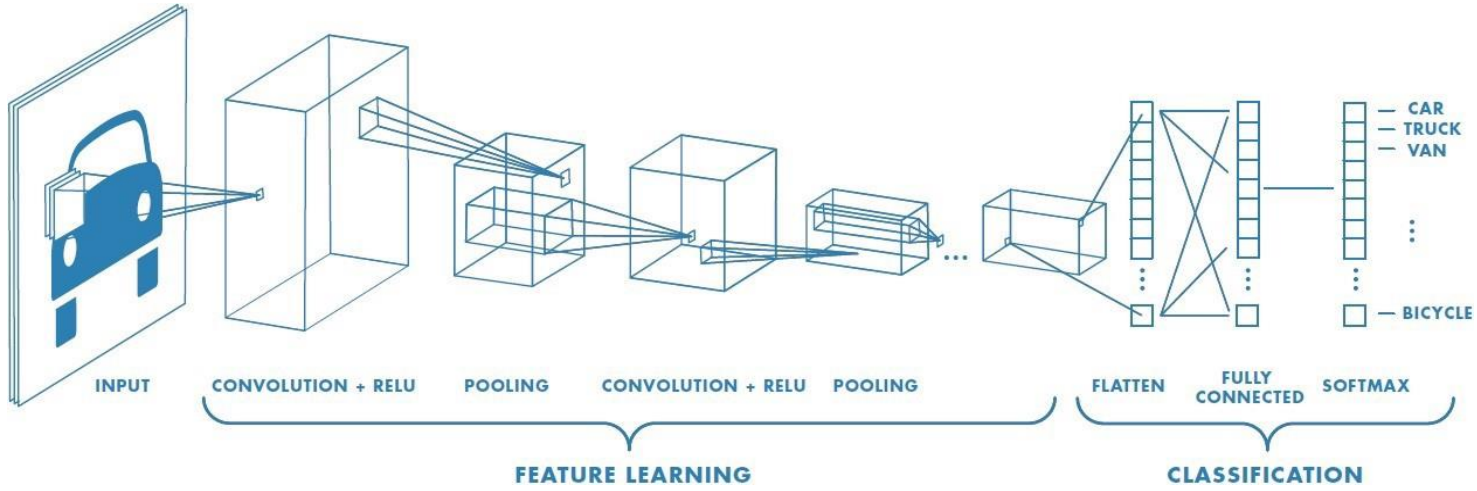
I
Convolutional
Neural Networks

Data Science Academy

What is CNN?

- Convolutional Neural Network (ConvNet/CNN) is a Deep Learning algorithm which can take in an input image, assign importance (learnable weights and biases) to various aspects/objects in the image and be able to differentiate one from the other.
- The architecture of a ConvNet is analogous to that of the connectivity pattern of Neurons in the Human Brain and was inspired by the organization of the Visual Cortex.
- CNN's are best known for their ability to recognize patterns present in images, and so the task chosen for the network described in this post was that of image classification.

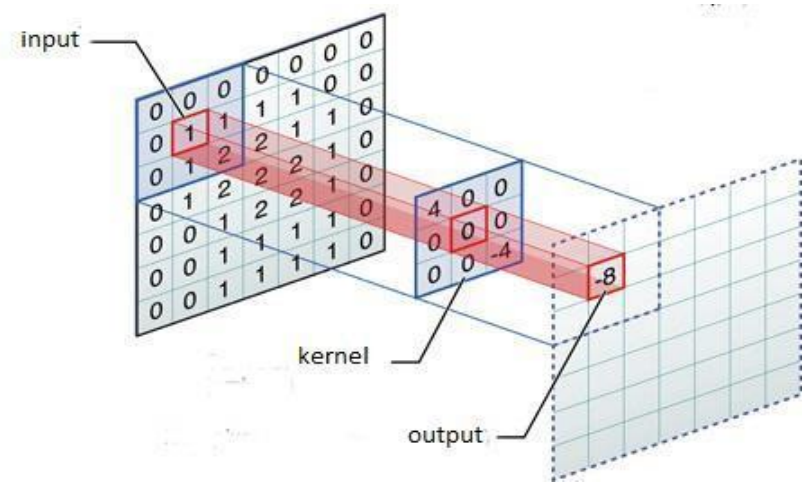


Bütün hüquqlar qorunur.

How Convolutional Neural Networks learn?

Convolutions

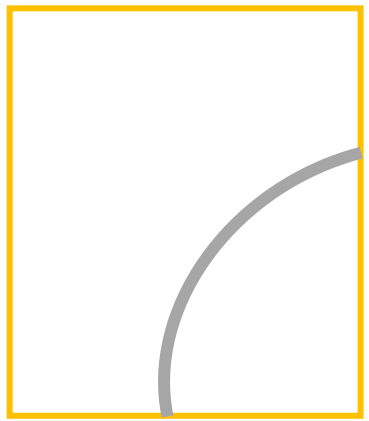
- CNN's make use of filters (also known as kernels), to detect what features, such as edges, are present throughout an image.
- A filter is just a matrix of values, called weights, that are trained to detect specific features. The filter moves over each part of the image to check if the feature it is meant to detect is present.
- To provide a value representing how confident it is that a specific feature is present, the filter carries out a convolution operation, which is an element-wise product and sum between two matrices.



Bütün hüquqlar qorunur.

Feature of an image

- When the feature is present in part of an image, the convolution operation between the filter and that part of the image results in a real number with a high value. If the feature is not present, the resulting value is low.
- In the following example, a filter that is in charge of checking for right-hand curves is passed over a part of the image.
- Since that part of the image contains the same curve that the filter is looking for, the result of the convolution operation is a large number (6600).



Visualization of the receptive field

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

Pixel representation of the receptive field

$*$

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

Pixel representation of filter

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

Bütün hüquqlar qorunur.

Feature of an image

- But when that same filter is passed over a part of the image with a considerably different set of edges, the convolution's output is small, meaning that there was no strong presence of a right hand curve.

Visualization of the filter on the image

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

Pixel representation of the receptive field

*

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

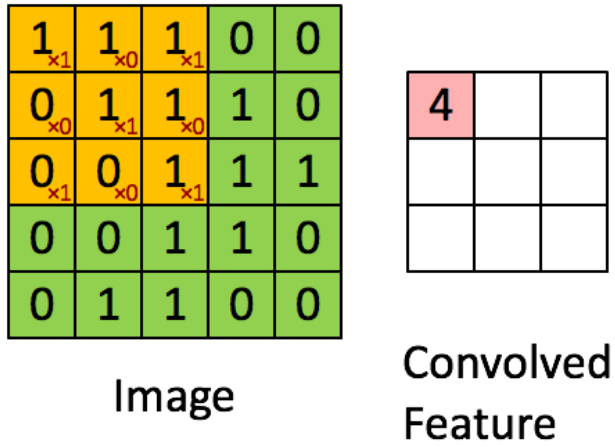
Pixel representation of filter

Multiplication and Summation= 0

Bütün hüquqlar qorunur.

Output Matrix

- The result of passing this filter over the entire image is an output matrix that stores the convolutions of this filter over various parts of the image.
- The filter must have the same number of channels as the input image so that the element-wise multiplication can take place.
- For instance, if the input image contains three channels (RGB, for example), then the filter must contain three channels as well.
- The convolution of a filter over a 2D image:



Bütün hüquqlar qorunur.

Stride Value

- A filter can be slid over the input image at varying intervals, using a stride value.
- The stride value dictates by how much the filter should move at each step.
- The output dimensions of a strided convolution can be calculated using the following equation:

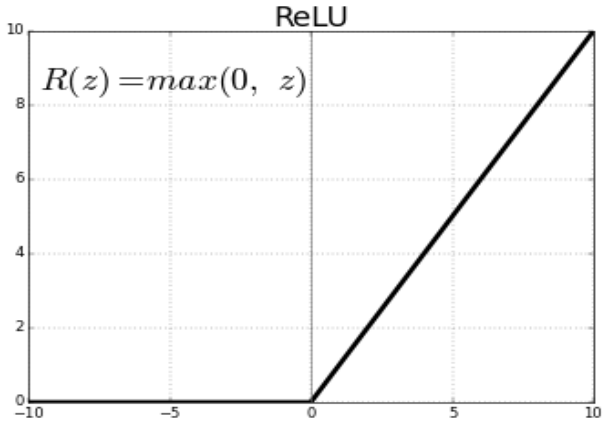
$$n_{out} = floor(\frac{n_{in} - f}{s}) + 1$$

- Where n_in denotes the dimension of the input image, f denotes the window size, and s denotes the stride.

Bütün hüquqlar qorunur.

Non-linear activation function

- Convolutional Neural Network can learn the values for a filter that detect features present in the input data, the filter must be passed through a non-linear mapping.
- The output of the convolution operation between the filter and the input image is summed with a bias term and passed through a non-linear activation function.
- The purpose of the activation function is to introduce non-linearity into our network.
- As we see, the ReLU function is quite simple; values that are less than or equal to zero become zero and all positive values remain the same.



- Usually, a network utilizes more than one filter per layer. When that is the case, the outputs of each filter's convolution over the input image are concatenated along the last axis, forming a final 3D output.

Bütün hüquqlar qorunur.

NumPy Implementation of CNN

- Using NumPy, we can program the convolution operation quite easily.
- The convolution function makes use of a for-loop to convolve all the filters over the image.
- Within each iteration of the for-loop, two while-loops are used to pass the filter over the image.
- At each step, the filter is multipled element-wise (*) with a section of the input image.
- The result of this element-wise multiplication is then summed to obtain a single value using NumPy's sum method, and then added with a bias term.

```
def convolution(image, filt, bias, s=1):
    (n_f, n_c_f, f, _) = filt.shape
    n_c, in_dim, _ = image.shape

    out_dim = int((in_dim - f)/s)+1
    out = np.zeros((n_f,out_dim,out_dim))
    for curr_f in range(n_f):
        curr_y = out_y = 0
        while curr_y + f <= in_dim:
            curr_x = out_x = 0
            while curr_x + f <= in_dim:
                out[curr_f, out_y, out_x] = np.sum(filt[curr_f] * \
                image[:,curr_y:curr_y+f, curr_x:curr_x+f]) + bias[curr_f]
                curr_x += s
                out_x += 1
            curr_y += s
            out_y += 1

    return out
```


Bütün hüquqlar qorunur.

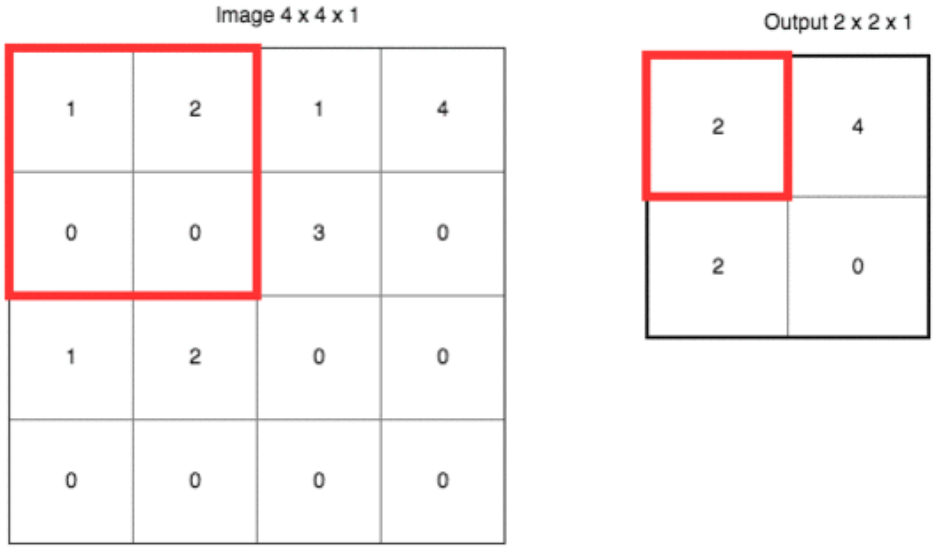
Downsampling

- After one or two convolutional layers, it is common to reduce the size of the representation produced by the convolutional layer.
- This reduction in the representation's size is known as downsampling.
- To speed up the training process and reduce the amount of memory consumed by the network, we try to reduce the redundancy present in the input feature.
- There are a couple of ways we can downsample an image, we will look at the most common one: max pooling.

Bütün hüquqlar qorunur.

Max Pooling

- In max pooling, a window passes over an image according to a set stride (how many units to move on each pass).
- At each step, the maximum value within the window is pooled into an output matrix, hence the name max pooling.
- In the following visual, a window of size f=2 passes over an image with a stride of 2.
  - f denotes the dimensions of the max pooling window (red box) and s denotes the number of units the window moves in the x and y-direction.
  - At each step, the maximum value within the window is chosen:



Bütün hüquqlar qorunur.

Max Pooling

- Max pooling significantly reduces the representation size, in turn reducing the amount of memory required and the number of operations performed later in the network.
- The output size of the max pooling operation can be calculated using the following equation:

$$n_{out} = floor(\frac{n_{in} - f}{s}) + 1$$

- Where n_in denotes the dimension of the input image, f denotes the window size, and s denotes the stride.
- An added benefit of max pooling is that it forces the network to focus on a few neurons instead of all of them which has a regularizing effect on the network, making it less likely to overfit the training data and hopefully generalize well.

Bütün hüquqlar qorunur.

NumPy Implementation of Max Pooling

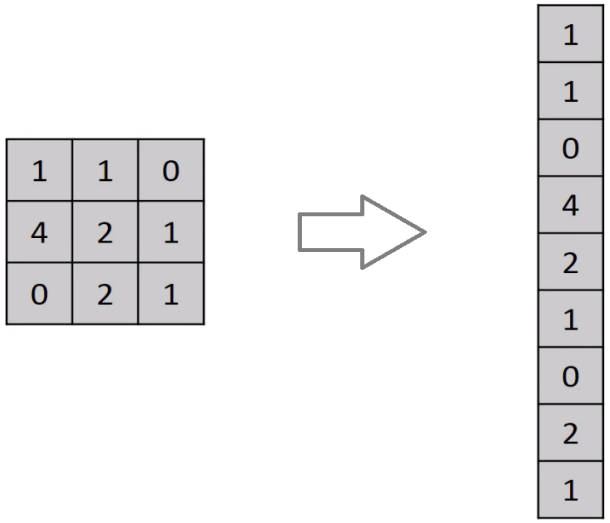
- The max pooling operation boils down to a for loop and a couple of while loops.
- The for-loop is used pass through each layer of the input image, and the while-loops slide the window over every part of the image.
- At each step, we use NumPy's max method to obtain the maximum value.

```
def maxpool(image, f=2, s=2):
    ```
    Downsample input `image` using a kernel size of `f` and a stride of `s`
    ```
    n_c, h_prev, w_prev = image.shape

    # calculate output dimensions after the maxpooling operation.
    h = int((h_prev - f)/s)+1
    w = int((w_prev - f)/s)+1

    # create a matrix to hold the values of the maxpooling operation.
    downsampled = np.zeros((n_c, h, w))

    # slide the window over every part of the image using stride s.
    # Take the maximum value at each step.
    for i in range(n_c):
        curr_y = out_y = 0
        # slide the max pooling window vertically across the image
        while curr_y + f <= h_prev:
            curr_x = out_x = 0
            # slide the max pooling window horizontally across the image
            while curr_x + f <= w_prev:
                # choose the maximum value within the window at
                # each step and store it to the output matrix
                downsampled[i, out_y, out_x] = np.max(image[i, curr_y:curr_y+f, curr_x:curr_x+f])
                curr_x += s
                out_x += 1
            curr_y += s
            out_y += 1
    return downsampled
```


Bütün hüquqlar qorunur.

Fully-Connected Layer

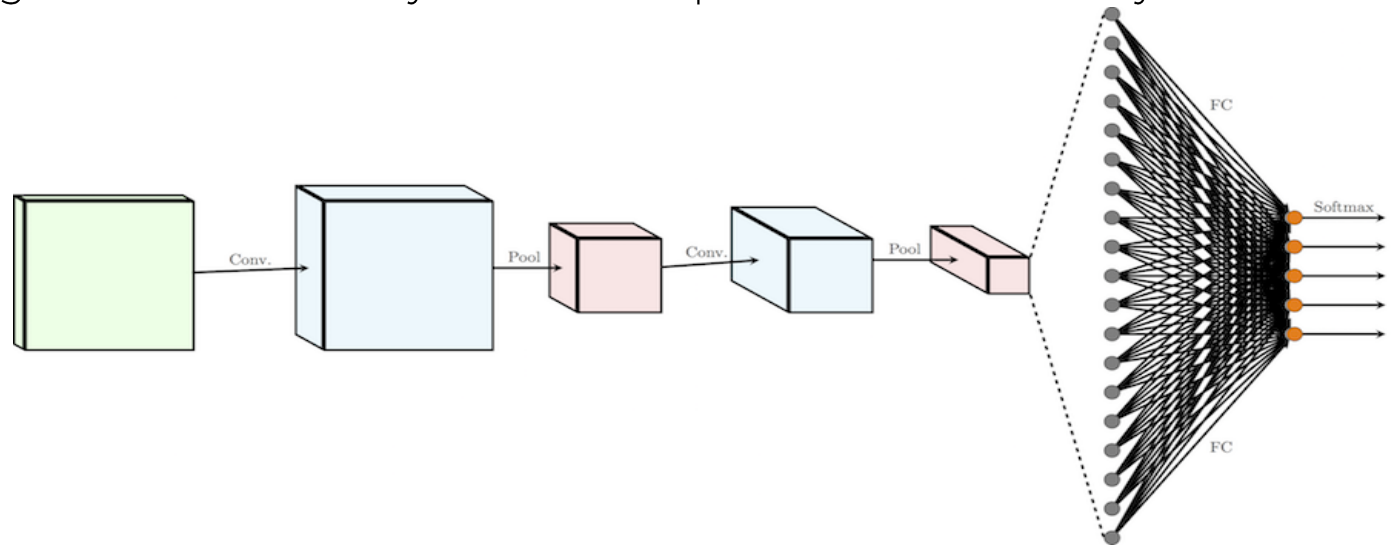
- After multiple convolutional layers and downsampling operations, the 3D image representation is converted into a feature vector that is passed into a Multi-Layer Perceptron, which merely is a neural network with at least three layers.
- This is referred to as a Fully-Connected Layer.
- In the fully-connected operation of a neural network, the input representation is flattened into a feature vector and passed through a network of neurons to predict the output probabilities.
- The following image describes the flattening operation:



Bütün hüquqlar qorunur.

Fully-Connected Layer

- The rows are concatenated to form a long feature vector. If multiple input layers are present, its rows are also concatenated to form an even longer feature vector.
- The feature vector is then passed through multiple dense layers. At each dense layer, the feature vector is multiplied by the layer's weights, summed with its biases, and passed through a non-linearity.
- The following image visualizes the fully connected operation and dense layers:



Bütün hüquqlar qorunur.

NumPy Implementation of Fully-Connected Layer

- NumPy makes it quite simple to program the fully connected layer of a CNN.
- As a matter of fact, you can do it in a single line of code using NumPy's reshape method:
- In this code snippet, we gather the dimensions of the previous layer (number of channels and height/width) then use them to flatten the previous layer into a fully connected layer.
- This fully connected layer is proceeded by multiple dense layers of neurons that eventually produce raw predictions:

```
(nf2, dim2, _) = pooled.shape

# flatten pooled layer
fc = pooled.reshape((nf2 * dim2 * dim2, 1))
```

```
# first dense layer
z = w3.dot(fc) + b3

# pass through ReLU non-linearity
z[z<=0] = 0

# second dense layer
out = w4.dot(z) + b4
```


Bütün hüquqlar qorunur.

Output Layer

- The output layer of a CNN is in charge of producing the probability of each class (each digit) given the input image.
- To obtain these probabilities, we initialize our final Dense layer to contain the same number of neurons as there are classes.
- The output of this dense layer then passes through the Softmax activation function, which maps all the final dense layer outputs to a vector whose elements sum up to one:

$$\sigma(x_j) = \frac{e^{x_j}}{\sum_i e^{x_i}}$$

- Where x denotes each element in the final layer's outputs.
- Once again, the softmax function can be written in a few lines in NumPy:

```
def softmax(raw_preds):
    # pass raw predictions through softmax activation function
    out = np.exp(raw_preds) # exponentiate vector of raw predictions
    return out/np.sum(out) # divide the exponentiated vector by its sum. All values in the output sum to 1.
```


Bütün hüquqlar qorunur.

Calculating the Loss

- To measure how accurate our network was in predicting the handwritten digit from the input image, we make use of a loss function.
- The loss function assigns a real-valued number to define the model's accuracy when predicting the output digit.
- A common loss function to use when predicting multiple output classes is the Categorical Cross-Entropy Loss function, defined as follows:

$$H(y, \hat{y}) = \sum_i y_i \log \frac{1}{\hat{y}_i} = -\sum_i y_i \log \hat{y}_i$$

- Here, ŷ is the CNN's prediction, and y is the desired output label. When making predictions over multiple examples, we take the average of the loss over all examples.

Bütün hüquqlar qorunur.

NumPy Implementation of Categorical Cross-Entropy loss function

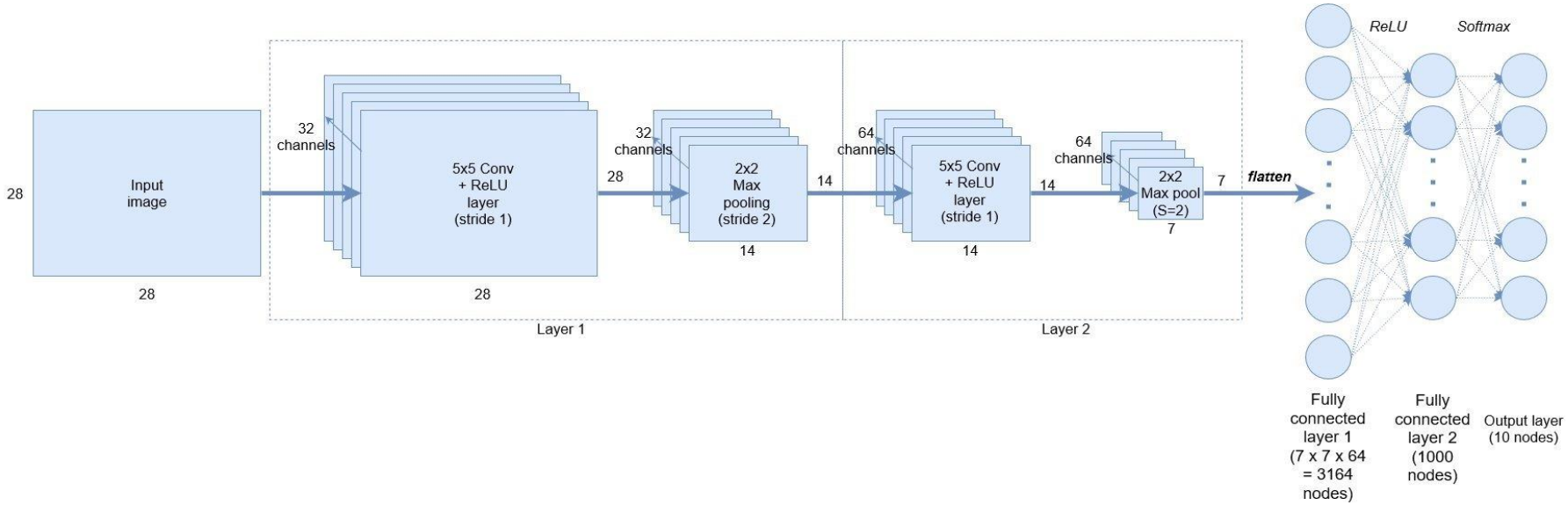
- The Categorical Cross-Entropy loss function can be easily programmed using two simple lines of code, which are a mirror of the equation shown in last slide:

```
def categoricalCrossEntropy(probs, label):
    # calculate the categorical cross-entropy loss of the predictions
    # Multiply the desired output label by the log of the prediction,
    # then sum all values in the vector
    return -np.sum(label * np.log(probs))
```


Bütün hüquqlar qorunur.

Implementing Convolutional Neural Networks in PyTorch

- The network we're going to build will perform MNIST digit classification
- The Convolutional Neural Network architecture that we are going to build can be seen in the diagram below:



Bütün hüquqlar qorunur.

The CNN architecture

- First up, we can see that the input images will be 28 x 28 pixel greyscale representations of digits.
- The first layer will consist of 32 channels of 5 x 5 convolutional filters + a ReLU activation, followed by 2 x 2 max pooling down-sampling with a stride of 2 (this gives a 14 x 14 output).
- In the next layer, we have the 14 x 14 output of layer 1 being scanned again with 64 channels of 5 x 5 convolutional filters and a final 2 x 2 max pooling (stride = 2) down-sampling to produce a 7 x 7 output of layer 2.
- After the convolutional part of the network, there will be a flatten operation which creates 7 x 7 x 64 = 3164 nodes, an intermediate layer of 1000 fully connected nodes and a softmax operation over the 10 output nodes to produce class probabilities. These layers represent the output classifier.

Bütün hüquqlar qorunur.

Loading the dataset

- PyTorch has an integrated MNIST dataset (in the torchvision package) which we can use via the DataLoader functionality.

```
# Hyperparameters
num_epochs = 5
num_classes = 10
batch_size = 100
learning_rate = 0.001

DATA_PATH = 'C:\\Users\Eljan\MyProjects\MNISTData'
MODEL_STORE_PATH = 'C:\\Users\Andy\MyProjects\pytorch_models\\'
```

- Next, we setup a transform to apply to the MNIST data, and also the data set variables:

```
# transforms to apply to the data
trans = transforms.Compose([transforms.ToTensor(), transforms.Normalize((0.1307,), (0.3081,))])

# MNIST dataset
train_dataset = torchvision.datasets.MNIST(root=DATA_PATH, train=True, transform=trans, download=True)
test_dataset = torchvision.datasets.MNIST(root=DATA_PATH, train=False, transform=trans)
```


Bütün hüquqlar qorunur.

transforms.Compose() function

- The first thing to note above is the transforms.Compose() function.
- This function comes from the torchvision package. It allows the developer to setup various manipulations on the specified dataset.
- Numerous transforms can be chained together in a list using the Compose() function. In this case, first we specify a transform which converts the input data set to a PyTorch tensor.
- A PyTorch tensor is a specific data type used in PyTorch for all of the various data and weight operations within the network.
- In its essence though, it is simply a multi-dimensional matrix. In any case, PyTorch requires the data set to be transformed into a tensor so it can be consumed in the training and testing of the network.

Bütün hüquqlar qorunur.

DataLoader

- Next, the train_dataset and test_dataset objects need to be created. These will subsequently be passed to the data loader.
- In order to create these data sets from the MNIST data, we need to provide a few arguments.
- First, the root argument specifies the folder where the train.pt and test.pt data files exist.
- The train argument is a boolean which informs the data set to pickup either the train.pt data file or the test.pt data file.
- The next argument, transform, is where we supply any transform object that we've created to apply to the data set – here we supply the trans object which was created earlier.
- Finally, the download argument tells the MNIST data set function to download the data (if required) from an online source.

```
train_loader = DataLoader(dataset=train_dataset, batch_size=batch_size, shuffle=True)
test_loader = DataLoader(dataset=test_dataset, batch_size=batch_size, shuffle=False)
```


Bütün hüquqlar qorunur.

Creating the model

- Given the relatively low amount of classes (10 in total) and the small size of each training image (28x28px.), a simple network architecture was chosen to solve the task of digit recognition.
- The network uses two consecutive convolutional layers followed by a max pooling operation to extract features from the input image.
- After the max pooling operation, the representation is flattened and passed through a Multi-Layer Perceptron (MLP) to carry out the task of classification.

```
class ConvNet(nn.Module):
    def __init__(self):
        super(ConvNet, self).__init__()
        self.layer1 = nn.Sequential(
            nn.Conv2d(1, 32, kernel_size=5, stride=1, padding=2),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2))
        self.layer2 = nn.Sequential(
            nn.Conv2d(32, 64, kernel_size=5, stride=1, padding=2),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2))
        self.drop_out = nn.Dropout()
        self.fc1 = nn.Linear(7 * 7 * 64, 1000)
        self.fc2 = nn.Linear(1000, 10)
```


Bütün hüquqlar qorunur.

The nn.Module

- The nn.Module is a very useful PyTorch class which contains all you need to construct your typical deep learning networks.
- The first step is to create some sequential layer objects within the class _init_ function.
- First, we create layer 1 (self.layer1) by creating a nn.Sequential object.
- This method allows us to create sequentially ordered layers in our network and is a handy way of creating a convolution + ReLU + pooling sequence.

Bütün hüquqlar qorunur.

Conv2d

- As can be observed, the first element in the sequential definition is the Conv2d nn.Module method – this method creates a set of convolutional filters.
- The first argument is the number of input channels – in this case, it is our single channel grayscale MNIST images, so the argument is 1.
- The second argument to Conv2d is the number of output channels – as shown in the model architecture diagram above, the first convolutional filter layer comprises of 32 channels, so this is the value of our second argument.
- The kernel_size argument is the size of the convolutional filter – in this case we want 5 x 5 sized convolutional filters – so the argument is 5.

Bütün hüquqlar qorunur.

MaxPool2d

- The last element that is added in the sequential definition for self.layer1 is the max pooling operation.
- The first argument is the pooling size, which is 2 x 2 and hence the argument is 2.
- Second – we want to down-sample our data by reducing the effective image size by a factor of 2. To do this, using the formula above, we set the stride to 2 and the padding to zero. Therefore, the stride argument is equal to 2.
- The padding argument defaults to 0 if we don't specify it – so that's what is done in the code above. From these calculations, we now know that the output from self.layer1 will be 32 channels of 14 x 14 "images".

Bütün hüquqlar qorunur.

nn.Dropout and nn.Linear

- Next, the second layer, self.layer2, is defined in the same way as the first layer. The only difference is that the input into the Conv2d function is now 32 channels, with an output of 64 channels. Using the same logic, and given the pooling down-sampling, the output from self.layer2 is 64 channels of 7 x 7 images.
- Next, we specify a drop-out layer to avoid over-fitting in the model.
- Finally, two two fully connected layers are created. The first layer will be of size 7 x 7 x 64 nodes and will connect to the second layer of 1000 nodes.
- To create a fully connected layer in PyTorch, we use the nn.Linear method.
- The first argument to this method is the number of nodes in the layer, and the second argument is the number of nodes in the following layer.

Bütün hüquqlar qorunur.

Forward Function

- It is important to call this function “forward” as this will override the base forward function in nn.Module and allow all the nn.Module functionality to work correctly.
- As can be observed, it takes an input argument x, which is the data that is to be passed through the model (i.e. a batch of data). We pass this data into the first layer (self.layer1) and return the output as “out”.
- This output is then fed into the following layer and so on.
- Note, after self.layer2, we apply a reshaping function to out, which flattens the data dimensions from 7 x 7 x 64 into 3164 x 1.
- Next, the dropout is applied followed by the two fully connected layers, with the final output being returned from the function.

```
def forward(self, x):
    out = self.layer1(x)
    out = self.layer2(out)
    out = out.reshape(out.size(0), -1)
    out = self.drop_out(out)
    out = self.fc1(out)
    out = self.fc2(out)
    return out
```


Bütün hüquqlar qorunur.

Training the model

```
model = ConvNet()
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)
# Train the model
total_step = len(train_loader)
loss_list = []
acc_list = []
for epoch in range(num_epochs):
    for i, (images, labels) in enumerate(train_loader):
        # Run the forward pass
        outputs = model(images)
        loss = criterion(outputs, labels)
        loss_list.append(loss.item())

        # Backprop and perform Adam optimisation
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        # Track the accuracy
        total = labels.size(0)
        _, predicted = torch.max(outputs.data, 1)
        correct = (predicted == labels).sum().item()
        acc_list.append(correct / total)

        if (i + 1) % 100 == 0:
            print('Epoch [{}/{}], Step [{}/{}], Loss: {:.4f}, Accuracy: {:.2f}%'
                  .format(epoch + 1, num_epochs, i + 1, total_step, loss.item(),
                          (correct / total) * 100))
```

- Before we train the model, we have to first create an instance of our ConvNet class, and define our loss function and optimizer:
- Next – the training loop is created:
- The training output will look something like this:

```
Epoch [1/6], Step [100/600], Loss: 0.2183, Accuracy: 95.00%
Epoch [1/6], Step [200/600], Loss: 0.1637, Accuracy: 95.00%
Epoch [1/6], Step [300/600], Loss: 0.0848, Accuracy: 98.00%
Epoch [1/6], Step [400/600], Loss: 0.1241, Accuracy: 97.00%
Epoch [1/6], Step [500/600], Loss: 0.2433, Accuracy: 95.00%
Epoch [1/6], Step [600/600], Loss: 0.0473, Accuracy: 98.00%
Epoch [2/6], Step [100/600], Loss: 0.1195, Accuracy: 97.00%
```


Bütün hüquqlar qorunur.

Testing the model

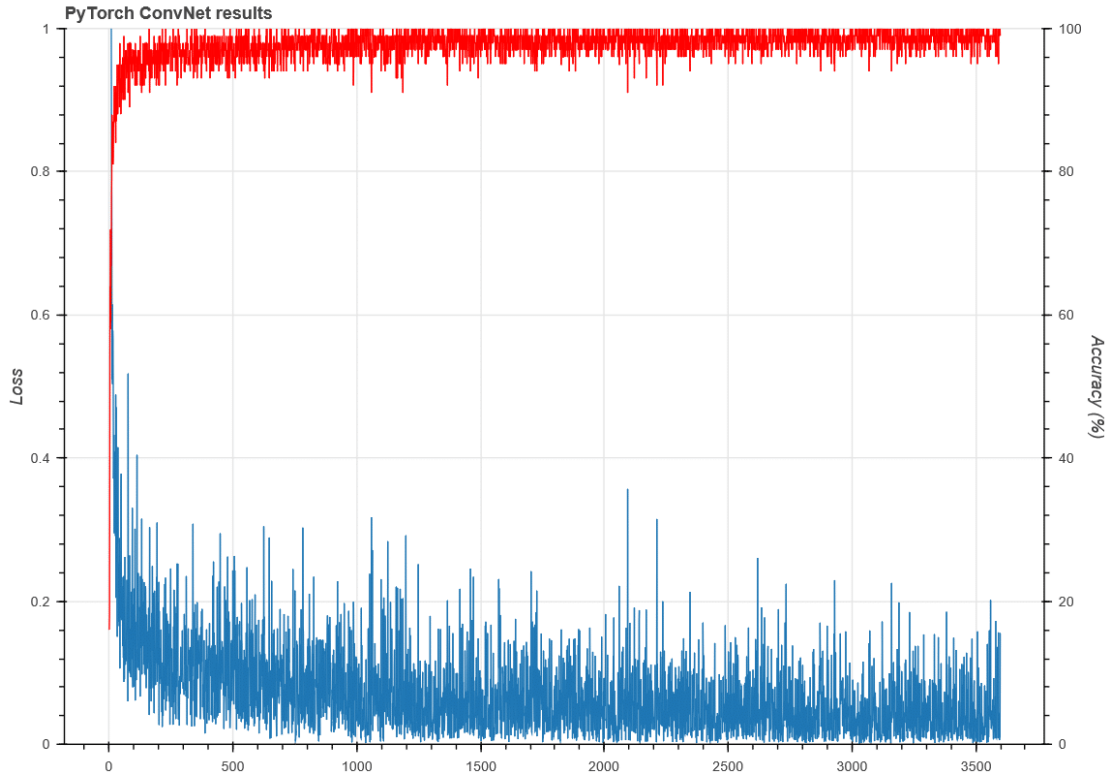
- To test the model, we use the following code:
- As a first step, we set the model to evaluation mode by running model.eval(). This is a handy function which disables any drop-out or batch normalization layers in your model.
- The torch.no_grad() statement disables the autograd functionality in the model.
- The rest is the same as the accuracy calculations during training, except that in this case, the code iterates through the test_loader.
- Finally, the result is output to the console, and the model is saved using the torch.save() function.

```
# Test the model
model.eval()
with torch.no_grad():
    correct = 0
    total = 0
    for images, labels in test_loader:
        outputs = model(images)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

    print('Test Accuracy of the model on the 10000 test images: {} %'.format((correct / total) *
100))

# Save the model and plot
torch.save(model.state_dict(), MODEL_STORE_PATH + 'conv_net_model.ckpt')
```


Bütün hüquqlar qorunur.

Result

- As can be observed, the network quite rapidly achieves a high degree of accuracy on the training set, and the test set accuracy, after 6 epochs, arrives at 99%. Certainly, better than the accuracy achieved in basic fully connected neural networks.



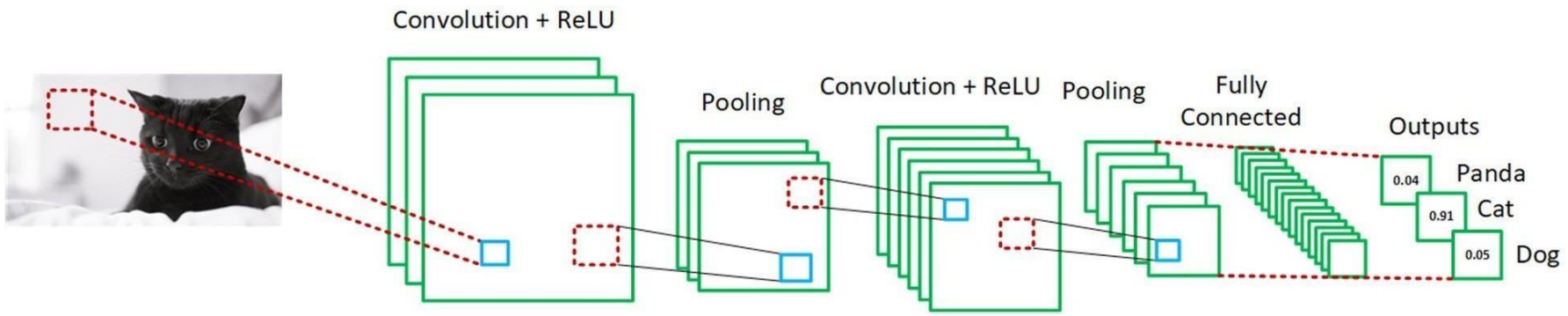
Bütün hüquqlar qorunur.

II
Machine View: Classic ConvNet Architectures:
LeNet-5 Classic, AlexNet, VGGNet, Resnet, Inception, Darknet

Data Science Academy

Quick Overview

- Convolutional Neural Networks are a type of deep learning neural network. These types of neural nets are widely use in computer vision and have pushed the capabilities of computer vision over the last few years, performing exceptionally better than older, more traditional neural networks;
- However, studies show that there are trade-offs related to training times and accuracy.
- The pre-processing required in ConvNet is much lower as compared to other classification
- Algorithms
- CNNs are perfectly suited for Image Classifications



Bütün hüquqlar qorunur.

ConvNet Architectures

There are several architectures in the field of Convolutional Networks that have a name. The most common are:

- LeNet
- AlexNet
- VGGNet
- ResNet

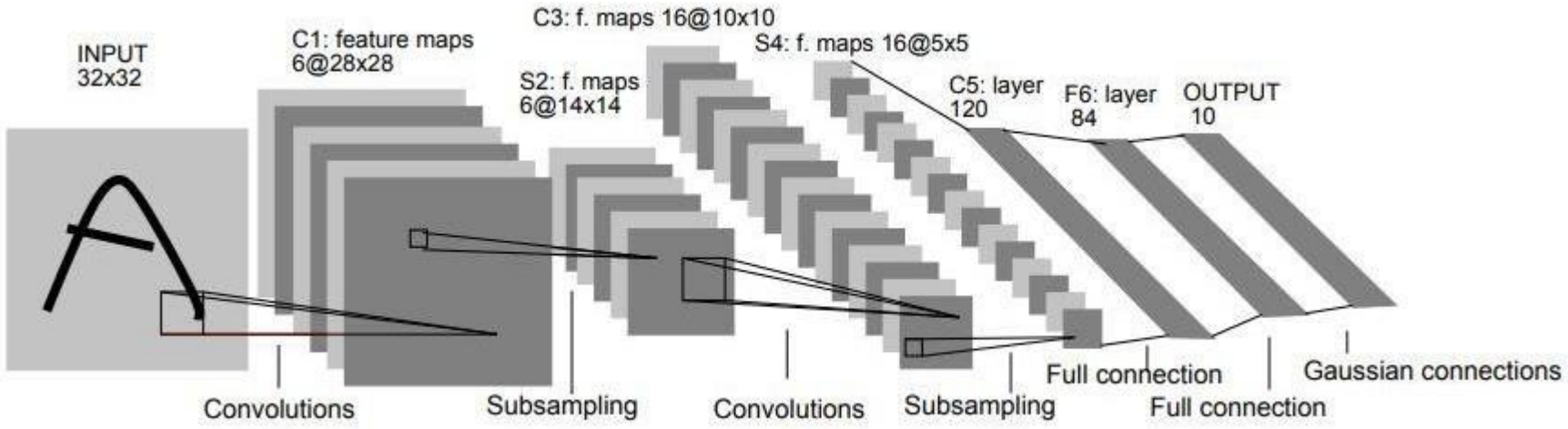
Bütün hüquqlar qorunur.

LeNet – 5 and AlexNet

Data Science Academy

LeNet – A Classic CNN Architecture

- Yann LeCun, Leon Bottou, Yosuha Bengio and Patrick Haffner proposed a neural network architecture for handwritten and machine-printed character recognition in 1990's which they called LeNet-5.
- The architecture is straightforward and simple to understand that's why it is mostly used as a first step for teaching Convolutional Neural Network.
- The LeNet-5 architecture consists of two sets of convolutional and average pooling layers, followed by a flattening convolutional layer, then two fully-connected layers and finally a softmax classifier.



Bütün hüquqlar qorunur.

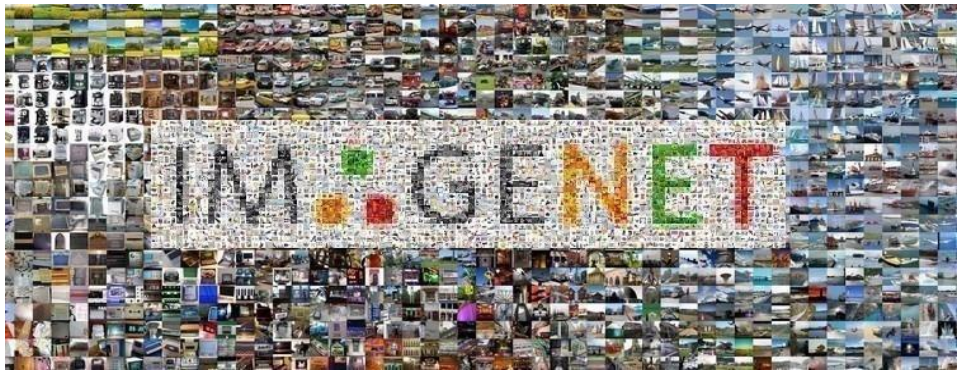
AlexNet: The Architecture that Challenged LeNet-5

- The recent availability of large datasets like ImageNet, which consist of hundreds of thousands to millions of labeled images, have pushed the need for an extremely capable deep learning model.
- Then came AlexNet.
- The architecture consists of eight layers: five convolutional ayers and three fully-connected layers.
- But this isn't what makes AlexNet special; here are some of the features used that are new approaches to convolutional neural networks:
  - ReLU Nonlinearity
  - Multiple GPUs
  - Overlapping Pooling

Bütün hüquqlar qorunur.

What is ImageNet

- ImageNet: a dataset made of more than 15 million high-resolution images labeled with 22 thousand classes.
- The key: web-scraping images and crowd-sourcing human labelers.
- ImageNet even has its own competition: the ImageNet Large-Scale Visual Recognition Challenge (ILSVRC).
- In this competition, data is not a problem; there are about 1.2 million training images, 50 thousand validation images, and 150 thousand testing images.
- The authors enforced a fixed resolution of 256x256 pixels for their images by cropping out the center 256x256 patch of each image.



Bütün hüquqlar qorunur.

AlexNet : The Results

- On the 2010 version of the ImageNet
- competition, the best model achieved 47.1% top-1 error and 28.2% top-5 error.
- AlexNet vastly outpaced this with a 37.5% top-1 error and a 17.0% top-5 error.
- AlexNet is able to recognize off-center objects and most of its top five classes for each image are reasonable.

Bütün hüquqlar qorunur.

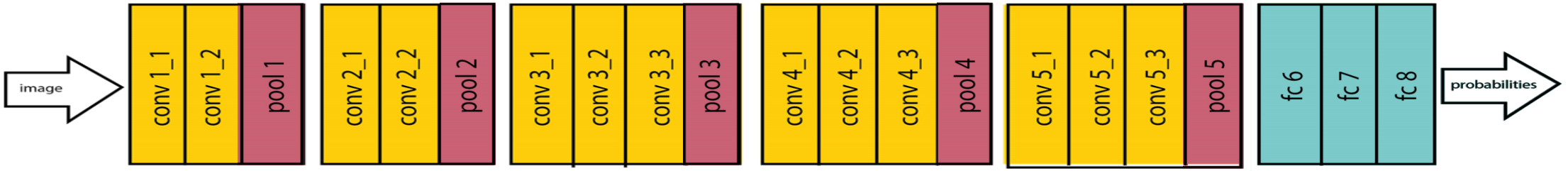
Other Architectures

VGGNet , ResNet , Inception_v3

Data Science Academy

VGGNet

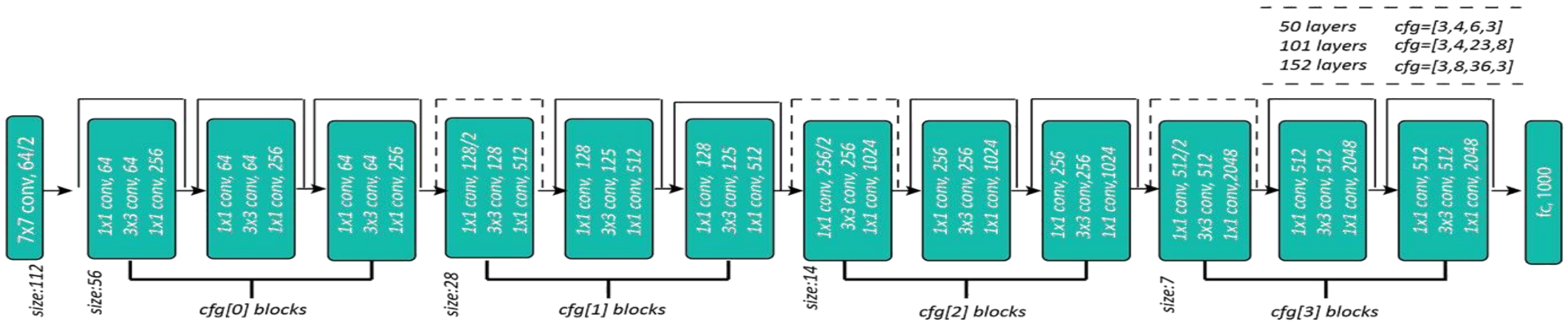
- VGGNet consists of 16 convolutional layers and is very appealing because of its very uniform architecture.
- Similar to AlexNet, only 3x3 convolutions, but lots of filters.
- It is currently the most preferred choice in the community for extracting features from images.
- The weight configuration of the VGGNet is publicly available and has been used in many other applications and challenges as a baseline feature extractor.
- However, VGGNet consists of 138 million parameters, which can be a bit challenging to handle.



Bütün hüquqlar qorunur.

ResNet

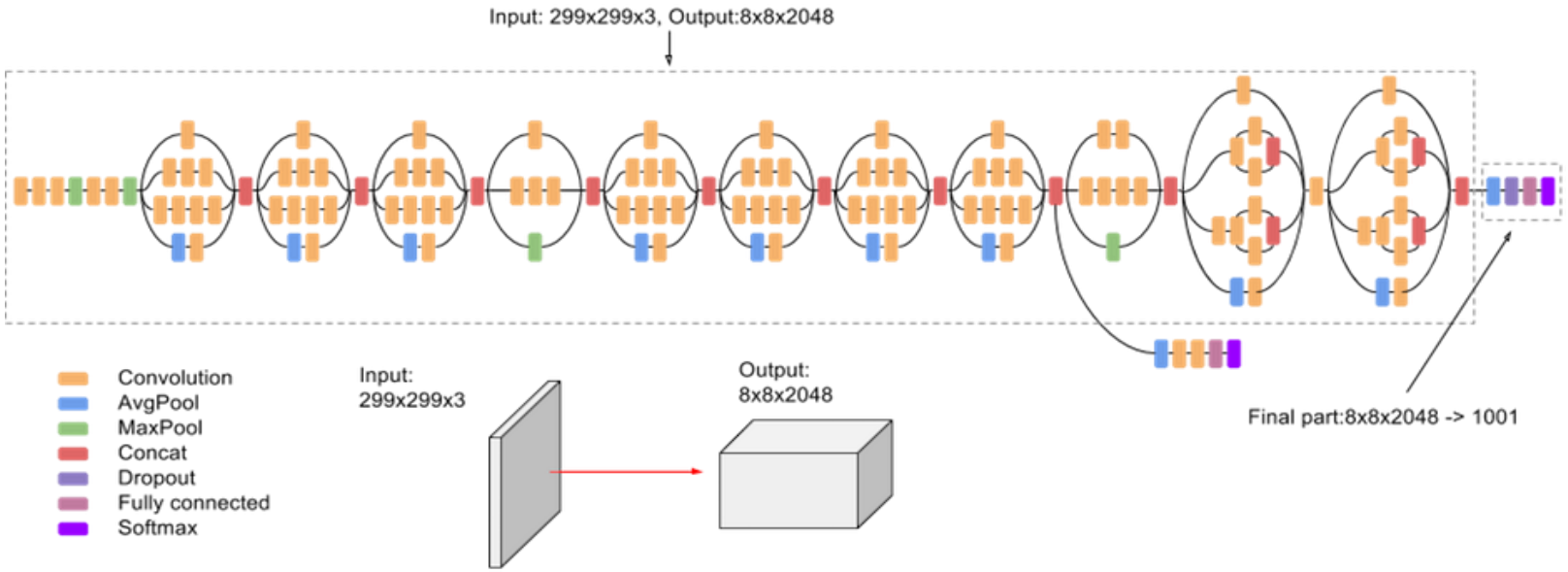
- At last, at the ILSVRC 2015, the so-called Residual Neural Network (ResNet) by Kaiming He et al introduced anovel architecture with "skip connections" and features heavy batch normalization.
- Such skip connections are also known as gated units or gated recurrent units and have a strong similarity to recent successful elements applied in RNNs.
- Thanks to this technique they were able to train a NN with 152 layers while still having lower complexity than VGGNet. It achieves a top-5 error rate of 3.57% which beats human-level performance on this dataset.



Bütün hüquqlar qorunur.

Inception_v3

- Inception V3 by Google is the 3rd version in a series of Deep Learning Convolutional Architectures.
- Inception V3 was trained using a dataset of 1,000 classes from the original ImageNet dataset which was trained with over 1 million training images, the Tensorflow version has 1,001 classes which is due to an additional "background' class not used in the original ImageNet.
- Inception V3 was trained for the ImageNet Large Visual Recognition Challenge where it was a first runner



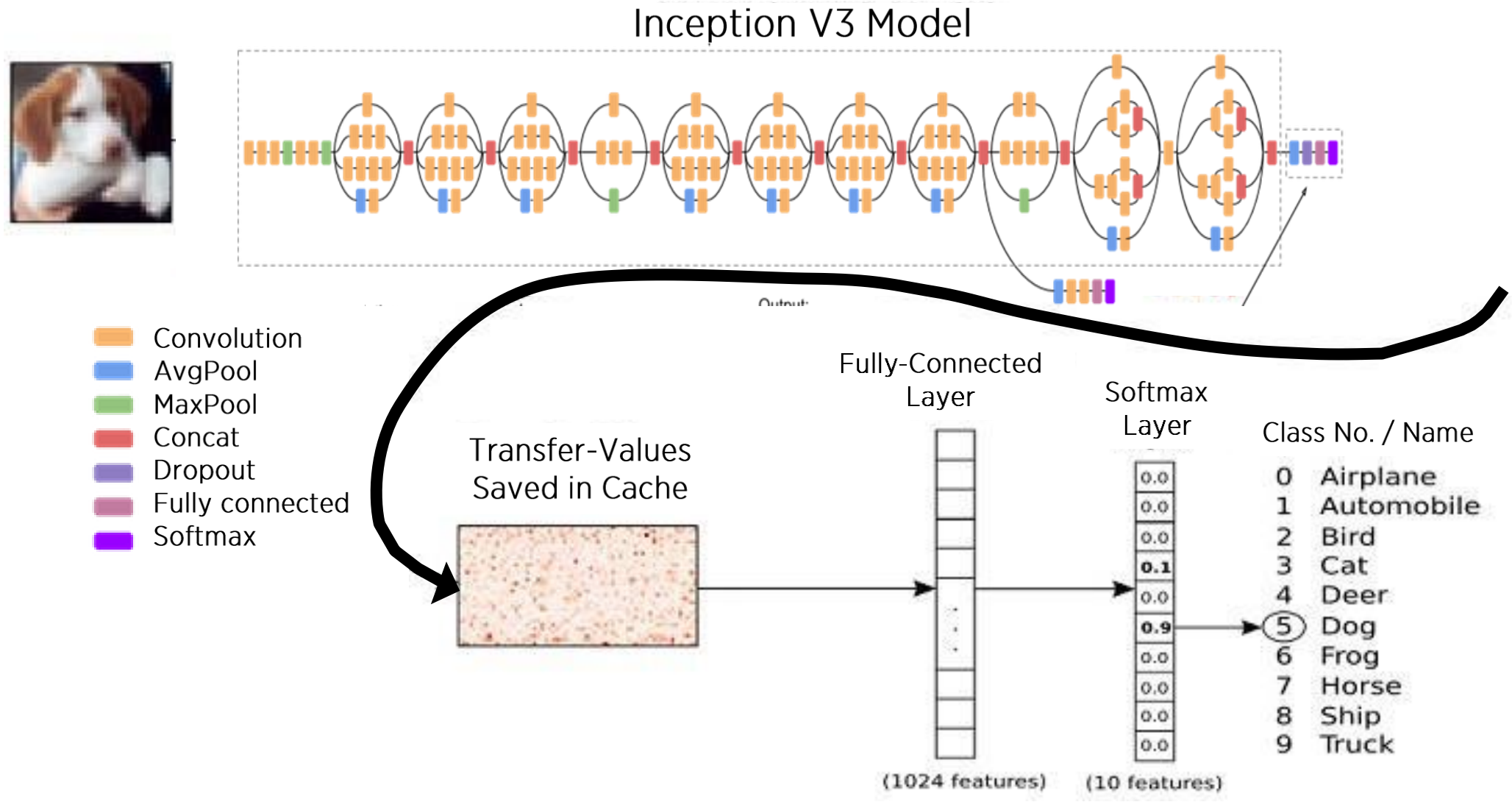
Bütün hüquqlar qorunur.

Transfer Learning

- Transfer learning allows you to retrain the final layer of an existing model, resulting in a significant decrease in not only training time, but also the size of the dataset required.
- One of the most famous models that can be used for transfer learning is Inception V3.
- As mentioned above, this model was originally trained on over a million images from 1,000 classes on some very powerful machines.
- Being able to retrain the final layer means that you can maintain the knowledge that the model had earned during its original training and apply it to your smaller dataset, resulting in highly accurate classifications without the need for extensive training and computational power.

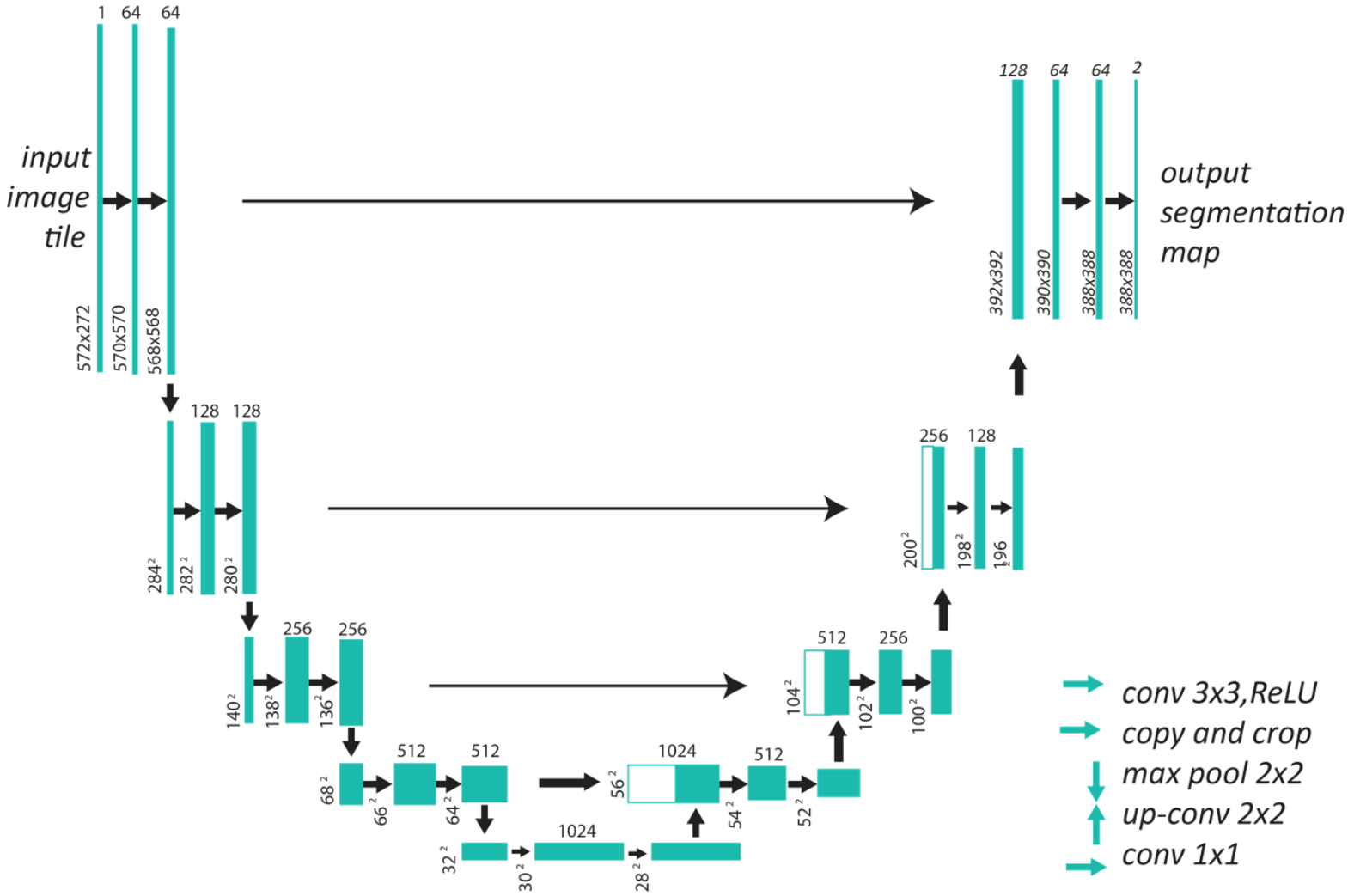
Bütün hüquqlar qorunur.

Transfer Learning



Bütün hüquqlar qorunur.

Unet



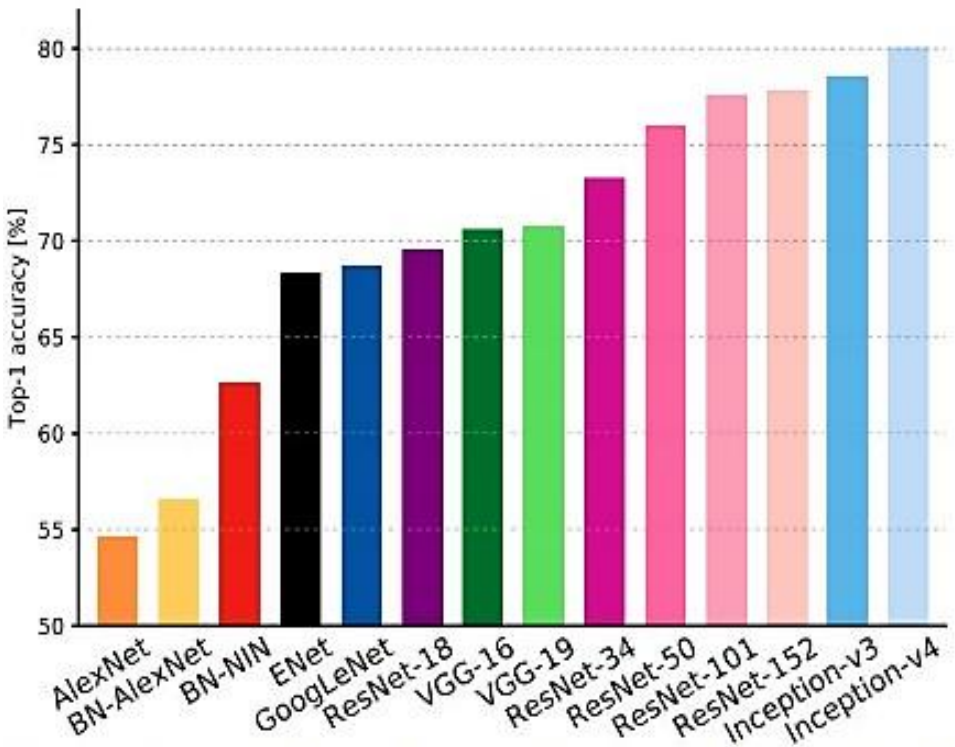
Bütün hüquqlar qorunur.

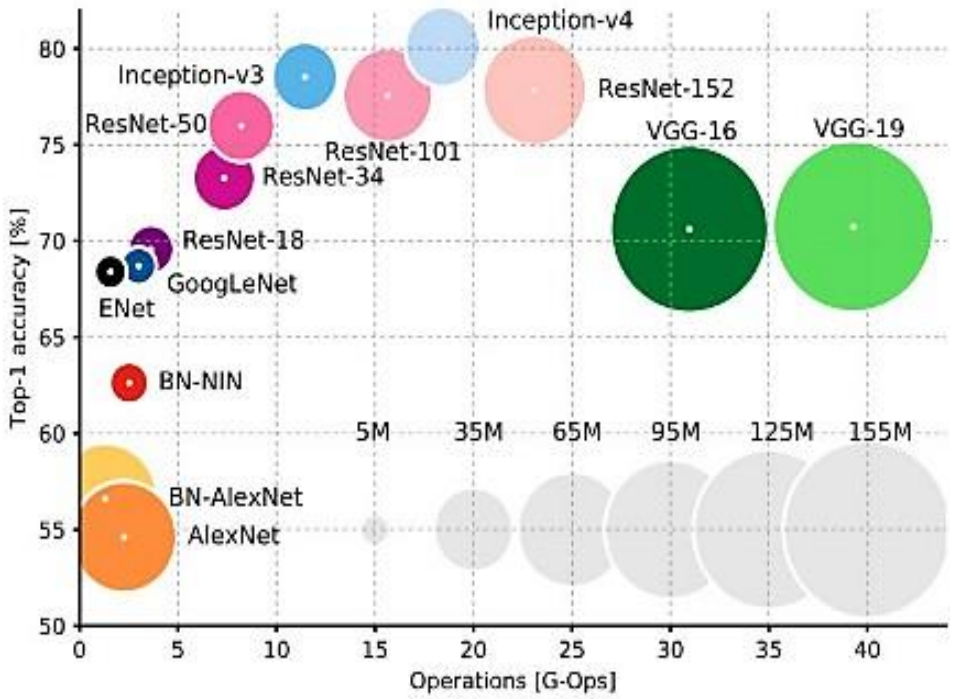
Unet

- First sight, it has a "U" shape.
- The architecture is symmetric and consists of two major parts
- The left part is called contracting path, which is constituted by the general convolutional process;
- The right part is expansive path, which is constituted by transposed 2d convolutional layers(you can think it as an upsampling technic for now).

Bütün hüquqlar qorunur.

Summary Accuracy Table





An Analysis of Deep Neural Network Models for Practical Applications, 2017.

Bütün hüquqlar qorunur.

Summary Table

Year	CNN	Developed by	Place	Top-5 error rate	No. of parameters
1998	AlexNet(7)	Yann LeCun et al			60 thousand
2012	LeNet(8)	Alex Krizhevsky,Geoffrey Hinton ,Ilya Sutskever	1st	15.3%	60 million
2013	ZFNet()	Matthew Zeiler and Rob Fergus	1st	14.8%	
2014	GoogLeNet(19)	Google	1st	6.67%	4 million
2014	VGG Net(16)	Simonyan,Zisserman	2nd	7.3%	138 million
2015	ResNet(152)	Kaiming He	1st	3.6%	

Bütün hüquqlar qorunur.

Computer Vision
in practice

Data Science
Academy

The Importance of Computer Vision

Today's AI systems can go a step further and take actions based on an understanding of the image. There are many types of computer vision that are used in different ways:

- Image segmentation partitions an image into multiple regions or pieces to be examined separately.
- Object detection identifies a specific object in an image. Advanced object detection recognizes many objects in a single image: a football field, an offensive player, a defensive player, a ball and so on.
- These models use an X,Y coordinate to create a bounding box and identify everything inside the box.
- Facial recognition is an advanced type of object detection that not only recognizes a human face in an image, but identifies a specific individual.
- Edge detection is a technique used to identify the outside edge of an object or landscape to better identify what is in the image.
- Pattern detection is a process of recognizing repeated shapes, colors and other visual indicators in mages. Image classification groups images into different categories.
- Feature matching is a type of pattern detection that matches similarities in images to help classify them.

Bütün hüquqlar qorunur.

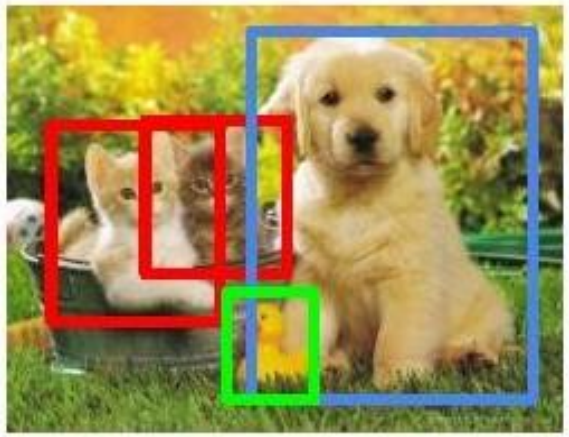
Classification, Localization, Object Detection and Segmentation

Classifaction

Classifaction
+ Localization

Object Detection

İnstance
Segmentation



CAT

CAT

CAT, DOG, DUCK

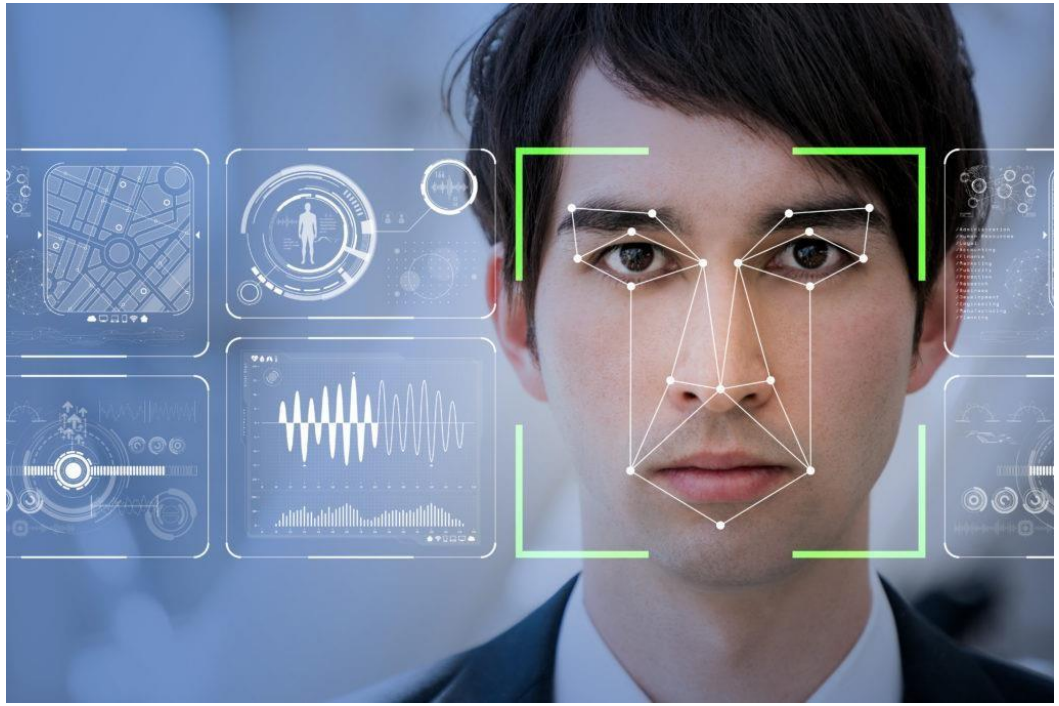
CAT, DOG, DUCK

Single object

Multiple object

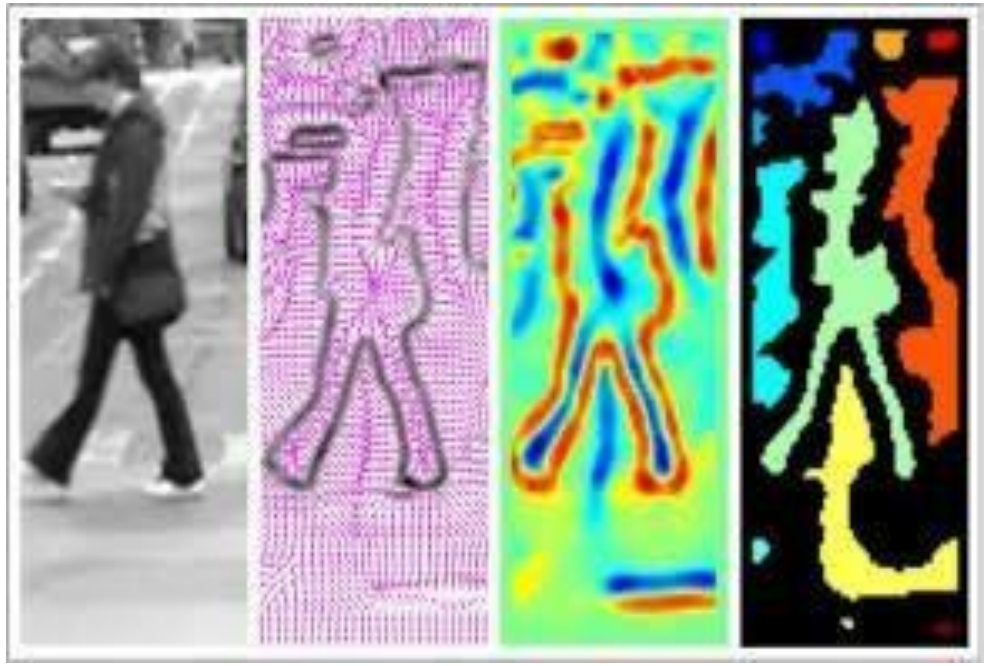
Bütün hüquqlar qorunur.

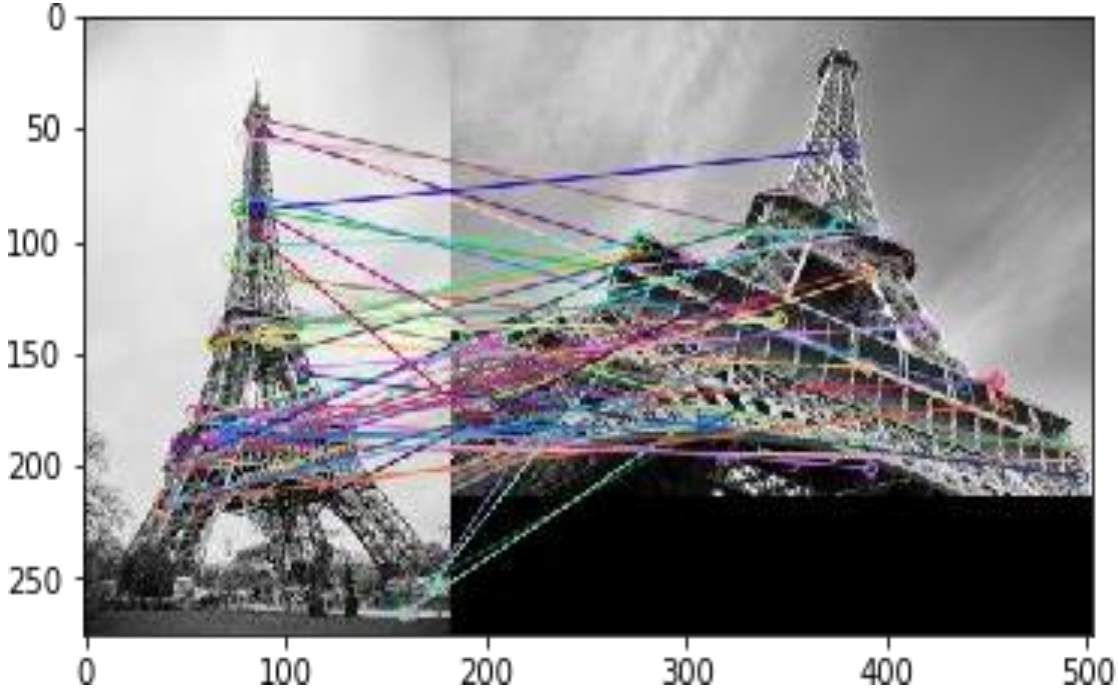
Facial Recognition & Edge Detection



Bütün hüquqlar qorunur.

Pattern Detection & Feature Matching



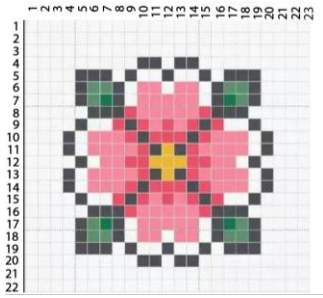


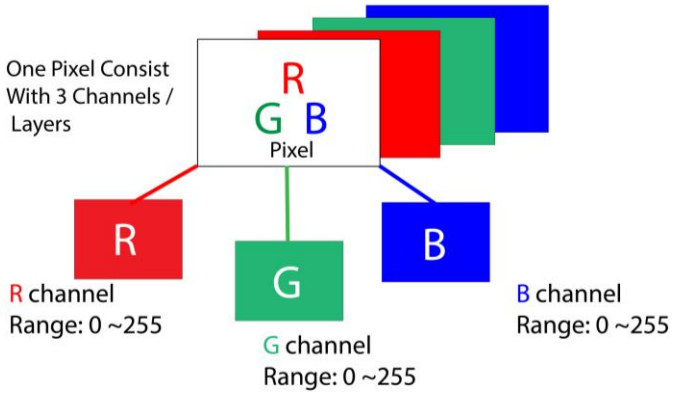
Bütün hüquqlar qorunur.

III
Working with Image Data: Channels, Matrices, Dimensions

Pixel

- Every photograph, in digital form, is made up of pixels. They are the smallest unit of information that makes up a picture. Usually round or square, they are typically arranged in a 2-dimensional grid.
- The word pixel means a picture element. A simple way to describe each pixel is using a combination of three colors, namely Red, Green, Blue. This is what we call an RGB image.
- In an RGB image, each pixel is represented by three 8 bit numbers associated to the values for Red, Green, Blue respectively.
- Each pixel coordinate (x, y) contains 3 values ranging for intensities of O to 255 (8-bit)
  -Red - Green – Blue
- Mixing different intensities of each color gives us the full color spectrum.

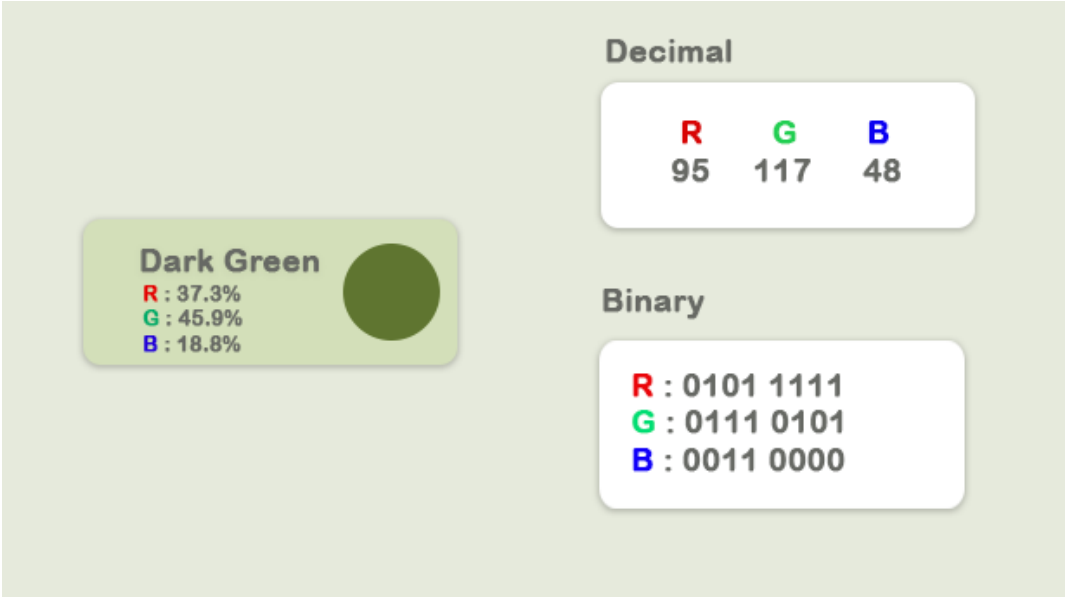




Bütün hüquqlar qorunur.

Pixel

- Now, if all three values are at full intensity, that means they're 255, it then shows as white and if all three colors are muted, or has the value of 0, the color shows as black.
- The combination of these three will, in turn, give us a specific shade of the pixel color. Since each number is an 8-bit number, the values range from 0-255.
- Combination of these three color will posses tends to the highest value among them. Since each value can have 256 different intensity or brightness value, it makes 16.8 million total shades.



Bütün hüquqlar qorunur.

Importing image

- Now, if all three values are at full intensity, that means they're 255, it then shows as white and if all three

```
if __name__ == '__main__':
    import imageio
    import matplotlib.pyplot as plt
    %matplotlib inline

    pic = imageio.imread('F:/demo_2.jpg')
    plt.figure(figsize = (15,15))

    plt.imshow(pic)
```

- Basic Properties of Image

```
print('Type : ' , type(pic))
print()
print('Shape of the image : {}'.format(pic.shape))
print('Image Hight {}'.format(pic.shape[0]))
print('Image Width {}'.format(pic.shape[1]))
print('Dimension of Image {}'.format(pic.ndim))
print('Image size {}'.format(pic.size))
print('Maximum RGB value in this image {}'.format(pic.max()))
print('Minimum RGB value in this image {}'.format(pic.min()))
```

```
Out [ ]  Type : <class 'imageio.core.util.Image'>
         Shape of the image : (562, 960, 3)
         Image Hight 562
         Image Width 960
         Dimension of Image 3
         Image size 1618560
         Maximum RGB value in this image 255
         Minimum RGB value in this image 0
```

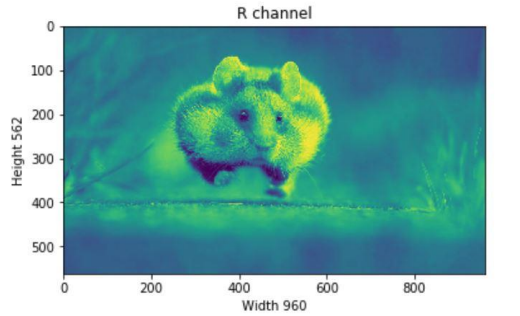

Bütün hüquqlar qorunur.

Image Channels

- Let’s take a view of each channels in the whole image.

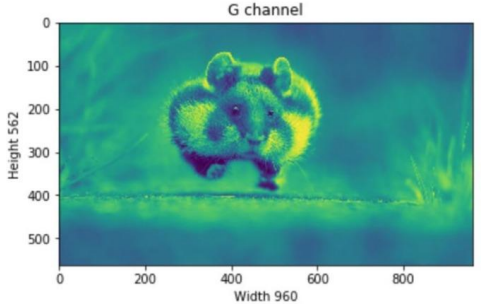
```
plt.title('R channel')
plt.ylabel('Height {}'.format(pic.shape[0]))
plt.xlabel('Width {}'.format(pic.shape[1]))

plt.imshow(pic[ : , : , 0])
plt.show()
```

Out [ ]



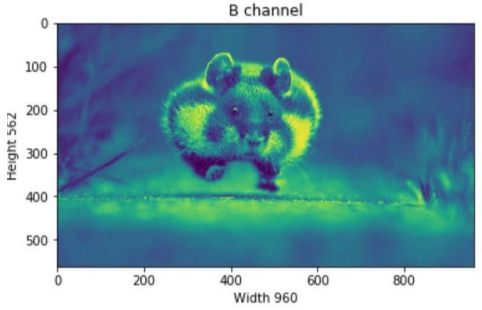

```
plt.title('G channel')
plt.ylabel('Height {}'.format(pic.shape[0]))
plt.xlabel('Width {}'.format(pic.shape[1]))

plt.imshow(pic[ : , : , 1])
plt.show()
```

Out [ ]




```
plt.title('B channel')
plt.ylabel('Height {}'.format(pic.shape[0]))
plt.xlabel('Width {}'.format(pic.shape[1]))

plt.imshow(pic[ : , : , 2])
plt.show()
```

Out [ ]



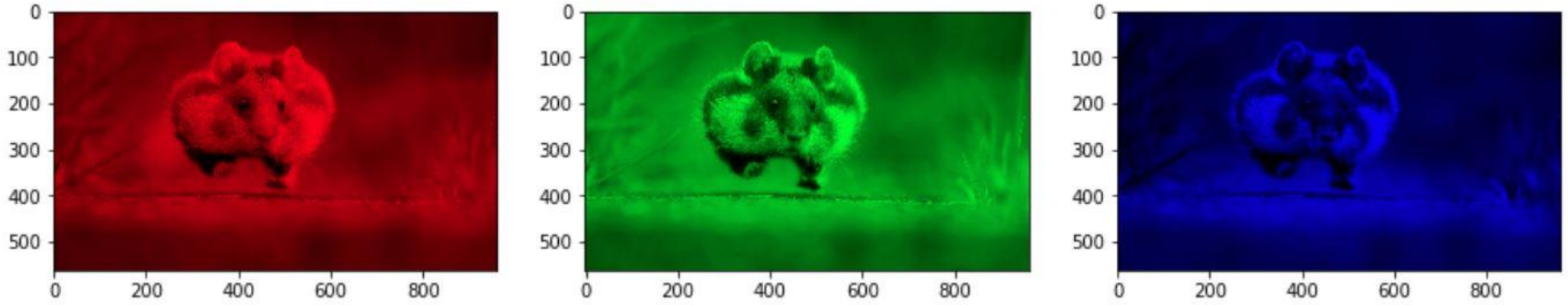
Bütün hüquqlar qorunur.

Splitting Layers

- We know that each pixel of the image is represented by three integers.
- Splitting the image into separate color components is just a matter of pulling out the correct slice of the image array.

```
import numpy as np
pic = imageio.imread('F:/demo_2.jpg')
fig, ax = plt.subplots(nrows = 1, ncols=3, figsize=(15,5))
for c, ax in zip(range(3), ax):
    # create zero matrix
    split_img = np.zeros(pic.shape, dtype="uint8") # 'dtype' by default: 'numpy.float64'

    # assing each channel
    split_img[ :, :, c] = pic[ :, :, c]

    # display each channel
    ax.imshow(split_img)
```

Out [ ]



Bütün hüquqlar qorunur.

Opening and writing to image files

- Writing an array to a file:

```
from scipy import ndimage
from scipy import misc
import imageio
f = misc.face()
imageio.imsave('face.png', f) # uses the Image module (PIL)
import matplotlib.pyplot as plt
plt.imshow(f)
plt.show()
```

- Creating a numpy array from an image file:

```
from scipy import misc
import imageio
face = misc.face()
imageio.imsave('face.png', face)
# First we need to create the PNG file

face = imageio.imread('face.png')
type(face)

face.shape, face.dtype
```


Bütün hüquqlar qorunur.

Displaying Images

- Use matplotlib and imshow to display an image inside a matplotlib figure:

```
f = misc.face(gray=True)  # retrieve a grayscale image
import matplotlib.pyplot as plt
plt.imshow(f, cmap=plt.cm.gray)
```

```
Out [ ]  <matplotlib.image.AxesImage object at 0x...>
```

- Increase contrast by setting min and max values:

```
plt.imshow(f, cmap=plt.cm.gray, vmin=30, vmax=200)

# Remove axes and ticks
plt.axis('off')
```

```
Out [ ] <matplotlib.image.AxesImage object at 0x...>
        (-0.5, 1023.5, 767.5, -0.5)
```

- Draw contour lines:

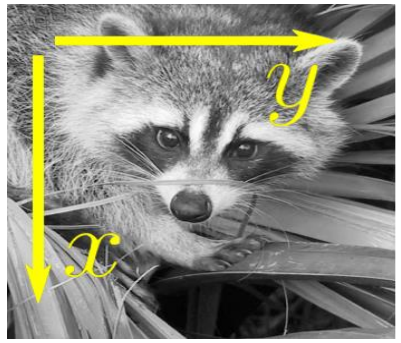
```
plt.contour(f, [50, 200])
```

```
Out [ ]  <matplotlib.contour.QuadContourSet ...>
```

Bütün hüquqlar qorunur.

Basic Manupilations

- Images are arrays: use the whole numpy machinery.



0	1	2
3	4	5
6	7	8

```
face = misc.face(gray=True)
face[0, 40]

# Slicing
face[10:13, 20:23]
face[100:120] = 255

lx, ly = face.shape
X, Y = np.ogrid[0:lx, 0:ly]
mask = (X - lx / 2) ** 2 + (Y - ly / 2) ** 2 > lx * ly / 4
# Masks
face[mask] = 0
# Fancy indexing
face[range(400), range(400)] = 255
```

```
Out [] 127      array([[141, 153, 145],0
                       [133, 134, 125],
                       [ 96, 92, 94]],   dtype=uint8)
```

Bütün hüquqlar qorunur.

Image Augmentations

Data Science Academy

Why Do We Need Image Augmentation?

- Deep learning models usually require a lot of data for training. In general, the more the data, the better the performance of the model. But acquiring massive amounts of data comes with its own challenges. Not everyone has the deep pockets of the big firms.

- And the problem with a lack of data is that our deep learning model might not learn the pattern or function from the data and hence it might not give a good performance on unseen data.

- So what can we do in that case? Instead of spending days manually collecting data, we can make use Of Image Augmentation techniques.

Bütün hüquqlar qorunur.

What is Image augmentations

- Image Augmentation is the process of generating new images for training our deep learning model.
- These new images are generated using the existing training images and hence we don't have to collect them manually.

Bütün hüquqlar qorunur.

Different Image Augmentation Techniques

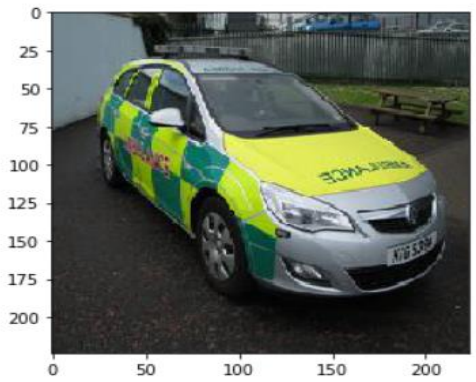
Image Rotation

- Image Rotation is one of the most commonly used augmentation techniques is image rotation. Even if you rotate the image, the information on the image remains the same. A car is a car, even if you see it from a different angle.

- Let's import the image and visualize it first:

```
# reading the image using its path
image = io.imread('image.jpg')

# shape of the image
print(image.shape)

# displaying the image
io.imshow(image)
```

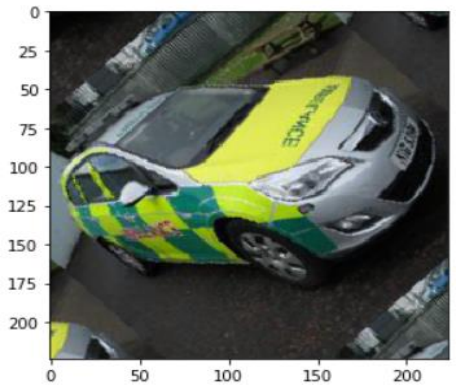
```
(224, 224, 3)
<matplotlib.image.AxesImage at 0x7f5ceb6d4240>
```



- Let's now see how we can rotate it.
  We will use the rotate function of the skimage library to rotate the image:

```
print('Rotated Image')
#rotating the image by 45 degrees
rotated = rotate(image, angle=45, mode = 'wrap')
#plot the rotated image
io.imshow(rotated)
```

```
Rotated Image
<matplotlib.image.AxesImage at 0x7f5ce9dfa470>
```

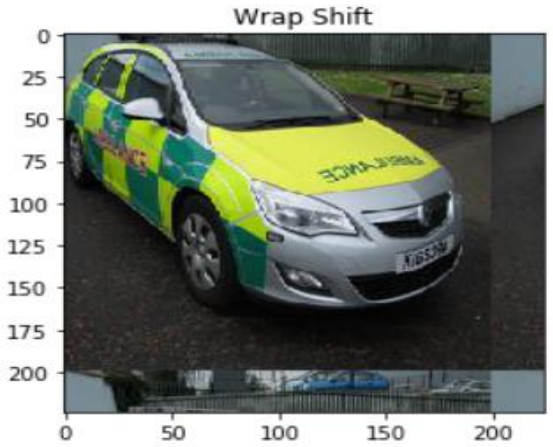


Bütün hüquqlar qorunur.

Shifting Images

- Image shift is a geometric transformation that maps the position of every object in the image to a new location in the final output image.
- There might be scenarios when the objects in the image are not perfectly central aligned. In these cases, image shift can be used to add shift-invariance to the images.
- By shifting the images, we can change the position of the object in the image and hence give more variety to the model. This will eventually lead to a more generalized model.
- After the shift operation, an object present at a location (x,y) in the input image is shifted to a new position (X, Y):
  - X = x + dx
  - Y = y + dy
- Here, dx and dy are the respective shifts along different dimensions. Let's see how we can apply shift to an image:

```
#apply shift operation
transform = AffineTransform(translation=(25,25))
wrapShift = warp(image,transform,mode='wrap')
plt.imshow(wrapShift)
plt.title('Wrap Shift')
```



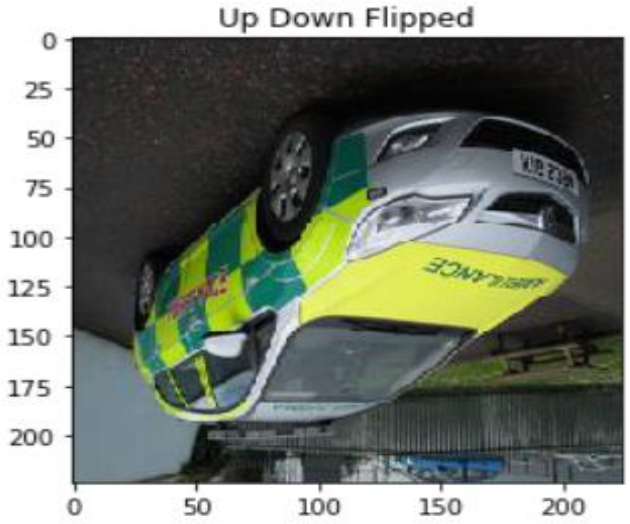
Bütün hüquqlar qorunur.

Flipping Images

- Flipping is an extension of rotation. It allows us to flip the image in the left-right as well as up-down direction. Let's see how we can implement flipping:
- We can use the fliplr or flipud function of NumPy to flip the image from left to right or up to down

```
#flip image left-to-right
flipLR = np.fliplr(image)

plt.imshow(flipLR)
plt.title('Left to Right Flipped')
```


```
#flip image up-to-down
flipUD = np.flipud(image)

plt.imshow(flipUD)
plt.title('Up Down Flipped')
```

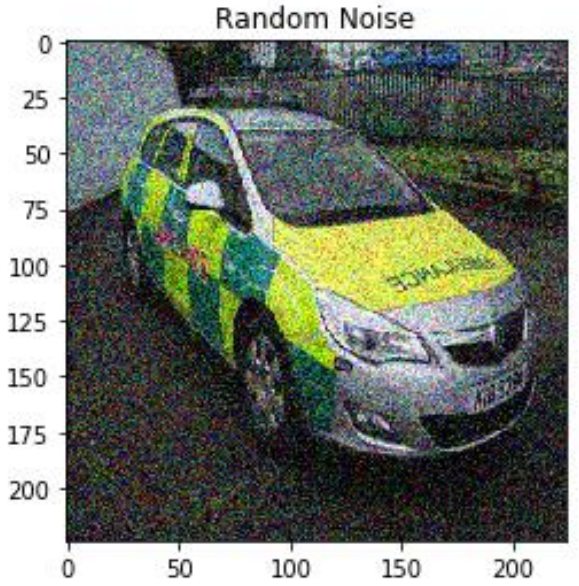


Bütün hüquqlar qorunur.

Adding Noise to Images

- Image noising is an important augmentation step that allows our model to learn how to separate signal from noise in an image. This also makes the model more robust to changes in the input.
- We will use the random_noise function of the skimage library to add some random noise to our original image.
- I will take the standard deviation of the noise to be added as 0.155 (you can change this value as well). Just keep in mind that increasing this value will add more noise to the image and vice versa:

```
#standard deviation for noise to be added in the image
sigma=0.155
#add random noise to the image
noisyRandom = random_noise(image,var=sigma**2)

plt.imshow(noisyRandom)
plt.title('Random Noise')
```

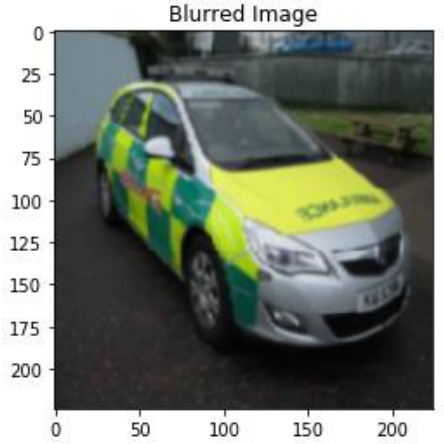


Bütün hüquqlar qorunur.

Blurring Images

- All photography lovers will instantly understand this idea.
- Images come from different sources. And hence, the quality of the images will not be the same from each source. Some images might be of very high quality while others might be just plain bad.
- In such scenarios, we can blur the image. How will that help? Well, this helps make our deep learning model more robust.
- Let's see how we can do that. We will use a Gaussian filter for blurring the image:

```
#blur the image
blurred = gaussian(image,sigma=1,multichannel=True)

plt.imshow(blurred)
plt.title('Blurred Image')
```



- Sigma here is the standard deviation for the Gaussian filter. I have taken it as 1. The higher the sigma value, the more will be the blurring effect. Setting *Multichannel* to true ensures that each channel of the image is filtered separately.

Bütün hüquqlar qorunur.

Basic Guidelines for Selecting the Right Augmentation Technique

- These are some of the image augmentation techniques which help to make our deep learning model robust and generalizable. This also helps increase the size of the training set.
- There are a few guidelines that are important while deciding the augmentation technique based on the problem that you are trying to solve. Here is a brief summary of these guidelines:
- The first step in any model building process is to make sure that the size of our input matches what is expected by the model. We also have to make sure that the size of all the images should be similar. For this, we can resize our images to the appropriate size.
- Let's say you are working on a classification problem and have relatively less number of data samples. In such scenarios, you can use different augmentation techniques like image rotation, image noising, flipping, shift, etc. Remember all these operations are applicable for classification problems where the location of objects in the image does not matter.
- If you are working on an object detection task, where the location of objects is what we want to detect, these techniques might not be appropriate.
- Normalizing image pixel values is always a good strategy to ensure better and faster convergence of the model. If there are some specific requirements of the model, we must pre-process the images as per the model's requirement.

Bütün hüquqlar qorunur.

Thank you!