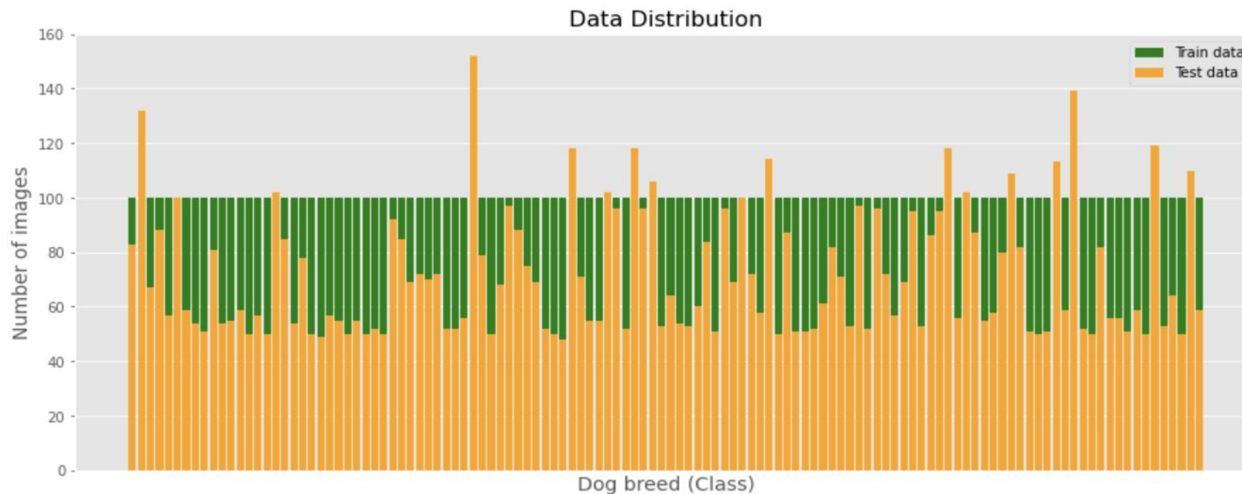
Classifying Dog Breeds with Convolutional Neural Networks

Group 23: Ioannis, Kimi, Linea, Maja and Samy

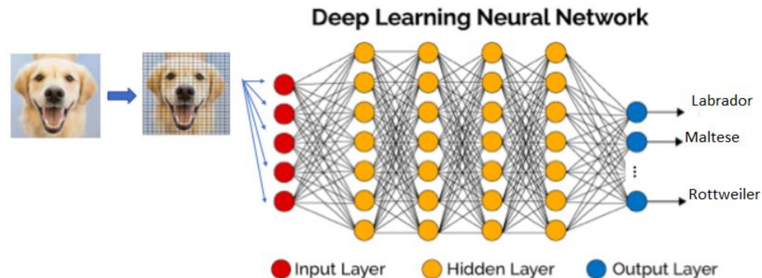
UNIVERSITY OF
COPENHAGEN

The Dataset

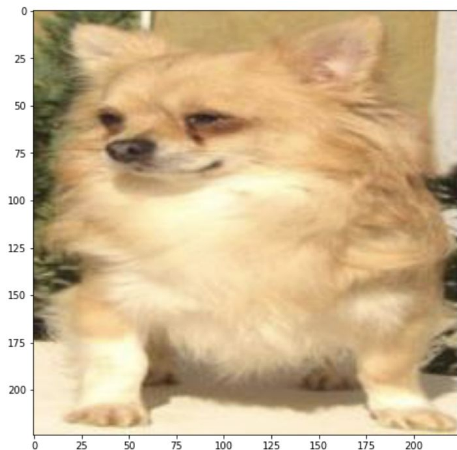
- 120 different dog breeds
  - Train set: 12000 images
  - Test set: 8580 images



Preprocessing

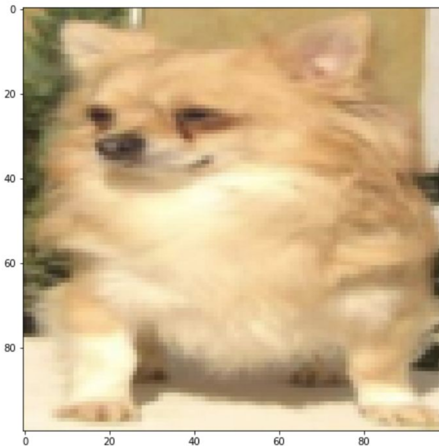


Original image:
224 x 224 x 3

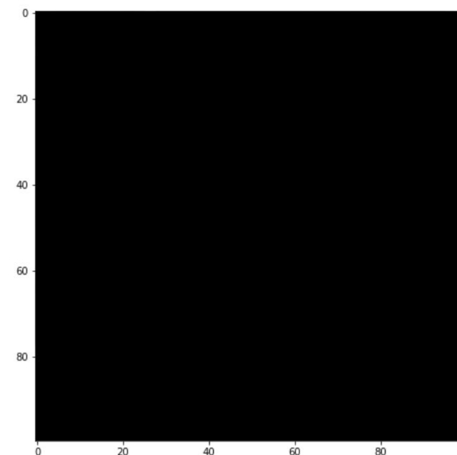


Resizing

Pixelated image:
100 x 100 x 3



Rescaling

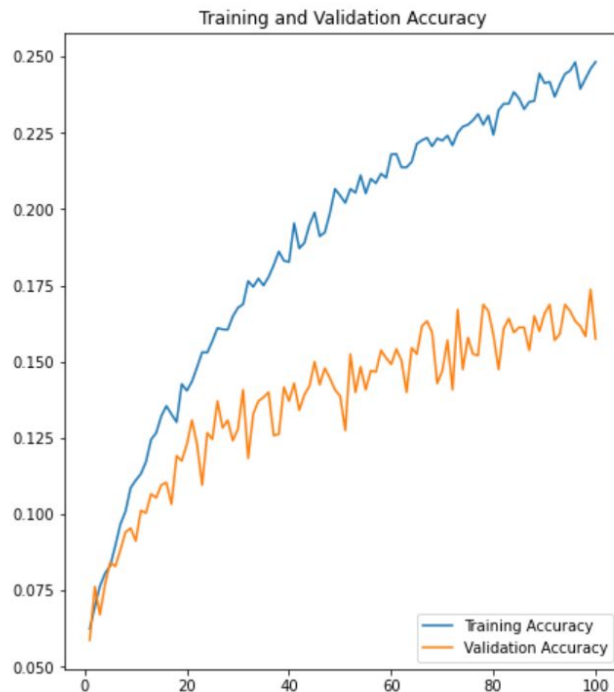


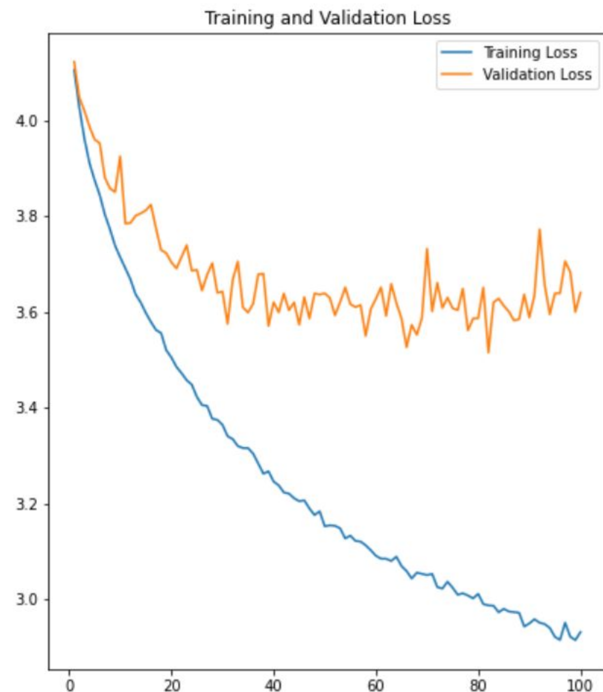
Deep Learning with Tensorflow

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 224, 224, 16)      448
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 112, 112, 16)      0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 112, 112, 32)      4640
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 56, 56, 32)        0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 56, 56, 64)        18496
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 28, 28, 64)        0
_________________________________________________________________
dropout (Dropout)            (None, 28, 28, 64)        0
_________________________________________________________________
flatten (Flatten)            (None, 50176)             0
_________________________________________________________________
dense (Dense)                (None, 128)               6422656
_________________________________________________________________
dense_1 (Dense)              (None, 120)               15480
=================================================================
Total params: 6,461,720
Trainable params: 6,461,720
Non-trainable params: 0
_________________________________________________________________
```

First model attempt ...

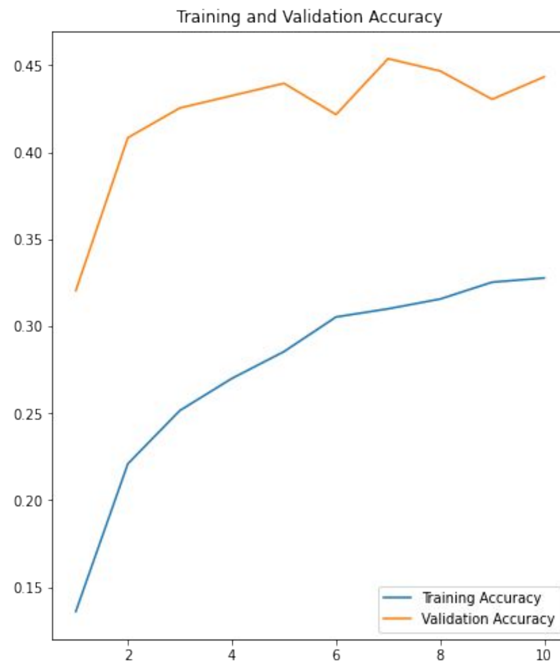
....was crap/neither deep nor learning

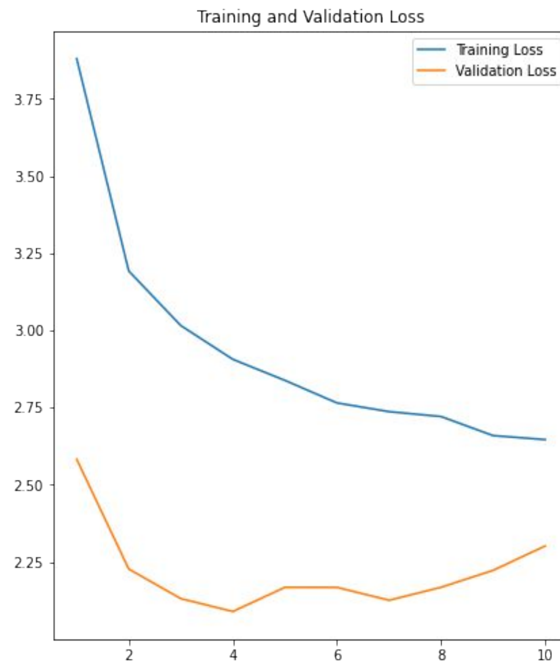




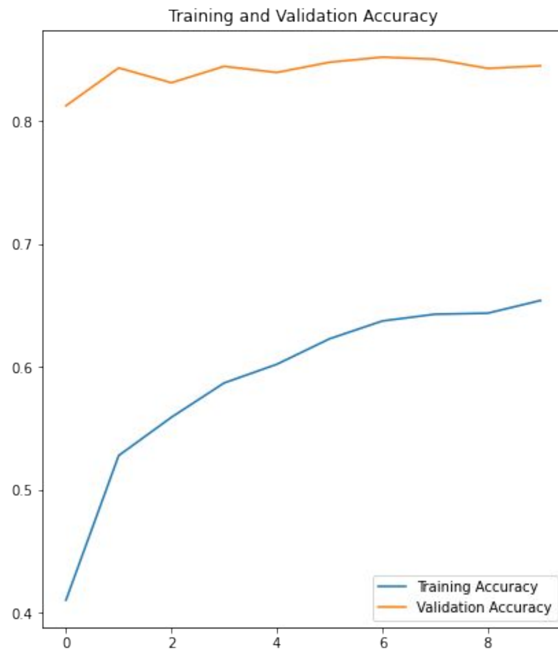
New attempt using pretrained Tensorflow model **Xception** on **100x100** cropped images:

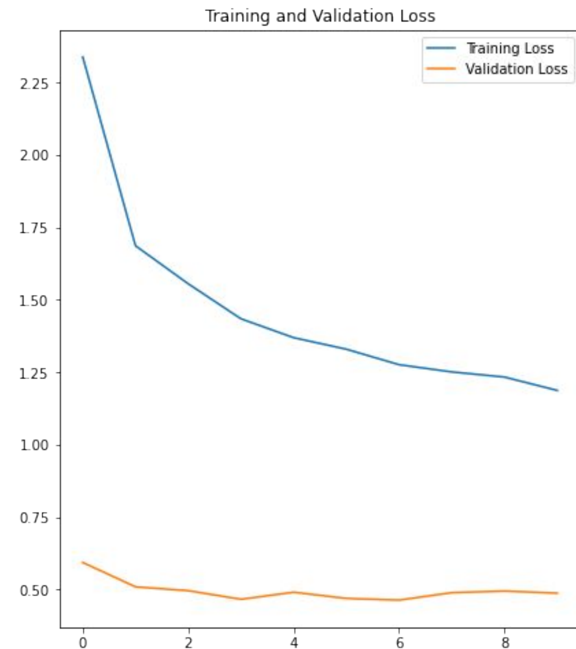
Xception is a convolutional neural network that is **71 layers deep**. You can load a pretrained version of the network trained on more than a million images from the ImageNet database





New attempt using pretrained Tensorflow model Xception on **224x224** cropped images:





```
(fc): Linear(in_features=512,
out_features=1000, bias=True)
```

UNIVERSITY OF COPENHAGEN

Deep Learning with Pytorch

Pretrained model Resnet18

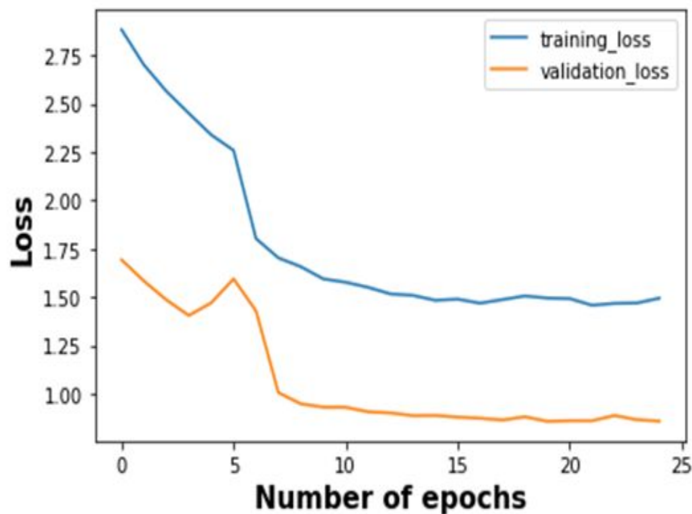
```
ResNet(

(conv1): Conv2d(3, 64, kernel_size=(7, 7),
stride=(2, 2), padding=(3, 3), bias=False)

(bn1):BatchNorm2d(64, eps=1e-05, momentum=0.1,
affine=True, track_running_stats=True)

(relu): ReLU(inplace=True)

(maxpool): MaxPool2d(kernel_size=3, stride=2,
padding=1, dilation=1, ceil_mode=False)
.
.
(fc): Linear(in_features=512,out_features=1000,
bias=True)
```

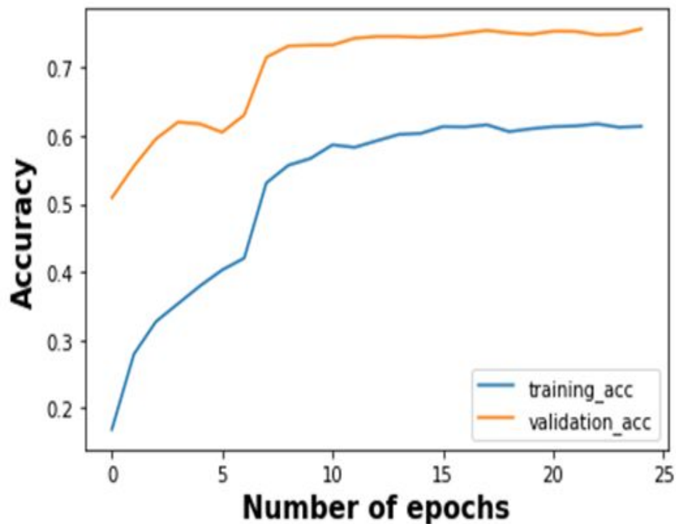
Libraries

- torchvision.transforms – performing transformation on image data
- torch.nn – defining the neural network
- torch.nn.functional – importing functions like ReLU
- torch.optim – implementing optimization algorithms such as Stochastic Gradient Descent (SGD)

ResNet-18 is a convolutional neural network that is 18 **layers** deep.

PyTorch






```
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(net.parameters(), lr=0.001, momentum=0.9)
```

```
model_ft = train_model(model_ft, criterion, optimizer_ft, exp_lr_scheduler, num_epochs=25)
```

Hyper Parameter Optimization - or well..

Surely we can optimize our models! Right?

skorch for PyTorch provides compatibility with sklearn - great!

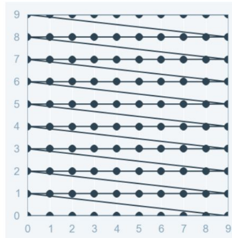
TensorFlow already integrated with sklearn - great!

- Random guess: 0.83 % chance of getting the breed correct
- Initial best with Tensorflow: ~13 %
- Initial best with PyTorch: ~71 %
- Best on Kaggle: 99.99 % correct

GridSearchCV and Grad Student Descent

- Test for best learning rate and max epochs with GridSearchCV

```
params = {
        'lr': [0.0001, 0.001, 0.1],
        'max_epochs': [5, 10, 15, 20, 25, 30]}
```



- GridSearchCV didn't work - manual approach instead
- Change Dropout, Learning Rate, Epochs
- Test amount and size of layers manually - where possible

And they got better

- Random guess: 0.83 % chance of getting the breed correct
- Final best with Tensorflow: ~86 %
- Final best with PyTorch: ~75 %
- Best on Kaggle: 99.99 % correct

Tensorflow was initially.. Not great, but turned out best

With more knowledge on data handling we can be as good as the Kagglers

UNIVERSITY OF
COPENHAGEN

Generative adversarial network

Real
images

Generator

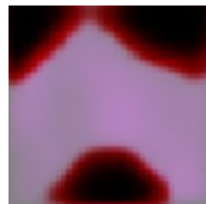
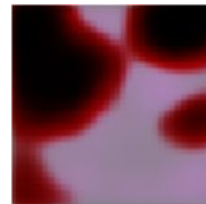
Generated
images

Discriminator

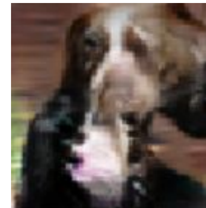
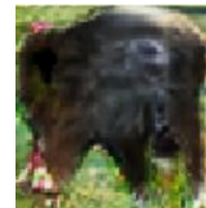
Is it real?

GAN results

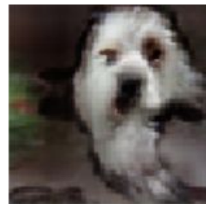
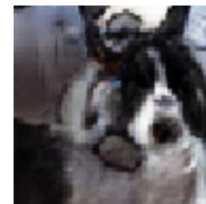
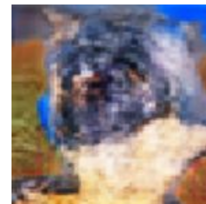
Epoch 0

Epoch 100

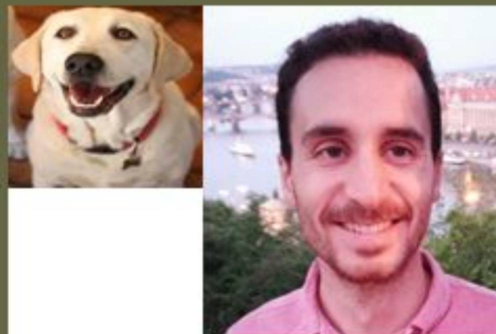
Epoch 200

Final Thoughts and Where to Go Next

- Tune hyperparameters
- Use different optimizers
- Image data augmentation
- Try more complex architectures such as the state of the art models of ImageNet
- Deal with overfitting
- Find more data

n02099712-Labrador_retriever



n02099712-Labrador_retriever

n02111500-Great_Pyrenees

n02111500-Great_Pyrenees

n02099712-Labrador_retriever

n02085936-Maltese_dog

APPENDIX

GAN

Inspiration for the GAN was this following kaggle competition:
https://www.kaggle.com/c/generative-dog-images

Discriminator model: Sequential keras model with 4 convolutions

Generator model: Sequential keras model with 4 convolutions as well as upsampling

Overall model: also uses weight normalisation

APPENDIX

For Pytorch approach:

```
data_transforms = {
    'train': transforms.Compose([
        transforms.RandomResizedCrop(224),
        transforms.RandomHorizontalFlip(),
        transforms.ToTensor(),
        transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
    ]),
    'test': transforms.Compose([
        transforms.Resize(256),
        transforms.CenterCrop(224),
        transforms.ToTensor(),
        transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
    ]),
}

data_dir = '/content/drive/MyDrive/Colab Notebooks/cropped/'
image_datasets = {x: datasets.ImageFolder(os.path.join(data_dir, x),
                                          data_transforms[x])
                  for x in ['train', 'test']}
dataloaders = {x: torch.utils.data.DataLoader(image_datasets[x], batch_size=4,
                                             shuffle=True, num_workers=4)
              for x in ['train', 'test']}
dataset_sizes = {x: len(image_datasets[x]) for x in ['train', 'test']}
class_names = image_datasets['train'].classes
```

Figure 1 : Defining and transforming the data.

All pre-trained models expect input images normalized in the same way.

APPENDIX

For Pytorch approach:

1. **ToTensor()** - converts the images into tesnors to be used with torch library
2. **Normalize (mean, std) -** The number of parameters we pass into the mean and std arguments depends on the modes of our images, i.e. for an RGB image we pass 3 parameters for both the mean and std
3. To normalize a dataset using standardization, we take every value **x** inside the dataset and transform it to its corresponding **z** value using the following formula:

$$z = \frac{x - mean}{std}$$

4. In NN in general we normalize to help the CNN perform better as it helps get data within a range and reduces the skewness since it centered around 0. This helps it learn faster and better.
5. The value **num_workers** allows pytorch to perform multi-process data loading. In our code we set 4 as the number of workers. This means that there are 4 workers simultaneously putting data into the computer’s RAM.

Now we are ready to define and load our train and test data.

1. We have used **datasets.ImageFolder** to upload the images and then **dataloaders** to pass our arguments
2. The name of the classes are followingly defined as **image_datasets.classes**

APPENDIX

Input image

```
imsize = 256
loader = transforms.Compose([transforms.Scale(imsize), transforms.ToTensor()])

def image_loader(image_name):
    """load image, returns cuda tensor"""
    image = Image.open(image_name)
    image = loader(image).float()
    image = Variable(image, requires_grad=True)
    image = image.unsqueeze(0)  #this is for VGG, may not be needed for ResNet
    return image.cuda()  #assumes that you're using GPU

image = image_loader('/content/drive/MyDrive/Colab Notebooks/cropped/labra.jpg')

outputs = model_ft(image)

_, preds = torch.max(outputs, 1)
print(class_names[preds[0]])
```

```
n02099712-Labrador retriever
```

Figure 2 : Pytorch. Predicting new images.

APPENDIX

Explaining the different layers - How Conv2d works:

This applies a 2D convolution and we turn several channels into feature/activation maps. Arguments: in_channels, out_channels, kernel_size:

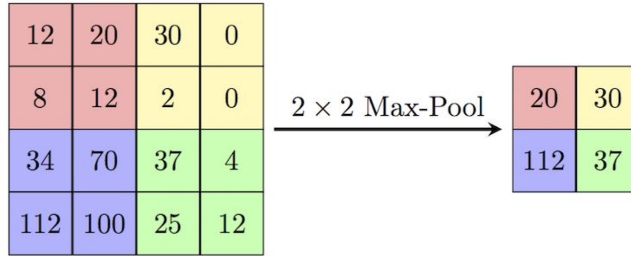
- in_channels = 3 (because our images are RGB)
- out_channels = 64
- kernel_size = 7 (that means that our square convolutional kernel is 7x7). Kernels are basically filters that act as feature detectors from the original input image. This filter moves around the image, detects the features, and produces the feature maps.

Example: Input Shape : (3, 9, 9) — Output Shape : (2, 3, 3) — K : (3, 3) — P : (1, 1) — S : (2, 2) — D : (2, 2) — G : 1
From:
https://towardsdatascience.com/conv2d-to-finally-understand-what-happens-in-the-forward-pass-1bbaafb0b148

APPENDIX

Explaining the different layers - How MaxPool works:

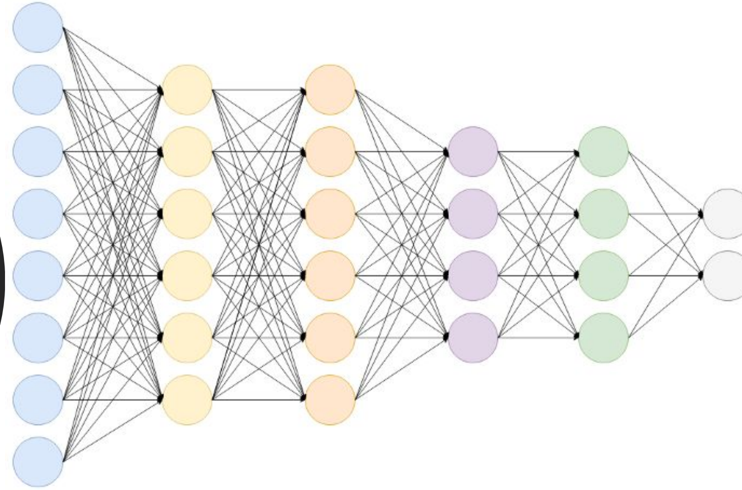
The main purpose of the Max Pooling is to down - sample the dimensions of our image to allow for assumptions to be made about the features in certain regions of the image. Long story short it reduces the dimensionality of the image keeping the important features



From:https://medium.com/bitgrit-data-science-publication/building-an-image-classification-model-with-pytorch-from-scratch-f10452073212

APPENDIX

Explaining the different layers - How fc - Fully Connected works:



From:
https://medium.com/bitgrit-data-science-publication/building-an-image-classification-model-with-pytorch-from-scratch-f10452073212

FC layers means that every neuron from the previous layers connects to all neurons in the next.

A good way to think about the fc layers is to use the concept of PCA principal component analysis that selects the good features among the feature space