Names: Chad Yu, Joshua Huang
NetIDs: cky25, jth239

CS 4740 HW 2 Questions

Q1.1.1: In less than three sentences, describe the effect of changing `min_freq` and `remove_frac` on the vocabulary size. How should one go about setting these two hyperparameters?

From the graphs above, we can see that the vocab size decreases in an exponential manner with respect to `min_freq` and linearly with respect to `remove_frac`, which leads me to believe that one should go about by setting `min_freq` to a lower number relatively (e.g., 5) to `remove_frac`, as not only does a high `min_freq` remove sufficiently decreases the vocab size for faster training, but it could remove named entities worth tagging, and could very quickly progress towards a distribution with almost all "O" tags. On the other hand, we could set `remove_frac` to something higher (e.g. 0.4), and the vocab size would be proportionately reduced, and not necessarily decimated like would be with `min_freq`.

Q1.2.1: Following our data distribution, (in under three sentences,) explain why is dynamic padding better than static padding?

Given our data distribution of the number of tokens in each sequence, we can see that it is not a uniform distribution, so static padding will be an erroneous approach; specifically, if we pad each sentence to the same length, then the padded data will misrepresent the actual distribution of sequence length, which is not uniform. Thus, to represent the varying sequence lengths, dynamic padding will do so, and with batching, it will also not specifically go into every sequence and do a padding catered to that sequence, which also saves us time.

Q3.1: What is the difference between `nn.CrossEntropyLoss` and `nn.NLLLoss`? What implication does using `nn.CrossEntropyLoss` have on the use of `nn.Softmax` before passing the logits to `nn.CrossEntropyLoss`?

The sole difference between `nn.CrossEntropyLoss` and `nn.NLLLoss` is that for the case where `ignore_index` is specified (which is what we do in the trainer), `nn.CrossEntropyLoss` internally applies `nn.LogSoftmax` and then takes the NLL loss on those results, so that we don't have to use `nn.LogSoftmax` on our outputs. The documentation explicitly says: "Note that this case is equivalent to applying `LogSoftmax` on an input, followed by `NLLLoss`"

Q3.2: Before proceeding, observe the `@torch.no_grad()` annotation on `Trainer._eval_epoch()`—why are we using "no grad" at evaluation?

The `no_grad` decorator disables gradient calculation for a given function, which makes sense for evaluation, since we are not minimizing loss anymore, and thus not executing backpropagation anymore as well. And since we are 100% not calling `loss.backward()` within evaluation, this allows us to reduce memory consumption by eliminating computations that would have `requires_grad=True`, as detailed in the documentation for `no_grad`.

Q4.3.1: In comparison to HMM and MEMM models from the previous assignment, how did your best FFNN model *perform*? Performance is more than just "performance on some metric"; it includes efficiency (e.g., training time), memory (e.g., weights storage), generalizability (e.g., on unknown words), etc. Choose any two dimensions and present your views.

In terms of performance on the entity F1 metric, the FFNN performed around the same as both the HMM and MEMM models. The FFNN had an F1 score of 0.41-0.42 while the HMM and MEMM models were both around 0.40-0.41. However, with regards to training time, the FFNN took more time to train than the HMM/MEMM models. Both the single layer FFNN and two layer FFNN took around 6 minutes to train for 10 epochs, while the HMM took merely 0.5 seconds to train, and the MEMM took around 3.5 minutes to train. It is also noted that we used Google Colab with dedicated GPUs to train the network, and it would probably be even slower on a local machine employing a CPU. Finally, we note that the FFNN clearly is less memory efficient than the HMM/MEMM models. We see that printing the parameters of the model that we have around 6.67 million parameters for the single layer FFNN and 6.93 million for the two layer FFNN, whereas for the HMM there were really only 3 trained parameters (the transition, emission, and start state probability matrices) and for MEMMs we just had the limited number of features we chose (7) as the trainable parameters.

Q4.3.2: We assume that you experimented with some (if not all) hyperparameters. Can you comment on some patterns you observed in hyperparameter tuning—e.g., variations in performance with batch size, number of layers, hidden dimension, embedding dimension, etc. Choose any two hyperparameters.

The hyperparameters that we experimented with were the batch size, number of training epochs, and the embedding dimension. For the batch size and number of epochs, we tested on 4 pairs of `(batch_size, num_epochs)` to see how changing these hyperparameters affected performance. For batch size, we hypothesized that increasing batch size would improve training time, as this would allow the number of batches to be trained to decrease, and thus decrease training time. For training epochs, we hypothesized that clearly training time would take longer, but also that F1 entity performance would be at least marginally better, given that convergence isn't absurdly fast, and that intuitively, more epochs gives the training more "chances" to perform better. Finally, for the increased embedding dimension, we hypothesized that training time would increase as well, but we weren't confident enough to conclude anything about the other variations of performance. We ran our experiments on the single layer FFNN model, and a table of our experiments and the results are shown below:

Batch Size	# of Epochs	Embedding Dimension	Training Time	Best Entity F1 (Validation)
128	10	300	6 min	0.398
128	5	300	3 min	0.3968
64	10	300	7 min	0.393
64	5	300	4 min	0.393
128	10	600	7 min	0.397

We can see from the table that the number of epochs decreases with the training time, but does not necessarily mean that the entity F1 is lower. Furthermore, with a decreased batch size, we have longer training time, and given the experiments we did, even worse entity F1 score. For the test where we doubled the embedding dimension, we had longer training time, and a marginally worse entity F1 score as compared to its corresponding experiment with the original embedding dimension.

Q4.3.3: We provide functionality to visualize activations of the modules of your trained FFNN. See how activations for named-entities vary (when compared to non named-entities) as you pass through the layers of a multi-layered FFNN. Do they get sparser as you get deeper into the network? Or is it the other way around? Maybe there's no specific pattern? Don't look for one example where some named entity has a specific pattern and base your answer on that; look for any interesting and general patterns.

The activations for named-entities seem to get denser as you get deeper into the network. Looking at the visualizations for `W` vs. `hid_to_hid[0]` (the tranform from first to second hidden layer), we see that the activation for the `hid_to_hid[0]` layer in general is much more dense overall than the activation for the `W` layer. We also notice that in both layers, the activations for the tokens whose tags are predicted correctly are sparser than the activations corresponding to the other tokens.

Q5.3.1: In comparison to your best FFNN model, how did your best RNN model *perform*? Again, performance is more than just "performance on some metric"; it includes efficiency (e.g., training time, convergence rate), memory (e.g., weights storage), generalizability (e.g., on unknown words), etc. Choose any two dimensions and present your views.

Both of our RNN models had a significantly higher entity F1 score than our best FFNN model, which means that it performed better in terms of purely the entity F1 metric. In terms of memory, the RNN is less memory efficient than the FFNN because it also has to store the parameters for the U matrix; we see that the RNN has a larger number of parameters than the corresponding FFNN with the same number of layers: the single layer RNN had 6.93 million trainable parameters, and the two layer RNN had 7.46 million trainable parameters. Training time is also worse than the FFNN: the FFNN has a training time of 6 minutes for both single and two layers with 10 epochs while the RNN with 10 epochs has a longer training time of 13 minutes for single layer RNN, and 16 minutes for two layer RNN.

Q5.3.2: From the (averaged) entity-level F1 score, we realize that RNN is far better performing than an FFNN (for the underlying task). Ignoring the training time, what happens if we used an RNN to process a really long sequence, say O(2^12) tokens (most large language models operate at this order)?

If we used the RNN to process a really long sequence, then we believe that the performance difference between the FFNN and the RNN would not be as drastic. As the RNN gets farther into its training layers, the information from the tokens from the beginning of the sequence gets lost, as we stack on computations of transformations, and the end of the sequence will be most represented in the output. Thus, this takes away from the entire point of the RNN, making it much more similar to the FFNN.

Q5.3.3: Consider the word "Bank" in two sequences: 1) "*I went to <u>Bank</u> of America to talk to the manager*", and 2) "*I went to <u>Bank</u> to draw cash*". Would our current RNN model be able to accurately classify "Bank" in sequence-1 as "B-LOC" and the "Bank" in sequence-2 as "O"? If your answer is *yes*, then justify your answer; if your answer is *no*, then provide a suitable fix.

Our answer is no. We believe that our current RNN model would output the same answer for both sequences; this means that one of the sequences would be predicted right while the other would be predicted wrong as they would have the same prediction. This is because our RNN model will only take previous words into consideration when processing the sequence up to the word "Bank", and we note that the two sequences are identical up to the point, so the model's prediction for "Bank" should be identical as well. One way we could fix this is by using a bidirectional RNN, as this would allow the model to capture information in the sequence that occurs after the word "Bank".