HW 1 - MLP and Character Language Modeling-4

January 29, 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 0x10c139530>
```

[ ]:
```
# 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))
```

```
# 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(l.weight, e.weight).all())
```

[ ]:
```
# 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
```

```
        # 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 = {}

        for i in range(len(chars)):
            dic_stoi[chars[i]] = i
            dic_itos[i] = chars[i]

        self.stoi = dic_stoi
        self.itos = dic_itos

    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 word in self.words

    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.
        return [self.stoi[char] for char in word]

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

    def __getitem__(self, idx):
        # This is used so we can loop over the data.
        word = self.words[idx]
        return self.encode(word)
```

[ ]:

```

```

[ ]:
```python
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 = [i for i in words if i] # Filter out all the empty words.

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

    # 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  = [MARKER * (window - 1) + word + MARKER * (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[0]
```

[ ]: [0, 0, 0, 14, 9, 25, 1, 13, 0, 0, 0]

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

[ ]: 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():
    number_parameters += 1
    print(name, param.shape, param.requires_grad)
print(f"Number of parameters: {number_parameters}")
```

```
H torch.Size([60, 20]) True
W torch.Size([60, 27]) True
U torch.Size([20, 27]) True
b torch.Size([27]) True
d torch.Size([20]) True
C.weight torch.Size([27, 20]) True
Number of parameters: 6
```

[ ]:
```

```

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)
```

```python
    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))
```

```
1000it [00:00, 2518.92it/s]

| epoch   1 |   500/15247 batches | perplexity    9.705 | loss    2.273
| epoch   1 |  1000/15247 batches | perplexity    8.738 | loss    2.168

1779it [00:00, 2581.16it/s]

| epoch   1 |  1500/15247 batches | perplexity    8.605 | loss    2.152
| epoch   1 |  2000/15247 batches | perplexity    8.323 | loss    2.119

2823it [00:01, 2396.74it/s]

| epoch   1 |  2500/15247 batches | perplexity    8.510 | loss    2.141

3323it [00:01, 2446.16it/s]

| epoch   1 |  3000/15247 batches | perplexity    8.326 | loss    2.119
| epoch   1 |  3500/15247 batches | perplexity    8.231 | loss    2.108

4314it [00:01, 2449.23it/s]

| epoch   1 |  4000/15247 batches | perplexity    8.015 | loss    2.081
| epoch   1 |  4500/15247 batches | perplexity    8.167 | loss    2.100

5351it [00:02, 2558.43it/s]

| epoch   1 |  5000/15247 batches | perplexity    8.002 | loss    2.080
| epoch   1 |  5500/15247 batches | perplexity    8.121 | loss    2.094

6404it [00:02, 2622.52it/s]

| epoch   1 |  6000/15247 batches | perplexity    8.067 | loss    2.088
| epoch   1 |  6500/15247 batches | perplexity    8.083 | loss    2.090
```

```
7463it [00:02, 2632.40it/s]

| epoch   1 |  7000/15247 batches | perplexity    7.990 | loss    2.078
| epoch   1 |  7500/15247 batches | perplexity    7.733 | loss    2.045

8529it [00:03, 2661.26it/s]

| epoch   1 |  8000/15247 batches | perplexity    8.021 | loss    2.082
| epoch   1 |  8500/15247 batches | perplexity    8.104 | loss    2.092

9329it [00:03, 2656.24it/s]

| epoch   1 |  9000/15247 batches | perplexity    7.951 | loss    2.073
| epoch   1 |  9500/15247 batches | perplexity    7.866 | loss    2.063

10388it [00:04, 2531.48it/s]

| epoch   1 | 10000/15247 batches | perplexity    7.756 | loss    2.049

10893it [00:04, 2451.57it/s]

| epoch   1 | 10500/15247 batches | perplexity    8.197 | loss    2.104

11413it [00:04, 2523.82it/s]

| epoch   1 | 11000/15247 batches | perplexity    8.008 | loss    2.080
| epoch   1 | 11500/15247 batches | perplexity    7.859 | loss    2.062

12450it [00:04, 2548.52it/s]

| epoch   1 | 12000/15247 batches | perplexity    7.530 | loss    2.019
| epoch   1 | 12500/15247 batches | perplexity    7.986 | loss    2.078

13503it [00:05, 2589.27it/s]

| epoch   1 | 13000/15247 batches | perplexity    8.037 | loss    2.084
| epoch   1 | 13500/15247 batches | perplexity    7.898 | loss    2.067

14286it [00:05, 2597.76it/s]

| epoch   1 | 14000/15247 batches | perplexity    7.566 | loss    2.024
| epoch   1 | 14500/15247 batches | perplexity    7.897 | loss    2.066

15247it [00:05, 2555.37it/s]

| epoch   1 | 15000/15247 batches | perplexity    7.815 | loss    2.056
-----------------------------------------------------------
| end of epoch   1 | time:  6.00s | valid perplexity    7.777 | valid loss
2.051
-----------------------------------------------------------

993it [00:00, 2526.53it/s]

| epoch   2 |   500/15247 batches | perplexity    7.514 | loss    2.017
| epoch   2 |  1000/15247 batches | perplexity    7.111 | loss    1.962

1789it [00:00, 2626.53it/s]
```

```
| epoch   2 |  1500/15247 batches | perplexity    7.164 | loss    1.969
| epoch   2 |  2000/15247 batches | perplexity    7.318 | loss    1.990

2855it [00:01, 2648.12it/s]

| epoch   2 |  2500/15247 batches | perplexity    7.256 | loss    1.982
| epoch   2 |  3000/15247 batches | perplexity    7.209 | loss    1.975

3923it [00:01, 2656.03it/s]

| epoch   2 |  3500/15247 batches | perplexity    7.180 | loss    1.971
| epoch   2 |  4000/15247 batches | perplexity    7.157 | loss    1.968

4993it [00:01, 2670.40it/s]

| epoch   2 |  4500/15247 batches | perplexity    7.158 | loss    1.968
| epoch   2 |  5000/15247 batches | perplexity    7.254 | loss    1.982

5794it [00:02, 2657.48it/s]

| epoch   2 |  5500/15247 batches | perplexity    7.012 | loss    1.948
| epoch   2 |  6000/15247 batches | perplexity    6.993 | loss    1.945

6854it [00:02, 2576.61it/s]

| epoch   2 |  6500/15247 batches | perplexity    7.224 | loss    1.977

7377it [00:02, 2526.97it/s]

| epoch   2 |  7000/15247 batches | perplexity    7.133 | loss    1.965
| epoch   2 |  7500/15247 batches | perplexity    7.176 | loss    1.971

8414it [00:03, 2539.61it/s]

| epoch   2 |  8000/15247 batches | perplexity    7.039 | loss    1.951
| epoch   2 |  8500/15247 batches | perplexity    7.034 | loss    1.951

9445it [00:03, 2536.19it/s]

| epoch   2 |  9000/15247 batches | perplexity    7.378 | loss    1.998
| epoch   2 |  9500/15247 batches | perplexity    6.957 | loss    1.940

10495it [00:04, 2605.65it/s]

| epoch   2 | 10000/15247 batches | perplexity    7.155 | loss    1.968
| epoch   2 | 10500/15247 batches | perplexity    7.195 | loss    1.973

11288it [00:04, 2617.55it/s]

| epoch   2 | 11000/15247 batches | perplexity    7.045 | loss    1.952
| epoch   2 | 11500/15247 batches | perplexity    7.052 | loss    1.953

12364it [00:04, 2673.55it/s]

| epoch   2 | 12000/15247 batches | perplexity    6.973 | loss    1.942
| epoch   2 | 12500/15247 batches | perplexity    7.191 | loss    1.973

13438it [00:05, 2676.16it/s]
```

```
| epoch   2 | 13000/15247 batches | perplexity    7.003 | loss    1.946
| epoch   2 | 13500/15247 batches | perplexity    7.090 | loss    1.959

14493it [00:05, 2583.35it/s]

| epoch   2 | 14000/15247 batches | perplexity    7.184 | loss    1.972
| epoch   2 | 14500/15247 batches | perplexity    7.057 | loss    1.954

15247it [00:05, 2601.10it/s]

| epoch   2 | 15000/15247 batches | perplexity    7.010 | loss    1.947
-----------------------------------------------------------
| end of epoch   2 | time:  5.89s | valid perplexity    7.023 | valid loss
1.949
-----------------------------------------------------------

1006it [00:00, 2543.75it/s]

| epoch   3 |   500/15247 batches | perplexity    6.910 | loss    1.933
| epoch   3 |  1000/15247 batches | perplexity    6.954 | loss    1.939

1800it [00:00, 2621.90it/s]

| epoch   3 |  1500/15247 batches | perplexity    7.033 | loss    1.951
| epoch   3 |  2000/15247 batches | perplexity    7.119 | loss    1.963

2877it [00:01, 2674.21it/s]

| epoch   3 |  2500/15247 batches | perplexity    7.005 | loss    1.947
| epoch   3 |  3000/15247 batches | perplexity    6.869 | loss    1.927

3921it [00:01, 2500.80it/s]

| epoch   3 |  3500/15247 batches | perplexity    7.055 | loss    1.954

4444it [00:01, 2557.78it/s]

| epoch   3 |  4000/15247 batches | perplexity    6.900 | loss    1.932
| epoch   3 |  4500/15247 batches | perplexity    6.883 | loss    1.929

5481it [00:02, 2573.08it/s]

| epoch   3 |  5000/15247 batches | perplexity    7.012 | loss    1.948
| epoch   3 |  5500/15247 batches | perplexity    7.241 | loss    1.980

6253it [00:02, 2487.04it/s]

| epoch   3 |  6000/15247 batches | perplexity    6.970 | loss    1.942

6780it [00:02, 2560.82it/s]

| epoch   3 |  6500/15247 batches | perplexity    7.060 | loss    1.954
| epoch   3 |  7000/15247 batches | perplexity    7.143 | loss    1.966

7841it [00:03, 2608.52it/s]

| epoch   3 |  7500/15247 batches | perplexity    6.882 | loss    1.929
| epoch   3 |  8000/15247 batches | perplexity    6.963 | loss    1.941
```

```
8871it [00:03, 2536.84it/s]

| epoch   3 |  8500/15247 batches | perplexity    6.906 | loss    1.932
| epoch   3 |  9000/15247 batches | perplexity    7.049 | loss    1.953

9887it [00:03, 2490.24it/s]

| epoch   3 |  9500/15247 batches | perplexity    7.069 | loss    1.956

10400it [00:04, 2528.15it/s]

| epoch   3 | 10000/15247 batches | perplexity    7.055 | loss    1.954
| epoch   3 | 10500/15247 batches | perplexity    6.960 | loss    1.940

11385it [00:04, 2381.10it/s]

| epoch   3 | 11000/15247 batches | perplexity    6.884 | loss    1.929

11869it [00:04, 2377.73it/s]

| epoch   3 | 11500/15247 batches | perplexity    6.933 | loss    1.936

12376it [00:04, 2457.37it/s]

| epoch   3 | 12000/15247 batches | perplexity    7.222 | loss    1.977
| epoch   3 | 12500/15247 batches | perplexity    6.977 | loss    1.943

13403it [00:05, 2484.25it/s]

| epoch   3 | 13000/15247 batches | perplexity    7.151 | loss    1.967

13923it [00:05, 2539.05it/s]

| epoch   3 | 13500/15247 batches | perplexity    7.082 | loss    1.958
| epoch   3 | 14000/15247 batches | perplexity    7.020 | loss    1.949

14962it [00:05, 2579.92it/s]

| epoch   3 | 14500/15247 batches | perplexity    7.114 | loss    1.962
| epoch   3 | 15000/15247 batches | perplexity    7.068 | loss    1.956

15247it [00:06, 2527.36it/s]

-----------------------------------------------------------
| end of epoch   3 | time:  6.06s | valid perplexity    6.994 | valid loss
1.945
-----------------------------------------------------------

902it [00:00, 2302.33it/s]

| epoch   4 |   500/15247 batches | perplexity    6.875 | loss    1.928

1384it [00:00, 2293.75it/s]

| epoch   4 |  1000/15247 batches | perplexity    7.027 | loss    1.950

1848it [00:00, 2274.47it/s]

| epoch   4 |  1500/15247 batches | perplexity    7.130 | loss    1.964

2328it [00:01, 2346.79it/s]
```

```
| epoch   4 |  2000/15247 batches | perplexity    7.178 | loss    1.971
2801it [00:01, 2335.84it/s]
| epoch   4 |  2500/15247 batches | perplexity    7.017 | loss    1.948
3284it [00:01, 2370.53it/s]
| epoch   4 |  3000/15247 batches | perplexity    6.986 | loss    1.944
3763it [00:01, 2374.53it/s]
| epoch   4 |  3500/15247 batches | perplexity    7.015 | loss    1.948
4260it [00:01, 2428.70it/s]
| epoch   4 |  4000/15247 batches | perplexity    7.055 | loss    1.954
| epoch   4 |  4500/15247 batches | perplexity    6.887 | loss    1.930
5308it [00:02, 2567.15it/s]
| epoch   4 |  5000/15247 batches | perplexity    6.832 | loss    1.922
| epoch   4 |  5500/15247 batches | perplexity    7.057 | loss    1.954
6332it [00:02, 2498.30it/s]
| epoch   4 |  6000/15247 batches | perplexity    7.094 | loss    1.959
| epoch   4 |  6500/15247 batches | perplexity    7.051 | loss    1.953
7381it [00:03, 2594.62it/s]
| epoch   4 |  7000/15247 batches | perplexity    7.015 | loss    1.948
| epoch   4 |  7500/15247 batches | perplexity    6.942 | loss    1.938
8404it [00:03, 2511.12it/s]
| epoch   4 |  8000/15247 batches | perplexity    6.986 | loss    1.944
| epoch   4 |  8500/15247 batches | perplexity    6.943 | loss    1.938
9414it [00:03, 2403.14it/s]
| epoch   4 |  9000/15247 batches | perplexity    6.885 | loss    1.929
9915it [00:04, 2448.18it/s]
| epoch   4 |  9500/15247 batches | perplexity    6.844 | loss    1.923
10407it [00:04, 2440.40it/s]
| epoch   4 | 10000/15247 batches | perplexity    7.073 | loss    1.956
10916it [00:04, 2498.99it/s]
| epoch   4 | 10500/15247 batches | perplexity    6.989 | loss    1.944
| epoch   4 | 11000/15247 batches | perplexity    7.023 | loss    1.949
11976it [00:04, 2616.68it/s]
| epoch   4 | 11500/15247 batches | perplexity    6.851 | loss    1.924
| epoch   4 | 12000/15247 batches | perplexity    6.905 | loss    1.932
```

```
12769it [00:05, 2631.49it/s]

| epoch   4 | 12500/15247 batches | perplexity    6.861 | loss    1.926
| epoch   4 | 13000/15247 batches | perplexity    6.967 | loss    1.941

13841it [00:05, 2668.72it/s]

| epoch   4 | 13500/15247 batches | perplexity    7.020 | loss    1.949
| epoch   4 | 14000/15247 batches | perplexity    7.028 | loss    1.950

14912it [00:05, 2667.90it/s]

| epoch   4 | 14500/15247 batches | perplexity    6.986 | loss    1.944
| epoch   4 | 15000/15247 batches | perplexity    7.137 | loss    1.965

15247it [00:06, 2488.35it/s]

-----------------------------------------------------------
| end of epoch   4 | time:  6.16s | valid perplexity    6.987 | valid loss
1.944
-----------------------------------------------------------

936it [00:00, 2354.60it/s]

| epoch   5 |   500/15247 batches | perplexity    7.017 | loss    1.948

1425it [00:00, 2401.27it/s]

| epoch   5 |  1000/15247 batches | perplexity    7.090 | loss    1.959

1919it [00:00, 2429.52it/s]

| epoch   5 |  1500/15247 batches | perplexity    7.062 | loss    1.955

2406it [00:01, 2409.92it/s]

| epoch   5 |  2000/15247 batches | perplexity    6.987 | loss    1.944

2920it [00:01, 2496.62it/s]

| epoch   5 |  2500/15247 batches | perplexity    7.005 | loss    1.947
| epoch   5 |  3000/15247 batches | perplexity    7.160 | loss    1.969

3996it [00:01, 2642.34it/s]

| epoch   5 |  3500/15247 batches | perplexity    7.224 | loss    1.977
| epoch   5 |  4000/15247 batches | perplexity    6.921 | loss    1.935

4794it [00:01, 2652.91it/s]

| epoch   5 |  4500/15247 batches | perplexity    7.065 | loss    1.955
| epoch   5 |  5000/15247 batches | perplexity    6.747 | loss    1.909

5839it [00:02, 2482.14it/s]

| epoch   5 |  5500/15247 batches | perplexity    7.056 | loss    1.954

6348it [00:02, 2512.80it/s]
```

```
| epoch   5 |  6000/15247 batches | perplexity    6.975 | loss    1.942
| epoch   5 |  6500/15247 batches | perplexity    6.820 | loss    1.920

7352it [00:02, 2370.39it/s]

| epoch   5 |  7000/15247 batches | perplexity    6.858 | loss    1.925

7851it [00:03, 2428.00it/s]

| epoch   5 |  7500/15247 batches | perplexity    7.035 | loss    1.951
| epoch   5 |  8000/15247 batches | perplexity    6.927 | loss    1.935

8817it [00:03, 2366.59it/s]

| epoch   5 |  8500/15247 batches | perplexity    6.937 | loss    1.937

9295it [00:03, 2371.18it/s]

| epoch   5 |  9000/15247 batches | perplexity    6.960 | loss    1.940

9784it [00:03, 2389.65it/s]

| epoch   5 |  9500/15247 batches | perplexity    7.090 | loss    1.959

10276it [00:04, 2420.88it/s]

| epoch   5 | 10000/15247 batches | perplexity    6.949 | loss    1.939

10770it [00:04, 2442.34it/s]

| epoch   5 | 10500/15247 batches | perplexity    6.912 | loss    1.933
| epoch   5 | 11000/15247 batches | perplexity    6.918 | loss    1.934

11811it [00:04, 2572.55it/s]

| epoch   5 | 11500/15247 batches | perplexity    6.971 | loss    1.942
| epoch   5 | 12000/15247 batches | perplexity    6.879 | loss    1.928

12879it [00:05, 2648.31it/s]

| epoch   5 | 12500/15247 batches | perplexity    6.909 | loss    1.933
| epoch   5 | 13000/15247 batches | perplexity    7.007 | loss    1.947

13948it [00:05, 2639.62it/s]

| epoch   5 | 13500/15247 batches | perplexity    7.042 | loss    1.952
| epoch   5 | 14000/15247 batches | perplexity    7.005 | loss    1.947

14978it [00:06, 2483.20it/s]

| epoch   5 | 14500/15247 batches | perplexity    7.007 | loss    1.947

15247it [00:06, 2486.01it/s]

| epoch   5 | 15000/15247 batches | perplexity    6.977 | loss    1.943
-----------------------------------------------------------
| end of epoch   5 | time:  6.16s | valid perplexity    6.974 | valid loss
1.942
-----------------------------------------------------------
```

```
Checking the results of test dataset.
test perplexity    7.121 | test loss    1.963
```

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.

[ ]:
```python
3 * [train_dataset.stoi[MARKER]]
```

[ ]:
```
[0, 0, 0]
```

0.2 Generate some text.

[ ]:
```python
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))
```

[ ]:
```python
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.
zailah.
ede.
rie.
gros.
aukh.
bamaka.
anyaarius.
kelee.
har.
jami.
naekshreem.
kaylen.
quyla.
naygusen.
mayanatram.
ahazoriexsunya.
shamonti.
hori.
evfiah.
rosie.
vaivel.
ynalaydin.
kenasia.
dar.
wun.
jayana.
ris.
nor.
ilyn.
marri.
alevante.
kalyn.

```
desleeshanaa.
daniellaenimariinilah.
```

[ ]:

```

```

[ ]:

```

```