PyTorch PAM

May 15, 2023

1 Imports

[1]:
```
import torch
from torch import nn
from torch.utils.data import Dataset, DataLoader
import numpy as np
from tqdm import tqdm
from matplotlib import pyplot as plt
import itertools
```

2 Data

Note that the below functions are almost identical to the ones used in PAM; since the task is merely to replicate the functionality using PyTorch, a lot can be copied.

[2]:
```
def get_data(L):
    symbols = [-3, -1, 1, 3]
    one_hot_symbols = {
        -3: [1, 0, 0, 0],
        -1: [0, 1, 0, 0],
        1: [0, 0, 1, 0],
        3: [0, 0, 0, 1]
    }
    input_sequence = np.random.choice(symbols, size=L)
    output_sequence = np.array([one_hot_symbols[i] for i in input_sequence])
    return input_sequence, output_sequence

def get_signal(input_data):
    a0 = np.zeros(input_data.size * (m), input_data.dtype)
    a0[::m] = input_data
    v = np.convolve(g_T, a0)
    return v

def add_noise(input_data, sigma):
    N = input_data.shape[0]
    noise = np.random.normal(loc=0, scale=sigma, size=N)
    return input_data + noise
```

```
def add_bias_dimension(input_data):
    N = input_data.shape[0]
    return np.column_stack((input_data, np.ones(N)))

def split_data(input_data, fraction):
    N = input_data.shape[0]
    part = int(fraction*N)
    return input_data[:part], input_data[part:]
```

[3]:
```
L = 20000
fraction = 3/4
noise_sigma = 3
T = 1
m = 10
Ts = T/m
ti = np.arange(-4*T, 4*T, Ts)
g_T = (np.cos(2*np.pi * ti/T)) / (1- (4* ti/T)**2)

input_sequence, output_sequence = get_data(L)
v = get_signal(input_sequence)
v_noisy = add_noise(v, noise_sigma)
start = np.argmax(np.convolve(g_T,g_T))
s = np.convolve(g_T, v_noisy)
a_hat = s[start:-start:m]
input_sequence = add_noise(input_sequence, noise_sigma)
training_input, testing_input = split_data(a_hat, fraction)
training_output, testing_output = split_data(output_sequence, fraction)
```

The below defines a custom dataset with the correct types, puts them in DataLoader (iterable datasets), and defines the network

[5]:
```
class Data(Dataset):
    def __init__(self, X, y):
        self.X = torch.from_numpy(X.astype(np.float32))
        self.X = torch.unsqueeze(self.X,-1)
        self.y = torch.from_numpy(y.astype(np.float32))
        self.len = self.X.shape[0]

    def __getitem__(self, index):
        return self.X[index], self.y[index]

    def __len__(self):
        return self.len
```

[6]:
```
batch_size = 64

train_data = Data(training_input, training_output)
```

```
train_dataloader = DataLoader(dataset=train_data, batch_size=batch_size,␣
 ↪shuffle=True)

test_data = Data(testing_input, testing_output)
test_dataloader = DataLoader(dataset=test_data, batch_size=batch_size,␣
 ↪shuffle=True)
```

We use a Sigmoid Activation Function on the hidden layer, and a softmax on the output.

[7]:
```
class NeuralNetwork(nn.Module):
    def __init__(self):
        super().__init__()
        self.flatten = nn.Flatten()
        self.function_stack = nn.Sequential(
            nn.Linear(1, 10),
            nn.Sigmoid(),
            nn.Linear(10, 4),
            nn.Softmax(dim=1),
        )



    def forward(self, x):
        x = self.flatten(x)
        logits = self.function_stack(x)
        return logits
```

[8]:
```
device = 'cpu'
model = NeuralNetwork().to(device)
print(model)
```

```
NeuralNetwork(
  (flatten): Flatten(start_dim=1, end_dim=-1)
  (function_stack): Sequential(
    (0): Linear(in_features=1, out_features=10, bias=True)
    (1): Sigmoid()
    (2): Linear(in_features=10, out_features=4, bias=True)
    (3): Softmax(dim=1)
  )
)
```

[9]:
```
loss_fn = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), lr=1e-3)
```

[10]:
```
num_epochs = 500
loss_values = []
```

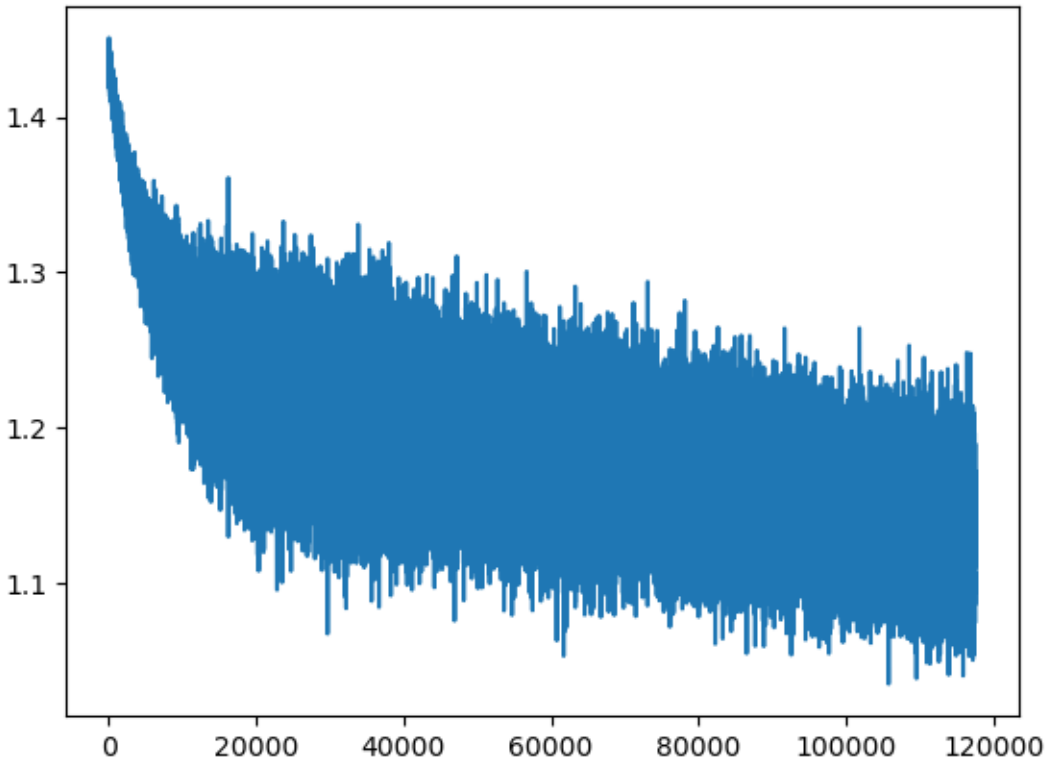
```
for epoch in tqdm(range(num_epochs)):
    for X, y in train_dataloader:
        # zero the parameter gradients
        optimizer.zero_grad()

        # forward + backward + optimize
        pred = model(X)
        loss = loss_fn(pred, y)
        loss_values.append(loss.item())
        loss.backward()
        optimizer.step()
```

```
100%|                    | 500/500 [01:10<00:00,  7.08it/s]
```

Below are the loss values & total error counts. The loss values are calculated for each batch of size 64, which is why there are so many data points. It can be seen that the error does converge somewhat, however it does not stabilise during training.

[11]:
```
plt.plot(loss_values)
```

[11]:
```
[<matplotlib.lines.Line2D at 0x7f77d592b3d0>]
```



[12]:
```
def count_errors_in_batch(pred, true):
    errors = 0
    for i,j in zip(pred, true):
        errors += 1 if np.argmax(i) != np.argmax(j) else 0
    return errors

def count_errors_in_batch(data, true):
    pred_guess, true_guess = np.argmax(model(data), axis=1), np.argmax(true,␣
 ↪axis=1)
    return sum(pred_guess != true_guess)
```

[13]:
```
with torch.no_grad():
    errors = 0
    for x, y in test_dataloader:
        errors += count_errors_in_batch(x, y)
    print(f'{errors} errors out of a total {len(test_data)}')
```

```
1832 errors out of a total 5000
```

Compared to the code written by hand to solve the problem, PyTorch actually has fewer errors (around 2500 with sigmoid when written myself). Why is not entirely clear, however it was quite a lot easier to implement in PyTorch, and the result turned out to be better.