# CSEP 517: Homework #3
# Text Classification with Perceptrons and Neural Networks

Due Sun. March 1, 2020 at 11:59pm.

**Submission instructions** Submit 2 files on Canvas:

1 **Code:** (`HW3.tgz`): You will submit your code together with a neatly written README file to instruct how to run your code with different settings. Code should use the provided scaffold and be written in Python. We assume that you always follow good practice of coding (commenting, structuring) and we will not grade your code based on such qualities of code writing practice.

2 **Report:** (`HW3.pdf`): You will also submit a typed report in the pdf format. The report will typically be about 3-4 pages in total. No restriction on font sizes, page margins, or number of pages. Consider technical writing as part of your educational training and organize your report neatly and articulate your thoughts clearly. We prefer quality over quantity. Thus, do not flood the report with tangential information such as low-level documentation of your code that should rather belong to the comments of your code or perhaps to the README file that you submit with the code. Similarly, when discussing the experimental results, do not copy and paste the entire system output directly to the report. Instead, consider creating tables and figures to organize the experimental results. When writing about *"error analysis"*, an important practice in any NLP project, try to make concrete arguments through examples.

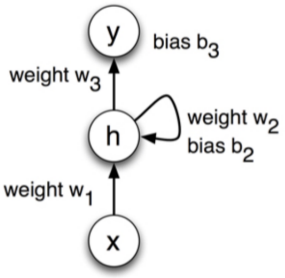
## [Written] Part 1: Recurrent Neural Networks (20 Points)

Consider a simplified RNN unit, in which input $x_t$, hidden state $h_t$, and output $y_t$ are all scalars, (i.e. $x_t$, $h_t$, $y_t \in \mathbb{R}$ for $t = 1, 2, ...$). $h_t$ and $y_t$ are computed with the following equations:

$$h_t = f(w_1 x_t + w_2 h_{t-1} + b_2)$$

$$y_t = g(w_3 h_t + b_3)$$

where $f$ and $g$ are activation functions.



Suppose that $f$ is a binary threshold function for its nonlinearity:

$$f(z) = \begin{cases} 1 & \text{if } z > 0 \\ 0 & \text{if } z \leq 0 \end{cases}$$

and $g$ is the identity function $g(z) = z$ and the input to this RNN $x_t$ is binary-valued ($x_t \in \{0, 1\}$). The hidden unit has an initial value of 0: $h_0 = 0$.

**Q1 (10 points) Provide a set of values for the weights and biases** ($w_1$, $w_2$, $b_2$, $w_3$, $b_3$) so that the RNN initially outputs 1, but as soon as it gets in input of 0, it switches to outputting 0 for all subsequent timesteps. For example, the sequence of inputs 1110101 produced the outputs 1110000. **Provide an explanation for how you choose the values of the weights and biases.**

**Q2 (10 points) Provide a set of values for the weights and biases** ($w_1$, $w_2$, $b_2$, $w_3$, $b_3$) so that the RNN outputs the opposite binary value from the input $x_t$; for example, an input of 0 produces an output of 1, and an input of 1 produces a 0, and the sequence of inputs 11001010 produced the outputs 00110101. **What effect do these weights have on the RNN?**

# [Programming] Part 2: Sentiment Classification with Perceptrons (40 points)

In this programming assignment, we will develop some basic models for sentiment analysis. In particular, we will work with a subset of the Stanford Sentiment Treebank (SST) [3]. Feel free to explore the larger dataset on the SST website.

## Dataset

Sentiment analysis entails analyzing text to infer the sentiment portrayed by the text. For example, given a movie review such as:

> This movie was actually neither that funny, nor super witty

We might say that the writer was not impressed by the movie because they described it as not funny nor witty. The SST dataset consists of movie reviews as well as their annotated sentiment. The original dataset consists of 5 fine-grain sentiment categories (very negative, negative, neutral, positive, and very positive). For our purpose, we have kept only positive and negative sentiment examples and further binarized these into two categories (negative and positive).

## Preliminaries

Please use Python 3.5+ for this project. This part requires PyTorch, a state-of-the-art deep learning package written in Python. Historically, hand-crafting features to incorporate domain expert knowledge has been a crucial to achieving good performance in NLP. In recent years, neural methods that leverage vast quantities of data, especially unlabelled data in the form of word embeddings and language models, have proven exceptionally useful for a wide variety of tasks. In this part of our assignment, we will write a Perceptron classifier and try our hand at hand-crafting features for the classifier. In the third part of our assignment, we will write neural models that leverage pretrained word embeddings.

### The Perceptron Classifier

Get started, we will implement a Perceptron classifier with unigram features. We will first need a way to map from a sentence (e.g. a list of strings) to a feature vector. A unigram feature vector is a sparse feature vector that indicate which words are present in the input sentence. There is no one right way to define unigram features. For example, do we want to throw out low-count words? Do we want to throw out stopwords? Do we want to lowercase words? For our purpose, we will extract features "on-the-fly" and define weights for these features as we grow our vocabulary. Note that we can use the native python class `Counter` to store elements of the sparse vector.

**Q3 Unigram perceptron (15 points)**
Implement **UnigramFeatureExtractor** and **PerceptronClassifier**. You can check your implementation by running `python sentiment_classifier.py --model PERCEPTRON --feats UNIGRAM`. The training and evaluation time (printed) should take less than 1 minute. You must get at least 74% accuracy on the development set. Feel free to change the `--learning_rate` and `--epoch`. **Report the top-10 unigrams with the highest perceptron weights and the top-10 unigrams with the lowest perceptron weights. Attach the resulting `test-blind.output.txt` as `unigram.output.txt` to your submission.**

**Q4 Learning curve (10 points)**
Set `--epoch` to 20, `--learning_rate` to 1e-3 and attach a plot or table of the train and development accuracy with respect to epoch number. How does the training and development accuracy change with respect to epoch? **Write 1-3 sentences describing what you see. Do you think development accuracy will significantly increase if we keep increasing `--epoch`?**

**Q5 Better features (15 points)** Unigram features are very basic. Can you think of some better features? For example, you extract higher-order ngrams, have features on first word of the sentence etc. Implement **BetterFeatureExtractor**. You can check your implementation by running `python sentiment_classifier.py --model PERCEPTRON --feats BETTER`. **Describe what you did and how its performance compares to unigram features. Set `--epoch` to 20, `--learning_rate` to 1e-3 and attach a plot or table of the train and development accuracy with respect to epoch number. Attach the resulting `test-blind.output.txt` as `better.output.txt` to your submission.**

## [Programming] Part 3: Sentiment Classification with Neural Models (40 points)

In this part, we will write neural models that leverage pretrained word embeddings. To make it easier to use word embeddings, we will use the **embeddings Python library**. Note that this library should have been installed when we installed dependencies. However, when we instantiate an embeddings class, the library will download a substantial amount of data corresponding to the word embedding parameters. In this case, we will use 300-dimensional GloVe vectors from Wikipedia and Gigawords [2]. The total download is around 850MB, and the expanded data files are around 1.5 GB. The data should be stored in your `/.embeddings` directory. You can optionally symlink this folder to another folder if you'd like to move the data location.

Our models will be implemented using PyTorch. There are many tutorials online on how to use PyTorch, including official ones from the PyTorch website. Feel free to have a look if you are interested, however this should not be necessary to complete this assignment because we will limit ourselves to a very small set of PyTorch functionalities.

**Q6 Multi-layer perceptron (15 points)**
Implement a feed-forward neural network (also named multi-layer perceptron) in **FNNClassifier**. In particular, implement a two layer model that consists of a 100-dimensional linear layer with Tanh

activation and a single-unit output layer with sigmoid activation. The model instantiation should be done in the `__init__` function.

Next, implement the `forward` function that takes in a word embeddings for each word, sums them, then gives this as input to the two-layer network. Note that PyTorch models anticipate a batch size which we do not use for this assignment. To get around this, we use the `unsqueeze(0)` function on the features to make a fake batch dimension.

Like we did for **PerceptronClassifier**, we'll implement the `extract_pred` function and `update_parameters` function. Note that for `update_parameters`, we should use binary cross entropy loss in Pytorch. The update should follow the procedure:

1. compute loss
2. run backprop
3. perform update with the optimizer `self.optim` (in this case we are using the ADAM optimizer [1])
4. clear the gradients

HINT: You might find the PyTorch docs helpful here, particularly docs on `nn.Linear`, and `nn.functional.binary_cross_entropy`. The Pytorch semantics for binary cross entropy is confusing. If your network activation already has been put through a sigmoid, you want `binary_cross_entropy`. If it has not been put through a sigmoid, you want `binary_cross_entropy_with_logits`.

You can check your implementation by running `python sentiment_classifier.py --model FNN`. **Describe what you did and how its performance compares to unigram features. Set `--epoch` to 10, `--learning_rate` to 1e-3 and report a plot or table of the train and development accuracy with respect to epoch number. Attach the resulting `test-blind.output.txt` as `fnn.output.txt` to your submission.**

**Q7 Recurrent neural networks (15 points)**

Next, we will implement **RNNClassifier**. Recurrent neural networks are a more powerful class of neural networks that propagate sequential information through time. Implement a 20-unit bidirectional LSTM model that runs on the sequence of word embeddings. Next, aggregate the LSTM output states by taking the maximum activation during each time step. Feed this "max-pooled" activation to a linear layer with 40 units followed by sigmoid activation.

HINT: when specifying the LSTM, use `bidirectional=True` and `batch_first=True`. You may find the docs on `nn.LSTM` and `Tensor.max` helpful.

You can check your implementation by running `python sentiment_classifier.py --model RNN`. **Describe what you did and how its performance compares to unigram features. Set `--epoch` to 10, `--learning_rate` to 1e-3 and report a plot or table of the train and development accuracy with respect to epoch number. Attach the resulting `test-blind.output.txt` as `rnn.output.txt` to your submission.**

**Q8 Experiment with your own model (10 points)**

Try your own neural networks model and see how far you can push the performance. You can check your implementation by running `python sentiment_classifier.py --model MyNN`.

**Describe what you did and how its performance compares to RNNClassifier. Set `--epoch` to 10, `--learning_rate` to 1e-3 and report a plot or table of the train and development accuracy with respect to epoch number. Attach the resulting `test-blind.output.txt` as `mynn.output.txt` to your submission.**

# References

[1] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *ICLR*, 2015.

[2] Jeffrey Pennington, Richard Socher, and Christopher D. Manning. Glove: Global vectors for word representation. In *EMNLP*, 2014.

[3] Richard Socher, Alex Perelygin, Jean Wu, Jason Chuang, Christopher D Manning, Andrew Y Ng, and Christopher Potts. Recursive deep models for semantic compositionality over a sentiment treebank. In *EMNLP*, 2013.