Task 4: Do research about and train two different models:

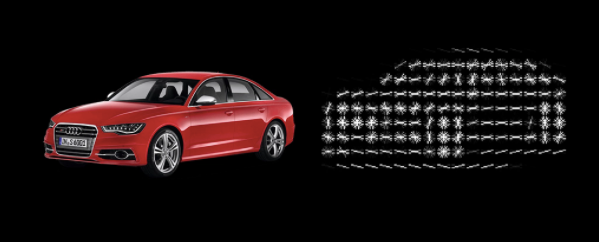
* Machine learning model: SVM using HOG features
* Deep learning: Facenet512

1. Histogram of Oriented Gradients (HOG)

**Feature Descriptors**

You may hear this name pop a lot throughout this article, but **feature descriptors** simply mean the representation of an image that simply extracts the useful information and disregards the unnecessary information from the image.

In the case of HOG feature descriptors, we also convert the image (width x height x channels) into a feature vector of length *n* chosen by the user. Although it may be hard to view these images, these images will be perfect for image classification algorithms like SVMs in order to produce good results.

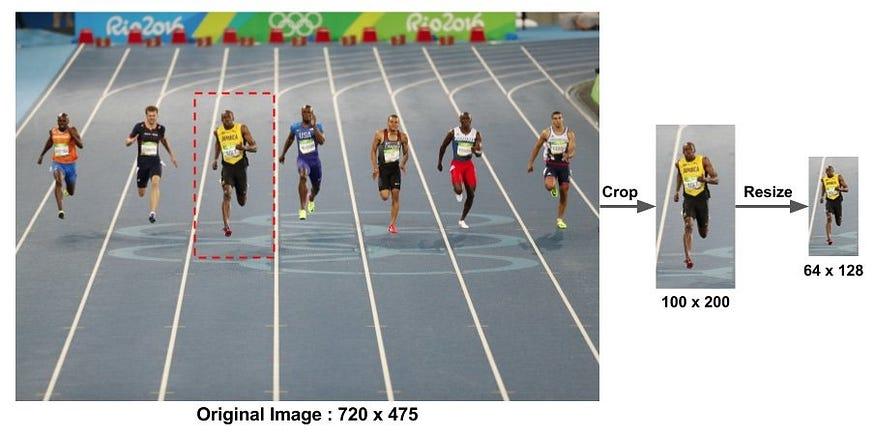


Example of HOG feature descriptor on images. Credit: [Analytics Vidhya](https://www.analyticsvidhya.com/blog/2019/09/feature-engineering-images-introduction-hog-feature-descriptor/)

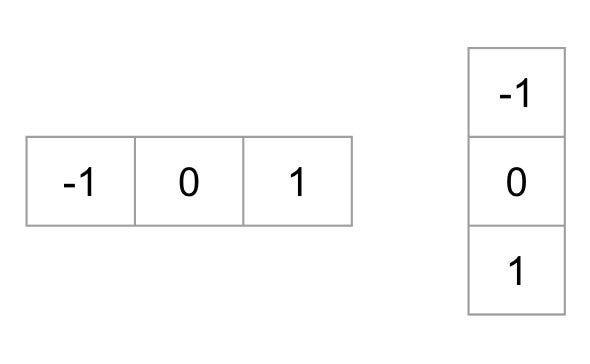
Now, you guys might be wondering how the HOG feature descriptor will actually sort through this unnecessary information. It simply does this by **using the histogram of gradients which are used as the features of an image**. Gradients are extremely important for checking for edges and corners in an image (through regions of intensity changes) since they often will pack much more information than flat regions.

## **Preprocessing**

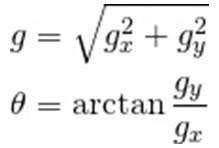
A key mistake that individuals often perform when doing HOG object detection is that they forget to preprocess the image so that it has a fixed aspect ratio. A common aspect ratio (width:height) is 1:2, so your images can be 100x200, 500x1000, etc.

For a particular image that you choose, make sure that you identify the section that you want so that it **correctly fits the aspect ratio** and allows for easier accessibility in the long run.**Calculating the Gradients**

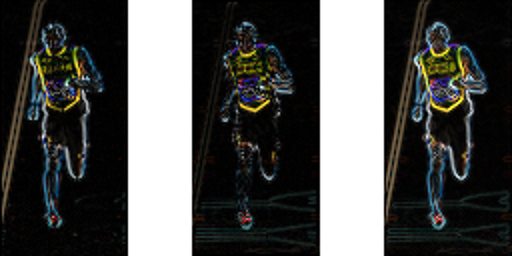
To make the HOG feature descriptor, as discussed above, we need to calculate the respective horizontal and vertical gradients to actually provide the histogram that can be used later in the algorithm. This can be done by simply filtering the image through these kernels:



Kernels like these are often used in image classification mainly in convolutional neural networks in order to find the edges and important points in a particular image. Then, the magnitude and the direction of the gradients can simply be found by using the following formulas (note that this is simply converting from **Cartesian to Polar coordinates** in a sense):



The main takeaway that you should get from gradients are that **the magnitude of the gradient increases wherever there is a sharp change in intensity.** The picture below highlights an optimal example of gradients do this as they fire around the edges of the images through these gradients. The unnecessary information is removed like the background and only the essential parts remain.



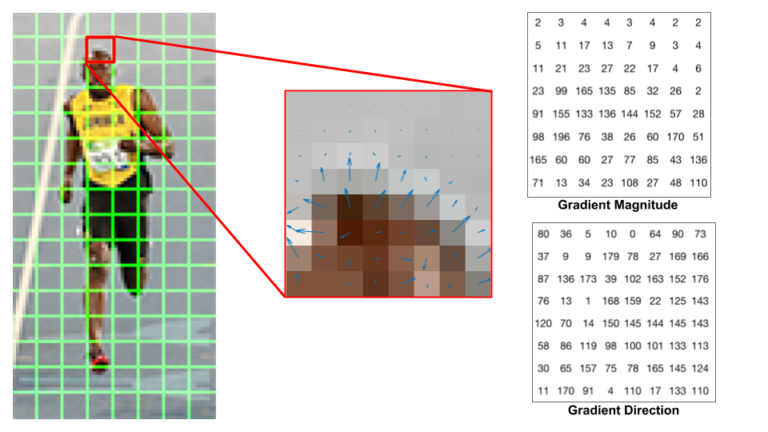
To summarize, understand that gradients have a magnitude and direction where the magnitude is calculated through the maximum of the magnitude of the gradients from the three color channels and the angle is calculated from the angle corresponding to the maximum gradient out of the three channels evaluated. These can produce images like the ones above where it can detect important information and disregard the unnecessary parts.

**Making A Histogram From These Gradients**

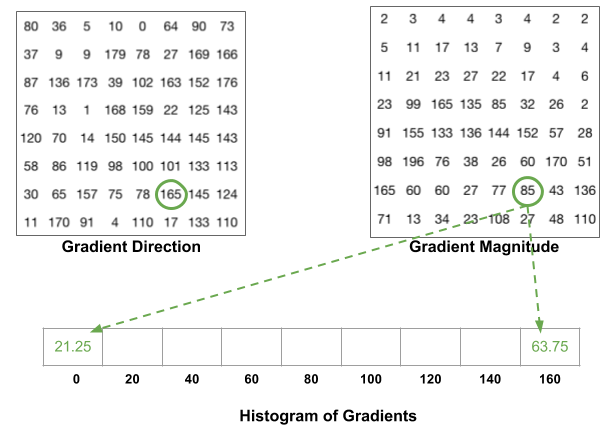
To move on to the next step of the HOG algorithm, make sure that the image is divided into cells so that the histogram of gradients can be calculated for each cell. For example, if you have a 64x128 image, divide your image into 8x8 cells (this will involve a bit of tweaking and guessing!).

Feature descriptors will allow for a concise and succinct representation of particular patches of the images; taking our example from above, an 8x8 cell can simply be explained using 128 numbers (8x8x2 where the last 2 are from the gradient magnitude and directional values). By further converting these numbers to calculate histograms, we allow for an image patch that is much **more robust to noise and more compact.**

For the histogram, make sure to split it up into **nine separate bins**, each corresponding to angles from 0–160 in increments of 20. Here’s an example of how an image with the respective gradient magnitudes and directions can look like (notice the arrows get larger depending on the magnitude).



In order to decide where each pixel goes inside of the histogram, a bin is selected depending on the direction chosen, and the value that is subsequently placed inside of the bin is dependent on the magnitude. Note that if a pixel is halfway between two bins, then it splits up the magnitudes accordingly depending on their distance away from each respective bin. After performing this process, a histogram can be formed, and the bins that have the most weight can easily be seen.

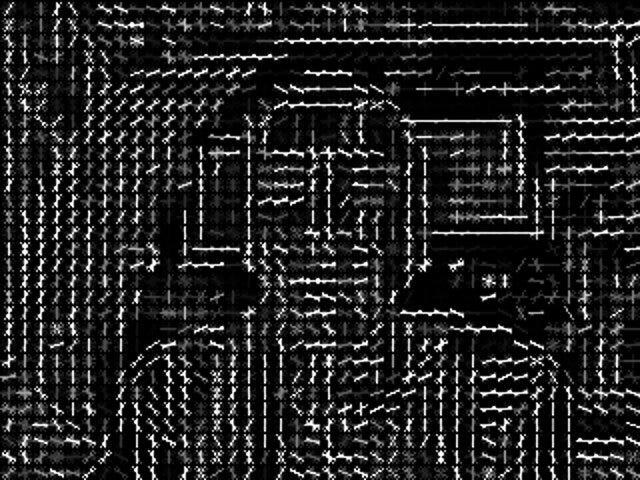


**Block Normalization (Optional)**

Lighting variations are another major factor that can mess up how these gradients are calculated. For example, if the picture was darker by 1/2 of the current brightness, then the gradient magnitudes and subsequently the histogram magnitudes would all decrease by half. Therefore, we want our descriptor to be **devoid of lighting variations so that it is unbiased and effective.**

The typical process of normalization occurs by simply calculating the length of a vector through its magnitude and then simply dividing all elements of that vector with the length. For example, if you had a vector of [1,2, 3], then the length of the vector, using basic mathematical principles, would be the square root of 14. By dividing the vector by this length, you arrive at your new normalized vector of [0.27, 0.53, 0.80].

This process of normalization can be performed **depending on your preference** (whether you want to perform on the 8x8 block or even a larger 16x16 block). Just remember to first turn these blocks into element vectors so that the normalization highlighted above can be performed.

**Image Visualization**

In many instances, the HOG descriptors are often visualized with the image on the right in order to get an accurate representation of the shape of the person. This visualization can be extremely useful in understanding where the gradients shift and knowing where the objects are inside of the image.

1. Support Vector Machine (SVM)

A Support Vector Machine (SVM) is a supervised machine learning algorithm used for both classification and regression tasks. While it can be applied to regression problems, SVM is best suited for classification tasks. The primary objective of the SVM algorithm is to identify the optimal hyperplane in an N-dimensional space that can effectively separate data points into different classes in the feature space. The algorithm ensures that the margin between the closest points of different classes, known as support vectors, is maximized.

The dimension of the hyperplane depends on the number of features. For instance, if there are two input features, the hyperplane is simply a line, and if there are three input features, the hyperplane becomes a 2-D plane. As the number of features increases beyond three, the complexity of visualizing the hyperplane also increases.

Consider two independent variables, x1 and x2, and one dependent variable represented as either a blue circle or a red circle.

* In this scenario, the hyperplane is a line because we are working with two features (x1 and x2).
* There are multiple lines (or hyperplanes) that can separate the data points.
* The challenge is to determine the best hyperplane that maximizes the separation margin between the red and blue circles.



Linearly Separable Data points

From the figure above it’s very clear that there are multiple lines (our hyperplane here is a line because we are considering only two input features x1, x2) that segregate our data points or do a classification between red and blue circles. So how do we choose the best line or in general the best hyperplane that segregates our data points?

## How does Support Vector Machine Algorithm Work?

One reasonable choice for the best hyperplane in a Support Vector Machine (SVM) is the one that maximizes the separation margin between the two classes. The maximum-margin hyperplane, also referred to as the hard margin, is selected based on maximizing the distance between the hyperplane and the nearest data point on each side.



Multiple hyperplanes separate the data from two classes

So we choose the hyperplane whose distance from it to the nearest data point on each side is maximized. If such a hyperplane exists it is known as the maximum-margin hyperplane/hard margin. So from the above figure, we choose L2. Let’s consider a scenario like shown below



Selecting hyperplane for data with outlier

Here we have one blue ball in the boundary of the red ball. So how does SVM classify the data? It’s simple! The blue ball in the boundary of red ones is an outlier of blue balls. The SVM algorithm has the characteristics to ignore the outlier and finds the best hyperplane that maximizes the margin. SVM is robust to outliers.



Hyperplane which is the most optimized one

So in this type of data point what SVM does is, finds the maximum margin as done with previous data sets along with that it adds a penalty each time a point crosses the margin. So the margins in these types of cases are called soft margins. When there is a soft margin to the data set, the SVM tries to minimize (1/margin+∧(∑penalty)). Hinge loss is a commonly used penalty. If no violations no hinge loss.If violations hinge loss proportional to the distance of violation.

Till now, we were talking about linearly separable data(the group of blue balls and red balls are separable by a straight line/linear line). What to do if data are not linearly separable?



Original 1D dataset for classification

Say, our data is shown in the figure above. SVM solves this by creating a new variable using a kernel. We call a point xi on the line and we create a new variable yi as a function of distance from origin o.so if we plot this we get something like as shown below



Mapping 1D data to 2D to become able to separate the two classes

In this case, the new variable y is created as a function of distance from the origin. A non-linear function that creates a new variable is referred to as a kernel.

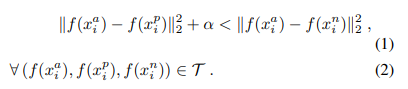
1. Implementation

In this project, in addition to using SVM as classifier, we also apply PCA (Principle component analysis), an unsupervised learning algorithm technique to reduce the dimensionality of a dataset while preserving the most important patterns or relationships between the variables without any prior knowledge of the target variables.

1. Facenet

FaceNet uses a deep convolutional network. We discuss two different core architectures: The Zeiler&Fergus style networks and the recent Inception type networks. The details of these networks are described in section 3.3. Given the model details, and treating it as a black box (see Figure 2), the most important part of our approach lies in the end-to-end learning of the whole system. To this end we employ the triplet loss that directly reflects what we want to achieve in face verification, recognition and clustering. Namely, we strive for an embedding f(x), from an image x into a feature space R d , such that the squared distance between all faces, independent of imaging conditions, of the same identity is small, whereas the squared distance between a pair of face images from different identities is large. Although we did not directly compare to other losses, e.g. the one using pairs of positives and negatives, as used in [14] Eq. (2), we believe that the triplet loss is more suitable for face verification. The motivation is that the loss from [14] encourages all faces of one identity to be projected onto a single point in the embedding space. The triplet loss, however, tries to enforce a margin between each pair of faces from one person to all other faces. This allows the faces for one identity to live on a manifold, while still enforcing the distance and thus discriminability to other identities. The following section describes this triplet loss and how it can be learned efficiently at scale. 3.1. Triplet Loss The embedding is represented by f(x) ∈ R d . It embeds an image x into a d-dimensional Euclidean space. Additionally, we constrain this embedding to live on the d-dimensional hypersphere, i.e. kf(x)k2 = 1. This loss is motivated in [19] in the context of nearest-neighbor classification. Here we want to ensure that an image x a i (anchor) of a specific person is closer to all other images x p i (positive) of the same person than it is to any image x n i (negative) of any other person. This is visualized in Figure 3. Thus we

want



where α is a margin that is enforced between positive and negative pairs. T is the set of all possible triplets in the training set and has cardinality N. The loss that is being minimized is then L =



Generating all possible triplets would result in many triplets that are easily satisfied (i.e. fulfill the constraint in Eq. (1)). These triplets would not contribute to the training and result in slower convergence, as they would still be passed through the network. It is crucial to select hard triplets, that are active and can therefore contribute to improving the model. The following section talks about the different approaches we use for the triplet selection. 3.2. Triplet Selection In order to ensure fast convergence it is crucial to select triplets that violate the triplet constraint in Eq. (1). This means that, given x a i , we want to select an x p i (hard positive) such that and similarly x n i (hard negative) such that . It is infeasible to compute the argmin and argmax across the whole training set. Additionally, it might lead to poor training, as mislabelled and poorly imaged faces would dominate the hard positives and negatives. There are two obvious choices that avoid this issue:

• Generate triplets offline every n steps, using the most recent network checkpoint and computing the argmin and argmax on a subset of the data.

• Generate triplets online. This can be done by selecting the hard positive/negative exemplars from within a mini-batch.

Here, we focus on the online generation and use large mini-batches in the order of a few thousand exemplars and only compute the argmin and argmax within a mini-batch.

To have a meaningful representation of the anchorpositive distances, it needs to be ensured that a minimal number of exemplars of any one identity is present in each mini-batch. In our experiments we sample the training data such that around 40 faces are selected per identity per minibatch. Additionally, randomly sampled negative faces are added to each mini-batch. Instead of picking the hardest positive, we use all anchorpositive pairs in a mini-batch while still selecting the hard negatives. We don’t have a side-by-side comparison of hard anchor-positive pairs versus all anchor-positive pairs within a mini-batch, but we found in practice that the all anchorpositive method was more stable and converged slightly faster at the beginning of training. We also explored the offline generation of triplets in conjunction with the online generation and it may allow the use of smaller batch sizes, but the experiments were inconclusive. Selecting the hardest negatives can in practice lead to bad local minima early on in training, specifically it can result in a collapsed model (i.e. f(x) = 0). In order to mitigate this, it helps to select x n i such that



We call these negative exemplars semi-hard, as they are further away from the anchor than the positive exemplar, but still hard because the squared distance is close to the anchorpositive distance. Those negatives lie inside the margin α. As mentioned before, correct triplet selection is crucial for fast convergence. On the one hand we would like to use small mini-batches as these tend to improve convergence during Stochastic Gradient Descent (SGD) [20]. On the other hand, implementation details make batches of tens to hundreds of exemplars more efficient. The main constraint with regards to the batch size, however, is the way we select hard relevant triplets from within the mini-batches. In most experiments we use a batch size of around 1,800 exemplars. 3.3. Deep Convolutional Networks In all our experiments we train the CNN using Stochastic Gradient Descent (SGD) with standard backprop [8, 11] and AdaGrad [5]. In most experiments we start with a learning rate of 0.05 which we lower to finalize the model. The models are initialized from random, similar to [16], and trained on a CPU cluster for 1,000 to 2,000 hours. The decrease in the loss (and increase in accuracy) slows down drastically after 500h of training, but additional training can still significantly improve performance. The margin α is set to 0.2. We used two types of architectures and explore their trade-offs in more detail in the experimental section. Their practical differences lie in the difference of parameters and FLOPS. The best model may be different depending on the application. E.g. a model running in a datacenter can have many parameters and require a large number of FLOPS, whereas a model running on a mobile phone needs to have few parameters, so that it can fit into memory. All our models use rectified linear units as the non-linear activation function.

1. Implementation

In this project, we use pretrained Facenet model with the help of a Python library named DeepFace. For anyone who is not familiar with DeepFace and its usage, it is the most lightweight face recognition and facial attribute analysis open-source library in Python which includes all leading-edge AI models for modern face recognition including Facenet. After that, we utilize the already-built feature extraction of DeepFace with model Facenet and detector mtcnn to get the embeddings vector for faces in our datasets. Finally, we create a SVM classifier to evaluate the performance of this model

Results: