DD143X Degree Project in Computer Science, First Level

Classification of Electroencephalographic Signals for Brain-Computer Interface

Fredrick Chahine

900505-2098

fchahine@kth.se

Supervisor: Pawel Herman

Abstract

Brain-computer interface is a promising research area that has the potential to aid impaired individuals in their daily lives. There are several different methods for capturing brain signals, both invasive and noninvasive. A popular noninvasive technique is electroencephalography (EEG). It is of great interest to be able to interpret EEG signals accurately so that a machine can carry out correct instructions. This paper looks at different machine learning techniques, both linear and nonlinear, in an attempt to classify EEG signals. It is found that support vector machines provide more satisfactory results than neural networks.

Contents

I. Introduction

Brain-computer interfaces promise the ability to control tools and perform tasks without lifting an arm. Such interfaces can even be used by individuals who are otherwise unable to move their limbs (Wolpaw, 2009). For reasons such as this, the brain-computer interface is a research area of great interest to scientists. Using various techniques, it is possible to measure electric activity in the brain. The signals may then be processed and features of particular interest elicited. Afterwards, these features may be interpreted as instructions to a connected device with the help of a translation algorithm.

There are various methods of measuring brain activity. These methods differ in means of execution as well as accuracy. Some methods are invasive, requiring the implanting electrodes into the brain. These methods often provide accurate signal data, but the physical presence of a foreign object inside the brain can lead to scar tissue and is thus usually unfavorable (Wolpaw, 2009). Invasive procedures are therefore rarely performed on humans. Other methods, such as electroencephalography (EEG), are noninvasive. EEG measures the activity on the scalp. It is applied on the outside of the head without causing any physical damage. As a result of its externality, however, EEG often provides less accurate data than do invasive methods. Consequently, being able to process such data accurately is of great value. EEG provides the opportunity to capture motor imagery, allowing a subject to mentally convey a movement, which can then be interpreted and replicated by a machine.

With the promising applications of EEG, it is of great interest to be able to classify EEG signals accurately. In this report, we shall examine how EEG signals may be classified using machine learning techniques.

II. Methodology

This paper will examine linear and nonlinear methods for classifying EEG data. We begin with the single-layer perceptron, move on to the multilayer perceptron, and conclude with the support vector machine. General opinion regarding classification of EEG signals favors simplicity, advocating the use of linear methods whenever

possible (Müller, Anderson, & Birch, 2003). However, more complicated, nonlinear methods may sometimes provide better results.

The experiment consists of classifying EEG signals collected from one subject imagining moving his left or right arm. There is no physical movement involved. There are 160 trials, 80 corresponding to imagining moving the left arm and the other 80 to imagining moving the right arm. The signals have been measured over a duration of nine seconds, but only the five most relevant seconds have been extracted, specifically the last five seconds. Signals were recorded for the mu and beta frequencies, with two channels for each. Hence the input data consists of four dimensions; it is processed using MATLAB. Labels, corresponding to the correct classifications, are given, making this a supervised learning experiment. The goal is to classify the data into two classes based on which arm the subject imagines moving.

We shall begin by attempting to classify the data using a linear model, namely a single layer perceptron. Then we shall proceed to use a more complicated, nonlinear model, the multilayer perceptron. Finally, we shall try to classify the data using support vector machines.

The first step is the processing of recorded data. The data is divided into two classes: data corresponding to the subject imagining moving his right arm, and data corresponding to imagining moving his left arm. We have 80 trials for each. To teach the network that there are two alternatives, it is necessary to combine the two classes of data in a fruitful way. The classes cannot be processed separately, otherwise there is no need for discrimination between the two and hence no need for learning.

The data is divided into 71 time windows, here referred to as time slices. The time slices cover the thought process as the subject imagines moving a particular arm. Hence we expect that at corresponding times in the thought process the subject is performing analogous imagination be it related to the left arm or the right. Therefore, it seems appropriate to process the data one time slice at a time. Thus we have data for 160 trials for the first time slice, then data from the 160 trials for the second time slice, and so on. At each time slice, each input is four-dimensional, corresponding to two frequencies and two channels.

III. Linear Classification: the Perceptron

A first attempt at classifying the EEG signals may be to use a (single-layer) perceptron. A perceptron is an artificial neural network that is a linear model for supervised learning. This suits our present situation given that we have the correct label for each 4-tuple of input. The network works by weighting each input parameter and summing these values. If the sum is greater than the threshold value for a neuron, the neuron fires. In this paper we will represent a threshold value using a bias input. Given target labels, the network classifies input by updating these weights based on the difference between the target value and the achieved value. A perceptron has been implemented in MATLAB and used to classify the EEG data.

We start by organizing the data into a pattern matrix. In this experiment, the pattern matrix is a 4 × 160 matrix where the rows correspond to the different dimensions of the input data and the columns correspond to the different trials. It is a good idea to make use of batch learning, so that the data for the 160 trials can be processed all at once. The values of the weights of the neural network will result in it classifying some input correctly and other input erroneously. The error, defined as the difference between the target output and the actual output, will be used to update the weights of each neuron the neural network in an attempt to more closely replicate the desired output. The error for each trial for a particular time slice contributes to the value that is added to the current weight to produce the new weight. The algorithm is run for 40 rounds or epochs. At the end of each epoch, the weights are updated.

The delta rule captures the weight update for a weight $w_{j,i}$ connecting node *j* to node *i*. This is given by the equation below, where x_i is the input to node *i*, t_j is the desired target output at node *j* and η is the learning rate, a constant that affects the rate of weight update. In this experiment, we choose η to be 0.001.

$$\Delta w_{j,i} = \eta x_i \sum_k \left(t_j - w_{j,k} x_k \right)$$

Rather than getting bogged down in calculating each weight update individually, we make use of matrix multiplication to perform batch learning. We then get the

following formula, where all the individual elements are placed in matrices. The summation follows from the definition of matrix multiplication.

$$\Delta W = \eta\left(T - WX\right)X^T$$

The weight matrix W initially contains small numbers that are then updated as the algorithm proceeds. The matrix T contains the target outputs, that is, an indication of the actual class corresponding to each 4-tuple of input. To simplify our calculations, we can choose a symmetric representation of the two classes of data, designating motor imagery for one arm by a target output of -1 and that for the other arm by a target output of 1. As a result, thresholding can then be done at zero. This means that any 4-tuple of input with an output greater than zero will be classified as motor imagery for one arm and any 4-tuple with an output less than zero will be deemed to correspond to motor imagery for the other arm.

To incorporate the notion of a threshold, we add a bias input to our data. This is represented by a row consisting entirely of 1s introduced as the last row of the pattern matrix. An extra column is appended to the weight matrix, corresponding to the weight for the bias input. This weight is then equal to the opposite of the threshold value for neural activation, and will be updated along with the other weights under the algorithm's execution.

The figure below shows the numbers of wrongly classified inputs per trial. We see that the best values are close to 20 wrong classifications.

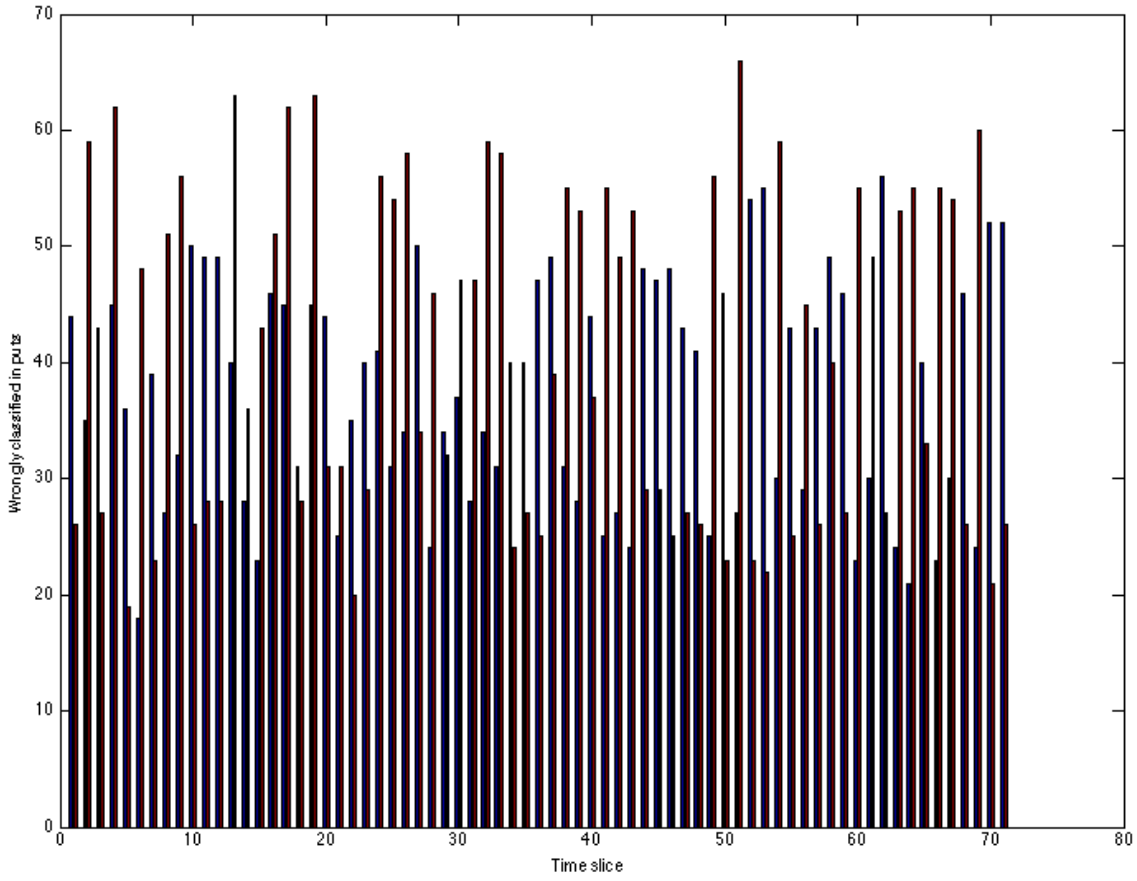

Figure 1 Wrongly classified inputs per time slice for perceptron

The table on the next page shows an excerpt from the data produced when the algorithm is run. There are 71 time slices in total – too redundant to reproduce here in their entirety. The headings Wrong 1 and Wrong 2 indicate the number of misclassifications of one type as another, i.e. classifying an imagined right arm movement as that of a left arm or vice versa. Since the weight matrix begins with small random weights, the values differ somewhat between executions.

Table 1 Sample of wrongly classified inputs per time slice for perceptron

Time Slice	Wrong 1	Wrong 2
1	49	26
2	35	56
3	43	21
10	47	35
25	43	26
48	46	37
71	26	61

Clearly there are a large number of errors considering that each class contains only 80 specimens. Since there are only two classes of data involved, if we were to go about guessing the class of an input instance we would be right 50% of the time. This corresponds to an expected number of 40 errors of each type in the above table, suggesting that the results above are poor. Perhaps the data itself is not linearly separable.

IV. The Multilayer Perceptron

The multilayer perceptron requires a more complicated approach but holds the promise of being able to classify linearly non-separable data. The network we will use consists of one hidden layer in addition to the output layer of neurons. The inputs to the hidden layer are known; these are namely the inputs to the network. The outputs of the hidden layer are unknown. Furthermore, the number of neurons in the hidden layer may be varied. The outputs of the hidden layer are the inputs to the output layer. Consequently, the inputs to this latter layer are unknown. The outputs of the layer are the network outputs and are therefore transparent.

The algorithm consists of three phases: a forward phase, a backward phase, and a phase in which the weights are updated (Marsland, 2009). The forward phase consists of pushing the input data through the network and producing output based on the input and the current values of the weights in the network. An error is then produced at the output end of the network as the actual output differs from the target

output. This error is then fed backwards through the network. In the third and final phase, this error is used to update the weights, whose new values are used during the next forward iteration.

To let the multilayer perceptron learn something new, we shall use a non-linear activation function. We choose a function whose derivative can be computed easily. Our function is

$$\varphi(x) = \frac{2}{1+e^{-x}} - 1$$

This function has a derivative that can be neatly expressed in terms of itself:

$$\varphi'(x) = \frac{\left[1+\varphi(x)\right]\left[1-\varphi(x)\right]}{2}$$

In MATLAB, where we need to compute the value of $\varphi(x)$ for different values of x, we can easily compute the value of $\varphi'(x)$.

The forward pass consists of propagating the four-dimensional input through the network. The activation of a node h_j in the hidden layer is equal to the sum of the weighted inputs $w_{j,i}x_i$. Hence we get

$$h_j = \sum_i w_{j,i}x_i$$

The transfer function is then applied to this value to produce the input to the next layer, namely the output layer. Thus we have the output o_k of node *k* equal to

$$o_k = \sum_j v_{k,j}\varphi(h_j).$$

It is convenient to express these computations using matrices. In addition we shall apply the transfer function to both the output from the hidden layer and the final output. We then get

$$H = \varphi(WX)$$

$$O = \varphi(VH)$$

We want the outer end of the network to produce values that mimic the target value, but in order to teach the network this we need to propagate the error backwards through the network. The output error δ_k^o for an output node *k* is the difference between the target value and the actual value. In other words, we have

$$\delta_k^o = t_k - \varphi(o_k)$$

This difference is multiplied by the derivative of the transfer function for the output value to result in gradient descent. The error in the previous layer can then be calculated as

$$\delta_j^h = \left(v_{k,j}\delta_k^o\right)\varphi(h_j)$$

We can then update the weights by multiplying the respective errors and inputs for each layer, summing them and multiplying by the learning rate:

$$\Delta W = \eta\delta^h X^T$$
$$\Delta V = \eta\delta^o H^T$$

We perform the three passes forty times just like for the single layer perceptron.

The results of the multilayer perceptron are somewhat better. The multilayer perceptron classifies about 65 out of 160 trials wrongly, meaning it has a success rate of about 59%. This is however not very far from the random chance of 50%.

V. Support Vector Machines

We can also attempt to classify the data using a support vector machine. A support vector machine seeks to find the best discriminant linear function by maximizing the distance between the graph of the function and the closest linear point. In addition, a support vector machine can model nonlinear functions using the kernel trick. MATLAB includes an implementation of a support vector machine that can be trained using input data and labels and then used to classify other data.

First, we create an SVMStruct object by supplying training data and labels using the following command

```
SVMStruct = svmtrain(training_data,labels)
```

The training data and labels contain the instances as rows, so the matrices used previously must be transposed. Then we use the SVMStruct object to investigate how many of the training data points were classified incorrectly. The command

```
Output = svmclassify(SVMStruct,Sample)
```

produces a column vector of 1s and -1s, depending on the classification of the data by the trained support vector machine. The table below shows the resulting errors when using different kernel functions with the support vector machine for the first time slice:

Table 2 Number of incorrect classifications with support vector machine

Kernel Function	Wrong 1	Wrong 2
Linear	21	30
Quadratic	14	39
Multilayer perceptron	37	44
Radial basis function	16	32
Polynomial of degree 3	16	24
Polynomial of degree 6	7	5

The best nominal result is achieved with a polynomial of degree 6 as the kernel function. This trend is maintained across the other time slices. For a polynomial of degree 7, the algorithm does not converge. Also interesting is that the multilayer perceptron kernel produces relatively poor results.

VI. Comparing the Results

The linear approach using the single layer perceptron proves to be inadequate to classify the EEG data. Similarly, the support vector machine with a linear kernel function performs poorly. This suggests that the data is not linearly separable.

The experimental data suggests that the best classifier is a support vector machine with a polynomial of degree 6 as a kernel function. This method surpasses by far any

other technique in this report. However, this may be due to overfitting, so that the method adapts especially well to the training data.

VII. Conclusion

The single and multilayer perceptrons have proven difficult to use when classifying electroencephalographic signals. The support vector machine implementation available in MATLAB proved to be more capable based on the same data. The inability of an advanced technique such as the multilayer perceptron to classify the data accurately suggests that the data has some noise involved. At the same time, the ability of the support vector machine implementation to accurately classify the data makes it seem that the data is highly intelligible. Finally, it has been suggested that improved data may be acquired by training individuals to shape brain signals (Blankertz, Curio, & Müller, n.d.).

References

Blankertz, B., Curio, G., & Müller, K. R. (n.d.). *Classifying single trial EEG: Towards brain computer interfacing.* Retrieved April 12, 2013, from http://books.nips.cc/papers/files/nips14/NS17.pdf

Marsland, S. (2009). *Machine learning: An algorithmic perspective*. Boca Raton, FL, USA: Chapman & Hall.

Müller, K. R., Anderson C. W., & Birch, G. E. (2003). *Linear and nonlinear methods for brain-computer interfaces*.

Wolpaw, J. R. (2009). *Brain-computer interface*. New York State Department of Health and State University of New York, Albany, NY, USA.

Appendix: MATLAB Code

```
function [ wrongs ] = delta(rate, patterns, T)

    epoch = 40;

    [~, colsX] = size(patterns);
    X = [patterns; ones(1, colsX)];

    [rowsX, ~] = size(X);
    [rowsT, ~] = size(T);
    W = randn(rowsT,rowsX);

    axis([-2, 2, -2, 2], 'square');

    [insize, ndata] = size(patterns);

    for i=1:epoch
        weight_change = -rate * (W*X - T) * X';
        W = W + weight_change;


    end
    final_weights = W;
    error = T-W*X;


    [~, colsT] = size(T);
    prod = W*X;
    wrong1 = 0;
    wrong2 = 0;
    for idx = 1:colsT

        if T(idx) > 0 && prod(idx) < 0
            wrong1 = wrong1 + 1;
        elseif T(idx) < 0 && prod(idx) > 0
            wrong2 = wrong2 + 1;
        end
    end

    disp('wrong1');
    disp(wrong1);
    disp('wrong2');
    disp(wrong2);


    wrongs = [wrong1 wrong2];

end


function [ final_weights ] = gen_delta(eta, patterns, T)

    epoch = 80;
    hidden = 40;
    scale = 0.9;

    [rowsPatterns, colsPatterns] = size(patterns);
```

```
    [rowsT, ~] = size(T);

    initial_weight_max = 0.1;

    w = initial_weight_max * randn(hidden,rowsPatterns + 1); %
columns in w == rows in patterns + bias row
    v = initial_weight_max * randn(rowsT,hidden+1);

    dw = zeros(size(w));
    dv = zeros(size(v));

    for i=1:epoch



        % The forward pass

        % w corresponds to the weights at the hidden layer
        hin = w * [patterns ; ones(1,colsPatterns)];
        hout = [2 ./ (1 + exp(-hin)) - 1; ones(1, colsPatterns)];

        % v corresponds to the weights at the output layer
        oin = v * hout;
        out = 2 ./ (1 + exp(-oin)) - 1;


        % The backward pass
        delta_o = (out - T) .* ((1 + out) .* (1 - out)) * 0.5;
        delta_h = (v' * delta_o) .* ((1 + hout) .* (1 - hout)) * 0.5;
        delta_h = delta_h(1:hidden, :); % removes row added for bias
term


        patterns = [patterns ; ones(1,colsPatterns)];
        %dw = -eta .* (delta_h * pat');
        %dv = -eta .* (delta_o * hout');

        % w has an extra column for the bias
        % Weight update
        dw = (dw .* scale) - (delta_h * patterns') .* (1 - scale);
        dv = (dv .* scale) - (delta_o * hout') .* (1 - scale);
        w = w + dw .* eta;
        v = v + dv .* eta;
        disp(i);




    end


end
```

```
for time_slice=1:71



    disp('time slice');
    disp(time_slice);
    patterns = [];
    for i = 1:160
        column_of_four = trial_features{i}(:,time_slice); % take out
the first time point from each trial (it contains four rows)
        patterns = [patterns column_of_four];
    end

    targets = [ones(1,80), -1 .*ones(1,80)];

    permute = randperm(160);
    patterns = patterns(:, permute);
    targets = targets(:,permute);

    gen_delta(0.001,patterns,targets);


end


for time_slice=1:71

    patterns = [];
    for i = 1:160
        column_of_four = trial_features{i}(:,time_slice); % take out
the first time point from each trial (it contains four rows)
        patterns = [patterns column_of_four];
    end

    targets = [ones(1,80), -1 .*ones(1,80)];

    training = patterns';
    group = targets';
    struct = svmtrain(training,group,'showplot',true,
'kernel_function','polynomial','polyorder',6);

    grouped = svmclassify(struct,training);

    wrong1 = 0;
    wrong2 = 0;
    [rows,~] = size(group);
    for idx = 1:rows
            if grouped(idx) > 0 && group(idx) < 0
                wrong1 = wrong1 + 1;
            elseif grouped(idx) < 0 && group(idx) > 0
                wrong2 = wrong2 + 1;
            end
    end

    disp('wrong1');
    disp(wrong1);
    disp('wrong2');
    disp(wrong2);
end
```