Momento

Objectivos

Aprender sobre puntos de silla, mínimos locales y ruido

Tabla de contenido

Preparación

In [1]:
```
import os
os.environ['KMP_DUPLICATE_LIB_OK']='True'
```

In [2]:
```
# These are the libraries that will be used for this lab.

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

torch.manual_seed(0)
```

Out[2]:
```
<torch._C.Generator at 0x1fef2fb84270>
```

La siguiente función graficará una función cúbica y los parámetros obtenidos vía gradiente descendente

In [3]:
```
# 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()
```

La siguiente función graficará una función de cuarto orden y los parámetros obtenidos vía el gradiente descendente. También puede agregarle ruido gaussiano con desviación estándar determinada por el parámetro std.

In [4]:
```
# Plot the fourth order function and the parameter values

def plot_fourth_order(w, optimizer, std=0, color='r', paramlabel='parameter values', c
    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 n0fig==1:
        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()
```

Módulo personalizado. Se comportará como un valor de parámetro único. Lo hacemos de esta forma para poder usar los optimizadores incorporados de PyTorch.

In [5]:
```
# 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
```

Creamos un objeto w, cuando lo llamemos con una entrada de 1, se comportará como un valor de parámetro individual. Es decir, w(1) es análogo a w

In [6]:
```
# Create a one_param object

w = one_param(1, 1)
```

Puntos de silla

Creamos una función cúbica que tiene puntos de silla:

In [7]:
```
# Define a function to output a cubic

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

Creamos un optimizador sin momentum:

In [8]:
```
# Create a optimizer without momentum

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

Ejecutamos varias iteraciones del gradiente descendente estocástico y graficamos. Vemos que los valores de los parámetros quedan atascados en el punto de silla.

In [9]:
```
# Plot the model

plot_cubic(w, optimizer)
```

Creamos un optimizador con momento igual a 0.9:

In [10]:
```
# Create a optimizer with momentum

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

Ejecutamos varias iteraciones del gradiente descendente estocástico con momento y graficamos. Vemos que ahora los valores de los parámetros NO se atascan en el punto de silla.

In [11]:
```
# Plot the model

plot_cubic(w, optimizer)
```

Mínimos locales

Crearemos un polinomio de cuarto grado que tiene un mínimo local en 4 y uno global en -2. Veremos que el parámetro de momento afecta la convergencia al mínimo global. El polinomio está dado por:

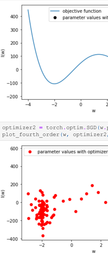
In [12]:
```
# Create a function to calculate the fourth order polynomial

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

Creamos un optimizador sin momento. Ejecutamos varias iteraciones del gradiente descendente estocástico y graficamos. Vemos que los valores de los parámetros quedan atascados en el mínimo local.

In [13]:
```
# Make the prediction without momentum

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



Creamos un optimizador con momento de 0.9. Ejecutamos varias iteraciones del gradiente descendente estocástico y graficamos. Vemos que los valores de los parámetros alcanzan el mínimo global.

In [14]:
```
# Make the prediction with momentum

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

Ruido

Al polinomio del ejemplo anterior le agregaremos ruido cuando se calcula el gradiente. Veremos cómo el parámetro de momento afecta la convergencia al mínimo global.

Sin momento, nos atascamos en el mínimo local:

In [15]:
```
# 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)
```

Con momento alcanzamos el mínimo global:

In [16]:
```
# 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)
```


Práctica

Crear 2 objetos SGD con tasa de aprendizaje 0.001. Use el momento por defecto para el primero y 0.9 para el segundo. Use la función `plot_fourth_order` con std= 100 para graficar diferentes pasos en cada uno.

In [17]:
```
# Practice: Create two SGD optimizer with lr = 0.001, and one without momentum and the

#
optimizer1 = torch.optim.SGD(w.parameters(), lr = 0.001)
plot_fourth_order(w, optimizer1, std = 100, color = 'black', paramlabel = 'parameter
```


In [18]:
```
optimizer2 = torch.optim.SGD(w.parameters(), lr = 0.001, momentum = 0.9)
plot_fourth_order(w, optimizer2, std = 100, color = 'red', paramlabel = 'parameter va
```