word_embeddings_tutorial

August 23, 2019

[1]:
```
# import jovian
# jovian.commit()
```

[1]:
```
%matplotlib inline
```

1 Word Embeddings: Encoding Lexical Semantics

Due to a LaTeX render issue with Jupyter, the supplementary material has been uploaded separately, please check here

[2]:
```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim

torch.manual_seed(1)
```

[2]:
```
<torch._C.Generator at 0x7f60840429f0>
```

[3]:
```
word_to_ix = {"hello": 0, "world": 1}
embeds = nn.Embedding(2, 5)  # 2 words in vocab, 5 dimensional embeddings
lookup_tensor = torch.tensor([word_to_ix["hello"]], dtype=torch.long)
hello_embed = embeds(lookup_tensor)
print(hello_embed)
```

```
tensor([[ 0.6614,  0.2669,  0.0617,  0.6213, -0.4519]],
       grad_fn=<EmbeddingBackward>)
```

An Example: N-Gram Language Modeling ~~~~~~

Due to a LaTeX render issue with Jupyter, the supplementary material has been uploaded separately, please check here

[4]:
```
CONTEXT_SIZE = 2
EMBEDDING_DIM = 10
# We will use Shakespeare Sonnet 2
test_sentence = """When forty winters shall besiege thy brow,
And dig deep trenches in thy beauty's field,
Thy youth's proud livery so gazed on now,
Will be a totter'd weed of small worth held:
```

```
Then being asked, where all thy beauty lies,
Where all the treasure of thy lusty days;
To say, within thine own deep sunken eyes,
Were an all-eating shame, and thriftless praise.
How much more praise deserv'd thy beauty's use,
If thou couldst answer 'This fair child of mine
Shall sum my count, and make my old excuse,'
Proving his beauty by succession thine!
This were to be new made when thou art old,
And see thy blood warm when thou feel'st it cold.""".split()
# we should tokenize the input, but we will ignore that for now
# build a list of tuples.  Each tuple is ([ word_i-2, word_i-1 ], target word)
trigrams = [([test_sentence[i], test_sentence[i + 1]], test_sentence[i + 2])
            for i in range(len(test_sentence) - 2)]
# print the first 3, just so you can see what they look like
print(trigrams[:3])

vocab = set(test_sentence)
word_to_ix = {word: i for i, word in enumerate(vocab)}


class NGramLanguageModeler(nn.Module):

    def __init__(self, vocab_size, embedding_dim, context_size):
        super(NGramLanguageModeler, self).__init__()
        self.embeddings = nn.Embedding(vocab_size, embedding_dim)
        self.linear1 = nn.Linear(context_size * embedding_dim, 128)
        self.linear2 = nn.Linear(128, vocab_size)

    def forward(self, inputs):
        embeds = self.embeddings(inputs).view((1, -1))
        out = F.relu(self.linear1(embeds))
        out = self.linear2(out)
        log_probs = F.log_softmax(out, dim=1)
        return log_probs


losses = []
loss_function = nn.NLLLoss()
model = NGramLanguageModeler(len(vocab), EMBEDDING_DIM, CONTEXT_SIZE)
optimizer = optim.SGD(model.parameters(), lr=0.001)

for epoch in range(10):
    total_loss = 0
    for context, target in trigrams:
```

```
        # Step 1. Prepare the inputs to be passed to the model (i.e, turn the␣
 ↪words
        # into integer indices and wrap them in tensors)
        context_idxs = torch.tensor([word_to_ix[w] for w in context],␣
 ↪dtype=torch.long)

        # Step 2. Recall that torch *accumulates* gradients. Before passing in a
        # new instance, you need to zero out the gradients from the old
        # instance
        model.zero_grad()

        # Step 3. Run the forward pass, getting log probabilities over next
        # words
        log_probs = model(context_idxs)

        # Step 4. Compute your loss function. (Again, Torch wants the target
        # word wrapped in a tensor)
        loss = loss_function(log_probs, torch.tensor([word_to_ix[target]],␣
 ↪dtype=torch.long))

        # Step 5. Do the backward pass and update the gradient
        loss.backward()
        optimizer.step()

        # Get the Python number from a 1-element Tensor by calling tensor.item()
        total_loss += loss.item()
    losses.append(total_loss)
print(losses)  # The loss decreased every iteration over the training data!
```

```
[(['When', 'forty'], 'winters'), (['forty', 'winters'], 'shall'), (['winters',
'shall'], 'besiege')]
[521.2105088233948, 518.474401473999, 515.756929397583, 513.059576511383,
510.38018465042114, 507.7175302505493, 505.07210540771484, 502.4404773712158,
499.82227659225464, 497.2171709537506]
```

1.0.1 Exercise: Computing Word Embeddings: Continuous Bag-of-Words

Due to a LaTeX render issue with Jupyter, the supplementary material has been uploaded separately, please check here

[7]:
```
def make_context_vector(context, word_to_ix):
    idxs = [word_to_ix[w] for w in context]
    return torch.tensor(idxs, dtype=torch.long)

def get_index_of_max(input):
    index = 0
    for i in range(1, len(input)):
        if input[i] > input[index]:
```

```
            index = i
    return index

def get_max_prob_result(input, ix_to_word):
    return ix_to_word[get_index_of_max(input)]


CONTEXT_SIZE = 2  # 2 words to the left, 2 to the right
EMBEDDING_DIM = 100
```

[9]:
```
CONTEXT_SIZE = 2  # 2 words to the left, 2 to the right
raw_text = """We are about to study the idea of a computational process.
Computational processes are abstract beings that inhabit computers.
As they evolve, processes manipulate other abstract things called data.
The evolution of a process is directed by a pattern of rules
called a program. People create programs to direct processes. In effect,
we conjure the spirits of the computer with our spells.""".split()

# By deriving a set from `raw_text`, we deduplicate the array
vocab = set(raw_text)
vocab_size = len(vocab)

word_to_ix = {word: i for i, word in enumerate(vocab)}
data = []
for i in range(2, len(raw_text) - 2):
    context = [raw_text[i - 2], raw_text[i - 1],
               raw_text[i + 1], raw_text[i + 2]]
    target = raw_text[i]
    data.append((context, target))
print(data[:5])


class CBOW(nn.Module):

    def __init__(self, vocab_size, embedding_dim, context_size):
        super(CBOW, self).__init__()
        self.embeddings = nn.Embedding(vocab_size, embedding_dim)
        self.linear1 = nn.Linear(embedding_dim, 128)
        self.linear2 = nn.Linear(128, vocab_size)

    def forward(self, inputs):
        # Here instead of cocatenating ebeddings we're going to add the␣
 ↪embeddings of the four context words
        embeds = self.embeddings(inputs).sum(0).view((1, -1))
        #print("\nembeds.shape: ")
        #print(embeds.shape) # torch.Size([1, 100])
        out1 = F.relu(self.linear1(embeds))
```

```
        out2 = self.linear2(out1)
        log_probs = F.log_softmax(out2, dim=1)
        return log_probs

# create your model and train.  here are some functions to help you make
# the data ready for use by your module
losses = []
loss_function = nn.NLLLoss()
model = CBOW(vocab_size, EMBEDDING_DIM, CONTEXT_SIZE)
optimizer = optim.SGD(model.parameters(), lr=0.001)
for epoch in range(10):
    total_loss = 0
    for context, target in data:
        context_to_idxs = make_context_vector(context, word_to_ix)
        model.zero_grad()
        log_probs = model(context_to_idxs)
        loss = loss_function(log_probs, torch.tensor([word_to_ix[target]],␣
 ↪dtype=torch.long))
        loss.backward()
        optimizer.step()
        total_loss += loss.item()
    losses.append(total_loss)

print("\nLosses: ")
print(losses)


def make_context_vector(context, word_to_ix):
    idxs = [word_to_ix[w] for w in context]
    return torch.tensor(idxs, dtype=torch.long)

make_context_vector(data[0][0], word_to_ix)  # example
```

```
[(['We', 'are', 'to', 'study'], 'about'), (['are', 'about', 'study', 'the'],
'to'), (['about', 'to', 'the', 'idea'], 'study'), (['to', 'study', 'idea',
'of'], 'the'), (['study', 'the', 'of', 'a'], 'idea')]

Losses:
[233.97173166275024, 224.90228986740112, 216.24900197982788, 207.93129968643188,
199.89470863342285, 192.14736413955688, 184.6737837791443, 177.4489290714264,
170.47974967956543, 163.71181631088257]
```

[9]:
```
tensor([23, 45, 13,  0])
```

[ ]:
```

```