

Character_Level_RNN_Exercise

February 28, 2019

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 [22]: 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 [23]: # 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 [24]: text[:100]
```

```
Out[24]: '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 [25]: # 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)) # set: only different characters, tuple: fixed order
int2char = dict(enumerate(chars)) # dict with keys: numbers, values: strings
char2int = {ch: ii for ii, ch in int2char.items()} # reverse dict

# encode the text
encoded = np.array([char2int[ch] for ch in text]) # encode every char in the text as nu

```

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

```
In [26]: encoded[:100]
```

```
Out[26]: array([69, 15,  6, 29, 40, 52, 75, 28,  7, 68, 68, 68, 14,  6, 29, 29, 32,
                28,  2,  6, 41, 76, 23, 76, 52, 10, 28,  6, 75, 52, 28,  6, 23, 23,
                28,  6, 23, 76, 47, 52, 18, 28, 52, 72, 52, 75, 32, 28, 31, 60, 15,
                 6, 29, 29, 32, 28,  2,  6, 41, 76, 23, 32, 28, 76, 10, 28, 31, 60,
                15,  6, 29, 29, 32, 28, 76, 60, 28, 76, 40, 10, 28, 56, 45, 60, 68,
                45,  6, 32, 42, 68, 68, 49, 72, 52, 75, 32, 40, 15, 76, 60])
```

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 [27]: 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 [28]: # 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 [29]: 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
        '''

        batch_size_total = batch_size * seq_length
```

```

## TODO: Get the number of batches we can make
n_batches = len(arr) // batch_size_total

## TODO: Keep only enough characters to make full batches
arr = arr[0:-(len(arr) % batch_size_total)]
#arr = arr[:n_batches * batch_size_total]

## TODO: Reshape into batch_size rows
arr = arr.reshape((batch_size, -1))
# arr = np.reshape(arr, (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): # start, stop, step
    # The features
    x = arr[:, n:n+seq_length]
    # The targets, shifted by one
    y = np.zeros_like(x)
    y[:, :-1] = x[:, 1:] # except for last value: y[i]=x[i+1] (next char is predicted)
    try:
        y[:, -1] = arr[:, n+seq_length]
    except IndexError:
        y[:, -1] = arr[:, 0] # sequence finished --> x[0] (or s.th. else)
    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 [30]: batches = get_batches(encoded, 8, 50)
         x, y = next(batches)

```

```

In [31]: # printing out the first 10 items in a sequence
         print('x\n', x[:10, :10])
         print('\ny\n', y[:10, :10])

```

```

x
[[69 15  6 29 40 52 75 28  7 68]
 [10 56 60 28 40 15  6 40 28  6]
 [52 60 67 28 56 75 28  6 28  2]
 [10 28 40 15 52 28 61 15 76 52]
 [28 10  6 45 28 15 52 75 28 40]
 [61 31 10 10 76 56 60 28  6 60]
 [28 54 60 60  6 28 15  6 67 28]
 [62 82 23 56 60 10 47 32 42 28]]

```

```

y
[[15  6 29 40 52 75 28  7 68 68]
 [56 60 28 40 15  6 40 28  6 40]

```

```
[60 67 28 56 75 28 6 28 2 56]
[28 40 15 52 28 61 15 76 52 2]
[10 6 45 28 15 52 75 28 40 52]
[31 10 10 76 56 60 28 6 60 67]
[54 60 60 6 28 15 6 67 28 10]
[82 23 56 60 10 47 32 42 28 22]]
```

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 for `y`’.

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 [32]: # 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 [33]: 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(tokens), n_hidden, n_layers,
                             dropout=drop_prob, batch_first=True)
        self.dropout = nn.Dropout(drop_prob)
        self.fc = nn.Linear(n_hidden, len(tokens))

    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_out, r_hidden = self.lstm(x, hidden)
        r_out = self.dropout(r_out)
        r_out = r_out.contiguous().view(-1, self.n_hidden)
        out = self.fc(r_out)
```

```

        # return the final output and the hidden state
        return out, r_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 [34]: def train(net, data, epochs=10, batch_size=10, seq_length=50, lr=0.001, clip=5, val_frac=0.1):
    ''' 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
    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 [35]: ## 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 [36]: 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=0.001)

Epoch: 1/20... Step: 10... Loss: 3.2684... Val Loss: 3.2087
Epoch: 1/20... Step: 20... Loss: 3.1507... Val Loss: 3.1320
Epoch: 1/20... Step: 30... Loss: 3.1439... Val Loss: 3.1232
Epoch: 1/20... Step: 40... Loss: 3.1123... Val Loss: 3.1182
Epoch: 1/20... Step: 50... Loss: 3.1420... Val Loss: 3.1169
Epoch: 1/20... Step: 60... Loss: 3.1141... Val Loss: 3.1139
Epoch: 1/20... Step: 70... Loss: 3.1053... Val Loss: 3.1105
Epoch: 1/20... Step: 80... Loss: 3.1158... Val Loss: 3.1024
Epoch: 1/20... Step: 90... Loss: 3.1002... Val Loss: 3.0823
Epoch: 1/20... Step: 100... Loss: 3.0871... Val Loss: 3.0835
Epoch: 1/20... Step: 110... Loss: 3.0463... Val Loss: 3.0246
Epoch: 1/20... Step: 120... Loss: 2.9344... Val Loss: 2.9166
Epoch: 1/20... Step: 130... Loss: 2.8705... Val Loss: 2.8229
Epoch: 2/20... Step: 140... Loss: 2.7516... Val Loss: 2.6944
Epoch: 2/20... Step: 150... Loss: 2.6542... Val Loss: 2.6089
Epoch: 2/20... Step: 160... Loss: 2.5858... Val Loss: 2.5391
Epoch: 2/20... Step: 170... Loss: 2.5063... Val Loss: 2.4901
Epoch: 2/20... Step: 180... Loss: 2.4772... Val Loss: 2.4515
Epoch: 2/20... Step: 190... Loss: 2.4265... Val Loss: 2.4192
Epoch: 2/20... Step: 200... Loss: 2.4244... Val Loss: 2.3981
Epoch: 2/20... Step: 210... Loss: 2.3893... Val Loss: 2.3628
Epoch: 2/20... Step: 220... Loss: 2.3482... Val Loss: 2.3314
Epoch: 2/20... Step: 230... Loss: 2.3401... Val Loss: 2.3043
Epoch: 2/20... Step: 240... Loss: 2.3131... Val Loss: 2.2750
Epoch: 2/20... Step: 250... Loss: 2.2495... Val Loss: 2.2554
Epoch: 2/20... Step: 260... Loss: 2.2250... Val Loss: 2.2275
Epoch: 2/20... Step: 270... Loss: 2.2375... Val Loss: 2.1993
Epoch: 3/20... Step: 280... Loss: 2.2259... Val Loss: 2.1778
Epoch: 3/20... Step: 290... Loss: 2.1798... Val Loss: 2.1516
Epoch: 3/20... Step: 300... Loss: 2.1552... Val Loss: 2.1291
Epoch: 3/20... Step: 310... Loss: 2.1412... Val Loss: 2.1172
Epoch: 3/20... Step: 320... Loss: 2.1088... Val Loss: 2.0895
Epoch: 3/20... Step: 330... Loss: 2.0719... Val Loss: 2.0713
Epoch: 3/20... Step: 340... Loss: 2.0884... Val Loss: 2.0479
Epoch: 3/20... Step: 350... Loss: 2.0806... Val Loss: 2.0424
Epoch: 3/20... Step: 360... Loss: 2.0079... Val Loss: 2.0193
Epoch: 3/20... Step: 370... Loss: 2.0386... Val Loss: 2.0009
Epoch: 3/20... Step: 380... Loss: 2.0106... Val Loss: 1.9837
Epoch: 3/20... Step: 390... Loss: 1.9874... Val Loss: 1.9676
Epoch: 3/20... Step: 400... Loss: 1.9531... Val Loss: 1.9507
```

Epoch: 3/20... Step: 410... Loss: 1.9645... Val Loss: 1.9390
Epoch: 4/20... Step: 420... Loss: 1.9570... Val Loss: 1.9248
Epoch: 4/20... Step: 430... Loss: 1.9412... Val Loss: 1.9098
Epoch: 4/20... Step: 440... Loss: 1.9247... Val Loss: 1.9026
Epoch: 4/20... Step: 450... Loss: 1.8763... Val Loss: 1.8824
Epoch: 4/20... Step: 460... Loss: 1.8582... Val Loss: 1.8733
Epoch: 4/20... Step: 470... Loss: 1.8922... Val Loss: 1.8626
Epoch: 4/20... Step: 480... Loss: 1.8648... Val Loss: 1.8484
Epoch: 4/20... Step: 490... Loss: 1.8773... Val Loss: 1.8385
Epoch: 4/20... Step: 500... Loss: 1.8731... Val Loss: 1.8280
Epoch: 4/20... Step: 510... Loss: 1.8430... Val Loss: 1.8150
Epoch: 4/20... Step: 520... Loss: 1.8564... Val Loss: 1.8073
Epoch: 4/20... Step: 530... Loss: 1.8212... Val Loss: 1.7952
Epoch: 4/20... Step: 540... Loss: 1.7779... Val Loss: 1.7877
Epoch: 4/20... Step: 550... Loss: 1.8256... Val Loss: 1.7739
Epoch: 5/20... Step: 560... Loss: 1.7871... Val Loss: 1.7680
Epoch: 5/20... Step: 570... Loss: 1.7711... Val Loss: 1.7588
Epoch: 5/20... Step: 580... Loss: 1.7569... Val Loss: 1.7452
Epoch: 5/20... Step: 590... Loss: 1.7576... Val Loss: 1.7357
Epoch: 5/20... Step: 600... Loss: 1.7436... Val Loss: 1.7299
Epoch: 5/20... Step: 610... Loss: 1.7271... Val Loss: 1.7250
Epoch: 5/20... Step: 620... Loss: 1.7313... Val Loss: 1.7157
Epoch: 5/20... Step: 630... Loss: 1.7430... Val Loss: 1.7121
Epoch: 5/20... Step: 640... Loss: 1.7191... Val Loss: 1.7043
Epoch: 5/20... Step: 650... Loss: 1.7016... Val Loss: 1.6931
Epoch: 5/20... Step: 660... Loss: 1.6745... Val Loss: 1.6859
Epoch: 5/20... Step: 670... Loss: 1.7025... Val Loss: 1.6790
Epoch: 5/20... Step: 680... Loss: 1.6974... Val Loss: 1.6704
Epoch: 5/20... Step: 690... Loss: 1.6822... Val Loss: 1.6623
Epoch: 6/20... Step: 700... Loss: 1.6761... Val Loss: 1.6596
Epoch: 6/20... Step: 710... Loss: 1.6662... Val Loss: 1.6492
Epoch: 6/20... Step: 720... Loss: 1.6513... Val Loss: 1.6442
Epoch: 6/20... Step: 730... Loss: 1.6645... Val Loss: 1.6387
Epoch: 6/20... Step: 740... Loss: 1.6253... Val Loss: 1.6329
Epoch: 6/20... Step: 750... Loss: 1.6154... Val Loss: 1.6324
Epoch: 6/20... Step: 760... Loss: 1.6496... Val Loss: 1.6237
Epoch: 6/20... Step: 770... Loss: 1.6304... Val Loss: 1.6198
Epoch: 6/20... Step: 780... Loss: 1.6187... Val Loss: 1.6144
Epoch: 6/20... Step: 790... Loss: 1.6084... Val Loss: 1.6100
Epoch: 6/20... Step: 800... Loss: 1.6152... Val Loss: 1.6060
Epoch: 6/20... Step: 810... Loss: 1.6134... Val Loss: 1.5996
Epoch: 6/20... Step: 820... Loss: 1.5685... Val Loss: 1.5915
Epoch: 6/20... Step: 830... Loss: 1.6135... Val Loss: 1.5842
Epoch: 7/20... Step: 840... Loss: 1.5728... Val Loss: 1.5826
Epoch: 7/20... Step: 850... Loss: 1.5869... Val Loss: 1.5781
Epoch: 7/20... Step: 860... Loss: 1.5721... Val Loss: 1.5712
Epoch: 7/20... Step: 870... Loss: 1.5781... Val Loss: 1.5685
Epoch: 7/20... Step: 880... Loss: 1.5864... Val Loss: 1.5633

Epoch: 7/20... Step: 890... Loss: 1.5723... Val Loss: 1.5644
Epoch: 7/20... Step: 900... Loss: 1.5597... Val Loss: 1.5657
Epoch: 7/20... Step: 910... Loss: 1.5351... Val Loss: 1.5579
Epoch: 7/20... Step: 920... Loss: 1.5609... Val Loss: 1.5536
Epoch: 7/20... Step: 930... Loss: 1.5424... Val Loss: 1.5480
Epoch: 7/20... Step: 940... Loss: 1.5396... Val Loss: 1.5451
Epoch: 7/20... Step: 950... Loss: 1.5545... Val Loss: 1.5430
Epoch: 7/20... Step: 960... Loss: 1.5513... Val Loss: 1.5331
Epoch: 7/20... Step: 970... Loss: 1.5551... Val Loss: 1.5271
Epoch: 8/20... Step: 980... Loss: 1.5359... Val Loss: 1.5248
Epoch: 8/20... Step: 990... Loss: 1.5223... Val Loss: 1.5233
Epoch: 8/20... Step: 1000... Loss: 1.5197... Val Loss: 1.5193
Epoch: 8/20... Step: 1010... Loss: 1.5582... Val Loss: 1.5171
Epoch: 8/20... Step: 1020... Loss: 1.5270... Val Loss: 1.5101
Epoch: 8/20... Step: 1030... Loss: 1.5072... Val Loss: 1.5101
Epoch: 8/20... Step: 1040... Loss: 1.5145... Val Loss: 1.5084
Epoch: 8/20... Step: 1050... Loss: 1.5053... Val Loss: 1.5059
Epoch: 8/20... Step: 1060... Loss: 1.5062... Val Loss: 1.5005
Epoch: 8/20... Step: 1070... Loss: 1.5046... Val Loss: 1.4957
Epoch: 8/20... Step: 1080... Loss: 1.5011... Val Loss: 1.4969
Epoch: 8/20... Step: 1090... Loss: 1.4861... Val Loss: 1.4951
Epoch: 8/20... Step: 1100... Loss: 1.4775... Val Loss: 1.4846
Epoch: 8/20... Step: 1110... Loss: 1.4869... Val Loss: 1.4813
Epoch: 9/20... Step: 1120... Loss: 1.4902... Val Loss: 1.4828
Epoch: 9/20... Step: 1130... Loss: 1.4838... Val Loss: 1.4819
Epoch: 9/20... Step: 1140... Loss: 1.4954... Val Loss: 1.4743
Epoch: 9/20... Step: 1150... Loss: 1.4978... Val Loss: 1.4782
Epoch: 9/20... Step: 1160... Loss: 1.4532... Val Loss: 1.4728
Epoch: 9/20... Step: 1170... Loss: 1.4671... Val Loss: 1.4666
Epoch: 9/20... Step: 1180... Loss: 1.4614... Val Loss: 1.4685
Epoch: 9/20... Step: 1190... Loss: 1.4870... Val Loss: 1.4685
Epoch: 9/20... Step: 1200... Loss: 1.4414... Val Loss: 1.4589
Epoch: 9/20... Step: 1210... Loss: 1.4502... Val Loss: 1.4580
Epoch: 9/20... Step: 1220... Loss: 1.4581... Val Loss: 1.4608
Epoch: 9/20... Step: 1230... Loss: 1.4387... Val Loss: 1.4588
Epoch: 9/20... Step: 1240... Loss: 1.4331... Val Loss: 1.4553
Epoch: 9/20... Step: 1250... Loss: 1.4449... Val Loss: 1.4480
Epoch: 10/20... Step: 1260... Loss: 1.4534... Val Loss: 1.4496
Epoch: 10/20... Step: 1270... Loss: 1.4428... Val Loss: 1.4517
Epoch: 10/20... Step: 1280... Loss: 1.4604... Val Loss: 1.4422
Epoch: 10/20... Step: 1290... Loss: 1.4328... Val Loss: 1.4473
Epoch: 10/20... Step: 1300... Loss: 1.4324... Val Loss: 1.4388
Epoch: 10/20... Step: 1310... Loss: 1.4441... Val Loss: 1.4406
Epoch: 10/20... Step: 1320... Loss: 1.4136... Val Loss: 1.4419
Epoch: 10/20... Step: 1330... Loss: 1.4229... Val Loss: 1.4390
Epoch: 10/20... Step: 1340... Loss: 1.4031... Val Loss: 1.4341
Epoch: 10/20... Step: 1350... Loss: 1.3981... Val Loss: 1.4281
Epoch: 10/20... Step: 1360... Loss: 1.4044... Val Loss: 1.4332

Epoch: 10/20... Step: 1370... Loss: 1.3949... Val Loss: 1.4353
 Epoch: 10/20... Step: 1380... Loss: 1.4440... Val Loss: 1.4246
 Epoch: 10/20... Step: 1390... Loss: 1.4439... Val Loss: 1.4215
 Epoch: 11/20... Step: 1400... Loss: 1.4464... Val Loss: 1.4210
 Epoch: 11/20... Step: 1410... Loss: 1.4523... Val Loss: 1.4225
 Epoch: 11/20... Step: 1420... Loss: 1.4378... Val Loss: 1.4181
 Epoch: 11/20... Step: 1430... Loss: 1.4078... Val Loss: 1.4215
 Epoch: 11/20... Step: 1440... Loss: 1.4411... Val Loss: 1.4144
 Epoch: 11/20... Step: 1450... Loss: 1.3629... Val Loss: 1.4178
 Epoch: 11/20... Step: 1460... Loss: 1.3871... Val Loss: 1.4140
 Epoch: 11/20... Step: 1470... Loss: 1.3857... Val Loss: 1.4116
 Epoch: 11/20... Step: 1480... Loss: 1.3982... Val Loss: 1.4076
 Epoch: 11/20... Step: 1490... Loss: 1.3895... Val Loss: 1.4067
 Epoch: 11/20... Step: 1500... Loss: 1.3656... Val Loss: 1.4130
 Epoch: 11/20... Step: 1510... Loss: 1.3555... Val Loss: 1.4109
 Epoch: 11/20... Step: 1520... Loss: 1.3970... Val Loss: 1.4000
 Epoch: 12/20... Step: 1530... Loss: 1.4469... Val Loss: 1.3955
 Epoch: 12/20... Step: 1540... Loss: 1.4052... Val Loss: 1.3971
 Epoch: 12/20... Step: 1550... Loss: 1.4030... Val Loss: 1.4009
 Epoch: 12/20... Step: 1560... Loss: 1.4153... Val Loss: 1.3944
 Epoch: 12/20... Step: 1570... Loss: 1.3614... Val Loss: 1.3987
 Epoch: 12/20... Step: 1580... Loss: 1.3425... Val Loss: 1.3958
 Epoch: 12/20... Step: 1590... Loss: 1.3393... Val Loss: 1.3960
 Epoch: 12/20... Step: 1600... Loss: 1.3643... Val Loss: 1.3932
 Epoch: 12/20... Step: 1610... Loss: 1.3541... Val Loss: 1.3946
 Epoch: 12/20... Step: 1620... Loss: 1.3552... Val Loss: 1.3901
 Epoch: 12/20... Step: 1630... Loss: 1.3772... Val Loss: 1.3844
 Epoch: 12/20... Step: 1640... Loss: 1.3575... Val Loss: 1.3904
 Epoch: 12/20... Step: 1650... Loss: 1.3203... Val Loss: 1.3908
 Epoch: 12/20... Step: 1660... Loss: 1.3822... Val Loss: 1.3861
 Epoch: 13/20... Step: 1670... Loss: 1.3511... Val Loss: 1.3841
 Epoch: 13/20... Step: 1680... Loss: 1.3646... Val Loss: 1.3790
 Epoch: 13/20... Step: 1690... Loss: 1.3416... Val Loss: 1.3787
 Epoch: 13/20... Step: 1700... Loss: 1.3485... Val Loss: 1.3828
 Epoch: 13/20... Step: 1710... Loss: 1.3227... Val Loss: 1.3844
 Epoch: 13/20... Step: 1720... Loss: 1.3394... Val Loss: 1.3805
 Epoch: 13/20... Step: 1730... Loss: 1.3724... Val Loss: 1.3762
 Epoch: 13/20... Step: 1740... Loss: 1.3372... Val Loss: 1.3760
 Epoch: 13/20... Step: 1750... Loss: 1.3078... Val Loss: 1.3788
 Epoch: 13/20... Step: 1760... Loss: 1.3368... Val Loss: 1.3736
 Epoch: 13/20... Step: 1770... Loss: 1.3511... Val Loss: 1.3693
 Epoch: 13/20... Step: 1780... Loss: 1.3256... Val Loss: 1.3717
 Epoch: 13/20... Step: 1790... Loss: 1.3203... Val Loss: 1.3727
 Epoch: 13/20... Step: 1800... Loss: 1.3493... Val Loss: 1.3694
 Epoch: 14/20... Step: 1810... Loss: 1.3410... Val Loss: 1.3683
 Epoch: 14/20... Step: 1820... Loss: 1.3296... Val Loss: 1.3645
 Epoch: 14/20... Step: 1830... Loss: 1.3501... Val Loss: 1.3654
 Epoch: 14/20... Step: 1840... Loss: 1.2946... Val Loss: 1.3654

Epoch: 14/20... Step: 1850... Loss: 1.2802... Val Loss: 1.3663
 Epoch: 14/20... Step: 1860... Loss: 1.3427... Val Loss: 1.3663
 Epoch: 14/20... Step: 1870... Loss: 1.3462... Val Loss: 1.3611
 Epoch: 14/20... Step: 1880... Loss: 1.3352... Val Loss: 1.3619
 Epoch: 14/20... Step: 1890... Loss: 1.3499... Val Loss: 1.3645
 Epoch: 14/20... Step: 1900... Loss: 1.3270... Val Loss: 1.3612
 Epoch: 14/20... Step: 1910... Loss: 1.3267... Val Loss: 1.3572
 Epoch: 14/20... Step: 1920... Loss: 1.3328... Val Loss: 1.3540
 Epoch: 14/20... Step: 1930... Loss: 1.2977... Val Loss: 1.3577
 Epoch: 14/20... Step: 1940... Loss: 1.3538... Val Loss: 1.3589
 Epoch: 15/20... Step: 1950... Loss: 1.3173... Val Loss: 1.3573
 Epoch: 15/20... Step: 1960... Loss: 1.3186... Val Loss: 1.3523
 Epoch: 15/20... Step: 1970... Loss: 1.3102... Val Loss: 1.3503
 Epoch: 15/20... Step: 1980... Loss: 1.3013... Val Loss: 1.3550
 Epoch: 15/20... Step: 1990... Loss: 1.3122... Val Loss: 1.3527
 Epoch: 15/20... Step: 2000... Loss: 1.2877... Val Loss: 1.3502
 Epoch: 15/20... Step: 2010... Loss: 1.3025... Val Loss: 1.3481
 Epoch: 15/20... Step: 2020... Loss: 1.3270... Val Loss: 1.3501
 Epoch: 15/20... Step: 2030... Loss: 1.2908... Val Loss: 1.3474
 Epoch: 15/20... Step: 2040... Loss: 1.3067... Val Loss: 1.3476
 Epoch: 15/20... Step: 2050... Loss: 1.2862... Val Loss: 1.3448
 Epoch: 15/20... Step: 2060... Loss: 1.3051... Val Loss: 1.3440
 Epoch: 15/20... Step: 2070... Loss: 1.3083... Val Loss: 1.3468
 Epoch: 15/20... Step: 2080... Loss: 1.3107... Val Loss: 1.3462
 Epoch: 16/20... Step: 2090... Loss: 1.3132... Val Loss: 1.3423
 Epoch: 16/20... Step: 2100... Loss: 1.2937... Val Loss: 1.3447
 Epoch: 16/20... Step: 2110... Loss: 1.2918... Val Loss: 1.3393
 Epoch: 16/20... Step: 2120... Loss: 1.2925... Val Loss: 1.3435
 Epoch: 16/20... Step: 2130... Loss: 1.2710... Val Loss: 1.3423
 Epoch: 16/20... Step: 2140... Loss: 1.2884... Val Loss: 1.3395
 Epoch: 16/20... Step: 2150... Loss: 1.3026... Val Loss: 1.3380
 Epoch: 16/20... Step: 2160... Loss: 1.2918... Val Loss: 1.3408
 Epoch: 16/20... Step: 2170... Loss: 1.2835... Val Loss: 1.3392
 Epoch: 16/20... Step: 2180... Loss: 1.2902... Val Loss: 1.3366
 Epoch: 16/20... Step: 2190... Loss: 1.2981... Val Loss: 1.3355
 Epoch: 16/20... Step: 2200... Loss: 1.2803... Val Loss: 1.3353
 Epoch: 16/20... Step: 2210... Loss: 1.2422... Val Loss: 1.3340
 Epoch: 16/20... Step: 2220... Loss: 1.2920... Val Loss: 1.3340
 Epoch: 17/20... Step: 2230... Loss: 1.2751... Val Loss: 1.3268
 Epoch: 17/20... Step: 2240... Loss: 1.2856... Val Loss: 1.3301
 Epoch: 17/20... Step: 2250... Loss: 1.2615... Val Loss: 1.3340
 Epoch: 17/20... Step: 2260... Loss: 1.2741... Val Loss: 1.3356
 Epoch: 17/20... Step: 2270... Loss: 1.2749... Val Loss: 1.3317
 Epoch: 17/20... Step: 2280... Loss: 1.2871... Val Loss: 1.3307
 Epoch: 17/20... Step: 2290... Loss: 1.2858... Val Loss: 1.3324
 Epoch: 17/20... Step: 2300... Loss: 1.2470... Val Loss: 1.3331
 Epoch: 17/20... Step: 2310... Loss: 1.2653... Val Loss: 1.3278
 Epoch: 17/20... Step: 2320... Loss: 1.2668... Val Loss: 1.3278

Epoch: 17/20... Step: 2330... Loss: 1.2614... Val Loss: 1.3283
Epoch: 17/20... Step: 2340... Loss: 1.2804... Val Loss: 1.3286
Epoch: 17/20... Step: 2350... Loss: 1.2763... Val Loss: 1.3263
Epoch: 17/20... Step: 2360... Loss: 1.2864... Val Loss: 1.3262
Epoch: 18/20... Step: 2370... Loss: 1.2520... Val Loss: 1.3239
Epoch: 18/20... Step: 2380... Loss: 1.2662... Val Loss: 1.3236
Epoch: 18/20... Step: 2390... Loss: 1.2720... Val Loss: 1.3270
Epoch: 18/20... Step: 2400... Loss: 1.2849... Val Loss: 1.3221
Epoch: 18/20... Step: 2410... Loss: 1.2810... Val Loss: 1.3243
Epoch: 18/20... Step: 2420... Loss: 1.2575... Val Loss: 1.3208
Epoch: 18/20... Step: 2430... Loss: 1.2686... Val Loss: 1.3221
Epoch: 18/20... Step: 2440... Loss: 1.2543... Val Loss: 1.3241
Epoch: 18/20... Step: 2450... Loss: 1.2468... Val Loss: 1.3196
Epoch: 18/20... Step: 2460... Loss: 1.2694... Val Loss: 1.3183
Epoch: 18/20... Step: 2470... Loss: 1.2646... Val Loss: 1.3185
Epoch: 18/20... Step: 2480... Loss: 1.2506... Val Loss: 1.3166
Epoch: 18/20... Step: 2490... Loss: 1.2490... Val Loss: 1.3123
Epoch: 18/20... Step: 2500... Loss: 1.2541... Val Loss: 1.3178
Epoch: 19/20... Step: 2510... Loss: 1.2518... Val Loss: 1.3148
Epoch: 19/20... Step: 2520... Loss: 1.2713... Val Loss: 1.3165
Epoch: 19/20... Step: 2530... Loss: 1.2720... Val Loss: 1.3177
Epoch: 19/20... Step: 2540... Loss: 1.2934... Val Loss: 1.3175
Epoch: 19/20... Step: 2550... Loss: 1.2495... Val Loss: 1.3147
Epoch: 19/20... Step: 2560... Loss: 1.2570... Val Loss: 1.3114
Epoch: 19/20... Step: 2570... Loss: 1.2389... Val Loss: 1.3113
Epoch: 19/20... Step: 2580... Loss: 1.2870... Val Loss: 1.3151
Epoch: 19/20... Step: 2590... Loss: 1.2358... Val Loss: 1.3084
Epoch: 19/20... Step: 2600... Loss: 1.2451... Val Loss: 1.3108
Epoch: 19/20... Step: 2610... Loss: 1.2535... Val Loss: 1.3127
Epoch: 19/20... Step: 2620... Loss: 1.2328... Val Loss: 1.3126
Epoch: 19/20... Step: 2630... Loss: 1.2364... Val Loss: 1.3094
Epoch: 19/20... Step: 2640... Loss: 1.2537... Val Loss: 1.3066
Epoch: 20/20... Step: 2650... Loss: 1.2544... Val Loss: 1.3066
Epoch: 20/20... Step: 2660... Loss: 1.2589... Val Loss: 1.3101
Epoch: 20/20... Step: 2670... Loss: 1.2660... Val Loss: 1.3070
Epoch: 20/20... Step: 2680... Loss: 1.2539... Val Loss: 1.3083
Epoch: 20/20... Step: 2690... Loss: 1.2438... Val Loss: 1.3116
Epoch: 20/20... Step: 2700... Loss: 1.2584... Val Loss: 1.3075
Epoch: 20/20... Step: 2710... Loss: 1.2243... Val Loss: 1.3098
Epoch: 20/20... Step: 2720... Loss: 1.2251... Val Loss: 1.3096
Epoch: 20/20... Step: 2730... Loss: 1.2169... Val Loss: 1.3067
Epoch: 20/20... Step: 2740... Loss: 1.2184... Val Loss: 1.3068
Epoch: 20/20... Step: 2750... Loss: 1.2306... Val Loss: 1.3053
Epoch: 20/20... Step: 2760... Loss: 1.2248... Val Loss: 1.3063
Epoch: 20/20... Step: 2770... Loss: 1.2615... Val Loss: 1.3041
Epoch: 20/20... Step: 2780... Loss: 1.2828... Val Loss: 1.3007