Face Detection using Convolutional Neural Networks

By Sai Prashant Chintalapudi and Anirudh Ravishankar

Part -1: Survey

Face Detection

Face detection is a necessary first-step in face recognition systems, with the purpose of localizing and extracting the face region from the background. It also has several applications in areas such as content-based image retrieval, video coding, video conferencing, crowd surveillance, facial expression recognition, face tracking, facial feature extraction, gender classification, identification system, document control and access control, clustering, biometric science, human computer interaction (HCI) system, digital cosmetics and many more. The human face is a dynamic object and has a high degree of variability in its appearance, which makes face detection a difficult problem in computer vision. A wide variety of techniques have been proposed, ranging from simple edge-based algorithms to composite high-level approaches utilizing advanced pattern recognition methods.

A face detector should tell whether an image of arbitrary size contains a human face and if so, where it is present in the image. One natural framework for considering this problem is that of binary classification, in which a classifier is constructed to minimize the misclassification risk. Since no objective distribution can describe the actual prior probability for a given image to have a face, the algorithm must minimize both the false negative and false positive rates to achieve an acceptable performance. This task requires an accurate numerical description of what sets human faces apart from other objects.

The Viola - Jones face detection framework is the first face detection framework to provide competitive face detection rates in real-time proposed in 2001 by Paul Viola and Michael Jones.

Convolutional Neural Networks for Face Detection

Ever since the seminal work of Paul Viola and Michael Jones, the boosted cascade with simple features became the most popular and effective design for practical face detection. The simple nature of the features enables fast evaluation and quick early rejection of false positive detections. Meanwhile, the boosted cascade constructs an ensemble of the simple features to achieve accurate face vs. non-face classification. The original Viola-Jones face detector uses the Haar feature which is fast to evaluate yet discriminative enough for frontal faces. However, due to the simple nature of the Haar feature, it is relatively weak in the uncontrolled environment where faces are in varied poses, expressions under unexpected lighting. Several improvements to the Viola-Jones face

detector have been proposed in the past decade. Most of them follow the boosted cascade framework with more advanced features. The advanced feature helps construct a more accurate binary classifier at the expense of extra computation. However, the number of cascade stages required to achieve the similar detection accuracy can be reduced. This observation suggests that it is possible to apply more advanced features in a practical face detection solution if the false positive detections can be rejected quickly in the early stages. In this project, we try to apply the Convolutional Neural Network (CNN) to face detection. Compared with the hand-crafted features other methods require, CNNs can automatically learn features to capture complex visual variations by leveraging a large amount of training images.

Convolutional Neural Networks

Convolutional Neural Networks are very similar to ordinary Neural Networks: they are made up of neurons that have learnable weights and biases. Each neuron receives some inputs, performs a dot product and optionally follows it with a non-linearity. The whole network still expresses a single differentiable score function: from the raw image pixels on one end to class scores at the other. The main difference is that CNN architectures make the explicit assumption that the inputs are images, which allows us to encode certain properties into the architecture. These then make the forward function more efficient to implement and vastly reduce the number of parameters in the network.

Neural networks receive an input (a single vector), and transform it through a series of hidden layers. Each hidden layer is made up of a set of neurons, where each neuron is fully connected to all neurons in the previous layer, and where neurons in a single layer function completely independently and do not share any connections. The last fully-connected layer is called the “output layer” and in classification settings it represents the class scores. Regular neural nets don’t scale well to full images.

Unlike a regular Neural Network, the layers of a CNN have neurons arranged in three dimensions: width, height, depth. They perceive images as volumes; i.e. three-dimensional objects, rather than flat canvases to be measured only by width and height. That’s because digital colour images have a red-blue-green (RGB) encoding, mixing those three colours to produce the colour spectrum humans perceive. A convolutional neural network ingests such images as three separate layers of colour stacked one on top of the other.

So, a convolutional network receives a normal colour image as a rectangular box whose width and height are measured by the number of pixels along those dimensions, and whose depth is three layers deep, one for each letter in RGB. Those depth layers are referred to as channels.

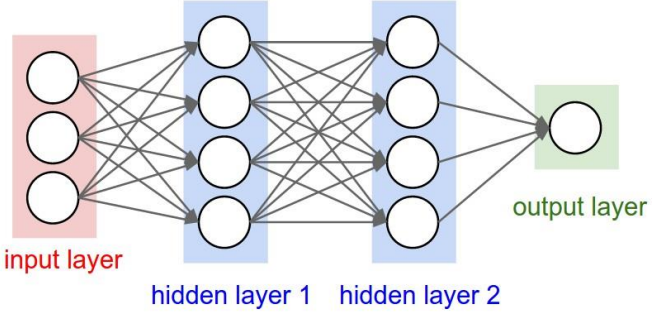

Fig 1. A regular 3-layer Neural Network.

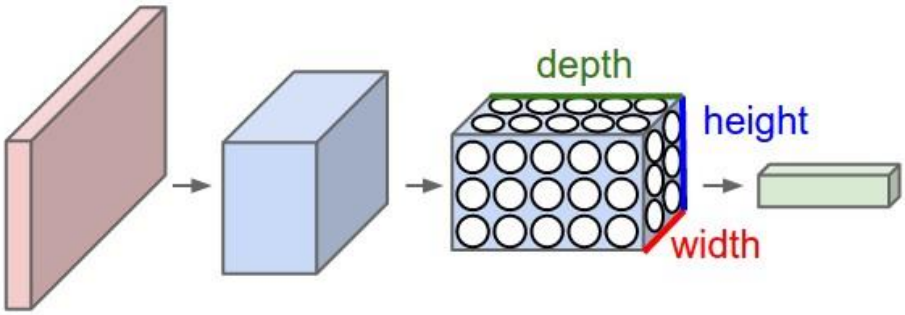

Fig 2. A CNN arranges its neurons in three dimensions (width, height, depth), as visualized in one of the layers. Every layer of a CNN transforms the 3D input volume to a 3D output volume of neuron activations.

Essentially, every image can be represented as a matrix of pixel values. An image from a standard digital camera will have three channels – red, green and blue. A grayscale image, on the other hand, has just one channel. The value of each pixel in the matrix will range from 0 to 255 – zero indicating black and 255 indicating white.

There are four main operations which are the basic building blocks of every CNN:

1) Convolution

CNNs derive their name from the "convolution" operator. The primary purpose of a convolution in case of a CNN is to extract features from the input image. Convolution preserves the spatial relationship between pixels by learning image features using small squares of input data.

The CONV layer's parameters consist of a set of learnable filters. Every filter is small spatially (along width and height), but extends through the full depth of the input volume. During the forward pass, we convolve each filter across the width and height of the input volume and compute dot products between the entries of the filter and the input at any position. As we slide the filter over the width and height of the input volume we will produce a 2D activation map that gives the responses of that filter at every spatial position as shown in Fig 3. Now, we will have an entire set of filters in each CONV layer, and each of them will produce a separate 2D activation map. We will stack these activation maps along the depth dimension and produce the output volume.

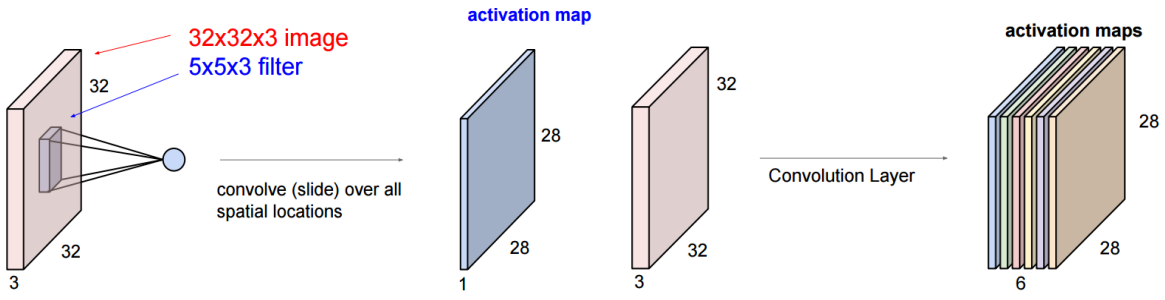

Fig 3. Stacking of activation maps as an output from a single convolutional layer.

In practice, a CNN learns the values of the filters on its own during the training process (although we still need to specify parameters such as number of filters, filter size, architecture of the network etc. before the training process). The more number of filters we have, the more image features get extracted and the better our network becomes at recognizing patterns in unseen images.

2) Non- Linearity (ReLU)

An additional operation called ReLU is used after every Convolution operation. ReLU stands for Rectified Linear Unit and is a non-linear operation. ReLU is an element wise operation (applied per pixel) and replaces all negative pixel values in the feature map by zero. The purpose of ReLU is to introduce non-linearity in our CNN, since most of the real-world data we would want our CNN to learn would be non-linear (Convolution is a linear operation – element wise matrix multiplication and addition, so we account for non-linearity by introducing a non-linear function like ReLU). Other nonlinear functions such as **tanh** or **sigmoid** can also be used instead of ReLU, but ReLU has been found to perform better in most situations.

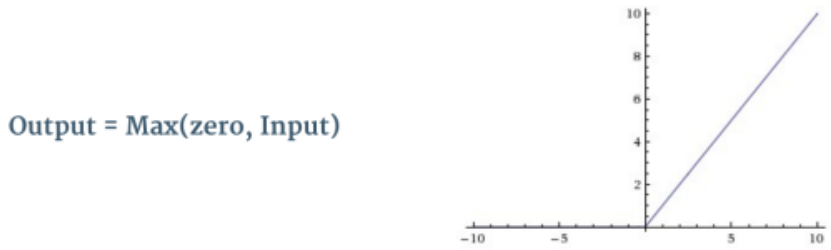

Fig 4. The Rectified Linear Unit (ReLU) function.

3) Pooling or Sub Sampling

Spatial Pooling (also called subsampling or down - sampling) reduces the dimensionality of each feature map but retains the most important information. Spatial Pooling can be of different types: Max, Average, Sum etc. In case of Max Pooling, we define a spatial neighbourhood (for example, a 2×2 window) and take the largest element from the rectified feature map within that window. Instead of taking the largest element we could also take the average (Average Pooling) or sum of all elements in that window. In practice, Max Pooling has been shown to work better.

Pooling makes the input representations (feature dimension) smaller and more manageable. It reduces the number of parameters and computations in the network, therefore, controlling overfitting. It also makes the network invariant to small transformations, distortions and translations in the input image (a small distortion in input will not change the output of Pooling – since we take the maximum / average value in a local neighbourhood). It also helps us arrive at an almost scale invariant representation of our image. This is very powerful since we can detect objects in an image no matter where they are located.

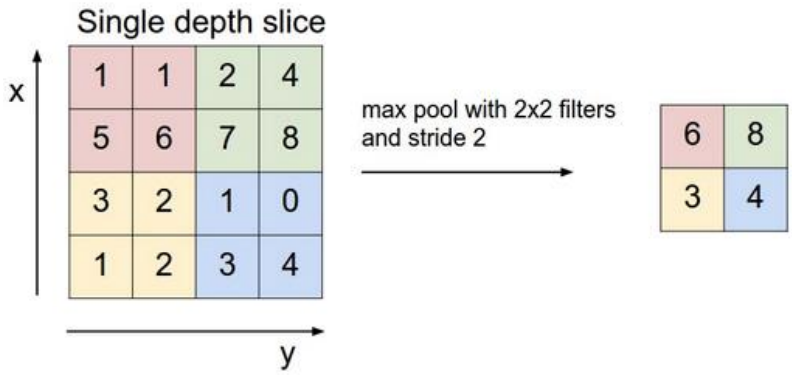

Fig 5. MaxPooling with a 2x2 filter.

4) Classification (Fully Connected Layer)

The Fully Connected layer is a traditional multi-layer perceptron that uses a SoftMax activation function in the output layer. The term "Fully Connected" implies that every neuron in the previous layer is connected to every neuron on the next layer. The output from the convolutional and pooling layers represent high-level features of the input image. The purpose of the fully connected layer is to use these features for classifying the input image into various classes based on the training dataset. The sum of output probabilities from the fully connected Layer is 1. This is ensured by using the SoftMax as the activation function in the output layer of the fully connected Layer. The SoftMax function takes a vector of arbitrary real-valued scores and squashes it to a vector of values between zero and one.

Part-2: Development

As mentioned in the section earlier, we try to train our own Convolutional Neural Network to detect faces in an image. We used the Labeled Faces in the Wild dataset provided by UMass as the positive samples for training. There are a total of 14,103 images in this dataset. As for the negative samples, we collected images from various sources like the negative training set of the INRIA dataset, the Caltech 101 dataset and the Street View House Numbers (SVHN) dataset. Since we wanted our training dataset to be balanced, we considered 14,103 images for the negative samples as well.

These images were stored in a subdirectory named "images" in the working directory. The positive samples were stored in a subdirectory "face" in the images directory and the negative samples in the "negatives" subdirectory. Storing the images in such a fashion makes it easier to load the dataset and parse it during training.

Training the Network

All the code is available in the "resources.zip" file. We used the Keras library over TensorFlow to train our CNN and the matplotlib library to create the accuracy and loss graphs.

1) The first step is to initialize matplotlib in the backend so that it can save the values for the loss and accuracy graphs in the background.

2) After importing all the required libraries and modules, we need to initialize our hyper – parameters: Learning rate, Epochs and Batch Size.

Learning rate determines how fast we would want our error function to converge to a minimum. A lower than optimal learning rate would result in a sluggish pace of training, whereas a higher than optimal rate could miss the solution altogether. There is no way to know the optimal learning rate apriori. We used a learning rate of e^{-3}. The number of epochs determine how many times the entire dataset would be fed into the network. Higher the number of epochs, greater is the probability for the network to learn intricate details about the dataset. However, one must be wary of overfitting the network. Overfitting causes the network to perform poorly on test images. Again, there is no method to decide the optimal number of epochs. We chose 250 epochs. We can't pass our entire dataset through the network in one pass; we must do it in batches. This is where Batch Size comes in. We decided to use a batch size of 32.

3) After initializing our hyper – parameters, we must load our training dataset. We lad all our image paths into a dictionary and shuffle them.

By shuffling the images and training on only a subset(batch) of them during a given iteration, the loss function changes with every iteration, and it is quite possible that no two iterations over the entire sequence of training iterations and epochs will be performed on the exact same loss function. This is important for us to "bounce" out of a local minimum.

4) After loading the dataset, we need to pre-process the images and assign class labels to them.

We resize the images to 28x28, convert them into an array and store it in a list "data". The subdirectory in which an image exists dictates its label. If the image exists in the "face" subdirectory, its label is 1, otherwise it is 0. The labels are stored in a list "labels". Both these lists are converted into NumPy arrays and the "data" NumPy array is normalized to contain values between 0 and 1.

5) The training data is then partitioned into training and validation sets with 75% for training and 25% for validation. We also augment the number of images we have using "ImageDataGenerator" to boost the performance over our validation set.

6) Now, we should define our model. Our model is a sequential one, with three sets of CONV, ReLU, POOL and Dropout layers.

The first set has:

- A convolutional 2D layer. This layer has 32 activation maps, each with the size of 3x3 and a rectifier activation function. This is the input layer, expecting images with the structure outline above [pixels][width][height].
- Another convolutional 2D layer with 32 activation maps each with a size of 3x3 and a rectifier activation function.
- A pooling layer MaxPooling2D with a pool size 2x2.
- And finally, a dropout layer with dropout ratio 0.25 which helps prevent overfitting in the network.

The second set of layers is like the first set, except that the two convolutional 2D layers create 64 activation maps each. The third set of layers is an exact replica of the second layer.

The final set of layers process the output from the first three sets and gives the output of the network.

- It has a Flatten layer which converts the 2D matrix data to a vector and allows the output to be processed by standard fully connected layers.
- The next layer is a fully connected layer with 512 neurons and a rectifier activation function.
- The next layer is a dropout layer with dropout ratio 0.5
- The last layer is the output layer that has a SoftMax activation function to output probability-like predictions for the face/not-face model.

The table below gives a summary of the different layers present in our CNN architecture.

#	Layer Name	Description
1	Convolutional 2D	Creates 32 activation maps, 3x3 filter, takes input vector of size 28x28x3
2	Convolutional 2D	Creates a new set of 32 activation maps, 3x3 filter
3	MaxPooling 2D	Applies the max function with pool size 2x2 on all 32 activation maps
4	Dropout	Drops 25% of activation maps chosen at random to prevent overfitting
5	Convolutional 2D	Creates 64 activation maps, 3x3 filter
6	Convolutional 2D	Creates a new set of 64 activation maps, 3x3 filter
7	MaxPooling 2D	Applies the max function with pool size 2x2 on all 64 activation maps
8	Dropout	Drops 25% of activation maps chosen at random to prevent overfitting
9	Convolutional 2D	Creates 64 activation maps, 3x3 filter
10	Convolutional 2D	Creates a new set of 64 activation maps, 3x3 filter
11	MaxPooling 2D	Applies the max function with pool size 2x2 on all 64 activation maps
12	Dropout	Drops 25% of activation maps chosen at random to prevent overfitting
13	Flatten	Flattens the 3D vector to a 1D vector
14	Dense	Fully Connected Layer with 512 neurons and rectifier activation function
15	Dropout	Drops 50% of activation maps chosen at random to prevent overfitting
16	Dense	Uses SoftMax to convert the scores of each class into probabilites

7) After defining our model, we can compile it. The loss function we used is the "binary_crossentropy" function and the optimizer used was the "Adam" optimizer. Finally, we can plot our Training Loss and Training Accuracy curves.

Testing the Network

To detect faces of different scales in an image, we need to have an image pyramid. An "image pyramid" is a multi-scale representation of an image. At the bottom of the pyramid we have the original image at its original size (in terms of width and height). And at each subsequent layer, the image is resized (subsampled) and optionally smoothed (usually via Gaussian blurring). The image is progressively subsampled until some stopping criterion is met, which is normally a minimum size has been reached and no further subsampling needs to take place.

Sliding windows play an integral role in object classification, as they allow us to localize exactly where in an image our object of interest resides. A sliding window is rectangular region of fixed width and height that slides across an image. For each of these windows, we take the window region and apply our CNN model to determine if the window has an object that interests us — in this case, a face.

Utilizing both a sliding window and an image pyramid we can detect faces in images at various scales and locations. For each window in each image in the pyramid, we apply our face/not-face model to detect faces if they return a confidence greater than 99%.

Results

Fig. 6 shows a few examples of the outputs from our classifier. Fig 7. Contains loss and accuracy curves for the network trained over the parameter mentioned in the previous section. We could achieve a final accuracy of 98.9%. As you can see in Fig 6. Validation accuracy and loss do not stray far from training accuracy and loss respectively. This is due to the data augmentation step that we perform while preparing before training our network. Without data augmentation, the probability of overfitting the network would be higher as the network would try to learn the training dataset "by-heart".

Fig. 6 Examples of outputs from our CNN face detector.

Fig. 7 (a) Training Accuracy (b) Training Loss

Conclusions and Future Work

In this work, we try to implement a convolutional neural network to an image dataset to build an image classifier for detecting faces in a test image. Convolutional neural networks have been shown to outperform earlier methods of face detection like the traditional Viola – Jones face detector because of the various advantages they offer. Viola – Jones detector only works well when the faces are facing the camera because of the simplistic nature of Haar features that they learn. CNNs on the other hand automatically learn complex features and are very robust to the differences in orientation of faces. Techniques like MaxPooling and parallel processing on GPUs ensure that CNNs train much faster than a traditional cascade classifier. There are several variations of CNNs like R-CNN and Faster-RCNN outperform CNNs.

Our CNN face detector works well, but there is a lot of room for improvement.

- The parameters used during training have scope for optimization. We just used the values we felt best from limited experimentation. These parameters include number of epochs, batch size, learning rate, training/testing split ratio and most importantly the model architecture.
- Pruning multiple detections for the same face. As can be seen from some of the examples in Fig. 7, the model detects multiple windows for the same face. This could be because of the sliding window/image pyramid approach to face detection. There are several post – processing steps to prune multiple instances which could be applied to clean up the output image.
- Optimizing testing parameters such as size and stride of the sliding window and scale of the image pyramid.
- Training on a larger set of training images would significantly improve the performance of our CNN.
- It is possible to implement our CNN model to detect faces in real time.

References

- ImageNet Classification with Deep Convolutional Neural Networks - Alex Krizhevsky, Ilya Sutskever, Geoffrey E. Hinton
- Visualizing and Understanding Convolutional Networks - Matthew D. Zeiler, Rob Fergus
- A Convolutional Neural Network Cascade for Face Detection - Haoxiang Li, Zhe Lin, Xiaohui Shen, Jonathan Brandt, Gang Hua
- Multi-view Face Detection Using Deep Convolutional Neural Networks - Sachin Sudhakar Farfade, Mohammad Saberian, Li-Jia Li
- Compact Convolutional Neural Network Cascade for Face Detection – Kalinovskii I.A., Spitsyn V.G.