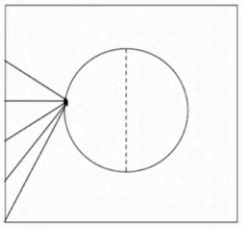
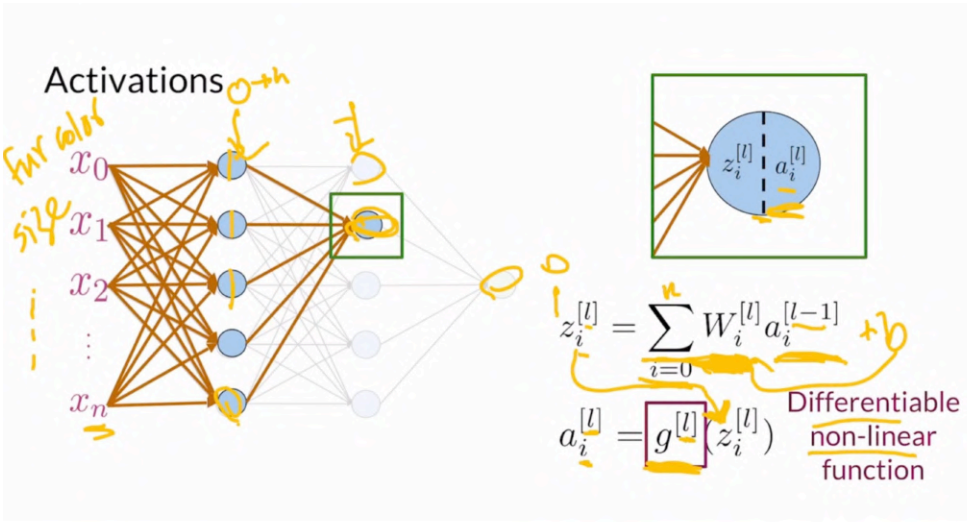
Outline

- What are activations
- Reasoning behind non-linear differential activations

Activations are functions that take any real number as input, also known as its domain, and outputs a number in a certain range using a non-linear differentiable function. We typically use them for classification between certain layers in deep neural networks and more specifically in games.

This section will go over what activations are, and the reason why they are both non-linear and need to be differentiable, which just means you can take the derivative of them, meaning they have a gradient.

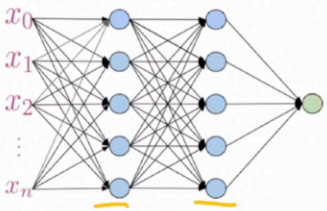




Activations

$a_i^{[l]} = g^{[l]}(z_i^{[l]})$ Differentiable non-linear function

1. Differentiable for backpropagation
2. Non-linear to compute complex features, **if not**:



$\equiv$ $WX + b$

Linear regression

Summary

- Activation functions are non-linear and differentiable
- Differentiable for backpropagation
- Non-linear to approximate complex functions

deeplearning.ai

Common Activation Functions

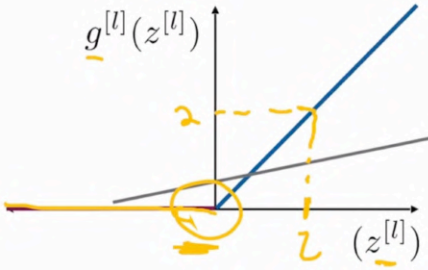
Outline

- Common activations and their structure
  - ReLU
  - Leaky ReLU
  - Sigmoid
  - Tanh

Activations: ReLU

ReLU = Rectified Linear Unit

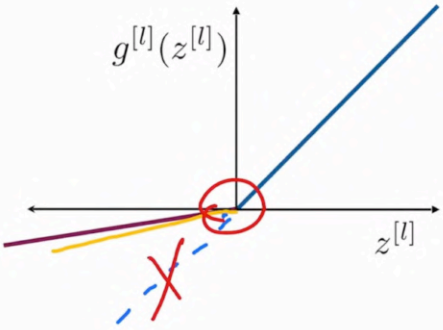
$$g^{[l]}(z^{[l]}) = \max(0, z^{[l]})$$



Dying ReLU problem

Activations: Leaky ReLU

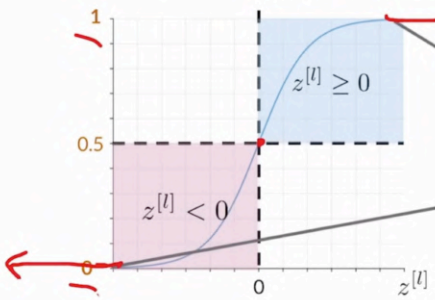
$$g^{[l]}(z^{[l]}) = max(az^{[l]}, z^{[l]})$$



Solves the dying ReLU problem

Activations: Sigmoid

$$g^{[l]}(z^{[l]}) = \frac{1}{1 + e^{-z^{[l]}}}$$

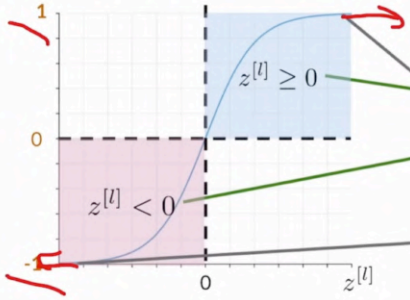


Values between 0 and 1

Vanishing gradient and saturation problems

Activations: Tanh

$$g^{[l]}(z^{[l]}) = tanh(z^{[l]})$$



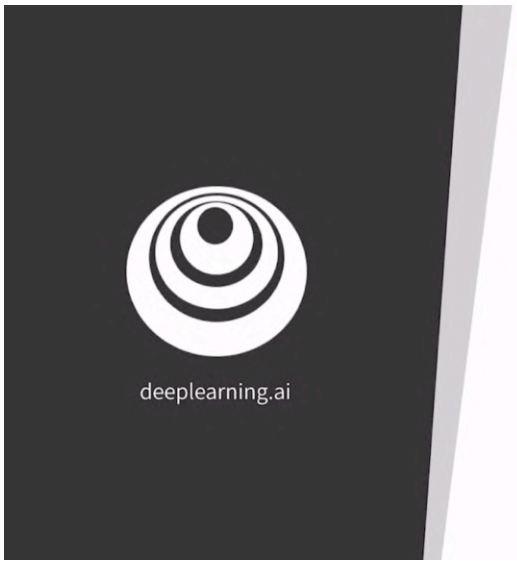
Values between -1 and 1

Keeps the sign of the input

Same issues as Sigmoid

Summary

- ReLU activations suffer from dying ReLU
- Leaky ReLU solve the dying ReLu problem
- Sigmoid and Tanh have vanishing gradient and saturation problems



Batch Normalization (Explained)

GANs take a lot of time to train, especially if you want to build them for really cool applications like the one discussing this course. GANs are often quite fragile when they learn to because they aren't as straightforward as a classifier. And sometimes the skills of the generator and discriminator aren't as aligned as they could be.

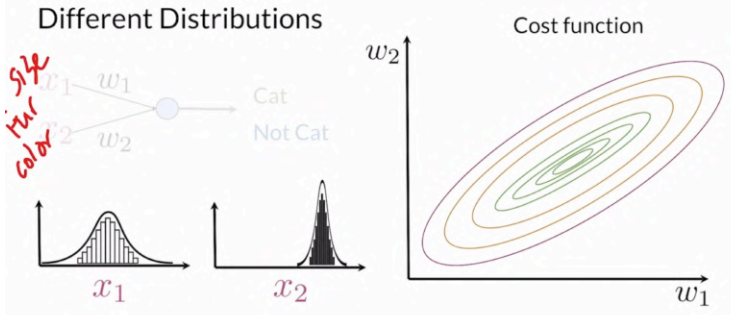
For these reasons, every trick that speeds up in stabilizes training is crucial for these models, and batch normalization has proven very effective to that end.

So to get started, review of the effects that normalization has an training. Review what internal covariate shift means and why it happens. And at the end, you'll get to connect it all the batch normalization and develop some intuition on why it speeds up and stabilizes training in neural networks.

Outline

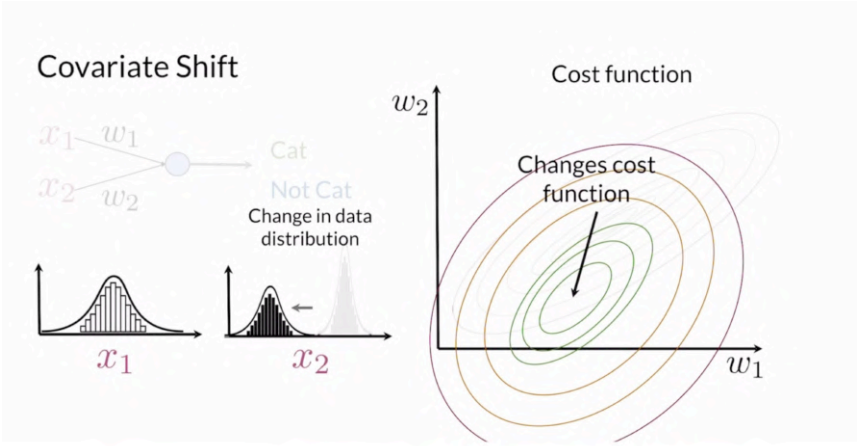
- How normalization helps models
- Internal covariate shift
- Batch normalization

So suppose you had a super simple neural network with two input variables x1, for size and x2, say fur color. In a single activation that outputs whether this example based on these features is a cat or not a cat. And let's say the distribution of x1, here the size, over your data set is normally distributed like this around a midsize example.



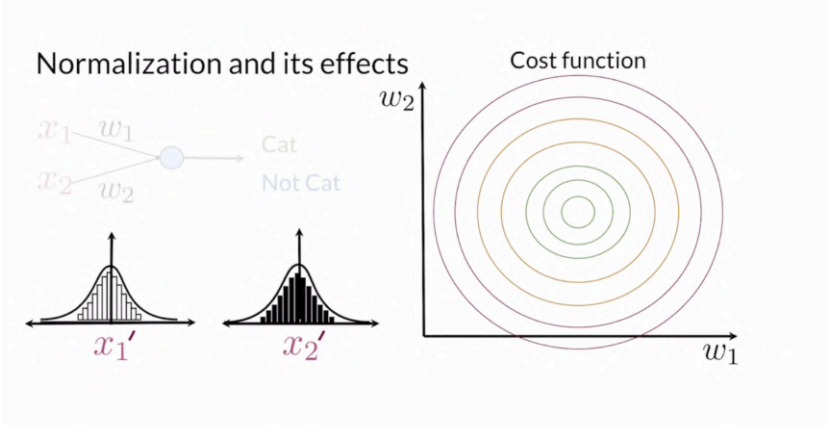
For example, with very few extremely small or extremely large examples. While the distribution of fur color x2 skews a little bit towards higher values over here with a much higher mean and lower standard deviation. And so let's say, this higher value here represents darker fur colors. And of course, notice that comparing the same value on fur color and size is a little bit like comparing apples to oranges, because dark fur color and large size don't really have a meaningful correlation.

However, the result of having these different distributions like this impacts away your neural network learns. For example, if it's trying to get to this local minimum here and it has these very different distributions across inputs, this cost function will be elongated. So that changes to the weights relating to each of the inputs will have kind of a different effect of varying impact on this cost function. And this makes training fairly difficult, makes it slower and highly dependent on how your weights are initialized.



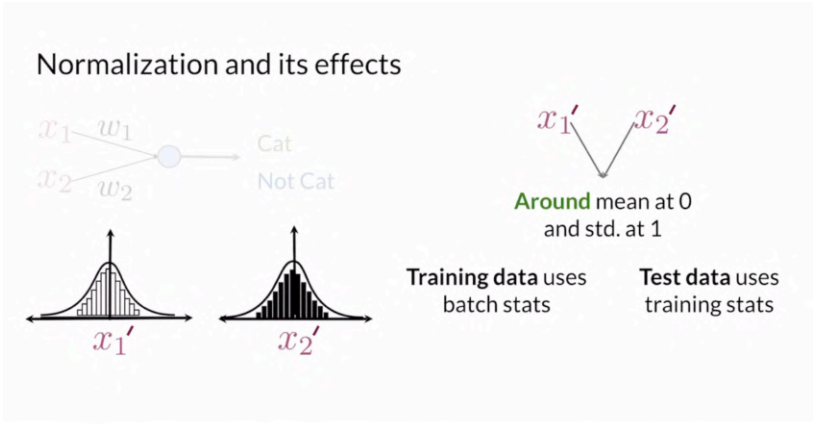
Additionally, if new training or test data has, let's say, really light for a color, so the state is distribution kind of shifts or changes in some way, then the form of the cost function could change too. So it's showing that it's a little bit more round here now and the location of the minimum could also move. Even if the ground truth of what determines whether something's a cat or not stays exactly the same. That is the labels on your images of whether something is a cat or not has not changed.

This is known as covariate shift. And this happens pretty often between training and test sets where precautions haven't been taken on how the data distribution is shifted.



But what if the input variables x1 and x2 are normalized? Normalized, meaning, the distribution of the new input variables x1 prime and x2 prime will be much more similar with say means equal to 0 and a standard deviation equal to 1.

Then the cost function will also look smoother and more balanced across these two dimensions. And as a result training would actually be much easier and potentially much faster.



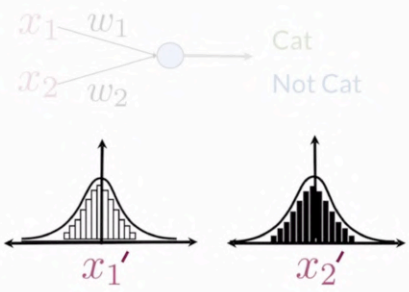
Additionally, no matter how much the distribution of the raw input variables change, for example from training to test, the mean and standard deviation of the normalized variables will be normalized to the same place. So around a mean of 0 and a standard deviation of 1.

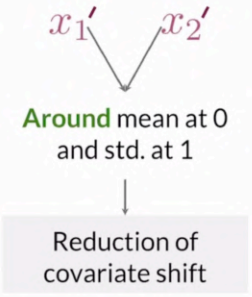
For the training data, this is done using the batch statistics. So as you train each batch you take the mean and standard deviation and you shift it to be around 0, and standard deviation of 1.

For the test data what you do is you can actually look at the statistics that were gathered overtime as you went through the training set and use those to center the test data to be closer to the training data. And using normalization, the effect of this covariate shift will be reduced significantly.

So those are the principle effects of normalization of input variables, smoothing that cost function out in reducing the covariate shift.

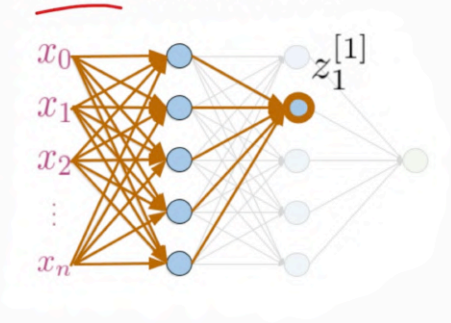
Normalization and its effects





However, covariate shift shouldn't be a problem if you just make sure that the distribution of your data set is similar to the task your modeling. So, the test set is similar to your training site in terms of how it's distributed

Internal Covariate Shift



Changes in weights

↓

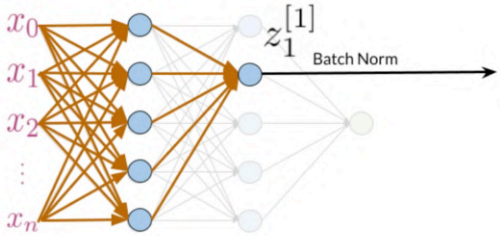
Changes in activation distribution

That being said, neural networks, like the one I am showing you here, are actually susceptible to something called internal covariate shift, which just means it's covariate shift in the internal hidden layers here.

So take the activation output of this first hidden layer of the neural network and look at this node right here. When training the model, all the weights that affect the activation value are updated. So all of these weights are updated. And consequently, the distribution of values contained in that activation changes in our influence over this course of training.

This makes the training process difficult due to the shifts similar to the changes you saw in the input variable distribution shifts like fur color. Only on the internal nodes that have a little bit more of an abstract meaning than fur color and size.

Batch Normalization



Normalizes the input for each neuron

Now batch normalization seeks to remedy the situation. And normalizes all these internal nodes based on statistics calculated for each input batch. And this is in order to reduce the internal covariate shift. And this has the added benefit of smoothing that cost function out and making the neural network easier to train and speeding up that whole training process.

So the word batch in batch normalization indicates the use of batch statistics

Summary

- Batch normalization smooths the cost function
- Batch normalization reduces the internal covariance shift
- Batch normalization speeds up learning!

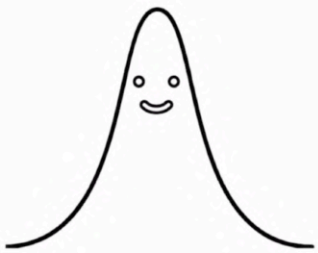
So in summary, batch normalization smooth the cost function and reduces the effect of internal covariate shift. It's used to speed up and stabilize training and comes in handy when building great GANs.

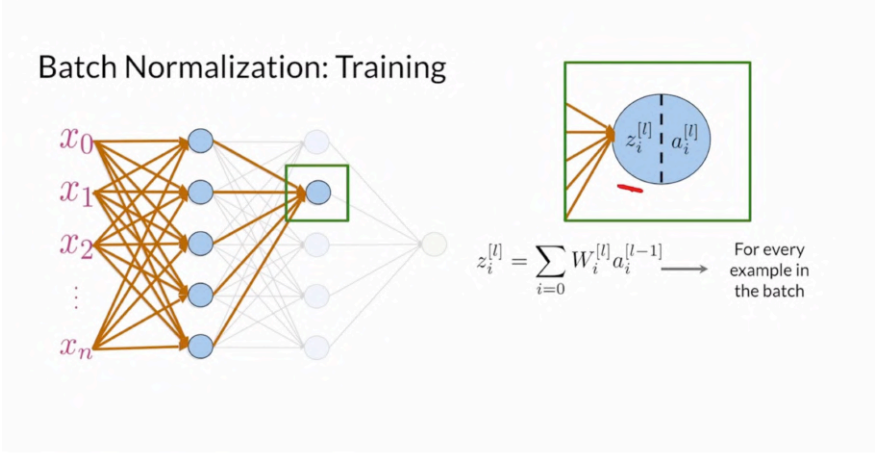
deeplearning.ai

Batch Norm. (Procedure)

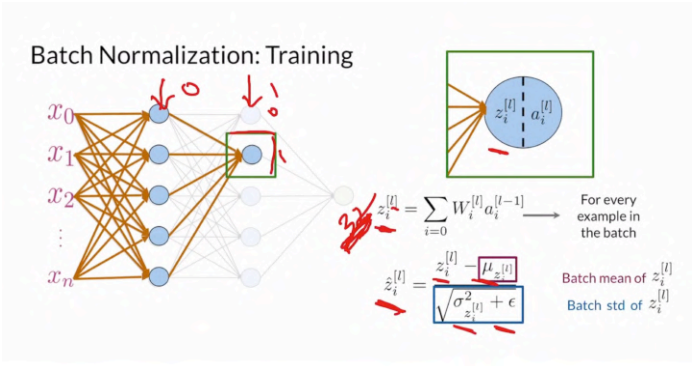
Outline

- Batch Norm for Training
- Batch Norm for Testing





Take this neural network with two hidden layers, multiple inputs and a single output. I'll focus on this internal hidden layer to explain how batch normalization works. From the inputs, z here is coming from all the previous nodes. Batch normalization considers every example z_i in the batch.

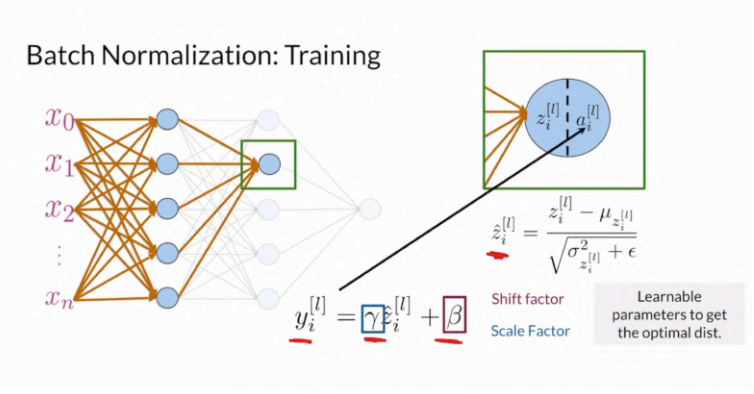


For example, you could have a batch size of 32, so you would have 32 zs here. Again, as a reminder, i here represents this is the ith node in this layer, and l represents this is the lth layer. For example, here is layer zero, layer one, or here is node zero, node i1.

Batch normalization takes the size of the batch, for example, 32 and it has 32 zs here. From those 32 zs, it wants to normalize it so that it has a mean of zero and a standard deviation of one.

What you do is you get the mean of the batch here Mu, and that's just the mean across all these 32 values. Then you also get the variance of the batch sigma squared, which is again the value from these 32 values.

To normalize these z values to a mean of zero and a standard deviation of one, you subtract the mean and you divide by the standard deviation, on here it's the square root of the variance, and you add an Epsilon here just to make sure that the denominator isn't zero. At the end, you obtain these normalized z values which I'll call z-hat.



After you get the normalized value z-hat, you have learned perimeters in the batch normalization layer. What that means is that you will have a value called Beta, which will be the shift factor and Gamma, which will be the scale factor. These parameters are learned during training to ensure that the distribution to which you're transforming z is the optimal one for your task. After you completely normalize things to z-hat, you then rescale them based on these learned values, Gamma and Beta.

This is the primary difference between normalization of inputs and batch normalization. Because here you are not forcing your distribution to have zero mean and standard deviation of one every single time, it's after normalizing things, then you can go on and rescale things to an unnecessary task.

What's key here is that batch normalization gives you control over what that distribution will look like moving forward in the neural network, and this final value after the shifting and scaling will be called y. This y is what then goes into this activation function.

Batch Normalization: Test

$$\hat{z}_i^{[l]} = \frac{z_i^{[l]} - \mathbf{E}(z_i^{[l]})}{\sqrt{\mathrm{Var}(z_i^{[l]}) + \epsilon}}$$

Running mean and running std from training

Frameworks like Tensorflow and Pytorch keep track of them.

Now, that was during training, and during testing, you want to prevent different batches from getting different means and standard deviations because that can mean the same example, but in a different batch would yield different results because it was normalized differently due to the specific batch mean or specific batch standard deviation. Instead, you want to have stable predictions during test time.

During testing, what you use is the running mean and standard deviation that was computed over the entire training set, and these values are now fixed after training, they don't move. But after that, you just follow a very similar process. Here is the expected value of those z values, so that's a running mean, and here's the variance of those z values, where when you take the square root here, will get you the standard deviation. You still have that Epsilon here to prevent that denominator from going to zero.

After that, you just follow the same process as you did in training and you feed these normalized values into the learn parameters and then the activation function. But don't worry too much about the details of this process. Frameworks like TensorFlow and PyTorch keep track of these statistics for you. All you have to do is create a layer called batch norm, and then when your model is put into the test mode, the running statistics will be computed over the whole data set for you or saved for you, so that's really, really nice.

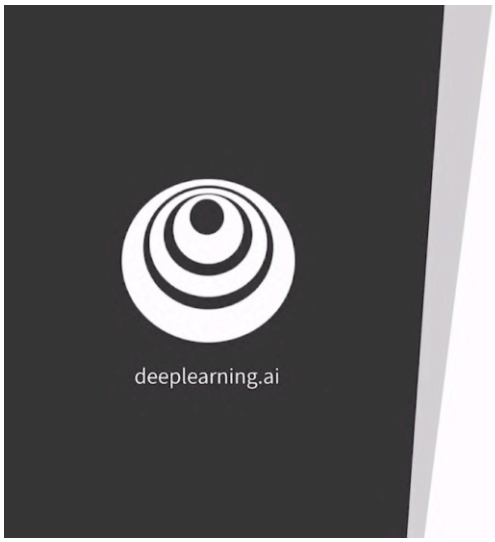
You might have also heard of test mode being called test-time inference time, evaluation or eval mode. These are roughly equivalent and you can think of them as not training time.

Summary

- Batch norm introduces learnable shift and scale factors
- During test, the running statistics from training are used
- Frameworks take care of the whole process

In summary, batch normalization differs from standard normalization because during training, you use this statistics from each batch, not the whole data set, and this reduces computation time and makes training faster with our waiting for the whole data set to be gone through before you can use batch normalization. Secondly, by introducing learnable shift and scale parameters, you don't force the target distribution to have a zero mean and a standard deviation of one. Finally, at test time, the running statistics from training are applied to the entire data set, which keeps predictions stable because the training values are independent and fixed.

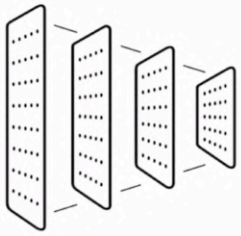
One important note is [inaudible] frameworks have this entire process implemented for you, for training and testing. All you need to do is know when to use it in your [inaudible] or some other spectacular model you want to train.



Review of Convolutions

Outline

- What convolutions are
- How they work



What is a convolution?

Eye Filter

5	3	1
10	23	1
1	42	4

Nose filter

2	21	5
23	45	12
3	32	4

Ear filter

32	2	24
25	12	66
3	45	2

What is a convolution?

SUM

50*1	50*0	0*-1	0	0
50*1	50*0	0*-1	0	0
50*1	50*0	0*-1	0	0
50	50	0	0	0
50	50	0	0	0

Grayscale image

*

Filter

1	0	-1
1	0	-1
1	0	-1

Convolution

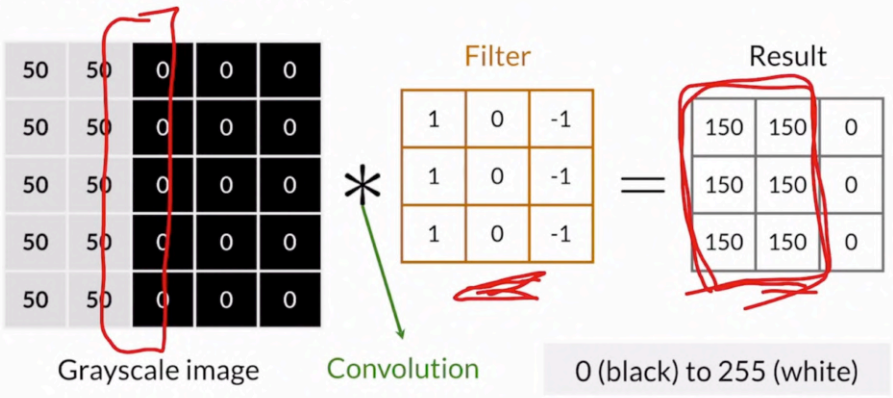
=

Result

150		

0 (black) to 255 (white)

What is a convolution?



Summary

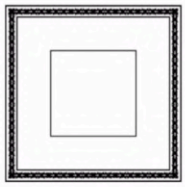
- Convolutions are useful layers for processing images
- They scan the image to detect useful features
- Just element-wise products and sums!

deeplearning.ai

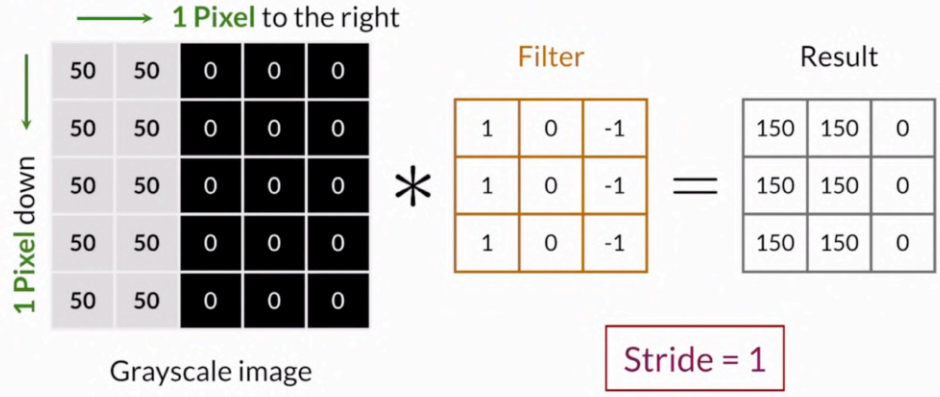
Padding and Stride

Outline

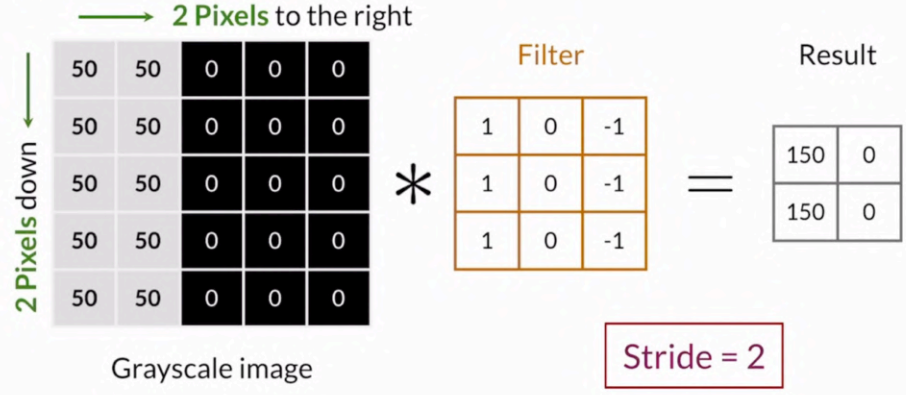
- Padding and stride
- The intuition behind padding



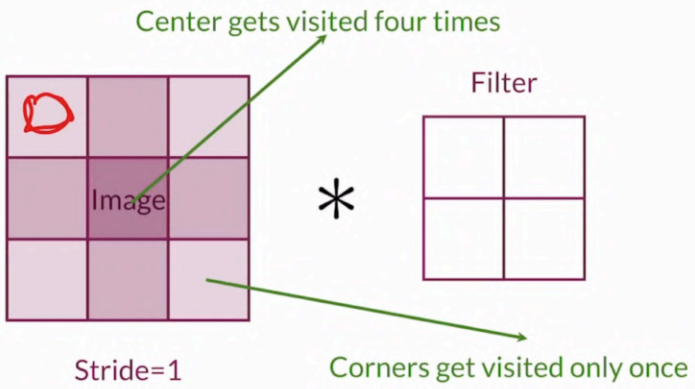
Stride



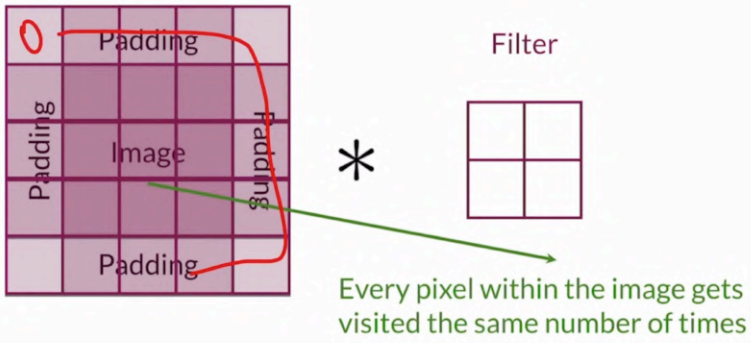
Stride



Padding



Padding



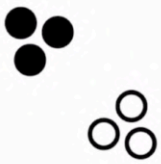
Summary

- Stride determines how the filter scans the image
- Padding is like a frame on the image
- Padding gives similar importance to the edges and the center

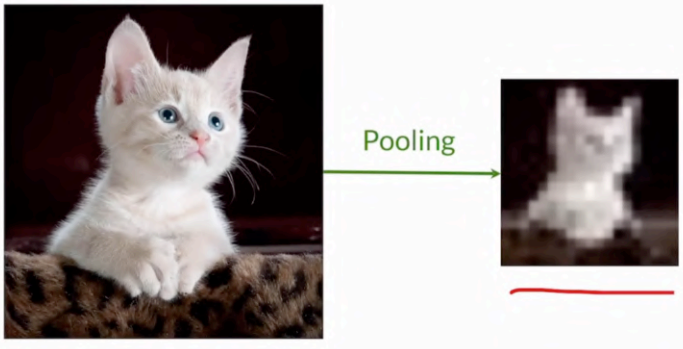
Pooling and Upsampling

Outline

- Pooling
- Upsampling and its relation to pooling

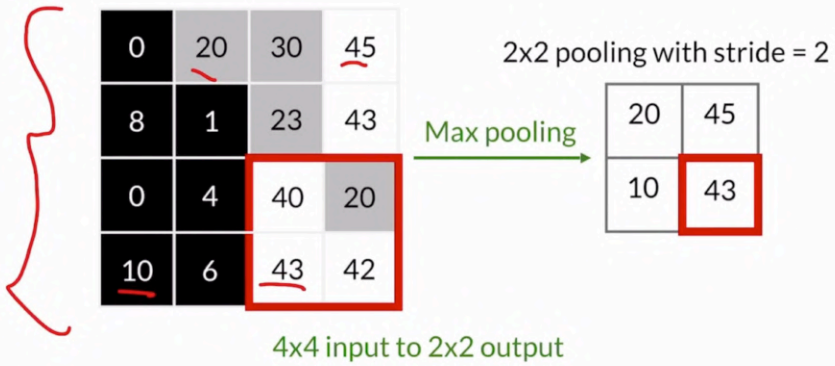


Pooling

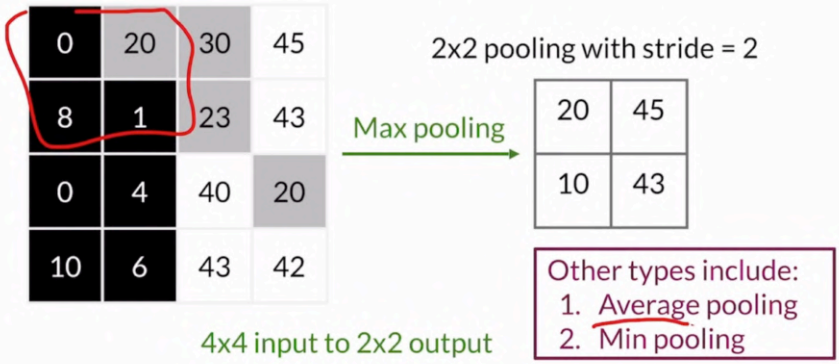


Pooling is used to lower the dimension of the input images by taking the mean or finding the maximum value of different areas. For instance, a picture of this cat after a pooling layer will result in this blurry image with a lower size or lower resolution. In this example, you can see that the color palette and the color distribution over both images is still very similar. The shapes on the blurry image still resemble the original. It'll be much less expensive to do computations on this pooled layer than it is on this original image. Pooling is really just trying to distill that information.

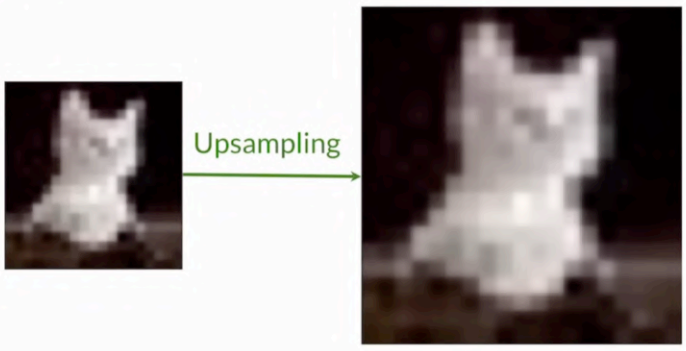
Max Pooling



Max Pooling



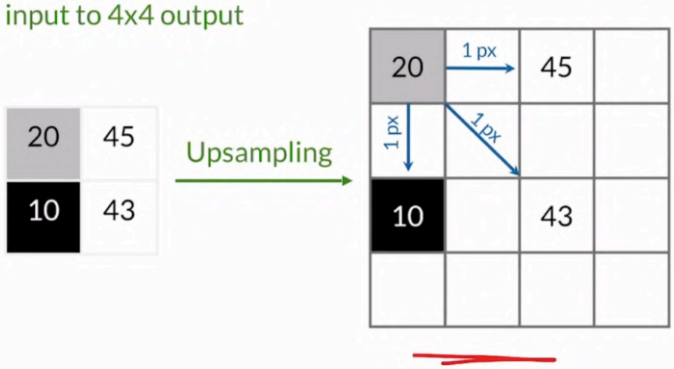
Upsampling



Up-sampling has the opposite effect of pooling. Given this lower resolution image, up-sampling has a goal of outputting one that has higher resolution. You see that it's not perfect. To do this, it actually requires inferring values for the additional pixels and there are a few different methods to do this.

Upsampling: Nearest Neighbors

2x2 input to 4x4 output

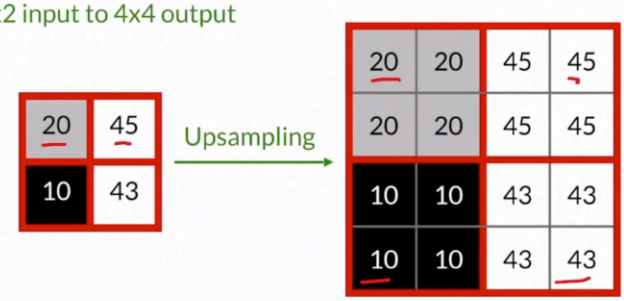


One easy way to up-sample is known as nearest neighbors up-sampling and using this method, you can copy the values of the pixels from your input multiple times to fill your output.

To illustrate, take this two-by-two grayscale image that you want to up-sample to this four-by-four output here. First, you assign the value in the top left corner from the input to the top left pixel in the output. The other values from the input to pixels that are added distance of two pixels from that top left corner and for other cases, this distance might be different depending on the size of the enlargement that you would want. There'll be exactly one pixel in between each of these, including this diagonal. Then assign the same value to every other pixel as it is to its nearest neighbor.

Upsampling: Nearest Neighbors

2x2 input to 4x4 output



For every two-by-two corner in the output, the pixel values will look the same. You can also think of this as putting these values into the corners first and for every other pixel finding its nearest neighbor. Again, pooling for up-sampling, there are many different ways to do up-sampling, like linear and bi-linear interpolation. Feel free to explore what those do and how those are implemented. But you really don't have to worry about implementing them because programming frameworks like TensorFlow and PyTorch actually take care of this whole process for you.

You just need to know that they infer the values from missing pixels in the output by looking at the known values from the input and just like pooling, up-sampling layers don't have any learnable parameters. It's just some fixed rule. These are just different fixed rules.

Upsampling: Nearest Neighbors

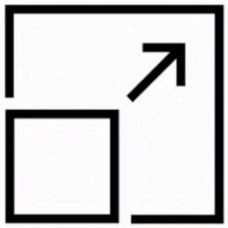
2x2 input to 4x4 output

Other types include:
1. Linear interpolation
2. Bi-linear interpolation

Summary

- Pooling reduces the size of the input
- Upsampling increases the size of the input
- No learnable parameters!

In summary, a pooling layer reduces the size of inputs while up-sampling does the opposite. Unlike convolutions, neither pooling nor up-sampling layers have those learnable parameters, just fixed rules.



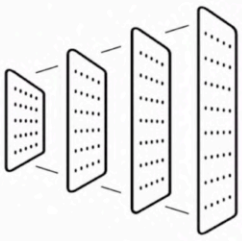
deeplearning.ai

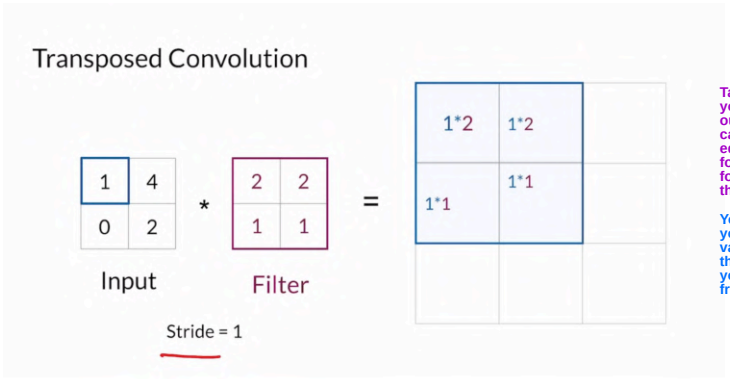
Transposed Convolutions

Outline

- Transposed convolutions as an upsampling technique
- Issues with Transposed Convolutions

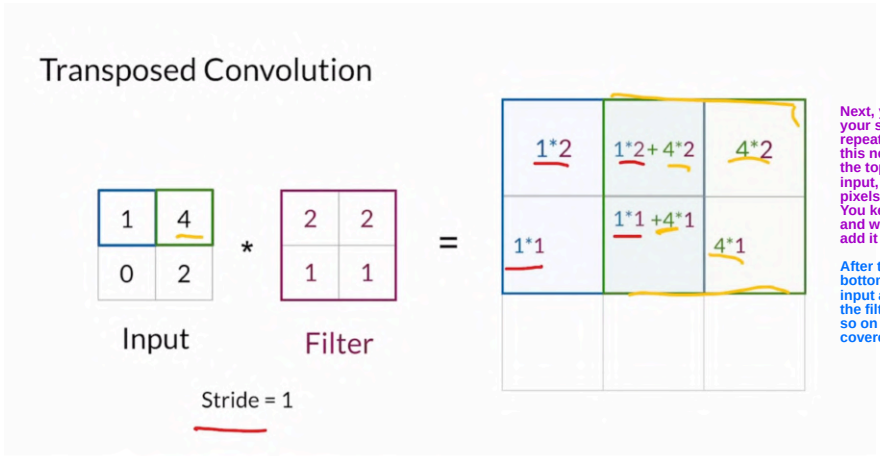
Review transposed convolutions and upsampling technique that uses a learnable filter to enlarge the size of your input, then I'll show you how the outputs from this method have a really particular issue related to the way the pixel values are calculated.





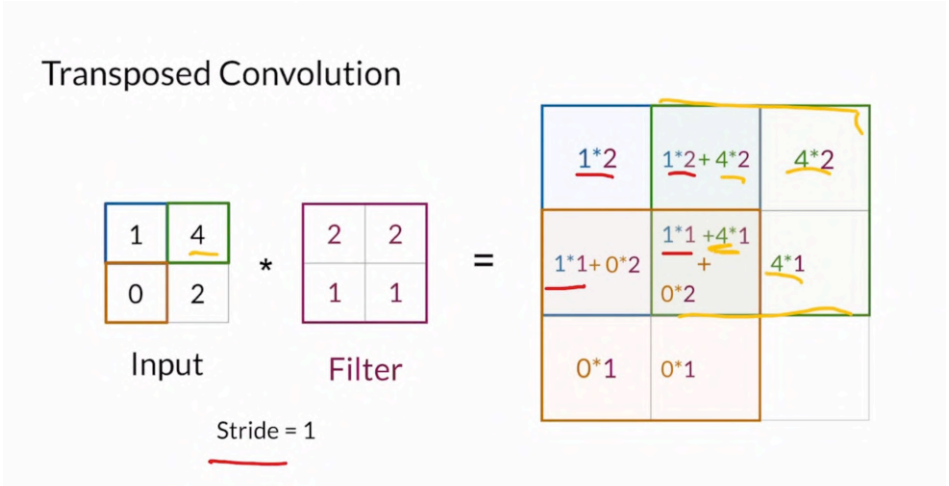
Take this example with a two-by-two input that you want to upsample to a three-by-three output. Using a transposed convolution, you can use a two-by-two learned filter with stride equal to one to accomplish this task by following a really similar procedure to the one for convolutions that I demonstrated earlier in this course.

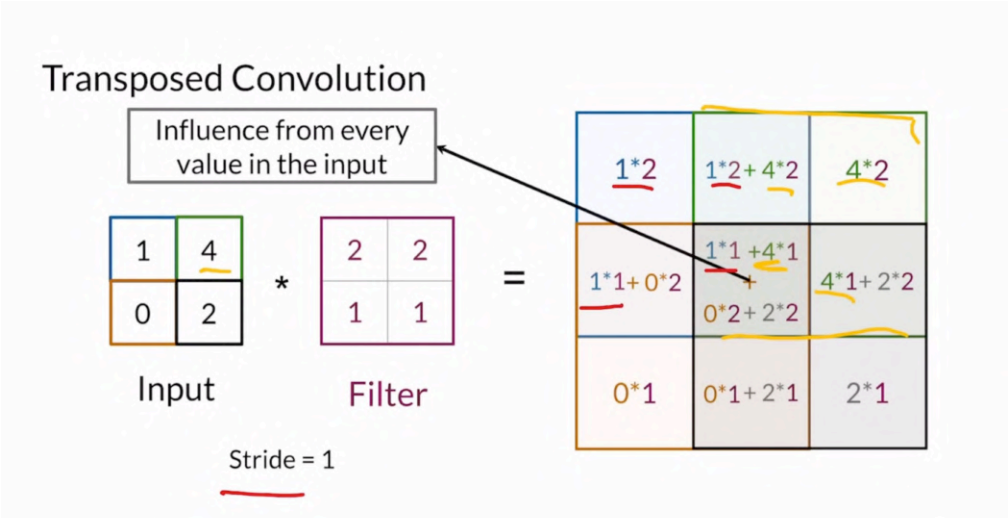
You start by taking the top-left value from your input and getting its product with every value in the two-by-two filter. Then you save this value in the top two-by-two left corner of your output. Here are the multiplied values from that operation.

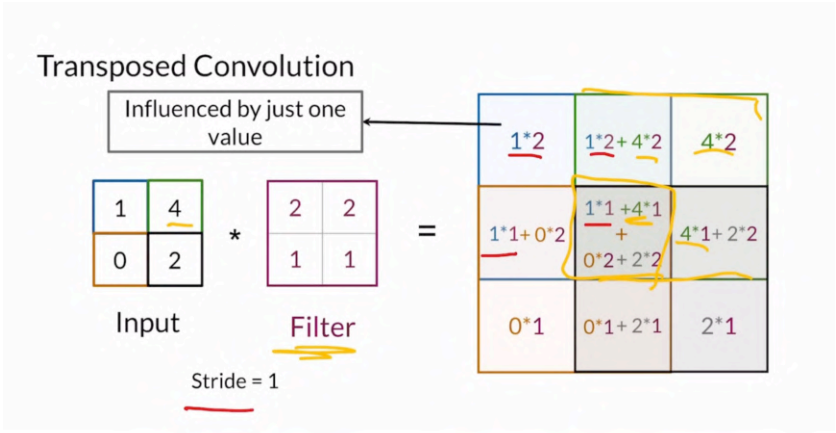


Next, you shift the filter by your stride of one and you repeat the same process on this next pixel and now it's at the top right corner of your input, as well as these four pixels out here in your output. You keep the products there and where there is overlap you add it to the previous product.

After that, you move to the bottom left corner of your input and take the product with the filter, send the result, and so on and so forth until you've covered your entire input.



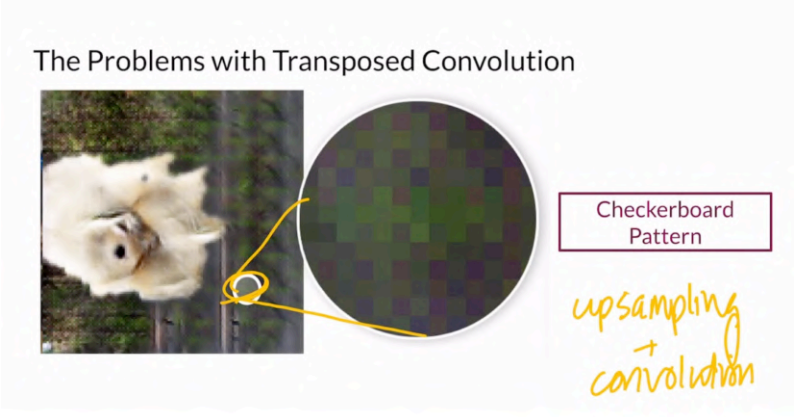




With this computation, you can see that some of the values in the output are influenced much more heavily by the values from the input. For instance, the center pixel in the output is influenced by all the values in the input, while the corners are influenced by just one value.

This is the transposed convolution operation and that's all it's doing in upsampling that. In the filter, these values are learned.

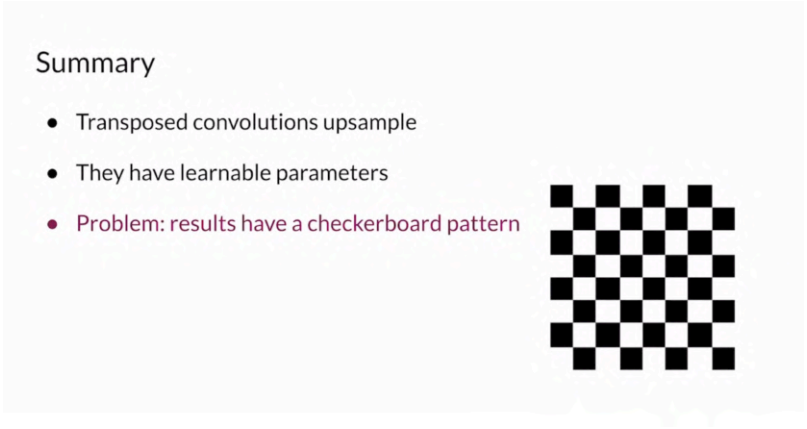
But there is an issue that this center pixel is visited four times and is influenced by all pixels while the other ones are not and this causes a common issue that arises when using transposed convolutions.



This image shows a real output from a transposed convolution where the circle is not actually generated. It's just zoom in to this checkerboard issue.

Notice that the image has a pattern resembling a checkerboard and this arises because when you upsample with a filter, some pixels are influenced much more heavily while the ones around it are not.

Despite this issue, however, transposed convolutions are still fairly popular in the research community, though, using upsampling followed by convolution is becoming a more popular technique now to avoid this checkerboard problem.



To recap, transposed convolutions are used as an upsampling method and they have learnable parameters unlike, upsampling layers.

A problem arising from the use of transposed convolutions, however, is that the output has a checkerboard problem. Nevertheless, these layers are still pretty popular and you'll use them for some of the models in the specialization.

(Optional) A Closer Look at Transposed Convolutions

Now that you have an idea of what transposed convolutions are (also commonly referred to as "deconvolutions") and the checkerboard pattern problem that comes with using them, let's take a closer look! This interactive paper demonstrates the checkerboard pattern problem and how they are not exclusive to GANs, but any neural network that employs them.

Odena, et al., "Deconvolution and Checkerboard Artifacts", Distill, 2016. http://doi.org/10.23915/distill.00003

https://distill.pub/2016/deconv-checkerboard/

(Optional) The DCGAN Paper

Curious about the paper behind the deep convolutional GAN (DCGAN) you just implemented? Check out the paper!

Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks (Radford, Metz, and Chintala, 2016): https://arxiv.org/abs/1511.06434

https://arxiv.org/abs/1511.06434

Works Cited

All of the resources cited in Course 1 Week 2, in one place. You are encouraged to explore these papers/sites if they interest you—for this week, both papers have been included as optional readings! They are listed in the order they appear in the lessons.

From the videos:

- Deconvolution and Checkerboard Artifacts (Odena et al., 2016): http://doi.org/10.23915/distill.00003

From the notebook:

- Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks (Radford, Metz, and Chintala, 2016): https://arxiv.org/abs/1511.06434
- MNIST Database: http://yann.lecun.com/exdb/mnist/