Character_Level_RNN_Exercise

May 15, 2020

1 Character-Level LSTM in PyTorch

In this notebook, I'll construct a character-level LSTM with PyTorch. The network will train character by character on some text, then generate new text character by character. As an example, I will train on Anna Karenina. **This model will be able to generate new text based on the text from the book!**

This network is based off of Andrej Karpathy's post on RNNs and implementation in Torch. Below is the general architecture of the character-wise RNN.

First let's load in our required resources for data loading and model creation.

```
In [9]: import numpy as np
        import torch
        from torch import nn
        import torch.nn.functional as F
```

1.1 Load in Data

Then, we'll load the Anna Karenina text file and convert it into integers for our network to use.

```
In [10]: # open text file and read in data as `text`
         with open('data/anna.txt', 'r') as f:
             text = f.read()
```

Let's check out the first 100 characters, make sure everything is peachy. According to the American Book Review, this is the 6th best first line of a book ever.

```
In [11]: text[:100]

Out[11]: 'Chapter 1\n\n\nHappy families are all alike; every unhappy family is unhappy in its ow
```

1.1.1 Tokenization

In the cells, below, I'm creating a couple **dictionaries** to convert the characters to and from integers. Encoding the characters as integers makes it easier to use as input in the network.

```
In [12]: # encode the text and map each character to an integer and vice versa

         # we create two dictionaries:
         # 1. int2char, which maps integers to characters
```

```
# 2. char2int, which maps characters to unique integers
chars = tuple(set(text))
int2char = dict(enumerate(chars))
char2int = {ch: ii for ii, ch in int2char.items()}

# encode the text
encoded = np.array([char2int[ch] for ch in text])
```

And we can see those same characters from above, encoded as integers.

In [13]:
```
encoded[:100]
```

Out[13]:
```
array([26, 65,  9, 10, 47, 68,  5, 39, 72, 55, 55, 55, 30,  9, 10, 10, 35,
       39, 42,  9, 11, 66, 31, 66, 68, 53, 39,  9,  5, 68, 39,  9, 31, 31,
       39,  9, 31, 66,  0, 68, 67, 39, 68, 70, 68,  5, 35, 39, 22, 58, 65,
        9, 10, 10, 35, 39, 42,  9, 11, 66, 31, 35, 39, 66, 53, 39, 22, 58,
       65,  9, 10, 10, 35, 39, 66, 58, 39, 66, 47, 53, 39, 32, 27, 58, 55,
       27,  9, 35, 49, 55, 55, 20, 70, 68,  5, 35, 47, 65, 66, 58])
```

1.2 Pre-processing the data

As you can see in our char-RNN image above, our LSTM expects an input that is **one-hot encoded** meaning that each character is converted into an integer (via our created dictionary) and *then* converted into a column vector where only it's corresponding integer index will have the value of 1 and the rest of the vector will be filled with 0's. Since we're one-hot encoding the data, let's make a function to do that!

In [14]:
```
def one_hot_encode(arr, n_labels):

    # Initialize the the encoded array
    one_hot = np.zeros((np.multiply(*arr.shape), n_labels), dtype=np.float32)

    # Fill the appropriate elements with ones
    one_hot[np.arange(one_hot.shape[0]), arr.flatten()] = 1.

    # Finally reshape it to get back to the original array
    one_hot = one_hot.reshape((*arr.shape, n_labels))

    return one_hot
```

In [15]:
```
# check that the function works as expected
test_seq = np.array([[3, 5, 1]])
one_hot = one_hot_encode(test_seq, 8)

print(one_hot)
```

```
[[[ 0.  0.  0.  1.  0.  0.  0.  0.]
  [ 0.  0.  0.  0.  0.  1.  0.  0.]
  [ 0.  1.  0.  0.  0.  0.  0.  0.]]]
```

1.3 Making training mini-batches

To train on this data, we also want to create mini-batches for training. Remember that we want our batches to be multiple sequences of some desired number of sequence steps. Considering a simple example, our batches would look like this:

In this example, we'll take the encoded characters (passed in as the `arr` parameter) and split them into multiple sequences, given by `batch_size`. Each of our sequences will be `seq_length` long.

1.3.1 Creating Batches

1. The first thing we need to do is discard some of the text so we only have completely full mini-batches.

Each batch contains $N \times M$ characters, where N is the batch size (the number of sequences in a batch) and M is the seq_length or number of time steps in a sequence. Then, to get the total number of batches, K, that we can make from the array `arr`, you divide the length of `arr` by the number of characters per batch. Once you know the number of batches, you can get the total number of characters to keep from `arr`, $N * M * K$.

2. After that, we need to split `arr` into N batches.

You can do this using `arr.reshape(size)` where `size` is a tuple containing the dimensions sizes of the reshaped array. We know we want N sequences in a batch, so let's make that the size of the first dimension. For the second dimension, you can use `-1` as a placeholder in the size, it'll fill up the array with the appropriate data for you. After this, you should have an array that is $N \times (M * K)$.

3. Now that we have this array, we can iterate through it to get our mini-batches.

The idea is each batch is a $N \times M$ window on the $N \times (M * K)$ array. For each subsequent batch, the window moves over by `seq_length`. We also want to create both the input and target arrays. Remember that the targets are just the inputs shifted over by one character. The way I like to do this window is use `range` to take steps of size `n_steps` from 0 to `arr.shape[1]`, the total number of tokens in each sequence. That way, the integers you get from `range` always point to the start of a batch, and each window is `seq_length` wide.

> **TODO:** Write the code for creating batches in the function below. The exercises in this notebook *will not be easy*. I've provided a notebook with solutions alongside this notebook. If you get stuck, checkout the solutions. The most important thing is that you don't copy and paste the code into here, **type out the solution code yourself.**

In [16]:
```
def get_batches(arr, batch_size, seq_length):
    '''Create a generator that returns batches of size
       batch_size x seq_length from arr.

       Arguments
       ---------
       arr: Array you want to make batches from
       batch_size: Batch size, the number of sequences per batch
       seq_length: Number of encoded chars in a sequence
    '''

    ## TODO: Get the number of batches we can make
```

```
        batch_size_total = batch_size * seq_length
        # total number of batches we can make
        n_batches = len(arr)//batch_size_total

        # Keep only enough characters to make full batches
        arr = arr[:n_batches * batch_size_total]
        # Reshape into batch_size rows
        arr = arr.reshape((batch_size, -1))

        ## TODO: Iterate over the batches using a window of size seq_length
        for n in range(0, arr.shape[1], seq_length):
            # The features
            x = arr[:, n:n+seq_length]
            # The targets, shifted by one
            y = np.zeros_like(x)
            try:
                y[:, :-1], y[:, -1] = x[:, 1:], arr[:, n+seq_length]
            except IndexError:
                y[:, :-1], y[:, -1] = x[:, 1:], arr[:, 0]
            yield x, y
```

1.3.2 Test Your Implementation

Now I'll make some data sets and we can check out what's going on as we batch data. Here, as an example, I'm going to use a batch size of 8 and 50 sequence steps.

```
In [17]: batches = get_batches(encoded, 8, 50)
         x, y = next(batches)
```

```
In [18]: # printing out the first 10 items in a sequence
         print('x\n', x[:10, :10])
         print('\ny\n', y[:10, :10])
```

```
x
 [[26 65  9 10 47 68  5 39 72 55]
 [53 32 58 39 47 65  9 47 39  9]
 [68 58 64 39 32  5 39  9 39 42]
 [53 39 47 65 68 39 73 65 66 68]
 [39 53  9 27 39 65 68  5 39 47]
 [73 22 53 53 66 32 58 39  9 58]
 [39 56 58 58  9 39 65  9 64 39]
 [69 12 31 32 58 53  0 35 49 39]]

y
 [[65  9 10 47 68  5 39 72 55 55]
 [32 58 39 47 65  9 47 39  9 47]
 [58 64 39 32  5 39  9 39 42 32]
 [39 47 65 68 39 73 65 66 68 42]
 [53  9 27 39 65 68  5 39 47 68]
```

```
 [22 53 53 66 32 58 39  9 58 64]
 [56 58 58  9 39 65  9 64 39 53]
 [12 31 32 58 53  0 35 49 39 50]]
```

If you implemented `get_batches` correctly, the above output should look something like "‘ x [[25 8 60 11 45 27 28 73 1 2][17 7 20 73 45 8 60 45 73 60] [27 20 80 73 7 28 73 60 73 65][17 73 45 8 27 73 66 8 46 27] [73 17 60 12 73 8 27 28 73 45][66 64 17 17 46 7 20 73 60 20] [73 76 20 20 60 73 8 60 80 73][47 35 43 7 20 17 24 50 37 73]]

y [[8 60 11 45 27 28 73 1 2 2][7 20 73 45 8 60 45 73 60 45] [20 80 73 7 28 73 60 73 65 7][73 45 8 27 73 66 8 46 27 65] [17 60 12 73 8 27 28 73 45 27][64 17 17 46 7 20 73 60 20 80] [76 20 20 60 73 8 60 80 73 17][35 43 7 20 17 24 50 37 73 36]] "`although the exact numbers may be different. Check to make sure the data is shifted over one step fory`'.

1.4 Defining the network with PyTorch

Below is where you'll define the network.

Next, you'll use PyTorch to define the architecture of the network. We start by defining the layers and operations we want. Then, define a method for the forward pass. You've also been given a method for predicting characters.

1.4.1 Model Structure

In `__init__` the suggested structure is as follows: * Create and store the necessary dictionaries (this has been done for you) * Define an LSTM layer that takes as params: an input size (the number of characters), a hidden layer size `n_hidden`, a number of layers `n_layers`, a dropout probability `drop_prob`, and a batch_first boolean (True, since we are batching) * Define a dropout layer with `dropout_prob` * Define a fully-connected layer with params: input size `n_hidden` and output size (the number of characters) * Finally, initialize the weights (again, this has been given)

Note that some parameters have been named and given in the `__init__` function, and we use them and store them by doing something like `self.drop_prob = drop_prob`.

1.4.2 LSTM Inputs/Outputs

You can create a basic LSTM layer as follows

```
self.lstm = nn.LSTM(input_size, n_hidden, n_layers,
                            dropout=drop_prob, batch_first=True)
```

where `input_size` is the number of characters this cell expects to see as sequential input, and `n_hidden` is the number of units in the hidden layers in the cell. And we can add dropout by adding a dropout parameter with a specified probability; this will automatically add dropout to the inputs or outputs. Finally, in the `forward` function, we can stack up the LSTM cells into layers using `.view`. With this, you pass in a list of cells and it will send the output of one cell into the next cell.

We also need to create an initial hidden state of all zeros. This is done like so

```
self.init_hidden()
```

In [19]:
```
# check if GPU is available
train_on_gpu = torch.cuda.is_available()
if(train_on_gpu):
    print('Training on GPU!')
else:
    print('No GPU available, training on CPU; consider making n_epochs very small.')
```

```
Training on GPU!
```

In [20]:
```
class CharRNN(nn.Module):

    def __init__(self, tokens, n_hidden=256, n_layers=2,
                               drop_prob=0.5, lr=0.001):
        super().__init__()
        self.drop_prob = drop_prob
        self.n_layers = n_layers
        self.n_hidden = n_hidden
        self.lr = lr

        # creating character dictionaries
        self.chars = tokens
        self.int2char = dict(enumerate(self.chars))
        self.char2int = {ch: ii for ii, ch in self.int2char.items()}

        ## TODO: define the layers of the model
        self.lstm = nn.LSTM(len(self.chars), n_hidden, n_layers,
                            dropout=drop_prob, batch_first=True)
        self.dropout = nn.dropout = nn.Dropout(drop_prob)

        ## TODO: define the final, fully-connected output layer
        self.fc = nn.Linear(n_hidden, len(self.chars))

    def forward(self, x, hidden):
        ''' Forward pass through the network.
            These inputs are x, and the hidden/cell state `hidden`. '''

        ## TODO: Get the outputs and the new hidden state from the lstm
        r_output, hidden = self.lstm(x, hidden)

        ## TODO: pass through a dropout layer
        out = self.dropout(r_output)

        # Stack up LSTM outputs using view
        # you may need to use contiguous to reshape the output
        out = out.contiguous().view(-1, self.n_hidden)
```

```
        ## TODO: put x through the fully-connected layer
        out = self.fc(out)

        # return the final output and the hidden state
        # return the final output and the hidden state
        return out, hidden


    def init_hidden(self, batch_size):
        ''' Initializes hidden state '''
        # Create two new tensors with sizes n_layers x batch_size x n_hidden,
        # initialized to zero, for hidden state and cell state of LSTM
        weight = next(self.parameters()).data

        if (train_on_gpu):
            hidden = (weight.new(self.n_layers, batch_size, self.n_hidden).zero_().cuda
                  weight.new(self.n_layers, batch_size, self.n_hidden).zero_().cuda())
        else:
            hidden = (weight.new(self.n_layers, batch_size, self.n_hidden).zero_(),
                      weight.new(self.n_layers, batch_size, self.n_hidden).zero_())

        return hidden
```

1.5 Time to train

The train function gives us the ability to set the number of epochs, the learning rate, and other parameters.

Below we're using an Adam optimizer and cross entropy loss since we are looking at character class scores as output. We calculate the loss and perform backpropagation, as usual!

A couple of details about training: >* Within the batch loop, we detach the hidden state from its history; this time setting it equal to a new *tuple* variable because an LSTM has a hidden state that is a tuple of the hidden and cell states. * We use `clip_grad_norm_` to help prevent exploding gradients.

```
In [21]: def train(net, data, epochs=10, batch_size=10, seq_length=50, lr=0.001, clip=5, val_fra
             ''' Training a network

                 Arguments
                 ---------

                 net: CharRNN network
                 data: text data to train the network
                 epochs: Number of epochs to train
                 batch_size: Number of mini-sequences per mini-batch, aka batch size
                 seq_length: Number of character steps per mini-batch
                 lr: learning rate
```

```python
    clip: gradient clipping
    val_frac: Fraction of data to hold out for validation
    print_every: Number of steps for printing training and validation loss

'''
net.train()

opt = torch.optim.Adam(net.parameters(), lr=lr)
criterion = nn.CrossEntropyLoss()

# create training and validation data
val_idx = int(len(data)*(1-val_frac))
data, val_data = data[:val_idx], data[val_idx:]

if(train_on_gpu):
    net.cuda()

counter = 0
n_chars = len(net.chars)
for e in range(epochs):
    # initialize hidden state
    h = net.init_hidden(batch_size)

    for x, y in get_batches(data, batch_size, seq_length):
        counter += 1

        # One-hot encode our data and make them Torch tensors
        x = one_hot_encode(x, n_chars)
        inputs, targets = torch.from_numpy(x), torch.from_numpy(y)

        if(train_on_gpu):
            inputs, targets = inputs.cuda(), targets.cuda()

        # Creating new variables for the hidden state, otherwise
        # we'd backprop through the entire training history
        h = tuple([each.data for each in h])

        # zero accumulated gradients
        net.zero_grad()

        # get the output from the model
        output, h = net(inputs, h)

        # calculate the loss and perform backprop
        loss = criterion(output, targets.view(batch_size*seq_length))
        loss.backward()
        # `clip_grad_norm` helps prevent the exploding gradient problem in RNNs / L
        nn.utils.clip_grad_norm_(net.parameters(), clip)
```

```
                    opt.step()

                    # loss stats
                    if counter % print_every == 0:
                        # Get validation loss
                        val_h = net.init_hidden(batch_size)
                        val_losses = []
                        net.eval()
                        for x, y in get_batches(val_data, batch_size, seq_length):
                            # One-hot encode our data and make them Torch tensors
                            x = one_hot_encode(x, n_chars)
                            x, y = torch.from_numpy(x), torch.from_numpy(y)

                            # Creating new variables for the hidden state, otherwise
                            # we'd backprop through the entire training history
                            val_h = tuple([each.data for each in val_h])

                            inputs, targets = x, y
                            if(train_on_gpu):
                                inputs, targets = inputs.cuda(), targets.cuda()

                            output, val_h = net(inputs, val_h)
                            val_loss = criterion(output, targets.view(batch_size*seq_length))

                            val_losses.append(val_loss.item())

                        net.train() # reset to train mode after iterationg through validation d

                        print("Epoch: {}/{}...".format(e+1, epochs),
                              "Step: {}...".format(counter),
                              "Loss: {:.4f}...".format(loss.item()),
                              "Val Loss: {:.4f}".format(np.mean(val_losses)))
```

1.6 Instantiating the model

Now we can actually train the network. First we'll create the network itself, with some given hyperparameters. Then, define the mini-batches sizes, and start training!

```
In [24]: ## TODO: set you model hyperparameters
         # define and print the net
         n_hidden= 512
         n_layers= 2

         net = CharRNN(chars, n_hidden, n_layers)
         print(net)
```

```
CharRNN(
  (lstm): LSTM(83, 512, num_layers=2, batch_first=True, dropout=0.5)
```

```
  (dropout): Dropout(p=0.5)
  (fc): Linear(in_features=512, out_features=83, bias=True)
)
```

1.6.1 Set your training hyperparameters!

In [26]:
```python
batch_size = 128
seq_length = 100
n_epochs = 20 # start small if you are just testing initial behavior

# train the model
train(net, encoded, epochs=n_epochs, batch_size=batch_size, seq_length=seq_length, lr=0
```

```
Epoch: 1/20... Step: 10... Loss: 3.2522... Val Loss: 3.1988
Epoch: 1/20... Step: 20... Loss: 3.1508... Val Loss: 3.1361
Epoch: 1/20... Step: 30... Loss: 3.1385... Val Loss: 3.1237
Epoch: 1/20... Step: 40... Loss: 3.1096... Val Loss: 3.1188
Epoch: 1/20... Step: 50... Loss: 3.1429... Val Loss: 3.1164
Epoch: 1/20... Step: 60... Loss: 3.1152... Val Loss: 3.1128
Epoch: 1/20... Step: 70... Loss: 3.1037... Val Loss: 3.1073
Epoch: 1/20... Step: 80... Loss: 3.1056... Val Loss: 3.0927
Epoch: 1/20... Step: 90... Loss: 3.0856... Val Loss: 3.0627
Epoch: 1/20... Step: 100... Loss: 3.0129... Val Loss: 3.0004
Epoch: 1/20... Step: 110... Loss: 2.9536... Val Loss: 2.9232
Epoch: 1/20... Step: 120... Loss: 2.8469... Val Loss: 2.8184
Epoch: 1/20... Step: 130... Loss: 2.7618... Val Loss: 2.7188
Epoch: 2/20... Step: 140... Loss: 2.6744... Val Loss: 2.6147
Epoch: 2/20... Step: 150... Loss: 2.5986... Val Loss: 2.5390
Epoch: 2/20... Step: 160... Loss: 2.5237... Val Loss: 2.4842
Epoch: 2/20... Step: 170... Loss: 2.4572... Val Loss: 2.4421
Epoch: 2/20... Step: 180... Loss: 2.4335... Val Loss: 2.4115
Epoch: 2/20... Step: 190... Loss: 2.3801... Val Loss: 2.3840
Epoch: 2/20... Step: 200... Loss: 2.3820... Val Loss: 2.3502
Epoch: 2/20... Step: 210... Loss: 2.3461... Val Loss: 2.3246
Epoch: 2/20... Step: 220... Loss: 2.3090... Val Loss: 2.2977
Epoch: 2/20... Step: 230... Loss: 2.3087... Val Loss: 2.2684
Epoch: 2/20... Step: 240... Loss: 2.2781... Val Loss: 2.2419
Epoch: 2/20... Step: 250... Loss: 2.2075... Val Loss: 2.2138
Epoch: 2/20... Step: 260... Loss: 2.1822... Val Loss: 2.1889
Epoch: 2/20... Step: 270... Loss: 2.2052... Val Loss: 2.1652
Epoch: 3/20... Step: 280... Loss: 2.1913... Val Loss: 2.1495
Epoch: 3/20... Step: 290... Loss: 2.1486... Val Loss: 2.1239
Epoch: 3/20... Step: 300... Loss: 2.1233... Val Loss: 2.1008
Epoch: 3/20... Step: 310... Loss: 2.1052... Val Loss: 2.0816
Epoch: 3/20... Step: 320... Loss: 2.0794... Val Loss: 2.0625
Epoch: 3/20... Step: 330... Loss: 2.0409... Val Loss: 2.0468
Epoch: 3/20... Step: 340... Loss: 2.0721... Val Loss: 2.0280
```

```
Epoch: 3/20... Step: 350... Loss: 2.0434... Val Loss: 2.0115
Epoch: 3/20... Step: 360... Loss: 1.9774... Val Loss: 1.9923
Epoch: 3/20... Step: 370... Loss: 2.0063... Val Loss: 1.9770
Epoch: 3/20... Step: 380... Loss: 1.9851... Val Loss: 1.9636
Epoch: 3/20... Step: 390... Loss: 1.9554... Val Loss: 1.9478
Epoch: 3/20... Step: 400... Loss: 1.9162... Val Loss: 1.9297
Epoch: 3/20... Step: 410... Loss: 1.9438... Val Loss: 1.9146
Epoch: 4/20... Step: 420... Loss: 1.9319... Val Loss: 1.8999
Epoch: 4/20... Step: 430... Loss: 1.9168... Val Loss: 1.8827
Epoch: 4/20... Step: 440... Loss: 1.8993... Val Loss: 1.8764
Epoch: 4/20... Step: 450... Loss: 1.8465... Val Loss: 1.8616
Epoch: 4/20... Step: 460... Loss: 1.8314... Val Loss: 1.8504
Epoch: 4/20... Step: 470... Loss: 1.8680... Val Loss: 1.8375
Epoch: 4/20... Step: 480... Loss: 1.8426... Val Loss: 1.8257
Epoch: 4/20... Step: 490... Loss: 1.8480... Val Loss: 1.8128
Epoch: 4/20... Step: 500... Loss: 1.8502... Val Loss: 1.8080
Epoch: 4/20... Step: 510... Loss: 1.8173... Val Loss: 1.7950
Epoch: 4/20... Step: 520... Loss: 1.8304... Val Loss: 1.7874
Epoch: 4/20... Step: 530... Loss: 1.7867... Val Loss: 1.7756
Epoch: 4/20... Step: 540... Loss: 1.7472... Val Loss: 1.7647
Epoch: 4/20... Step: 550... Loss: 1.7942... Val Loss: 1.7511
Epoch: 5/20... Step: 560... Loss: 1.7632... Val Loss: 1.7466
Epoch: 5/20... Step: 570... Loss: 1.7513... Val Loss: 1.7309
Epoch: 5/20... Step: 580... Loss: 1.7309... Val Loss: 1.7239
Epoch: 5/20... Step: 590... Loss: 1.7316... Val Loss: 1.7137
Epoch: 5/20... Step: 600... Loss: 1.7219... Val Loss: 1.7129
Epoch: 5/20... Step: 610... Loss: 1.7018... Val Loss: 1.7007
Epoch: 5/20... Step: 620... Loss: 1.7054... Val Loss: 1.6934
Epoch: 5/20... Step: 630... Loss: 1.7218... Val Loss: 1.6881
Epoch: 5/20... Step: 640... Loss: 1.6802... Val Loss: 1.6809
Epoch: 5/20... Step: 650... Loss: 1.6816... Val Loss: 1.6696
Epoch: 5/20... Step: 660... Loss: 1.6584... Val Loss: 1.6671
Epoch: 5/20... Step: 670... Loss: 1.6823... Val Loss: 1.6567
Epoch: 5/20... Step: 680... Loss: 1.6795... Val Loss: 1.6537
Epoch: 5/20... Step: 690... Loss: 1.6573... Val Loss: 1.6485
Epoch: 6/20... Step: 700... Loss: 1.6501... Val Loss: 1.6481
Epoch: 6/20... Step: 710... Loss: 1.6389... Val Loss: 1.6321
Epoch: 6/20... Step: 720... Loss: 1.6251... Val Loss: 1.6286
Epoch: 6/20... Step: 730... Loss: 1.6528... Val Loss: 1.6222
Epoch: 6/20... Step: 740... Loss: 1.6135... Val Loss: 1.6186
Epoch: 6/20... Step: 750... Loss: 1.5928... Val Loss: 1.6157
Epoch: 6/20... Step: 760... Loss: 1.6316... Val Loss: 1.6075
Epoch: 6/20... Step: 770... Loss: 1.6082... Val Loss: 1.6043
Epoch: 6/20... Step: 780... Loss: 1.5921... Val Loss: 1.5952
Epoch: 6/20... Step: 790... Loss: 1.5852... Val Loss: 1.5907
Epoch: 6/20... Step: 800... Loss: 1.5961... Val Loss: 1.5875
Epoch: 6/20... Step: 810... Loss: 1.5901... Val Loss: 1.5815
Epoch: 6/20... Step: 820... Loss: 1.5511... Val Loss: 1.5781
```

```
Epoch: 6/20... Step: 830... Loss: 1.5911... Val Loss: 1.5727
Epoch: 7/20... Step: 840... Loss: 1.5576... Val Loss: 1.5709
Epoch: 7/20... Step: 850... Loss: 1.5641... Val Loss: 1.5623
Epoch: 7/20... Step: 860... Loss: 1.5481... Val Loss: 1.5579
Epoch: 7/20... Step: 870... Loss: 1.5580... Val Loss: 1.5521
Epoch: 7/20... Step: 880... Loss: 1.5562... Val Loss: 1.5486
Epoch: 7/20... Step: 890... Loss: 1.5523... Val Loss: 1.5447
Epoch: 7/20... Step: 900... Loss: 1.5352... Val Loss: 1.5442
Epoch: 7/20... Step: 910... Loss: 1.5031... Val Loss: 1.5392
Epoch: 7/20... Step: 920... Loss: 1.5229... Val Loss: 1.5347
Epoch: 7/20... Step: 930... Loss: 1.5180... Val Loss: 1.5314
Epoch: 7/20... Step: 940... Loss: 1.5288... Val Loss: 1.5298
Epoch: 7/20... Step: 950... Loss: 1.5325... Val Loss: 1.5231
Epoch: 7/20... Step: 960... Loss: 1.5330... Val Loss: 1.5172
Epoch: 7/20... Step: 970... Loss: 1.5455... Val Loss: 1.5161
Epoch: 8/20... Step: 980... Loss: 1.5061... Val Loss: 1.5160
Epoch: 8/20... Step: 990... Loss: 1.5090... Val Loss: 1.5094
Epoch: 8/20... Step: 1000... Loss: 1.5036... Val Loss: 1.5050
Epoch: 8/20... Step: 1010... Loss: 1.5457... Val Loss: 1.5019
Epoch: 8/20... Step: 1020... Loss: 1.4956... Val Loss: 1.5000
Epoch: 8/20... Step: 1030... Loss: 1.4934... Val Loss: 1.4942
Epoch: 8/20... Step: 1040... Loss: 1.5029... Val Loss: 1.4967
Epoch: 8/20... Step: 1050... Loss: 1.4795... Val Loss: 1.4919
Epoch: 8/20... Step: 1060... Loss: 1.4825... Val Loss: 1.4848
Epoch: 8/20... Step: 1070... Loss: 1.4863... Val Loss: 1.4873
Epoch: 8/20... Step: 1080... Loss: 1.4870... Val Loss: 1.4811
Epoch: 8/20... Step: 1090... Loss: 1.4696... Val Loss: 1.4801
Epoch: 8/20... Step: 1100... Loss: 1.4580... Val Loss: 1.4782
Epoch: 8/20... Step: 1110... Loss: 1.4699... Val Loss: 1.4715
Epoch: 9/20... Step: 1120... Loss: 1.4784... Val Loss: 1.4717
Epoch: 9/20... Step: 1130... Loss: 1.4728... Val Loss: 1.4701
Epoch: 9/20... Step: 1140... Loss: 1.4786... Val Loss: 1.4640
Epoch: 9/20... Step: 1150... Loss: 1.4826... Val Loss: 1.4629
Epoch: 9/20... Step: 1160... Loss: 1.4484... Val Loss: 1.4603
Epoch: 9/20... Step: 1170... Loss: 1.4532... Val Loss: 1.4588
Epoch: 9/20... Step: 1180... Loss: 1.4551... Val Loss: 1.4585
Epoch: 9/20... Step: 1190... Loss: 1.4827... Val Loss: 1.4567
Epoch: 9/20... Step: 1200... Loss: 1.4282... Val Loss: 1.4544
Epoch: 9/20... Step: 1210... Loss: 1.4341... Val Loss: 1.4494
Epoch: 9/20... Step: 1220... Loss: 1.4433... Val Loss: 1.4485
Epoch: 9/20... Step: 1230... Loss: 1.4140... Val Loss: 1.4446
Epoch: 9/20... Step: 1240... Loss: 1.4335... Val Loss: 1.4396
Epoch: 9/20... Step: 1250... Loss: 1.4350... Val Loss: 1.4393
Epoch: 10/20... Step: 1260... Loss: 1.4476... Val Loss: 1.4385
Epoch: 10/20... Step: 1270... Loss: 1.4319... Val Loss: 1.4372
Epoch: 10/20... Step: 1280... Loss: 1.4371... Val Loss: 1.4335
Epoch: 10/20... Step: 1290... Loss: 1.4334... Val Loss: 1.4296
Epoch: 10/20... Step: 1300... Loss: 1.4257... Val Loss: 1.4303
```

```
Epoch: 10/20... Step: 1310... Loss: 1.4206... Val Loss: 1.4285
Epoch: 10/20... Step: 1320... Loss: 1.4018... Val Loss: 1.4247
Epoch: 10/20... Step: 1330... Loss: 1.4100... Val Loss: 1.4263
Epoch: 10/20... Step: 1340... Loss: 1.3882... Val Loss: 1.4244
Epoch: 10/20... Step: 1350... Loss: 1.3872... Val Loss: 1.4202
Epoch: 10/20... Step: 1360... Loss: 1.3883... Val Loss: 1.4197
Epoch: 10/20... Step: 1370... Loss: 1.3826... Val Loss: 1.4151
Epoch: 10/20... Step: 1380... Loss: 1.4113... Val Loss: 1.4158
Epoch: 10/20... Step: 1390... Loss: 1.4241... Val Loss: 1.4141
Epoch: 11/20... Step: 1400... Loss: 1.4201... Val Loss: 1.4115
Epoch: 11/20... Step: 1410... Loss: 1.4378... Val Loss: 1.4098
Epoch: 11/20... Step: 1420... Loss: 1.4223... Val Loss: 1.4059
Epoch: 11/20... Step: 1430... Loss: 1.3954... Val Loss: 1.4061
Epoch: 11/20... Step: 1440... Loss: 1.4196... Val Loss: 1.4092
Epoch: 11/20... Step: 1450... Loss: 1.3541... Val Loss: 1.4041
Epoch: 11/20... Step: 1460... Loss: 1.3701... Val Loss: 1.4017
Epoch: 11/20... Step: 1470... Loss: 1.3597... Val Loss: 1.4046
Epoch: 11/20... Step: 1480... Loss: 1.3903... Val Loss: 1.3985
Epoch: 11/20... Step: 1490... Loss: 1.3702... Val Loss: 1.3998
Epoch: 11/20... Step: 1500... Loss: 1.3523... Val Loss: 1.4003
Epoch: 11/20... Step: 1510... Loss: 1.3410... Val Loss: 1.3969
Epoch: 11/20... Step: 1520... Loss: 1.3842... Val Loss: 1.3902
Epoch: 12/20... Step: 1530... Loss: 1.4396... Val Loss: 1.3932
Epoch: 12/20... Step: 1540... Loss: 1.3905... Val Loss: 1.3918
Epoch: 12/20... Step: 1550... Loss: 1.3862... Val Loss: 1.3886
Epoch: 12/20... Step: 1560... Loss: 1.4039... Val Loss: 1.3848
Epoch: 12/20... Step: 1570... Loss: 1.3510... Val Loss: 1.3891
Epoch: 12/20... Step: 1580... Loss: 1.3249... Val Loss: 1.3870
Epoch: 12/20... Step: 1590... Loss: 1.3292... Val Loss: 1.3857
Epoch: 12/20... Step: 1600... Loss: 1.3531... Val Loss: 1.3853
Epoch: 12/20... Step: 1610... Loss: 1.3385... Val Loss: 1.3851
Epoch: 12/20... Step: 1620... Loss: 1.3482... Val Loss: 1.3783
Epoch: 12/20... Step: 1630... Loss: 1.3660... Val Loss: 1.3799
Epoch: 12/20... Step: 1640... Loss: 1.3424... Val Loss: 1.3796
Epoch: 12/20... Step: 1650... Loss: 1.3226... Val Loss: 1.3772
Epoch: 12/20... Step: 1660... Loss: 1.3715... Val Loss: 1.3730
Epoch: 13/20... Step: 1670... Loss: 1.3464... Val Loss: 1.3769
Epoch: 13/20... Step: 1680... Loss: 1.3571... Val Loss: 1.3725
Epoch: 13/20... Step: 1690... Loss: 1.3290... Val Loss: 1.3684
Epoch: 13/20... Step: 1700... Loss: 1.3394... Val Loss: 1.3681
Epoch: 13/20... Step: 1710... Loss: 1.3091... Val Loss: 1.3700
Epoch: 13/20... Step: 1720... Loss: 1.3207... Val Loss: 1.3725
Epoch: 13/20... Step: 1730... Loss: 1.3595... Val Loss: 1.3679
Epoch: 13/20... Step: 1740... Loss: 1.3308... Val Loss: 1.3696
Epoch: 13/20... Step: 1750... Loss: 1.2979... Val Loss: 1.3715
Epoch: 13/20... Step: 1760... Loss: 1.3292... Val Loss: 1.3654
Epoch: 13/20... Step: 1770... Loss: 1.3431... Val Loss: 1.3649
Epoch: 13/20... Step: 1780... Loss: 1.3191... Val Loss: 1.3620
```

```
Epoch: 13/20... Step: 1790... Loss: 1.3125... Val Loss: 1.3586
Epoch: 13/20... Step: 1800... Loss: 1.3356... Val Loss: 1.3573
Epoch: 14/20... Step: 1810... Loss: 1.3378... Val Loss: 1.3682
Epoch: 14/20... Step: 1820... Loss: 1.3216... Val Loss: 1.3608
Epoch: 14/20... Step: 1830... Loss: 1.3362... Val Loss: 1.3560
Epoch: 14/20... Step: 1840... Loss: 1.2818... Val Loss: 1.3564
Epoch: 14/20... Step: 1850... Loss: 1.2700... Val Loss: 1.3583
Epoch: 14/20... Step: 1860... Loss: 1.3295... Val Loss: 1.3581
Epoch: 14/20... Step: 1870... Loss: 1.3361... Val Loss: 1.3550
Epoch: 14/20... Step: 1880... Loss: 1.3298... Val Loss: 1.3570
Epoch: 14/20... Step: 1890... Loss: 1.3335... Val Loss: 1.3592
Epoch: 14/20... Step: 1900... Loss: 1.3168... Val Loss: 1.3528
Epoch: 14/20... Step: 1910... Loss: 1.3158... Val Loss: 1.3505
Epoch: 14/20... Step: 1920... Loss: 1.3114... Val Loss: 1.3487
Epoch: 14/20... Step: 1930... Loss: 1.2777... Val Loss: 1.3482
Epoch: 14/20... Step: 1940... Loss: 1.3411... Val Loss: 1.3455
Epoch: 15/20... Step: 1950... Loss: 1.3011... Val Loss: 1.3526
Epoch: 15/20... Step: 1960... Loss: 1.3114... Val Loss: 1.3483
Epoch: 15/20... Step: 1970... Loss: 1.3029... Val Loss: 1.3448
Epoch: 15/20... Step: 1980... Loss: 1.3048... Val Loss: 1.3474
Epoch: 15/20... Step: 1990... Loss: 1.2911... Val Loss: 1.3461
Epoch: 15/20... Step: 2000... Loss: 1.2755... Val Loss: 1.3434
Epoch: 15/20... Step: 2010... Loss: 1.2984... Val Loss: 1.3419
Epoch: 15/20... Step: 2020... Loss: 1.3062... Val Loss: 1.3449
Epoch: 15/20... Step: 2030... Loss: 1.2871... Val Loss: 1.3454
Epoch: 15/20... Step: 2040... Loss: 1.2962... Val Loss: 1.3373
Epoch: 15/20... Step: 2050... Loss: 1.2870... Val Loss: 1.3399
Epoch: 15/20... Step: 2060... Loss: 1.3063... Val Loss: 1.3383
Epoch: 15/20... Step: 2070... Loss: 1.2991... Val Loss: 1.3341
Epoch: 15/20... Step: 2080... Loss: 1.2937... Val Loss: 1.3327
Epoch: 16/20... Step: 2090... Loss: 1.3005... Val Loss: 1.3354
Epoch: 16/20... Step: 2100... Loss: 1.2800... Val Loss: 1.3318
Epoch: 16/20... Step: 2110... Loss: 1.2695... Val Loss: 1.3323
Epoch: 16/20... Step: 2120... Loss: 1.2888... Val Loss: 1.3313
Epoch: 16/20... Step: 2130... Loss: 1.2711... Val Loss: 1.3325
Epoch: 16/20... Step: 2140... Loss: 1.2755... Val Loss: 1.3310
Epoch: 16/20... Step: 2150... Loss: 1.3072... Val Loss: 1.3322
Epoch: 16/20... Step: 2160... Loss: 1.2828... Val Loss: 1.3352
Epoch: 16/20... Step: 2170... Loss: 1.2788... Val Loss: 1.3330
Epoch: 16/20... Step: 2180... Loss: 1.2791... Val Loss: 1.3289
Epoch: 16/20... Step: 2190... Loss: 1.2980... Val Loss: 1.3279
Epoch: 16/20... Step: 2200... Loss: 1.2716... Val Loss: 1.3303
Epoch: 16/20... Step: 2210... Loss: 1.2362... Val Loss: 1.3253
Epoch: 16/20... Step: 2220... Loss: 1.2877... Val Loss: 1.3220
Epoch: 17/20... Step: 2230... Loss: 1.2547... Val Loss: 1.3254
Epoch: 17/20... Step: 2240... Loss: 1.2697... Val Loss: 1.3258
Epoch: 17/20... Step: 2250... Loss: 1.2535... Val Loss: 1.3261
Epoch: 17/20... Step: 2260... Loss: 1.2569... Val Loss: 1.3278
```

```
Epoch: 17/20... Step: 2270... Loss: 1.2713... Val Loss: 1.3239
Epoch: 17/20... Step: 2280... Loss: 1.2852... Val Loss: 1.3212
Epoch: 17/20... Step: 2290... Loss: 1.2796... Val Loss: 1.3215
Epoch: 17/20... Step: 2300... Loss: 1.2470... Val Loss: 1.3250
Epoch: 17/20... Step: 2310... Loss: 1.2703... Val Loss: 1.3214
Epoch: 17/20... Step: 2320... Loss: 1.2638... Val Loss: 1.3193
Epoch: 17/20... Step: 2330... Loss: 1.2474... Val Loss: 1.3173
Epoch: 17/20... Step: 2340... Loss: 1.2733... Val Loss: 1.3177
Epoch: 17/20... Step: 2350... Loss: 1.2734... Val Loss: 1.3151
Epoch: 17/20... Step: 2360... Loss: 1.2726... Val Loss: 1.3150
Epoch: 18/20... Step: 2370... Loss: 1.2498... Val Loss: 1.3163
Epoch: 18/20... Step: 2380... Loss: 1.2560... Val Loss: 1.3183
Epoch: 18/20... Step: 2390... Loss: 1.2576... Val Loss: 1.3183
Epoch: 18/20... Step: 2400... Loss: 1.2845... Val Loss: 1.3179
Epoch: 18/20... Step: 2410... Loss: 1.2796... Val Loss: 1.3152
Epoch: 18/20... Step: 2420... Loss: 1.2615... Val Loss: 1.3152
Epoch: 18/20... Step: 2430... Loss: 1.2640... Val Loss: 1.3158
Epoch: 18/20... Step: 2440... Loss: 1.2495... Val Loss: 1.3169
Epoch: 18/20... Step: 2450... Loss: 1.2379... Val Loss: 1.3135
Epoch: 18/20... Step: 2460... Loss: 1.2698... Val Loss: 1.3128
Epoch: 18/20... Step: 2470... Loss: 1.2439... Val Loss: 1.3067
Epoch: 18/20... Step: 2480... Loss: 1.2540... Val Loss: 1.3106
Epoch: 18/20... Step: 2490... Loss: 1.2418... Val Loss: 1.3057
Epoch: 18/20... Step: 2500... Loss: 1.2449... Val Loss: 1.3069
Epoch: 19/20... Step: 2510... Loss: 1.2530... Val Loss: 1.3089
Epoch: 19/20... Step: 2520... Loss: 1.2552... Val Loss: 1.3126
Epoch: 19/20... Step: 2530... Loss: 1.2663... Val Loss: 1.3100
Epoch: 19/20... Step: 2540... Loss: 1.2717... Val Loss: 1.3105
Epoch: 19/20... Step: 2550... Loss: 1.2419... Val Loss: 1.3091
Epoch: 19/20... Step: 2560... Loss: 1.2529... Val Loss: 1.3057
Epoch: 19/20... Step: 2570... Loss: 1.2497... Val Loss: 1.3048
Epoch: 19/20... Step: 2580... Loss: 1.2710... Val Loss: 1.3086
Epoch: 19/20... Step: 2590... Loss: 1.2252... Val Loss: 1.3055
Epoch: 19/20... Step: 2600... Loss: 1.2245... Val Loss: 1.3028
Epoch: 19/20... Step: 2610... Loss: 1.2436... Val Loss: 1.3053
Epoch: 19/20... Step: 2620... Loss: 1.2266... Val Loss: 1.3076
Epoch: 19/20... Step: 2630... Loss: 1.2293... Val Loss: 1.2981
Epoch: 19/20... Step: 2640... Loss: 1.2467... Val Loss: 1.2956
Epoch: 20/20... Step: 2650... Loss: 1.2556... Val Loss: 1.3012
Epoch: 20/20... Step: 2660... Loss: 1.2496... Val Loss: 1.3075
Epoch: 20/20... Step: 2670... Loss: 1.2435... Val Loss: 1.3042
Epoch: 20/20... Step: 2680... Loss: 1.2509... Val Loss: 1.3043
Epoch: 20/20... Step: 2690... Loss: 1.2397... Val Loss: 1.3036
Epoch: 20/20... Step: 2700... Loss: 1.2476... Val Loss: 1.2999
Epoch: 20/20... Step: 2710... Loss: 1.2180... Val Loss: 1.3019
Epoch: 20/20... Step: 2720... Loss: 1.2103... Val Loss: 1.3062
Epoch: 20/20... Step: 2730... Loss: 1.2150... Val Loss: 1.3017
Epoch: 20/20... Step: 2740... Loss: 1.2109... Val Loss: 1.3009
```