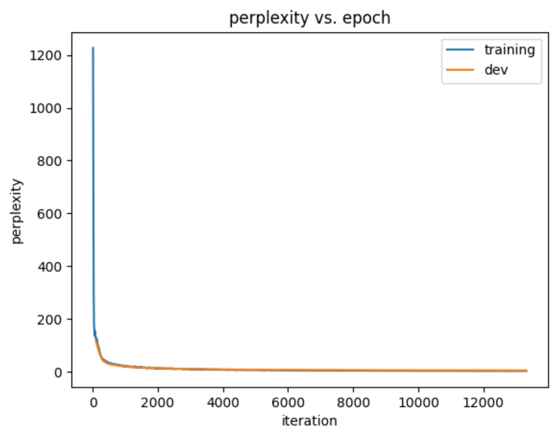

Figure 1: Training Iterations vs. (Dev, Training) Perplexity

Q1

Part 1

See Code.

Part 2

See Code.

Part 3

See Code.

Part 4

French: 1308 tokens
English: 1279 tokens
Training Samples: 8701 sentences between en/fr
Batch Size: 32
Batches/Epoch: 272
Epochs: our model ran for 50 epochs and did not early stop.

Training perplexity is reported every 320 samples or 10 batches. Dev perplexity is reported every 3200 samples or 100 batches. It seems both training and dev perplexity do fall although slower over time. See Figure 1. It seems the gains after a few dozen iteration sets are not very significant.

The translation of the dev set from english to french has a BLEU score of 39.32 (higher the better). The quality of the translation is rather poor, but the fluency seems to be okay on the dev set. There seems to be significant errors for words that have been Byte Pair Encoded (BPE .e.g., @@ words).

Q2

Part 1

The BLEU score with different decoder (128) and encoder (64) hidden layer sizes is 34.17.

Part 2

The BLEU score for additive attention is 42.00.

The BLEU score for dot-product attention is 38.89.

Part 3

Architectural changes include stacking more layers into the LSTM. There are also many other forms of attention. Also maybe ReLu-ing some of the hidden layer projections (not sure intuitively why this helps, but yolo). And I guess you can always go transformers!

In terms of improvements to speed, the early stopping probably can be improved along with a better set of hyper-parameter search. Also another mechanism is to optimize using a difference kind of loss (dunno what off the top of my head) or another dev metric other than perplexity.

The training algorithm iterates until either early stopping or until max epoch. It computes loss first by summing the loss along the sequence dimension (returned by model.forward()) in var *example_losses*. Then it is batch summed into a cumulative loss that is then divided by the batch size to get the average cumulative loss across the sentences, which then backpropagation occurs so the update weights is an average across the batch. Avg Loss is the loss across batches and Avg Perplexity is the exponential of that loss. Early stopping is checked every var *rgs.valid_-niter* times, where the metric perplexity is used where the cumulative loss across all dev words is divided by the number of words to translate. If a better model based on the metric is found it is saved to model.bin and model.bin.optim and patience count reset. Otherwise patience count is incremented for this dev evaluation. If patience count hits a limit, then this trial path is given up on if no improvement was found and num_trials incremented. If num_trials hit a args.max_num_trials then early stopping is triggered. If a trial path is given up on, learning rate decay is triggered on the optimizer and the model is reloaded from the best found at the start when patience=0.

Decode loads the model from the saved model.bin and model.bin.optim parameters along with the test target and src sentences and vocab. It calls beam search and extracts the top most likely hypotheses sentence to compare to the target sentence. It uses the nltk corpus_bleu function to calculus a BLEU score form 0-100 with lower the better.

The beam search decoding algorithm can be found in *nmt_model.py* beam_search function. In this function it uses the trained model parameters for the encoding and embedding layer to first create the inputs to the decoder. Then the beam search takes it step by step, creating beam search width number of sentences each time. A sentence ends either via '</s>' token or hitting the maximum number of tokens allowed. The current set of completed hypothesis sentences are kept in the var *completed_hypothesis* and the log_prob score of such a sentence is tracked in var *hyp_scores* while the current sentences are tracked in list[list[]] var *hypothesis*. In calculating the top K next words, the current scores in *hyp_scores* is expanded to the length of the vocab and the output log probabilities of the vocab is then broadcast-added to the *hyp_scores* from which the top K words is found in the matrix (K x words so far) using torch.topk(...). This is why the next operations are an integer division to figure out the root index of the sentence so far and a remainder operation to figure out the new word. The next hidden and cell states after the initial '<s>' token is a batch (K x hidden_size) which is fished out by the var *live_hyp_ids* (which of the K hidden states from decoder step is the topk choice to use as *prev_hyp_id* in the next iteration).

Q3

Part 1

A

The downside of maximizing over $\Sigma_{(w,c)\in D} log(\frac{e^{v_c \cdot u_w}}{\Sigma_{c^{'} \in V} e^{v_{c^{'}} \cdot u_w}})$ is that the corpus V can be really large and this can be computationally expensive as the denominator is over all V rather than just select words N. Also it feels as if this loss function favours overfitting of the word embeddings.

B

Yes, ideally the more likely words that appear more often that do not appear in the context would be more likely to be chosen. If an uniform distribution is applied, some more likely words would get closer to the target word as it becomes less likely to be chosen as negative samples and words that appear less will be further away from the target word as it would be more likely to be chosen as negative samples . It feels like this would not be as a representative language model (underfitting).

Part 2

The larger the context window, the more costly the computation as there are more context words to optimize over to update the hidden vector. But more importantly, a context window of 1 basically means that there is no context. A context window of 5 means we understand the features of a sentence and context window of 100 means we may extrapolate the features of a paragraph.

Part 3

With this smoothing distribution, the unigram frequency distribution becomes more uniform for $\alpha < 0$ and less uniform for $alpha > 0$. A more uniform distribution leads to underfitting and is a form of regularization. The similarity with most words will decrease as the rare words would be more likely to be chosen as negative samples to optimize away from then the words they are in the context for which would optimize them close to some other words (there are more words not in rare words' context, than there are words in rare words' context).

Part 4

The similarity can be rewritten as:

$$cos(y, b - a + x) = cos(y, b) - cos(y, a) + cos(y, x) \tag{1}$$

It may be that $cos(y, b) - cos(y, a) \approx 0$ as London and England occur pretty simultaneously in the same context and with only Baghdad as the input, then similar words that co-occur in similar context may not be necessarily Iraq but other Iraqi cities.