Advanced Natural Language Processing

Assignment 2: Character-level Language Modelling with LSTM

December 24, 2022

1 Build the Model

Q. Explain in your own words what the code does when you use the flag `--default_train`.

```
./assignment.py
    parser = argparse.ArgumentParser(
        description='Train LSTM'
    )

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

The code uses the python **argparse** library, which is a standard built-in library used to parse command-line arguments and sub-commands. We do this by initializing a ArgumentParser, which will look for the argument `--default_train`. If the user while calling the python file provides the argument, then the intended code which utilises that argument as a parameter is executed. As for the `--default_train` argument, we follow the standard training procedure mentioned under the files `assignment.py`.

```
./assignment.py
    if args.default_train:
        n_epochs = config["default"]["num_epochs"]
        print_every = 100
        plot_every = 10
        hidden_size = config["default"]["hidden_size"]
        n_layers = config["default"]["num_layers"]
        lr = config["default"]["lr"]

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

        decoder_optimizer = get_optimizer(
            decoder=decoder,
            optim=config["default"]["optimizer"],
            lr=lr
        )
```

Note: for simplicity and better adaptability for debugging, I've shifted the hyperparameter initialization to the file `config.yaml`.

Q. What do the defined parameters do?

We refer to these defined parameters as hyperparameters since they indirectly affect the model's performance. This includes, i) `# of Epochs`, ii) `dimensionality of hidden layer`, iii) `# of layers`, iv) `learning rate` etc. We utilize these hyperparameters while training the model, e.g. in our case, the hyperparameter `n_epochs` determines how many times the model will be iterated over the entire training data. Similarly, the hyperparameter `hidden_size` determines the internal representation of input data and how varying the dimensionality will affect the model's performance toward predicting the results.

Q. And what is happening inside the training loop?

./assignment.py

```
    start = time.time()
        all_losses = []
        loss_avg = 0

        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('[{} ({} {}%) {:.4f}]'.format(
                    time_since(start),
                    epoch,
                    epoch/n_epochs * 100,
                    loss
                ))
                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, decoder_optimizer, inp, target):

    # push PyTorch model and other associated params. and attr. to same device "cuda" OR "cpu"
    decoder = decoder.to(device)
    inp = inp.to(device)
    target = target.to(device)

    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
```

We first initialize a PyTorch model, which in our case is a char-RNN model under file `./model/model.py`. Next, we initialize an optimizer, which primarily determines the algorithm we're going to use to optimize our model while training. Additionally, we do require a loss function, which is initialized under the `train` function call. Next, the training loop iterates based on the `n_epochs` parameter. We first calculate the CrossEntropyLoss using the model's `output` & `target`. In PyTorch, the backward propagation is quite simpler and is done by calling the `loss.backward()` function. And then we perform the optimization step. The loss is then reported back for inference.

2 Plotting the Training Losses

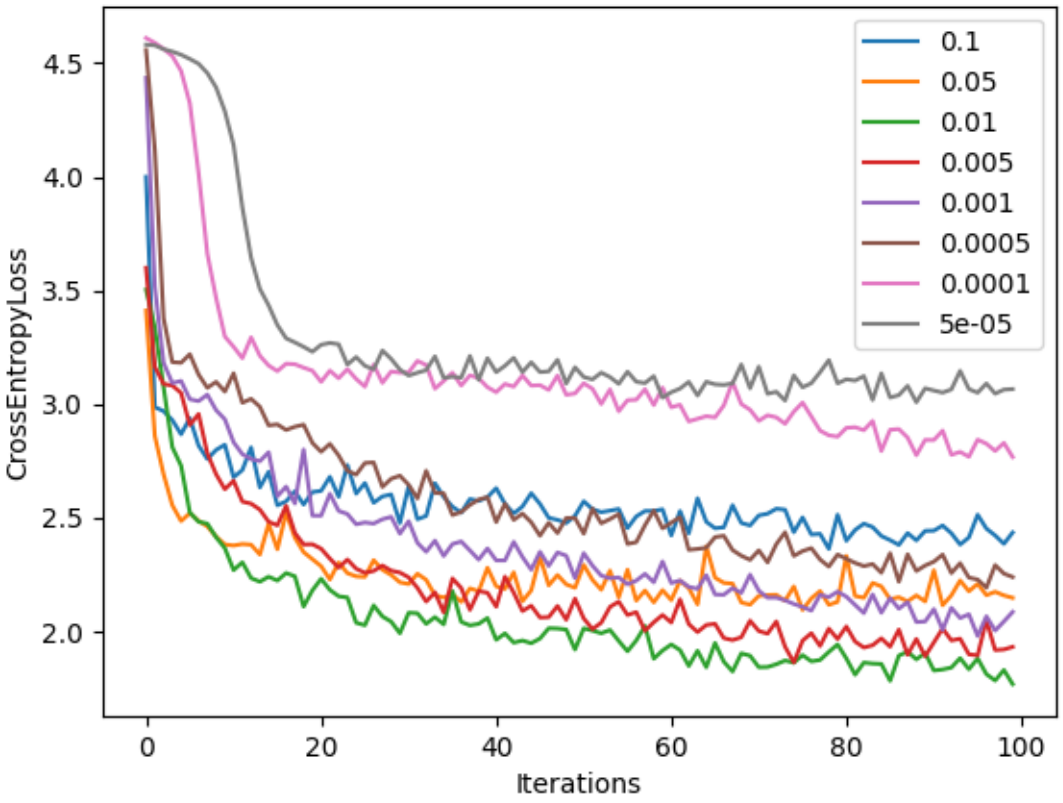

Figure 1: Shows the decrease in CrossEntropyLoss over time. Here the values are recorded after every 10 training steps. Also, learning rates and their respective color indication are mentioned in the pane on the top-right side of the graph.

3 Generating at different temperatures

Q. 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?

Note: The model is trained using the `dickens_train.txt` *data provided under the* `./data` *folder. We used the default hyperparameter setting to train the model.*

Temperature determines how risky the RNN should be toward predicting the next character given the input string of characters. The model's predictive power is inversely proportional to temperature. With

a low-temperature value, the model will be more cautious in predicting the next character, whereas with a high-temperature value close to 1, the model will choose to consider rare and less frequent characters, putting the predictive power at risk.

We show this under the file `./output/diff_temp_output.txt`. For a low-temperature value of **0.5**, the model prediction closely follows the trend of including word/sequence-of-characters that occur too often in the training data, such as *was, it, the, of.* Moreover, these are the words, which we often classify as stopwords, however, when increasing the temperature to **0.85**, the outputs are more sensible and legit. There is the inclusion of words like *experiencing, less, weather*, however, since we're using character-level RNN, some of the words are being misspelled. Which we can overcome using more diverse and large training data. Next, at a much high value of temperature often too close to **1**, we notice a drastic change in the model outputs. The predictions include more gibberish words which sometimes intends as if the model is trying to start incorporating proper nouns and adjectives.

Q. 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, or the person who uses the outputs? What cautions should we take when using language models for this purpose?

An automatic language model is unaware of the information of its environment since it is deprived of the initial information which is needed to be processed hence to generate the most probable output. Next, logically the person who scripts the model is more or less the person who builds it and thus has the access to modify the hyperparameters responsible to generate the model, which indirectly is responsible for updating the original parameters (weights & biases). Also, the person who uses the output might not have any implicit effect on the model functioning, however, can have the right to the variability of the model's input and other parameters like temperature, etc.