Neural Networks Lab 2

The Multi-Layer Perceptron

1 Introduction

In this lab we will consider the Multi-Layer Perceptron, or MLP. The MLP is a feed-forward neural network that has numerous applications. It can be found in image processing, pattern recognition and stock market predictions, just to name a few. During the 1980s it was proven that the MLP with just one hidden layer can be trained such that it can approximate an arbitrary function with arbitrary precision [1]. Therefore, in essence it is not necessary to implement multiple hidden layers, but it can be considerably more efficient to do so, because multiple layers are able to compactly represent a high degree of non-linearity [2].

The training of neural networks with hidden neurons has been an issue for years because of the *credit assignment problem*. In 1986 Rumelhart et al. described the backpropagation-algorithm [3]. This algorithm used the error of the previous layer to estimate the error in the current layer and it is still widely applied to train MLPs. This lab assignment will not be an exception.

A significant limitation of the TLU of last week is that the classification of points in the input space has to be linearly separable. Fortunately, the MLP can deal with non-linearity.

2 Aim of this assignment

The aim is to get familiar with the MLP and the backpropagation-algorithm. We will learn how it can be implemented and how it can be used to solve classification problems and to approximate functions. Network pruning will be covered in the bonus exercise.

3 Theory questions (1.5 pt.)

a) What is the *credit assignment problem*?

b) What are the two factors that determine the error of a neuron in a hidden layer after the forward pass has finished?

c) Give the mathematical definition of the sigmoid function. Clearly state which symbols are constants and what the variable is. How do the constants of the function affect the shape? Also write down the derivative.

d) Why is it impractical to initialize the weights with very high values?

e) Describe three criteria that you can use to determine when to stop learning. Provide one disadvantage per criterion.

f) How can you speed up the learning of a network besides increasing the learning rate? Explain.

g) How can you verify that a network is generalizing for some training data?

h) Describe the problem of *overfitting*.

i) What is network pruning? Describe at least two ways of accomplishing network pruning.

4 An MLP on paper (2 pt.)

a) Draw a two-layer neural network with 2 input neurons, 3 hidden neurons and 2 output neurons. Enumerate the neurons from top to bottom.

b) Why is this considered a two-layer network instead of a 3-layered network?

Let W^h and W^o be the matrices that represent the weights between the input and the hidden layer and the weights between the hidden and the output layer respectively.

In W^h the k-th row will correspond to the connections that go from the k-th input neuron to the hidden layer. For W^o the weights are represented similarly: the weights from the k-th row correspond to the connections that go from the k-th hidden neuron to the output layer.

c) What are the dimensions (number of rows and columns) of W^h and W^o?

d) How do the weights in a single column of a weight matrix relate to each other?

Let a_{ij} be the entry at row i and column j in an $m \times n$ matrix A. Note that matrix indexing starts at 1 (and not at 0).

e) Mark w^h_{12} and w^o_{21} in your drawing by giving the connections a color.

f) Extend the drawing of your network by creating an augmented input layer. What are the dimensions of W^h and W^o now?

g) All thresholds at the input are set to 0.5. Table 1 shows the connections between the first two input nodes and the first hidden node.

Table 1: Connections from the two original input neurons to the first hidden neuron.

input neuron	weights to first hidden neuron
1	0.3
2	0.6

Extend the table with a third row that represents the augmented part of the input. Where can we find the values of this table in W^h?

h) The input at the first input neuron is 1. At the second input neuron it is 0. Let $\vec{\boldsymbol{x}}$ be the row vector containing the inputs of the augmented network. Compute the activation of the first hidden neuron using $\vec{\boldsymbol{x}}$ and the values in Table 1.

i) Explain why we can compute the activation of all hidden neurons by performing the following vector-matrix multiplication:

$$\vec{\boldsymbol{a}}^{\,h} = \vec{\boldsymbol{x}} W^h$$

where $\vec{\boldsymbol{a}}^{\,h}$ is the activation vector where the k-th element corresponds to the k-th activation in the hidden layer.

j) Let $\vec{\boldsymbol{y}}^h$ be the output of the hidden layer. How do we compute the activation of the output layer $\vec{\boldsymbol{a}}^o$? Write down the multiplication in terms of one or more vectors and/or matrices. Transpose the terms if necessary.

5 Implementing an MLP in Matlab (3.5 pt.)

Now we are going to implement an MLP in Matlab. On Nestor you can find the files `mlp.m`, `sigmoid.m` and `d_sigmoid.m` and `output_function.m`. Download and save the files. Start up Matlab and make sure the files are in your working directory.

We will create the MLP step-by-step. The MLP will be able to learn the XOR function using the backpropagation algorithm. Make sure your final algorithm is generic in the sense that it should work for any number of neurons.

5.1 Computing the activation (forward pass)

Have a look at the code. You will see a number of declared variables that will be used later on. The weight matrices $W^h \in \mathbb{R}^{(n_{in} \times n_{hidden})}$ and $W^o \in \mathbb{R}^{(n_{hidden} \times n_{out})}$ are initialized with random values. The range and average of these values can be set manually.

In the matrix $W^h \in \mathbb{R}^{(n_{in} \times n_{hidden})}$ each row corresponds to all connections of one input neuron to the hidden layer. Each column corresponds to all connections from the input layer to one hidden neuron.

In the matrix $W^o \in \mathbb{R}^{(n_{hidden} \times n_{out})}$ each row corresponds to all connections of one hidden neuron to the output layer. Each column corresponds to all connections from the hidden layer to one output neuron.

5.1.1 The hidden layer

The `for`-loop is executed for every row in the matrix `input_data`. Compute the `hidden_activation` using matrix vector multiplication. You do not have to account for an activation function yet.

Hint Contrary to last week, we will now use the *augmented input vector*. The final input is now a constant and so you do not have to subtract the threshold from the activation anymore. The final weight will represent the threshold and the activation of a hidden node is simplified to determining an inner product.

5.1.2 Determining the output at the hidden layer

The hidden layer uses a logistic function to compute the output. This function is implemented in `sigmoid.m`. Here you can choose the function to be:

$$\sigma(x) = \frac{1}{1 + \exp(-x)}$$

Make sure that the function is able to handle a vector $\vec{\boldsymbol{x}}$ by performing an element-wise sigmoid function on $\vec{\boldsymbol{x}}$. Test your function in the command window:

```
x = -3:0.1:3;
y = sigmoid(x);
plot(x,y);
```

Now there should be a graph of a sigmoid curve. If your graph looks OK, you can compute the output of the hidden layer.

5.1.3 The activation of the output

Now that we have the output of the hidden layer we can compute the activation of the output layer easily. Think in terms of matrix-vector multiplications.

5.1.4 The output function

The output function is the function that the output neurons apply to their activation to compute the output. This function has to be implemented in `output_function.m`. Implement this function such that it uses the sigmoid function (`result = sigmoid(input)`). This distinction seems pointless for now, but later on we will implement it differently.

5.2 The backward pass: backpropagation

5.2.1 The slope: $\frac{d}{dx}\text{sigmoid}(x)$

In the file `d_sigmoid.m` the derivative of the sigmoid function is implemented. The function only works if you have properly defined the sigmoid function. Create a file named `d_output_function` and implement the derivative of the output function. Again, making this distinction seems pointless, but it will matter later on.

5.2.2 The local gradient at the output layer

Use the goal matrix and the output of the output layer to determine the local gradient at the output layer. For the output neurons the k-th local gradient δ_k^o can be found as follows:

$$\delta_k^o = \sigma'(a_k^o)(t_k - y_k^o)$$

where t_k is the k-th target (goal) value for the current pattern, y_k^o is the k-th output in the output layer and a_k^o is the k-th activation in the output layer.

In this case there is only one local gradient (a scalar).

5.2.3 The local gradient at the hidden layer

Use the local gradient at the output layer, the matrix W^o and the activation $\vec{\boldsymbol{a}}^h$ to compute the local gradient at the hidden layer. The k-th local gradient can be computed as follows:

$$\delta_k^h = \sigma'(a_k^h) \sum_j \delta_j^o w_{kj}.$$

where $\sigma'(a_k^h)$ is now the derivative of the output function.

Remark

$\sum_j \delta_j^o w_{kj}$ is effectively an inner product. You can compute it for all k using just one matrix-vector multiplication (the result will be a vector).

5.2.4 Computing ΔW

Remember how we used a_{ij} to denote the entry at row i and column j of a matrix A. The elements of the delta weight matrix ΔW are given by:

$$\Delta w_{ij} = \alpha x_i \delta_j$$

where x_i is the i-th input and δ_j is the j-th local gradient of the corresponding layer.

Compute both ΔW^h and ΔW^o.

Remark

The matrices can be computed using so-called *outer-products*.

5.2.5 Updating the weight matrices

Now that we have ΔW^h and ΔW^o we can update the weight matrices. Implement this in your code. The algorithm should work now. Make sure that you let the network train long enough (5000 epochs should do).

5.3 Implementing the stop criterion

We will implement a stop criterion to make sure that we do not always have to wait until the maximum number of epochs has passed. Change the program such that the while loop is terminated when the error drops below a certain value. Call this value `min_error` and set it to 0.01.

6 Testing the MLP (1.5 pt.)

Set the `learn_rate` to 0.2, the number of hidden neurons to 2 and the `noise_level` to $5\% = 0.05$. Answer the following questions:

a) Is it guaranteed that the network finds a solution. Why so?

Set the number of hidden neurons to 20.

b) How many epochs are needed to find a solution?

c) Set the `noise_level` to 0.5. Explain what happens.

d) Set the `noise_level` to 0.2. Change the `weight_spread` to 5. What can you observe? Explain your results using the delta-rule.

e) Set the noise_level to 1%. Leave the `weight_spread` at 5. There are two *qualitatively* different solutions to the XOR problem. What are these two? Include a figure of both solutions.

f) Which shape does the graph of the error usually have? Explain the shape with the use of the delta-rule.

7 Another function (1.5 pt.)

As stated before, it is proven that an MLP can approximate an arbitrary function. This means that we could make our MLP learn to approximate any function. The file `mlp_sinus.m` is a lot like `mlp.m` and can be found on Nestor. The `for`-loop still needs to be implemented. Copy the body of the `for`-loop from `mlp.m` to `mlp_sinus.m`. Do not forget to implement the stop-criterion.

This network has 1 output and 1 input. Given an input x, the network should give the output $\sin(x)$. To learn a sine the network also needs to put out negative values. Therefore, we have to change the output function. We will need a linear output function. Modify the code such that the output uses the identity function ($f(x) = x$) for the activation. Make sure you use the derivative of the identity function in the delta rule.

Set the learning rate to 0.05 and the maximum number of epochs to 5000. Use 20 neurons in the hidden layer. The network will now attempt to learn a sine in the domain $0 \leq x < 2\pi$. Answer the following questions:

a) Is the network capable of learning the sine function?

b) Set `n_examples` in the top of the file to 5. Rerun the simulation. What can you observe? With which feature of neural networks does this phenomenon correspond?

c) Set `plot_bigger_picture` to `true`. How is the domain of the network determined? What happens if the input is outside of this domain?

d) At least how many neurons are required to learn a sine?

e) You have modified the output function for this part of the lab assignment. Does the XOR learning network still work? Why so?

8 Bonus: Network pruning

Copy the file `mlp_sinus.m` and name it `mlp_sinus_pruning.m`. Train this network to approximate a sine with 50 hidden neurons. Set `n_examples` to 20. Train the network until the error drops below 0.01. Obviously, we could do with less than 50 neurons. Let's prune the network:

1. Set the incoming and outgoing weights of a single neuron to 0. Make a backup of the old weights.

2. Compute the mean squared error of the network over all example inputs.

3. Repeat 1 and 2 for all neurons in the hidden layer. The neuron that is least significant can be pruned by definitively setting it to 0. The least significant neuron is the neuron that minimizes the increase of the error.

4. Repeat 1 to 3 as long as the error stays under 0.1.

Answer the following questions:

a) How many neurons are required to approximate a sine when pruning a network of 50 hidden neurons?

b) Compare your answer with your answer to (d) in section 7. What can you say about brute force pruning?

9 What to hand in

Write a report that adheres to the guidelines on Nestor. Include all of your code and provide meaningful comments to the code.

References

[1] K. Hornik, M. Stinchcombe, and H. White, “Multilayer feedforward networks are universal approximators,” *Neural Networks*, vol. 2, no. 5, pp. 359–366, 1989.

[2] H. Larochelle, Y. Bengio, J. Louradour, and P. Lamblin, “Exploring strategies for training deep neural networks,” *The Journal of Machine Learning Research*, vol. 10, pp. 1–40, 2009.

[3] D. Rumelhart, G. Hinton, and R. Williams, “Learning representations by back-propagating errors,” *Nature*, vol. 323, no. 6088, pp. 533–536, 1986.