Chapter 15 - Backpropagation

The backpropagation algorithm is the classical feedforward artificial neural network. It is the technique still used to train large deep learning networks. In this tutorial, you will discover how to implement the backpropagation algorithm from scratch with Perl. After completing this tutorial, you will know:

- How to forward-propagate an input to calculate an output.
- How to backpropagate error and train a network.
- How to apply the backpropagation algorithm to a real-world predictive modeling problem.

Let's get started.

15.1 Description

This section provides a brief introduction to the Backpropagation Algorithm and the Wheat Seeds dataset that we will be using in this tutorial.

15.1.1 Backpropagation Algorithm

The Backpropagation algorithm is a supervised learning method for multilayer feedforward networks from the field of Artificial Neural Networks. Feedforward neural networks are inspired by the information processing of one or more neural cells, called a neuron. A neuron accepts input signals via its dendrites, which pass the electrical signal down to the cell body. The axon carries the signal out to synapses, which are the connections of a cell's axon to other cell's dendrites.

The principle of the backpropagation approach is to model a given function by modifying internal weightings of input signals to produce an expected output signal. The system is trained using a supervised learning method where the error between the system's output and a known expected output is presented to the system and used to modify its internal state.

Technically, the backpropagation algorithm is a method for training the weights in a multilayer feedforward neural network. As such, it requires a network structure to be defined of one or more layers where one layer is fully connected to the next layer. A standard network structure is one input layer, one hidden layer, and one output layer. Backpropagation can be used for both classification and regression problems, but we will focus on classification in this tutorial.

In classification problems, best results are achieved when the network has one neuron in the output layer for each class value. For example, a 2-class or binary classification problem with the class values of A and B. These expected outputs would have to be transformed into binary vectors with one column for each class value. Such as [1, 0] and [0, 1] for A and B respectively. This is called a one hot encoding.

15.1.2 Wheat Seeds Dataset

In this tutorial we will use the Wheat Seeds Dataset. This dataset involves the prediction of the species of wheat seeds. The baseline performance on the problem is approximately 28%. You can learn more about it in Appendix A, Section A.10. Download the dataset and save it into your current working directory with the filename seeds dataset.csv. The dataset is in tab-separated format, so you must convert it to CSV using a text editor or a spreadsheet program.

15.2 Tutorial

This tutorial is broken down into 6 parts:

1. Initialize Network.
2. Forward-Propagate.
3. Backpropagate Error.
4. Train Network.
5. Predict.
6. Wheat Seeds Case Study.

These steps will provide the foundation that you need to implement the backpropagation algorithm from scratch and apply it to your own predictive modeling problems.

15.2.1 Initialize Network

Let's start with something easy: the creation of a new network ready for training. Each neuron has a set of weights that need to be maintained. One weight for each input connection and an additional weight for the bias. We will need to store additional properties for a neuron during training, therefore we will use a dictionary to represent each neuron and store properties by names such as weights for the weights.

A network is organized into layers. The input layer is really just a row from our training dataset. The first real layer is the hidden layer. This is followed by the output layer that has one neuron for each class value. We will organize layers as arrays of dictionaries and treat the whole network as an array of layers. It is good practice to initialize the network weights to small random numbers. In this case, will we use random numbers in the range of 0 to 1. Below is a function named initialize_network() that creates a new neural network ready for training. It accepts three parameters: the number of inputs n_inputs, the number of neurons to have in the hidden layer n_hidden and the number of outputs n_outputs. You can see that for the hidden layer we create n_hidden neurons and each neuron in the hidden layer has n_inputs + 1 weights, one for each input column in a dataset and an additional one for the bias.

You can also see that the output layer that connects to the hidden layer has n_output neurons, each with n_hidden + 1 weights. This means that each neuron in the output layer connects to (has a weight for) each neuron in the hidden layer.

In [1]:
```perl
use strict;
use warnings;
use Data::Dump qw(dump);
use List::Util qw(sum zip shuffle max uniq);
use sml; # Statistical Machine Learning Library
IPerl->load_plugin('Chart::Plotly'); # Cambiar por use Chart::Plotly qw(show_plot); si lo ejecutas fuera del entorno Jupyter
```

```
# Dentro de sml.pm carregar la librería Plotly:
# use Chart::Plotly::Plot;
# use Chart::Plotly::Trace::Scatter;
```

In [2]:
```
# Function To Initialize a Multilayer Perceptron Network.
# Initialize a network

sub initialize_network{
  my ($self, $n_inputs, $n_hidden, $n_outputs) = @_;

  my @network;
  my @hidden_layer
     = map { {'weights' => [map { rand() } 0 .. $n_inputs]} }
     (1 .. $n_hidden);
  push @network, \@hidden_layer;

  my @output_layer
  = map { {'weights' => [map { rand() } 0 .. $n_hidden]} }
  (1 .. $n_outputs);
  push @network, \@output_layer;

  return \@network;
}

sml->add_to_class('initialize_network', \&{'initialize_network'});
```

Out[2]:
```
*sml::initialize_network
```

Let's test out this function. Below is a complete example that creates a small network.

In [3]:
```
# Example of initializing a network
# Example of Initializing a Multilayer Perceptron Network.

# Test initializing a network
srand(1);
my $network = sml->initialize_network(2, 1, 2);

# Imprimir resultados
my @layer_names = ('Hidden', 'Output');
for my $layer (zip \@layer_names, $network){
  printf "\n%s layer:\n", $layer->[0];
  while (my ($i, $neuron) = each @{$layer->[1]}){
    printf "Neuron[%d] weights & bias:%s\n", $i, dump $neuron->{'weights'};
  }
}
```

```
Hidden layer:
Neuron[0] weights & bias:[0.0416303447718782, 0.454492444728629, 0.834817218166915]

Output layer:
Neuron[0] weights & bias:[0.3359860301452, 0.565489403566136]
Neuron[1] weights & bias:[0.00176691239174431, 0.18758951699996]
```

Running the example, you can see that the code prints out each layer one by one. You can see the hidden layer has one neuron with 2 input weights plus the bias. The output layer has 2 neurons, each with 1 weight plus the bias.

Sample Output from Initializing a Network.

Hidden layer:
Neuron[0] weights & bias:[0.0416303447718782, 0.454492444728629, 0.834817218166915]

Output layer:
Neuron[0] weights & bias:[0.3359860301452, 0.565489403566136]
Neuron[1] weights & bias:[0.00176691239174431, 0.18758951699996]

Now that we know how to create and initialized a network, let's see how we can use it to calculate an output.

15.2.2 Forward-Propagate

We can calculate an output from a neural network by propagating an input signal through each layer until the output layer outputs its values. We call this forward-propagation. It is the technique we will need to generate predictions during training that will need to be corrected, and it is the method we will need after the network is trained to make predictions on new data. We can break forward-propagation down into three parts:

1. Neuron Activation.
2. Activation Function(Neuron Transfer).
3. Forward-Propagation.

Neuron Activation

The first step is to calculate the activation of one neuron given an input. The input could be a row from our training dataset, as in the case of the hidden layer. It may also be the outputs from each neuron in the hidden layer, in the case of the output layer. Neuron activation is calculated as the weighted sum of the inputs. Much like linear regression.

$$\text{activation} = \text{bias} + \sum_{i=1}^{n} \text{weight}_i \times \text{input}_i \tag{15.1}$$

Where weight is a network weight, input is an input value, i is the index of a weight or an input and bias is a special weight that has no input to multiply with (or you can think of the input as always being 1.0). Below is an implementation of this in a function named activate(). You can see that the function assumes that the bias is the last weight in the list of weights. This helps here and later to make the code easier to read.

In [4]:
```
# Calculate neuron activation for an input
# Function To Activate a Neuron.

sub activate{
  my ($self, $weights, $inputs) = @_;

  my $activation = $weights->[-1];
  for (my $i = 0; $i < scalar(@$weights) -1; $i++) {
    $activation += $weights->[$i] * $inputs->[$i];
  }
  return $activation;
}

sml->add_to_class('activate', \&{'activate'});
```

Out[4]:
```
*sml::activate
```

Now, let’s see how to use the neuron activation.

Activation Function(Neuron Transfer)

Once a neuron is activated, we need to transfer the activation to see what the neuron output actually is. Different transfer functions can be used. It is traditional to use the sigmoid activation function, but you can also use the tanh (hyperbolic tangent) function to transfer outputs. More recently, the rectifier transfer function has been popular with large deep learning networks.

The sigmoid activation function looks like an S shape: it’s also called the logistic function. It can take any input value and produce a number between 0 and 1 on an S-curve. It is also a function of which we can easily calculate the derivative (slope) that we will need later when backpropagating error. We can transfer an activation function using the sigmoid function as follows:

$$\text{output} = \frac{1}{1 + e^{-\text{activation}}} \tag{15.2}$$

Where e is the base of the natural logarithms (Euler’s number). Below is a function named transfer() that implements the sigmoid equation.

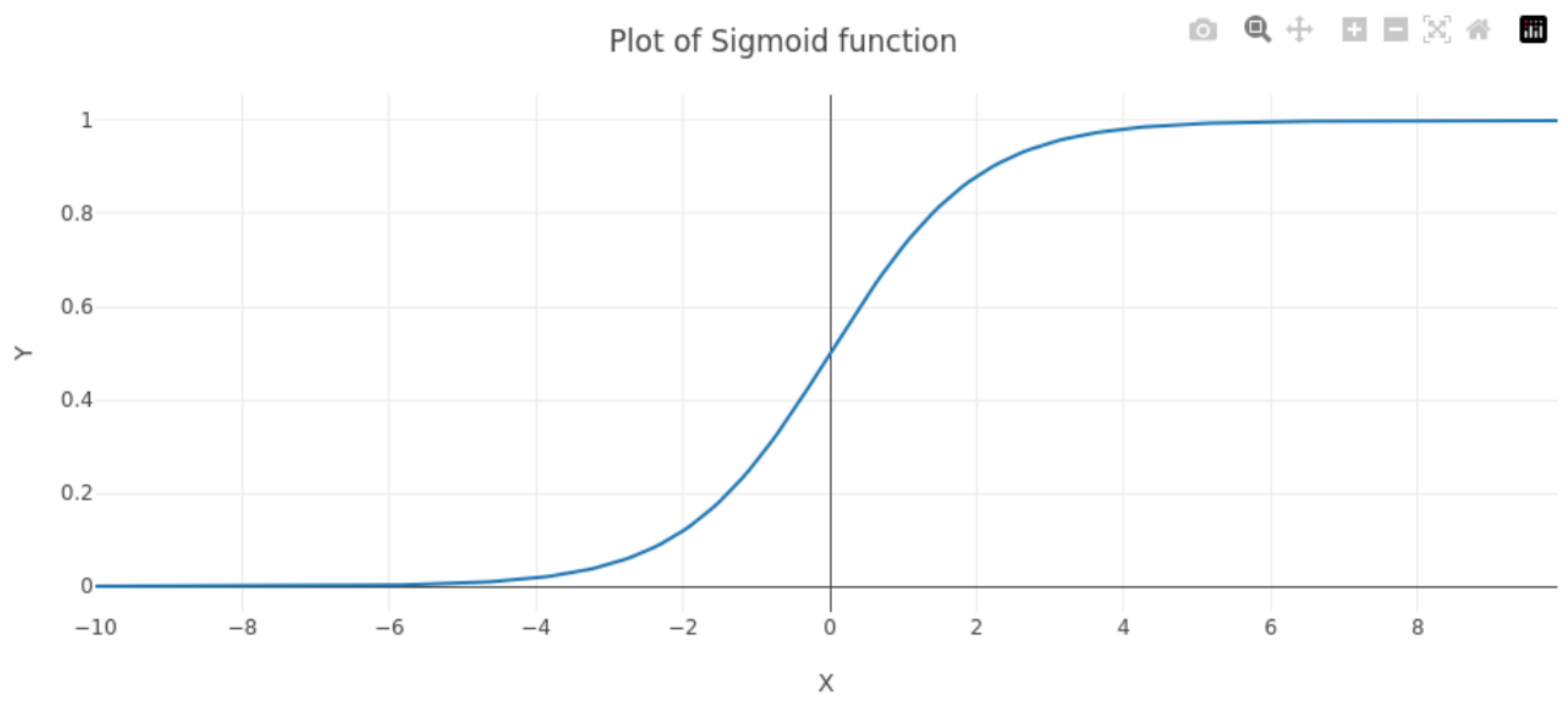
In [5]:
```
# Transfer neuron activation
# Function To Transfer a Neuron’s Activation.

sub transfer{
  my ($self, $activation) = @_;
  return 1.0 / (1.0 + exp(-$activation));
}
```

```
sml->add_to_class('transfer', \&{'transfer'});
```

```
Out[5]: *sml::transfer
```

In [6]:
```
my $X = [map {$_ * 0.1} (-100 .. 99)];
my $Y = [map {sml->transfer($_)} @$X];

my $trace = [new Chart::Plotly::Trace::Scatter(x      => $X,
                                              y      => $Y,
                                              name   => 'Sigmoid',
                                              mode   => 'lines')];

my $layout = {title => {text => 'Plot of Sigmoid function'},
              xaxis => {title => 'X'}, yaxis => {title => 'Y'},
              width  => 900, height => 400,
              margin => { l => 50, r => 0, t => 50, b => 50 }};

my $plot = new Chart::Plotly::Plot(traces => $trace,
                                   layout => $layout);

IPerl->display($plot); # Cambiar por la función show_plot($plot); si lo ejecutas fuera del entorno Jupyter
```

In [7]:
```
sml->embedplot($plot, width=>900, height=>450); # Este comando es de uso exclusivo del docente
```



Now that we have the pieces, let’s see how they are used.

Forward-Propagation

Forward-propagating an input is straightforward. We work through each layer of our network calculating the outputs for each neuron. All of the outputs from one layer become inputs to the neurons on the next layer. Below is a function named forward propagate() that implements the forward-propagation for a row of data from our dataset with our neural network.

You can see that a neuron’s output value is stored in the neuron with the name output. You can also see that we collect the outputs for a layer in an array named new inputs that becomes the array inputs and is used as inputs for the following layer. The function returns the outputs from the last layer also called the output layer.

Let’s put all of these pieces together and test out the forward-propagation of our network. We define our network inline with one hidden neuron that expects 2 input values and an output layer with two neurons.

In [8]:
```
# Forward-propagate input to a network output
# Function To Forward-Propagate Input Through a Network.

sub forward_propagate{
  my ($self, $network, $row) = @_;

  my @inputs = @$row;
  for my $layer (@$network){
    my @new_inputs = ();
    for my $neuron (@$layer){
      my $activation      = sml->activate($neuron->{'weights'}, \@inputs);
      $neuron->{'output'} = sml->transfer($activation);
      push @new_inputs, $neuron->{'output'};
    }
    @inputs = @new_inputs;
  }

  return \@inputs;
}

sml->add_to_class('forward_propagate', \&{'forward_propagate'});
```

```
Out[8]: *sml::forward_propagate
```

Let's put all of these pieces together and test out the forward-propagation of our network. We define our network inline with one hidden neuron that expects 2 input values and an output layer with two neurons.

```
In [9]: # Example of forward propagating input
        # Example of Forward-Propagating an Input Through a Network.

        # test forward propagation
        $network = [[{'weights' => [0.13436424411240122, 0.8474337369372327, 0.763774618976614]}],
                    [{'weights' => [0.2550690257394217, 0.49543508709194095]},
                     {'weights' => [0.4494910647887381, 0.651592972722763]}]];

        my $row    = [1, 0, undef];
        my $output = sml->forward_propagate($network, $row);

        print dump $output;
```

```
[0.662997012985289, 0.725316072527975]
Out[9]: 1
```

Running the example propagates the input pattern [1, 0] and produces an output value that is printed. Because the output layer has two neurons, we get a list of two numbers as output. The actual output values are just nonsense for now, but next, we will start to learn how to make the weights in the neurons more useful.

[0.6629970129852887, 0.7253160725279748]
Sample Output from Forward-Propagate Input Through a Network.

15.2.3 Backpropagate Error

The backpropagation algorithm is named for the way in which weights are trained. Error is calculated between the expected outputs and the outputs forward-propagated from the network. These errors are then propagated backward through the network from the output layer to the hidden layer, assigning blame for the error and updating weights as they go. The math for backpropagating error is rooted in calculus, but we will remain high level in this section and focus on what is calculated and how rather than why the calculations take this particular form.

This part is broken down into two sections

1. Transfer Derivative.
2. Error Backpropagation

Transfer Derivative

Given an output value from a neuron, we need to calculate its slope. We are using the sigmoid transfer function, the derivative of which can be calculated as follows:

$$\text{derivative} = \text{output} \times (1.0 - \text{output}) \tag{15.3}$$

Below is a function named **transfer derivative()** that implements this equation.

In [10]:
```
# Calculate the derivative of an neuron output
# Function To Calculate the Derivative of a Neuron’s Output.

sub transfer_derivative{
  my ($self, $output) = @_;
  return $output * (1.0 - $output);
}

sml->add_to_class('transfer_derivative', \&{'transfer_derivative'});
```

Out[10]:
```
*sml::transfer_derivative
```

Now, let’s see how this can be used.

Error Backpropagation

The first step is to calculate the error for each output neuron; this will give us our error signal (input) to propagate backwards through the network. The error for a given neuron can be calculated as follows:

$$\text{error} = \text{expected - output} \times \text{transfer\_derivative}(\text{output}) \tag{15.4}$$

Where expected is the expected output value for the neuron, output is the output value for the neuron and transfer derivative() calculates the slope of the neurons output value, as shown above. This error calculation is used for neurons in the output layer. The expected value is the class value itself. In the hidden layer, things are a little more complicated.

The error signal for a neuron in the hidden layer is calculated as the weighted error of each neuron in the output layer. Think of the error traveling back along the weights of the output layer to the neurons in the hidden layer. The backpropagated error signal is accumulated and then used to determine the error for the neuron in the hidden layer, as follows:

$$\text{error} = (\text{weight}_j \times \text{error}_j) \times \text{transfer\_derivative}(\text{output}) \tag{15.5}$$

Where error j is the error signal from the jth neuron in the output layer, weight (15.5) k is the weight that connects the kth neuron to the current neuron and output is the output for the current neuron. Below is a function named backward propagate error() that implements this procedure. You can see that the error signal calculated for each neuron is stored with the name delta.

You can see that the layers of the network are iterated in reverse order, starting at the output and working backwards. This ensures that the neurons in the output layer have delta values calculated rst that neurons in the hidden layer can use in the subsequent iteration. I chose the name delta to re ect the change the error implies on the neuron (e.g. the weight delta).

You can see that the error signal for neurons in the hidden layer is accumulated from neurons in the output layer where the hidden neuron number j is also the index of the neurons weight in the output layer neuron[weights][j].

In [11]:

```perl
# Backpropagate error and store in neurons
# Function To Backpropagate Error Through a Network.

sub backward_propagate_error{
  my ($self, $network, $expected) = @_;

  my ($layer, @errors, $error);
  for my $i (reverse 0 .. $#{$network}){
    $layer = $network->[$i];
    @errors = ();
    if ($i != $#{$network}){
      for my $j (0 .. $#{$layer}){
        $error = 0.0;
        for my $neuron (@{$network->[$i + 1]}){
          $error += $neuron->{'weights'}[$j] * $neuron->{'delta'};
        }
        push @errors, $error;
      }
    }else{
      for my $j (0 .. $#{$layer}){
        my $neuron = $layer->[$j];
        push @errors, $expected->[$j] - $neuron->{'output'};
      }
    }

    for my $j (0 .. $#{$layer}){
      my $neuron = $layer->[$j];
      $neuron->{'delta'} = $errors[$j] * sml->transfer_derivative($neuron->{'output'});
    }
  }
}

sml->add_to_class('backward_propagate_error', \&{'backward_propagate_error'});
```

Out[11]:
```
*sml::backward_propagate_error
```

Lets put all of the pieces together and see how it works. We de ne a xed neural network with output values and backpropagate an expected output pattern. The complete example is listed below.

In [12]:
```
# Example of backpropagating error
# Example of Backpropagating Error Through a Network.

# test backpropagation of error
$network = [[{'output' => 0.7105668883115941, 'weights' => [0.13436424411240122, 0.8474337369372327, 0.763774618976614]}],
            [{'output' => 0.6213859615555266, 'weights' => [0.2550690257394217, 0.49543508709194095]},
             {'output' => 0.6573693455986976, 'weights' => [0.4494910647887381, 0.651592972722763]}]];

my $expected = [0, 1];
sml->backward_propagate_error($network, $expected);
for my $layer (@$network){
  printf "%s\n", dump $layer;
}
```

```
[
  {
    delta   => -0.000534804804661052,
    output  => 0.710566888311594,
    weights => [0.134364244112401, 0.847433736937233, 0.763774618976614],
  },
]
[
  {
    delta   => -0.146190646835828,
    output  => 0.621385961555527,
    weights => [0.255069025739422, 0.495435087091941],
  },
  {
    delta   => 0.0771723774346327,
    output  => 0.657369345598698,
    weights => [0.449491064788738, 0.651592972722763],
  },
]
```

Running the example prints the network after the backpropagation of error is complete. You can see that error values are calculated and stored in the neurons for the output layer and the hidden layer.

[{ ' output ' : 0.7105668883115941,
' weights ' : [0.13436424411240122, 0.8474337369372327, 0.763774618976614],

```
' delta ' : -0.0005348048046610517}]
[{ ' output ' : 0.6213859615555266,
' weights ' : [0.2550690257394217, 0.49543508709194095],
' delta ' : -0.14619064683582808},
{ ' output ' : 0.6573693455986976,
' weights ' : [0.4494910647887381, 0.651592972722763],
' delta ' : 0.0771723774346327}]
```

Sample Output from Backpropagate Error Through a Network.

Now lets use the backpropagation of error to train the network.

15.2.4 Train Network

The network is trained using stochastic gradient descent. Gradient descent was introduced and described in Section 8.1.2. The procedure involves multiple iterations of exposing a training dataset to the network and for each row of data forward-propagating the inputs, backpropagating the error and updating the network weights. This part is broken down into two sections:

1. Update Weights.
2. Train Network.

Update Weights

Once errors are calculated for each neuron in the network via the backpropagation method above, they can be used to update weights. Network weights are updated as follows:

$$\text{weight} = \text{weight} + \text{learning rate} \times \text{error} \times \text{input} \tag{15.6}$$

Where weight is a given weight, learning (15.6) rate is a parameter that you must specify, error is the error calculated by the backpropagation procedure for the neuron and input is the input value that caused the error. The same procedure can be used for updating the bias weight, except there is no input term, or input is the xed value of 1.0.

Learning rate controls how much to change the weight to correct for the error. For example, a value of 0.1 will update the weight 10% of the amount that it possibly could be updated. Small learning rates are preferred that cause slower learning over a large number of training iterations. This increases the likelihood of the network nding a good set of weights across all layers rather than the fastest set of weights that minimize error (called premature convergence).

Below is a function named update weights() that updates the weights for a network given an input row of data, a learning rate and assume that a forward and backward propagation have already been performed. Remember that the input for the output layer is a collection of outputs from the hidden layer.

In [19]:
```
# Update network weights with error
# Function To Update Weights in a Network.

sub update_weights{
  my ($self, $network, $row, $l_rate) = @_;
  for my $i (0 .. $#{$network}){
    my @inputs = @$row[0 .. $#$row -1];
    if ($i != 0) {
      @inputs = map { $_->{'output'} } @{$network->[$i -1]};
    }
    for my $neuron (@{$network->[$i]}){
      for my $j (0 .. $#inputs){
        $neuron->{'weights'}[$j] += $l_rate * $neuron->{'delta'} * $inputs[$j];
      }
      $neuron->{'weights'}[-1] += $l_rate * $neuron->{'delta'};
    }
  }
}

sml->add_to_class('update_weights', \&{'update_weights'});
```

Out[19]:
```
*sml::update_weights
```

Now that we know how to update network weights, lets see how we can do it repeatedly.

Train Network

As mentioned, the network is updated using stochastic gradient descent. This involves rst looping for a xed number of epochs and within each epoch updating the network for each row in the training dataset.

Because updates are made for each training pattern, this type of learning is called online learning. If errors were accumulated across an epoch before updating the weights, this is called batch learning or batch gradient descent. Below is a function that implements the training of an already initialized neural network with a given training dataset, learning rate, xed number of epochs and an expected number of output values.

The expected number of output values is used to transform class values in the training data into a one hot encoding. That is a binary vector with one column for each class value to match the output of the network. This is required to calculate the error for the output layer. You can also see that the sum squared error between the expected output and the network output is accumulated each epoch and printed. This is helpful to create a trace of how much the network is learning and improving each epoch.

In [13]:
```
# Train a network for a fixed number of epochs
# Function To Train a Neural Network on a Dataset.

sub train_network{
  my ($self, $network, $train, $l_rate, $n_epoch, $n_outputs, $test) = @_;

  my (@train_losses, @test_losses, $sum_error, $outputs, $expected);
  for my $epoch (1 .. $n_epoch){
    $sum_error = 0.0;
    for my $row (@$train){
      $outputs     = sml->forward_propagate($network, $row);
      $expected    = [(0) x $n_outputs];
      $expected->[$row->[-1]] = 1;
      $sum_error += sum(map { ($expected->[$_] - $outputs->[$_]) ** 2 } 0 .. $#$expected);
      sml->backward_propagate_error($network, $expected);
      sml->update_weights($network, $row, $l_rate);
    }
    printf " >epoch=%d, lrate=%.3f, error=%.3f\n", $epoch, $l_rate, $sum_error if $epoch % 10 == 0 || $epoch == 1;

    # Storing performance data for the plot:
    push @train_losses, $sum_error;
    if (defined $test){
      $sum_error = 0.0;
      for my $row (@$test){
        $outputs     = sml->forward_propagate($network, $row);
        $expected    = [(0) x $n_outputs];
        $expected->[$row->[-1]] = 1;
        $sum_error += sum(map { ($expected->[$_] - $outputs->[$_]) ** 2 } 0 .. $#$expected);
      }
      push @test_losses, $sum_error;
    }
  }

  return $network, \@train_losses, \@test_losses;
}

sml->add_to_class('train_network', \&{'train_network'});
```

Out[13]:
```
*sml::train_network
```

We now have all of the pieces to train the network. We can put together an example that includes everything weve seen so far including network initialization and train a network on a small dataset. Below is a small contrived dataset that we can use to test out training our neural network.

In [14]:
```
# Small Contrived Dataset for Testing Backpropagation.

my $dataset = [
    [2.7810836, 2.550537003, 0],
    [1.465489372, 2.362125076, 0],
    [3.396561688, 4.400293529, 0],
    [1.38807019, 1.850220317, 0],
    [3.06407232, 3.005305973, 0],
    [7.627531214, 2.759262235, 1],
    [5.332441248, 2.088626775, 1],
    [6.922596716, 1.77106367, 1],
    [8.675418651, -0.242068655, 1],
    [7.673756466, 3.508563011, 1],
];

printf "%-15s %-15s %s\n", "X1", "X2", "Y";
for my $row (@$dataset) {
  printf "%-15.9f %-15.9f %d\n", @$row;
}
```

```
X1              X2              Y
2.781083600     2.550537003     0
1.465489372     2.362125076     0
3.396561688     4.400293529     0
1.388070190     1.850220317     0
3.064072320     3.005305973     0
7.627531214     2.759262235     1
5.332441248     2.088626775     1
6.922596716     1.771063670     1
8.675418651     -0.242068655    1
7.673756466     3.508563011     1
```

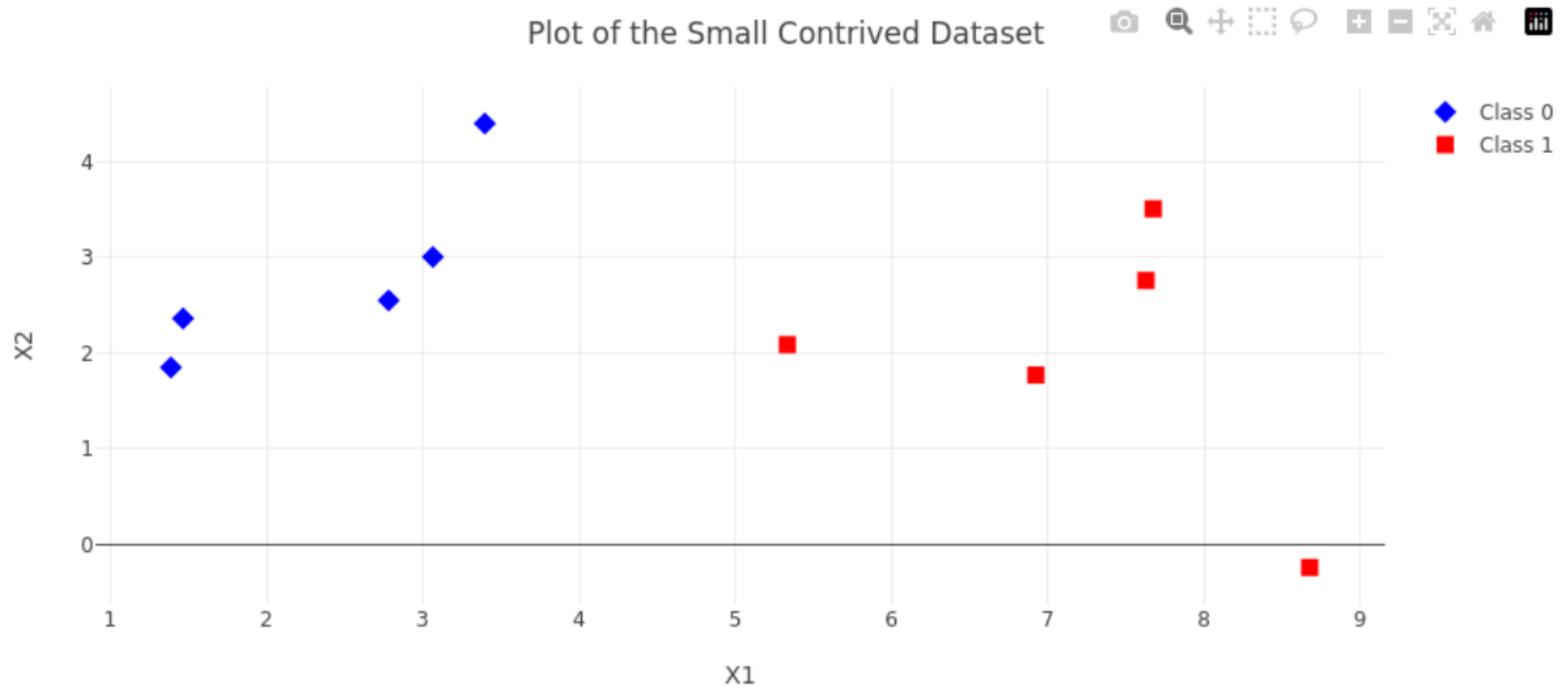
Below is a plot of the dataset using different colors to show the different classes for each point.

In [15]:

```
my @classes = uniq map {$_->[-1]} @$dataset;
my %colors  = map {$_->[0] => $_->[1]} (zip \@classes, ['blue', 'red']);
my %symbols = map {$_->[0] => $_->[1]} (zip \@classes, ['diamond', 'square']);

my (@traces, $data_by_class, $X1, $X2) = ();
for my $class (@classes){
  $data_by_class = [grep {$_->[-1] eq $class} @$dataset];
  ($X1, $X2) = (zip @$data_by_class);
  push @traces, new Chart::Plotly::Trace::Scatter(x      => $X1,
                                                  y      => $X2,
                                                  name   => "Class $class",
                                                  mode   => 'markers',
                                                  marker => { symbol => $symbols{$class},
                                                              color  => $colors{$class},
                                                              size   => 10 });
}

$layout = {title => {text => 'Plot of the Small Contrived Dataset'},
              xaxis => {title => 'X1'}, yaxis => {title => 'X2'},
              width  => 900, height => 400,
              margin => { l => 50, r => 0, t => 50, b => 50 }};

$plot = new Chart::Plotly::Plot(traces => \@traces,
                                  layout => $layout);
```

```
IPerl->display($plot); # Cambiar por la función show_plot($plot); si lo ejecutas fuera del entorno Jupyter
```

In [16]:
```
sml->embedplot($plot, width=>900, height=>450); # Este comando es de uso exclusivo del docente
```



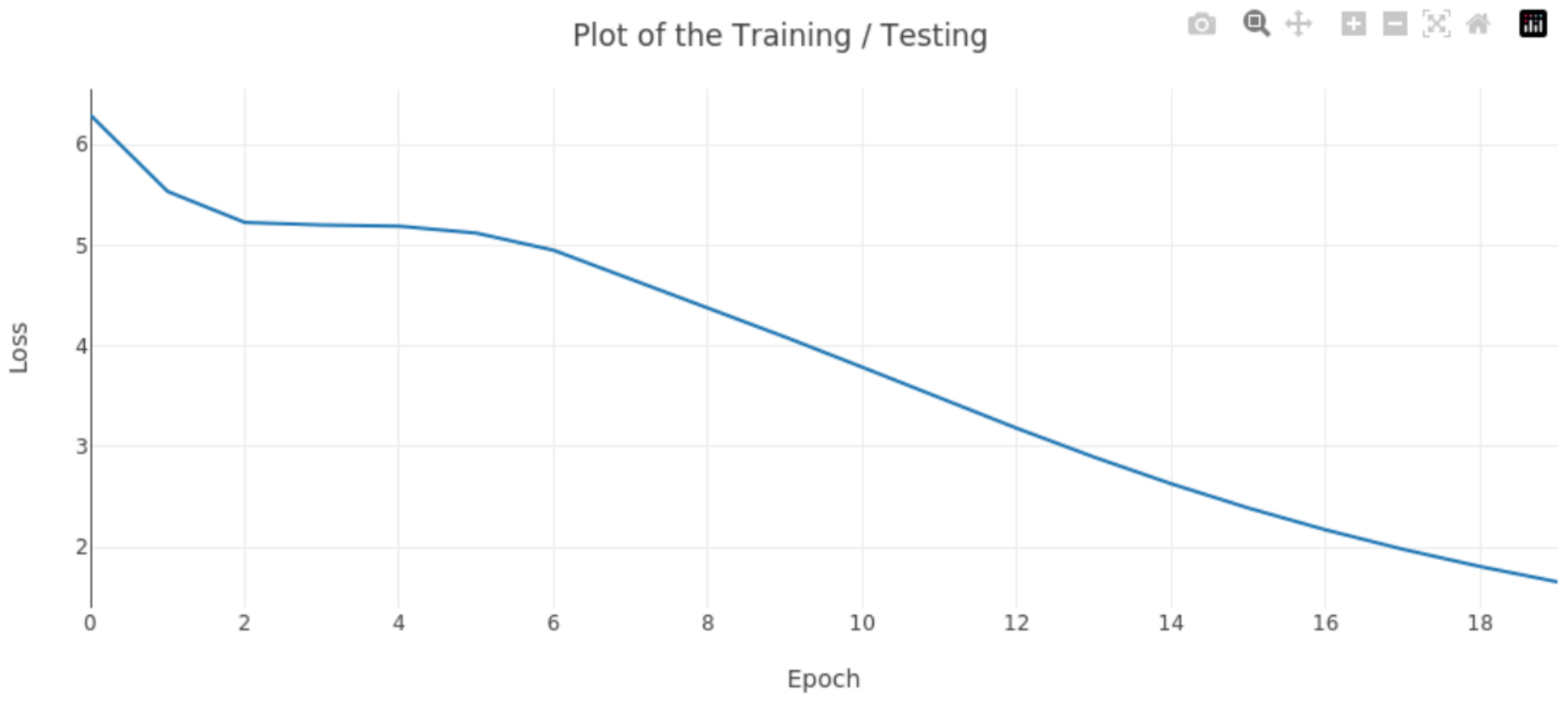
Below is the complete example. We will use 2 neurons in the hidden layer. It is a binary classification problem (2 classes) so there will be two neurons in the output layer. The network will be trained for 20 epochs with a learning rate of 0.5, which is high because we are training for so few iterations.

In [20]:
```
# Example of training a network by backpropagation
# Example of Training a Network on the Contrived Dataset.

my $n_inputs  = $#{$dataset->[0]};
my $n_outputs = keys %{{ map { $_->[-1] => undef } @$dataset }};

$network = sml->initialize_network($n_inputs, 2, $n_outputs);
my ($weights, $train_losses, undef) = sml->train_network($network, $dataset, 0.5, 20, $n_outputs);

for my $layer (@$network){
  printf "%s\n", dump $layer;
}
```

```
>epoch=1, lrate=0.500, error=6.295
>epoch=10, lrate=0.500, error=4.089
>epoch=20, lrate=0.500, error=1.658
[
  {
    delta   => 0.0110421614368629,
    output  => 0.94110178199232,
    weights => [1.3326211533855, -1.93304302946038, -0.274163463903728],
  },
  {
    delta   => -3.2656411400222e-06,
    output  => 0.999903999373995,
    weights => [0.854113470594309, 0.745590204348853, 0.0807329763705832],
  },
]
[
  {
    delta   => -0.0435454661809677,
    output  => 0.23924904901634,
    weights => [-2.31194663576695, 0.465528768784266, 0.490667239362583],
  },
  {
    delta   => 0.0443266049755316,
    output  => 0.758210566000323,
    weights => [2.26396671576714, -0.266614816483855, -0.657188211876175],
  },
]
```

In [21]:
```
@traces = (new Chart::Plotly::Trace::Scatter(x => [0 .. $#$train_losses],
                                             y => $train_losses,
                                             name => 'Train',
                                             mode => 'lines'));

$layout = {title => {text => 'Plot of the Training / Testing'},
              xaxis => {title => 'Epoch'}, yaxis => {title => 'Loss'},
              width  => 900, height => 400,
              margin => { l => 50, r => 0, t => 50, b => 50 }};

$plot = new Chart::Plotly::Plot(traces => \@traces,
                                layout => $layout);

IPerl->display($plot); # Cambiar por la función show_plot($plot); si lo ejecutas fuera del entorno Jupyter
```

In [22]:
```
sml->embedplot($plot, width=>900, height=>450); # Este comando es de uso exclusivo del docente
```



15.2.5 Predict

Making predictions with a trained neural network is easy enough. We have already seen how to forward-propagate an input pattern to get an output. This is all we need to do to make a prediction. We can use the output values themselves directly as the probability of a pattern belonging to each output class.

It may be more useful to turn this output back into a crisp class prediction. We can do this by selecting the class value with the larger probability. This is also called the arg max function. Below is a function named predict() that implements this procedure. It returns the index in the network output that has the largest probability. It assumes that class values have been converted to integers starting at 0.

In [23]:
```
# Make a prediction with a network
# Function To Make a Prediction With a Network.

sub predict{
  my ($self, $network, $row) = @_;
  my $outputs = sml->forward_propagate($network, $row);
  return (sort { $outputs->[$b] <=> $outputs->[$a] } 0 .. $#$outputs)[0];
}

sml->add_to_class('predict', \&{'predict'});
```

Out[23]:
```
*sml::predict
```

We can put this together with our code above for forward-propagating input and with our small contrived dataset to test making predictions with an already-trained network. The example hardcodes a network trained from the previous step. The complete example is listed below.

In [24]:
```
# Test making predictions with the network
# Example of Making a Prediction on the Contrived Dataset.

$network = [[{'weights' => [-1.482313569067226, 1.8308790073202204, 1.078381922048799]},
             {'weights' => [0.23244990332399884, 0.3621998343835864, 0.40289821191094327]}],
            [{'weights' => [2.5001872433501404, 0.7887233511355132, -1.1026649757805829]},
             {'weights' => [-2.429350576245497, 0.8357651039198697, 1.0699217181280656]}]];

foreach my $row (@$dataset) {
  my $prediction = sml->predict($network, $row);
  printf "Expected=%s, Got=%s\n", $row->[-1], $prediction;
}
```

```
Expected=0, Got=0
Expected=0, Got=0
Expected=0, Got=0
Expected=0, Got=0
Expected=0, Got=0
Expected=1, Got=1
Expected=1, Got=1
Expected=1, Got=1
Expected=1, Got=1
Expected=1, Got=1
```

Running the example prints the expected output for each record in the training dataset, followed by the crisp prediction made by the network. It shows that the network achieves 100% accuracy on this small dataset.

```
Expected=0, Got=0
Expected=0, Got=0
Expected=0, Got=0
Expected=0, Got=0
Expected=0, Got=0
Expected=1, Got=1
```

Expected=1, Got=1
Expected=1, Got=1
Expected=1, Got=1
Expected=1, Got=1

Now we are ready to apply our backpropagation algorithm to a real world dataset.

15.2.6 Wheat Seeds Case Study

This section applies the Backpropagation algorithm to the wheat seeds dataset. The first step is to load the dataset and convert the loaded data to numbers that we can use in our neural network. For this we will use the helper function load_csv() to load the file, str_column_to_float() to convert string numbers to floats and str_column_to_int() to convert the class column to integer values.

Input values vary in scale and need to be normalized to the range of 0 and 1. It is generally good practice to normalize input values to the range of the chosen transfer function, in this case, the sigmoid function that outputs values between 0 and 1. The dataset_minmax() and normalize_dataset() helper functions were used to normalize the input values.

We will evaluate the algorithm using k-fold cross-validation with 5 folds. This means that 201/5 = 40.2 or 40 records will be in each fold. We will use the helper functions evaluate_algorithm() to evaluate the algorithm with cross-validation and accuracy_metric() to calculate the accuracy of predictions.

A new function named back_propagation() was developed to manage the application of the Backpropagation algorithm, first initializing a network, training it on the training dataset and then using the trained network to make predictions on a test dataset. The complete example is listed below.

In [25]:
```
# Backprop on the Seeds Dataset
# Backpropagation Algorithm on the Wheat Seeds Dataset.

# Backpropagation Algorithm With Stochastic Gradient Descent
sub back_propagation{
  my ($self, $train, $test, %args) = (splice(@_, 0, 3), l_rate   => undef,
                                                        n_epoch  => undef,
                                                        n_hidden => undef, @_);

  my $n_inputs  = scalar(@{$train->[0]}) -1;
  my $n_outputs = scalar(uniq map {$_->[-1]} @$train);
  my $network   = sml->initialize_network($n_inputs, $args{n_hidden}, $n_outputs);
  my ($trained_network, $train_losses, $test_losses) = sml->train_network($network,
                                                                  $train, $args{l_rate},
                                                                  $args{n_epoch},
                                                                  $n_outputs,
                                                                  $test);

  my @predictions = ();
  for my $row (@$test){
    push @predictions, sml->predict($trained_network, $row);
  }
```

```perl
  return (\@predictions, $train_losses, $test_losses);
}

sml->add_to_class('back_propagation', \&{'back_propagation'});

# Test Backprop on Seeds dataset
srand(1);
my $filename = '../data/seeds_dataset.csv';
($dataset, my $header) = sml->load_csv($filename);

# convert string attributes to integers
for my $i (0 .. $#{$dataset->[0]} -1){
  sml->str_column_to_float($dataset, $i);
}
# convert class column to integers
sml->str_column_to_int($dataset, -1);
# normalize input variables
my $minmax = sml->dataset_minmax($dataset);
sml->normalize_dataset($dataset, $minmax);

# Evaluate algorithm
my $n_folds  = 2;
my $l_rate   = 0.3;
my $n_epoch  = 250;
my $n_hidden = 5;

# En el método evaluate_algorithm, asegúrate de pasar los datos de prueba también a train_network
(my $scores, $train_losses, my $test_losses) = sml->evaluate_algorithm($dataset, \&{'sml::back_propagation'},
                                                                        n_folds  => $n_folds,
                                                                        l_rate   => $l_rate,
                                                                        n_epoch  => $n_epoch,
                                                                        n_hidden => $n_hidden,
                                                                        metric   => 'accuracy');
printf "Scores: %s\n", dump($scores);
printf "Mean Accuracy: %.3f%% \n", sum(@$scores) / @$scores;
```

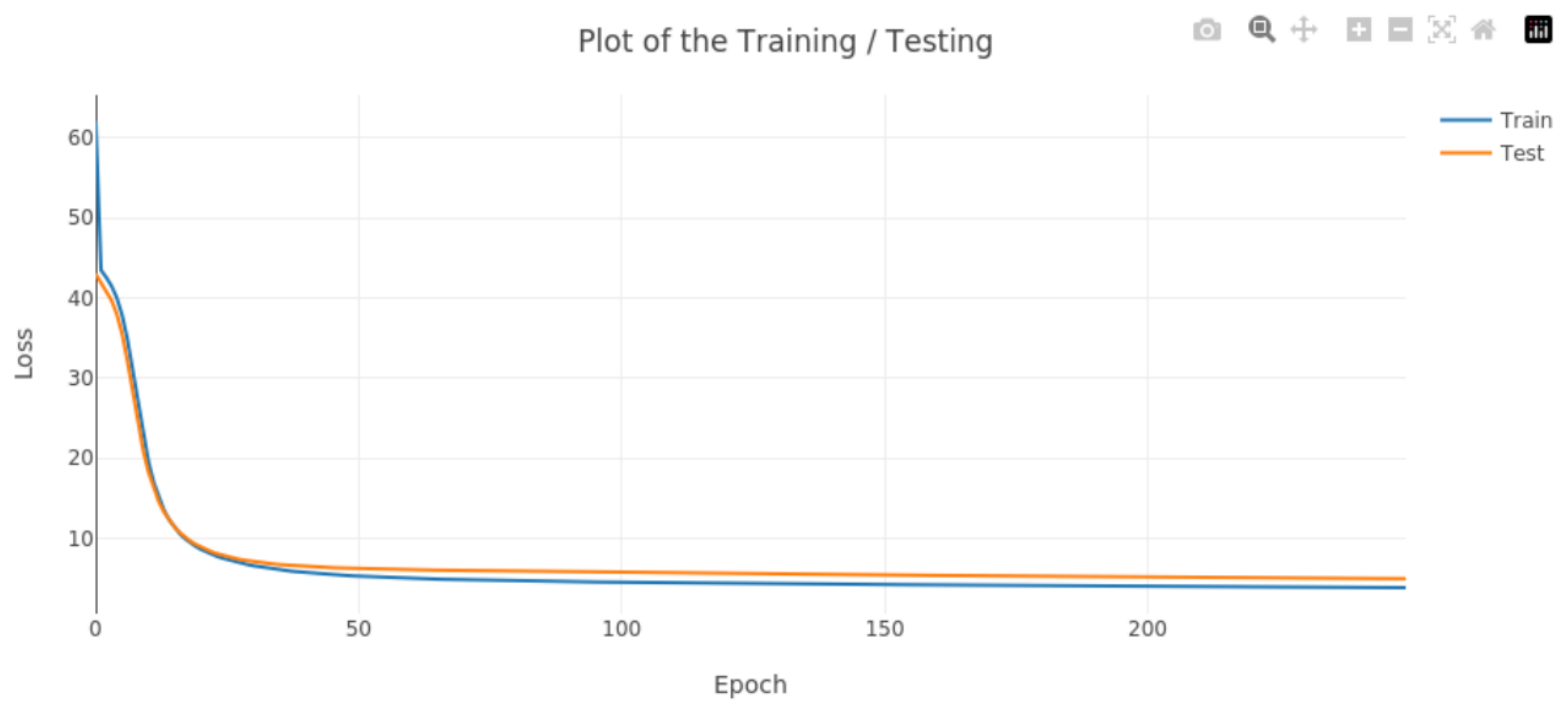
```
>epoch=1, lrate=0.300, error=62.212
>epoch=10, lrate=0.300, error=22.178
>epoch=20, lrate=0.300, error=10.660
>epoch=30, lrate=0.300, error=8.693
>epoch=40, lrate=0.300, error=7.939
>epoch=50, lrate=0.300, error=7.572
>epoch=60, lrate=0.300, error=7.361
>epoch=70, lrate=0.300, error=7.218
>epoch=80, lrate=0.300, error=7.108
>epoch=90, lrate=0.300, error=7.015
>epoch=100, lrate=0.300, error=6.932
>epoch=110, lrate=0.300, error=6.855
>epoch=120, lrate=0.300, error=6.783
>epoch=130, lrate=0.300, error=6.715
>epoch=140, lrate=0.300, error=6.651
>epoch=150, lrate=0.300, error=6.590
>epoch=160, lrate=0.300, error=6.532
>epoch=170, lrate=0.300, error=6.477
>epoch=180, lrate=0.300, error=6.425
>epoch=190, lrate=0.300, error=6.375
>epoch=200, lrate=0.300, error=6.327
>epoch=210, lrate=0.300, error=6.280
>epoch=220, lrate=0.300, error=6.235
>epoch=230, lrate=0.300, error=6.191
>epoch=240, lrate=0.300, error=6.148
>epoch=250, lrate=0.300, error=6.107
>epoch=1, lrate=0.300, error=61.950
>epoch=10, lrate=0.300, error=23.879
>epoch=20, lrate=0.300, error=7.405
>epoch=30, lrate=0.300, error=4.796
>epoch=40, lrate=0.300, error=3.707
>epoch=50, lrate=0.300, error=3.126
>epoch=60, lrate=0.300, error=2.776
>epoch=70, lrate=0.300, error=2.549
>epoch=80, lrate=0.300, error=2.394
>epoch=90, lrate=0.300, error=2.283
>epoch=100, lrate=0.300, error=2.200
>epoch=110, lrate=0.300, error=2.135
>epoch=120, lrate=0.300, error=2.083
>epoch=130, lrate=0.300, error=2.039
>epoch=140, lrate=0.300, error=2.000
>epoch=150, lrate=0.300, error=1.966
>epoch=160, lrate=0.300, error=1.935
>epoch=170, lrate=0.300, error=1.905
>epoch=180, lrate=0.300, error=1.876
>epoch=190, lrate=0.300, error=1.848
>epoch=200, lrate=0.300, error=1.820
```

```
 >epoch=210, lrate=0.300, error=1.791
 >epoch=220, lrate=0.300, error=1.761
 >epoch=230, lrate=0.300, error=1.730
 >epoch=240, lrate=0.300, error=1.698
 >epoch=250, lrate=0.300, error=1.665
Scores: [65.66, 57.58]
Mean Accuracy: 61.620%
```

Out[25]: 1

In [26]:
```
my @train_loss_mean = map {sml->mean($_)} (zip @$train_losses);
my @test_loss_mean  = map {sml->mean($_)} (zip @$test_losses);

@traces = (new Chart::Plotly::Trace::Scatter(x => [0 .. $#train_loss_mean],
                                             y => \@train_loss_mean,
                                             name => 'Train',
                                             mode => 'lines'),
           new Chart::Plotly::Trace::Scatter(x => [0 .. $#test_loss_mean],
                                             y => \@test_loss_mean,
                                             name => 'Test',
                                             mode => 'lines'));

$layout = {title => {text => 'Plot of the Training / Testing'},
              xaxis => {title => 'Epoch'}, yaxis => {title => 'Loss'},
              width  => 900, height => 400,
              margin => { l => 50, r => 0, t => 50, b => 50 }};

$plot = new Chart::Plotly::Plot(traces => \@traces,
                                layout => $layout);

IPerl->display($plot); # Cambiar por la función show_plot($plot); si lo ejecutas fuera del entorno Jupyter
```

In [27]:
```
sml->embedplot($plot, width=>900, height=>450); # Este comando es de uso exclusivo del docente
```



A network with 5 neurons in the hidden layer and 3 neurons in the output layer was constructed. The network was trained for 500 epochs with a learning rate of 0.3. These parameters were found with a little trial and error, but you may be able to do much better. Running the example prints the average classification accuracy on each fold as well as the average performance across all folds.

```
Scores: [95.23809523809523, 97.61904761904762, 95.23809523809523, 92.85714285714286, 95.23809523809523]
Mean Accuracy: 95.238%
```

Example Output from the Backpropagation Algorithm on the Wheat Seeds Dataset.

You can see that backpropagation and the chosen configuration achieved a mean classification accuracy of 95.238% which is dramatically better than the baseline of 28% accuracy.

15.3 Extensions

This section lists extensions to the tutorial that you may wish to explore.

- **Tune Algorithm Parameters**. Try larger or smaller networks trained for longer or shorter. See if you can get better performance on the seeds dataset.
- **Additional Methods**. Experiment with different weight initialization techniques (such as small random numbers) and different transfer functions (such as tanh).

- **More Layers**. Add support for more hidden layers, trained in just the same way as the one hidden layer used in this tutorial.
- **Regression**. Change the network so that there is only one neuron in the output layer and that a real value is predicted. Pick a regression dataset to practice on. A linear transfer function could be used for neurons in the output layer, or the output values of the chosen dataset could be scaled to values between 0 and 1.
- **Batch Gradient Descent**. Change the training procedure from online to batch gradient descent and update the weights only at the end of each epoch.

15.4 Review

In this tutorial, you discovered how to implement the Backpropagation algorithm from scratch. Specifically, you learned:

- How to forward-propagate an input to calculate a network output.
- How to backpropagate error and update network weights.
- How to apply the backpropagation algorithm to a real world dataset.

15.4.1 Further Reading

- Section 18.7. Artificial Neural Networks, page 717, Artificial Intelligence: A Modern Approach, 2010. http://amzn.to/2e3lFqP
- Section 7.1 Neural Networks, page 141 and Section 13.2 Neural Networks, page 333, Applied Predictive Modeling, 2013 http://amzn.to/2e3lNXF
- Chapter 5. Back-Propagation, page 39, Neural Smithing: Supervised Learning in Feedforward Artificial Neural Networks, 1999 http://amzn.to/2fmFWUC

15.4.2 Next

This ends Part 3 on nonlinear algorithms. Next, in Part 4 you will discover ensemble algorithms. In the next tutorial, you will discover how to implement and apply the Bootstrap Aggregation algorithm for classification.