Object Detection

Image classification involves assigning a class label to an image, whereas object localization involves drawing a bounding box around one or more objects in an image. Object detection is more challenging and combines these two tasks and draws a bounding box around each object of interest in the image and assigns them a class label. Together, all of these problems are referred to as object recognition.

As such, we can distinguish between these three computer vision tasks:

- **Image Classification**: Predict the type or class of an object in an image.
  Input: An image with a single object, such as a photograph.
  Output: A class label (e.g. one or more integers that are mapped to class labels).

- **Object Localization**: Locate the presence of objects in an image and indicate their location with a bounding box.
  Input: An image with one or more objects, such as a photograph.
  Output: One or more bounding boxes (e.g. defined by a point, width, and height).

- **Object Detection**: Locate the presence of objects with a bounding box and types or classes of the located objects in an image.
  Input: An image with one or more objects, such as a photograph.
  Output: One or more bounding boxes (e.g. defined by a point, width, and height), and a class label for each bounding box.

  One further extension to this breakdown of computer vision tasks is *object segmentation*, also called “object instance segmentation” or “semantic segmentation,” where instances of recognized objects are indicated by highlighting the specific pixels of the object instead of a coarse bounding box.

Today, there is a plethora of pre-trained models for object detection (YOLO, RCNN, Fast RCNN, Mask RCNN, Multibox etc.).

How computers learn patterns? Convolutions! (Look at the figure above while reading this) Convolution is a mathematical operation between two matrices to give a third matrix. The smaller matrix, which we call filter or kernel (3x3 in figure 1) is operated on the matrix of image pixels. Depending on the numbers in the filter matrix, the output matrix can recognize the specific patterns present in the input image

---

Object Localization

To make classification with localization we use a Conv Net with a softmax attached to the end of it and a four numbers bx, by, bh, and bw to tell you the location of the class in the image. The dataset should contain this four numbers with the class too.

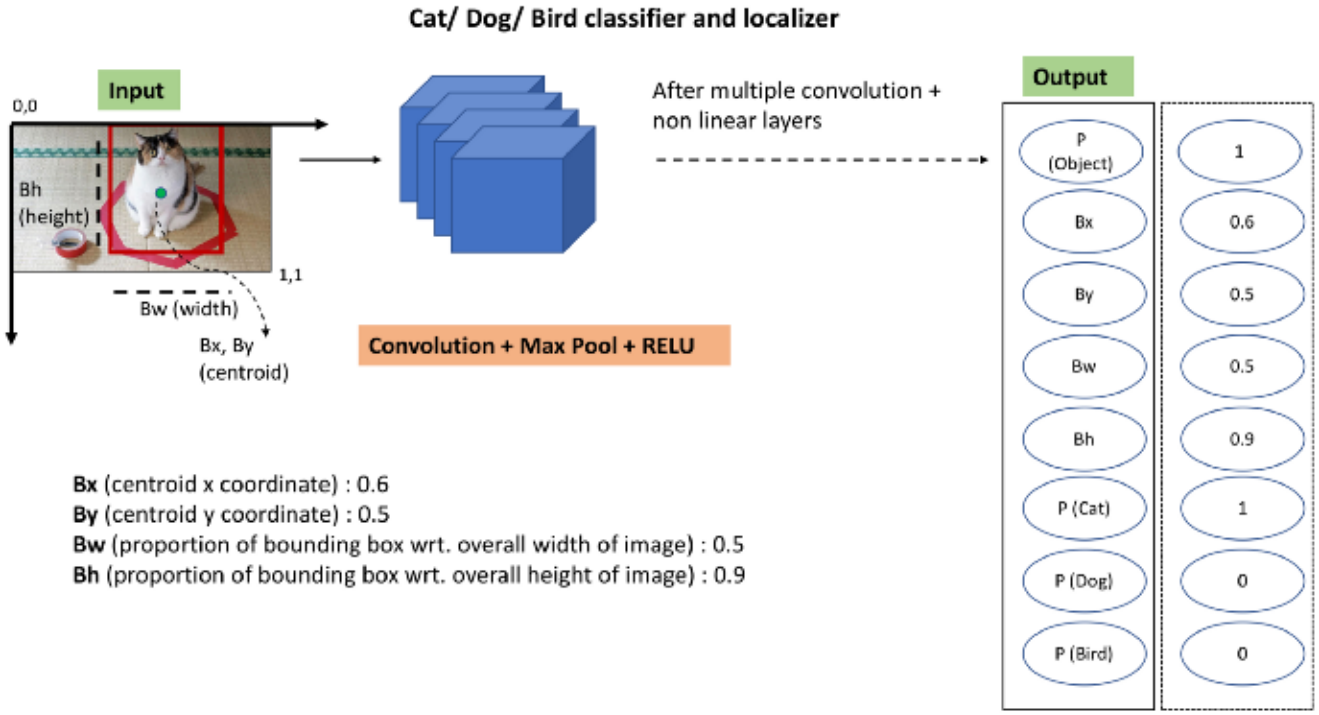
Defining the target label Y in classification with localization problem:

```
Y = [
        Pc                  # Probability of an object is presented
        bx                  # Bounding box
        by                  # Bounding box
        bh                  # Bounding box
```

```
                bw                                  # Bounding box
                c1                                  # The classes
                c2
                ...
]
```

- Example (Object is present):

```
○  Y = [
                1               # Object is present
                0
                0
                100
                100
                0
                1
                0
            ]
```



Src - https://towardsdatascience.com/evolution-of-object-detection-and-localization-algorithms-e241021d8bad

---

Landmark Detection

Face landmark detection is the process of finding points of interest in an image of a human face. It has recently seen rapid growth in the computer vision community because it has many compelling applications. For example, we have shown the ability to detect emotion through facial gestures, estimating gaze direction, changing facial appearance (**face swap**), augmenting faces with graphics, and puppeteering of virtual characters.

This is mostly used in face filter in mobiles nowadays. Emotion detection
Links to follow
https://towardsdatascience.com/tagged/facial-landmarks
https://medium.com/@rishiswethan.c.r/emotion-detection-using-facial-landmarks-and-deep-learning-b7f54fe551bf

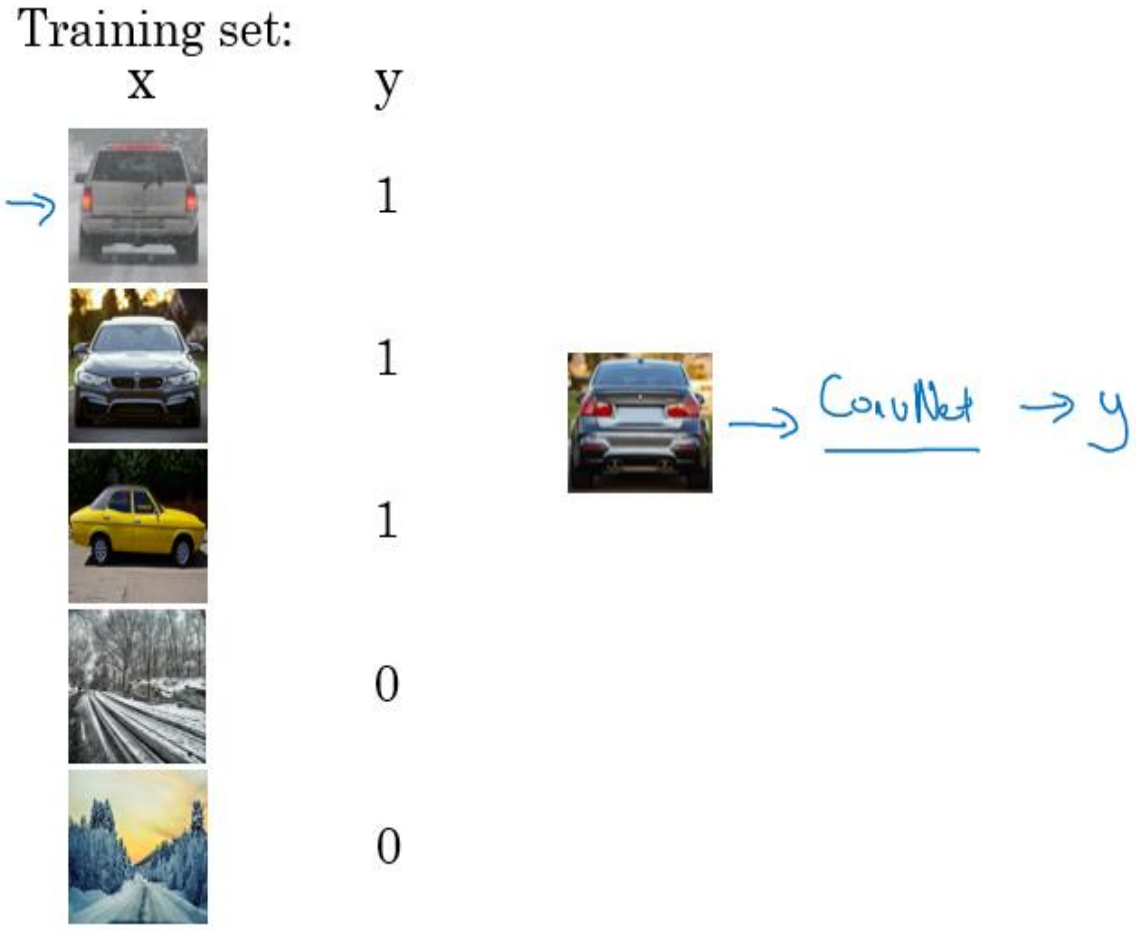
https://levelup.gitconnected.com/facial-landmark-detection-in-opencv4-616f9c1737a5

https://dzone.com/articles/drowsy-detection-using-facial-landmarks-extraction

---

Object Detection

- We will use a Conv net to solve the object detection problem using a technique called the sliding windows detection algorithm.
- For example, lets say we are working on Car object detection.
- The first thing, we will train a Conv net on cropped car images and non-car images.



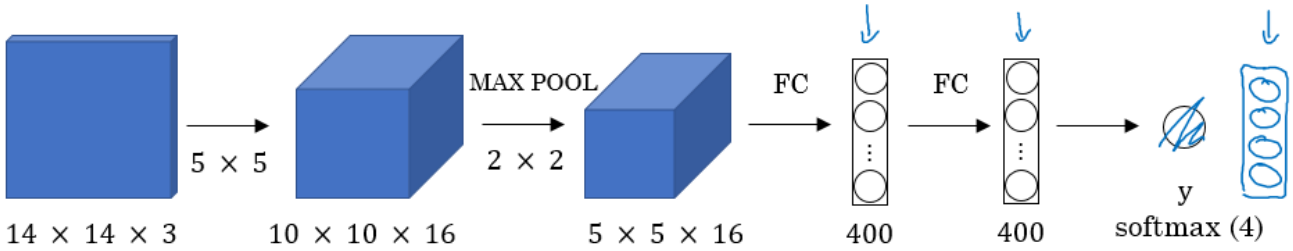
- After we finish training of this Conv net we will then use it with the sliding windows technique.
- Sliding windows detection algorithm:
  i. Decide a rectangle size.

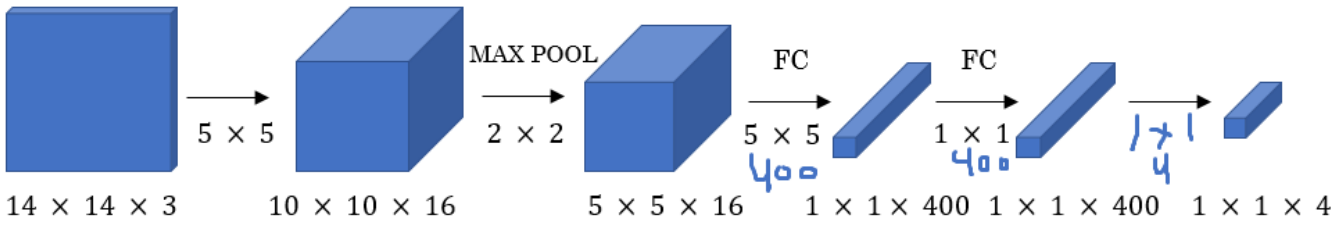
ii. Split your image into rectangles of the size you picked. Each region should be covered. You can use some strides.
iii. For each rectangle feed the image into the Conv net and decide if its a car or not.
iv. Pick larger/smaller rectangles and repeat the process from 2 to 3.
v. Store the rectangles that contains the cars.
vi. If two or more rectangles intersects choose the rectangle with the best accuracy.

- Disadvantage of sliding window is the computation time.
- In the era of machine learning before deep learning, people used a hand crafted linear classifiers that classifies the object and then use the sliding window technique. The linear classier make it a cheap computation. But in the deep learning era that is so computational expensive due to the complexity of the deep learning model.
- To solve this problem, we can implement the sliding windows with a Convolutional approach.

Convolutional Implementation of Sliding Windows

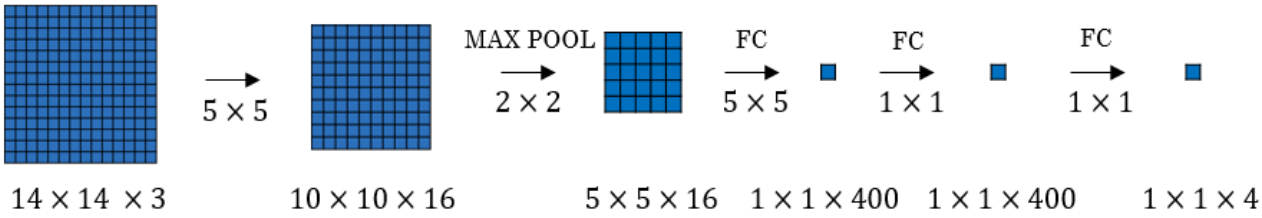
- Turning FC layer into convolutional layers (predict image class from four classes):



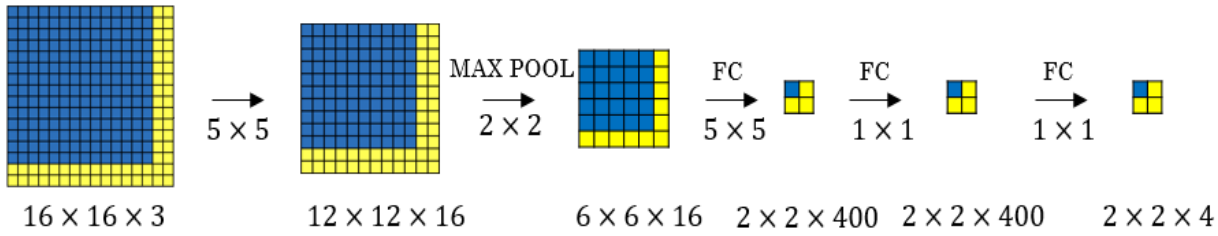


- As you can see in the above image, we turned the FC layer into a Conv layer using a convolution with the width and height of the filter is the same as the width and height of the input.

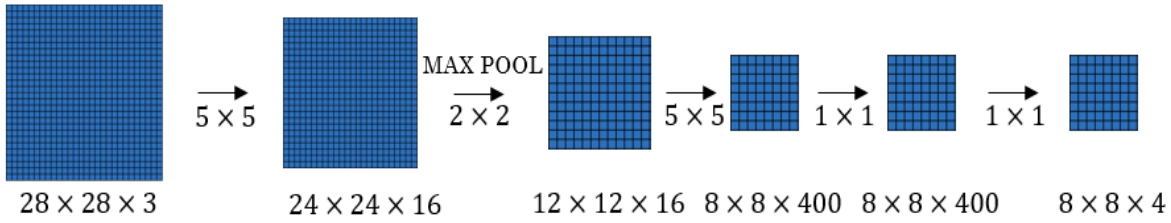
- **Convolution implementation of sliding windows**:
  - First lets consider that the Conv net you trained is like this (No FC all is conv layers):



- Say now we have a 16 x 16 x 3 image that we need to apply the sliding windows in. By the normal implementation that have been mentioned in the section before this, we would run this Conv net four times each rectangle size will be 16 x 16.
- The convolution implementation will be as follows:



- Simply we have feed the image into the same Conv net we have trained.
- The left cell of the result "The blue one" will represent the the first sliding window of the normal implementation. The other cells will represent the others.
- Its more efficient because it now shares the computations of the four times needed.
- Another example would be:



  - This example has a total of 16 sliding windows that shares the computation together.
  - [Sermanet et al., 2014, OverFeat: Integrated recognition, localization and detection using convolutional networks]
- The weakness of the algorithm is that the position of the rectangle wont be so accurate. Maybe none of the rectangles is exactly on the object you want to recognize.

YOLO

The YOLO model was first described by Joseph Redmon, et al. in the 2015 paper titled “You Only Look Once: Unified, Real-Time Object Detection.” Note that Ross Girshick, developer of R-CNN, was also an author and contributor to this work, then at Facebook AI Research.
The approach involves a single neural network trained end to end that takes a photograph as input and predicts bounding boxes and class labels for each bounding box directly.

YOLO (You Only Look Once), is a network for object detection. The object detection task consists in determining the location on the image where certain objects are present, as well as classifying those objects. Previous methods for this, like R-CNN and its variations, used a pipeline to perform this task in multiple steps. This can be slow to run and also hard to

optimize, because each individual component must be trained separately. YOLO, does it all with a single neural network

There are a few different algorithms for object detection, and they can be split into two groups:

1. **Algorithms based on classification** – they work in two stages. In the first step, we're selecting from the image interesting regions. Then we're classifying those regions using convolutional neural networks. This solution could be very slow because we have to run prediction for every selected region. Most known example of this type of algorithms is the Region-based convolutional neural network (RCNN) and their cousins Fast-RCNN and Faster-RCNN.
2. **Algorithms based on regression** – instead of selecting interesting parts of an image, we're predicting classes and bounding boxes for the whole image **in one run of the algorithm**. Most known example of this type of algorithms is **YOLO (You only look once)** commonly used for real-time object detection.

FROM THE YOLO PAPER -

We reframe object detection as a single regression problem, straight from image pixels to bounding box coordinates and class probabilities.

A single convolutional network simultaneously predicts multiple bounding boxes and class probabilities for those boxes. YOLO trains on full images and directly optimizes detection performance. This unified model has several benefits over traditional methods of object detection. First, YOLO is extremely fast. Since we frame detection as a regression problem we don't need a complex pipeline. We simply run our neural network on a new image at test time to predict detections. Our base network runs at 45 frames per second with no batch processing on a Titan X GPU and a fast version runs at more than 150 fps. This means we can process streaming video in real-time with less than 25 milliseconds of latency.

Second, YOLO reasons globally about the image when making predictions. Unlike sliding window and region proposal-based techniques, YOLO sees the entire image during training and test time so it implicitly encodes contextual information about classes as well as their appearance. Fast R-CNN, a top detection method, mistakes background patches in an image for objects because it can't see the larger context. YOLO makes less than half the number of background errors compared to Fast R-CNN.

Third, YOLO learns generalizable representations of objects. When trained on natural images and tested on artwork, YOLO outperforms top detection methods like DPM and R-CNN by a wide margin. Since YOLO is highly generalizable it is less likely to break down when applied to new domains or unexpected inputs.

Our network uses features from the entire image to predict each bounding box. It also predicts all bounding boxes across all classes for an image simultaneously. This means our network reasons globally about the full image and all the objects in the image. The YOLO design enables end-to-end training and realtime speeds while maintaining high average precision.

Our system divides the input image into an S × S grid. If the center of an object falls into a grid cell, that grid cell is responsible for detecting that object.

Each grid cell predicts B bounding boxes and confidence scores for those boxes. These confidence scores reflect how confident the model is that the box contains an object and also how accurate it thinks the box is that it predicts. Formally we define confidence as Pr(Object) ∗ IOU . If no object exists in that cell, the confidence scores should be zero. Otherwise we want the confidence score to equal the intersection over union (IOU) between the predicted box and the ground truth

Each bounding box consists of 5 predictions: x, y, w, h, and confidence. The (x, y) coordinates represent the center of the box relative to the bounds of the grid cell. The width and height are predicted relative to the whole image. Finally the confidence prediction represents the IOU between the predicted box and any ground truth box. Each grid cell also predicts C conditional class probabilities, Pr(Classi |Object). These probabilities are conditioned on the grid cell containing an object. We only predict one set of class probabilities per grid cell, regardless of the number of boxes B.

At test time we multiply the conditional class probabilities and the individual box confidence predictions,

$$Pr(Classi|Object)*Pr(Object)*IOU = Pr(Classi)*IOU$$

, which gives us class-specific confidence scores for each box. These scores encode both the probability of that class appearing in the box and how well the predicted box fits the object

Implementations of YOLO

Currently there are 3 main implementations of YOLO, each one of them with advantages and disadvantages

1. **Darknet** (https://pjreddie.com/darknet/). This is the "official" implementation, created by the same people behind the algorithm. It is written in C with CUDA, hence it supports GPU computation. It is actually a complete neural network framework, so it really can be used for other objectives besides YOLO detection. The disadvantage is that, since it is written from the ground up (not based on a stablished neural network framework) it may be more difficult to find answers for errors you might encounter (happened to me more than once).

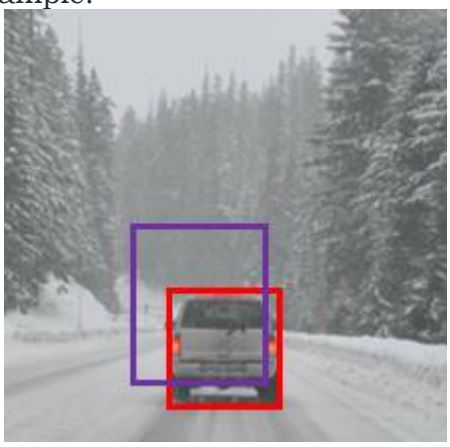
2. **AlexeyAB/darknet** (https://github.com/AlexeyAB/darknet). Here I am actually cheating a little bit because it is actually a fork of Darknet to support Windows and Linux. I haven't actually used this one, but I have checked the README many times, it is an excellent source to find tips and recommendations about YOLO in general, how to prepare you training set, how to train the network, how to improve object detection, etc.

3. **Darkflow** (https://github.com/thtrieu/darkflow/). This is port of Darknet to work over TensorFlow. This is the system I have used the most, mainly because I started this project without having a GPU to train the network and apparently using CPU-only

Darkflow is several times faster than the original Darkent. AFAIK the main disadvantage is that it has not been updated to YOLOv3.

All these implementations come “ready to use”, which means you only need to download and install them to start detecting images or videos right away using already trained weights available to download. Naturally this detection will be limited to classes contained in the datasets used to obtain this weights.

Intersection Over Union

- Intersection Over Union is a function used to evaluate the object detection algorithm.
- It computes size of intersection and divide it by the union. More generally, *IoU is a measure of the overlap between two bounding boxes*.
- For example:
  - 
  - The red is the labeled output and the purple is the predicted output.
  - To compute Intersection Over Union we first compute the union area of the two rectangles which is "the first rectangle + second rectangle" Then compute the intersection area between these two rectangles.
  - Finally IOU = intersection area / Union area
- If IOU >=0.5 then its good. The best answer will be 1.
- The higher the IOU the better is the accuracy.

Links

https://machinelearningmastery.com/object-recognition-with-deep-learning/

https://github.com/mbadry1/DeepLearning.ai-Summary/tree/master/4-%20Convolutional%20Neural%20Networks#object-detection

https://www.analyticsvidhya.com/blog/2018/12/practical-guide-object-detection-yolo-framewor-python/

Face Recognition

One-shot Learning

One Shot Learning: A recognition system is able to recognize a person, learning from one image. The idea here is that we need to learn an object class from only a few data

We make this work with a **similarity function**:

- d(**img1**, **img2**) = degree of difference between images.
- We want d result to be low in case of the same faces.
- We use tau T as a threshold for d:
  - If d(**img1**, **img2**) <= T Then the faces are the same.

Similarity function helps us solving the one shot learning. Also its robust to new inputs.

In face recognition systems, we want to be able to recognize a person's identity by just feeding one picture of that person's face to the system. And, in case, it fails to recognize the picture, it means that this person's image is not stored in the system's database.

To solve this problem, we cannot use only a convolutional neural network for two reasons:

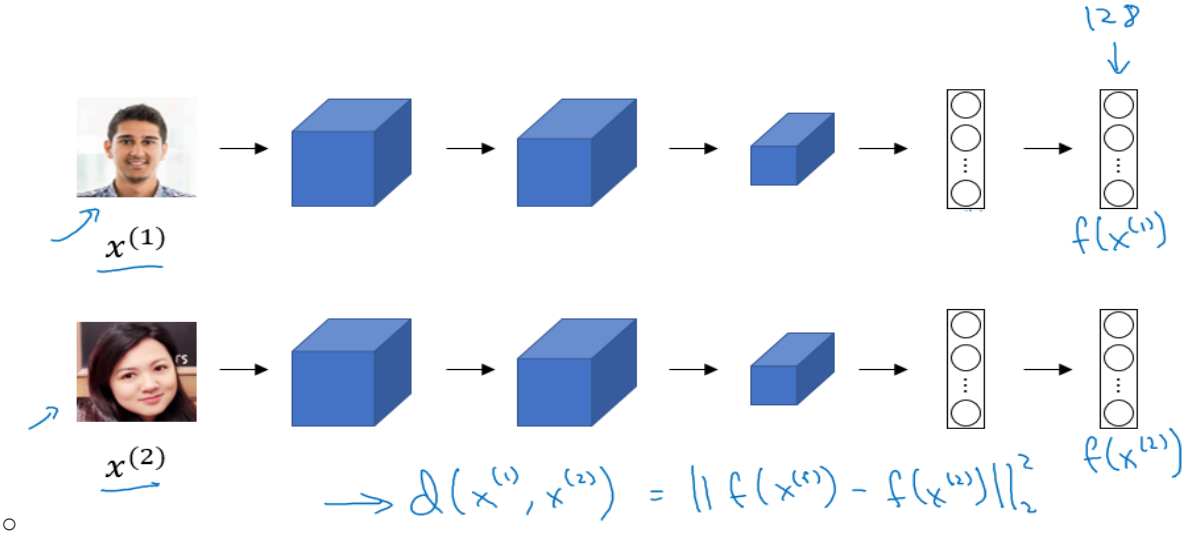
1) CNN doesn't work on a small training set.

2) It is not convenient to retrain the model every time we add a picture of a new person to the system. However, we can use Siamese neural network for face recognition.

Siamese network

We will implement the similarity function using a type of NNs called Siamease Network in which we can pass multiple inputs to the two or more networks with the same architecture and parameters.

Siamese network architecture are as the following:



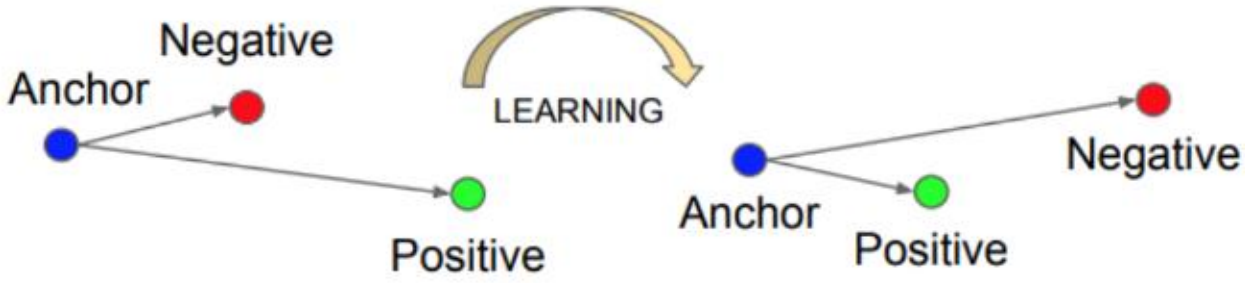
- We make 2 identical conv nets which encodes an input image into a vector. In the above image the vector shape is (128,)
- The loss function will be d(x1, x2) = || f(x1) - f(x2) ||^2
- If X1, X2 are the same person, we want d to be low. If they are different persons, we want d to be high.

- And this is working for any two images *xi* and *xj*.

If $x^{(i)}, x^{(j)}$ are the same person, $\left\|f\left(x^{(i)}\right)-f\left(x^{(j)}\right)\right\|^2$ is small.
If $x^{(i)}, x^{(j)}$ are different persons, $\left\|f\left(x^{(i)}\right)-f\left(x^{(j)}\right)\right\|^2$ is large.

So, how can we learn the parameters in order to get a good encoding for the input image?

We can apply gradient descent on a **triplet loss function** which is simply a loss function using **three** images: an anchor image A, a positive image P(same person as the anchor), as well as a negative image N (different person than the anchor). So, we want the distance d(A, P) between the encoding of the anchor and the encoding of the positive example **to be less than or equal to** the distance d(A, N) between the encoding of the anchor and the encoding of the negative example. In other words, we want pictures of the same person to be close to each other, and pictures of different persons to be far from each other.



The Triplet Loss minimizes the distance between an anchor and a positive, both of which have the same identity, and maximizes the distance between the anchor and a negative of a different identity. from the paper **FaceNet: A Unified Embedding for Face Recognition and Clustering**

The problem here is that the model can learn to make the same encoding for different images, which means that distances will be zero, and unfortunately, it will satisfy the triplet loss function. For this reason, we are adding a margin alpha (hyperparameter), to prevent this from happening, and to always have a gap between A and P versus A and N.

$$d(A,P) + \alpha \leq d(A,N)$$

$$\|f(A)-f(P)\|^2 + \alpha \leq \|f(A)-f(N)\|^2$$

$$\boxed{\|f(A)-f(P)\|^2 - \|f(A)-f(N)\|^2 + \alpha \leq 0}$$

Triplet loss function:

$$L(A,P,N) = \mathbf{max}\left(\|f(A)-f(P)\|^2 - \|f(A)-f(N)\|^2 + \alpha,\ 0\right)$$

The max means as long as d(A, P) — d(A, N)+ alpha is less than or equal to zero, the loss L(A, P, N) is zero, but if it is greater than zero, the loss will be positive, and the function will try to minimize it to zero or less than zero.

The Cost function is the sum of all individual losses on different triplets from all the training set.

$$\text{Cost function: } J = \sum_{i=1}^{n} L(A^{(i)}, P^{(i)}, N^{(i)})$$

Training set:

The training set should contain **multiple pictures of the same person** to have the pairs A and P, then once the model is trained, we'll be able to recognize a person with only one picture.

How do we choose the triplets to train the model?

If we choose them randomly, it will be so easy to satisfy the constraint of the loss function because the distance is going to be most of the time so large. And the gradient descent will not learn much from the training set. For this reason, we need to find A, P, and N so that A

and P are so close to N. Our objective is to make it harder to train the model to push the gradient descent to learn more.

Available implementations for face recognition using deep learning includes:

- Openface
- FaceNet
- DeepFace

What is neural style transfer?

Neural style transfer is one of the application of Conv nets.

Neural style transfer takes a content image C and a style image S and generates the content image G with the style of style image.

Cost Function

First, let's look at the cost function needed to build a neural style transfer algorithm. Minimizing this cost function will help in getting a better generated image (G). Defining a cost function:

$$J(G) = \alpha * J_{Content}(C,G) + \beta * J_{Style}(S,G)$$

Here, the content cost function ensures that the generated image has the same content as that of the content image whereas the generated cost function is tasked with making sure that the generated image is of the style image fashion.

Below are the steps for generating the image using the content and style images:

1. We first initialize G randomly, say G: 100 X 100 X 3, or any other dimension that we want
2. We then define the cost function J(G) and use gradient descent to minimize J(G) to update G:
   $G = G - d/dG(J(G))$

Suppose the content and style images we have are:

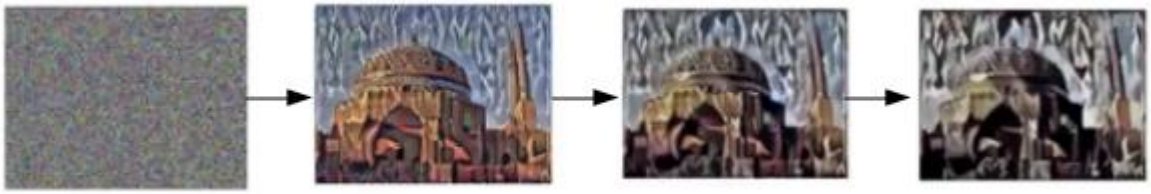
Content

Style

First, we initialize the generated image:

Initialized
generated image

After applying gradient descent and updating G multiple times, we get something like this:



Not bad! This is the outline of a neural style transfer algorithm. It's important to understand both the content cost function and the style cost function in detail for maximizing our algorithm's output.

Content Cost Function

Suppose we use the *lth* layer to define the content cost function of a neural style transfer algorithm. Generally, the layer which is neither too shallow nor too deep is chosen as the *lth* layer for the content cost function. We use a pretrained ConvNet and take the activations of its *lth* layer for both the content image as well as the generated image and compare how similar their content is. With me so far?

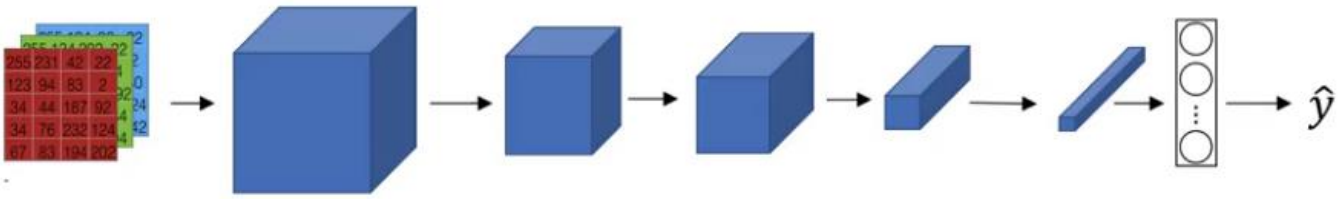
Now, we compare the activations of the *lth* layer. For the content and generated images, these are $a^{[l](C)}$ and $a^{[l](G)}$respectively. If both these activations are similar, we can say that the images have similar content. Thus, the cost function can be defined as follows:

$$J_{Content}(C,G) = \frac{1}{2} * || a^{[l](C)} - a^{[l](G)} ||^2$$

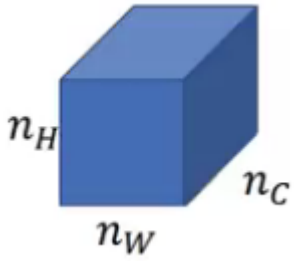
We try to minimize this cost function and update the activations in order to get similar content. Next, we will define the style cost function to make sure that the style of the generated image is similar to the style image.

Style Cost Function

Suppose we pass an image to a pretrained ConvNet:



We take the activations from the *lth* layer to measure the style. We define the style as the correlation between activations across channels of that layer. Let's say that the *lth* layer looks like this:



We want to know how correlated the activations are across different channels:

$a_{i,j,k}$ = activations at (i,j,k)

Here, i is the height, j is the width, and k is the channel number. We can create a correlation matrix which provides a clear picture of the correlation between the activations from every channel of the *lth* layer:

$$G^{[l](S)}_{kk'} = \sum_{i=1}^{n_H^{[l]}} \sum_{j=1}^{n_W^{[l]}} a^{[l](S)}_{ijk} \, a^{[l](S)}_{ijk'}$$

where k and k' ranges from 1 to $n_c^{[l]}$. This matrix is called a **style matrix**. If the activations are correlated, $G_{kk'}$ will be large, and vice versa. S denotes that this matrix is for the style image. Similarly, we can create a style matrix for the generated image:

$$G^{[l](G)}_{kk'} = \sum_{i=1}^{n_H^{[l]}} \sum_{j=1}^{n_W^{[l]}} a^{[l](G)}_{ijk} \, a^{[l](G)}_{ijk'}$$

Using these two matrices, we define a style cost function:

$$J^{[l]}_{style}(S,G) = \frac{1}{\left(2n_H^{[l]} n_W^{[l]} n_C^{[l]}\right)^2} \sum_k \sum_{k'} \left(G^{[l](S)}_{kk'} - G^{[l](G)}_{kk'}\right)$$

This style cost function is for a single layer. We can generalize it for all the layers of the network:

Finally, we can combine the content and style cost function to get the overall cost function:

$$J(G) = \alpha * J_{Content}(C,G) + \beta * J_{Style}(S,G)$$

References

https://towardsdatascience.com/one-shot-learning-with-siamese-networks-using-keras-17f34e75bb3d

https://towardsdatascience.com/one-shot-learning-face-recognition-using-siamese-neural-network-a13dcf739e

https://www.analyticsvidhya.com/blog/2018/12/guide-convolutional-neural-network-cnn/