Word Embeddings

In [2]:

```
import numpy as np
import math
import matplotlib.pyplot as plt
import torch
import torchvision
import torchvision.transforms as transforms
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
import torch.nn.parameter
```

Consider the limited vocabulary below

In [31]:

```
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 which maps each word to a vector where it's location in the vocab list is 1 and all other entries are zero. For example "quick" should map to a torch tensor of dimension 1 with entries 0,1,0.... Create an extra category for words not in the vocabulary

In [32]:

```
def one_hot_embedding(token, vocab):
    x=np.zeros(len(vocab))
    x[vocab.index(token)]=1
    return torch.tensor(x,dtype=torch.int8)
```

In [33]:

```
one_hot_embedding("jumped",vocab)
```

Out[33]:

```
tensor([0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0], dtype=torch.int8)
```

Create a nn.module that takes in a single sentence, encodes the words as embeddings and averages them. The input should be a python list and the output a torch vector of size D. For each word you will encode it as a D dimensional vector and average the final embeddings.

In [63]:

```
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),dim))

    def forward(self, inputList):
#         print(self.EmbeddingTable.shape)
        emb=[]
        for w in inputList:
            emb.append(self.EmbeddingTable[vocab.index(w)])
#             one_hot_embedding(w,vocab)
#         print(emb[0].shape)
        mean = torch.mean(torch.stack(emb), dim=0)
#         print(mean)
        return mean
```

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

In [64]:

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

#Instantiate model
my_model = MyWordEmbeddingBag(100)

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

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.

In [66]:

```
fox=my_model(["fox"])
dog=my_model(["dog"])
((fox-dog)**2).sum(axis=0)
```

Out[66]:

```
tensor(202.1752, grad_fn=<SumBackward1>)
```

Recurrent Neural Networks

We will experiment with recurrent networks using the MNIST dataset.

In [1]:

```
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/issue
s/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([t
ransforms.ToTensor()]), train=False)
test_indices = torch.arange(0, 10000)
test_dataset = Subset(dataset, test_indices)
```

In [3]:

```
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)
```

Consider the following script (modified from https://github.com/yunjey/pytorch-tutorial/blob/master/tutorials/02-intermediate/recurrent_neural_network/main.py (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. We will use this to familiarize ourselves with the nn.RNN module and the nn.LSTM module. First run the cell below

In [19]:

```
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, hidd
en_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)
#         print('predicted',predicted)
#         print('labels',labels)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

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

```
Epoch [1/2], Step [10/157], Loss: 2.8488
Epoch [1/2], Step [20/157], Loss: 2.4338
Epoch [1/2], Step [30/157], Loss: 2.3588
Epoch [1/2], Step [40/157], Loss: 2.3468
Epoch [1/2], Step [50/157], Loss: 2.3909
Epoch [1/2], Step [60/157], Loss: 2.4215
Epoch [1/2], Step [70/157], Loss: 2.4364
Epoch [1/2], Step [80/157], Loss: 2.4261
Epoch [1/2], Step [90/157], Loss: 2.4107
Epoch [1/2], Step [100/157], Loss: 2.3317
Epoch [1/2], Step [110/157], Loss: 2.4936
Epoch [1/2], Step [120/157], Loss: 2.3340
Epoch [1/2], Step [130/157], Loss: 2.3364
Epoch [1/2], Step [140/157], Loss: 2.4472
Epoch [1/2], Step [150/157], Loss: 2.4065
Epoch [2/2], Step [10/157], Loss: 2.4360
Epoch [2/2], Step [20/157], Loss: 2.4541
Epoch [2/2], Step [30/157], Loss: 2.4861
Epoch [2/2], Step [40/157], Loss: 2.4331
Epoch [2/2], Step [50/157], Loss: 2.4758
Epoch [2/2], Step [60/157], Loss: 2.3438
Epoch [2/2], Step [70/157], Loss: 2.3488
Epoch [2/2], Step [80/157], Loss: 2.3547
Epoch [2/2], Step [90/157], Loss: 2.4209
Epoch [2/2], Step [100/157], Loss: 2.2982
Epoch [2/2], Step [110/157], Loss: 2.4746
Epoch [2/2], Step [120/157], Loss: 2.4202
Epoch [2/2], Step [130/157], Loss: 2.3574
Epoch [2/2], Step [140/157], Loss: 2.4096
Epoch [2/2], Step [150/157], Loss: 2.3290
Test Accuracy of the model on the 10000 test images: 10.09 %
```

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.

In [21]:

```
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()
            print('RNN Model weight_ih_l0 grad norm:',model.rnn.weight_ih_l0.grad.data.
norm(1))
            optimizer.step()
```

```
RNN Model weight_ih_l0 grad norm: tensor(9.5919e-05)
RNN Model weight_ih_l0 grad norm: tensor(0.0001)
```

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 recurrent network

In [22]:

```
# Recurrent neural network (many-to-one)
class RNN_LSTM(nn.Module):
    def __init__(self, input_size, hidden_size, num_layers, num_classes):
        super(RNN_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_lengt
h, hidden_size)

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

model = RNN_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)
#         print('predicted',predicted)
#         print('labels',labels)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

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

```
Epoch [1/2], Step [10/157], Loss: 2.0252
Epoch [1/2], Step [20/157], Loss: 1.5607
Epoch [1/2], Step [30/157], Loss: 1.3003
Epoch [1/2], Step [40/157], Loss: 1.5051
Epoch [1/2], Step [50/157], Loss: 1.0838
Epoch [1/2], Step [60/157], Loss: 1.2134
Epoch [1/2], Step [70/157], Loss: 1.4657
Epoch [1/2], Step [80/157], Loss: 0.7524
Epoch [1/2], Step [90/157], Loss: 0.6947
Epoch [1/2], Step [100/157], Loss: 0.6530
Epoch [1/2], Step [110/157], Loss: 0.7962
Epoch [1/2], Step [120/157], Loss: 0.5565
Epoch [1/2], Step [130/157], Loss: 0.5382
Epoch [1/2], Step [140/157], Loss: 0.4951
Epoch [1/2], Step [150/157], Loss: 0.6407
Epoch [2/2], Step [10/157], Loss: 0.4220
Epoch [2/2], Step [20/157], Loss: 0.5531
Epoch [2/2], Step [30/157], Loss: 0.2820
Epoch [2/2], Step [40/157], Loss: 0.6886
Epoch [2/2], Step [50/157], Loss: 0.4063
Epoch [2/2], Step [60/157], Loss: 0.5552
Epoch [2/2], Step [70/157], Loss: 0.3824
Epoch [2/2], Step [80/157], Loss: 0.1164
Epoch [2/2], Step [90/157], Loss: 0.2096
Epoch [2/2], Step [100/157], Loss: 0.2115
Epoch [2/2], Step [110/157], Loss: 0.1607
Epoch [2/2], Step [120/157], Loss: 0.2391
Epoch [2/2], Step [130/157], Loss: 0.3427
Epoch [2/2], Step [140/157], Loss: 0.0819
Epoch [2/2], Step [150/157], Loss: 0.2998
Test Accuracy of the model on the 10000 test images: 91.95 %
```

In [23]:

```
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()
            print('LSTM Model weight_ih_l0 grad norm:',model.lstm.weight_ih_l0.grad.dat
a.norm(1))
            optimizer.step()
```

```
LSTM Model weight_ih_l0 grad norm: tensor(20.1510)
LSTM Model weight_ih_l0 grad norm: tensor(22.0459)
```

Standard RNNs (Recurrent Neural Networks) suffer from vanishing and exploding gradient problems.

LSTMs (Long Short Term Memory) deal with these problems by introducing new gates, such as input and forget gates, which allow for a better control over the gradient flow and enable better preservation of “long-range dependencies”.

The long range dependency in RNN is resolved by increasing the number of repeating layer in LSTM.

Test Accuracy of 10000 test images with RNN=10.09 %, LSTM=91.95 %

RNN Model weight_ih_l0 grad norm: tensor(9.5919e-05)

RNN Model weight_ih_l0 grad norm: tensor(0.0001)

LSTM Model weight_ih_l0 grad norm: tensor(20.1510)

LSTM Model weight_ih_l0 grad norm: tensor(22.0459)

In [ ]:

```
```