Advanced Methods in NLP Lecturer: Jonathan Berant

Home Assignment 2: Language Models

Due Date: *11:00pm, April 30, 2019*

In this home assignment we will implement an n-gram, a neural n-gram language model and an RNN language model.
Download the data and supporting code from the following link: `https://drive.google.com/file/d/1zln3xNJhICDxX3hJeWYbSVAacMSxlNWr/view?usp=sharing`. Note, you will need to re-use some of the code implemented for assignment1 - `gradcheck.py`, `sgd.py`, `sigmoid.py` and `softmax.py`.
Submit your solution through Moodle, create a zip file named `<id1>_<id2>_<id3>.zip` (where id1 refers to the ID of the first student). This zip file for the submission should include all the .py files to run the code, the .ipynb file and the .npy file for the saved parameters, as well as a written solution named answers.pdf. Only one student needs to submit.
Your code should run properly on Python 2.7. Your code will be tested on the School of Computer Science operating system, installed on `nova` and other similar machines. Please make sure your code runs there. If your code does not run on `nova`, you will lose points.

1 N-gram language model

(a) Implement a trigram language model with linear interpolation. Fill your implementation in `ngram_model.py`. Your implementation should include the training algorithm and model evaluation function. Use `data/lm/ptb-train.txt` to train the n-gram models. Use `data/lm/ptb-dev.txt` to evaluate your perplexity.

(b) Implement grid search to tune the linear interpolation coefficients (λ_i). Find the perplexity for every setting of the coefficients and the setting that maximizes perplexity on the dev set. Add the results of this search to your written solution. Make sure to evaluate with $\lambda_1 = 0$ and $\lambda_1 = \lambda_2 = 0, \lambda_3 = 1$, which will effectively be training bigram and unigram language models. [1]

2 Neural Bigram language model

(a) Derive the gradient with regard to the inputs of a softmax function when cross entropy loss is used for evaluation, i.e., find the gradients with respect to the softmax input vector $\boldsymbol{\theta}$, when the prediction is made by $\hat{\boldsymbol{y}} = \text{softmax}(\boldsymbol{\theta})$. Cross entropy and softmax

[1]Note: To evalute whether the perplexity values you are getting are correct, consult this paper: `https://static.googleusercontent.com/media/research.google.com/iw//pubs/archive/46183.pdf`.

are defined as:

$$\mathrm{CE}(\boldsymbol{y}, \hat{\boldsymbol{y}}) = -\sum_i y_i \cdot \log(\hat{y}_i)$$

$$, \mathrm{softmax}(\boldsymbol{\theta})_i = \frac{\exp(\theta_i)}{\sum_j \exp(\theta_j)}.$$

The gold vector $\boldsymbol{y}$ is a one-hot vector, and the predicted vector $\hat{\boldsymbol{y}}$ is a probability distribution over the output space.

(b) Derive the gradients with respect to the inputs $\boldsymbol{x}$ in a one-hidden-layer neural network (i.e., find $\frac{\partial J}{\partial \boldsymbol{x}}$, where J is the cross entropy loss $\mathrm{CE}(\boldsymbol{y}, \hat{\boldsymbol{y}})$). The neural network employs a sigmoid activation function for the hidden layer, and a softmax for the output layer. Assume a one-hot label vector $\boldsymbol{y}$ is used. The network is defined as:

$$\boldsymbol{h} = \sigma(\boldsymbol{x}\boldsymbol{W_1} + \boldsymbol{b_1}),$$
$$\hat{\boldsymbol{y}} = \mathrm{softmax}(\boldsymbol{h}\boldsymbol{W_2} + \boldsymbol{b_2}).$$

The dimensions of the vectors and matrices are $\boldsymbol{x} \in \mathbb{R}^{1\times D_x}, \boldsymbol{h} \in \mathbb{R}^{1\times D_h}, \hat{\boldsymbol{y}} \in \mathbb{R}^{1\times D_y}$. The dimensions of the parameters are $\boldsymbol{W_1} \in \mathbb{R}^{D_x\times D_h}, \boldsymbol{W_2} \in \mathbb{R}^{D_h\times D_y}, \boldsymbol{b_1} \in \mathbb{R}^{1\times D_h}, \boldsymbol{b_2} \in \mathbb{R}^{1\times D_y}$.

(c) Implement the forward and backward passes for a neural network with one sigmoid hidden layer. Fill in your implementation in `neural.py`. Sanity check your implementation with `python neural.py`.

(d) Use the neural network to implement a bigram language model in `neural-lm.py`. Use glove word embeddings to represent the vocabulary (`data/lm/vocab.embeddings.glove.txt`). Implement the `lm_wrapper` function, that is used by `sgd` to sample the gradient, and the `eval_neural_lm` function that is used for model evaluation. Add the results to your written solution. Include in your submission the trained parameters.

3 Recurrent Neural Networks language model

In this section, youll implement a recurrent neural network (RNN) used for language modeling and use a GPU to train it. Given a sequence of words (represented as one-hot vectors) $\boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)} \ldots, \boldsymbol{x}^{(t)}$, a RNN language model predicts the next word $\boldsymbol{x}^{(t+1)}$ by modeling: $P(\boldsymbol{x}^{(t+1)} = \boldsymbol{w}_j | \boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)} \ldots, \boldsymbol{x}^{(t)})$ where $\boldsymbol{w}_j$ is a word in the vocabulary. The model is as follows:

$$\boldsymbol{h}^{(t)} = RNN(\boldsymbol{h}^{(t-1)}, \boldsymbol{x}^{(t)}),$$
$$\hat{\boldsymbol{y}}^{(t)} = \mathrm{softmax}(\boldsymbol{h}^{(t)}\boldsymbol{W} + \boldsymbol{b}).$$

Where the RNN can either be a basic RNN, a GRU or an LSTM, $\boldsymbol{W} \in \mathbb{R}^{\boldsymbol{D}_h\times|V|}$, and $\boldsymbol{b} \in \mathbb{R}^{1\times|V|}$.

(a) We will regularize our network using a regularization method called *Dropout*. During training this randomly sets units in the hidden layer $\boldsymbol{h}$ to zero with probability p_{drop} and then multiplies $\boldsymbol{h}$ by a constant scalar γ (dropping different units each minibatch). We can write this as

$$\boldsymbol{h}_{drop} = \gamma \boldsymbol{d} \circ \boldsymbol{h}$$

where $\boldsymbol{d} \in \{0,1\}^{\boldsymbol{D}_h}$ ($\boldsymbol{D}_h$ is the size of $\boldsymbol{h}$) is a mask vector where each entry is 0 with probability p_{drop} and 1 with probability $1 - p_{drop}$. And γ is chosen such that the value of $\boldsymbol{h}_{drop}$ in expectation equals $\boldsymbol{h}$:

$$\mathrm{E}_{p_{drop}}[\boldsymbol{h}_{drop}]_i = h_i$$

for all $0 < i < \boldsymbol{D}_h$. What must γ equal in terms of p_{drop}? Briefly justify your answer.

(b) Complete the implementation from the following notebook `https://colab.research.google.com/drive/1hJjN1Ue8XBNC9Wg-E8FR4w1SG-LvUrZO`. Implement the rest of the RNN model. This will involve implementing the `add_placeholder`, `add_rnn_cell`, `add_embedding_op`, `add_prediction_op` and `add_loss_op` functions. Follow the instructions within the code. Note that you should only modify the parts between "YOUR CODE HERE" and "END YOUR CODE"

(c) Train your model using the main cell in the notebook. Run your code for 20 Epochs with the default hyperparameters. The time to evaluate your code should be around 8 minutes. Make sure you switch your runtime in Google Colab to GPU (Runtime $\rightarrow$ Change runtime type). Include in your submission the perplexity plot you got for each epoch of the training, and the final validation perplexity.

(d) Download the .IPYNB file (File $\rightarrow$ Download .ipynb) and include it within your submission.