NN for NLP
Fundamental Concepts and Modelling Approaches

- Deep learning models have achieved remarkable results in computer vision and speech recognition in recent years.
- Within NLP, much of the work with deep learning methods has involved learning word vector representations through neural language models and performing compositions over the learned word vectors for classification tasks.
- Word vectors (words projected from a sparse encoding onto a lower dimensional dense vector space via a hidden layer), are essentially feature extractors that encode semantic features of words in specified number of dimensions.
- Words that are close—in euclidean or cosine distance—in the lower dimensional vector space are assumed to be semantically close (it actually works in practice).

NN - FUNDAMENTAL CONCEPTS

Tensors

- A tensor is a **generalization of matrices**: you could think of it as a matrix with higher dimensions (dim1, dim2, …, dim N) as opposed to regular matrix 2 dims: "rows" and "cols".
- Notice the potential source of confusion with the wording: "dimensions" in traditional matrix language are often also the actual values of the row and col ... DIMENSIONS! in tensor language.
- In NN we are almost always dealing with 3D tensors.
- Why? A **more compact representation** of data and **more efficient computations.**
- A clear example: for a 3D tensor comes when dealing with CNN on images. An image is given as a row x column matrix, but let's say you want to process a batch of these, then you need another dimension for your input so you end up with (batch size) *x* (row) *x* (columns): a 3D tensor. The most common dim number is 3-4 but there can be more.
- We could perfectly flatten the data into a regular matrix, perform computations and deal with the whole problem as a matrix. Just **better with tensors** for simplicity and efficiency.

Layers

- In general, a multilayer NN architecture takes the input sentence and learns several layers of feature extraction that process the inputs.
- The **first layer** extracts features for each word:
  - **Words are fed to our architecture** as indices taken from a finite dictionary. The first layer of our network maps each of these **word indices** into a feature vector via lookup table .
  - Given a task of interest, **a relevant representation of each word** is then given by the corresponding lookup table feature vector, which is trained by backpropagation, starting from a random initialization.
  - Feature vectors produced by the lookup table layer need to be **combined in subsequent layers** of the NN to produce a tag decision for each word in the sentence.
  - So, we use these trainable features as input to further layers of trainable feature extractors, that can represent groups of words and then sentences.

- The **second layer** extracts features from a window of words or from the whole sentence, treating it as some *sequence* with structure (i.e. not like a bag of words). Further layers are refinements of this.

Embeddings : what are thoooooose? version 745678900987654

- Assume there is a "brain" that receives inputs (say, texts) and gives it meaning. "Give meaning" here is assigning the message with a score in one or more compartiments. In principle, the number of compartiments (dimensions) is infinite -- an infinitely complex brain would decode information into extremely fine-grained "boxes" in order to interpret, represent and ultimately map sentences to actual meanings -> this is what embeddings are.
- Our human brains *seem* close to this extreme - being capable of processing and understanding infinite number of minutely distinct concepts, and each concept or sentence or document or anything that can carry a meaning can be represented as a point in it.
- Let's assume we have a not very smart brain that can only represent text messages within a 5-dimensional vector space.
- Given a text message, this brains assigns it a score of 0.61 in aspect (dim) 1, a score of 0.12 in dim 2 and so on until dim 5, and that' final 5dim vectors represent how it conveys meaning to it. Like this, every point becomes a 5-dimensional vector ... it *embeds data.*
- This brain's behavior is what NN mimic and word embeddings is how they can be given useful inputs to learn from.
- These are not random vectors (at least not initially) - the aim is that related words (e.g. or words used in a similar context) are close to each other while antonyms end up far apart in the vector space. What is "close" or "far" - some given notion of vector distance: cosine, euclidean norm, etc.
- Word embeddings are the **building blocks for using NN to do NLP**: nice starting point for training any NN taking text as its input since they capture similarity and relations.
- Word embedding algorithms are **unsupervised learning** techniques and thus can be trained on any corpus without the need for human annotation! Two common:
  - Word2vec (context and counts)
    - CBOW
    - Skip-grams
  - GloVe (bayesian probs)
  - FastText
- When using pre-trained embeddings, words not present in the set of pre-trained words are just initialized randomly.

PREDICTION

- **Producing tags** for each element in variable length sequences (sequence of words are often our sequences in NLP and the sequence size can vary from one to |V|) depending on the task at hand: PoS for one word? Predicting next word given k previous?, topic?, sentiment/toxicity of the whole text?.
- Common approaches: a word window approach, convolutional (CNN), recurrent (RNN) approaches:

- **Window approach:** word neighborhood conveys meaning

    - A window approach assumes the tag of a word depends mainly on its **neighboring words**. Given a word to tag, we consider a fixed size *ksz* (a hyper-parameter) window of words around this word. Each word in the window is first passed through the lookup table layer (1) or (2), producing a matrix of word features of fixed size $dwrd \times ksz$.
    - The fixed size vector can then be fed to one or several standard neural network layers (via affine, hardtanh, etc.).

- **CNN:** sentence conveys meaning

    - While a window approach performs well for most natural language processing tasks, it fails when the tag of a word depends on some predicate (for instance, in SRL). Now correctly tagging a word requires considering the *whole* sentence.
    - When using NN, the natural choice to tackle this problem becomes a convolutional approach. CNN use layers with <u>convolving</u> filters applied to the network inputs.
    - A **convolutional layer** can be seen as a generalization of a window approach: given a sequence represented by columns in a matrix $f\,l-1$ (in our lookup table matrix (1)), **a matrix-vector operation is applied to each window of successive windows** in the sequence.
    - Convolutional layers extract **local features around each window** of the given sequence. As for standard affine (linear) layers, convolutional layers are often stacked to extract higher level features, each followed by a non-linearity.

    - Local feature vectors extracted by the convolutional layers have to be combined to obtain a **global feature vector,** with a fixed size independent of sentence length, in order to apply subsequent layers.
    - Traditional CNN often apply an average or a max operation over the "time" (i.e. position in sentence in this context) t of the sequence . It depends on the task at hand which is best.
    - Some variants of the CNN model.
        - CNN-rand: common baseline model where all words are randomly initialized and then modified during training.
        - CNN-static: A model with pre-trained vectors from *word2vec*. All words—including the unknown ones that are randomly initialized—are kept static and only the other parameters of the model are learned.
        - CNN-non-static: Same as above but the pre-trained vectors are fine-tuned for each task.
        - CNN-multichannel: A model with two sets of word vectors. Each set of vectors is treated as a 'channel' and each filter is applied to both channels, but gradients are back- propagated only through one of the channels.

Hence the model is able to fine-tune one set of vectors while keeping the other static. Both channels are initialized with word2vec.

- **RNN**
  - Recurrent nets (RNN) is a class of neural networks whose connections between neurons form a directed cycle, and so, unlike feedforward NN, they can use their internal "memory" to process a sequence of inputs, which makes it popular for processing sequential information.
  - RNN are designed to recognize patterns in **sequences** of data (text, genomes, handwriting, speech, numerical times series). We could say they have a **temporal dimension**. Time, in this case, is simply expressed by a well-defined, ordered series of calculations linking one time step to the next.
  - RNNs are called *recurrent* because they perform the **same task for every element of a sequence, with the output being depended on the previous computations** - they have a "memory" which captures ("remembers") information about what has been calculated *so far*.
  - Recall that NN, recurrent or not, are simply nested composite functions like f(g(h(x))). Technically, adding a time element (going RNN) only **extends the series of functions for which we calculate derivatives** with chain rule.
  - Training a RNN is similar to training a traditional NN - we use backpropagation with a twist: the parameters are shared by all time steps in the network, so the gradient at each output depends not only on the calculations of the current time step, but also the previous time steps. That's what *backpropagation through time*, or BPTT does.
  - Most common RNN variant:
    - Long short-term memory (LSTM): strategically save previous info as opposed to using all the info from arbitrarily long sequences.

TRAINING/ VALIDATION

- In principle, we grid-search to find the best model in terms of our preferred combination of performance metrics, as we've done in regular ML pipelines. We just iterate over a **different set of hyperparameters**: e.g. batchsize, number of epochs, optimizer, loss functions, regularization parameter, dropout rate (dropout is a common practice to avoid overfitting), and most importantly, learning rate.
- Collober et al. 2011 (widely cited):
  - Benchmark CNN on four NLP tasks: POS, chunking (CHUNK), NER, and Semantic Role Labeling (SRL).
  - Train NN by maximizing likelihood over the training data using stochastic gradient ascent.
  - Tune hyper-parameters by trying only a few different architectures by validation and report the **F1 score** for each task on the validation set, with respect to the number of hidden units.

    - Considering the **variance related to the network initialization, they chose the smallest network achieving "reasonable" performance**, rather than picking the network achieving the top performance on a single run.
    - Among other things, they find that the choice of (most) hyperparameters provided they are large enough, has a limited impact on the generalization performance.

TOXICITY+BIAS - WHAT KAGGLE COMPETITION PEOPLE HAVE USED SO FAR

Models

- Non-NN benchmark model (sklearn): here.
- Bidirectional LSTM (RNN) with Glove and FastText embeddings: links here (keras) and here (PyTorch with suggestions for improvement).
- Temporal CNN in keras: here. Similar to what a RNN does.
- CNN using folds in keras: here.

Metrics

- Overall AUC: link here

Bias

- Metric: Subgroup AUC aggregation: link here
- Scores (aggregate AUC): **~ 0.925**

REFERENCES

- Deep Learning with Python (Chollet, 2018 ed)
- Embeddings:
    - https://techblog.gumgum.com/articles/deep-learning-for-natural-language-processing-part-1-word-embeddings
- CNN:
    - Kim (2014)
    - Collobert et al. (2011)
- RNN:
    - Zhang et al. (2017)
    - https://skymind.ai/wiki/lstm
    - http://www.wildml.com/2015/09/recurrent-neural-networks-tutorial-part-1-introduction-to-rnns/
- Bias
    - Borkan et al. (2019)