California State University, Long Beach
Department of Computer Engineering and Computer Science
CECS 553 Sec 02 11792 (Machine Vision) – Fall 2022
Assignment 06 – Tuesday, 10/18/2022

Logistic Regression With Mini-Batch Gradient Descent

Objective

- Represent your data as a Dataset object
- Create a Logistic Regression Model using PyTorch
- Set a Criterion to calculate Loss
- Create a Data Loader and set the Batch Size
- Create an Optimizer to update Model Parameters and set Learning Rate
- Train a Model

Table of Contents

In this lab, you will learn how to train a PyTorch Logistic Regression model using Mini-Batch Gradient Descent.

- Load Data
- Create the Model and Total Loss Function (Cost)
- Setting the Batch Size using a Data Loader
- Setting the Learning Rate
- Train the Model via Mini-Batch Gradient Descent
- Question

Estimated Time Needed: **30 min**

Preparation

We'll need the following libraries:

In [ ]:
```
# Import the libraries we need for this lab

# Allows us to use arrays to manipulate and store data
import numpy as np
# Used to graph data and loss curves
import matplotlib.pyplot as plt
from mpl_toolkits import mplot3d
# PyTorch Library
import torch
# Used to help create the dataset and perform mini-batch
from torch.utils.data import Dataset, DataLoader
# PyTorch Neural Network
import torch.nn as nn
```

The class `plot_error_surfaces` is just to help you visualize the data space and the parameter space during training and has nothing to do with Pytorch.

In [ ]:
```
# Create class for plotting and the function for plotting

class plot_error_surfaces(object):

    # Construstor
    def __init__(self, w_range, b_range, X, Y, n_samples = 30, go = True):
        W = np.linspace(-w_range, w_range, n_samples)
        B = np.linspace(-b_range, b_range, n_samples)
        w, b = np.meshgrid(W, B)
        Z = np.zeros((30, 30))
        count1 = 0
        self.y = Y.numpy()
        self.x = X.numpy()
        for w1, b1 in zip(w, b):
            count2 = 0
            for w2, b2 in zip(w1, b1):
                yhat= 1 / (1 + np.exp(-1*(w2*self.x+b2)))
                Z[count1,count2]=-1*np.mean(self.y*np.log(yhat+1e-16) +(1-se
                count2 += 1
            count1 += 1
        self.Z = Z
        self.w = w
        self.b = b
        self.W = []
        self.B = []
        self.LOSS = []
        self.n = 0
        if go == True:
            plt.figure()
            plt.figure(figsize=(7.5, 5))
            plt.axes(projection='3d').plot_surface(self.w, self.b, self.Z, r
```

```
            plt.title('Loss Surface')
            plt.xlabel('w')
            plt.ylabel('b')
            plt.show()
            plt.figure()
            plt.title('Loss Surface Contour')
            plt.xlabel('w')
            plt.ylabel('b')
            plt.contour(self.w, self.b, self.Z)
            plt.show()

     # Setter
    def set_para_loss(self, model, loss):
        self.n = self.n + 1
        self.W.append(list(model.parameters())[0].item())
        self.B.append(list(model.parameters())[1].item())
        self.LOSS.append(loss)

    # Plot diagram
    def final_plot(self):
        ax = plt.axes(projection='3d')
        ax.plot_wireframe(self.w, self.b, self.Z)
        ax.scatter(self.W, self.B, self.LOSS, c='r', marker='x', s=200, alph
        plt.figure()
        plt.contour(self.w, self.b, self.Z)
        plt.scatter(self.W, self.B, c='r', marker='x')
        plt.xlabel('w')
        plt.ylabel('b')
        plt.show()

    # Plot diagram
    def plot_ps(self):
        plt.subplot(121)
        plt.ylim
        plt.plot(self.x[self.y==0], self.y[self.y==0], 'ro', label="training
        plt.plot(self.x[self.y==1], self.y[self.y==1]-1, 'o', label="trainir
        plt.plot(self.x, self.W[-1] * self.x + self.B[-1], label="estimated
        plt.xlabel('x')
        plt.ylabel('y')
        plt.ylim((-0.1, 2))
        plt.title('Data Space Iteration: ' + str(self.n))
        plt.show()
        plt.subplot(122)
        plt.contour(self.w, self.b, self.Z)
        plt.scatter(self.W, self.B, c='r', marker='x')
        plt.title('Loss Surface Contour Iteration' + str(self.n))
        plt.xlabel('w')
        plt.ylabel('b')

# Plot the diagram
```

```
def PlotStuff(X, Y, model, epoch, leg=True):

    plt.plot(X.numpy(), model(X).detach().numpy(), label=('epoch ' + str(epc
    plt.plot(X.numpy(), Y.numpy(), 'r')
    if leg == True:
        plt.legend()
    else:
        pass
```

Set the random seed:

In [ ]:
```
# Setting the seed will allow us to control randomness and give us reproduci
torch.manual_seed(0)
```

Load Data

The Dataset class represents a dataset. Your custom dataset should inherit Dataset which we imported above and override the following methods:

`__len__` so that len(dataset) returns the size of the dataset.

`__getitem__` to support the indexing such that dataset[i] can be used to get ith sample

Below we will create a sample dataset

In [ ]:
```
# Create the custom Data class which inherits Dataset
class Data(Dataset):

    # Constructor
    def __init__(self):
        # Create X values from -1 to 1 with step .1
        self.x = torch.arange(-1, 1, 0.1).view(-1, 1)
        # Create Y values all set to 0
        self.y = torch.zeros(self.x.shape[0], 1)
        # Set the X values above 0.2 to 1
        self.y[self.x[:, 0] > 0.2] = 1
        # Set the .len attribute because we need to override the __len__ met
        self.len = self.x.shape[0]

    # Getter that returns the data at the given index
    def __getitem__(self, index):
        return self.x[index], self.y[index]

    # Get length of the dataset
    def __len__(self):
        return self.len
```

Make `Data` object

In [ ]:
```
# Create Data object
data_set = Data()
```

We can see the X values of the dataset

In [ ]:
```
data_set.x
```

We can see the Y values of the dataset which correspond to the class of the X value

In [ ]:
```
data_set.y
```

We can get the length of the dataset

In [ ]:
```
len(data_set)
```

We can get the label y as well as the x for the first sample

In [ ]:
```
x,y = data_set[0]
print("x = {},  y = {}".format(x,y))
```

We can get the label y as well as the x for the second sample:

In [ ]:
```
x,y = data_set[1]
print("x = {},  y = {}".format(x,y))
```

We can see we can separate the one-dimensional dataset into two classes:

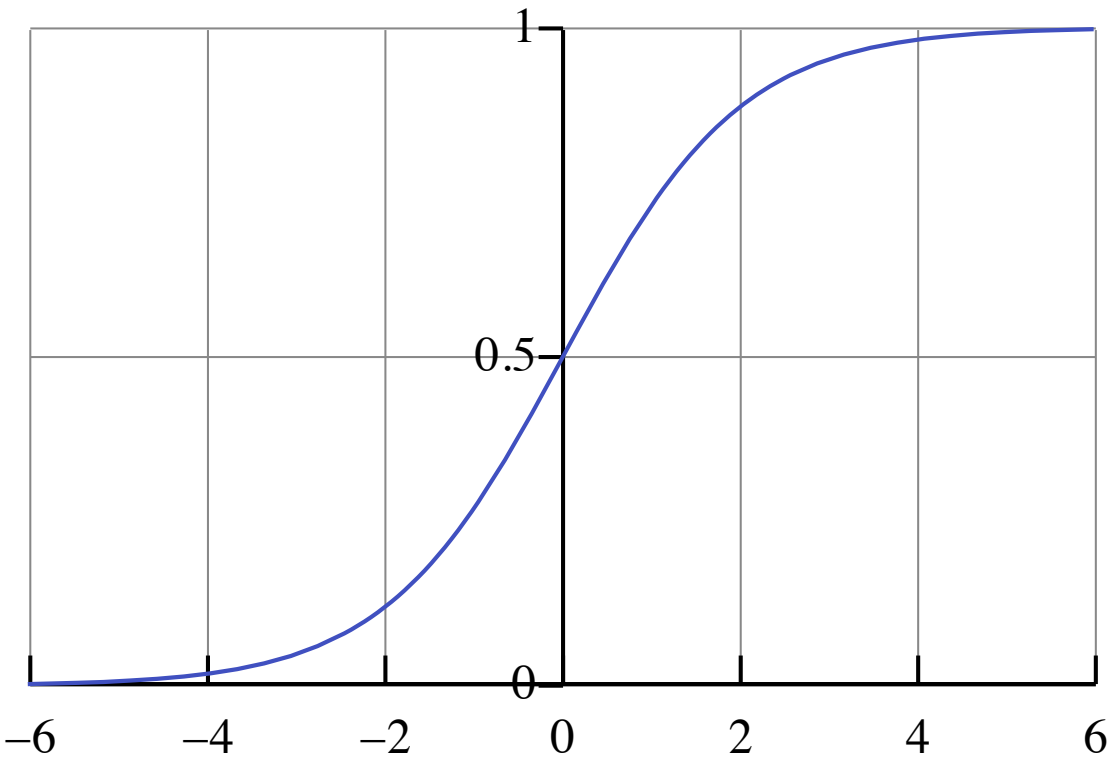
In [ ]:
```
plt.plot(data_set.x[data_set.y==0], data_set.y[data_set.y==0], 'ro', label="
plt.plot(data_set.x[data_set.y==1], data_set.y[data_set.y==1]-1, 'o', label=
plt.xlabel('x')
plt.legend()
```

Create the Model and Total Loss Function (Cost)

For Logistic Regression typically we would not use PyTorch instead we would use Scikit-Learn as it is easier to use and set up. We are using PyTorch because it is good practice for deep learning. Scikit-Learn is typically used for Machine Learning while PyTorch is used for Deep Learning.

We will create a custom class that defines the architecture of Logistic Regression using PyTorch. Logistic Regression has a single layer where the input is the number of features an X value of the dataset has (dimension of X) and there is a single output. The output of the layer is put into a sigmoid function which is a function between 0 and 1. The larger the output of the layer the closer it is to 1 and the smaller the output is the closer it is to 0. The sigmoid function will allow us to turn this output into a classification problem. If the output value is closer to 1 it is one class if it is closer to 0 it is in another.

Sigmoid Function



In [ ]:
```
# Create logistic_regression class that inherits nn.Module which is the base
class logistic_regression(nn.Module):

    # Constructor
    def __init__(self, n_inputs):
        super(logistic_regression, self).__init__()
        # Single layer of Logistic Regression with number of inputs being n_
        self.linear = nn.Linear(n_inputs, 1)

    # Prediction
    def forward(self, x):
        # Using the input x value puts it through the single layer defined a
        yhat = torch.sigmoid(self.linear(x))
        return yhat
```

We can check the number of features an X value has, the size of the input, or the dimension of X

In [ ]:
```
x,y = data_set[0]
len(x)
```

Create a logistic regression object or model, the input parameter is the number of dimensions.

In [ ]:
```
# Create the logistic_regression result

model = logistic_regression(1)
```

We can make a prediction sigma σ this uses the forward function defined above

In [ ]:
```
x = torch.tensor([-1.0])

sigma = model(x)
sigma
```

We can also make a prediction using our data

In [ ]:
```
x,y = data_set[2]

sigma = model(x)
sigma
```

Create a `plot_error_surfaces` object to visualize the data space and the learnable parameters space during training:

We can see on the Loss Surface graph, the loss value varying across w and b values with yellow being high loss and dark blue being low loss which is what we want

On the Loss Surface Contour graph we can see a top-down view of the Loss Surface graph

In [ ]:
```
# Create the plot_error_surfaces object

# 15 is the range of w
# 13 is the range of b
# data_set[:][0] are all the X values
# data_set[:][1] are all the Y values

get_surface = plot_error_surfaces(15, 13, data_set[:][0], data_set[:][1])
```

We define a criterion using Binary Cross Entropy Loss. This will measure the difference/loss between the prediction and actual value.

In [ ]:
```
criterion = nn.BCELoss()
```

We have our samples:

In [ ]:
```
x, y = data_set[0]
print("x = {},  y = {}".format(x,y))
```

We can make a prediction using the model:

In [ ]:
```
sigma = model(x)
sigma
```

We can calculate the loss

In [ ]:
```
loss = criterion(sigma, y)
loss
```

Setting the Batch Size using a Data Loader

You have to use data loader in PyTorch that will output a batch of data, the input is the `dataset` and `batch_size`

In [ ]:
```
batch_size=10
```

In [ ]:
```
trainloader = DataLoader(dataset = data_set, batch_size = 10)
```

In [ ]:
```
dataset_iter = iter(trainloader)
```

In [ ]:
```
X,y=next(dataset_iter )
```

We can see here that 10 values the same as our batch size

In [ ]:
```
X
```

Setting the Learning Rate

We can set the learning rate by setting it as a parameter in the optimizer along with the parameters of the logistic regression model we are training. The job of the optimizer, torch.optim.SGD, is to use the loss generated by the criterion to update the model parameters according to the learning rate. SGD stands for Stochastic Gradient Descent which typically means that the batch size is set to 1, but the data loader we set up above has turned this into Mini-Batch Gradient Descent.

In [ ]:
```
learning_rate = 0.1
```

In [ ]:
```
optimizer = torch.optim.SGD(model.parameters(), lr = learning_rate)
```

Train the Model via Mini-Batch Gradient Descent

We are going to train the model using various Batch Sizes and Learning Rates.

Mini-Batch Gradient Descent

In this case, we will set the batch size of the data loader to 5 and the number of epochs to 250.

First, we must recreate the get_surface object again so that for each example we get a Loss Surface for that model only.

In [ ]:
```
get_surface = plot_error_surfaces(15, 13, data_set[:][0], data_set[:][1], 30
```

Train the Model

In [ ]:
```
# First we create an instance of the model we want to train
model = logistic_regression(1)
# We create a criterion which will measure loss
criterion = nn.BCELoss()
# We create a data loader with the dataset and specified batch size of 5
trainloader = DataLoader(dataset = data_set, batch_size = 5)
# We create an optimizer with the model parameters and learning rate
optimizer = torch.optim.SGD(model.parameters(), lr = .01)
# Then we set the number of epochs which is the total number of times we wil
epochs=500
# This will store the loss over iterations so we can plot it at the end
loss_values = []

# Loop will execute for number of epochs
for epoch in range(epochs):
    # For each batch in the training data
    for x, y in trainloader:
        # Make our predictions from the X values
        yhat = model(x)
        # Measure the loss between our prediction and actual Y values
        loss = criterion(yhat, y)
        # Resets the calculated gradient value, this must be done each time
        optimizer.zero_grad()
        # Calculates the gradient value with respect to each weight and bias
        loss.backward()
        # Updates the weight and bias according to calculated gradient value
        optimizer.step()
        # Set the parameters for the loss surface contour graphs
        get_surface.set_para_loss(model, loss.tolist())
        # Saves the loss of the iteration
        loss_values.append(loss)
    # Want to print the Data Space for the current iteration every 20 epochs
    if epoch % 20 == 0:
        get_surface.plot_ps()
```

We can see the final values of the weight and bias. This weight and bias correspond to the orange line in the Data Space graph and the final spot of the X in the Loss Surface Contour graph.

In [ ]:
```
w = model.state_dict()['linear.weight'].data[0]
b = model.state_dict()['linear.bias'].data[0]
print("w = ", w, "b = ", b)
```

Now we can get the accuracy of the training data

In [ ]:
```
# Getting the predictions
yhat = model(data_set.x)
# Rounding the prediction to the nearedt integer 0 or 1 representing the cla
yhat = torch.round(yhat)
# Counter to keep track of correct predictions
correct = 0
# Goes through each prediction and actual y value
for prediction, actual in zip(yhat, data_set.y):
    # Compares if the prediction and actualy y value are the same
    if (prediction == actual):
        # Adds to counter if prediction is correct
        correct+=1
# Outputs the accuracy by dividing the correct predictions by the length of
print("Accuracy: ", correct/len(data_set)*100, "%")
```

Finally, we plot the Cost vs Iteration graph, although it is erratic it is downward sloping.

In [ ]:
```
plt.plot(loss_values)
plt.xlabel("Iteration")
plt.ylabel("Cost")
```

Stochastic Gradient Descent

In this case, we will set the batch size of the data loder to 1 so that the gradient descent will be performed for each example this is referred to as Stochastic Gradient Descent. The number of epochs is set to 100.

Notice that in this example the batch size is decreased from 5 to 1 so there would be more iterations. Due to this, we can reduce the number of iterations by decreasing the number of epochs. Due to the reduced batch size, we are optimizing more frequently so we don't need as many epochs.

First, we must recreate the `get_surface` object again so that for each example we get a Loss Surface for that model only.

In [ ]:
```
get_surface = plot_error_surfaces(15, 13, data_set[:][0], data_set[:][1], 30
```

Train the Model

In [ ]:
```
# First we create an instance of the model we want to train
model = logistic_regression(1)
# We create a criterion which will measure loss
criterion = nn.BCELoss()
# We create a data loader with the dataset and specified batch size of 1
trainloader = DataLoader(dataset = data_set, batch_size = 1)
# We create an optimizer with the model parameters and learning rate
optimizer = torch.optim.SGD(model.parameters(), lr = .01)
# Then we set the number of epochs which is the total number of times we wil
epochs=100
# This will store the loss over iterations so we can plot it at the end
loss_values = []

# Loop will execute for number of epochs
for epoch in range(epochs):
    # For each batch in the training data
    for x, y in trainloader:
        # Make our predictions from the X values
        yhat = model(x)
        # Measure the loss between our prediction and actual Y values
        loss = criterion(yhat, y)
        # Resets the calculated gradient value, this must be done each time
        optimizer.zero_grad()
        # Calculates the gradient value with respect to each weight and bias
        loss.backward()
        # Updates the weight and bias according to calculated gradient value
        optimizer.step()
        # Set the parameters for the loss surface contour graphs
        get_surface.set_para_loss(model, loss.tolist())
        # Saves the loss of the iteration
        loss_values.append(loss)
    # Want to print the Data Space for the current iteration every 20 epochs
    if epoch % 20 == 0:
        get_surface.plot_ps()
```

We can see the final values of the weight and bias. This weight and bias correspond to the orange line in the Data Space graph and the final spot of the X in the Loss Surface Contour graph.

In [ ]:
```
w = model.state_dict()['linear.weight'].data[0]
b = model.state_dict()['linear.bias'].data[0]
print("w = ", w, "b = ", b)
```

Now we can get the accuracy of the training data

In [ ]:
```
# Getting the predictions
yhat = model(data_set.x)
# Rounding the prediction to the nearedt integer 0 or 1 representing the cla
yhat = torch.round(yhat)
# Counter to keep track of correct predictions
correct = 0
# Goes through each prediction and actual y value
for prediction, actual in zip(yhat, data_set.y):
    # Compares if the prediction and actualy y value are the same
    if (prediction == actual):
        # Adds to counter if prediction is correct
        correct+=1
# Outputs the accuracy by dividing the correct predictions by the length of
print("Accuracy: ", correct/len(data_set)*100, "%")
```

Finally, we plot the Cost vs Iteration graph, although it is erratic it is downward sloping.

In [ ]:
```
plt.plot(loss_values)
plt.xlabel("Iteration")
plt.ylabel("Cost")
```

High Learning Rate

In this case, we will set the batch size of the data loder to 1 so that the gradient descent will be performed for each example this is referred to as Stochastic Gradient Descent. This time the learning rate will be set to .1 to represent a high learning rate and we will observe what will happen when we try to train.

First, we must recreate the `get_surface` object again so that for each example we get a Loss Surface for that model only.

In [ ]:
```
get_surface = plot_error_surfaces(15, 13, data_set[:][0], data_set[:][1], 30
```

Train the Model

In [ ]:
```
# First we create an instance of the model we want to train
model = logistic_regression(1)
# We create a criterion that will measure loss
criterion = nn.BCELoss()
# We create a data loader with the dataset and specified batch size of 1
trainloader = DataLoader(dataset = data_set, batch_size = 1)
# We create an optimizer with the model parameters and learning rate
optimizer = torch.optim.SGD(model.parameters(), lr = 1)
# Then we set the number of epochs which is the total number of times we wil
epochs=100
# This will store the loss over iterations so we can plot it at the end
loss_values = []

# Loop will execute for number of epochs
for epoch in range(epochs):
    # For each batch in the training data
    for x, y in trainloader:
        # Make our predictions from the X values
        yhat = model(x)
        # Measure the loss between our prediction and actual Y values
        loss = criterion(yhat, y)
        # Resets the calculated gradient value, this must be done each time
        optimizer.zero_grad()
        # Calculates the gradient value with respect to each weight and bias
        loss.backward()
        # Updates the weight and bias according to calculated gradient value
        optimizer.step()
        # Set the parameters for the loss surface contour graphs
        get_surface.set_para_loss(model, loss.tolist())
        # Saves the loss of the iteration
        loss_values.append(loss)
    # Want to print the Data Space for the current iteration every 20 epochs
    if epoch % 20 == 0:
        get_surface.plot_ps()
```

Notice in this example the due to the high learning rate the Loss Surface Contour graph has increased movement over the previous example and also moves in multiple directions due to the minimum being overshot.

We can see the final values of the weight and bias. This weight and bias correspond to the orange line in the Data Space graph and the final spot of the X in the Loss Surface Contour graph.

In [ ]:
```
w = model.state_dict()['linear.weight'].data[0]
b = model.state_dict()['linear.bias'].data[0]
print("w = ", w, "b = ", b)
```

Now we can get the accuracy of the training data

In [ ]:
```
# Getting the predictions
yhat = model(data_set.x)
# Rounding the prediction to the nearedt integer 0 or 1 representing the cla
yhat = torch.round(yhat)
# Counter to keep track of correct predictions
correct = 0
# Goes through each prediction and actual y value
for prediction, actual in zip(yhat, data_set.y):
    # Compares if the prediction and actualy y value are the same
    if (prediction == actual):
        # Adds to counter if prediction is correct
        correct+=1
# Outputs the accuracy by dividing the correct predictions by the length of
print("Accuracy: ", correct/len(data_set)*100, "%")
```

Finally, we plot the Cost vs Iteration graph, although it is erratic it is downward sloping.

In [ ]:
```
plt.plot(loss_values)
plt.xlabel("Iteration")
plt.ylabel("Cost")
```

Question

Using the following code train the model using a `learning rate` of `.01`, `120 epochs`, and `batch_size` of `1`.

In [ ]:
```
get_surface = plot_error_surfaces(15, 13, data_set[:][0], data_set[:][1], 30
```

Train the Model

In [ ]:

```
# First we create an instance of the model we want to train
model = logistic_regression(1)
# We create a criterion which will measure loss
criterion = nn.BCELoss()
# We create a data loader with the dataset and specified batch size of 1
trainloader = DataLoader(dataset = data_set, batch_size = "SET_BATCH_SIZE")
# We create an optimizer with the model parameters and learning rate
optimizer = torch.optim.SGD(model.parameters(), lr = "SET_LEARNING_RATE")
# Then we set the number of epochs which is the total number of times we wil
epochs= "SET_NUMBER_OF_EPOCHS"
# This will store the loss over iterations so we can plot it at the end
loss_values = []

# Loop will execute for number of epochs
for epoch in range(epochs):
    # For each batch in the training data
    for x, y in trainloader:
        # Make our predictions from the X values
        yhat = model(x)
        # Measure the loss between our prediction and actual Y values
        loss = criterion(yhat, y)
        # Resets the calculated gradient value, this must be done each time
        optimizer.zero_grad()
        # Calculates the gradient value with respect to each weight and bias
        loss.backward()
        # Updates the weight and bias according to calculated gradient value
        optimizer.step()
        # Set the parameters for the loss surface contour graphs
        get_surface.set_para_loss(model, loss.tolist())
        # Saves the loss of the iteration
        loss_values.append(loss)
    # Want to print the Data Space for the current iteration every 20 epochs
    if epoch % 20 == 0:
        get_surface.plot_ps()
```