Text generation using a Char-RNN model

We're going to train a Recurrent Neural Network (RNN) to understand and generate text character by character. To do this, we'll provide the RNN with a large piece of text and ask it to learn the likelihood of the next character based on the sequence of previous characters.

Let's break it down with a simple example: Imagine our vocabulary consists of just four letters, "helo," and our training sequence is "hello." In this case, we have four separate training examples:

- The RNN should learn that when it sees "h", the next character "e" is likely.
- When it encounters "he", it should expect "l" to come next.
- Similarly, when it has "hel" as input, it should predict "l".
- Finally, after "hell", it should anticipate "o".

To make this happen, we'll represent each character as a vector using a technique called 1-of-k encoding, where each character is uniquely identified by a specific position in the vector. We'll then feed these character vectors into the RNN one at a time using a step function. The RNN will produce a sequence of output vectors, each with four dimensions, corresponding to the likelihood of the next character in the sequence.

In essence, we're training the RNN to understand and generate text character by character, and it will predict the next character based on the context of the preceding characters.

In [1]:
```
import torch
import torch.nn as nn
import torch.optim as optim
import string
import random
import numpy as np
```

Some pre-processing

We will train our model using a text file of Shakespeare's plays.

The first step is create a mapping from characters to integers, so as to represent each string as a list of integers. This is essential since we can only pass in numbers to our model, not strings or characters. Using this mapping, we now have our corpus of text mapped into a list of numbers.

In [2]:
```
# Create a character-to-index and index-to-character mapping
chars = np.load('chars.npy')
# np.save('chars.npy', chars)
char_to_index = {char: i for i, char in enumerate(chars)}
index_to_char = {i: char for i, char in enumerate(chars)}
```

Let's examine the mapping between integers and characters

(a) By looking at the dictionary `char_to_index` , answer the following questions:

- How many characters are we considering?
- What is the code for A?

Now let's read in Shakespeare's plays and convert the text to integers.

In [3]:
```
text = open('shakespeare_plays.txt', 'r').read()

# Convert the text to a numerical sequence
text_as_int = [char_to_index[char] for char in text]

data = list(text)
for i, ch in enumerate(data):
    data[i] = char_to_index[ch]

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

# data tensor on device
data = torch.tensor(data).to(device)
data = torch.unsqueeze(data, dim=1)
```

In [8]:
```
print(len(char_to_index))
```

```
66
```

In [7]:
```
print(char_to_index['A'])
```

```
12
```

(b) What is the length of the corpus in characters?

Defining our model

Initialization:

The `__init__` method initializes the RNN model with the following parameters:

- input_size: The size of the character vocabulary. This indicates the number of unique characters that the model can work with.
- output_size: The size of the output vocabulary. It's typically set to the same value as input_size for character generation tasks.
- hidden_size: The number of hidden units in the LSTM (Long Short-Term Memory) layer.
- num_layers: The number of LSTM layers stacked on top of each other.

Embedding Layer:

Inside the `__init__` method, an `nn.Embedding` layer is created. This layer is used to convert character indices (input) into dense vectors of fixed size.

LSTM Layer:

The `nn.LSTM layer` is defined with the specified `input_size`, `hidden_size`, and `num_layers`. This LSTM layer will process the embedded character sequence to capture dependencies and patterns within the sequence.

Decoder Layer:

After the LSTM layer, there is a linear (fully connected) layer defined as `nn.Linear`, which takes the output from the LSTM layer and maps it to the desired output size.

Forward Pass:

The forward method is where the actual computation occurs. It takes an input sequence (`input_seq`) and a hidden state (`hidden_state`) as input arguments.

First, the input sequence is passed through the embedding layer to convert the character indices into dense embeddings.

Then, these embeddings are fed into the LSTM layer, which processes the sequence. The LSTM layer produces an output sequence (output) and an updated hidden state.

Finally, the output from the LSTM is passed through the linear decoder layer to generate the predictions for the next characters in the sequence.

The forward method returns the output sequence and the updated hidden state.

Note that the `self.rnn` is actually an LSTM. This is used since LSTM's are known to outperform RNNs in most language tasks. We can very well replace this with an RNN, but would expect the model not to perform that well.

In [9]:
```
# Define the Char-RNN Model
class CharRNN(nn.Module):
    def __init__(self, input_size, output_size, hidden_size, num_layer
        super(CharRNN, self).__init__()
        self.embedding = nn.Embedding(input_size, input_size)
        self.rnn = nn.LSTM(input_size=input_size, hidden_size=hidden_s
        self.decoder = nn.Linear(hidden_size, output_size)

    def forward(self, input_seq, hidden_state):
        embedding = self.embedding(input_seq)
        output, hidden_state = self.rnn(embedding, hidden_state)
        output = self.decoder(output)
        return output, (hidden_state[0].detach(), hidden_state[1].deta
```

Defining a dataset class

In this part of the tutorial, we'll create a custom PyTorch dataset called `TextDataset`. This dataset is designed for training character-level text generation models like CharRNN. The dataset allows you to prepare your text data for training by converting characters to integer indices and creating input-target pairs for the model.

Initialization:

Accepts three parameters: `text`, `seq_length`, and `char_to_index`.

- `text`: The input text data you want to train the model on.
- `seq_length`: The length of sequences to be used during training (e.g., 50 characters per sequence).
- `char_to_index`: A dictionary mapping characters to integer indices.

Conversion of Text to Integers:

Inside the constructor, the input text is converted into an integer representation by mapping characters to their corresponding integer indices using the `char_to_index` dictionary.

`__len__`:

Defines the length of the dataset. You can specify a fixed length (e.g., 10,000) for your dataset, but this can be adjusted based on your dataset size. What you can also do is simply set length as `len(text) - self.seq_length`. This would result in a much larger set of samples and you wouldn't need to randomly sample an index (as described next).

`__getitem__`:

Retrieves individual training examples from the dataset.

- Randomly selects a starting index within the range `[0, len(text) - seq_length)` for each training example.
- Creates an input sequence (`input_seq`) containing characters from the selected `index` to `index + seq_length`.
- Creates a target sequence (`target_seq`) containing characters from `index + 1` to `index + seq_length + 1`.
- Returns a tuple with `input_seq` and `target_seq`.

In [10]:
```
from torch.utils.data import Dataset, DataLoader

class TextDataset(Dataset):
    def __init__(self, text, seq_length, char_to_index):
        self.seq_length = seq_length
        self.char_to_index = char_to_index
        self.text_as_int = [char_to_index[char] for char in text]

    def __len__(self):
        return 10000

    def __getitem__(self, idx):
        idx = random.randint(0, len(self.text_as_int) - self.seq_lengt
        input_seq = torch.tensor(self.text_as_int[idx:idx + self.seq_l
        target_seq = torch.tensor(self.text_as_int[idx + 1:idx + self.
        return input_seq, target_seq

# Create the dataset
seq_length = 100
text_dataset = TextDataset(text, seq_length, char_to_index)

# Create a data loader
batch_size = 2048
data_loader = DataLoader(text_dataset, batch_size=batch_size, shuffle=
```

In [11]:
```
# Define the training loop

input_size = len(chars)
output_size = len(chars)
hidden_size = 512
num_layers = 3

model = CharRNN(input_size, output_size, hidden_size, num_layers)

# Define the loss function and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)

# Training parameters
num_epochs = 15

# Training device
model = model.to(device)
```

We can now train our model using the following code. For ease of use, a pre-trained model has been provided since training the model can be a long process especially if you don't have GPUs set up on your local machine.

In [12]:
```
## NO NEED TO RUN THIS CELL

# for i_epoch in range(1, num_epochs+1):

#     n = 0
#     running_loss = 0
#     hidden_state = None

#     for i_data,(input_seq, target_seq) in enumerate(data_loader):
#         print(i_data)
#         # forward pass
#         input_seq = input_seq.to(device)
#         target_seq = target_seq.to(device)
#         output, hidden_state = model(input_seq, hidden_state)
#         print(output.shape,target_seq.shape)
#         # compute loss
#         loss = criterion(output.view(-1,output_size), target_seq.vie
#         running_loss += loss.item()

#         # compute gradients and take optimizer step
#         optimizer.zero_grad()
#         loss.backward()
#         optimizer.step()

#         n +=1


#     # print loss and save weights after every epoch
#     print("Epoch: {0} \t Loss: {1:.8f}".format(i_epoch, running_loss
#     torch.save(model.state_dict(), './model_{}.pth'.format(i_epoch))
```

Let's load the pretrained weights

In [13]:
```
model.load_state_dict(torch.load('./CharRNN_shakespeare.pth',map_locat
model = model.cpu()
model.eval()
```

Out[13]:
```
CharRNN(
  (embedding): Embedding(66, 66)
  (rnn): LSTM(66, 512, num_layers=3)
  (decoder): Linear(in_features=512, out_features=66, bias=True)
)
```

In [15]:
```
len(text_dataset)
```

Out[15]:
```
10000
```

(c) When you ran the cell above, you should have gotten an input and output size of 66. Where is this number coming from?

Time to generate some Shakespeare!

In [17]:
```
input_seq = data[25:26].cpu()
hidden_state = None
o_len = 0
output_len = 2000
while o_len < output_len:
    # forward pass
    output, hidden_state = model(input_seq, hidden_state)
    # construct categorical distribution and sample a character
    output = torch.nn.functional.softmax(torch.squeeze(output), dim=0)
    dist = torch.distributions.Categorical(output)
    index = dist.sample()
    # index = torch.argmax(output)
    # print the sampled character
    print(index_to_char[index.item()], end='')

    # next input is current output
    input_seq[0][0] = index.item()
    o_len += 1
```

```
 on your
majesty. Yet to it in the ill horse.

Boy:
Your good time Gloucester, adieu?

KING HENRY V:
Give it this, my liege, a fellow and discipline
Dwelling in monstick'd stone constables;
```

And in all nature's obedience, not parhable.
There is film'd septious of all grace. Knave
a man for each poorer as
heur no tackle and pritapy, though lips gave you a
ring, know this tripman, God to be to
side yourself, take a night. You have
'em—nailies! why shites it again?

QUELINLUS:
I'll can come for his honour: how art thou so?

KING HENRY V:
Nay, and, prance, sweet French miscrimps! your weak devorys save the
conseque and us, on thy blood and might in
Exeminature forsooths, and her fellows, feeble is
dishenitors and entorman: did this broke better,
By our offer'd French for Herculess. What
follog nymph's friend, for the tail of lessenour
mercy, yeared ton of person aps discourse?

KING HENRY V:
It is good as this a little fleet as those our monds,
And fistre into good as if God, an hour of it,
In morning that to pay porperon;
Pintens behollest insead rivaked, luxure, for other creature
I writ in honour upon the bell: you know he honours,
Lend Scotchman.

REY:
An objects! I dare not.

KING HENRY V:
Dear instanced Monmouth, les than they speak it couse
on them of linen cap of your daughters reason.
Postience have an impenient word.

GOWER:
My lord, take your abouts, to greet my vous word!

MACKIN:
I will not, my young illurning: this I be all
Welcome with appointed bull of graces, and
like the que is your glove to thine offer all,
With his pleasure; close till I give and ten,
Give persuade up my elication, his hinds,
Therefore come hither, thou hast good Master George
Pediance of, good morrow, I will be justs any.

FLUELLEN:
Go to. Now, Trospur name!

```
KING OF C Constable:
France, Thare; I say jest holy to live
i' the rivable regar and heur.

Hostes:
I had nothing sounds.

ORLEANS:
'In the bull of few,' and so art thou a
fellow?

FLUELLEN:
I'll take those was a qore; his mothe
```

It is 66 since we have 66 characters (found this on part a) , and the output should also be 66 characters. The RNN hidden layers can be longer, but it should predict between the 66 characters.

In [ ]: