Image Feature Analysis

Nicolas Langley 904433991

December 21, 2014

1 Introduction

This project looks to perform analysis of image characteristics using a Convolutional Neural Network in order to identify specific image features. This report will explore the datasets used for testing and their composition. It will also examine the different techniques explored and implemented for attempting to identify specific image features from a set of images.

The approach for using these techniques will be examined as will proposals for further work on this topic.

2 Dataset Overview

An overview of the datasets used in this project will be presented as follows:

1. Dataset Contents
2. Dataset Origins
3. Relevance to Project

2.1 MNIST Dataset

1. The MNIST dataset contains a large number of handwritten digits. It contains 50,000 training examples as well as 10,000 testing samples. Each of the samples is a single handwritten digit from 0 to 9 centered in a $28x28$ image. This centering is done by computing the center of mass of the pixels in the data and translating the image so this point is at the center of the $28x28$ image.

2. The dataset is a subset of the a database compiled by the National Institute of Standards and Technology (NIST) comprised of handwritten digits by both Census Bureau Employees and High School students. The MNIST dataset is a re-mixed subset where half of the training images and half of the testing images were taken from each of the Census Bureau Employee and High School student sets respectively.

3. This dataset is very commonly used as an initial dataset for the testing of different machine learning techniques. For this project, it was used as the initial dataset for testing the implementation of the convolutional neural network (CNN) approach used. It was also used in the testing of the PCA-based technique. For the purpose of working with the CNN, the training dataset was divided into both a training and a validation data set

Figure 1: Sample of the MNIST dataset

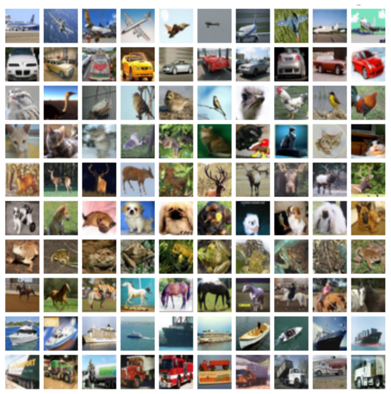
2.2 CIFAR Dataset

1. The CIFAR datasets are two datasets that contain a set of labelled $32x32$ color images. There are two versions of the dataset: one where the images are divided into 10 classes (CIFAR-10) and one where they are divided into 100 different classes (CIFAR-100). For this project, the CIFAR-100 dataset was used with a focus of the "coarse" labels provided (there are 20 coarse labels and 100 fine labels)

2. These datasets are labelled subsets of the Tiny Images Dataset. The Tiny Images Dataset is a dataset of 80 million different $32x32$ images. The CIFAR datasets are comprised of a sampling of these images where the chosen images have been divided into classes and labelled.

3. Within the context of this project, the CIFAR dataset was used as input to the Convolutional Neural Network as a more complex dataset than the MNIST dataset with real world (useful) classes that could be used in the identification process. A copy of the dataset was constructed where the images have been simplified by reducing their size to $28x28$ and converted to grayscale images. The images have also been whitened using PCA

(a) Sample of original CIFAR dataset



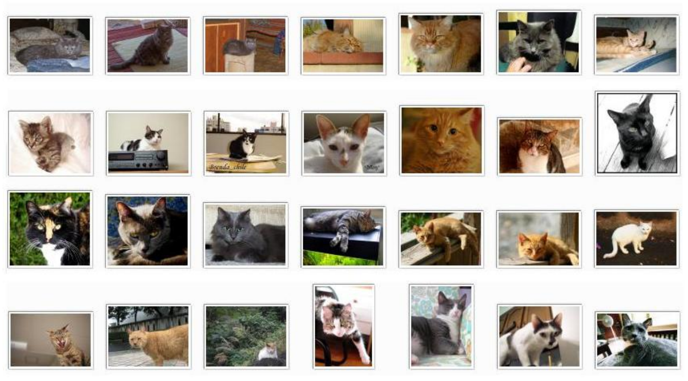
2.3 Custom Dataset

1. The custom dataset used is a smaller subset of images then the two datasets previously discussed. It is composed of 3 smaller sets: one training, one validation and one testing set. For the custom dataset that was assembled, there are 300 images in the training set, 40 in the validation set and 40 in the testing set. The training set is composed of pictures of cats that are a variety of different sizes. These have all been normalized using a Whitening Transformation and resized to be of size $28x28$. Both the validation and testing sets contain 20 images of cats and 20 images of buildings. These sets have also been normalized and resized.

2. The data for these datasets was taken from two seperate larger datasets. The pictures of cats have been taken from the Asirra data set, which contains millions of images of cats and dogs labelled manually by people at animal shelters and provided to Microsoft Research. The images of buildings come from a dataset of buildings in different orientations.

3. This dataset is the most important for the purpose of this project as it is intended to provide a clear look at the features of a single class of object, in this case a cat. Ideally, examining the results achieved using this dataset will be helpful in further exploring what the differences and similarities can be looked at these images.

(a) Cat images for custom dataset



3 Overview of Techniques

This project involved the exploration of a number of techniques used to determine the features of an image. The technique that was settled on and used for the attempted gathering of results was the Convolutional Neural Network, a machine learning technique commonly used in computer vision problems. A Convolutional Neural Network relies on concepts taken from MultiLayer Perceptrons as well as Logistic Regression classifiers. These two concepts were also explored in detail and their details and implementations will be detailed in this section. Principal Components Analysis was used in the data preprocessing stage. This application of PCA will also be explained. This section will detail the approach taken for testing each of these techniques as well as their contributions with respect to the goal of the project.

3.1 Principal Components Analysis and Whitening

3.1.1 Technique Overview

Principal Components Analysis (PCA) is a technique for converting a set of correlated variables into a set of linearly uncorrelated variables called Principal Components. The Principal Components are orthogonal to each other and starting with the first Principal Component, represent (in decreasing order) the largest possible variance. These Principal Components can be found by taking the eigendecomposition of the covariance matrix of a set of data. The Principal Components correspond to the eigenvectors of this decomposition.

In this project, PCA was initially used as an approach for extracting the features of an image. This approach involves looking at a large set of like-images and extracting the Principal Components. These Principal Components will represent the areas of highest variance within the image and can be interpreted as being the features of the image set. This approach is very sensitive to image orientation and did not lend itself to the goal due to these restrictions.

Within the context of a Convolutional Neural Network, PCA was used in preparation of the image data used as a part of the Whitening Transformation. The Whitening Transformation is a decorrelation transformation that transforms a set of random variables with a covariance matrix, M into a set of random variables with the Identity matrix as their covariance matrix. This transformation serves to change the input vector into a white noise vector where all the random variables are decorrelated and their variance is 1. This transformation involves the following steps:

1. Given a data matrix $X_{k,n}$, center the data by subtracting the mean.
2. Compute the covariance matrix $\Sigma = XX^T/n$
3. Take the Eigenvalue Decomposition of Σ to be $\Sigma = EDE^T$
4. Re-arrange to get $D = E^T \Sigma E$
5. Multiply the data X by $D^{-\frac{1}{2}}E^T$ to get the 'whitened' data

3.1.2 Implementation

The implementation for this project was done in Python using a collection of scientific libraries. See the implementation of the Whitening Transformation below:

```
def whitening_transform(X):
```

```
''' Whiten the image - decorrelate and variance to 1 '''
# Subtract the mean from X
X_mean = np.mean(X)
X_norm = X / X_mean
# Get covariance matrix X^T X
cov = np.dot(X_norm.T,X_norm)
# Get eigendecomposition of covariance matrix
d, V = np.linalg.eigh(cov)
D = np.diag(1. / np.sqrt(d))
# Whitening matrix
W = np.dot(np.dot(V, D), V.T)
X_white = np.dot(X, W)
return np.array(X_white)
```

3.2 Logistic Regression

3.2.1 Technique Overview

Logistic Regression is a basic probabilistic classification model. The purpose of the Logistic Regression technique is to predict the outcome of a class label based on a set of features. The classification process can be viewed as projecting an input vector onto a set of hyperplanes and determining the distance of the input to each hyperplane. The hyperplanes correspond to classes and distance of the input vector to any given hyperplane is used to determine the probability that the input matches the class defined by that hyperplane.

The model takes as parameters a matrix W, which is a weight matrix, and b, which is a bias vector. With this information, the probability that an input vector x is a member of class i is:

$$\begin{aligned} P(Y = i|x, W, b) &= \frac{e^{W_i x + b_i}}{\Sigma_j e^{W_j x + b_j}} \\ &= softmax(Wx + b) \end{aligned}$$

The $softmax$ function transforms a K-dimensional vector of real values into a K-dimensional vector of real values in the range [01] which represents a categorical probability distribution.

Once the probability of the input belonging to any given class has been determined, the chosen class prediction is the class for which the highest probability was found i.e. $y_{pred} = max_i(P(Y = i|x, W, b))$

Using a Logistic Regression classifier involves training the parameters of the model using a training data set. Training is done by attempting to minimize a loss function. For this project, the negative log-likelihood was used as a measure of loss. Gradient descent is then used to generate the optimal parameters. Once a model has been trained, it is possible to supply a test input vector and generate a predicted class label.

3.2.2 Implementation

Implementing a basic Logistic Regression classifier was done in Python using the Theano library. Below is the *LogisticRegression* class:

```
import numpy as np
import theano as th
import theano.tensor as T

class LogisticRegression(object):

    def __init__(self, input, n_in, n_out):
        # Initialize the weight matrix W with zeros
        self.W = th.shared(value=np.zeros((n_in,n_out),dtype=th.config.floatX),
                           name='W',
                           borrow=True)
        # Initialize the biases b as a vector of zeros
        self.b = th.shared(value=np.zeros((n_out,),dtype=th.config.floatX),
                           name='b',
                           borrow=True)
        # Symbolic expresion for computing the probability matrix of class-membership
        # i.e. P(Y|x,W,b)
        self.py_given_x = T.nnet.softmax(T.dot(input, self.W) + self.b)
        # Symbolic expression for computing the prediction of the class with max-probability
        self.y_pred = T.argmax(self.py_given_x, axis=1)
        # Set the parameters of the model
        self.params = [self.W, self.b]

    def negative_log_likelihood(self, y):
        # Define the likelihood
        log_likelihood_probs = T.log(self.py_given_x)
        minibatch_examples = T.arange(y.shape[0])
        # Return mean log-likelihood across the minibatch
        return -T.mean(log_likelihood_probs[minibatch_examples, y])

    def errors(self, y):
        # Check that y has the same dimensions as y_pred
        if y.ndim != self.y_pred.ndim:
```

```
        raise TypeError('y should have the same shape as self.y_pred',
                        ('y', y.type, 'y_pred', self.y_pred.type))
    # Check that y has the right datatype
    if y.dtype.startswith('int'):
        # Return any mistakes in prediction (a vector of 1s and 0s)
        return T.mean(T.neq(self.y_pred, y))
    else:
        raise NotImplementedError()
```

3.3 Multilayer Perceptron

3.3.1 Technique Overview

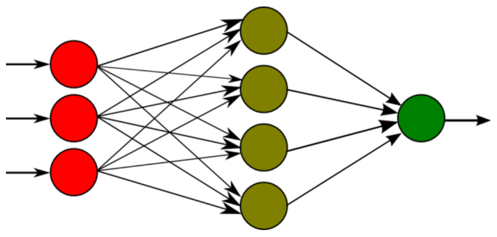
The term Multilayer Perceptron (MLP) refers to a form of feed forward network. At a basic level, this can be viewed as an extension on a logistic regression classifier where the input is first transformed using a non-linear transformation, Φ. This transformation is learned according to some given training data. This transformation projects the input data (still an input vector) into a space where it is linearly separable, i.e. sets of input data are mapped onto a set of appropriate outputs.

A MLP is formally represented as a finite directed acyclic graph where the nodes (sometimes called neurons) can be classified according to the following criterion:

- nodes that have no incoming connections are called **Input Neurons**
- nodes that have no outgoing connections are called **Output Neurons**
- all other nodes are called **Hidden Neurons**

The nodes of the graph (neurons) are organized into layers. Below is an example of a single-layer MLP:

Figure 4: Single Layer MLP



The nodes in a MLP are called neurons because MLPs were developed to model the workings of biological neurons in the brain. Each of the neurons in an MLP has an activation function. This function maps the weighted inputs to the output of each neuron. The activation function used is nonlinear and models the frequency of action potentials (this is what happens in the brain). The activation function used is typically either $y(v_i) = tanh(v_i)$ or $y(v_i) = (1+e^{-v_i})^{-1}$. Note that both of these functions are sigmoids (S-shaped curves).

In this project, the MLP is parameterized by a set of weight matrices, $W_{i,j}$, and bias vectors, b_i. The weight matrices correspond to the weights of the connections between the nodes in layer i to the nodes in layer j. Similarly, the bias vector b_i, defines any biases towards certain nodes in layer i.

In order for a MLP to be an effective tool, the transformation Φ must be learned. This amounts to determining the appropriate bias and weight values. The transformation can be formally expressed as:

$$\Phi(x) = s(b_i + W_i x)$$

In this formula x is the input to layer i, b is the bias vector, W is the weight matrix and s is the activation function.

Learning the set of required parameters Θ is done using a gradient descent approach similar to that used in the logistic regression implementation. The gradients, $\frac{\partial \ell}{\partial \Theta}$, are obtained using the backpropogation algorithm.

To use an MLP for classification of input, logistic regression can be used to determine the output. When this is the case, the output of an MLP can be formally described as:

$$o(x) = softmax(b_n + W_n \Phi(x))$$

Where n is the number of layers in the MLP.

3.3.2 Implementation

The implementation of a MLP was done in Python using the Theano library. Below is shown the code for the class that represents the hidden layer of the MLP. This layer is connected to an instance of the*LogisticRegression* classifier.

```
# Class for the Hidden Layer of the MLP
class HiddenLayer(object):
    def __init__(self, rng, input, n_in, n_out, W=None, b=None, activation=T.tanh):
        # Weights of the hidden layer are uniformly sampled from a symmetric interval
        # Weights are dependant on the choice of activation function
        if W is None:
            # For tanh as activation function, this is the interval to use
            W_values = np.asarray(rng.uniform(low=-np.sqrt(6. / (n_in + n_out)),
                                              high=np.sqrt(6. / (n_in + n_out)),
                                              size=(n_in, n_out)),
                                  dtype=th.config.floatX)
            # If the activation function is a sigmoid, multiply the weights by 4
            if activation == T.nnet.sigmoid:
                W_values *= 4
            # Make a shared Theano variable for the weight matrix W
            W = th.shared(value=W_values, name='W', borrow=True)
        # Set the biases to be an array of zeros
        if b is None:
            b_values = np.zeros((n_out,), dtype=th.config.floatX)
            b = th.shared(value=b_values, name='b', borrow=True)

        self.W = W
        self.b = b
        # Compute the output of the activation function for the hidden layer
        # output is h(x) = s(b + Wx) where s is the activation function
        lin_output = T.dot(input, self.W) + self.b
        self.output = (lin_output if activation is None
                       else activation(lin_output)) # Apply activation function to lin_output
        # Set the paramters of the model
        self.params = [self.W, self.b]
```

3.4 Convolutional Neural Network

3.4.1 Technique Overview

A Convolutional Neural Network (CNN) is a variant of the previously seen Multilayer Perceptron. CNNs are inspired by the structure of the visual cortex and its complex arrangement of cells.

The visual cortex has been shown to be comprised of cells that are sensitive to small sub-regions of the visual-field. These sub-regions can be tiled to cover the entire visual field and are called receptive fields. These cells act as local filters over the input space and result in an exploitation of the spatial local correlation that occurs in natural images.

Convolutional Neural Networks were designed to mirror the functionality of the visual cortex. They do this by enforcing a local connectivity pattern between neurons of adjacent layers to exploit spatial local correlation as well as using convolution of the input with a linear filter to repeatedly apply a function across sub-regions of the entire input.

There are a number of CNNs modeled after the visual cortex. The architecture followed in this project is for the LeNet CNN developed by Yann LeCun at NYU.

CNNs enforce local connectivity by having the inputs to the nodes in a hidden layer h be from a subset of units in layer $h-1$ that have spatially contiguous receptive fields. This results in each node being unresponsive to variations outside of its 'receptive field', ensuring the strongest response to a spatially local input pattern.

Convolution is used in the creation of a feature map. A feature map is a set of replicated nodes (replicated by applying a filter h_i across the visual field) that share the same parameterization. One of these feature maps is obtained by convolution of the input with a linear filter, adding a bias and applying a non-linear function. A feature map h_k can be obtained by:

$$h^k_{i,j} = s((W_k * x)_{i,j} + b_k)$$

In the above formula, $*$ is the convolution operator, W is the weight matrix from layer $k-1$ to k, b is the bias vector at layer k and s is the non-linear function used.

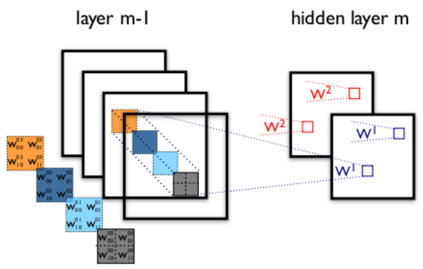
The definition of convolution on a 2D signal is:

$$o(m,n) = f(m,n) * g(m,n) = \sum_{u=-\infty}^{\infty} \sum_{v=-\infty}^{\infty} f(u,v)g(m-u,n-v)$$

The implementation of a CNN, typically has each hidden layer composed of multiple feature maps to form a richer representation of the data. In this case, the connections, and weights, between two layers $m-1$ and m are from each pixel of the kth feature map in layer m to the pixel at coordinates (i,j)

of the ith feature map of layer $m-1$. The weights are then $W_{i,j}^{k,l}$. An example of a convolutional layer can be seen below:

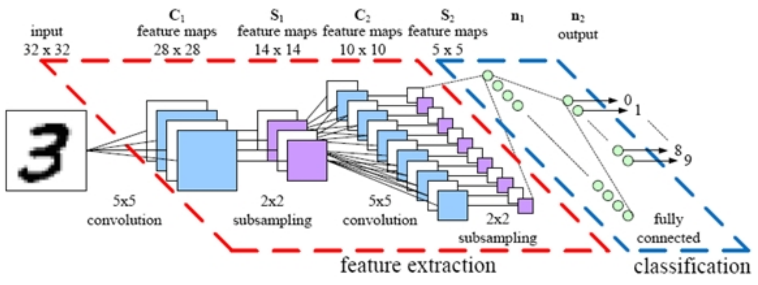
Figure 5: Convolutional Layer Example



In addition to a convolutional layer, CNNs also employ a form of non-linear down-sampling called **max pooling**. Max-pooling partitions an input into a set of non-overlapping rectangles and outputs the maximum value for each sub region. This step serves to reduce the computation required in upper layers of the CNN by eliminating non-maximal values as well as providing additional robustness to position.

The CNN constructed for this project involves two convolutional layers, two max-pooling layers and one fully-connected hidden-layer (like that in an MLP) used for classification. An example of the architecture is:

Figure 6: Convolutional Neural Network Example for MNIST dataset



3.4.2 Implementation

The Convolutional Neural Network used in this project was written in Python using the Theano library. Below is the source code for the Convolutional and Max-Pooling layers of a CNN:

```
class ConvPoolLayer(object):
    ''' Convolution and Max-Pooling Layer of a LeNet convolutional network '''
    def __init__(self, rng, input, filter_shape, image_shape, poolsize=(2,2)):
        # Ensure image and filter have same size
        assert image_shape[1] == filter_shape[1]
        self.input = input

        # Set the number of inputs to hidden layer
        # There are "num_input_feature_maps * filter-height * filter_width" inputs to hidden layer
        fan_in = np.prod(filter_shape[1:])
        # Lower layer gets gradient from "inputs / pooling size"
        fan_out = filter_shape[0] * np.prod(filter_shape[2:]) / np.prod(poolsize)

        # Initialize random weights
        W_bound = np.sqrt(6. / (fan_in + fan_out))
        self.W = th.shared(np.asarray(rng.uniform(low=-W_bound,
                                                  high=W_bound,
                                                  size=filter_shape),
                                      dtype=th.config.floatX),
                           borrow=True)

        # Bias is a 1d tensor with one bias per ouput feature map
        b_values = np.zeros((filter_shape[0],), dtype=th.config.floatX)
        self.b = th.shared(value=b_values, borrow=True)

        # Convolve input feature maps with the filters
        conv_out = conv.conv2d(input=input,
                               filters=self.W,
                               filter_shape=filter_shape,
                               image_shape=image_shape)

        # Downsample each feature map using maxpooling
        pooled_out = downsample.max_pool_2d(input=conv_out,
                                            ds=poolsize,
                                            ignore_border=True)
        # Broadcast bias across mini-batches and feature map
        self.output = T.tanh(pooled_out + self.b.dimshuffle('x', 0, 'x', 'x'))
        # Store the model parameters
        self.params = [self.W, self.b]
```

The CNN is comprised of four layers:

- Two **Convolution and Max-Pooling layers**

- One **Fully-connected Hidden Layer**
- One **Logistic Regression Classifier**

Below is the source code for a complete CNN implementation:

```
class CNN(object):
    '''
        Convolutional Neural Network with 2 convolutional pooling layers
        NOTE: Dataset is required to be 28x28 images with three sub data sets
    '''
    def __init__(self, datasets, batch_size=500, nkerns=[20, 50], img_size=(28, 28), learning_rate=0.1):

        train_set_x, train_set_y = datasets[0]
        valid_set_x, valid_set_y = datasets[1]
        test_set_x, test_set_y = datasets[2]

        self.batch_size = batch_size
        # compute number of minibatches for training, validation and testing
        self.n_train_batches = train_set_x.get_value(borrow=True).shape[0]
        self.n_valid_batches = valid_set_x.get_value(borrow=True).shape[0]
        self.n_test_batches = test_set_x.get_value(borrow=True).shape[0]
        self.n_train_batches /= batch_size
        self.n_valid_batches /= batch_size
        self.n_test_batches /= batch_size

        # allocate symbolic variables for the data
        self.index = T.lscalar()  # index to a [mini]batch
        self.x = T.matrix('x')
        self.y = T.ivector('y')

        rng = np.random.RandomState(23455)

        layer0_input = self.x.reshape((batch_size, 1, img_size[0], img_size[1]))

        # Create the two convolutional layers that also perform downsampling using maxpooling
        self.layer0 = ConvPoolLayer(rng,
                                    input=layer0_input,
                                    image_shape=(batch_size, 1, img_size[0], img_size[1]),
                                    filter_shape=(nkerns[0], 1, 5, 5),
                                    poolsize=(2,2))

        self.layer1 = ConvPoolLayer(rng,
                                    input=self.layer0.output,
                                    image_shape=(batch_size, nkerns[0], 12, 12),
                                    filter_shape=(nkerns[1], nkerns[0], 5, 5),
                                    poolsize=(2,2))

        layer2_input = self.layer1.output.flatten(2)

        # Create the hidden layer of the MLP
        self.layer2 = HiddenLayer(rng,
                                  input=layer2_input,
                                  n_in=nkerns[1] * 4 * 4,
                                  n_out=500,
                                  activation=T.tanh)

        # Create the logistic regression layer for classifiying the results
        self.layer3 = LogisticRegression(input=self.layer2.output, n_in=500, n_out=10)

        self.cost = self.layer3.negative_log_likelihood(self.y)

        self.params = self.layer3.params + self.layer2.params + self.layer1.params + self.layer0.params

        self.grads = T.grad(self.cost, self.params)

        # Update list for the paramters to be used when training the model
```

```
updates = [(param_i, param_i - learning_rate * grad_i)
           for param_i, grad_i in zip(self.params, self.grads)]

# This function updates the model parameters using Stochastic Gradient Descent
self.train_model = th.function([self.index],
                               self.cost, # This is the negative-log-likelihood of the Logisitc Regression layer
                               updates=updates,
                               givens={self.x: test_set_x[self.index * batch_size: (self.index + 1) * batch_size],
                                       self.y: test_set_y[self.index * batch_size: (self.index + 1) * batch_size]})

# These are Theano functions for testing performance on our test and validation datasets
self.test_model = th.function([self.index],
                              self.layer3.errors(self.y),
                              givens={self.x: test_set_x[self.index * batch_size: (self.index + 1) * batch_size],
                                      self.y: test_set_y[self.index * batch_size: (self.index + 1) * batch_size]})

self.validate_model = th.function([self.index],
                                  self.layer3.errors(self.y),
                                  givens={self.x: valid_set_x[self.index * batch_size: (self.index + 1) * batch_size],
                                          self.y: valid_set_y[self.index * batch_size: (self.index + 1) * batch_size]})
```

4 Application of Techniques

This section will outline the usage of the techniques described above.

4.1 Goal

The desired outcome of this project is to gain insight into the similarity of sets of images. To this end, the project looks to examine the analysis of a set of like-images resulting in a description of the unique 'features' of that set that can then be used to examine similarity of images with respect to the discovered features. This task is closely intertwined with current computer vision techniques used in the identification of images and so another intended result was an exploration of the techniques currently used for image identification tasks.

4.2 Approach

This section will outline the usage of the different techniques outlined in the previous section. It will also examine the process used to attempt to achieve the proposed goal.

As described in the previous section, the techniques used in this project are as the follows:

1. Principal Components Analysis
2. Logistic Regression Classifier
3. Convolutional Neural Network (incorporates Multilayer Perceptron)

What follows is an outline of the how the above techniques were used to attempt to extract the features of a given dataset D:

- **Whitening the Data**
  Principal Components Analysis was used to apply the Whitening Transformation to the input data. This results in pre-processing the data to be uncorrelated with a variance of 1. This boosts the areas of high contrast in the image and increases the accuracy of the analysis techniques

- **Training the Convolutional Neural Network** The dataset D is used to train a Convolutional Neural Network. This process causes the parameters of the CNN model to be unique to the given dataset. In other words, what is obtained by training a CNN, is a unique set of parameters Θ that corresponds to the weights and biases of the different layers.

- **Verify of Accuracy of the CNN Classification** After a Convolutional Neural Network has been trained, verification of the results obtained is necessary to see if the parameters yield a satisfactory result in labeling new data.

The above process outlines the steps necessary to take a dataset D and obtain a unique description of the dataset in the form of a set of weights and bias vectors for the different layers of the CNN. For this approach to work, the dataset must be images that have been identified to be alike. In testing this process, the custom dataset outlined in section 2.3 was used.

This initial approach was undertaken with the idea that the weights learned by the CNN on a specific set of images would provide a description of the features of a class of images, e.g. a cat. In practice, however, the data gathered by using the custom dataset in 2.3 did not clarify anything in terms of providing a possible outline of the features of a cat (in the case of the described dataset). The values generated by the custom dataset show no noticeable differences from the weights generated on the MNIST dataset (2.1) which clearly yields different features. Below are a small sample of the weights generated between the first and second layers of the CNN after training on the custom dataset for 10 epochs:

$$\begin{pmatrix} 0.004747 & 0.004747 & -0.04272 \\ 0.00119 & 0.00119 & -0.010782 \\ -0.006756 & -0.006756 & 0.060812 \\ 0.006241 & 0.00624135 & -0.05617 \\ -0.007345460 & -0.0073454 & 0.06610919 \end{pmatrix}$$

The above example is a small excerpt of the trained weights, but in general there is no clearly discernable feature representation within the weights. While the initial approach was largely unsuccessful in yielding any insight into the feature makeup of images, I will now propose the possible direction for future work to be done.

Moving forward, a modification to the approach of identifying images will

be applied. Instead of gaining information about the features of an image, and as a result being able to judge the similarity of two images, similarity and "features" could be examined at a higher level of abstraction. By taking the Convolutional Neural Network approach from above and, instead of focusing on the results of applying a single CNN to a set of images, training multiple CNNs on a base set of broad features, it would be possible to apply each of the CNNs in turn to an image and gain some insight into the feature composition of the image based on which of these broad, pre-defined, features are shown to occur in the image. An outline of this approach is as follows:

- **Train multiple CNNs on image data sets that exemplify a pre-defined set of features** The challenge in this step is obtaining a set of data that corresponds to a number of pre-defined features. An initial approach to this would be to, for example, focus on a small subset of images that have recognizable (and easily-identifiable) features, e.g. cat ears and cat eyes.

- **For a given image, determine which of the CNNs match the image with the class (feature) of that CNN** By applying each of the CNNs to an input image, and analyzing the outputted labels, we can determine which of the features match.

- **Use the obtained feature details for comparison or classification purposes** With an initial set of broad features generated using the above step, it will be possible to perform a myriad of comparison and classification tasks on a given set of images. An example of this type of task can be seen if we look to examine two images and provide insight into how similar they are. By looking at the features that the two images have been shown to demonstrate, it is easy to look at which features match and perform an analysis as a result.

This approach will look to allow for the ability to compare and classify images based on their features.

5 Conclusion and Commments

This project was very interesting. The problem area is a difficult one to navigate and it was never clear if the chosen approach would lead to tangible results. In the end, the goal of gaining insight into the features of an image and performing some comparison was not completely achieved. However, I was able to learn a significant amount about the techniques used to classify images and how the cutting-edge neural network techniques are applied. An interesting application of PCA techniques was used in examining the Whitening transformation as well. Working on this project also was able to lead to insight into a potential approach for further exploring how to distinguish images based on their features.

6 Technical Challenges

6.1 Python Libraries

This project is implemented in Python using a collection of libraries. The most important of these are Scipy (and by extension NumPy) as well as Theano. A brief overview of these libraries and the challenges involved in using them follows.

6.1.1 SciPy

SciPy is the de-facto library for scientific computing in Python. It's core module, NumPy, provides datatypes that easily allow for the representation of matrices and other useful structures. Using SciPy and NumPy easily allows for a MATLAB-like workflow. There are, however, a large amount of small differences that makes transferring basic MATLAB knowledge directly to Python difficult.

6.1.2 Theano

Theano is a machine learning library developed primarily by researchers at the University of Montreal. It provides a framework for defining and evaluating mathematical expressions involving multi-dimensional arrays efficiently. This library is used extensively for the development of the CNN used in this project. The challenges with using Theano that I came across is the focus on running computations on a GPU as well as the extensive use of symbolic functions. An example of the symoblic function usage is that many functions are defined as taking input that is itself only a symbolic representation. This relies on supplying of the value of this input at a later date, at which point all of the values within a function defined in terms of this input are evaluated. Theano also makes use of a concept of shared variables which are stored on the GPU. This workflow is very different from that of a system like Matlab where much of these details are abstracted out.