# Word Sense Disambiguation with Attention

**Huge thanks to Uri Sherman for coding, testing, and organizing this exercise!**

This exercise aims to dive into the details of the attention models learned in Lesson 2. We will use the task of word sense disambiguation (WSD) as an example scenario for where attention can be applied as a contextualization function. You will implement various forms of attention and use the performance on WSD, as well as visualization tools, to learn about how different forms of attention behave.

## Instructions

Follow the steps in the README and wsd_model.ipynb notebook. Before you submit download the notebook as html (File -> Download as -> HTML) and place it the attention_exercise directory. Zip the attention_exercise directory (without the data) and upload to moodle.

Name the zip file as your IDs separated by underscores. For example: 123_456.zip

Each group should submit only once.

## Introduction

## Coding

Most of the code you will need to implement will either be in `model.py` , or in the `wsd_model.ipynb` notebook (a.k.a. **the** notebook). The notebook will walk you through parts 1-3 of the exercise, after which you should be familiar enough with the code to implement the rest. Feel free to change any part of the code.

You will need a GPU to train the model. There are two options:

- Google Colab: free and easy to access from your browser. It is a good choice for this exercise. Remember to upload the three python files `data_loader.py` , `model.py` , and `traineval.py` for the notebook to be able to use them.
- The CS school also has a GPU cluster for courses. We will supply additional access instructions soon.

It is recommended to develop and debug locally, and only use Colab for the full datasets once you have a properly working model.

## Dataset

We will work on an annotated dataset called SemCor https://www.gabormelli.com/RKB/SemCor_Corpus. The dataset is comprised of a collection of sentences. We will consider each one of them as a seperate context. In each sentence, every word with a clear WordNet sense is labeled, resulting in roughly 30% annotated words. The remaining 70% of the words are given a "no_sense" label.

The SemCor dataset is already tokenized and split into train (80%), dev (10%) and test(10%) (see the build_dataset.py script).

## Loading the Data

If you look at `data_loader.py` , you will see 3 classes:

- Vocabulary: Maps words (token types) and senses to integers.
- WSDDataset: Serves query-based attention, in which an example consists of the 3-tuple (sentence, query, label). The query is the index (position in the sentence) of the word we want to disambiguate. The label is the annotated sense. All three tensors are integer encoded, and their string representation can be looked up using the vocabulary's decode() method. Note there is a special key for unlabeled words -- "no_sense". This implementation skips these examples.
- WSDSentencesDataset: Serves self-attention, in which an example consists of the 2-tuple (sentence, labels). In this case, the sentence itself serves as both the query and the context, and the labels provide an annotated sense for each word in the sentence.

In order to facilitate batching, the sentences (in both dataset implementations) and labels (in the 2nd one) are padded to the maximum sentence length using a special "PAD" token.

We use the following dimension variables:

- `B` : Batch dimension, will be 100 unless you change it.
- `N` : Sentence length, automatically computed by WSDDataset according to the longest sentence.
- `D` : Embedding dimension, set to 300 by default.

## Part 1: Query-Based Attention

Take a look at model.py. The WSD model is already initialized with embeddings and attention matrices with variable names following the notation from the lecture. Fill in your code at the "TODO Part 1" placeholders.

The `v_q` argument representing the query is optional, since we will use the same model to implement self-attention in Part 3. Don't worry for now about the mask argument passed to the attention function -- we'll deal with that later.

Use the notebook to train your model (with the specified hyperparameters), and plot the loss and accuracy to validate convergence.

Once the model has converged, visualize the model's attention using the provided API (demonstrated in the notebook). Is there anything weird about the attention?

## Part 2: Padding

Apparently, the model learns to attend to the padded indices as if they were real tokens. Padding is introduced to solve a technical problem -- fitting variable-length sequences into fixed-length tensors -- so the model should completely ignore these tokens. How can we make sure that the attention mechanism places zero weight on padding tokens?

Use the `mask` argument in your freshly implemented attention function to "zero out" any padding attention factors.

Hint: softmax is an exponential function; what input `z` will ensure that `exp(z) == 0` ?

## Part 3: Self-Attention

Change your implementation so that it can handle self-attention.

Recall that in self-attention, the model is given only the sentences matrix ( `M_s` ), i.e. the optional `v_q` argument is not passed. Take a look at the training / eval implementation -- the logic there switches between the two modes according to a flag on the dataset object: `sample_type='word'` or `sample_type='sentence'` . The notebook will take you through the simple process of converting the word level datasets to sentence level ones.

Follow the notebook to run training, loss inspection, and visualization of attention. What changed in the training process? How does the performance change?

**NOTE:** Depending on implementation, the visualization code might need to be changed as it currently supports only query-based attention models.

## Part 4: Position-Sensitive Attention

Extend your model to add relative positions. Specifically, implement the simplified version of (Shaw et al., 2018), as shown in the lesson.

Retrain, evaluate, and analyze the model. How does performance change? Use the visualization tool to verify that the model prefers to attend on local neighbors.

## Part 5: Causal Attention

How can we use relative positions to implement causal attention? Define a property of the relative position function such that the attention weights are causal, i.e. queries cannot attend on contexts that appear later in the sequence. Implement causal attention in your WSD model.

Retrain, evaluate, and analyze the model. How did causal attention affect the model's performance? Use the visualization tool to verify that the model does not attend on "future" tokens.