Contents

Convolutional neural network (CNN)

- Most generally, we can think of a CNN as an artificial neural network specialization for being able to *pick out or detect patterns*. This pattern detection is what makes CNN so useful for image analysis.a
- If a CNN is just an artificial neural network, though, then what differentiates it from a standard multilayer perceptron or MLP?

  CNN have hidden layers called *convolutional* layers, and these layers are what make a CNN, well... a CNN. CNN can, and usually do, have other, non-convolutional layers as well, but the basis of a CNN is the convolutional layers.

Convolutional layers

- Just like the other layer, convolutional layer receives input, transform the input in some way and then outputs the transformed input to the next layer.

The inputs to convolutional layers are called input channels, and outputs are called output channels.

- Let's look at a high-level idea of what convolutional layers are doing

Filters and convolution operations

- We know that convolutional neural networks can detect patterns in images.

With each convolutional layer, we need to specify the number of filters the layer should have. These filters are what detect the patterns.

Patterns

- We said that filters can detect the patterns. Think about how much may be going on in any single image. Multiple edges, shapes, textures, objects etc. These are what we mean by patterns

  Edges

  Shapes

  Textures

  Curves

Objects

Colors

- One type of patten that a filter can detect in an image is edges, so this filter would be called an edge detector. Aside from the edges filter can detect corners, circles square This type of simple geometric are what we'd see at the start of a convolutional neural network.
- The deeper the network goes, the more sophisticated the filters be come. In later layer, rather than edges and simple shapes, our filter may be able to detect specific objects like ears, hair, eyes, feather, scales etc.
- In even deeper object filter can detect more sophisticated objects like full dog, cats, lizards, and birds.
- To understand what's happening here with these convolutional layers and their respective filters, let's look at an example.

Filter

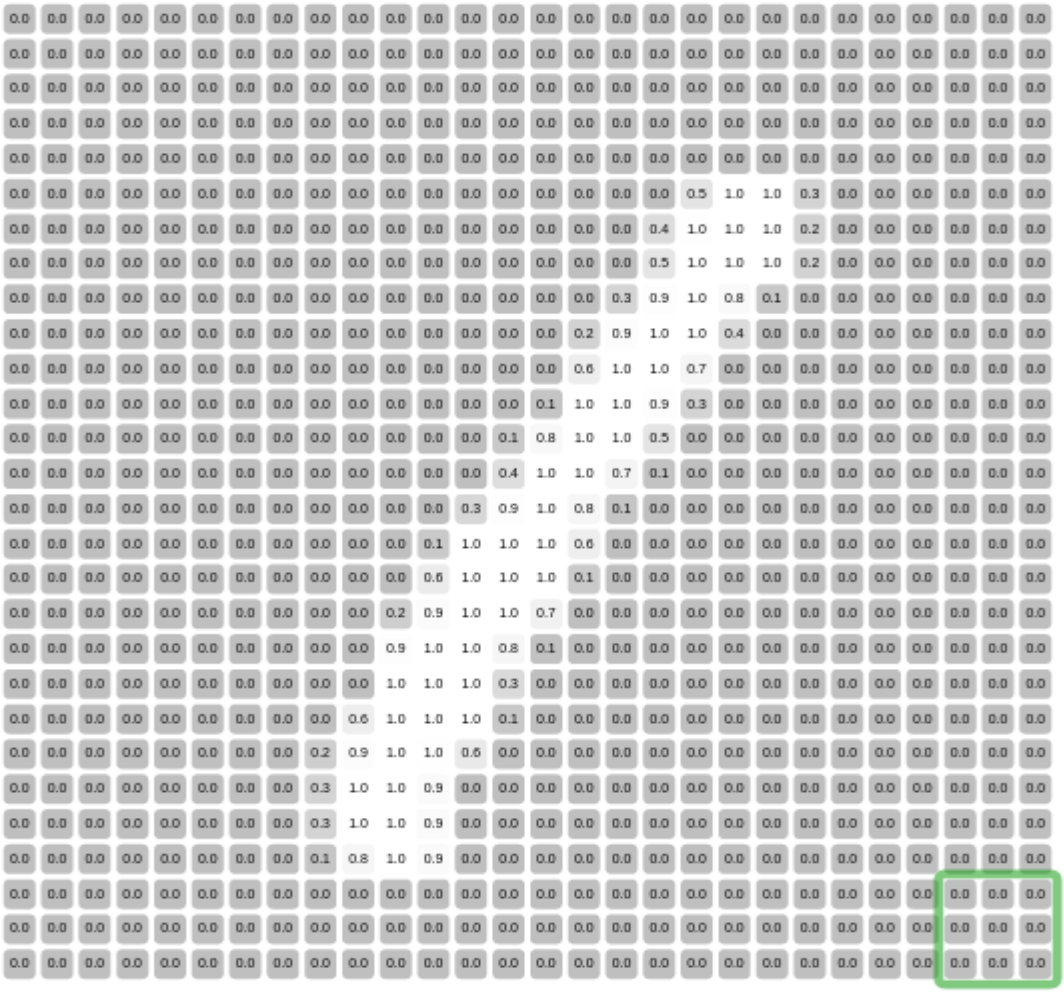
- ***Filter can be technically though as relatively small metrics,***
- for which we decide the number of rows and number of columns that this matrix has, and values within the matrix are initialized with random numbers.
- So, for this convolutional layers in this example of ours we are going to specify layers that contain one filter of size 3 by 3.
- Now this convolutional receives input, the filter will slide over each 3 * 3 set of pixels from the input itself until it slides over every 3 * 3 block of pixel from entire images. This sliding is referred to as convolving, so we should say that filter is going to convolve across each 3 * 3 block pixel from the input.

Example: -

- Let's say we have to identity the hand-written number 1 from the image

- Handwritten number 1 could look like this in image

This is



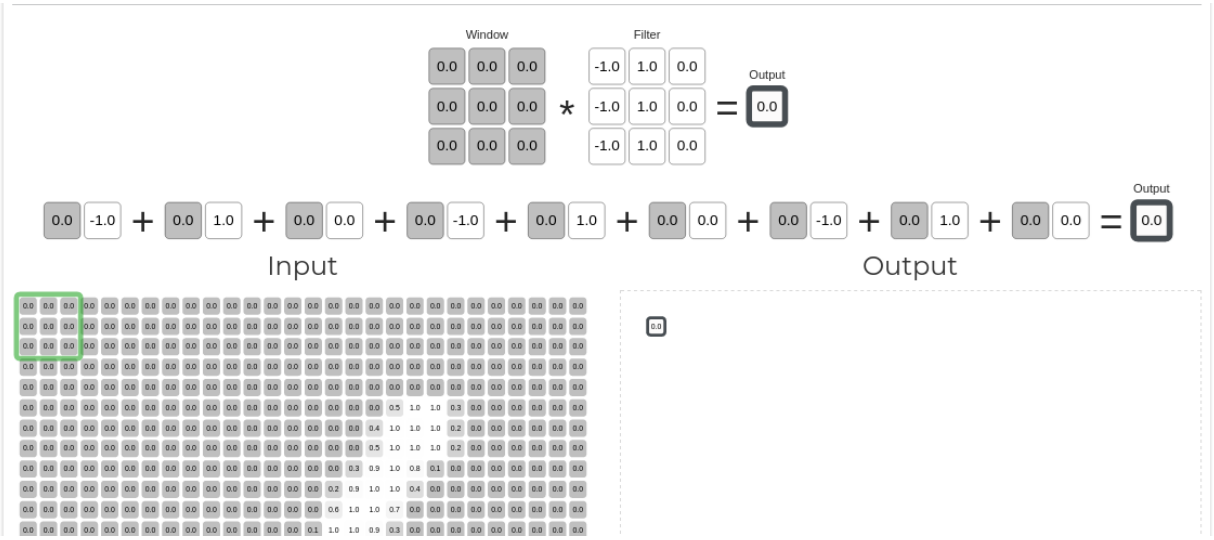
Grey-scale image of number 1 (***python of function that could transform image into this***).

- We all know that CNN have convolutional layer, in that layer we have filter that could detect pattern.
- Filter can be assumed as a small metrics (let's say 3*3 metrics here)

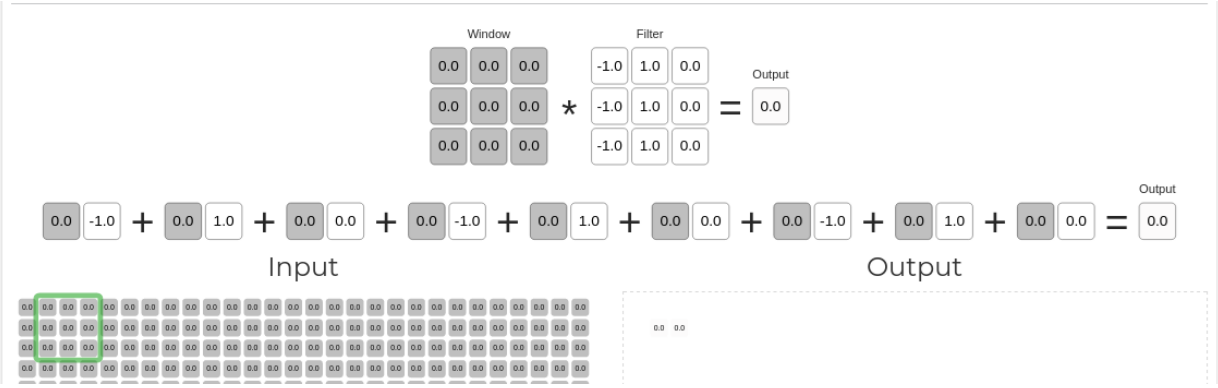
Now it will take 3 * 3 images from the above image and multiply that with filter (which means small metrics)

There are different types of filters available (like top edge filter, bottom edge filter, right edge filter etc.)

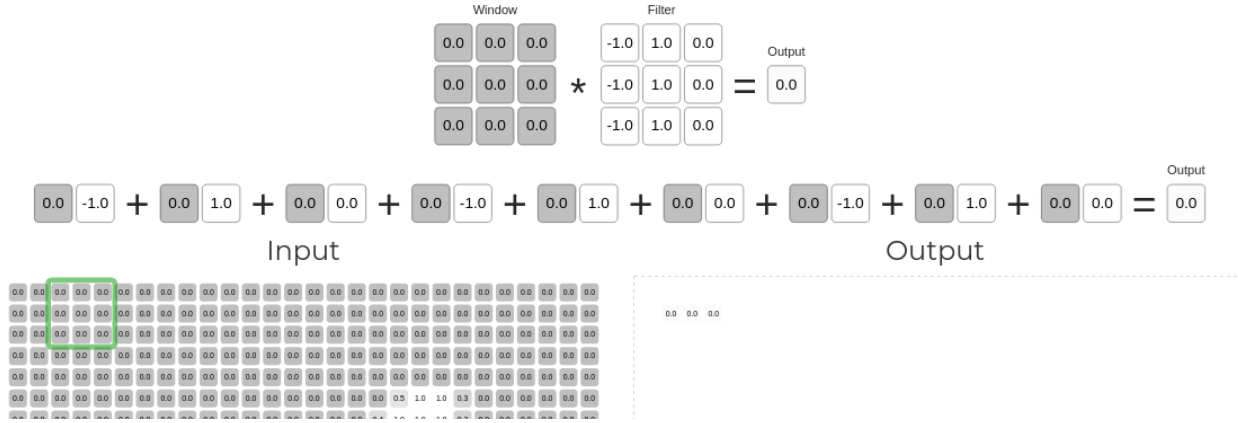
- At first it will take first 3 * 3 image like this, and multiply with filter (



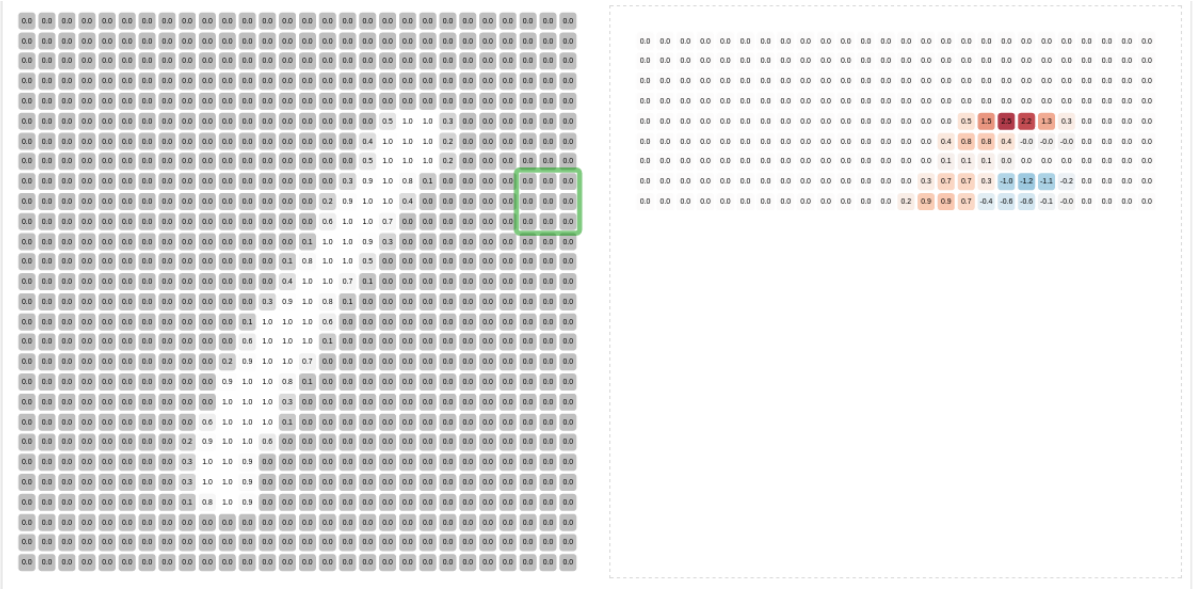
- Then in second step it look like this



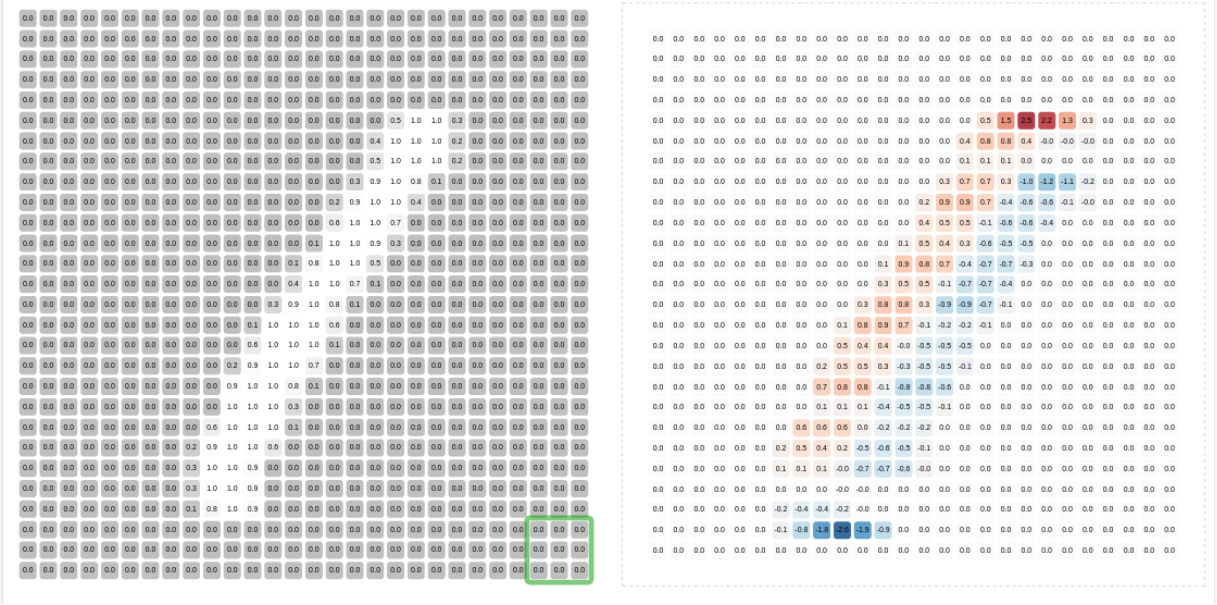
- In third step it look like this



- In some middle step it looks like this



- At the end of this it looks like this



- At the end of this it looks like this. Here you can see that it has detected some pattern. One in Reddish and other is in blue color. Where Dark red indicate the strong positive, and dark blue is consider as strong negative.

Convolutional layer

- Convolutional layers receive an input channel, and the filter will slide over each 3 * 3 set of pixels of the input itself until it's slid over 3*3 block of pixels from the entire image.

Convolution operation

- The input is passed to a convolutional layer.
- As just discussed, we've specified the first convolutional layer to only have one filter, and this filter is going to convolve across each 3 x 3 block of pixels from the input. When the filter lands on its first 3 x 3 block of pixels, the dot product of the filter itself with the 3 x 3 block of pixels from the input will be computed and stored. This will occur for each 3 x 3 block of pixels that the filter convolves.
- For example, we take the dot product of the filter with the first 3 x 3 block of pixels, and then that result is stored in the output channel. Then, the filter slides to the next 3 x 3 block, computes the dot product, and stores the value as the next pixel in the output channel.

A note about the usage of the "dot product"

- Technically what we're doing is summing the element-wise products of each pair of elements

For example, suppose we have two `3 x 3` matrices A and B as follows.

$$A = \begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,1} & a_{3,2} & a_{3,3} \end{bmatrix}$$

$$B = \begin{bmatrix} b_{1,1} & b_{1,2} & b_{1,3} \\ b_{2,1} & b_{2,2} & b_{2,3} \\ b_{3,1} & b_{3,2} & b_{3,3} \end{bmatrix}$$

Then, we sum the pairwise products like this:

$$a_{1,1}b_{1,1} + a_{1,2}b_{1,2} + \cdots + a_{3,3}b_{3,3}$$

in the two matrices

- So, technically this operation is the *summation of the element-wise products*. Even so, you may still encounter the term "dot product" used loosely to refer to this operation

Quiz

- A CNN develops ***filters*** during training that act as pattern detectors
- A filter can technically just be thought of as a relatively small ***matrices***, for which we decide the number of rows and columns

Visualizing Convolutional Filter From A CNN

- We are going to use Keras, a neural network API, to visualize the filter of the convolutional layers from the VGG16 network.

  VGG16 is a CNN that won ImageNet comparison. This is a competition where teams build algorithm to compete on visual recognition tasks.

- First step is to import pre-trained **VGG16** model

```
# build the VGG16 network with ImageNet weights
model = vgg16.VGG16(weights='imagenet', include_top=False)
```

- Then we define loss function

  Loss function has an objective to maximize the activation of a given filter within given layer.

  Then we calculate **gradient ascent** with regard tour filter's activation loss.

```
# we build a loss function that maximizes the activation
# of the nth filter of the layer considered
layer_output = layer_dict[layer_name].output
if K.image_data_format() == 'channels_first':
    loss = K.mean(layer_output[:, filter_index, :, :])
else:
    loss = K.mean(layer_output[:, :, :, filter_index])

# we compute the gradient of the input picture wrt this loss
grads = K.gradients(loss, input_img)[0]
```

- **Gradient ascent** maximizes loss function.
- **Gradient decent** minimize loss function

- Here we can think maximizing our loss here as basically trying to activate the filter a mush as possible, in order to able to visually inspect what type of pattern is detecting by filter.

- Then we pass the network as plain grey image with some random noise as input

```
# we start from a gray image with some random noise
if K.image_data_format() == 'channels_first':
    input_img_data = np.random.random((1, 3, img_width, img_height))
else:
    input_img_data = np.random.random((1, img_width, img_height, 3))
input_img_data = (input_img_data - 0.5) * 20 + 128
```

- After we maximize the loss, we're then able to obtain a visual representation of what sort of input maximizes the activation for each filter in each layer.

```
# save the result to disk
save_img('stitched_filters_%dx%d.png' % (n, n), stitched_filters)
```

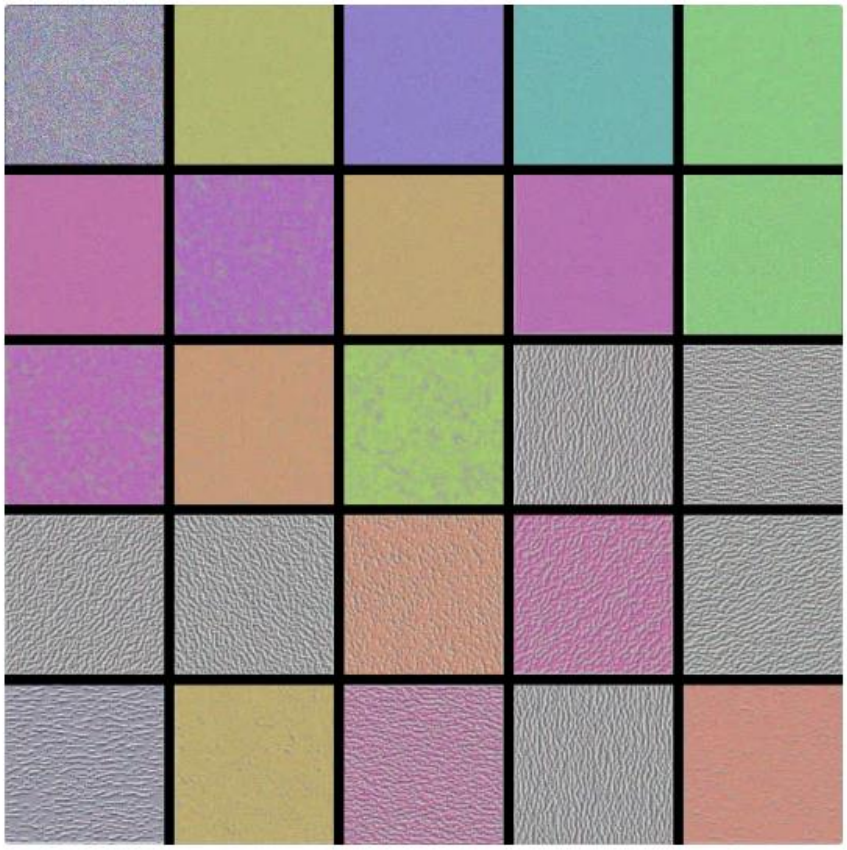
- This is generated from the original gray image that we supplied the network.

  To run this code, it did take a bit of time running on a CPU. Maybe about an hour to generate all the visualizations.

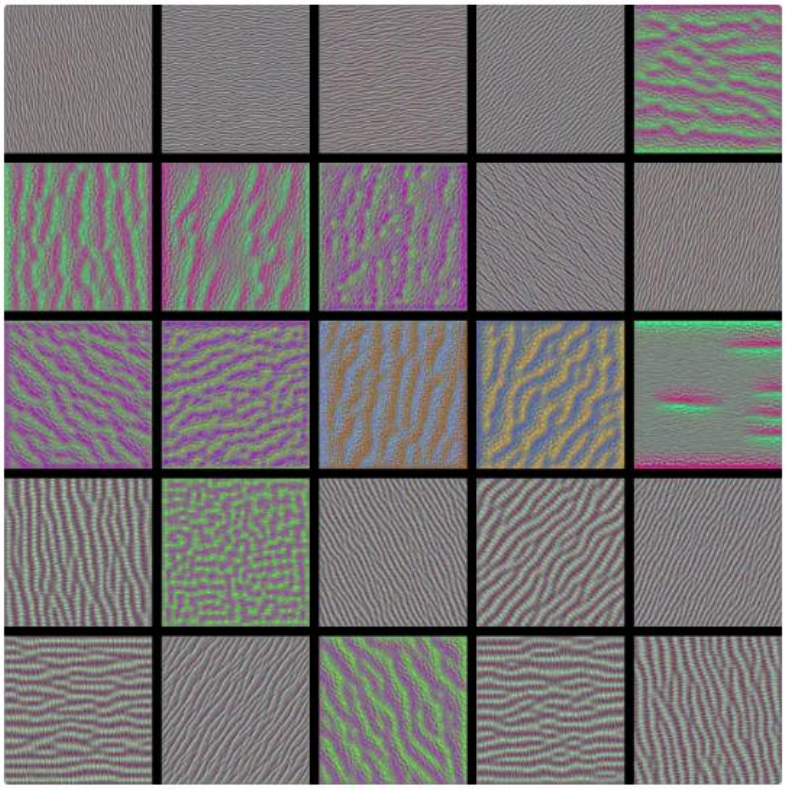
  That's a summary of what our code is doing. Now, let's get to the cool part and step through some of these generated visualizations from each convolutional layer

Generated CNN Layer Visualizations

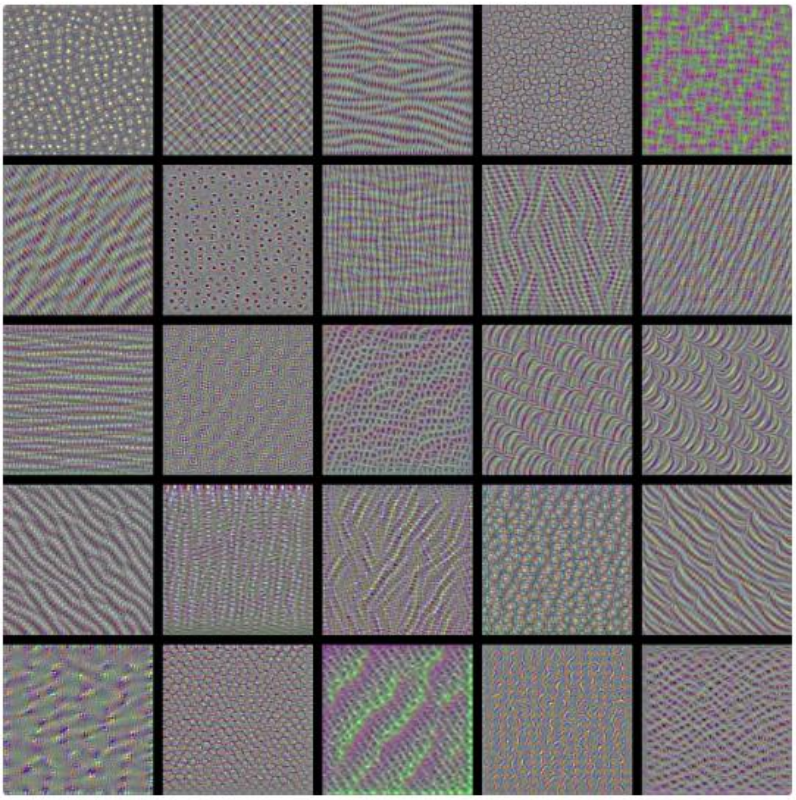
1st Conv Layer From The 1st Conv Block

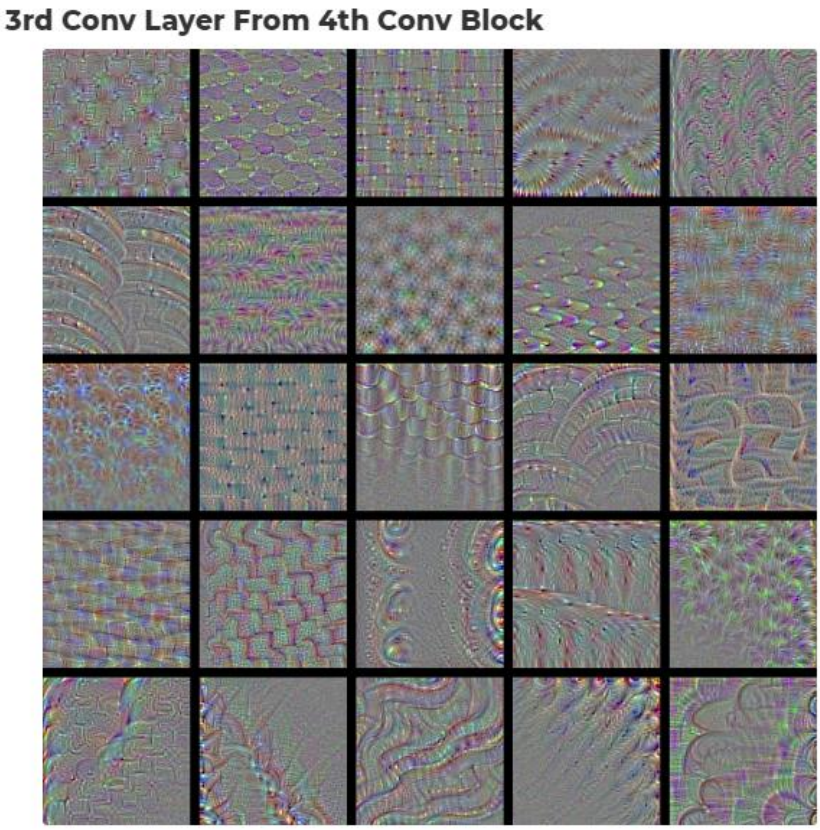


2nd Conv Layer From The 2nd Conv Block

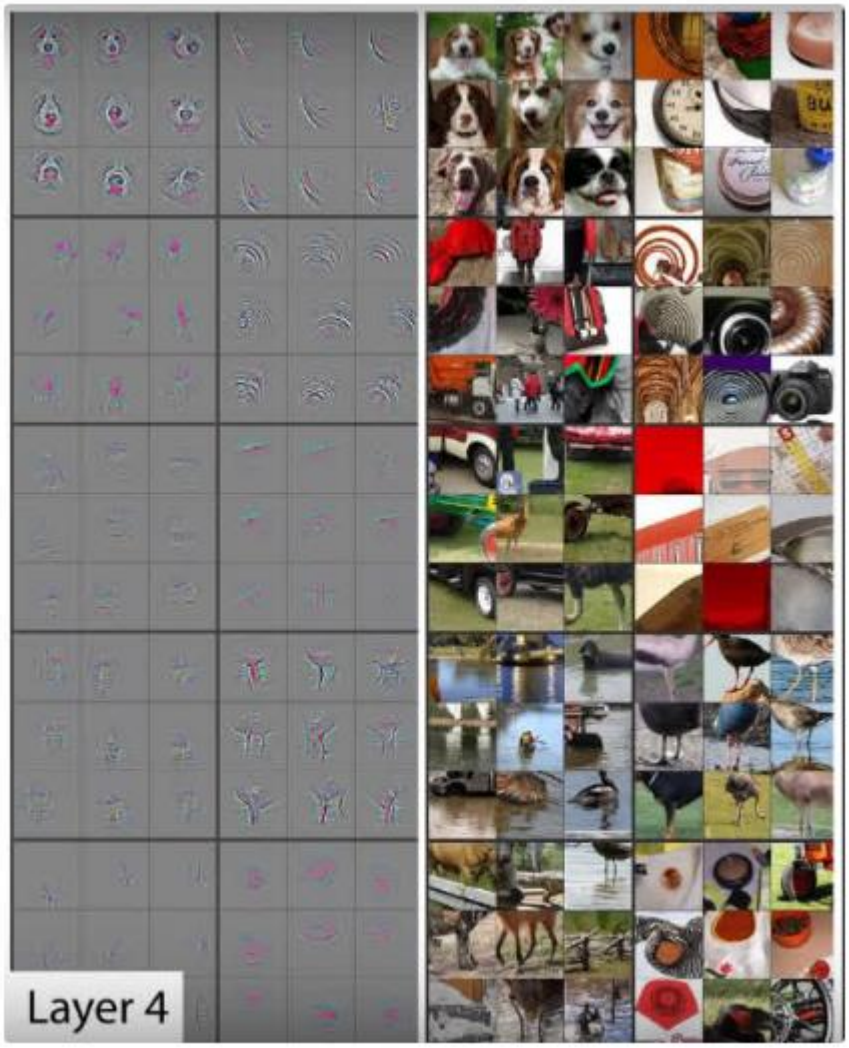


2nd Conv Layer From The 3rd Conv Block



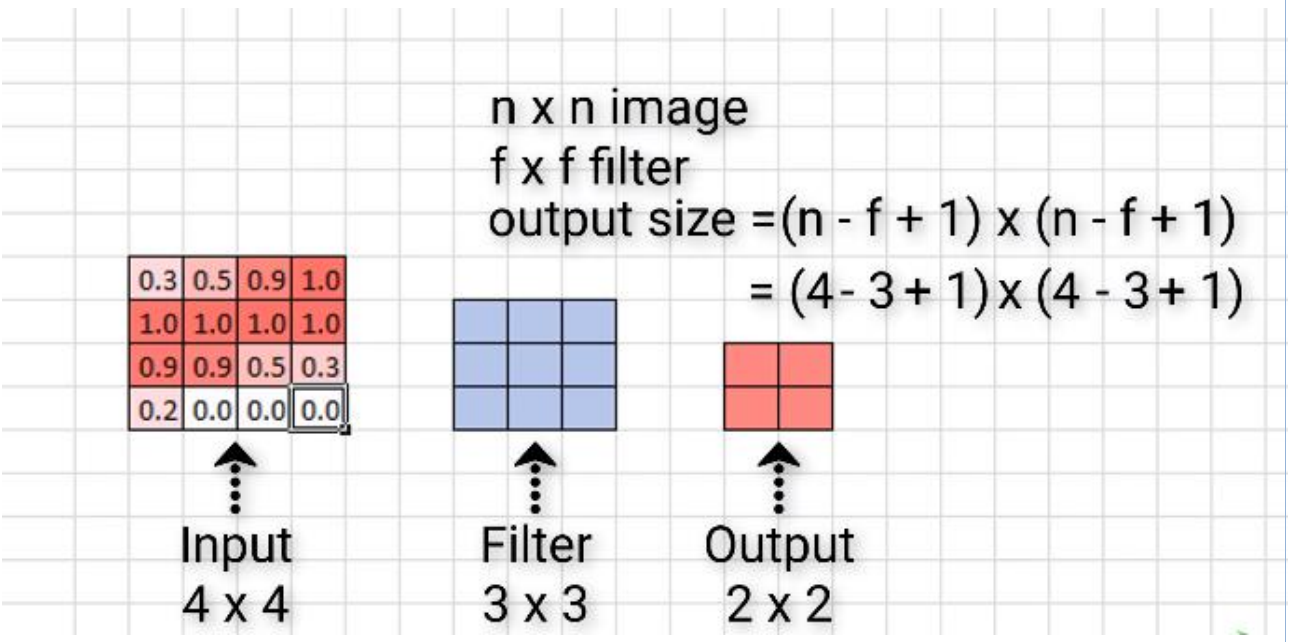


- Notice how with each deeper convolutional layer, we're getting more complex and more interesting visualizations.


Layer 4

Zero Padding in Convolutional Neural Networks Explained

- Image dimensions are reduced.
- Our original size of metrics will be shrined when we do convolutional



Here if our input size in 4*4, let's consider input by n

And filter size is 3*3, let's lay filter as f

Then output will be = (n-f+1) * (n-f+)

So, (4-3+1)*(4-3+1)

2*2

So our output is reduced to 2*2

Output is smaller than input

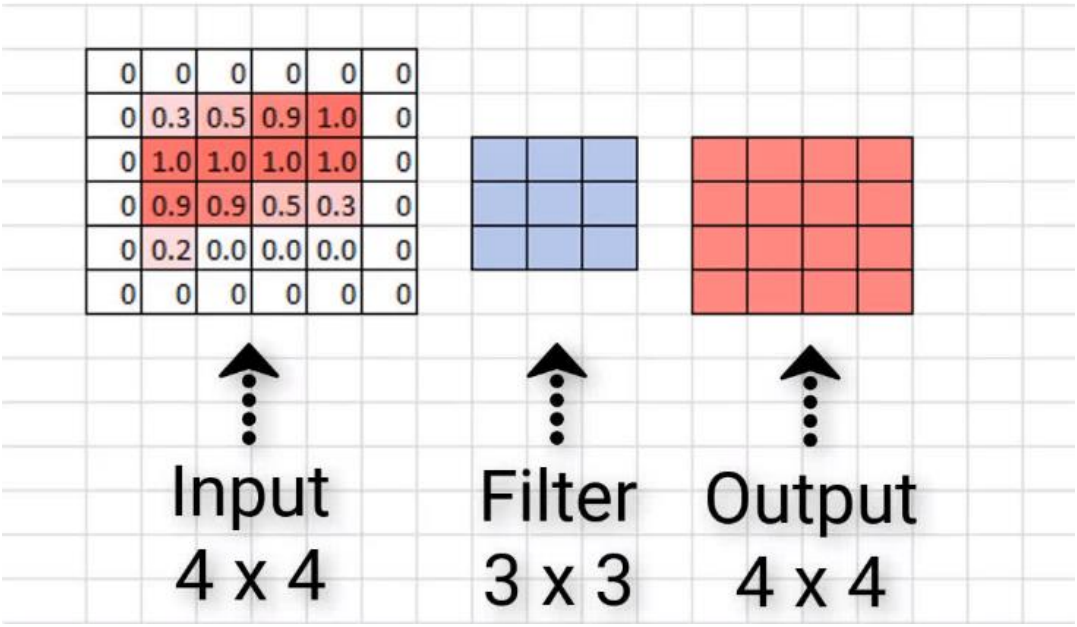
This is just in one convolutional, so what happened when we have so many convolutional in our neural network?

It could be very small with no meaning, so to overcome from this we use zero padding

- Zero padding keep the original size of image (which means size of input image is equal to size of output image)
- *Zero padding* is a technique that allows us to preserve the original input size

- This is something that we specify on a per-convolutional layer basis. With each convolutional layer, just as we define how many filters to have and the size of the filters, we can also specify whether to use padding.

- Zero padding occurs when we add a border of pixels all with value zero around the edges of the input images.



Types of zero padding

- Valid = add no padding
- Same = add zero padding

Padding Type	Description	Impact
Valid	No padding	Dimensions reduce
Same	Zeros around the edges	Dimensions stay the same

Zero padding in code

We'll start with some imports:

```
import keras
from keras.models import Sequential
from keras.layers import Activation
from keras.layers.core import Dense, Flatten
from keras.layers.convolutional import *
```

Now, we'll create a completely arbitrary CNN.

```
model_valid = Sequential([
    Dense(16, input_shape=(20,20,3), activation='relu'),
    Conv2D(32, kernel_size=(3,3), activation='relu', padding='valid'),
    Conv2D(64, kernel_size=(5,5), activation='relu', padding='valid'),
    Conv2D(128, kernel_size=(7,7), activation='relu', padding='valid'),
    Flatten(),
    Dense(2, activation='softmax')
])
```

- It has 3 dense layers
- Then 3 convolutional layers fallowed by dense output layer
- We have specified that input size of images that are coming to CNN is 20*20
- Our first convolutional layer has filter size of 3*3, which is specified in kernel_size. Then second conv layer has 5*5 filter size, then third 7*7.
- With model we are specifying padding as valid (which means no padding). Which is a default for conv layers in Keras. Since we are using valid padding, we are expecting a dimension of output from each layer to decrease
- Let’s check

```
> model_valid.summary()

_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_2 (Dense)              (None, 20, 20, 16)        64
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 18, 18, 32)        4640
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 14, 14, 64)        51264
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 8, 8, 128)         401536
_________________________________________________________________
flatten_1 (Flatten)          (None, 8192)              0
_________________________________________________________________
dense_3 (Dense)              (None, 2)                 16386
=================================================================
Total params: 473,890
Trainable params: 473,890
Non-trainable params: 0
_________________________________________________________________
```

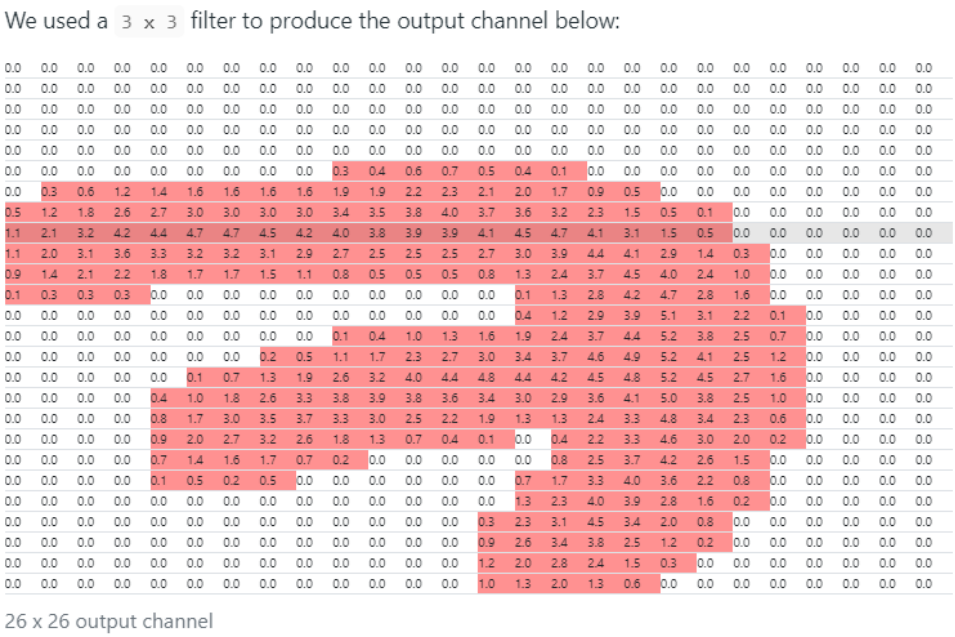
- The first two integer specify the dimension of the output in height and width
- We can see the output shape in each layer.
- We start size with 20*20
- Once we get the output of second, we get 18*18, in third it decreases to 14*14, and finally in last conv layers it decreases to 8*8

- Let's see in padding same

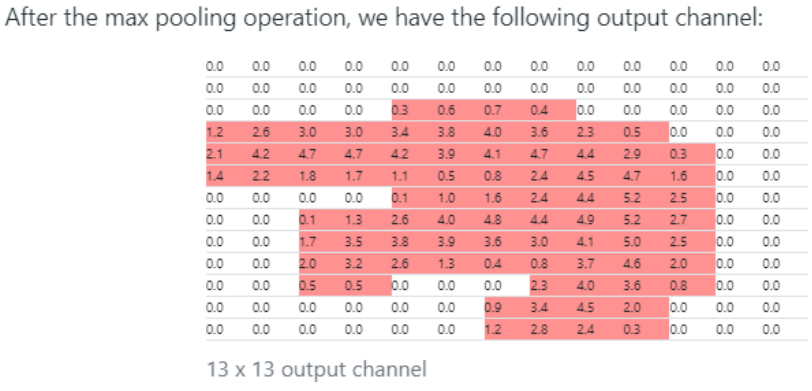
```
> model_same.summary()

_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_6 (Dense)              (None, 20, 20, 16)        64
_________________________________________________________________
conv2d_7 (Conv2D)            (None, 20, 20, 32)        4640
_________________________________________________________________
conv2d_8 (Conv2D)            (None, 20, 20, 64)        51264
_________________________________________________________________
conv2d_9 (Conv2D)            (None, 20, 20, 128)       401536
_________________________________________________________________
flatten_3 (Flatten)          (None, 51200)             0
_________________________________________________________________
dense_7 (Dense)              (None, 2)                 102402
=================================================================
Total params: 559,906
Trainable params: 559,906
Non-trainable params: 0
_________________________________________________________________
```

Max Pooling in Convolutional Neural Network

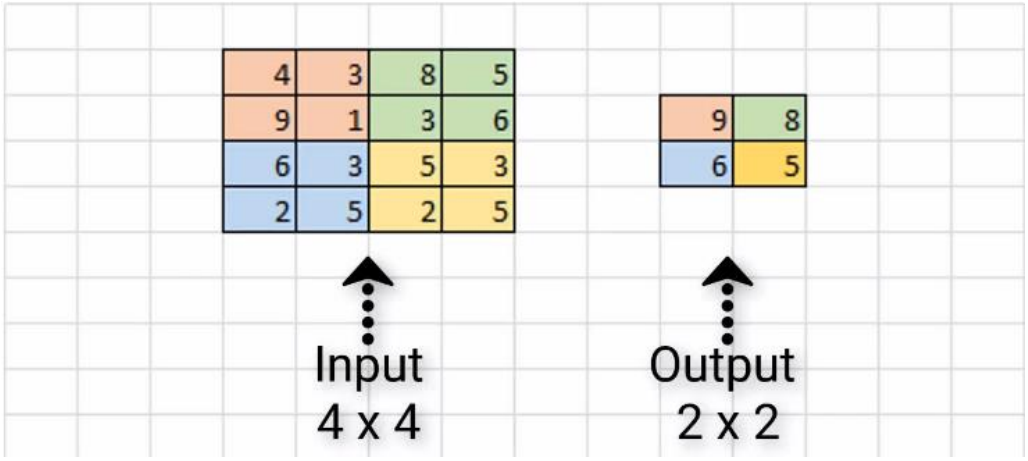
- Max pooling is a type of operation that is typically added to CNN following individual conv layer
- Max pooling reduce the dimensionality of images by reducing the number of pixels in the output from the previous layer.
- Input, Example 1:-



- Output



- Stride determines how many units the filter slides
- Example 2:



Here we have 4*4 input

We are using 2*2 filter with stride 2

In first 2*2 metrics max number is 9, so we are pooling 9 from that metrics,

Then we have 2 strides, so we move to second (green) metric, where max value is 8 so we are pooling 8 from that

From third we have 6 and from forth we have 5

So, we get 2*2 output

- Why use max pooling

  To reduce computational load

  Reduce overfitting

- Just like max pooling we have average pooling too which take the average

Max pooling in code

- Example: -

```
import keras
from keras.models import Sequential
from keras.layers import Activation
from keras.layers.core import Dense, Flatten
from keras.layers.convolutional import *
from keras.layers.pooling import *
```

Here, we have a completely arbitrary CNN.

```
model_valid = Sequential([
    Dense(16, input_shape=(20,20,3), activation='relu'),
    Conv2D(32, kernel_size=(3,3), activation='relu', padding='same'),
    MaxPooling2D(pool_size=(2, 2), strides=2, padding='valid'),
    Conv2D(64, kernel_size=(5,5), activation='relu', padding='same'),
    Flatten(),
    Dense(2, activation='softmax')
])
```

- It has input of 20*20*3 dimension
- At first dense layer, then it is followed by convolutional layer and max pooling, then convolutional layer, which is followed by output layer.
- Since our convolutional layer is 2D we are using the `MaxPooling2D` layer.
- **Output**

```
> model_valid.summary()

_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_2 (Dense)              (None, 20, 20, 16)        64
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 20, 20, 32)        4640
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 10, 10, 32)        0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 10, 10, 64)        51264
_________________________________________________________________
flatten_1 (Flatten)          (None, 6400)              0
_________________________________________________________________
dense_2 (Dense)              (None, 2)                 12802
=================================================================
Total params: 68,770
Trainable params: 68,770
Non-trainable params: 0
_________________________________________________________________
```

- In the above we can see max pooling have cut dimension in half, this is because a filter size of 2*2 along with Stride 2.
- Lastly max pooling layer is followed by one last convolutional layer that is using the padding same, se we can see that output shape for this layer maintains the 10*10 dimensions form the

above max pooling layer.

Quiz

1. Stride refers to ***how many units of data the filter slides between each operation***

Backpropagation in Neural Network

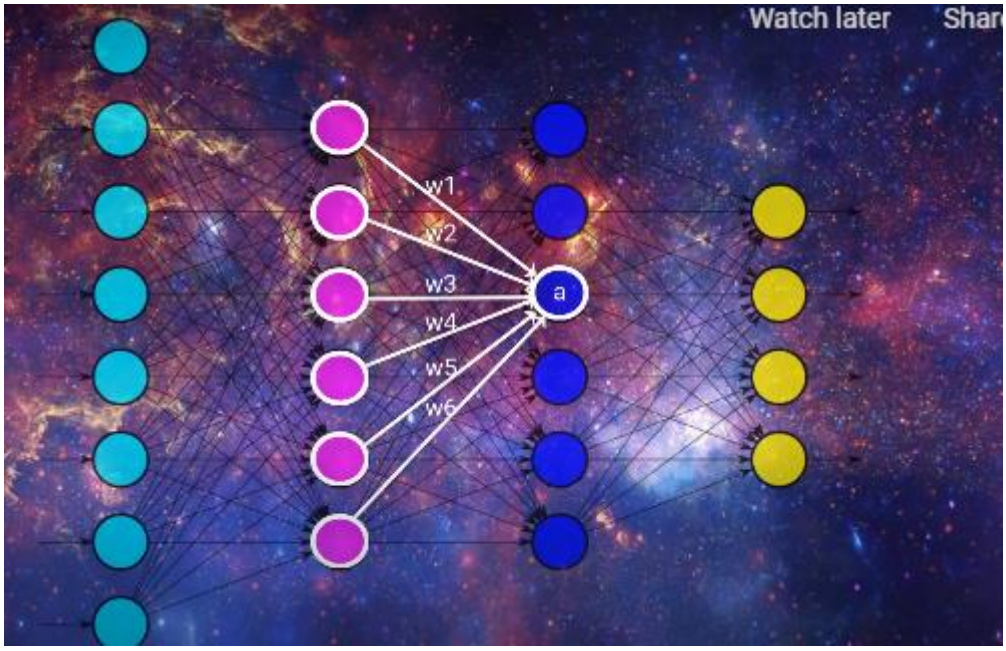
- Back propagation is about calculus
- During the training SGD (sophisticated gradient descent) is used to minimize the loss function by updating the weights with each epoch
- Act of calculating gradient in order to update weights occur through called backpropagation.

- **Quick Reminder**

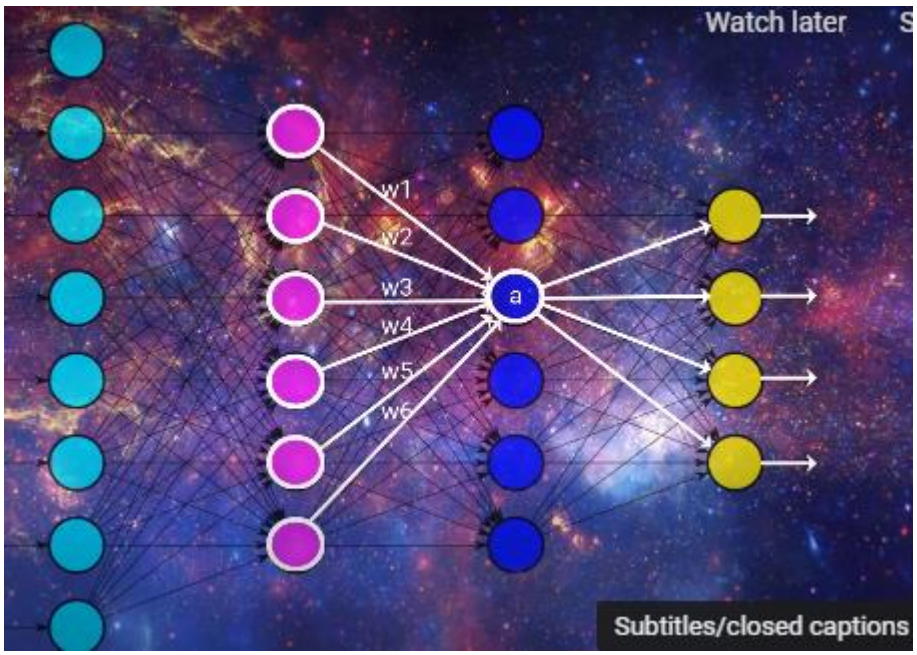
  Whenever we pass the data to the model, data is propagated forward through the network until it reaches output

  We also know that each node receives input from previous layer, and that input is sum of the weights of each of these connections multiplied by previous layers output.

  We then pass that sum to activation function,



  Then result from activation function is the output for a particular node and is then passed as part of input for the nodes in next layer.



And this happened for each layer in the network until we reach the output layer, and this process is referred to as forward propagation.

Once we reach the output layer, we obtain the resulting output from the model for given input.

Each of these output nodes will correspond to different type of animal (let's say we are classifying the animal here). Output with highest activation will be the output that model think best match for corresponding input.

From the obtain result loss is calculated for the predicted output

Loss function is done by taking derivative

d(loss)/d(weight)

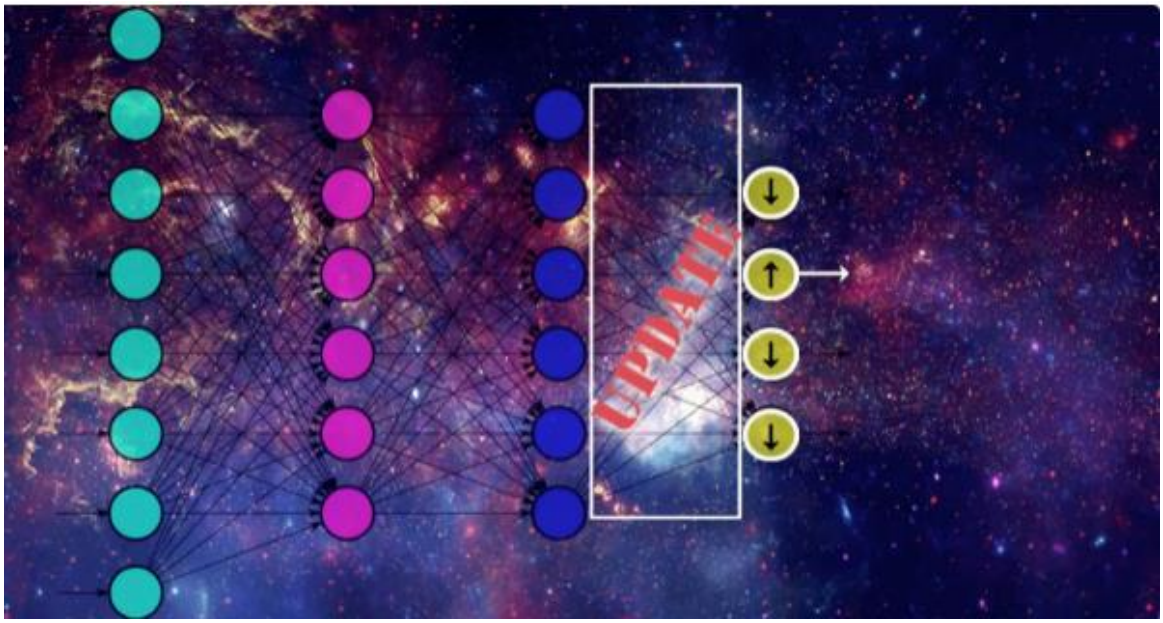
Backpropagations is the tool that gradient descent use to calculate gradient of the loss function. And gradient function updates the weight of input using backpropagation in order to minimize the loss.

Backpropagation intuition

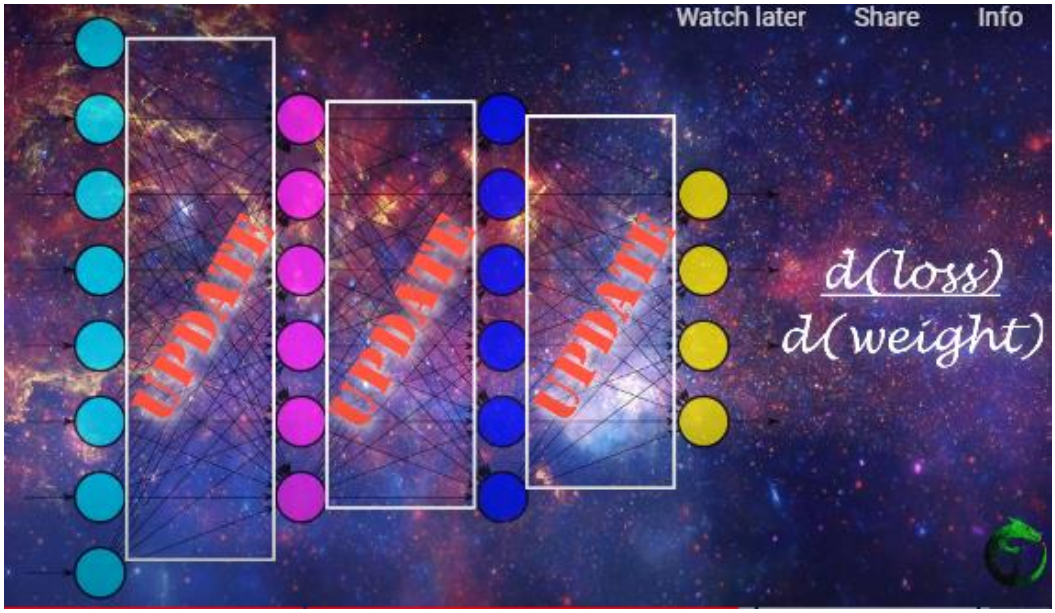
- To update the weights, gradient descent is going to start by looking at the activation outputs from our output nodes.

  Let's say this output (with arrow up) maps the output for given input, if that's the case then gradient descent understands value of this output should increase, and value from all other

output nodes should decrease, by doing this SGD lower the loss for this input.



- We know that the value of the output nods comes from the weighted sum of the weights for the connection in the output layer * output from the previous layer then pass to the activation function.



Therefore, if we want to update the values for the output nodes in a way we discussed, one way is **by updating the weight for these connections that are connected to the output layer**. Another way is **by changing the activation output from the previous layer**.

But we can't directly change the activation output because it is calculation bases on the weigh

and previous layers output. But we can indirectly influence a change in layer's activation output by jumping backwards until we reach at last and update the weight herein the same way we discussed.

Note:- Gradient descent use backpropagation

Backpropagation uses **chain rule** of derivative to calculate gradient descent

Vanishing & Exploding Gradient | A Problem Resulting from Backpropagation

- Vanishing gradient problem causes major difficulty when training a neural network. More specifically, this is a problem that involves weights in earlier layers of the network.
- During training stochastic gradient descent (SGD) works to calculate the gradient of the loss with respect to weights in the neural network.

  Now sometimes the gradient with respect to weights in the earlier layers of the network becomes small like vanishingly small, hence vanishing gradient

Ok what's the big deal with small gradient?

- Once SGD calculate the gradient with respect to weight, it uses this value update the weight. If the gradient is vanishingly small, then update is, in turn, going to be vanishing small as well. Therefore, if this newly update valuate weight has just barely moved from its original then its not going to doing much for the network. This change is not going to help to reduce the loss. As a result, this weight becomes kind of stuck, never really updating enough to even optimal value.

How exactly this problem occurs?

- We know that the gradient of loss with respect to any given weight is going to be the product of some of derivatives that depends on components that reside later in the network.

  The key here is that what if the term in this product, or at least some of the are small (less than one). Well, the product of bunch of numbers less than one is also going to give us even mall number

- After this smaller number is obtained, we subtract the number from the weight and result of this difference is going to be the value of updated weight.

  Summary

  We know gradient vanishing occur with weights in the network due to product of at least some, relatively small values.

Exploding gradient

- Gradients explode is just opposite of gradient vanishing
- We know gradient vanishing occur with weights in the network due to product of at least some, relatively small values.
- Gradient explode occur due to product of at leas some large values (say greater than 1)

  Well, if we multiply bunch of terms together that are greater than one, we are going to get something greater than one, and perhaps even lot greater than one. Thus, essentially exploding in size
- This time instead of barely moving our weight, we are going to greatly move it, so optimal value won't be achieved because of proportion to which the weight becomes updated with each epoch is just too large and continues to move further and further away from its optimal value.

Quiz

1. Vanishing gradient is most concerned with **earlier weights** and is caused by multiplying values against each other that are **less than one** when calculating the **gradient**.

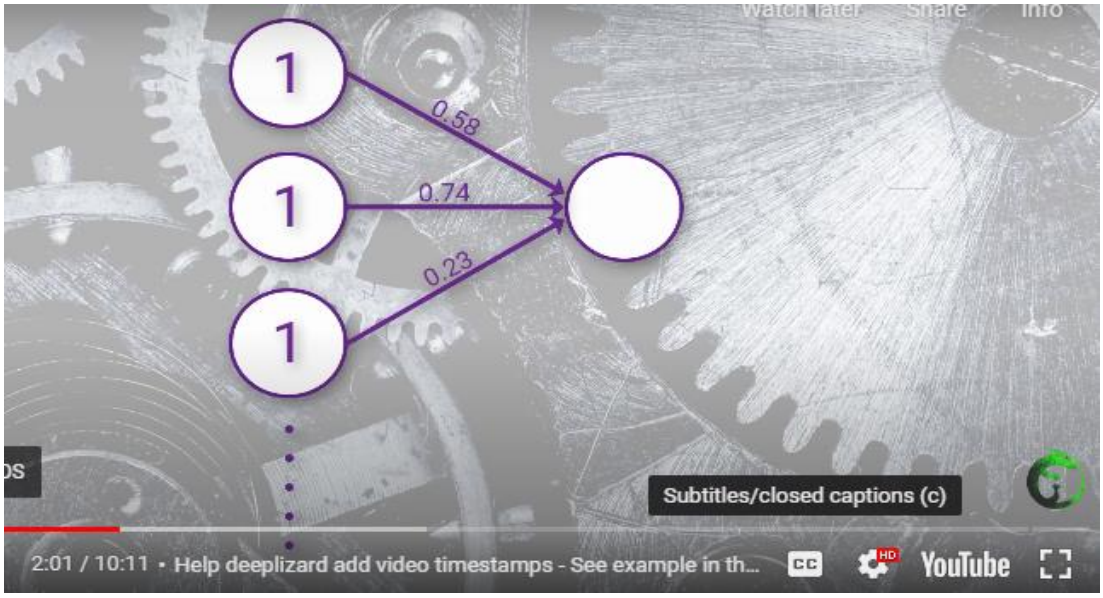
Weight initializations | A way to reduce the vanishing gradient problem

- Weights are randomly initialized
- Whenever we build or compile network values for the weights will be set with random number, one random number per weight.

  These random number will be normally distributed with mean of 0 and standard deviation of 1.

- **Example**

  Let's say our neural network have 250 nodes, for simplicity the value of each of these 250 nodes is 1, now let's focus only on the weight that connect the input layer to a single node in first hidden layer. In total there will 250 weights connecting the nodes in our first hidden layer to all the nodes in the input layer, now each of these weights were randomly generated and normally distributed with a mean of 0 and standard deviation of 1.



So here weighted sum of **z** that this node accepts as input note that in our case all the input nodes have a value of 1, so each weight and **z** will be multiplied by a 1 so **z** become just the sum of weight.

..

…

…

…

Problems With Random Initialization

If the desired output from our activation function is on the opposite side from where it saturated, then during training, when SGD updates the weights in attempts to influence the activation output, it will only make very small changes in the value of this activation output, barely even incrementally moving it in the right direction.

Thus, the network's ability to learn becomes hindered, and training is stuck running in this slow and inefficient state.

These problems that we've discussed so far with weight initialization also contribute to the vanishing and exploding gradient problem.

Given this random weight initialization causes issues and instabilities with training, can we do anything to help ourselves out here? Can we change how weights are initialized?

Xavier initialization

- Since the variance of the input for a given node is determined by variance of the weights connected to this node from the previous layer, we need to shrink the variance of these weights, which will shrink variance of the weighted sum.

Also, note that, given how we defined n as being the number of weights connected to a given node from the previous layer, we can see that this weight initialization process occurs on a per-layer basis.

Another thing also worth noting that when this Xavier initialization was originally announced, it was suggested to use $2/n_{in} + n_{out}$ as the variance where n_{in} is defined as the number of weights coming into this neuron, and n_{out} is the number of weights coming out of this neuron. You may still still see this value referenced in some places.

Now, we've talked a lot about Xavier initialization. Aside from this one, there are other initialization techniques that you can explore, but this Xavier is currently one of the most popular and has an aim to reduce the vanishing and exploding gradient problem.

Some researchers identified a value for the variance of the weights that seems to work pretty well to mitigate the earlier problems we discussed. The value for the variance of the weights connected to a given node is $1/n$, where n is the number of weights connected to this node from the previous layer.

So, rather than the distribution of these weights be centered around 0 with a variance of 1, which is what we had earlier, they are now still centered around 0, but with a significantly smaller variance, $1/n$.

It turns out that, to get these weights to have this variance of $1/n$, what we do is, after randomly generating the weights centered around 0 with variance 1, we multiply each of them by $\sqrt{1/n}$. Doing this causes the variance of these weights to shift from 1 to $1/n$. This type of initialization is referred to as *Xavier initialization* and also *Glorot initialization*.

It's important to note that actually, if we're using relu as our activation function, which is highly likely, then this ideal value for the variance is $2/n$ rather than $1/n$. Besides that, everything else stated so far for this solution is the same. This value just happens to be what works better for relu.

Also, note that, given how we defined n as being the number of weights connected to a given node from the previous layer, we can see that this weight initialization process occurs on a per-layer basis.

Another thing also worth noting that when this Xavier initialization was originally announced, it was suggested to use $2/n_{in} + n_{out}$ as the variance where n_{in} is defined as the number of weights coming into this neuron, and n_{out} is the number of weights coming out of this neuron. You may still still see this value referenced in some places.

Now, we've talked a lot about Xavier initialization. Aside from this one, there are other initialization techniques that you can explore, but this Xavier is currently one of the most popular and has an aim to reduce the vanishing and exploding gradient problem.

Weight initialization in Keras

- Example

```
from keras.models import Sequential
from keras.layers import Dense, Activation

model = Sequential([
    Dense(16, input_shape=(1,5), activation='relu'),
    Dense(32, activation='relu', kernel_initializer='glorot_uniform'),
    Dense(2, activation='softmax')
])
```

- **kernel_initializer = 'glorot_uniform'** in second hidden layer. Here we set **glorot_uniform**, which is the Xavier initialization using uniform distribution. We can also use **glorot_normal** for Xavier initialization.
- If you specify nothing at all, be default Keras initializes the weights in each layer with the **glorot_uniform** initialization. And this is true for other layer types as well not just Dense. Convolutional layers, for example also use the **glorot_uniform** initializer initializer by default as well.

Quiz

How does Keras prevent the variance of the weights from becoming relatively too large or small?

- By using **glorot_uniform** weight initialization by default.

Bias in an artificial neural network

- Each neuron has a bias
- The value assigned to biases are learnable
- During training SGD learn and update weights via backpropagation via during training its also learning and updating biases.
- Bias determined if a neuron is activated or not.
- Biases increased the flexibility of the model to fit the given data

Where Bias fits in

- We know each neuron receives weighted sum of input from the previous layer, and then that weighted sum gets passed to an activation function. Well bias for neuron I going to fit right here within this process. Here what we do is rather than pas the weighted sum directly to the activation function, we instead pas the weighted sum plus the bias term to the activation function

Example of bias

- Let's say we have neural network that has an input layer of just two nodes. Suppose the first node has a value of **1** and second has value of **2**.

  Here we are going to focus on a single neuron within the first hidden layer that directly follows the input layer.

  The activation function will use by first hidden layer is relu. And we are going to assign the some randomly generated weight to our connections.

  Let's see that would be the output for this node without using any bias

  (1 * -0.55) + (2 * 0.10) = -0.35

  Here **-0.55** and **0.10** are randomly generated weight

  Now we pass this value to relu. We know that the value of relu at given input will be maximum of either **0** or the input itself. In our case we have

  Relu (-0.35) = 0

  With an activation output of **0**, the neuron is considered not to be activated, or not firing. In

fact, with relu any neuron with weighted sum of input is less than or equal to **0** will not be firing, so no information from these non-activated neurons will be passed forward through to the rest of the network.

Here **0** is threshold for weighted sum in determining whether a neuron is firing or not.

Well, what if we want to shift our threshold? what if instead of zero, we determined that neuron should fire if its input is greater than or equal to **-1**?

This is where bias comes into play

Bias gets added to the weighted sum before pass into the activation function***. The value we assign to our bias is just opposite of his called threshold value*** (for example if we want threshold of **-1,** we use 1, if we want threshold **5** then we use **-5**)

Continuing with our example we want the threshold to move from **0** to **-1**, then bias would be opposite of **-1**, which is **1**.(1 * -09.55) + (2 * 0.10) + 1 = -0.35 + 1 = 0.65

Relu (0.65) = 0.65

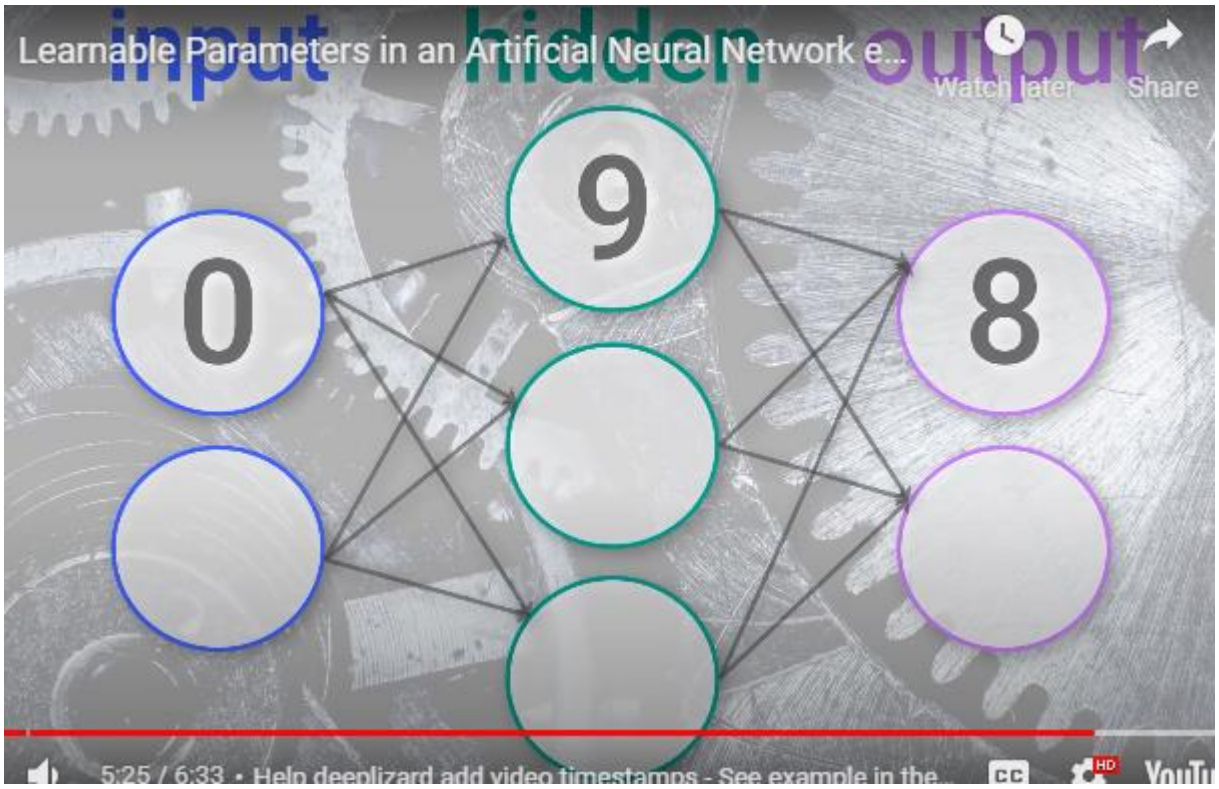
The neuron is considered firing now

The model now has a bit of increased flexibility in fitting the data since it now has a broader range in what values it considers as being activated or not.

Learnable parameters in an artificial neural network

- During the training process, we have discussed how stochastic gradient descent (SGD) works to learn and optimize the weights and biases in neural network. These ***weights*** are *biases* are indeed learnable parameters.
- Any parameter within our model which are learned during training via SGD are considered as learnable parameters. They are also known by ***trainable parameters***, since they are optimized during training process.

Example



- **First**

  First two blues are input nodes here, and three green nodes are output for first input layer

  (2 * 3) +3

  = **9**

  2 = first input

  3 = output for the first hidden layer

  3 = last three is number of bias (since output is three it would have 3 biases here)

From this we get **9** learnable parameters

- **Second**

  In second we have three input (green color) and two output (pink is output here)

  So,

  (3 * 2) + 2

  = **8**

  3 = input

  2 = (middle one) output

  2 = (last one) is the number of bias (because it has **2** output)

  In total we have **9 + 8 = 17** learnable parameters here

Quiz

The number of learnable parameters for a densely connected layer in a neural network can be calculated using following formulas

- Inputs * outputs + biases

Learnable parameters in an Convolutional neural network

- Learnable parameter is same parameter that we saw in standard fully connected network. That is weight and biases.
- **How learnable parameter is calculated in neural network?**

  So just like in standard network, with CNN we will calculate the number of parameters per layer then we will sum up the parameter in each layer to get the total learnable parameter in the entire network.

```
// pseudocode
let sum = 0;
network.layers.forEach(function(layer) {
    sum += layer.getLearnableParameters().length;
}
```

- **Let's see what convolutional layer has that a dense layer doesn't have?**

  Convolutional layer has filters, also known as kernels, we need to determine how many filters are in a convolutional layer as well as how large these filters are, and we need to consider these things in our calculations.

  With this in mind, we will modify our formula for determining the number of learnable parameters in convolutional layer.

- **So, what is the input to be for a given convolutional layer? Well, that is going to determine by what type of previous layer was?**
  - If the previous layer was a dense layer, the input to the convolutional layer is just the number of nodes in the previous dense layer
  - If the previous layer was convolutional layer, the input will be number of filters from that previous convolutional layer.
- The number of biases, well that will just be equal to the number of filters in the layer.
- So overall, we have the same general setup for the number of learnable parameters in the layer being calculated as the number of **inputs times the number of outputs plus the**

number of biases

(input * output) + biases

- ## Calculate the number of learnable parameters in CNN

  Suppose we have a CNN made up of input layer, two hidden convolutional layers, and a dense output layer

  - input layer
  - hidden convolutional layer
  - hidden convolutional layer
  - dense layer

- Our input layer is made up of input data from image of size 20*20*3

  Where 20*20 specifies the width and height of the images and 3 specifies the number of channels. The three channels indicate that our image is in RGB color scale, and these three channels will represent the input features in this layer.

  First convolutional layer is made up of 2 filter size of 3*3

  Second convolutional layer is made up of 3 filters of size 3*3

  Output layer is dense layer with 2 nodes

  We will assume that network contains bias, and we are using zero padding throughout the network to maintain the dimension of images.

  - input layer - images of size 20*20*3
  - hidden convolutional layer - 2 filters of size 3*3
  - hidden convolutional layer - 3 filters of size 3*3
  - dense output layer - 2 nodes

- **Input layer**

  The input layer has no learnable parameter since it just contains the input data.

- **Convolutional layer 1**

  We have 3 input layers; we know that the number of outputs is **the number of filters times the filter size**. So, we have two filter each of size **3*3** so **2*33** = **18**.

Multiplying our three inputs by **18** outputs, we have **54** weights.

- **Convolutional layer 2**

  In this layer we will have three filter again with size of **3*3**, so that **3*3*3 = 27** outputs. Multiplying **2** outputs by **27** we get **54** weights in this layer. Adding **3** bias terms from three filters we have **57** learnable parameters in this layer.

- **Output layer**

  Into output layer, how many inputs? We may think just three, right, since that's the number of filters in the last convolutional layer. But that's not quite right. If you have fallowed the Keras series, you know that before passing output from a convolutional layer to dense layer, that we have to flatten the output by multiplying the dimension of the data from the convolutional layer by the number of filters int that layer. In our case the is image data.

  Since we are assuming that this network use zero padding, the dimension of our image of size 20*20 haven't changed by the time we ger to this layer. So, multiplying 20*20 by the three filters give us total of 1200 inputs coming into our output layer.

  Since this layer is dense layer, the number of outputs is equal to number of nodes in this layer, so we have two outputs, multiplying 1200*2 give us 2400 weights. Adding two biases from this layer we have 2402 learnable parameters n this layer

- **Result**

  Summing up the parameters from all the layers gives us a total of 2515 learnable parameters within the entire network

Quiz

To calculate the number of learnable parameters in a convolutional layer, we multiply the layer input by the output and add the bias

Regularization in neural network

- In general regularization help to reduce overfitting and variance in our network by penalizing (correcting or fining) for complexity.
- To implement regularization, we simply add the term tour loss function that penalize for large weights.

L2 regularization

- The most common regularization is **L2 regularization**

L2 regularization term

- In L2 regularization term, what we add to the loss function is the sum of the squared norms of the weight matrices multiplied by small constant.

$$\sum_{j=1}^{n} \left\| w^{[j]} \right\|^2,$$

$$\frac{\lambda}{2m}.$$

Norms are positive

- In general term nom is just a function that assign strictly positive length or size for each

vector in a vector space. The vector space we are working here depends on the size of our weight's matrices.

- Rather than going on a linear algebra tangent about norms in this moment, we will continue with the general idea about regularization.
- Norms are fundamental concept of linear algebra
- To oversimplify, know for now that the norm of each of our weight matrices is just going to be a positive number.
- Suppose that v is a vector in a vector space. The norm of **v** is denoted as **‖v‖,** and it is required that

$$\|v\| \geq 0.$$

Adding The Term To The Loss

Let's look at what L2 regularization looks like. We have

$$loss + \left(\sum_{j=1}^{n} \left\|w^{[j]}\right\|^2\right) \frac{\lambda}{2m}.$$

The table below gives the definition for each variable in the expression above.

Variable	Definition
n	Number of layers
$w^{[j]}$	Weight matrix for the j^{th} layer
m	Number of inputs
λ	Regularization parameter

- The term **λ** is called the regularization parameter, and this is another hyperparameter that we'll have to choose and then test and tune in order to choose the correct number for our specific model.
- **To summarize**, we now know that regularization is just a technique that penalize for relatively large weights in our model, and behind the scenes, the implementation of regularization is just the addition of term to our existing loss functions.

Impact of regularization

- Well, using L2 regularization as an example, if we were to set **λ** to be large, then it would incentivize the model to set the weights close to zero because the objective of **SGD** is to minimize the loss function. Remember our original loss function is now being summed with

the sum of the squared matrix norms, which is multiplied by lambda by $2m$.

If λ is large, then this term, $\lambda/2m$, will continue to stay relatively large, and if we're multiplying that by the sum of the squared norms, then the product may be relatively large depending on how large our weights are. This means that our model is incentivized to make the weights small so that the value of this entire function stays relatively small in order to minimize loss.

Intuitively, we could think that maybe this technique will set the weights so close to zero, that it could basically zero-out or reduce the impact of some of our layers. If that's the case, then it would conceptually simplify our model, making our model less complex, which may in turn reduce variance and overfitting.

- Example of regularization in Keras

Batch size in neural network

- The batch size is the number of samples that are passed to the network at once. It is also known as mini batch.
- Now, recall that an **epoch** is one single pass over the entire training set to the network. The batch size and an epoch are not the same thing.

Batches in epoch

- Example

  Let's say we have 1000 of images of dogs that we want to train our network to identify different dog breed. And let's say our batch size is 10. This means that 10 images of dog will be passed as batch, at one time to the network.

  We know that single epoch one single pass of all the data through network is, it will take 100 batches to make up full epoch. We have 1000 image divided by a batch size of 10, which means 100 total batches.

  batches in epoch = training set size / batch_size

Why uses batches

- Larger the batch size, quicker our model will complete each epoch during training.
- Our model can even handle very large batches, the quality of model may degrade as we set our batch large and may ultimately cause the model to be unable to generalize well on data it hasn't seen before.
- Batch size is hyperparameter that we must test, and tune based on how our specific model is performing during training.

Mini batch gradient decent

- Additionally, note if using *mini-batch gradient descent*, which is normally the type of gradient descent algorithm used by most neural network APIs like Keras by default, the gradient update will occur on a per-batch basis. The size of these batches is determined by the batch size.

Working with batch size in in Keras

```
model = Sequential([
    Dense(units=16, input_shape=(1,), activation='relu'),
    Dense(units=32, activation='relu', kernel_regularizer=regularizers.l2(0.01)),
    Dense(units=2, activation='sigmoid')
])
```

- model.fit() we know this function will train our model

```
model.fit(
    x=scaled_train_samples,
    y=train_labels,
    validation_data=valid_set,
    batch_size=10,
    epochs=20,
    shuffle=True,
    verbose=2
)
```

- we'll be passing in **10** samples at a time until we eventually pass in all the training data to complete one single epoch. Then, we'll start the same process over again to complete the next epoch

Fine-tuning a neural network

- A fine-tuning is very closely linked with transfer learning
- Transfer learning occurs when we use knowledge that was gained from solving one problem and apply it to a new but related problem

  For example, knowledge gained from learning to recognize cars could be applied in a problem of recognizing trucks
- Fine turning is a way of applying or utilizing transfer learning. Especially fine tuning is process that takes a model that has already been trained for one given task and then tunes or tweaks the model to make it perform to a second similar task

Why use fine tuning?

- Using an artificial neural network that has already been designed and trained allow us to take advantage of what the model has already learned without to develop it from scratch.
- When building a model from scratch, we usually must try many approaches through trial-and-error.

  For example, we have to choose how many layers we're using, what types of layers we're using, what order to put the layers in, how many nodes to include in each layer, decide how much regularization to use, what to set our learning rate as, etc. Building and validating our model can be a huge task, depending on what data we're training it on.

  This is what makes the fine-tuning approach so attractive. If we can find a trained model that already does one task well, and that task is similar to ours in at least some remote way, then we can take advantage of everything the model has already learned and apply it to our specific task.

Now, of course, if the two tasks are different, then there will be some information that the model has learned that may not apply to our new task, or there may be new information that the model needs to learn from the data regarding the new task that wasn't learned from the previous task.

For example, a model trained on cars is not going to have ever seen a truck bed, so this feature is something new the model would have to learn about. However, think about everything our model for recognizing trucks could use from the model that was originally trained on cars.

This already trained model has learned to understand edges and shapes and textures and more objectively, head lights, door handles, windshields, tires, etc. All these learned features are things we could benefit from in our new model for classifying trucks.

How to fine tune

- Going back to the example we just mentioned, if we have a model that has already been trained to recognize cars and we want to fine-tune this model to recognize trucks, we can first import our original model that was used on the cars problem.
- For simplicity purposes, let's say we remove the last layer of this model. The last layer would have previously been classifying whether an image was a car or not. After removing this, we want to add a new layer back that's purpose is to classify whether an image is a truck or not.
- In some problems, we may want to remove more than just the last single layer, and we may want to add more than just one layer. This will depend on how similar the task is for each of the models.
- Layers at the end of our model may have learned features that are very specific to the original task, whereas layers at the start of the model usually learn more general features like edges, shapes, and textures.
- After we've modified the structure of the existing model, we then want to freeze the layers in our new model that came from the original model
- **Freezing wight**

  By *freezing*, we mean that we don't want the weights for these layers to update whenever we train the model on our new data for our new task. We want to keep all of these weights the same as they were after being trained on the original task. We only want the weights in our new or modified layers to be updating.

  After we do this, all that's left is just to train the model on our new data. Again, during

this training process, the weights from all the layers we kept from our original model will stay the same, and only the weights in our new layers will be updating.

Batch normalization (Batch form)

- When we train a neural network, we want to normalize or standardize our data in some way ahead of time as a part of the pre-processing step. This is the step where we prepare our data to get it ready for training.
- Normalization and standardization have same objective of transforming the data to put all the data points on the same scale.
- A typical normalization process consists of scaling numerical data down to be on a scale from **0** to **1**, and a typical standardization process consists of subtracting the mean of the dataset from each data point, and then dividing that difference by the data set's standard deviation. This forces the standardized data to take on a mean of zero and a standard deviation of one. In practice, this standardization process is often just referred to as normalization as well.

Why use normalization techniques?

- Well, if we didn't normalize our data in some way, we can imagine that we may have

some numerical data points in our data set that might be very high, and other that might be very low.

For example, suppose we have data on the number of miles individuals have driven a car over the last 5 years. We may have someone who has driven 100,000 miles total, and we may have someone else who's only driven 1000 miles total. This data has a relatively wide range and isn't necessarily on the same scale.

Additionally, each one of the features for each of our samples could vary widely as well. If we have one feature which corresponds to an individual's age and the other feature corresponds to the number of miles that individual has driven a car over the last five years, then, again, we can see that these two pieces of data, age and miles driven, will not be on the same scale.

The larger data points in these non-normalized data sets can cause **instability** in neural networks because the relatively large inputs can cascade (fall or drop) down through the layers in the network, which may cause imbalanced gradients, which may therefore cause the famous **exploding gradient problem**.

- For now, understand that this imbalanced, non-normalized data may cause problems with our network that make it drastically harder to train. Additionally, non-normalized data can significantly decrease our training speed.
- When we normalize our inputs, however, we put all of our data on the same scale, in attempts to increase training speed as well as avoid the problem we just discussed because we won't have this relatively wide range between data points.

Applying Batch normalization to layers

- ***Batch normalization* is applied to layers.**
- When applying batch norm to a layer, the first thing batch norm does is normalize the output from the activation function.
- After normalizing the output from the activation function, batch norm multiplies this

normalized output by some arbitrary parameter and then adds another arbitrary parameter to this resulting product.

s

Step	Expression	Description
1	$z = \frac{x - mean}{std}$	Normalize output x from activation function.
2	$z * g$	Multiply normalized output z by arbitrary parameter g.
3	$(z * g) + b$	Add arbitrary parameter b to resulting product $(z * g)$.

Batch Normalization Process

Trainable parameter

- This calculation with the two arbitrary parameters sets a new standard deviation and mean for the data. The two arbitrarily set parameters, g and b are trainable, meaning that they will be become learned and optimized during the training process.

Parameter	Trainable
g	Yes
b	Yes

Hyperparameters

- This process makes it so that the weights within the network don't become imbalanced with extremely high or low values since the normalization is included in the gradient process.
- This addition of batch norm to our model can greatly increase the speed in which training occurs and reduce the ability of outlying large weights to over-influence the training process.
- When we spoke about normalizing our input data in the pre-processing step before training occurs, we understand that this normalization happens to the data before being passed to the input layer.
- With batch norm, we can normalize the output data from the activation functions for individual layers within our model as well. This means we have normalized data coming in, and we also have normalized data within the model

Normalizing Per Batch

- Everything we just mentioned about the batch normalization process occurs on a per-batch basis, hence the name *batch norm*.
- These batches are determined by the batch size we set when we train our model. If we're not yet familiar with training batches or batch size, check out this post on the topic.

Working with Keras

- Example:-

```
model = Sequential([
    Dense(units=16, input_shape=(1,5), activation='relu'),
    Dense(units=32, activation='relu'),
    BatchNormalization(axis=1),
    Dense(units=2, activation='softmax')
])
```

We have a model with two hidden layers with 16 and 32 nodes respectively, both using relu() as their activation functions, and an output layer with two output categories using the softmax() activation function.
The only difference here is the line between the last hidden layer and the output layer

```
BatchNormalization(axis=1)
```

- This is how we specify batch normalization in Keras
- The only parameter that we're specifying for BatchNormalization is the axis parameter, and that is just to specify the axis from the data that should be normalized, which is typically the features axis.
- There are several other parameters that we can optionally specify, including two called beta_initializer and gamma_initializer. These are the initializers for the arbitrarily set parameters that we mentioned when we were describing how batch norm works.
- These are set by default to 0 and 1 by Keras, but we can optionally change these, along with several other optionally specified parameters.