MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.036—Introduction to Machine Learning
Spring 2016

Project 2: Handwritten Digit Recognition   Issued: Tue., 3/1 Due: Fri., 4/01 at 9am

Project Submission: Please submit a .zip file of your project on the Stellar web site by 9am, 4/01. The zip file should contain: a PDF of your project report and your Python code.

TAs: Amruth, David H., Haoyang, Jonas, Tony (in alphabetic order)

Introduction

For this project we will use the MNIST[1] (Mixed National Institute of Standards and Technology) database, which contains binary images of handwritten digits commonly used to train image processing systems. The digits were collected from among Census Bureau employees and high school students. The database contains $60,000$ training digits and $10,000$ testing digits, all of which have been size-normalized and centered in a fixed-size image of 28×28 pixels. Many methods have been tested with this dataset and in this project, you will get a chance to experiment with the task of classifying these images.

You will first implement multinomial logistic regression (aka softmax regression) with gradient descent and run your model on a test dataset. Next, you will use PCA to reduce high-dimensional vectors of pixels to a low-dimensional vector of features. You will also experiment with the reverse by increasing the dimension using kernels and then apply the regression model again, except to these new features. Lastly, you will apply neural network models to the same task.

Other than your code, you will submit a PDF with relevant plots and discussion. What to put in the PDF for each part is indicated with **"include"** annotations below.

[1] `<http://yann.lecun.com/exdb/mnist/`

0. Setup

As with the last project, please use Python's NumPy numerical library for handling arrays and array operations; use `matplotlib` for producing figures and plots.

Download `project2.zip` from Stellar and unzip it into a working directory. The zip file contains the image dataset in `mnist.pkl.gz`, several relevant Python files, and a `main.py` file where you will be running your code.

1. Multinomial/Softmax Regression and Gradient Descent [40pts]

In this section, you will implement multinomial regression and the gradient descent algorithm to learn a set of parameters used to classify images as digits from 0-9. You will work with the raw pixel values of each image. That is, each pixel in the original image corresponds to a feature in our feature vector.

To get warmed up to the MNIST data set run `main.py`. This file provides code that reads the data from `mnist.pkl.gz` by calling the function `getMNISTData` that is provided for you in `utils.py`.
The call to `getMNISTData` returns Numpy arrays:

- `trainX` : A matrix of the training data. Each row of `trainX` contains the features of one image, which are simply the raw pixel values flattened out into a vector of length $784 = 28^2$. The pixel values are float values between 0 and 1 (0 stands for black, 1 for white, and various shades of gray in-between).
- `trainY` : The labels for each training datapoint, aka the digit shown in the corresponding image (a number between 0-9).
- `testX` A matrix of the test data, formatted like `trainX`.
- `testY` The labels for the test data, which should *only* be used to evaluate the accuracy of different classifiers in your report.

Next, we call the function `plotImages` to display the first 20 images of the training set. Look at these images and get a feel for the data (don't include these in your write-up).

The main function which you will call to run the code you will implement in this section is `runSoftmaxOnMNIST` in `main.py` (already implemented). Below we describe a number of the methods that are already implemented for you in `softmax_skeleton.py` that will be useful.

1. `augmentFeatureVector`: adds the $x_0^{(i)} = 1$ feature for each data point $x^{(i)}$ for $i = 1, 2, ..., n$. The inputs are:

   - `X`, an $n \times (d-1)$ Numpy array of n data points, each with $d-1$ features.

   The function returns:

   - `X_augment`, an $n \times d$ Numpy array of n data points, each with d features.

   The motivation for doing this is to form a compact representation of $\theta \cdot x + \theta_0$. For example, if θ and x were both $(d-1)$ dimensional vectors, we could define a d dimensional vector $\theta' = [\theta_0, \theta_1, \theta_2, \ldots, \theta_{d-1}]$ and $x' = [1, x_1, x_2, \ldots x_{d-1}]$. Then $\theta \cdot x + \theta_0 = \theta' \cdot x'$.

2. `softmaxRegression`: runs batch gradient descent for a specified number of iterations on a dataset, with θ initialized to the all-zeros array. Here, θ is a $k \times d$ Numpy array, where row j represents the parameters of our model for label j for $j = 0, 1, ..., k - 1$. The inputs are:
   - `X`, an $n \times d$ Numpy array of n data points, each with d features.
   - `Y`, an $n \times 1$ Numpy array containing the labels (a number between 0-9) for each data point.
   - α, the learning rate (scalar).
   - λ, the regularization constant (scalar).
   - `k`, the number of labels (scalar).
   - `numIterations`, the number of iterations to run gradient descent (scalar).

   The function returns:
   - θ', a $k \times d$ Numpy array that is the final value of θ.
   - `costFunctionProgression`, a Python list containing the cost calculated at each step of gradient descent.

3. `getClassification`: makes predictions by classifying a given dataset. The inputs are:
   - `X`, an $n \times d$ Numpy array of n data points, each with d features.
   - θ, a $k \times d$ Numpy array, where row j represents the parameters of our model for label j.

   The function returns:
   - `Ŷ`, an $n \times 1$ Numpy array, containing the predicted label (a number between 0-9) for each data point.

4. `runSoftmaxOnMNIST` in `main.py`: runs the above `softmaxRegression` on the MNIST training set and computes the test error using the test set. It uses the following values for parameters: $\alpha = 0.3$, $\lambda = 10^{-4}$, and $numIterations = 150$. It also plots the cost function over the number of iterations. Once softmax regression has run, you will get the final model parameters θ. We have called the function `writePickleData` that will save the model parameters to a file called `theta.pkl.gz` in the current directory. Rather than rerunning softmax regression, you can read in your model parameters by calling the function `readPickleData` that is provided in `utils.py`. **This model must be saved with this exact name and will be used during grading**.

Below are the three methods that *you* are responsible for. Each method is described in detail with regards to the inputs and the outputs. We have included some methods in a file called `softmax_verification.py` to help you verify that the methods you have implemented are behaving sensibly.

1. Write a function `computeProbablities` [10pts] that computes, for each data point $x^{(i)}$, the probability that $x^{(i)}$ is labeled as j for $j = 0, 1, ..., k - 1$. The inputs are:
   - `X`, an $n \times d$ Numpy array of n data points, each with d features.
   - θ, a $k \times d$ Numpy array, where row j represents the parameters of our model for label j.

   The function returns:
   - `H`, a $k \times n$ Numpy array, where entry (j, i) is the probability that $x^{(i)}$ is labeled as j.

See the appendix for a hint about dealing with overflow errors. `verifyFirstIterationProbabilities` and `verifySecondIterationProbabilities` are available to help you reason that your code is correct. Note that `verifySecondIterationProbabilities` will not pass until you implement `runGradientDescentIteration`.

2. Write a function `computeCostFunction` [10pts] that computes the total cost over every data point. The inputs are:

   - `X`, an $n \times d$ Numpy array of n data points, each with d features.
   - `Y`, an $n \times 1$ Numpy array containing the labels (a number between 0-9) for each data point.
   - θ, a $k \times d$ Numpy array, where row j represents the parameters of our model for label j.
   - λ, the regularization constant (scalar).

   The function returns:

   - `c`, the cost value (scalar).

   For a reminder on the cost function formula, consult the appendix.

   You may use `verifyFirstIterationCostFunction` and `verifySecondIterationCostFunction` to help you reason that your code is correct. Note that `verifySecondIterationCostFunction` will not pass until you implement `runGradientDescentIteration`.

3. Write a function `runGradientDescentIteration` [10pts] that runs one step of batch gradient descent. The inputs are:

   - `X`, an $n \times d$ Numpy array of n data points, each with d features.
   - `Y`, an $n \times 1$ Numpy array containing the labels (a number between 0-9) for each data point.
   - θ, a $k \times d$ Numpy array, where row j represents the parameters of our model for label j.
   - α, the learning rate (scalar).
   - λ, the regularization constant (scalar).

   The function returns:

   - θ', a $k \times d$ Numpy array that is the next value of θ after one step of gradient descent.

   For a hint on the gradient descent step, consult the appendix.

Finally, in your report **include** the final test error [10pts].

If you have implemented everything correctly, the error on the test set should be around 0.1, which implies the linear softmax regression model is able to recognize MNIST digits with around 90% accuracy.

Important: To make sure you receive credit for this section, run the file `checker.py` which will try to check that your functions work with the specified input and return the specified output. If it errors that means there is an issue in your code. If it does not error it will say whether the simple tests in place to check types pass or fail.

2. Using manually crafted features [40pts]

The performance of most learning algorithms depends heavily on the representation of the training data. In this section, we will try representing each image using different features in place of the raw pixel values. Subsequently, we will investigate how well our regression model from the previous section performs when fed different representations of the data.

Dimensionality Reduction via PCA. Principal Components Analysis[2] (PCA) is the most popular method for linear dimension reduction of data and is widely used in data analysis. Briefly, this method finds (orthogonal) directions of maximal variation in the data. By projecting an $n \times d$ dataset X onto $k \ll d$ of these directions, we get a new dataset of lower dimension that reflects more variation in the original data than any other k-dimensional linear projection of X. By going through some linear algebra, it can be proven that these directions are equal to the k eigenvectors corresponding to the k largest eigenvalues of the covariance matrix $\widetilde{X}^T\widetilde{X}$, where $\widetilde{X}$ is a centered version of our original data.

Remark: The best implementations of PCA actually use the Singular Value Decomposition of $\widetilde{X}$ rather than the more straightforward approach outlined here, but these concepts are beyond the scope of this course.

Here are some functions that we have provided for you to use in this part (all located in `features.py`):

`centerData` centers the data (by subtracting off the mean of each feature).
The input is: X, an $n \times d$ Numpy array of n data points, each with d features.
The function returns: $n \times d$ Numpy array $\widetilde{X}$, where for each $i = 1, \ldots, n$ and $j = 1, \ldots, d$:

$$\widetilde{X}_{i,j} = X_{i,j} - \mu_j \quad \text{with } \mu_j = \frac{1}{n}\sum_{i=1}^{n} X_{i,j}$$

`principalComponents` computes the principal component directions of an input data matrix X ($n \times d$ Numpy array). This function first calculates the covariance matrix, $\widetilde{X}^T\widetilde{X}$ and then find its eigenvectors.
The function returns: $d \times d$ matrix (Numpy array) whose columns are the principal component directions sorted in descending order by the amount of variation each direction (these are equivalent to the d eigenvectors of $\widetilde{X}^T\widetilde{X}$ sorted in descending order of eigenvalues, so the first column corresponds the eigenvector with largest eigenvalue).

`plotPC` produces a scatter-plot of data after it has first been projected down to its 2-D representation in the first 2 principal components.

`reconstructPC` tries to reconstruct an original data sample from its lower-dimensional PCA-representation. Note that to reconstruct an observation from its representation in the top k principal components, $x_{\text{pca}} \in \mathbb{R}^{1\times k}$ we can use matrix algebra: $x_{\text{pca}} \cdot V^T + \mu$, where V is the $d \times k$ matrix whose columns are the top k eigenvectors of $\widetilde{X}^T\widetilde{X}$, and μ is a $1 \times d$ vector containing the mean of each feature (see the definition of its jth element μ_j above).

Below are the steps you are responsible for in this section:

1. Fill in function `projectOntoPC` in `features.py` that implements PCA dimensionality reduction of dataset X [10 pts]. Your input should be:

[2] In-depth exposition: `www.cs.otago.ac.nz/cosc453/student_tutorials/principal_components.pdf`

- `X`, a $n \times d$ Numpy array of n data points, each with d features.
- `pcs`, a $d \times d$ matrix whose columns are the d eigenvectors of $\widetilde{X}^T \widetilde{X}$, ordered in *descending* order of eigenvalues (i.e. the output of `principalComponents`).
- `n_components`, the number of principal components to use in the PCA representation.

Your function should return a $n \times$`n_components` Numpy array, whose rows represent low-dimensional feature vectors corresponding to the projection of dataset X onto its first `n_components` principal components (we refer to this as the PCA representation).
Note that to project a given $n \times d$ dataset X into its k-dimensional PCA representation, one can use matrix multiplication (after first centering X): $\widetilde{X}V$, where V is the $d \times k$ matrix whose columns are the top k eigenvectors of $\widetilde{X}^T \widetilde{X}$. This is because the eigenvectors are of unit-norm, so there is no need to divide by their length.

2. Use `projectOntoPC` to compute a 20-dimensional PCA representation of the MNIST training and test datasets, as illustrated in `main.py`.

   Retrain your softmax regression model (using the same settings the previous section) on the MNIST training dataset and report its error on the test data, this time using these 20-dimensional PCA-representations rather than the raw pixel values (**Include** the error in your report [5 pts]).

   If your PCA implementation is correct, the model should perform nearly as well when only given 20 numbers encoding each image as compared to the 784 in the original data (error on the test set using PCA features should be around 0.14). This is because PCA ensures these 20 feature values capture the maximal amount of variation in the original 784-dimensional data.

   Remark: Note that we only use the training dataset to determine the principal components. It is *improper* to use the test dataset for anything except evaluating the accuracy of our predictive model. If the test data is used for other purposes such as selecting good features, it is possible to overfit the test set and obtain overconfident estimates of a model's performance.

3. Use the call to `plotPC` in `main.py` to visualize the first 100 MNIST images, as represented in the space spanned by the first 2 principal components of the training data.
   Include this plot in your report [5 pts].

   Remark: Two dimensional PCA plots offer a nice way to visualize some global structure in high-dimensional data, although approaches based on nonlinear dimension reduction may be more insightful in certain cases. Notice that for our data, the first 2 principal components are insufficent for fully separating the different classes of MNIST digits.

4. Use the calls to `plotImages()` and `reconstructPC` in `main.py` to plot the reconstructions of the first two MNIST images (from their 20-dimensional PCA-representations) alongside the originals. **Include** these plots in your report [5 pts].

Quadratic Features.

In this section, we will work with a *quadratic feature* mapping which maps an input vector $x = [x_1, \ldots, x_d]$ into a new feature vector $\phi(x)$, defined so that for any $x, x' \in \mathbb{R}^d$:

$$\phi(x)^T \phi(x') = (x^T x' + 1)^2$$

5. In 2-D, let $x = [x_1, x_2]$. Write down the explicit quadratic feature mapping $\phi(x)$ as a vector (**include** in your report).
   Hint: $\phi(x)$ should be a 6-dimensional vector.

   Now, fill in function `quadraticFeatures` in `features.py`. Given an input dataset (with d-dimensional features where d is not necessarily equal to 2), this function returns a new dataset where each observation is now represented using quadratic features [10 pts].

6. If we explicitly apply the quadratic feature mapping to the original 784-dimensional raw pixel features, the resulting representation would be of massive dimensionality. Instead, we will apply the quadratic feature mapping to the 20-dimensional PCA representation of our training data which we computed previously. After applying the quadratic feature mapping to the PCA representations for both the train and test datasets, retrain the softmax regression model using these new features and report the resulting test set error (**Include** this error in your report [5 pts]).

   If you have done everything correctly, softmax regression should perform better (on the test set) using these features than either the principal components or raw pixels. The error on the test set using quadratic features should only be around 0.04, demonstrating the power of nonlinear classification models.

Remark: Note that the feature mapping we have been working with actually corresponds to the quadratic kernel, which is defined as: $K(x, x') = (x^T x' + 1)^2$. Recall, that the feature mapping ϕ associated with a kernel K satisfies $K(x, x') = \phi(x)^T \phi(x')$ for any $x, x' \in \mathbb{R}^d$.

In practice, this kernel function is all that is required to perform nonlinear kernel regression and one never has to actually compute the feature mapping (recall the kernel trick). Nevertheless, we have explicitly worked with the feature mapping corresponding to this kernel for educational purposes. Note that working with the explicit representations induced by the feature mapping can vastly increase in the dimensionality of the problem. This is why the kernel trick is a key component of kernel methods, allowing them to run efficiently in practice!

3. Classification using deep neural networks [20pts]

In this section, we are going to use deep neural networks to perform the same classification task as in previous sections. In particular, we will use Keras, a python library built upon the deep learning framework Theano. **Refer to the Appendix for instructions on setting up the necessary deep learning environment**.

1. **Fully-Connected Neural Networks**

   First, we will employ the most basic form of a deep neural network, in which the neurons in adjacent layers are fully connected to one another.

   (a) We have provided a toy example `mnist_nnet_fc.py` in which we have implemented for you a simple neural network. This network has one hidden layer of 128 neurons with a rectified linear unit (ReLU) nonlinearity, as well as an output layer of 10 neurons (one for each digit class). Finally, a softmax function normalizes the activations of the output neurons so that they specify a probability distribution. Reference the Keras website[3] and read through the documentation in order to gain a better understanding of the code. Then, try running the code on your computer with the command `python mnist_nnet_fc.py`. This will train the network with 10 epochs, where an epoch is a complete pass through the training dataset. Total training time of your network should take no more than a couple of minutes; if you find the training to be going slowly on your machine, try using an Athena[4] cluster machine. At the end of training, your model should have an accuracy around 92 to 93% on the test set (**test accuracy**).

(b) Modify your model to reach over 98% **test accuracy** after 10 epochs of training. Things you can, but don't have to, tweak include the number of fully-connected layers, the number of neurons in each layer, or the parameters of the stochastic gradient descent (SGD) optimizer (such as the learning rate).
Include a description of your final model architecture in your report, and give its accuracy on the test data. What did you try that worked, and what did you try that didn't work? [10 pts]

2. **Convolutional neural networks**

Next, we are going to apply convolutional neural networks to the same task. These networks have demonstrated great performance on many deep learning tasks, especially in computer vision.

(a) We provide a skeleton code `mnist_nnet_cnn_skeleton.py` which includes examples of some (**not all**) of the new layers you will need in this part. Using the Keras documentation website[5], complete the parts of the code called "Design the model here" and "Import the layer types needed" to implement a convolutional neural network with following layers in order:

- A convolutional layer with 32 filters of size 3×3
- A ReLU nonlinearity
- A max pooling layer with size 2×2
- A convolutional layer with 64 filters of size 3×3
- A ReLU nonlinearity
- A max pooling layer with size 2×2
- A flatten layer
- A fully connected layer with 128 neurons
- A dropout layer with drop probability 0.5
- A fully-connected layer with 10 neurons
- A softmax layer

Without GPU acceleration, you will likely find that this network takes quite a long time to train. For that reason, we don't expect you to actually train this network until convergence. Implementing the layers and verifying that you get approximately 93% **training accuracy** and 98% **validation accuracy** after one training epoch (this should take less than 10 minutes) is enough for this project. If you are curious, you can let the model train for a few hours; if implemented correctly, your model should achieve >99% **test accuracy** after 10 epochs of training. If you have access to a CUDA compatible GPU, you could even try configuring Keras to use your GPU. Include the code for your CNN in your submission. [10 pts]

[3] `http://keras.io/`
[4] `http://ist.mit.edu/athena`
[5] `http://keras.io/`

(b) (**Optional**) To understand why convolutional neural networks work so well, we often need to visualize the intermediate filters to see what's going on. blog[6] and try visualizing the features learned in each layer of your CNN network.

REMEMBER Submit a .zip file containing your source code and your report in PDF format to Stellar.

[6]`http://blog.keras.io/how-convolutional-neural-networks-see-the-world.html`

4. Appendix: some background and details

Dealing with overflow errors when computing H

Computing h of a particular $x^{(i)}$ requires computing

$$h(x^{(i)}) = \frac{1}{\sum_{j=1}^{k} e^{\theta_j \cdot x^{(i)}}} \begin{bmatrix} e^{\theta_1 \cdot x^{(i)}} \\ e^{\theta_2 \cdot x^{(i)}} \\ \vdots \\ e^{\theta_k \cdot x^{(i)}} \end{bmatrix}$$

While the probabilities themselves are in the range [0,1], $e^{\theta_j \cdot x^{(i)}}$ is not. To deal with this we can simply subtract some fixed amount c from each exponent to keep the resulting number from getting too large. See that

$$\begin{aligned} h(x^{(i)}) &= \frac{1}{\sum_{j=1}^{k} e^{\theta_j \cdot x^{(i)}}} \begin{bmatrix} e^{\theta_1 \cdot x^{(i)}} \\ e^{\theta_2 \cdot x^{(i)}} \\ \vdots \\ e^{\theta_k \cdot x^{(i)}} \end{bmatrix} \\ &= \frac{e^{-c}}{e^{-c}\sum_{j=1}^{k} e^{\theta_j \cdot x^{(i)}}} \begin{bmatrix} e^{\theta_1 \cdot x^{(i)}} \\ e^{\theta_2 \cdot x^{(i)}} \\ \vdots \\ e^{\theta_k \cdot x^{(i)}} \end{bmatrix} \\ &= \frac{1}{\sum_{j=1}^{k} e^{\theta_j \cdot x^{(i)} - c}} \begin{bmatrix} e^{\theta_1 \cdot x^{(i)} - c} \\ e^{\theta_2 \cdot x^{(i)} - c} \\ \vdots \\ e^{\theta_k \cdot x^{(i)} - c} \end{bmatrix} \end{aligned}$$

Thus subtracting some fixed amount from each exponent will not change the final probabilities. A good choice for this fixed amount is $c = \max_j \theta_j \cdot x^{(i)}$.

Softmax Cost Function

The cost function $J(\theta)$ is given by:

$$J(\theta) = -\frac{1}{m}\left[\sum_{i=1}^{m}\sum_{j=1}^{k}[[y^{(i)} == j]] \log \frac{e^{\theta_j \cdot x^{(i)}}}{\sum_{l=1}^{k} e^{\theta_l \cdot x^{(i)}}}\right] + \frac{\lambda}{2}\sum_{i=1}^{k}\sum_{j=0}^{d-1}\theta_{ij}^2$$

Softmax Gradient

The derivative of $J(\theta)$ wrt a particular θ_j is given by:

$$\bigtriangledown_{\Theta_j} J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}[x^{(i)}([[y^{(i)} == j]] - p(y^{(i)} = j | x^{(i)}, \theta))] + \lambda\theta_j$$

Environment setup for deep learning

We provide two ways to set up the computing environment for Keras. You are free to choose either one, but we strongly recommend solution 1 as it automatically takes care of version dependency across different packages. This tutorial only works for Mac and Linux users. **Windows users please use MIT Athena**[7] . We have successfully tested the miniconda environment and DNN code on the Athena machines, so if all else fails (or if you find the DNN code to run too slowly on your machine) consider using an Athena machine.

Solution 1

- Download miniconda for **Python 2.7** from here[8]
- Follow the instructions here [9] to install miniconda.
  - When asked to modify the environment config file in your home directory (∼/.bash_profile for Mac or ∼/.bashrc for Linux), **please approve**.
  - After installation, close the terminal and reopen it. Then type 'which python' to make sure the path has 'miniconda' in it.
- We provided a conda enviroment file called `environment.yml` . In the same folder as this file, run `conda env create -f environment.yml` and then `source activate ml6036` in terminal. You will see a 'ml6036' label in front of every line in your terminal now.
- Run `pip install git+git://github.com/Theano/Theano.git` in the terminal
- Additional step for Mac users: please first run `mdfind -name libmkl_intel_lp64.dylib`. Copy the output on the screen. Then run `export DYLD_FALLBACK_LIBRARY_PATH=YOUR_PATH` where `YOUR_PATH` is the output of the previous command
- Things to note:
  - **Every time you close and reopen the terminal, you will have to redo** `source activate ml6036` **to enter this environment again.**
  - At any point, you can switch back to the default python by simply removing the line 'export PATH=/yourhomepath/miniconda2/bin:$PATH' from your environment config file (see the second bulletin point above) and restart your terminal. When you feel you want to switch back to the conda python environment which enables you to run the code in this part, you will then need to first add back this line in your environment config file, restart your terminal, and run `source activate ml6036` in terminal.

Solution 2

Run the following commands to install all the necessary packages. Note that this solution works for Mac/Linux only and is more prone to trouble. If you encounter a package version conflict and don't know how to solve it, we suggest you choose solution 1.

[7] `http://ist.mit.edu/athena`
[8] `http://conda.pydata.org/miniconda.html`
[9] `http://conda.pydata.org/docs/install/quick.html`

```
sudo pip install numpy
sudo pip install scipy
sudo pip install yaml
sudo pip install cython
sudo pip install h5py
sudo pip install git+git://github.com/Theano/Theano.git
sudo pip install keras
```

Image features

Many other feature representations of images have been developed besides the approaches considered here. Some examples of widely used image features include HOG, SIFT, and GIST. All of them are variations on the same theme: abstract away from raw pixels, and form more general representations of image content. This can be done manually by histogramming and averaging measurements in different image regions. These measurements are often pixel intensities (grayscale or color) or gradients (edges). A histogram over an image region just specifies how much of each measurement there is in a region (e.g. how many horizontal edges, vertical edges, etc.) without specifying the exact pixel location. This allows for a more general representation that will still match images that have similar distributions of features but are not identical at the pixel level.

Recently, deep learning approaches have outperformed complex manually-engineered features like HOG/SIFT in large image datasets. The original appeal of neural networks was a general purpose model that could learn useful features directly from raw pixels. However, in the past few years, the state of the art for computer vision tasks like object recognition has been significantly improved by spending effort to develop complex CNN architectures[10] in place of feature-engineering.

[10] GoogleNet architecture: `http://homes.cs.washington.edu/~jmschr/lectures/googlenet.png`
For more famous CNN architectures, see: `http://cs231n.github.io/convolutional-networks/#case`