A Curious Learning Task

Suppose we want to fit the following dataset.

In [1]:
```
using Plots
using LinearAlgebra
using Statistics
using Interact

gr()
```

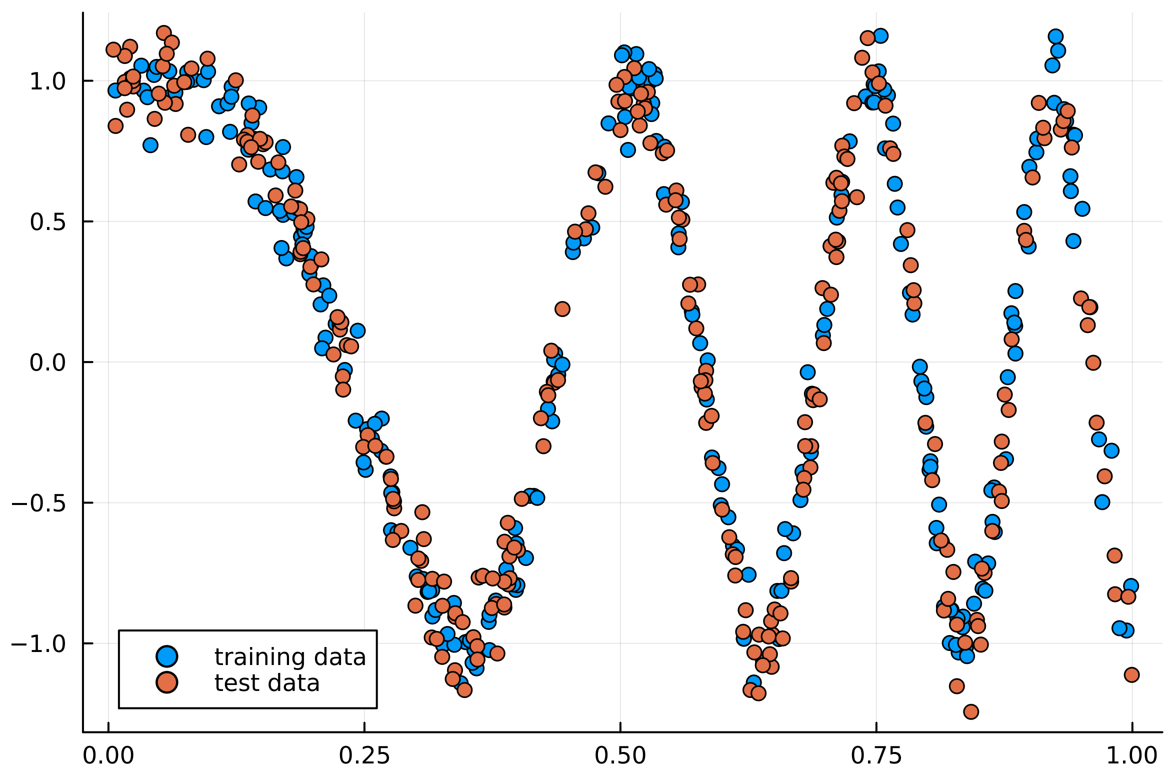
Out[1]: `Plots.GRBackend()`

In [2]:
```
function gen_data(n::Int64)
    xs = rand(n);
    ys = cos.(2*xs .+ 20*xs.^2) .+ 0.1 * randn(n);
    return (xs, ys);
end

n = 256;
(xs, ys) = gen_data(n);
(xs_test, ys_test) = gen_data(n);
```

In [3]:
```
scatter(xs, ys; label="training data");
scatter!(xs_test, ys_test; label="test data")
```

Out[3]:



How will a linear model perform on this task?

Let's look at linear regression.

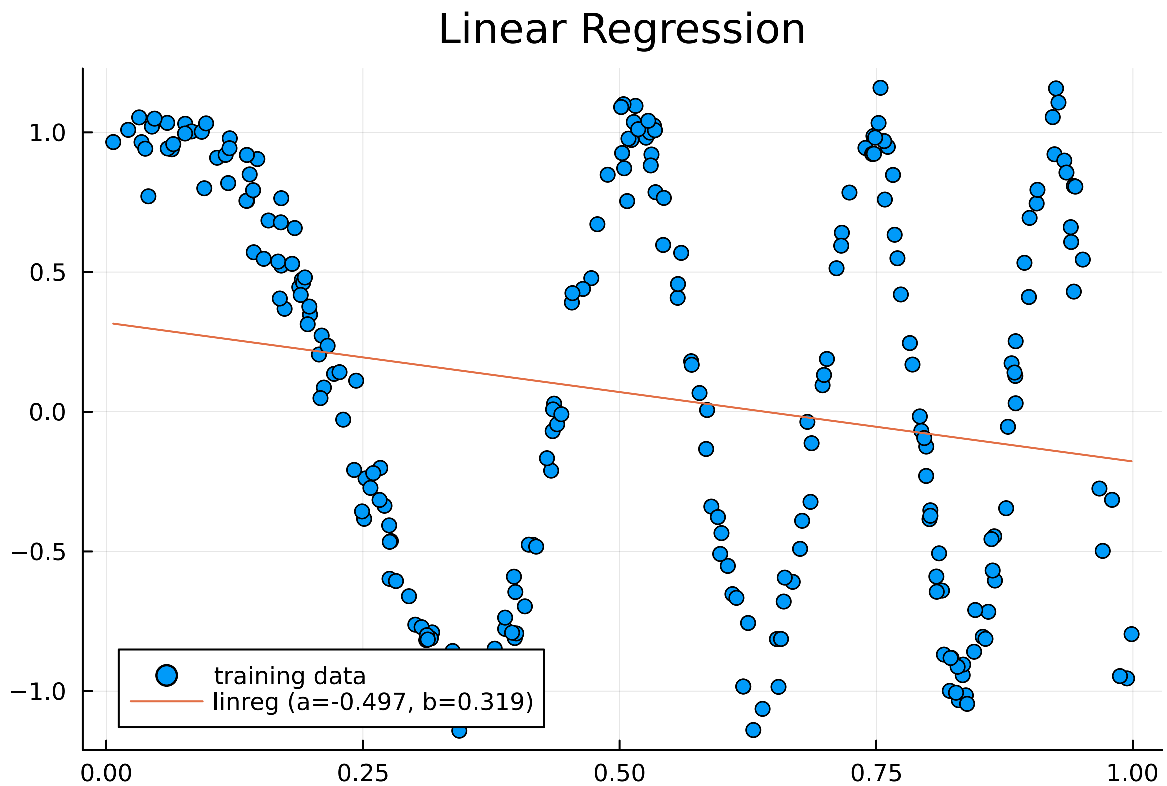
$$\min_{a,b} \sum_{i=1}^{n} (ax_i + b - y_i)^2.$$

In [4]:
```
Xs_homogeneous = vcat(xs', ones(1, n));

(a,b) = inv(Xs_homogeneous * Xs_homogeneous') * Xs_homogeneous * ys;
```

In [5]:
```
scatter(xs, ys; label="training data", title="Linear Regression");
plot!(sort(xs), a * sort(xs) .+ b; label="linreg (a=$(round(a;digits=3)), b
```

Out[5]:



In [6]:
```
# what's the average mean-squared error?

training_loss = mean((a * xs[i] + b - ys[i])^2 for i = 1:n);
test_loss = mean((a * xs_test[i] + b - ys_test[i])^2 for i = 1:n);

println("training loss = $training_loss")
println("test loss = $test_loss")
```

```
training loss = 0.4969151130212958
test loss = 0.49099568808295857
```

Can we do better?

One way to do this is to use a more sophisticated model. One such model is the piecewise linear model.

In [7]:
```
ise_linear_eval(x::Float64, zs::Array{Tuple{Float64,Float64},1})
ength(zs)
>= zs[i][1]) && (x <= zs[i+1][1]))
, y0) = zs[i];
, y1) = zs[i+1];
urn ((x - x0)/(x1 - x0)) * (y1 - y0) + y0;





ise_linear_widget(xs::Array{Float64,1}, ys::Array{Float64,1})
.36, 0.5, 0.62, 0.74, 0.82, 0.95];
rt(rand(6); rev=true);
(0.0:0.01:1.0, label="x1", value=xinits[1]);
(-1.2:0.01:1.2, label="y1", value=yinits[1]);
(0.0:0.01:1.0, label="x2", value=xinits[2]);
(-1.2:0.01:1.2, label="y2", value=yinits[2]);
(0.0:0.01:1.0, label="x3", value=xinits[3]);
(-1.2:0.01:1.2, label="y3", value=yinits[3]);
(0.0:0.01:1.0, label="x4", value=xinits[4]);
(-1.2:0.01:1.2, label="y4", value=yinits[4]);
(0.0:0.01:1.0, label="x5", value=xinits[5]);
(-1.2:0.01:1.2, label="y5", value=yinits[5]);
(0.0:0.01:1.0, label="x6", value=xinits[6]);
(-1.2:0.01:1.2, label="y6", value=yinits[6]);
 Interact.@map sort([(0.0,1.0), (&x1,&y1), (&x2,&y2), (&x3,&y3), (&x4,&y4),
act.@map mean((piecewise_linear_eval(xs[i], &keypoints) - ys[i])^2 for i = 1
act.@map begin
(xs, ys; label="training data", title="Piecewise Linear Model (err=$(round(&
z[1] for z in &keypoints], [z[2] for z in &keypoints]; label="piecewise line

t([
 "x1" => x1, "y1" => y1,
 "x2" => x2, "y2" => y2,
 "x3" => x3, "y3" => y3,
 "x4" => x4, "y4" => y4,
 "x5" => x5, "y5" => y5,
 "x6" => x6, "y6" => y6], output = plt)
g hbox(plt, vbox(hbox(:x1, :y1), hbox(:x2, :y2), hbox(:x3, :y3), hbox(:x4, :
```

Out[7]:
```
piecewise_linear_widget (generic function with 1 method)
```

In [8]:
```
piecewise_linear_widget(xs, ys)
```

This error is MUCH lower than what we got from the linear regression model!

But how can we learn this?

Problem: the way we have parameterized this model is not continuous!

To solve this problem, note that we can always represent a piecewise linear function as a sum of shifted and scaled ReLU functions. The ReLU function (**RE**ctified **L**inear **U**nit) is defined as

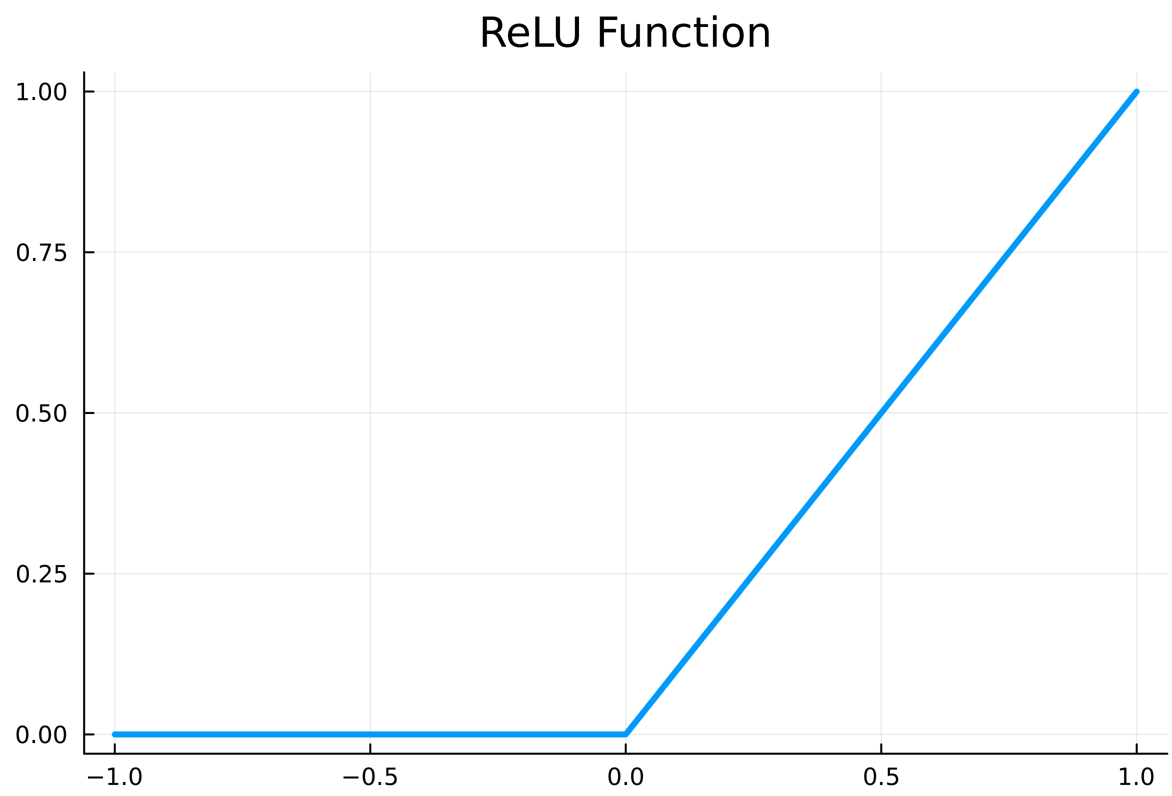
$$\mathrm{ReLU}(x) = \begin{cases} x & \text{if } x \geq 0 \\ 0 & \text{if } x \leq 0 \end{cases} = \max(x, 0).$$

We can visualize this as follows.

In [9]:
```
function ReLU(x::Float64)
    return max(x, 0);
end

us = collect(-1.0:0.01:1.0);
plot(us, ReLU.(us); linewidth=3, label="", title="ReLU Function")
```

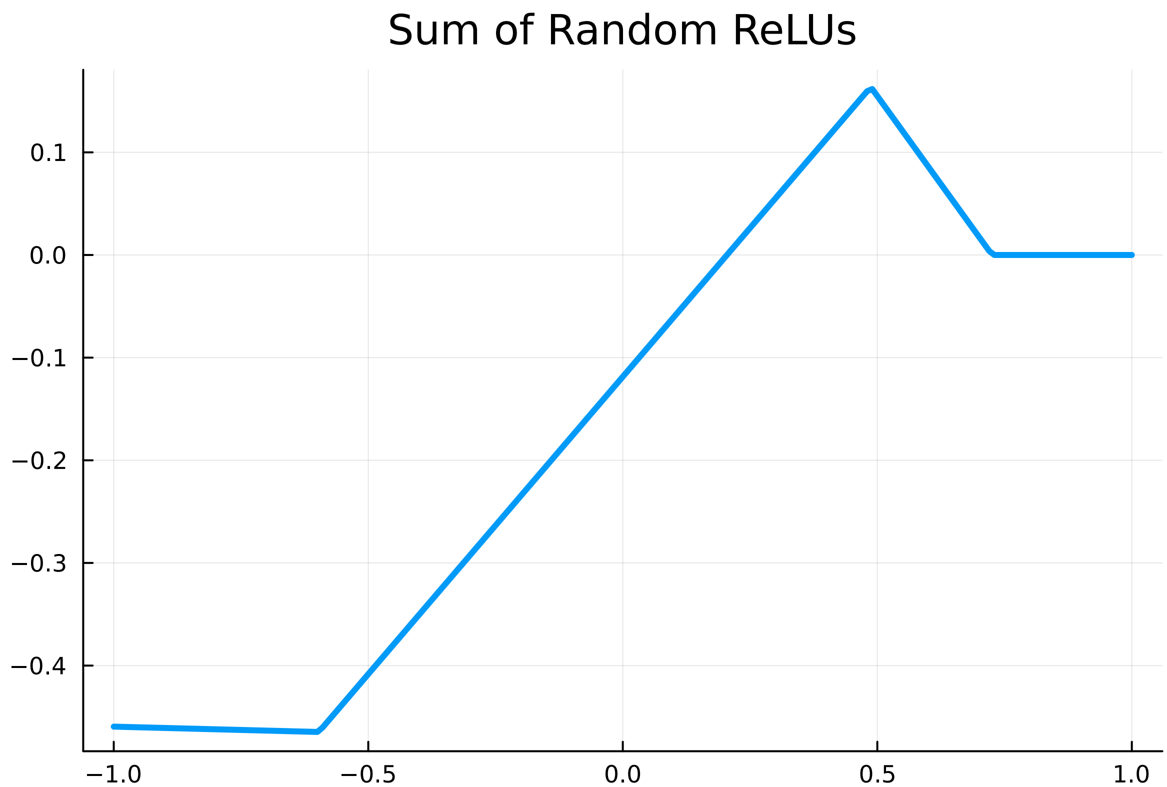
Out[9]:



In [10]:
```
# to illustrate, here's a sum of some randomly shifted and scaled relus

us = collect(-1.0:0.01:1.0);
plot(us, sum(rand([-1.0,1.0]) * ReLU.(randn() * us .+ randn()) for i = 1:5)
```

Out[10]:



We can try to parameterize our model as the sum of ReLU functions as follows.

$$h_{a,b,w}(x) = w_1 \cdot \mathrm{ReLU}(a_1 \cdot x + b_1) + w_2 \cdot \mathrm{ReLU}(a_2 \cdot x + b_2) + \cdots = \sum_{i=1}^{d} w_i \cdot \mathrm{ReLU}(a_i \cdot x + $$

This is guaranteed to be continuous in the parameters $a, b, w \in \mathbb{R}^d$. (Why?)

We can train this using SGD. To compute the gradient with respect to a loss function, observe that

$$\frac{\partial}{\partial w_i} \frac{1}{2} \left(h_{a,b,w}(x) - y \right)^2 = \left(h_{a,b,w}(x) - y \right) \cdot \mathrm{ReLU}(a_i \cdot x + b_i)$$

$$\frac{\partial}{\partial a_i} \frac{1}{2} \left(h_{a,b,w}(x) - y \right)^2 = \left(h_{a,b,w}(x) - y \right) \cdot w_i \cdot \mathrm{ReLU}'(a_i \cdot x + b_i) \cdot x$$

$$\frac{\partial}{\partial b_i} \frac{1}{2} \left(h_{a,b,w}(x) - y \right)^2 = \left(h_{a,b,w}(x) - y \right) \cdot w_i \cdot \mathrm{ReLU}'(a_i \cdot x + b_i)$$

In [11]:

```
function dReLU(x::Float64)
    return (x > 0.0) ? 1.0 : 0.0;
end

function heval(x::Float64, a::Array{Float64,1}, b::Array{Float64,1}, w::Arr
    d = length(a);
    @assert(length(b) == d);
    @assert(length(w) == d);

    return sum(w .* ReLU.(a .* x .+ b));
end

function grad(x::Float64, y::Float64, a::Array{Float64,1}, b::Array{Float64
    d = length(a);
    @assert(length(b) == d);
    @assert(length(w) == d);

    axb = a .* x .+ b;
    dh = sum(w .* ReLU.(axb)) - y;

    dw = dh .* ReLU.(axb);
    da = dh .* w .* dReLU.(axb) .* x;
    db = dh .* w .* dReLU.(axb);

    return [da,db,dw];
end
```

Out[11]:

```
grad (generic function with 1 method)
```

In [12]:

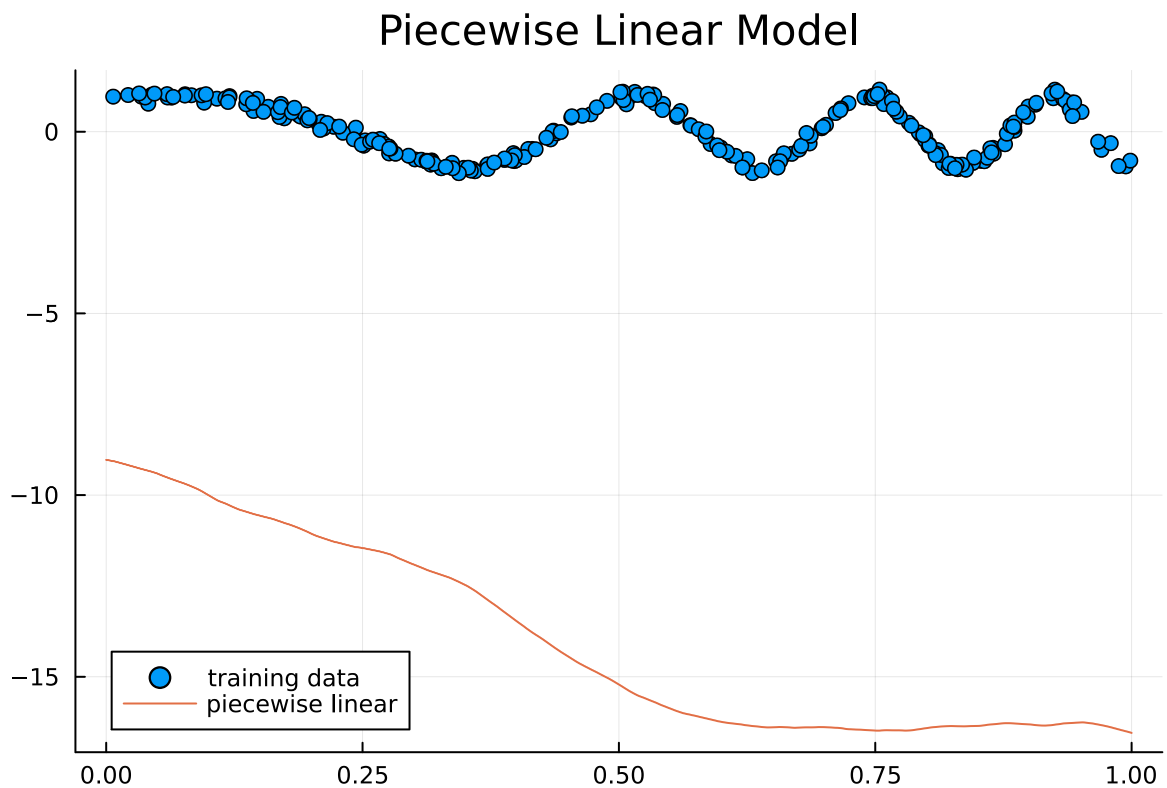
```
d = 1024;
a = rand([1.0,-1.0],d);
b = 2 * rand(d) .- 1;
w = randn(d);

alpha = 0.0001;
```

In [13]:
```
# what does the model look like when we initialize?

us = collect(0.0:0.001:1.0);
scatter(xs, ys; label="training data", title="Piecewise Linear Model");
plot!(us, [heval(u,a,b,w) for u in us]; label="piecewise linear")
```

Out[13]:

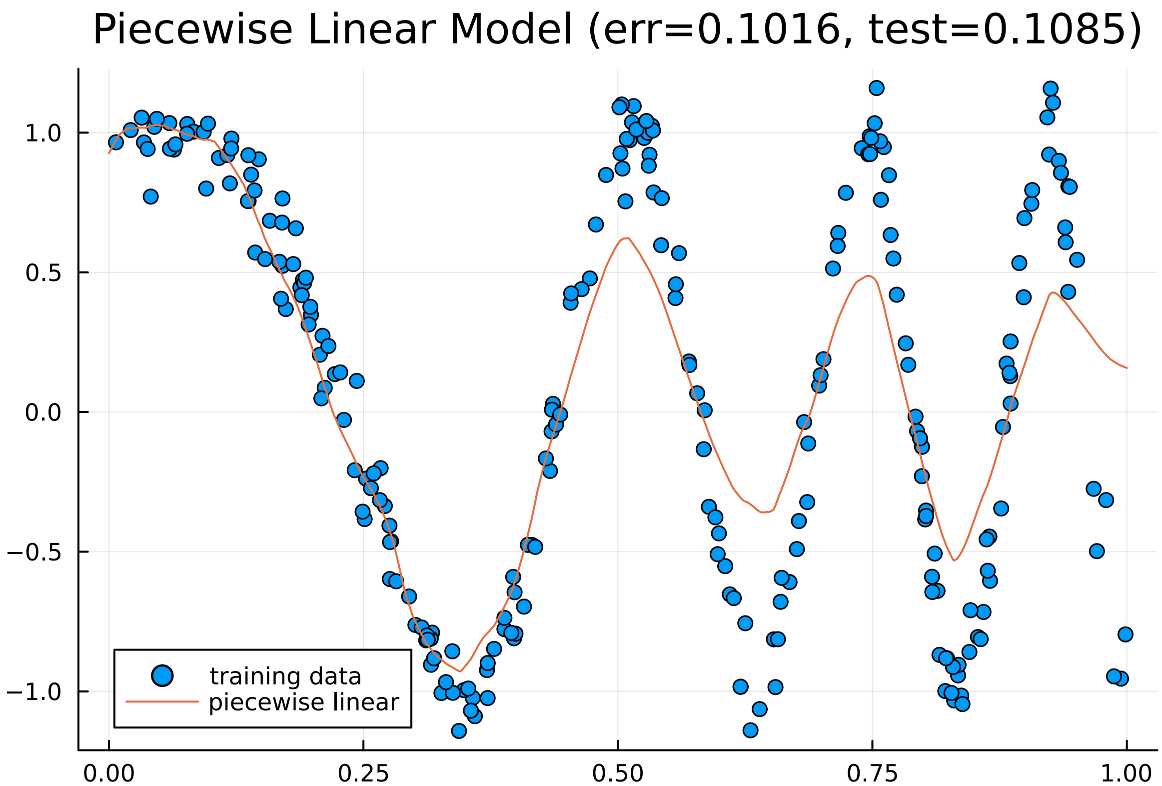


In [14]:
```
# let's train this with SGD

for k = 1:10000
    i = rand(1:n)
    (a, b, w) = (a, b, w) .- alpha .* grad(xs[i], ys[i], a, b, w);
end

err = mean((heval(xs[i],a,b,w)-ys[i])^2 for i = 1:n);
test_err = mean((heval(xs_test[i],a,b,w)-ys_test[i])^2 for i = 1:n);
us = collect(0.0:0.001:1.0);
scatter(xs, ys; label="training data", title="Piecewise Linear Model (err=$
plot!(us, [heval(u,a,b,w) for u in us]; label="piecewise linear")
```

Out[14]:

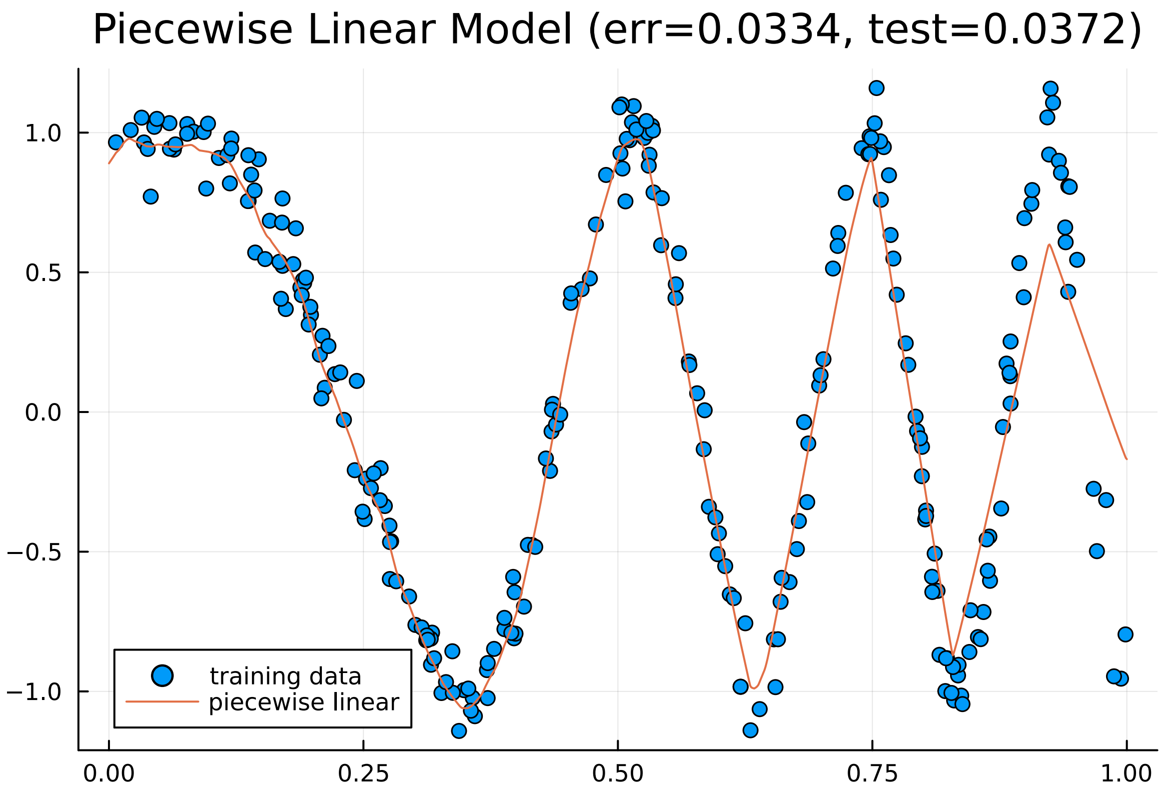


In [15]:
```
# train for many more iterations

for k = 1:100000
    i = rand(1:n)
    (a, b, w) = (a, b, w) .- alpha .* grad(xs[i], ys[i], a, b, w);
end

err = mean((heval(xs[i],a,b,w)-ys[i])^2 for i = 1:n);
test_err = mean((heval(xs_test[i],a,b,w)-ys_test[i])^2 for i = 1:n);
us = collect(0.0:0.001:1.0);
scatter(xs, ys; label="training data", title="Piecewise Linear Model (err=$
plot!(us, [heval(u,a,b,w) for u in us]; label="piecewise linear")
```

Out[15]:



The model did a pretty good job!

Okay but hold on...what does this have to do with deep neural networks? Well, it turns out the model we just trained **is** a neural network! It's a particular type of DNN that uses "ReLU activations."

- One way to think of a ReLU neural network is as a piecewise linear model.

Deep Neural Networks more generally

In machine learning we usually want to make predictions not just from a single-dimensional input but from high-dimensional input feature vectors.

How can we represent a piecewise linear function from a vector space to a vector space?

ReLUs let us do this easily. For example, the function

$$f(x) = W \cdot \mathrm{ReLU}(Ax + b)$$

where the ReLU function is now taken elementwise on the vector $Ax + b$, is piecewise linear.

In [ ]:

In [ ]: