Instructions: This assignment is intended to introduce you to hybrid DNN-HMM models and End-to-End Recurrent Neural Network (RNN) models for speech recognition, and explore the performance differences between the two. Using a GPU is recommended for the hybrid DNN-HMM part. If you don't have access to one, let the TA know at ankitap@ttic.edu so that we can set-up your account on the TTIC slurm cluster. Please ask for help if you feel stuck!
Please submit answers to all underlined questions below in the writeup as a PDF file via the course Canvas site. The tex file used to create this PDF has also been provided so that you can enter your answers directly into this document. Additionally, please run through and execute all of the cells of the ipython notebook (including the parts you added which are marked by `# TODO`) and submit the notebook via Canvas.
Lateness policy: Late submissions submitted up to Thursday 5/28/20 7:00 pm (the "late deadline") will receive a 20% reduction in credit. Submissions that are later than this will receive some non-zero credit, but we cannot guarantee how much. We cannot guarantee that we will carefully grade or give feedback on submissions after the "late deadline". In addition, you have four free "late days" that you can use throughout the term to extend homework (not project) deadlines without penalty. If you are using any of your "late days" for this assignment, mention that in the comment while submitting this assignment. You should state how many "late days" you are using and how many remaining "late days" you have. The number of late days used for any given homework must be an integer.
Collaboration: You are encouraged to discuss assignments and any aspect of the course material with others, but any material you submit (writeup, code, figures) should be produced on your own.

*** Before starting this homework, you'll need to make sure you have python3 and the following packages on your system: `numpy, scipy, jupyter, matplotlib, editdistance, pytorch`. You may use any installer of your choice. Alternatively, you an follow the steps from HW2 to create a python3 environment and install the required packages using Miniconda. You will need some additional packages that you can install using the following commands within your python environment.

(a) `pip install editdistance` to compute the word error rate (WER) for evaluation.

(b) Go to https://pytorch.org/getstarted/previousversions/#v101 and follow instructions to download pytorch 1.0.1 on whichever platform you're using.

(c) The zip-file from canvas includes the required data, library files, and two jupyter notebooks that you will be editing.

Once everything is set up, you can open a notebook with the command: `jupyter notebook hw4_hybrid.ipynb` and `jupyter notebook hw4_rnn.ipynb`. Make sure that the kernel is set to Python 3 in the notebook. Please let us know if you need help!

0. Please fill out this questionnaire. This is purely to help us calibrate assignment load and let us know what may need to be made clearer in class. Your responses will receive a small amount of credit, independent of the actual answers.

(i) Did you collaborate on this assignment, and if so, with whom? **No.**

(ii) Approximately how many hours did this assignment take to complete? **15 hours excluding training time. The TTIC slurm queue was pretty long.**

(iii) On a scale of 1 to 5 (where 1 = trivial and 5 = impossible), what was the difficulty of this assignment? **4.**

(iv) On a scale of 1 to 5 (where 1 = useless and 5 = essential), how useful has this assignment been to your understanding of the material? **5. The two coding problems made what we learned a lot more concrete.**

1. Paper exercises

   (a) Show that count$(w_{i-1}w_i)$/count(w_{i-1}) is indeed the maximum-likelihood estimate of the bigram probability $p(w_i|w_{i-1})$. (Hint: Think Lagrange multipliers. If you would like to start with a warm-up exercise, consider tossing a biased coin T times, and show that the maximum-likelihood estimate of the probability of heads is count(heads)/T. Then answer the original question about bigrams.)

   Answer:
   For the ease of notation, define count(w_kw_l) as $c(w_kw_l)$. Reindex w_{i-1} and w_i as w_k and w_l.
   WTS:

$$MLE : \hat{p}(w_l|w_k) = \frac{c(w_kw_l)}{\sum_{j=1}^{n} c(w_kw_j)}$$

   When computing the likelihood, we treat the probability of the first word (which doesn't have any previous word to condition on) as a constant.

$$L(\mathbf{w}) \propto \prod_{i=1}^{n}\prod_{j=1}^{n} p(w_j|w_i)^{c(w_iw_j)}$$
$$\log L(\mathbf{w}) \propto \sum_{i=1}^{n}\sum_{j=1}^{n} c(w_iw_j)\log p(w_j|w_i)$$

   We have the constraint that when conditioned on a fixed word w_i, the probabilities of all words in the corpus should sum up to 1.

$$\sum_{j=1}^{n} p(w_j|w_i) = 1$$

   We define the following function to be used with Lagrange multipliers $\lambda_i, i = 1, ..., n$:

$$g(w_i) = -1 + \sum_{j=1}^{n} p(w_j|w_i)$$

   The following Lagragian is the function that we seek to maximize:

$$\begin{aligned}\mathcal{L} &= \log L(\mathbf{w}) + \sum_{i=1}^{n}\lambda_i g(w_i) \\ &= \sum_{i=1}^{n}\sum_{j=1}^{n} c(w_iw_j)\log p(w_j|w_i) + \sum_{i=1}^{n}\lambda_i\Big[-1 + \sum_{j=1}^{n} p(w_j|w_i)\Big] \\ &\propto \sum_{i=1}^{n}\sum_{j=1}^{n} c(w_iw_j)\log p(w_j|w_i) + \sum_{i=1}^{n}\lambda_i\sum_{j=1}^{n} p(w_j|w_i)\end{aligned}$$

   Take the derivative with respect to $p(w_l|w_k)$:

$$\frac{\partial\mathcal{L}}{\partial p(w_l|w_k)} = \frac{c(w_kw_l)}{p(w_l|w_k)} + \lambda_k = 0 \implies p(w_l|w_k) = -\frac{c(w_kw_l)}{\lambda_k}$$

   Plug this back into the sum-to-one constraint:

$$\sum_{j=1}^{n} p(w_j|w_i) = \sum_{j=1}^{n} -\frac{c(w_iw_j)}{\lambda_i} = -\frac{1}{\lambda_i}\sum_{j=1}^{n} c(w_iw_j) = 1 \implies \lambda_i = -\sum_{j=1}^{n} c(w_iw_j)$$

   Plug this back into the expression of $p(w_l|w_k)$ to complete the proof.

$$p(w_l|w_k) = -\frac{c(w_kw_l)}{\lambda_k} = \frac{c(w_kw_l)}{\sum_{j=1}^{n} c(w_kw_j)}$$

(b) In lecture, we discussed the "fundamental equation of speech recognition",
$\mathbf{w}^* = \text{argmax}_{\mathbf{w}}\, p(\mathbf{O}|\mathbf{w})p(\mathbf{w}) = \text{argmax}_{\mathbf{w}} \log p(\mathbf{O}|\mathbf{w}) + \log p(\mathbf{w})$.
In practice, we do not use this equation precisely but rather,
$\mathbf{w}^* = \text{argmax}_{\mathbf{w}} \log p(\mathbf{O}|\mathbf{w}) + \alpha \log p(\mathbf{w}) + \lambda|\mathbf{w}|$
where, α is the language model scaling factor, λ is an "insertion penalty", and $|\mathbf{w}|$ is the number of words in the hypothesized sequence. Explain why we might want to use the insertion penalty, whether you would expect a good value of λ to be positive or negative, and why. (A warm-up exercise if needed: if the language model is uniform, i.e. $p(w_i|w_1, \ldots, w_{i-1}) = 1/|V|$ where $|V|$ is the size of the vocabulary, then do you think λ should be positive or negative? Based on your answer to this, try to reason about the general case with a typical arbitrary language model. Also note: "Insertion penalty" is a bit of a misnomer, so don't get hung up on the term.)

Answer:
We want to use the insertion penalty so as to encourage a longer hypothesized sequence and to give the language model more say in so that it matches up to the acoustic model. We'd use a positive λ for this. Since we are multiplying probabilities in $p(\mathbf{w})$, the value gets smaller as $|\mathbf{w}|$ increases. This will lead to a lower likelihood. To make up for this decreased likelihood, we add back some likelihood that is proportional to $|\mathbf{w}|$, the size of the hypothesized sequence.

Submission files for problem 2 and 3: best.pkl for RNN, dnn_model.pkl and log_emission_0.0.npz for DNN; hw4_hybrid_dropout.ipynb, hw4_hybrid_nlayers.ipynb, hw4_rnn.ipynb

2. Digit sequence recognition using hybrid HMM/DNN models:
Note: To parallelize training, I used different notebook files for different sets of experiments. They share the same core code. Please refer to different submissions with names: hw4_hybrid_dropout.ipynb, hw4_hybrid_nlayers.ipynb

   For this part you will use a hybrid HMM/DNN model to do digit sequence recognition. The evaluation will be measured via word error rate (WER), where each word is a digit between 0 and 9, and where 0 is pronounced 'oh' and not 'zero'.

   (a) You have been provided a trained HMM/GMM system for digit sequence recognition. The default setup provided gives a word error rate of 24.5%. You are not expected to train or tune the model in this part of the assignment. You will use this model only for generating forced alignments as ground-truth frame labels for the neural network classifier.

   (b) Next, finish the forced alignment code (marked by `# TODO`) to get the decoded state sequences. The generated state sequences will be saved in the hybrid/data/ directory as a `.npz` file.
   Done. Please refer to any of the notebook files.

   (c) Using the state labels you have generated, train a neural network frame classifier. Create a setup for tuning hyperparameters within the ipynb (see the area marked by `# TODO`). This can take a variety of forms, but it just needs to be a setup that allows you to train models on the training set and tune hyperparameters by evaluating models, during training, on the development set. The development set evaluation criterion for this part is the neural network classification accuracy.
   Done. Please refer to any of the notebook files.

   (d) Experiment with the feedforward model by changing one or more of the following hyperparameters: number of layers, hidden layer dimensionality, input context size, parameter initialization, optimizer and related hyperparameters, batch size, and temperature parameter for the prior used for normalizing as follows

   $$\texttt{log\_emission} = \texttt{log\_posterior} - \texttt{(temp\_parameter)*log\_prior}$$

   (e) Report the following:

   - Plots of the loss and accuracy vs. epoch on train and dev
   - Any plot/table of your choice for the set of tuning experiments you performed on the dev set. This plot/table should contain enough information to show how one would choose the best model given your tuning experiments. Note that you are tuning here to get the best frame classification accuracy and not the word error rate.

Figure 1: Plot of loss and accuracy vs. epoch on train and dev

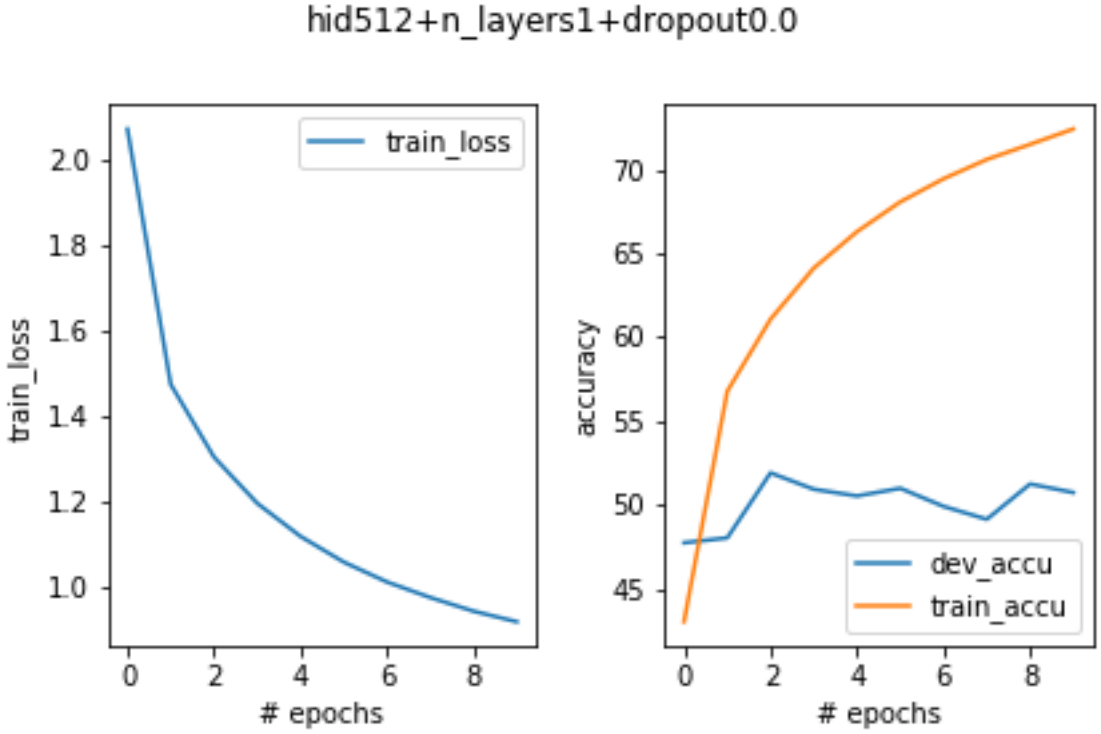


Figure 2: Plot of best accuracy and loss when dropout is 0.3, 0.5, 0.8

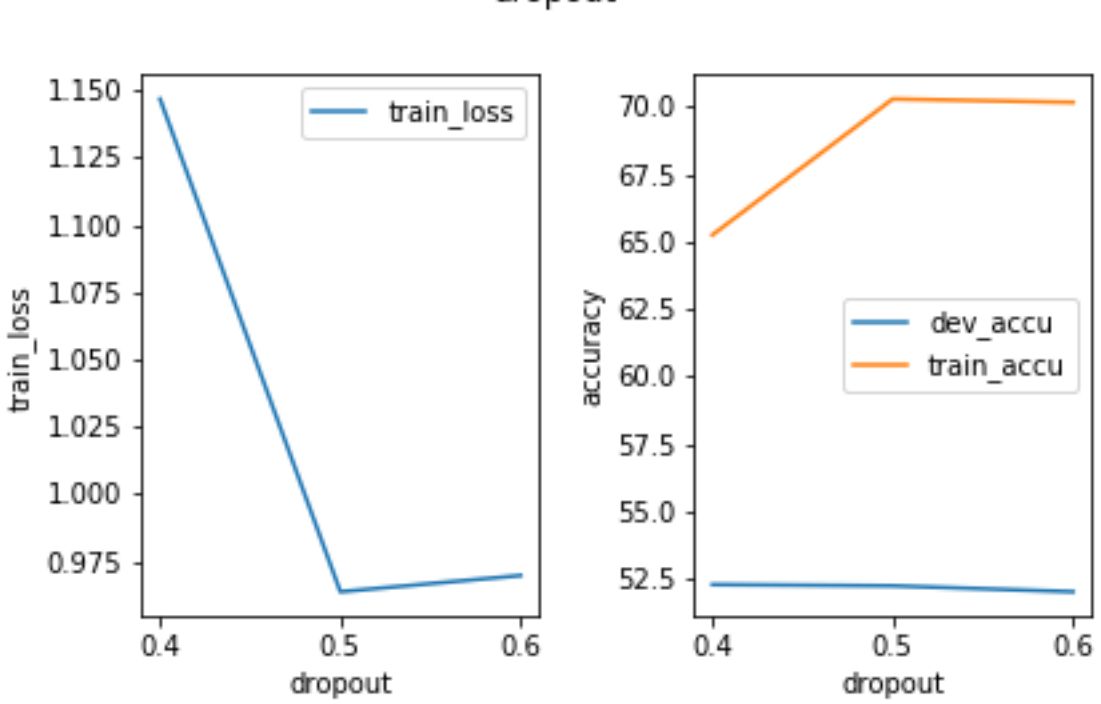
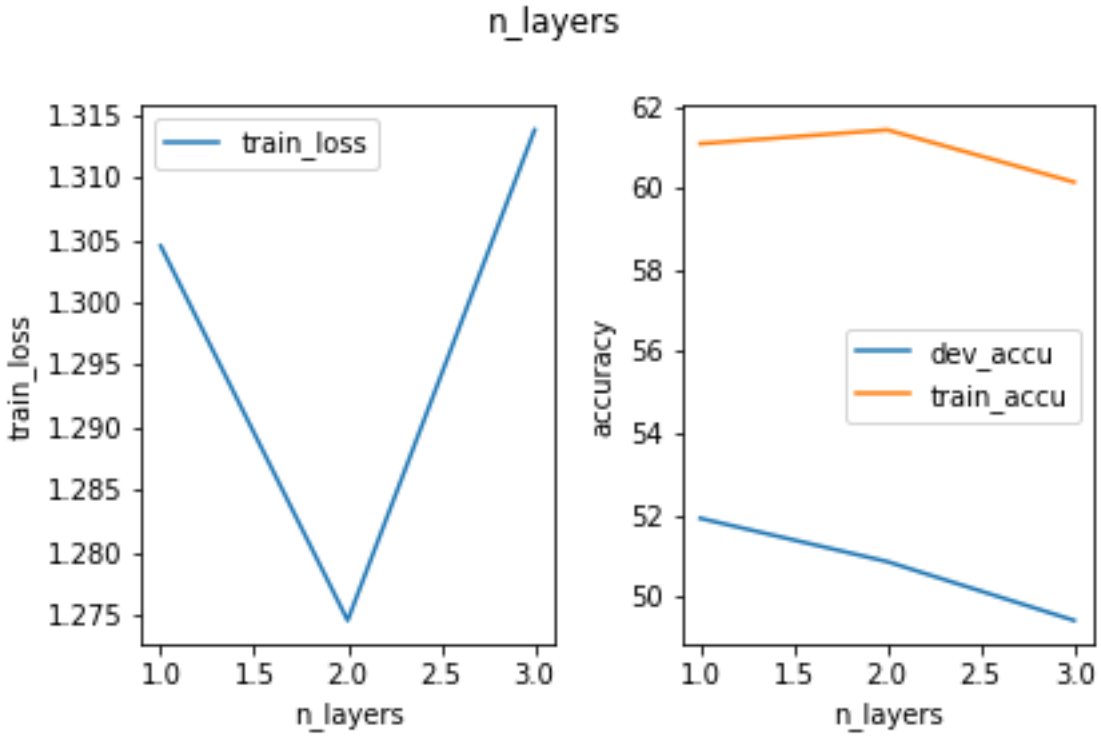


Figure 3: Plot of best accuracy and loss when n_layers is 1, 2, 3 (dropout fixed at 0.5)



(f) Once you have the best model saved, save the state posterior probabilities and use them to decode the test set with the tuned HMM/DNN. You must include the state posterior probabilities (a npz file saved by default at hybrid/data/log_emission) for your best model as a part of your submission.
Submitted the unnormalized one, log_emission_0.0.npz

(g) Report the WER you obtain on the test set, using the model found to be best by tuning on dev, and describe in 1-2 paragraphs how you chose the best hyperparameter setting based on dev set performance, results you obtained, and any other comments about your experiments. *(Note: Unlike in previous assignments, we are now tuning on dev and evaluating on test, i.e. we are no longer cheating. To stay honest to this setting you must evaluate on the test set only once using the model that gave you the best error rate on the development set).*

Reflection
I used the model that achieves the highest dev accuracy of 52.35% as my best model, which gives me a WER of **18.64%** on the test set. The model configuration is as follows: 512 hidden units per layer, 1 hidden layer, and a dropout probability of 0.5. The model uses normalized log_emission (setting temperature parameter to 1.0). Please refer to **hw4_hybrid_dropout.ipynb**
In the first figure, we observe that while training loss is strictly decreasing and training accuracy strictly increasing, the plot of dev accuracy is in a zig-zag fashion. What's interesting is that dev accuracy is actually higher in the first two epochs than in later epochs where the model likely overfits. The sharp decrease in training loss from epoch 0 to 2 also shows that the model has learned substantially during these initial epochs. In my hyperparameter tuning experiments, I experimented with dropout probabilities of 0.3, 0.5, 0.8; and 1, 2, or, 3 hidden layers. In the dropout experiment, although a dropout rate of 0.5 results in higher training loss and lower training accuracy than dropout rates of 0.3 or 0.8, the accuracy on the dev set turns out to be better. Examining the training log print-outs in the notebook file shows some correlation between dropout rates and the best epoch number, as briefly summarized in the table below.

dropout	**0**	**0.3**	**0.5**	**0.8**
best epoch	2	4	2	5
best dev accuracy	51.93	51.71	52.35	51.89

It seems that in general, as dropout probability increases, the model will achieve its best dev accuracy at a

later epoch. This could be because leaving out nodes essentially temporarily reduces the number of hidden units, and the model needs more epochs to learn before it achieves the best dev accuracy. A model achieving its best dev accuracy in the second epoch out of all 10 epochs may indicate that the later 8 epochs are overfitting the training set. In that regard, dropping out could be an effective measure against overfitting.

Surprisingly, in my experiment with 1, 2, and 3 hidden layers (while fixing dropout at 0.5), I found that dev accuracy decreases with increasing numbers of layers. This doesn't align with my intuition that a deeper network is usually better. The dropout rate could have some effect on this, but I didn't have enough computing resource to conduct another experiment. A tentative explanation to this phenomenon could be that the task of digit sequence recognition is relatively simple, and a deeper network tends to overfit the train set way too soon.

3. Digit sequence recognition using an RNN encoder-decoder model

   For this part you will use an RNN encoder-decoder model to do digit sequence recognition. The evaluation will be measured by word error rate (WER) where each word is a digit between 0-9 (where 0 is pronounced 'oh' and not 'zero').

   (a) Create a setup for tuning hyperparameters within the ipynb (see the area marked by `# TODO`).
   Done. Please refer to the notebook file for RNN. This can take a variety of forms, but it just needs to be a setup that allows you to train models on the training set and tune hyperparameters by evaluating models, during training, on the development set. This time the tuning criterion is the WER on the development set.

   (b) Experiment with RNN digit sequence recognition to see if you can improve the WER on the **dev** (tuning) set.
   Answered in details below.

   (c) End-to-end RNN-based ASR models typically use an attention mechanism to focus on the parts of the acoustic signal that are relevant for each predicted output. The default (and one of the most basic) attention mechanism is given by

$$\mathbf{a}_t = Softmax(\mathbf{H}\mathbf{d}_t)$$
$$\mathbf{c}_t = \sum_{i=1}^{T} a_{it}\mathbf{h}_i$$

   where $\mathbf{H}$ is the matrix given by the hidden output of the encoder with shape T x n, where T is the number of total input frames and n is the encoder and decoder hidden state dimensionality; $\mathbf{d}_t$ is the hidden state vector of the decoder at decoding step t; and $\mathbf{c}_t$ is the resulting context vector at time t.

   You will implement another common attention mechanism given by

$$u_{it} = \mathbf{v}^T \tanh(\mathbf{W}_1\mathbf{h}_i + \mathbf{W}_2\mathbf{d}_t)$$
$$\mathbf{a}_t = Softmax(\mathbf{u}_t)$$
$$c_t = \sum_{i=1}^{T} a_{it}h_i$$

   where $\mathbf{v}$, $\mathbf{W}_1$, and $\mathbf{W}_2$ are additional learned parameters. (Note: here the encoder and decoder state sizes do not need to be equal.) Implement this attention mechanism where indicated by `# TODO` in the code.
   Done. Please refer to the notebook for RNN.

   (d) Optionally, try changing other hyperparamters (see `config.json`):

   - encoder hidden_size
   - decoder input_size
   - num_layers
   - learning_rate
   - batch_size
   - dropout (applied between LSTM layers)
   - sampling_prob (for scheduled sampling) *Note: scheduled sampling has not been covered in class but it refers to the decoding done during training. At each time step, a new prediction is made (in this case a digit in 0-9). With probability* `sample_prob`*, the previous prediction is fed as the input to the next step of the decoder; otherwise the ground truth is used. If* `sample_prob` *is set to 0, then the ground truth is always fed as input at each time step.*
   - or any other aspect that you find interesting to experiment with!

   (e) Report 3 plots:

   - Plot the loss and word error rate (WER) vs. epoch on train and dev, for one of the two attention models above.
   - For 3-5 models trained with a range of scheduled sampling rates, plot loss and WER vs. sampling probability.
   - Come up with your own! For example, you could plot the loss and WER vs. time for models trained with different optimizers or learning rates. Choose whatever is most interesting to you to investigate.

Figure 4: Plot of the loss and WER vs. epoch for the default attention mechanism

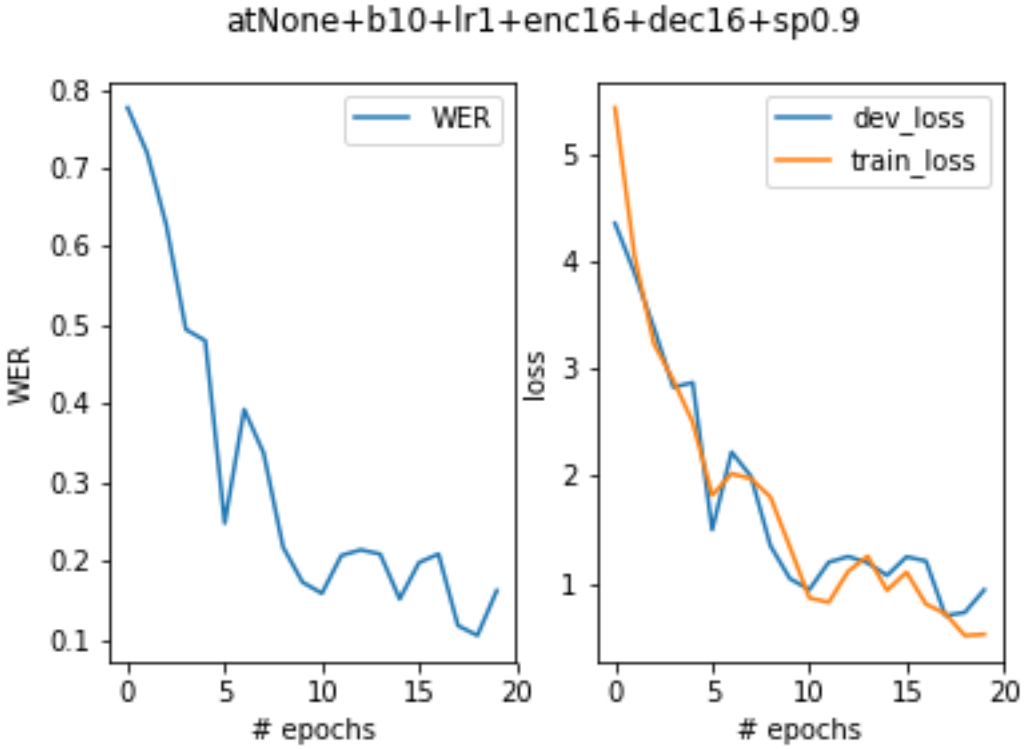


Figure 5: Plot of the best WER and loss vs. sampling probabilities of 0.3, 0.6, 0.9

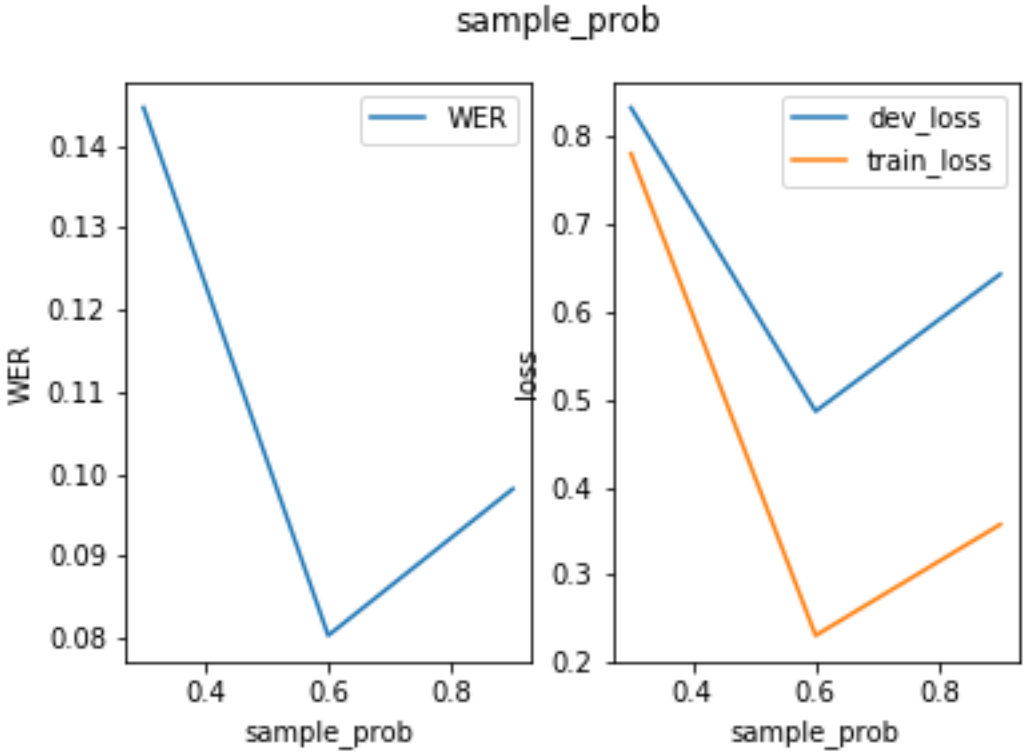
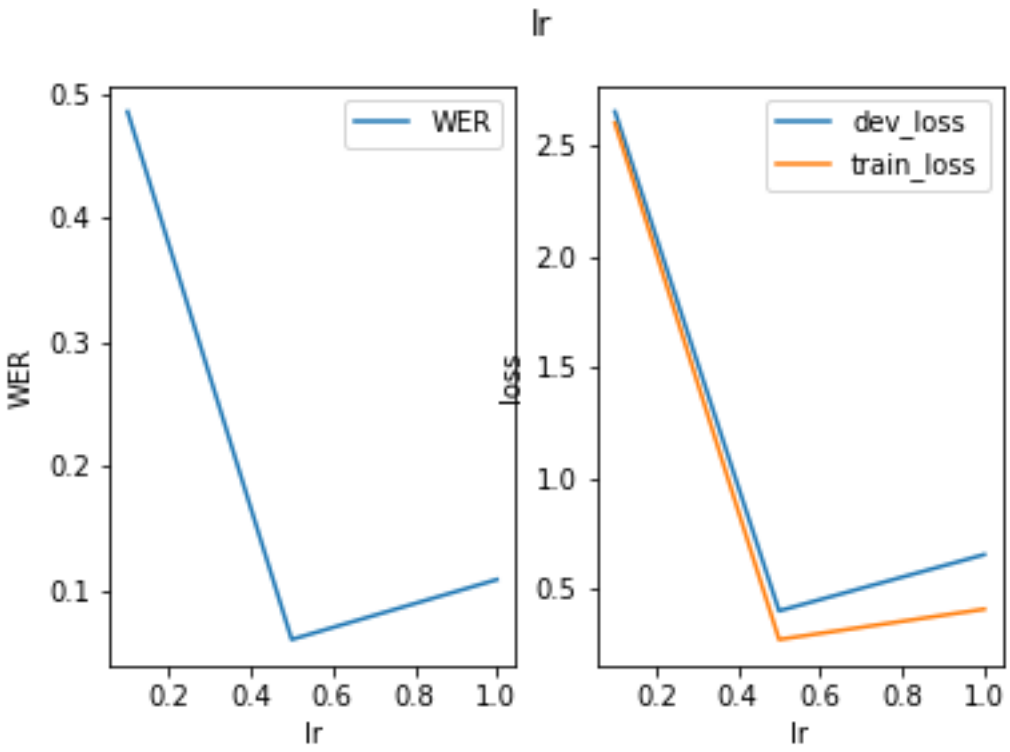


Figure 6: Plot of the best WER and loss vs. learning rates of 0.1, 0.5, and 1.0



(f) Describe in 1-2 paragraphs what you did, results you obtained, how did you choose the best hyperparameter setting, and any additional comments about your experiments (e.g. the trade-off between performance and efficiency, number of iterations required, types of errors you observed, ...).

Reflection

In addition to experimenting with sampling probabilities per the instructions, I also experimented with different learning rates.

As the first figure indicates, training loss, dev loss, and dev WER follow an overall decreasing trend, but experience some bursts of increase over the 20 epochs. This might be a sign of overfitting, which led me to experiment with a slightly lower learning rate as described below.

In my experiment with sampling probabilities of 0.3, 0.6, and 0.9, I didn't observe a direct correlation between the magnitude of sampling rate and performance. A sampling probability of 0.3 results in a much higher best WER than the other two, which may indicate that feeding the model ground truth too frequently isn't as constructive to its learning as feeding it the previous prediction at times. The WER and loss differences between a sampling probability of 0.6 vs. 0.9 don't appear to be too significant. However, a 0.6 sampling probability might still be better since the model will see a balanced mixture of ground truth and its previous predictions, as opposed to seeing only one kind of the two. This should create a good balance between training performance and efficiency. (In a previous course, I've trained an LSTM to predict the next sentence given the previous. I only fed in previous predictions, which corresponds to a sampling probability of 1 here. Consequently, the model learned extremely slowly and took many epochs to train.)

In my experiment with different learning rates, I observed that a learning rate of 0.5 led to the best-performing model. When the learning rate is as low as 0.1, the model learns slowly and the dev WER is almost strictly decreasing (at a slow rate) throughout the 20 epochs and ended on a WER of 48.57%. This indicates that a learning rate this low is too conservative and not too efficient in training. On the other hand, a learning rate as great as 1.0 sometimes results in oscillating dev WER between different epochs. This indicates that we may be taking too big a step when descending the gradients. A moderate learning rate of 0.5 appeared to offer a more steady yet still efficient decrease in WER during training.

Regarding the temperature parameter, the unnormalized (temperature 0.0) posteriors always outperformed the normalized one (temperature 1.0). A temperature parameter of 0.5 results in a WER in-between the two.

temperature	**N/A, baseline**	**1.0, normalized**	**0.5**	**0.0, unnormalized**
WER (%)	21.36	20.91	20.00	18.64

Some additional comments about my experiments: I found that training 20 epochs is about enough. Training more epochs than that doesn't contribute significantly to improving dev WER and may easily result in overfitting.

(g) Finally, report results on the test set for only the best model (found by tuning on dev). You must submit your best model file (saved by default as rnn/checkpoints/best) as a part of your submission. *(The same note applies here, regarding testing only once on the test set, as for Question 2(g).)*

Result

I used the model that achieves the lowest dev WER of 7.14% as my best model, which gives me a pretty decent WER of **4.52%** on the test set. The model configuration is as follows: default attention, a learning rate of 0.5, encoder and decoder hidden sizes of 16, a sampling probability of 0.9, training for 20 epochs.

(h) (EXTRA CREDIT, 5 points) There will be extra credit for the best final test set performance.