CS 559 - Fall 2021
Homework #6

Krishna Garg
kgarg8@uic.edu

1 Default Hyperparameters

Epochs: 25

Optimizer: Adam

Weight Decay: 0.03

Batch size: 100

Learning rate: 1e-3

Scheduler: StepLR

Scheduler step size: 1

Scheduler gamma: 0.7

Loss: CrossEntropyLoss

Dropout Probability: 0.5

Hidden size (LSTM): 512

2 Network

I created a neural network with 2 LSTM layers of hidden size 512 in each layer and with a dropout probability of 0.5. The LSTM layers were followed by the linear layer for classification.

3 Loss function

$$Loss_{cross-entropy} = \sum_{i=1}^{n} t_i * log(p_i)$$

where n refers to number of labels (27 in our case) and p_i refers to softmax probability value for the corresponding label.

In figure 1, we can see that the loss value initially decreases sharply since the model starts to learn from the dataset but soon it converges to a loss value of $\sim$1.533 after 10 epochs or so. I think the network reaches saturation at this point given the current hyperparameters. We can also observe the corresponding intuitive behavior in the accuracy plot since the accuracy also saturates at around 10 epochs and it is kind of low (58%). I tried tuning the hyperparameters a lot but nothing worked out. I could have tried other loss functions like BCEWithLogitsLoss but I didn't have the sufficient time.

4 Generation Output

4.1 Names starting with 'a'

['aniineenanie', 'aniaeeaannineiae', 'aine', 'aaen', 'aaeie', 'aeniaaienii', 'aaieaiineenin', 'aei', 'aie', 'aniiaa', 'aeeeeeaina', 'aiane', 'aaeniineaean', 'aniee', 'aiee', 'ainnna', 'aeaeeeieeeneniie', 'aaienannnenaa', 'ainaeai', 'aiaeinneeiee']

4.2 Names starting with 'x'

['xnaienii', 'xan', 'xnain', 'xenan', 'xaaei', 'xna', 'xaanan', 'xaieen', 'xiiienei', 'xeieiae', 'xiiaanaaaeni', 'xnai', 'xiiniina', 'xeaeineaieaannne', 'xnii', 'xeiaeeea', 'xieeneiieiieanaa', 'xiiaaaiiie', 'xnee', 'xieene']

4.3 Design Choice

I randomly picked up one of the top-5 logits from the model output and appended it to the generated string until I reached <EON> or reached the maximum sequence length of 15.

$TOPK = 5$, $MAX\_OUTPUT\_LEN = 15$ were the only hyperparameters for the generation.

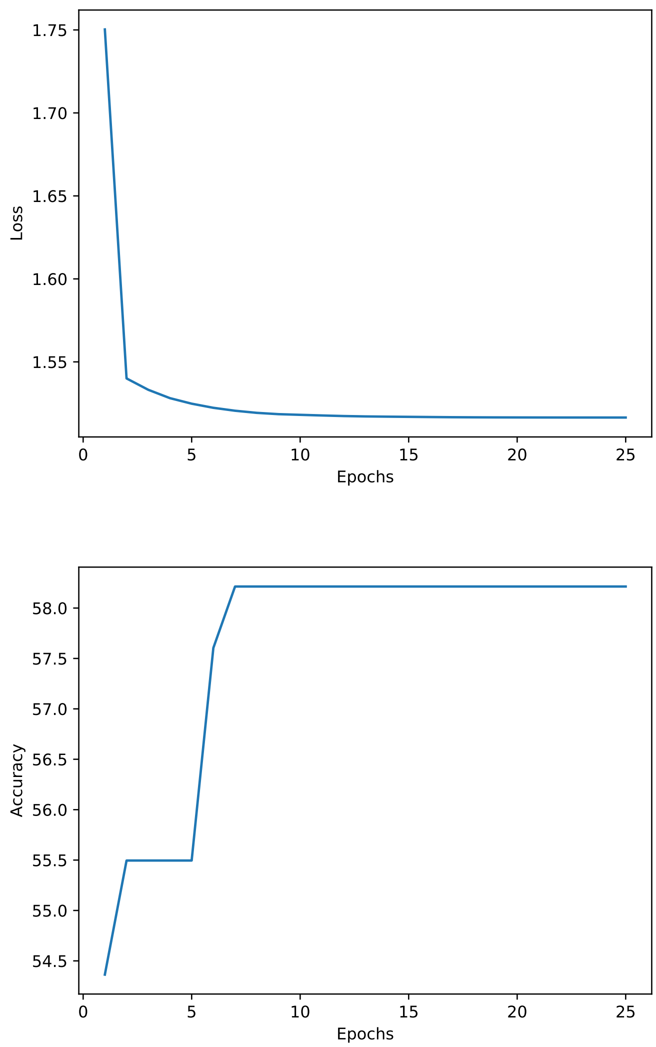

Figure 1: Plots: (a) Loss vs Epochs, (b) Accuracy vs Epochs for training

Code - 0701-656377418-Garg.py

```
from torch.optim.lr_scheduler import StepLR

EON    = '<EON>'
alphabet = 'abcdefghijklmnopqrstuvwxyz'
alphabet = [char for char in alphabet] # convert to list of chars
alphabet.append(EON)

alphabet = {v:k for k, v in enumerate(alphabet)} # convert to dict
device   = torch.device("cuda" if torch.cuda.is_available() else "cpu")
MAX_SEQ_LEN = 11

def one_hot_encoding(letter):
   val = alphabet[letter]
   encoding = [0 for i in range(27)]
   encoding[val] = 1
   return encoding


class CharLSTM(nn.Module):
   def __init__(self, input_size, hidden_size, output_size, num_layers,
       batch_size, dropout_prob):
      super(CharLSTM, self).__init__()
      self.input_size = input_size
      self.hidden_size = hidden_size
      self.output_size = output_size
      self.num_layers = num_layers
      self.batch_size = batch_size
      self.dropout_prob = dropout_prob
      self.lstm     = nn.LSTM(self.input_size, self.hidden_size,
         self.num_layers, batch_first=True, dropout=self.dropout_prob)
      # self.lstm2    = nn.LSTM(self.hidden_size, self.hidden_size,
         self.num_layers, batch_first=True, dropout=self.dropout_prob)
      self.dropout  = nn.Dropout(self.dropout_prob)
      self.linear   = nn.Linear(self.hidden_size, self.output_size)
      self.h_0      = torch.zeros(self.num_layers, self.batch_size,
         self.hidden_size).to(device)
      self.c_0      = torch.zeros(self.num_layers, self.batch_size,
         self.hidden_size).to(device)

   def forward(self, x):
      x, (h_n, c_n) = self.lstm(x.float(), (self.h_0, self.c_0))
      # x, (h_n, c_n) = self.lstm2(x, (h_n, c_n))
      # x = self.dropout(x)
      x = x.reshape(-1, self.hidden_size)
      x = self.linear(x)
      return x, (h_n, c_n)


def load_data():
   file = open('names.txt', 'r')
   data = file.readlines()
   input_data, output_data = [], []
   for name in data:
      name = name.replace('\n','').lower()
      name = [char for char in name]
      while len(name) < MAX_SEQ_LEN: # pad EONs
         name.append(EON)

      # ground truths will be the next character
      gt_output = name[1:]
      gt_output.append(EON)
```

```python
        name_tensor  = torch.tensor([one_hot_encoding(leter) for leter in
            name])
        gt_output_tensor = torch.tensor([one_hot_encoding(leter) for leter
            in gt_output])

        input_data.append(name_tensor)
        output_data.append(gt_output_tensor)

    return input_data, output_data


def get_batches(input_data, output_data, batch_size):
    num_batches = len(input_data) // batch_size
    for i in range(num_batches):
        x = input_data[i*batch_size: (i+1)*batch_size]
        y = output_data[i*batch_size: (i+1)*batch_size]
        x = torch.stack(x) # (batch_size, max_seq_len, len(alphabet))
        y = torch.stack(y)
        yield x, y


def train(args, model, input_data, output_data, optimizer, epoch):
    model.train()
    tot_loss = 0
    correct = 0
    for batch_idx, (data, target) in enumerate(get_batches(input_data,
        output_data, args.batch_size)):
        data, target = data.to(device), target.to(device)
        optimizer.zero_grad()
        output, (h_n, c_n) = model(data)
        target = target.reshape(-1, target.shape[2])
        target = target.argmax(axis=1)
        loss = torch.nn.CrossEntropyLoss()(output, target)
        tot_loss = tot_loss + loss.item()
        loss.backward()
        optimizer.step()

        pred  = output.argmax(dim=1)
        correct += pred.eq(target).sum().item()

        if batch_idx % args.log_interval == 0:
            print('Train Epoch: {} [{}/{} ({:.0f}%)]\tLoss: {:.6f},
                Accuracy: {:.2f}%'.format(
                epoch, batch_idx * len(data), len(input_data), 100. *
                    batch_idx * args.batch_size / len(input_data),
                tot_loss/ (batch_idx+1),
                    100.0*correct/((batch_idx+1)*args.batch_size*MAX_SEQ_LEN)))

    loss = tot_loss / (batch_idx+1)
    acc = 100.0 * correct / (len(input_data) * MAX_SEQ_LEN)
    print('End of Epoch: {}'.format(epoch))
    print('Training Loss: {:.6f}, Training Accuracy:
        {:.2f}%'.format(loss, acc))
    return loss, acc


def main():
    parser = argparse.ArgumentParser(description='PyTorch MNIST Example')
    parser.add_argument('--batch-size', type=int, default=16, help='input
        batch size for training (default: 16)')
    parser.add_argument('--epochs', type=int, default=25, help='number of
        epochs to train (default: 10)')
    parser.add_argument('--lr', type=float, default=1e-3, help='learning
        rate (default: 1e-3)')
```

```
    parser.add_argument('--dropout_prob', type=float, default=0.5,
        help='dropout probability (default: 0.5)')
    parser.add_argument('--hidden_size', type=int, default=512,
        help='size of hidden layers in LSTM (default: 512)')
    parser.add_argument('--num-layers', type=int, default=2, help='number
        of layers in LSTM')
    parser.add_argument('--gamma', type=float, default=0.7,
        help='Learning rate step gamma (default: 0.7)')
    parser.add_argument('--weight_decay', type=float, default=0.03,
        help='Weight decay for Optimizer (default: 0.03)')
    parser.add_argument('--seed', type=int, default=112, help='random
        seed (default: 112)')
    parser.add_argument('--log-interval', type=int, default=32, help='how
        many batches to wait before logging training status')
    parser.add_argument('--save-model', action='store_true',
        default=True, help='For Saving the current Model')
    args = parser.parse_args()
    print(args)

    random.seed(args.seed)
    np.random.seed(args.seed)
    torch.manual_seed(args.seed)
    torch.backends.cudnn.deterministic = True
    torch.backends.cudnn.benchmark = False

    input_data, output_data = load_data()

    model   = CharLSTM(len(alphabet), args.hidden_size, len(alphabet),
        args.num_layers, args.batch_size, args.dropout_prob).to(device)
    optimizer = optim.Adam(model.parameters(), lr=args.lr,
        weight_decay=0.03)
    scheduler = StepLR(optimizer, step_size=1, gamma=args.gamma)

    train_losses, train_accs = [], []
    for epoch in range(1, args.epochs + 1):
       train_loss, train_acc = train(args, model, input_data,
           output_data, optimizer, epoch)
       train_losses.append(train_loss)
       train_accs.append(train_acc)
       scheduler.step()

    if args.save_model:
       torch.save(model.state_dict(), "0702-656377418-Garg.pt")

    # plot
    epoch_arr = [(i+1) for i in range(args.epochs)]
    plt.figure()
    plt.xlabel('Epochs')
    plt.ylabel('Loss')
    plt.plot(epoch_arr, train_losses)
    plt.savefig('Loss vs. Epochs.pdf')

    plt.figure()
    plt.xlabel('Epochs')
    plt.ylabel('Accuracy')
    plt.plot(epoch_arr, train_accs)
    plt.savefig('Accuracy vs. Epochs.pdf')


if __name__ == '__main__':
    main()
```

Code - 0703-656377418-Garg.py

```python
# generation
import torch, torch.nn as nn, torch.nn.functional as F, random, numpy as
    np, pdb

EON    = '<eon>'
alphabet = 'abcdefghijklmnopqrstuvwxyz'
alphabet = [char for char in alphabet] # convert to list of chars
alphabet.append(EON)

alphabet  = {v:k for k, v in enumerate(alphabet)} # convert to dict
num_to_alpha = {k:v for k, v in enumerate(alphabet)} # convert to dict
device    = torch.device("cuda" if torch.cuda.is_available() else "cpu")
random.seed(112)

def one_hot_encoding(letter):
   val = alphabet[letter]
   encoding = [0 for i in range(27)]
   encoding[val] = 1
   return encoding


class CharLSTM(nn.Module):
   def __init__(self, input_size, hidden_size, output_size, num_layers,
       batch_size, dropout_prob):
      super(CharLSTM, self).__init__()
      self.input_size = input_size
      self.hidden_size = hidden_size
      self.output_size = output_size
      self.num_layers = num_layers
      self.batch_size = batch_size
      self.dropout_prob = dropout_prob
      self.lstm     = nn.LSTM(self.input_size, self.hidden_size,
          self.num_layers, batch_first=True, dropout=self.dropout_prob)
      self.dropout  = nn.Dropout(self.dropout_prob)
      self.linear   = nn.Linear(self.hidden_size, self.output_size)
      self.h_0      = torch.zeros(self.num_layers, self.batch_size,
          self.hidden_size).to(device)
      self.c_0      = torch.zeros(self.num_layers, self.batch_size,
          self.hidden_size).to(device)

   def forward(self, x):
      x, (h_n, c_n) = self.lstm(x.float(), (self.h_0, self.c_0))
      x = x.reshape(-1, self.hidden_size)
      x = self.linear(x)
      return x, (h_n, c_n)


def generate(initial_char, num_names, model):


   names = []
   while(len(names) < num_names):

      # include initial_char
      input    = torch.tensor(one_hot_encoding(initial_char.lower()))
      input    = input.unsqueeze(0).unsqueeze(0).to(device) # unsqueeze
          twice to make 3-D tensor
      input_char = initial_char

      output_char = '' # initialize to anything other than EON
      output_string = initial_char
      MAX_OUTPUT_LEN = 15 # max_length of generated names

      while (output_char!=EON and MAX_OUTPUT_LEN > 0):
```

```
            new_input = torch.tensor(one_hot_encoding(input_char.lower()))
            new_input = new_input.unsqueeze(0).unsqueeze(0).to(device)
            input = torch.cat((input, new_input), dim=1) # append new char
                to the current string

            output, (h_n, c_n) = model(input)

            TOPK = 5
            _, idxs   = output[-1].topk(TOPK)
            idx       = random.randint(0, TOPK-1)
            pred      = idxs[idx].item()
            output_char = num_to_alpha[pred]
            if output_char != EON: # dont append eon to name
               output_string += output_char

            input_char = output_char
            MAX_OUTPUT_LEN -= 1

         # print(output_string)
         if output_string not in names and len(output_string) > 2: # names
             at least 3 chars long
            names.append(output_string)

      return names

# hyperparamters
hidden_size = 512
num_layers = 2
dropout_prob = 0.5
batch_size = 1
num_names = 20

saved_model_path = '0702-656377418-Garg.pt'
checkpoint    = torch.load(saved_model_path, map_location=device)
model         = CharLSTM(len(alphabet), hidden_size, len(alphabet),
    num_layers, batch_size, dropout_prob).to(device)
model.load_state_dict(checkpoint)
model.eval()
print(generate('a', num_names, model))
print(generate('x', num_names, model))
```