Human Action Recognition via Various Learning Methods

UNIVERSITY OF BATH

DEPARTMENT OF COMPUTER SCIENCE: CS50249
CANDIDATE NUMBER: 02151
10/05/2018

Human action recognition is a key topic in computer vision with applications in many aspects of video analysis and human machine interaction. This report focuses on performing basic human action recognition on the Microsoft Research Cambridge-12 gesture dataset using a variety of machine learning methods and critically reviewing each methods successes and failures.

1 Introduction

Human action recognition is a thriving topic within computer vision. Human action recognition is becoming much more prevalent with the increase in commercial VR/AR applications which demand human interaction. As well as thism there are a large number of potential applications in video analysis such as in surveillance, sports analysis, video retrieval, and human-machine interfaces.

An action is defined as a sequence of body motions which may, or may not, involve several body parts. The goal is to match the observation of an action (via video or other method) with pre-defined movements and assign an action label. Actions can be classified based on their complexity; gestures, actions, interactions and group activities [1]. In this report only gestures will be considered from the Microsoft Research Cambridge-12 gesture dataset [2].

The major technical components of this task include feature extraction, learning, and classification. The focus of this report is on the learning and classification, various learning methods will be tested with the dataset. Ranging in complexity, the learning methods trialled will include K-Nearest Neighbours, Support Vector Machines, Multi-Layer Perceptrons, Random Forests, and Convolutional Neural Networks. Each of these learning methods will be explained in full within section 2 and their results displayed and discussed in section 3. Of particular note; the Multi-Layer Perceptron and Random Forest methods will be implemented from scratch and have the algorithms explained in section 2.

Each learning algorithm has clear pros and cons for use with this dataset and it is the aim of this report to identify these and demonstrate the uses of each method as well as discuss the issues and successes arising. To summarises, this report will first explain the problem and dataset in more detail, as well as give an introduction to each of the learning algorithms and relevant code in section 2. Secondly, results will be presented and discussed in section 3. Finally, the report will be concluded in section 4.

2 Method

To effectively compare the results from each of the learning algorithms, the same pipeline will be used each time with the same training and test sets used for all algorithms so as to eliminate any bias due to training. The data set will be split into training and test sets of 4356 and 1263 exemplars respectively. Mean-squared errors between the predicted test set classifications and the desired response classifications will be computed for each algorithm for comparison of results between methods.

2.1 Gesture Dataset

The dataset contains sequences of human skeletal body part movements and was collected as part of a research project by Fothergill [2]. The full data set contains 6244 examples of 12 different gestures and was captured using a Microsoft Kinect. An example of a gesture within the dataset can be seen in figure 1. The gestures themselves fit into 12 classifiers:

1. Start Music/Raise Volume
2. Crouch
3. Navigate to next menu
4. Put on goggles
5. Wind up music
6. Shoot a pistol

7. Take a bow
8. Throw an object
9. Protest the music
10. Change weapon
11. Speed up tempo
12. Kick

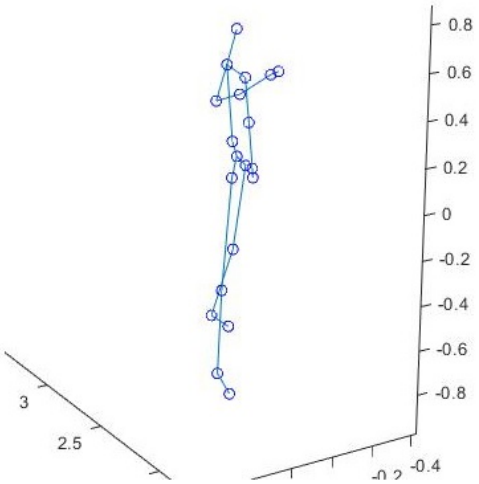

Figure 1: A still from an animation demonstrating the 'Throw' gesture. Each skeletal body part can be clearly seen.

The dataset is then processed into a format for use with all learning algorithms, this involves feature extraction which extracts 1830 features and results in a $n * m$ matrix where n is the number of exemplars and m is the number of features. This also generates a $n * 1$ vector of the corresponding classification labels.

2.2 Learning Algorithms

This section will discuss each of the classification learning algorithms that will be applied to the dataset, as well as the implementations performed for multi-layer perceptrons and random forests with the corresponding pseudo code.

2.2.1 K-Nearest Neighbours

The K-nearest neighbours algorithm (k-NN) is the most simple of the algorithms used on the dataset. In k-NN each exemplar is classified by the majority vote of a number of its nearest neighbours in feature space specified by the hyperparamater k [3]. In this instance, the inbuilt MATLAB $fitcknn$ function will be used from the 'Statistics and Machine Learning' toolbox. For this comparison the inbuilt optimization tools will be used as the main goal is to acquire the best model for prediction so as to compare to the other methods.

The main hyperparamaters of note are the k value and the distance measure. Typical measures that are often used for k-NN include Euclidean distance (for continuous variables) and Hamming distance (for discrete variables). Euclidean distance between two points is simply the length of the line connecting them:

$$||q - p|| = \sqrt{||p||^2 + ||q||^2 - 2p \cdot q} \qquad (1)$$

where q and p are the position vectors of the exemplars. For multiple dimensions in feature space this just adds the same vector calculation for each dimensional vector of p and q.

Alternatively the Hamming distance can be used to measure the distance between two discrete datasets such as letter strings. Here each different individual feature (or letter in the string example) accounts for an increase in distance. Many other distance measures exist and will be trialled as part of the optimization process, the optimal metric that is found will be discussed in section 3.

2.2.2 Support Vector Machines

The support vector machine (SVM) learning algorithm is the first of the more complex methods considered, although the basic premise of the method is rather straight forward. In SVM classification is performed by finding a hyper-plane that differentiates each class from each other [4]. These planes can exist between any pairs of dimensions in feature space and are used to separate and classify each exemplar. The support vectors are simply the co-ordinates of each individual observation that require classification.

Constructing the best hyper-plane between each class is done by maximizing the distances between the nearest data points and the hyper-plane itself, known as the margin and shown in figure 2. The hyper-planes do not have to be linear as kernel parameters can be used to define non-linear kernels for the separation. However, this dataset has a high number of features so it is likely that the data is linearly separable in the space.

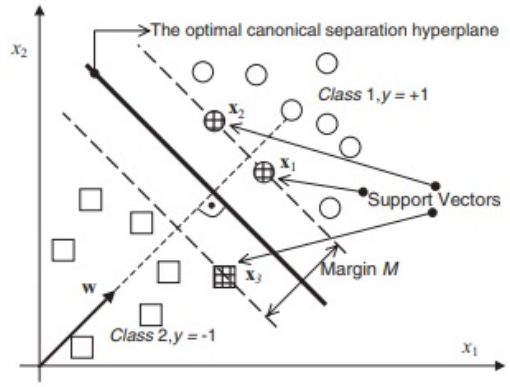

Figure 2: An example of a simple SVM where a single hyper-plane fully separates the data [4]. This simple schematic can be expanded into multiple dimensions with multiple hyper-planes separating the data and do not all have to be linear.

For use in this report, the inbuilt *fitcecoc* function from the 'Statistics and Machine Learning' toolbox will be used and optimized using inbuilt tools. The main hyperparameters of interest here are the number of binary learners used and their types. The optimized hyperparameters will be discussed in section 3.

2.2.3 Multi-Layer Perceptrons

A multi-layer perceptron (MLP) is an artificial neural network consisting of three or more layers of nodes; an input layer, one or more hidden layers, and an output layer [5]. Each node, excluding the input layer, is a neuron known as a perceptron. The MLP then uses backpropagation to train the network.

A single perceptron is a straightforward algorithm for mapping an input, x, to an output value, $f(x)$, as

$$f(x) = \begin{cases} 1 \text{ if } w \cdot x > 0 \\ 0 \text{ otherwise} \end{cases} \quad (2)$$

where w is a vector of weights, a dimension for each dimension of x. The output value of $f(x)$ is then used to classify x. In an MLP this process can be multiple levels deep with each neuron using a non-linear activation function. A typical activation function, and the one used in this implementation is a sigmoid:

$$Z = \frac{1}{1 + e^{-v}} \quad (3)$$

where Z is the output of the node and v is the weighted input. This output then propagates through to the next layer and the process repeats until reaching the output layer. Each node has an associated weight vector which is multiplied by the incoming matrix of features from the previous layer. It is this weight vector which is trained to produce an accurate model.

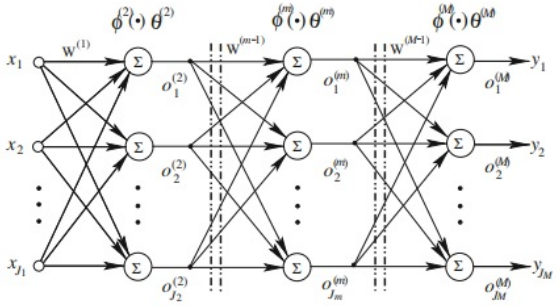

Figure 3: A schematic of a typical MLP. The input layer on the left has a node for each feature x_j, each node applies a sigmoid activation function labelled Σ, between each layer the weight vector w^m is applied, the final layer outputs y as a binary classifier [5].

The network learns via backpropagation. In the implementation carried out here, a gradient descent method is used to change the weight vector at each node. Firstly the errors at the output nodes are computed as

$$E = d_j(n) - y_j(n) \quad (4)$$

where d is the desired response and y is the actual response from the network for the j^{th} node of the n^{th} data point. Next, the gradient decent is carried out to minimize the whole networks error by adjusting the weights at each node, W, by

$$\Delta W = -\eta \frac{\delta E}{\delta W} \quad (5)$$

where η is the learning rate. This new weight for the output node is then used to pass back the original error to the previous node and the process repeats to the first layer. Multiple iterations of this process train the network to an acceptable level.

When optimising an MLP many hyperparameters can be considered, however here the only hyperparameters that will be experimented with are the number of hidden layers, the number of nodes in each hidden layer, the learning rate, and the number of iterations.

The pseudo code for the implementation performed for this report is as follows:

Result: Trains a MLP from a set of input exemplars X (features * n), response vector Y (binary classes * n), and vector of hidden layers h (Layers * 1)
for *each layer, L* **do**
  W_L = random values;
end
for *each Epoch* **do**
  for *each layer* **do**
    V = W_L*X_L : apply weights;
    $Z_L = \frac{1}{1+e^V}$: activation function;
  end
  Compute error, E, at output node;
  for *each layer* **do**
    $G = Z_{L+1}.*(1 - Z_{L+1})$: compute gradient;
    $\Delta W = \eta * Z_L * G * E$: compute ΔW;
    Adjust old weight by ΔW;
    $E = W_L * G.*E$: update error for propagation;
  end
end
Return final weights;

Algorithm 1: Train MLP

To then test the network, a test set can be passed forward through the network and the output is the predicted classification.

2.2.4 Random Forests

First created by Tin Kam Ho [6], but with the modern bagging method being created by Leo Breiman [7], random forests (RF) are an ensemble learning method that constructs multiple decision trees and output a result based on the average prediction of each of the individual trees.

The method implemented here creates a specified number of trees by splitting a the data a specified number of times to a designated maximum depth. The data is split based on maximising the information gain at each split. Where the information gain is defined as

$$IG = H_p - H_c \tag{6}$$

where H_p is the entropy of the parent and H_c is the weighted sum of the entropy of the children. The entropy is defined as

$$H = -\sum p_i log_2 p_i \tag{7}$$

where p_i are fractions that represent the percentage of each class present in the child. The trained forest is then used to make classification predictions on the test dataset by passing the test features through each tree in the forest and taking the average outcome as the result.

The pseudo code for the implementation carried out here can be seen below.

Result: Trains a random forest for classification based on a set of forest parameters, depth, number of trees and number of splits.
for *number of trees* **do**
  for *depth of tree* **do**
    Get relevant data at each node;
    Check not a leaf;
    for *number of splits* **do**
      Randomly split data;
      Evaluate information gain at split;
      if *best split* **then**
        Keep split;
      end
    end
    Pass relevant data down the new split;
  end
  Assign classes to leaf nodes and check distribution;
end
Return RF model;

Algorithm 2: Train forest

2.2.5 Convolutional Neural Networks

The final learning algorithm that will be considered is a convolutional neural network (CNN). A CNN is similar to the previously discussed MLP in that it is a feed-forward artificial neural network, but is much deeper in terms of hidden layers.

Although designed around a similar concept to MLPs, CNNs are biologically inspired with the main difference to traditional neural networks being that convolution is used in place of general matrix multiplication [8]. The following equation defines the convolution of functions f and g:

$$(f * g)(t) = \int_{\infty}^{\infty} f(\tau) g(t - \tau) d\tau \tag{8}$$

Simply, convolution is the integral of the product of functions after one is reversed and shifted by the target variable.

This difference in mathematics allows three distinct features of CNNs to be realised [8]. Firstly the neurons in a CNN are mapped in three dimensions and can either be locally or completely connected through the net. This moves on to the second key difference in that CNNs can be localised spatially enforcing only connections between adjacent neurons rather than the whole net. The final major

difference is that a CNN shares weights through the entire network, meaning each neuron in a given convolutional layer responds to the same feature within their local connectivity with the same weighting, this allows features to be detected regardless of spatial position.

A CNN will not be implemented in this report, instead MATLAB's 'Neural Network' toolbox will be used, specifically the $trainNetwork$ and $classify$ functions to create a network using a stochastic gradient descent solver which is then used to classify the test data. The data used in this method is of a slightly different format, instead of using extracted features this algorithm will be trained with time series sequences of the 60 x, y, z coordinates of the joint locations with each time step being a frame of the capture. The network will then solve each case as a sequence classification problem.

3 Results & Discusion

In this section the optimisation of each method will be discussed as well as the respective results of the test set classification. There will be discussion throughout in regards to each methods performance and the section will conclude with a discussion on the comparison of each of the methods performance on the dataset.

3.1 k-NN Results

The k-NN algorithm was optimised using the inbuilt MATLAB optimisation tools, which selected an optimal distance parameter and k value which minimised the expected objective function value.

The optimised hyperparameters were found to be the 'city block' distance metric with a k value of 1. The optimisation can be seen in figure 4. The city block metric is a special case of the Minkowski distance and is computed as:

$$d_{st} = \sum_{j=1}^{n} |x_{sj} - x_{tj}| \tag{9}$$

where d_{st} is the distance between data points x_s and x_t. It is likely that such a low value of k was found to be optimal due to the high dimensionality of the problem, nearest neighbours in some of the feature dimensions may be more important to the classification than in others. It is easy to justify that the best classifier will be the single closest training set point with a dimensionality this high (1830 features per data point). Despite the simplicity of the method, the algorithm achieved successful classification 42.36% of the time which is surprisingly successful given the complexity of the dataset.

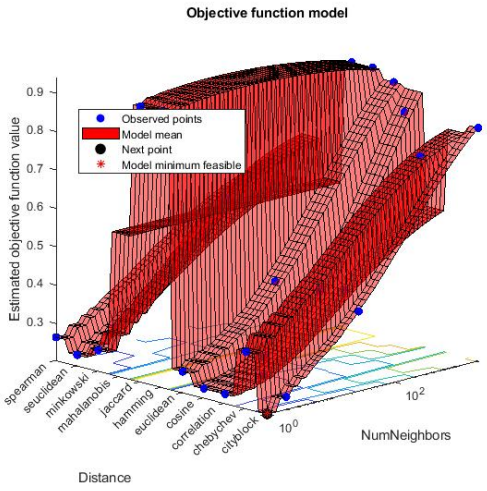

Figure 4: The optimisation of the k-NN model, it can be seen the 'cityblock' distance metric with a k value of 1 minimised the expected objective function value.

3.2 SVM Results

The SVM algorithm was again optimised using the inbuilt MATLAB Bayesian optimisation tools to select the optimal kernel scale, and box constraint for the algorithm. The optimisation processes evaluated 30 functions and concluded the objective function was minimised with a a box constraint of 993.03 and a kernel scale of 4.9103.

The SVM algorithm was fairly successful in predicting the classification of the test dataset, achieving 82.50% correct classification. The model dealt with the high dimensionality far more successfully than the k-NN algorithm due to using multiple linear hyper-planes to divide the training dataset across many dimensions. Perhaps using non-linear kernels for the generation of non-linear hyper-planes would further increase the accuracy of classification but this is beyond the scope of this report.

3.3 MLP Results

Optimisation of the MLP model was carried out manually due to the manual implementation of the method. Due to the time constraints with the computation and manual adjustment of the hyperparameters only a rough optimisation was carried out. The results of the various tests can be seen in the below table.

h	η	Epoch	% correct
(30, 20, 12)	0.0002	5000	39.67
(50, 40, 30, 24, 12)	0.0002	5000	39.75
(48, 24, 12)	0.0002	5000	57.72
(48, 24, 12)	0.0001	5000	36.74
(48, 24, 12)	0.0003	5000	38.95
(50, 24, 12)	0.0002	5000	28.27
(96, 48, 24, 12)	0.0002	5000	55.50
(48, 24, 12)	0.0002	8000	57.96

The best layer structure found was to have three hidden layers with 48, 24, and 12 nodes in each layer. A learning rate of 0.0002 was found to be optimal and with 8000 epochs a correct classification was achieved on 57.96% of the test dataset. More Epochs would most likely have continued to improve the network however this would be at the expense of computational time. Overall the MLP model was able to classify the data but operated very slowly and is particularly hard to optimise the layer structure.

3.4 RF Results

Similarly to MLP, the RF optimisation was implemented manually. Again, due to time constraints, full optimisation will take too long so only a rough optimisation was carried out. The results of which can be seen below.

Depth	Trees	Splits	% correct
5	300	5	39.03
5	500	5	37.77
10	300	5	46.08
15	300	5	48.06
12	300	5	47.51
12	500	7	49.80

The best classification on the test dataset was achieved with a tree depth of 12, 7, splits and 500 trees trained. These hyperparameters led to a 49.80% successful classification. Results could have been improved further with more trees and more optimisation of depth and splits, however, computing time was long so this result was accepted.

3.5 CNN Results

The creation of the CNN with the inbuilt MATLAB function took a large amount of time on CPU, over 2 hours. Lacking the hardware for GPU computation it was decided that adjusting parameters and training multiple, larger, CNNs for optimisation purposes is infeasible. The net created had 100 hidden layers and was trained over 100 epochs, it successfully classified only 13.22% of the test dataset which was far less than anticipated.

3.6 Discussion

The best classification algorithm on this dataset was found to be the SVM algorithm which successfully classified 82.50% of the test dataset. None of the other algorithms performed to this level with the next best being the MLP which predicted only 57.96%. This is most likely due to the high dimensionality of the dataset, with 1830 features the algorithms that struggle with high dimensionality such as k-NN proved to be difficult to optimise.

The k-NN, MLP, and RF algorithms all scored similarly on the test dataset scoring 42.36%, 57.96%, and 49.80% respectively. These algorithms all function differently and deal with different aspects of data better and worse than each other. This further supports the argument that the high dimensionality of the dataset renders the models insufficient for more accurate classification.

It is worth considering the fact that due to the manual optimisation of both the MLP and RF algorithms, they could be optimised to a much greater level. This would most likely increase the classification accuracy. Key hyperparameters that require further optimisation include the hidden layer structure of the MLP and increasing the number of epochs, both of which at the expense of computation time. Similarly with RF more trees could be trained and the depth and splits further optimised.

The performance of the CNN was most surprising, it was anticipated that the network would achieve a greater classification score than the 13.22% it did achieve. 13.22% is only slightly better than random selection which would be a score of 8.33%. It is unknown as to why this was the case, it is also not feasible to optimise and continuously retrain a CNN due to the very long computation time required. It is possible that a larger training set would be required to effectively train the CNN.

As previously identified, the SVM algorithm was the most successful in generating a model for classifying the test dataset. The model effectively deals with the high dimensionality by using multiple linear hyper-planes to divide the training dataset across the many dimensions. The high success of the SVM algorithm suggests that most features are linearly separable despite the large dimensionality and potential complex feature relationships.

4 Conclusions

To conclude, multiple machine learning algorithms were tested to create an effective action pose estimator for a Microsoft Kinect gesture dataset. It was found that a support vector machine algorithm created the most accurate model which successfully classified 82.50% of the test dataset. K-Nearest Neighbours, multilayer perceptron, and random forest algorithms were also tested and all scored between 40 and 60% accuracy, with the reduction in accuracy most likely due to the high dimensionality of the dataset. A convolutional neural network was also trained and tested but was substantially less accurate with only 13.22% successful classification. It is predicted this is due to the small training set and the small amount of hidden layers and epochs used to train the network, unfortunately due to the large computation time on a CPU, and a lack of appropriate GPU for computation, a more complex network could not be trained.

References

[1] J.K. Aggarwal and M.S. Ryoo. Human activity analysis: A review. *ACM Comput. Surv.*, 43(3):16:1–16:43, April 2011.

[2] Simon Fothergill, Helena Mentis, Pushmeet Kohli, and Sebastian Nowozin. Instructing people for training gestural interactive systems. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '12, pages 1737–1746, New York, NY, USA, 2012. ACM.

[3] L. Devroye G. Biau. *Lectures on the Nearest Neighbor Method*. Springer, Cham, 2015.

[4] V. Kecman. *Support Vector Machines – An Introduction*, pages 1–47. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005.

[5] Ke-Lin Du and M. N. S. Swamy. *Multilayer Perceptrons: Architecture and Error Backpropagation*, pages 83–126. Springer London, London, 2014.

[6] Tin Kam Ho. Random decision forests. In *Proceedings of 3rd International Conference on Document Analysis and Recognition*, volume 1, pages 278–282 vol.1, Aug 1995.

[7] Leo Breiman. Random forests. *Machine Learning*, 45(1):5–32, Oct 2001.

[8] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.