Recurrent Neural Network Example

Xây dựng mạng RNN với PyTorch

Tổng quan về RNN

Tài liệu tham khảo:

- Long Short Term Memory (http://deeplearning.cs.cmu.edu/pdfs/Hochreiter97_lstm.pdf), Sepp Hochreiter & Jurgen Schmidhuber, Neural Computation 9(8): 1735-1780, 1997.

Tổng quan về bộ dữ liệu MNIST

Ví dụ này sử dụng bộ dữ liệu về chữ số viết tay MNIST. Bộ dữ liệu chữa 60k mẫu cho huấn luyện và 10k mẫu cho kiểm thử.

Để phân loại hình ảnh sử dụng RNN, chúng ta sẽ coi mỗi hàng là 1 chuỗi pixels. Bởi vì kích thước ảnh là 28*28px, ta sẽ sử lý 28 chuỗi của 28 timesteps cho tất cả các sample.

In [ ]:
```
from __future__ import absolute_import, division, print_function

import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
import torchvision
import torchvision.transforms as transforms
from torch.autograd import Variable
import numpy as np
```

In [ ]:
```
# Chuẩn bị dữ liệu
from tensorflow.keras.datasets import mnist
(x_train, y_train), (x_test, y_test) = mnist.load_data()
# Chuyển đổi sang định dạng float32.
x_train, x_test = np.array(x_train, np.float32), np.array(x_test, np.float3
x_train, x_test = x_train.reshape([-1, 28, 28]), x_test.reshape([-1, 28, 28
# Chuẩn hóa ảnh từ from [0, 255] to [0, 1].
x_train, x_test = x_train / 255., x_test / 255.
x_train, x_test, y_train, y_test = torch.from_numpy(x_train), torch.from_nu
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-
datasets/mnist.npz (https://storage.googleapis.com/tensorflow/tf-keras-da
tasets/mnist.npz)
11493376/11490434 [==============================] - 0s 0us/step
11501568/11490434 [==============================] - 0s 0us/step
```

In [ ]:
```
# x_train.shape
```

In [ ]:
```
trainloader = []
for (i,j) in zip(x_train, y_train):
    trainloader.append([i,j])
trainloader = torch.utils.data.DataLoader(trainloader, shuffle=True, batch_

testloader = []
for (i,j) in zip(x_test, y_test):
    testloader.append([i,j])
testloader = torch.utils.data.DataLoader(testloader, shuffle=True, batch_si
```

In [ ]:
```
# Create RNN Model
class RNNModel(nn.Module):
    def __init__(self, input_dim, hidden_dim, layer_dim, output_dim):
        super(RNNModel, self).__init__()

        # Number of hidden dimensions
        self.hidden_dim = None

        # Number of hidden layers
        self.layer_dim = None

        # RNN
        self.rnn = nn.RNN(None, None, None, batch_first=True, nonlinearity=

        # Readout layer
        self.fc = nn.Linear(hidden_dim, None)

    def forward(self, x):

        # Initialize hidden state with zeros
        h0 = Variable(torch.zeros(self.layer_dim, x.size(0), self.hidden_di

        # One time step
        out, hn = self.rnn(x, h0)
        out = self.fc(out[:, -1, :])
        return out
```

In [ ]:
```
# Create RNN
input_dim = 28    # input dimension
hidden_dim = 100  # hidden layer dimension
layer_dim = 1     # number of hidden layers
output_dim = 10   # output dimension

model = BiRNNModel(None, None, None, None)

# Cross Entropy Loss
import torch.optim as optim
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(model.parameters(), lr=0.001, momentum=0.9)
```

In [ ]:
```
for epoch in range(2):  # loop over the dataset multiple times

    running_loss = 0.0
    for i, data in enumerate(None, 0):
        # get the inputs; data is a list of [inputs, labels]
        inputs, labels = None
        # zero the parameter gradients
        optimizer.zero_grad()

        # forward + backward + optimize
        outputs = model(None)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        # print statistics
        running_loss += loss.item()
        if i % 1000 == 999:    # print every 2000 mini-batches
            print('[%d, %5d] loss: %.3f' %
                  (epoch + 1, i + 1, running_loss / 2000))
            running_loss = 0.0

print('Finished Training')
```

```
[1,  2000] loss: 2.237
[2,  2000] loss: 0.619
Finished Training
```

In [ ]:
```
correct = 0
total = 0
# since we're not training, we don't need to calculate the gradients for ou
with torch.no_grad():
    for data in testloader:
        images, labels = data
        # calculate outputs by running images through the network
        outputs = model(images)
        # the class with the highest energy is what we choose as prediction
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

print('Accuracy of the network on the 10000 test images: %d %%' % (
    100 * correct / total))
```

```
Accuracy of the network on the 10000 test images: 87 %
```

In [ ]:
```

```