A Developer Diary

{about:"code learn and share"}

April 21, 2019 By Abhisek Jana — Leave a Comment (Edit)

Implement Neural Network using PyTorch

PyTorch is gaining popularity specially among students since it's much more developer friendly. PyTorch helps to focus more on core concepts of deep learning unlike TensorFlow which is more focused on running optimized model on production system. In this tutorial we will Implement Neural Network using PyTorch and understand some of the core concepts of PyTorch.

This tutorial is more like a follow through of the previous tutorial on Understand and Implement the Backpropagation Algorithm From Scratch In Python.If you need a refresher on this please review my previous article.

Understand and Implement the Backpropagation Algorithm From Scratch In Python

It's very important have clear understanding on how to implement a simple Neural Network from scratch. In this Understand and Implement the Backpropagation Algorithm From Scratch In Python tutorial we go through step by step process of understanding and implementing a Neural Network. We will start from Linear Regression and use the same concept to … Continue reading

A Developer Diary 24

Notes on PyTorch:

- PyTorch models cannot be deployed to a production system directly. It needs to be converted to Caffe2 using ONNX, then deploy to production.
- PyTorch supports computations using GPU(cuda) for faster processing. I will explain how to do this in this tutorial.
- In other deep learning frameworks such as TensorFlow or Theano, you can just feed the input data in NumPy format to the model.It's easy to implement this way, specially when you are trying out for the first time or learning. However `batching`, `shuffling`, `parallel` data loading etc needs to be taken care manually when you are looking for real implementation. PyTorch provides all these functionalities out of the box

using the `torch.utils.data.Dataset` and `torch.utils.data.DataLoader`
- PyTorch automatically calculates derivate of any function, hence our backpropagation will be very easy to implement.
- PyTorch provides Modules, which are nothing but abstract class or interface. If you are familiar with OOPS then you already know about inheritance. Modules helps to integrate our custom code with the PyTorch core framework.

Dataset:

We will be using the MNIST dataset. It has 60K training images, each 28X28 pixel in gray scale. There are total 10 classes to classify. You can find more details about it in the following sites:

https://en.wikipedia.org/wiki/MNIST_database
http://yann.lecun.com/exdb/mnist/index.html

Implementation:

In PyTorch we need to define our Neural Network using a class. We will name our class as ANN. We will also add the `fit()` and `predict()` function so that we can invoke them from the `main()` function.

__main__():

Lets look at our simple main method. We will first get the data from the `get_data()` function. I am using an external library to load the MNIST data. You can install it using the below command.

```
pip install python-mnist
```

we have defined the `device` variable before main function. This will help to detect if the machine has cuda supported GPU so that we can run our model faster.

First we will run through our pre-processing step where we are normalizing the data. Then instantiate the `ANN` class by passing the `layers_size`. We will code the ANN class such way that we can define the layers dynamically.

Then we will call the `fit()` and `predict()` function.

```
device = torch.device("cuda:0" if
torch.cuda.is_available() else "cpu")

if __name__ == '__main__':
    train_x_orig, train_y_orig, test_x_orig, test_y_orig
= get_data()
    train_x, train_y, test_x, test_y =
pre_process_data(train_x_orig, train_y_orig, test_x_orig,
test_y_orig)

    model = ANN(layers_size=[196, 10])
    model.fit(train_x, train_y, learning_rate=0.1,
n_iterations=1000)
    model.predict(test_x, test_y)
    model.plot_cost()
```

__init__() of ANN Class:

As discussed earlier, PyTorch provides Modules for specific type of Neural Networks. We will be extending the `torch.nn.Module` while creating the ANN class.

The `__init__()` method is very srtaight forward. In the first line we will call the `__init__()` method of the parent class `torch.nn.Module`.

```
class ANN(nn.Module):
    def __init__(self, layers_size):
        super(ANN, self).__init__()
        self.layers_size = layers_size
        self.L = len(layers_size)
        self.costs = []
```

initialize_parameters():

In the `initialize_parameters()` function we will define our Layes with W's and b's. Since we dont want to create fixed set of layers, we will loop through our `self.layers_size` list and call `nn.Linear()` function.

There are two important points to note here:

- We will be calling `nn.Linear().to(device)` so that PyTorch can select GPU (if available) for computation.
- `add_module()` function is part of `torch.nn.Module`. PyTorch provides this function so that we can define all the layers dynamically.

```
def initialize_parameters(self):
        for l in range(0, self.L):
                self.add_module("fc" + str(l + 1),
nn.Linear(self.layers_size[l], self.layers_size[l +
1]).to(device))
```

forward():

The `forward()` is inherited from the `torch.nn.Module`, which means you need to always define a function named `forward()`. Otherwise PyTorch wont be able to execute this function.

The logic in this function is very easy to understand. We will loop through all the different layers that was added by calling the `self.add_module` and both Z and A was calculated. (Z is the output before Activation and A is the output of the Activation)

We are using Relu as activation function for all the hidden layers except for the last layer. That's why we are not calculating that for the last layer `L` inside the loop.

We are calling `torch.nn.functional.log_softmax()` function for the `Softmax` activation.

```
def forward(self, X):
        for l, (name, m) in
enumerate(self.named_modules()):
                if l > 0:
                        if l == self.L - 1:
                                X = m(X)
                        else:
                                X = F.relu(m(X))

        return F.log_softmax(input=X)
```

fit():

The `fit()` function drives all the work for us, hence we will break it down to understand fully.

The `self.to()` is a built in function which is part of the `torch.nn.Module`. We will pass the `device` here so that PyTorch knows whether to execute the computation in CPU or GPU.

Next we will insert the feature size to the `self.layers_size` list since technically `X` is the `layer 0`.

Invoke `self.initialize_parameters()` to create the required layers. Use `torch.optim.SGD()` for updating the parameters using Stochastic Gradient Descent. We need pass the parameters by calling `self.parameters()` (which is again part of `torch.nn.Module`) and the `learning rate`.

We can define the negative log likelihood loss function just by calling `torch.nn.NLLLoss()`.

```
def fit(self, X, Y, learning_rate=0.1, 
n_iterations=2500):
        self.to(device)
        self.layers_size.insert(0, X.shape[1])
        self.initialize_parameters()

        optimizer = torch.optim.SGD(self.parameters(), 
lr=learning_rate)
        criterion = nn.NLLLoss()
```

We are all set to run our training iterations. However as discussed earlier, we need to make sure PyTorch can retrieve the data using the `torch.utils.data.DataLoader` class.

PyTorch DataLoader:

We need to inherit the `torch.utils.data.Dataset` class and provide implementation of the necessary methods.

Here is the structure of our class `MyDataLoader`. Here in the `__init__()` method will initialize `data` and `target`. We can actually read the data from the file in the init method itself since it will be executed only once, however in order to make the code simple, we will just pass our already loaded numpy data there.

__len__():

The `__len__()` method needs to return the length of the dataset.

__getitem__():

At runtime PyTorch will call `__getitem__()` method and create the mini batch randomly. We just need to return the feature row vector and target class as a tuple, based on the index that was passed. Here we will convert the numpy array to `torch.Tensor`.

Also, remember that we don't have to transform the target using OneHotEncoding, since PyTorch will take care of that automatically.

We will just convert the target class to `int` since PyTorch does not integrate directly with NumPy.

```
class MyDataLoader(data.Dataset):
        def __init__(self, X, Y):
                self.data = X
                self.target = Y
                self.n_samples = self.data.shape[0]

        def __len__(self):
                return self.n_samples

        def __getitem__(self, index):
```

```
            return torch.Tensor(self.data[index]),
int(self.target[index])
```

Once we have the `MyDataLoader` class completed, we can create an install of the class by passing our train feature matrix and target class vector.

Next we will pass the instance of `MyDataLoader` to the `torch.utils.data.DataLoader` class. We also need to provide the `batch_size` and `num_workers`. I have selected batch size of 2048 and num_workers will be mostly be the number of CPU core you have you. Since I have 32, I have provided the same.

The PyTorch's `DataLoader` class takes care of `batching`, `shuffling`, `parallel` data loading etc. Nice !

```
train_dataset = self.MyDataLoader(X, Y)
data_loader =
torch.utils.data.DataLoader(dataset=train_dataset,
batch_size=2048, num_workers=32)
```

Training Loop:

Back to our training loop inside the `fit()` function. First we will loop through the `n_iterations` and then the `data_loader`.

The `data_loader` will return a batch of train data. In order to use them for training we need to send them to the appropriate device such as CPU or GPU. Just call the `.to` function so that the data can be moved to GPU memory or stay in on-board memory.

Then we will reset the gradient by calling `optimizer.zero_grad()`. `self(inputs)` will automatically execute the `forward()` function.

Next, the loss will be calculated using the predicted value and ground truth. Afterwards, call `loss.backward()` for computing the backpropagation and update the parameters using the `optimizer.step()` function.

```
for epoch in range(n_iterations):
        for k, (inputs, target) in
enumerate(data_loader):
                inputs, target = inputs.to(device),
target.to(device)

                optimizer.zero_grad()
                forward = self(inputs)
                loss = criterion(forward, target)
                loss.backward()
                optimizer.step()
```

predict():

We will use the `MyDataLoader` class for loading the test data too. Here we will use `with torch.no_grad()` in order to inform PyTorch that there is no need to track for gradients (This will save some computation).

Below code is very straight forward. I will have you go through and ask question as needed.

```
def predict(self, X, Y):
        dataset = self.MyDataLoader(X, Y)
        data_loader =
torch.utils.data.DataLoader(dataset=dataset,
batch_size=2048, num_workers=32)
        with torch.no_grad():
```

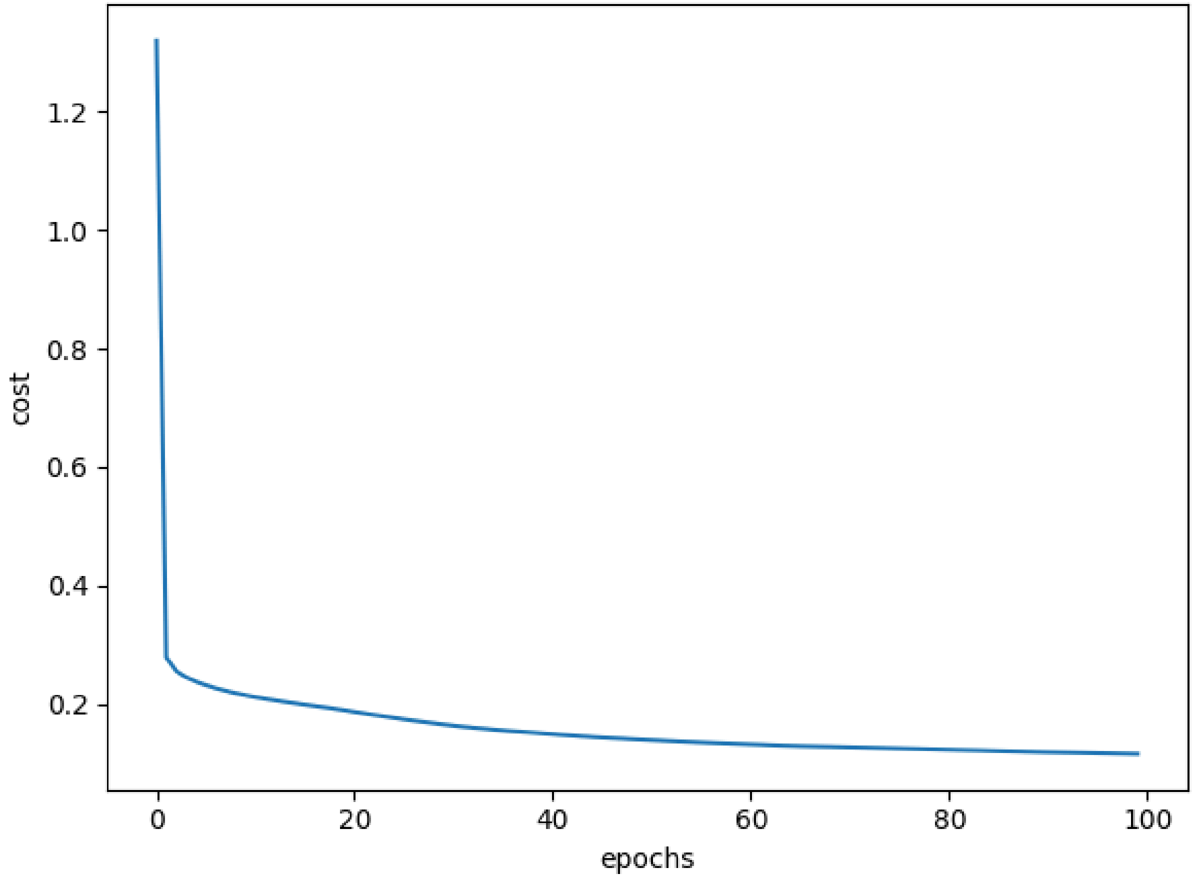
```
                correct = 0
                total = 0
                for inputs, target in data_loader:
                        inputs, target =
inputs.to(device), target.to(device)
                        forward = self(inputs)
                        _, predicted =
torch.max(forward.data, 1)
                        total += target.size(0)
                        correct += (predicted ==
target).sum().item()

                print('Accuracy of the network on the {}
images: {} %'.format(Y.shape[0], 100 * correct / total))
```

Results:

Just using 2-Layes, `[196, 10]` we can achieve 92.77% Accuracy in the Test set.

```
Train Epoch: 0  Loss: 1.382688
Train Epoch: 100        Loss: 0.211236
Train Epoch: 200        Loss: 0.187463
Train Epoch: 300        Loss: 0.164462
Train Epoch: 400        Loss: 0.149739
Train Epoch: 500        Loss: 0.140264
Train Epoch: 600        Loss: 0.133282
Train Epoch: 700        Loss: 0.127745
Train Epoch: 800        Loss: 0.122800
Train Epoch: 900        Loss: 0.118388
Train Accuracy: 94.23 %
Accuracy of the network on the 10000 images: 92.77 %
```

Here is the plot of the Cost function.



Try using different layers and hidden units and see how the accuracy changes.

Full ANN Class:

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.utils.data as data
import datasets.mnist.loader as mnist
import matplotlib.pyplot as plt
import numpy as np

device = torch.device("cuda:0" if
torch.cuda.is_available() else "cpu")
```

```
class ANN(nn.Module):
    class MyDataLoader(data.Dataset):
        def __init__(self, X, Y):
            self.data = X
            self.target = Y
            self.n_samples = self.data.shape[0]

        def __len__(self):
            return self.n_samples

        def __getitem__(self, index):
            return torch.Tensor(self.data[index]),
int(self.target[index])

    def __init__(self, layers_size):
        super(ANN, self).__init__()
        self.layers_size = layers_size
        self.L = len(layers_size)
        self.costs = []

    def initialize_parameters(self):
        for l in range(0, self.L):
            self.add_module("fc" + str(l + 1),
nn.Linear(self.layers_size[l], self.layers_size[l +
1]).to(device))

    def forward(self, X):

        for l, (name, m) in
enumerate(self.named_modules()):
            if l > 0:
                if l == self.L - 1:
                    X = m(X)
```

```
                else:
                    X = F.relu(m(X))

        return F.log_softmax(input=X)

    def fit(self, X, Y, learning_rate=0.1,
n_iterations=2500):

        self.to(device)

        self.layers_size.insert(0, X.shape[1])

        self.initialize_parameters()

        optimizer = torch.optim.SGD(self.parameters(),
lr=learning_rate)
        criterion = nn.NLLLoss()

        train_dataset = self.MyDataLoader(X, Y)
        data_loader =
torch.utils.data.DataLoader(dataset=train_dataset,
batch_size=2048, num_workers=32)

        for epoch in range(n_iterations):
            for k, (inputs, target) in
enumerate(data_loader):
                inputs, target = inputs.to(device),
target.to(device)

                optimizer.zero_grad()
                forward = self(inputs)
                loss = criterion(forward, target)
                loss.backward()
```

```
                optimizer.step()

            if epoch % 100 == 0:
                print('Train Epoch: {} \tLoss:
{:.6f}'.format(epoch, loss.item()))

            if epoch % 10 == 0:
                self.costs.append(loss.item())

        with torch.no_grad():
            correct = 0
            total = 0
            for inputs, labels in data_loader:
                inputs, labels = inputs.to(device),
labels.to(device)
                outputs = self(inputs)
                _, predicted = torch.max(outputs.data, 1)
                total += labels.size(0)
                correct += (predicted ==
labels).sum().item()
            print('Train Accuracy: {:.2f} %'.format(100 *
correct / total))

    def plot_cost(self):
        plt.figure()
        plt.plot(np.arange(len(self.costs)), self.costs)
        plt.xlabel("epochs")
        plt.ylabel("cost")
        plt.show()

    def predict(self, X, Y):
        dataset = self.MyDataLoader(X, Y)
        data_loader =
```

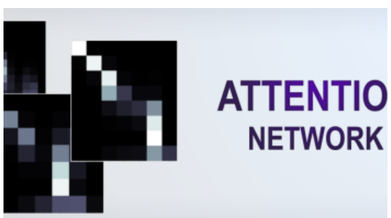
```
torch.utils.data.DataLoader(dataset=dataset,
batch_size=2048, num_workers=32)
        with torch.no_grad():
            correct = 0
            total = 0
            for inputs, target in data_loader:
                inputs, target = inputs.to(device),
target.to(device)
                forward = self(inputs)
                _, predicted = torch.max(forward.data, 1)
                total += target.size(0)
                correct += (predicted ==
target).sum().item()

            print('Accuracy of the network on the {}
images: {} %'.format(Y.shape[0], 100 * correct / total))


def pre_process_data(train_x, train_y, test_x, test_y):
    # Normalize
    train_x = train_x / 255.
    test_x = test_x / 255.

    return train_x, train_y, test_x, test_y


if __name__ == '__main__':
    train_x_orig, train_y_orig, test_x_orig, test_y_orig
= mnist.get_data()
    train_x, train_y, test_x, test_y =
pre_process_data(train_x_orig, train_y_orig, test_x_orig,
test_y_orig)
```

```
    model = ANN(layers_size=[196, 10])
    model.fit(train_x, train_y, learning_rate=0.1,
n_iterations=1000)
    model.predict(test_x, test_y)
    model.plot_cost()
```

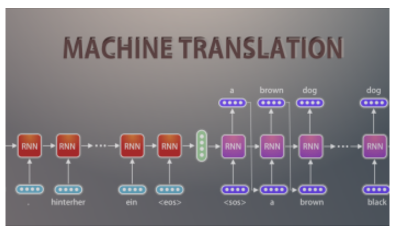
Please find the full project here:

Code

Related



Machine Translation using Attention with PyTorch
In "Data Science"



Machine Translation using Recurrent Neural Network and PyTorch
In "Data Science"

Understanding and implementing Neural Network with SoftMax in Python from scratch
In "Data Science"

Filed Under: Data Science, Deep Learning | Tagged With: Deep Learning, Machine Learning, MNIST, Neural Network, Python, PyTorch

Subscribe to stay in loop

* indicates required

Email Address *

Subscribe

Leave a Reply

Logged in as Abhisek Jana. Edit your profile. Log out? Required fields are marked *

Comment *

Post Comment

This site uses Akismet to reduce spam. Learn how your comment data is processed.