ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Natural Language Processing
Spring 2021

Prof. Ryan Cotterell

Course Assignment Episode 1

22/04/2021 - 16:01h

Question 1: Backpropagation (6 pts)

Consider the following fully-connected network $f : \mathbb{R}^3 \to (0, 1)$ used for binary classification:

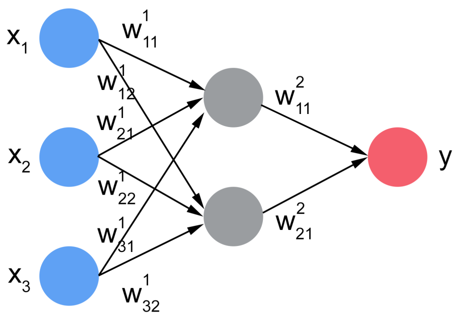


Figure 1: Neural Network f

where:

- All hidden units have ReLU activation.
- The output layer has sigmoid activation.
- There are no biases in the layers.
- All weights are initialised as ones.

a) Draw the computation graph of f. You can use $\mathit{ReLU}()$ and $\sigma()$ blocks in your drawings.

b) Consider binary cross entropy loss:[1]

$$L_{BCE} = -(y \log(\hat{y}) + (1 - y) \log(1 - \hat{y})) \tag{1}$$

where $\hat{y} = f(\mathbf{x})$ is the prediction from our network and y is the true value. Consider the following data-point: $(\mathbf{x}_0, y_0) = ((1, 1, 1), 0)$

i) Evaluate the graph of a) at $(\mathbf{x}_0, y_0)$ to compute the value of $\hat{y}$.

ii) Compute partial derivatives for the network weights using the graph at point a) then evaluate them at $(\mathbf{x}_0, y_0)$.

[1]This loss is usually defined with respect to a collection of data points; we define it with respect to a single data point here for simplicity

iii) Compute the values of L_{BCE} and $\frac{\partial L_{BCE}}{\partial \hat{y}}$ at the point $(\mathbf{x}_0, y_0)$.

iv) Run a step of weight optimisation over f following a standard gradient descent (GD) optimization procedure with learning rate $\eta = 0.1$.

c) How could we change our computation graph from b.ii. to reduce the number of computations while producing the same result? Show your improvements via a revised computation graph. Besides improving efficiency, what other benefits might this change have?

Question 2: Log-linear models (10 pts)

In this problem, you will implement your own binary log-linear model class. A skeleton class (in Python) has been provided for you (see `assignment_log_linear_models.ipynb`); while you are not required to use Python for this component, all attributes of the class shown in this file must be implemented for full credit.

You will compare your model implementation to that of a standard Python machine learning library: sklearn. Use the sklearn `make_classification` function to create 3 different datasets with increasing sample sizes (n_samples $= 100, 1000, 10000$) and varying number of informative features; all parameters are specified in the `assignment_log_linear_models.ipynb` file. If you are not using Python, you may export these datasets to, e.g., a csv file, to use with your model.

You are required to fill in the above jupyter notebook with your answers to this question, i.e., all of your code as well as your plots/tables and explanations comparing the aspects in b). Note that there are various kernels available for Jupyter notebooks, which will allow you to use one with virtually any programming language you want. Make sure your code is well documented and understandable, especially if you are using a language other than R or Python since we may not be able to give partial credit if your code does not run properly and we can not infer what you were trying to do.

(a) Implement your log-linear model class in an object-oriented programming language of your choice with the following attributes:

(i) A constructor, which takes as arguments
- A feature function $f : \mathcal{X} \times \mathcal{Y} \rightarrow \mathbb{R}^m$
- A fixed learning rate η to be used by gradient descent
- The number of iterations to run gradient descent for during `fit`
- A loss function
- The gradient of the loss function
- The verbosity level of the class

(ii) A `fit` method, which uses gradient descent to estimate the parameters $\boldsymbol{\theta}$ of the model. If `verbose=True`, the class should print the following during `fit` after each iteration of gradient descent.
- The number of the current iteration (e.g., `Iteration:  1 / 100`)
- The change in loss after this gradient step
- The largest absolute change in any of the single parameters $\theta_i \in \boldsymbol{\theta}$ after this gradient step

(iii) A `gradient_descent` method, which will be used by the `fit` method and updates model parameters based on the gradient of the loss function.

(iv) A `predict` method, which takes inputs X of the same format as `fit` and provides predictions using the current model parameters.

Please refer to `assignment_log_linear_models.ipynb` for an overview of all required methods and attributes (including type hints).

(b) You will now compare the performance of your custom log-linear model class to that of Python's sklearn implementation of logistic regression, `LogisticRegression`. Please initialize `LogisticRegression` with its' default parameters (i.e., don't set any parameter explicitly such that they all get initialized to their defaults).

For your own model, use the negative log-likelihood as the loss function (and thus the gradient of the negative log-likelihood as the gradient of the loss function); recall that this implies the parameters of your log-linear model will be estimated using maximum likelihood. Further, initialize your model so it runs gradient descent for 100 iterations during `fit`.

Fit each of the models on all datasets generated in the `assignment_log_linear_-models.ipynb` file, using a train-test split (use `train_test_split` from sklearn or the equivalent in the language of your choice) of 80% training data and 20% testing data. Compare the following aspects between the two models:

(i) Computation time (both training time and prediction time, separately)

(ii) In-sample accuracy (i.e., accuracy on the training set)

(iii) Out-of sample accuracy (i.e., accuracy on the test set)

(iv) Coefficient values

You can freely choose the exposition of these comparisons (one plot for each of the above aspects plus a short paragraph is sufficient). Identify some of the possible reasons for the discrepancies in performance (if any).

NB: If you are not using Python, please compare your model to a canonical implementation in the language of your choice (e.g., `glm` in R).

Question 3: Skip-Gram (10 pts)

The goal of this problem is to implement one of the two word2vec variants: the skip-gram model, presented in lecture 4.

We provide starter code in Python (see `assignment_skip_gram.ipynb`). You are **required to fill in this notebook and return it for grading, along with the embedding matrix you compute**. Your code should be clear, commented and executable; we cannot give partial credit if we cannot understand your code! We encourage you to do the exercise using Python, however you are free to choose another language for which there is a notebook kernel available. If you use a language other than Python, your code has to be very well commented and must run without any issue; we may not be able to give partial credit for questions done incorrectly when the language is one the TA team is not familiar with. Note that, this exercise also contains theoretical questions that you have to answer in the notebook using Mardown syntax. Finally, note that you must code all the algorithms for this problem; you cannot use external libraries that compute word embeddings for you.

Question 4: Language modeling (10 pts)

Mary is exploring a very simple language in which strings are formed by sequentially drawing each token from the vocabulary V independently and uniformly at random. Mary wants to construct a corpus in this language. She does so by sampling $x_1, x_2, x_3, \dots$ from her vocabulary until the corpus is the size she wants.

(a) Mary draws n tokens to form her corpus.

(i) What is the expected number of distinct tokens that appear?

(ii) What is the probability that all $|V|$ of the words from the vocabulary have appeared?

(b) Mary wants the bigram *work hard* to appear in the corpus.

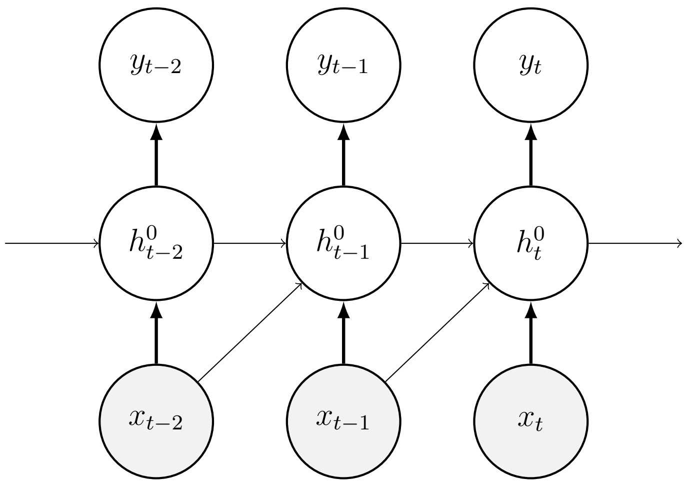
(i) What is the expected number of tokens that will be drawn from the vocabulary before this bigram appears?

(ii) Mary decided that this number is quite big. Instead, she would at least like to ensure that the word *work* appears in the text with probability $\geq 95\%$. What is the number of tokens she needs to draw before this happens?

(c) One thing Mary dislikes is when the same words appear next to each other in her corpus.

(i) What is the expected number of draws before two of the same tokens appear next to each other?

(ii) Mary doesn't trust her computation from the previous part and decides to implement a neural network that, at each step, will output the number of bigrams encountered so far that consist of two of the same token. She proceeds as follows. First, she builds a network that will output 1 whenever the current bigram (consisting of the previous word and the current one) contains two of the same token. She uses the architecture from the figure below. Your job is to help her find the unknowns.



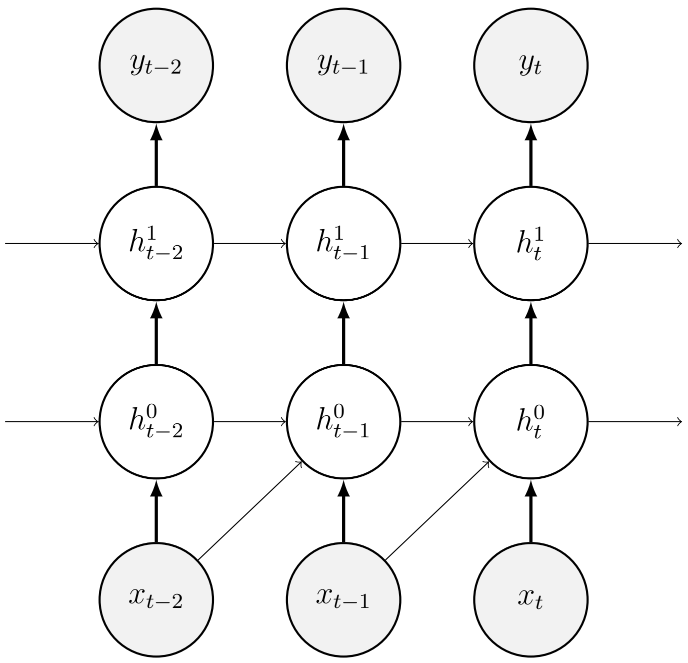
Formally, we have

$$h^0_t = f(w_0 x_{t-1} + w_1 x_t + w_2 h^0_{t-1} + b_0)$$

$$y_t = g(w_3 h_t^0 + b_1)$$

Find all of f, g, w_i for $i \in \{1, 2, 3\}$ and b_j for $j \in \{0, 1\}$. What kind of activation function f is needed and why? Can we set f to be a linear function of the input?

(iii) Now she would actually like to finish her problem and build what is needed using the architecture from the figure below. As in the previous problem, your job is to find the unknowns.



Where we can formally define the components of the network as

$$h_t^0 = f(w_0 x_{t-1} + w_1 x_t + w_2 h_{t-1}^0 + b_0)$$
$$h_t^1 = g(w_3 h_{t-1}^1 + w_4 h_t^0 + b_1)$$
$$y_t = h(w_5 h_t^1 + b_2)$$

(iv) Mary is considering having a non-uniform unigram distribution to avoid the problem of seeing two of the same tokens next to each other. Consider a single bigram. Which model has a bigger chance of sequentially drawing two of the same token, a uniform unigram language model or a non-uniform unigram language model?

Question 5: Dijkstra's algorithm (10 pts)

(a) Recall from lecture that we define the *decoding problem* with respect to some scoring function score$(\cdot, \mathbf{x})$ as the problem of finding the highest scoring item $\mathbf{y}^\star$. Formally, this can be written as:

$$\mathbf{y}^\star \leftarrow \underset{\mathbf{y}' \in \mathcal{Y}}{\text{argmax}} \; \text{score}(\mathbf{y}', \mathbf{x}) \tag{2}$$

In lecture, you learned how to decode from a Conditional Random Field (CRF) using the Viterbi algorithm. The Viterbi algorithm, however, is not the only one that can be used for solving this task. In fact, if you interpret the decoding problem as a shortest path problem, you can use many well-known algorithms: Dijkstra's, Bellman-Ford, Floyd-Warshall, and many others! In this problem, you will explore how to perform decoding for a CRF using Dijkstra's algorithm. Consider a first order CRF that outputs scores over sequences in the output space $\mathcal{Y}$ of length N, where the sequence consists of elements from some set Y, e.g., POS-tags:

$$p(\mathbf{y} \mid \mathbf{x}) = \frac{\exp\{\sum_{n=1}^{N} \text{score}(\langle y_{n-1}, y_n \rangle, \mathbf{x})\}}{\sum_{\mathbf{y}' \in \mathcal{Y}} \exp\{\sum_{n=1}^{N} \text{score}(\langle y'_{n-1}, y'_n \rangle, \mathbf{x})\}} \tag{3}$$

(i) We will represent our problem using a graph $G = \langle V, E, W \rangle$, where every path through G is some $\mathbf{y} \in \mathcal{Y}$. State what the vertices V, edges E and weights W should be in terms of the defined variables. Provide pseudocode for solving the decoding problem for Eq. 3 using a prioritized[2] version of Dijkstra's algorithm.

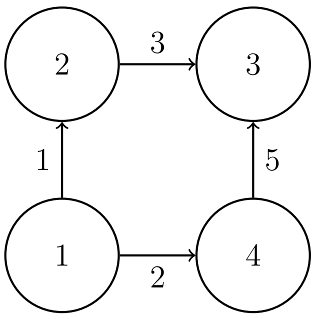
(ii) What is the time complexity of your algorithm in terms of the cardinality of the sets V and E? Compare this to the time complexity of the Viterbi algorithm. Is either strictly better than the other?

(b) Dynamic programming algorithms rely on the distributive property to efficiently compute quantities over a large number of terms. Dijkstra's algorithm, despite generally being classified as a greedy algorithm, can also be seen as a dynamic programming algorithm, thus it can be semiring-ified.

(i) Rewrite your pseudocode from part a(i) in semiring notation and identify the semiring that was used.

(ii) Give a suitable semiring that can be used with your pseudocode from part b(i) for computing the weight of the longest path between 2 nodes in a weighted directed graph. You may assume that all the weights are negative and you can define your own comparator operation that is used by the priority queue to determine in what order the nodes should be explored.

(iii) The widest path problem involves finding a path between two vertices by maximizing the weight of the minimum-weight edge in the path. For instance, consider the following graph:



There are 2 paths from vertex 1 to vertex 3: $1 \to 2 \to 3$ and $1 \to 4 \to 3$. The widest path between vertices 1 and 3 is $1 \to 4 \to 3$ because the smallest weight on the path (2) is larger than the smallest weight on the path $1 \to 2 \to 3$ (1).
Give a suitable semiring that can be used with your pseudocode from part b(i) for computing the weight of the widest path between 2 nodes in a directed graph. You may assume that all the weights are non-negative.

[2] A prioritized Dijkstra's algorithm is the version of the algorithm that uses a priority queue.