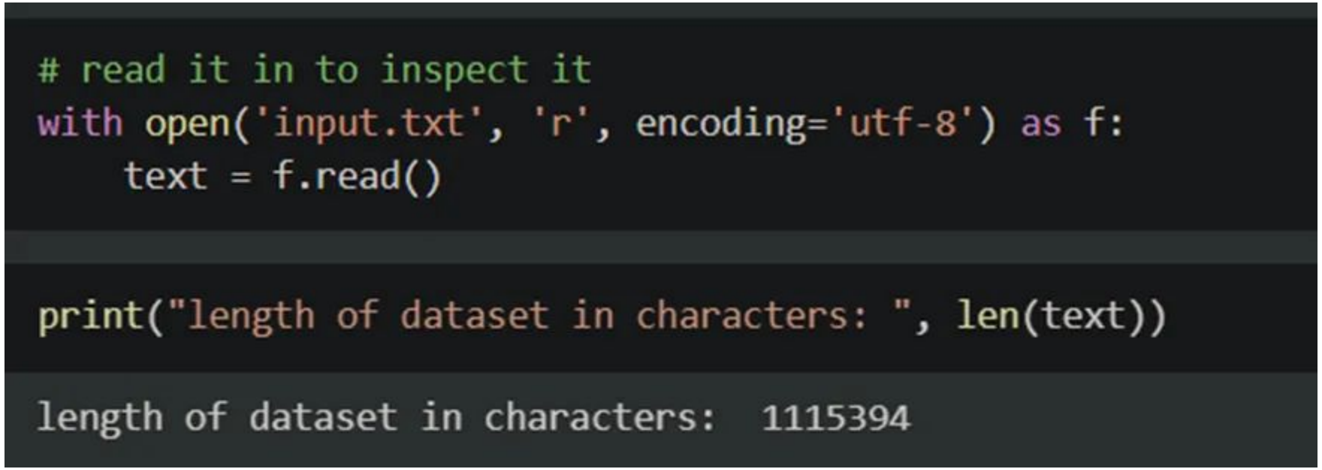
GPT from scratch

- Data preprocessing

Reading and Exploring the Data

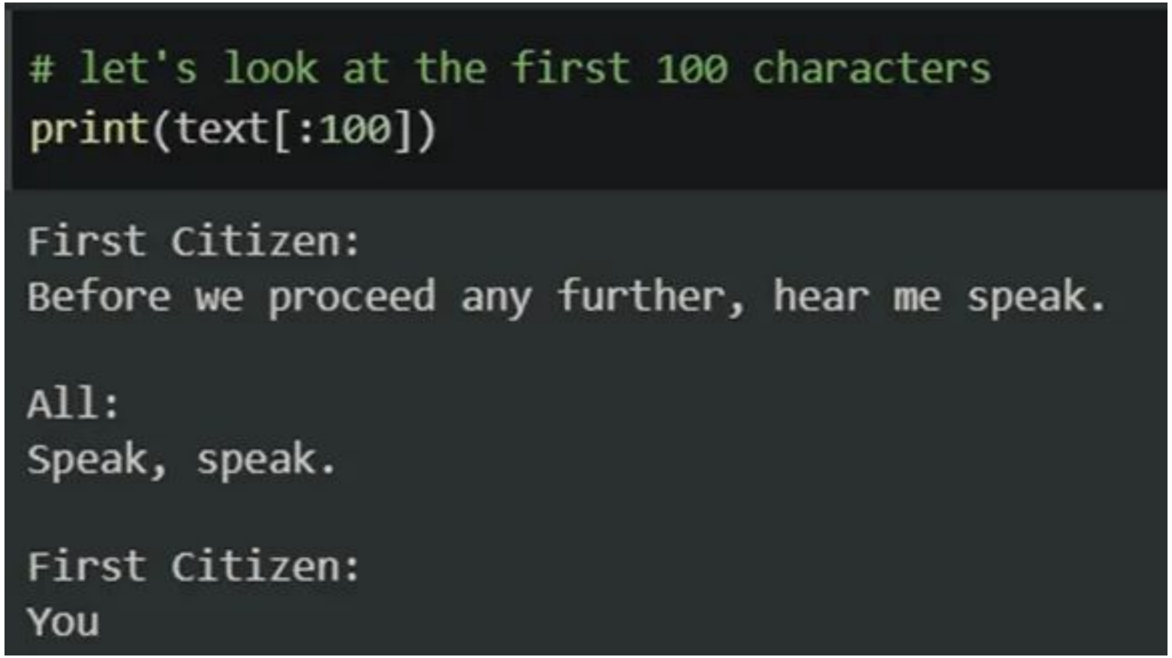


```
# read it in to inspect it
with open('input.txt', 'r', encoding='utf-8') as f:
    text = f.read()
```

```
print("length of dataset in characters: ", len(text))
```

```
length of dataset in characters:  1115394
```

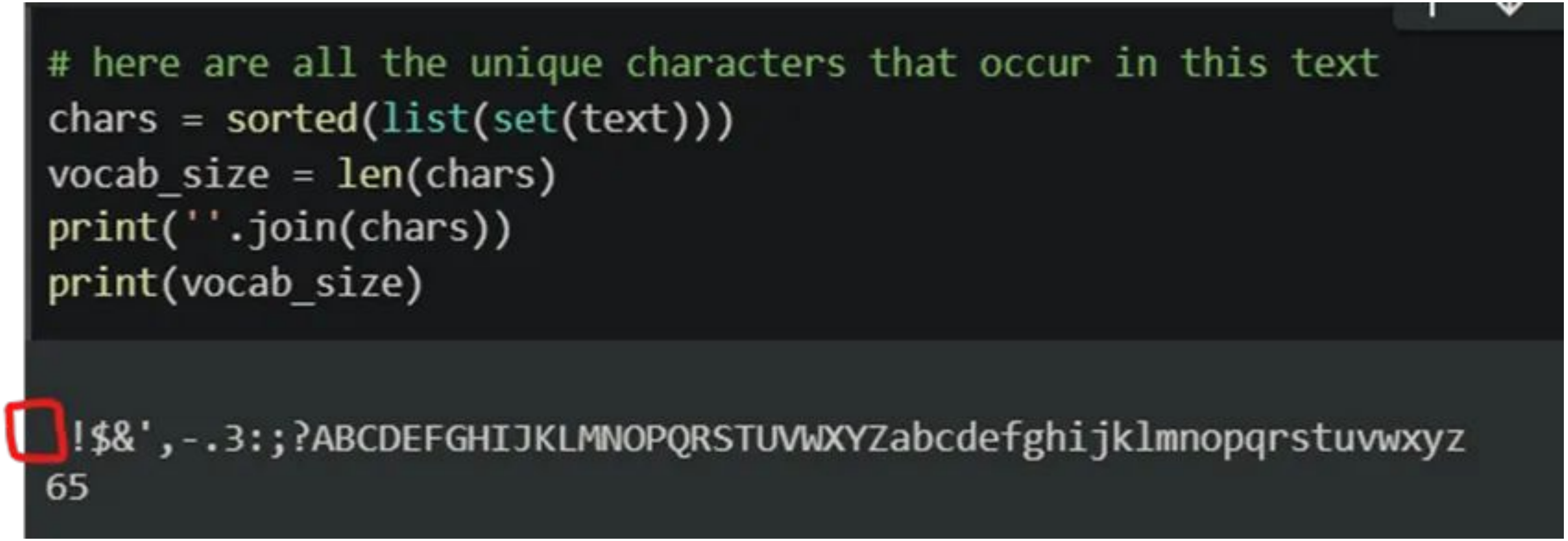
Reading and Exploring the Data



```python
# let's look at the first 100 characters
print(text[:100])
```

```
First Citizen:
Before we proceed any further, hear me speak.

All:
Speak, speak.

First Citizen:
You
```

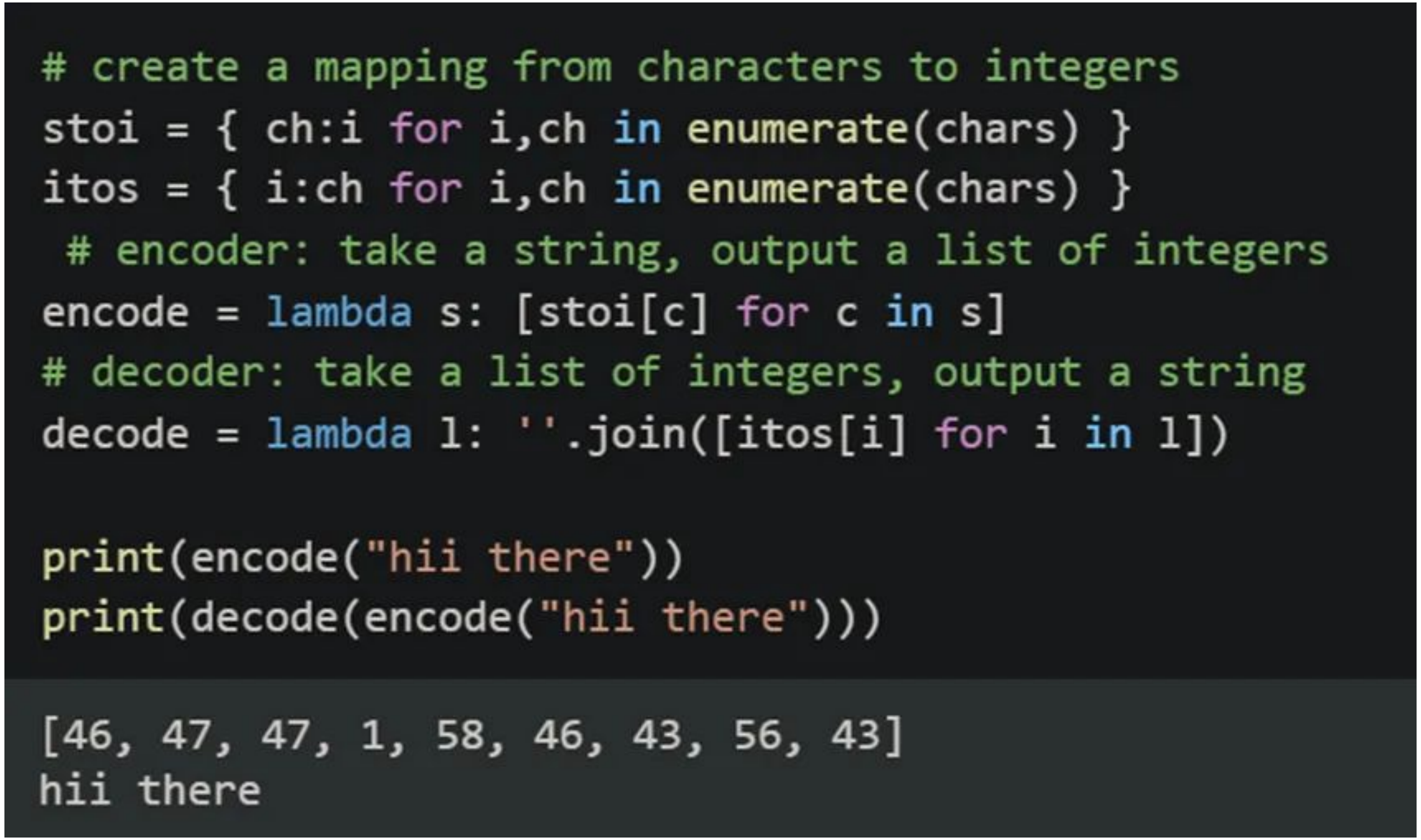
Reading and Exploring the Data



```
# here are all the unique characters that occur in this text
chars = sorted(list(set(text)))
vocab_size = len(chars)
print(''.join(chars))
print(vocab_size)
```

```
 !$&',-.3:;?ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz
65
```

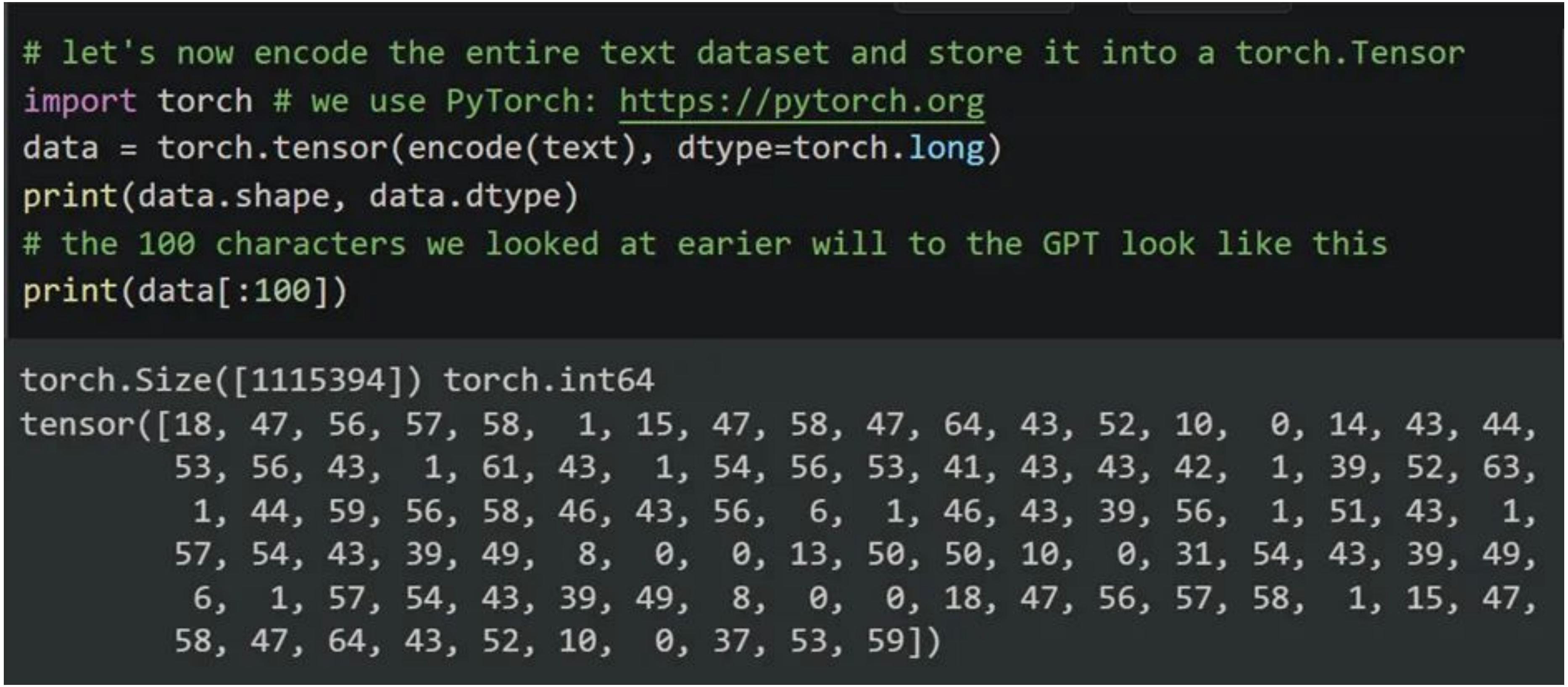
Tokenization and train/validation split



```python
# create a mapping from characters to integers
stoi = { ch:i for i,ch in enumerate(chars) }
itos = { i:ch for i,ch in enumerate(chars) }
 # encoder: take a string, output a list of integers
encode = lambda s: [stoi[c] for c in s]
# decoder: take a list of integers, output a string
decode = lambda l: ''.join([itos[i] for i in l])

print(encode("hii there"))
print(decode(encode("hii there")))
```

```
[46, 47, 47, 1, 58, 46, 43, 56, 43]
hii there
```

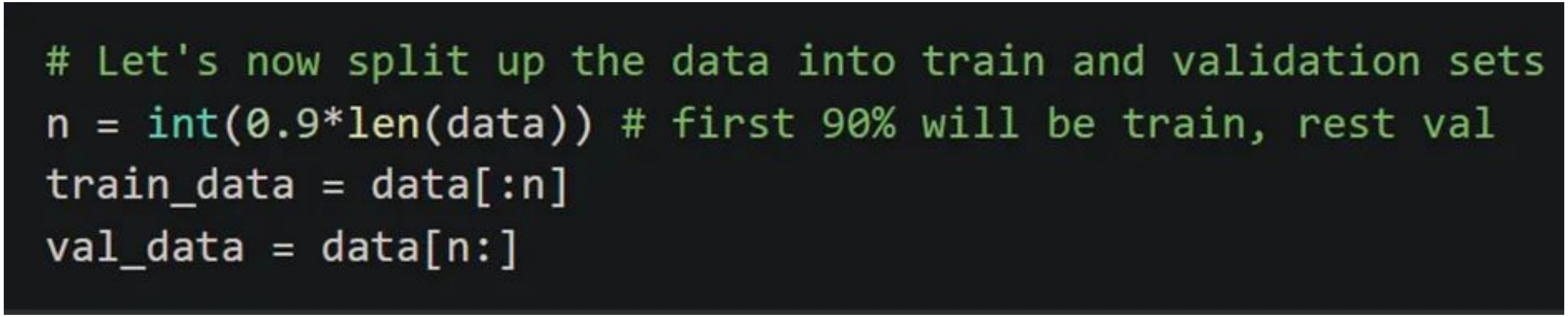
Tokenization and train/validation split



```python
# let's now encode the entire text dataset and store it into a torch.Tensor
import torch # we use PyTorch: https://pytorch.org
data = torch.tensor(encode(text), dtype=torch.long)
print(data.shape, data.dtype)
# the 100 characters we looked at earier will to the GPT look like this
print(data[:100])
```

```
torch.Size([1115394]) torch.int64
tensor([18, 47, 56, 57, 58,  1, 15, 47, 58, 47, 64, 43, 52, 10,  0, 14, 43, 44,
        53, 56, 43,  1, 61, 43,  1, 54, 56, 53, 41, 43, 43, 42,  1, 39, 52, 63,
         1, 44, 59, 56, 58, 46, 43, 56,  6,  1, 46, 43, 39, 56,  1, 51, 43,  1,
        57, 54, 43, 39, 49,  8,  0,  0, 13, 50, 50, 10,  0, 31, 54, 43, 39, 49,
         6,  1, 57, 54, 43, 39, 49,  8,  0,  0, 18, 47, 56, 57, 58,  1, 15, 47,
        58, 47, 64, 43, 52, 10,  0, 37, 53, 59])
```

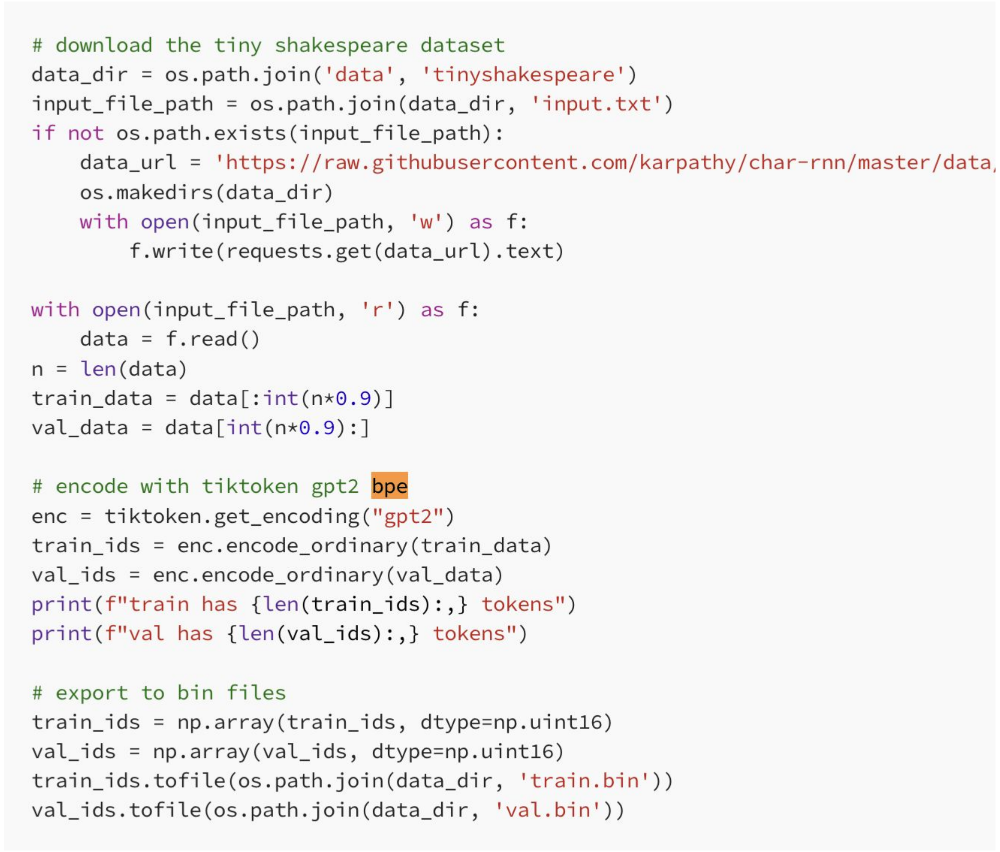
Tokenization and train/validation split



```
# Let's now split up the data into train and validation sets
n = int(0.9*len(data)) # first 90% will be train, rest val
train_data = data[:n]
val_data = data[n:]
```

What are we missing so far?

Bype Pair Tokenization!



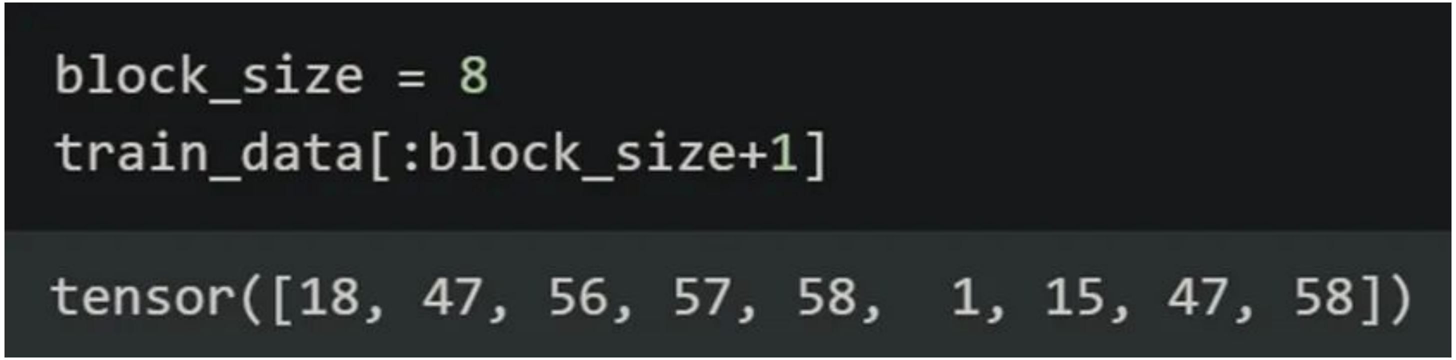
```
# download the tiny shakespeare dataset
data_dir = os.path.join('data', 'tinyshakespeare')
input_file_path = os.path.join(data_dir, 'input.txt')
if not os.path.exists(input_file_path):
    data_url = 'https://raw.githubusercontent.com/karpathy/char-rnn/master/data
    os.makedirs(data_dir)
    with open(input_file_path, 'w') as f:
        f.write(requests.get(data_url).text)

with open(input_file_path, 'r') as f:
    data = f.read()
n = len(data)
train_data = data[:int(n*0.9)]
val_data = data[int(n*0.9):]

# encode with tiktoken gpt2 bpe
enc = tiktoken.get_encoding("gpt2")
train_ids = enc.encode_ordinary(train_data)
val_ids = enc.encode_ordinary(val_data)
print(f"train has {len(train_ids):,} tokens")
print(f"val has {len(val_ids):,} tokens")

# export to bin files
train_ids = np.array(train_ids, dtype=np.uint16)
val_ids = np.array(val_ids, dtype=np.uint16)
train_ids.tofile(os.path.join(data_dir, 'train.bin'))
val_ids.tofile(os.path.join(data_dir, 'val.bin'))
```

Data Loader: Batches of Chunks of Data

Right now we are working with a very small dataset, but practical approaches or commercial projects requires large amount of data to work with.

In these cases, it does matter that data must be passed in chunks (i.e., small sizes of train data) into transformers to avoid memory error. Chunks are also called **block_size** in terms of coding.

Data Loader: Batches of Chunks of Data

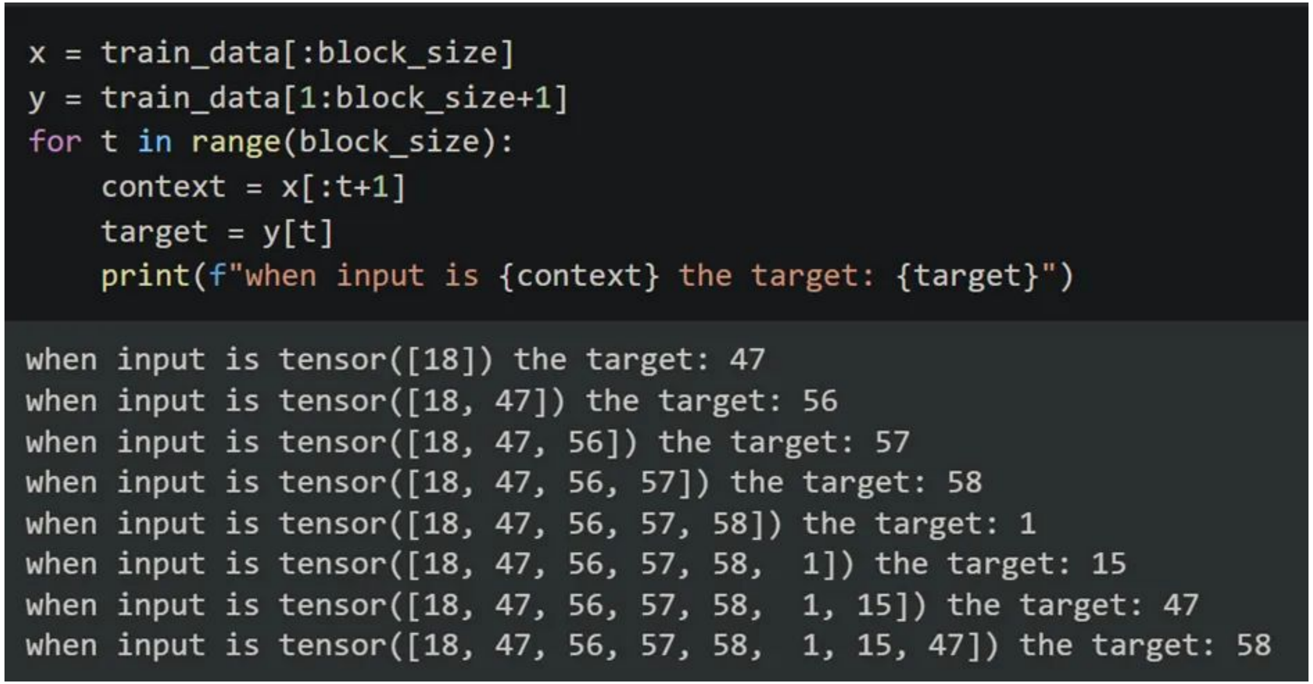


```
block_size = 8
train_data[:block_size+1]
```

```
tensor([18, 47, 56, 57, 58,  1, 15, 47, 58])
```

If we set our chunk size or block size value to 8, it will give us a small portion of our training data that is 9 characters long. We add 1 to the block size parameter because it will only target 8 characters, not including the first character (which makes it a total of 9)

Data Loader: Batches of Chunks of Data

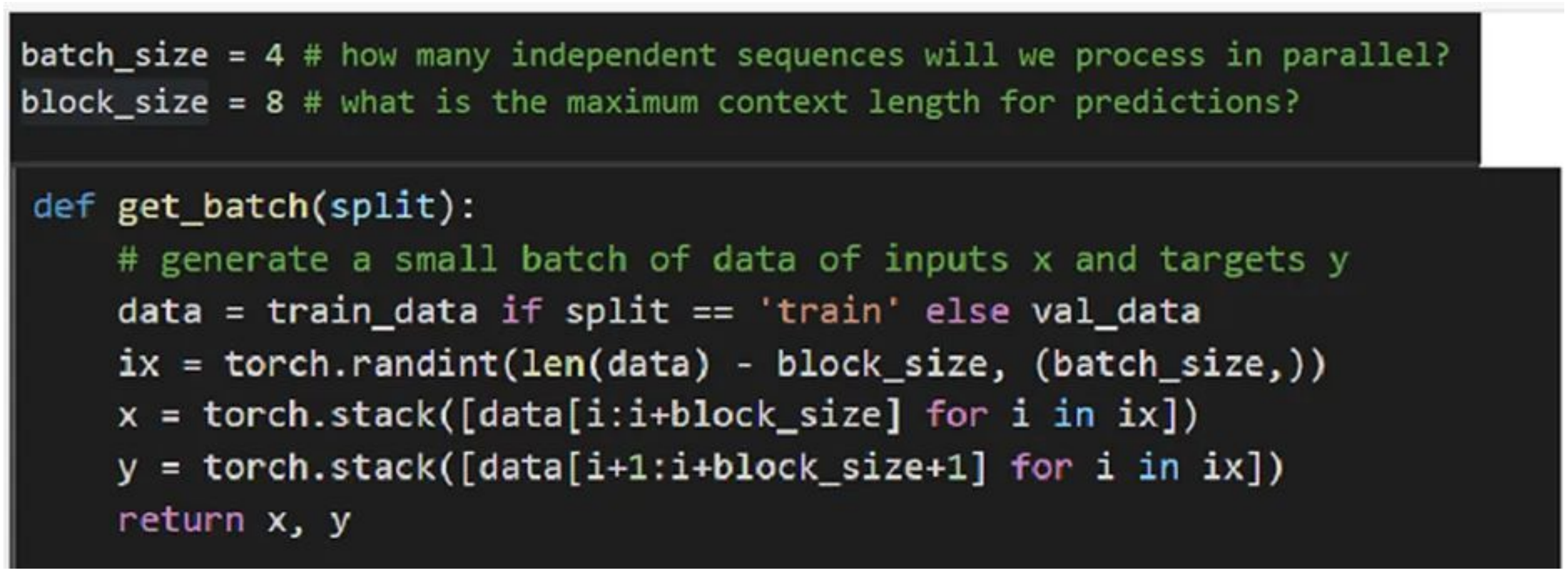


```python
x = train_data[:block_size]
y = train_data[1:block_size+1]
for t in range(block_size):
    context = x[:t+1]
    target = y[t]
    print(f"when input is {context} the target: {target}")
```

```
when input is tensor([18]) the target: 47
when input is tensor([18, 47]) the target: 56
when input is tensor([18, 47, 56]) the target: 57
when input is tensor([18, 47, 56, 57]) the target: 58
when input is tensor([18, 47, 56, 57, 58]) the target: 1
when input is tensor([18, 47, 56, 57, 58,  1]) the target: 15
when input is tensor([18, 47, 56, 57, 58,  1, 15]) the target: 47
when input is tensor([18, 47, 56, 57, 58,  1, 15, 47]) the target: 58
```

starting with the character 18, the model predicts the next character 47. Then, using the characters 18 and 47 together, it predicts the next character 56.

Data Loader: Batches Size

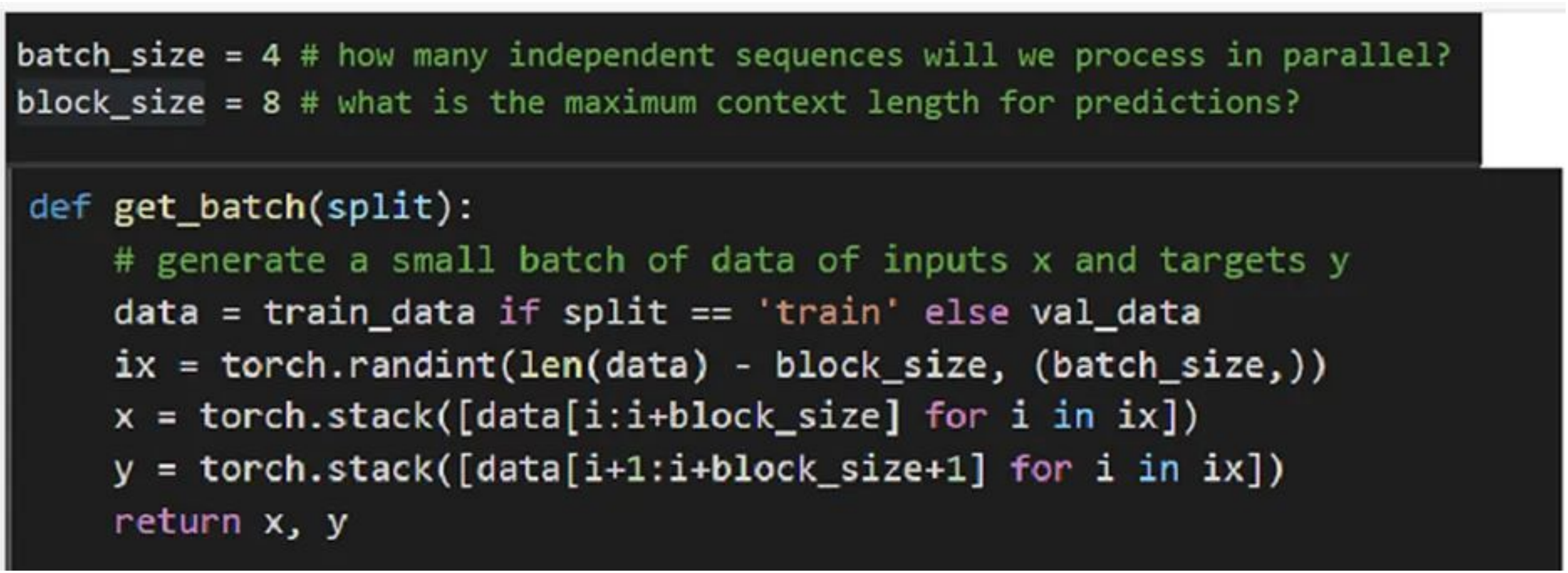


```
batch_size = 4 # how many independent sequences will we process in parallel?
block_size = 8 # what is the maximum context length for predictions?
```

```
def get_batch(split):
    # generate a small batch of data of inputs x and targets y
    data = train_data if split == 'train' else val_data
    ix = torch.randint(len(data) - block_size, (batch_size,))
    x = torch.stack([data[i:i+block_size] for i in ix])
    y = torch.stack([data[i+1:i+block_size+1] for i in ix])
    return x, y
```

A batch refers to a collection of multiple chunks of data. We’ll be feeding multiple batches (each containing multiple chunks) into our Transformer model

Data Loader: Batches Size

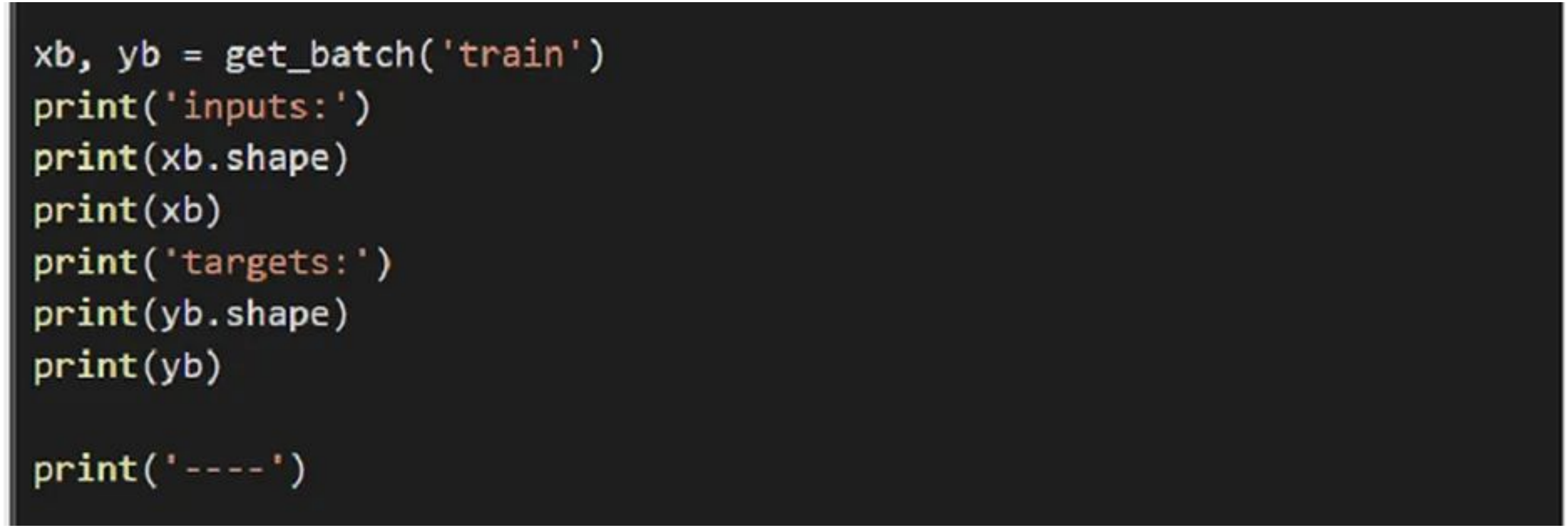


```
batch_size = 4 # how many independent sequences will we process in parallel?
block_size = 8 # what is the maximum context length for predictions?
```

```
def get_batch(split):
    # generate a small batch of data of inputs x and targets y
    data = train_data if split == 'train' else val_data
    ix = torch.randint(len(data) - block_size, (batch_size,))
    x = torch.stack([data[i:i+block_size] for i in ix])
    y = torch.stack([data[i+1:i+block_size+1] for i in ix])
    return x, y
```

The batch size is 4 which means each batch contains 4 examples of chunks (independent to each other) and block size is 8 which means each chunk size is 8 characters long, get_batch function is creating a 4x8 matrix based on split parameter (train or valid).

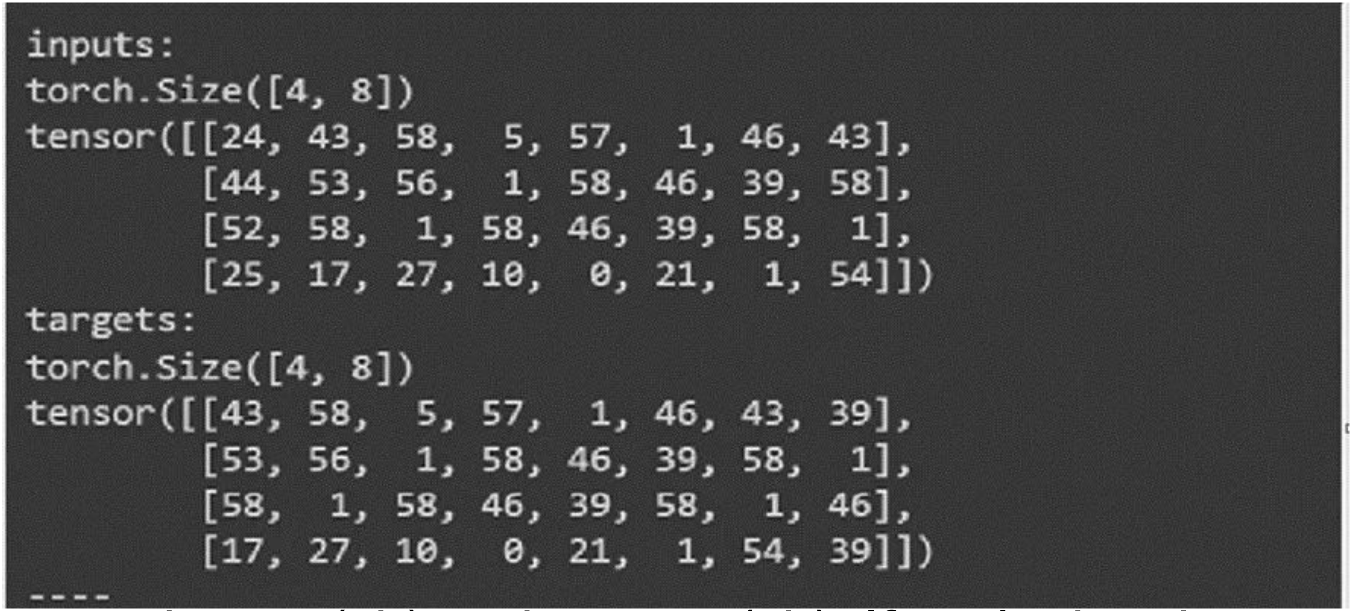
Data Loader: Batches Size



```
xb, yb = get_batch('train')
print('inputs:')
print(xb.shape)
print(xb)
print('targets:')
print(yb.shape)
print(yb)

print('----')
```

It will return two matrices: x are the inputs while y are the targets, both are 4x8 matrices.

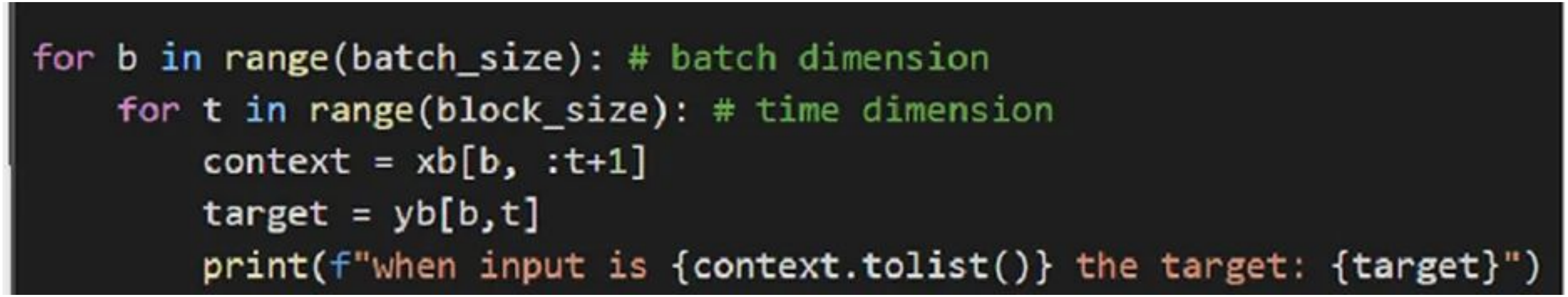
Data Loader: Batches Size



```
inputs:
torch.Size([4, 8])
tensor([[24, 43, 58,  5, 57,  1, 46, 43],
        [44, 53, 56,  1, 58, 46, 39, 58],
        [52, 58,  1, 58, 46, 39, 58,  1],
        [25, 17, 27, 10,  0, 21,  1, 54]])
targets:
torch.Size([4, 8])
tensor([[43, 58,  5, 57,  1, 46, 43, 39],
        [53, 56,  1, 58, 46, 39, 58,  1],
        [58,  1, 58, 46, 39, 58,  1, 46],
        [17, 27, 10,  0, 21,  1, 54, 39]])
----
```

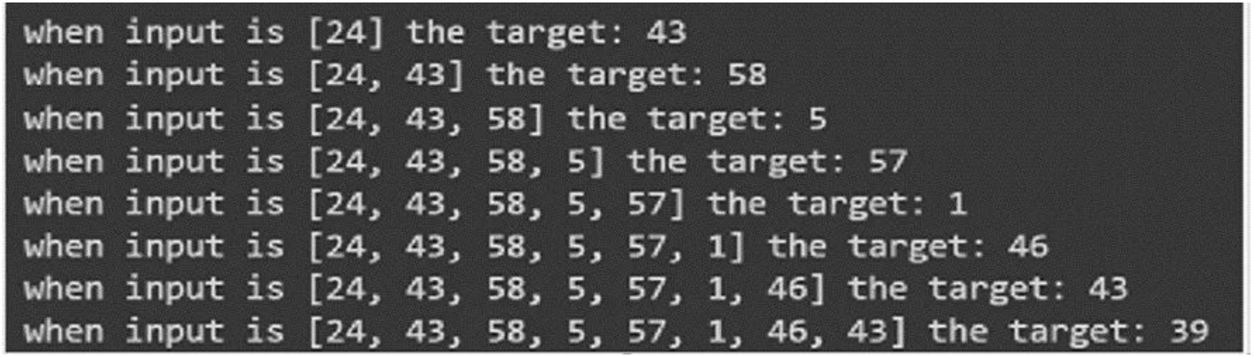
The outputs are inputs (xb) and targets (yb). If we look at input matrix first element 24, the target will be 43 from targets matrix. For 24 and 43 from inputs matrix the target is 58 from targets matrix, and for 52,58 and 1 as a combined input from the third row of input matrix the target is 58 …

Data Loader: Batches Size



```
for b in range(batch_size): # batch dimension
    for t in range(block_size): # time dimension
        context = xb[b, :t+1]
        target = yb[b,t]
        print(f"when input is {context.tolist()} the target: {target}")
```

This is what it looks like in terms of coding, And the output:



```
when input is [24] the target: 43
when input is [24, 43] the target: 58
when input is [24, 43, 58] the target: 5
when input is [24, 43, 58, 5] the target: 57
when input is [24, 43, 58, 5, 57] the target: 1
when input is [24, 43, 58, 5, 57, 1] the target: 46
when input is [24, 43, 58, 5, 57, 1, 46] the target: 43
when input is [24, 43, 58, 5, 57, 1, 46, 43] the target: 39
```

GPT from scratch

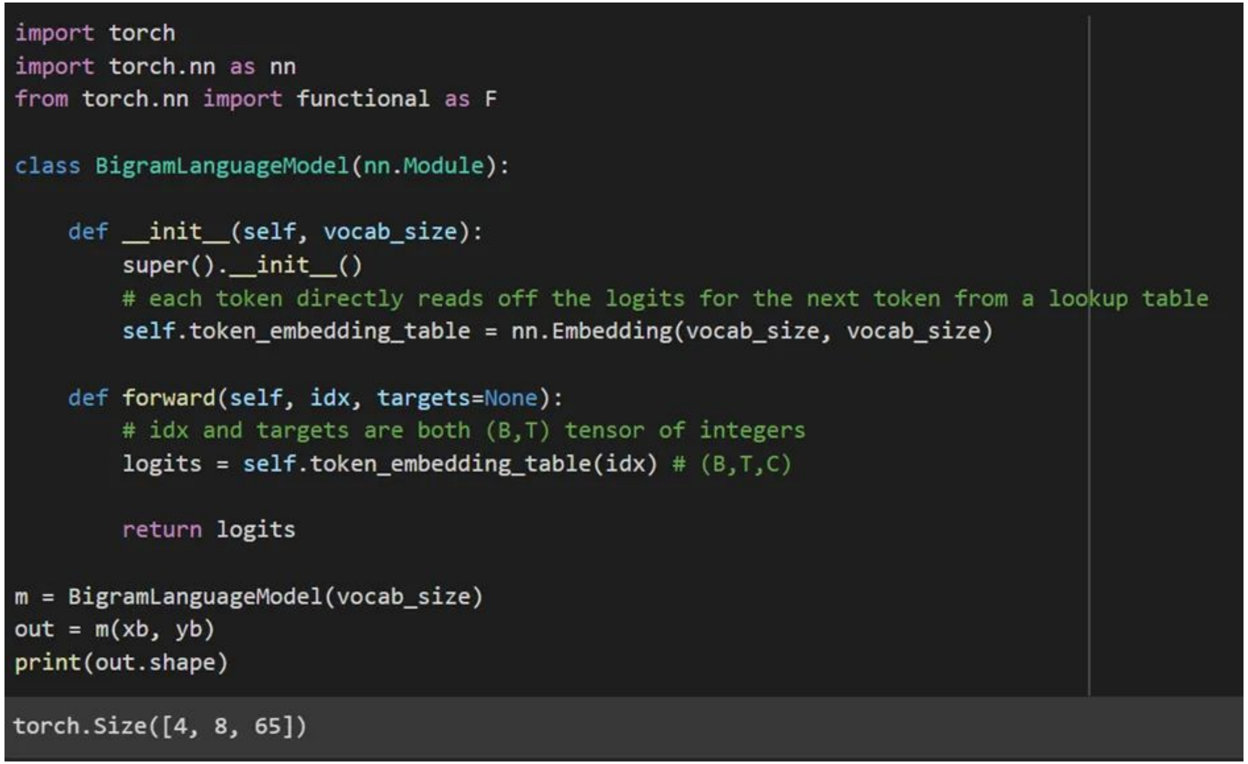
- Neural Bigram Baseline: Model, Loss, Generation

Neural Bigram Language Model, Loss, Generation

Up until now, we have encoded our text data, split it into train/Val and discussed the importance of chunks and batch size.

Now, let’s attempt to feed this data into neural networks. The most basic neural network for language modelling is the bigram language model.

Neural Bigram: Model

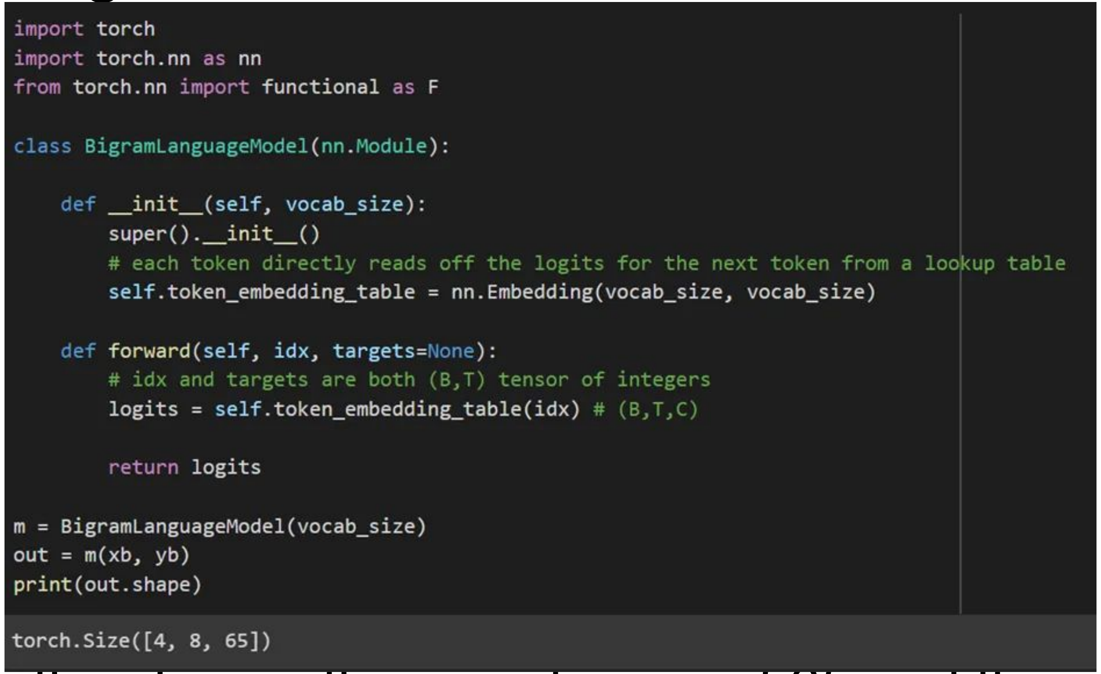


```python
import torch
import torch.nn as nn
from torch.nn import functional as F

class BigramLanguageModel(nn.Module):

    def __init__(self, vocab_size):
        super().__init__()
        # each token directly reads off the logits for the next token from a lookup table
        self.token_embedding_table = nn.Embedding(vocab_size, vocab_size)

    def forward(self, idx, targets=None):
        # idx and targets are both (B,T) tensor of integers
        logits = self.token_embedding_table(idx) # (B,T,C)

        return logits

m = BigramLanguageModel(vocab_size)
out = m(xb, yb)
print(out.shape)
```

```
torch.Size([4, 8, 65])
```

We will be using torch library to implement Bi-Gram language model

Neural Bigram: Model

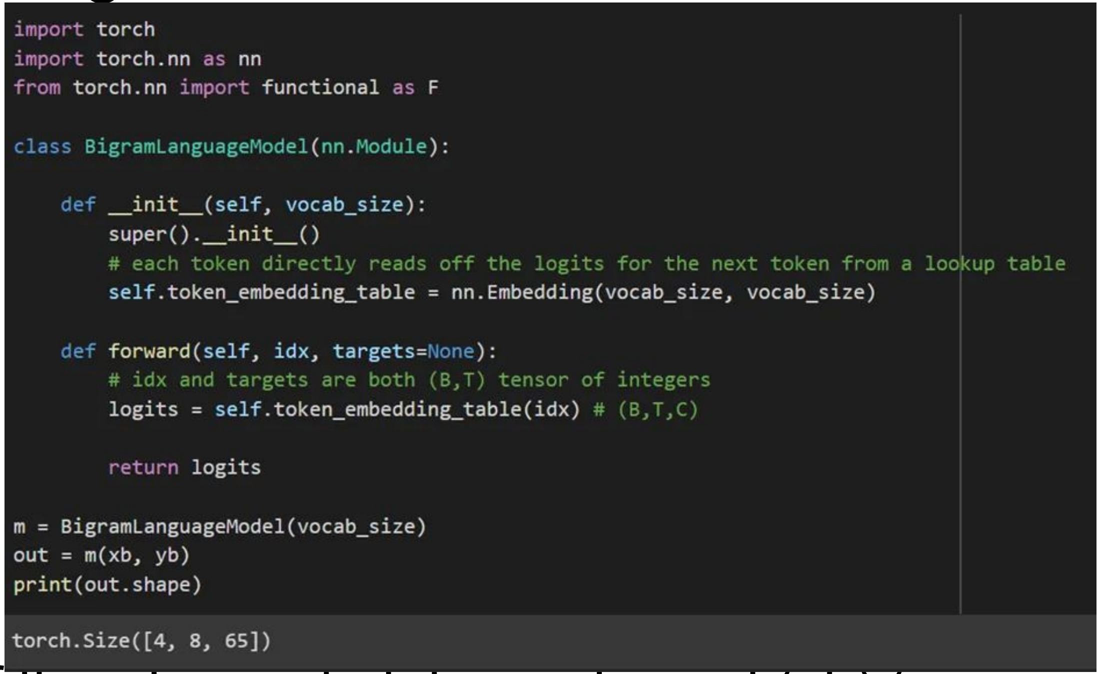


```
import torch
import torch.nn as nn
from torch.nn import functional as F

class BigramLanguageModel(nn.Module):

    def __init__(self, vocab_size):
        super().__init__()
        # each token directly reads off the logits for the next token from a lookup table
        self.token_embedding_table = nn.Embedding(vocab_size, vocab_size)

    def forward(self, idx, targets=None):
        # idx and targets are both (B,T) tensor of integers
        logits = self.token_embedding_table(idx) # (B,T,C)

        return logits

m = BigramLanguageModel(vocab_size)
out = m(xb, yb)
print(out.shape)
```

```
torch.Size([4, 8, 65])
```

we first initialize the class with a vocab_size of 65 and then pass input (xb) and targets (yb).

Neural Bigram: Model

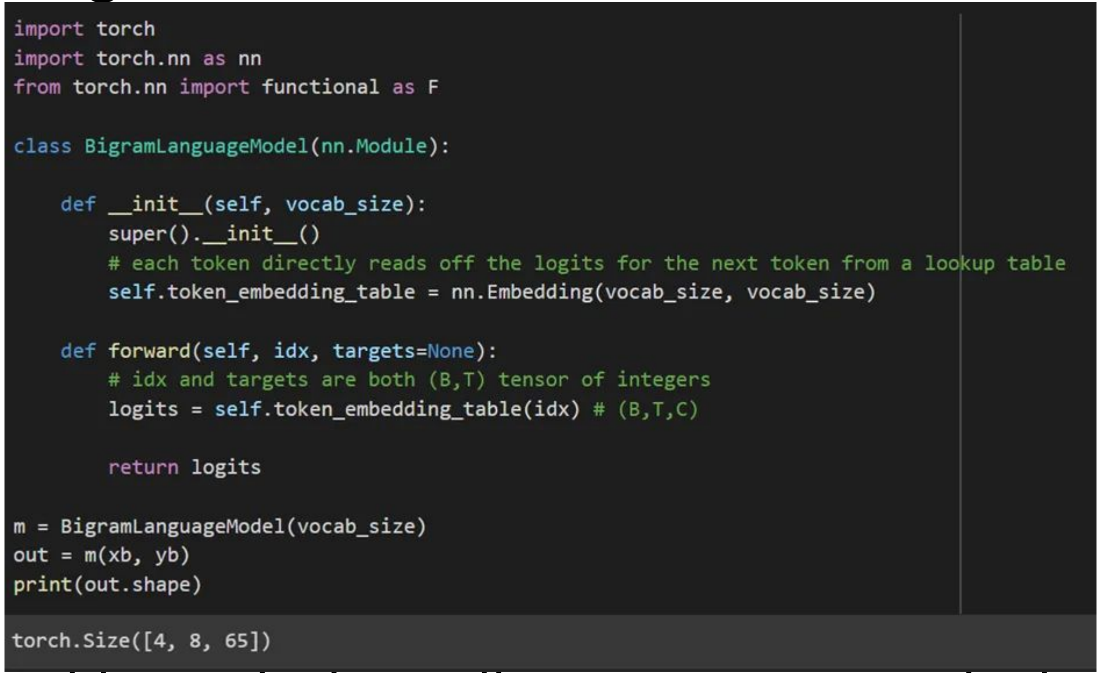


```python
import torch
import torch.nn as nn
from torch.nn import functional as F

class BigramLanguageModel(nn.Module):

    def __init__(self, vocab_size):
        super().__init__()
        # each token directly reads off the logits for the next token from a lookup table
        self.token_embedding_table = nn.Embedding(vocab_size, vocab_size)

    def forward(self, idx, targets=None):
        # idx and targets are both (B,T) tensor of integers
        logits = self.token_embedding_table(idx) # (B,T,C)

        return logits

m = BigramLanguageModel(vocab_size)
out = m(xb, yb)
print(out.shape)
```

```
torch.Size([4, 8, 65])
```

The purpose of this class is to take each input (xb) (renamed to idx) and pass them into a token embedding table, which is created in the Constructor and is of size vocab_size x vocab_size.

Neural Bigram: Model



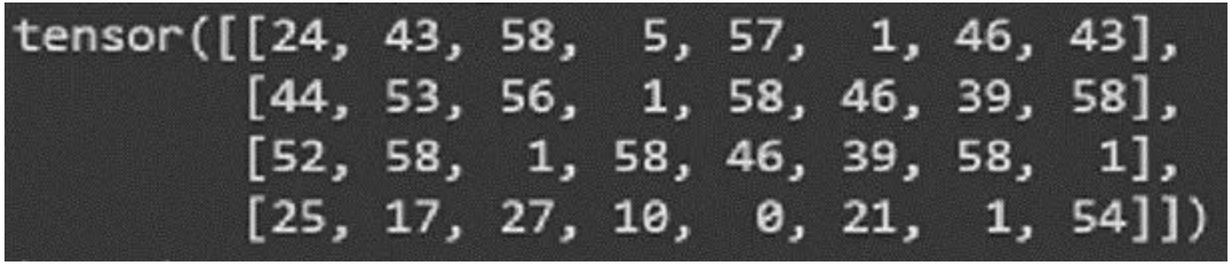
```
import torch
import torch.nn as nn
from torch.nn import functional as F

class BigramLanguageModel(nn.Module):

    def __init__(self, vocab_size):
        super().__init__()
        # each token directly reads off the logits for the next token from a lookup table
        self.token_embedding_table = nn.Embedding(vocab_size, vocab_size)

    def forward(self, idx, targets=None):
        # idx and targets are both (B,T) tensor of integers
        logits = self.token_embedding_table(idx) # (B,T,C)

        return logits

m = BigramLanguageModel(vocab_size)
out = m(xb, yb)
print(out.shape)
```

```
torch.Size([4, 8, 65])
```

We use nn.embedding, which is a thin wrapper around a tensor of shape vocab_size by vocab_size. When idx is passed in, each integer in the input refers to a row of the embedding table corresponding to its index.

Neural Bigram: Model

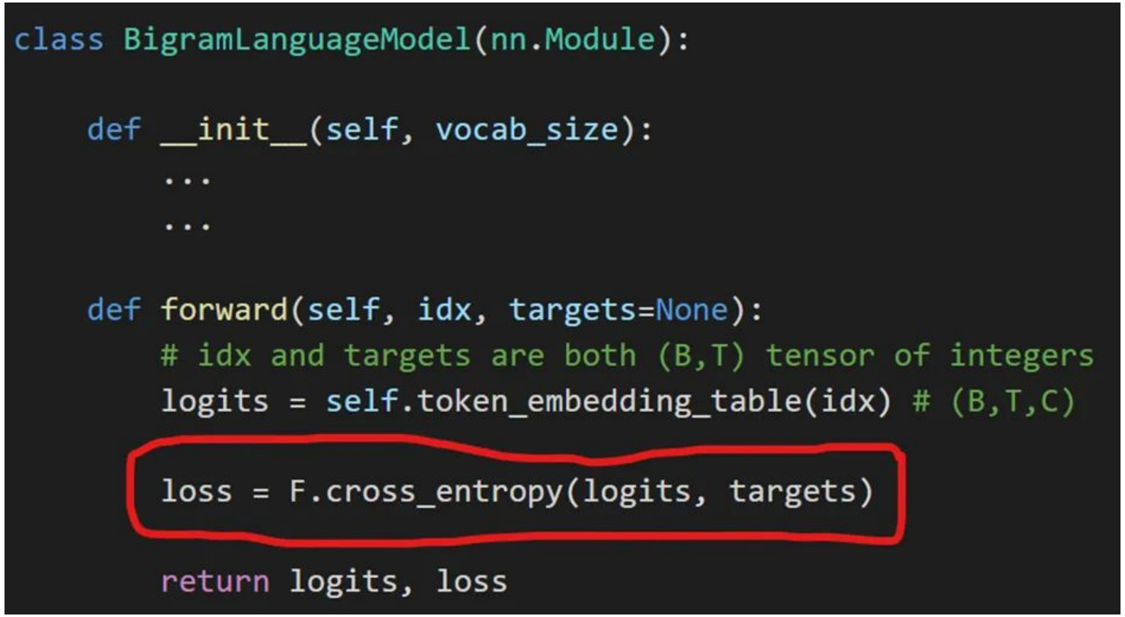
For example, remember our input matrix:



```
tensor([[24, 43, 58,  5, 57,  1, 46, 43],
        [44, 53, 56,  1, 58, 46, 39, 58],
        [52, 58,  1, 58, 46, 39, 58,  1],
        [25, 17, 27, 10,  0, 21,  1, 54]])
```

the integer 24 will refer to the 24th row of the embedding table, similarly 43 represent the 43th row and so on. The model will select and arrange all the rows into a (B, T, C) format, where B represents the batch size of 4, T represents the time of 8, and C represents the channels of the vocabulary size (i.e., 65). This format we called **logits** (the scores for the next character in the sequence). The output is [4,8,65] which means we get **scores** for each 4x8 matrix positions of our inputs (xb)

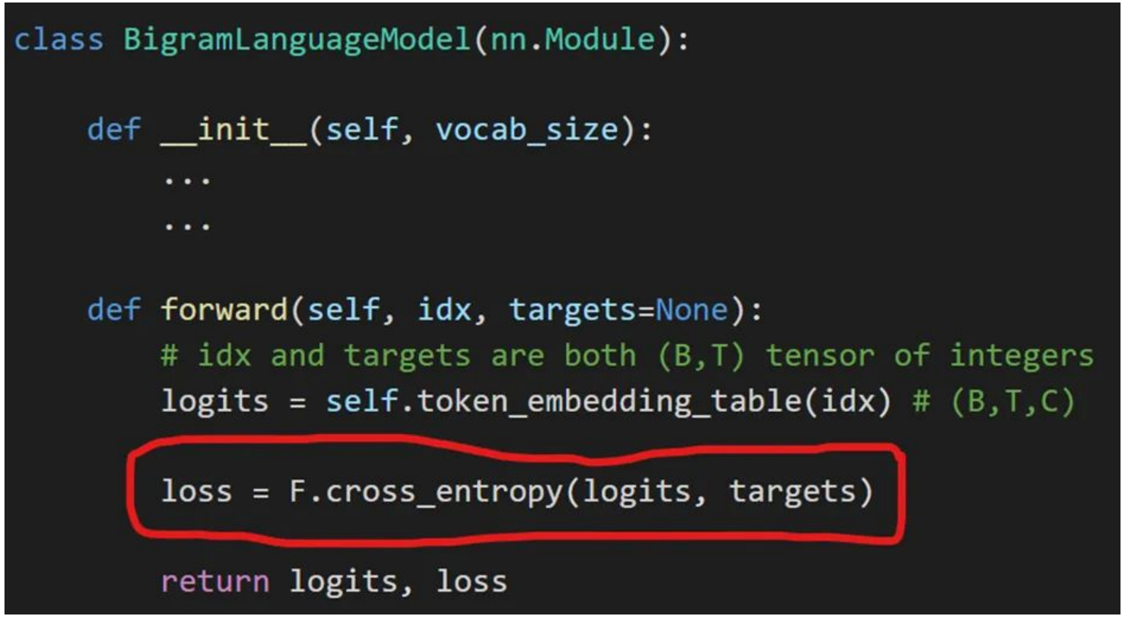
Neural Bigram: Loss



```
class BigramLanguageModel(nn.Module):

    def __init__(self, vocab_size):
        ...
        ...

    def forward(self, idx, targets=None):
        # idx and targets are both (B,T) tensor of integers
        logits = self.token_embedding_table(idx) # (B,T,C)

        loss = F.cross_entropy(logits, targets)

        return logits, loss
```

Now that we make predictions of what comes next, we must evaluate the loss function. It is used to measure the quality of the prediction. We will be using **negative log likelihood loss**, it is available in PyTorch under the name *cross_entropy*

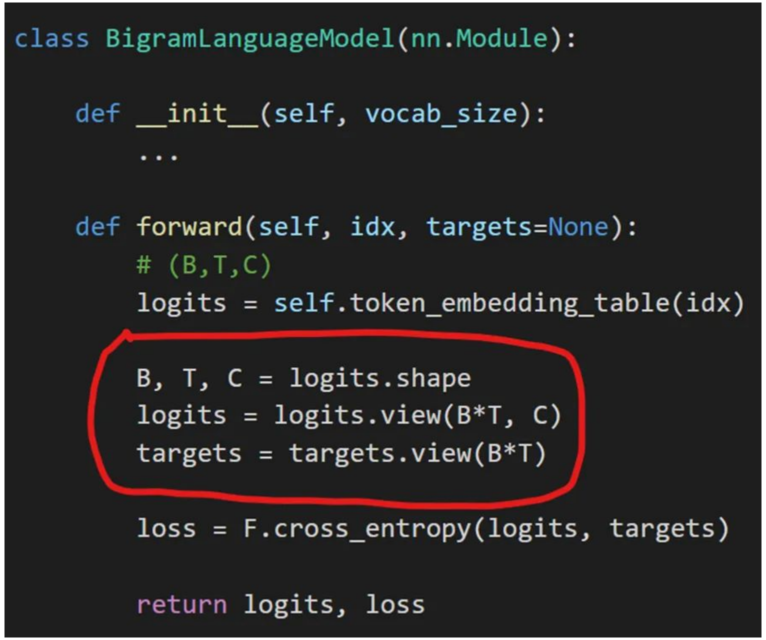
Neural Bigram: Loss



```
class BigramLanguageModel(nn.Module):

    def __init__(self, vocab_size):
        ...
        ...

    def forward(self, idx, targets=None):
        # idx and targets are both (B,T) tensor of integers
        logits = self.token_embedding_table(idx) # (B,T,C)

        loss = F.cross_entropy(logits, targets)

        return logits, loss
```

calling the Big-gram class results in an **error**. The reason for this is that the shape of our logits is multi-dimensional (B, T, C), while PyTorch requires that for multi-dimensional tensors, the channels (C) should be in the second position of the (B, T, C) format.

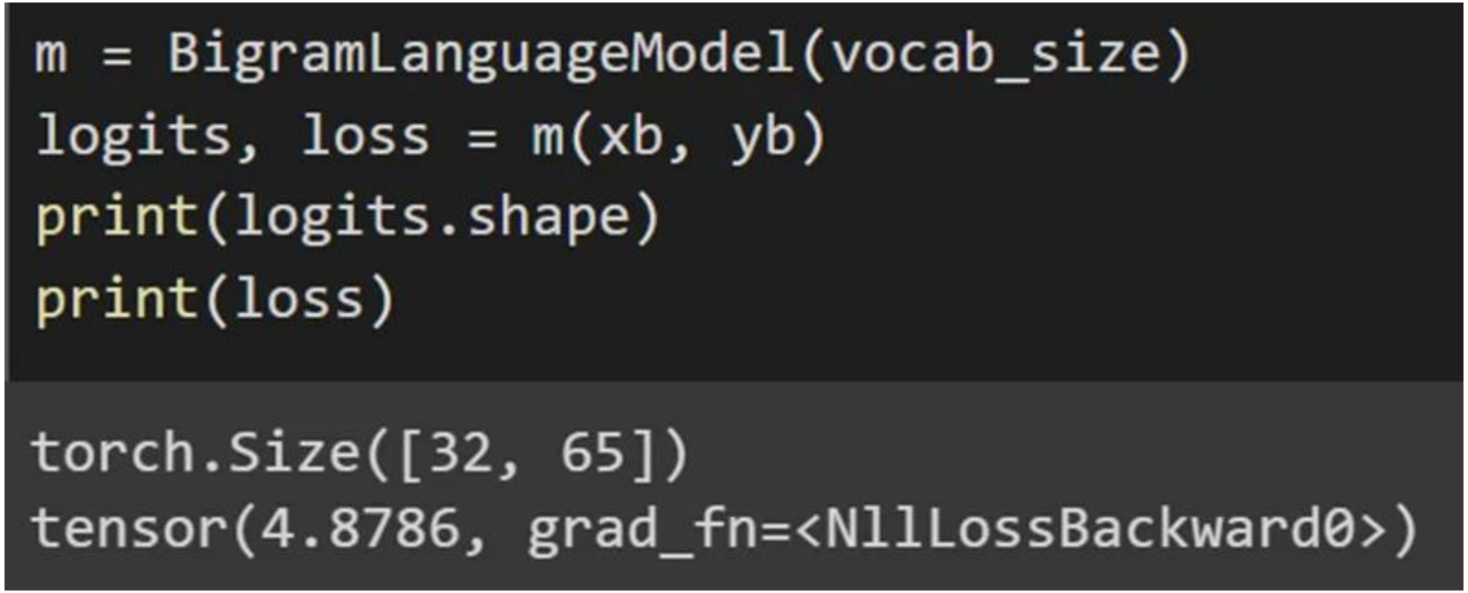
Neural Bigram: Loss



```
class BigramLanguageModel(nn.Module):

    def __init__(self, vocab_size):
        ...

    def forward(self, idx, targets=None):
        # (B,T,C)
        logits = self.token_embedding_table(idx)

        B, T, C = logits.shape
        logits = logits.view(B*T, C)
        targets = targets.view(B*T)

        loss = F.cross_entropy(logits, targets)

        return logits, loss
```

We convert Logits into 2-dimensional array, preserve channel dimension as second dimension. Similarly, we must also convert the targets from a 2-dimensional to a 1-dimensional array to maintain the same format.

Neural Bigram: Loss

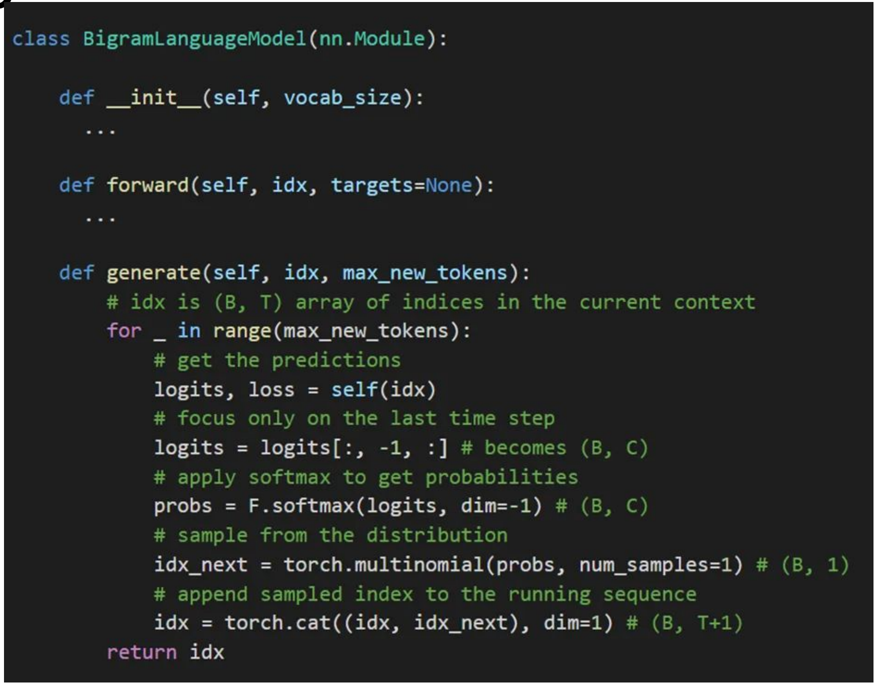


```
m = BigramLanguageModel(vocab_size)
logits, loss = m(xb, yb)
print(logits.shape)
print(loss)
```

```
torch.Size([32, 65])
tensor(4.8786, grad_fn=<NllLossBackward0>)
```

Calling the bi-gram class should return the output with no error.

Neural Bigram: Generation

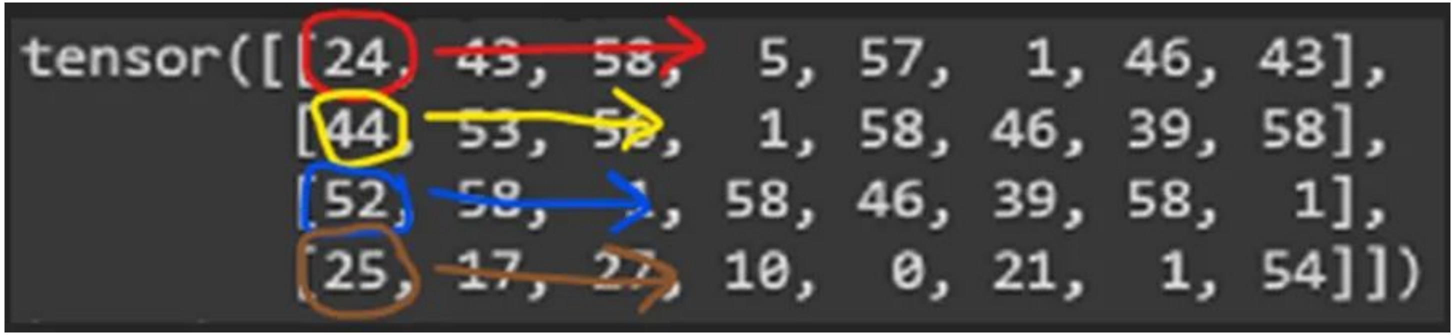


```
class BigramLanguageModel(nn.Module):

    def __init__(self, vocab_size):
      ...

    def forward(self, idx, targets=None):
      ...

    def generate(self, idx, max_new_tokens):
        # idx is (B, T) array of indices in the current context
        for _ in range(max_new_tokens):
            # get the predictions
            logits, loss = self(idx)
            # focus only on the last time step
            logits = logits[:, -1, :] # becomes (B, C)
            # apply softmax to get probabilities
            probs = F.softmax(logits, dim=-1) # (B, C)
            # sample from the distribution
            idx_next = torch.multinomial(probs, num_samples=1) # (B, 1)
            # append sampled index to the running sequence
            idx = torch.cat((idx, idx_next), dim=1) # (B, T+1)
        return idx
```

We define a generate function inside our Bi-Gram class. Two parameters have been passed inside the generate function. Idx and max_new_tokens.

Neural Bigram: Generation

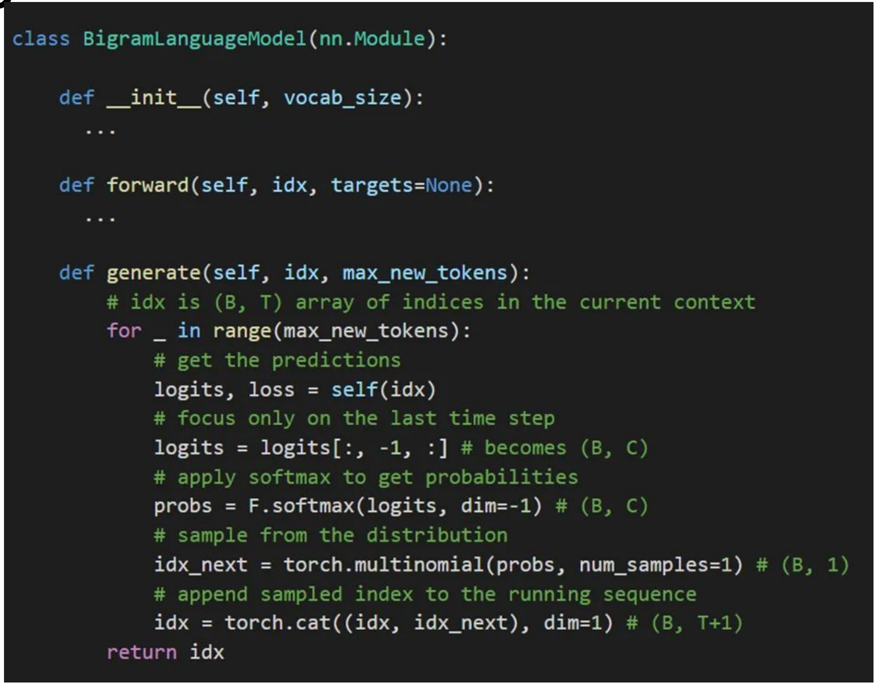
idx refers to current context of characters in some batch, (B,T) dimension and extend it to (B,T)+1,+2,+3…. And so on until max_new_tokens, let me show visually how it is working in our train input matrix diagram:



```
tensor([[24, 43, 58,  5, 57,  1, 46, 43],
        [44, 53, 56,  1, 58, 46, 39, 58],
        [52, 58,  1, 58, 46, 39, 58,  1],
        [25, 17, 27, 10,  0, 21,  1, 54]])
```

Each coloured number is used to predicting to the next number of with respect to the row it belongs to. For example, 24 is predicting 43, then 24 and 43 predicting 58, this will continue till total number of predictions reaches to max_new_tokens value and this approach which we stated earlier is called character level language modelling.

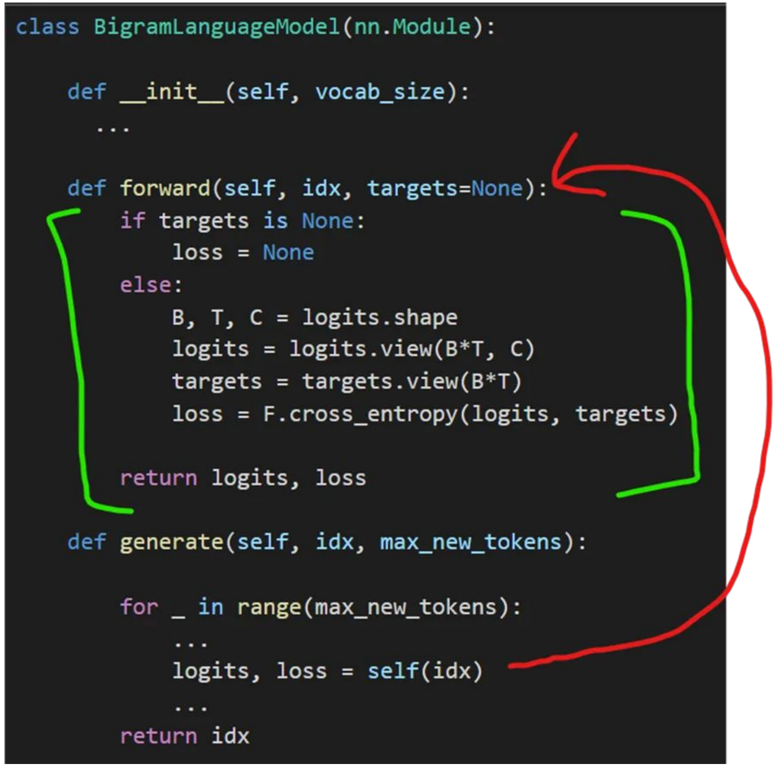
Neural Bigram: Generation



```python
class BigramLanguageModel(nn.Module):

    def __init__(self, vocab_size):
      ...

    def forward(self, idx, targets=None):
      ...

    def generate(self, idx, max_new_tokens):
        # idx is (B, T) array of indices in the current context
        for _ in range(max_new_tokens):
            # get the predictions
            logits, loss = self(idx)
            # focus only on the last time step
            logits = logits[:, -1, :] # becomes (B, C)
            # apply softmax to get probabilities
            probs = F.softmax(logits, dim=-1) # (B, C)
            # sample from the distribution
            idx_next = torch.multinomial(probs, num_samples=1) # (B, 1)
            # append sampled index to the running sequence
            idx = torch.cat((idx, idx_next), dim=1) # (B, T+1)
        return idx
```

idx value updated based on probabilities which is calculated using softmax and attaching each new predicted character with the previous value from which it got predicted and returning it in the last line.

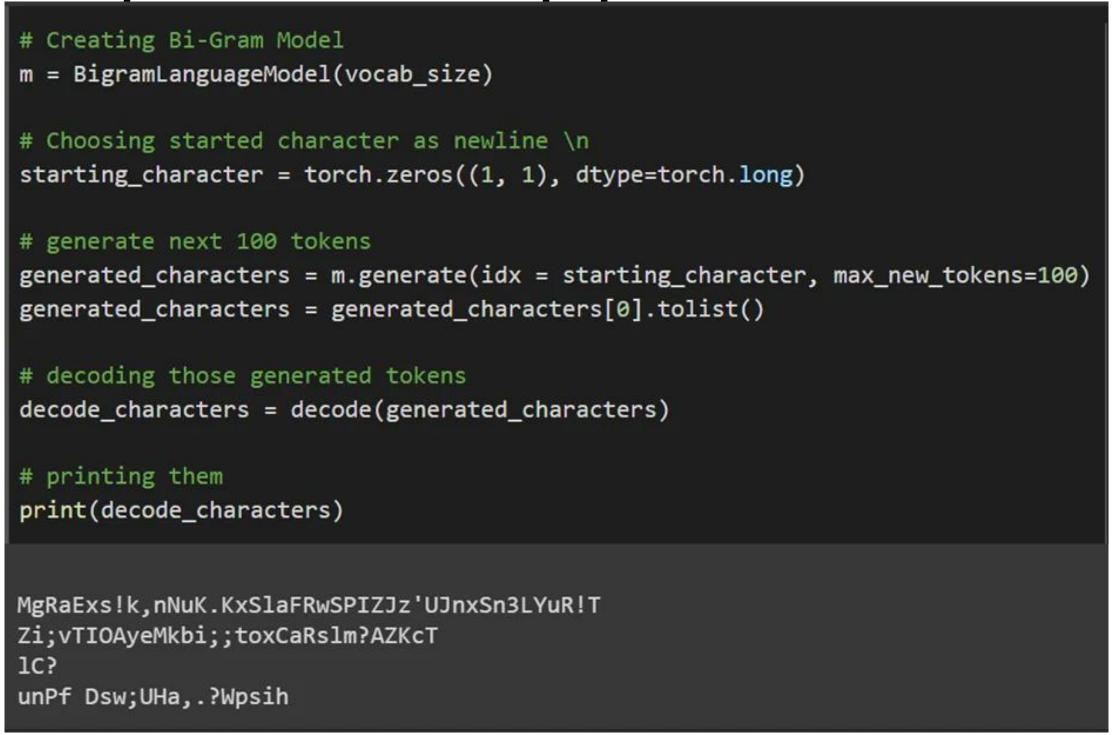
Neural Bigram: Generation



```
class BigramLanguageModel(nn.Module):

    def __init__(self, vocab_size):
      ...

    def forward(self, idx, targets=None):
        if targets is None:
            loss = None
        else:
            B, T, C = logits.shape
            logits = logits.view(B*T, C)
            targets = targets.view(B*T)
            loss = F.cross_entropy(logits, targets)

        return logits, loss

    def generate(self, idx, max_new_tokens):

        for _ in range(max_new_tokens):
            ...
            logits, loss = self(idx)
            ...
        return idx
```

if condition we call tell our forward function that if not target was passed no loss will be calculated otherwise it should be.

Neural Bigram: Whole pipeline

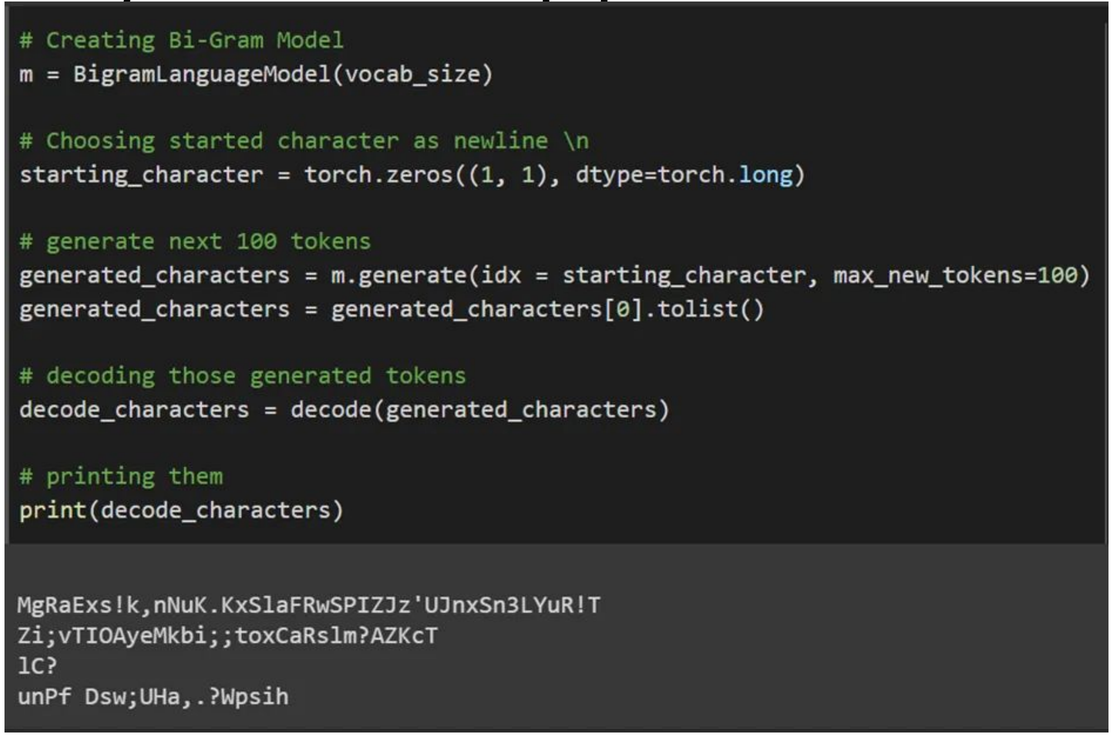


```
# Creating Bi-Gram Model
m = BigramLanguageModel(vocab_size)

# Choosing started character as newline \n
starting_character = torch.zeros((1, 1), dtype=torch.long)

# generate next 100 tokens
generated_characters = m.generate(idx = starting_character, max_new_tokens=100)
generated_characters = generated_characters[0].tolist()

# decoding those generated tokens
decode_characters = decode(generated_characters)

# printing them
print(decode_characters)
```

```
MgRaExs!k,nNuK.KxSlaFRwSPIZJz'UJnxSn3LYuR!T
Zi;vTIOAyeMkbi;;toxCaRslm?AZKcT
lC?
unPf Dsw;UHa,.?Wpsih
```

I choose starting character as new line using torch.zeroes(1, 1), predicting next 100 tokens and then decoding them and printing.

Neural Bigram: Whole pipeline

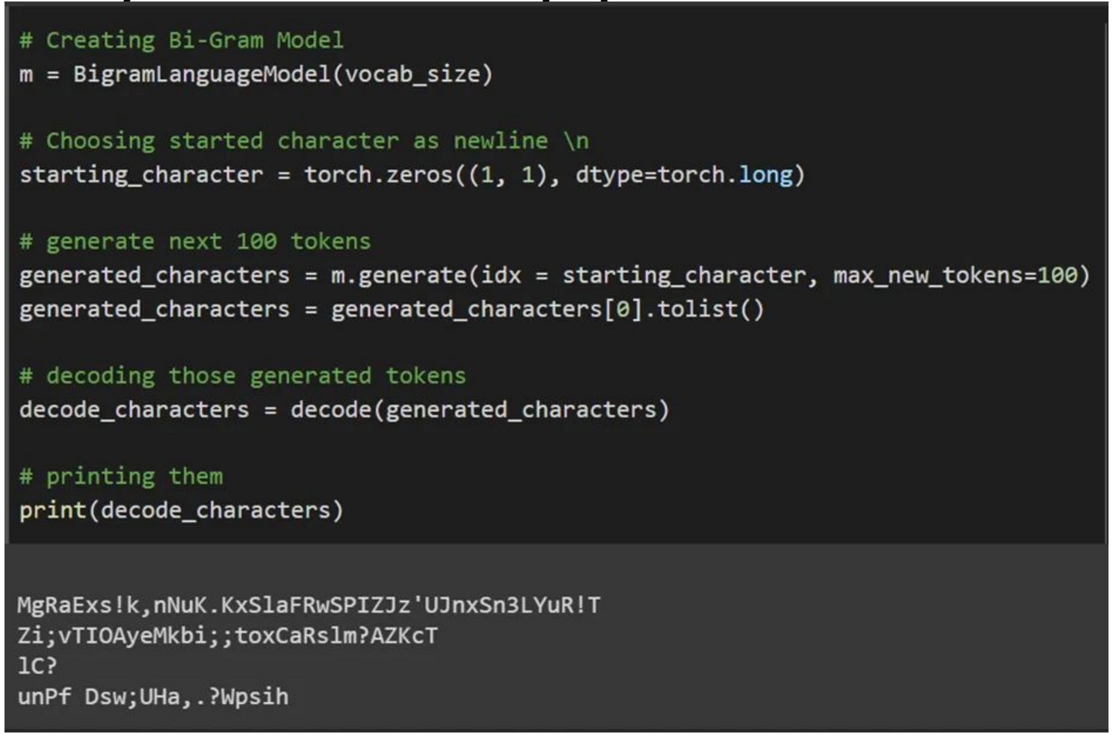


```python
# Creating Bi-Gram Model
m = BigramLanguageModel(vocab_size)

# Choosing started character as newline \n
starting_character = torch.zeros((1, 1), dtype=torch.long)

# generate next 100 tokens
generated_characters = m.generate(idx = starting_character, max_new_tokens=100)
generated_characters = generated_characters[0].tolist()

# decoding those generated tokens
decode_characters = decode(generated_characters)

# printing them
print(decode_characters)
```

```

MgRaExs!k,nNuK.KxSlaFRwSPIZJz'UJnxSn3LYuR!T
Zi;vTIOAyeMkbi;;toxCaRslm?AZKcT
lC?
unPf Dsw;UHa,.?Wpsih
```

Why garbage predictions?

Neural Bigram: Whole pipeline

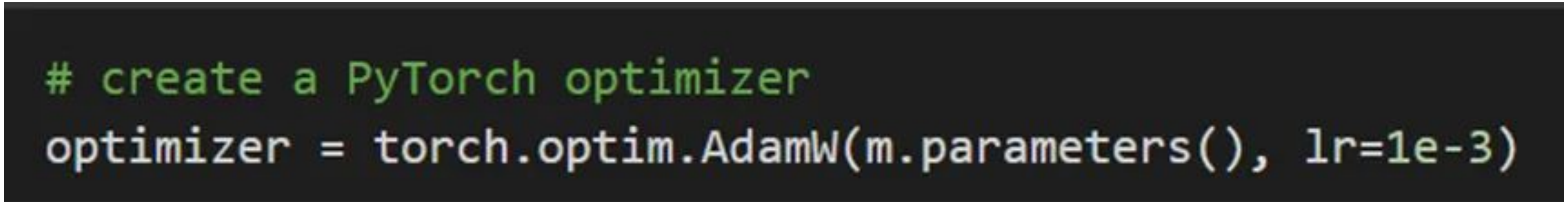


```
# Creating Bi-Gram Model
m = BigramLanguageModel(vocab_size)

# Choosing started character as newline \n
starting_character = torch.zeros((1, 1), dtype=torch.long)

# generate next 100 tokens
generated_characters = m.generate(idx = starting_character, max_new_tokens=100)
generated_characters = generated_characters[0].tolist()

# decoding those generated tokens
decode_characters = decode(generated_characters)

# printing them
print(decode_characters)
```

```
MgRaExs!k,nNuK.KxSlaFRwSPIZJz'UJnxSn3LYuR!T
Zi;vTIOAyeMkbi;;toxCaRslm?AZKcT
lC?
unPf Dsw;UHa,.?Wpsih
```

Exactly! We need still to train the model

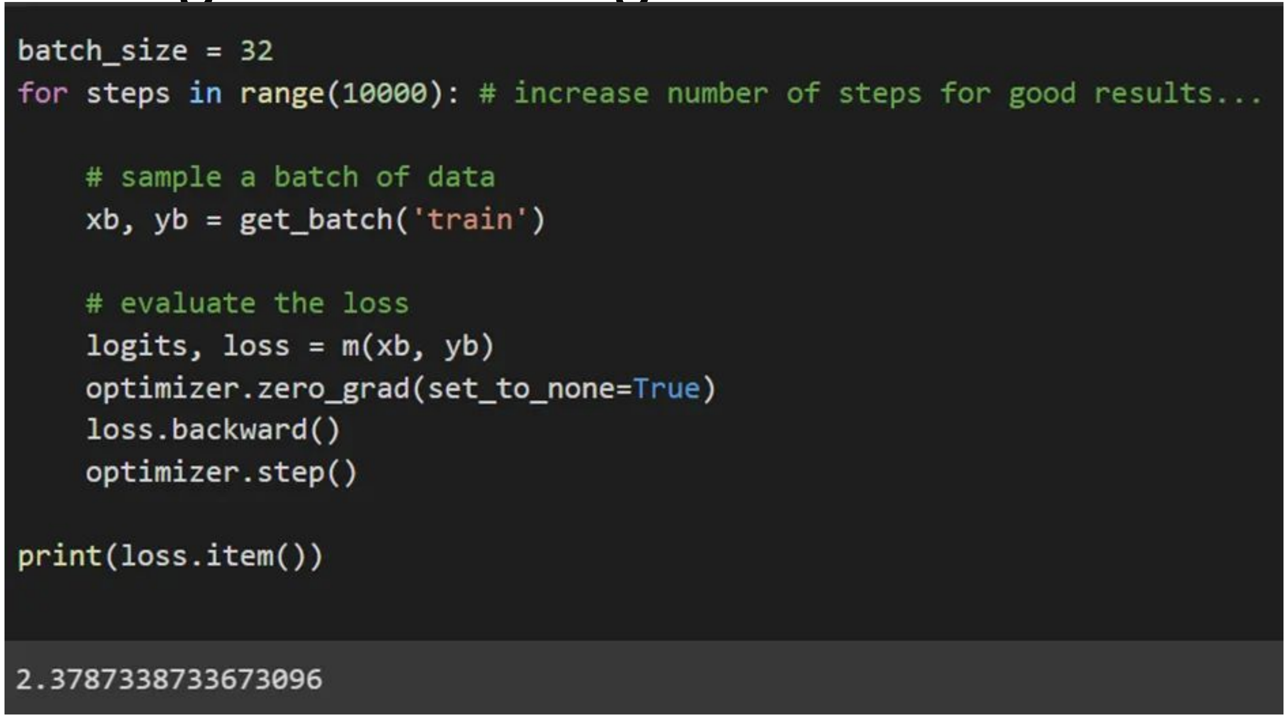
Neural Bigram: Training



```python
# create a PyTorch optimizer
optimizer = torch.optim.AdamW(m.parameters(), lr=1e-3)
```

The learning rate is roughly 1e-3 but for very small networks you can use a much higher learning rate.

Neural Bigram: Training

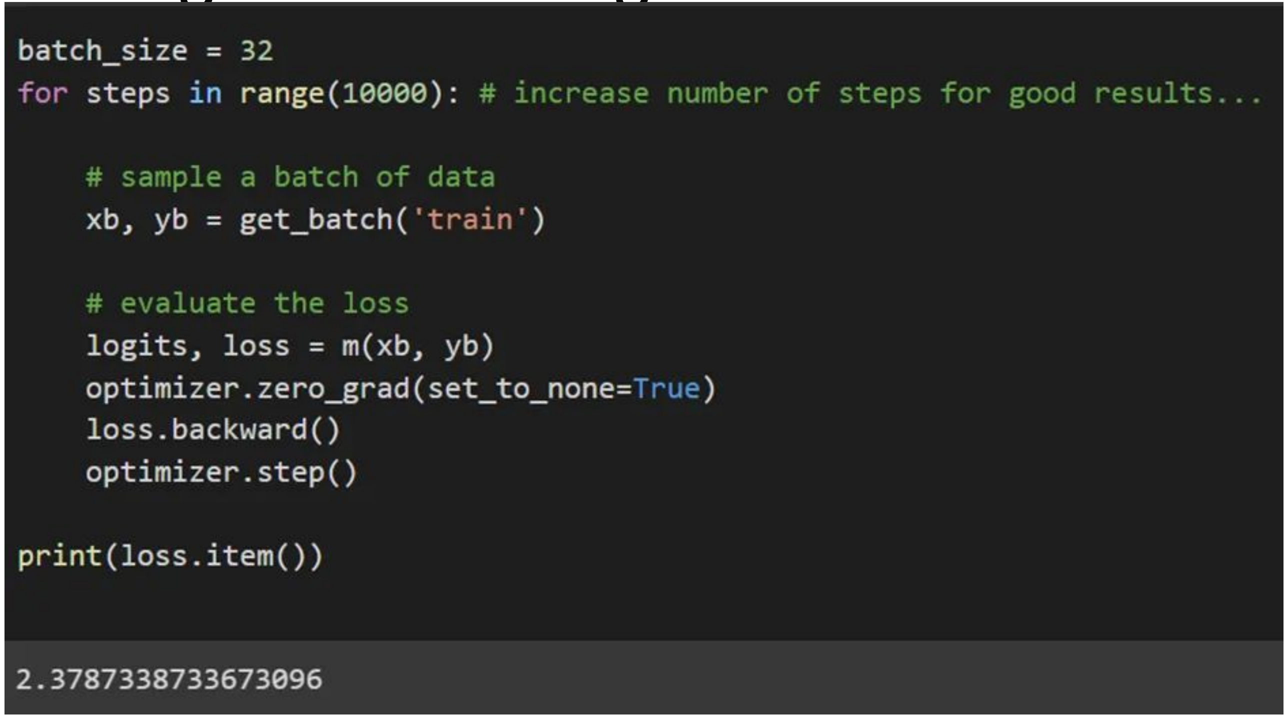


```
batch_size = 32
for steps in range(10000): # increase number of steps for good results...

    # sample a batch of data
    xb, yb = get_batch('train')

    # evaluate the loss
    logits, loss = m(xb, yb)
    optimizer.zero_grad(set_to_none=True)
    loss.backward()
    optimizer.step()

print(loss.item())
```

```
2.3787338733673096
```

We were working with batch_size 4, but now we increase it to reduce the loss

Neural Bigram: Training

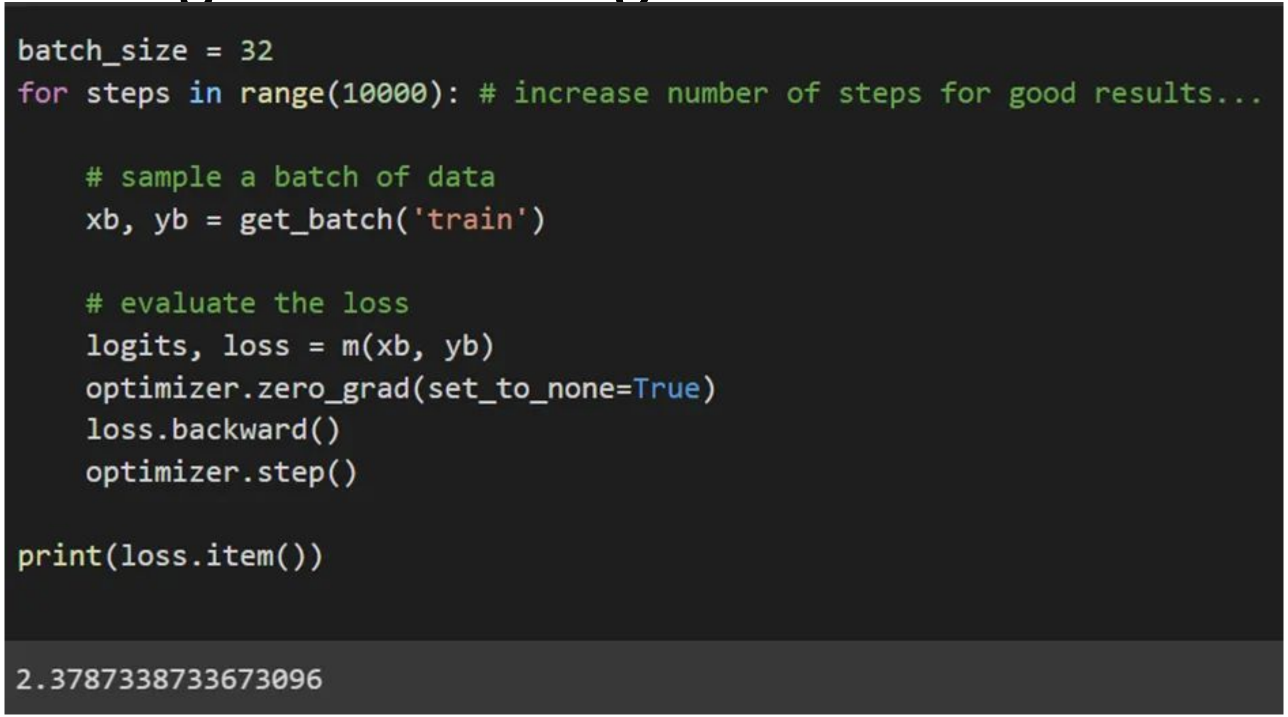


```
batch_size = 32
for steps in range(10000): # increase number of steps for good results...

    # sample a batch of data
    xb, yb = get_batch('train')

    # evaluate the loss
    logits, loss = m(xb, yb)
    optimizer.zero_grad(set_to_none=True)
    loss.backward()
    optimizer.step()

print(loss.item())
```

```
2.3787338733673096
```

optimizer.zero_grad is zeroing out all the gradients from the previous step and getting the gradients for all the parameters and then using those gradients to update our parameters

Neural Bigram: Training

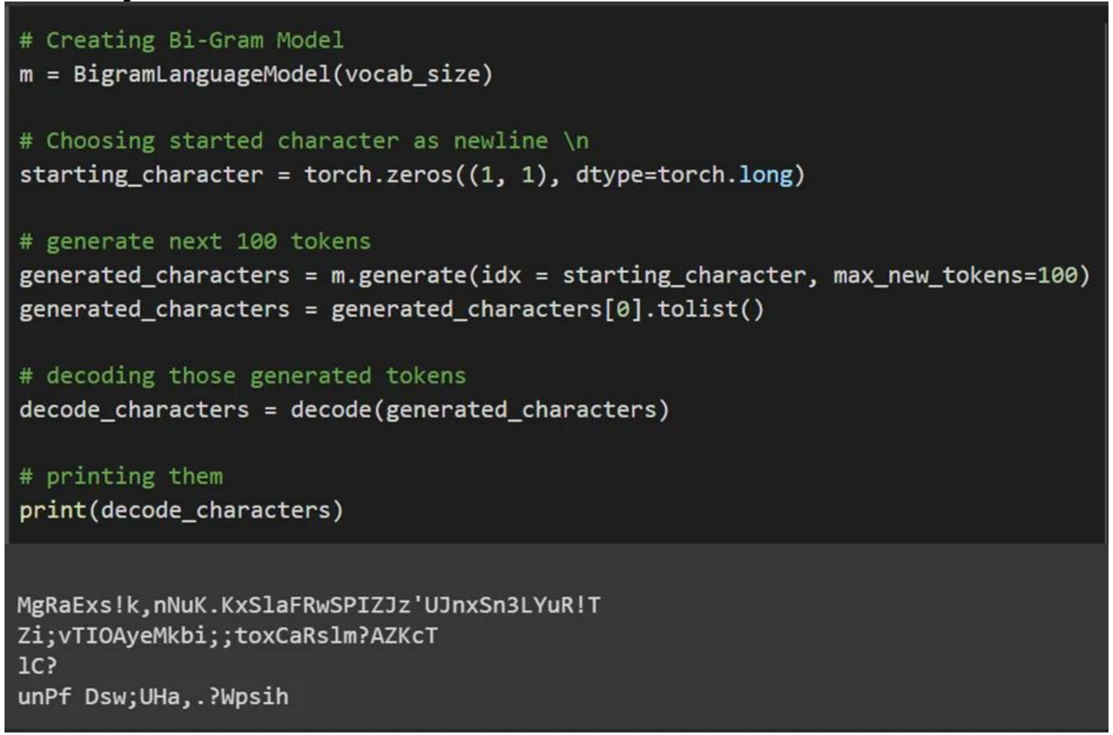


```
batch_size = 32
for steps in range(10000): # increase number of steps for good results...

    # sample a batch of data
    xb, yb = get_batch('train')

    # evaluate the loss
    logits, loss = m(xb, yb)
    optimizer.zero_grad(set_to_none=True)
    loss.backward()
    optimizer.step()

print(loss.item())
```

```
2.3787338733673096
```

Running about 10k iterations reduces the loss to 2.3787 from 4.8786, you can reduce it further by running more iterations (although you shall use early stopping in the end).

Neural Bigram: Generation



```
# Creating Bi-Gram Model
m = BigramLanguageModel(vocab_size)

# Choosing started character as newline \n
starting_character = torch.zeros((1, 1), dtype=torch.long)

# generate next 100 tokens
generated_characters = m.generate(idx = starting_character, max_new_tokens=100)
generated_characters = generated_characters[0].tolist()

# decoding those generated tokens
decode_characters = decode(generated_characters)

# printing them
print(decode_characters)
```

```
MgRaExs!k,nNuK.KxSlaFRwSPIZJz'UJnxSn3LYuR!T
Zi;vTIOAyeMkbi;;toxCaRslm?AZKcT
lC?
unPf Dsw;UHa,.?Wpsih
```

I choose starting character as new line using torch.zeroes(1, 1), predicting next 100 tokens and then decoding them and printing.

GPT from scratch

- Words and Corpora

GPT from scratch

- Words and Corpora

GPT from scratch

- Words and Corpora

GPT from scratch

- Words and Corpora