In general, it may be difficult to

determine the appropriate architecture for convolutional neural network. With that in mind, in this section, we're going to discuss some different

architectures, which is help provide a framework as you move towards building

out your own convolutional neural nets. Now, let's go over our learning goals for

this section. In this section, we're going discuss

different architectures using convolutional neural networks. And we're going to specifically talk

about some commonly used network types. When you start off with LeNet,

which is an earlier architecture, so it's going to be more of

a motivating architecture. And is one of the first successes, and

it was used on black and white images. We'll then discuss AlexNet, which is not

really made convolutional neural networks popular as it won the 2012 ImageNet

competition by a landslide. Then we'll discuss VGG, which is a means

of coming up with a simpler overall architecture that still able to

identify more complex features. We'll then discuss inception, which will

be a means of combining different types of layers together within a single layer. And we'll see what that

means in just a bit. And then finally, we have resonant which

is going to be a means of working with much much deeper networks and

still getting high accuracy. So starting off with LeNet, LeNet was

created by Yann Lecun in the 1990s. So again, it's one those earlier

architectures and LeNet was built for MNIST. And the MNIST data set is specifically

numerical values that are hand written between zero and nine. And we want to identify which number

is written for a given image, so they're all going to be black and

white on grayscale. So we're only going to have one channel, if we think back to our

discussion about image data. And he was able to use this

concept of convolutions for the first time to efficiently learn these

features that are built into the data set. Whether those are those edges,

or those loops, or those sharp turns that you

may see in any numerical value. So let's walk through the actual

architecture of working with LeNet. So we have the actual structure

diagram here in front of us and we start off with this input,

which is a 32 by 32 grayscale image. Again, in the original data set we

are working with numerical values in that MNIST data set. And that's why our output at the end is

going to be ten different values is going to predict whether it's a 0,

1, 2, 3, etc., through nine. So ten different possible outputs. Here we have an A, you can just

imagine that is a handwritten digit. And we also have 0 depth here since we had

this on the grayscale, it's in black and white. So we don't have to worry about having

different channels just going to have a depth of 1. So then we have our first

convolutional layer and that's going to be a 5 by 5

convolutional layer with astride of 1. So it's going to be moving across

the image one step at a time, and then down the image one step at a time. And this will have the resulting

output with invention of 28 by 28. And the reason for this, as we discuss, if we're moving that 5 by 5 filter

across our image and down our image. We're actually going to be reducing

the number of dimensions each time we take those steps, especially,

if there's no padding. We also are going to use a depth of six,

so this means we will results in six

different kernels that are being learned. So our filter will have

six different kernels and will have that output that we

have here of 6 by 28 by 28. So the next layer does have a depth and

that depth is six. So you want to think how many

weights do we need to learn for this particular layer? And if we think about what the size

of our kernel is and that is 5 by 5. So we have 25 weights there,

then we add on the bias term, so we end up with 26 weights. And then we think about

the depth of that filter. So we multiply that 26,

which is just one kernel times 6 for the depth to come to 156 weights

being learned at that first layer. Next, we have a pooling layer

with stride equal to 2. So it's going to be no weights needed to

be learned as pooling is just say fixed operation. But we want to note that here, given that we're working with

this older architecture. The original paper actually does

a more complicated pooling than max or average pooling. But this is essentially

considered obsolete by now. So if you are going to be using this, you would probably use

something like max pooling. We then have another 5 by 5 filter, again, with stride equal to 1 and

with no padding. So again, we're going to be

reducing our size even further. So this time we go to depth of 16,

so we went from 6 out to 16 and we reduced that output size

to 10 by 10 as we discussed. As we moved out 5 by 5 filter,

it will reduce the actual size of that next layer, and

then we have that depth again of 16. The kernels will be taking in the full

depth of that previous layer and that full depth is equal to 6. So each of 5 by 5 kernel now

looks at 6 times 5 by 5 pixels. Not just the 5 by 5 or the 32 by 32,

as we saw in the original layer. But now, in order to calculate each one

of the individual pixels in each one of our 16 dimensions, we now have to

look at 6 times 5 times 5 pixels. So because each kernelized 6 times 5 times 5 different weights

that are being learned. We have 150 weights plus that bias term,

so equal to 151 for each particular kernel. And then to get the total weights for

this layer, we multiply this by our new depth,

which is 16. So we're learning here 2,416 weights,

which is just 16 times 151. And we have our output here,

which is 16 by 5 by 5. We can then flatten this into a vector,

which is a 400 vector. And now, we are just working

with fully connected layers, so we can go from 400 down to 120,

and then from 120 down to 84. Then ultimately, from 84 down to 10 and

allowed us to ultimately predict which class we are actually working

with for our numbers zero through 9. So that softmax output size 10 for

each one of those ten digits. So how many weights did we actually have

to train as we walked through this LeNet structure? So you think about that first layer, we only had 156 weights in

that convolutional layer. Then 2,416, and then moving to

those fully connected layers. That's when the numbers really jump up and

we see 48,000 and 10,000, then 850, and then ultimately, we had a total of 61,000 different

weights or 61,706 weights. Now, this is always going to be

less than the equivalent for a fully connected layer. And with that, we want to know in the

major takeaway that we want to take from here is that convolutional layers

are generally going to have relatively few weights compared to these

fully connected networks. This structure that we

just walked through, the LeNet structure is still used today

in regards to that convolutional layer. Then the pooling layer and again,

the convolutional layer, ultimately, leading to those fully

connected layers at the end. That closed out discussion

here in regards to LeNet. In the next video, we will discuss

the AlexNet structure which was used to win ImageNet back in 2012. All right, I'll see you there.

Now let's discuss AlexNet. AlexNet is named after Alex Krivhevsky,

one of its main creator, and if you watched the intro

course that we had here for all the courses within this learning path,

you would recall that this was when convolutional neural

Nets really hit the mainstage. And that was due to the fact that it

won the competition here on ImageNet. The goal of this competition was to

predict the correct label from among 1000 different classes and this was amongst

1.2 million different images, so working with a very large data set or the

large classification problem in general. Now again, AlexNet is considered the flash

point for modern deep learning in general. This is due to the fact that

it demolished the competition, had a top 5 error rate of 15.4%

whereas the next best was 26.2%. So let's dive into AlexNet under the hood. Now here we have an actual

diagram of the AlexNet. Now don't be too nervous about

this breakdown of layers. The reason why we have two separate

paths that this network is walking through is that in order to

run model on such a large data set, what they actually did was split

it up into two parallel paths. So rather than thinking about, say,

this first layer that we have here which we see is 55 by 55 by 48 twice, you can

imagine it using your normal convolutional layer that this would be something

along the lines of 55 by 55 by 96. Where that depth of 96 is split

into two parallel paths, and then you see the same with the next

layer in every layer moving forward. So in the next layer you can

imagine rather again and 27 by 27 by 128,

it would be 27 by 27 by 256. And when we look at this large

network along with those dense fully connected layers at the back end, there was actually 60 million different

parameters that had to be learned. So that parallelization

was very important and also this would take weeks

to actually learn but again, had this very high performance of

knocking out the competition with that difference that we just

saw of 15% to around 26%. So now I want to go over

just a few details, a few more details in regards to AlexNet. So first off, the AlexNet developers

performed data augmentation before feeding through these

images through the network. And they did things such as cropping,

horizontal flipping and other manipulations, and

that augmentation helped with overfitting. So if you think about working with

an image of a cat for example, and you were to crop down the image but

still had an image of a cat, or did a horizontal flipping, but again

doing the horizontal flip of an image, you'd still have that image of a cat. You'd be able to avoid over

fitting to those exact images and learn the extra features that

actually go into what makes up a cat. Now the basic template which we just saw

in the last slide is that we'll have convolutions with ReLUs and ReLUs has

those activation functions that is, and ReLUs were fairly

new to use at the time. And that was a major part of why

they were able to create this huge breakthrough to train such a large

network and with that, they would sometimes add on that maxpooling

later after convolutional layers. And as we've seen before at the end, there

was a fully connected layer that led to that softmax classifier that allow you

to identify the class of that image. Now that closes out AlexNet. Now I want to move on to VGG. And with VGG, now AlexNet that is we were talking

more about things that have worked and they were similar to what we saw when we

discussed convolutional Nets in general. With VGG, we're going to see how

to greatly simplify implementation convolutional neural

networks in general and will be a powerful baseline as well

as just a powerful model in general. So we simplify our network by avoiding

having to make manual choices for our convolution sizes. That is, whether we need three by three,

five by five and so on. And we do so by just sticking with

three by three convolutions, but we make our networks very deep. And as well see, this will effectively

give rise to larger convolutions to account for the fact that we're limiting

ourselves to only three by three filters. So here we have an actual

diagram of VGG and in a lot Places similar to

what we've seen so far. Where all the way out to the left, we have

our input, which is going to be an image. We see it's a colored image, so

it's going to have three channels. We then pass that through

those convolutional layers. We see those convolutional layers

are the ones that have no color or black background versus the max pooling,

which is our red. Which will reduce the size of

each one of our different layers. And then at the end all

the way out to the right, we have our fully connected

layers as well as our softmax. So in that regard, it's going to be

very similar to what we've seen so far with Lynette and Alex Net. What differentiates VGG is that it

only uses these 3 by 3 filters. And we're going to see how those 3 by

3 filters an a deeper network will accomplish similar goals to working with

a 7 by 7 or a 5 by 5 filter as well. So thinking through the way it would

work in regards to moving from our input in layer 1 all the way

through to say here layer 3. We can imagine that we're doing our

convolutional operation on this red portion of the pixels that

we have here in our input. And that would lead to that single square

that we have here in layer 2, right? In order to calculate layer 2,

we have to do a convolutional operation on those nine squares that

we have in layer 1. Now we can then say that be receptive

fields of layer 2 is 3 by 3 or in other words, each cell in layer 2 was calculated using a 3 by

3 input from layer 1. Now we know that layer 2 for each one

of the cells it took some 3 by 3 input. Now what about layer 3? We know that we're sticking

with using that 3 by 3 filter. That's going to be at the core

of this VGG architecture. So each cell in layer 3 will now

be made up of a convolutional operation of the 3 by

3 Patch from layer 2. And that's again for each cell that we see

here in layer 3 is going to be made up of that 3 by 3 convolution. So what does this mean though,

in regards to how much of layer 1? Layer 2 is actually using for

each one of our cells. So let's move back to layer 1. We know that each one of

these layer 2 components or made up of a 3 by 3

portion from that layer 1. And if we think through all

the cells from layer 1 that made up the 3 by 3 filter that

we're now using in layer 2. We start to see all the particular

cells used to make that layer, and we can move across. And as we continue to move along,

we see that we use that top 3 by 5 cells. So we have 3 ReLu, then 5 columns

that made up that top row of layer 2. We can then move down and

repeat these steps for that next layer. And continue to do this until we've

made up the entire convolutional layer that ultimately led to

the calculation that we have in layer 3. So each square in layer 3. If you look back to that layer 1

from which that layer 3 value was calculated is actually seeing

a 5 by 5 grid from that layer 1. Because again, that 5 by 5 grid was

able to make up that 3 by 3 grid that we have in layer 2, which ultimately

led to that one value in layer 3. So with this in mind, let's talk through how VGG actually ends

up reducing the number of weights that we actually need to learn compared to working

with these 7 by 7 or 5 by 5 layers. So we saw just before

that two 3 by 3 layers or the stride of 1 is equivalent

to one 5 by 5 filter. And on that same note, you can extend the

logic that we just used to find that three 3 by 3 convolutions is equivalent

to one 7 by 7 convolution. Now, if you think about the amount

of weights that we need to learn for one 3 by 3 layer. We obviously only need to use 9. Whereas for one 7 by 7 layer we're going

to need to learn 49 different weights. And If we were to think of. Three three-by-three layers which we just

established is going to be equivalent to one seven-by-seven layer. That will just be 3 times at

9 that we just discussed. There for

27 weights that we need to learn. So we see here the reduction and

the number of weights that we need to learn going from the 49 if we

were to use a seven-by-seven filter. Going down to three three-by-threes,

which will be equal to 27 weights. So we actually have to learn

less weights if we use VGG, that will come to similar solutions. So VGG is one of the first architectures

to really experiment with many layers and come up with this idea

of working with more is better in regards

to convolutional nets. And it showed how we can use multiple

three-by-three convolutions to simulate larger kernels while learning less

parameters as we just saw in the last slide. And served as base model for future works. Understanding how we can use these smaller

convolutions build up to those larger convolutions. Now that closes our discussion

here on VGG, and again, VGG is specific to working with the same

filter here, three-by-three filters, and how that can build up to larger filters. In the next video we're going to discuss

how within a layer we can have many different filters and

many different types of layers. And we can bring them altogether

in regards to one layer, and I'll make this clear in the next video,

using a framework called Inception. All right, I'll see you there.

Now, I'd like to talk about

the Inception architecture. Now, with Inception, the idea

is perhaps you don't know exactly what type of filter or what type of layer you

want at each step, so you may want to combine or try a bunch of them together. But this can be

computationally expensive. We probably want to

accomplish this with some level of

computational efficiency. We're also going to want to ensure that we can

reduce the total number of activations that are needed to run through our

entire network. Our solution with the

Inception architecture, would be to turn each individual

layer into branches of convolutions rather than just working with a

single filter type. Each of these

branches are going to handle a small portion

of the workload, and then each layer

will concatenate the different branches to

complete a single layer. Let's see a visual of this. What we see here is

we are moving through our previous layer to the next layer using one

of these inception blocks. This is our first idea of this inception block

that we see here, where we have the previous layer, and then we're going

to concatenate many different types

of convolutions as well as max pooling, which will make up

that next layer. We'll have one-by-one

convolutions make up a certain depth, three-by-three convolutions,

five-by-five, and so on. Then we concatenate all those together to get our full depth. Then we can run our

activation function through that concatenated

version of that layer. Now, the way that it's laid out, if we use the reducing filters

from the previous layer using three-by-three and

five-by-five convolutions, so we run three-by-three and

five-by-five convolutions, through the full depth

of the previous layer, we should recall that

we're going to have to have a value for every

single channel or every level of depth of that prior layer in order to

get each individual value. Thus, we end up

requiring a ton of operations to complete the

calculation for that filter. Now, instead of what we just saw, what we can do is first, as we've look at these

one-by-one convolutions, is we can run this

one-by-one convolution, and that one-by-one convolution may at first seem meaningless, but recall that we're also

working with the entire depth, so we would have a

different single number for each level of the new depth that we're trying to calculate. Similar to when we work with different three-by-three

kernels to come up with different depths. With a three-by-three kernel, we do the same with a one-by-one, but this time we're just

multiplying by a single value. By first doing these

one-by-one convolutions, we can reduce the depth without nearly as many

calculations as would be needed if we did a three-by-three or a five-by-five filter. Then once we have

reduced that depth, then we can do our

five-by-five convolutions, or our three-by-three

convolutions with much fewer

operations required. Thus reducing the

computational complexity. Now with that, we also have this pooling that we see all

the way out to the right. For max pooling, we still

are going to end up with that same number of channels or that same depth if we did

pooling on that previous layer. Doing the one-by-one convolutions

after the max pooling, allows us to again reduce that depth to whatever

depth we want before concatenating together

all the different types of convolutions that we have

here in our inception layer. This whole block serves the function of that previous

convolutional layer. We combine these all together, and then once we combine

these all together, then we pass it through

our activation function. To see what this looks

like for a full network, we have our input coming

in from the left, and we see that we have

multiple different convolutions within the single layer where we have triple and five, so three-by-three or five-by-five convolutions as well as

perhaps an average pool. Then once we concatenate

those all together, then we can run the softmax, and we can continue to do

that with different types of convolutions at each one

of the different layers, feeding all the way

through to our output. Now I'd like to

talk briefly about our final architecture,

the ResNet architecture. I want to start off with

the motivation behind it. Now, researchers were building deeper and deeper

networks as they started to realize the power of convolutional neural nets. But they started to find that as they built out

these deeper networks, they were actually

tending to have worse performance

on deeper networks. We see this here on the

training error with the 20 layer versus

the 56 layer network, that we actually have a

higher training error with that higher layer network. Hopefully you don't

think that this is intuitive because

this is intuitive as we see here for perhaps the test error on our holdout set. But when focusing just on

our training set, ideally, we should just be getting

better and better in regards to the training error just

on that training set that we learned the model on. This is surprising, again because deeper networks should overfit more and should do better

on the training set. So what was happening? Earlier layers of deep networks were very slow to adjust so it was hard to adjust those earlier layers

within the network. Now it gets to that

vanishing gradient issue as we move towards the front, as we do back propagation

and move towards the front of our neural network. This is happening

that we are having this lower performance on the training set, when in theory, we should be able to just have an identity

transformation that makes the deeper network behave

just like the shallower one. If our 20 layer network

is doing well and we add on another layer

or another 30 layers, there's no reason why

we would do worse, because we can just add on identity layers that keep

it exactly the same. But our Convolutional

Neural Nets, due to the vanishing

gradient issue, weren't able to learn these identity matrices or identity transformations

in any type of way. The assumption that will

make ResNet possible, so ResNet is the solution

to this problem, will be that the best

transformation over multiple layers will be

close to F of x plus x. Where x is going to be our

input to the series of layers, and we see that here in our

diagram where we have that x and F of x is our function

represented by several layers, say convolutions here with their ReLU activations as we see in the diagram in between. We can then take that layer

as linear transformations, as well as the added on

initial weights that x, and pass them through

that current ReLU. This shortcut allows for

the information from those earlier layers to easily

pass through our network. We can continue to

do this throughout the network to ensure

that prior layers, say two layers, priors, and not just that initial input, as you may have

thought about with x, can continue to be added to the output of the

most recent layer. The idea basically is keep passing that initial information unchanged to the next layer, as well as that transformed information as we move

along our network. Now, this will actually allow you to continue to pass through past information if we just set our weights to zero

for our new layers. So it's possible to

just have ReLU of that shortcut connection

represented by the loop. What goes wrong is that as

we go deeper and deeper, it becomes difficult to

even learn something like that identity function due to the vanishing gradient issue. But if we allow for that, again, initial value to

be passed through, then we can hold on to that value from

that earlier layer. Add again, zero weights to

that new information coming in and effectively have that

identity transformation as we move through our network. Now to recap, in this section, we discussed many different common architectures

that we should be aware of when we're working with convolutional neural nets. We started off with LeNet, which was an earlier

version where we saw the framework of that

convolutional layer, then the pooling layer, and then those fully

connected layers first being introduced

using the m-nest data set. Then we discussed AlexNet, which was introducing

ReLUs into the equation, as well as other breakthroughs

and efficiencies to create that complex network

that ultimately blew away the ImageNet

competition back in 2012. We then discussed VGG and how

that allowed for a simpler, powerful framework

that accommodated for its simplicity

with deeper networks. We discussed the Inception model, which allowed us to include

multiple layer types within a single layer while maintaining computational

efficiency. Then finally, we discuss ResNet, which allowed for

maintaining information from earlier in the

network to ensure that we can build out incredibly

deep networks while still continuing to

reduce training error. Now that I've closed

out discussion on Convolutional Neural Networks, which are very powerful

for image data. But in the next

video we're going to introduce our next

major framework that's going to be

powerful for working with text and time series data, namely recurrent neural networks. All right, I'll see you there.

SUMMARY

# Summary/Review

## **CNNs**

Convolutional Layers have relatively few weights and more layers than other architectures. In practice, data scientists add layers to CNNs to solve specific problems using Transfer Learning.

## **Transfer Learning**

The main idea of Transfer Learning consists of keeping early layers of a pre-trained network and re-train the later layers for a specific application.

Last layers in the network capture features that are more particular to the specific data you are trying to classify.

Later layers are easier to train as adjusting their weights has a more immediate impact on the final result.

**Guiding Principles for Fine Tuning**

While there are no rules of thumb, these are some guiding principles to keep in mind:

* The more similar your data and problem are to the source data of the pre-trained network, the less intensive fine-tuning will be.
* If your data is substantially different in nature than the data the source model was trained on, Transfer Learning may be of little value.

## **CNN Architectures**

**LeNet-5**

* Created by Yann LeCun in the 1990s
* Used on the MNIST data set.
* Novel Idea: Use convolutions to efficiently learn features on data set.

**AlexNet**

* Considered the “flash point” for modern deep learning
* Created in 2012 for the ImageNet Large Scale Visual Recognition Challenge (ILSVRC).
* Task: predict the correct label from among 1000 classes.
* Dataset: around 1.2 million images.

AlexNet developers performed data augmentation for training.

* Cropping, horizontal flipping, and other manipulations.

Basic AlexNet Template:

* Convolutions with ReLUs.
* Sometimes add maxpool after convolutional layer.
* Fully connected layers at the end before a softmax classifier.

**VGG**

Simplify Network Structure: has same concepts and ideas from LeNet, considerably deeper.

This architecture avoids Manual Choices of Convolution Size and has very Deep Network with 3x3 Convolutions.

These structures tend to give rise to larger convolutions.

This was one of the first architectures to experiment with many layers (More is better!). It can use multiple 3x3 convolutions to simulate larger kernels with fewer parameters and it served as ”base model” for future works.

**Inception**

Ideated by Szegedy et al 2014, this architecture was built to turn each layer of the neural network into further branches of convolutions. Each branch handles a smaller portion of workload.

The network concatenates different branches at the end. These networks use different receptive fields and have sparse activations of groups of neurons.

Inception V3 is a relevant example of an Inception architecture.

**ResNet**

Researchers were building deeper and deeper networks but started finding these issues:

In theory, the very deep (56-layer) networks should fit the training data better (even if they overfit) but that was not happening.

Seemed that the early layers were just not getting updated and the signal got lost (due to vanishing gradient type issues).

These are the main reasons why adding layers does not always decrease training error:

* Early layers of Deep Networks are very slow to adjust.
* Analogous to “Vanishing Gradient” issue.
* In theory, should be able to just have an “identity” transformation that makes the deeper network behave like a shallow one.

In a nutshell, a ResNet:

* Has several layers such as convolutions
* Enforces “best transformation” by adding “shortcut connections”.
* Adds the inputs from an earlier layer to the output of current layer.
* Keeps passing both the the initial unchanged information and the transformed information to the next layer.