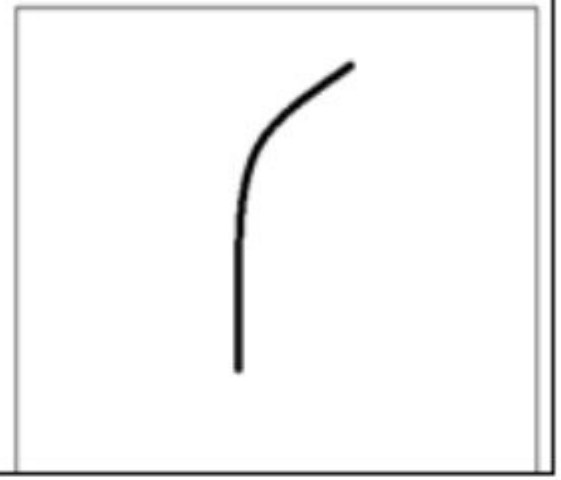
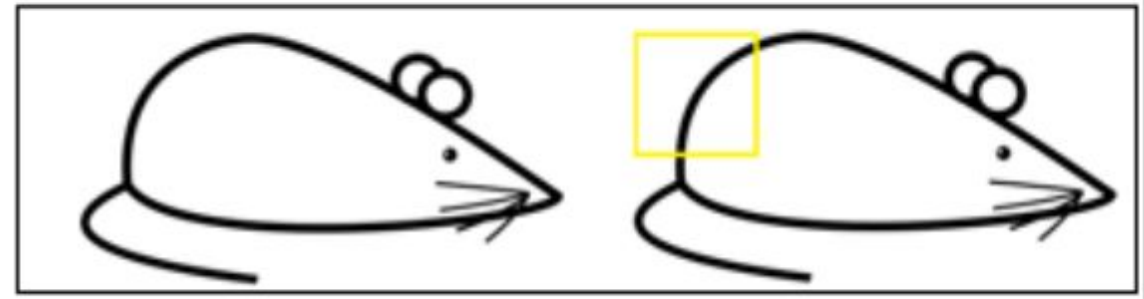
RECAP

Topic Already Covered:

- Computer Vision Introduction
- Convolution Operation
- Edge Detection
- Padding
- Stride

0	0	0	0	0	30	0
0	0	0	0	30	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	0	0	0	0





0	0	0	0	0	0	30
0	0	0	0	50	50	50
0	0	0	20	50	0	0
0	0	0	50	50	0	0
0	0	0	50	50	0	0
0	0	0	50	50	0	0
0	0	0	50	50	0	0

*

0	0	0	0	0	30	0
0	0	0	0	30	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	0	0	0	0

Multiplication and Summation = (50*30)+(50*30)+(50*30)+(20*30)+(50*30) = 6600 (A large number!)

0	0	0	0	0	0	0
0	40	0	0	0	0	0
40	0	40	0	0	0	0
40	20	0	0	0	0	0
0	50	0	0	0	0	0
0	0	50	0	0	0	0
25	25	0	50	0	0	0

*

0	0	0	0	0	30	0
0	0	0	0	30	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	0	0	0	0

Multiplication and Summation = 0

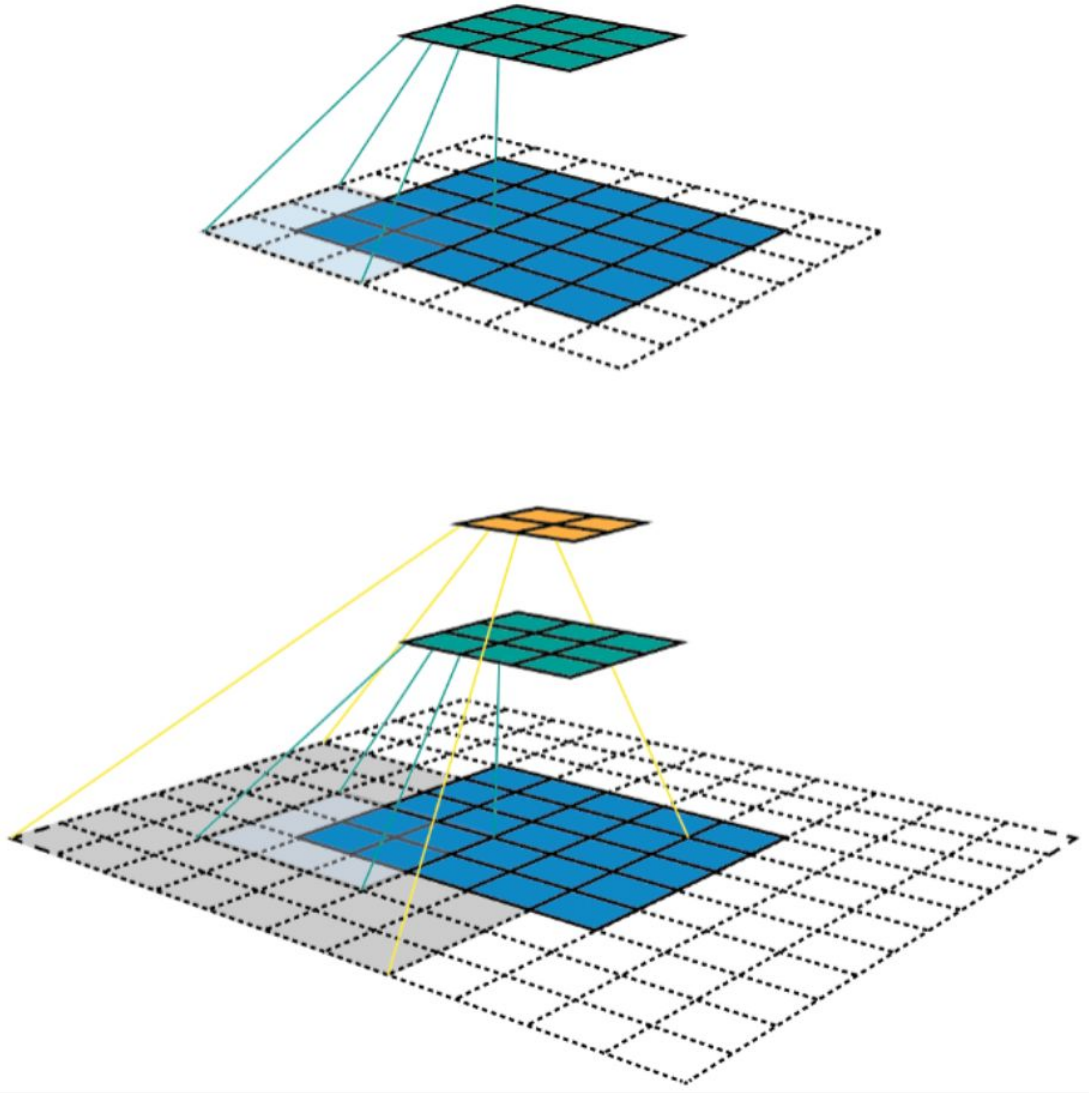
Today we will see..

- Receptive Field
- Pooling
- Dropout
- Batch Normalisation
- Augmentation

Receptive Field

- *The **receptive field** is defined as the region in the input space that a particular CNN's feature is looking at (i.e. be affected by).*
- A receptive field of a feature can be described by its **center location** and its **size**.

Visualization of Receptive field



Number of layers to accommodate whole of the input

2*num_of_layers + 1 >= r

2 * x + 1 >= 5
=> x >= 2 so, 3 Layers!!

What about for a 1000-by-1000 image??

Ref: https://www.youtube.com/watch?v=70A3uYfM1qA&ab_channel=AppliedAICourse

Problem of Diminishing Gradients

- The gradients to be learnt at a later stage in the layer hierarchy might be very very different from the gradients to be learnt at an early stage.
- We tend to lose gradients which are smaller - this is the problem of diminishing gradients
- How to solve? - Reduce the number of layers?

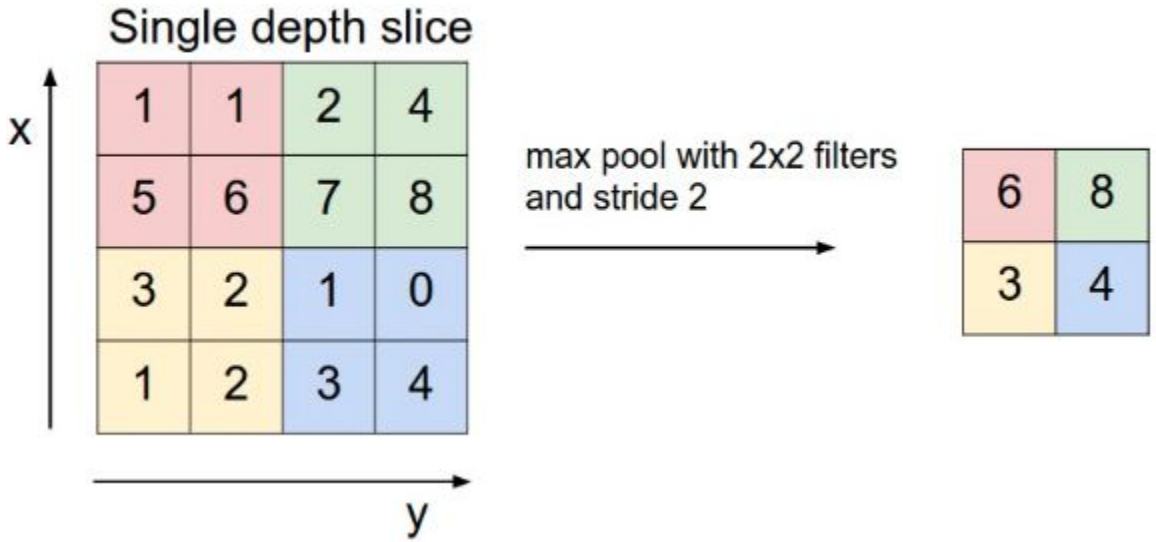
Ref: https://www.youtube.com/watch?v=KKmCnwGzSv8&ab_channel=Serrano.Academy

How to reduce the number of layers?

- **Pooling! - One solution..**
- **WHAT EXACTLY IS REDUCED? - Size of features!!!**

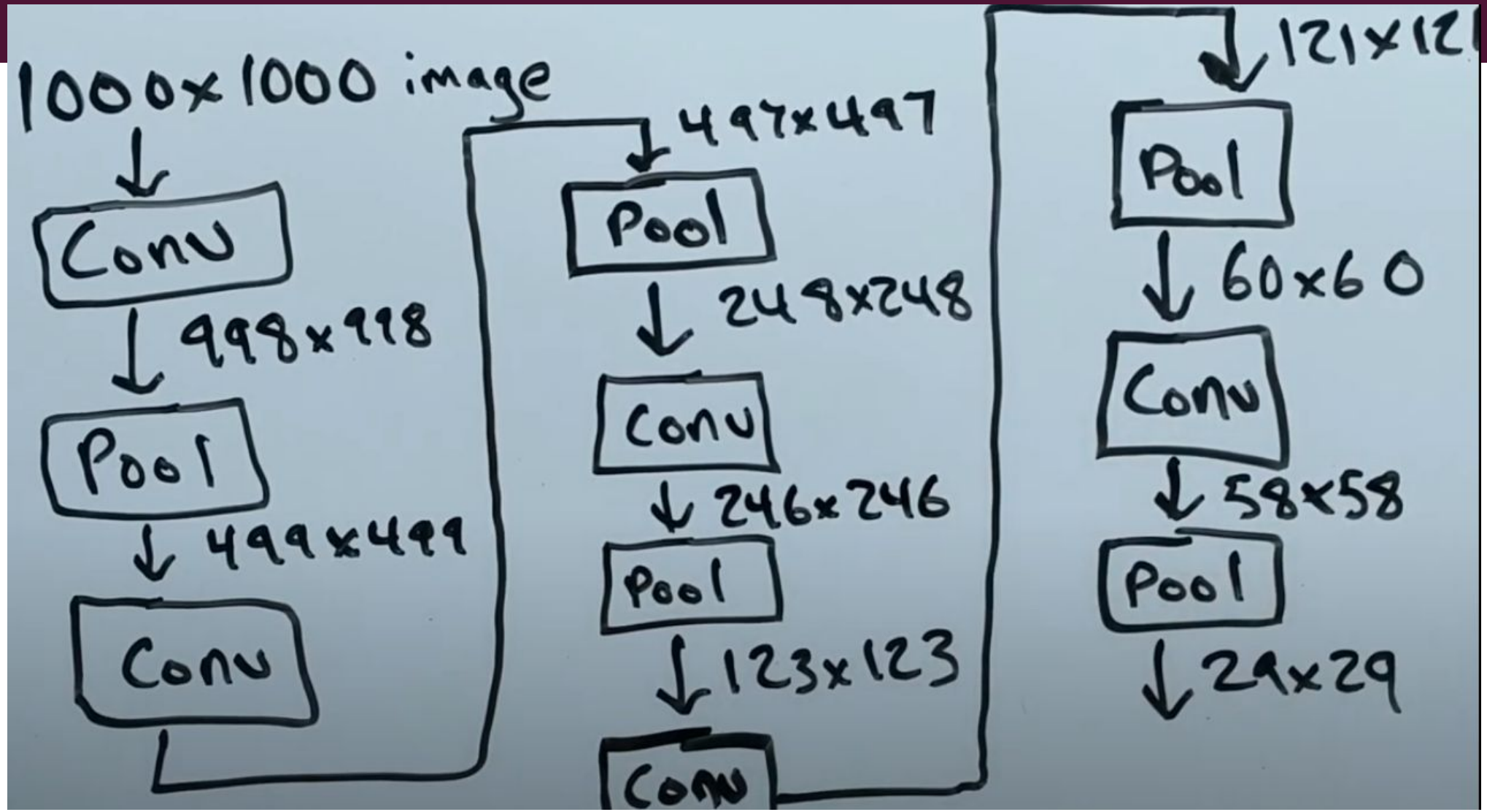
Pooling techniques

Max pooling where we take largest of the pixel values of a segment.

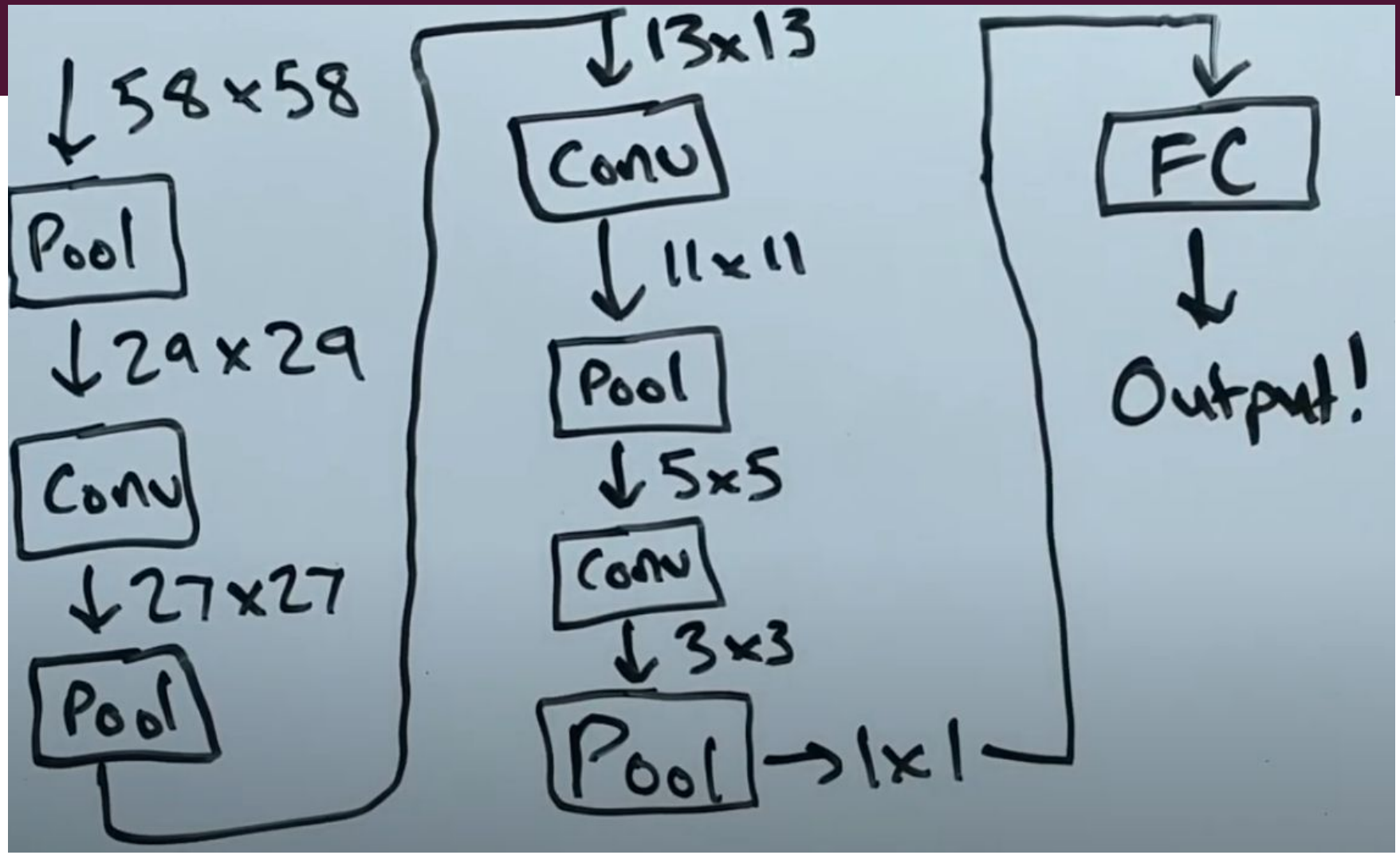


Ref: https://cs231n.github.io/convolutional-networks/#pool

Reductions in a 1000-by-1000 image



Reductions in a 1000-by-1000 image



Ref: https://www.youtube.com/watch?v=70A3uYfM1qA&ab_channel=AppliedAICourse

New number of layers?

Instead of 500 CONV layers, we now have:

8 CONV layers,
8 Pooling layers,
1 FC layer!!

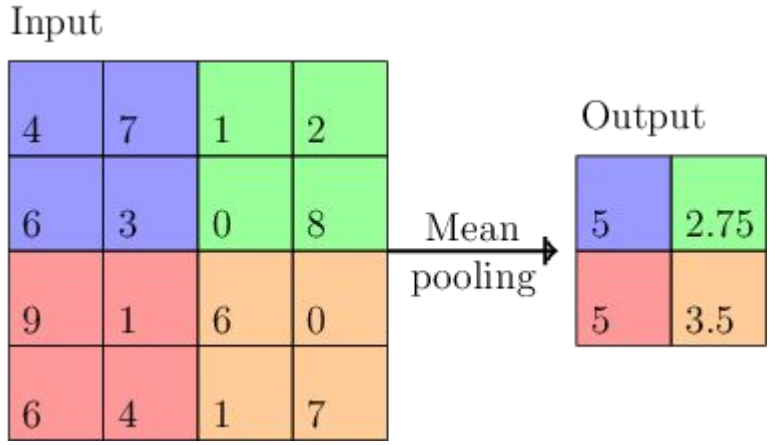
Total = 17 layers!

Pooling

- Pooling is nothing other than down sampling of an image.
- The most common pooling layer filter is of size 2x2, which discards three fourth of the activations.
- Role of pooling layer is to reduce the resolution of the feature map but retaining features of the map required for classification through translational and rotational invariants.
- In addition to spatial invariance robustness, pooling will reduce the computation cost by a great deal - **no learnable parameters!**

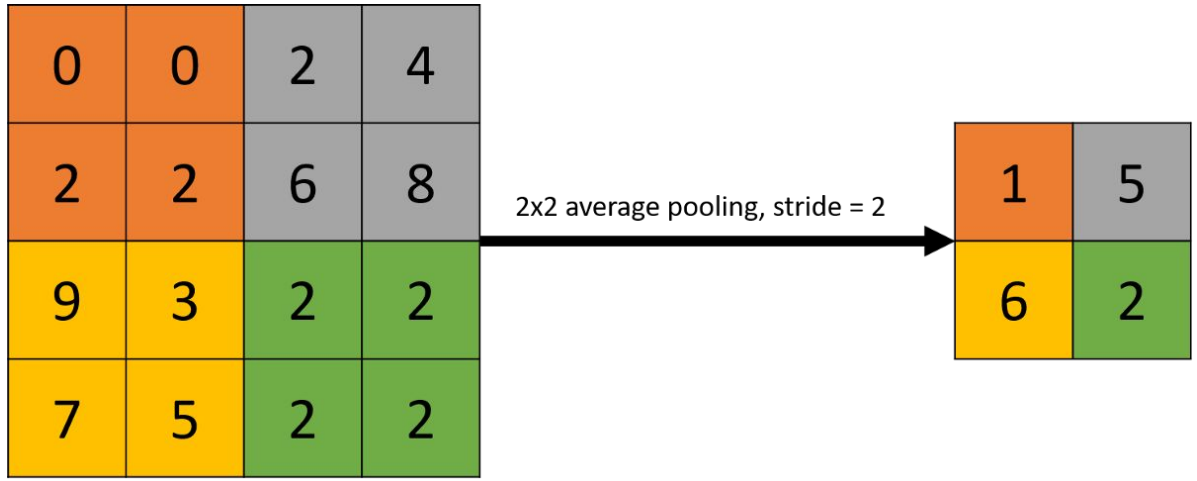
Pooling techniques

Mean pooling where we take largest of the pixel values of a segment



Pooling techniques

Avg pooling where we take largest of the pixel values of a segment.



Other pooling

- Min pooling

- wavelet pooling

- tree pooling

- max-avg pooling

- spatial pyramid pooling

Dropout

Deep learning neural networks are likely to quickly overfit a training dataset with few examples.

Dropout

A single model can be used to simulate having a large number of different network architectures by randomly dropping out nodes during training.

This is called ***dropout*** and offers a very computationally cheap and remarkably effective regularization method to reduce overfitting and improve generalization error in deep neural networks of all kinds

Dropout

Dropout is a regularization method that approximates training a large number of neural networks with different architectures in parallel.

Dropout

- During training, some number of layer outputs are randomly ignored or "dropped out."
- This has the effect of making the layer look-like and be treated-like a layer with a different number of nodes and connectivity to the prior layer.
- In effect, each update to a layer during training is performed with a different "view" of the configured layer.

Dropout

- Dropout has the effect of making the training process noisy, forcing nodes within a layer to probabilistically take on more or less responsibility for the inputs.

- This conceptualization suggests that perhaps dropout breaks-up situations where network layers co-adapt to correct mistakes from prior layers, in turn making the model more robust.

How to Dropout

- Dropout is implemented per-layer in a neural network.
- It can be used with most types of layers, such as dense fully connected layers, convolutional layers, and recurrent layers such as the long short-term memory network layer.
- Dropout may be implemented on any or all hidden layers in the network as well as the visible or input layer.
- It is not used on the output layer.

Dropout

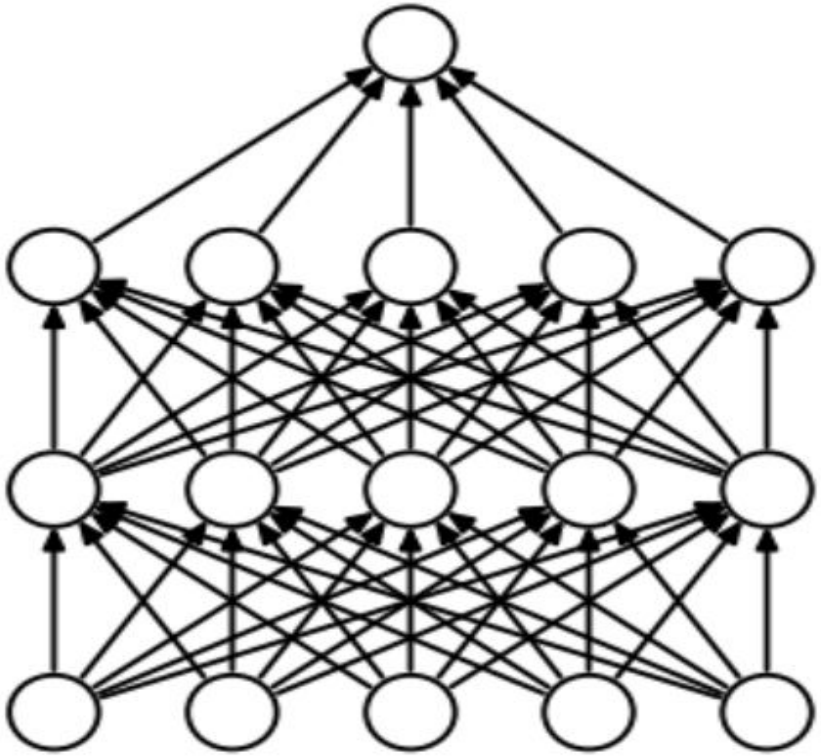
Dropout is not used after training when making a prediction with the fit network.

The weights of the network will be larger than normal because of dropout.

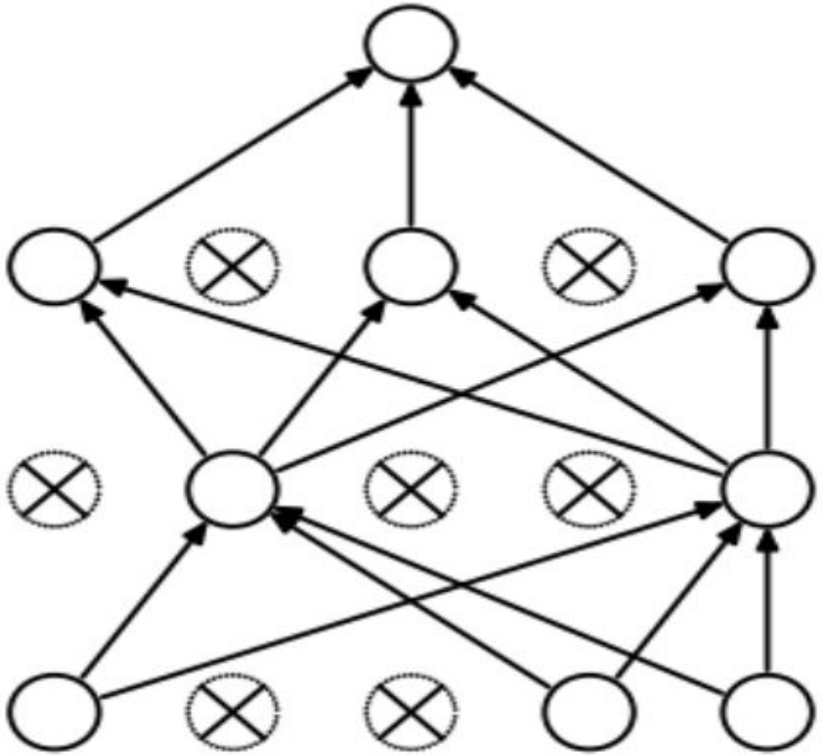
Therefore, before finalizing the network, the weights are first scaled by the chosen dropout rate.

The network can then be used as per normal to make predictions.

Dropout



(a) Standard Neural Net



(b) After applying dropout.

Batch normalization

Training deep neural networks with tens of layers is challenging as they can be sensitive to the initial random weights and configuration of the learning algorithm.

Batch normalization

Batch normalization is a technique for training very deep neural networks that standardizes the inputs to a layer for each mini-batch.

This has the effect of stabilizing the learning process and dramatically reducing the number of training epochs required to train deep networks.

We know how scaling and normalization of input improves the model

Batch normalization is similar but not on input but at hidden layer

It makes weights at deeper layer more robust to change

Thus helps in generalization of the model

Batch normalization as a regularizer

Each mini batch is scaled by mean/variance computed just on mini batch

It handles data at one batch at a time

Thus it adds some noise to hidden layer and thus gives slight regularization effect

And makes sure that the weights in network do not become extremely high or low

But deep learning libraries like keras provide with batch norm function

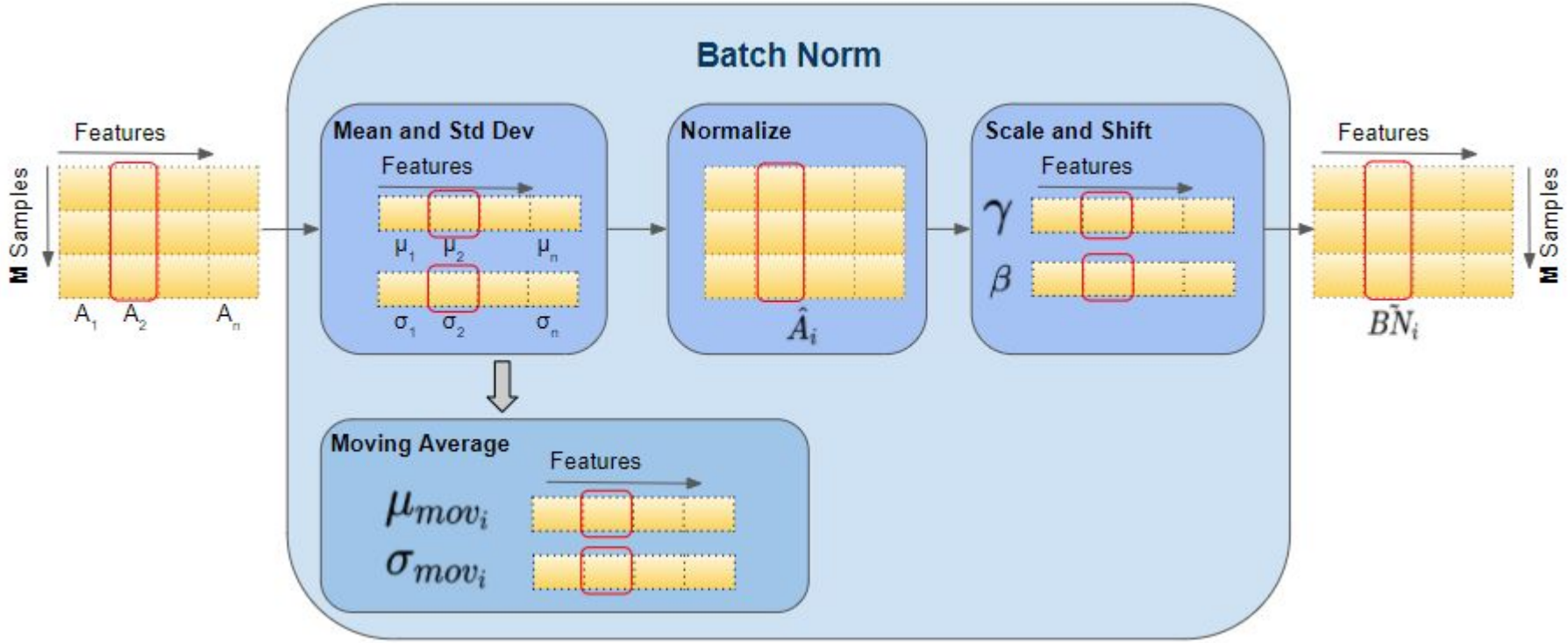
```
model = Sequential
model.add(Dense(32))
model.add(BatchNormalization())
model.add(Activation('relu'))
```

```
from keras.layers import BatchNormalization
...
model.add(Conv2D(32, (3,3), activation='relu'))
model.add(Conv2D(32, (3,3), activation='relu'))
model.add(BatchNormalization())
model.add(MaxPooling2D())
model.add(Dense(1))
```

What are the advantages of Batch Normalisation?

- The model is less delicate to hyperparameter tuning.
- Diminishes the reliance of gradients on the scale of the parameters or their underlying values
- Weight initialization is a smidgen less significant at this point
- Dropout can be evacuated for regularisation

Batch normalization



We can write batch norm code as

```python
def batchnorm_backward_alt(dout, cache):
    """
    Alternative backward pass for batch normalization.
    Work out the derivatives for the batch normalizaton backward pass on paper
    and simplify as much as possible.

    Note: This implementation receives the same cache variable as
    batchnorm_backward, but does not use all of the values in the cache.
    Inputs / outputs: Same as batchnorm_backward
    """
    N = dout.shape[0]
    x_norm, x_centered, std, gamma = cache

    dgamma = (dout * x_norm).sum(axis=0)
    dbeta = dout.sum(axis=0)

    dx_norm = dout * gamma
    dx = 1/N / std * (N * dx_norm -
                      dx_norm.sum(axis=0) -
                      x_norm * (dx_norm * x_norm).sum(axis=0))

    return dx, dgamma, dbeta
```

Augmentation

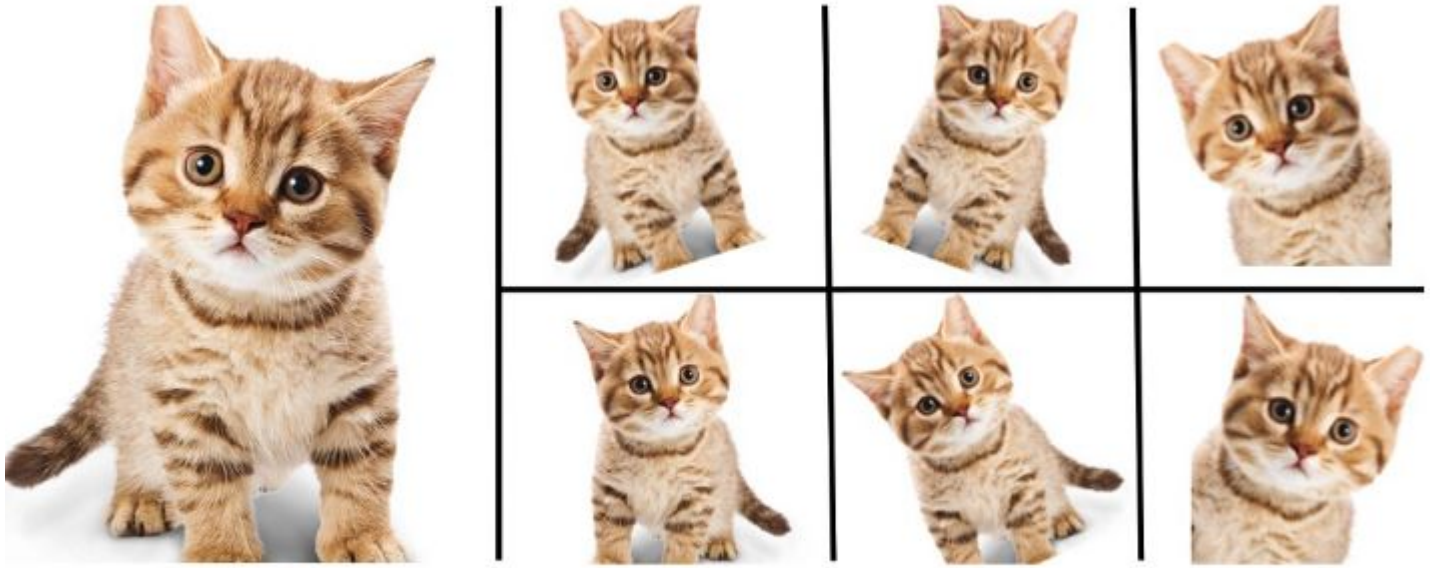
CNN is not rotation and scale invariant

Eg we train our model on best pictures of flowers, roses, tulip and sunflower but if we test it over not so clear, properly scales, or rotated picture it may not be so accurate

This problem is handled by data augmentation

For example

We generate 4 samples by one and thus by doing so we make our model more robust and increase its efficiency



How do we do it

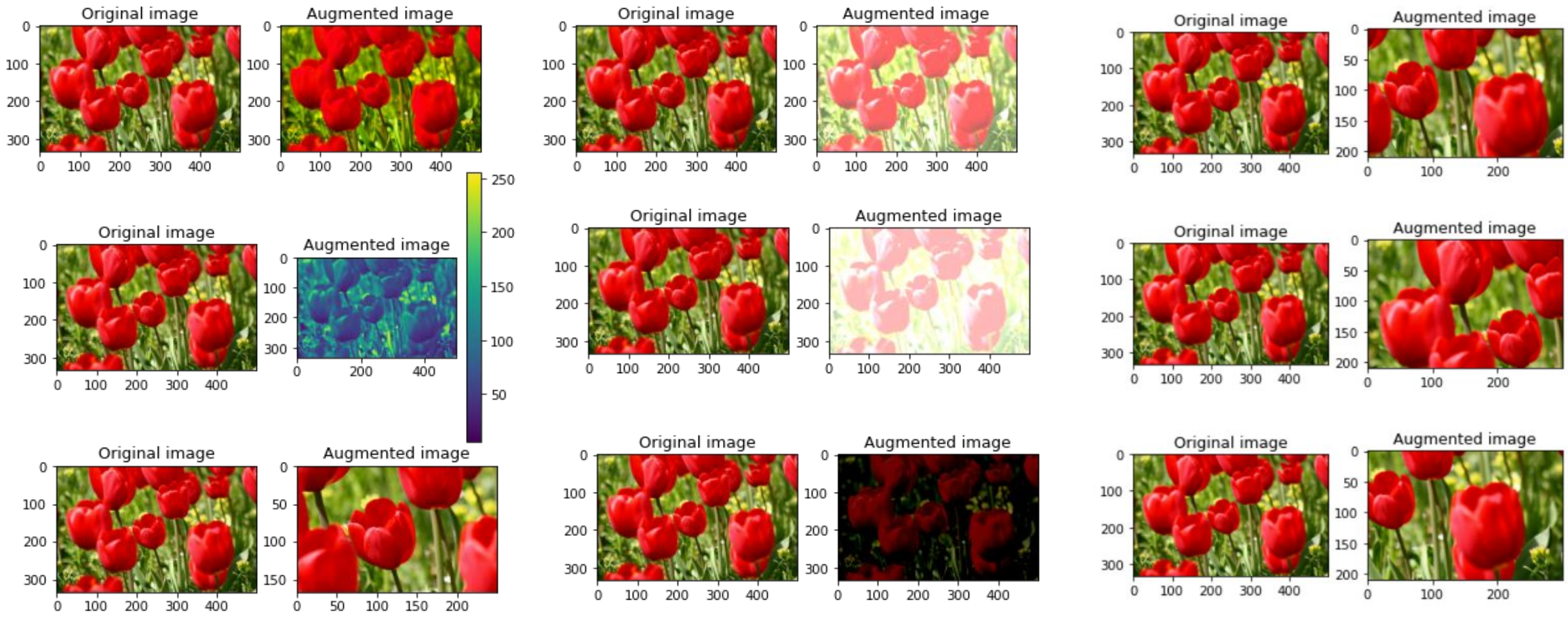
- Use the Keras preprocessing layers, such as ***tf.keras.layers.Resizing, tf.keras.layers.Rescaling, tf.keras.layers.RandomFlip, and tf.keras.layers.RandomRotation.***

- Use the tf.image methods, such as ***tf.image.flip_left_right, tf.image.rgb_to_grayscale, tf.image.adjust_brightness, tf.image.central_crop, and tf.image.stateless_random*.***

Example

```
resize_and_rescale = tf.keras.Sequential([
  layers.Resizing(IMG_SIZE, IMG_SIZE),
  layers.Rescaling(1./255)
])
```

Example


Original image
Augmented image
Original image
Augmented image
Original image
Augmented image
Original image
Augmented image
Original image
Augmented image
Original image
Augmented image
Original image
Augmented image
Original image
Augmented image
Original image
Augmented image