2023_Lab6_Ex

March 2, 2023

1 Lab 6: Word Embeddings and RNNs

This lab covers the following topics: - Word encodings and embeddings. - Recurrent neural networks (RNNs). - Long-short term memory (LSTM).

1.1 Exercise 1: Word Embeddings

1.1.1 Exercise 1.1

Consider the limited vocabulary list below

[ ]:
```
vocab = ["the", "quick", "brown", "sly", "fox", "jumped", "over", "a", "lazy",␣
↪"dog", "and","found","lion"]
print(len(vocab))
```

```
13
```

Write a function to create **one hot encodings** of the words. The function maps each word to a vector, where it's location in the vocab list is indicated by 1 and all other entries are zero.

For example "quick" should map to a torch tensor of dimension 1 with entries [0,1,0....0].

Create an extra category for words not in the vocabulary

[ ]:
```
import torch
import torch.nn as nn
def one_hot_embedding(token, vocab):
    """
    Token should be a list of words or an indvidual word of length W.
    The output shouild be a torch tensor fo size W x (V+1) which gives the one␣
↪hot encoding for all W tokens
    """
    vector = torch.zeros(len(vocab))
    vector[vocab.index(token)] = 1
    return vector

one_hot_embedding("brown", vocab)
```

[ ]: tensor([0., 0., 1., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.])

1.1.2 Exercise 1.2

Create a `nn.module` that:

1. Takes in a single sentence (a python list).
2. Finds the one hot encoding of each word using the function created in exercise 1.1.
3. Finds the "word embedding" of each word that is D-dimensional using the `EmbedddingTable`.
4. Returns the average of the word embeddings as a torch vector of size D.

[ ]:
```
import torch.nn as nn

class MyWordEmbeddingBag(nn.Module):
    def __init__(self, dim):
        super(MyWordEmbeddingBag, self).__init__()

        self.EmbeddingTable = nn.Parameter(torch.randn(len(vocab)+1,dim))

    def forward(self, inputList):
        # Your answer here
        w_embed=[]
        for word in inputList:
          w_embed.append(self.EmbeddingTable[vocab.index(word)])

        return torch.mean(torch.stack(w_embed),dim=0)
```

1.1.3 Exercise 1.3

Instantiate the model with vectors of size D= 100 and forward pass the following sentences through your module

[ ]:
```
sent1 = ["the", "quick", "brown"]
sent2 = ["the", "sly", "fox", "jumped"]
sent3 = ["the", "dog", "found","a","lion"]

#Instantiate model
my_model = MyWordEmbeddingBag(100)

#forward pass sentences
assert(len(my_model(sent1))==100)
assert(len(my_model(sent2))==100)
assert(len(my_model(sent3))==100)
```

1.1.4 Exercise 1.4

Compute the euclidean distance between "fox" and "dog" using the randomly initialized embedding table in your model above.

Note: As this is randomly initialized, the distances will also be random in this case. However a trained model using word embeddings will often exhibit closer distances between related words, depending on objective.

[ ]:
```
fox = my_model(["fox"])
dog = my_model(["dog"])
print(((fox-dog)**2).sum())
```

```
tensor(176.4262, grad_fn=<SumBackward0>)
```

1.2 Exercise 2: Recurrent Neural Networks

We will experiment with recurrent networks using the MNIST dataset.

[ ]:
```
import torchvision
import torch
import torchvision.transforms as transforms

from torch.utils.data import Subset

### Hotfix for very recent MNIST download issue https://github.com/pytorch/
↪vision/issues/1938
from six.moves import urllib
opener = urllib.request.build_opener()
opener.addheaders = [('User-agent', 'Mozilla/5.0')]
urllib.request.install_opener(opener)
###

dataset = torchvision.datasets.MNIST('./', download=True, transform=transforms.
↪Compose([transforms.ToTensor()]), train=True)
train_indices = torch.arange(0, 10000)
train_dataset = Subset(dataset, train_indices)

dataset=torchvision.datasets.MNIST('./', download=True, transform=transforms.
↪Compose([transforms.ToTensor()]), train=False)
test_indices = torch.arange(0, 10000)
test_dataset = Subset(dataset, test_indices)
```

```
Downloading http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz to
./MNIST/raw/train-images-idx3-ubyte.gz

100%|      | 9912422/9912422 [00:00<00:00, 11887581.64it/s]

Extracting ./MNIST/raw/train-images-idx3-ubyte.gz to ./MNIST/raw

Downloading http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz to
./MNIST/raw/train-labels-idx1-ubyte.gz
```

```
100%|      | 28881/28881 [00:00<00:00, 21067077.19it/s]

Extracting ./MNIST/raw/train-labels-idx1-ubyte.gz to ./MNIST/raw

Downloading http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz to
./MNIST/raw/t10k-images-idx3-ubyte.gz

100%|      | 1648877/1648877 [00:00<00:00, 11942935.83it/s]

Extracting ./MNIST/raw/t10k-images-idx3-ubyte.gz to ./MNIST/raw

Downloading http://yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.gz to
./MNIST/raw/t10k-labels-idx1-ubyte.gz

100%|      | 4542/4542 [00:00<00:00, 8608463.07it/s]

Extracting ./MNIST/raw/t10k-labels-idx1-ubyte.gz to ./MNIST/raw
```

[ ]:
```
train_loader = torch.utils.data.DataLoader(train_dataset, batch_size=64,
                                           shuffle=True, num_workers=0)

test_loader = torch.utils.data.DataLoader(test_dataset, batch_size=16,
                                          shuffle=False, num_workers=0)
```

[ ]:
```

```

1.2.1 Exercise 2.1

Consider the following script (modified from https://github.com/yunjey/pytorch-tutorial/blob/master/tutorials/02-intermediate/recurrent_neural_network/main.py) which trains an RNN on the MNIST data.

Here we can consider each column of the image as an input for each step of the RNN. After 28 steps the model applies a linear layer + cross-entropy loss. We will use this to familiarize ourselves with the nn.RNN module and the nn.LSTM module.

First run the cell below

[ ]:
```
import torch
import torch.nn as nn
import torchvision
import torchvision.transforms as transforms


# Device configuration
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

# Hyper-parameters
```

```
sequence_length = 28
input_size = 28
hidden_size = 128
num_layers = 2
num_classes = 10
batch_size = 100
num_epochs = 2
learning_rate = 0.01


# Recurrent neural network (many-to-one)
class RNN(nn.Module):
    def __init__(self, input_size, hidden_size, num_layers, num_classes):
        super(RNN, self).__init__()
        self.hidden_size = hidden_size
        self.num_layers = num_layers
        self.rnn = nn.RNN(input_size, hidden_size, num_layers, batch_first=True)
        self.fc = nn.Linear(hidden_size, num_classes)

    def forward(self, x):
        # Set initial hidden and cell states
        h0 = torch.zeros(self.num_layers, x.size(0), self.hidden_size).to(device)

        # Forward propagate RNN
        out , _ = self.rnn(x, h0)  # out: tensor of shape (batch_size, seq_length, hidden_size)

        # Decode the hidden state of the last time step
        out = self.fc(out[:, -1, :])
        return out

model = RNN(input_size, hidden_size, num_layers, num_classes).to(device)


# Loss and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)

# Train the model
total_step = len(train_loader)
for epoch in range(num_epochs):
    for i, (images, labels) in enumerate(train_loader):
        images = images.reshape(-1, sequence_length, input_size).to(device)
        labels = labels.to(device)

        # Forward pass
```

```
        outputs = model(images)
        loss = criterion(outputs, labels)

        # Backward and optimize
        optimizer.zero_grad()
        loss.backward()
        #Gradient clipping
        #torch.nn.utils.clip_grad_norm_(model.parameters(), 0.2)

        optimizer.step()

        if (i+1) % 10 == 0:
            print ('Epoch [{}/{}], Step [{}/{}], Loss: {:.4f}'
                   .format(epoch+1, num_epochs, i+1, total_step, loss.item()))

# Test the model
model.eval()
with torch.no_grad():
    correct = 0
    total = 0
    for images, labels in test_loader:
        images = images.reshape(-1, sequence_length, input_size).to(device)
        labels = labels.to(device)
        outputs = model(images)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

    print('Test Accuracy of the model on the 10000 test images: {} %'.
 ↪format(100 * correct / total))
```

```
Epoch [1/2], Step [10/157], Loss: 2.2728
Epoch [1/2], Step [20/157], Loss: 2.5359
Epoch [1/2], Step [30/157], Loss: 2.3939
Epoch [1/2], Step [40/157], Loss: 2.3857
Epoch [1/2], Step [50/157], Loss: 2.3960
Epoch [1/2], Step [60/157], Loss: 2.4181
Epoch [1/2], Step [70/157], Loss: 2.3044
Epoch [1/2], Step [80/157], Loss: 2.3873
Epoch [1/2], Step [90/157], Loss: 2.4043
Epoch [1/2], Step [100/157], Loss: 2.4596
Epoch [1/2], Step [110/157], Loss: 2.3815
Epoch [1/2], Step [120/157], Loss: 2.4080
Epoch [1/2], Step [130/157], Loss: 2.4153
Epoch [1/2], Step [140/157], Loss: 2.4072
Epoch [1/2], Step [150/157], Loss: 2.5312
Epoch [2/2], Step [10/157], Loss: 2.3948
```

```
Epoch [2/2], Step [20/157], Loss: 2.3945
Epoch [2/2], Step [30/157], Loss: 2.4038
Epoch [2/2], Step [40/157], Loss: 2.4777
Epoch [2/2], Step [50/157], Loss: 2.3980
Epoch [2/2], Step [60/157], Loss: 2.3129
Epoch [2/2], Step [70/157], Loss: 2.3320
Epoch [2/2], Step [80/157], Loss: 2.2729
Epoch [2/2], Step [90/157], Loss: 2.4703
Epoch [2/2], Step [100/157], Loss: 2.3820
Epoch [2/2], Step [110/157], Loss: 2.2799
Epoch [2/2], Step [120/157], Loss: 2.2774
Epoch [2/2], Step [130/157], Loss: 2.4864
Epoch [2/2], Step [140/157], Loss: 2.4522
Epoch [2/2], Step [150/157], Loss: 2.4579
Test Accuracy of the model on the 10000 test images: 10.09 %
```

1.2.2 Exercise 2.2

Modify the above code (no need to create a new cell) to print the gradient norm of some of the parameters after backward in the the first minibatch.

Do this for the following weight parameter: model.rnn.weight_ih_l0.

[ ]:
```python
model.train()
for epoch in range(num_epochs):
    for i, (images, labels) in enumerate(train_loader):
        if i==0:
            images = images.reshape(-1, sequence_length, input_size).to(device)
            labels = labels.to(device)

            # Forward pass
            outputs = model(images)
            loss = criterion(outputs, labels)

            # Backward and optimize
            optimizer.zero_grad()
            loss.backward()
            #Gradient clipping
            #torch.nn.utils.clip_grad_norm_(model.parameters(), 0.2)
            print("grad norm:",model.rnn.weight_ih_l0.grad.norm(1))
            optimizer.step()
```

```
grad norm: tensor(7.5160e-05, device='cuda:0')
grad norm: tensor(7.7694e-05, device='cuda:0')
```

1.2.3 Exercise 2.3

Modify the code (in a new cell below) to use LSTM (and remove the gradient clipping) and rerun the code.

Note: This is essentially what is done in the original script linked above which you may check for reference or if you get stuck.

Run with LSTM and compare the accuracy and the gradient norm for weight_ih_l0 of the RNN.

[ ]:
```
class MY_LSTM(nn.Module):
    def __init__(self, input_size, hidden_size, num_layers, num_classes):
        super(MY_LSTM, self).__init__()
        self.hidden_size = hidden_size
        self.num_layers = num_layers
        self.lstm = nn.LSTM(input_size, hidden_size, num_layers,␣
↪batch_first=True)
        self.fc = nn.Linear(hidden_size, num_classes)

    def forward(self, x):
        # Set initial hidden and cell states
        h0 = torch.zeros(self.num_layers, x.size(0), self.hidden_size).
↪to(device)
        c0 = torch.zeros(self.num_layers, x.size(0), self.hidden_size).
↪to(device)

        # Forward propagate LSTM
        out, _ = self.lstm(x, (h0, c0))  # out: tensor of shape (batch_size,␣
↪seq_length, hidden_size)

        # Decode the hidden state of the last time step
        out = self.fc(out[:, -1, :])
        return out

model = MY_LSTM(input_size, hidden_size, num_layers, num_classes).to(device)

# Loss and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)

# Train the model
total_step = len(train_loader)
for epoch in range(num_epochs):
    for i, (images, labels) in enumerate(train_loader):
        images = images.reshape(-1, sequence_length, input_size).to(device)
        labels = labels.to(device)

        # Forward pass
        outputs = model(images)
```

```
        loss = criterion(outputs, labels)

        # Backward and optimize
        optimizer.zero_grad()
        loss.backward()
        #Gradient clipping
        #torch.nn.utils.clip_grad_norm_(model.parameters(), 0.2)

        optimizer.step()

        if (i+1) % 10 == 0:
            print ('Epoch [{}/{}], Step [{}/{}], Loss: {:.4f}'
                   .format(epoch+1, num_epochs, i+1, total_step, loss.item()))

# Test the model
model.eval()
with torch.no_grad():
    correct = 0
    total = 0
    for images, labels in test_loader:
        images = images.reshape(-1, sequence_length, input_size).to(device)
        labels = labels.to(device)
        outputs = model(images)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

    print('Test Accuracy of the model on the 10000 test images: {} %'.
 ↪format(100 * correct / total))
```

```
Epoch [1/2], Step [10/157], Loss: 2.1385
Epoch [1/2], Step [20/157], Loss: 1.2572
Epoch [1/2], Step [30/157], Loss: 1.1613
Epoch [1/2], Step [40/157], Loss: 1.2074
Epoch [1/2], Step [50/157], Loss: 1.0134
Epoch [1/2], Step [60/157], Loss: 1.0101
Epoch [1/2], Step [70/157], Loss: 0.6097
Epoch [1/2], Step [80/157], Loss: 0.8361
Epoch [1/2], Step [90/157], Loss: 0.7477
Epoch [1/2], Step [100/157], Loss: 0.3957
Epoch [1/2], Step [110/157], Loss: 0.7036
Epoch [1/2], Step [120/157], Loss: 0.4359
Epoch [1/2], Step [130/157], Loss: 0.5402
Epoch [1/2], Step [140/157], Loss: 0.5273
Epoch [1/2], Step [150/157], Loss: 0.2990
Epoch [2/2], Step [10/157], Loss: 0.6606
Epoch [2/2], Step [20/157], Loss: 0.5517
```

```
Epoch [2/2], Step [30/157], Loss: 0.3870
Epoch [2/2], Step [40/157], Loss: 0.3031
Epoch [2/2], Step [50/157], Loss: 0.2829
Epoch [2/2], Step [60/157], Loss: 0.2779
Epoch [2/2], Step [70/157], Loss: 0.4032
Epoch [2/2], Step [80/157], Loss: 0.3485
Epoch [2/2], Step [90/157], Loss: 0.1622
Epoch [2/2], Step [100/157], Loss: 0.0465
Epoch [2/2], Step [110/157], Loss: 0.2175
Epoch [2/2], Step [120/157], Loss: 0.2364
Epoch [2/2], Step [130/157], Loss: 0.1728
Epoch [2/2], Step [140/157], Loss: 0.1812
Epoch [2/2], Step [150/157], Loss: 0.1956
Test Accuracy of the model on the 10000 test images: 87.48 %
```

[ ]:

```python
model.train()
for epoch in range(num_epochs):
    for i, (images, labels) in enumerate(train_loader):
        if i==0:
            images = images.reshape(-1, sequence_length, input_size).to(device)
            labels = labels.to(device)

            # Forward pass
            outputs = model(images)
            loss = criterion(outputs, labels)

            # Backward and optimize
            optimizer.zero_grad()
            loss.backward()
            #Gradient clipping
            #torch.nn.utils.clip_grad_norm_(model.parameters(), 0.2)
            print("grad norm:",model.lstm.weight_ih_l0.grad.norm(1))
            optimizer.step()
```

```
grad norm: tensor(13.3063, device='cuda:0')
grad norm: tensor(14.0086, device='cuda:0')
```