tut-11-ann

November 10, 2023

1 Implementing A Logistic Regression Model from Scratch with PyTorch

In this tutorial, we are going to implement a logistic regression model from scratch with PyTorch. The model will be designed with neural networks in mind and will be used for a simple image classification task. I believe this is a great approach to begin understanding the fundamental building blocks behind a neural network. Additionally, we will also look at best practices on how to use PyTorch for training neural networks.

After completing this tutorial the learner is expected to know the basic building blocks of a logistic regression model. The learner is also expected to apply the logistic regression model to a binary image classification problem of their choice using PyTorch code.

---

[1]:
```
## Import the usual libraries
import torch
import torchvision
import torch.nn as nn
from torchvision import datasets, models, transforms
import os
import numpy as np
import matplotlib.pyplot as plt

%matplotlib inline

print(torch.__version__)
```

```
2.1.0+cu118
```

[2]:
```
## configuration to detect cuda or cpu
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
print (device)
```

```
cuda:0
```

1.1 Importing Dataset

In this tutorial we will be working on an image classification problem.

The objective of our model is to learn to classify between "bee" vs. "no bee" images.

Since we are using Google Colab, we will need to first import our data into our environment using the code below:

[3]:
```
!gdown 19xtUfic7D0q4KvvJcPS7lIxf3RuRgye0
!unzip -q ./hymenoptera_data.zip
!ls
```

```
Downloading…
From: https://drive.google.com/uc?id=19xtUfic7D0q4KvvJcPS7lIxf3RuRgye0
To: /content/hymenoptera_data.zip
100% 47.3M/47.3M [00:02<00:00, 17.5MB/s]
hymenoptera_data  hymenoptera_data.zip  sample_data
```

1.2 Data Transformation

This is an image classification task, which means that we need to perform a few transformations on our dataset before we train our models. I used similar transformations as used in this tutorial. For a detailed overview of each transformation take a look at the official torchvision documentation.

The following code block performs the following operations: - The `data_transforms` contains a series of transformations that will be performed on each image found in the dataset. This includes cropping the image, resizing the image, converting it to tensor, reshaping it, and normalizing it. - Once those transformations have been defined, then the `DataLoader` function is used to automatically load the datasets and perform any additional configuration such as shuffling, batches, etc.

[4]:
```
## configure root folder on your gdrive
data_dir = './hymenoptera_data'

## custom transformer to flatten the image tensors
# class ReshapeTransform:
#     def __init__(self, new_size):
#         self.new_size = new_size

#     def __call__(self, img):
#         result = torch.reshape(img, self.new_size)
#         return result

## transformations used to standardize and normalize the datasets
data_transforms = {
    'train': transforms.Compose([
        transforms.Resize(224),
        transforms.CenterCrop(224),
        transforms.ToTensor(),
        transforms.Lambda(torch.flatten)
    ]),
    'val': transforms.Compose([
        transforms.Resize(224),
        transforms.CenterCrop(224),
        transforms.ToTensor(),
        transforms.Lambda(torch.flatten)
    ]),
}

## load the correspoding folders
image_datasets = {x: datasets.ImageFolder(os.path.join(data_dir, x),
                                          data_transforms[x])
                  for x in ['train', 'val']}

## load the entire dataset; we are not using minibatches here
train_dataset = torch.utils.data.DataLoader(image_datasets['train'],
                                            ␣
 ↪batch_size=len(image_datasets['train']),
                                            shuffle=True)
test_dataset = torch.utils.data.DataLoader(image_datasets['val'],
                                           ␣
 ↪batch_size=len(image_datasets['val']),
                                           shuffle=True)
```

1.3 Print sample

It's always a good practise to take a quick look at the dataset before training your models. Below we print out an example of one of the images from the `train_dataset`.

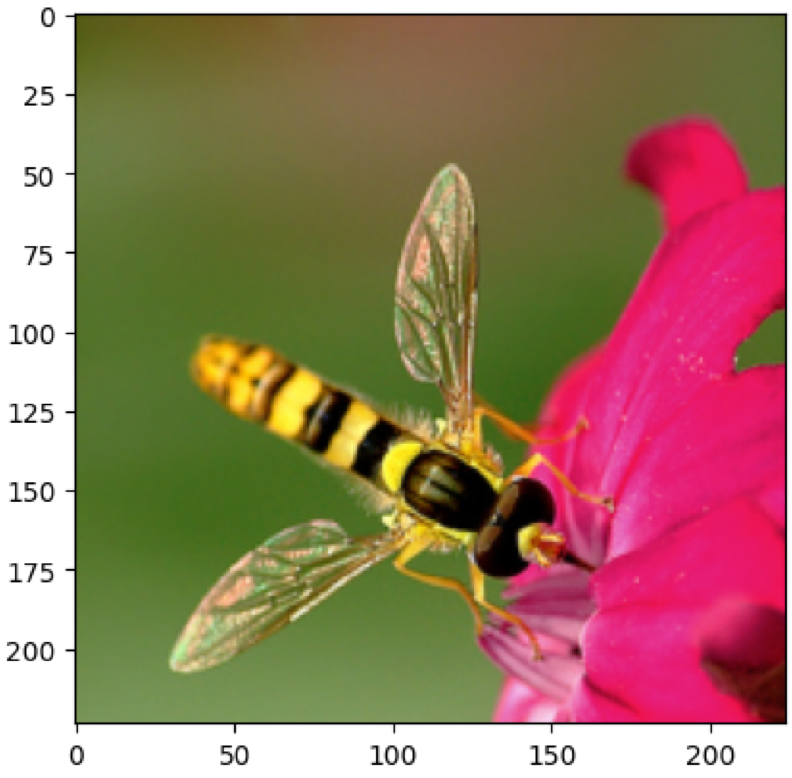
[5]:
```
## load the entire dataset
x, y = next(iter(train_dataset))

## print one example
dim = x.shape[1]
print("Dimension of image:", x.shape, "\n",
      "Dimension of labels", y.shape)

plt.imshow(x[160].reshape(1, 3, 224, 224).squeeze().T.numpy())
```

```
Dimension of image: torch.Size([244, 150528])
 Dimension of labels torch.Size([244])
```

```
<ipython-input-5-c890a31777ed>:9: UserWarning: The use of `x.T` on tensors of
dimension other than 2 to reverse their shape is deprecated and it will throw an
error in a future release. Consider `x.mT` to transpose batches of matrices or
`x.permute(*torch.arange(x.ndim - 1, -1, -1))` to reverse the dimensions of a
tensor. (Triggered internally at ../aten/src/ATen/native/TensorShape.cpp:3614.)
```

```
plt.imshow(x[160].reshape(1, 3, 224, 224).squeeze().T.numpy())
```

```
[5]: <matplotlib.image.AxesImage at 0x7f8adbfcbd60>
```



1.4 Building the Model

Let's now implement our logistic regression model. Logistic regression is one in a family of machine learning techniques that are used to train binary classifiers. They are also a great way to understand the fundamental building blocks of neural networks, thus they can also be considered the simplest of neural networks where the model performs a `forward` and `backward` propagation to train the model on the data provided.

If you don't fully understand the structure of the code below, I strongly recommend you to read the following Week 2 of Andrew Ng's Deep Learning Specialization course for all the explanation, intuitions, and details of the different parts of the neural network such as the `forward`, `sigmoid`, `backward`, and `optimization` steps.

In short: - The `__init__` function initializes all the parameters (`W`, `b`, `grad`) that will be used to train the model through backpropagation. - The goal is to learn the `W` and `b` that minimimizes the cost function which is computed as seen in the `loss` function below.

Note that this is a very detailed implementation of a logistic regression model so I had to explicitly move a lot of the computations into the GPU for faster calcuation, `to(device)` takes care of this in PyTorch.

[6]:
```
class ANN(nn.Module):
    def __init__(self, dim, lr=torch.scalar_tensor(0.01)):
        super(ANN, self).__init__()
        # intialize parameters
        self.w = torch.zeros(dim, 1, dtype=torch.float).to(device)
        self.b = torch.scalar_tensor(0).to(device)
        self.grads = {"dw": torch.zeros(dim, 1, dtype=torch.float).to(device),
                      "db": torch.scalar_tensor(0).to(device)}
        self.lr = lr.to(device)

    def sigmoid(self, z):
        return 1/(1 + torch.exp(-z))

    def forward(self, x):
        ## compute forward
        z = torch.mm(self.w.T, x)
        a = self.sigmoid(z)
        return a #h_\theta(x)

    def backward(self, x, yhat, y):
        ## compute backward
        self.grads["dw"] = (1/x.shape[1]) * torch.mm(x, (yhat - y).T)
        self.grads["db"] = (1/x.shape[1]) * torch.sum(yhat - y)

    def optimize(self):
        ## optimization step
        self.w = self.w - self.lr * self.grads["dw"]
        self.b = self.b - self.lr * self.grads["db"]

## utility functions
def loss(yhat, y):
    # m = y.size()[1]
    return -(1/y.size()[1]) * torch.sum(y * torch.log(yhat) + (1 - y) * torch.
 ↪log(1-yhat))

def predict(yhat, y):
    y_prediction = torch.zeros(1, y.size()[1])
    for i in range(yhat.size()[1]):
        if yhat[0, i] <= 0.5:
            y_prediction[0, i] = 0
        else:
            y_prediction[0, i] = 1
    return 100 - torch.mean(torch.abs(y_prediction - y)) * 100
```

1.5 Pretesting the Model

It is also good practice to test your model and make sure the right steps are taking place before training the entire model.

[7]:
```
## model pretesting
x, y = next(iter(train_dataset))

## flatten/transform the data
x_flatten = x.T
y = y.unsqueeze(0)

## num_px is the dimension of the images
dim = x_flatten.shape[0]

## model instance
model = ANN(dim).to(device)
yhat = model.forward(x_flatten.to(device))
yhat = yhat.data.cpu()

## calculate loss
cost = loss(yhat, y)
prediction = predict(yhat, y)
print("Cost: ", cost)
print("Accuracy: ", prediction)

## backpropagate
model.backward(x_flatten.to(device), yhat.to(device), y.to(device))
model.optimize()
```

```
Cost:  tensor(0.6931)
Accuracy:  tensor(50.4098)
```

1.6 Train the Model

It's now time to train the model.

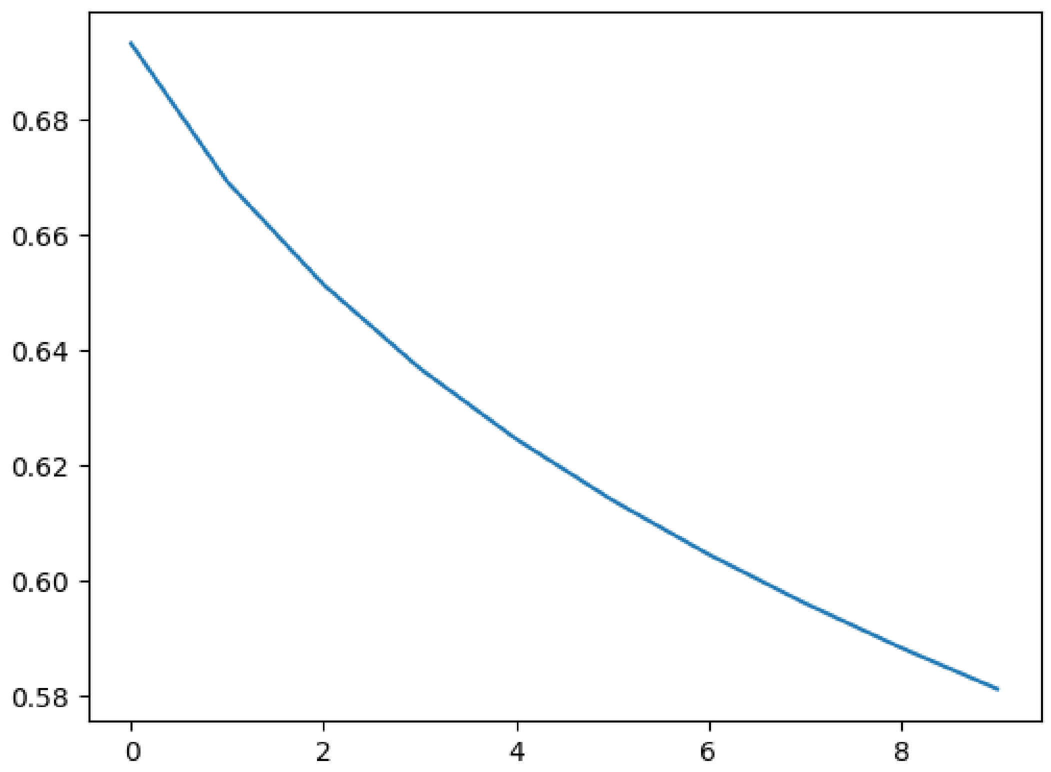
[8]:
```
%%time

## hyperparams
costs = []
dim = x_flatten.shape[0]
learning_rate = torch.scalar_tensor(0.0001).to(device)
num_iterations = 100
lrmodel = ANN(dim, learning_rate)
lrmodel.to(device)

## transform the data
def transform_data(x, y):
    x_flatten = x.T
    y = y.unsqueeze(0)
    return x_flatten, y

## training the model
for i in range(num_iterations):
    x, y = next(iter(train_dataset))
    test_x, test_y = next(iter(test_dataset))
    x, y = transform_data(x, y)
    test_x, test_y = transform_data(test_x, test_y)

    # forward
    yhat = lrmodel.forward(x.to(device))
    cost = loss(yhat.data.cpu(), y)
    train_pred = predict(yhat, y)

    # backward
    lrmodel.backward(x.to(device),
                     yhat.to(device),
                     y.to(device))
    # Update the weights
    lrmodel.optimize()

    ## test
    yhat_test = lrmodel.forward(test_x.to(device))
    test_pred = predict(yhat_test, test_y)

    if i % 10 == 0:
        costs.append(cost)

    if i % 10 == 0:
        print("Cost after iteration {}: {} | Train Acc: {} | Test Acc: {}".
↪format(i,
                                                                                ␣
↪    cost,
                                                                                ␣
↪    train_pred,
                                                                                ␣
↪    test_pred))
```

```
Cost after iteration 0: 0.6931472420692444 | Train Acc: 50.40983581542969 | Test
Acc: 45.75163269042969
Cost after iteration 10: 0.669146716594696 | Train Acc: 64.3442611694336 | Test
Acc: 54.24836730957031
Cost after iteration 20: 0.651317834854126 | Train Acc: 68.44261932373047 | Test
Acc: 54.24836730957031
Cost after iteration 30: 0.6367819905281067 | Train Acc: 68.03278350830078 |
Test Acc: 54.24836730957031
Cost after iteration 40: 0.6245331764221191 | Train Acc: 69.67213439941406 |
Test Acc: 54.90196228027344
Cost after iteration 50: 0.6139220595359802 | Train Acc: 70.90164184570312 |
Test Acc: 56.20914840698242
Cost after iteration 60: 0.604523241519928 | Train Acc: 72.54098510742188 | Test
Acc: 56.86274337768555
Cost after iteration 70: 0.5960509181022644 | Train Acc: 74.18032836914062 |
Test Acc: 57.51633834838867
Cost after iteration 80: 0.5883082151412964 | Train Acc: 73.77049255371094 |
Test Acc: 57.51633834838867
Cost after iteration 90: 0.58115553855896 | Train Acc: 74.59016418457031 | Test
Acc: 58.1699333190918
CPU times: user 3min 47s, sys: 15.5 s, total: 4min 3s
Wall time: 4min 5s
```

1.7 Result

From the loss curve below you can see that the model is sort of learning to classify the images given the decreas in the loss. I only ran the model for `100` iterations. Train the model for many more rounds and analyze the results. In fact, I have suggested a couple of experiments and exercises at the end of the tutorial that you can try to get a more improved model.

[9]:
```
## the trend in the context of loss
plt.plot(costs)
plt.show()
```



1.8 References

- Understanding the Impact of Learning Rate on Neural Network Performance
- TRANSFER LEARNING FOR COMPUTER VISION TUTORIAL
- Deep Learning Specialization by Andrew Ng