UNIT V

Feature Mapping Using the SIFT Algorithm

Scale-invariant feature transform

Scale-invariant feature transform (SIFT) descriptors provide an alternative representation for image regions. They are very useful for matching images. As demonstrated earlier, simple corner detectors work well when the images to be matched are similar in nature (with respect to scale, orientation, and so on). But if they have different scales and rotations, the SIFT descriptors are needed to be used to match them. SIFT is not only just scale invariant, but it still obtains good results when rotation, illumination, and viewpoints of the images change as well.

Let's discuss the primary steps involved in the SIFT algorithm that transforms image content into local feature coordinates that are invariant to translation, rotation, scale, and other imaging parameters.

Algorithm to compute SIFT descriptors

Scale-space extrema detection: Search over multiple scales and image locations, the location and characteristic scales are given by DoG detector **Keypoint localization**: Select keypoints based on a measure of stability, keep only the strong interest points by eliminating the low-contrast and edge **keypoints Orientation assignment**: Compute the best orientation(s) for each keypoint region, which contributes to the stability of matching Keypoint **descriptor computation:** Use local image gradients at selected scale and rotation to describe each keypoint region

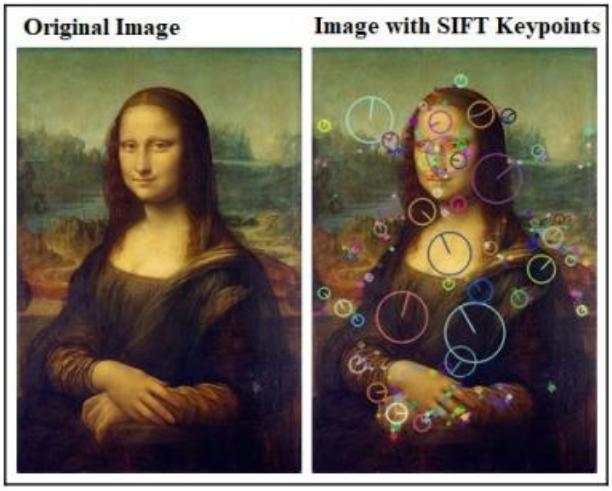
As discussed, SIFT is robust with regard to small variations in illumination (due to gradient and normalization), pose (small affine variation due to orientation histogram), scale (by DoG), and intra-class variability (small variations due to histograms).

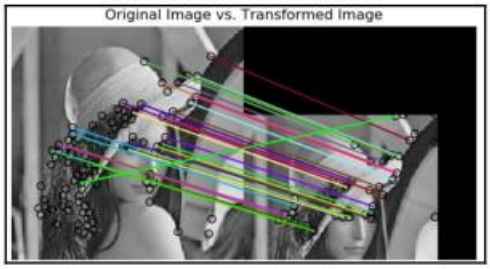
With opencv and opencv-contrib

We will first construct a SIFT object and then use the detect() method to compute the keypoints in an image. Every keypoint is a special feature, and has several attributes. For example, its (x, y) coordinates, angle (orientation), response (strength of keypoints), size of the meaningful neighborhood, and so on.

```
import cv2
import numpy as np
import matplotlib.pyplot as plt

# Load grayscale image
img_path = "images/monalisa.jpg"
img = cv2.imread(img_path, cv2.IMREAD_GRAYSCALE)
if img is None:
    raise FileNotFoundError(f"Image not found at {img_path}")
```

```python
# Create SIFT detector
sift = cv2.SIFT_create()
keypoints, descriptors = sift.detectAndCompute(img, None)


# Draw keypoints
img_kp = cv2.drawKeypoints(img, keypoints, None,
flags=cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)


# Convert grayscale to RGB for matplotlib
img_kp_rgb = cv2.cvtColor(img_kp, cv2.COLOR_BGR2RGB)


# Display
plt.figure(figsize=(10, 8))
plt.imshow(img_kp_rgb)
plt.axis('off')
plt.title(f"SIFT Keypoints (Total: {len(keypoints)})")
plt.show()


# Print descriptor info
print("Number of keypoints:", len(keypoints))
print("Descriptor shape:", descriptors.shape)
```





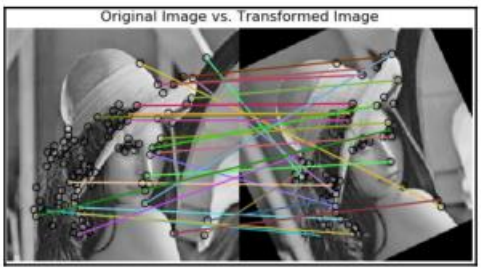


Image Registration Using the RANSAC Algorithm

In this example, we will match an image with its affine transformed version; they can be considered as if they were taken from different view points. The following steps describe the image matching algorithm:

1. First, we will compute the interest points or the Harris Corners in both the images.

2. A small space around the points will be considered, and the correspondences inbetween the points will then be computed using a weighted sum of squared differences. This measure is not very robust, and it's only usable with slight viewpoint changes.

3. A set of source and corresponding destination coordinates will be obtained once the correspondences are found; they are used to estimate the geometric transformations between both the images.

4. A simple estimation of the parameters with the coordinates is not enough—many of the correspondences are likely to be faulty.

5. The RANdom SAmple Consensus (RANSAC) algorithm is used to robustly estimate the parameters, first by classifying the points into inliers and outliers, and then by fitting the model to inliers while ignoring the outliers, in order to find matches consistent with an affine transformation. The next code block shows how to implement the image matching using the Harris Corner features:

```
# Read and prepare image
temple = rgb2gray(img_as_float(imread('../images/temple.JPG')))
image_original = np.zeros(list(temple.shape) + [3])
image_original[..., 0] = temple
```

```
gradient_row, gradient_col = np.mgrid[0:image_original.shape[0],
0:image_original.shape[1]] / float(image_original.shape[0])

image_original[..., 1] = gradient_row

image_original[..., 2] = gradient_col

image_original = rescale_intensity(image_original)

image_original_gray = rgb2gray(image_original)


# Apply affine transform

affine_trans = AffineTransform(scale=(0.8, 0.9), rotation=0.1, translation=(120, -20))

image_warped = warp(image_original, affine_trans.inverse,
output_shape=image_original.shape)

image_warped_gray = rgb2gray(image_warped)


# Detect corners

coords_orig = corner_peaks(corner_harris(image_original_gray), threshold_rel=0.01,
min_distance=5)

coords_warped = corner_peaks(corner_harris(image_warped_gray), threshold_rel=0.01,
min_distance=5)


# Refine corners to subpixel accuracy

coords_orig_subpix = corner_subpix(image_original_gray, coords_orig, window_size=9)

coords_warped_subpix = corner_subpix(image_warped_gray, coords_warped, window_size=9)


# Plot results

fig, axes = plt.subplots(1, 2, figsize=(20, 10))

axes[0].imshow(image_original_gray, cmap='gray')

axes[0].plot(coords_orig_subpix[:, 1], coords_orig_subpix[:, 0], 'r.', markersize=5)

axes[0].set_title('Original image with corners')

axes[0].axis('off')


axes[1].imshow(image_warped_gray, cmap='gray')

axes[1].plot(coords_warped_subpix[:, 1], coords_warped_subpix[:, 0], 'r.', markersize=5)

axes[1].set_title('Warped image with corners')

axes[1].axis('off')


plt.show()
```

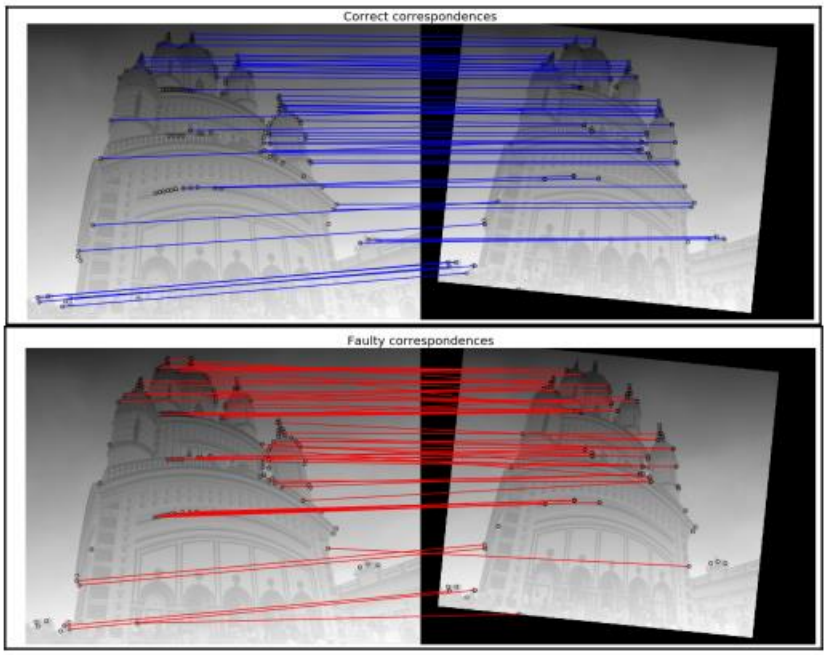


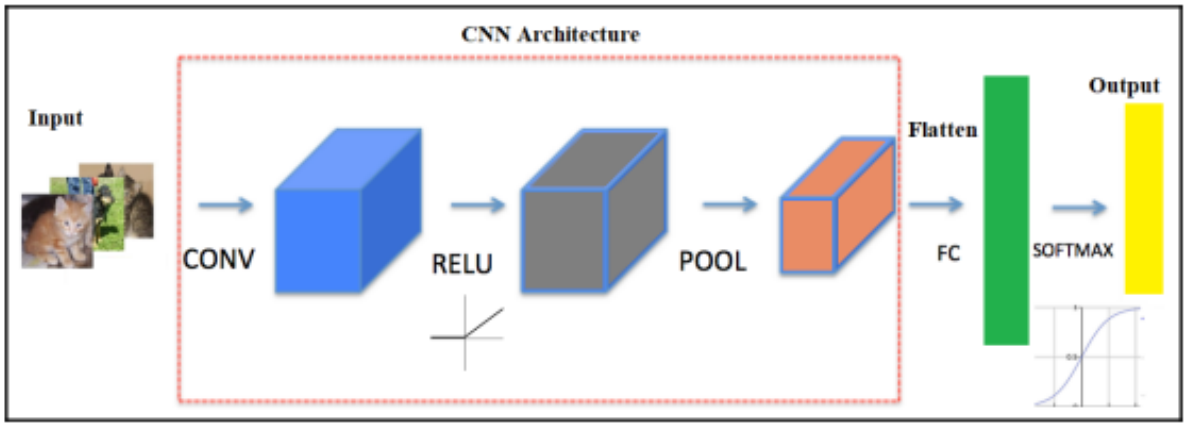
Image Classification Using CNNs

CNNs are deep neural networks for which the primarily used input is images. CNNs learn the filters (features) that are hand-engineered in traditional algorithms. This independence from prior knowledge and human effort in feature design is a major advantage. They also reduce the number of parameters to be learned with their shared-weights architecture and possess translation invariance characteristics. In the next subsection, we'll discuss the general architecture of a CNN and how it works.

Conv or pooling or FC layers – CNN architecture and how it works

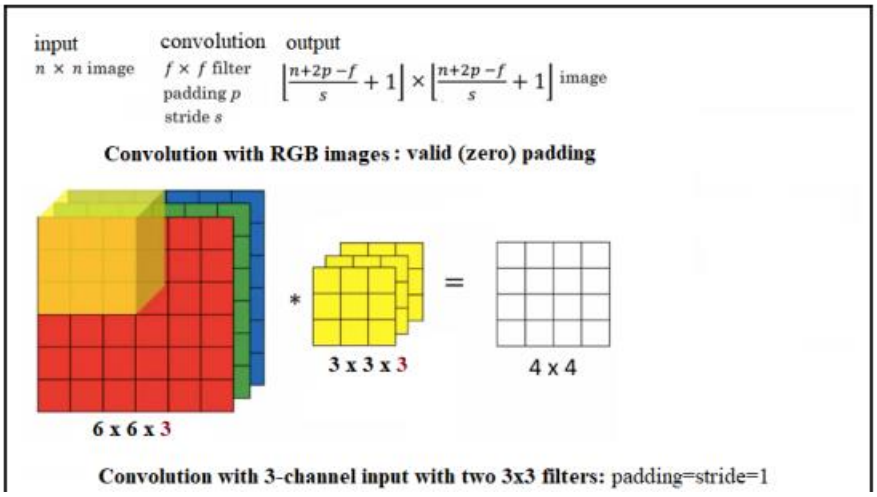
The next screenshot shows the typical architecture of a CNN. It consists of one or more convolutional layer, followed by a nonlinear ReLU activation layer, a pooling layer, and, finally, one (or more) **fully connected (FC)** layer, followed by an FC softmax layer, for example, in the case of a CNN designed to solve an image classification problem.

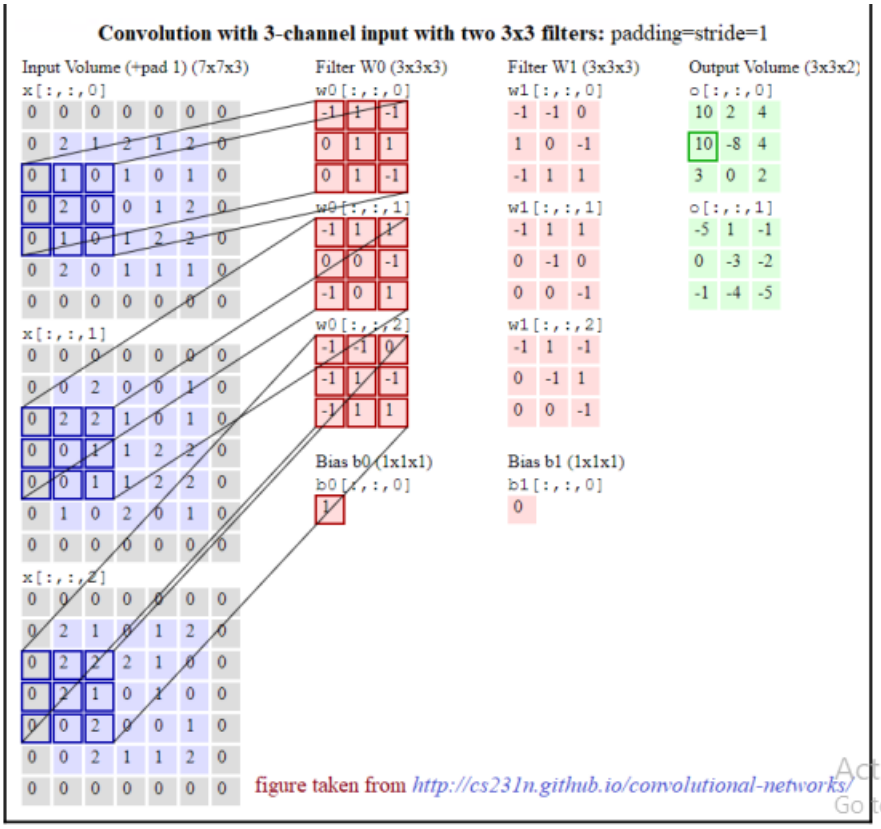
There can be multiple convolution ReLU pooling sequences of layers in the network, making the neural network deeper and useful for solving complex image processing tasks, as seen in the following diagram



Convolutional layer

The main building block of CNN is the convolutional layer. The convolutional layer consists of a bunch of convolution filters (kernels), which we already discussed in detail in Chapter 2(refere Text Book2), Sampling, Fourier Transform, and Convolution. The convolution is applied on the input image using a convolution filter to produce a feature map. On the left side is the input to the convolutional layer; for example, the input image. On the right is the convolution filter, also called the kernel. As usual, the convolution operation is performed by sliding this filter over the input. At every location, the sum of element-wise matrix multiplication goes into the feature map. A convolutional layer is represented by its width, height (the size of a filter is width x height), and depth (number of filters). Stride specifies how much the convolution filter will be moved at each step (the default value is 1). Padding refers to the layers of zeros to surround the input (generally used to keep the input and output image size the same, also known as same padding). The following screenshot shows how 3 x 3 x 3 convolution filters are applied on an RGB image, the first with valid padding and the second with the computation with two such filters with the size of the stride=padding=1





Pooling layer

After a convolution operation, a pooling operation is generally performed to reduce dimensionality and the number of parameters to be learned, which shortens the training time, requires less data to train, and combats overfitting. Pooling layers downsample each feature map independently, reducing the height and width, but keeping the depth intact. The most common type of pooling is max pooling, which just takes the maximum value in the pooling window. Contrary to the convolution operation, pooling has no parameters. It slides a window over its input and simply takes the max value in the window. Similar to a convolution, the window size and stride for pooling can be specified.

Non-linearity – ReLU layer

For any kind of neural network to be powerful, it needs to contain non-linearity. The result of the convolution operation is hence passed through the non-linear activation function. ReLU activation is used in general to achieve non-linearity (and to combat the vanishing gradient problem with sigmoid activation). So, the values in the final feature maps are not actually the sums, but the relu function applied to them.

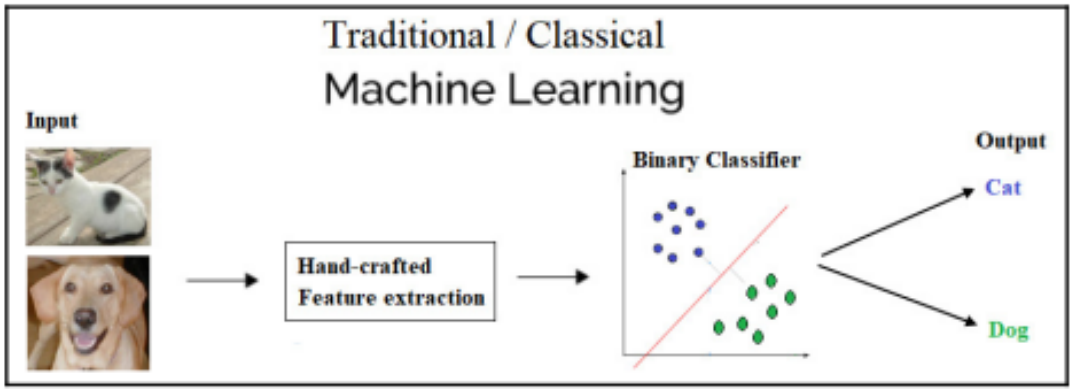
FC layer

After the convolutional and pooling layers, generally a couple of FC layers are added to wrap up the CNN architecture. The output of both convolutional and pooling layers are 3D volumes, but an FC layer expects a 1D vector of numbers. So, the output of the final pooling layer needs to be flattened to a vector, and that becomes the input to the FC layer. Flattening is simply arranging the 3D volume of numbers into a 1D vector.

Dropout

Dropout is the most popular regularization technique for deep neural networks. Dropout is used to prevent overfitting, and it is typically used to increase the performance (accuracy) of the deep learning task on the unseen dataset. During training time, at each iteration, a neuron is temporarily dropped or disabled with some probability, p. This means all the input and output to this neuron will be disabled at the current iteration. This hyperparameter p is called the dropout rate, and it's typically a number around 0.5, corresponding to 50% of the neurons being dropped out.

Image Classification Using Machine Learning Approaches:Decision Trees,Support Vector Machines, Logistics Regression,Code,Important Terms



Decision Trees

-Supervised learning technique for classification or regression problems
-We create a model that predicts the value of a target variable by learning simple decision rules inferred from the data features
-Not as effective as the best supervised machine learning techniques
improve the performance◊-Boosting and random forests
-Has several advantages + disadvantages

Advantages
-Simple to understand and to interpret + trees can be visualized
-No need for data preparations such as normalization or dummy variables
-Logarithmic O(logN) running time

Distadvantages
-Decision-tree learners can create over-complex trees that do not generalize to the data well
-This is the problem of overfitting // pruning somethime helps

-Decision trees can be unstable because small variations in the data might result in a completely different tree being generated
-The problem of learning an optimal decision tree is known to be NP-complete !!!
-Practical decision-tree learning algorithms are based on heuristic algorithms such as the greedy algorithm

bias: error from misclassifications in the learning algorithm
the algorithm misses the relevant relationships◊ High bias
between features and target outputs !!!

ERROR DUE TO MODEL MISMATCH

variance: error from sensitivity to small changes in the training set
can cause overfitting◊ High variance

VARIATION DUE TO TRAINING SAMPLE AND RANDOMIAZTION

Bias / variance tradeoff
~ we are not able to optimize both bias and variance at the same time

high variance◊ low bias
high bias◊ low variance

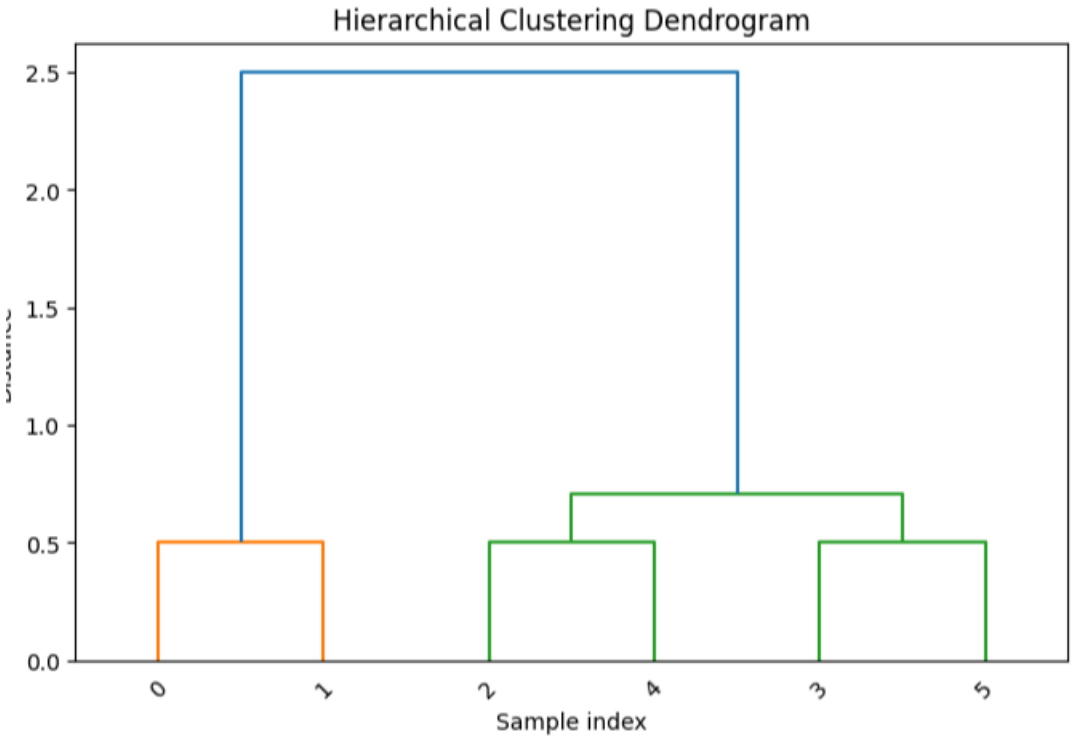
Pruning
-Usually decision trees are likely to overfit the data leading to poor test performance
better predictor at the cost of a little bias◊-Smaller tree + fewer splits
-Better solution: grow a large tree and then prune it back to a smaller subtree
„weakest link pruning"

Random forests
Why is it good?
they will become correlated◊If one or a few features are very strong predictors for the response variable (target output), these features will be selected in many of the decision trees
at some point the variance stops decreasing no matter how many more trees we add to our random forest + it is not going to produce overfitting !!!◊Huge advantage

Boosting
-It can be used for classification and regression too
-Helps to reduce variance and bias !!!
combining all the trees to make predictions◊ constructs several decision trees on the copies ◊-Bagging: creates multiple copies of the original data
-THESE TREES ARE INDEPENDENT FROM EACH OTHER !!!
each tree is grown using information from previously grown trees◊-Boosting: here the decision trees are grown sequentially

-THESE TREES ARE NOT INDEPENDENT FROM EACH OTHER !!!
-Can a set of weak learners create a single strong learner?
-Yes, we can turn a weak learner into a strong learner !!!
overfitting◊-Fit a large decision tree to the data
-The boosting algorithm learns slowly instead
-By fitting small trees we slowly improve the final result in cases when it does not perform well

```
import numpy as np
from scipy.cluster.hierarchy import linkage, dendrogram
import matplotlib.pyplot as plt

# Data points
x = np.array([[1,1],[1.5,1],[3,3],[4,4],[3,3.5],[3.5,4]])

# Scatter plot of data points
plt.scatter(x[:,0], x[:,1], s=50, color='blue')
plt.title("Data points")
plt.xlabel("X1")
plt.ylabel("X2")
plt.show()

# Compute hierarchical clustering (linkage)
linkage_matrix = linkage(x, method='single')  # "single" linkage

# Plot dendrogram
plt.figure(figsize=(8,5))
dendrogram(linkage_matrix, truncate_mode='none', leaf_rotation=45, leaf_font_size=10)
plt.title("Hierarchical Clustering Dendrogram")
plt.xlabel("Sample index")
plt.ylabel("Distance")
plt.show()
```



Support Vector Machines:

-Very popular and widely used supervised learning classification algorithm
-The great benefit: it can operates even in infinite dimensions !!!
between the data points in multidimensional space◊-It defines a margin / boundary
-Goal: find a flat boundary („hyperplane") that leads to a homogeneous partition of the data
-A good separation is achieved by the hyperplane that has the largest distance to the nearest training-data point of any class since in general the larger the margin the lower the generalization error of the classifier
-So we have to maximize the margin

-Can be applied to almost everything
-Classifications or numerical predictions
-Widely used in pattern recognition
-Identify cancer or genetic diseases
-Text classification: classify texts based on the language
-Detecting rare events: earthquakes or engine failures

Non-linear spaces
-In many real-world applications, the relationships between variables are non-linear
-A key feature of SVMs is their ability to map the problem into a higher dimensional space using a process known as the „kernel trick"
-Non-linear relationship may suddenly appears to be quite linear

$f(x) = \text{sign}(\sum_{i=1}^{N} \alpha_i y_i K(x_i, x) + b)$

Here:

- `Σ` → summation
- `α_i, y_i, x_i` → subscripted
- `K(x_i, x)` → kernel function
- `b` → bias term

Optional: Showing **Kernel types** in

1️⃣ Linear Kernel

$K(x_i, x_j) = x_i \cdot x_j$

2️⃣ Polynomial Kernel

$K(x_i, x_j) = (x_i \cdot x_j + c)^d$

- `c` → constant (can be 0 or 1)
- `d` → degree of the polynomial

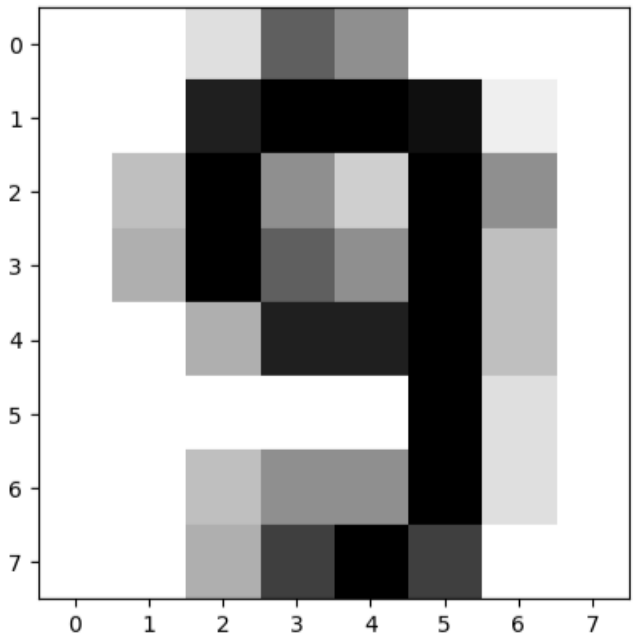
-SVMs with non-linear kernels add additional dimensions to the data in order to create separation in this way
process of adding new features that express mathematical relationships between measured characteristics◊-Kernel trick
-This allows the SVM to learn concepts that were not explicitly measured in the original data

Advantages
-SVM can be used for regression problems as well as for classifications
-Not overly influenced by noisy data
-Easier to use than neural networks
Disadvantages
-Finding the best model requires testing of various combinations of kernels and model parameters
especially when the input dataset has a large number of features◊-Quite slow
-Black box model: very hard to understand !!!

```
import matplotlib.pyplot as plt
from sklearn import datasets, svm, metrics
from sklearn.metrics import accuracy_score

# The digits dataset
digits = datasets.load_digits()
```

```
#print("Digits\n", digits)

images_and_labels = list(zip(digits.images, digits.target))

n_samples = len(digits.images)
data = digits.images.reshape((n_samples, -1))
#print("Data\n",data)

# Create a classifier: a support vector classifier
classifier = svm.SVC(gamma=0.001)

# We learn the digits on the first half of the digits
trainTestSplit = int(n_samples*0.75)
classifier.fit(data[:trainTestSplit], digits.target[:trainTestSplit])

# Now predict the value of the digit on the second half:
expected = digits.target[trainTestSplit:]
predicted = classifier.predict(data[trainTestSplit:])

#print("Classification report for classifier %s:\n%s\n"
#% (classifier, metrics.classification_report(expected, predicted)))
print("Confusion matrix:\n%s" % metrics.confusion_matrix(expected, predicted))
print(accuracy_score(expected, predicted))


# let's test on the last few images
plt.imshow(digits.images[-2], cmap=plt.cm.gray_r, interpolation='nearest')
print("Prediction for test image: ", classifier.predict(data[-2].reshape(1,-1)))

plt.show()
```

```
 [ 0  0  0  0  0 44  1  0  0  0]
 [ 0  0  0  0  0  0 47  0  0  0]
 [ 0  0  0  0  0  0  0 45  0  0]
 [ 0  1  0  0  0  0  0  0 40  0]
 [ 0  0  0  1  0  1  0  0  0 43]]
0.9666666666666667
Prediction for test image:  [9]
```



Logistics Regression

$p(y) = e^{(b_0 + b_1 x)} / (1 + e^{(b_0 + b_1 x)})$

The p(x) = P(default=1 | balance = x) is the probability of default when we know the balance !!!

It has a value between 0 and 1

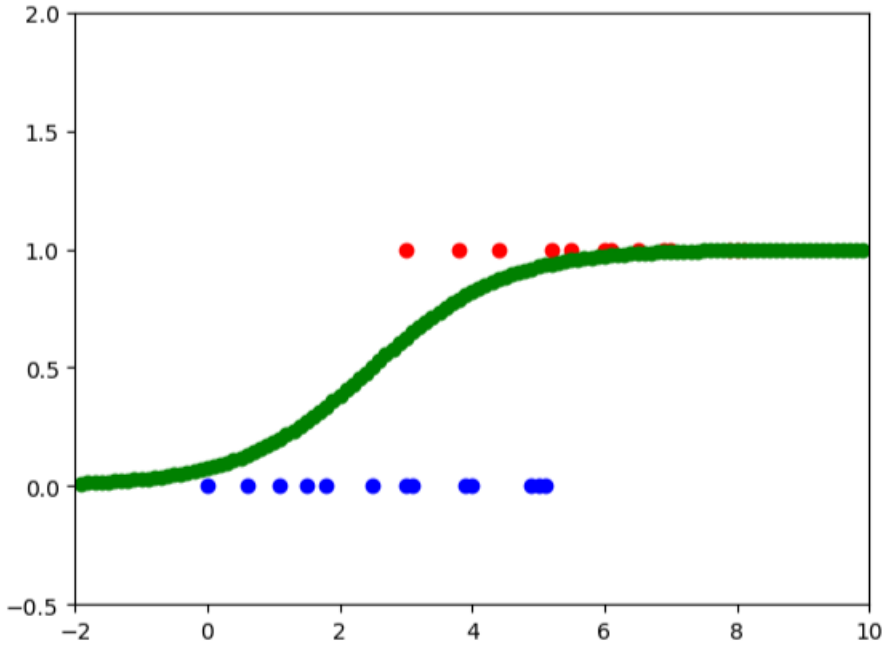
Logistic regression fits the b and b parameters, these are the regression parameters

This fitted curve is not linear: we can make it linear with the help of the logit transformation

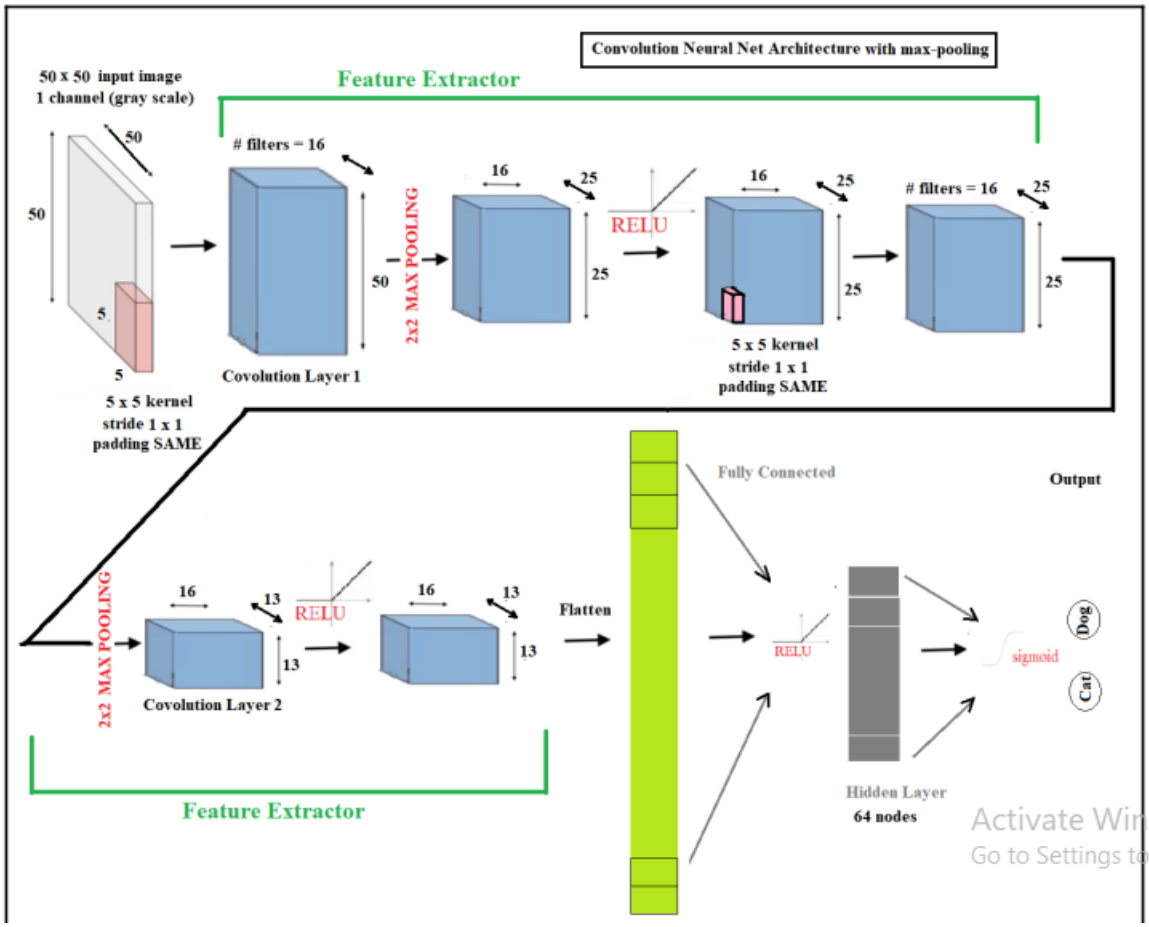
```
import numpy as np
```

```
from matplotlib import pyplot as plt
from sklearn.linear_model import LogisticRegression

# p i = 1 / 1 + exp[ - ( b0 + b1 * x )]

x1 = np.array([0,0.6,1.1,1.5,1.8,2.5,3,3.1,3.9,4,4.9,5,5.1])
y1 = np.array([0,0,0,0,0,0,0,0,0,0,0,0,0])

x2 = np.array([3,3.8,4.4,5.2,5.5,6.5,6,6.1,6.9,7,7.9,8,8.1])
y2 = np.array([1,1,1,1,1,1,1,1,1,1,1,1,1])

X =
np.array([[0],[0.6],[1.1],[1.5],[1.8],[2.5],[3],[3.1],[3.9],[4],[4.9],[5],[5.1],[3],[3.8]
,[4.4],[5.2],[5.5],[6.5],[6],[6.1],[6.9],[7],[7.9],[8],[8.1]])
y = np.array([0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,1,1,1,1,1,1,1,1])

plt.plot(x1,y1,'ro',color='blue')
plt.plot(x2,y2,'ro',color='red')

model = LogisticRegression()
model.fit(X,y)

print("b0 is:", model.intercept_)
print("b1 is:", model.coef_)

def logistic(classifier, x):
        return 1/(1+np.exp(-(model.intercept_ + model.coef_ * x)))

for i in range(1,120):
        plt.plot(i/10.0-2,logistic(model,i/10.0),'ro',color='green')

plt.axis([-2,10,-0.5,2])
plt.show()

# pred = model.predict_proba(1)
# print("Prediction: ", pred)
pred = model.predict_proba([[1]])
print("Prediction: ", pred)
```

```
b1 is: [[1.00401882]]
```




```
Prediction:  [[0.97061988 0.02938012]]
```



Introduction to Real-Time Use Cases:

Few Real time Ways

A fully convolutional model for detecting objects: YOLO (v2)

Deep segmentation with DeepLab (v3)

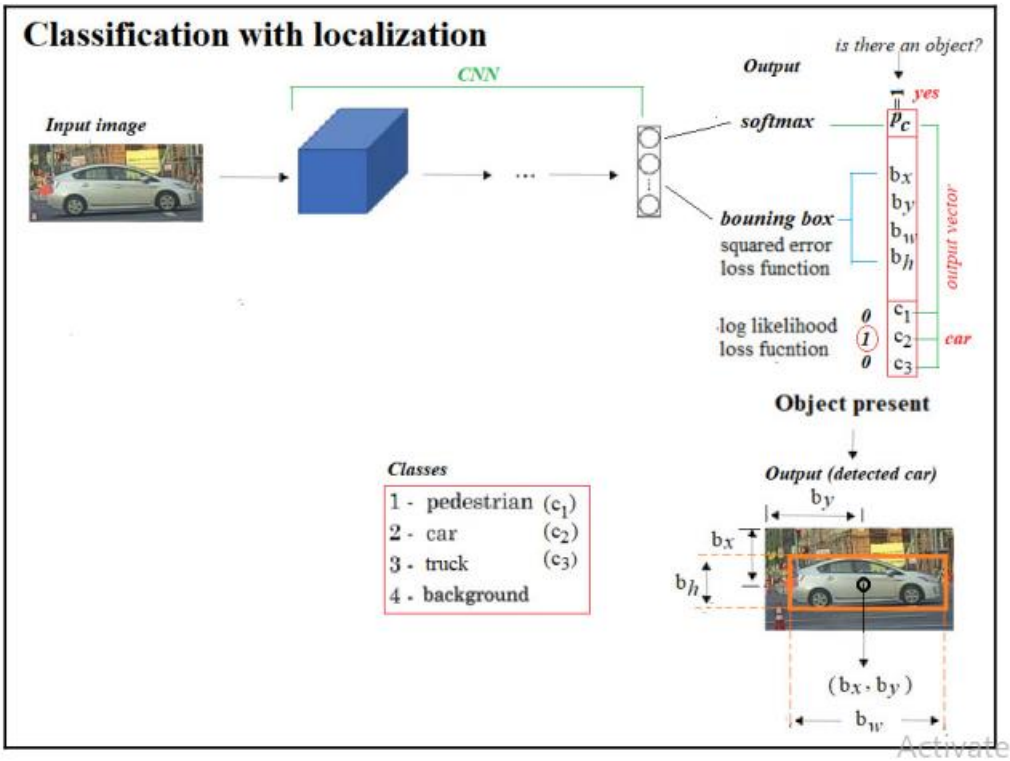
Transfer learning: what is it and when to use it

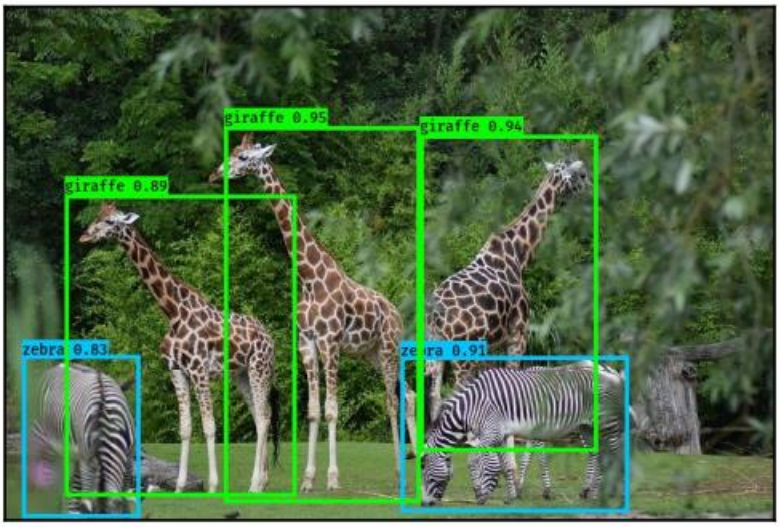
Deep style transfer with cv2 using a pretrained torch-based deep learning model

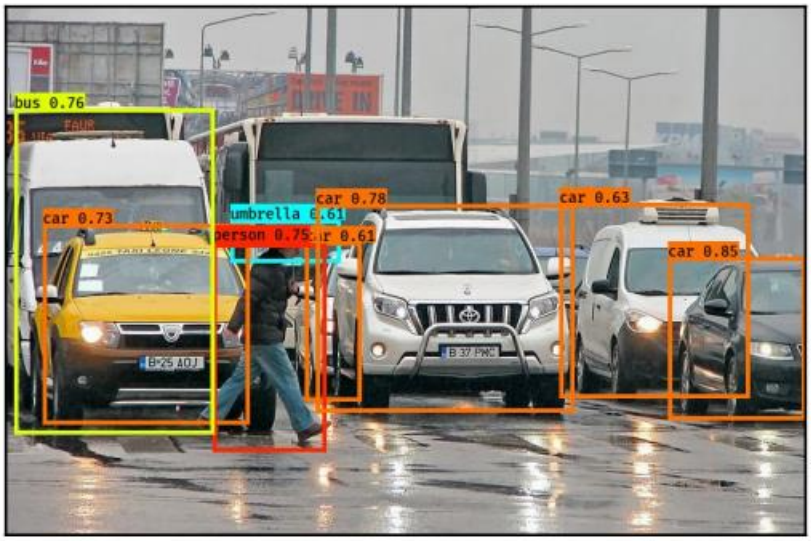
Introducing YOLO v2 :

YOLO, is a very popular and fully conventional algorithm that is used for detecting images. It gives a very high accuracy rate compared to other algorithms, and also runs in real time. As the name suggests, this algorithm looks only once at an image. This means that this algorithm requires only one forward propagation pass to make accurate predictions.

In this section, we will detect objects in images with a fully convolutional network (FCN) deep learning model. Given an image with some objects (for example, animals, cars, and so on), the goal is to detect objects in those images using a pre-trained YOLO model, with bounding boxes.







Deep semantic segmentation with DeepLab V3+

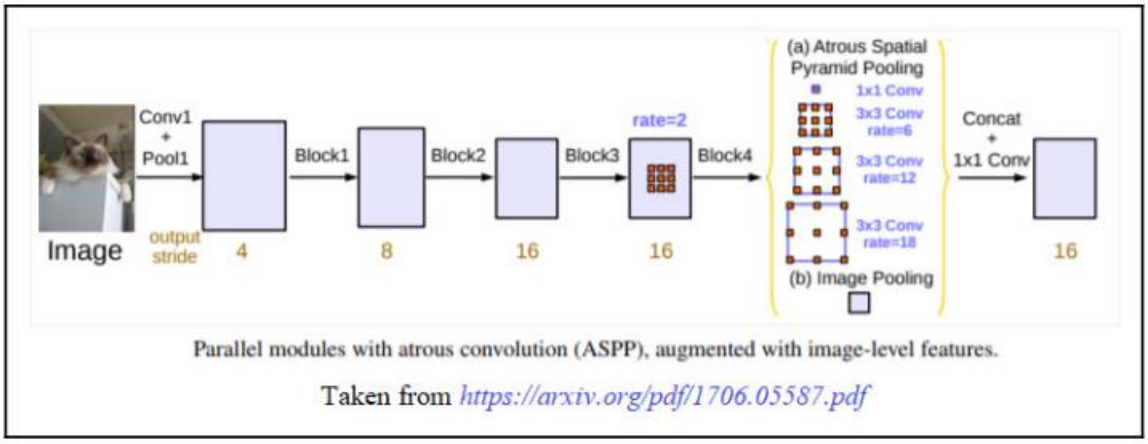
In this section, we'll discuss how to use a deep learning FCN to perform semantic segmentation of an image. Before diving into further details, let's clear the basic concepts.

Semantic segmentation

Semantic segmentation refers to an understanding of an image at pixel level; that is, when we want to assign each pixel in the image an object class (a semantic label). It is a natural step in the progression from coarse to fine inference. It achieves fine-grained inference by making dense predictions that infer labels for every pixel so that each pixel is labeled with the class of its enclosing object or region.

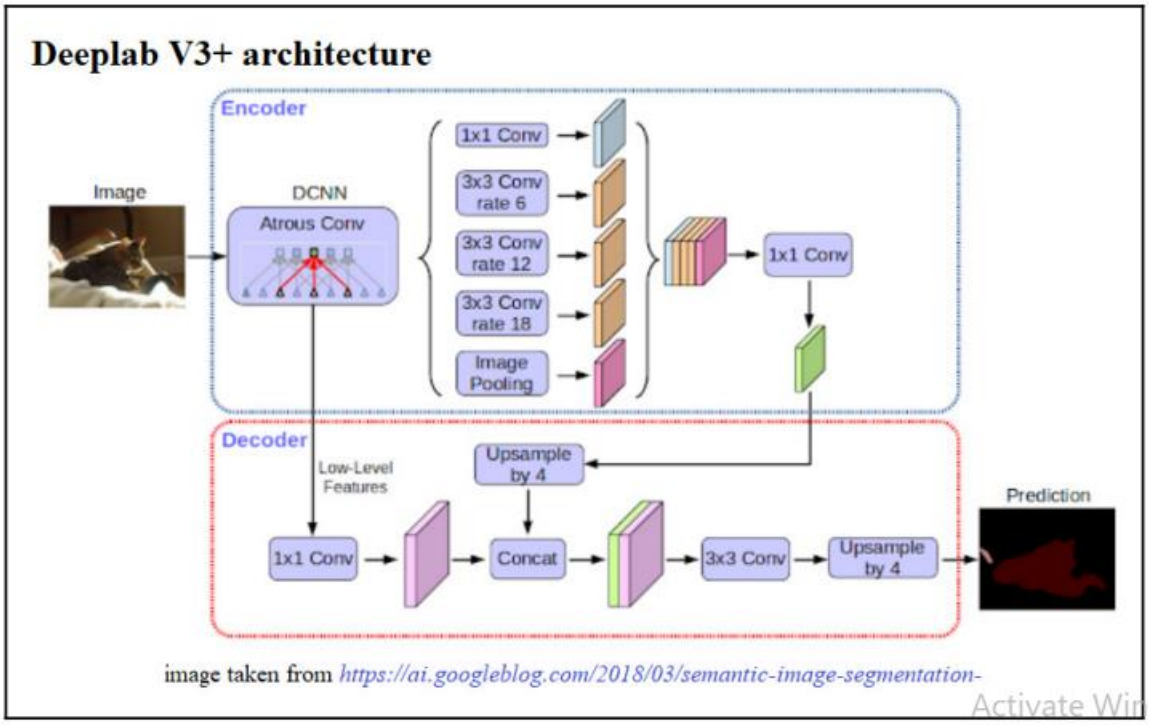
DeepLab v3 architecture

The image shows the parallel modules with atrous convolution:



Parallel modules with atrous convolution (ASPP), augmented with image-level features.

Taken from *https://arxiv.org/pdf/1706.05587.pdf*

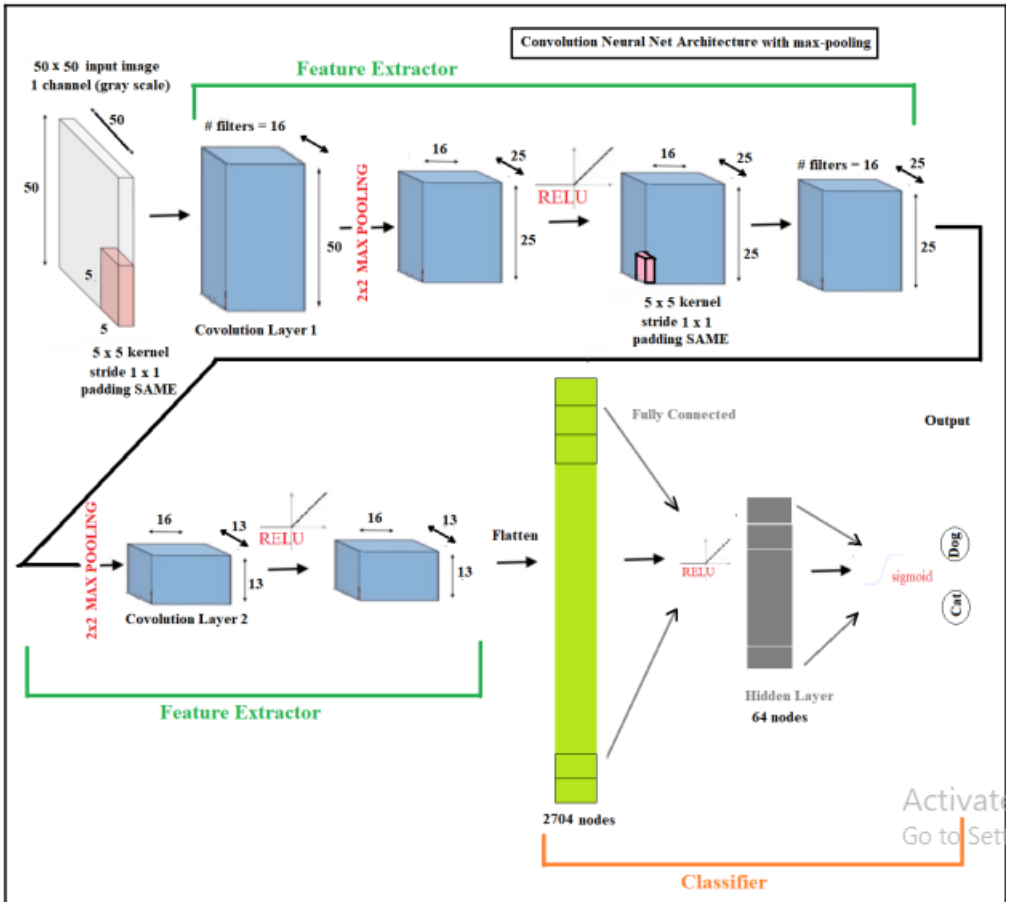


Transfer learning – what it is, and when to use it

Transfer learning is a deep learning strategy that reuses knowledge gained from solving one problem by applying it to a different, but related, problem. For example, let's say we have three types of flowers, namely, a rose, a sunflower, and a tulip. We can use the standard pre-trained models, such as VGG16/19, ResNet50, or InceptionV3 models (pretrained on ImageNet with 1000 output classes, which can be found at https://gist.github.com/yrevar/942d3a0ac09ec9e5eb3a) to classify the flower images, but our model wouldn't be able to correctly identify them because these flower categories were not learned by the models. In other words, they are classes that the model is not aware of.

Transfer learning with Keras

Training of pre-trained models is done on many comprehensive image classification problems. The convolutional layers act as a feature extractor, and the fully connected (FC) layers act as classifiers, as shown in the following diagram, in the context of cat vs. dog image classification with a conv net:


Convolution Neural Net Architecture with max-pooling
50 x 50 input image
1 channel (gray scale)
Feature Extractor
50
50
5
5
5 x 5 kernel
stride 1 x 1
padding SAME
filters = 16
50
Covolution Layer 1
2x2 MAX POOLING
16
25
25
RELU
16
25
25
5 x 5 kernel
stride 1 x 1
padding SAME
filters = 16
25
25
2x2 MAX POOLING
16
13
13
RELU
16
13
13
Covolution Layer 2
Feature Extractor
Flatten
2704 nodes
Fully Connected
RELU
Hidden Layer
64 nodes
sigmoid
Output
Dog
Cat
Classifier