Natural Language Processing **Tel Aviv University**

Assignment 2: Language Models

Due Date: *Tuesday, May 30, 2023* Lecturer: Maor Ivgi

1 Word-Level Neural Bi-gram Language Model

(a) Derive the gradient with respect to the input 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 are defined as:

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

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

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 input $\boldsymbol{x}$ in a one-hidden-layer neural network (i.e., find $\frac{\partial J}{\partial \boldsymbol{x}}$, where J is the cross entropy loss $\text{CE}(\boldsymbol{y}, \hat{\boldsymbol{y}})$). 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:

$$\boldsymbol{h} = \sigma(\boldsymbol{x}\boldsymbol{W}_1 + \boldsymbol{b}_1),$$
$$\hat{\boldsymbol{y}} = \text{softmax}(\boldsymbol{h}\boldsymbol{W}_2 + \boldsymbol{b}_2).$$

The dimensions of the vectors and matrices are $\boldsymbol{x} \in \mathbb{R}^{1\times D_x}, \boldsymbol{h} \in \mathbb{R}^{1\times D_h}, \hat{\boldsymbol{y}} \in \mathbb{R}^{1\times D_y}, \boldsymbol{y} \in \mathbb{R}^{1\times D_y}$. 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 `q1c_neural.py`. Sanity check your implementation with `python q1c_neural.py`.

(d) GloVe (Global Vectors) embeddings are a type of word embeddings that represent words as vectors in a high-dimensional space, based on the co-occurrence statistics of words in a corpus. They are related to the skip-gram embeddings you saw in class in that they both aim to capture the semantic and syntactic relationships between words, but GloVe embeddings incorporate global corpus-level information in addition to local context information. In this section you will be using GloVe embeddings to represent the vocabulary. Use the neural network to implement a bigram language model in `q1d_neural_lm.py`. Use GloVe 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. Report the dev perplexity in your written solution. Don't forget to include `saved_params_40000.npy` in your submission zip!

2 Theoretical Inquiry of a Simple RNN Language Model

In this section we will perform a short theoretical analysis of a simple RNN language model, adapted from a paper by Tomas Mikolov, et al.[1]. Formally, for every timestep t, the model is defined as follows:

$$\begin{aligned}
\boldsymbol{e}^{(t)} &= \boldsymbol{x}^{(t)}\boldsymbol{L} \\
\boldsymbol{h}^{(t)} &= \text{sigmoid}\left(\boldsymbol{h}^{(t-1)}\boldsymbol{H} + e^{(t)}\boldsymbol{I} + \boldsymbol{b}_1\right) \\
\hat{\boldsymbol{y}}^{(t)} &= \text{softmax}\left(\boldsymbol{h}^{(t)}\boldsymbol{U} + \boldsymbol{b}_2\right)
\end{aligned} \tag{1}$$

where $\boldsymbol{h}^{(0)} \in \mathbb{R}^{D_h}$ is some initialization vector for the hidden layer and $\boldsymbol{x}^{(t)}\boldsymbol{L}$ is the product of $\boldsymbol{L}$ with the one-hot vector $\boldsymbol{x}^{(t)}$ representing index of the current word. The parameters are:

$$\boldsymbol{L} \in \mathbb{R}^{|V|\times d} \qquad \boldsymbol{H} \in \mathbb{R}^{D_h\times D_h} \qquad \boldsymbol{I} \in \mathbb{R}^{d\times D_h} \qquad \boldsymbol{b}_1 \in \mathbb{R}^{D_h} \qquad \boldsymbol{U} \in \mathbb{R}^{D_h\times|V|} \qquad \boldsymbol{b}_2 \in \mathbb{R}^{|V|} \tag{2}$$

where $\boldsymbol{L}$ is the embedding matrix, $\boldsymbol{I}$ is the input word weight matrix, $\boldsymbol{H}$ is the hidden state weight matrix, $\boldsymbol{U}$ is the output word transformation matrix, and $\boldsymbol{b}_1$ and $\boldsymbol{b}_2$ are biases. As for the dimensions, $|V|$ is the vocabulary size, d is the embedding dimension, and D_h is the hidden state dimension.

The output vector $\hat{\boldsymbol{y}}^{(t)} \in \mathbb{R}^{|V|}$ is a probability distribution over the vocabulary, and we optimize the cross-entropy loss:

$$J^{(t)}(\theta) = \text{CE}(\boldsymbol{y}^{(t)}, \hat{\boldsymbol{y}}^{(t)}) = -\sum_{i=1}^{|V|} {y_i}^{(t)} \log(\hat{y}_i^{(t)})$$

where $\boldsymbol{y}^{(t)}$ is the one-hot vector corresponding to the target word (which in our case is equal to $\boldsymbol{x}^{(t+1)}$). Note that $J^{(t)}(\theta)$ is a loss for a single timestep.

(a) Compute the gradients for all model parameters at a single point in time (time-step) t:

$$\frac{\partial J^{(t)}}{\partial \boldsymbol{U}} \qquad \frac{\partial J^{(t)}}{\partial \boldsymbol{L}_{\boldsymbol{x}^{(t)}}} \qquad \left.\frac{\partial J^{(t)}}{\partial \boldsymbol{I}}\right|_{(t)} \qquad \left.\frac{\partial J^{(t)}}{\partial \boldsymbol{H}}\right|_{(t)}$$

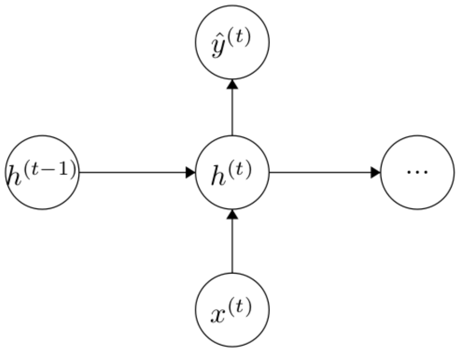
Where $\boldsymbol{L}_{\boldsymbol{x}^{(t)}}$ is the row of $\boldsymbol{L}$ corresponding to the current input word $\boldsymbol{x}^{(t)}$ and $\Big|_{(t)}$ denotes the gradient for the appearance of that parameter at time t. (Equivalently, $\boldsymbol{h}^{(t-1)}$ is taken to be fixed, and you don't need to back-propagate to earlier time-steps just yet - you'll do that in part (b)). Additionally, compute the derivative with respect to the *previous* hidden layer value:[2]

$$\frac{\partial J^{(t)}}{\partial \boldsymbol{h}^{(t-1)}}$$

[1] `http://www.fit.vutbr.cz/research/groups/speech/publi/2010/mikolov_interspeech2010_IS100722.pdf`. You might Recognize Mikolov from `https://arxiv.org/abs/1301.3781`.

[2] For those of you who took Intro to ML, this derivative is also known as an "error term", $\boldsymbol{\delta}^{(t-1)}$.

(b) Below is a sketch of the network at a single time-step:



Draw the unrolled network for 3 time-steps and compute the "back-propagation-through-time" gradients:

$$\frac{\partial J^{(t)}}{\partial \boldsymbol{L}_{\boldsymbol{x}^{(t-1)}}} \qquad \left.\frac{\partial J^{(t)}}{\partial \boldsymbol{H}}\right|_{(t-1)} \qquad \left.\frac{\partial J^{(t)}}{\partial \boldsymbol{I}}\right|_{(t-1)} \qquad \left.\frac{\partial J^{(t)}}{\partial \boldsymbol{b}_1}\right|_{(t-1)}$$

where $\Big|_{(t-1)}$ denotes the gradient from the appearance of that parameter at time $(t-1)$. Because parameters are used multiple times in a forward computation, to implement an RNN we need to compute the gradient for each time they appear.

Use back-propagation rules and express your answer in the terms you computed in part (a). You can also use any other term mentioned in the introduction of Section 2. (This might prove easier than you expect, due to the elegance of back-propagation).

Note that the true gradient with respect to a training example requires us to run back-propagation all the way back to $t = 0$. *In practice, however, we generally truncate this and only back-propagate for a fixed number of time-steps.*

3 Generating Shakespeare Using a Character-level Language Model

In this section we will train a language model and use it to generate text.
Follow the instructions, complete the code, and answer the questions from this Google Colab notebook[3]:
`https://colab.research.google.com/drive/1WIUACyCAgrPiuKzNBwXNChOzWrecLnCF?usp=sharing`

4 Perplexity

(a) Show that perplexity calculated using the natural logarithm $\ln(x)$ is equal to perplexity calculated using $\log_2(x)$. i.e:

$$2^{-\frac{1}{M}\sum_{i=1}^{M}\log_2 p(s_i|s_1,\ldots,s_{i-1})} = e^{-\frac{1}{M}\sum_{i=1}^{M}\ln p(s_i|s_1,\ldots,s_{i-1})}$$

(b) In this section you will be computing the perplexity of your previous trained models on two different passages. Please provide your results in the PDF file, as well as attach the code to your code files. The two different passages appear in the .zip file you've got. Their names are:

[3] Feel free to comment inside the notebook.

`shakespeare_for_perplexity.txt` which contains a subset from the Shakespeare dataset, and `wikipedia_for_perplexity.txt` which contains a certain passage from Wikipedia. Please compute the perplexity of the bi-gram LM, and the model from section 3, on both these passages.

(c) Try to explain the results you've got. Particularly, why there might be large gaps in perplexity, while looking on different passages.

5 Deep Averaging Networks

In this question we will implement Deep Averaging Networks (DAN), and work on the IMDB dataset. The IMDB dataset consists of positive and negative reviews for movies.
This exercise will also introduce you to the popular `transformers` package from Hugging Face. It will also require reading some of the documentation of `pytorch`. The total number of lines of code you need to write for implementing the model itself is roughly 10 or less (i.e., not a lot). Complete the code, and answer the following questions: `https://colab.research.google.com/drive/141W2qonpIYahv0wj-iJ-xqAhRFEa1muV?usp=sharing`

Note: The model in the paper uses GloVe embeddings, in this exercise, you will implement this model using GloVe embedding trained on slightly less data, so you should expect different results then the ones shown in the paper.

(a) Implement the DAN model as described in section 3 in the paper. In a nutshell, the DAN model proposes to average the GloVe word embeddings to represent the sentence, and then pass this sentence representation through a multi-layer feed-forward network (or multi-layer perceptron). Your best model should get accuracy of at least 83.5% on the evaluation set (anything below will result in partial credit). Add feed-forward layers, and tune the learning rate and batch size as necessary. Include a plot of the evaluation accuracy as a function of the number of epochs.

(b) Word dropout is a popular method for regularizing the model and avoiding over-fitting. Read section 3.1 of the paper and add word dropout as described (see `pytorch` documentation), and include a plot of the accuracy of the model across different values of the dropout rate (See Figure 2 in the paper).

(c) Train 4 models with an increasing number of hidden layers (from 0 to 3 hidden layers), and compare the accuracy as the number of layers increases. Before you start training, think about when will we start seeing the effect of diminishing returns, are the results the way you expected them to be? Did the linear model outperform the model with 4 hidden layers? Include a plot of the accuracy as a function of the number of layers.

(d) Use *nn.Relu* and 2 other activation functions (of your choosing) from the torch documentation, include a plot of the accuracy across epochs. What have you learned from this experiment?

(e) For your best model, sample 5 examples from the evaluation set that the model classified incorrectly and for each example try to explain why the model classified it incorrectly.

6 Right-to-left vs left-to-right Estimation

Let $x_0, x_1, ..., x_n$ be any sentence, where x_0 is the start symbol and x_n is the end symbol. Prove that estimating $P(x_0, x_1, \ldots, x_n)$ with a left-to-right count-based bi-gram model (which uses $P(x_i \mid x_{i-1})$) is equal to estimating it with a right-to-left count-based bi-gram model ($P(x_i \mid x_{i+1})$):

$$P(x_0 x_1 ... x_n) = P(x_n)P(x_{n-1} \mid x_n) \ldots P(x_0 \mid x_1) = P(x_0)P(x_1 \mid x_0) \ldots P(x_n \mid x_{n-1}).$$