Contents

(75pts) COS495 Midterm

Your name:

(15pts) Short answers

(2pts) What is the object being embedded (i.e. a vector representing this object is computed) when one uses

- the pair-pattern matrix? **pairs of words/the semantic relations between them**
- the word-context matrix? **words**

(2pts) Bigrams are consecutive tokens. Let x_1 = 'dog bites man more' and x_2 = 'man bites dog less'. Write down the dense bigram indicator feature vectors for this dataset, label the dimensions.

	(dog,bites)	(bites,man)	(man,more)	(man,bites)	(bites,dog)	(dog,less)
x_1	1	1	1	0	0	0
x_2	0	0	0	1	1	1

(3pts) When faced with a big softmax over the vocabulary, training can be slow. Describe and explain the hierarchical softmax for speeding up training used in word2vec. Identify its parameters and explain the savings.

For a given word w' and vector v we wish to approximate the softmax $\mathbb{P}(w|v) = \frac{v_w^T v}{\sum_{w'} v_{w'}^T v}$ without computing a sum over V inner products in the denominator (which has complexity $\mathcal{O}(V)$), where V is the number of words in the vocabulary. Instead we construct a Huffman tree of depth $\log V$ with one word (and its associated embedding) assigned to each leaf node and a parameter vector at all other nodes. The probability is then approximated as

$$\mathbb{P}(w|v) \approx \prod_{n \in P(w)} \frac{1}{1 + \exp\left((-1)^{L_{w,n}} v_n^T v\right)}$$

where $P(w)$ is the path from the root node to w and $L_{w,n}$ is 1 if w is n or a descendant of the left child of n and -1 otherwise. This can be computed in $\mathcal{O}(\log V)$ time.

GloVe has objective $\sum_{ij} f(X_{ij})(w_i \cdot \tilde{w}_j + b_i + \tilde{b}_j - \log X_{ij})^2$

- **(2pt)** What do the indices i, j represent and what is X_{ij}?
  i and j represent words and X_{ij} represents the number of times they co-occur within a fixed-size window.
- **(3pt)** What is an issue with the alternative objective $\sum_{ij}(w_i \cdot \tilde{w}_j + b_i + \tilde{b}_j - \log X_{ij})^2$?
- since the sum is taken over all word pairs, this expression is infinite/undefined if there exist pairs of words that never co-occur (which is usually the case).
- this objectives gives equal weight to words that co-occur rarely as words that co-occur frequently, even though the confidence we have in the latter is much higher.

(3pts) Let $z = Wa$, $W \in R^{m \times n}$, and consider the backward pass. Suppose we have i.i.d samples $\frac{dL}{dz_i} \sim N(0,1)$ and $W_{ij} \sim N(0, \sigma^2)$. We would like $\text{Var}[\frac{dL}{da_i}] = 1$ by setting σ appropriately. Compute σ and show your steps.

$$\frac{dL}{da_i} = W_i^T \nabla_z L = \sum_j W_{ji} \frac{dL}{z_j}$$

Setting the desired variance to 1 and using the fact that the variance of a product is the product of variances (for mean-zero i.r.v.) we have

$$1 = \text{Var}\left[\frac{dL}{da_i}\right] = \sum_j \text{Var}[W_{ji}] \,\text{Var}\left[\frac{dL}{dz_i}\right] = m\sigma^2 \qquad \Longrightarrow \qquad \sigma = \frac{1}{\sqrt{m}}$$

(5pts) Unequal loss

Not all mistakes are created equal. We define $\text{cost}(y, y')$ as the cost when the label is y and the prediction is y'. Then the hinge loss is

$$\max(0, \max_{y'} \mathrm{cost}(y, y') + s_{y'} - s_y).$$

* **(3pts)** Show that this hinge loss is an upperbound of $\mathrm{cost}(y, y')$.
y' being the prediction means that $y' = \arg\max_{\tilde{y}} s_{\tilde{y}}$, and in particular $s_{y'} \geq s_y$.
Then,

$$\max_{\tilde{y}} \mathrm{cost}(y, \tilde{y}) + s_{\tilde{y}} - s_y \geq \mathrm{cost}(y, y') + s_{y'} - s_y \geq \mathrm{cost}(y, y').$$

Noting $\max(0, x) \geq x$ gives the result.

* **(2pts)** Give the necessary condition on s_i for this upperbound to be tight and an example when it is loose.
For the bound to be tight means the prediction has the same score as the target $s_{y'} = s_y$, and everything else clears the margin, for all $\tilde{y}$, $\mathrm{cost}(y, \tilde{y}) + s_{\tilde{y}} \leq \mathrm{cost}(y, y') + s_y$
This bound is loose whenever there is a clear wrong prediction, $s_{y'} > s_y$,
or if there is $\hat{y}$ for which $\mathrm{cost}(y, \hat{y}) + s_{\hat{y}} > \mathrm{cost}(y, y') + s_y$.

(15pts) About LSTMs

Here are the LSTM equations and the corresponding illustration

$$\begin{aligned}
f_t &= \sigma(W_f[h_{t-1}, x_t] + b_f) \\
i_t &= \sigma(W_i[h_{t-1}, x_t] + b\_i) \\
o_t &= \sigma(W_o[h_{t-1}, x_t] + b_o) \\
\tilde{c}_t &= \tanh(W_c[h_{t-1}, x_t] + b_c) \\
c_t &= f_t \odot c_{t-1} + i_t \odot \tilde{c}_t \\
h_t &= o_t \odot \tanh(c_t)
\end{aligned}$$

LSTM-Chain
LSTM-Notation
(illustration from Chris Olah)

- **(5pt)** Label each of $f_t, i_t, o_t, c_t, \tilde{c}_t, c_{t-1}$ on the illustration. Make sure there is no ambiguity in your labeling.

Suppose we have a one-layer LSTM primitive $[h_1, \ldots, h_T] = \mathrm{LSTM}([x_1, \ldots, x_T]) \in R^{h \times T}$.

- **(2pts)** Represent a two-layer LSTM using LSTM
  $[h_1, \ldots, h_T] = \mathrm{LSTM}(\mathrm{LSTM}([x_1, \ldots, x_T]))$

- **(3pts)** Represent a bidirectional LSTM and use its hidden states to perform a sequence tagging task, where the tag at each timestep depends only on forward and backward hidden states at that time step. Clearly define your operations.
  A BiLSTM can be expressed as $[h_1, \dots, h_T] = \text{Concat}\left(\text{LSTM}([x_1, \dots, x_T]), \text{LSTM}([x_T, \dots, x_1])\right)$ where $\text{Concat}([a_1, \dots, a_T], [b_1, \dots, b_T]) = \left[\begin{pmatrix} a_1 \\ b_1 \end{pmatrix}, \dots, \begin{pmatrix} a_T \\ b_T \end{pmatrix}\right]$. We can then train sequence tagging via a softmax loss: $L([x_1, \dots, x_T], y_t) = w_{y_t}^T h_t - \log \sum_y \exp w_y^T h_t$.

LSTM is supposed to alleviate the vanishing and exploding gradient problem. Let us consider a loss function that only depends on the final step $L(c_T)$ where T is the final step.

- **(5pts)** Express $\frac{dL}{dc_t}$ in terms of $\frac{dL}{dc_{t+1}}$, $\frac{dL}{dh_t}$, and non-recursive terms.

$$\begin{aligned}\frac{dL}{dc_t} &= \frac{dL}{dc_{t+1}} \frac{dc_{t+1}}{dc_t} + \frac{dL}{dh_t} \frac{dh_t}{dc_t} \\ &= \frac{dL}{dc_{t+1}} \odot f_t + \frac{dL}{dh_t} \odot o_t \odot \tanh'(c_t)\end{aligned}$$

- **(5pts)** By analyzing your expression for $\frac{dL}{dc_1}$, explain why the long term dependency does not vanish when the forget gate is $f_t = 1$.
  Assuming the forget gate is 1, $\frac{dL}{c_1} = \frac{dL}{c_T} + \dots$ so rather than vanishing, the gradient due to the initial state is around the same magnitude as that due to the final state, so the long-term dependency will be reflected in the loss function.

(25pts) Modular implementation of neural networks

Modules makes it easy to construct neural networks with various structures. Each module supports the `forward` and `backward` functions, might maintain internal state, and might be be composed of other modules. `Linear` and `Sequential` are two examples.

```
class Linear(Module):
    def __init__(self, W, b):
        self.W, self.b = W, b

    def forward(self, input):
        return np.dot(self.W, input) + self.b

    def backward(self, gradout):
        return np.dot(self.W.T, gradout)

class Sequential(Module):
    def __init__(self, children):
        self.children = children

    def forward(self, input):
        for child in self.children:
            input = child.forward(input)
        return input

    def backward(self, gradout):
        for child in reversed(self.children):
            gradout = child.backward(gradout)
        return gradout
```

(3pt) While `Linear` is computing the right outputs, it is missing a step that would enable learning in `Linear` itself. What step is missing? What else needs to be stored in order to perform this step?
A step to compute and store the gradients needed for its own parameter updates is missing in `backward`.

(5pt) Implement the `Sigmoid` module which computes the element-wise function $\sigma(x) = \frac{1}{1+\exp(-x)}$

```
# pseudo code or python code are both acceptable
# clarity and precision is required, being able to run is not required
class Sigmoid(Module):
    def forward(self, input):
            self.output = 1.0 / (1.0+np.exp(-input))
            return self.output

    def backward(self, gradout):
            return self.output * (1.0-self.output) * gradout
```

(5pt) Implement the `Dropout` module (space below)

```
class Dropout(Module):
        def __init__(self, p):
        self.p = p # with prob. p, the input stays
    def forward(self, input):
            self.keep = (np.random.rand(*input.shape) < self.p).astype(input.dtype)
            return self.keep * input

    def backward(self, gradout):
            return self.keep * gradout
```

(7pt) Implement `EMul` that takes some children modules, give all children the same inputs, and outputs the elementwise product of the outputs of all children.

```
class EMul(Module):
    def __init__(self, children):
        self.children = children

    def forward(self, input):
            self.factors = [child.forward(input) for child in self.children]
            self.output = np.copy(self.factors[0])
            for factor in self.factors[1:]:
                    self.output *= factor
                return self.output

    def backward(self, gradout):
                return self.output * sum(child.backward(gradout)/factor for child, factor in
```

(5pt) Use existing modules to implement the following network function by writing a single statement invoking constructors of `Sequential`, `EMul`, `Sigmoid`, and `Linear` given weights `W1, W2, W3`

$$\begin{aligned} a_1 &= \sigma(W_1 x) \\ a_2 &= \sigma(W_2 x) \\ z &= a_1 \odot a_2 \\ y &= W_3 z \end{aligned}$$

```
Network = Sequential(EMul(Sequential(Linear(W1, 0.0), Sigmoid()),
Sequential(Linear(W2, 0.0), Sigmoid())), Linear(W3, 0.0))
```

(15pts) Design an RNN

We would like an RNN to recognize a string of well-balanced parenthesis. Well balanced parenthesis are strings such as `(())` or `(())()` where

- the number of `(` is the same as the number of `)` in the whole string
- any prefix string does not contain more `)` than `(`.

Let $x = x_1, x_2, \ldots, x_T$ where $x_i \in R^{2\times 1}$ with $[0, 1]$ for `)` and $[1, 0]$ for `(`.
This RNN should perform the following updates

$$h_{i+1} = \text{relu}(Wh_i + Ux_{i+1})$$

In addition, we start with h_0 (dimension 3 with 0 initial value is recommended, but you are free to use anything you like). At the end, a prediction $y = Vh_T$ is made, where $y \leq 0$ means balanced and $y > 0$ otherwise.

- **(7pts)** Clearly state your strategy for the RNN
  Store the number of times each character has been seen in first two dimensions. At each step compute the difference between the two and store in the last two dimensions (#`(`-#`)` and #`)`-#`(`). Make the fifth dimension positive whether or not a prefix string has ever had more `)` than `(` (i.e. whether the fourth dimension has ever been positive). Then if any of the last three dimensions of h_T are positive the string is not well-balanced.

- **(8pts)** Specify explicit values of W, U, V to achieve it

$$h_0 = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}, \quad W = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 1 & -1 & 0 & 0 & 0 \\ -1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{pmatrix}, \quad U = \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 1 & -1 \\ -1 & 1 \\ 0 & 0 \end{pmatrix}, \quad V = \begin{pmatrix} 0 & 0 & 1 & 1 & 1 \end{pmatrix}$$

- Solution with 3 states

- 3-state solution

$$h_0 = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}, \quad W = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & -1 & B \end{pmatrix}, \quad U = \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 1 & -1 \end{pmatrix}, \quad V = \begin{pmatrix} 1 & -1 & B \end{pmatrix}$$

  h_3 stores if the sequence has failed for having too many `)`, which will remain positive once it got to be positive B (B needs to be big enough e.g. B=2). If sequence did not fail for too many `)`s, $Vh_T > 0$ when there are more `(` than `)`.