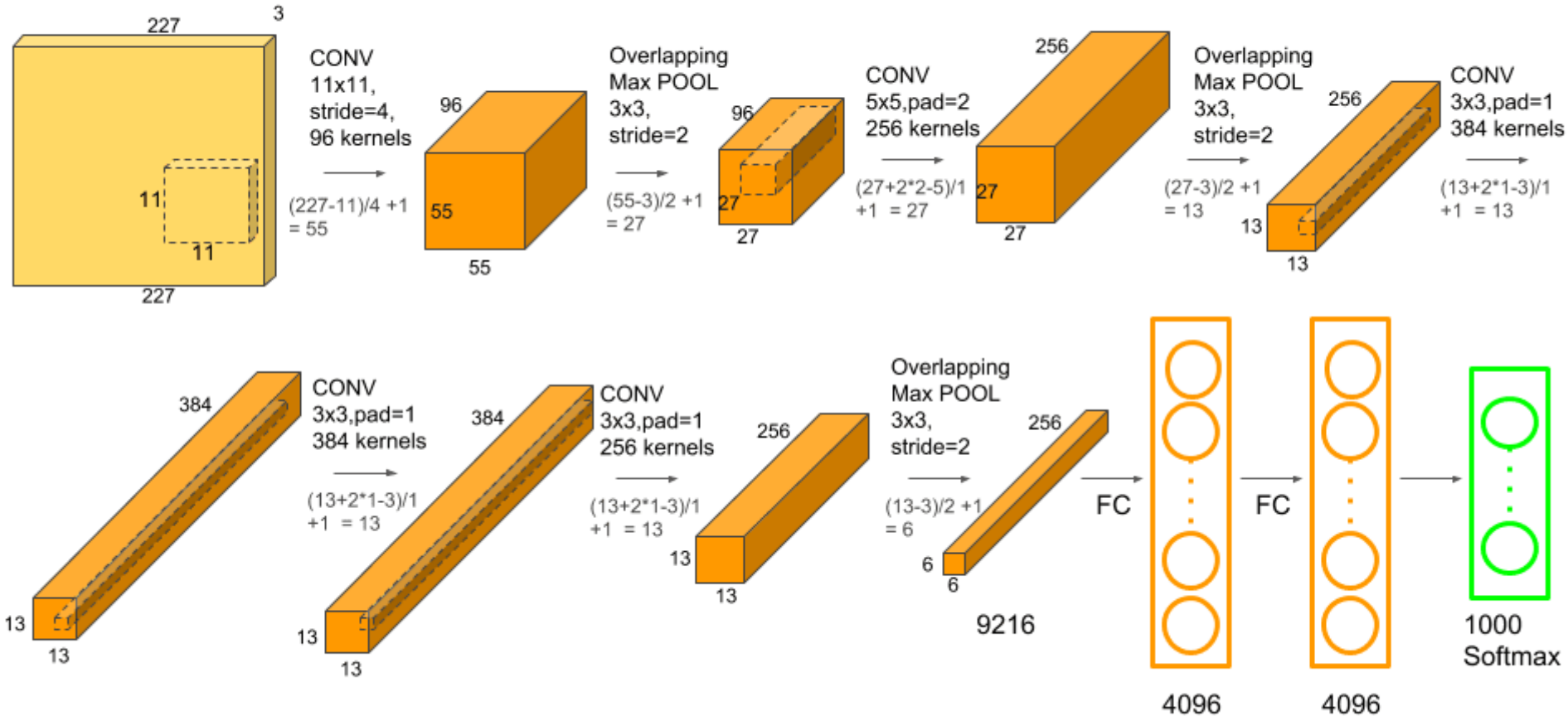
Today we are going to learn about AlexNet and built it completely from Scratch.

Problems ranging from image recognition to image generation and tagging have benefited greatly from various Deep Learning architectural advancements. Understanding the details of different Deep Learning models will help you understand the evoultion of the field, and find the right fit for the problems you're trying to solve. Over a past couple of years many architectures have risen up varying in many aspects, such as the type of layers, hyperparameters, etc. This AlexNet Tutorial is focused on exploring AlexNet which became one of the most popular CNN architectures.

AlexNet



AlexNet is one the most popular neural network upto date. It was proposed by Alex Krizhevsky for the ImageNet Large Scale Visual Recognition Challenge (ILSVRV), and is based on convolutional neural networks. ILSVRV evaluates algorithms for Object Detection and Image Classification. In 2012, Alex Krizhevsky et al. published ImageNet Classification with Deep Convolutional Neural Networks. This is when AlexNet was first heard of.

The challenge was basically to develop a Deep Convolutional Neural Network to classify the 1.2 million high-resolution images in the ImageNet LSVRC-2010 dataset into more than 1000 different categories. The architecture achieved a top-5 error rate (the rate of not finding the true label of a given image among a model's top-5 predictions) of 15.3%. The next best result trailed far behind at 26.2%.

Research paper : https://proceedings.neurips.cc/paper/2012/hash/c399862d3b9d6b76c8436e924a68c45b-Abstract.html

Download the original research paper of AlexNet through the above provided link. And read the research paper before moving ahead.

Reference video : https://youtu.be/Nq3auVtvd9Q

Go through the above mentioned video if you are not able to understand the research paper.

History of AlexNet

Alexnet was designed by Alex Krizhevsky. It was published with Ilya Sutskever and Krizhevsky's doctoral advisor Geoffrey Hinton, and is a Convolutional Neural Network or CNN.

After competing in ImageNet Large Scale Visual Recognition Challenge, AlexNet shot to fame. It achieved a top-5 error of 15.3%. This was 10.8% lower than that of runner up.

The primary result of the original paper was that the depth of the model was absolutely required for its high performance. This was quite expensive computationally but was made feasible due to GPUs or Graphical Processing Units, during training.

CNN Architecture

Before getting into AlexNet it is esssential to understand what is a convolutional neural network.

So Convolutional neural networks are one of the variants of neural networks where hidden layers consist of convolutional layers, pooling layers, fully connected layers, and normalization layers.

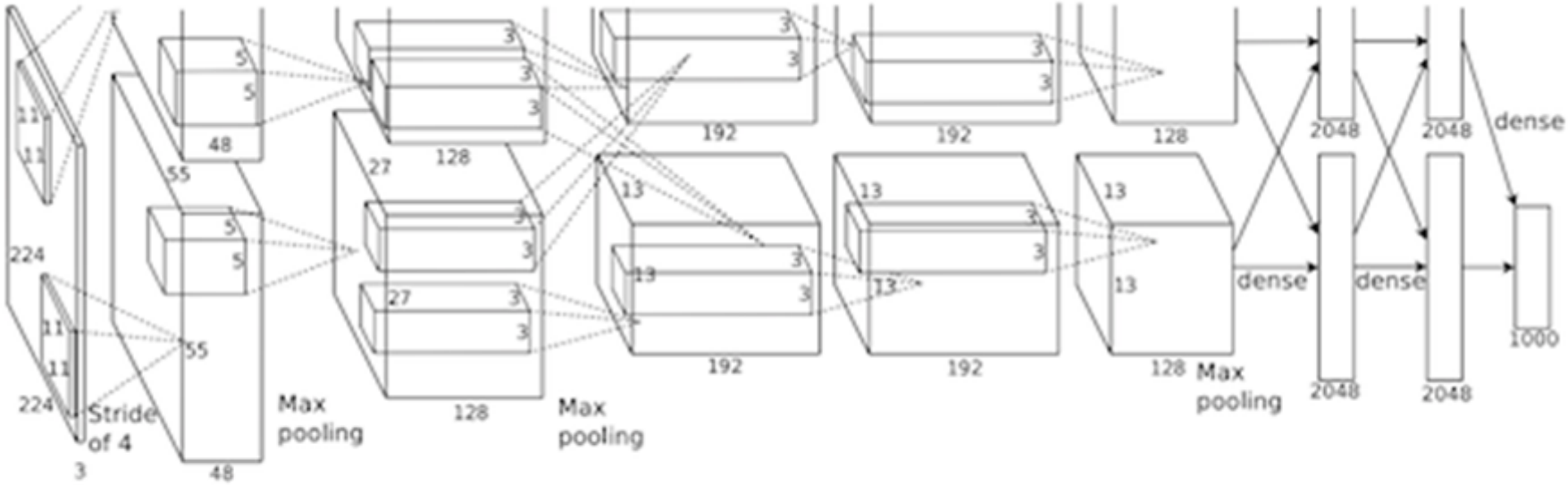
Convolution is basically the process of applying a filter over an image or signal to modify it. Now what is pooling? It is a sample-based discretization process. The main reason is to reduce the dimensionality of the input. Thus, allowing assumptions to be made about the features contained in the sub-regions binned.

A stack of distinct layers that transform input volume into output volume with the help of a differentiable function is known as CNN Architecture.

In other words, one can understand a CNN architecture to be a specific arrangement of the above mentioned layers. Numerous variations of such arrangements have developed over the years resulting in several CNN architectures. The most common amongst them are:

1. LeNet-5 (1998)
2. AlexNet (2012)
3. ZFNet (2013)
4. GoogleNet / Inception(2014)
5. VGGNet (2014)
6. ResNet (2015)

Alexnet Architecture



AlexNet architecture (May look weird because there are two different "streams". This is because the training process was so computationally expensive that they had to split the training onto 2 GPUs)

AlexNet was the first convolution network which used GPU to boost performance

Now, Let's take a deep dive into the architectue of Alexnet

Reference video : https://youtu.be/YgA-v6tJTz8

NOTE : Watch this video till 27:34 only

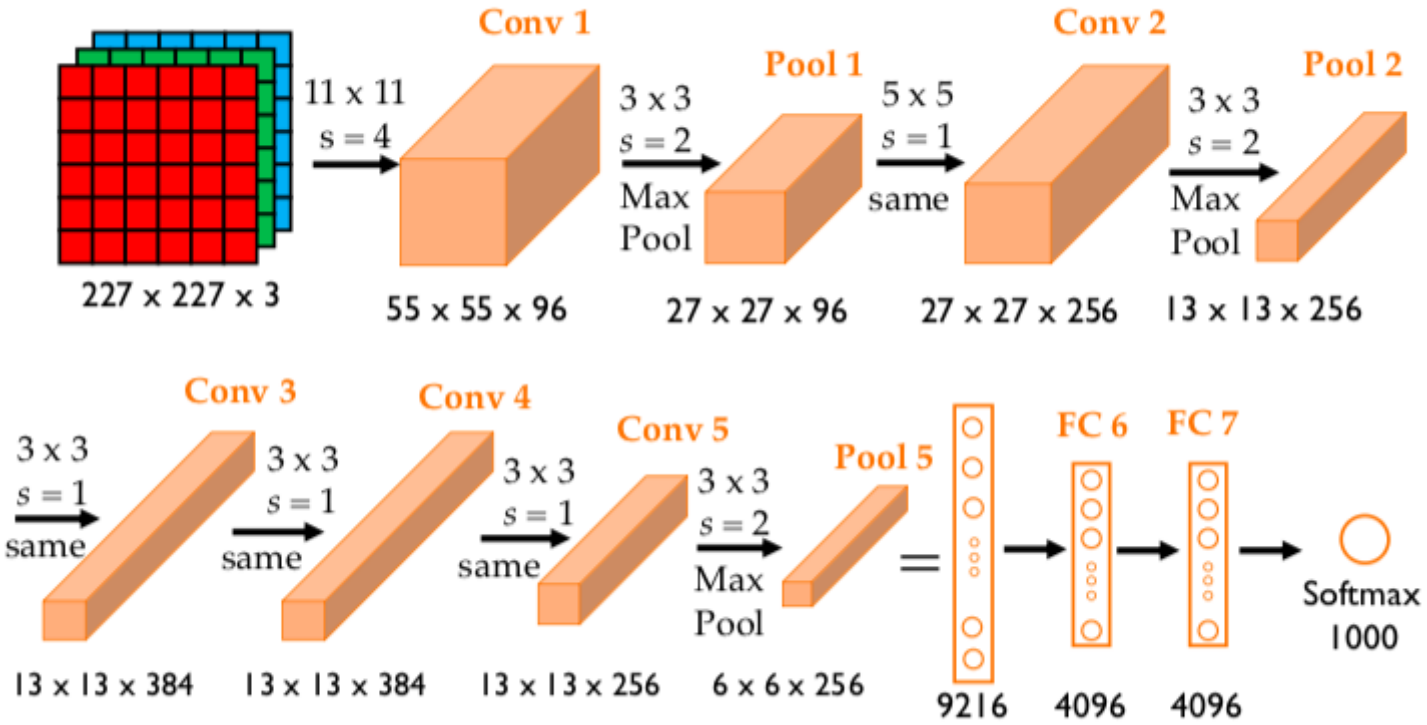
The AlexNet neural network architecture consists of 8 learned layers of which 5 are convolution layers, few are max-pooling layers, 3 are fully connected layers, and the output layer is a 1000 channel softmax layer. The pooling used here is Max pool.

1. AlexNet architecture consists of 5 convolutional layers, 3 max-pooling layers, 2 normalization layers, 2 fully connected layers, and 1 softmax layer.

2. Each convolutional layer consists of convolutional filters and a nonlinear activation function ReLU.
3. The pooling layers are used to perform max pooling.
4. Input size is fixed due to the presence of fully connected layers.
5. The input size is mentioned at most of the places as 224x224x3 but due to some padding which happens it works out to be 227x227x3
6. AlexNet overall has 60 million parameters.

Why 1000 channels of softmax layers are taken??

The AlexNet neural network architecture consists of 8 learned layers of which 5 are convolution layers, few are max-pooling layers, 3 are fully connected layers, and the output layer is a 1000 channel softmax layer. The pooling used here is Max pool.



The input to the AlexNet network is 227 x 227 size RGB image, so it is having 3 different channels i.e., red, green, blue.

Then we have the First Convolution Layer in the AlexNet that has 96 different kernels with each kernel's size of 11 x 11 and with stride equals 4. So the output of the first convolution layer gives you 96 different channels or feature maps because there are 96 different kernels and each feature map contains features of size 55 x 55.

Reference video: https://youtu.be/Y1qxI-Df4L

Calculations:

- Size of Input: N = 227 x 227
- Size of Convolution Kernels: f = 11 x 11
- No. of Kernels: 96
- Strides: S = 4
- Padding: P = 0

Size of each feature map = [(N — f + 2P)/S] + 1

- Size of each feature map = (227−11+0)/4+1 = 55

```
import tensorflow as tf

# Input Layer
inputs = tf.keras.Input(shape=(224, 224, 3), name="alexnet_input")

# Layer 1 - Convolutions

# Define a Conv2D layers with required parameters
l1_g1 = tf.keras.layers.Conv2D(filters=48, kernel_size=11, strides=4, padding="same")(inputs)

# Define a BatchNormalization layer
l1_g1 = tf.keras.layers.BatchNormalization()(l1_g1)

# Define a ReLU layer
l1_g1 = tf.keras.layers.ReLU()(l1_g1)

# Define a MaxPooling2D layer
l1_g1 = tf.keras.layers.MaxPooling2D(pool_size=3, strides=2)(l1_g1)

# Define a Conv2D layers with required parameters
l1_g2 = tf.keras.layers.Conv2D(filters=48, kernel_size=11, strides=4, padding="same")(inputs)
```

```
# Define a BatchNormalization layer
l1_g2 = tf.keras.layers.BatchNormalization()(l1_g2)

# Define a ReLU layer
l1_g2 = tf.keras.layers.ReLU()(l1_g2)

# Define a MaxPooling2D layer
l1_g2 = tf.keras.layers.MaxPooling2D(pool_size=3, strides=2)(l1_g2)
```

So every feature map after the first convolution layer is of the size 55 x 55.

After this convolution, we have an **Overlapping Max Pool Layer**, where the max-pooling is done over a window of 3 x 3, and stride equals 2. So, here we'll find that as our max pooling window is of size 3 x 3 but the stride is 2 that means max pooling will be done over an overlapped window. After this pooling, the size of the feature map is reduced to 27 x 27 and the number of feature channels remains 96.

Calculations :

- Size of Input: N = 55 x 55
- Size of Convolution Kernels: f = 3 x 3
- Strides: S = 2
- Padding: P = 0
- *Size* of each feature map = (55–3+0)/2+1 = 27

So every feature map after this pooling is of the size 27 x 27.

Then we have **Second Convolution Layer** where kernel size is 5 x 5 and in this case, we use the padding of 2 so that the output of the convolution layer remains the same as the input feature size. Thus, the size of feature maps generated by this second convolution layer is 27 x 27 and the number of kernels used in this case is 256 so that means from this convolution layer output we'll get 256 different channels or feature maps and every feature map will be of size 27 x 27.

Calculation

- Size of Input: N = 27 x 27
- Size of Convolution Kernels: f = 5 x 5
- No. of Kernels: 256
- Strides: S = 1
- Padding: P = 2
- Size of each feature map = (27−5+4)/1+1 = 27

So every feature map after the second convolution layer is of the size 27 x 27.

```
# Layer 2 - Convolutions

# Define a Conv2D layers with required parameters
l2_g1 = tf.keras.layers.Conv2D(filters=128, kernel_size=5, strides=1, padding="same")(l1_g1)

# Define a BatchNormalization layer
l2_g1 = tf.keras.layers.BatchNormalization()(l2_g1)

# Define a ReLU layer
l2_g1 = tf.keras.layers.ReLU()(l2_g1)

# Define a MaxPooling2D layer
l2_g1 = tf.keras.layers.MaxPooling2D(pool_size=3, strides=2)(l2_g1)

# Define a Conv2D layers with required parameters
l2_g2 = tf.keras.layers.Conv2D(filters=128, kernel_size=5, strides=1, padding="same")(l1_g2)

# Define a BatchNormalization layer
l2_g2 = tf.keras.layers.BatchNormalization()(l2_g2)

# Define a ReLU layer
l2_g2 = tf.keras.layers.ReLU()(l2_g2)
```

```
# Define a MaxPooling2D layer
l2_g2 = tf.keras.layers.MaxPooling2D(pool_size=3, strides=2)(l2_g2)
```

Now again we have an Overlapping Max Pool Layer, where the max-pooling is again done over a window of 3 x 3, and stride equals 2 which means max-pooling is done over overlapping windows and output of this become 13 x 13 feature maps and number of channels we get is 256.

Calculations:

- Size of Input: N = 27 x 27
- Size of Convolution Kernels: f = 3 x 3
- Strides: S = 2
- Padding: P = 0
- Size of each feature map = (27−3+0)/2+1 = 13

So every feature map after this pooling is of the size 13 x13

Then we have Three Consecutive Convolution Layers of which the first convolution layer is having the kernel size of 3 x 3 with padding equal to 1 and 384 kernels give you 384 feature maps of size 13 x 13 which passes through the next convolution layer.

Calculations:

- Size of Input: N = 13 x 13
- Size of Convolution Kernels: f = 3 x 3
- No. of Kernels: 384
- Strides: S = 1
- Padding: P = 1
- Size of each feature map = (13−3+2)/1+1 = 13

```
# Layer 3 - Convolutions
```

```
# Concatenate previous last layers
l3_concat = tf.keras.layers.concatenate([l2_g1, l2_g2], axis=-1)

# Define a Conv2D layers with required parameters
l3_g1 = tf.keras.layers.Conv2D(filters=192, kernel_size=3, strides=1, padding="same")(l3_concat)

# Define a ReLU layer
l3_g1 = tf.keras.layers.ReLU()(l3_g1)

# Define a Conv2D layers with required parameters
l3_g2 = tf.keras.layers.Conv2D(filters=192, kernel_size=3, strides=1, padding="same")(l3_concat)

# Define a ReLU layer
l3_g2 = tf.keras.layers.ReLU()(l3_g2)
```

In the second convolution, the kernel size is 13 x 13 with padding equal to 1 and it has 384 number of kernels that means the output of this convolution layer will have 384 channels or 384 feature maps and every feature map is of size 13 x 13. As we have given padding equals 1 for a 3 x 3 kernel size and that's the reason the size of every feature map at the output of this convolution layer is remaining the same as the size of the feature maps which are inputted to this convolution layer.

Calculations:

- Size of Input: N = 13 x 13
- Size of Convolution Kernels: f = 3 x 3
- No. of Kernels: 384
- Strides: S = 1
- Padding: P = 1
- Size of each feature map = (13−3+2)/1+1 = 13

```
# Layer 4 - Convolutions
```

```
# Define a Conv2D layers with required parameters
l4_g1 = tf.keras.layers.Conv2D(filters=192, kernel_size=3, strides=1, padding="same")(l3_g1)

# Define a ReLU layer
l4_g1 = tf.keras.layers.ReLU()(l4_g1)

# Define a Conv2D layers with required parameters
l4_g2 = tf.keras.layers.Conv2D(filters=192, kernel_size=3, strides=1, padding="same")(l3_g2)

# Define a ReLU layer
l4_g2 = tf.keras.layers.ReLU()(l4_g2)
```

The output of this second convolution is again passed through a convolution layer where kernel size is again 3 x 3 and padding equal to 1 which means the output of this convolution layer generates feature maps of the same size of 13 x 13. But in this case, AlexNet uses 256 kernels so that means at the input of this convolution we have 384 channels which now get converted to 256 channels or we can say 256 feature maps are generated at the end of this convolution and every feature map is of size 13 x 13.

Calculations:

- Size of Input: N = 13 x 13
- Size of Convolution Kernels: f = 3 x 3
- No. of Kernels: 256
- Strides: S = 1
- Padding: P = 1
- Size of each feature map = (13−3+2)/1+1 = 13

```
# Layer 5 - Convolutions

# Define a Conv2D layers with required parameters
l5_g1 = tf.keras.layers.Conv2D(filters=128, kernel_size=3, strides=1, padding="same")(l4_g1)

# Define a ReLU layer
```

```
l5_g1 = tf.keras.layers.ReLU()(l5_g1)

# Define a MaxPooling2D layer
l5_g1 = tf.keras.layers.MaxPooling2D(pool_size=3, strides=2)(l5_g1)

# Define a Conv2D layers with required parameters
l5_g2 = tf.keras.layers.Conv2D(filters=128, kernel_size=3, strides=1, padding="same")(l4_g2)

# Define a ReLU layer
l5_g2 = tf.keras.layers.ReLU()(l5_g2)

# Define a MaxPooling2D layer
l5_g2 = tf.keras.layers.MaxPooling2D(pool_size=3, strides=2)(l5_g2)
```

Followed by the above is the next Overlapping Max Pool Layer, where the max-pooling is again done over a window of 3 x 3 and stride equal to 2 and that gives you the output feature maps and the number of channels remains same as 256 and the size of each feature map is 6 x 6.

Calculations:

- Size of Input: N = 13 x 13
- Size of Convolution Kernels: f = 3 x 3
- Strides: S = 2
- Padding: P = 0
- Size of each feature map = (13−3+0)/2+1 = 6

```
# Layer 6 - Dense
l6_pre = tf.keras.layers.concatenate([l5_g1, l5_g2], axis=-1)

# Define a Flatten layer
l6_pre = tf.keras.layers.Flatten()(l6_pre)

# Define a Dense layer
l6 = tf.keras.layers.Dense(units=4096)(l6_pre)
```

```
# Define a ReLU layer
l6 = tf.keras.layers.ReLU()(l6)

# Define a Dropout layer
l6 = tf.keras.layers.Dropout(rate=0.5)(l6)


# Layer 7 - Dense

# Define a Dense layer
l7 = tf.keras.layers.Dense(units=4096)(l6)

# Define a ReLU layer
l7 = tf.keras.layers.ReLU()(l7)

# Define a Dropout layer
l7 = tf.keras.layers.Dropout(rate=0.5)(l7)


# Layer 8 - Dense

# Define a Dense layer
l8 = tf.keras.layers.Dense(units=10)(l7)

# Define a Softmax layer
l8 = tf.keras.layers.Softmax(dtype=tf.float32, name="alexnet_output")(l8)
```

Now we have a fully connected layer which is the same as a multi-layer perception. The first two fully-connected layers have 4096 nodes each. After the above mentioned last max-pooling, we have a total of 6x6x256 i.e. 9216 nodes or features and each of these nodes is connected to each of the nodes in this fully-connected layer. So the number of connections we'll have in this case is 9216x4096. And then every node from this fully connected convolution layer provides input to every node in the second fully connected layer. So here we'll have a total of 4096x4096 connections as in the second fully connected layer also we have 4096 nodes.

And then, in the end, we have an output layer with 1000 softmax channels. Thus the number of connections between the second fully

```
# Create a model using the layers created
alexnet = tf.keras.models.Model(inputs=inputs, outputs=l8)
```

Now let's look at the summary

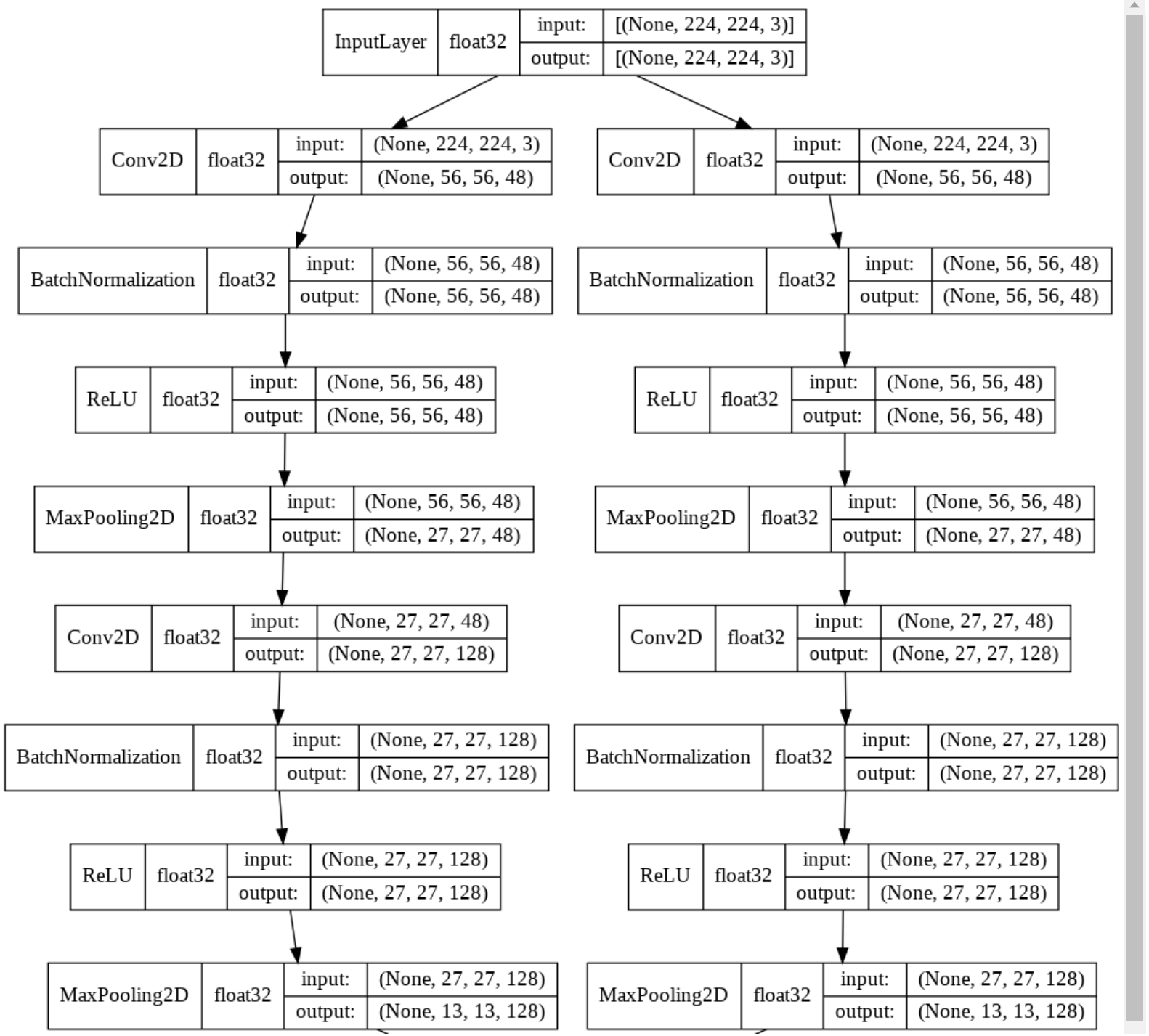
```
# Model Summary
alexnet.summary()
```

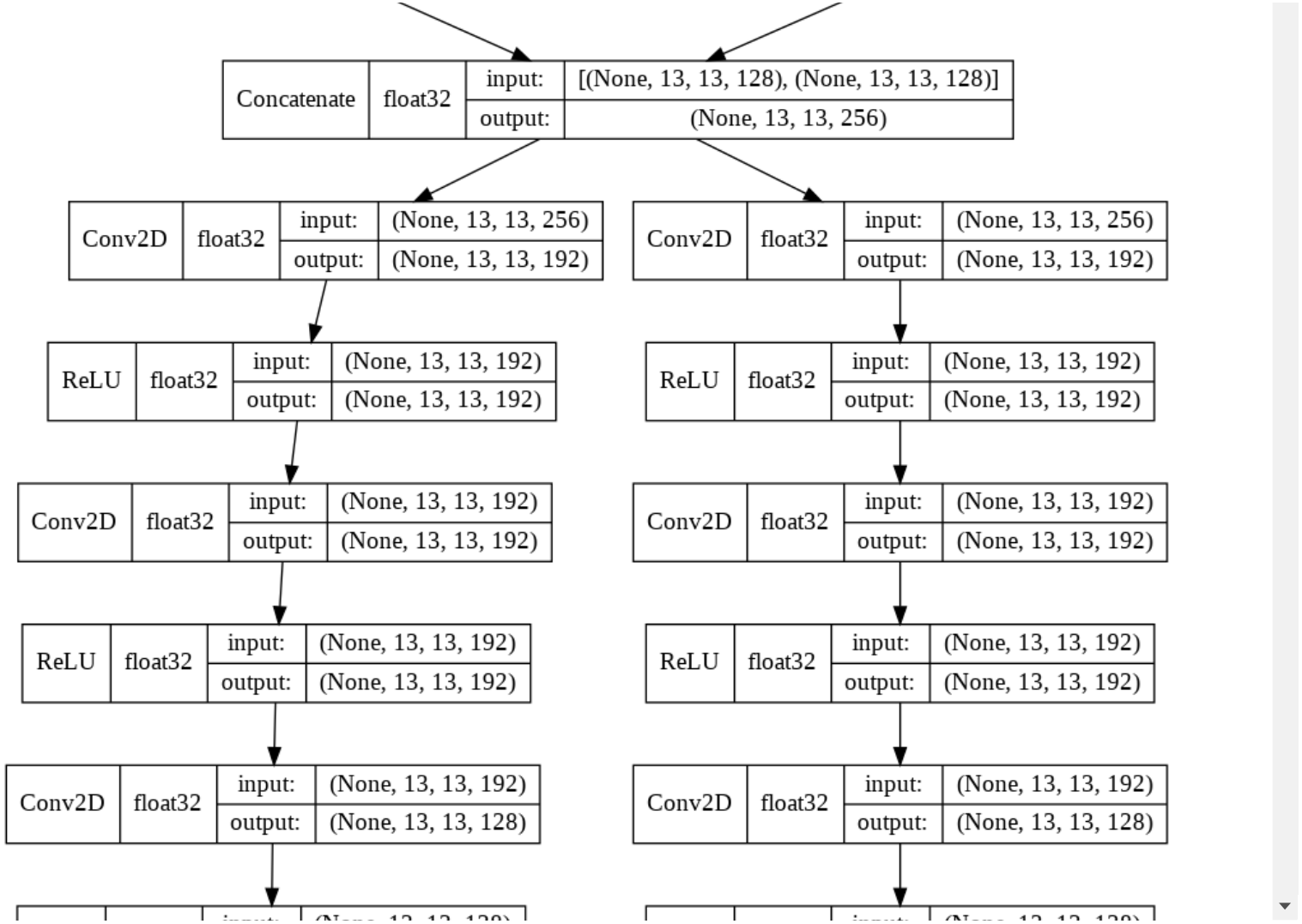
```
Model: "model"
_________________________________________________________________________________________________
 Layer (type)                   Output Shape         Param #     Connected to
==================================================================================================
 alexnet_input (InputLayer)     [(None, 224, 224, 3  0           []
                                )]

 conv2d (Conv2D)                (None, 56, 56, 48)   17472       ['alexnet_input[0][0]']

 conv2d_1 (Conv2D)              (None, 56, 56, 48)   17472       ['alexnet_input[0][0]']

 batch_normalization (BatchNorm  (None, 56, 56, 48)  192         ['conv2d[0][0]']
 alization)

 batch_normalization_1 (BatchNo  (None, 56, 56, 48)  192         ['conv2d_1[0][0]']
 rmalization)

 re_lu (ReLU)                   (None, 56, 56, 48)   0           ['batch_normalization[0][0]']

 re_lu_1 (ReLU)                 (None, 56, 56, 48)   0           ['batch_normalization_1[0][0]']

 max_pooling2d (MaxPooling2D)   (None, 27, 27, 48)   0           ['re_lu[0][0]']

 max_pooling2d_1 (MaxPooling2D)  (None, 27, 27, 48)  0           ['re_lu_1[0][0]']

 conv2d_2 (Conv2D)              (None, 27, 27, 128)  153728      ['max_pooling2d[0][0]']

 conv2d_3 (Conv2D)              (None, 27, 27, 128)  153728      ['max_pooling2d_1[0][0]']
```

```
batch_normalization_2 (BatchNo  (None, 27, 27, 128)  512        ['conv2d_2[0][0]']
rmalization)

batch_normalization_3 (BatchNo  (None, 27, 27, 128)  512        ['conv2d_3[0][0]']
rmalization)

re_lu_2 (ReLU)                 (None, 27, 27, 128)  0           ['batch_normalization_2[0][0]']

re_lu_3 (ReLU)                 (None, 27, 27, 128)  0           ['batch_normalization_3[0][0]']

max_pooling2d_2 (MaxPooling2D)  (None, 13, 13, 128)  0          ['re_lu_2[0][0]']

max_pooling2d_3 (MaxPooling2D)  (None, 13, 13, 128)  0          ['re_lu_3[0][0]']

concatenate (Concatenate)      (None, 13, 13, 256)  0           ['max_pooling2d_2[0][0]',
                                                                 'max_pooling2d_3[0][0]']

conv2d_4 (Conv2D)              (None, 13, 13, 192)  442560      ['concatenate[0][0]']

conv2d_5 (Conv2D)              (None, 13, 13, 192)  442560      ['concatenate[0][0]']

re_lu_4 (ReLU)                 (None, 13, 13, 192)  0           ['conv2d_4[0][0]']

re_lu_5 (ReLU)                 (None, 13, 13, 192)  0           ['conv2d_5[0][0]']

conv2d_6 (Conv2D)              (None, 13, 13, 192)  331968      ['re_lu_4[0][0]']

conv2d_7 (Conv2D)              (None, 13, 13, 192)  331968      ['re_lu_5[0][0]']
```

Visualising the architecture

```
# Architecture Visualized
tf.keras.utils.plot_model(alexnet, show_layer_names=False, show_shapes=True, show_dtype=True)
```


InputLayer | float32 | input: [(None, 224, 224, 3)] | output: [(None, 224, 224, 3)]
Conv2D | float32 | input: (None, 224, 224, 3) | output: (None, 56, 56, 48)
Conv2D | float32 | input: (None, 224, 224, 3) | output: (None, 56, 56, 48)
BatchNormalization | float32 | input: (None, 56, 56, 48) | output: (None, 56, 56, 48)
BatchNormalization | float32 | input: (None, 56, 56, 48) | output: (None, 56, 56, 48)
ReLU | float32 | input: (None, 56, 56, 48) | output: (None, 56, 56, 48)
ReLU | float32 | input: (None, 56, 56, 48) | output: (None, 56, 56, 48)
MaxPooling2D | float32 | input: (None, 56, 56, 48) | output: (None, 27, 27, 48)
MaxPooling2D | float32 | input: (None, 56, 56, 48) | output: (None, 27, 27, 48)
Conv2D | float32 | input: (None, 27, 27, 48) | output: (None, 27, 27, 128)
Conv2D | float32 | input: (None, 27, 27, 48) | output: (None, 27, 27, 128)
BatchNormalization | float32 | input: (None, 27, 27, 128) | output: (None, 27, 27, 128)
BatchNormalization | float32 | input: (None, 27, 27, 128) | output: (None, 27, 27, 128)
ReLU | float32 | input: (None, 27, 27, 128) | output: (None, 27, 27, 128)
ReLU | float32 | input: (None, 27, 27, 128) | output: (None, 27, 27, 128)
MaxPooling2D | float32 | input: (None, 27, 27, 128) | output: (None, 13, 13, 128)
MaxPooling2D | float32 | input: (None, 27, 27, 128) | output: (None, 13, 13, 128)


Concatenate | float32 | input: | [(None, 13, 13, 128), (None, 13, 13, 128)]
output: | (None, 13, 13, 256)
Conv2D | float32 | input: | (None, 13, 13, 256)
output: | (None, 13, 13, 192)
Conv2D | float32 | input: | (None, 13, 13, 256)
output: | (None, 13, 13, 192)
ReLU | float32 | input: | (None, 13, 13, 192)
output: | (None, 13, 13, 192)
ReLU | float32 | input: | (None, 13, 13, 192)
output: | (None, 13, 13, 192)
Conv2D | float32 | input: | (None, 13, 13, 192)
output: | (None, 13, 13, 192)
Conv2D | float32 | input: | (None, 13, 13, 192)
output: | (None, 13, 13, 192)
ReLU | float32 | input: | (None, 13, 13, 192)
output: | (None, 13, 13, 192)
ReLU | float32 | input: | (None, 13, 13, 192)
output: | (None, 13, 13, 192)
Conv2D | float32 | input: | (None, 13, 13, 192)
output: | (None, 13, 13, 128)
Conv2D | float32 | input: | (None, 13, 13, 192)
output: | (None, 13, 13, 128)

```
# Imports
import tensorflow as tf
from tensorflow import keras
import matplotlib.pyplot as plt
import os
import time


# Load Cifar100 dataset from Keras Datasets
(train_images, train_labels), (test_images, test_labels) = keras.datasets.cifar10.load_data()
```

```
Downloading data from https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz
170500096/170498071 [==============================] - 2s 0us/step
170508288/170498071 [==============================] - 2s 0us/step
```

```
# Split the data in training, validation and testing sets
validation_images, validation_labels = train_images[:5000], train_labels[:5000]
train_images, train_labels = train_images[5000:], train_labels[5000:]


# Make a train. test, validation dataset object
train_ds = tf.data.Dataset.from_tensor_slices((train_images, train_labels))
test_ds = tf.data.Dataset.from_tensor_slices((test_images, test_labels))
validation_ds = tf.data.Dataset.from_tensor_slices((validation_images, validation_labels))


# Function to process images
def process_images(image, label):
    # Normalize images to have a mean of 0 and standard deviation of 1
    image = tf.image.per_image_standardization(image)
    # Resize images from 32x32 to 277x277
    image = tf.image.resize(image, (224,224))
    return image, label
```

```
# Get size of the train, test and validation set
train_ds_size = tf.data.experimental.cardinality(train_ds).numpy()
test_ds_size = tf.data.experimental.cardinality(test_ds).numpy()
validation_ds_size = tf.data.experimental.cardinality(validation_ds).numpy()
print("Training data size:", train_ds_size)
print("Test data size:", test_ds_size)
print("Validation data size:", validation_ds_size)
```

```
Training data size: 45000
Test data size: 10000
Validation data size: 5000
```

```
# Define batch and shuffle config for train, test and validation set
train_ds = (train_ds
                  .map(process_images)
                  .shuffle(buffer_size=train_ds_size/10)
                  .batch(batch_size=1, drop_remainder=True))
test_ds = (test_ds
                  .map(process_images)
                  .shuffle(buffer_size=train_ds_size/10)
                  .batch(batch_size=1, drop_remainder=True))
validation_ds = (validation_ds
                  .map(process_images)
                  .shuffle(buffer_size=train_ds_size/10)
                  .batch(batch_size=1, drop_remainder=True))
```

```
# Compile the model by passing loss and optimizer
alexnet.compile(loss='sparse_categorical_crossentropy', optimizer=tf.optimizers.SGD(lr=0.001), metrics=['accuracy'])
```

```
/usr/local/lib/python3.7/dist-packages/keras/optimizer_v2/gradient_descent.py:102: UserWarning: The `lr` argument is
  super(SGD, self).__init__(name, **kwargs)
```

```
# Fit the model upto 5 epochs with dataset and other required parameters
alexnet.fit(train_ds,
```