Using A Multilayer Perceptron To Classify Facial Expressions

Jacob Gavin s4658981
Laura Pijnacker s4332199

9th of June 2016

1 Abstract

Neural networks are getting more attention for everyday. They are starting to be implemented and used in a lot of different ways. There are many different kinds of Neural networks and they serve different purposes. In this report we are trying to classify happy and sad facial expressions with the help of a Multilayer Perceptron (MLP). The MLP consist of one input layer, one hidden layer, and one output layer. It uses backpropagation to update the weights based on error. The data consist of 9000 faces where 8000 were used for training and 1000 for testing. The training phase ran for a 1000 iterations with a learning rate of 0.04 which resulted in a mean squared-error of 304 with an accuracy of 69.4% correct predictions for our 1000 test cases.

2 Introduction

In this report we will try to predict whether input images were happy or sad faces. In order to make this prediction will train a Neural Network to classify an image as either happy(1) or sad(0). The Multilayer Perceptron (MLP) is a feedforward network that consist of three layers. The input layer consist of 2304 units (one unit per pixel), the hidden layer consist of 25 hidden units and the output layer which consist of 1 unit. The reason for only having one output unit is that we only have two different classes, happy(1) and sad(0). Once we have propagated forward in the network we need to perform a backpropagation to update the weights based on the error in the expected result. This is also the reason why we choose to use a MLP. Since it's a supervised learning model, it allows us to change change the weights based on the mean squared error for the whole layer, which we anticipate will be very useful because we know the target values.

Figure 1: Input image of a happy face

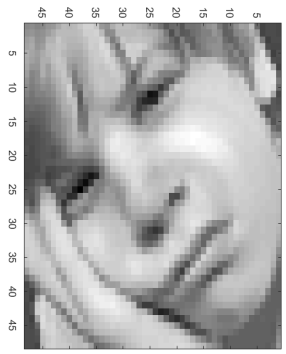

Figure 2: Input image of a sad face

3 Method

The Multilayer perceptron with two layers (one hidden) can be represented as

$$y = f(W^2 f(W^1 x))$$

where f is the activation function applied to all input elements, in this case the sigmoid function. $s = 1/(1 + e^{-a})$.

The first thing that happens in the MLP is the forward propagation to compute the output of the second layer(hidden), which is the activation function applied to the first weights times the input $h = f(W^1 x)$.

Further on we compute the output of the second layer by passing it through the activation function

$$o = f(W^2 h)$$

The sigmoid function also returns the gradient of the two layers output. We choose to use the sigmoid function as activation function for both layers even though the MLP allows for using a linear function for the second layer. Once we have computed the output for the output layer we can compute the output of the error function like such: Let $w = (vec(W^1); vec(W^2))$. The squared loss error function is given by

$$E(w) = \sum_n E^2(w)$$

where

$$E^n(w) = \frac{1}{2} \sum_n (y_k^n - t_k^n)^2$$

is the squared loss for a single data point.

The next step is to perform the backpropagation where we begin by computing the errors of the first and second layer. To get the error (delta) of the second layer (output layer) we use the chain rule

$$\delta_j = \frac{\partial E^n}{\partial a_j} = \frac{\partial E^n}{\partial y_j} \frac{\partial y_j}{\partial a_j} = \frac{\partial E^n}{\partial y_j} \frac{\partial f(a_j)}{\partial a_j}$$

$\frac{\partial E^n}{\partial y^n}$ is the derivative of the activation function which we can call $f'(a_j)$. And we know that

$$\frac{\partial E^n}{\partial y^n} = (y^n - t^n)$$

. So to make life easier we can get the error for the output layer like this

$$\delta_j = (y^n - t^n) f'(a_j)$$

It's a little bit trickier to compute the deltas for the hidden units since we don't really know what they do (represent). So instead of looking at the change in activity, we will look at the change in error that results from this hidden

activity. Because hidden activities can have multiple effects on the error we have to combine theses effects. To do this we again use the chain rule

$$\delta_j = \frac{\partial E^n}{\partial a_j} = \sum_k \frac{\partial E^n}{\partial a_k} \frac{\partial a_k}{\partial a_j}$$

where k are the output units that hidden unit j connects to. We know how to compute the first partial derivative from the previous section. Computing the second partial derivative and combining this with the first one leads us to the formula that we use to obtain the delta terms for the hidden units

$$\delta_j = f'(a_j)\Big[\sum_k \partial_k w_{kj}\Big]$$

We then use these results to compute the gradients of the error function. For the output units we use:

$$\frac{\partial E^n}{\partial w_{ji}} = \delta_j = (y^n - t^n) f'(a_j) h_i$$

For the hidden units we use:

$$\frac{\partial E^n}{\partial w_{ji}} = f'(a_j)\Big[\sum_k \partial_k w_{kj}\Big] x_i$$

4 Results

Each time we run the network it obviously gives us different results since the weights are initialized at random for each time we run it. The lowest mean error squared was 304 with an accuracy of 69.4% correct predictions for our 1000 test cases. For that performance we used a learning rate of 0.04 with 25 hidden units when iterating over the network for 1000 times. We ran the code a few more times with learning rates varying between 0.4 and 0.0001, and in the results accuracy varied between 65% and 70%. With a learning rate of 0.001 we could get the error down to 172, but even then the accuracy of our predictions was only 68.6%.

5 Discussion

Our network was able to predict outputs with an accuracy of almost 70 percent. We think this result is decent considering the difficulty of the problem. This idea was supported by the fact that we were able to reduce the error, but doing this did not lead to better predictions on the test data. That means that trying to improve our results this way could lead to overfitting and would not result in more accurate predictions, but instead might even lead to less accurate predictions. To improve our algorithm we would suggest using an approach like bagging to allow us to refine our model without risking overfitting.

References

G. Cybenko. Approximation by superpositions of a sigmoidal function. Math. Control. Signals, Syst., 2:303–314, 1989.
Kurt Hornik. Approximation capabilities of multilayer feedforward networks. Neural Networks, 4(2):251–257, 1991.

6 Appendix

This code was based on the code given in Exercise three of this course.

Listing 1: runMLP

```
load data.mat
learningrate = 0.001;
numOfIterations = 1000;
hiddenUnits = 40;

a = -(sqrt(6) / sqrt(8000));
b = sqrt(6) / sqrt(8000);

W_1 = (b-a) .* rand(50, 2304) + a;
W_2 = (b-a) .* rand(1, 50) + a;

[f_a_3, E_W, W_1, W_2] = mainMLP(numOfIterations,
    learningrate, ytrain, W_1, W_2, Xtest, Xtrain);

%% Compute final accuracy
results = round(f_a_3).';
totalerror = numel(find(results~=ytest))
```

Listing 2: mainMLP

```
% Please write your name and student number below
%
% Name: Jacob Gavin, Laura Pijnacker
% Student number: s4658981, s4332199
%
% Weights that needs to be initialized
function [f_a_3, E_W, W_1, W_2] = mainMLP(
    iteration_number, learning_rate, ytrain, W_1, W_2,
    Xtest, Xtraining)
    % Training %%%%%%%%%%%%
    % %
    % We have to transpose ytrain since it should be on
        the other side

    Y_trainNEW = transpose(ytrain);
```

```
    X_trainingNEW = transpose(Xtraining);
    X_testNEW = transpose(Xtest);

    %face = reshape(X_trainingNEW(:,1),48,48);
    %imagesc(face);
    %colormap(gray);

    for iteration = 1 : iteration_number %
        % %
         disp(['Iteration: ' num2str(iteration) ' / '
             num2str(iteration_number)]); %
        % %
         [E_W, grad_E_W_1, grad_E_W_2] = error(Y_trainNEW
             , W_1, W_2, X_trainingNEW); %
        % %
        disp(['Error: ' num2str(E_W)]); %
        % After each traning iteration we update the
            weights based on learning
        % rate and the gradient of the error function
         W_1 = W_1 - learning_rate * grad_E_W_1; %
         W_2 = W_2 - learning_rate * grad_E_W_2; %
        % %
    end
    % %
    %%%%%%%%%%
    % Test %%%%%%%%%%%%%%%%%%
    % %
    [~, f_a_3] = forwardprop(W_1, W_2, X_testNEW); %
    % %
    %%%%%%
end
% You should now implement the sigmoid function. Your
    implementation should
% compute and return the output of the sigmoid function
    and its gradient.

function [f_a, grad_f_a] = sigmoid(a)
    % First, compute the output of the sigmoid function (
        i.e. f_a). Write your
    % code below:
    %
      ----------------------------------------------------------------------

     f_a = 1 ./ (1+exp(-a));
    %
      ----------------------------------------------------------------------
```

```
    % Once you have computed f_a, use it to compute the
        gradient of the sigmoid
    % function (i.e. grad_f_a). Write your code below:
    %
      ---------------------------------------------------------------------

     grad_f_a = f_a .* (1-f_a);
    %
      ---------------------------------------------------------------------
```

end

```
% You should now implement the forward propagation
    function. Your
% implementation should compute and return the outputs of
     the second and
% third layer units as well as their gradients.

function [f_a_2, f_a_3, grad_f_a_2, grad_f_a_3] =
    forwardprop(W_1, W_2, X)
    % First, compute the inputs of the second layer units
        (i.e. a_2). Write
    % your code below:
    %
      ---------------------------------------------------------------------


    a_2 = W_1 * X;  % where X is Xtrain and W_1 are
        weights of first layer
    %
      ---------------------------------------------------------------------

    % Once you have computed a_2, use it with the sigmoid
        function that you
    % have implemented (i.e. sigmoid) to compute the
        outputs of the second
    % layer units (i.e. f_a_2) and their gradients (i.e.
        grad_f_a_2). Write
    % your code below:
    %
      ---------------------------------------------------------------------


    [f_a_2, grad_f_a_2] = sigmoid(a_2);
```

```
    %
    ------------------------------------------------------------------------

    % Then, compute the inputs of the third layer units (
        i.e. a_3). Write your
    % code below:
    %
    ------------------------------------------------------------------------

    a_3 = W_2 * f_a_2;
    %
    ------------------------------------------------------------------------

    % Once you have computed a_3, use it with the sigmoid
        function that you
    % have implemented (i.e. sigmoid) to compute the
        outputs of the third layer
    % units (i.e. f_a_3) and their gradients (i.e.
        grad_f_a_3). Write your code
    % below:
    %
    ------------------------------------------------------------------------


     [f_a_3, grad_f_a_3] = sigmoid(a_3);

    %
    ------------------------------------------------------------------------

end
% You should now implement the back propagation function.
    Your
% implementation should compute and return the gradients
    of the error
% function.
function [grad_E_W_1, grad_E_W_2] = backprop(f_a_2, f_a_3
    , grad_f_a_2, grad_f_a_3, T, W_2, X)
    % First, compute the errors of the second and third
        layer units (i.e.
    % delta_2 and delta_3). Write you code below:
    %
    ------------------------------------------------------------------------

    delta_3 = grad_f_a_3 .* (f_a_3 - T);
    delta_2 = grad_f_a_2 .* (W_2' * delta_3);
```

```
    %
    % Once you have computed delta_2 and delta_3, use them to compute the
    % gradients of the error function (i.e. grad_E_W_1 and grad_E_W_2). Write
    % your code below:
    %

    grad_E_W_1 = delta_2 * X';
    grad_E_W_2 = delta_3 * f_a_2';
    %

end
% You should now implement the error function. Your implementation should
% compute and return the output of the error function and its gradient.

function [E_W, grad_E_W_1, grad_E_W_2] = error(T, W_1, W_2, X)
    % First, use the forward propagation function that you have implemented
    % (i.e. forwardprop) to compute the outputs of the second and third layer
    % units (i.e. f_a_2 and f_a_3) as well as their gradients (i.e. grad_f_a_2
    % and grad_f_a_3). Write your code below:
    %

    [f_a_2, f_a_3, grad_f_a_2, grad_f_a_3] = forwardprop(W_1, W_2, X);
    %

    % Once you have computed f_a_3, use it to compute the output of the error
    % function (i.e. E_W). Write your code below:
    %
```

```
    E_W = 0.5 * sum(sum((f_a_3 - T) .^ 2));
    %
        --------------------------------------------------------------------------

    % Once you have computed f_a_2, f_a_3, grad_f_a_2 and
        grad_f_a_3, use them
    % with the back propagation function that you have
        implemented (i.e.
    % backprop) to compute the gradients of the error
        function (i.e. grad_E_W_1
    % and grad_E_W_2). Write your code below:
    % ---------------------------------------------------
    [grad_E_W_1, grad_E_W_2] = backprop(f_a_2, f_a_3,
        grad_f_a_2, grad_f_a_3, T, W_2, X);
    %
        --------------------------------------------------------------------------

end
```