Character-level Language Modelling with LSTM

Kshitij Kar

January 2024

1 Build the Model

1.1 Explain in your own words what does the code do when you use the flag *default_train*.

./assignment3.py

```
def main():
  parser = argparse.ArgumentParser(
        description='Train LSTM'
    )

    parser.add_argument(
        '--default_train', dest='default_train',
        help='Train LSTM with default hyperparameter',
        action='store_true'
    )
```

The **main()** function is run when the name of the file is run in the command line. The **argparse** library is standard python library used for parsing through the command line arguments.

Here's a brief explanation:

(a) **argparse.ArgumentParser** is used to create a command-line argument parser. The description parameter provides a brief description of the script or application.

(b) **parser.add_argument** is used to define a command-line option. In this case:

- **–default_train** is the name of the command-line option.
- **dest='default_train'** specifies the attribute in the namespace where the value of the option will be stored.
- **help** provides a short description of what the option does.
- **action='store_true'** indicates that if the option is present in the command line, the **default_train** attribute will be set to True. If the option is not present, it remains False.

The purpose of this code is to allow the script or application to be run with the –**default_train** option, which triggers the training of an LSTM model with default hyper-parameters. The presence of this option is used to determine whether the script should perform a default training run.

This then runs the following code:
./assignment3.py

```
if args.default_train:
    print(f"Start Time: {datetime.datetime.now()}")
    n_epochs = 2000
    print_every = 100
    plot_every = 10
    hidden_size = 128
    n_layers = 2
    lr = 0.005

    decoder = LSTM(
        input_size = n_characters,
        hidden_size = hidden_size,
        num_layers = n_layers,
        output_size = n_characters)

    decoder_optimizer = torch.optim.Adam(decoder.parameters(),
        lr=lr)
    print("Parameters of LSTM: \n")
    print(decoder)
    print("Parameters of optimizer: \n")
    print(f"Learning Rate: {lr}")
    print(decoder_optimizer)
    start = time.time()
    all_losses = []
    loss_avg = 0

    print("---------------------------------STARTING TRAINING
        ---------------------------------")

    for epoch in range(1, n_epochs+1):
        loss = train(decoder, decoder_optimizer, *
            random_training_set())
        loss_avg += loss

        if epoch % print_every == 0:
            print(f'[{time_since(start)} ({epoch} {np.round(
                epoch/n_epochs * 100,4)} %) {np.round(loss,4)}]
                ')
            print(generate(decoder, 'A', 100), '\n')

        if epoch % plot_every == 0:
            all_losses.append(loss_avg / plot_every)
            loss_avg = 0
```

1.2 What do the defined parameters do?

- **n_epochs** : Integer input. Defines the number of data points the model is trained with.
- **print_every** : Integer input. Defines the number of training instances between which the average loss is printed.
- **plot_every** : Integer input. Defines the number of training instances between which the average loss is saved to a list used for plotting later.
- **hidden_size** : Integer input. Defines the feature vector size of the hidden dimension.
- **n_layers** : Integer input. Define the number of LSTM layers to be stacked.
- **lr** : Float input. Defines the learning rate used for the optimization algorithm to reduce the loss.

1.3 What is happening inside the training loop?

The training loop involves two chunks of code :

./assignment3.py

```
for epoch in range(1, n_epochs+1):
        loss = train(decoder, decoder_optimizer, *
            random_training_set())
        loss_avg += loss

        if epoch % print_every == 0:
            print(f'[{time_since(start)}-({epoch}-{np.round(
                epoch/n_epochs-*-100,4)}-%)-{np.round(loss,4)}]
                ')
            print(generate(decoder, 'A', 100), '\n')

        if epoch % plot_every == 0:
            all_losses.append(loss_avg / plot_every)
            loss_avg = 0
```

./language_model.py

```
def train(decoder : object, decoder_optimizer : object,
      inp : torch.TensorType, target : torch.TensorType) ->
          float:
"""
Train the decoder model for a single step using the given input
     and target sequences.

Args:
    decoder (object): The decoder model to be trained.
    decoder_optimizer (object): The optimizer for updating the
        decoder's parameters.
    inp (torch.TensorType): The input sequence tensor.
    target (torch.TensorType): The target sequence tensor.

Returns:
```

```
        float: The normalized loss for the current training step,
            averaged over the sequence length.
    """
    hidden, cell = decoder.init_hidden()
    decoder.zero_grad()
    loss = 0
    criterion = nn.CrossEntropyLoss()

    for c in range(CHUNK_LEN):
        output, (hidden, cell) = decoder(inp[:, c], hidden, cell)
        loss += criterion(output, target[:, c].view(1))

    loss.backward()
    decoder_optimizer.step()

    return loss.item() / CHUNK_LEN
```

The hidden and cell tensors are created using the **decoder.init_hidden()** function that creates zero tensors of size (n_layers,batch_size,hidden_size). **decoder.zero_grad()** resets the gradients to zero.
criterion = nn.CrossEntropyLoss() initializes the loss calculation class. The "for loop" runs for c values ranging from 0 to CHUNK_LEN. Each individual string in inp chuck is fed to the LSTM sequentially and compares the predicted string to the actual string using the CrossEntropyLoss method. The total loss is then prepared for back propagation through the model using the **loss.backward()** method, ie. the gradients are calculated. The parameters of the model are the updated according to the gradients using the **decoder_optimizer.step()** method. The train function then returns the average loss given an input text chuck and the output text chuck.

The above for loop saves the returned loss and then prints the average loss to the terminal. It also generates text starting with the letter A using the trained LSTM model.

2 Hyper-parameter Tuning

n_epochs	hidden_size	n_layers	learning rate	optimizer	bits per character	loss
2000	128	2	0.005	Adam	2.60749	1.7316
3000	128	2	0.005	Adam	2.491	1.6158
2000	64	2	0.005	Adam	2.80389	1.939
2000	256	2	0.005	Adam	2.46321	1.6705
2000	128	1	0.005	Adam	2.59603	1.759
2000	128	3	0.005	Adam	2.67253	1.8535
2000	128	2	0.1	Adam	3.42708	2.3311
2000	128	2	0.01	Adam	2.59886	1.682
2000	128	2	0.0001	Adam	3.61083	2.5484
2000	128	2	0.005	AdamW	2.54474	1.806
2000	128	2	0.005	RMSprop	2.46079	1.6703

3 Plotting the Training Losses

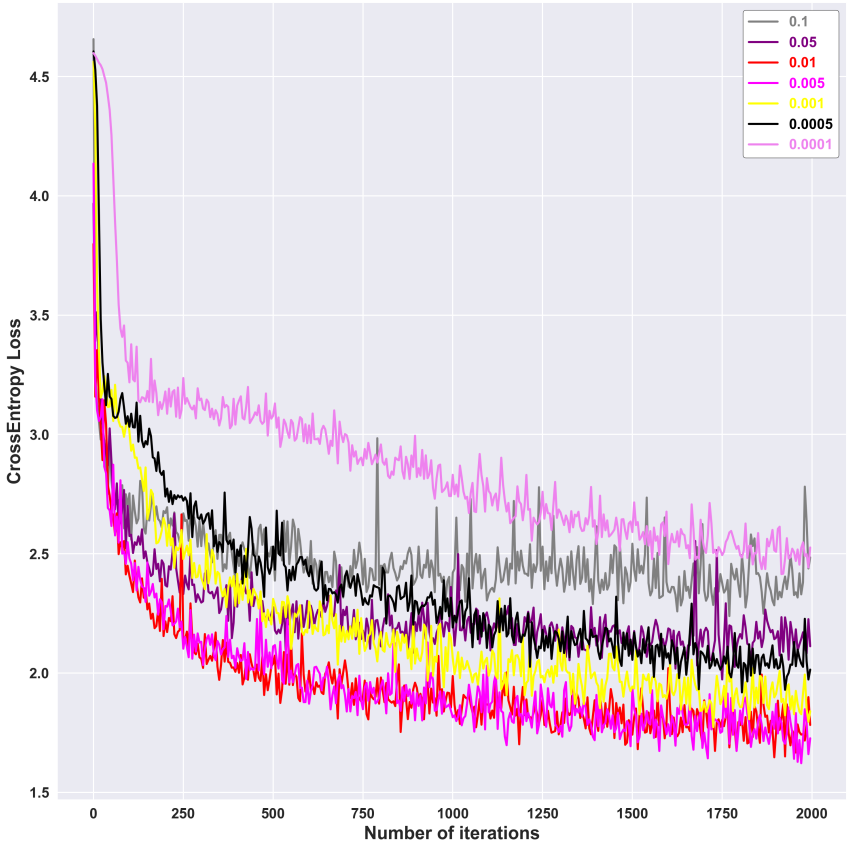

Figure 1: Graph of loss vs number of training iterations for different learning rates

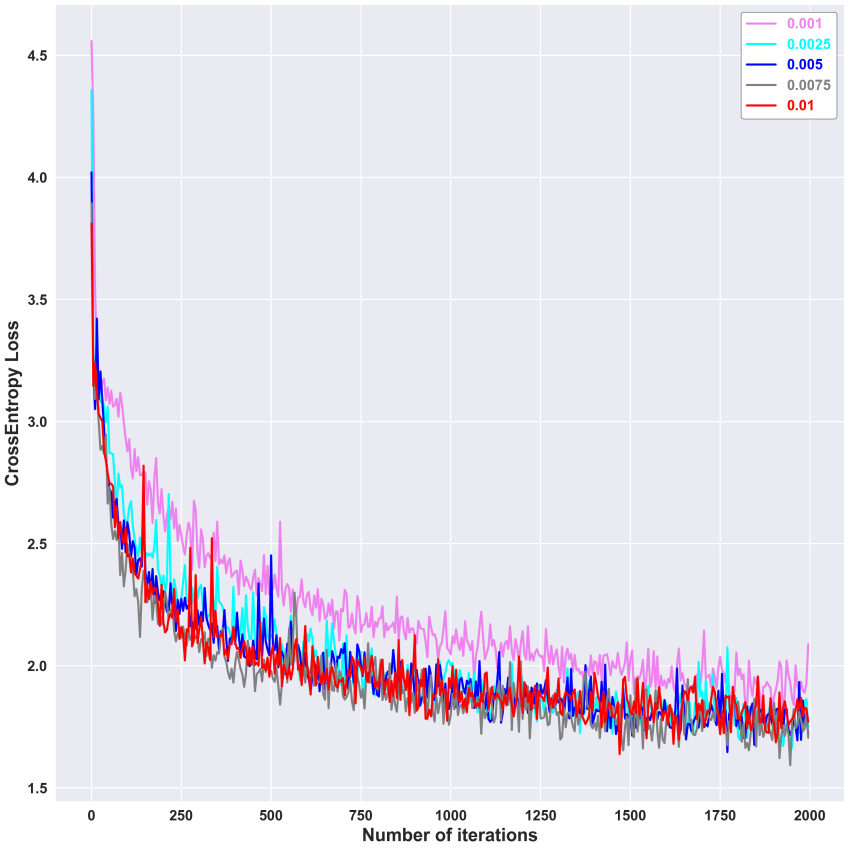

Figure 2: Closer investigation of best learning rates.

4 Generating at different temperatures

4.1 Discuss what you observe in the output when you increase the temperature values? In your understanding, why does changing the temperature affect the output as the way you observed?

Temperature determines the amount of risk the RNN is willing to take while choosing the next character. As we can see in **./output/diff_temp.txt**, for lower temperature values such as *0.4*, the model is more cautious with the character is chooses whereas with higher temperature values like *0.9*, it can choose more rare and infrequent characters. For lower temperature values, the character level model generates words that occur often in the training data such as *of,was,the,it*. At a temperature of **0.7**, the model misspells longer words because its a character level model but it chooses words that are more legitimate such as *spiriting, night, thought*. At a yet higher temperature value of **0.9**, the model starts generating gibberish words that do not make any linguistic sense.

TEMPERATURE : 0.4
PREDICTION : and almost the pillow of the back and the sirds and the friend. I said, and she had no could have the convinced that is he had been and the will nothing, and the could a surpering the possible was sort of t
ORIGINAL TEXT : and almost immediately called for a candle, and went to bed, as if he were not quite safe anywhere else. He did not actually stagger under the negus; but I should think his placid little pulse must h

TEMPERATURE : 0.7
PREDICTION : ellent of earness, that have comple, my aunt, she was a her night, you have begs, an you thought of much my like in her. I did him to down as of the contreed the spiriting and for the are thim that true is l
ORIGINAL TEXT : ellent of women, as he would know full well if he knew her better. The mere notion of the possibility of his ever seeing her again, appeared to terrify him. He replied with a small pale smile, 'Is sh

TEMPERATURE : 0.9
PREDICTION : bury today, in this dear of the faurt.

"They ancy unhibles when I disent. "The spreites. Bust count ease, and had book on, with the face the good hand.'

'Discounted the quarted before) handed his new I she
ORIGINAL TEXT : bury today without stopping, if I had been coming to anyone but her.

She was pleased, but answered, 'Tut, Trot; MY old bones would have kept till tomorrow!' and softly patted my hand again, as I sat l

5 Discussion

5.1 Discuss what are the risks of having a language model that generates text automatically? Who is responsible for the output: the person who builds the model, the person who writes the generating script, the person who uses the outputs? What cautions should we take when using language models for this purpose?

A language model that generates text automatically has a very narrow context window of its environment in which it is generating the text. This is a risky as it lacks sufficient initial context to generate the most suitable and probable output. The person who builds the model has control of its hyper-parameters and the data that its trained on and hence is responsible for the language model. However, the person that generates the output and then uses the output is also responsible for the application of the language model and its derivatives. When using language models, before using the output generated, the party using the output should verify and double check the output before using it. They should look out for cases of "hallucinations" where the model generates something completely wrong or unrelated to the task assigned.