40216952_2023_Lab6_Ex

March 5, 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

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
  """
  if type(token) == str:
    return one_hot_embedding([token], vocab)

  # Define the special "out-of-vocabulary" category
  oov_category = torch.zeros(len(vocab)+1)
  oov_category[-1] = 1

  # Create a dictionary mapping words to indices in the vocabulary
  vocab_indices = {word: i for i, word in enumerate(vocab)}
  #print (vocab_indices)

    # Encode each word using list comprehension
  encodings = torch.stack([torch.eye(len(vocab)+1)[vocab_indices[word]] if word␣
 ↪in vocab_indices else oov_category for word in token])

  return encodings

print (one_hot_embedding("the brown".split(), vocab))
```

```
tensor([[1., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
        [0., 0., 1., 0., 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
        encoding = one_hot_embedding(inputList, vocab)

        embed = nn.functional.embedding(encoding.argmax(dim=1), self.
 ↪EmbeddingTable)

        vector= embed.mean(dim=0)


        return vector
```

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.

```
[ ]: eucledean_distance = nn.PairwiseDistance(p=2)

     #Compute distance between sentences
     dist1 = eucledean_distance(my_model("fox"), my_model("dog"))

     print(dist1)
```

```
tensor(13.2394, grad_fn=<NormBackward1>)
```

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

100.0%

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

100.0%


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
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

100.0%
100.0%

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
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()

        #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.
        if i == 0:
            print("Gradient norm for model.rnn.weight_ih_l0: ", torch.
↪norm(model.rnn.weight_ih_l0.grad))

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

```python
# 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))
```

```
Gradient norm for model.rnn.weight_ih_l0:  tensor(0.0510, device='cuda:0')
Epoch [1/2], Step [10/157], Loss: 2.5554
Epoch [1/2], Step [20/157], Loss: 2.3050
Epoch [1/2], Step [30/157], Loss: 2.3172
Epoch [1/2], Step [40/157], Loss: 2.4606
Epoch [1/2], Step [50/157], Loss: 3.2187
Epoch [1/2], Step [60/157], Loss: 2.5557
Epoch [1/2], Step [70/157], Loss: 2.6076
Epoch [1/2], Step [80/157], Loss: 2.7772
Epoch [1/2], Step [90/157], Loss: 2.7901
Epoch [1/2], Step [100/157], Loss: 2.3962
Epoch [1/2], Step [110/157], Loss: 2.4115
Epoch [1/2], Step [120/157], Loss: 2.3288
Epoch [1/2], Step [130/157], Loss: 2.3941
Epoch [1/2], Step [140/157], Loss: 2.3687
Epoch [1/2], Step [150/157], Loss: 2.3273
Gradient norm for model.rnn.weight_ih_l0:  tensor(1.7137e-05, device='cuda:0')
Epoch [2/2], Step [10/157], Loss: 2.5240
Epoch [2/2], Step [20/157], Loss: 2.3542
Epoch [2/2], Step [30/157], Loss: 2.3781
Epoch [2/2], Step [40/157], Loss: 2.4032
Epoch [2/2], Step [50/157], Loss: 2.4610
Epoch [2/2], Step [60/157], Loss: 2.3707
Epoch [2/2], Step [70/157], Loss: 2.3899
Epoch [2/2], Step [80/157], Loss: 2.3505
Epoch [2/2], Step [90/157], Loss: 2.4332
Epoch [2/2], Step [100/157], Loss: 2.3872
Epoch [2/2], Step [110/157], Loss: 2.3527
Epoch [2/2], Step [120/157], Loss: 2.4164
Epoch [2/2], Step [130/157], Loss: 2.3904
Epoch [2/2], Step [140/157], Loss: 2.3096
Epoch [2/2], Step [150/157], Loss: 2.3693
Test Accuracy of the model on the 10000 test images: 11.35 %
```

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.

[ ]:
```
#Shown above
```

```
Gradient norm for model.rnn.weight_ih_l0:  tensor(2.4822e-06, 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 LSTM(nn.Module):
    def __init__(self, input_size, hidden_size, num_layers, num_classes):
        super(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 = 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()

        #print the gradient norm of some of the parameters after backward in␣
 ↪the the first minibatch. Do this for the following weight parameter: model.
 ↪lstm.weight_ih_l0.
        if i == 0:
            print("Gradient norm for model.lstm.weight_ih_l0: ", torch.
 ↪norm(model.lstm.weight_ih_l0.grad))

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

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))
```

```
Gradient norm for model.lstm.weight_ih_l0:  tensor(0.0113, device='cuda:0')
Epoch [1/2], Step [10/157], Loss: 2.1425
Epoch [1/2], Step [20/157], Loss: 2.0451
Epoch [1/2], Step [30/157], Loss: 1.6909
Epoch [1/2], Step [40/157], Loss: 1.4442
Epoch [1/2], Step [50/157], Loss: 1.6527
Epoch [1/2], Step [60/157], Loss: 1.3032
Epoch [1/2], Step [70/157], Loss: 0.9947
Epoch [1/2], Step [80/157], Loss: 1.4260
Epoch [1/2], Step [90/157], Loss: 0.9072
Epoch [1/2], Step [100/157], Loss: 0.5481
Epoch [1/2], Step [110/157], Loss: 0.8512
Epoch [1/2], Step [120/157], Loss: 0.6215
Epoch [1/2], Step [130/157], Loss: 0.4429
Epoch [1/2], Step [140/157], Loss: 0.3663
Epoch [1/2], Step [150/157], Loss: 0.5722
Gradient norm for model.lstm.weight_ih_l0:  tensor(0.5293, device='cuda:0')
Epoch [2/2], Step [10/157], Loss: 0.4629
Epoch [2/2], Step [20/157], Loss: 0.4683
Epoch [2/2], Step [30/157], Loss: 0.4384
Epoch [2/2], Step [40/157], Loss: 0.3925
Epoch [2/2], Step [50/157], Loss: 0.4820
Epoch [2/2], Step [60/157], Loss: 0.6155
Epoch [2/2], Step [70/157], Loss: 0.3477
Epoch [2/2], Step [80/157], Loss: 0.2657
Epoch [2/2], Step [90/157], Loss: 0.3310
Epoch [2/2], Step [100/157], Loss: 0.0984
Epoch [2/2], Step [110/157], Loss: 0.4163
Epoch [2/2], Step [120/157], Loss: 0.3081
Epoch [2/2], Step [130/157], Loss: 0.3922
Epoch [2/2], Step [140/157], Loss: 0.4304
Epoch [2/2], Step [150/157], Loss: 0.3853
Test Accuracy of the model on the 10000 test images: 91.24 %
```