# CS4740/5740 Introduction to NLP
# Fall 2017
# Neural Language Models and Classifiers

Final report: due via Gradescope & CMS by Friday 12/1 11:59PM

## 1 Overview

In this project you will gain familiarity and working knowledge of neural language models. This time **we provide the bulk of the code** in Python, using the `dynet` and `pytorch` libraries. You will have to understand the code, extend it slightly, and run various experiments for the report. Choose your library among the two, and make sure you have the latest release installed. We recommend `dynet` for NLP tasks, since it is both easier to code for and about 2-3x faster on CPUs.

**Useful reading.**

- Chapter 8 in Jurafsky & Martin, 3rd ed. Especially 8.3 and 8.5. (Note that Chapter 8 is **required reading** anyway, so do read it. (`https://web.stanford.edu/~jurafsky/slp3/8.pdf`)
- Graham Neubig's slides (`http://phontron.com/class/nn4nlp2017/assets/slides/nn4nlp-02-lm.pdf`) and lecture notes (`http://phontron.com/class/mtandseq2seq2017/mt-spring2017.chapter5.pdf`)

**Note on other programming languages.** If you have been working in some other language, we **strongly encourage** giving Python a try for this assignment, since it allows you to complete the programming part of the assignment with writing probably 10 lines of code. Give it a shot, and if you still think you would rather work from scratch in a different language, come talk to us.

## 2 Dataset

We're back to movie reviews for this assignment, but we are providing a larger version of the dataset from Pang & Lee this time. **Do not** down-

load the full dataset from it's on-line site: we are providing you with a slightly reorganized version via CMS, in order to make for a more interesting project.

**Task 1.** Download the data and the code from CMS. Look through the `preprocess.py` file and understand what it does.[a] **What is the size of the preprocessed vocabulary?** Explain how you obtained your answer.

[a] Along the way, try to identify some things you were making overcomplicated in your own preprocessing in previous assignments.

# 3 Sentiment classification

We will first train a simple neural sentiment classifier using a very simple model: a *deep averaging network* (DAN) with one hidden layer.[1] Importantly, we're using this model as a generic introduction to how neural networks are constructed, trained and evaluated.

**What is a hidden layer?** One of the fundamental concepts in neural networks is that of the *hidden layer*. This is a transformation that takes an input vector of size $n$ (i.e., $x \in \mathbb{R}^n$) and produces an output $f(x) \in \mathbb{R}^m$ of size $m$. Most typically, hidden layers consist of an *affine transformation* followed by an element-wise nonlinearity

$$f(x) = \sigma(b + W \cdot x) \qquad W \in \mathbb{R}^{m \times n}, b \in \mathbb{R}^n$$

where common nonlinearities $\sigma$ include the hyperbolic tangent `tanh`

$$\sigma(z)_i = \frac{\exp(2z_i) - 1}{\exp(2z_i) + 1}$$

or the rectified linear unit `ReLU`. We'll stick with `tanh`.

**Word vectors.** Neural networks are based on continuous mathematical functions applied to numbers. This fits like a glove to data from continuous sensor measurements, or image data (pixel intensities), but natural language sentences come in the form of lists of discrete words

$$\text{sentence} = [w_1, w_2, ..., w_{\text{len(sentence)}}]$$

where, for simplicity, we can assume each word is represented by a non-negative integer index into a vocabulary: $w_i \in \mathbb{N}$. To get from this discrete representation to a vector of continuous values $x \in \mathbb{R}^n$, a common

[1] Iyyer, Mohit, Varun Manjunatha, Jordan Boyd-Graber, and Hal Daumé III. "Deep unordered composition rivals syntactic methods for text classification." In: ACL 2015.

approach is to limit the vocabulary size to a value $|V|$ and learn a matrix of word vectors $E \in \mathbb{R}^{|V| \times n}$. Then, we can use $E_w \in \mathbb{R}^n$ as a vector representation of word $w$.

**Special word indices.** Commonly, we reserve the first few indices as special word vectors for special meanings. In this assignment, we will use the index $w = 0$ for unknown words, $w = 1$ for the beginning-of-sentence marker, and $w = 2$ for the end-of-sentence marker. (Double-check this yourself with `preprocess.py`!)

**Deep Averaging Network.** A deep averaging network is characterized by taking, as input, the average of all word vectors in the input sentence

$$x = \frac{1}{\text{len(sentence)}} \sum_{i=1}^{\text{len(sentence)}} E_{w_i}$$

It then passes $x$ through several hidden layers. (In our case, just one.)

$$h = \sigma(b + W \cdot x)$$

Finally, since we are solving a binary classification problem (where 1 indicates a sentence from a positive review and a 0 indicates a sentence from a negative review), we compute a single score

$$z = b_{\text{out}} + w_{\text{out}} \cdot x$$

where $w_{\text{out}} \in \mathbb{R}^m$, $b_{\text{out}} \in \mathbb{R}$. We then convert $z$ to a probability by "squashing" it to the $[0, 1]$ interval

$$p = P(\text{positive}|\text{sentence}) = \text{logistic}(z) = \frac{1}{1 + \exp(-z)}$$

**Code:** Look at the `_predict` method in the `dan_classifier.py` script to see how this is implemented. The method takes a batch of labeled sentences and returns the predicted probability $p$ for each given sentence.

**Training.** We want to find the best model parameters ($E$, $W$, $b$, $W_{\text{out}}$, $b_{\text{out}}$). To do this, we optimize with respect to the **binary logistic loss**

$$\ell(\text{sentence}, y) = -y \log(p) - (1 - y) \log(1 - p)$$

where $y$ is the true label: $y = 1$ if the sentence is truly positive, else $y = 0$.

In this assignment we use **mini-batch training** which is faster: we group all of the training data into sets of (say) 32 sentences. We then compute the batch-level loss:

$$\ell(\text{batch}) = \sum_{(\text{sentence}, y) \in \text{batch}} \ell(\text{sentence}, y)$$

and make a parameter update with respect to this batch-level loss.

**Evaluation.** We will monitor the accuracy on the validation set after every full pass over all training batches. (A full pass is commonly called an *epoch*.) Keeping with the "batching" paradigm, we need a function that takes a validation batch and returns the **number of correctly classified sentences** (i.e., sentences for which $y = [[p > 0.5]]$ when the gold standard says that $y$ is of positive sentiment and sentences for which $y = [[p <= 0.5]]$ when the gold standard says that $y$ is of negative sentiment. A *stub* of this function is provided, but it always returns 0.

**Task 2.** Finish the implementation of the `num_correct` method. Train the model and plot the training loss and the validation set accuracy after each *epoch*. What do you observe?

**Overfitting and regularization.** If after a few epochs, the validation performance stops improving and it actually starts getting worse, but the training loss keeps decreasing, the model is overfitting. (Even if this is not the case in your experiment, you still need to complete this section.) To address this, it is common to *regularize* our model. A common regularization strategy for neural nets is *dropout*, which amounts to randomly setting some elements in a vector to 0, regardless of their value. In `dynet`, dropout can be applied to a vector using `dy.dropout` and in `pytorch` we may use `nn.functional.dropout`, look up the documentation of these functions for more details. **Warning:** Dropout should only be applied during training, but never when computing validation or test scores. Use the `train` argument to the `_predict` method to tell the difference.

**Task 3.** Apply dropout with probability 0.5 to each word embedding, prior to averaging. Run again (as in Task 2), and compare the results. (Add the curves to the same plots.)

# 4 Neural language models

In the previous sections, we trained a neural model to estimate the classification probability $P(\text{sentiment=positive}|\text{sentence})$. In this section, we will apply similar techniques to predict the *language model* probability: $P(w_k = j|w_{k-1}, w_{k-2}, w_{k-3}...)$. (Looks familliar?)

**We will start with a bigram model** $P(w_k = j|w_{k-1})$. We will use a very simple feed-forward model. We use, again, an embedding matrix $E$. The biggest differences from the DAN model is that (1) this time we make a prediction for each word, rather than for each sentence, and (2) instead of a binary prediction, we need to predict a probability for all $|V|$ possible

next words. Both these reasons mean that **the language model will be much slower to train than the classifier!**

At positition $k > 0$ in a sentence, a simple, bigram feed-forward language model performs the following steps:

1. Get the embedding of the previous (context) word $w_{k-1}$: $c_k = E_{w_{k-1}}$
2. Pass the embedding through a hidden layer: $h_k = \sigma(b + W \cdot c_k)$
3. Compute a non-normalized score vector $z_k = W_{\text{out}} \cdot h_k \in \mathbb{R}^{|V|}$
4. Compute the normalized probabilities using a softmax (or, equivalently, in log space)

$$P(w_k = j | w_{k-1}) = \frac{\exp(z_j)}{\sum_{i=0}^{|V|} \exp(z_i)}$$
$$\log P(w_k = j | w_{k-1}) = z_j - \log \sum_{i=0}^{|V|} \exp(z_i)$$

For training, the loss at word $k$ is given simply by

$$\ell_k(\text{sentence}) = -\log P(w_k = y | w_{k-1})$$

where $y$ is the index of the actual word observed at position $k$. This is also known as the *negative log-likelihood* (NLL). The batch loss is the sum of all word-level losses in the batch

$$\ell(\text{batch}) = \sum_{\text{sentence} \in \text{batch}} \sum_{k=1}^{\text{len(sentence)}} \ell_k(\text{sentence})$$

**Code:** Study the implementation of `batch_loss` in `simple_lm.py`.

**Evaluation.** As usual, we evaluate a language model using perplexity. Unlike the classifier, where we needed a separate method for evaluation, here we can reuse `batch_loss(..., train=False)` because of a **fundamental connection** between perplexity and the total NLL:

$$\text{NLL(dataset)} = \sum_{\text{batch} \in \text{dataset}} \ell(\text{batch}) = \sum_{\text{sentence} \in \text{dataset}} \sum_{k=1}^{\text{len(sentence)}} \ell_k(\text{sentence})$$

**Task 4.** What is the (mathematical) connection between NLL(dataset) and perplexity? (Hint: this is already used in the validation part of the provided code.) Given this connection, how can we interpret what our language model is minimizing over the training set?

**Using unlabeled data.** An observant reader will notice that we are not using the sentiment labels at all in training our language model.[2] This means we can throw the unlabeled data in the mix and maybe get a better language model.

> **Task 5.** Load the unlabeled data (use the commented code to help you) and combine it with the labeled data, then train a language model on the resulting bigger dataset. Plot the validation perplexity for the two cases (i.e., with and without the extra unlabeled data) on the same graph, and discuss what you observe. **Do not** plot the training perplexities on the same plot: they are not comparable to each other. Why are they not comparable?

**Generalization to n-gram contexts.** Predicting the next word using only one previous word seems a bit simplistic. How much better can we get by incorporating wider contexts?

To accomplish this, we will simply concatenate the word embeddings in the context. Concretely, to get a 3-word context (and thus essentially a 4-gram LM), we replace $c_k = E_{w_{k-1}}$ with $c_k = [E_{w_{k-1}}, E_{w_{k-2}}, E_{w_{k-3}}]$ where $[\ ]$ denotes concatenation.[3] (`dy.concatenate`/`torch.cat`).

> **Task 6.** What is the dimension of $c_k$, as a function of the embedding dimension $n$, when using $p$ words in the context? Assuming we don't want to make $W_{\text{out}}$ any bigger, how must we change the size of $W$?
>
> Using these observations, extend the implementation to a 3-gram and a 4-gram LM (i.e., context windows of 2 and 3 words, respectively). Train under both scenarios: labeled data only, then using all data.
>
> Report the best perplexities achieved for each configuration (context size = 1, 2, 3; unlabeled=True/False). Discuss the results.

# 5 Reusing learned word embeddings

In this section we will investigate whether we can get better performance in **sentiment classification** if we initialize the word embeddings in DAN with the values learned by the **language model**.

The provided language model scripts always save the trained embeddings at the end. We may use the saved embeddings to initialize our DAN

[2] Unlike in Project 1, we're not seeking separate positive and negative language models.

[3] This leaves open the question of what to do if, e.g., $k-2$ falls before the beginning of the sentence. You can pick any reasonable approach here, we suggest using the index of the end-of-sentence token.

classifier. In `dynet`, use `clf.embed.populate(filename, "/embed")`, and in `pytorch` use `clf.embed.weight = torch.load(filename)`.

**Task 7.** For every different language model trained in the previous section, train a DAN sentiment classifier preinitialized with the corresponding embeddings. Discuss the validation accuracy.

Make note of what the best embeddings are **according to the validation set**. Modify `dan_classifier.py` to print the test accuracy. Report the test accuracy of the model without pretrained embeddings, and with the best embeddings. **Do not** use the test scores to guide any decisions, and do not train anything further after seeing test scores.

# 6 Report

The required tasks are highligted with borders in this PDF. Follow the instructions in the boxes to the letter. Make sure to answer all additional questions: every question mark inside a border of this PDF must have a clearly-worded answer in your report.

**You will not** submit any code. Instead, every time a task requires you to modify the code, indicate in your report what changes you made. (Don't rely on line numbers, because as you modify the code the offsets can change. Keep changes minimal, don't include plotting code.) For instance, to explain how you load the test set, you'd write something like[4]

> **Code modification.** Right after the lines
>
> ```
> with open(os.path.join('processed', 'valid_ix.pkl'),
>           'rb') as f:
>     valid_ix = pickle.load(f)
> ```
>
> we added
>
> ```
> with open(os.path.join('processed', 'test_ix.pkl'),
>           'rb') as f:
>     test_ix = pickle.load(f)
> ```

**Rough grading rubric**

- Task 1: 10p
- Task 2: 15p
- Task 3: 10p
- Task 4: 10p
- Task 5: 15p
- Task 6: 15p
- Task 7: 15p
- Writing (clarity, quality, attention to detail): 10p

[4] We recommend using the *minted* package in LaTeX for nice-looking code.