POLITECNICO DI TORINO

Corso di Laurea Magistrale
in Data Science and Engineering

Numerical Optimization Report
Exam session: Winter 2020

Multi-Layer Perceptron using MATLAB

Emanuele Fasce

INTRODUCTION

In this report I have built a neural network, also called MLP (Multi-Layer Perceptron), to approximate the values of a function, using the Matlab programming language.
I chose this project since in the Data Science Lab exam I built a MLP in Python using the Keras Library for classifying TripAdvisor reviews as positive or negative, and therefore I became curious to understand how to implement a MLP using Matlab.
To evaluate the results I used two metrics that I used in the regression laboratories, R^2 and SSE.

IMPLEMENTATION

This is the first part of the main script, in which I built the useful data structures for the algorithm and called the training_neural function, that implements the actual training.
Here as an example I used 5 hidden neurons, mu=0.01, tol=1e-6 and 500 as the maximum number of epochs, but the hyperparameter tuning will be discussed later on.

```
%This is the function I want to model
f=@(x) 3+x(1)^2+5*x(1)*x(2);

%First I generate 10 random couple of values for x(1) and x(2), storing
%them in training_x, and I compute f(x(1),x(2)), storing the results in
%training_y

n_train=10;
dimensions=2;
training_x=rand(n_train, dimensions);
training_y=zeros(n_train,1);

for i=1:n_train
    training_y(i)=f(training_x(i,:));
end

%Now_w I add a b_wias to every couple of values x_w(1) and x_w(2)
bias=ones(n_train, 1);
training_x=[training_x, bias];

%I set the hyperparameters of the model: these can be modified in order to
%try to achieve a greater accuracy on the test set.
n_epochs=500;
n_hidden=5;
n_input=3;
mu=0.01;
tol=1e-6;

%Training my neural network with these hyperpameters: this function returns
%all the updated weights of the models, the number of iterations, and the
%SSE on the training set.
[w_w,b_w,x_w,SSE,i]=training_neural(training_x,training_y,n_epochs,tol,mu,n_input,n_hidden,
n_train);
```

Here it is the commented code for the training_neural function.

```
function [w_w,b_w,x_w,SSE,
i]=training_neural(training_x,training_y,n_epochs,tol,mu,n_input,n_hidden, n_train)

    i=0;
    old_sse=+Inf;
    new_sse=0;

    %Building the input-hidden weight matrix, initializing it with random
    %numbers from 0 to 0.5
    w_w=rand(n_input,n_hidden);
    for i=1:n_input
        for j=1:n_hidden
            w_w(i,j)=w_w(i,j)/2;
        end
    end

    %Building x_w array and initializing it in the same way as w_w
    x_w=rand(n_hidden);
    for i=1:n_hidden
        x_w(i)=x_w(i)/2;
    end

    %Initializing the bias weight
     b_w=rand(1);
     b_w=b_w/2;

    %Matrix v_star
    v_star=zeros(n_train,n_hidden);

    %Output array
    out=zeros(n_train, 1);

    %Flag
    sse_flag = 1;

    while (i < n_epochs && sse_flag==1)

        %Building v_star
        for p = 1:n_train
            for h=1:n_hidden
                s=0;
                for j=1:n_input
                    s=s+w_w(j,h)*training_x(p,j);
                end
                v_star(p,h)=s;
            end
        end

        %Updating v_star through the sigmoid function
        for p=1:n_train
            for h=1:n_hidden
                v_star(p,h)=sigmoid(v_star(p,h));
            end
        end
```

```
        %Updating output array
        for p=1:n_train
            s=0;
            for h=1:n_hidden
                s=s+x_w(h)*v_star(p,h);
            end
            out(p)=b_w+s;
        end

        %I store the squared difference of the ouput array and training_y in
        %diff
        dif=training_y-out;
        dif_v=dif.^2;
        diff=sum(dif_v);

        %Updating b_w
        b_w=b_w+mu*sum(dif);


        %Updating w_w
        for i=1:n_input
            for h=1:n_hidden
                s=0;
                for p=1:n_train
                    s=s+(training_y(p)-out(p))*x_w(h)*v(p,h)*(1-v(p,h))*training_x(p,i);
                end
                w_w(i,h)=w_w(i,h)+mu*s;
            end
        end

        %Updating x_w
        for h = 1:n_hidden
            s=0;
            for p = 1:n_train
                s=s+(training_y(p)-out(p))*v(p,h);
            end
            x_w(h)=x_w(h)+mu*s;
        end

        new_sse=diff;

        %Stopping criterium
        if abs(new_sse - old_sse) < tol
            sse_flag=0;
        else
            old_sse=new_sse;
        end

        i=i+1;

    end

    SSE=new_sse;
end
```

This is the code for the neural_test function.

```
function [output] =neural_test(test_x_w,w_h,b_w,x_w, n_hidden, n_input)

    v_star=zeros(n_hidden);

    %Building v_star
    for h = 1:n_hidden
        s=0;
        for j=1:n_input
            s=s+w_h(j,h)*test_x_w(j);
        end
        v_star(h)=s;
    end

    %Updating v_star through sigmoid
    for h=1:n_hidden
        [v_star(h)]=sigmoid(v_star(h));
    end

    %Computing the output value
    out=0;
    for h = 1:n_hidden
        out=out+x_w(h)*v_star(h);
    end
    output=out+b_w;
end
```

In order to test the model, I create other 10 random couple of points and call the neural_test function using each couple as input, with the following code.
I store the real values of f(x(1),x(2)) in true_y and the predicted values in test_y. True_y will be used to evaluate the results.

```
%Preparing the data structures
n_test=10;
test_x=rand(n_test, 2);
test_y=zeros(n_test,1);
true_y=zeros(n_test,1);
bias=ones(n_test,1);

%Storing the real values of the function into true_y
for p=1:n_test
    true_y(p)=f(test_x(p,:));
end

%Adding the bias to test_x
test_x=[test_x, bias];

for p=1:n_test
    test_y(p)=neural_test(test_x(p,:),w_w,b_w,x_w,n_hidden, n_input);
end
```

VALIDATION AND HYPER-PARAMETER TUNING

I decided to use two metrics for the validation: R^2 and SSE.
For the SSE the explanation is straight-forward: obtaining low values means good results.
R^2 instead takes into account also the variance of the samples to evaluate: if values are close to 1 then the model approximates well the function.

Coefficient of Determination: $R^2 = \frac{SSR}{SST} = 1 - \frac{SSE}{SST}$

Sum of Squares Total: $SST = \sum (y - \bar{y})^2$

In order to be more accurate, since 10 training couples of random points and 10 test couples of random points could be subject to a lot of variability, I decided to compute the mean of the metrics. In the Data Science lab exam, I read this is a good practice.

```
%Hyperparameters to modify
n_epochs=1000;
n_hidden=5;
mu=0.005;
tol=1e-8;

cycles=10;
r2_arr=zeros(cycles,1);
sse_arr=zeros(cycles,1);

for n=1:cycles

[w_w,b_w,x_w,SSE,i]=training_neural(training_x,training_y,n_epochs,tol,mu,n_input,n_hidden,
n_train);

%Preparing the data structures
n_test=10;
test_x=rand(n_test, 2);
test_y=zeros(n_test,1);
true_y=zeros(n_test,1);
bias=ones(n_test,1);

%Storing the real values of the function into true_y
for p=1:n_test
    true_y(p)=f(test_x(p,:));
end

%Adding the bias to test_x
test_x=[test_x, bias];

for p=1:n_test
    test_y(p)=neural_test(test_x(p,:),w_w,b_w,x_w,n_hidden, n_input);
end

dif_y=sum((test_y-true_y).^2);
sse=dif_y;
rss=dif_y;
```

```
tss=mean(true_y);
r2=1-rss/tss;

r2_arr(n)=r2;
sse_arr(n)=sse;
r2_average=mean(r2_arr);
sse_average=mean(sse_arr);

end
```

The table with the attempts is the following:

#Epochs	#Hidden	Tolerance	Mu	R^2	SSE
500	5	1,00E-06	0.01	0.82	0.78
1000	5	1,00E-06	0.01	0.84	0.72
2000	5	1,00E-06	0.01	0.97	0.12
1000	10	1,00E-06	0.01	0.89	0.47
1000	30	1,00E-06	0.01	0.97	0.1
2000	30	1,00E-06	0.01	0.82	0.81
1000	30	1,00E-08	0.01	0.99	0.03
1000	30	1,00E-08	0.005	0.77	1.01
1000	30	1,00E-06	changing	0.95	0.19

First, I focused on changing the epochs, and I found a really good result with 2000 epochs.
Then I played around with the number of hidden nodes, keeping the number of epochs fixed at 1000, and got a good result with 30 hidden. So, I tried with 2000 epochs and 30 hidden neurons, mixing the best approaches, but the result was not good at all. I probably incurred into overfitting. I tried then to change the tolerance from 1e-6 to 1e-8 and, even though I had to wait some minutes, the result was surprising, outperforming all the other trials.
I decided to change also mu from 0.01 to 0.005 but the algorithm did not perform well.
I tried to modify the training_neural function to change the value of mu at every iteration and the model performed slightly worse than with the fixed mu.
In conclusion, the model that fits the function better, giving a great result (0.03 as average SSE) is found with 30 hidden nodes, tolerance=1e-8, 1000 epochs and mu=0.01.