

AlexNet architecture

AlexNet model contains five convolutional and three fully-connected layers. Removing any convolutional layer resulted in inferior performance.

The first convolutional layer filters the $227 \times 227 \times 3$ input image with 96 kernels of size $11 \times 11 \times 3$ with a stride of 4 pixels (this is the distance between the receptive field centers of neighboring neurons in a kernel map). The second convolutional layer takes as input the (response-normalized and pooled) output of the first convolutional layer and filters it with 256 kernels of size $5 \times 5 \times 48$. The third, fourth, and fifth convolutional layers are connected to one another without any intervening pooling or normalization layers. The third convolutional layer has 384 kernels of size $3 \times 3 \times 256$ connected to the (normalized, pooled) outputs of the second convolutional layer. The fourth convolutional layer has 384 kernels of size $3 \times 3 \times 192$, and the fifth convolutional layer has 256 kernels of size $3 \times 3 \times 192$. The fully-connected layers have 4096 neurons each.

The input to the AlexNet network is a 227×227 size RGB image, so it's having 3 different channels- red, green, and blue.

There are no hard rule on the number of filters, also there are no paper discuss about why AlexNet model use a specific number of filter in each convolutional layers.

First Convolution Layer in the AlexNet that has 96 different kernels with each kernel's size of 11×11 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 55×55 .

Calculations:

- **Size of Input:** $N = 227 \times 227$
- **Size of Convolution Kernels:** $f = 11 \times 11$
- **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** $= (227 - 11 + 0)/4 + 1 = 55$

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

Overlapping Max Pool Layer, where the max-pooling is done over a window of 3×3 , and stride equals 2. So, here we'll find that as our max pooling window is of size 3×3 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 27×27 and the number of feature channels remains 96.

Calculations:

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

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

Second Convolution Layer where kernel size is 5×5 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 27×27 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 27×27 .

Calculations:

- **Size of Input:** $N = 27 \times 27$
- **Size of Convolution Kernels:** $f = 5 \times 5$
- **No. of Kernels:** 256
- **Strides:** $S = 1$
- **Padding:** $P = 2$
- **Size of each feature map** $= [(N - f + 2P)/S] + 1$
- **Size of each feature map** $= (27 - 5 + 4)/1 + 1 = 27$

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

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

Calculations:

- **Size of Input:** $N = 27 \times 27$
- **Size of Convolution Kernels:** $f = 3 \times 3$
- **Strides:** $S = 2$
- **Padding:** $P = 0$
- **Size of each feature map** $= [(N - f + 2P)/S] + 1$
- **Size of each feature map** $= (27 - 3 + 0)/2 + 1 = 13$

So, every feature map after this pooling is of the size 13×13 .

Three Consecutive Convolution Layers

The *first convolution layer* is having the kernel size of 3×3 with padding equal to 1 and 384 kernels give you 384 feature maps of size 13×13 which passes through the next convolution layer.

Calculations:

- **Size of Input:** $N = 13 \times 13$
- **Size of Convolution Kernels:** $f = 3 \times 3$
- **No. of Kernels:** 384
- **Strides:** $S = 1$
- **Padding:** $P = 1$
- **Size of each feature map** $= [(N - f + 2P)/S] + 1$
- **Size of each feature map** $= (13 - 3 + 2)/1 + 1 = 13$

In the *second convolution*, the kernel size is 3×3 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 13×13 . As we have given padding equals 1 for a 3×3 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:** $N = 13 \times 13$
- **Size of Convolution Kernels:** $f = 3 \times 3$
- **No. of Kernels:** 384
- **Strides:** $S = 1$
- **Padding:** $P = 1$
- **Size of each feature map** $= [(N - f + 2P)/S] + 1$
- **Size of each feature map** $= (13 - 3 + 2)/1 + 1 = 13$

The output of this second convolution is again passed through a convolution layer where kernel size is again 3×3 and padding equal to 1 which means the output of this convolution layer generates feature maps of the same size of 13×13 . 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:** $N = 13 \times 13$
- **Size of Convolution Kernels:** $f = 3 \times 3$
- **No. of Kernels:** 256
- **Strides:** $S = 1$
- **Padding:** $P = 1$
- **Size of each feature map** $= [(N - f + 2P)/S] + 1$
- **Size of each feature map** $= (13 - 3 + 2)/1 + 1 = 13$
-

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:** $N = 13 \times 13$
- **Size of Convolution Kernels:** $f = 3 \times 3$
- **Strides:** $S = 2$
- **Padding:** $P = 0$
- **Size of each feature map** $= [(N - f + 2P)/S] + 1$
- **Size of each feature map** $= (13 - 3 + 0)/2 + 1 = 6$

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 $6 \times 6 \times 256$ 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 will have in this case is 9216×4096 . And then every node from this fully connected convolution layer provides input to every node in the second fully connected layer. So here we will have a total of 4096×4096 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 connected layer and the output layer is 4096×1000 .

AlexNet requires a constant input dimensionality.

Pooling layers in CNNs summarize the outputs of neighboring groups of neurons in the same kernel map.

AlexNet model use overlapping pool

ReLU (Rectified Linear Unit) activation function

In terms of training time with gradient descent, these saturating nonlinearities are much slower than the non-saturating nonlinearity $f(x) = \max(0, x)$

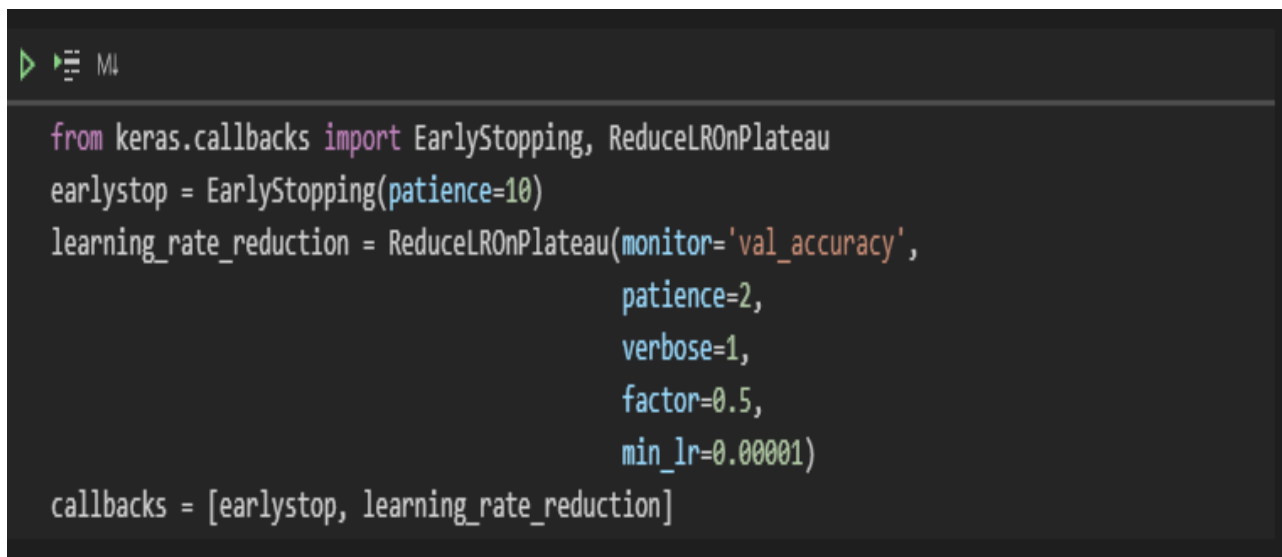
Problem of overfitting

Because AlexNet model has 60 million parameters to be trained which make the model become complexing. Therefore, this would lead to the problem of overfitting which means the network would be able to learn or memorize the training data very properly. So to reduce this overfitting issue additional augmented data was generated from the existing data and the augmentation was done by **rotating, rescale, zoom, vertical and horizontal flip**. Another method by which the problem of overfitting was taken care of is by using dropout regularization

Dropout

In this randomly selected neuron or randomly selected nodes, which are selected with a probability of 0.5 are dropped from the network temporarily. So the probability of the nodes that are removed and the probability of the nodes that are retained both will be equal to 0.5. So the dropout means that the node which has been dropped out will not pass its output to the subsequent nodes in the subsequent layers downstream and for the same nodes during backward propagation, no updating will take place as removing the nodes will also remove its subsequent connections as well. But this would increase the number of iterations required for training the model but this would make the model less vulnerable to overfitting thus, generalizing the model.

EarlyStopping and ReduceLROnPlateau



```
from keras.callbacks import EarlyStopping, ReduceLROnPlateau
earlystop = EarlyStopping(patience=10)
learning_rate_reduction = ReduceLROnPlateau(monitor='val_accuracy',
                                             patience=2,
                                             verbose=1,
                                             factor=0.5,
                                             min_lr=0.00001)

callbacks = [earlystop, learning_rate_reduction]
```

EarlyStopping is used to stop training when a monitored metric has stopped improving.

```
tf.keras.callbacks.EarlyStopping(
    monitor="val_loss",
    min_delta=0,
    patience=0,
```

```

verbose=0,
mode="auto",
baseline=None,
restore_best_weights=False,
)

```

Arguments:

- **monitor:** Quantity to be monitored.
- **min_delta:** Minimum change in the monitored quantity to qualify as an improvement, i.e. an absolute change of less than min_delta, will count as no improvement.
- **patience:** Number of epochs with no improvement after which training will be stopped.
- **verbose:** verbosity mode.
- **mode:** One of {"auto", "min", "max"}. In min mode, training will stop when the quantity monitored has stopped decreasing; in "max" mode it will stop when the quantity monitored has stopped increasing; in "auto" mode, the direction is automatically inferred from the name of the monitored quantity.
- **baseline:** Baseline value for the monitored quantity. Training will stop if the model doesn't show improvement over the baseline.
- **restore_best_weights:** Whether to restore model weights from the epoch with the best value of the monitored quantity. If False, the model weights obtained at the last step of training are used.

ReduceLROnPlateau is reduce learning rate when a metric has stopped improving. Models often benefit from reducing the learning rate by a factor of 2-10 once learning stagnates. This callback monitors a quantity and if no improvement is seen for a 'patience' number of epochs, the learning rate is reduced.

```

tf.keras.callbacks.ReduceLROnPlateau(
    monitor="val_loss",
    factor=0.1,
    patience=10,
    verbose=0,
    mode="auto",
    min_delta=0.0001,
    cooldown=0,
    min_lr=0,
    **kwargs
)

```

Arguments

- **monitor:** quantity to be monitored.
- **factor:** factor by which the learning rate will be reduced. $\text{new_lr} = \text{lr} * \text{factor}$.
- **patience:** number of epochs with no improvement after which learning rate will be reduced.
- **verbose:** int. 0: quiet, 1: update messages.
- **mode:** one of {'auto', 'min', 'max'}. In 'min' mode, the learning rate will be reduced when the quantity monitored has stopped decreasing; in 'max' mode it will be reduced when the quantity monitored has stopped increasing; in 'auto' mode, the direction is automatically inferred from the name of the monitored quantity.
- **min_delta:** threshold for measuring the new optimum, to only focus on significant changes.

- **cooldown:** number of epochs to wait before resuming normal operation after lr has been reduced.
- **min_lr:** lower bound on the learning rate.

References:

https://keras.io/api/callbacks/early_stopping/

https://keras.io/api/callbacks/reduce_lr_on_plateau/

<https://stackoverflow.com/questions/37674306/what-is-the-difference-between-same-and-valid-padding-in-tf-nn-max-pool-of-t>

<https://www.machinecurve.com/index.php/2020/02/08/how-to-use-padding-with-keras/>

<https://medium.com/analytics-vidhya/the-architecture-implementation-of-alexnet-135810a3370>

<https://www.xpertup.com/blog/deep-learning/tensorflow-tensor-shape-dtype-graph-session/>

<https://androidkt.com/batch-size-step-iteration-epoch-neural-network/>

http://d2l.ai/chapter_convolutional-neural-networks/index.html

http://d2l.ai/chapter_convolutional-modern/alexnet.html

https://www.tensorflow.org/api_docs/python/tf/keras/preprocessing/image/ImageDataGenerator#used-in-the-notebooks_1

<https://nanonets.com/blog/data-augmentation-how-to-use-deep-learning-when-you-have-limited-data-part-2/>