Statistical Natural Language Processing
Assignment 1

October 21, 2024

Victor Pekkari

Question 2

2 (a)

I had three layers all containing 300 neurons. My learning rate was set to 0.001. I had a weight decay of 1e-5. I drop out percentage of 20% after layer 2 and layer 3. And I used the embedding layer with 300 dimensions. These hyperparameters gave me over 80% accuracy on the development set. I used drop out and weight decay to prevent overfitting.
Weight decay prevents overfitting because it penlaises large weights which is common in overfitted networks. Without dropout a network can more easily fit outliers, but you need many neurons working together to fit outliers, thats why drop out prevents this. I also didn't want my network architecture to become too complex as it increases the risk of overfitting.

2 (b)

Using random initialized weights for the embedding layer made training a lot slower (timewise), and made it converge to a "stable" accuracy score in more epochs. It also made the model overfit to the training data more, than with the pre-trained embedding layer. I think the perfect combination would have been to start with the pre-initialized embedding layer, and then fine tuning it.

Question 3

3 (a)

$$P(dog|the) = \frac{1}{2} \tag{1}$$

$$P(cat|the) = \frac{1}{2} \tag{2}$$

3 (b)

$$w_{dog} = w_{cat} \implies v_{the}^t \cdot w_{dog} = v_{the}^t \cdot w_{cat} = z \tag{1}$$

$$P(X = \text{the} \mid Y = \text{dog}) = \frac{\exp(v_{\text{the}}^t \cdot w_{\text{dog}})}{\sum_{y'} \exp(v_{\text{the}}^t \cdot w_{\text{y'}})} \tag{2}$$

$$= \frac{\exp(z)}{\sum_{y'} \exp(v_{\text{the}}^t \cdot w_{\text{y'}})} = P(X = \text{the} \mid Y = \text{cat}) \tag{3}$$

$$\implies \forall v_{the} \in \mathbb{R}^2 \colon\ P(cat|the) = P(dog|the) = \frac{1}{2}$$

3 (c)

$$\{(the, dog), (dog, the), (the, cat), (cat, the), (a, dog), (dog, a), (a, cat), (cat, a)\}$$

3 (d)

our training examples tell us that the:

- "a" occurs in the same way with "dog" and "cat"
- "the" occurs in the same way with "dog" and "cat"
- "dog" occurs in the same way with "the" and "a"
- "cat" occurs in the same way with "the" and "a"

We want to treat "dog" and "cat" similarly since they always occur in the same context. We also want to treat "the" and "a" similarly because of the same reason.

Our desired probabilites are:

$$P(dog|the) \approx 0.5$$
$$P(cat|the) \approx 0.5$$
$$P(dog|a) \approx 0.5$$
$$P(cat|a) \approx 0.5$$

By using the similar vector-encodings, and context-encodings for similar words we get:

$$v_{the} = v_a = (1, 0)$$
$$v_{dog} = v_{cat} = (0, 1)$$

and the following context vectors:

$$w_{the} = w_a = (1, 0)$$
$$w_{cat} = w_{dog} = (0, 1)$$

PA1

October 20, 2024

CSE 156 PA1, Part 1

1 CSE 156 PA1, PyTorch Basics (25 points)

1.0.1 Due: October 18, 2024 at 10pm

IMPORTANT: After copying this notebook to your Google Drive, paste a link to it below. To get a publicly-accessible link, click the *Share* button at the top right, then click "Get shareable link" and copy the link.

Link: https://colab.research.google.com/drive/1VC_k4_pjtwhP8u947O2N_1Z_y_JIlfu1?usp=sharing

---

Notes:

Make sure to save the notebook as you go along.

Submission instructions are located at the bottom of the notebook.

The code should run fairly quickly (a couple of minutes at most even without a GPU), if it takes much longer than that, its likely that you have introduced an error.

2 Part 1: PyTorch Basics (25 Points)

We will use PyTorch, a machine learning framework for the rest of the course. The first part of this assigment focuses on PyTorch and how it is used for NLP. If you are new to PyTorch, it is highly recommended to work through the Stanford PyTorch Tutorial

##Question 1.1 (2.5 points)

In state-of-the-art NLP, words are represented by low-dimensional vectors, referred to as *embeddings*. When processing sequences such as sentences, movie, reviews, or entire paragraphs, word embeddings are used to compute a vector representation of the sequence, denoted by x. In the cell below, the embeddings for the words in the sequence "I like NLP" are provided. Your task is to combine these embeddings into a single vector representation x, using element-wise vector addition. This method is a simple way to obtain a sequence representation, namely, it is a *continuous bag-of-words (BoW) representation* of a sequence.

[7]:
```
import torch
torch.set_printoptions(sci_mode=False)
```

```python
# Seed the random number generator for reproducibility
torch.manual_seed(0)

input_sequence = 'I like NLP'

# Initialize an embedding matrix
# We have a vocabulary of 5 words, each represented by a 10-dimensional␣
 ↪embedding vector.
embeddings = torch.nn.Embedding(num_embeddings=5, embedding_dim=10)
vocab = {'I': 0, 'like': 1, 'NLP': 2, 'classifiers': 3, '.': 4}


# Convert the words to integer indices. These indices will be used to
# retrieve the corresponding embeddings from the embedding matrix.
# In PyTorch, operations are performed on Tensor objects, so we need to  convert
# the list of indices to a LongTensor.
indices = torch.LongTensor([vocab[w] for w in input_sequence.split()])

input_sequence_embs = embeddings(indices)
#print(embeddings(torch.LongTensor([1])))

print('sequence embedding tensor size: ', input_sequence_embs.size())

# The input_sequence_embs tensor contains the embeddings for each word in the␣
 ↪input sequence.
# The next step is to aggregate these embeddings into a single vector␣
 ↪representation.
# You will use  element-wise addition to do this.
# Write the code to add the embeddings element-wise and store the result in the␣
 ↪variable "x".

print(input_sequence_embs)
### YOUR CODE HERE!
# Replace with the actual computation
x = torch.sum(input_sequence_embs, dim=0)  # Sum along the first dimension

### DO NOT MODIFY THE LINE BELOW
print('input sequence embedding sum (continuous BoW): ', x)
```

```
sequence embedding tensor size:  torch.Size([3, 10])
tensor([[-1.1258, -1.1524, -0.2506, -0.4339,  0.8487,  0.6920, -0.3160, -2.1152,
          0.3223, -1.2633],
        [ 0.3500,  0.3081,  0.1198,  1.2377,  1.1168, -0.2473, -1.3527, -1.6959,
          0.5667,  0.7935],
        [ 0.5988, -1.5551, -0.3414,  1.8530,  0.7502, -0.5855, -0.1734,  0.1835,
          1.3894,  1.5863]], grad_fn=<EmbeddingBackward0>)
input sequence embedding sum (continuous BoW):  tensor([-0.1770, -2.3993,
```

```
-0.4721,  2.6568,  2.7157, -0.1408, -1.8421, -3.6277,
         2.2783,  1.1165], grad_fn=<SumBackward1>)
```

##Question 1.2 (2.5 points)

2.0.1 Answer:

Element-wise addition doesn't treat the order of words. It represents: - I like cars the same as - cars like I (This doesn't make sense grammaticly) that is the biggest weakness with element wise addition

---

##Question 1.3 (5 points) The softmax function is used in nearly all the neural network architectures we will look at in this course. The softmax is computed on an n-dimensional vector $< x_1, x_2, \ldots, x_n >$ as $\text{softmax}(x_i) = \frac{e^{x_i}}{\sum_{1 \leq j \leq n} e^{x_j}}$. Given the sequence representation x we just computed, we can use the softmax function in combination with a linear projection using a matrix W to transform x into a probability distribution p over the next word, expressed as $p = \text{softmax}(Wx)$. Let us look at this in the cell below:

[8]:
```
# Initialize a random matrix W of size 10x5. This will serve as the weight␣
 ↪matrix
# for the linear projection of the vector x into a 5-dimensional space.
W = torch.rand(10, 5)


# Project the vector x to a 5-dimensional space using the matrix W. This␣
 ↪projection is achieved through
# matrix multiplication. After the projection, apply the softmax function to␣
 ↪the result,
# which converts the 5-dimensional projected vector into a probability␣
 ↪distribution.
# You can find the softmax function in PyTorch's  API (torch.nn.functional.
 ↪softmax).
# Store the resulting probability distribution in the variable "probs".

### YOUR CODE HERE
# Replace with the actual computation
projected_x = torch.matmul(x, W)
probs = torch.nn.functional.softmax(projected_x, dim=0) # This should be␣
 ↪replaced with the actual computation


### DO NOT MODIFY THE BELOW LINE!
print('probability distribution', probs)
```

```
probability distribution tensor([0.0718, 0.0998, 0.1331, 0.6762, 0.0191],
grad_fn=<SoftmaxBackward0>)
```

##Question 1.4 (10 points)

In the example so far, we focused on a single sequence ("I like NLP"). However, in practical applications, it's common to process multiple sequences simultaneously. This practice, known as *batching*, allows for more efficient use of GPU parallelism. In batching, each sequence is considered an example within a larger batch

For this question, you will redo the previous computation, but with a batch of two sequences instead of just one. The final output of this cell should be a 2x5 matrix, where each row represents a probability distribution for a sequence. **Important: Avoid using loops in your solution, as you will lose points**. The code should be fully vectorized.

[9]:
```
import torch
import torch.nn.functional as F

# For this example, we replicate our previous sequence indices to create a␣
 ↪simple batch.
# Normally, each example in the batch would be different.
batch_indices = torch.cat(2 * [indices]).reshape((2, 3))
batch_embs = embeddings(batch_indices)
print('Batch embedding tensor size: ', batch_embs.size())

# To process the batch, follow these steps:
# Step 1: Aggregate the embeddings for each example in the batch into a single␣
 ↪representation.
# This is done through element-wise addition. Use torch.sum with the␣
 ↪appropriate 'dim' argument
# to sum across the sequence length (not the batch dimension).
aggregated_batch_embs = torch.sum(batch_embs, dim=1)

# Step 2: Project each aggregated representation into a 5-dimensional space␣
 ↪using the matrix W.
# This involves matrix multiplication, ensuring the resulting batch has the␣
 ↪shape 2x5.
W = torch.rand(10, 5)

# Step 3: Apply the softmax function to the projected representations to obtain␣
 ↪probability distributions.
# Each row in the output matrix should sum to 1, representing a probability␣
 ↪distribution for each batch example.
projected_batch = torch.matmul(aggregated_batch_embs, W)

### YOUR CODE HERE
# Replace with the actual computation
batch_probs = F.softmax(projected_batch, dim=1)

### DO NOT MODIFY THE BELOW LINE
print("Batch probability distributions:", batch_probs)
```

```
Batch embedding tensor size:  torch.Size([2, 3, 10])
```

```
Batch probability distributions: tensor([[0.0655, 0.0028, 0.0023, 0.1176,
0.8118],
        [0.0655, 0.0028, 0.0023, 0.1176, 0.8118]], grad_fn=<SoftmaxBackward0>)
```

##Question 1.5 (5 points)

When processing a text sequence, how should the system handle words that are not present in the existing vocabulary? In the current implementation, the presence of such out-of-vocabulary words causes the code to fail, as in the cell below. To address this issue, a simple solution is to use the special token `<UNK>`, added to the vocabulary to serve as a placeholder for any unknown words.

Modify the indexing function to ensure that it checks each word against the known vocabulary and substitutes any out-of-vocabulary words with the `<UNK>` token. Make sure not to add any new words to the vocabulary except for the `<UNK>` token. Don't forget to adjust the embedding table.

[10]:
```
import torch

torch.set_printoptions(sci_mode=False)
# Seed the random number generator for reproducibility
torch.manual_seed(0)

input_sequence = 'I like linear'


# Initialize an embedding matrix
# We have a vocabulary of 5 words, each represented by a 10-dimensional␣
 ↪embedding vector.
embeddings = torch.nn.Embedding(num_embeddings=6, embedding_dim=10)
vocab = {'I': 0, 'like': 1, 'NLP': 2, 'classifiers': 3, '.': 4, '<UNK>': 5}


#indices = torch.LongTensor([vocab[w] for w in input_sequence.split()]) ###␣
 ↪MODIFY THIS INDEXING

indices = torch.LongTensor([vocab.get(w, vocab['<UNK>']) for w in␣
 ↪input_sequence.split()])

input_sequence_embs = embeddings(indices)
print('sequence embedding tensor size: ', input_sequence_embs.size())
```

```
sequence embedding tensor size:  torch.Size([3, 10])
```

Submission Instructions

1. Click the Save button at the top of the Jupyter Notebook.
2. Select Cell -> All Output -> Clear. This will clear all the outputs from all cells (but will keep the content of all cells).
3. Select Cell -> Run All. This will run all the cells in order, and will take several minutes.

4. Once you've rerun everything, convert the notebook to PDF, you can use tools such as nbconvert, which requires first downloading the ipynb to your local machine, and then running "nbconvert" . (If you have trouble using nbconvert, you can also save the webpage as pdf. Make sure all your solutions are displayed in the pdf, it's okay if the provided codes get cut off because lines are not wrapped in code cells).
5. Look at the PDF file and make sure all your solutions are there, displayed correctly. The PDF is the only thing your graders will see!
6. Submit your PDF on Gradescope, along with Parts 2 and 3 of the Assignment as described in the PA handhout.