGG16 network. That is, when we perform our gradient descent, we should not mess with the weights of the network that generate the 25,888 features. That part of the network is already great. And if we include that in the gradient descent, we’ll be undoing all of Karen and Andrew's hard work. Now to do this, we set conv_base.trainable equal to False. After doing this, if we look at our summary, we see that our model now has only 25,089 trainable parameters, and the others will be left alone during the gradient descent process. They're frozen.

There's a second thing we should do. Now this one is not as important as freezing the pre-trained network, but it can make a significant impact on model accuracy. So what is that second thing? Well, it turns out that the VGG16 network was trained with the data in a very specific, somewhat non-standard format. For example, the blue channel came before the red channel in that dataset. However, in our setup, the red channel is first. So luckily a function exists in Keras that can manipulate our 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 our neural network by using the functional API. And so I'm showing that version here.

If you're a little confused, take a moment to pause the video and compare the functional API implementation of this network with the sequential API implementation that we were just discussing. As always, the functional and the sequential API implementations, they result in the exact same network. So now if we look at the resulting model summary, the first three layers represent the pre-processing. So don't worry too much about what these steps in the network mean exactly, the whole idea of this applications.vgg16.preprocess_input function is that we don't want to worry about it, so we can ignore those.

Now that we've done those two important things, we simply train the model as usual. Recall that the convolutional neural network that we trained, totally from scratch, was only able to get 70% accuracy on the validation set. By contrast, our new model, which leverages the existing VGG16 model to featurize each image, is able to get 97% accuracy on the validation set. That is just an absolutely gigantic improvement. So you can see why this technique was worth seeing. 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's because the bottoms of these networks can featurize data in a way that is incredibly useful for all kinds of tasks.

And in many contexts, it's 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 we can squeeze out just a bit more performance by fine-tuning our model. Now a warning in advance that for our specific problem here, fine-tuning is only going to make a tiny difference, but in some cases, it can help a lot more. So that's why I'm showing it to you. So fine-tuning a model is a relatively minor tweak to our workflow. After we train our combined model, we'll then unfreeze the last few layers of the huge model that we're using as a base. So for this example, we'll unfreeze the last four layers of VGG16. We'll then train again, jointly train the parameters of these last few layers of their network, along with 25,089 parameters of our sigmoidal neuron.

Note that this model summary shown, it only lets us see the trainable parameters and the convolutional base. And it doesn't show the 25,089 parameters of our sigmoidal neuron, which will also be trained. So in other words, after we unfreeze the last four layers, the last 7,079,424 parameters that are part of the last three convolutional 2D stages, they're going to be part of our gradient descent process. So why did I pick these three layers specifically? Well, I should acknowledge that, again, I'm inspired by the book, *Deep Learning with Python*, by Francois Chollet, which contains a somewhat similar exercise to what we're doing now. And let me again say that I highly recommend this book. Now, conceptually, the reason why we only unfreeze the topmost layers is that the bottom layers capture something more universal about all image classification. Whereas the top few layers, they're conceptually higher-level features that are more likely to be usefully informed by our dataset. Now, I will acknowledge that is a vague explanation. But the philosophy and intuitions behind much of the art of deep learning can be tricky to follow until you have a lot more experience.

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're not trainable. So it's sort of a quirky piece of syntax. Just note that's how it's done. So now the third and final step in our fine-tuning process is to call fit, but on our existing already trained model. So in this example shown, I set the learning rate to a smaller value. And the idea here is that we don't want to make large gradient descent steps, because we're just trying to eke out a bit more performance. So the first figure that we see is as shown. It shows the accuracy of our VGG16 base model, when we train it from scratch.

Now the second figure we see gives us the evolution of the accuracy after we fine-tune. Now, as I warned you earlier, we see that in this case, fine-tuning has only a tiny impact, bringing us from 97.4% to 97.5% validation set accuracy. Now I'll 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, I should note that you will find that in some real-world problems, that fine-tuning, where you unfreeze the base you're using, it can give you a small, but sometimes really significant, performance boost.

Video 5: Interpreting Neural Networks for Vision

Throughout the last video, I mentioned that models like VGG16, they 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 exactly, and how can we 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. In case you're curious, he searches the space of images by conducting a gradient descent on the input, instead of on the model. But that's beyond the scope of our class. So what do we do after identifying an image which highly activates a target feature? Well, since the image highly activates that feature, we can think of our network as trying to identify images that look like the image we've generated. Note that it's possible that there maybe are several completely different-looking images, which might activate a particular feature of a neuron. And if that's the case, Fabio's procedure will identify only one such, what I'll call an exemplar, image.

So let's see an example. The image I'm showing here, it highly activates the 12th filter of one of the early layers of VGG16. Whereas this 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, it's looking for images with diagonal banding. Whereas the 110th filter, it looks for what I might describe as 90° edges. Now as we go deeper into the network, the textures that activate the filters look increasingly complex. For example, the image shown, it highly activates the 190th filter of a later layer of VGG16. As an example, we 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.

Now going even deeper into the model, we see more interesting exemplar images. For example, this image, it 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, I think it looks like chains. Now keep in mind, that's just one example of a potentially huge number of images that highly activate this filter.

So, as an experiment, after identifying an exemplar image, you can go grab a real-world image that you think resembles that generated exemplar image. So if we want to test the hypothesis that this given filter is a chain detector, we can do that. So, for example, Fabio does an experiment where he feeds this image shown of a chain and he finds that, indeed, filter 265 in the final layer is actually highly activated. Now that's true, even though the exemplar image that he generated is a more complex linkage of chains. It's some kind of chain recognizing neuron. While generating such exemplar images may seem like a purely aesthetic pursuit that we're doing just for fun, this sort of thinking is actually useful. Notably, it gives us a way of interpreting our neural networks. So even though VGG16 has tens of millions of parameters in its convolutional base, we can get some sense of what it's 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's 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, it 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. So, for example here, altering the single pixel makes the network decide that rather than being most likely a teapot, with 25% confidence, that this is, in fact, a joystick, with 37.39% confidence. So in a sense, what we've identified amounts to an optical illusion, to which our neural network is vulnerable.

To me, it is fascinating that models like VGG16 perform so 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's not clear to me that this fact has any practical consequence. But it does suggest that maybe these networks are not so smart as they might appear. And I'll note that area remains an open topic of research.

Video 6: Neural Networks for Text or Time Series

We've seen how, with CNNs, we adapted the architecture of neural networks to work with data of a particular type, namely images. In this module, we will learn how to create neural networks for processing sequential data. We've encountered sequential data before when we 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. These models and their variations are a topic of intense current research. Here we will go over the fundamental ideas, in order to give a sense of how they work.

Let's say we have a sequential dataset consisting of a long list of numbers or words. And our goal is to guess the next few items in the list. We'll need some notation. Let's use k to denote our current position in the list. Then k plus 1 is the upcoming position and k minus 1, k minus 2, et cetera, are points in the past. We will base our prediction on a finite history, let's say, for the sake of the example, the most recent three data points. I 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 we are currently focused on. We will denote the prediction that we make at time, k, with y_k, which may also be a list. Our goal will be for our prediction, y_k, to match the ground truth as closely as possible.

Let's first consider a naive approach in which we feed the historical data into a plain vanilla dense neural network with y_k as the output. This is similar to what we tried when we introduced 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. And so at time, k, we also have available the previous prediction from y_{k-1}. That prediction was made on the basis of data from k minus 3 to k minus 1. So if we include y_{k-1} in our model, then we implicitly gain access to X_{k-3}. And thus, we extend our historical information by one data point. But the argument goes further, because y_{k-1} was itself computed using y_{k-2}. And so it includes information about X_{k-4}. And so, ad infinitum.

Simply by including the previous prediction in our model, we automatically gain implicit access to all of the historical data, without having to store it in memory. This is not a new idea. We 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's a graphical representation of the idea. 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 we initialize an RNN, we must populate its memory with some initial state. And we will typically set it all to zeros or to some random noise. Then as we feed information through the network, such as text or music, it will continually update its state, much as we do when we read a book. We hold in our 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 our comprehension of the text.

Here we 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 we call C. This state is not exported from the cell. It's a so-called hidden state, but it serves as a repository of long-term information.

Let's see how it works. In this diagram, you'll 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. Similarly, σ_2 regulates the amount of the input, U, that is added to C at every time step. 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. This model is much more complicated than the simple RNN we saw previously. Each LSTM cell contains within it four dense layers and two memory states.

Next, we 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 we 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. And our 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 we've seen, has to do with how the data's prepared. The single long array must be split into a series of historical and forecasting windows. The function shown here takes the data along with the lengths of the two windows—h, in our 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 we will test are, first, a plain vanilla dense neural network with a 16-unit hidden layer and a ReLU activation function. 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.

In every case, the model was trained on the mean squared error loss function. We trained for 500 epochs with a batch size of 32. And here we see 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. In this rolling horizon animation, we see on the left the 6 hours of input data. This is followed by the 24-hour predictions for the three models. The light, dashed black line is the ground truth. 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.

So that's it for recurrent neural networks. This is a very large topic and we have only scratched the surface here. But with this introduction, you will hopefully be able to understand the use of RNNs in other contexts. In the next video, Josh will give us a more in-depth look at the use of neural networks for regression problems.

Video 7: Neural Networks for Regression

As we've seen time and time again earlier in our course, a good machine learning idea for classification can also be used for regression, and vice versa. That means we can also do regression using neural networks. The only difference is that at our output layer, rather than having a sigmoidal or a softmax activation, we instead have a None activation. That is, the final neuron simply returns the weighted sum of its inputs, plus an offset. In other words, our 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 that we saw way back in an earlier module. A selection of some rows and columns from that dataset is as shown. Let's suppose we want to predict the sale price from the other 79 columns, of which I'm only showing here 4. After dropping all of the non-numeric columns, we're left with 37 features. And after dropping all rows with missing numeric values, we're left with just over 1,000 rows. So next, we split our data into a training and a validation set using the code shown. This gives us a training set of 1,000 samples, and a dev_set of 121 samples. Since the sale price 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 we 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.

So here the architecture that I picked was entirely arbitrary. Now the resulting mean absolute error is as shown. We see that, in other words, our model, after a lot of training epochs, is able to get within $30,000 of the price on average for unseen houses in our validation set. After training out to 5,000 epochs, we see that the model does not appear to overfit, even given a pretty long time to train. Now I should note, it is possible that if we increase the number of training epochs, we might actually begin to see a rise in the dev_set mean absolute error. And actually, it's possible that even beyond that, we would start to see the dev mean absolute error drop again, because of the double descent phenomenon we discussed in an earlier module. However, my suspicion is that this arbitrary model architecture, it simply isn't 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 doesn't look like it's getting there.

Now I should note that there are many techniques we could try to use to reduce overfitting, to improve generalization, and improve training time. So those include ideas like dropout, standardizing our data, tweaking the model architecture, and more. So we'll not do so here, but you'll have a chance to try this out on the homework.