Computer Vision

MoSIG M2

James L. Crowley and Nachwa Aboubakr

Fall Semester 15 October 2020

Lesson 2 - Practical Part

Face Detection in Images and Multi-Layer Perceptrons

Lesson Outline:

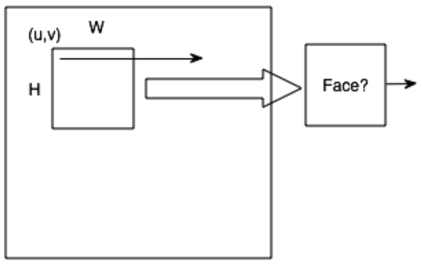
1. Face Detectors

As explained in our first lecture, we will use the problem of face detection as a running example to compare different techniques for detecting, classifying and tracking information in images. For each lecture, you will be asked to implement a form of face detector, and evaluate its performance. In most cases, the implementation will illustrate some point from the theoretical exercises. After the sixth lecture, you will be asked to write a report on you experiments. This report will count for half of your grade.

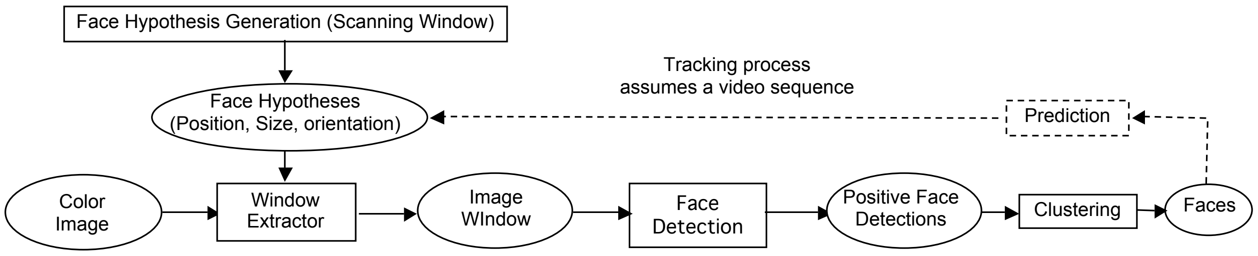
We will use the FDDB data available on the course web site, as our first source of training and test data. The FDDB data set contains 2845 images with a total of 5171 faces extracted from news articles selected using an automatic face detector. Faces with a height or width less than 20 pixels, as well as faces that are looking away from the camera were rejected. The remaining 5171 faces have been noted in a ground-truth data set and labeled with a bounding box. The images in this data set exhibit large variations in scale, pose, lighting, background and appearance due to factors such as motion, occlusions, and facial expressions, which are characteristic of the unconstrained setting for image acquisition. This can be challenging for many computer vision algorithms.

1.1. Sliding Window Face Detectors

Faces can occur at many different positions, orientations and sizes (scales) in an image. With sufficient computing power, we could test could test for faces at all possible positions, sizes and orientations in parallel. However, lacking a massively parallel computer, we can use a sliding window detector.



A sliding window detector is a brute force method to test if a pattern can be found in an image. This approach was made popular by the Viola-Jones face detector, now found in most smart phones and tablets. A typical architecture for a sliding window detector looks like this:


Face Hypothesis Generation (Scanning Window)
Face Hypotheses (Position, Size, orientation)
Tracking process assumes a video sequence
Prediction
Color Image
Window Extractor
Image WIndow
Face Detection
Positive Face Detections
Clustering
Faces

Rectangular windows over a range of positions and sizes (and possibly orientations) are extracted and projected into a standard size feature vector. This feature vector is run through a binary classifier (a detection function) that may produce a binary decision (P, N) or a likelihood of a P. The detection function will typically provide a positive detection over a range of position and sizes near an actual target. Adjacent detections may be grouped into clusters and used to provide a precise estimate for the position and size (and orientation) for each face.

This process can be used to build an efficient tracking system for video sequence. In this case, face detection is limited to a small range of positions and sizes near where faces have been found in the previous image. Our problem for this week is to construct and evaluate a simple face detector using a Multi-layer Perceptron (a fully connected neural network).

1.2. Scale vs Resolution

Scale is the size at which a pattern occurs in an image. With modern cameras, image can range to 4K pixels by 4K pixels (or larger) and faces can and do occur at all sizes depending on the optics and distance from the camera. We say that faces exist at multiple scales.

Resolution is the number of pixels used to represent a pattern. It is well known that for a pattern to be recognizable as a face, it must be represented by an array of at least 8 x 8 pixels, and that error rates will improve with larger arrays. It can also be shown that windows larger than 32 x 32 pixel provide little or no gain in detection rate and may actually degrade the error rate. Thus there is a "sweet spot" of around 24 by 24 pixels that provides the optimum resolution for face detection. In addition, faces tend to be oval in shape, and it is often possible to improve detection by using a rectangular window with a larger vertical dimension, such as a 4 to 3 or similar ratio.

Most machine learning algorithms for classifiers require specifying a fixed size input vector. Flattening a 24 x 24 pixel array to 1-D gives a vector with 576 components, which is large, but not unreasonable for a perceptron. A part of our task will be determine how the detection error rate, and requirements for training vary as a function of resolution changes from, say 8x8 pixels to 24x24 pixels.

1.3. Building a Data set

In order to perform a proper evaluation, we need to train and test our detector with a similar number of P and N examples. Thus we need to build a "balanced" data set. Using a Balanced Data set will also be very important for successful learning of a detection function using any machine learning technique. The data set will be composed of known P and N examples.

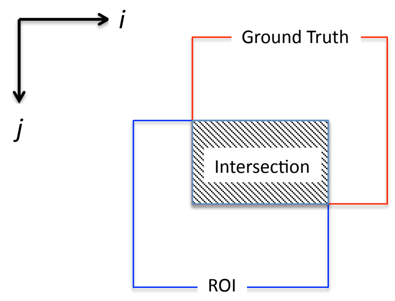
The images in FDDB have been divided into 10 "folds". You should train your detector with some of the folds and use other folds for testing. We have prepared a standard "evaluation" test set to allow programming teams to compare the results of their detectors.

The easiest way to build a set of Positive face examples from FDDB is to simply extract an example of a Positive ROI for each face listed in the FDDB ground truth. We can we transform the pixels in each face bounding box to a standard resolution to be chosen for training and test.

For a balanced data set, we need a similar number of Negative examples, M_n (#N). We can do this by using a random number generator to choose the upper left corner a negative face window of the same size for each positive face. Normally, most randomly chosen windows will be negative (N). However, occasionally a random window will overlap a Face. Thus we need to set a criterion for when a window can be confirmed as a true negative (TN).

1.4. Intersection over Union (IUO)

A rectangular window can be considered to be true positive when it has sufficient overlap with a Ground Truth Bounding Box. Overlap is measured as the ratio of Intersection over Union (IOU).



IOU is area of intersection divide by the area of Union of the rectangles: $IOU = \frac{A_I}{A_U}$

Assume image coordinates with the origin at the top left corner. A rectangular window can be represented by the coordinate of their top-left (*l, t*) and bottom right (r, b), corners (l, t, r, b). This is often called a "Region of Interest" or ROI.

l - "left" - first column of the window.
t - "top" - first row of the window.
r - "right" - last column of the window.
b - "bottom" - last row of the window

The area of a rectangle, $\vec{R}$ is: $A = w \cdot h = (l - r + 1) \cdot (b - t + 1)$

For two rectangles: $\vec{R}_1$ and $\vec{R}_2$, the area of the intersection of two rectangles is

(t_i, l_i, b_i, r_i)

where $l_i = \max(l_1, l_2)$ $r_i = \min(r_1, r_2)$ $t_i = \max(t_1, t_2)$ $b_i = \min(b_1, b_2)$

The area of the intersection is $A_i = (l_i - r_i + 1) \cdot (b_i - t_i + 1)$

The area of the Union of two rectangles is: $A_U = A_1 + A_2 - A_i$

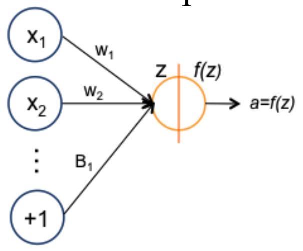
Thus IOU is: $IOU = \frac{A_i}{A_U} = \frac{A_i}{A_1 + A_2 - A_i}$

A typical threshold for True Positive is IOU > 0.5. A True Negative requires an IOU to be ≤ 0.5. We can reject any randomly generated window with an IOU>0.5 and choose a new upper left corner.

2. Multi-layer Perceptrons (Artificial Neural Networks)

2.1. The Perceptron Model

The simplest possible neural network is composed of a single percetron.



A perceptron (or "artificial neuron") is a computational unit that integrates information from a vector of features, $\vec{X}$, to compute the likelihood of an activation, a.

$$a = f(z)$$

The neuron is composed of a weighted sum of input values

$$z = w_1x_1 + w_2x_2 + \ldots + w_Dx_D + b$$

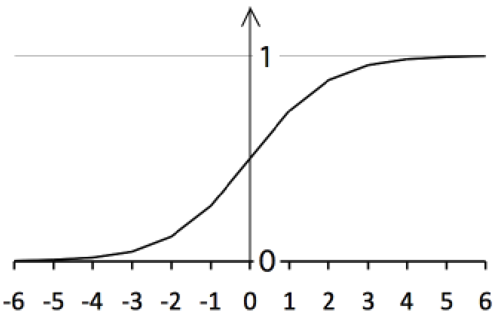
followed by a non-linear "activation" function, $f(z)$

$$a = f(\vec{w}^T\vec{X} + b)$$

2.2. Nonlinear Activation Function.

A non-linear activation function makes it possible to learn the network parameters. Many different non-linear activation functions may be used for $f(z)$

A popular choice for activation function is the sigmoid: $\sigma(z) = \frac{1}{1+e^{-z}}$



The sigmoid is useful because the derivative is: $\frac{d\sigma(z)}{dz} = \sigma(z)(1-\sigma(z))$

Other popular decision functions include the hyperbolic tangent, relu and softmax.

The hyperbolic Tangent: $f(z) = \tanh(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}}$

The rectified linear function is popular for deep learning because of a trivial derivative:

Relu: $relu(z) = \max(0, z)$

While Relu is discontinuous at z=0, for $z > 0$: $\frac{d(relu(z))}{dz} = 1$

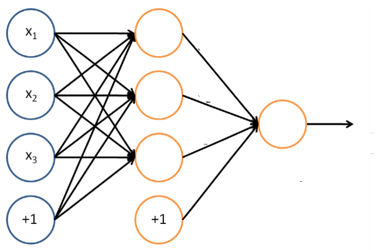
The softmax function is often used for multi-class networks. For K classes:

$$f(z_k) = \frac{e^{z_k}}{\sum_{k=1}^{K} e^{z_k}}$$

Note that the choice of decision function will determine the target variable "y" for supervised learning.

2.3. The Neural Network model

A neural network is a multi-layer assembly of neurons. For example, this is a 2-layer network:



The circles labeled +1 are the bias terms.
The circles on the left are the input terms. Some authors, notably in the Stanford tutorials, refer to this as Level 1.

We will NOT refer to this as a level (or, if necessary, level L=0).
The rightmost circle is the output layer, also called L.
The circles in the middle are referred to as a "hidden layer". In this example there is a single hidden layer and the total number of layers is L=2.

The parameters carry a superscript, referring to their layer.

We will use the following notation:

L	The number of layers (Layers of non-linear activations).
l	The layer index. l ranges from 0 (input layer) to L (output layer)
$N^{(l)}$	The number of units in layer l. $N^{(0)}=D$
$a_j^{(l)}$	The activation output of the jth neuron of the lth layer.
$w_{ij}^{(l)}$	The weight from the unit i of layer l-1 for the unit j of layer l.
$b_j^{(l)}$	The bias term for j^{th} unit of the l^{th} layer
$f(z)$	A non-linear activation function, such as a sigmoid, tanh, or soft-max

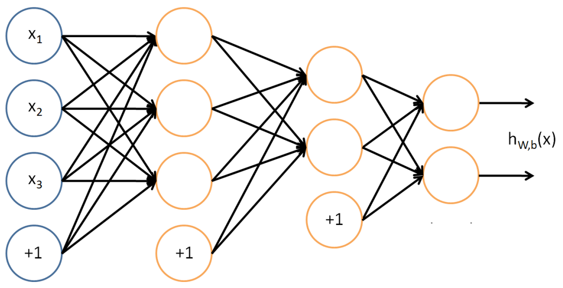
For example: $a_1^{(2)}$ is the activation output of the first neuron of the second layer.
$W_{13}^{(2)}$ is the weight for neuron 1 from the first level to neuron 3 in the second level.

The above network would be described by:

$$a_1^{(1)} = f(w_{11}^{(1)}X_1 + w_{21}^{(1)}X_2 + w_{31}^{(1)}X_3 + b_1^{(1)})$$
$$a_2^{(1)} = f(w_{12}^{(1)}X_1 + w_{22}^{(1)}X_2 + w_{32}^{(1)}X_3 + b_2^{(1)})$$
$$a_3^{(1)} = f(w_{13}^{(1)}X_1 + w_{23}^{(1)}X_2 + w_{33}^{(1)}X_3 + b_3^{(1)})$$
$$a_1^{(2)} = f(w_{11}^{(2)}a_1^{(1)} + w_{21}^{(2)}a_2^{(1)} + w_{31}^{(2)}a_3^{(1)} + b_1^{(2)})$$

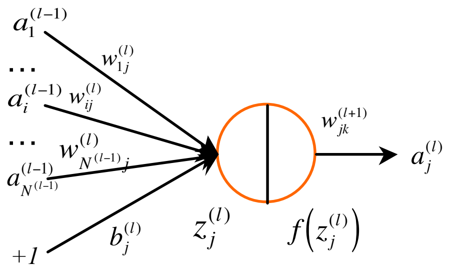
2.4. Multi-Layer networks.

The perceptron model can be generalized to multiple layers. For example:



Where $\vec{a}^{(out)} = \begin{pmatrix} a_1^{(out)} \\ \vdots \\ a_K^{(out)} \end{pmatrix}$ is the vector of network outputs, with each output neuron indicating the "likelihood" of the class "k".

For a sliding window detector, the 2-D window must be flattened into a 1-D Vector that is input to the network. For example an 8x8 Sliding window would require a network with 64 inputs. Each unit is defined as follows:



The equations for each neuron are:

$$z_j^{(l)} = \sum_{i=1}^{N^{(l-1)}} w_{ij}^{(l)} a_i^{(l-1)} + b_j^{(l)} \qquad a_j^{(l)} = f\left(\sum_{i=1}^{N^{(l-1)}} w_{ij}^{(l)} a_i^{(l-1)} + b_j^{(l)} \right)$$

Where

$\vec{a}^{(0)} = \vec{X}$ is the input layer. $a_i^{(0)} = X_d$

l is the current layer under discussion.

$N^{(l)}$ is the number of activation units in layer l. $N^{(0)} = D$

i,j,k Unit indices for layers *l-1, l* and *l+1*: $i \to j \to k$

$w_{ij}^{(l)}$ is the weight for the unit *i* of layer *l-1* feeding to unit *j* of layer *l*.

$a_j^{(l)}$ is the activation output of the *j*th unit of the layer *l*

$b_j^{(l)}$ the bias term feeding to unit *j* of layer *l*.

$z_j^{(l)} = \sum_{i=1}^{N^{(l-1)}} w_{ij}^{(l)} a_i^{(l-1)} + b_j^{(l)}$ is the weighted input to *j*th unit of layer *l*

f(z) is a non-linear decision function, such as a sigmoid, tanh(), or soft-max

$a_j^{(l)} = f(z_j^{(l)})$ is the activation output for the *j*th unit of layer *l*

2.5. Matrix Notation

It can be more convenient to represent this using vectors:

$$\vec{z}^{(l)} = \begin{bmatrix} z_1^{(l)} \\ z_2^{(l)} \\ \vdots \\ z_{N_l}^{(l)} \end{bmatrix} \qquad \vec{a}^{(l)} = \begin{bmatrix} a_1^{(l)} \\ a_2^{(l)} \\ \vdots \\ a_{N_l}^{(l)} \end{bmatrix}$$

and to write the weights and bias at each level l as a k by j Matrix,

$$W^{(l)} = \begin{pmatrix} w_{11}^{(l)} & \cdots & w_{1i}^{(l)} & \cdots & w_{1N^{(l-1)}}^{(l)} \\ \vdots & \ddots & \vdots & \cdot^{\cdot^{\cdot}} & \vdots \\ w_{j1}^{(l)} & \cdots & w_{ji}^{(l)} & \cdots & w_{jN^{(l-1)}}^{(l)} \\ \vdots & \cdot^{\cdot^{\cdot}} & \vdots & \ddots & \vdots \\ w_{N^{(l)}1}^{(l)} & \cdots & w_{N^{(l)}i}^{(l)} & \cdots & w_{N^{(l)}N^{(l-1)}}^{(l)} \end{pmatrix} \qquad \vec{b}^{(l)} = \begin{pmatrix} b_1^l \\ \vdots \\ b_i^l \\ \vdots \\ b_{N^{(l)}}^l \end{pmatrix}$$

(note: To respect matrix notation, we have reversed the order of i and j in the subscripts.)

We can see that the weights are a 3rd order Tensor or vector of matrices, with one matrix for each level, The biases are a matrix (vector of vectors) with a vector for each level.

$$\vec{z}^{(l)} = W^{(l)}\vec{a}^{(l-1)} + \vec{b}^{(l)} \quad \text{and} \quad \vec{a}^{(l)} = f(\vec{z}^{(l)}) = f(W^{(l)}\vec{a}^{(l-1)} + \vec{b}^{(l)})$$

We can assemble the set of matrices $W^{(l)}$ into an 3rd order Tensor (Vector of matrices), W, and represent $\vec{a}^{(l)}$, $\vec{z}^{(l)}$ and $\vec{b}^{(l)}$ as matrices (vectors of vectors): A, Z, B.

2.6. Training with Gradient Descent using Back-propagation

The network weights and biases are trained by Gradient Descent using a set of M sample windows $\{\vec{X}_m\}$ with ground truth labels (indicator variables), $\{y_m\}$. For our face detector, $\{\vec{X}_m\}$ would be a set of M flattened face windows, and the ground truth with be binary {P, N} indictors. The results would be tested with a separate set of sample windows with ground truth labels (a test set). Note that it is important to have a similar number of P and N examples in the training and test data.

Training is typically performed using the using the back-propagation algorithm.

The Back-propagation algorithm can be summarized as:

1) Initialize the network and a set of correction vectors:

$$\forall_{i,j,l}\, w_{ji}^{(l)} = \mathcal{N}(0;\varepsilon)$$
$$\forall_{i,l}\, b_j^{(l)} = \mathcal{N}(0;\varepsilon)$$
$$\forall_{i,j,l}\, \Delta w_{ji}^{(l)} = 0$$
$$\forall_{i,l}\, \Delta b_j^{(l)} = 0$$

where $\mathcal{N}$ is a sample from a normal density, and ε is a small value.

2) For each training sample, $\vec{x}_m$, propagate $\vec{x}_m$ through the network (forward propagation) to obtain a network activation $a_m^{(L)}$. Compute the error and propagate this back through the network:

a) Compute the network error term: $\delta_m^{out} = \left(a_m^{(L)} - y_m\right)$

b) Compute the error term at Layer L: $\delta_m^{(L)} = \frac{\partial f(z_j^{(l)})}{\partial z_j^{(l)}}\delta_m^{out}$

c) Propagate the error back from *l=L-1* to *l=0*: $\delta_{j,m}^{(l)} = \frac{\partial f(z_j^{(l)})}{\partial z_j^{(l)}}\sum_{k=1}^{N^{(l+1)}} w_{jk}^{(l+1)}\delta_{k,m}^{(l+1)}$

d) Use the error at each layer to set a vector of correction weights.

$$\Delta w_{ij,m}^{(l)} = -a_i^{(l-1)}\delta_{j,m}^{(l)} \qquad \Delta b_{j,m}^{(l)} = -\delta_{j,m}^{(l)}$$

3) For all layers, *l=1 to L*, update the weights and bias using a learning rate, η

$$w_{ij}^{(l)} \leftarrow w_{ij}^{(l)} + \eta \cdot \Delta w_{ij,m}^{(l)}$$
$$b_j^{(l)} \leftarrow b_j^{(l)} + \eta \cdot \Delta b_{j,m}^{(l)}$$

Note that this last step can be done with an average correction matrix obtained from many training samples (Batch mode), providing a more efficient algorithm.

2.7. Key Equations for L layers:

Feed Forward from layer *i* to *j*: $a_j^{(l)} = f\left(\sum_{i=1}^{N^{(l-1)}} w_{ij}^{(l)} a_i^{(l-1)} + b_j^{(l)}\right)$

Feed Forward from layer *j* to *k*: $a_k^{(l+1)} = f\left(\sum_{j=1}^{N^{(l)}} w_{jk}^{(l+1)} a_j^{(l)} + b_k^{(l+1)}\right)$

Ouput error for training sample *m*: $\delta_m^{out} = \left(a_m^{(L)} - y_m\right)$

Error for unit at layer *L*: $\delta_m^{(L)} = \frac{\partial f(z_j^{(l)})}{\partial z_j^{(l)}} \delta_m^{out}$

Back Propagation from Layer k to j: $\delta_{j,m}^{(l)} = \frac{\partial f(z_j^{(l)})}{\partial z_j^{(l)}} \sum_{k=1}^{N^{(l+1)}} w_{jk}^{(l+1)} \delta_{k,m}^{(l+1)}$

Weight and Bias Corrections for layer j:

$$\Delta w_{ij,m}^{(l)} = -a_i^{(l-1)} \delta_{j,m}^{(l)}$$

$$\Delta b_{j,m}^{(l)} = -\delta_{j,m}^{(l)}$$

Network Update Formulas:

$$w_{ij}^{(l)} \leftarrow w_{ij}^{(l)} + \eta \cdot \Delta w_{ij,m}^{(l)}$$

$$b_j^{(l)} \leftarrow b_j^{(l)} + \eta \cdot \Delta b_{j,m}^{(l)}$$

Example Keras Code ??

```
     model = Sequential()
     model.add(Flatten(input_shape=(N, N, 3)))
     model.add(Dense(units = N * N , activation='sigmoid'))
     model.add(Dense(units=2, activation='softmax'))
     model.compile(loss='sparse_categorical_crossentropy',
     optimizer='adam',
     metrics=['accuracy']


#Deteremine P ou N from IOU with Face Box
     train_df = pd.read_csv('annotations.csv')
     train_X = pd.Series(train_df['image_name'])
     train_Y = pd.Series(train_df['IOU'])

# Allocate Training Data, x, and Ground Truth Indicators y
     x = []
     y = []
     for i in range(0, len(train_X)):
          img = cv2.imread(train_X[i])
          x.append(img)
          if(train_Y[i] > 0.5):
               y.append(1)
          else:
               y.append(0)
          y = numpy.array(y)
          x = numpy.array(x)
#Train the model
      model.fit(x = x, y = y, epochs = 10)

# Save the model
     model.save("model.h5")
    x = []
    img = cv2.imread("./Training/01_32/0.png")
    x.append(img)
    img = cv2.imread("./Training/01_32/81.png")
    x.append(img)
    x = numpy.array(x)
    classes = model.predict(x)
    print(classes)
```