3. N-Gram Model

So far we only considered latent variable models. Let's see what happens if we use a regular -gram model and an autoregressive setting. That is, we aim to predict the next character given the current characters one character at a time. For this implement the following:

1. Split data into (x, y) pairs as before, just that we now use very short subsequences, e.g. only 5 characters. That is, `But Brutus` turns into the tuples (`(But B, r)` , `(ut Br, u)` , `(t Bru, t)` , `( Brut, u)` , `(Brutu, s)`).
2. Use one-hot encoding for each character separately and combine them all.
   - In one case use a sequential encoding to obtain an embedding proportional to the length of the sequence.
   - Use a bag of characters encoding that sums over all occurrences.
3. Implement an MLP with one hidden layer and 256 hidden units.
4. Train it to output the next character.

How accurate is the model? How does the number of operations and weights compare to an RNN, a GRU and an LSTM discussed above?

0. Preparation

In [1]:
```python
import urllib3
import collections
import re
shakespeare = 'http://www.gutenberg.org/files/100/100-0.txt'

http = urllib3.PoolManager()
text = http.request('GET', shakespeare).data.decode('utf-8')
raw_dataset = ' '.join(re.sub('[^A-Za-z]+', ' ', text).lower().split())

print('number of characters: ', len(raw_dataset))
print(raw_dataset[0:70])
```

```
number of characters:  5032359
project gutenberg s the complete works of william shakespeare by willi
```

In [2]:
```
idx_to_char = list(set(raw_dataset))
char_to_idx = dict([(char, i) for i, char in enumerate(idx_to_char)])
vocab_size = len(char_to_idx)
corpus_indices = [char_to_idx[char] for char in raw_dataset]
sample = corpus_indices[:20]
print('chars:', ''.join([idx_to_char[idx] for idx in sample]))
print('indices:', sample)

train_indices = corpus_indices[:-100000]
test_indices = corpus_indices[-100000:]
```

```
chars: project gutenberg s
indices: [19, 11, 24, 10, 20, 15, 23, 25, 5, 6, 23, 20, 17, 4, 20, 11, 5, 25, 13, 25]
```

In [3]:
```
import d2l
import math
from mxnet import autograd, gluon, init, nd
from mxnet.gluon import loss as gloss, nn, rnn
import time
import mxnet
```

1. Encoding

In [4]:
```
ctx = d2l.try_gpu()
embedding = 5 # embedding dimension for autoregressive model
T = len(raw_dataset)
features = nd.zeros((T-embedding, embedding), ctx = ctx)
```

In [5]:
```
for i in range(embedding):
    features[:,i] = corpus_indices[i:T-embedding+i]
labels = nd.array(corpus_indices[embedding:],ctx =ctx)
```

In [6]:
```
predict_dataset = 'but brutus is an honorable man'
corpus_indices_predict = [char_to_idx[char] for char in predict_dataset]
T_predict = len(predict_dataset)
predict_features = nd.zeros((T_predict-embedding, embedding),ctx = ctx)
```

In [7]:
```
for i in range(embedding):
    predict_features[:,i] = corpus_indices_predict[i:T_predict-embedding+i]
#predict_labels = nd.array(corpus_indices_predict[embedding:],ctx = ctx)
```

2 .One-Hot

2.1 Sequence Encoding is implemented as concating all the one-hot sequence together.

To explain, we have vocab_size of 27. Then we encode each character into a (1, 27) vector. Then we concat them together into (1, 135) vector with 5 characters of sequence.

In [10]:
```
def to_onehot_length(X, vocab_size):
    inputs = [nd.one_hot(x, vocab_size) for x in X]
    matrix_size = inputs[0].size
    for i in range(len(inputs)):
        inputs[i] = inputs[i].reshape(1,matrix_size)
    return inputs
```

In [11]:
```
inputs = to_onehot_length(features[:2000000], vocab_size)
print('finished')
#outputs = to_onehot_output(labels, vocab_size)
```

```
finished
```

2.2 Sum the one-hot result together.

To explain, we have vocab_size of 27. Then we encode each character into a (1, 27) vector. Then we sum them together along axis = 0. The one-hot vector is still (1,27).

In [8]:
```
def to_onehot_sum(X, vocab_size):
    inputs = [nd.one_hot(x, vocab_size) for x in X]
    matrix_size = inputs[0].size
    for i in range(len(inputs)):
        inputs[i] = nd.sum(inputs[i],axis = 0).reshape(1, vocab_size)
    return inputs
```

In [10]:
```
#print('finished')
inputs_sum = to_onehot_sum(features[:2000000], vocab_size)
print('finished')
```

```
finished
```

3. MLP

3.1 Sequence Encoding proportional to its length

In [12]:
```
net = gluon.nn.Sequential()
net.add(gluon.nn.Dense(256, activation='relu'))
net.add(gluon.nn.Dense(vocab_size))
net.initialize(init.Xavier(),ctx = ctx)
loss = gloss.SoftmaxCrossEntropyLoss()
```

In [13]:
```
#ntrain = 4932354
ntrain = 1800000
train_data = gluon.data.ArrayDataset(inputs[:ntrain], labels[:ntrain])
test_data  = gluon.data.ArrayDataset(inputs[ntrain:2000000], labels[ntrain:2000000])

batch_size = 25
trainer = gluon.Trainer(net.collect_params(), 'sgd', {'learning_rate': 0.5})
num_epochs = 20

train_iter = gluon.data.DataLoader(train_data, batch_size, shuffle=True)
test_iter = gluon.data.DataLoader(test_data, batch_size, shuffle=True)
```

In [14]:
```
d2l.train_ch5(net, train_iter, test_iter, batch_size, trainer, ctx,
              num_epochs)
```

```
training on gpu(0)
epoch 1, loss 1.6389, train acc 0.495, test acc 0.513, time 220.7 sec
epoch 2, loss 1.5242, train acc 0.525, test acc 0.523, time 195.8 sec
epoch 3, loss 1.4996, train acc 0.532, test acc 0.526, time 192.7 sec
epoch 4, loss 1.4868, train acc 0.534, test acc 0.530, time 190.0 sec
epoch 5, loss 1.4784, train acc 0.537, test acc 0.531, time 186.3 sec
epoch 6, loss 1.4728, train acc 0.538, test acc 0.530, time 184.4 sec
epoch 7, loss 1.4682, train acc 0.539, test acc 0.528, time 182.2 sec
epoch 8, loss 1.4647, train acc 0.540, test acc 0.534, time 185.2 sec
epoch 9, loss 1.4624, train acc 0.540, test acc 0.530, time 181.0 sec
epoch 10, loss 1.4600, train acc 0.541, test acc 0.536, time 184.3 sec
epoch 11, loss 1.4579, train acc 0.541, test acc 0.535, time 176.1 sec
epoch 12, loss 1.4563, train acc 0.542, test acc 0.536, time 179.2 sec
epoch 13, loss 1.4550, train acc 0.542, test acc 0.538, time 178.5 sec
epoch 14, loss 1.4538, train acc 0.542, test acc 0.534, time 175.7 sec
epoch 15, loss 1.4523, train acc 0.542, test acc 0.534, time 178.5 sec
epoch 16, loss 1.4512, train acc 0.543, test acc 0.535, time 171.8 sec
epoch 17, loss 1.4503, train acc 0.543, test acc 0.536, time 178.7 sec
epoch 18, loss 1.4493, train acc 0.543, test acc 0.538, time 176.7 sec
epoch 19, loss 1.4485, train acc 0.543, test acc 0.538, time 173.8 sec
epoch 20, loss 1.4476, train acc 0.544, test acc 0.536, time 171.6 sec
```

In [18]:
```
net_2 = gluon.nn.Sequential()
net_2.add(gluon.nn.Dense(256, activation='relu'))
net_2.add(gluon.nn.Dense(vocab_size))
net_2.initialize(init.Xavier(),ctx = ctx)

loss = gloss.SoftmaxCrossEntropyLoss()
```

In [20]:
```
ntrain = 180000
batch_size = 30
num_epochs = 15
trainer_2 = gluon.Trainer(net_2.collect_params(), 'sgd', {'learning_rate': 0.4})

train_data_2 = gluon.data.ArrayDataset(inputs_sum[:ntrain], labels[:ntrain])
test_data_2  = gluon.data.ArrayDataset(inputs_sum[ntrain:2000000], labels[ntrain:2000000])

train_iter_2 = gluon.data.DataLoader(train_data_2, batch_size, shuffle=True)
test_iter_2 = gluon.data.DataLoader(test_data_2, batch_size, shuffle=True)
```

In [21]:
```
d2l.train_ch5(net_2, train_iter_2, test_iter_2, batch_size, trainer_2, ctx,
              num_epochs)
```

```
training on gpu(0)
epoch 1, loss 2.5489, train acc 0.257, test acc 0.254, time 78.1 sec
epoch 2, loss 2.4526, train acc 0.288, test acc 0.269, time 77.7 sec
epoch 3, loss 2.4136, train acc 0.300, test acc 0.274, time 77.0 sec
epoch 4, loss 2.3899, train acc 0.308, test acc 0.281, time 77.2 sec
epoch 5, loss 2.3741, train acc 0.312, test acc 0.282, time 76.6 sec
epoch 6, loss 2.3627, train acc 0.315, test acc 0.284, time 79.7 sec
epoch 7, loss 2.3535, train acc 0.318, test acc 0.277, time 79.0 sec
epoch 8, loss 2.3460, train acc 0.319, test acc 0.284, time 79.8 sec
epoch 9, loss 2.3400, train acc 0.321, test acc 0.284, time 77.7 sec
epoch 10, loss 2.3342, train acc 0.323, test acc 0.287, time 77.4 sec
epoch 11, loss 2.3298, train acc 0.324, test acc 0.287, time 75.7 sec
epoch 12, loss 2.3247, train acc 0.325, test acc 0.289, time 77.1 sec
epoch 13, loss 2.3214, train acc 0.326, test acc 0.287, time 75.1 sec
epoch 14, loss 2.3180, train acc 0.327, test acc 0.279, time 74.5 sec
epoch 15, loss 2.3154, train acc 0.328, test acc 0.286, time 73.5 sec
```

In [15]:
```python
# This function is saved in the d2l package for future use
def predict_ngram_length(prefix, num_chars, net, vocab_size, ctx):

    output = [char_to_idx[prefix[0]]]
    for t in range(num_chars + len(prefix) - 6):
        if t < len(prefix) - 1:
            output.append(char_to_idx[prefix[t + 1]])
        else:
            out = nd.array(output[-5:],ctx = ctx)
            features = nd.zeros((1,135), ctx = ctx)

            for i in range(len(out)):
                features[:,27*i:27+27*i] = nd.one_hot(out[i], vocab_size)

            Y = net(features).as_in_context(ctx)
            output.append(int(Y[0].argmax(axis=0).asscalar()))

    return ''.join([idx_to_char[i] for i in output])
```

In [17]:
```python
prefix = 'but brutus is an honorable man'
predict_ngram_length(prefix, 200, net, vocab_size, ctx)
```

Out[17]:
```
'but brutus is an honorable man and the so much a come the so much a come the so much a come the so much a come the so muc
h a come the so much a come the so much a come the so much a come the so much a come the so much a come '
```

In [22]:
```
def predict_ngram_sum(prefix, num_chars, net, vocab_size, ctx):

    output = [char_to_idx[prefix[0]]]
    for t in range(num_chars + len(prefix) - 6):
        if t < len(prefix) - 1:
            output.append(char_to_idx[prefix[t + 1]])
        else:
            out = nd.array(output[-5:],ctx = ctx)
            features = nd.zeros((1,27), ctx = ctx)

            for i in range(len(out)):
                features+= nd.one_hot(out[i], vocab_size)
                #print(features)
            Y = net(features).as_in_context(ctx)
            output.append(int(Y[0].argmax(axis=0).asscalar()))

    return ''.join([idx_to_char[i] for i in output])
```

In [25]:
```
prefix = 'but brutus is an honorable man'
predict_ngram_sum(prefix, 200, net_2, vocab_size, ctx)
```

Out[25]:
```
'but brutus is an honorable man e der e three  thrat th ere the re the re the re the re the re the re the re the re the re
the re the re the re the re the re the re the re the re the re the re the re the re the re the re the r'
```

How accurate is the model?

The model is not so accurate because they are predicting based on previous predictions after few sequences. I think this is why they are prodducing the same words after few characters.

How does the number of operations and weights compare to an RNN, a GRU and an LSTM discussed above?

The number of operations are much smaller with regard to the computation of neural netwokrs since this model doesn't have 'states' variable which feed it into networks. However, when it comes to preprocess, this model needs more operations to a constant number of degree.

In [ ]:
```

```