CS 224n: Assignment #2

Due date: 2/8 11:59 PM PST (You are allowed to use 3 late days maximum for this assignment)

These questions require thought, but do not require long answers. Please be as concise as possible.

? We encourage students to discuss in groups for assignments. **However, each student must finish the problem set and programming assignment individually, and must turn in her/his assignment**. We ask that you abide the university Honor Code and that of the Computer Science department, and make sure that all of your submitted work are done by yourself.

Please review any additional instructions posted on the assignment page at http://cs224n.stanford.edu/assignments.html. When you are ready to submit, please follow the instructions on Piazza.

1 Tensorflow Softmax (25 points)

In this question, we will implement a linear classifier with loss function

$$J(\boldsymbol{W}) = CE(\boldsymbol{y}, \text{softmax}(\boldsymbol{x}\boldsymbol{W} + \boldsymbol{b}))$$

Where $\boldsymbol{x}$ is a vector of features, $\boldsymbol{W}$ is the model's weight matrix, and $\boldsymbol{b}$ is a bias term. We will use TensorFlow's automatic differentiation capability to fit this model to provided data.

(a) (5 points, coding) Implement the softmax function using TensorFlow in `q1_softmax.py`. Remember that

$$\text{softmax}(\boldsymbol{x})_i = \frac{e^{\boldsymbol{x}_i}}{\sum_j e^{x_j}}$$

Note that you may **not** use `tf.nn.softmax` or related built-in functions. You can run basic (nonexhaustive tests) by running `python q1_softmax.py`.

(b) (5 points, coding) Implement the cross-entropy loss using TensorFlow in `q1_softmax.py`. Remember that

$$CE(\boldsymbol{y}, \hat{\boldsymbol{y}}) = -\sum_{i=1}^{N_c} y_i \log(\hat{y}_i)$$

where $\boldsymbol{y} \in \mathbb{R}^{N_c}$ is a one-hot label vector and N_c is the number of classes. This loss is summed over all examples in a minibatch. Note that you may **not** use TensorFlow's built-in cross-entropy functions for this question. You can run basic (non-exhaustive tests) by running `python q1_softmax.py`.

(c) (5 points, coding/written) Carefully study the Model class in `model.py`. Briefly explain the purpose of placeholder variables and feed dictionaries in TensorFlow computations. Fill in the implementations for `add_placeholders` and `create_feed_dict` in `q1_classifier.py`.

Hint: Note that configuration variables are stored in the Config class. You will need to use these configuration variables in the code.

Solution: Placeholder variables and feed dictionaries make it possible to feed data (such as training examples for a neural network) into the computational graph.

(d) (5 points, coding) Implement the transformation for a softmax classifier in the function `add_prediction_op` in `q1_classifier.py`. Add cross-entropy loss in the function `add_loss_op` in the same file. Use the implementations from the earlier parts of the problem (already imported for you), **not** TensorFlow built-ins.

(e) (5 points, coding/written) Fill in the implementation for `add_training_op` in `q1_classifier.py`. Explain in a few sentences what happens when the model's `train_op` is called (what gets computed during forward propagation, what gets computed during backpropagation, and what will have changed after the op has been run?). Verify that your model is able to fit to synthetic data by running python `q1_classifier.py` and making sure that the tests pass.

Hint: Make sure to use the learning rate specified in `Config`.

Solution: During forwards propagation, the value of the loss will be computed. Then during backwards propagation, the derivative of the loss with respect to the `tf.Variables` we defined will be computed. Lastly, the variables in our graph will be updated according to Stochastic Gradient Descent.

2 Neural Transition-Based Dependency Parsing (50 points)

In this section, you'll be implementing a neural-network based dependency parser. A dependency parser analyzes the grammatical structure of a sentence, establishing relationships between "head" words and words which modify those heads. Your implementation will be a *transition-based* parser, which incrementally builds up a parse one step at a time. At every step it maintains a partial parse, which is represented as follows:

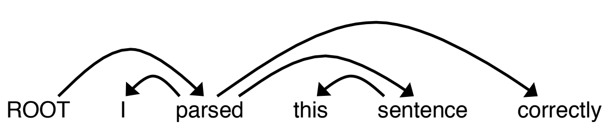
- A *stack* of words that are currently being processed.
- A *buffer* of words yet to be processed.
- A list of *dependencies* predicted by the parser.

Initially, the stack only contains ROOT, the dependencies lists is empty, and the buffer contains all words of the sentence in order. At each step, the parse applies a *transition* to the partial parse until its buffer is empty and the stack is size 1. The following transitions can be applied:

- `SHIFT`: removes the first word from the buffer and pushes it onto the stack.
- `LEFT-ARC`: marks the second (second most recently added) item on the stack as a dependent of the first item and removes the second item from the stack.
- `RIGHT-ARC`: marks the first (most recently added) item on the stack as a dependent of the second item and removes the first item from the stack.

Your parser will decide among transitions at each state using a neural network classifier. First, you will implement the partial parse representation and transition functions.

(a) (6 points, written) Go through the sequence of transitions needed for parsing the sentence *"I parsed this sentence correctly"*. The dependency tree for the sentence is shown below. At each step, give the configuration of the stack and buffer, as well as what transition was applied this step and what new dependency was added (if any). The first three steps are provided below as an example.



stack	buffer	new dependency	transition
[ROOT]	[I, parsed, this, sentence, correctly]		Initial Configuration
[ROOT, I]	[parsed, this, sentence, correctly]		`SHIFT`
[ROOT, I, parsed]	[this, sentence, correctly]		`SHIFT`
[ROOT, parsed]	[this, sentence, correctly]	parsed→I	`LEFT-ARC`

Solution:

stack	buffer	new dependency	transition
[ROOT, parsed, this]	[sentence, correctly]		`SHIFT`
[ROOT, parsed, this, sentence]	[correctly]		`SHIFT`
[ROOT, parsed, sentence]	[correctly]	sentence→this	`LEFT-ARC`
[ROOT, parsed]	[correctly]	parsed→sentence	`RIGHT-ARC`
[ROOT, parsed, correctly]	[]		`SHIFT`
[ROOT, parsed]	[]	parsed→correctly	`RIGHT-ARC`
[ROOT]	[]	ROOT→parsed	`RIGHT-ARC`

(b) (2 points, written) A sentence containing n words will be parsed in how many steps (in terms of n)? Briefly explain why.

Solution: Each word of the sentence must be shifted onto the stack and then reduced away, so a sentence containing n words will be parsed in $2n$ steps.

(c) (6 points, coding) Implement the `__init__` and `parse_step` functions in the `PartialParse` class in `q2_parser_transitions.py`. This implements the transition mechanics your parser will use. You can run basic (not-exhaustive) tests by running `python q2_parser_transitions.py`.

(d) (6 points, coding) Our network will predict which transition should be applied next to a partial parse. We could use it to parse a single sentence by applying predicted transitions until the parse is complete. However, neural networks run much more efficiently when making predictions about batches of data at a time (i.e., predicting the next transition for a many different partial parses simultaneously). We can parse sentences in minibatches with the following algorithm.

Algorithm 1 Minibatch Dependency Parsing

Input: `sentences`, a list of sentences to be parsed and `model`, our model that makes parse decisions

Initialize `partial_parses` as a list of partial parses, one for each sentence in `sentences`
Initialize `unfinished_parses` as a shallow copy of `partial_parses`
while `unfinished_parses` is not empty **do**
  Take the first `batch_size` parses in `unfinished_parses` as a minibatch
  Use the `model` to predict the next transition for each partial parse in the minibatch
  Perform a parse step on each partial parse in the minibatch with its predicted transition
  Remove the completed (empty buffer and stack of size 1) parses from `unfinished_parses`
end while

Return: The `dependencies` for each (now completed) parse in `partial_parses`.

Implement this algorithm in the `minibatch_parse` function in `q2_parser_transitions.py`. You can run basic (not-exhaustive) tests by running `python q2_parser_transitions.py`.

Note: You will need `minibatch_parse` to be correctly implemented to evaluate the model you will build in part (h). However, you do not need it to train the model, so you should be able to complete most of part (h) even if `minibatch_parse` is not implemented yet.

We are now going to train a neural network to predict, given the state of the stack, buffer, and dependencies, which transition should be applied next. First, the model extracts a feature vector representing the current state. We will be using the feature set presented in the original neural dependency parsing paper: *A Fast and Accurate Dependency Parser using Neural Networks*[1]. The function extracting these features has been implemented for you in `parser_utils`. This feature vector consists of a list of tokens (e.g., the last word in the stack, first word in the buffer, dependent of the second-to-last word in the stack if there is one, etc.). They can be represented as a list of integers

$$[w_1, w_2, ..., w_m]$$

where m is the number of features and each $0 \leq w_i < |V|$ is the index of a token in the vocabulary ($|V|$ is the vocabulary size). First our network looks up an embedding for each word and concatenates them into a single input vector:

$$\boldsymbol{x} = [\boldsymbol{E}_{w_1}, ..., \boldsymbol{E}_{w_m}] \in \mathbb{R}^{dm}$$

where $\boldsymbol{E} \in \mathbb{R}^{|V| \times d}$ is an embedding matrix with each row $\boldsymbol{E}_i$ as the vector for a particular word i. We then compute our prediction as:

$$\boldsymbol{h} = \mathrm{ReLU}(\boldsymbol{x}\boldsymbol{W} + \boldsymbol{b}_1)$$
$$\hat{\boldsymbol{y}} = \mathrm{softmax}(\boldsymbol{h}\boldsymbol{U} + \boldsymbol{b}_2)$$

(recall that $\mathrm{ReLU}(z) = \max(z, 0)$). We will train the model to minimize cross-entropy loss:

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

To compute the loss for the training set, we average this $J(\theta)$ across all training examples.

(e) (4 points, coding) In order to avoid neurons becoming too correlated and ending up in poor local minimina, it is often helpful to randomly initialize parameters. One of the most frequent initializations used is called Xavier initialization[2].

Given a matrix $\boldsymbol{A}$ of dimension $m \times n$, Xavier initialization selects values A_{ij} uniformly from $[-\epsilon, \epsilon]$, where

$$\epsilon = \frac{\sqrt{6}}{\sqrt{m+n}}$$

Implement the initialization in `xavier_weight_init` in `q2_initialization.py`. You can run basic (nonexhaustive tests) by running `python q2_initialization.py`. This function will be used to initialize $\boldsymbol{W}$ and $\boldsymbol{U}$.

(f) (2 points, written) We will regularize our network by applying Dropout[3]. 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 γ (dropping different units each minibatch). We can write this as

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

[1] Chen and Manning, 2014, http://cs.stanford.edu/people/danqi/papers/emnlp2014.pdf

[2] This is also referred to as Glorot initialization and was initially described in `http://jmlr.org/proceedings/papers/v9/glorot10a/glorot10a.pdf`

[3] Srivastava et al., 2014, https://www.cs.toronto.edu/ hinton/absps/JMLRdropout.pdf

where $\boldsymbol{d} \in \{0,1\}^{D_h}$ (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 $(1 - p_{drop})$. γ is chosen such that the value of $\boldsymbol{h}_{drop}$ in expectation equals $\boldsymbol{h}$:

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

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

Solution:

$$\mathbb{E}_{p_{drop}}[\boldsymbol{h}_{drop}]_i = \mathbb{E}_{p_{drop}}[\gamma d_i h_i] = p_{drop}(0) + (1 - p_{drop})\gamma h_i = (1 - p_{drop})\gamma h_i = h_i$$

So γ must equal $1/(1 - p_{drop})$

(g) (4 points, written) We will train our model using the Adam[4] optimizer. Recall that standard SGD uses the update rule

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \alpha \nabla_{\boldsymbol{\theta}} J_{minibatch}(\boldsymbol{\theta})$$

where $\boldsymbol{\theta}$ is a vector containing all of the model parameters, J is the loss function, $\nabla_{\boldsymbol{\theta}} J_{minibatch}(\boldsymbol{\theta})$ is the gradient of the loss function with respect to the parameters on a minibatch of data, and α is the learning rate. Adam uses a more sophisticated update rule with two additional steps[5].

(i) First, Adam uses a trick called *momentum* by keeping track of $\boldsymbol{m}$, a rolling average of the gradients:

$$\begin{aligned} \boldsymbol{m} &\leftarrow \beta_1 \boldsymbol{m} + (1 - \beta_1)\nabla_{\boldsymbol{\theta}} J_{minibatch}(\boldsymbol{\theta}) \\ \boldsymbol{\theta} &\leftarrow \boldsymbol{\theta} - \alpha \boldsymbol{m} \end{aligned}$$

where β_1 is a hyperparameter between 0 and 1 (often set to 0.9). Briefly explain (you don't need to prove mathematically, just give an intuition) how using $\boldsymbol{m}$ stops the updates from varying as much. Why might this help with learning?

Solution: Each update will be mostly the same as the previous one (only $1 - \beta_1$ of $\boldsymbol{m}$ changes each step), so the updates won't vary as much. One way of thinking about this is that it will stop the model parameters from "bouncing around as much" when moving towards a local optimum. Another way is that doing the rolling average is a bit like computing the gradient over a larger minibatch, so each update will be closer to the true gradient over the whole dataset (i.e., lower variance means each gradient estimate is closer to the mean).

(ii) Adam also uses *adaptive learning rates* by keeping track of $\boldsymbol{v}$, a rolling average of the magnitudes of the gradients:

$$\begin{aligned} \boldsymbol{m} &\leftarrow \beta_1 \boldsymbol{m} + (1 - \beta_1)\nabla_{\boldsymbol{\theta}} J_{minibatch}(\boldsymbol{\theta}) \\ \boldsymbol{v} &\leftarrow \beta_2 \boldsymbol{v} + (1 - \beta_2)(\nabla_{\boldsymbol{\theta}} J_{minibatch}(\boldsymbol{\theta}) \circ \nabla_{\boldsymbol{\theta}} J_{minibatch}(\boldsymbol{\theta})) \\ \boldsymbol{\theta} &\leftarrow \boldsymbol{\theta} - \alpha \circ \boldsymbol{m} / \sqrt{\boldsymbol{v}} \end{aligned}$$

where $\circ$ and $/$ denote elementwise multiplication and division (so $\boldsymbol{z} \circ \boldsymbol{z}$ is elementwise squaring) and β_2 is a hyperparameter between 0 and 1 (often set to 0.99). Since Adam divides the update by $\sqrt{\boldsymbol{v}}$, which of the model parameters will get larger updates? Why might this help with learning?

Solution: The parameters with the smallest gradients (on average) will get the larger updates. This means parameters that are at a place where the loss with respect to them is pretty flat will get larger updates, helping them move off the flat areas.

[4]Kingma and Ma, 2015, https://arxiv.org/pdf/1412.6980.pdf

[5]The actual Adam update uses a few additional tricks that are less important, but we won't worry about them for this problem.

(h) (20 points, coding/written) In `q2_parser_model.py` implement the neural network classifier governing the dependency parser by filling in the appropriate sections. We will train and evaluate our model on the Penn Treebank (annotated with Universal Dependencies).Run python `q2_parser_model.py` to train your model and compute predictions on the test data (make sure to turn off debug settings when doing final evaluation).

Hints:

- When debugging, pass the keyword argument `debug=True` to the `main` method (it is set to `true` by default). This will cause the code to run over a small subset of the data, so the training the model won't take as long.
- This code should run within 1 hour on a CPU.
- When running with `debug=True`, you should be able to get a loss smaller than 0.2 and a UAS larger than 65 on the dev set (although in rare cases your results may be lower, there is some randomness when training). When running with `debug=False`, you should be able to get a loss smaller than 0.08 on the train set and an Unlabeled Attachment Score larger than 87 on the dev set. For comparison, the model in the original neural dependency parsing paper gets 92.5 UAS. If you want, you can tweak the hyperparameters for your model (hidden layer size, hyperparameters for Adam, number of epochs, etc.) to improve the performance (but you are not required to do so).

Deliverables:

- Working implementation of the neural dependency parser in `q2_parser_model.py`. (We'll look at, and possibly run this code for grading).
- Report the best UAS your model achieves on the dev set and the UAS it achieves on the test set.
- List of predicted labels for the test set in the file `q2_test.predicted`.

(i) **Bonus** (2 points). Add an extension to your model (e.g., l2 regularization, an additional hidden layer) and report the change in UAS on the dev set. Briefly explain what your extension is, what UAS the resulting model gets on the dev and test sets, and why the extension helps (or hurts!) the model. Some extensions may require tweaking the hyperparameters in `Config` to make them effective. For both bonus points, your extended model should get better UAS than the baseline model without an extension. You should turn in your extension with your code, but **your extension should be turned off by default**. An easy way of doing this is add an `extension_on` boolean to your model's `Config` that is set to `False` when you turn in the code. If your extension does not improve over your baseline, do not turn in those predictions when you submit `q2_test.predicted`, use the baseline's ones instead (we will make sure these predictions produce a good UAS when grading)!

3 Recurrent Neural Networks: Language Modeling (25 points)

In this section, you'll analyze a recurrent neural network (RNN) used for language modeling.

Language modeling is a central task in NLP, and language models can be found at the heart of speech recognition, machine translation, and many other systems. Given a sequence of words (represented as one-hot vectors) $\boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}, \ldots, \boldsymbol{x}^{(t)}$, a language model predicts the next word $\boldsymbol{x}^{(t+1)}$ by modeling:

$$P(\boldsymbol{x}^{(t+1)} = \boldsymbol{w}_j \mid \boldsymbol{x}^{(t)}, \ldots, \boldsymbol{x}^{(1)})$$

where $\boldsymbol{w}_j$ is a word in the vocabulary.

Your job is to compute the gradients of a recurrent neural network language model, which uses a hidden layer to represent the "history" $\boldsymbol{x}^{(t)}, \boldsymbol{x}^{(t-1)}, \ldots, \boldsymbol{x}^{(1)}$ of previous words. Formally, the model[6] is:

$$\begin{aligned}
\boldsymbol{e}^{(t)} &= \boldsymbol{E}\boldsymbol{x}^{(t)} \\
\boldsymbol{h}^{(t)} &= \text{sigmoid}\left(\boldsymbol{W}_h \boldsymbol{h}^{(t-1)} + \boldsymbol{W}_e \boldsymbol{e}^{(t)} + \boldsymbol{b}_1\right) \\
\hat{\boldsymbol{y}}^{(t)} &= \text{softmax}\left(\boldsymbol{U}\boldsymbol{h}^{(t)} + \boldsymbol{b}_2\right) \\
\bar{P}(\boldsymbol{x}^{(t+1)} = \boldsymbol{w}_j \mid \boldsymbol{x}^{(t)}, \ldots, \boldsymbol{x}^{(1)})) &= \hat{\boldsymbol{y}}_j^{(t)}
\end{aligned}$$

where $\boldsymbol{h}^{(0)} = \boldsymbol{h}_0 \in \mathbb{R}^{D_h}$ is some initialization vector for the hidden layer and $\boldsymbol{E}\boldsymbol{x}^{(t)}$ is the product of $\boldsymbol{E}$ with the one-hot vector $\boldsymbol{x}^{(t)}$ representing the current word. Essentially, the first equation embeds the current word, the second equation produces a new hidden state given the current word's embedding and previous hidden state, and the last equation predicts what the next word will be. The parameters are:

$$\boldsymbol{E} \in \mathbb{R}^{d \times |V|} \quad \boldsymbol{W}_h \in \mathbb{R}^{D_h \times D_h} \quad \boldsymbol{W}_e \in \mathbb{R}^{D_h \times d} \quad \boldsymbol{b}_1 \in \mathbb{R}^{D_h} \quad \boldsymbol{U} \in \mathbb{R}^{|V| \times D_h} \quad \boldsymbol{b}_2 \in \mathbb{R}^{|V|} \tag{1}$$

where $\boldsymbol{E}$ is the embedding matrix, $\boldsymbol{W}_e$ the input word representation matrix, $\boldsymbol{W}_h$ the hidden state transformation matrix, and $\boldsymbol{U}$ is the output word representation matrix. $\boldsymbol{b}_1$ and $\boldsymbol{b}_2$ are biases. d is the embedding size, $|V|$ is the vocabulary size, and D_h is the hidden layer size.

The output vector $\hat{\boldsymbol{y}}^{(t)} \in \mathbb{R}^{|V|}$ is a probability distribution over the vocabulary. The model is trained by minimizing the (un-regularized) cross-entropy loss:

$$J^{(t)}(\theta) = CE(\boldsymbol{y}^{(t)}, \hat{\boldsymbol{y}}^{(t)}) = -\sum_{j=1}^{|V|} y_j^{(t)} \log \hat{y}_j^{(t)}$$

where $\boldsymbol{y}^{(t)}$ is the one-hot vector corresponding to the target word (which here is equal to $\boldsymbol{x}^{(t+1)}$). We average the cross-entropy loss across all examples (i.e., words) in a sequence to get the loss for a single sequence.

(a) (4 points, written) Conventionally, when reporting performance of a language model, we evaluate on *perplexity*, which is defined as:

$$\text{PP}^{(t)}\left(\boldsymbol{y}^{(t)}, \hat{\boldsymbol{y}}^{(t)}\right) = \frac{1}{\bar{P}(\boldsymbol{x}_{\text{pred}}^{(t+1)} = \boldsymbol{x}^{(t+1)} \mid \boldsymbol{x}^{(t)}, \ldots, \boldsymbol{x}^{(1)})} = \frac{1}{\sum_{j=1}^{|V|} y_j^{(t)} \cdot \hat{y}_j^{(t)}}$$

i.e. the inverse probability of the correct word, according to the model distribution $\bar{P}$.

(i) (2 points) Show how you can derive perplexity from the cross-entropy loss (*Hint: remember that* $\boldsymbol{y}^{(t)}$ *is one-hot!*). ***This should be a very short problem - not too perplexing!***

Solution: Since $y^{(t)}$ is one-hot, suppose without loss of generality that $y_i^{(t)}$ is the only nonzero element of $y^{(t)}$. Then

$$\text{CE}(y^{(t)}, \hat{y}^{(t)}) = -\log \hat{y}_i^{(t)} = \log \frac{1}{\hat{y}^{(t)}}$$

$$\text{PP}(y^{(t)}, \hat{y}^{(t)}) = \frac{1}{\hat{y}_i^{(t)}}$$

[6] This model is adapted from a paper by Toma Mikolov, et al. from 2010: `http://www.fit.vutbr.cz/research/groups/speech/publi/2010/mikolov_interspeech2010_IS100722.pdf`

It follows that

$$\mathrm{CE}(y^{(t)}, \hat{y}^{(t)}) = \log \mathrm{PP}(y^{(t)}, \hat{y}^{(t)})$$

(ii) (1 point) Now use this relationship between perplexity and cross-entropy to show that minimizing the geometric mean perplexity, $\left(\prod_{t=1}^{T} PP(\boldsymbol{y}^{(t)}, \hat{\boldsymbol{y}}^{(t)})\right)^{1/T}$, is equivalent to minimizing the arithmetic mean cross-entropy loss, $\frac{1}{T}\sum_{t=1}^{T} CE(\boldsymbol{y}^{(t)}, \hat{\boldsymbol{y}}^{(t)})$, across the training set. (*Hint: for any positive function f, minimizing* $\log(f)$ *is equivalent to minimizing f itself*)

Solution: Minimizing the log of the geometric mean of the perplexity is equivalent to minimizing the arithmetic mean of the cross entropy:

$$\log\left[\left(\prod_{t=1}^{T} PP(\boldsymbol{y}^{(t)}, \hat{\boldsymbol{y}}^{(t)})\right)^{1/T}\right] = \frac{1}{T}\sum_{t=1}^{T} \log PP(\boldsymbol{y}^{(t)}, \hat{\boldsymbol{y}}^{(t)}) = \frac{1}{T}\sum_{t=1}^{T} CE(\boldsymbol{y}^{(t)}, \hat{\boldsymbol{y}}^{(t)})$$

(iii) (1 point) Suppose you have a vocabulary of $|V|$ words and your "language model" works by picking the next word uniformly at random from the vocabulary (mathematically, $\bar{P}(\boldsymbol{x}^{(t+1)} = \boldsymbol{w}_j \mid \boldsymbol{x}^{(t)}, \ldots, \boldsymbol{x}^{(1)})) = 1/|V|$ for every word $\boldsymbol{w}_j$ in the vocabulary). What would the perplexity $PP(\boldsymbol{y}^{(t)}, \hat{\boldsymbol{y}}^{(t)})$ for a single word to be? Compute the corresponding cross-entropy loss when $|V| = 10000$. **Solution:** $E[\hat{y}_i^{(t)}] = \frac{1}{|V|}$. Given that $y^{(t)}$ is one-hot, the expected value of the perplexity is $1/(1/|V|) = |V|$. Since the cross-entropy is the logarithm of perplexity, the expected cross-entropy if $|V| = 10000$ is $\log 10000 \approx 9.21$.

(b) (7 points, written) Compute the gradients of the loss J (the cross-entropy loss) with respect to the following model parameters at a single point in time t (to save a bit of time, you don't have to compute the gradients with the respect to the biases $\boldsymbol{b}_1$ and $\boldsymbol{b}_2$):

$$\frac{\partial J^{(t)}}{\partial \boldsymbol{U}} \qquad \frac{\partial J^{(t)}}{\partial \boldsymbol{e}^{(t)}} \qquad \left.\frac{\partial J^{(t)}}{\partial \boldsymbol{W}_e}\right|_{(t)} \qquad \left.\frac{\partial J^{(t)}}{\partial \boldsymbol{W}_h}\right|_{(t)}$$

where $\big|_{(t)}$ denotes the gradient for the appearance of that parameter at time t (in other words, $\boldsymbol{h}^{(t-1)}$ is taken to be fixed, so you don't need to take it's derivative when applying the chain rule and backpropagate to earlier timesteps - you'll do that in part (c)).

Additionally, compute the derivative with respect to the *previous* hidden layer value:

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

Hint: you may want to define the following variables:

$$\boldsymbol{z}^{(t)} = \boldsymbol{W}_h \boldsymbol{h}^{(t-1)} + \boldsymbol{W}_e \boldsymbol{e}^{(t)} + \boldsymbol{b}_1$$
$$\boldsymbol{\theta}^{(t)} = \boldsymbol{U}\boldsymbol{h}^{(t)} + \boldsymbol{b}_2$$

and define (and compute the value of) the following error terms:

$$\boldsymbol{\delta}_1^{(t)} = \frac{\partial J}{\partial \boldsymbol{\theta}^{(t)}}$$
$$\boldsymbol{\delta}_2^{(t)} = \frac{\partial J}{\partial \boldsymbol{z}^{(t)}} = \boldsymbol{\delta}_1^{(t)} \frac{\partial \boldsymbol{\theta}^{(t)}}{\partial \boldsymbol{h}^{(t)}} \frac{\partial \boldsymbol{h}^{(t)}}{\partial \boldsymbol{z}^{(t)}}$$

Solution:

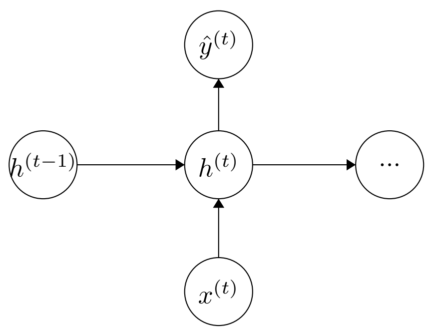
Recall that $\frac{d}{dz}\text{sigmoid}(z) = \text{sigmoid}(z)(1 - \text{sigmoid}(z))$. We can define the error signals (using Jacobian formulation)

$$\boldsymbol{\delta}_1^{(t)} = \frac{\partial J}{\partial \boldsymbol{\theta}^{(t)}} = (\hat{\boldsymbol{y}}^{(t)} - \boldsymbol{y}^{(t)})^T$$
$$\boldsymbol{\delta}_2^{(t)} = \frac{\partial J}{\partial \boldsymbol{z}^{(t)}} = \boldsymbol{\delta}_1^{(t)} \frac{\partial \boldsymbol{\theta}^{(t)}}{\partial \boldsymbol{h}^{(t)}} \frac{\partial \boldsymbol{h}^{(t)}}{\partial \boldsymbol{z}^{(t)}} = \boldsymbol{\delta}_1^{(t)} U \circ \boldsymbol{h}^{(t)} \circ (1 - \boldsymbol{h}^{(t)})$$

This results in the gradients

$$\frac{\partial J}{\partial \boldsymbol{U}} = \frac{\partial J}{\partial \boldsymbol{\theta}^{(t)}} \frac{\partial \boldsymbol{\theta}^{(t)}}{\partial \boldsymbol{U}} = \boldsymbol{\delta}_1^{(t)T} \boldsymbol{h}^{(t)T}$$
$$\frac{\partial J}{\partial \boldsymbol{e}^{(t)}} = \frac{\partial J}{\partial \boldsymbol{z}^{(t)}} \frac{\partial \boldsymbol{z}^{(t)}}{\partial \boldsymbol{e}^{(t)}} = (\boldsymbol{\delta}_2^{(t)} \boldsymbol{W}_e)^T$$
$$\frac{\partial J}{\partial \boldsymbol{W}_e} = \frac{\partial J}{\partial \boldsymbol{z}^{(t)}} \frac{\partial \boldsymbol{z}^{(t)}}{\partial \boldsymbol{W}_e} = \boldsymbol{\delta}_2^{(t)T} \boldsymbol{e}^{(t)T}$$
$$\frac{\partial J}{\partial \boldsymbol{W}_h} = \frac{\partial J}{\partial \boldsymbol{z}^{(t)}} \frac{\partial \boldsymbol{z}^{(t)}}{\partial \boldsymbol{W}_h} = \boldsymbol{\delta}_2^{(t)T} (\boldsymbol{h}^{(t-1)})^T$$
$$\frac{\partial J}{\partial \boldsymbol{h}^{(t-1)}} = \frac{\partial J}{\partial \boldsymbol{z}^{(t)}} \frac{\partial \boldsymbol{z}^{(t)}}{\partial \boldsymbol{h}^{(t-1)}} = (\boldsymbol{\delta}_2^{(t)} \boldsymbol{W}_h)^T$$

(c) (7 points, written) Below is a sketch of the network at a single timestep:

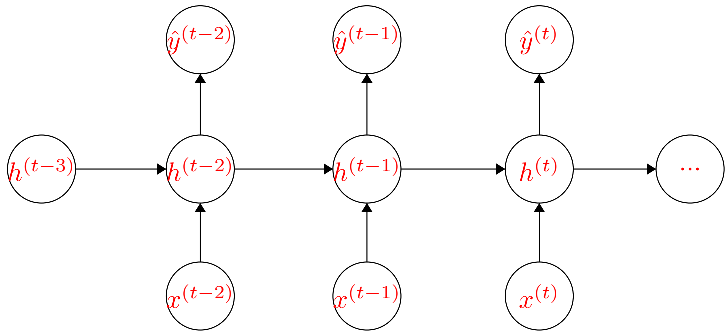


Draw the "unrolled" network for 3 timesteps (your picture should show $h^{(t-3)}$ to $h^{(t)}$. Next, compute the following backpropagation-through-time gradients:

$$\frac{\partial J^{(t)}}{\partial \boldsymbol{e}^{(t-1)}} \qquad \left.\frac{\partial J^{(t)}}{\partial \boldsymbol{W}_e}\right|_{(t-1)} \qquad \left.\frac{\partial J^{(t)}}{\partial \boldsymbol{W}_h}\right|_{(t-1)}$$

where $\big|_{(t-1)}$ denotes the gradient for the appearance of that parameter at time $(t-1)$. Use the error term $\boldsymbol{\gamma}^{(t-1)} = \frac{\partial J^{(t)}}{\partial \boldsymbol{h}^{(t-1)}}$ to represent the derivative you computed in the previous part (it should show up in three gradients you compute).

We have to compute multiple gradients with respect to parameters like $\boldsymbol{W}_e$ because they are used multiple times in the feed-forward computation. The final gradient for the parameters will be the sum of all their gradients over time (e.g., the gradient $\frac{\partial J^{(t)}}{\partial \boldsymbol{W}_e} = \sum_{i=0}^{t} \frac{\partial J^{(t)}}{\partial \boldsymbol{W}_e}\big|_{(i)}$).



Solution:

Let $\sigma'(\boldsymbol{z}^{(t-1)}) = \frac{\partial \boldsymbol{h}^{(t-1)}}{\partial \boldsymbol{z}^{(t-1)}} = \text{diag}(\boldsymbol{h}^{(t-1)} \circ (1 - \boldsymbol{h}^{(t-1)}))$

$$\frac{\partial J}{\partial \boldsymbol{e}^{(t-1)}} = \frac{\partial J}{\partial \boldsymbol{h}^{(t-1)}} \frac{\partial \boldsymbol{h}^{(t-1)}}{\partial \boldsymbol{z}^{(t-1)}} \frac{\partial \boldsymbol{z}^{(t-1)}}{\partial \boldsymbol{e}^{(t-1)}} = (\gamma^{(t-1)} \sigma'(\boldsymbol{z}^{(t-1)}) \boldsymbol{W}_e)^T$$
$$\frac{\partial J}{\partial \boldsymbol{W}_e} = \frac{\partial J}{\partial \boldsymbol{h}^{(t-1)}} \frac{\partial \boldsymbol{h}^{(t-1)}}{\partial \boldsymbol{z}^{(t-1)}} \frac{\partial \boldsymbol{z}^{(t-1)}}{\partial \boldsymbol{W}_e} = (\gamma^{(t-1)})^T \sigma'(\boldsymbol{z}^{(t-1)}) \boldsymbol{e}^{(t-1)T}$$
$$\frac{\partial J}{\partial \boldsymbol{W}_h} = \frac{\partial J}{\partial \boldsymbol{h}^{(t-1)}} \frac{\partial \boldsymbol{h}^{(t-1)}}{\partial \boldsymbol{z}^{(t-1)}} \frac{\partial \boldsymbol{z}^{(t-1)}}{\partial \boldsymbol{W}_h} = (\gamma^{(t-1)})^T \sigma'(\boldsymbol{z}^{(t-1)}) \boldsymbol{h}^{(t-2)T}$$

(d) (3 points, written) Given $\boldsymbol{h}^{(t-1)}$, how many operations are required to perform backpropagation for a single timestep (i.e., compute the terms you found in part (b)). Express your answer in big-O notation in terms of the dimensions d, D_h and $|V|$ (Equation 1). Don't worry about the gradients you didn't compute (with respect to $\boldsymbol{b}_1$ and $\boldsymbol{b}_2$), they actually don't change the answer!

(*Hint: You only have to worry about matrix multiplications, the other operations do not change the big-O runtime. Multiplying a vector by an n by m matrix* takes $O(nm)$ operations.)

Solution: $O(|V|D_h + dD_h + D_h^2)$

(e) (3 points, written) Now suppose you have a sequence of T words. How many operations are required to compute the gradient of the loss with respect to the model parameters across the entire sequence ($\sum_{t=1}^{T} J^{(t)}(\theta)$)? Assume we backpropagate through time all the way to $t = 0$ for each word.

Hint: Look at the computational graph you drew in part (c). Will we have to pass an error signal (upstream gradient) into $h^{(0)}$ T times (once for the loss from every word), or can we be more efficient that that? Therefore, how many times will we have to do the single timestep (your answer to (d)) computations?

Solution: Backpropping through time for a single loss $J^{(t)}(\theta)$ takes $O(T)$ steps, so it looks like the runtime will be $O(T^2(...))$. However, efficient backprop will aggregate gradients into error terms and only go back along each edge in the computational graph once. This results in the runtime.

$O(T(|V|D_h + dD_h + D_h^2))$

(f) (1 point, written) Which term in your big-O expressions is likely the largest? Which layer in the RNN is that term from?

Solution: Slow part is the $O(|V|D_h)$ term from the matrix multiply when computing a probability distribution over next words (assuming $|V| >> D_h$).

Bonus (1 point, written) Given your knowledge of similar models (i.e. `word2vec`), suggest a way to speed up this part of the computation. Your approach can be an approximation, but you should argue why it's a good one. The paper "Extensions of recurrent neural network language model" (Mikolov, et al. 2013) may be of interest here.

Solution: Hierarchical softmax, negative sampling (although the latter only improves training time and not testing time)