Watson Studio democratizes machine learning and deep learning to accelerate infusion of AI in your business to drive innovation. Watson Studio provides a suite of tools and a collaborative environment for data scientists, developers and domain experts.

(http://cocl.us/pytorch_link_top)

Momentum

Table of Contents

In this lab, you will deal with several problems associated with optimization and see how momentum can improve your results.

Estimated Time Needed: **25 min**

Preparation

Import the following libraries that you'll use for this lab:

In [1]:

```
# These are the libraries that will be used for this lab.

import torch
import torch.nn as nn
import matplotlib.pylab as plt
import numpy as np

torch.manual_seed(0)
```

Out[1]:

```
<torch._C.Generator at 0x7f0b54067190>
```

This function will plot a cubic function and the parameter values obtained via Gradient Descent.

In [2]:

```
# Plot the cubic

def plot_cubic(w, optimizer):
    LOSS = []
    # parameter values
    W = torch.arange(-4, 4, 0.1)
    # plot the loss fuction
    for w.state_dict()['linear.weight'][0] in W:
        LOSS.append(cubic(w(torch.tensor([[1.0]]))).item())
    w.state_dict()['linear.weight'][0] = 4.0
    n_epochs = 10
    parameter = []
    loss_list = []

    # n_epochs
    # Use PyTorch custom module to implement a ploynomial function
    for n in range(n_epochs):
        optimizer.zero_grad()
        loss = cubic(w(torch.tensor([[1.0]])))
        loss_list.append(loss)
        parameter.append(w.state_dict()['linear.weight'][0].detach().data.item())
        loss.backward()
        optimizer.step()
    plt.plot(parameter, loss_list, 'ro', label='parameter values')
    plt.plot(W.numpy(), LOSS, label='objective function')
    plt.xlabel('w')
    plt.ylabel('l(w)')
    plt.legend()
```

This function will plot a 4th order function and the parameter values obtained via Gradient Descent. You can also add Gaussian noise with a standard deviation determined by the parameter `std` .

In [3]:

```
# Plot the fourth order function and the parameter values

def plot_fourth_order(w, optimizer, std=0, color='r', paramlabel='parameter values'
, objfun=True):
    W = torch.arange(-4, 6, 0.1)
    LOSS = []
    for w.state_dict()['linear.weight'][0] in W:
        LOSS.append(fourth_order(w(torch.tensor([[1.0]]))).item())
    w.state_dict()['linear.weight'][0] = 6
    n_epochs = 100
    parameter = []
    loss_list = []

    #n_epochs
    for n in range(n_epochs):
        optimizer.zero_grad()
        loss = fourth_order(w(torch.tensor([[1.0]]))) + std * torch.randn(1, 1)
        loss_list.append(loss)
        parameter.append(w.state_dict()['linear.weight'][0].detach().data.item())
        loss.backward()
        optimizer.step()

    # Plotting
    if objfun:
        plt.plot(W.numpy(), LOSS, label='objective function')
    plt.plot(parameter, loss_list, 'ro',label=paramlabel, color=color)
    plt.xlabel('w')
    plt.ylabel('l(w)')
    plt.legend()
```

This is a custom module. It will behave like a single parameter value. We do it this way so we can use PyTorch's build-in optimizers .

In [4]:

```
# Create a linear model

class one_param(nn.Module):

    # Constructor
    def __init__(self, input_size, output_size):
        super(one_param, self).__init__()
        self.linear = nn.Linear(input_size, output_size, bias=False)

    # Prediction
    def forward(self, x):
        yhat = self.linear(x)
        return yhat
```

We create an object `w` , when we call the object with an input of one, it will behave like an individual parameter value. i.e `w(1)` is analogous to w

In [5]:

```
# Create a one_param object

w = one_param(1, 1)
```

Saddle Points

Let's create a cubic function with Saddle points

In [6]:

```
# Define a function to output a cubic

def cubic(yhat):
    out = yhat ** 3
    return out
```

We create an optimizer with no momentum term

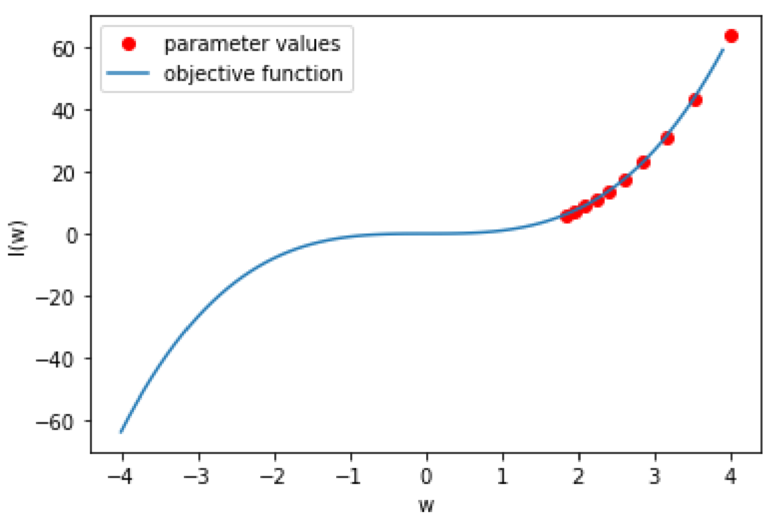
In [7]:

```
# Create a optimizer without momentum

optimizer = torch.optim.SGD(w.parameters(), lr=0.01, momentum=0)
```

We run several iterations of stochastic gradient descent and plot the results. We see the parameter values get stuck in the saddle point.

In [8]:

```
# Plot the model

plot_cubic(w, optimizer)
```



we create an optimizer with momentum term of 0.9

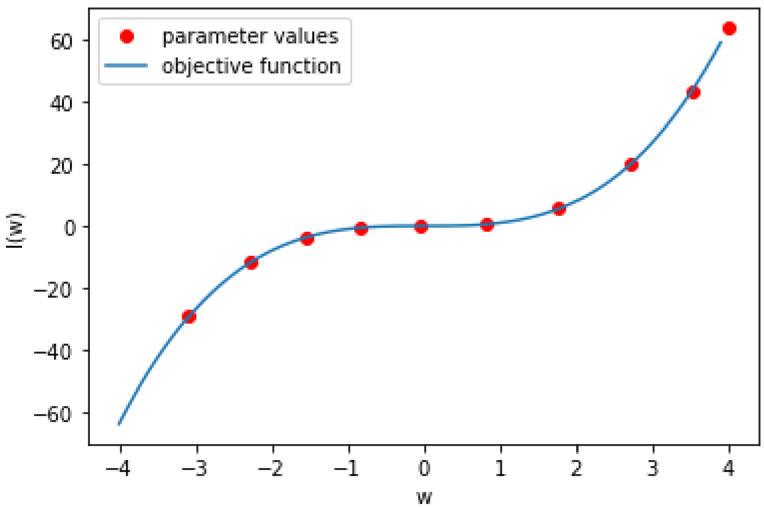
In [9]:

```
# Create a optimizer with momentum

optimizer = torch.optim.SGD(w.parameters(), lr=0.01, momentum=0.9)
```

We run several iterations of stochastic gradient descent with momentum and plot the results. We see the parameter values do not get stuck in the saddle point.

In [10]:

```
# Plot the model

plot_cubic(w, optimizer)
```



Local Minima

In this section, we will create a fourth order polynomial with a local minimum at *4* and a global minimum a *-2*. We will then see how the momentum parameter affects convergence to a global minimum. The fourth order polynomial is given by:

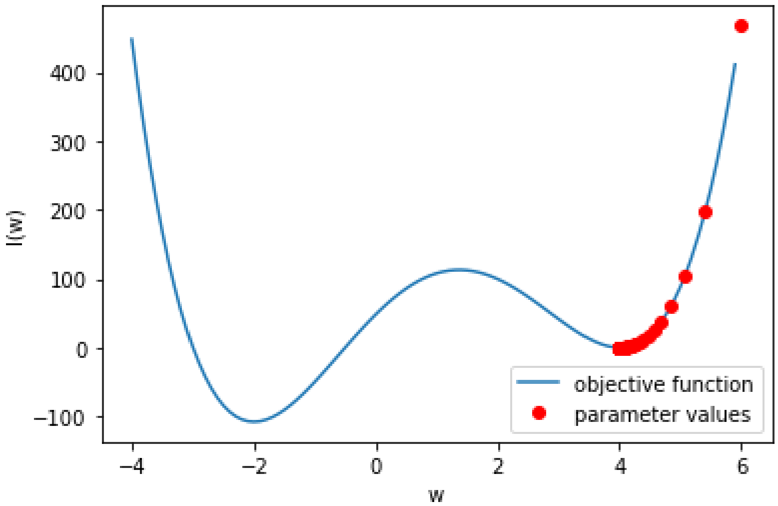
In [11]:

```
# Create a function to calculate the fourth order polynomial

def fourth_order(yhat):
    out = torch.mean(2 * (yhat ** 4) - 9 * (yhat ** 3) - 21 * (yhat ** 2) + 88 * yh
at + 48)
    return out
```

We create an optimizer with no momentum term. We run several iterations of stochastic gradient descent and plot the results. We see the parameter values get stuck in the local minimum.

In [12]:

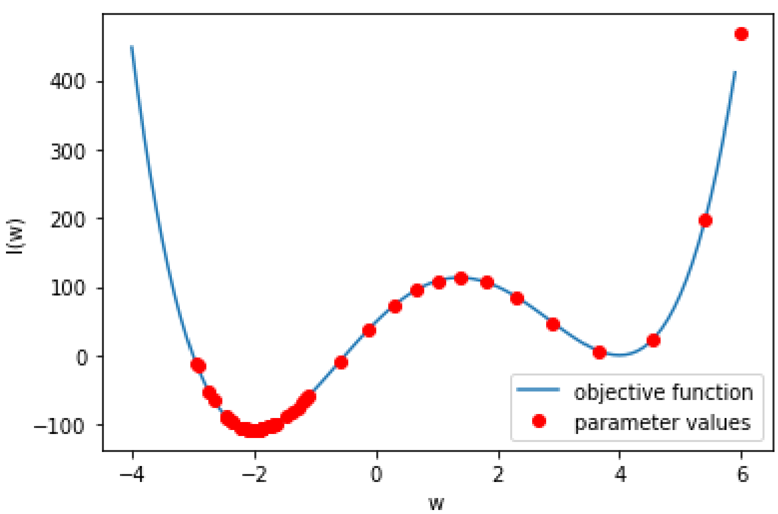
```
# Make the prediction without momentum

optimizer = torch.optim.SGD(w.parameters(), lr=0.001)
plot_fourth_order(w, optimizer)
```



We create an optimizer with a momentum term of 0.9. We run several iterations of stochastic gradient descent and plot the results. We see the parameter values reach a global minimum.

In [13]:

```
# Make the prediction with momentum

optimizer = torch.optim.SGD(w.parameters(), lr=0.001, momentum=0.9)
plot_fourth_order(w, optimizer)
```



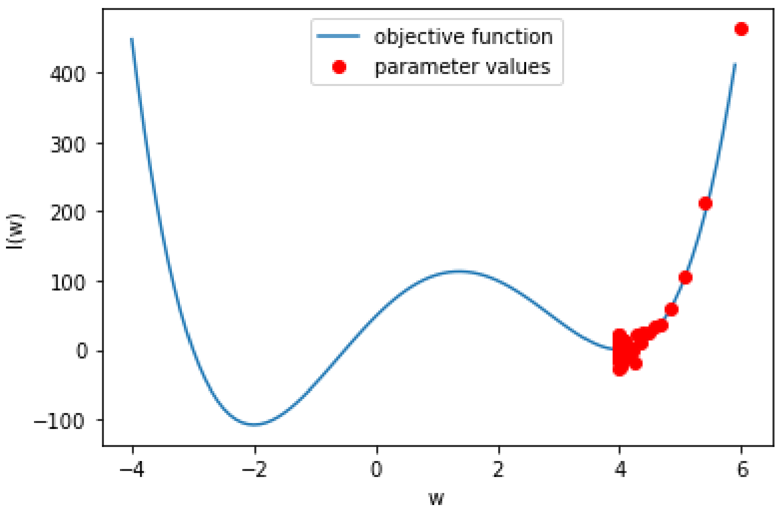
Noise

In this section, we will create a fourth order polynomial with a local minimum at 4 and a global minimum a -2, but we will add noise to the function when the Gradient is calculated. We will then see how the momentum parameter affects convergence to a global minimum.

with no momentum, we get stuck in a local minimum

In [14]:

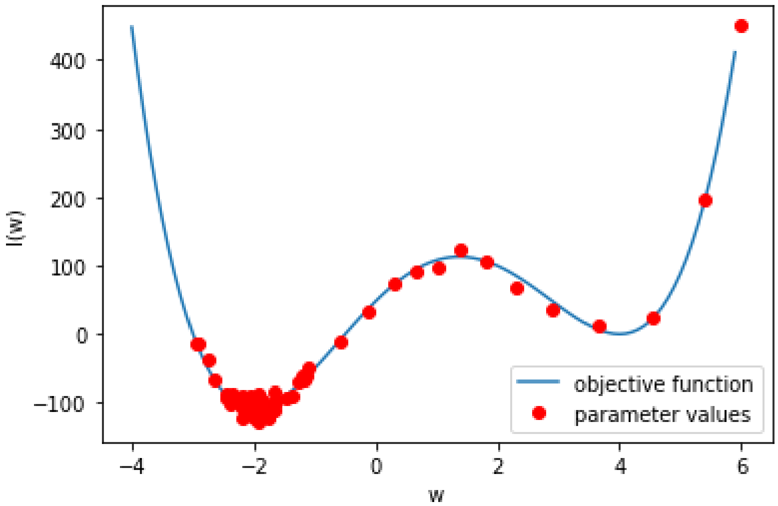
```
# Make the prediction without momentum when there is noise

optimizer = torch.optim.SGD(w.parameters(), lr=0.001)
plot_fourth_order(w, optimizer, std=10)
```



with momentum, we get to the global minimum

In [15]:

```
# Make the prediction with momentum when there is noise

optimizer = torch.optim.SGD(w.parameters(), lr=0.001,momentum=0.9)
plot_fourth_order(w, optimizer, std=10)
```

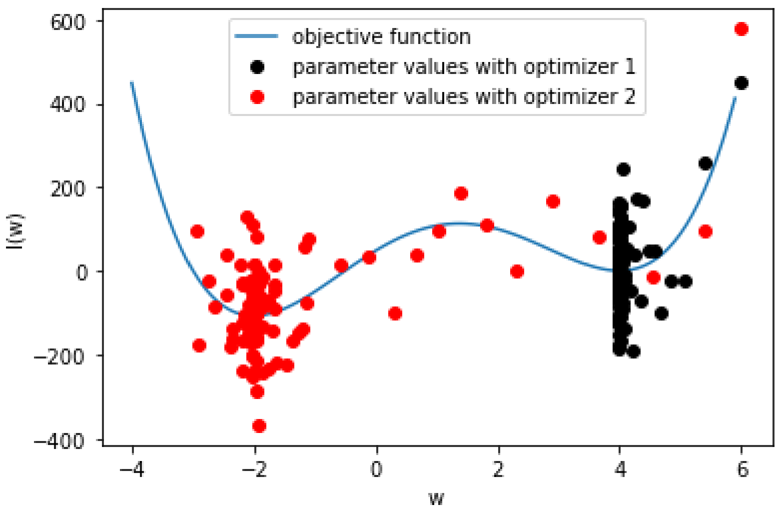


Practice

Create two SGD objects with a learning rate of 0.001. Use the default momentum parameter value for one and a value of 0.9 for the second. Use the function `plot_fourth_order` with an `std=100`, to plot the different steps of each. Make sure you run the function on two independent cells.

In [17]:

```
# Practice: Create two SGD optimizer with lr = 0.001, and one without momentum and
 the other with momentum = 0.9. Plot the result out.

optimizer1 = torch.optim.SGD(w.parameters(), lr = 0.001)
plot_fourth_order(w, optimizer1, std = 100, color = 'black', paramlabel = 'paramete
r values with optimizer 1')

optimizer2 = torch.optim.SGD(w.parameters(), lr = 0.001, momentum = 0.9)
plot_fourth_order(w, optimizer2, std = 100, color = 'red', paramlabel = 'parameter
 values with optimizer 2', objfun = False)
```



Double-click **here** for the solution.

About the Authors:

Joseph Santarcangelo (https://www.linkedin.com/in/joseph-s-50398b136/) has a PhD in Electrical Engineering, his research focused on using machine learning, signal processing, and computer vision to determine how videos impact human cognition. Joseph has been working for IBM since he completed his PhD.

Other contributors: Michelle Carey (https://www.linkedin.com/in/michelleccarey/), Mavis Zhou (www.linkedin.com/in/jiahui-mavis-zhou-a4537814a)
