HW 1 - MLP and Character Language Modeling-4_TaichenZhou

January 30, 2023

[ ]:
```
import torch
from torch.utils.data import DataLoader
from torch.utils.data.dataset import random_split
import torch.nn as nn
from torch.utils.data import Dataset
from torch.utils.data import DataLoader, TensorDataset
import time
from tqdm import tqdm
```

0.0.1 Information

- We will do a few preliminary exercises and also build a character level MLP language model.
- This model will be similar to the model we did in class, except that we will have characters as tokens, not words.
- You will need a conda environment for this, here is general information on this.
- https://docs.conda.io/projects/conda/en/latest/user-guide/install/index.html
- PyTorch: https://anaconda.org/pytorch/pytorch

In the code below, FILL-IN the code necessary in the hint string provided.

[ ]:
```

```

0.0.2 Preliminary exercises

- Please fill in the cells below with the asked for data.

[ ]:
```
torch.manual_seed(1)
```

[ ]:
```
<torch._C.Generator at 0x7fdeaf91acb0>
```

[ ]:
```
# Create an embedding layer for a vocabulary of size 10 and the word vectors␣
 ↪are each of dimension 5.
e = nn.Embedding(10,5)

# Extract the embedding for the word whose token index is 3. What is the shape␣
 ↪of this vector?
v = e(torch.tensor(3))
print("Shape of the vector: ", v.size())
```

```
# Extract the weight matrix from the layer e.
# Create a linear layer (with no bias) of size 10 by 5 and set it's data to the
 ↪embedding matrix.
l = nn.Linear(5,10, bias = False)
l.weight = e.weight

# Insert inside of the assert below some sort of equality check between l.
 ↪weight and e.weight; it should pass to true.
# Hint: look up torch.all() and torch.eq()
assert(torch.eq(e.weight, l.weight).all())
```

```
Shape of the vector:  torch.Size([5])
```

[ ]:
```
# Create a batch of size 2 with entries [0, 1, 2] and [2, 3, 4] in the data
 ↪batch.
x = torch.tensor([[0, 1, 2], [2, 3, 4]])
```

[ ]:
```
# What is the dimesion of this batch ran through the embeding layer?
assert(e(x).shape == torch.Size([2,3,5]))
```

[ ]:
```

```

0.0.3 Constants and configs used below.

[ ]:
```
DEVICE = "cpu"
LR = 4.0
BATCH_SIZE = 16
NUM_EPOCHS = 5
MARKER = '.'
# N-gram level; P(w_t | w_{t-1}, ..., w_{t-n+1}).
# We use 3 words to predict the next word.
n = 4
# Hidden layer dimension.
h = 20
# Word embedding dimension.
m = 20
```

[ ]:
```

```

0.0.4 Get the dataset and the tokenizer.

[ ]:
```
class CharDataset(Dataset):
    def __init__(self, words, chars):
        self.words = words
        self.chars = chars
```

```python
        # Inverse dictionaries mapping char tokens to unique ids and the␣
↪reverse.
        # Tokens in this case are the unique chars we passed in above.
        # Each token should be mappend to a unique integer and MARKER should␣
↪have token 0.
        # For example, stoi should be like {'.' -> 0, 'a' -> 1, 'b' -> 2} if I␣
↪pass in chars = '.ab'.
        dic_stoi, dic_itos = {}, {}
        count = 0
        for ele in chars:
            dic_stoi[ele] = count
            dic_itos[count] = ele
            count += 1
        self.stoi = dic_stoi
        self.itos = dic_itos # Inverse mapping.

    def __len__(self):
        # Number of words.
        return len(self.words)

    def contains(self, word):
        # Check if word is in self.words and return True/False if it is, is not.
        return True if word in self.words else False

    def get_vocab_size(self):
        # Return the vocabulary size.
        return len(self.chars)

    def encode(self, word):
        # Express this word as a list of int ids. For example, maybe ".abc" ->␣
↪[0, 1, 2, 3].
        # This assumes 'a' -> 1, etc.
        result = []
        for char in word:
            result.append(self.stoi[char])
        return result

    def decode(self, tokens):
        # For a set of tokens, return back the string.
        # For example, maybe [1, 1, 2] -> "aac"
        result = []
        for tok in tokens:
            result.append(self.itos[tok])
        return result

    def __getitem__(self, idx):
        # This is used so we can loop over the data.
```

```
        word = self.words[idx]
        return self.encode(word)
```

[ ]:

```

```

[ ]:

```
def create_datasets(window, input_file = 'names.txt'):
    """
    This takes a file of words and separates all the words.
    It then gets all the characters present in the universe of words and then␣
↪ouputs the statistics.
    """
    with open(input_file, 'r') as f:
        data = f.read()
    # Split the file by new lines. You should get a list of names.
    words = data.split('\n')
    words = [word.replace(' ', '') for word in words] # This gets rid of any␣
↪trailing and starting white spaces.
    words = [word for word in words if word] # Filter out all the empty words.


    chars = sorted(list(set([char for word in words for char in word]))) # This␣
↪gets the universe of all characters.

    # Will force chars to have MARKER having index 0.
    chars= [MARKER] + chars

    # Pad each word with a context window of size n-1.
    # Why? a word like "abc" should becomes "..abc.." if the window is size 3.
    # This is some we can get pair of (x, y) data like this: ".." -> "a", ".a"␣
↪-> "b", "ab" -> "c", "bc" -> ".", "c." -> "."
    # I.e. this allows us to know that "a" is a start character.
    # So you should get something like ["ab", "c"] -> ["..ab..", "..c.."], for␣
↪example.
    words = [('.'*(window-1))+word+('.'*(window-1)) for word in words]

    print(f"The number of examples in the dataset: {len(words)}")
    print(f"The number of unique characters in the vocabulary: {len(chars)}")
    print(f"The vocabulary we have is: {''.join(chars)}")

    # Partition the input data into a training, validation, and the test set.
    out_of_sample_set_size = min(2000, int(len(words) * 0.1)) # We use 10% of␣
↪the training set, or up to 2000 examples.
    test_set_size = 1500

    # First, get a random permutation of randomly permute of size len(words).
    # Then, convert this to a list.
```

```
    # This index list is used below to get the train, validation, and test sets.
    rp = torch.randperm(len(words)).tolist()

    # Get train, validation, and test set.
    train_words = [words[i] for i in rp[:-out_of_sample_set_size]]
    validation_words = [words[i] for i in rp[-out_of_sample_set_size:
↪-test_set_size]]
    test_words = [words[i] for i in rp[-test_set_size:]]

    print(f"We've split up the dataset into {len(train_words)},␣
↪{len(validation_words)}, {len(test_words)} training, validation, and test␣
↪examples")

    # But the data in the data set objects.
    train_dataset = CharDataset(train_words, chars)
    validation_dataset = CharDataset(validation_words, chars)
    test_dataset = CharDataset(test_words, chars)

    return train_dataset, validation_dataset, test_dataset
```

[ ]:
```
train_dataset, validation_dataset, test_dataset = create_datasets(n)
```

```
The number of examples in the dataset: 32033
The number of unique characters in the vocabulary: 27
The vocabulary we have is: .abcdefghijklmnopqrstuvwxyz
We've split up the dataset into 30033, 500, 1500 training, validation, and test
examples
```

0.1 Explore the data

[ ]:
```
# Get the first word in "train_dataset"
train_dataset.words[0]
```

[ ]: '…niyam…'

[ ]:
```
# Get the stoi map of train_dataset. How many keys does it have?
print(len(train_dataset.stoi))
print(train_dataset.get_vocab_size())
```

```
27
27
```

[ ]:
```
```

0.1.1 Get the dataloader

[ ]:
```
def create_dataloader(dataset, window):
    x_list = []
    y_list = []
    # For ech word.
    for i, word in enumerate(dataset):
        # Grab a context of size window and window-1 characters will be in x, 1␣
↪will be in y.
        for j, _ in enumerate(word):
            # If there is no widow of size window left, break.
            if j + window > len(word) - 1:
                break
            word_window = word[j:j+window]
            x, y = word_window[:window-1], word_window[-1]
            x_list.append(x)
            y_list.append(y)

    return DataLoader(
        TensorDataset(torch.tensor(x_list), torch.tensor(y_list)),
        BATCH_SIZE,
        shuffle=True
    )
```

[ ]:
```
train_dataloader = create_dataloader(train_dataset, n)
validation_dataloader = create_dataloader(validation_dataset, n)
test_dataloader = create_dataloader(test_dataset, n)
```

[ ]:
```

```

0.1.2 Set up the model

- Identical to lecture. Please look over that!

[ ]:
```
# One of the first Neural language models!
class CharacterNeuralLanguageModel(nn.Module):
    def __init__(self, V, m, h, n):
        super(CharacterNeuralLanguageModel, self).__init__()

        # Vocabulary size.
        self.V = V

        # Embedding dimension, per word.
        self.m = m

        # Hidden dimension.
        self.h = h
```

```
        # N in "N-gram"
        self.n = n

        # Can you change all this stuff to use nn.Linear?
        # Ca also use nn.Parameter(torch.zeros(V, m)) for self.C but then we␣
↪need one-hot and this is slow.
        self.C = nn.Embedding(V, m)
        self.H = nn.Parameter(torch.zeros((n-1) * m, h))
        self.W = nn.Parameter(torch.zeros((n-1) * m, V))
        self.U = nn.Parameter(torch.zeros(h, V))

        self.b = torch.nn.Parameter(torch.ones(V))
        self.d = torch.nn.Parameter(torch.ones(h))

        self.init_weights()

    def init_weights(self):
        # Intitialize C, H, W, U in a nice way. Use xavier initialization for␣
↪the weights.
        # On a first run, just pass.
        with torch.no_grad():
            torch.nn.init.xavier_uniform_(self.C.weight)
            torch.nn.init.xavier_uniform_(self.H)
            torch.nn.init.xavier_uniform_(self.W)
            torch.nn.init.xavier_uniform_(self.U)


    def forward(self, x):

        # x is of dimenson N = batch size X n-1

        # N X (n-1) X m
        x = self.C(x)

        # N
        N = x.shape[0]

        # N X (n-1) * m
        x = x.view(N, -1)

        # N X V
        y = self.b + torch.matmul(x, self.W) + torch.matmul(nn.Tanh()(self.d +␣
↪torch.matmul(x, self.H)), self.U)

        return y
```

[ ]:

0.1.3 Set up the model.

[ ]:
```
# Identical to lecture.
criterion = torch.nn.CrossEntropyLoss().to(DEVICE)
model = CharacterNeuralLanguageModel(
    train_dataset.get_vocab_size(), m, h, n).to(DEVICE)
optimizer = torch.optim.SGD(model.parameters(), lr=LR)
scheduler = torch.optim.lr_scheduler.StepLR(optimizer, 1.0, gamma=0.1)
```

[ ]:
```
# How many parameters does the neural network have?
# Hint: look up model.named_parameters and the method "nelement" on a tensor.
# See also the XOR notebook where we count the gradients that are 0.
# There, we loop over the parameters.
number_parameters = 0
for name, param in model.named_parameters():
    print(name, "Number of elements: ", param.numel())
    number_parameters += param.numel()
print("Total number of parameter is {}".format(number_parameters))
print()
```

```
H Number of elements:  1200
W Number of elements:  1620
U Number of elements:  540
b Number of elements:  27
d Number of elements:  20
C.weight Number of elements:  540
Total number of parameter is 3947
```

[ ]:
```

```

0.1.4 Train the model.

[ ]:
```
def calculate_perplexity(total_loss, total_batches):
    return torch.exp(torch.tensor(total_loss / total_batches)).item()
```

[ ]:
```
def train(dataloader, model, optimizer, criterion, epoch):
    model.train()
    total_loss, total_batches = 0.0, 0.0
    log_interval = 500

    for idx, (x, y) in tqdm(enumerate(dataloader)):
        optimizer.zero_grad()

        logits = model(x)

        # Get the loss.
```

```
        loss = criterion(input=logits, target=y.view(-1))

        # Do back propagation.
        loss.backward()

        # Clip the gradients so they don't explode. Look at how this is done in␣
 ↪lecture.
        torch.nn.utils.clip_grad_norm_(model.parameters(), 0.1)

        # Do an optimization step.
        optimizer.step()
        total_loss += loss.item()
        total_batches += 1

        if idx % log_interval == 0 and idx > 0:
            perplexity = calculate_perplexity(total_loss,  total_batches)
            print(
                "| epoch {:3d} "
                "| {:5d}/{:5d} batches "
                "| perplexity {:8.3f} "
                "| loss {:8.3f} "
                .format(
                    epoch,
                    idx,
                    len(dataloader),
                    perplexity,
                    total_loss / total_batches,
                )
            )
            total_loss, total_batches = 0.0, 0
```

[ ]:
```
def evaluate(dataloader, model, criterion):
    model.eval()
    total_loss, total_batches = 0.0, 0

    with torch.no_grad():
        for idx, (x, y) in enumerate(dataloader):
            logits = model(x)
            total_loss += criterion(input=logits, target=y.squeeze(-1)).item()
            total_batches += 1
    return total_loss / total_batches, calculate_perplexity(total_loss, ␣
 ↪total_batches)
```

[ ]:
```
for epoch in range(1, NUM_EPOCHS + 1):
    epoch_start_time = time.time()
    train(train_dataloader, model, optimizer, criterion, epoch)
    loss_val, perplexity_val = evaluate(validation_dataloader, model, criterion)
```

```
    scheduler.step()
    print("-" * 59)
    print(
        "| end of epoch {:3d} "
        "| time: {:5.2f}s "
        "| valid perplexity {:8.3f} "
        "| valid loss {:8.3f}".format(
            epoch,
            time.time() - epoch_start_time,
            perplexity_val,
            loss_val
        )
    )
    print("-" * 59)

print("Checking the results of test dataset.")
loss_test, perplexity_test = evaluate(test_dataloader, model, criterion)
print("test perplexity {:8.3f} | test loss {:8.3f} ".format(perplexity_test,
 ↪loss_test))
```

```
652it [00:00, 1204.18it/s]

| epoch   1 |   500/15247 batches | perplexity    7.055 | loss    1.954

1143it [00:01, 1133.42it/s]

| epoch   1 |  1000/15247 batches | perplexity    7.039 | loss    1.952

1644it [00:01, 1233.75it/s]

| epoch   1 |  1500/15247 batches | perplexity    7.031 | loss    1.950

2167it [00:01, 1288.65it/s]

| epoch   1 |  2000/15247 batches | perplexity    7.092 | loss    1.959

2679it [00:02, 1148.78it/s]

| epoch   1 |  2500/15247 batches | perplexity    7.226 | loss    1.978

3122it [00:02, 1072.63it/s]

| epoch   1 |  3000/15247 batches | perplexity    7.010 | loss    1.947

3688it [00:03, 1071.82it/s]

| epoch   1 |  3500/15247 batches | perplexity    7.047 | loss    1.953

4141it [00:03, 1106.59it/s]

| epoch   1 |  4000/15247 batches | perplexity    7.235 | loss    1.979

4589it [00:04, 1074.25it/s]

| epoch   1 |  4500/15247 batches | perplexity    7.189 | loss    1.973
```

```
5143it [00:04, 1033.44it/s]
| epoch   1 |  5000/15247 batches | perplexity    7.137 | loss    1.965
5727it [00:05, 1139.80it/s]
| epoch   1 |  5500/15247 batches | perplexity    7.000 | loss    1.946
6206it [00:05, 1173.37it/s]
| epoch   1 |  6000/15247 batches | perplexity    6.884 | loss    1.929
6694it [00:05, 1169.13it/s]
| epoch   1 |  6500/15247 batches | perplexity    7.068 | loss    1.956
7192it [00:06, 1229.33it/s]
| epoch   1 |  7000/15247 batches | perplexity    7.078 | loss    1.957
7680it [00:06, 1196.86it/s]
| epoch   1 |  7500/15247 batches | perplexity    6.963 | loss    1.941
8174it [00:07, 1221.80it/s]
| epoch   1 |  8000/15247 batches | perplexity    7.056 | loss    1.954
8668it [00:07, 1204.11it/s]
| epoch   1 |  8500/15247 batches | perplexity    7.246 | loss    1.980
9167it [00:08, 1197.72it/s]
| epoch   1 |  9000/15247 batches | perplexity    7.292 | loss    1.987
9652it [00:08, 1199.49it/s]
| epoch   1 |  9500/15247 batches | perplexity    6.988 | loss    1.944
10122it [00:08, 1068.01it/s]
| epoch   1 | 10000/15247 batches | perplexity    7.038 | loss    1.951
10693it [00:09, 1118.09it/s]
| epoch   1 | 10500/15247 batches | perplexity    6.982 | loss    1.943
11154it [00:09, 1145.58it/s]
| epoch   1 | 11000/15247 batches | perplexity    6.939 | loss    1.937
11731it [00:10, 1141.02it/s]
| epoch   1 | 11500/15247 batches | perplexity    7.001 | loss    1.946
12193it [00:10, 1132.54it/s]
| epoch   1 | 12000/15247 batches | perplexity    7.148 | loss    1.967
12656it [00:11, 1135.12it/s]
| epoch   1 | 12500/15247 batches | perplexity    7.186 | loss    1.972
```

```
13120it [00:11, 1126.13it/s]
| epoch   1 | 13000/15247 batches | perplexity    7.096 | loss    1.960
13676it [00:12, 1051.38it/s]
| epoch   1 | 13500/15247 batches | perplexity    7.105 | loss    1.961
14188it [00:12, 995.58it/s]
| epoch   1 | 14000/15247 batches | perplexity    7.136 | loss    1.965
14670it [00:13, 934.88it/s]
| epoch   1 | 14500/15247 batches | perplexity    7.016 | loss    1.948
15125it [00:13, 886.58it/s]
| epoch   1 | 15000/15247 batches | perplexity    7.147 | loss    1.967
15247it [00:13, 1093.59it/s]
------------------------------------------------------------
| end of epoch   1 | time: 14.03s | valid perplexity    7.021 | valid loss
1.949
------------------------------------------------------------
609it [00:00, 1037.22it/s]
| epoch   2 |   500/15247 batches | perplexity    6.957 | loss    1.940
1135it [00:01, 1018.60it/s]
| epoch   2 |  1000/15247 batches | perplexity    7.110 | loss    1.962
1662it [00:01, 1047.35it/s]
| epoch   2 |  1500/15247 batches | perplexity    6.972 | loss    1.942
2196it [00:02, 1058.78it/s]
| epoch   2 |  2000/15247 batches | perplexity    7.136 | loss    1.965
2617it [00:02, 1035.40it/s]
| epoch   2 |  2500/15247 batches | perplexity    6.980 | loss    1.943
3141it [00:03, 1024.39it/s]
| epoch   2 |  3000/15247 batches | perplexity    7.032 | loss    1.950
3676it [00:03, 1062.58it/s]
| epoch   2 |  3500/15247 batches | perplexity    6.759 | loss    1.911
4108it [00:03, 1048.53it/s]
| epoch   2 |  4000/15247 batches | perplexity    7.159 | loss    1.968
4638it [00:04, 1005.38it/s]
| epoch   2 |  4500/15247 batches | perplexity    6.869 | loss    1.927
```

```
5168it [00:05, 1025.52it/s]
| epoch   2 |  5000/15247 batches | perplexity    6.795 | loss    1.916
5706it [00:05, 1069.70it/s]
| epoch   2 |  5500/15247 batches | perplexity    6.923 | loss    1.935
6142it [00:05, 1067.11it/s]
| epoch   2 |  6000/15247 batches | perplexity    6.947 | loss    1.938
6688it [00:06, 1085.27it/s]
| epoch   2 |  6500/15247 batches | perplexity    7.020 | loss    1.949
7123it [00:06, 1042.26it/s]
| epoch   2 |  7000/15247 batches | perplexity    7.104 | loss    1.961
7665it [00:07, 1081.58it/s]
| epoch   2 |  7500/15247 batches | perplexity    6.864 | loss    1.926
8222it [00:07, 1098.26it/s]
| epoch   2 |  8000/15247 batches | perplexity    7.043 | loss    1.952
8660it [00:08, 1079.57it/s]
| epoch   2 |  8500/15247 batches | perplexity    6.930 | loss    1.936
9196it [00:08, 1052.74it/s]
| epoch   2 |  9000/15247 batches | perplexity    7.094 | loss    1.959
9618it [00:09, 1038.58it/s]
| epoch   2 |  9500/15247 batches | perplexity    6.997 | loss    1.945
10152it [00:09, 1061.67it/s]
| epoch   2 | 10000/15247 batches | perplexity    7.102 | loss    1.960
10681it [00:10, 1047.86it/s]
| epoch   2 | 10500/15247 batches | perplexity    6.988 | loss    1.944
11212it [00:10, 1049.16it/s]
| epoch   2 | 11000/15247 batches | perplexity    6.888 | loss    1.930
11626it [00:11, 1008.37it/s]
| epoch   2 | 11500/15247 batches | perplexity    7.183 | loss    1.972
12165it [00:11, 1069.11it/s]
| epoch   2 | 12000/15247 batches | perplexity    6.987 | loss    1.944
12707it [00:12, 1075.67it/s]
| epoch   2 | 12500/15247 batches | perplexity    6.994 | loss    1.945
```

```
13145it [00:12, 1070.03it/s]

| epoch   2 | 13000/15247 batches | perplexity    7.114 | loss    1.962

13686it [00:13, 964.71it/s]

| epoch   2 | 13500/15247 batches | perplexity    7.067 | loss    1.955

14108it [00:13, 1030.37it/s]

| epoch   2 | 14000/15247 batches | perplexity    6.905 | loss    1.932

14645it [00:14, 1062.80it/s]

| epoch   2 | 14500/15247 batches | perplexity    6.713 | loss    1.904

15190it [00:14, 1079.92it/s]

| epoch   2 | 15000/15247 batches | perplexity    6.973 | loss    1.942

15247it [00:14, 1036.19it/s]

-----------------------------------------------------------
| end of epoch   2 | time: 14.79s | valid perplexity    6.954 | valid loss
1.939
-----------------------------------------------------------

616it [00:00, 1021.77it/s]

| epoch   3 |   500/15247 batches | perplexity    6.985 | loss    1.944

1144it [00:01, 1042.22it/s]

| epoch   3 |  1000/15247 batches | perplexity    7.018 | loss    1.948

1674it [00:01, 1046.67it/s]

| epoch   3 |  1500/15247 batches | perplexity    7.055 | loss    1.954

2092it [00:02, 975.77it/s]

| epoch   3 |  2000/15247 batches | perplexity    6.947 | loss    1.938

2619it [00:02, 1051.28it/s]

| epoch   3 |  2500/15247 batches | perplexity    6.924 | loss    1.935

3158it [00:03, 1067.16it/s]

| epoch   3 |  3000/15247 batches | perplexity    7.069 | loss    1.956

3691it [00:03, 1051.07it/s]

| epoch   3 |  3500/15247 batches | perplexity    6.921 | loss    1.935

4116it [00:03, 1046.33it/s]

| epoch   3 |  4000/15247 batches | perplexity    6.909 | loss    1.933

4653it [00:04, 1041.21it/s]

| epoch   3 |  4500/15247 batches | perplexity    7.151 | loss    1.967
```

```
5199it [00:05, 1066.68it/s]
| epoch   3 |  5000/15247 batches | perplexity    7.057 | loss    1.954
5627it [00:05, 1056.04it/s]
| epoch   3 |  5500/15247 batches | perplexity    6.970 | loss    1.942
6173it [00:05, 1066.16it/s]
| epoch   3 |  6000/15247 batches | perplexity    6.988 | loss    1.944
6592it [00:06, 956.46it/s]
| epoch   3 |  6500/15247 batches | perplexity    6.681 | loss    1.899
7138it [00:06, 1080.95it/s]
| epoch   3 |  7000/15247 batches | perplexity    6.911 | loss    1.933
7672it [00:07, 1020.12it/s]
| epoch   3 |  7500/15247 batches | perplexity    7.028 | loss    1.950
8186it [00:07, 1002.14it/s]
| epoch   3 |  8000/15247 batches | perplexity    6.895 | loss    1.931
8687it [00:08, 975.45it/s]
| epoch   3 |  8500/15247 batches | perplexity    6.968 | loss    1.941
9175it [00:09, 956.85it/s]
| epoch   3 |  9000/15247 batches | perplexity    7.063 | loss    1.955
9678it [00:09, 980.54it/s]
| epoch   3 |  9500/15247 batches | perplexity    7.067 | loss    1.955
10201it [00:10, 1040.02it/s]
| epoch   3 | 10000/15247 batches | perplexity    6.895 | loss    1.931
10611it [00:10, 960.69it/s]
| epoch   3 | 10500/15247 batches | perplexity    6.952 | loss    1.939
11145it [00:11, 1038.05it/s]
| epoch   3 | 11000/15247 batches | perplexity    7.031 | loss    1.950
11686it [00:11, 1076.30it/s]
| epoch   3 | 11500/15247 batches | perplexity    7.097 | loss    1.960
12110it [00:11, 1032.21it/s]
| epoch   3 | 12000/15247 batches | perplexity    6.827 | loss    1.921
12628it [00:12, 996.25it/s]
| epoch   3 | 12500/15247 batches | perplexity    6.893 | loss    1.930
```

```
13144it [00:12, 1006.18it/s]
| epoch   3 | 13000/15247 batches | perplexity    7.159 | loss    1.968
13657it [00:13, 1013.87it/s]
| epoch   3 | 13500/15247 batches | perplexity    6.861 | loss    1.926
14167it [00:13, 1011.67it/s]
| epoch   3 | 14000/15247 batches | perplexity    7.091 | loss    1.959
14682it [00:14, 1009.22it/s]
| epoch   3 | 14500/15247 batches | perplexity    6.747 | loss    1.909
15196it [00:15, 1014.91it/s]
| epoch   3 | 15000/15247 batches | perplexity    6.895 | loss    1.931
15247it [00:15, 1009.29it/s]
-----------------------------------------------------------
| end of epoch   3 | time: 15.19s | valid perplexity    6.966 | valid loss
1.941
-----------------------------------------------------------
634it [00:00, 1077.43it/s]
| epoch   4 |   500/15247 batches | perplexity    6.854 | loss    1.925
1177it [00:01, 1061.95it/s]
| epoch   4 |  1000/15247 batches | perplexity    7.060 | loss    1.954
1611it [00:01, 1073.61it/s]
| epoch   4 |  1500/15247 batches | perplexity    7.061 | loss    1.955
2156it [00:02, 923.48it/s]
| epoch   4 |  2000/15247 batches | perplexity    7.131 | loss    1.964
2684it [00:02, 1027.31it/s]
| epoch   4 |  2500/15247 batches | perplexity    6.991 | loss    1.945
3107it [00:03, 1026.95it/s]
| epoch   4 |  3000/15247 batches | perplexity    7.127 | loss    1.964
3633it [00:03, 1038.35it/s]
| epoch   4 |  3500/15247 batches | perplexity    6.851 | loss    1.924
4146it [00:04, 996.15it/s]
| epoch   4 |  4000/15247 batches | perplexity    6.921 | loss    1.935
4645it [00:04, 968.86it/s]
| epoch   4 |  4500/15247 batches | perplexity    6.954 | loss    1.939
```

```
5123it [00:05, 930.66it/s]
| epoch   4 |  5000/15247 batches | perplexity    6.987 | loss    1.944
5640it [00:05, 1012.91it/s]
| epoch   4 |  5500/15247 batches | perplexity    7.008 | loss    1.947
6193it [00:06, 1093.29it/s]
| epoch   4 |  6000/15247 batches | perplexity    7.032 | loss    1.950
6638it [00:06, 1093.34it/s]
| epoch   4 |  6500/15247 batches | perplexity    6.978 | loss    1.943
7209it [00:07, 1127.85it/s]
| epoch   4 |  7000/15247 batches | perplexity    6.798 | loss    1.917
7672it [00:07, 1142.31it/s]
| epoch   4 |  7500/15247 batches | perplexity    7.008 | loss    1.947
8140it [00:07, 1146.75it/s]
| epoch   4 |  8000/15247 batches | perplexity    6.988 | loss    1.944
8741it [00:08, 1187.13it/s]
| epoch   4 |  8500/15247 batches | perplexity    6.830 | loss    1.921
9216it [00:08, 1166.01it/s]
| epoch   4 |  9000/15247 batches | perplexity    6.866 | loss    1.927
9699it [00:09, 1187.21it/s]
| epoch   4 |  9500/15247 batches | perplexity    6.822 | loss    1.920
10191it [00:09, 1218.40it/s]
| epoch   4 | 10000/15247 batches | perplexity    7.004 | loss    1.946
10675it [00:10, 1165.36it/s]
| epoch   4 | 10500/15247 batches | perplexity    6.773 | loss    1.913
11150it [00:10, 1178.16it/s]
| epoch   4 | 11000/15247 batches | perplexity    6.936 | loss    1.937
11623it [00:10, 1162.19it/s]
| epoch   4 | 11500/15247 batches | perplexity    7.019 | loss    1.949
12223it [00:11, 1173.17it/s]
| epoch   4 | 12000/15247 batches | perplexity    6.883 | loss    1.929
12698it [00:11, 1162.94it/s]
| epoch   4 | 12500/15247 batches | perplexity    6.999 | loss    1.946
```

```
13168it [00:12, 1162.55it/s]

| epoch   4 | 13000/15247 batches | perplexity    7.013 | loss    1.948

13636it [00:12, 1160.65it/s]

| epoch   4 | 13500/15247 batches | perplexity    6.918 | loss    1.934

14233it [00:13, 1169.91it/s]

| epoch   4 | 14000/15247 batches | perplexity    7.012 | loss    1.948

14708it [00:13, 1173.48it/s]

| epoch   4 | 14500/15247 batches | perplexity    6.940 | loss    1.937

15182it [00:13, 1161.72it/s]

| epoch   4 | 15000/15247 batches | perplexity    7.098 | loss    1.960

15247it [00:14, 1087.72it/s]

-----------------------------------------------------------
| end of epoch   4 | time: 14.10s | valid perplexity    6.970 | valid loss
1.942
-----------------------------------------------------------

707it [00:00, 1187.39it/s]

| epoch   5 |   500/15247 batches | perplexity    7.191 | loss    1.973

1175it [00:01, 1138.15it/s]

| epoch   5 |  1000/15247 batches | perplexity    6.992 | loss    1.945

1637it [00:01, 1138.82it/s]

| epoch   5 |  1500/15247 batches | perplexity    6.993 | loss    1.945

2231it [00:01, 1167.64it/s]

| epoch   5 |  2000/15247 batches | perplexity    7.074 | loss    1.956

2705it [00:02, 1175.67it/s]

| epoch   5 |  2500/15247 batches | perplexity    7.073 | loss    1.956

3175it [00:02, 1158.83it/s]

| epoch   5 |  3000/15247 batches | perplexity    6.991 | loss    1.945

3644it [00:03, 1159.47it/s]

| epoch   5 |  3500/15247 batches | perplexity    7.031 | loss    1.950

4238it [00:03, 1179.81it/s]

| epoch   5 |  4000/15247 batches | perplexity    6.854 | loss    1.925

4703it [00:04, 1094.43it/s]

| epoch   5 |  4500/15247 batches | perplexity    6.967 | loss    1.941
```

```
5159it [00:04, 1122.98it/s]
| epoch   5 |  5000/15247 batches | perplexity    7.063 | loss    1.955
5606it [00:04, 1068.61it/s]
| epoch   5 |  5500/15247 batches | perplexity    6.762 | loss    1.911
6181it [00:05, 1122.80it/s]
| epoch   5 |  6000/15247 batches | perplexity    7.006 | loss    1.947
6627it [00:05, 1083.98it/s]
| epoch   5 |  6500/15247 batches | perplexity    6.683 | loss    1.900
7204it [00:06, 1124.46it/s]
| epoch   5 |  7000/15247 batches | perplexity    6.794 | loss    1.916
7656it [00:06, 1110.63it/s]
| epoch   5 |  7500/15247 batches | perplexity    7.109 | loss    1.961
8228it [00:07, 1109.91it/s]
| epoch   5 |  8000/15247 batches | perplexity    7.047 | loss    1.953
8691it [00:07, 1146.16it/s]
| epoch   5 |  8500/15247 batches | perplexity    6.876 | loss    1.928
9157it [00:08, 1140.77it/s]
| epoch   5 |  9000/15247 batches | perplexity    6.899 | loss    1.931
9622it [00:08, 1111.76it/s]
| epoch   5 |  9500/15247 batches | perplexity    6.762 | loss    1.911
10210it [00:09, 1143.14it/s]
| epoch   5 | 10000/15247 batches | perplexity    7.013 | loss    1.948
10672it [00:09, 1131.32it/s]
| epoch   5 | 10500/15247 batches | perplexity    6.910 | loss    1.933
11130it [00:09, 1114.87it/s]
| epoch   5 | 11000/15247 batches | perplexity    6.975 | loss    1.942
11723it [00:10, 1161.06it/s]
| epoch   5 | 11500/15247 batches | perplexity    7.009 | loss    1.947
12189it [00:10, 1138.55it/s]
| epoch   5 | 12000/15247 batches | perplexity    6.963 | loss    1.941
12658it [00:11, 1139.37it/s]
| epoch   5 | 12500/15247 batches | perplexity    7.024 | loss    1.949
```

```
13126it [00:11, 1151.95it/s]

| epoch   5 | 13000/15247 batches | perplexity    6.981 | loss    1.943

13724it [00:12, 1161.02it/s]

| epoch   5 | 13500/15247 batches | perplexity    6.939 | loss    1.937

14186it [00:12, 1127.77it/s]

| epoch   5 | 14000/15247 batches | perplexity    6.975 | loss    1.942

14627it [00:13, 1066.08it/s]

| epoch   5 | 14500/15247 batches | perplexity    6.990 | loss    1.944

15182it [00:13, 1084.14it/s]

| epoch   5 | 15000/15247 batches | perplexity    7.000 | loss    1.946

15247it [00:13, 1117.38it/s]

-----------------------------------------------------------
| end of epoch   5 | time: 13.73s | valid perplexity    6.965 | valid loss
1.941
-----------------------------------------------------------
Checking the results of test dataset.
test perplexity    7.099 | test loss    1.960
```

Hint: For the above, you should see your loss around 2.0 and going down. Similarly to perplexity which should be aroud 7 to 8.

[ ]:
```

```

0.2 Generate some text.

[ ]:
```
def generate_word(model, dataset, window):
    generated_word = []
    # Set the context to a window-1 length array having just the MARKER␣
↪character's token_id.
    context = (window - 1)*[dataset.stoi[MARKER]]

    while True:
        logits = model(torch.tensor(context).view(1, -1))

        # Get the probabilities from the logits.
        # Hint: softmax!
        probs = nn.Softmax(dim=1)(logits)

        # Get 1 sample from a multinomial having the above probabilities.
        token_id = torch.multinomial(probs,1).item()

        # Append the token_id to the generated word.
```

```
        generated_word.append(token_id)

        # Move the context over 1, drop the first (oldest) token and apped the␣
↪new one above.
        # The size of the resulting context should be the same.
        # For exaple, if it was "[0, 1, 2]" and you generated 4, it should now␣
↪be [1, 2, 4].
        context = context[1:] + [token_id]

        if token_id == 0:
            # If you generate token_id = 0, i.e. '.', break out.
            break
    # Return and decode the generated word to a string.
    return ''.join(dataset.decode(generated_word))
```

[ ]:
```
torch.manual_seed(1)
for _ in range(50):
    print(generate_word(model, train_dataset, n))
```

```
ama.
ele.
lia.
aldi.
jarorsse.
dez.
bria.
jairestlei.
revy.
madlais.
hoanna.
dacelian.
alalie.
shais.
maya.
jouston.
zafi.
tye.
karie.
gros.
auhl.
bamaka.
alyaariu.
dera.
ejhar.
jami.
naekshreem.
kaylen.
```

```
quyla.
naygusen.
mayanatram.
ahazorie.
sunya.
shamonti.
hori.
ecfiah.
rosierouston.
ynalah.
cirk.
jasia.
dar.
wun.
jayana.
ris.
nor.
ilyn.
marri.
alavante.
kaly.
marca.
```

[ ]:

```

```