A REPORT

ON

Assignment II

Kinship Verification

BY

PRITHVI RAJ	2016A7PS0013P
DHAIVATA PANDYA	2016A7PS0020P
ADITCHANDRA	2016A3PS0256P

IN PARTIAL FULFILLMENT OF THE COURSE
MACHINE LEARNING (BITS F464)
FOR I SEMESTER, 2018-19

BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE, PILANI

NOVEMBER 2018

Table of Contents

1. Introduction

1.1 Problem Statement

Kinship Verification: To verify kinship relationship from facial images of parents and children. Also try to derive sibling relationships using without training a separate model.

Learn about metric learning and apply it in the context of the above problem.

Also compare and contrast deep learning techniques with traditional machine learning techniques and try to implement model for kinship verification using CNN.

1.2 Objectives

- Model the problem of kinship verification as a machine learning problem.
- Collect an appropriate dataset.
- Pre-process the data accordingly.
- Implement a solution for the problem using metric learning.
- Implement a solution for the problem using CNN.
- Train and test the models.
- Compare deep learning with traditional machine learning techniques.
- Analyse the results.

2. Metric Learning

2.1 Motivating Example

Fig 1. Sample face image dataset

Consider the images in Figure 1, and imagine a scenario in which we must compute a similarity measure pairs of images (for example, for clustering or nearest neighbor classification). The problem is precisely how to assess the similarity between a pair of images. For instance, our goal could be to find matching faces based on identity or it could be to determine the pose of an individual. Clearly the two situations should not be using the same similarity measure between images. To handle multiple similarity or distance metrics, we could attempt to determine by hand an appropriate distance function for each task, by an appropriate choice of features and the combination of those features. However, this approach may require significant effort and may not be robust to changes in the data. An alternative is to apply metric learning, which aims to automate this process and learn task-specific distance functions in a supervised manner.

2.2 Global Metric Learning

Formally, the metric learning problem can be stated as follows: given an input distance function $d(x, y)$ between objects x and y (for example, the Euclidean distance), along with supervised information regarding an ideal distance (relative or absolute), construct a new distance function $d'(x,y)$ which is "better" than $d(x,y)$ at achieving the ideal distance.

If the distance function $d'(x,y)$ is of the form $d(f(x), f(y))$ for some function f, then it is called global metric learning. That is, we learn some global mapping function to be applied to all the points and use the original distance function over the mapped data.

For our discussion, we will be using euclidean distance as the distance function $d(x,y)$.

2.3 The Mahalanobis distance

In the metric learning literature, the term "Mahalanobis distance" is often used to denote any distance function of the form:

$$d_A(x,y) = (x-y)^T A(x-y)$$

where A is some positive semi-definite matrix (i.e., its eigenvalues are non-negative). We can view this distance simply as applying a linear transformation of the input data: since A is positive semi-definite, we factorize it as $A = G^T G$ and simple algebraic manipulations shows that $d_A(x,y) = ||Gx - Gy||_2^2$. Thus, this generalized notion of a Mahalanobis distance exactly captures the idea of learning a global linear transformation.

If A is an identity matrix, then the Mahalanobis distance is equal to the euclidean distance.

2.4 Unsupervised Metric Learning and Dimensionality Reduction

A number of classical dimensionality reduction methods may be viewed as Mahalanobis distance learning methods. For instance, consider principal components analysis (PCA), which finds a linear transformation to map the data from an input space to a lower-dimensional space such that the lower-dimensional projected data is as informative as possible, in that it captures as much of the variance of the data as possible. Principal components analysis can be viewed as constructing a linear transformation P to be applied globally to the data, in an unsupervised manner. The resulting distance between objects is therefore $d(Px, Py)$, and one may claim that this is also a form of metric learning.

2.5 Regularized Transformation Learning

Let us suppose that we have a set of points $x_1, x_2, \ldots x_n$ in the euclidean space. Let $X = x_1, x_2, \ldots x_n$ be the matrix of all data points.

To encode the supervision, we assume that we are given a collection of *m* loss functions, which we will denote as $c_1, c_2, \ldots c_m$; The assumption is that each loss function depends on the data only through the inner product matrix $X^T AX$. For instance, one loss function might encode the squared loss between the target distance between x_i and x_j and the squared Euclidean distance between x_i and x_j using the Mahalanobis distance with *A*.

The second part of the model is a regularizer on the model, which we will denote as *r*(*A*), and will be a function of *A*. Putting these two together, we obtain the general model, a linear combination between these two components of the model:

$$\mathcal{L}(A) = r(A) + \lambda \sum_{i=1}^{m} c_i(X^T AX)$$

The λ term is a trade-off between the regularizer and the loss. The goal will be to find the minimum of *L*(*A*) over the domain of *A*.

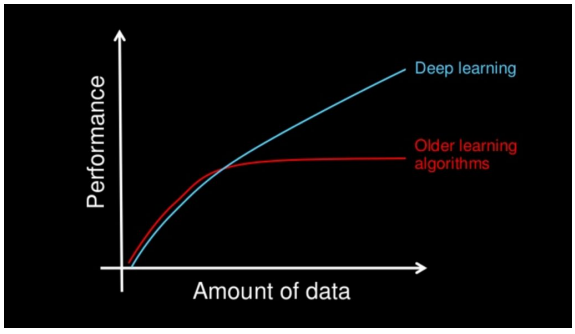
3. Deep Learning

3.1 Introduction to Deep Learning

Deep learning is a type of machine learning based on algorithms inspired by the functioning and structure of the brain called artificial neural networks. Deep learning is based on learning data representations rather than specific algorithms. This type of learning can be supervised, semi-supervised or unsupervised.

Deep learning architectures have been used in numerous fields such as computer vision, machine translation, natural language processing, speech recognition, social network filtering, bio-informatics, drug design and medical image analysis, and it has been seen that they produced results similar and at times, even better than the ones given by human domain experts.

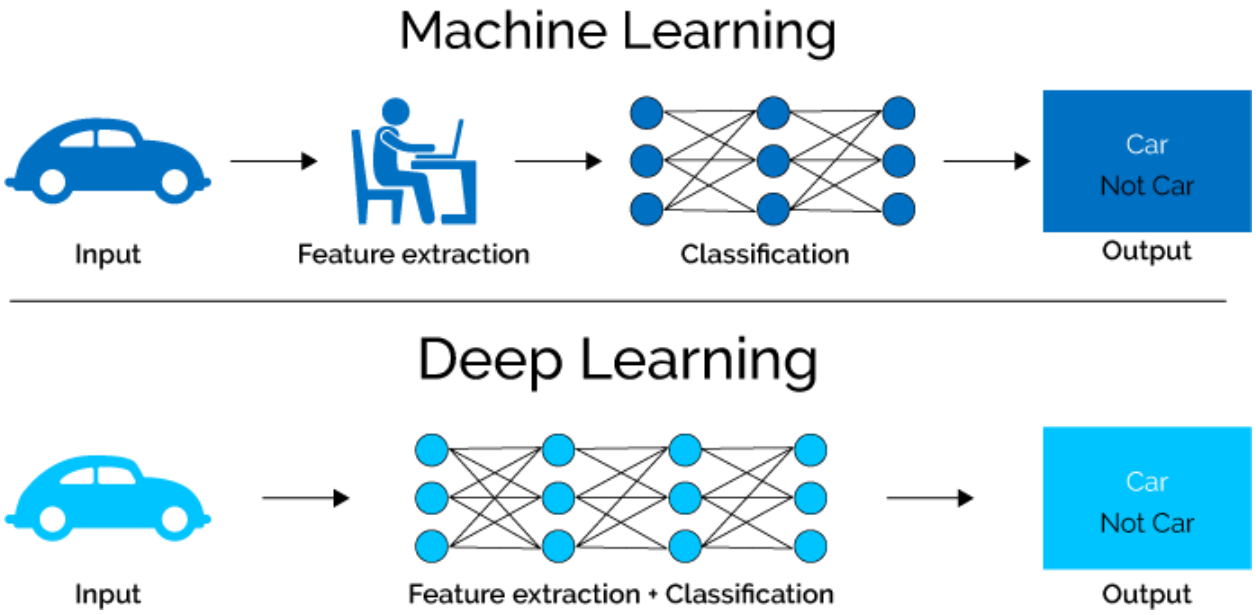
The heart of deep learning is that there are sufficiently fast computers and sufficient data to train large NNs. Deep learning techniques are more scalable than traditional machine learning techniques. As we make larger NNs and train them with more and more data, their performance continues to increase. This is generally different to other ML techniques that reach a saturation in performance.



3.2 Deep Learning Compared to Traditional ML Techniques

A major advantage of deep learning is that is is powered by a massive amount of data. In terms of computer architecture requirements, deep learning requires high performance GPUs compared to traditional ML techniques.

In conventional ML techniques, a domain expert is required to identify most of the applied features in order to reduce the complexity of the data and make patterns more visible to learning algorithms to work well. This is where deep learning algorithms come into play as they learn high-level features from data in an incremental manner. This eliminates the need for feature extraction and domain expertise.



Deep learning techniques tend to solve any problem end to end, whereas traditional ML methods need to break down the problem into different sub problems to be solved first and then obtain their results to be combined at a later stage.

Due to the immense number of parameters, deep learning models usually take a long time to train. Some algorithms like ResNet may even take a fortnight to finish training from scratch. However, their traditional machine learning counterparts take comparatively lesser time, from a few seconds to hours. Conversely, in the testing phase, deep learning algorithms usually take lesser time to run and on being compared to ML algorithms like *k*-NN, it is revealed that test phase time increases on increasing the

size of data. However, it is to be noted that this is not applicable to all ML algorithms, as there are some exceptions with smaller test times as well.

One reason why many sectors prefer traditional ML techniques to deep learning is interpretability. For instance, let us assume that we have used a deep learning model to compute the relevance score of a document. Although it performs well and the results comparable to human performance, it does not reveal why it has given that score. Mathematically, we can find out which nodes of a deep neural network were activated, but we don't know what there neurons were supposed to model and what these layers of neurons were doing collectively. However, when it comes to traditional ML techniques like decision trees, logistic regression, etc. this issue does not exist and they are quite straightforward when it comes to interpretation.

3.3 Deep Learning in the context of Kinship Verification

The following architecture for a CNN was proposed by Zhang et. al [2]. for solving the problem of kinship verification using convolutional neural network.

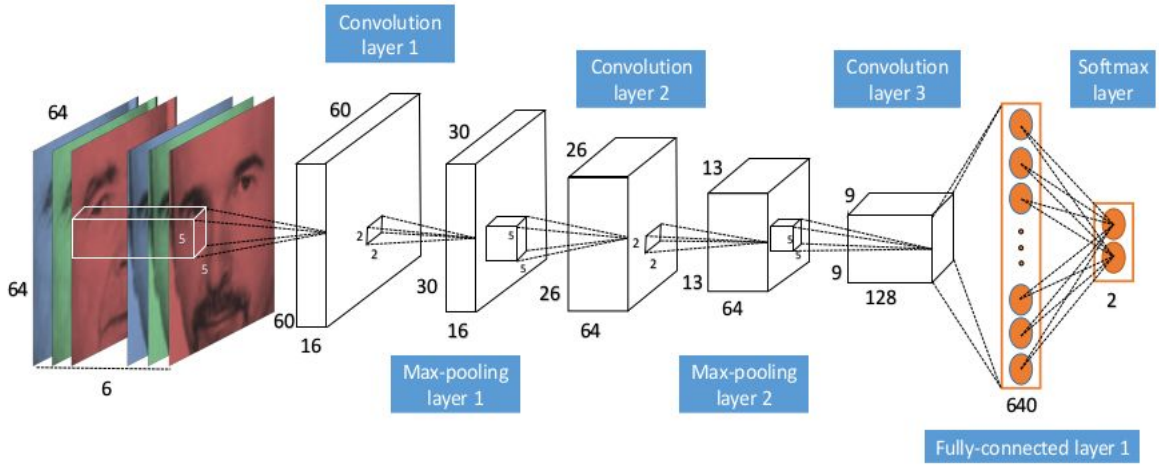


Fig 2. CNN architecture

Overview of the structure: The basic structure of CNN used in this work contains three convolutional layers, followed by a fully-connected layer and a soft-max layer. As shown in Figure 2, the input is a pair of 64 × 64 images with three channels (RGB). Following the input, the first convolutional layer is generated after convolving the input via 16 filters with a stride of 1. Each filter is

with the size 5 × 5 × 6. The second convolutional layer filters the input of the previous layer with 64 kernels of size 5 × 5 × 16. The third convolutional layer contains 128 kernels of the size 5×5×64. After the convolutional layers, a fully-connected layer projects the extracted features into a subspace with 640 neurons. Max-pooling layers follow the first and second convolutional layers. We adopt the ReLU function as the activation function of the convolution layers. Finally, this network is trained via a two-way soft-max classifier at the top layer.

4. Solution

4.1 Neighbourhood Repulsed Metric Learning

The intuition behind the Neighbourhood Repulsed Metric Learning technique is simple. We wish to minimize the distance (in the metric space) between faces for which the given relationship exists (father-daughter, father-son, mother-daughter or mother-son), while maximising the distance between pairs of faces not included in the relationship. A simple visualization for the problem and its solution is shown in Figure 3.

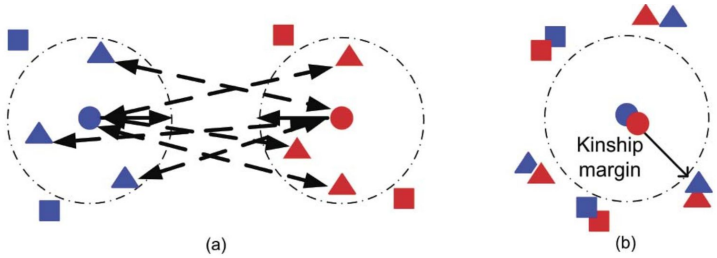


Fig 3. (a) The original face images with/without kinship relations in the high-dimensional feature space. The samples in the left denote face images of parents, and those in the right denote face images of children. (b) The expected distributions of face images in the learned metric space, where the similarity of the circle pair (with a kinship relation) is increased and those of the circle and triangle pairs are decreased.

Let $S = \{(x_i, y_i) \mid i = 1, 2, \ldots, N\}$ be the training set of N pairs of kinship images, where x_i and y_i, each of which is an m-dimensional column vector, represent the ith parent image and child image pair, respectively. The NRML technique aims to find a good distance metric d such that the distance between x_i and y_j $(i = j)$ is as small as possible, while simultaneously maximising the distance between x_i and y_j $(i \neq j)$.

We can define the distance metric d for some arbitrary x_i and y_j as follows,

$$d(x_i, y_j) = \sqrt{(x_i - y_j)^T A (x_i - y_j)}$$

where A is an $m \times m$ square matrix, and $1 \leq i, j \leq N$. Since d is a distance metric, $d(x_i, y_j)$ should have the properties of non-negativity and symmetry, and should obey the triangle inequality. Hence, A must be a symmetric and positive semidefinite square matrix.

Furthermore, due to these properties, A can be written as WW^T for some matrix W with dimensions $m \times l$, where $l \leq m$, which allows us to write d as follows,

$$d(x_i, y_j) = \sqrt{(x_i - y_j)^T A(x_i - y_j)} = \sqrt{(x_i - y_j)^T WW^T(x_i - y_j)} = \sqrt{(u_i - v_j)^T(u_i - v_j)}$$

Formally, this problem can be stated as an optimization problem for the function $J(A)$ as follows

$$\begin{aligned}
\max_A J(A) &= J_1(A) + J_2(A) - J_3(A) \\
&= \frac{1}{Nk}\sum_{i=1}^{N}\sum_{t_1=1}^{k} d^2(x_i, y_{it_1}) + \frac{1}{Nk}\sum_{i=1}^{N}\sum_{t_2=1}^{k} d^2(x_{it_2}, y_i) - \frac{1}{N}\sum_{i=1}^{N} d^2(x_i, y_i) \\
&= \frac{1}{Nk}\sum_{i=1}^{N}\sum_{t_1=1}^{k}(x_i - y_{it_1})^T A(x_i - y_{it_1}) \\
&\quad + \frac{1}{Nk}\sum_{i=1}^{N}\sum_{t_2=1}^{k}(x_{it_2} - y_i)^T A(x_{it_2} - y_i) \\
&\quad - \frac{1}{N}\sum_{i=1}^{N}(x_i - y_i)^T A(x_i - y_i),
\end{aligned}$$

where, J_1 represents the mean of distances of parents from their k nearest not-children

J_2 represents the mean of distances of children from their k nearest not-parents

J_3 represents the mean of distances of parents from their children

There exists no closed-form solution for such an optimization problem. Hence, the problem is solved in an iterative manner. However, this may seem like a chicken-and-egg scenario, since the distance metric d needs to be known for computing the k-nearest neighbors of x_i and y_i. The basic idea is to first use the euclidean metric to search the k-nearest neighbors of x_i and y_i, and solve d sequentially. In subsequent iterations, we use the previous value of A to compute d, and hence improve A. The algorithm terminates when A does not change significantly, limited by the number of iterations specified beforehand.

The method of calculating A, after some simplification, relies on solving the eigenvalue problem

$$(H_1 + H_2 - H_3)w = \lambda w$$

where H_1, H_2, H_3 are square matrices which depend only on the k-nearest neighbours. Of course, computing these neighbours requires A, and the old version of A is used as a sufficient approximation.

4.2 Support Vector Machines

Support vector machines (SVMs) are supervised learning models with associated learning algorithms that analyze data used for classification and regression analysis. For a given set of training examples, each labelled (marked as belonging to one or the other of two categories), an SVM training algorithm builds a model that assigns test samples (new examples) to one category or the other, which makes it a non-probabilistic binary linear classifier. An SVM model is a representation of the data as points in an *N*-dimensional space (usually in the hundreds or thousands), mapped so that the examples of the separate categories are divided by a hyperplane with their margin from the two categories being as wide as possible. This hyperplane is "supported" by the closest data points, or feature vectors, to it, hence the name. New examples are then mapped into that same space and predicted as belonging to a class depending on which side of the hyperplane they fall.

Since kinship verification is a binary classification problem and support vector machines have an exceptional track record with demonstrably excellent performance for a variety of similar tasks, we chose to use SVM for classification. The input for the SVC was the distance vector between the two input face image vectors x_i and y_i, and the class labels were 1 and 0, representing verification and contradiction. We used the radial basis function kernel in the experiments as it provided better accuracy than other kernels with relatively little parameter tuning required.

4.3 *k*-Nearest Neighbours

We chose the kNN classifier as an alternate to the SVM, which classifies data points according to some weighted average of their *k*-nearest neighbours. It works on a similar principle to the SVM, and in practice shows similar results to the SVM across a variety of problems.

Since the number of data points was low, we decided to consider only the 1st nearest neighbour for the classification. The input vectors and labels for the kNN classifier were the same as that of the SVM. The algorithm used for computing the nearest neighbours was a ball tree.

5. Implementation

5.1 Platform

The platform chosen for the implementation was Python (v3.7.1), due to its vast support libraries, like SciKit (learn- v0.20.1; image- v0.14.1) and NumPy (v1.15.4), along with extensive documentation on both. The CNN model was implemented using Keras (v2.2.4) Python library, which was built against a TensorFlow (v1.12.0) backend.

All experiments were conducted on a 2.5 GHz Intel i7 MacBook Pro, with 16GB of main memory.

5.2 Preprocessing

The datasets for the experiments - KinFaceW-I and KinFaceW-II were made available by the authors of the NRML paper. These were used for the experiments, as they were both labelled appropriately and cropped to the concerned facial area, excluding irrelevant attributes such as background and hair. This also prevented the need to apply convolution and pooling to extract relevant sections from the image, which is quite common for machine learning tasks related to image processing.

Since some images in the dataset were not available in color, all images were converted to grayscale to make the length of feature vectors equal for all images. If we had not done this, we would have had to process colored and grayscale images separately, resulting in downsizing of an already small dataset. The dimensions of each image were 64⁄64, hence the original image vector was 4096D. Histogram equalization was applied on each image separately, with 256 bins, to mitigate any illumination discrepancies amongst the images.

Feature descriptors were extracted for each image and used as the representative vectors for the images. Experiments were performed using vectors from the original image (4096D), local binary patterns (4096D), histogram of oriented gradients (2916D) and DAISY (7200D).

5.3 Training and Testing

Since the problem is a classification problem, we must have a good number of samples of all classes. In this case, the problem is a binary classification problem, i.e., positive and negative. These were given the labels 1 and 0 for convenience. The positive samples were already provided, in fact, that is what the entire dataset is. For getting the negative samples, we paired up a parent with a child that was not theirs, i.e., each negative sample is a pair of parent and child feature vectors (x_i , y_j), where $i \neq j$. Additionally, we ensured that in the negative samples, each parent and each child is present the same number of times. This is done to prevent biasing towards any particular sample vector.

SVM: For the SVM classifier, we generated an amount of negative samples which equal the number of positive samples, since SVMs are good at generalising and do not need a large number of negatives.
The gamma value was fixed at 0.00017 and the C parameter was set to 0.8 after hand tuning the SVM's performance.

kNN: For the kNN classifier, the number of negative samples generated was 4 times the number of positive samples. The nearest neighbour calculation should be near correct in all regions of the vector space. Since the positive samples lie in one cluster and the negative samples are spread out over the rest of the space, an equal number of samples would result in the kNN classifier producing incorrect labels for negative data points which lie away from the training negative data points a majority of the time. The generated ball tree had a leaf size of 8.

CNN: For the CNN classifier, the number of negative samples generated was 5 times the number of positive samples. This was done since the size of the original dataset was too small to perform deep learning, which generally requires thousands of data points.
The CNN is trained by back-propagation with logistic loss over the predicted scores using the soft-max function. To initialize weights, we use a Gaussian distribution with zero mean and a standard deviation of 0.01. The biases are initialized as zeros. In each iteration, we update all the weights after learning the mini-batch with the size of 128. In all layers, the momentum is set as 0.9 and the weight decay is set as 0.005. To expand the training set, we also randomly flip images and add grayscales during training (dataset augmentation).

6. Results

6.1 NRML

Classification Accuracy (Percent) of Different Feature Representation Methods
on Different Subsets of the **KinFaceW-I** Data Set using the SVM classifier

Method	Feature	F_S	F_D	M_S	M_D	Mean
NRML (paper)	LBP	62.7	60.2	54.4	61.4	59.7
	LE	66.1	59.1	58.9	68.0	63.0
	SIFT	65.5	59.0	55.5	55.4	58.8
	TPLBP	56.3	60.5	56.0	62.2	58.7
NRML (ours)	LBP	59.7	59.3	53.6	61.1	58.4
	HOG	58.2	57.9	54.4	58.4	57.2
	DAISY	61.3	63.0	56.8	62.7	61.0

Classification Accuracy (Percent) of Different Feature Representation Methods
on Different Subsets of the **KinFaceW-II** Data Set using the SVM classifier

Method	Feature	F_S	F_D	M_S	M_D	Mean
NRML (paper)	LBP	64.0	63.5	62.8	63.0	63.3
	LE	69.8	66.1	72.8	72.0	69.9
	SIFT	60.0	56.9	54.8	55.4	56.8
	TPLBP	64.4	60.6	60.8	62.9	62.2
NRML (ours)	LBP	62.3	61.4	59.8	62.0	61.4
	HOG	59.0	58.4	56.2	60.1	58.4
	DAISY	64.8	64.1	66.7	68.9	66.1

Classification Accuracy (Percent) of Different Methods on the **KinFaceW-I** Data Set
(average accuracy over different relationships FD, FS, MD, MS)

Method	Feature	SVM	KNN
NRML (paper)	LBP	59.7	63.4
	LE	63.0	64.3
	SIFT	58.8	63.5
	TPLBP	58.7	62.5
NRML (ours)	LBP	58.4	58.2
	HOG	57.2	57.2
	DAISY	61.0	60.4
(Non-NRML)	Raw image	52.4	51.5

Classification Accuracy (Percent) of Different Methods on the **KinFaceW-II** Data Set
(average accuracy over different relationships FD, FS, MD, MS)

Method	Feature	SVM	KNN
NRML (paper)	LBP	63.3	63.4
	LE	69.9	64.3
	SIFT	56.8	63.5
	TPLBP	62.2	62.5
NRML (report)	LBP	61.4	61.4
	HOG	58.4	58.3
	DAISY	66.1	65.7
(Non-NRML)	Raw image	54.5	52.5

6.2 Deep CNN

Classification accuracy (percentage) of CNN technique on different subsets of **KinfaceW-I** dataset

	F_S	**F_D**	**M_S**	**M_D**	**Mean**
CNN (paper)	75.7	70.8	73.4	79.4	74.8
CNN (ours)	73.4	67.2	70.5	76.5	71.9

Classification accuracy (percentage) of CNN technique on different subsets of **KinfaceW-II** dataset

	F_S	**F_D**	**M_S**	**M_D**	**Mean**
CNN (paper)	84.9	79.6	88.3	88.5	85.3
CNN (ours)	81.9	76.1	86.2	86.9	82.8

7. Conclusions

- Metric learning is a useful technique for automatically deriving a new metric based on the type similarity that we want to measure between two data points.
- The results in our experiments show that the larger the feature descriptor vector, the more accurate the classifier is (HOG (2916D) < LBP (4096D) < DAISY(7200D)), but this is at the additional cost penalty of increased computation time during the NRML optimization stage.
- However, it also shows that any type of feature extraction makes the classifier more accurate, even if it loses data (HOG (2916D) vs raw image (4096D)), which suggests that while NRML makes the classifier better, it cannot be used by itself to solve such tasks and requires appropriate preprocessing.
- Between the two classifier (SVC and kNN), there is very little difference in the performance. However, the kNN is significantly faster during the recognition (testing) stage due to the small number of samples provided during training. Note that the time complexity of recognition grows differently for both algorithms - for SVC it grows with the number of features (or dimensions), while for kNN it grows with number of samples (or data points).
- The discrepancy between our results and the paper results is down to imperfect tuning of the parameters for the classifiers.
- The CNN technique clearly outperformed the metric learning technique by quite a wide margin. Hence, for image processing tasks such as kinship verification, convolutional neural networks are still better.

References

i. J. Lu, X. Zhou, Y. Tan, Y. Shang and J. Zhou, "Neighborhood Repulsed Metric Learning for Kinship Verification," in *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 36, no. 2, pp. 331-345.

ii. Kaihao Zhang, Yongzhen Huang, Chunfeng Song, Hong Wu and Liang Wang. Kinship Verification with Deep Convolutional Neural Networks. In Xianghua Xie, Mark W. Jones, and Gary K. L. Tam, editors, Proceedings of the British Machine Vision Conference (BMVC), pages 148.1-148.12. BMVA Press, September 2015.

Appendix

A. Code

main.py

```
import preprocess as pp
import nrml
from classifier import ApnaSVM, ApnaKNN
import sys
import numpy as np

relationship = sys.argv[2]
featureSet = int(sys.argv[1])
k = 5
T = 5
eps = 0.00001

allImages = pp.getFlats(relationship, featureSet)

print("NRML")
W = nrml.trainNRML(allImages, k, T, eps)
print("\t...done")

knn = ApnaKNN(allImages, W)
svc = ApnaSVM(allImages, W)
knn.train()
svc.train()
knn.test()
svc.test()
```

preprocess.py

```
import os
from skimage import data, exposure, feature
from numpy import array as np_array
import numpy as np

N_BINS=256

def prepareImage(fileName):
     return exposure.equalize_hist(data.imread(fileName,
                                               as_gray=True))

def getPreparedImages(dir):
     fileNames = [x for x in os.listdir(dir) if
x.endswith('.jpg')]
     fileNames.sort()
     parent = []
     child = []

     for i in range(0,len(fileNames),2):
          parent.append(prepareImage(os.path.join(dir,
                                          fileNames[i])))
          child.append(prepareImage(os.path.join(dir,
                                          fileNames[i+1])))

     return np_array([np_array(parent), np_array(child)])

def flattenedImages(images):
     return np_array([i.flatten() for i in images])

def extractLBP(images):
     LBP_BLOCK_MAX_X = 4
     LBP_BLOCK_MAX_Y = 4
     l,b = images[0].shape
     BLOCK_L = l//LBP_BLOCK_MAX_X
     BLOCK_B = b//LBP_BLOCK_MAX_Y
     featureVecs = []
```

```
     for image in images:
          vector = []
          _x = 0
          for x in range(LBP_BLOCK_MAX_X):
               x_ = _x + BLOCK_L
               _y = 0
               for y in range(LBP_BLOCK_MAX_Y):
                    y_ = _y + BLOCK_B
                    newHist, _ =
exposure.histogram(image[_x:x_,_y:y_],nbins=N_BINS)
                    vector.extend(newHist)
                    _y = y_
               _x = x_
          featureVecs.append(vector)

     featureVecs = np_array(featureVecs)
     return featureVecs

def extractLE(images):
     return flattenedImages(images)

def extractSIFT(images):
     return flattenedImages(images)

def extractTPLBP(images):
     return flattenedImages(images)

def extractHOG(images):
     featureVecs = []

     for image in images:
          featureVecs.append(feature.hog(image,
                                         orientations=9,
                                         pixels_per_cell=(8,8),
                                         cells_per_block=(3,3),
                                         block_norm='L1',
                                         feature_vector=True))

     featureVecs = np_array(featureVecs)
```

```
     return featureVecs

def extractDAISY(images):
     featureVecs = []

     for image in images:
          featureVecs.append(feature.daisy(image,
                                           step=8,
                                           radius=8,
                                           rings=3).flatten())

     featureVecs = np_array(featureVecs)
     return featureVecs

def getFlats(dir, whichFeatures):
     allImages = getPreparedImages(dir)

     getFeatures = None
     if whichFeatures == 0:
          getFeatures = flattenedImages
     elif whichFeatures == 1:
          getFeatures = extractLBP
     elif whichFeatures == 2:
          getFeatures = extractLE
     elif whichFeatures == 3:
          getFeatures = extractSIFT
     elif whichFeatures == 4:
          getFeatures = extractTPLBP
     elif whichFeatures == 5:
          getFeatures = extractHOG
     elif whichFeatures == 6:
          getFeatures = extractDAISY

     allVectors = np_array([getFeatures(allImages[0]),
                            getFeatures(allImages[1])])

     print(allVectors.shape)
     return allVectors
```

```
def getCNNSamples(dir, mult=5):
     allVectors = getPreparedImages(dir)
     newShape = (allVectors.shape[0],
                 allVectors.shape[1],
                 allVectors.shape[2],1)
     allVectors.reshape(newShape)
     N = allVectors.shape[1]

     posPairs, negPairs = [], []
     for i in range(N):
          posPairs.append(np.concatenate((allVectors[0][i],
                         allVectors[1][i]),axis=3))

     OGRange = np_array(range(N),dtype=int)
     for i in range(mult):
          negIndex = np_array(range(N),dtype=int)
          np.random.shuffle(negIndex)
          while 0 in (negIndex-OGRange):
               np.random.shuffle(negIndex)

          for i in range(N):
               negPairs.append(np.concatenate((allVectors[0][i],
                         allVectors[1][negIndex[i]]),axis=3))

     return np_array(posPairs),np_array(negPairs)

def getRelationships(dir):
     fileNames = [x for x in os.listdir(dir) if
os.path.isdir(os.path.join(dir,x))]
     fileNames.sort()
     parents, childrens = [], []

     for relationship in fileNames:
          newParent, newChild =
getRelationshipImages(os.path.join(dir,relationship))
          parents.append(newParent)
          childrens.append(newChild)

     return np_array([np_array(parents), np_array(childrens)])
```

nrml.py

```
import os
import numpy as np
from sklearn.neighbors import NearestNeighbors

class FastMatMul():
     def __init__(self, allVectors):
          self.allVectors = allVectors
          N = allVectors.shape[1]
          m = allVectors.shape[2]

     def getMatMulEasy(self, i, j, allVectors):
          diff = allVectors[0][i] - allVectors[1][j]
          diffT = np.transpose(diff)
          return np.matmul(diff,diffT)

     def getMatMul(self, i, j, allVectors):
          return self.getMatMulEasy(i, j, allVectors)

def distanceMetric(x1, x2, WT):
     diff = x1 - x2
     WTdiff = np.matmul(WT,diff)
     return (np.dot(WTdiff,WTdiff))**0.5

def getkNNs(vectors, k, W):
     WT = np.transpose(W)
     metric = lambda x1,x2: distanceMetric(x1,x2,WT)
     neigh = NearestNeighbors(n_neighbors=k+1,
                              metric=metric,
                              n_jobs=-1)
     neigh.fit(vectors)
     _, kNNs = neigh.kneighbors(vectors)

     return kNNs[:,1:]

def getAllkNNs(allVectors, k, W):
     allkNNs = np.array([getkNNs(allVectors[0],k,W),
                         getkNNs(allVectors[1],k,W)])
```

```
     return allkNNs

def getNewH(allVectors, allkNNs, fMM):
     N = allVectors.shape[1]
     m = allVectors.shape[2]
     k = allkNNs.shape[1]

     H1 = np.zeros((m,m))
     for i in range(N):
          for t1 in allkNNs[0][i]:
               H1 += fMM.getMatMul(i, t1, allVectors)
     H1 /= (N*k)

     H2 = np.zeros((m,m))
     for i in range(N):
          for t2 in allkNNs[1][i]:
               H2 += fMM.getMatMul(t2, i, allVectors)
     H2 /= (N*k)

     H3 = np.zeros((m,m))
     for i in range(N):
          H3 += fMM.getMatMul(i, i, allVectors)
     H3 /= N

     return H1, H2, H3

def getNewW(H1, H2, H3, l):
     w, v = np.linalg.eig(H1+H2-H3)
     return v[:,(-w).argsort()][:,:l]
def trainNRML(allVectors, k, T, eps, l=None):
     N = allVectors.shape[1]
     m = allVectors.shape[2]
     fMM = FastMatMul(allVectors)
     if l == None:
          l = int(0.8*m)

     # print("init")
     allkNNs = getAllkNNs(allVectors, k, np.eye(m))
     Wr_1 = None
```

```
Wr = None

for r in range(T):
     print(r)
     Wr_1 = Wr
     H1, H2, H3 = getNewH(allVectors, allkNNs, fMM)
     Wr = getNewW(H1, H2, H3, l)

     if r > 1:
          if np.allclose(Wr,Wr_1,atol=eps):
               break

return Wr
```

classifier.py

```
from sklearn.svm import SVC
from sklearn.neighbors import NearestNeighbors
import numpy as np

class ApnaSVM():
     def __init__(self, samples, W):
          self.WT = np.transpose(W)
          self.N = samples.shape[1]*4//5
          self.samples = samples
          self.vectors = self.getVectors(samples, self.WT)
          self.posPairsTrain =
self.getPosPairs(self.vectors[:,:self.N])
          self.negPairsTrain =
self.getNegPairs(self.vectors[:,:self.N])
          self.posPairsTest =
self.getPosPairs(self.vectors[:,self.N:])
          self.negPairsTest =
self.getNegPairs(self.vectors[:,self.N:])

          self.svc = SVC(C=0.8,
                         gamma=0.000017,
                         kernel='rbf')

     def getVectors(self, samples, WT):
          N = samples.shape[1]
          vectors1 = []
          vectors2 = []
          for i in range(N):
               vectors1.append(np.real(np.matmul(WT,
                                         samples[0][i])))
               vectors2.append(np.real(np.matmul(WT,
                                         samples[1][i])))

          vectors = np.array([vectors1,vectors2])

          return vectors
```

```
    def getPosPairs(self, vectors):
        N = vectors.shape[1]
        posPairs = []
        for i in range(N):
            posPairs.append(np.array([vectors[0][i]-
                                vectors[1][i]]).flatten())

        posPairs = np.array(posPairs)
        return posPairs

    def getNegPairs(self, vectors):
        N = vectors.shape[1]
        OGRange = np.array(range(N),dtype=int)
        negIndex = np.array(range(N),dtype=int)

        np.random.shuffle(negIndex)
        while 0 in (negIndex-OGRange):
            np.random.shuffle(negIndex)

        negPairs = []
        for i in range(N):
            negPairs.append(np.array([vectors[0][i]-
                      vectors[1][negIndex[i]]]).flatten())

        negPairs = np.array(negPairs)
        return negPairs

    def train(self):
        trainX = np.concatenate((self.posPairsTrain,
                                 self.negPairsTrain))
        trainY =
np.concatenate((np.ones(self.posPairsTrain.shape[0]),
                np.zeros(self.negPairsTrain.shape[0])))
        self.svc.fit(trainX,trainY)

    def test(self):
        testX = np.concatenate((self.posPairsTest,
                                self.negPairsTest))
```

```
          testY =
np.concatenate((np.ones(self.posPairsTest.shape[0]),
                np.zeros(self.negPairsTest.shape[0])))
          score = self.svc.score(testX,testY)
          print("SVC accuracy = %.2f%"%(100*score))

class ApnaKNN():
     def __init__(self, samples, W):
          self.WT = np.transpose(W)
          self.N = samples.shape[1]*4//5
          self.samples = samples
          self.vectors = self.getVectors(samples, self.WT)
          self.posPairsTrain =
self.getPosPairs(self.vectors[:,:self.N])
          self.negPairsTrain =
self.getNegPairs(self.vectors[:,:self.N],mult=4)
          self.posPairsTest =
self.getPosPairs(self.vectors[:,self.N:])
          self.negPairsTest =
self.getNegPairs(self.vectors[:,self.N:],mult=1)
          self.neigh = NearestNeighbors(n_neighbors=1,
                                        algorithm='ball_tree',
                                        leaf_size=8,
                                        metric='euclidean',
                                        n_jobs=-1)

     def getVectors(self, samples, WT):
          N = samples.shape[1]
          Vectors1, vectors2 = [], []
          for i in range(N):
               vectors1.append(np.real(np.matmul(WT,
                                        samples[0][i])))
               vectors2.append(np.real(np.matmul(WT,
                                        samples[1][i])))

          vectors = np.array([vectors1,vectors2])

          return vectors
```

```
    def getPosPairs(self, vectors):
        N = vectors.shape[1]
        posPairs = []
        for i in range(N):
            posPairs.append(np.array([vectors[0][i]-
                            vectors[1][i]]).flatten())

        posPairs = np.array(posPairs)
        return posPairs

    def getNegPairs(self, vectors, mult):
        N = vectors.shape[1]
        OGRange = np.array(range(N),dtype=int)

        negPairs = []
        for i in range(mult):
            negIndex = np.array(range(N),dtype=int)

            np.random.shuffle(negIndex)
            while 0 in (negIndex-OGRange):
                np.random.shuffle(negIndex)

            for i in range(N):
                negPairs.append(np.array([vectors[0][i]-
                    vectors[1][negIndex[i]]]).flatten())

        negPairs = np.array(negPairs)
        return negPairs

    def train(self):
        trainX = np.concatenate((self.posPairsTrain,
                                 self.negPairsTrain))
        self.trainY =
np.concatenate((np.ones(self.posPairsTrain.shape[0]),

np.zeros(self.negPairsTrain.shape[0])))
        self.neigh.fit(trainX)

    def test(self):
```

```
            testX = np.concatenate((self.posPairsTest,
                                    self.negPairsTest))
            testY =
np.concatenate((np.ones(self.posPairsTest.shape[0]),

np.zeros(self.negPairsTest.shape[0])))
            _, kNNs = self.neigh.kneighbors(testX)
            results = self.trainY[kNNs[:,0]]
            wrongs = sum(np.abs(results-testY))
            score = (len(testX)-wrongs)*1.0/len(testX)
            print("kNN accuracy = %.2f%"%(100*score))
```

cnn.py

```
import os
import numpy as np
import cv2
import pickle

from imutils import paths
import skimage.data
import skimage.transform

from sklearn.preprocessing import LabelBinarizer
from sklearn.metrics import classification_report
from sklearn.model_selection import train_test_split

from keras.utils import to_categorical
from keras.models import Sequential
from keras.models import Model
from keras.optimizers import SGD
from keras.layers.convolutional import Conv2D
from keras.layers.convolutional import MaxPooling2D
from keras.layers.core import Activation
from keras.layers.core import Flatten
from keras.layers.core import Dropout
from keras.layers.core import Dense
from keras.layers import Concatenate
from keras.preprocessing.image import ImageDataGenerator
from keras.engine.input_layer import Input
from keras.utils.vis_utils import plot_model
from keras import backend as K

# set the matplotlib backend so figures can be saved in the
background
import matplotlib
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
matplotlib.use("Agg")

import preprocess as pp
```

```
import sys

data_dir = sys.argv[1]

posPairs, negPairs = pp.getCNNSamples(data_dir,1)

posY = np.ones((posPairs.shape[0],1),int)
negY = np.zeros((negPairs.shape[0],1),int)


data_X = np.concatenate((posPairs,negPairs),axis=0)
data_Y = np.concatenate((posY,negY),axis=0)

(trainX, testX, trainY, testY) = train_test_split(data_X,data_Y,
test_size=0.2, random_state=0)

trainY = to_categorical(trainY,num_classes=2,dtype='int32')
testY = to_categorical(testY,num_classes=2,dtype='int32')

#bulding the CNN
#input layer
ip = Input(shape=trainX[0].shape)

#first convolutional layer
c1 = Conv2D(filters=16, kernel_size=(5,5), strides=1,
padding="same", activation = 'relu')(ip)

#first pooling layer
p1 = MaxPooling2D(pool_size = (2,2), strides=2)(c1)

#second convolutional layer
c2 = Conv2D(filters=64, kernel_size=(3,3), strides=1,
padding="same",activation= 'relu')(p1)

#second pooling layer
p2 = MaxPooling2D(pool_size = (2,2), strides=2)(c2)

#third convolutional layer
```

```
c3 = Conv2D(filters=128, kernel_size=(5,5), strides=1,
padding="same",activation= 'relu')(p2)

#flattening outputs of c3
fl1 = Flatten()(c3)

#fully connected layer
fc1 = Dense(640)(fl1)

#fully connected (final layer - 2 outputs)
fc6 = Dense(2, activation = 'softmax')(fc1)

#model
model = Model(inputs = ip, outputs = fc6)

#model
model = Model(inputs = ip, outputs = fc6)

# initialize our initial learning rate and # of epochs to train
for
INIT_LR = 0.0000000001
MOMENTUM = 0.000000003
DECAY =  0.005

# compile the model using SGD as our optimizer and binary
# cross-entropy loss (logistic regression loss)
opt = SGD(lr = INIT_LR, momentum = MOMENTUM, decay = DECAY)
# opt = SGD(lr = INIT_LR, momentum = MOMENTUM, )
model.compile(loss="binary_crossentropy", optimizer=opt,
metrics=["accuracy"])

print(model.summary())
plot_model(model, show_shapes= True)

model_diagram=skimage.data.imread("model.png")
plt.figure(figsize=(10,10),dpi=512)
plt.imshow(model_diagram,aspect='auto')
plt.axis("off");
```

```
#train the model
EPOCHS = 500
BATCH_SIZE = 128

gen = ImageDataGenerator(horizontal_flip=True)
# H = model.fit_generator(gen.flow(trainX,trainY),
steps_per_epoch = len(trainX)//BATCH_SIZE, epochs=EPOCHS)
H = model.fit_generator(gen.flow(trainX,trainY),
validation_data=(testX,testY) , steps_per_epoch =
len(trainX)//BATCH_SIZE, epochs=EPOCHS)

# evaluate the network
predictions = model.predict(testX, batch_size=32)
print(classification_report(testY.argmax(axis=1),predictions.arg
max(axis=1), labels=[0,1]))
```