openSAP
Enterprise Deep Learning with TensorFlow

Week 4 Unit 1

00:00:07 Hi, and welcome to Week 4 of the course "Enterprise Deep Learning with TensorFlow". My name is Christian Reissweg, I am a senior data scientist at the Deep Learning Center of Excellence

00:00:16 in the Innovation Center Network of SAP, and I will lead you through this week of the course.

00:00:21 Last week, we covered deep networks and sequence models, and this week we will talk about convolutional networks.

00:00:29 I will start with an introduction to convolutional neural networks in this first unit. In particular, we will discuss the biological inspiration that is behind convolutional networks.

00:00:40 We will look at challenges for computer vision, why the problem is hard. We will discuss the need for spatial invariance,

00:00:48 and also have a look at a naïve approach to how a neural network could be built for computer vision. And finally, this leads us to convolutional neural networks.

00:01:00 So let's have a look at how nature does things. If you have the primate visual cortex, the retina of the eye is connected to the brain,

00:01:10 and the brain has subsequent areas, or a hierarchy of areas that process the signal. And this is exactly what we like to do with neural networks as well.

00:01:20 We like to have the signal coming in and being processed layer-wise in a hierarchical way. Of course, this correspondence is not perfect because in the primate visual vortex,

00:01:32 there are many connections – forward connections, backward connections. Our models will be fairly simple – we really want to stick to a layer-wise approach

00:01:40 where the signal is processed by one layer after the other. In the 50s and 60s, Hubel and Wiesel studied the feline visual cortex,

00:01:51 and they identified two types of cells: "Simple" cells, which fire in response to stimuli of a particular shape and orientation.

00:01:59 For instance, some cells may react to vertical lines, some may react to some diagonal lines with a particular orientation,

00:02:08 and some may react to circles, ellipses, and all kinds of other basic shapes. There are also "complex" cells and they, on top of that, fire when the stimulus moves in a particular direction.

00:02:24 So if you try to recognize a dog in some image, this entails combining many irregular features into a whole.

00:02:33 So what does this mean? If an animal's brain tries to identify whether there's a dog in this image,

00:02:40 then first it needs to identify edges and basic shapes, like the arc forming the outline of an eye. This needs to be combined into an abstract representation of an eye.

00:02:52 You may identify a second eye and other body parts like ears, the snout, legs, and so on. And finally, you arrive at a more global abstract representation of a dog.

00:03:11 The problem with this approach, or the problem in general is that if you just have image data, images are made up of a lot of pixels.

00:03:19 And even if you have a fairly low-resolution image of 512 x 512 pixels with three color channels, you already have in the order of 800,000 features.

00:03:31 Another problem is that we have all kinds of invariances built in. So if you ask: "Does this image contain a dog?",

00:03:39 then a naïve machine learning model would need to learn spatial invariances. It doesn't matter where the dog in this image is located.

00:03:52 Also, the scale doesn't matter – it doesn't matter whether the dog is enlarged or reduced. And also the angle doesn't matter.

00:04:02 And, of course, the orientation of the dog in 3D space doesn't matter either. And even its color or shape, and what parts of the body are actually visible,

00:04:13 and additional transformations, like color transformations, that also need to be taken into account. So let's have a closer look at spatial invariance.

00:04:26 As I mentioned already, where the dog appears in this image is completely irrelevant. The same is also true for all related component questions, like:

00:04:36 Is there a given edge of some given orientation? Is there an eye? Is there a second eye? Does it have legs?

00:04:42 All these kinds of things are invariant to the spatial location. However, the spatial relationships between the various components do matter, of course.

00:04:57 Let's say, for this particular example here on the right-hand side, if you split up this image of the dog into many bits and pieces,

00:05:05 it is very hard to recognize that this is actually a And if you play this game to the max,

00:05:11 then it will be impossible to identify that this image actually contains a dog. Now, a very naïve approach in computer vision for actually dealing with spatial invariance

00:05:25 is to just take one image and augment it by generating many translations of that image. You see here that we just applied some spatial offset and cropped the image in various different ways.

00:05:42 The problem with this approach is that the training time is massively increased. The dataset will be much larger, and therefore also the training time will be much longer.

00:05:51 And still a naïve feed-forward network requires a lot of parameters, because you need to connect to each pixel in the input image.

00:06:02 So let's have a closer look – how would this work? A naïve feed-forward network connects each pixel in the input image –

00:06:11 that is the "image width x image height" number of pixels – to all neurons in the next hidden layer. And so you have a lot of parameters,

00:06:23 namely the number of pixels times the number of neurons in the next hidden layer. A convolutional neural network, on the other hand, operates in a slightly different way.

00:06:37 It also connects pixels to the next hidden layer, but rather than operating on this entire input image, it operates on just a small patch of the input image.

00:06:50 Okay? We only connect this small patch, and that patch is being shifted across the entire image, all the way to the end.

00:07:00 And the parameters between the patches are shared. So rather than having a large set of parameters as in standard feed-forward networks,

00:07:13 multilayer perceptrons, which are fully connected, we just have a reduced set of parameters,

00:07:19 namely the number of pixels that are actually located in the patch times the size of the hidden layer. One other interesting aspect is that, especially in convolutional neural networks,

00:07:32 as you go to deeper layers, the features become effectively larger and more abstract. So a lower-level layer would perhaps cover this kind of region of the image highlighted here,

00:07:46 and the deeper layer would already have a larger field of view and incorporate more global structures and features.

00:07:55 Well, the advantages are perhaps fairly obvious. You have fewer parameters to tune,

00:08:02 the spatial invariance is already mathematically built into the network, so no need for data augmentation,

00:08:08 and we retain spatial relationships between the various features and components. In the next unit, we will focus on this in more detail,

00:08:21 but before I move on to the next unit, I would like to have a first look at our Jupyter notebook. In this notebook, we want to build a classifier for traffic signs.

00:08:35 Let me first show you what the samples look like. These are German traffic signs – in total, we have 43 different types.

00:08:43 That means we have a classification problem of 43 classes. We have these image snippets here.

00:08:50 And the task is to recognize whether this is a stop sign, a speed limit sign,... And this is what the network gets as input, this particular snippet of the image,

00:09:00 and it is supposed to output a particular number, the class number. So before discussing the details coming up in the next few units,

00:09:08 let me first give you an overview of the structure of this notebook and what kind of API we are actually using.

00:09:16 We are making use of the Estimator API. The Estimator API is a new kind of module that allows one to more easily wrap TensorFlow models.

00:09:28 In particular, an estimator wraps the training loop, the evaluation stage, and also the prediction stage. The first thing an estimator requires is a model function.

00:09:40 The model function sets up the network graph, it defines losses and the training operator, it defines evaluation metrics, accuracy metrics,

00:09:49 and it also sets up what the network is actually outputting, or predicting. The model function always looks the same.

00:09:57 It requires as input some features – in our case, those are just images. Also labels – during the training and the evaluation stages, you require labels.

00:10:07 And then there are some additional dictionaries that one can pass in, like the parameters. Those are arbitrary parameters that the user – that's us – can set up.

00:10:17 And also, importantly, we have a mode parameter, which defines whether this is actually being called in training mode, in evaluation mode, or prediction mode.

00:10:27 In prediction mode, we don't have the labels. Now, the estimator wraps – as I mentioned – the training loop, and it takes the model as input.

00:10:40 For actually executing the training, it requires us to set up an input function. This input function is supposed to return a tuple of features and labels in the training case.

00:10:52 In the prediction case, we don't return labels – we don't have labels there. But during the training stage, the input function has to set up tensors

00:11:02 which return the features and the labels. And once we have defined that, we can actually execute the training loop.

00:11:10 Now, there's one caveat: Because everything is now wrapped inside some kind of function call, we don't have access to the loop.

00:11:17 But after or before each training step, or at the beginning or end of training, perhaps we still want to execute certain things that the user defines.

00:11:29 So we can actually build hooks. Let me quickly show you what a hook looks like – I've created a hook in this notebook.

00:11:44 In this example, we have an OutputHook. It's derived from the SessionRunHook, and all it does is... the interface defines basic functions.

00:11:54 Most importantly, so in our case, the functions before_run and after_run, and as the names suggest, those routines are being called by the estimator internally,

00:12:05 before and after each training step is executed. And particularly here, after one training step has been performed,

00:12:16 we are outputting the loss and some accuracy metrics to disk, actually to a file. Then I also created a ValidationHook,

00:12:29 and the sole purpose of this ValidationHook is that every, let's say, 1,000 training iterations, I would like to be able to run the validation loop,

00:12:42 I want to have a pass over the entire validation dataset and see how well my model is performing. And therefore, after each training step,

00:12:52 I'm checking whether it's actually time to run the validation loop, and then I'm just executing the validation loop.

00:13:00 And that's, again, wrapped by our estimator. There's an estimator.evaluate function.

00:13:06 Now, just like the training function of the estimator that takes as input some kind of input function that defines where the data is coming from,

00:13:16 we can do the same for the validation loop. The validation data typically is different, we have a different set of samples,

00:13:24 and that's why it allows you to set up a different input function. That's exactly what we're doing here.

00:13:35 Well, rather than actually having two separate functions, we have a flag indicating whether we are currently training or not,

00:13:43 and we get as input the set of samples, and in the training stage, the set of samples is different compared to the validation stage.

00:13:54 In this particular case, our sample set is just a list of file names. And since this is a classification problem,

00:14:01 and the files on disk are arranged according to class in separate subdirectories, we can just extract the label for a given file – that is, for a given image – from the file name.

00:14:19 Now, then, we have our list of file names and our labels, and we can actually put this into our dataset API and create a dataset,

00:14:30 and by just inputting this list, the list of file names and the list of associated labels, we have a dataset that consists of those particular samples, of file names and labels.

00:14:42 Now this is, of course, not what we desire. We would like to have image data, and therefore we need to apply some kind of mapping function

00:14:49 which maps the file names to actual image data. So our mapping function is essentially just loading the image.

00:14:56 And we can do that concurrently on multiple threads, and ahead of time – there's a buffer – so that we can do it while the training loop is still executing.

00:15:08 Now, I mentioned that I have a switch statement – I decide whether I'm in the training stage or in the validation stage. If I'm training, then I would like to repeat the dataset, or the samples from the dataset, indefinitely.

00:15:19 That's what's being done by the call to the dataset.repeat function. I would like to shuffle the training samples after I've seen all training samples once,

00:15:30 I would like to reshuffle them arbitrarily. And then finally, I want to batch them together.

00:15:36 I want to have groups of samples which are passed as one minibatch of samples to the network. Finally, I need to create an iterator which is actually iterating over the samples

00:15:49 and grabbing samples from the dataset. And in order to actually get the tensors associated to the samples,

00:15:59 I need to call the iterator.get_next function. And then finally, I can return a tuple – features and labels – which the estimator,

00:16:10 together with the model function, can use to execute training or the validation loop, or also the prediction stage.

00:16:19 Now, very quickly, let me scroll down. I have a function where I set up the network architecture – we will have a closer look at this in the next units.

00:16:31 I will have the actual model function which is just calling the architecture function that sets up the graph. The difference is that the architecture function where I set up the graph

00:16:46 is really only setting up the graph and the output, but it's not defining any training operator or any loss function,

00:16:53 and that's what's actually being handled here in this model function. And once that has been defined, I can actually set up the estimator.

00:17:05 And you see here in this call, this is the constructor of the estimator, it's just getting the model function as input,

00:17:11 a certain set of parameters which are passed to the model function, and some additional run configuration parameters

00:17:20 which define the output directory and how often we would like to checkpoint. Yeah, and then finally, we can execute the training.

00:17:32 But all that is for upcoming units. Let me switch back to the presentation – that concludes this unit.

00:17:40 In the next unit, we will discuss "CNN Architecture, Part 1". This means we will have a closer look at convolutions,

00:17:46 how those convolutions and their kernels work in detail. We will discuss non-linearities that we have in our neural network.

00:17:54 And finally, also have a look at the important topic of weight initialization. Thanks a lot and see you in the next unit.

Week 4 Unit 2

00:00:07 Welcome to Week 4, Unit 2: "Convolutional Neural Network Architecture, Part 1". In the last unit, we were inspired by biology and how nature is doing things,

00:00:18 we looked at the need for spatial invariance, and we conveyed the basic idea of convolutional neural networks.

00:00:24 In this unit, we will cover aspects of a convolutional neural network architecture. Especially in this first part, we will cover convolutions in more detail,

00:00:34 we will look at non-linearities – activation functions which are located in between convolution layers – and we will have a look at weight initialization.

00:00:45 Let me start by discussing convolutions. So here on the left-hand side we have some input image,

00:00:55 on the right-hand side we have some output image, and each pixel in the output image is connected to a 3x3 patch in the input image.

00:01:05 So this one particular red pixel here on the output image is a weighted sum of those nine input pixels. And this is our set of weights.

00:01:21 And we say that this particular patch, this 3x3 region in the input image, is the field of view for that given output pixel.

00:01:30 Now, if we slide the same set of weights across the entire input image, all the way to the end, we arrive at our full result on the left-hand side,

00:01:44 and this is actually what imparts spatial invariance to the network. We're recycling, we're reusing the same set of weights across the entire input image

00:01:52 to produce some output image. And, of course, this reduces the number of weights because in a naïve approach,

00:01:58 we would have a separate weight for each input pixel to each output pixel. Now, what we've just seen is a "valid" convolution.

00:02:08 You notice that the output image has a reduced resolution, because we have to somehow deal with the boundary points.

00:02:20 In order to actually have the same output resolution, we could just add zero padding to our input image, and in TensorFlow, this is known as a "same" convolution.

00:02:32 Now, convolutions don't just work in 2D – they also work in 1D. For instance, for natural language processing tasks, we can represent a sentence in terms of characters

00:02:43 where each character corresponds to some one-hot vector, and we can just slide a set of weights over those characters to produce some output.

00:02:55 Audio signals are also just one-dimensional, and we can apply the same scheme here. This also generalizes to 3D.

00:03:02 In 3D settings, a prominent example would be a stream, a video signal, a stream of images, and here we would not just slide over individual pixels in the input image,

00:03:13 but also across the time direction of the video signal. There are various different kernel shapes,

00:03:22 and the most popular kernel shapes that we have are those 3x3 kernels that we've just seen. We can also have a 1x1 kernel – it will become clear shortly why we actually want to have such a kernel.

00:03:36 We can have asymmetric kernels. We can have kernels that are shifted with a stride of 2 over the input image.

00:03:45 This effectively means that rather than shifting one pixel to the next, one pixel to the right, we shift the input patch every second pixel to the right,

00:03:55 and this effectively gives us a downsampling operation. We can also have a dilated kernel.

00:04:02 This means that rather than having immediate neighbors as stencil points in our kernel patch, we have, for instance, every second pixel in our set of weights.

00:04:16 And this has the effect that the field of view is enlarged, but we still only have nine weights. We only sum across nine of those input pixels in this field of view here.

00:04:28 So the same computational complexity, but increased field of view, so this could be a desirable property.

00:04:36 Also, we have transposed convolutions, which are also known as "deconvolutions", or "fractionally strided" convolutions.

00:04:44 Here, the idea is that they effectively reverse, or undo the effect of downsampling. So going from some coarse input resolution image to a high input resolution image,

00:04:58 doing some upsampling, learning some upsampling operation. And internally how this is done is you just place immediate pixels which are just set to zero

00:05:10 between the pixels, the original coarse pixels. Now, convolutions don't just operate on the pixel neighborhood like a 3x3 kernel

00:05:24 looking at the neighboring pixels. You're not just summing across that, but also you sum over input channels.

00:05:31 An RGB image, for instance, has three input channels, and you can also sum across those input channels.

00:05:38 And now it becomes obvious why a 1x1 convolution makes sense: Because we just sum across all input channels rather than neighboring pixels.

00:05:48 So in total, this means that our weight matrix is now of rank 3. We have the x and the y directions that we're connecting, and we also have the depth of the channels.

00:06:02 Okay. We don't just have one set of weights as output,

00:06:11 but we can also have a multiple set of weights that correspond to multiple output channels, and this makes up our complete convolution.

00:06:21 So we have some set of input channels, we sum across those input channels and the pixels in the neighborhood of our current central pixel,

00:06:29 and we can have a separate set of weights and repeat that operation multiple times to produce multiple output channels.

00:06:37 And this is, of course, wrapped just in one single convolution. Then we can chain those convolutions to build a deeper network and have multiple layers.

00:06:49 We can go deeper. One interesting property that arises while doing that is that the learn filters, the learn features

00:07:01 respond to various different patterns. So the first layers right after the image respond to fairly low-level features,

00:07:13 just very basic local texture information. The mid layers, they already have some kind of more local context – not just very local,

00:07:21 but a little bit more global information already. You see that here in this middle image.

00:07:27 And then finally, the higher levels are even more global and they incorporate more complicated structure, they respond to more complicated structure and patterns.

00:07:37 And that's exactly what we were aiming for and hoping for. As we go deeper, the network is building more complex shapes,

00:07:44 it's reacting and responding to more complex shapes, to a hierarchy of more complex shapes. Now, if you look at what a single pixel in some layer is actually seeing,

00:08:00 or what the field of view with respect to the input image is, you notice that even though we always only have 3x3 convolutions going from layer to layer,

00:08:13 in the input image, the field of view is effectively increased. And this allows the network to actually incorporate more and more global context.

00:08:28 So right now, we have a network which consists just of convolutions. This is, of course, not so desirable because this network will only be linear.

00:08:37 Linear networks don't have that much expressive power and can't fit any non-linear function. We need to introduce some non-linearities.

00:08:46 And the very well-known non-linearity that is used very widely nowadays is the "rectified linear unit". And I've just inserted this rectified linear unit right after each convolution as an activation function.

00:09:03 And in most frameworks and especially also in TensorFlow, this is actually just directly wrapped inside a single layer.

00:09:12 You don't need to specify that extra – you just write down the convolution and you can specify the activation function that this convolution layer should call.

00:09:24 Now, what does this activation function look like? On the left-hand side, you see a plot of a rectified linear unit.

00:09:33 On the negative x axis it's just 0, and on the positive x axis it's just a linear function. And the property of that is it does not saturate on the positive x axis.

00:09:45 If you look on the right-hand side, on the classic activation function, a Tanh activation function that was typically used in the past,

00:09:54 you notice that on the far right, for x, for a large x, or for a very small x in the negative regime, the activation function saturates.

00:10:04 That means that the derivative approach is 0. A zero derivative is not so desirable because when you learn in the network,

00:10:11 you apply the backpropagation algorithm. You will also multiply with the derivative of the activation function, and if that is close to 0,

00:10:20 then the product becomes very small and the learning signal becomes very small. And this is known as the "vanishing gradient problem".

00:10:28 Now, another nice property that the rectified linear unit induces is actually sparsity. However, the rectified linear unit has one problem:

00:10:37 On the negative x axis, it's just 0, and once some signal is pushed into that regime, it's hard for the network to recover from there because the derivative is 0 there,

00:10:52 so this unit may never activate again, the neuron associated with this non-linearity may never activate. And this is known as the "dying ReLU problem".

00:11:04 To circumvent this, you can come up with the leaky rectified linear unit, especially the parametric leaky rectified linear unit,

00:11:13 which allows you to have a non-zero slope also on the negative x axis that can even be learned from the network during training.

00:11:23 Now, finally, let's move on to the topic of weight initialization. Typically, you need to randomly initialize the weights in a neural network,

00:11:31 but they can't be arbitrarily initialized. You want to make sure that the signal is neither amplified nor damped when you apply

00:11:44 those randomly initialized weights to some kind of input distribution. On the right, you see an example of that.

00:11:49 You have some input distribution with a given variance and a given mean, you apply the action of the weights, which could be a convolution,

00:11:57 it could also just be a metrics multiplication in a fully connected layer, and if the weights were initialized such that the signal is amplified,

00:12:07 then you get an output distribution which has a different variance and a different scale. Now, if you apply this multiple times throughout the entire network,

00:12:19 then what happens is that the signal will get amplified as you move deeper into the network. And this is undesirable because, especially in the early stages during training,

00:12:28 it means that your network training becomes unstable and perhaps forces you to use a smaller learning rate than would have been necessary otherwise.

00:12:39 Vice versa, if the action of the weights is such that it actually dents the input signal, then learning may not progress very well at the beginning

00:12:50 and you make slow progress during network training. So ideally what you would like to have is to initialize the weights such that you get

00:12:58 some output distribution which has identical variance to the input distribution. And that, of course, is a property that depends on the number of input and output connections,

00:13:09 and also on the particular non-linearity that is being employed. And especially for the rectified linear unit,

00:13:16 one very well-known method is the "He's method", developed by Kaiming He. It's also known as the Gaussian initialization,

00:13:25 or in TensorFlow, it's called the "variance scaling initialization". And what you're doing is you're randomly sampling the weights from a Gaussian distribution

00:13:37 with a particular variance, and the variance is fixed by the square root of 2/N, where N is either the number of input connections or the number of output connections

00:13:46 or the average of the two. Now, before we conclude, I would like to actually show you an example of what this looks like

00:13:56 and how we set up a convolution. I'm back at our Jupyter notebook and at the function where we actually defined the network graph.

00:14:04 And this network accepts as input some feature tensor, which in our case are the images. Let me scroll down, we will come back to the initial part, it's not so important right now.

00:14:19 So this is our first convolution – this is how it looks when we have a convolution in TensorFlow. In the first convolution, we just input the features and we produce some output.

00:14:34 And notice that we have to select a number of output channels. In this case, I think it's eight – I left it to be parametric, for you to experiment with that.

00:14:47 You may also notice that this particular function call does not specify the kernel shape or the stride. Now, because I'm reusing the same arguments for each convolutional layer in the following,

00:15:02 I decided to actually specify default arguments, and that's what I'm doing here in this "with" construction.

00:15:10 I'm using TensorFlow's arg_scope functionality to define for each convolutional layer type the new default arguments.

00:15:20 And I'm setting default arguments, there I'm setting the kernel size – in this case, it's a 3x3 convolution. I'm also setting the non-linearity that should be called right after the convolution – in our case,

00:15:31 it's the rectified linear unit. And I'm also specifying how the weights should be initialized.

00:15:37 In this particular case, I'm using the variance scaling method. After the first convolution and the corresponding activation that is already implicitly being called,

00:15:52 we pass the output right into the next convolution with the same number of channels, and the next convolution, again, with the same number of channels,

00:16:02 and then we arrive at some operation which will be covered in the next unit. In the next unit, we will continue with our architecture discussion.

00:16:17 We will have CNN Architecture, Part 2, where we discuss the pooling operation, dense layers, dropout, and finally, softmax classification.

00:16:26 Thank you very much and see you in the next unit.

Week 4 Unit 3

00:00:07 Welcome to Week 4, Unit 3, "Convolutional Neural Network Architecture Part II". In the last unit, we covered "CNN Architecture Part I", where we had a closer look at convolutions,

00:00:19 at activation functions, the particular non-linearities, especially rectified linear units, and we looked at weight initialization.

00:00:28 In this unit, we will cover the remaining aspects of our network that we want to build. These are pooling, global pooling, dense layers.

00:00:37 We will discuss a technique called "dropout", and we will have a look at softmax classification. Now so far, our network looks exactly like this.

00:00:49 We have some input, we have convolutions, followed by some non-linearity,

00:00:53 another set of convolutions followed by some non-linearity, and we can continue the game in theory as much as we like.

00:01:00 However, it's computationally not as efficient. The main reason being we don't...

00:01:09 well, we just maintain the same resolution after each convolution. However, it may be desirable to actually decrease the resolution by having some downsampling operator.

00:01:21 The main reason being, first of all, of course, computational complexity, and second, it also increases spatial invariance.

00:01:30 Now after each pooling operator, a typical scheme is – for the next convolution that directly follows the pooling –

00:01:39 to double the number of output channels. So after each pooling we have reduced the resolution by perhaps a factor of 2 –

00:01:48 that's the most common factor – and then you double the number of output channels of each convolution.

00:01:55 And then you can repeat that game. You can have another pooling layer after a certain number of convolutions,

00:02:01 and reduce the resolution again by a factor of 2, and again increase the number of output channels for the subsequent convolutions.

00:02:09 And you can repeat that game multiple times, and it's not uncommon to reduce the resolution by a factor of 32 or even larger.

00:02:19 Now how does pooling work? On the left, you see the input image.

00:02:25 And I've created four buckets. They correspond exactly to what is known as 2x2 pooling with a stride of 2x2.

00:02:34 And the idea is to average in each bucket all the values. And this yields the average sum output map that is reduced in resolution by a factor of 2,

00:02:45 and just averaged. The more popular choice is what is known as max pooling.

00:02:52 Here the idea is to take the maximum value in each bucket. This again gives you a downsampled version.

00:02:59 And max pooling is actually better at capturing edges and preserving crisp and sharp features.

00:03:05 Average pooling corresponds more to a little bit of blurring. Now you can take this to the extreme and apply global average pooling.

00:03:16 And global average pooling just takes the entire input feature map. It averages across all the pixels to produce just one single number for a given channel.

00:03:25 Then of course you can apply this to all the input channels. So if you have six channels as output channels from a given convolution,

00:03:34 you get six numbers out of that. Let me switch to the Jupyter notebook and actually show you what this looks like in action.

00:03:48 In the last unit, we stopped just at the last convolutional layer here, so we have defined three convolutions.

00:03:54 And now the new operator that we've just discussed is the max pooling operator. And again, notice that it just takes the output of the last convolution as input.

00:04:06 I have not specified any parameters. That's again because I am making use of default arguments

00:04:13 in this arg_scope construction here, where I specify a stride of 2 and a kernel size of 2. And then I continue with convolutions.

00:04:24 I again have three convolutions. They are now operating on just half of the input resolution.

00:04:29 Notice, however, that now the number of output channels for each of those convolutions has doubled. Then again, I have some max pooling operator, which again downsamples by a factor of 2.

00:04:41 So now after that, we effectively have downsampled the input image by a factor of 4. And I continue with another convolutional block, again made up of three convolutions.

00:04:54 This time with four times the number of base channels. So I'm again doubling with respect to the previous convolutional block the number of output channels.

00:05:05 And then finally, I arrive at the global pooling. And I've implemented it as some kind of average pooling operator,

00:05:13 which just has a kernel size of the entire output feature map. Okay, let me switch back.

00:05:24 So... If you want to build a classifier, then most likely we also want to have some dense layer.

00:05:34 A dense layer following a convolution will connect each output pixel in each output channel with all the neurons in the dense layer.

00:05:46 And the weight matrix therefore becomes very large. Now it's much better to actually first apply global pooling.

00:05:56 You've already seen that in the Jupyter notebook. This means that rather than connecting all the neurons to each pixel of all the output channels of the previous convolution,

00:06:09 we only now connect to this one single value that has been produced as the average value across an entire feature map. And if in this particular example, we had six output channels,

00:06:22 and we applied global pooling, that means we would have just six numbers – one number per output channel.

00:06:28 And then we're just connecting to that, rather than to all the individual pixels. So now our weight matrix is way smaller.

00:06:37 So let's depict it here. We have the global pooling operator.

00:06:42 The number of channels equals the number of neurons. And then we can have the first dense layer.

00:06:49 And the dense layer is also followed directly by a rectified linear unit. Let me see what that looks like on a Jupyter notebook.

00:07:02 So we had here the average... the global pooling operation. Then I'm passing it through a flattening operation.

00:07:10 The flattening is necessary because, after the average pooling, our tensors are still in 4D shape.

00:07:18 The number of samples the mini batch, the number of channels, and then the resolution of that image –

00:07:25 in that case, it's just 1x1 because we're left with one single pixel. And the flattening operation will just turn it into a matrix.

00:07:33 And the number of samples in the mini batch and the number of output channels that we had. And then we can just directly connect this to the next dense layer, to the fully connected layer.

00:07:45 In this particular case, I'm not making use of default arguments because here in this Jupyter notebook it's just

00:07:53 the only fully-connected layer with these particular sets of parameters. I'm using 100 neurons.

00:08:01 Again, I'm using the rectified linear unit. And then we arrive at another layer that incorporates the technique known as dropout.

00:08:12 Let me explain that. So we have a layer called dropout.

00:08:18 And the purpose of dropout is to circumvent overfitting. So if you have a dense layer, you have a lot of parameters still.

00:08:29 The network is prone to overfitting. And dropout lets you basically circumvent this problem.

00:08:37 The idea of dropout is to take some input, some dense layer, for instance, and randomly during training set some neurons, like 50% of the neurons.

00:08:49 It's a free parameter, you can also specify 20%, for instance. But it's very common to set 50% of the neurons, let's say, to 0,

00:09:00 And at each training step, those are different neurons. And that forces the network to learn more entangled representation.

00:09:07 It forces other weights to make up for neurons that are currently being forced to 0 to still incorporate the same amount of information.

00:09:18 Now during inference, you do not drop out anything, you do not set neurons to 0.

00:09:24 But in order to make up for the new output distribution, you have to rescale all the neurons by a certain factor.

00:09:37 We can also apply dropout to convolutions, of course, not just dense layers. You simply drop out individual pixels across all the feature maps completely randomly.

00:09:48 Or you can apply something which is also known as "spatial dropout", where you, rather than dropping out individual pixels, drop out entire output channels.

00:10:01 Let me actually see what dropout looks like in code. Let's go back to the Jupyter notebook.

00:10:07 Right after our fully connected layer, we apply the dropout operation. We can specify a keep probability – the probability that a neuron keeps its value.

00:10:19 And also we have to specify whether we are currently in training mode, evaluation mode, or prediction mode. In training mode, as I mentioned earlier, dropout is already being applied,

00:10:32 but during prediction it's not – it's just a rescaling that is being applied. Let's switch back to the presentation.

00:10:40 Here in this particular case, I've just added another dense layer, another rectified linear unit, and another dropout layer just for the fun of it.

00:10:50 And then finally we arrive at our classification. And classification works such that for the number of classes that we have,

00:10:57 we have one associated neuron that is responsible for that class. And this is essentially also a dense layer followed by a "softmax activation function"

00:11:11 And the softmax activation function incorporates class probabilities. It's essentially normalizing the output across all the neurons,

00:11:24 such that they sum to one and can be interpreted as a probability. And that class, that neuron that has the highest probability, is the winning class.

00:11:33 So we just take the argmax across all the neurons in the final dense layer. And if we have 10 classes, let's say,

00:11:44 or in our traffic sign recognition benchmark example where we have 43 classes, we have to have 43 neurons.

00:11:53 Now during training, what we're doing is we're providing the training, the label probability distribution. And that's of course all the probability mass, all the probabilities just located at that one single neuron that is associated with that class.

00:12:09 So during training we try to drive the softmax output probability distribution towards the distribution as given by that class label. So let's check what that actually looks like in code.

00:12:28 Right after the dropout layer I have another fully connected layer which now is responsible for classification.

00:12:34 And the number of neurons is identical to the number of classes. I actually do not pass in the softmax activation function.

00:12:45 The sole reason being that for prediction, actually, I don't need a softmax activation function. All that I'm interested in is the neuron which has the maximum value.

00:12:56 I don't need to normalize it. But, of course, for defining the loss function I need to have the softmax.

00:13:06 The loss function involves a certain expression which can be numerically stabilized. And therefore the softmax has been incorporated already into that loss function.

00:13:17 Let me scroll to the loss function where I specify it. I'm actually scrolling now to our model function that the Estimator expects.

00:13:27 In this model function, the first thing we do is we actually build up the network graph. I just came from there.

00:13:33 The output of our network is the argmax on the last layer. And then if we're not in prediction mode, that means in the mode where we have labels available,

00:13:48 I'm calling loss function. And in this case, it's the sparse_softmax_cross_entropy loss.

00:13:54 And as the name suggests, the softmax is already incorporated in that. Now let me briefly continue to discuss what's going on.

00:14:02 Next here in this model function, we have specified the loss function. This is our function that we aim to minimize.

00:14:11 And in order to minimize that, we need to set up some kind of training optimizer, some training operator. And that, of course, only makes sense when we're in training mode.

00:14:21 So in the model function, if we are in training mode, we initialize some optimizer. In our case, it still has the gradient descent with momentum.

00:14:32 And then we say that we want to minimize our loss function. And that's our training operator.

00:14:45 We can also specify accuracy metrics, evaluation metrics – in this case, I'm specifying some kind of classification accuracy.

00:14:56 That's the measure of how many samples have been correctly classified. That's output during training and also during evaluation, over the validation data set.

00:15:09 And then, finally, the model function returns the training operator, the loss function, the particular mode we are in,

00:15:18 our evaluation metric operator, which incorporates this accuracy measure. And then we can actually launch training,

00:15:31 using the input function that loops over the training data set. And I'm not executing the training loop now because it takes a little bit of time,

00:15:42 but I can nevertheless show you the result of it. In this particular plot, I'm just loading the output of the training steps,

00:15:50 the loss function as a function of iterations – and that's what you see here. And hopefully, if the training went well, you should see that the loss function is decreasing

00:16:02 and plateauing at some value where training does not progress anymore. And that's the case here.

00:16:09 And then, also importantly, we would like to measure the accuracy on the validation data set, how many samples have been correctly classified – that's our accuracy.

00:16:21 And that's shown here. In this plot here, I'm just loading the output of the validation accuracy as a function of validation steps.

00:16:33 And you see that it's initially strongly increasing, it's plateauing around 99.5%,

00:16:41 and then I just stop training because it looked like it wouldn't increase anymore. All right, let me switch back to the presentation.

00:16:52 This was Unit 3. Coming up next, we have Unit 4, "Accelerating Deep Convolutional Neural Network Training".

00:17:00 Here we will look at computational considerations, we will discuss a technique called batch normalization.

00:17:06 And we will also have a look at transfer learning, and finally, briefly discuss residual networks.

00:17:11 Thank you very much and see you in the next unit.

Week 4 Unit 4

00:00:07 Welcome to Week 4, Unit 4, "Accelerating Deep Convolutional Neural Network Training". In the last unit, we had a closer look at the "Convolutional Neural Network Architecture Part

00:00:17 We covered, in particular, pooling operators. We had a look at dense layers, we covered dropout, and the softmax classification layer.

00:00:27 In this unit, we will cover all aspects of accelerating deep convolutional neural network training. We'll have a look at computational considerations.

00:00:34 We'll look at a technique that accelerates learning, called batch normalization. We will discuss transfer learning.

00:00:41 And finally, we will have a brief and quick look at residual networks. Let's begin with computational considerations.

00:00:51 Now suppose you want to have a fairly large field of view, you take a 7 x 7 convolution,

00:00:59 meaning that your input patch is covering 7 pixels in the x direction and 7 pixels in the y direction. In total, this means we have 49 parameters.

00:01:10 This makes up for, if you have 100 channels per layer, 49 x 100^2 parameters to optimize.

00:01:19 Now if you compare that to 3 x 3 convolutions, you have about 80% fewer parameters.

00:01:26 So you only have 9 weights that you need to have your weighted sum with in order to arrive at the output. However, the 3 x 3 convolution has a much smaller field of view.

00:01:41 However, if you replace the 7 x 7 convolution with two subsequent 3 x 3 convolutional layers, you still arrive at the same effective field of view.

00:01:54 You still have fewer parameters than the 7 x 7 convolution. Therefore, you have less overfitting.

00:02:00 And because you've actually now split this layer into two layers, you have more non-linearity in between

00:02:05 because – remember – after each convolution we have a non-linearity and a non-linear activation function. Now the next thing one can do is to actually reduce the number of fully connected layers that one has

00:02:21 right after some kind of convolutional network. On the left side, we have a network called the VGG network.

00:02:29 It consists of 19 convolutional layers. And then following that, we have a max pooling operation for further reducing the dimensionality of the input.

00:02:41 And then we have three fully connected layers. Now those three fully connected layers contain a lot of parameters – in total 23 million parameters.

00:02:49 So a valid question to ask is: "Can we perhaps reduce the number of parameters and still get comparable accuracy?" The answer is yes.

00:03:01 And there's some follow-up work, and a network called the Inception network. It essentially consists of 22 convolutional layers.

00:03:08 And then following that, we have a global pooling operator. We already discussed that in the last unit.

00:03:15 And then just one single fully connected layer. And in total, this network only has 1 million parameters.

00:03:22 So this is less prone to overfitting, it's also faster to optimize. Computational complexity is lower, and that's desirable.

00:03:31 Let's have a look at the next topic – batch normalization. Batch normalization is a technique for accelerating deep neural network training.

00:03:40 During training, suppose you have some kind of network consisting of multiple layers, what happens at some given training step is a convolutional layer produces an output distribution.

00:03:55 This output distribution is fed into the next layer, and that next layer again produces some other output distribution, and so on and so forth.

00:04:05 Now because we're during training, the weights in each layer get updated. And that effectively means that our output distribution changes now.

00:04:16 So in the first layer now, we updated the weights, and the effect of the convolutional layer, due to the weight updates, is now yielding a different output distribution.

00:04:28 Now this is problematic because the following layer has to adapt to the changes in the output distribution. And this is what is known as internal covariate shift, which makes the training rather hard.

00:04:43 So the idea of batch normalization is to actually normalize all the output distributions of each layer. So if each convolutional layer, or any layer, approximately outputs the distribution of zero mean and unit variance,

00:05:01 then we hope to get rid of this effect which hampers training. That's exactly what batch normalization is doing.

00:05:12 The idea is to operate on the current mini-batch, calculate the mean and the variance of that mini- batch,

00:05:19 and produce a normalized output. Now ideally, of course, we would need to operate on the entire training data set,

00:05:26 take the mean and the variance over the entire training data set, but this is infeasible during training.

00:05:31 And so we take an approximation and just approximate the mean and variance on the current mini-batch. How it works in more detail...

00:05:45 So first we have an input, xi, the samples in the mini-batch. We subtract the mean of the mini-batch and divide by the variance of that mini-batch.

00:05:55 And that produces the normalized samples in the mini-batch. So next we introduce two new parameters, gamma (γ) and beta (β).

00:06:06 And then we just multiply gamma by our normalized mini-batch and add an additional shift variable. Now you'll notice that actually gamma and beta are able to, in principle, undo the effect of the normalization.

00:06:22 However, we let the network learn those parameters, and it means that we let the network decide what the best way to do it is.

00:06:34 In practice, when you look at results and compare trainings with and without batch normalization, here on the right-hand side in the plot, we compare two of those trainings,

00:06:47 and we compare the classification accuracy of a network. The blue curve is without batch normalization.

00:06:55 As you can see, it flattens much earlier. Training does not progress as well in the later stages.

00:07:02 And the green curve continues to learn and is not as flat as the blue curve. As a nice side effect, we have a reduction in overfitting and we require fewer dropout layers as well.

00:07:19 Before continuing, let me just show you in the notebook how you can define batch normalization. Let me scroll to the function where we define the network graph.

00:07:37 So this is the place where we define the convolutions. This is the first convolution in the network where we just have the features as input.

00:07:46 Now we don't notice anything about batch normalization. But that's because if you look at the default arguments that we discussed earlier for the convolutional layer,

00:07:57 we are able to actually feed in the "normalizer" function. This normalizer function, in our case, will be batch normalization.

00:08:06 That's a separate layer in TensorFlow. And essentially the convolutional layer in TensorFlow wraps also batch normalization.

00:08:15 And if we provide that as a default argument, then there is no need to specify it. And batch normalization itself also requires some parameters.

00:08:25 And we have just specified this at the very beginning. We have to distinguish between training mode and evaluation mode.

00:08:35 In training mode, I just described that we are calculating the mean and the variance only on the current mini-batch. But in fact, at test time we don't want to do this dynamically for each individual mini-batch.

00:08:47 We want to have a fixed mean and a fixed variance. Now the fixed mean and the fixed variance are calculated on the fly as a moving average during training.

00:09:00 And those parameters... like the decay parameter is, for instance, the factor for the moving average. Okay, let's jump back to the presentation.

00:09:17 And let's move on to the next topic – transfer learning. Now it's typically the case that deep neural networks require large amounts of training data

00:09:27 and a lot of computational resources. So its computational complexity for building a network from scratch is rather high.

00:09:37 On the other hand, a lot of research groups have already produced plenty of nicely performing models, for instance the VGGNet, GoogleNet, and Microsoft ResNet networks.

00:09:48 Those networks have been trained on millions of images, mostly the ImageNet challenge. Multiple GPUs were used for training over the course of days or even weeks.

00:10:01 And... the idea of transfer learning is to actually take those models,

00:10:09 recycle it somehow, and adapt it to our current problem task. Why would you think that this would actually work?

00:10:21 Well, as we already discussed earlier, a neural network will learn a hierarchy of features.

00:10:29 And let's say you have a classifier, you have trained your network on a classifier that detects different types of animals.

00:10:40 And now you want to have a different classification task. You want to detect, let's say, different makes of cars.

00:10:49 You would expect that the first features in the lowest layer can be recycled because the lowest layers are responsible for responding to shapes, very basic shapes,

00:11:05 of particular orientation – circles, lines. And that, of course, can be recycled,

00:11:13 also for our classifier that detects or classifies cars. We've already seen that figure.

00:11:23 The first filters in the early layers respond to rather local features. The next mid-level layers react or respond to mid-level or somewhat more global features,

00:11:40 and the higher-level layers to more global features, Until the very end, we have layers that respond to rather fine-tuned or tailored responses

00:11:57 for the task at which the network has been trained. For instance, in this particular case, we see that this network in the last layer responds to what looks like a dog –

00:12:07 an abstract representation of a dog, a cat, and an elephant in this case. And this, of course, cannot be reused when we then try to classify, for instance, different cars.

00:12:20 We can have two different scenarios. In the first scenario, we can assume that most of the features in the all the earlier layers

00:12:29 can be recycled and they can just remain fixed. And all we have to do is to replace the last layer, and then retrain the last layer.

00:12:39 The nice thing about this is that we only need fewer training samples. Our data set now doesn't need to be very large.

00:12:47 We can just do a little bit of fine tuning on a small data set. If our task is deviating more from the original task on which the network has been pretrained,

00:12:57 then we may be interested in replacing more layers and retraining more layers. The drawback, of course, is that you then again start to require a larger data set again.

00:13:09 But still, this may be more feasible than redoing everything from scratch. Let's finally have a look at residual networks.

00:13:20 Our intuition is that deeper networks are more expressive. However, they are also harder to train due to the vanishing gradient problem.

00:13:28 The idea of ResNets is to actually assume that the modeled function of each building block has a higher a resemblance to the identity function than to the zero function.

00:13:41 So what does this mean in more detail? A layer that is receiving some input typically has to rebuild the entire signal from scratch.

00:13:51 But if we have some kind of representation, or some kind of way of letting the network to model just the deviation away from the signal,

00:14:00 then this would be more optimal. And this is actually what the residual network is doing.

00:14:04 So see a basic module here on the right. The module receives x as an input,

00:14:12 and just bypasses it directly to the end, and sums it on the outcome of two, let's say, convolutional neural network blocks.

00:14:22 Those don't need to be convolutions, they can also be fully connected layers. The effect of this shortcut path is exactly what I just mentioned.

00:14:35 This forces the weights to model deviations away from the input signal. And now you can repeat that pattern.

00:14:45 You can have that building block repeated multiple times to arrive at a very deep network. And since we have multiple skip connections,

00:14:55 the signal can actually directly skip multiple layers right till the very end, right to the bottom layers. And vice versa, also the gradients can flow much easier back to the beginning,

00:15:07 making training much faster. Now this way, we can have very deep network architectures with typically more than 100 layers.

00:15:17 That concludes the unit. In the final unit of this week, we will have a look at applications of convolutional neural networks.

00:15:25 In particular, we will look at object detection and semantic image segmentation. Thank you very much.

Week 4 Unit 5

00:00:07 Welcome to the final unit of Week 4, "Applications of Convolutional Neural Networks". In the last unit, we covered aspects of how to accelerate deep convolutional neural network training.

00:00:17 We had a look at computational considerations. We looked at a technique called batch normalization for speeding up training.

00:00:23 We discussed transfer learning, and also we briefly discussed residual networks.

00:00:30 In this unit, we'll have a look at other applications of convolutional networks other than the classifier that we built in the notebook.

00:00:38 In particular, we will have a look at object detection – "two-stage detectors" and "one-stage detectors" – and we'll also look at the task of image segmentation.

00:00:49 Now let's first discuss object detection and how to build a two-stage detector. Now this is a particular example of what object detection is actually like.

00:01:00 So we have some kind of input image. And then we would like to detect certain objects in our image.

00:01:06 For instance, we may want to detect buildings, we may want to detect persons,

00:01:12 and we may want to detect traffic signs. Detection means we actually draw a bounding box around that object.

00:01:22 That's the location. And we can have multiple different classes,

00:01:26 so we don't... well, we could have a detector that only focuses on detecting buildings, but nothing is stopping us from also detecting all kinds of different classes of objects in one go.

00:01:43 Now what does a naïve approach to object detection look like? Typically, we would specify some kind of sliding window that we slide over the input.

00:01:54 And then for each sliding window, we run a pretrained classifier, which just classifies whether the object is in that sliding window or not.

00:02:07 So we have to run a lot of sliding windows. We have to run the classifier on a lot of sliding windows,

00:02:12 all the way until we finally arrive at a sliding window where perhaps the object is located. And then that shape of the sliding window defines the location of that object.

00:02:25 A problem, of course, is that this is computationally prohibitive. It's intractable even for small images.

00:02:31 You have to have a sliding window – one for each pixel that you shift over the input image in the most extreme case.

00:02:39 This is, of course, not desirable. A much better way is to actually have some kind of proposal algorithm, a kind of selective search,

00:02:48 that, based on texture, color, or intensity information, actually proposes locations in the image.

00:02:58 It's depicted here. We have a proposal algorithm that spits out a bunch of proposals.

00:03:04 And then we evaluate those proposals with our classifier. And there are way fewer than if we just had the naïve sliding window approach.

00:03:17 And that's what some of the first works have actually done using deep neural networks.

00:03:24 And there's a particular two-stage detector called R-CNN. And this is exactly how it's pursuing the idea.

00:03:33 It's taking the input image. There's some kind of external proposal algorithm which is cropping proposals from the original image.

00:03:42 Then those crops are warped to the same target resolution. That's because the convolutional neural network requires a fixed input resolution.

00:03:53 And then this convolutional neural network is producing features. And those features at the very end are fed into some support vector machine

00:04:01 to classify whether there is an object or whether it's background, and what kind of object this is.

00:04:06 And then also we may have an additional linear regressor, which can improve our bounding boxes.

00:04:16 Advantages are that this is several orders of magnitude faster compared to the naïve sliding window approach. It also allows for improved bounding boxes.

00:04:24 We don't just have a bounding box for that fixed sliding window, but we can adjust the bounding box a little bit,

00:04:30 arriving at this more detailed region where the object is located. The disadvantages for this particular model are that the convolutional neural network

00:04:41 basically does not feel the influence of the support vector machine – it's separate training. And also the convolutional neural network evaluates bounding boxes which are overlapping.

00:04:51 And that means a lot of computation is actually wasted because you spend computational resources on a lot of overlapping regions.

00:04:59 And that's why, in follow-up work, the same sorts of offers had a similar network, which they call Fast R-CNN,

00:05:07 and that tried to improve the situation. Now the first thing they did is they replaced the support vector machine at the linear regressor

00:05:14 with a fully-connected neural network with a softmax output and a linear regression output. So they combined that.

00:05:24 And then they attached a convolutional neural network to it. And now, rather than operating on individual proposals as input,

00:05:34 this convolutional network takes as the input the entire input image. And only later, only in the last feature layer,

00:05:41 in the output layer before the features are fed into the fully-connected neural network, the classifier and the regressor,

00:05:51 only then is the feature map cropped. Only then do we have some kind of region proposal generator

00:05:57 which directly crops in the last feature map. And this, of course, speeds things up

00:06:06 because now we don't have overlapping computation for the CNN – we just have the entire input image once.

00:06:12 We only crop in the last feature map. Once this problem is removed,

00:06:20 the proposal generator actually becomes the next bottleneck. So the solution to that is to actually also have a Region Proposal Network directly inside the neural network,

00:06:31 rather than using some kind of external proposal algorithm. And the idea of that is as follows:

00:06:37 Again, you get an entire input image that you feed into a convolutional neural network. You produce at the output of the convolutional neural network some kind of feature map.

00:06:48 Then you have a Region Proposal Network, a sub-network, which just proposes bounding boxes or regions.

00:06:59 And those region proposals, or bounding boxes, are taken by the network, and you, directly on the fly, crop those region proposals from the last feature map.

00:07:11 You again apply region-of-interest pooling, or warping, to have a fixed size. And then you just attach the classifier and the regressor to it.

00:07:25 If you look at the performance numbers, the first thing I would like to draw your attention to is the mean average precision that is reported here at the bottom.

00:07:34 You notice that across all models – the R-CNN model, the Fast R-CNN model, and the Faster R-CNN model – it's about the same.

00:07:44 But the test time per image is drastically reduced. While R-CNN requires 50 seconds per image,

00:07:54 Faster R-CNN operates on a fraction of a second. Now let's move on to "one-stage detectors".

00:08:07 Even though Faster R-CNN already seems to be very fast compared to the other two-stage detectors of the very simple models, two-stage detectors are still fairly slow,

00:08:20 simply because they operate in two stages. A valid question to ask is: "Can we make this faster?"

00:08:27 And yes, we can. And we can also make it simpler.

00:08:31 We can come up with one-stage connectors which just propose bounding boxes and classify objects in one go. And we don't have an intermediate region proposal stage.

00:08:46 How does this work in more detail? For instance, a particular one-stage detector called YOLO(9000),

00:08:53 there's a variety of different one-stage detectors. The basic idea is that you take the input image,

00:08:59 you feed it into a convolutional neural network, which we also call an encoder.

00:09:06 We call it an encoder because it's encoding an abstract representation of features. It's this concept of convolutional layers, then pooling.

00:09:18 Convolutional layers with reduced resolution, again pooling. So the encoder at the very end arrives at some kind of feature map,

00:09:26 which has a lot of channels but only very low resolution. Now the next thing we do is we upsample again,

00:09:37 and we call it the decoder. And the reason we like to upsample is because we want to output a mask.

00:09:45 In the extreme case, for each pixel we are outputting a mask whether the object is located in that pixel or not, whether the current pixel belongs to a certain object or not –

00:09:56 what kind of class of object that is. Of course, doing it for all the pixels on the original input resolution is overkill.

00:10:06 Typically, one downsamples by a factor of 4, or something like that. So, in addition to the mask, we also require a bounding box regression algorithm.

00:10:16 And that's why we have a separate output branch, which is actually for each pixel in the mask predicting 4 coordinates,

00:10:23 the 4 coordinates of the bounding box. And then you can merge the two pieces of information,

00:10:27 the mask information together with all the predicted coordinates at each pixel, and we can arrive at one single bounding box.

00:10:39 Actually, usually because we have multiple pixels belonging to the same object and we have a larger set of bounding boxes,

00:10:46 you also would need to run an algorithm, like non-maximum suppression. which merges adjacent bounding boxes,

00:10:57 or bounding boxes that largely overlap. Now there are also more modern detectors,

00:11:06 the single shot multibox detector (SSD), and a variety of different models as well.

00:11:13 Now one of the problems with one-stage detectors is that even though they are faster and simpler to design, they are less accurate,

00:11:22 simply due to the fact that two-stage detectors have this proposal algorithm, and they spit proposals on the order of maybe 2,000 proposals.

00:11:34 So they operate on 2,000... they need to regress 2,000 bounding boxes and classify them,

00:11:40 whereas a one-stage detector has to do it for each pixel, or for all the pixels in the input image, in the most extreme case.

00:11:52 And even when the resolution is reduced by a factor of 4, it's still on the order of 100,000 proposals that the one-stage detector has to work on.

00:12:06 And since most of the pixels belong to the background, that means we have a drastic class imbalance between foreground and background objects.

00:12:15 And in the two-stage detectors, we have the luxury of actually sampling and balancing foreground vs. background objects. Therefore, one-stage detectors typically are less accurate.

00:12:30 But then, fairly recently, there was a paper that came up with a new loss function – the "Focal Loss" function. And here the idea is to downweigh contributions in the loss for examples that are very easy.

00:12:47 And typically, easy examples are the background samples. We can just take our cross entropy loss for classification

00:12:56 and dynamically downweigh the loss based on the confidence for that class. I strongly invite you to check out this paper for details.

00:13:09 I won't discuss it in more detail here. Now let's move on to a separate task in computer vision – that's image segmentation.

00:13:23 Here we have an example – we have an input image and the task of image segmentation means that we want to classify each pixel in that image.

00:13:35 We want to create some kind of mask. In this particular image, I'm classifying background vs. person.

00:13:43 And this is what the result ideally should be. Now, this is very simple –

00:13:50 it's just object vs. no object. In a real-world scenario, you can have many classes, and you want to classify each pixel.

00:14:02 For instance, for autonomous driving or so, you want to classify pixels that belong to a traffic sign, you want to have pixels that belong to the street,

00:14:09 pixels that belong to buildings, to other cars, and persons. Now the very generic architecture of such a segmentation network is fairly similar to a one-stage detector

00:14:24 for object detection. We get this input image,

00:14:28 we feed it into a convolutional neural network, which we call the encoder, which is building up an abstract representation of the objects.

00:14:37 But since the final feature map again has a very coarse resolution, we need to upsample somehow. And that's done by the decoder.

00:14:47 And to make things simpler for the network to learn, we actually recycle the low-level but high-resolution feature from the earlier layers,

00:14:59 and merge them back into our later layers that are supposed to upsample our image. This is a very rough idea of perhaps a generic architecture of a segmenter.

00:15:12 So let's have a closer look at a particular architecture, the "U-Net" architecture in this example.

00:15:20 The encoder basically takes some kind of image as an input – a 32 x 32 image, fairly small. But it could also be larger – 512 x 512 pixels, for instance, or even larger.

00:15:30 So this RGB image is fed into the first convolutional layer, which has 64 output channels.

00:15:36 Then we have a second convolution – again with 64 output channels. And then we encounter the first max pooling operator,

00:15:44 which downsamples by a factor of 2. Now we have a feature map that has a resolution of 16 x 16.

00:15:52 And that's again fed into the next convolutional layer. Here we double the number of channels.

00:15:58 It's the same scheme that we already saw for our classification example in earlier units. And we feed it again through a second convolution.

00:16:12 And finally again, a max pooling layer. And now we arrive at a resolution that is fairly coarse – 8 x 8.

00:16:18 Where we have downsampled by a factor of 4 in total, that's actually not that much. There are also encoders which downsample much more aggressively.

00:16:28 However, for this simple architecture we just downsample by a factor of 4. And we have a final convolutional layer in the encoder, consisting of 256 output channels.

00:16:40 Now we try to upsample again with a decoder. And the decoder essentially consists of convolutions, just like the encoder.

00:16:50 And it's exactly a mirrored structure, actually. The only difference is that the downsampling – which is done by the max pooling operators in the encoder –

00:17:01 is replaced by either replaced with either transposed convolution, which we briefly introduced in Unit 3, or with a static bilinear upsampling operator.

00:17:13 Now finally, the output we are interested in is pixelwise classification. That could be some kind of softmax classification layer

00:17:23 which now acts at each individual pixel. Now we also have already discussed that...

00:17:34 I mentioned earlier that we want to recycle earlier features which have a higher resolution to help us in the upsampling.

00:17:43 And that's what's being done here. So we take from the end of the first convolutional block,

00:17:49 which is operating at the same resolution as the input image. We just have a shortcut connection,

00:17:54 which directly shortcuts to the corresponding convolution at the same resolution in the last decoder convolutional block.

00:18:04 And the same with the convolution which is downsampled by a factor of 2. So this architecture can actually be more complex.

00:18:17 The downsampling could be more aggressive, you could have way more layers. But this architecture nicely summarizes the key concepts.

00:18:29 Of course there's another fairly simple network – it's SegNet. But I like it because it's a nice and easy baseline.

00:18:36 There are also more complicated networks – for instance, the "pyramid scene parsing" network,

00:18:41 which has additional modules which we can't cover in this unit here. Now the challenges for image segmentation –

00:18:52 it's always an efficiency and accuracy trade-off. You have some kind of encoder downsampling.

00:18:59 You lower the resolution to build some kind of abstract representation and to be more spatially invariant. On the other hand, you want to have a dense prediction –

00:19:07 you want to have a prediction for each pixel. So rather than having some aggressive downsampling to reduce the computational complexity,

00:19:17 we can instead introduce dilated convolutions, which we also briefly mentioned in Unit 2.

00:19:27 And we can make use of feature fusion – that's the concept of shortcutting low-level features from the high-resolution layers.

00:19:36 From the early high-resolution layers into the late layers that are upsampled to the same resolution. And then another aspect is data availability.

00:19:47 Ground truth is very costly to acquire because the images have to be hand-segmented up front. And that's obviously a laborious task.

00:20:00 So data augmentation becomes essential. We can have a number of random, on-the-fly modifications of our samples.

00:20:09 We can have color augmentations. We can have all kinds of distortions, cropping...

00:20:15 and this will help improve our network training. Thanks a lot. This concludes the last unit of this week.

00:20:24 I hope you enjoyed this week, and I wish you a lot of luck with your assignments.

00:20:31 In the next unit, or in the next week, we will have a closer look at industry applications of deep learning. We will discuss machine learning in customer service, machine learning in banking,

00:20:42 and end with medical image segmentation. Thanks a lot.

www.sap.com