01.112 Project

Gabriel Wong (1002299) Lee Tze How (1002033) Koh Jing Yu (1002045)

December 8, 2018

Abstract

In this report, we document our code for the 01.112 project, as well as detail our experiments and approach for the design challenge. We use a multi-layer perceptron to achieve an entity F-score of 0.723 on EN and 0.683 on FR for the validation dataset.

Contents

2 Maximum Likelihood Estimation

Our code for Part 2 is within the files *Part2.py* and *Part2.ipynb*.

2.1 Emission Parameters Estimation

In order to derive the count values, we process the file as follows.
Each line in the file represents a word followed by its label. For each label, we store the number of occurrences of each word in a dictionary with the key as the word, and the value as the number of occurrences. For a particular label, this results in a dictionary that looks something like the following:

```
{'Nous': 100, 'avons': 101, 'tout': 105, 'aim': 7, '.': 1067, ... }
```

We store this dictionary as the value of another dictionary, where the key represents the label. This results in the following dictionary:

```
emissions = {'B-negative': {'"': 1, 'Accueil': 7, 'Ambiance': 2, ...},
             'B-positive': { ... }, ... }
```

This allows us to efficiently retrieve Count($y \rightarrow x$) using

```
emissions[y][x]
```

Similarly, for computational efficiency, we store the counts of each label in a separate dictionary:

```
emission_counts = {'B-negative': 675, 'B-neutral': 113, 'B-positive': 810,
    'I-negative': 233, 'I-neutral': 43, 'I-positive': 181, 'O': 24512}
```

We can then easily retrieve Count(y) using

```
emission_counts[x]
```

This allows us to efficiently retrieve the emission parameters $e(x|y)$, at the cost of a minor space complexity increase.

2.2 Laplace Smoothing

For smoothing, we implement a getter function for the emission parameters. If the given word x does not exist in our parameters learnt from training data, we return the smoothed version of the parameters.

```
def get_emission_parameters(emissions, emission_counts, x, y, k=1):
    ''' Returns the MLE of the emission parameters based on the emissions dictionary '''
    state_data = emissions[y]
    count_y = emission_counts[y] #sum(state_data.values()) # Denominator

    # If x == "#UNK#", it will return the following
    count_y_x = k

    # If x exists in training, return its MLE instead
    if x != "#UNK#":
        count_y_x = state_data[x] # Numerator

    e = count_y_x / (count_y + k)
    return e{}
```

2.3 Sequence Labeling

For sequence labeling, we implement a function that takes as input the sentence to be labeled, and the emission parameters as described in Section 2.1:

```
def label_sequence(sentence, emissions, emission_counts):
    ''' Takes a list `sentence` that contains words of a sentence as strings '''
    tags = []

    for word in sentence:
        predicted_label = ""
        max_prob = -1

        # Find y with maximum probability
        for y in emissions:

            if word not in observations:
                word = "#UNK#"

            if (word in emissions[y]) or (word == "#UNK#"):
                prob = get_emission_parameters(emissions, emission_counts, word, y)

                # If this is higher than the previous highest, use this
                if prob > max_prob:
                    predicted_label = y
                    max_prob = prob

        # Add prediction to list
        tags.append(predicted_label)

    return tags
```

For each word, it finds the arg max over y, and assigns this as the tag. The final output of this function is a list containing the predicted tags.

2.4 Results

Our results on the four datasets are as follows:

Dataset	**F-Score**	
	Entity	**Entity Type**
EN	0.6297	0.4595
SG	0.2952	0.1501
CN	0.1929	0.1134
FR	0.2751	0.1169

3 First-Order Hidden Markov Model

3.1 Transition Parameters Emission

In order to derive the transition values, we process the file as follow.
Each line in the file represents a word followed by its label. For each label, we create a transition denoted by a tuple `(previous label, current label)`. For the first and last label in each sentence, we also add the transitions `(START, label)` and `(label, STOP)` respectively. This results in a dictionary that looks something like the following.

```
{('START', 'O'): 1474, ('O', 'O'): 21452, ('O', 'STOP'): 1621, ...}
```

In addition, for each tuple `(a, b)` that we add to the dictionary, we also add the counts of `a` and `b` to a separate dictionary. This allows us to obtain the counts of each transition, as follows.

```
{'START': 1632, 'O': 24512, 'B-positive': 810, 'I-positive': 181,
    'B-negative': 675, 'B-neutral': 113, 'I-negative': 233, 'I-neutral': 43}
```

This allows us to efficiently retrieve $\text{Count}(y_{i-1}, y_i)$ and $\text{Count}(y_{i-1})$, and obtain the transition parameters

$$q(y_i|y_{i-1}) = \frac{\text{Count}(y_{i-1}, y_i)}{\text{Count}(y_{i-1})}$$

Note that `STOP` is not necessary in the second dictionary, since there will never be a transition *from* `STOP`. In addition, when $\text{Count}(y^*_{i-1}) = 0$, we simply allow $q(y_i|y^*_{i-1}) = 0$.

3.2 Viterbi Algorithm

Combining Parts 2 and 3, we now have the estimated transition and emission parameters. We will denote them as

$$a_{u,v} = q(v|u) = \frac{\text{Count}(u, v)}{\text{Count}(u)}$$

$$b_u(x_j) = e(x|y) = \frac{\text{Count}(u \to x_j)}{\text{Count}(u)}$$

For each sequence , a `pandas.DataFrame` object of size $|\mathcal{T}| \times (n+2)$ is used to store the probabilities of a given state at a given time, where n is the length of the sequence, and $\mathcal{T}$ is the set of all states in training. $(n+2)$ is used to include the `START` and `STOP` observations as well.

Similarly, another `pandas.DataFrame` object of the same size is created as a backpointer table to perform the backtracing algorithm. The probability table will be known as `P` while the backpointer table will be known as `B`.

3.2.1 Initialization

We begin by initializing $\pi(0, \texttt{START}) = 1$.

```
P.loc['START', 0] = 1
```

3.2.2 Recursion

Next, as per the Viterbi algorithm:

$$\pi(j, v) = \max_{u \in \mathcal{T}} \{\pi(j-1, u) \cdot a_{u,v} \cdot b_v(x_j)\}, \; \forall \; v \in \mathcal{T}$$

In addition, we also use the backpointer table to store the state transition (y_{j-1}, y_j):

$$\beta(j, v) = \arg\max_{u \in \mathcal{T}} \{\pi(j-1, u) \cdot a_{u,v} \cdot b_v(x_j)\}, \; \forall \; v \in \mathcal{T}$$

```
x_j = obs_seq[j-1]
for v in states:  # current state
    for u in states:  # previous state
        p = P.loc[u, j-1] * a(u, v) * b(v, x)
        if p > P.loc[v, j]:  # keep larger probability
        P.loc[v, j] = p  # update probability
        B.loc[v, j] = u  # update backpointer
```

This operation is repeated for $j = \{1, ..., n+1\}$, where $x_{n+1} = \texttt{STOP}$.

3.2.3 Backtracking

Finally, once the Viterbi algorithm is complete, we use the backpointer table to obtain the state sequence.

$$y_n = \beta(n+1, \texttt{STOP})$$

$$y_{j-1} = \beta(j, y_j),\ \forall\ j \in \{n-1, ..., 2, 1\}$$

The Python implementation of this is as follows:

```
state_seq = ['STOP']  # initialization
for i in range(n-1, 0, -1):  # {n-1, ..., 2, 1}
    curr_state = state_seq[-1]  # y_j
    prev_state = B.loc[curr_state, i]  # y_{j-1}
    state_seq.append(prev_state)
```

In special cases where there is no observed transition (i.e. $P(y_{j-1}|y_j) = 0$), we might obtain an observation sequence $x_1, x_2, ..., x_n$ that has a probability of 0. In this scenario, we simply predict a sequence of `O`'s.

3.3 Results

Our results on the four datasets are as follows:

Dataset	**F-Score**	
	Entity	**Entity Type**
EN	0.6379	0.5556
SG	0.4185	0.2531
CN	0.3955	0.2700
FR	0.3994	0.2232

4 Second-Order Hidden Markov Model

The Second Order Hidden Markov Model works on a similar principle to the First Order, except now we also want to consider the *prior* state in addition to the current state during our expectation maximization. Our Viterbi algorithm is thus a modified version of the algorithm discussed in Section 3.2.

We initialize the system by cleaning words as they are input - genericizing punctuations, hash mentions, @-mentions, and the like. We also introduce an `#UNK#` term to optimize our results further, for terms being seen for the first time. Further elaboration on this data cleaning can be found in Section 5.1.2.

4.1 Emissions Matrix Training

Our emissions matrix is trained identically to that in Section 2.1, with emissions mapped to the current state via Maximum Likelihood Estimation (MLE).

4.2 Transition Matrix Training

Our transition matrix training process is now a modified version of that of the initial process. As ours is now a second order Hidden Markov Model, we now need to map the likelihood of *sequence pairs* leading to output states for our transition matrix.

Mathematically, our previous state, current state, and next state are represented by t, u, v respectively. Our modified matrices are hence:

$$a_{t,u,v} = q(v|t,u) = \frac{\text{Count}(t,u,v)}{\text{Count}(t,u)}$$

$$b_u(x_j) = e(x|y) = \frac{\text{Count}(u \to x_j)}{\text{Count}(u)}$$

Our Python implementation for the modified training of the transition matrix is as follows:

```
for line in lines:
# data_split is a 2-element list, in the structure [{word}, {state}].
    data_split = line.strip().rsplit(' ', 1)

    # line breaks -> new sequence
    if len(data_split) < 2:
        transitions[(prev_prev_state, prev_state, 'STOP')] += 1
        prev_prev_state = 'PRESTART'
        prev_state = 'START'
        continue

    # Get state transitioned to
    obs, curr_state = data_split
    transitions[(prev_prev_state, prev_state, curr_state)] += 1

    # Update past-2-states history
    prev_prev_state = prev_state
    prev_state = curr_state
```

4.3 Modified Viterbi Algorithm

Our Viterbi algorithm is also now modified to accommodate this two-element transition source. Both the probability and backpointer tables now have a state pair as the index, and n columns where n is the number of words in the sentence being evaluated.

At each time step of the Viterbi algorithm, we look at a pair of previous states to determine the next state. Let $\mathcal{T}$ be the set of all observed states. It follows that there are $|\mathcal{T}|^2$ pairs of states. For each pair, we then determine the next state by iterating through all states in $\mathcal{T}$. This operation is thus $O(|\mathcal{T}|^2 \cdot |\mathcal{T}|)$. Since there are n time steps, where n is the length of the observed sequence, the time complexity of the modified Viterbi algorithm is $O(n|\mathcal{T}|^3)$.

4.3.1 Initialization

We begin by initializing $\pi(\texttt{PRESTART}, \texttt{START}, 0) = 1$. Similar to the original Viterbi algorithm, we have:

- A matrix, P, which serves as our probability table
- A matrix, B, which serves as our backpointer table

Both matrices have an index being the total possible unique combinations of our states.

```
P.loc[('PRESTART','START'),0] = 1
```

4.3.2 Recursion

Next, as per the Viterbi algorithm:

$$\pi(j, v) = \max_{t,u \in \mathcal{T}} \{\pi(j-1, t, u) \cdot a_{t,u,v} \cdot b_v(x_j)\}, \forall\, v \in \mathcal{T}$$

In addition, we also use the backpointer table to store the state transition (y_{j-1}, y_j):

$$\beta(j, v) = \arg\max_{t,u \in \mathcal{T}} \{\pi(j-1, u, v) \cdot a_{t,u,v} \cdot b_v(x_j)\}, \forall\, v \in \mathcal{T}$$

```
for j in range(1, n-1):
  x = clean_word(obs_seq[j-1])
  if x not in dictionary:
    x = "#UNK#"
  for v in states:  # curr state
    for u in states:  # parent
      for t in states: # grandparent
        p = P.loc[(t, u), j-1] * a(t, u, v) * b(v, x)
        if p > P.loc[(u, v), j]:
          P.loc[(u, v), j] = p  # update probability
          B.loc[(u, v), j] = t  # update backpointer - t is the grandfather
```

This operation is repeated for $j = \{1, ..., n\}$.

4.3.3 Termination

At the end of the sentence, we perform one termination iteration, where the last state is $STOP$.

```
j = n - 1
  v = 'STOP'
  for u in states: # parent state
      for t in states: # grandparent state
          p = P.loc[(t, u), j-1] * a(t, u, v)
          if p > P.loc[(u, v), j]:
              P.loc[(u, v), j] = p  # probability
              B.loc[(u, v), j] = t  # backpointer
```

4.3.4 Backtracing

After our probability and backpointer tables have been fully populated, we generate our predicted state sequence by accessing parent states via the backtracing table.

```
state_pair = P[n-1].idxmax()
    state_seq = []
    for i in range(n-1, 0, -1):
        next_state = B.loc[state_pair, i]
        # next_state might be numpy.nan or 0 if not populated
        # in which case predictions fail; return 0s
        if isinstance(next_state, str):
            state_seq.append(state_pair[1])
            state_pair = (next_state, state_pair[0])
        else: # edge case: no possible transition to START
            for j in range(int(i)):
                state_seq.append('O')
            break
    state_seq = state_seq[::-1][:-1]  # reverse and drop STOP
```

4.4 Results

Our results on the EN and FR datasets are as follows:

Dataset	**F-Score**	
	Entity	**Entity Type**
EN	0.650	0.570
FR	0.528	0.325

5 Design Challenge

5.1 Preprocessing

5.1.1 String tokenization

Each file can be considered a corpus, the documents being each sentence in the file. Tokenizing was carried out by converting each sentence into a list of words. During tokenization, all whitespace was stripped from the beginning and end of each word. Also, each token is normalized by converting to lowercase.

5.1.2 Replacing irrelevant words

In the initial problem, the training data contains a large vocabulary of words which occurred only a single time. To mitigate this issue, several types of words were converted into various placeholders instead, as they do not carry semantic meaning.

The placeholders used are as follows:

1. `#PUNC#`: strings that do not contain alphanumeric characters
2. `#HASH#`: hashtags (strings beginning with a `#` character)
3. `#AT#`: `@` mentions (strings beginning with a `@` character)
4. `#NUM#`: strings that contain only digits
5. `#URL#`: strings that begin with `http:` and end with `.com`

5.1.3 Stop words

Stop words were also initially converted to a token labeled `#STOP#`. However, during evaluation on the `EN` and `SG` dataset with Part 3, it was shown to reduce the F_1 score on both the *Correct Entity* and *Correct Entity Type* tasks. It was also shown to produce poorer F_1 scores on the `EN` dataset with Part 5.

Taking into account the negative correlation between removing stop words and F_1 scores, as well as the fact that we do not have French stop words, we decided not to remove stop words from the training data.

5.2 Simple vectorization

We discuss an implementation to convert each word into a vector, allowing us to train a recurrent neural network (RNN) with the text sequences.

5.2.1 Tokenizing

Tokenizing transforms each text into a sequence of integers, which allows us to vectorize the text corpus.

We first obtain a vocabulary of all unique words that have appeared in the training data. We then assign a unique integer to each word, as well as 1 additional integer for the `#UNK#` token, which denotes words not present in the vocabulary.

```
{'rt': 0,
 '#AT#': 1,
 '#PUNC#': 2,
 'encore': 3,
 ...,
 'bpa': 2502,
 '#UNK#': 2503}
```

The following is a sample output sequence that has been tokenized:

```
# input: 'rt #AT# #PUNC# encore #PUNC# #AT# for ... #PUNC#'
[ 0,  1,  2,  3,  2,  1,  4,  ...,  2]
```

5.2.2 One-hot encoding

Once each word has been converted into a corresponding integer, each integer is then converted into a vector via one-hot encoding.

Let $\mathcal{V}$ be the set of all words in the training vocabulary, including `#UNK#`. Then, one-hot encoding maps each input to a vector, $f : \mathbb{R} \to \mathbb{R}^{|\mathcal{V}|}$.

$$f(n)_i = \begin{cases} 1, \text{ if } i = n \\ 0, \text{ otherwise} \end{cases}$$

5.3 word2vec

word2vec is used as a feature extraction method, to convert the sparse one-hot encoding into dense vectors.

5.3.1 Skip-gram

A skip-gram architecture with a window size of 5 is used for the training of the *word2vec*. This means that given a sequence of words $\boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}, ..., \boldsymbol{x}^{(n)}$, for an input variable $\boldsymbol{x}^{(i)}$, the target variable $\boldsymbol{y}^{(i)} \in \{\boldsymbol{x}^{(i-2)}, \boldsymbol{x}^{(i-1)}, \boldsymbol{x}^{(i+1)}, \boldsymbol{x}^{(i+2)}\}$.

We initialize two sets of weights, $\boldsymbol{W} \in \mathbb{R}^{300 \times |\mathcal{V}|}$ and $\boldsymbol{U} \in \mathbb{R}^{|\mathcal{V}| \times 300}$, with each value drawn from a normal distribution $\mathcal{N} \sim (0, 0.1^2)$.

5.3.2 Forward

The model is defined as such:

$$\begin{aligned} \boldsymbol{x}^{(i)}, \boldsymbol{y}^{(i)} &\in \mathbb{R}^{|\mathcal{V}|} \\ \boldsymbol{h}^{(i)} &= \boldsymbol{W}\boldsymbol{x}^{(i)} \\ \boldsymbol{o}^{(i)} &= \boldsymbol{U}\boldsymbol{h}^{(i)} \\ \hat{\boldsymbol{y}}^{(i)} &= \text{softmax}(\boldsymbol{o}^{(i)}) \\ \mathcal{L} &= -\sum_{y^{(i)}} \boldsymbol{y}^{(i)} \cdot \log(\hat{\boldsymbol{y}}^{(i)}) \end{aligned} \tag{5.1}$$

The loss function is the sum of the cross-entropy loss between all target variables in the context, and the predicted variable.

5.3.3 Backward

$$\begin{aligned} \frac{\partial \mathcal{L}^{(i)}}{\partial \boldsymbol{o}^{(i)}} &= \text{softmax}(\boldsymbol{o}^{(i)}) - \boldsymbol{y}^{(i)} \\ &= \hat{\boldsymbol{y}}^{(i)} - \boldsymbol{y}^{(i)} \end{aligned} \tag{5.2}$$

$$\begin{aligned} \frac{\partial \mathcal{L}^{(i)}}{\partial \boldsymbol{U}} &= \frac{\partial \mathcal{L}^{(i)}}{\partial \boldsymbol{o}^{(i)}} \frac{\partial \boldsymbol{o}^{(i)}}{\partial \boldsymbol{U}} \\ &= (\hat{\boldsymbol{y}}^{(i)} - \boldsymbol{y}^{(i)}) \boldsymbol{h}^{(i)^T} \end{aligned} \tag{5.3}$$

$$\begin{aligned}\frac{\partial \mathcal{L}^{(i)}}{\partial \boldsymbol{h}^{(i)}} &= \frac{\partial \mathcal{L}^{(i)}}{\partial \boldsymbol{o}^{(i)}} \frac{\partial \boldsymbol{o}^{(i)}}{\partial \boldsymbol{h}^{(i)}} \\ &= \boldsymbol{U}^T(\hat{\boldsymbol{y}}^{(i)} - \boldsymbol{y}^{(i)})\end{aligned} \tag{5.4}$$

$$\begin{aligned}\frac{\partial \mathcal{L}^{(i)}}{\partial \boldsymbol{W}} &= \frac{\partial \mathcal{L}^{(i)}}{\partial \boldsymbol{h}^{(i)}} \frac{\partial \boldsymbol{h}^{(i)}}{\partial \boldsymbol{W}} \\ &= \boldsymbol{U}^T(\hat{\boldsymbol{y}}^{(i)} - \boldsymbol{y}^{(i)}){\boldsymbol{x}^{(i)}}^T\end{aligned} \tag{5.5}$$

5.3.4 Training

Positive sampling
For each word in every sentence, we take that word to be the center word. Then, for each word in its context, we update the weights according to the equations above.

Negative sampling
For each word in every sentence, we take that word to be the center word. We also obtain the context words. Next, we sample 20 words from the vocabulary, excluding the center and context words, as recommended for small datasets [1]. The probability of sampling a word w is given by the following equation, to reduce the probability of oversampling frequent words:

$$P(w) = \frac{\text{Count}(w)^{\frac{3}{4}}}{\sum_{w'}(\text{Count}(w')^{\frac{3}{4}})}$$

The corresponding loss function for the negative samples is then given by:

$$\mathcal{L} = -\sum_{y^{(i)}} -\boldsymbol{y}^{(i)} \cdot \log(\hat{\boldsymbol{y}}^{(i)})$$

5.3.5 Visualization

We use t-SNE to reduce each word vector to 2 dimensions for visualization.
The results are shown in Figure 1.

5.3.6 Inference

During inference, both weights $\boldsymbol{W}, \boldsymbol{U}$ are combined via averaging [2].

$$\boldsymbol{h} = \frac{1}{2}(\boldsymbol{W} + \boldsymbol{U}^T)\boldsymbol{x}$$

It is also common to find words in the validation and test sets that are not found in the training set. As such, these words are not in the *word2vec* vocabulary.

Instead of simply treating these words as an `#UNK#` token, they were represented as the mean of their context, within a window of size 5.

$$x_i = \frac{1}{4}(x_{i-2} + x_{i-1} + x_{i+1} + x_{i+2})$$

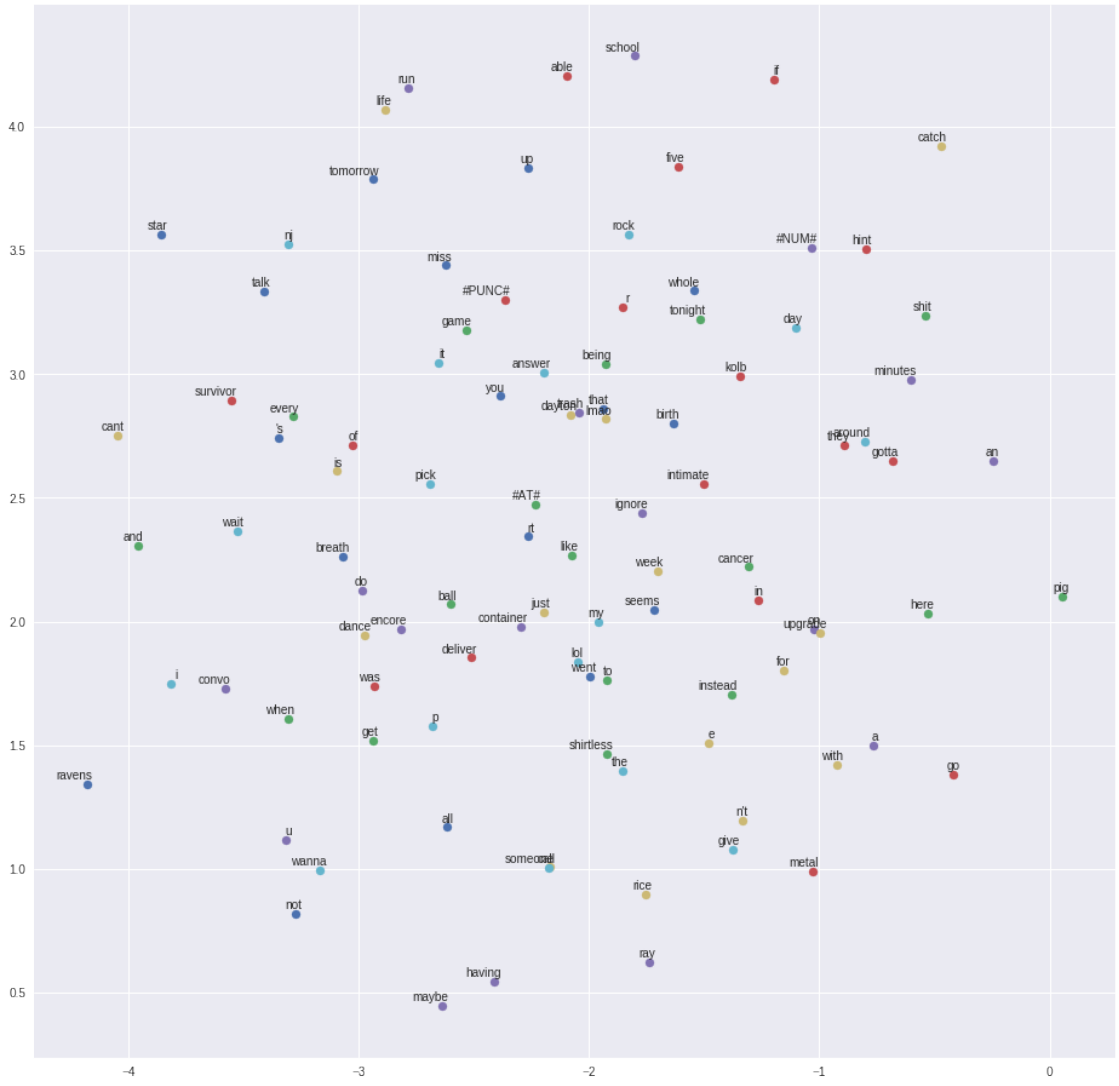


Figure 1: t-SNE visualization of *word2vec* model on EN dataset

5.4 Recurrent neural network (RNN)

5.4.1 Forward

We use a RNN, with a computational graph as shown in Figure 2. The RNN maps an input sequence $\boldsymbol{x}$ to an output sequence $\boldsymbol{o}$. The predicted labels are then given by $\hat{\boldsymbol{y}} = \text{softmax}(\boldsymbol{o})$.

The loss function $\mathcal{L}$ compares the predicted labels $\hat{\boldsymbol{y}}$ with the target labels $\boldsymbol{y}$ through cross-entropy loss, where C is the number of classes in the classification task.

$$\mathcal{L} = -\sum_{i}^{C} \boldsymbol{y}_i \cdot \log(\hat{\boldsymbol{y}}_i)$$

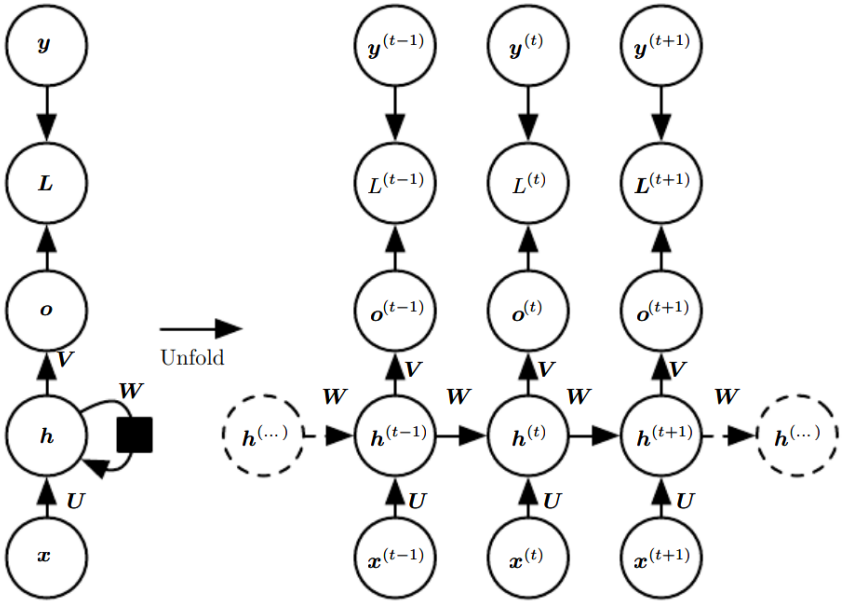


Figure 2: RNN model architecture [3]

The remaining nodes in the graph are computed as shown in Equation 5.6.

$$\begin{aligned} \boldsymbol{a}^{(t)} &= \boldsymbol{b} + \boldsymbol{U}\boldsymbol{x}^{(t)} + \boldsymbol{W}\boldsymbol{h}^{(t-1)} \\ \boldsymbol{h}^{(t)} &= \tanh(\boldsymbol{a}^{(t)}) \\ \boldsymbol{o}^{(t)} &= \boldsymbol{c} + \boldsymbol{V}\boldsymbol{h}^{(t)} \\ \hat{\boldsymbol{y}}^{(t)} &= \text{softmax}(\boldsymbol{o}^{(t)}) \end{aligned} \tag{5.6}$$

$\boldsymbol{U}, \boldsymbol{W}, \boldsymbol{V}$ are the weight matrices, while $\boldsymbol{b}, \boldsymbol{c}$ are biases. By choice of hyperparameters, $\boldsymbol{h}^{(t)} \in \mathbb{R}^{128\times 1}$.

For an input sequence, $\boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}, ..., \boldsymbol{x}^{(n)}$, we first compute the hidden units $\boldsymbol{h}^{(t)}$ as shown in Equation 5.7. At $t = 1$, the hidden unit is computed using only $x^{(1)}$. Once we have the value of $\boldsymbol{h}^{(t)}$, we can then compute its predicted state $\hat{\boldsymbol{y}}^{(t)}$ with Equation 5.6.

$$\begin{aligned} \boldsymbol{h}^{(1)} &= \tanh(\boldsymbol{b} + \boldsymbol{U}\boldsymbol{x}^{(1)}) \\ \boldsymbol{h}^{(t)} &= \tanh(\boldsymbol{b} + \boldsymbol{U}\boldsymbol{x}^{(t)} + \boldsymbol{W}\boldsymbol{h}^{(t-1)}),\ \forall\, t \in \{2, ..., n\} \end{aligned} \tag{5.7}$$

5.4.2 Backward

To perform backpropagation, our objective is to obtain the partial differential of $\mathcal{L}$ with respect to the weights and biases. The gradients are taken to be the **sum** over all time steps $t \in \{1, ..., n\}$.

Output layer

$$\begin{aligned} \frac{\partial \mathcal{L}^{(t)}}{\partial \boldsymbol{o}^{(t)}} &= \text{softmax}(\boldsymbol{o}^{(t)}) - \boldsymbol{y}^{(t)} \\ &= \hat{\boldsymbol{y}}^{(t)} - \boldsymbol{y}^{(t)} \end{aligned} \tag{5.8}$$

$$\begin{aligned}\frac{\partial \mathcal{L}}{\partial \boldsymbol{c}} &= \sum_{t=1}^{n} \frac{\partial \mathcal{L}^{(t)}}{\partial \boldsymbol{o}^{(t)}} \frac{\partial \boldsymbol{o}^{(t)}}{\partial \boldsymbol{c}} \\ &= \sum_{t=1}^{n} (\hat{\boldsymbol{y}}^{(t)} - \boldsymbol{y}^{(t)}) \frac{\partial (\boldsymbol{c} + \boldsymbol{V}\boldsymbol{h}^{(t)})}{\partial \boldsymbol{c}} \\ &= \sum_{t=1}^{n} (\hat{\boldsymbol{y}}^{(t)} - \boldsymbol{y}^{(t)})\end{aligned} \tag{5.9}$$

$$\begin{aligned}\frac{\partial \mathcal{L}}{\partial \boldsymbol{V}} &= \sum_{t=1}^{n} \frac{\partial \mathcal{L}^{(t)}}{\partial \boldsymbol{o}^{(t)}} \frac{\partial \boldsymbol{o}^{(t)}}{\partial \boldsymbol{V}} \\ &= \sum_{t=1}^{n} (\hat{\boldsymbol{y}}^{(t)} - \boldsymbol{y}^{(t)}) \frac{\partial (\boldsymbol{c} + \boldsymbol{V}\boldsymbol{h}^{(t)})}{\partial \boldsymbol{V}} \\ &= \sum_{t=1}^{n} (\hat{\boldsymbol{y}}^{(t)} - \boldsymbol{y}^{(t)})\ {\boldsymbol{h}^{(t)}}^{T}\end{aligned} \tag{5.10}$$

Hidden layer

To compute the gradients of the hidden units $\frac{\partial \mathcal{L}}{\partial \boldsymbol{h}^{(t)}}$, we initialize the gradient at $t = n$, since it has no subsequent time step, and its gradient is dependent only on $\boldsymbol{o}^{(n)}$.

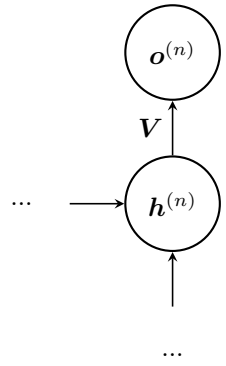


Figure 3: Computational graph showing gradients for $\boldsymbol{h}^{(n)}$

$$\begin{aligned}\frac{\partial \mathcal{L}^{(n)}}{\partial \boldsymbol{h}^{(n)}} &= \frac{\partial \mathcal{L}^{(n)}}{\partial \boldsymbol{o}^{(n)}} \frac{\partial \boldsymbol{o}^{(n)}}{\partial \boldsymbol{h}^{(n)}} \\ &= (\hat{\boldsymbol{y}}^{(n)} - \boldsymbol{y}^{(n)}) \frac{\partial (\boldsymbol{c} + \boldsymbol{V}\boldsymbol{h}^{(n)})}{\partial \boldsymbol{h}^{(n)}} \\ &= \boldsymbol{V}^{T}(\hat{\boldsymbol{y}}^{(n)} - \boldsymbol{y}^{(n)})\end{aligned} \tag{5.11}$$

Now, we can formulate the rest of the gradients recursively.

$$\frac{\partial \mathcal{L}^{(t)}}{\partial \boldsymbol{h}^{(t)}} = \frac{\partial \mathcal{L}^{(t)}}{\partial \boldsymbol{o}^{(t)}} \frac{\partial \boldsymbol{o}^{(t)}}{\partial \boldsymbol{h}^{(t)}} + \frac{\partial \mathcal{L}^{(t+1)}}{\partial \boldsymbol{h}^{(t+1)}} \frac{\partial \boldsymbol{h}^{(t+1)}}{\partial \boldsymbol{h}^{(t)}}$$

From the result in Equation 5.11, we have that

$$\frac{\partial \mathcal{L}^{(t)}}{\partial \boldsymbol{o}^{(t)}} \frac{\partial \boldsymbol{o}^{(t)}}{\partial \boldsymbol{h}^{(t)}} = \boldsymbol{V}^{T}(\hat{\boldsymbol{y}}^{(t)} - \boldsymbol{y}^{(t)})$$

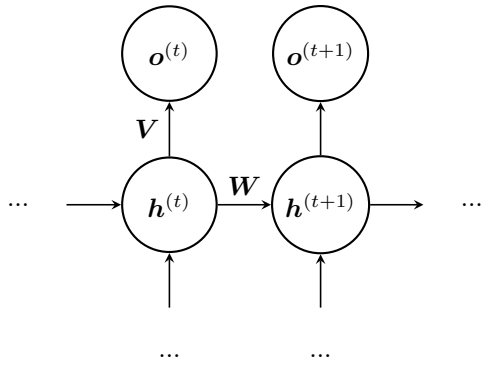


Figure 4: Computational graph showing gradients for $\boldsymbol{h}^{(t)}$

$$\begin{aligned}
\frac{\partial \mathcal{L}^{(t+1)}}{\partial \boldsymbol{h}^{(t+1)}} \frac{\partial \boldsymbol{h}^{(t+1)}}{\partial \boldsymbol{h}^{(t)}} &= \frac{\partial \mathcal{L}^{(t+1)}}{\partial \boldsymbol{h}^{(t+1)}} \frac{\partial \boldsymbol{h}^{(t+1)}}{\partial \boldsymbol{a}^{(t+1)}} \frac{\partial \boldsymbol{a}^{(t+1)}}{\partial \boldsymbol{h}^{(t)}} \\
&= \frac{\partial \mathcal{L}^{(t+1)}}{\partial \boldsymbol{h}^{(t+1)}} \frac{\partial(\tanh(\boldsymbol{a}^{(t+1)}))}{\partial \boldsymbol{a}^{(t+1)}} \frac{\partial(\boldsymbol{b} + \boldsymbol{U}\boldsymbol{x}^{(t+1)} + \boldsymbol{W}\boldsymbol{h}^{(t)})}{\partial \boldsymbol{h}^{(t)}} \\
&= \boldsymbol{W}^T \frac{\partial \mathcal{L}^{(t+1)}}{\partial \boldsymbol{h}^{(t+1)}} \frac{\partial(\tanh(\boldsymbol{a}^{(t+1)}))}{\partial \boldsymbol{a}^{(t+1)}} \\
&= \boldsymbol{W}^T \mathrm{diag}(1 - (\boldsymbol{h}^{(t+1)})^2) \frac{\partial \mathcal{L}^{(t+1)}}{\partial \boldsymbol{h}^{(t+1)}}
\end{aligned}$$

Combining the above results, we obtain

$$\begin{aligned}
\frac{\partial \mathcal{L}^{(t)}}{\partial \boldsymbol{h}^{(t)}} &= \frac{\partial \mathcal{L}^{(t)}}{\partial \boldsymbol{o}^{(t)}} \frac{\partial \boldsymbol{o}^{(t)}}{\partial \boldsymbol{h}^{(t)}} + \frac{\partial \mathcal{L}^{(t+1)}}{\partial \boldsymbol{h}^{(t+1)}} \frac{\partial \boldsymbol{h}^{(t+1)}}{\partial \boldsymbol{h}^{(t)}} \\
&= \boldsymbol{V}^T(\hat{\boldsymbol{y}}^{(t)} - \boldsymbol{y}^{(t)}) + \boldsymbol{W}^T \mathrm{diag}(1 - (\boldsymbol{h}^{(t+1)})^2) \frac{\partial \mathcal{L}}{\partial \boldsymbol{h}^{(t+1)}}
\end{aligned} \tag{5.12}$$

It is next imperative to show that the Jacobian $J^{(t+1)} = \frac{\partial \boldsymbol{h}^{(t+1)}}{\partial \boldsymbol{a}^{(t+1)}} = \frac{\partial(\tanh(\boldsymbol{a}^{(t+1)}))}{\partial \boldsymbol{a}^{(t+1)}} = \mathrm{diag}(1 - (\boldsymbol{h}^{(t+1)})^2)$.

We begin with the knowledge that $\frac{d}{dx}\tanh(x) = 1 - \tanh^2(x)$.

$$J_{ij}^{(t+1)} = \frac{\partial \boldsymbol{h}_i^{(t+1)}}{\partial \boldsymbol{a}_j^{(t+1)}}$$

For $i \neq j$, $J_{ij}^{(t)} = 0$.
For $i = j$,

$$\begin{aligned}
J_{ii}^{(t+1)} = \frac{\partial \boldsymbol{h}_i^{(t+1)}}{\partial \boldsymbol{a}_i^{(t+1)}} &= \frac{\partial(\tanh(\boldsymbol{a}^{(t+1)})_i)}{\partial \boldsymbol{a}_i^{(t+1)}} \\
&= 1 - \tanh^2(\boldsymbol{a}^{(t+1)})_i \\
&= 1 - (\boldsymbol{h}_i^{(t+1)})^2
\end{aligned}$$

$$\therefore J^{(t)} = \frac{\partial \boldsymbol{h}^{(t)}}{\partial \boldsymbol{a}^{(t)}} = \begin{pmatrix} 1 - (\boldsymbol{h}_1^{(t)})^2 & 0 & ... & 0 \\ 0 & 1 - (\boldsymbol{h}_2^{(t)})^2 & ... & 0 \\ ... & ... & 1 - (\boldsymbol{h}_i^{(t)})^2 & ... \\ 0 & 0 & ... & 1 - (\boldsymbol{h}_n^{(t)})^2 \end{pmatrix} = \mathrm{diag}(1 - (\boldsymbol{h}^{(t)})^2)$$

We can then use this result to obtain the gradients with respect to $\boldsymbol{b}, \boldsymbol{W}, \boldsymbol{U}$.

$$\begin{aligned}\frac{\partial \mathcal{L}}{\partial \boldsymbol{b}} &= \sum_{t=1}^{n} \frac{\partial \mathcal{L}^{(t)}}{\partial \boldsymbol{h}^{(t)}} \frac{\partial \boldsymbol{h}^{(t)}}{\partial \boldsymbol{a}^{(t)}} \frac{\partial \boldsymbol{a}^{(t)}}{\partial \boldsymbol{b}} \\ &= \sum_{t=1}^{n} \operatorname{diag}(1 - (\boldsymbol{h}^{(t)})^2) \frac{\partial \mathcal{L}^{(t)}}{\partial \boldsymbol{h}^{(t)}} \frac{\partial (\boldsymbol{b} + \boldsymbol{U}\boldsymbol{x}^{(t)} + \boldsymbol{W}\boldsymbol{h}^{(t-1)})}{\partial \boldsymbol{b}} \\ &= \sum_{t=1}^{n} \operatorname{diag}(1 - (\boldsymbol{h}^{(t)})^2) \frac{\partial \mathcal{L}^{(t)}}{\partial \boldsymbol{h}^{(t)}}\end{aligned} \tag{5.13}$$

$$\begin{aligned}\frac{\partial \mathcal{L}}{\partial \boldsymbol{W}} &= \sum_{t=1}^{n} \frac{\partial \mathcal{L}^{(t)}}{\partial \boldsymbol{h}^{(t)}} \frac{\partial \boldsymbol{h}^{(t)}}{\partial \boldsymbol{a}^{(t)}} \frac{\partial \boldsymbol{a}^{(t)}}{\partial \boldsymbol{W}} \\ &= \sum_{t=1}^{n} \operatorname{diag}(1 - (\boldsymbol{h}^{(t)})^2) \frac{\partial \mathcal{L}^{(t)}}{\partial \boldsymbol{h}^{(t)}} \frac{\partial (\boldsymbol{b} + \boldsymbol{U}\boldsymbol{x}^{(t)} + \boldsymbol{W}\boldsymbol{h}^{(t-1)})}{\partial \boldsymbol{W}} \\ &= \sum_{t=1}^{n} \operatorname{diag}(1 - (\boldsymbol{h}^{(t)})^2) \frac{\partial \mathcal{L}^{(t)}}{\partial \boldsymbol{h}^{(t)}} \boldsymbol{h}^{(t-1)^T}\end{aligned} \tag{5.14}$$

$$\begin{aligned}\frac{\partial \mathcal{L}}{\partial \boldsymbol{U}} &= \sum_{t=1}^{n} \frac{\partial \mathcal{L}^{(t)}}{\partial \boldsymbol{h}^{(t)}} \frac{\partial \boldsymbol{h}^{(t)}}{\partial \boldsymbol{a}^{(t)}} \frac{\partial \boldsymbol{a}^{(t)}}{\partial \boldsymbol{U}} \\ &= \sum_{t=1}^{n} \operatorname{diag}(1 - (\boldsymbol{h}^{(t)})^2) \frac{\partial \mathcal{L}^{(t)}}{\partial \boldsymbol{h}^{(t)}} \frac{\partial (\boldsymbol{b} + \boldsymbol{U}\boldsymbol{x}^{(t)} + \boldsymbol{W}\boldsymbol{h}^{(t-1)})}{\partial \boldsymbol{U}} \\ &= \sum_{t=1}^{n} \operatorname{diag}(1 - (\boldsymbol{h}^{(t)})^2) \frac{\partial \mathcal{L}^{(t)}}{\partial \boldsymbol{h}^{(t)}} \boldsymbol{x}^{(t)^T}\end{aligned} \tag{5.15}$$

Summary

$$\begin{aligned}\frac{\partial \mathcal{L}}{\partial \boldsymbol{U}} &= \sum_{t=1}^{n} \operatorname{diag}(1 - (\boldsymbol{h}^{(t)})^2) \frac{\partial \mathcal{L}^{(t)}}{\partial \boldsymbol{h}^{(t)}} \boldsymbol{x}^{(t)^T} \\ \frac{\partial \mathcal{L}}{\partial \boldsymbol{W}} &= \sum_{t=1}^{n} \operatorname{diag}(1 - (\boldsymbol{h}^{(t)})^2) \frac{\partial \mathcal{L}^{(t)}}{\partial \boldsymbol{h}^{(t)}} \boldsymbol{h}^{(t-1)^T} \\ \frac{\partial \mathcal{L}}{\partial \boldsymbol{b}} &= \sum_{t=1}^{n} \operatorname{diag}(1 - (\boldsymbol{h}^{(t)})^2) \frac{\partial \mathcal{L}^{(t)}}{\partial \boldsymbol{h}^{(t)}} \\ \frac{\partial \mathcal{L}}{\partial \boldsymbol{V}} &= \sum_{t=1}^{n} (\hat{\boldsymbol{y}}^{(t)} - \boldsymbol{y}^{(t)})\, \boldsymbol{h}^{(t)^T} \\ \frac{\partial \mathcal{L}}{\partial \boldsymbol{c}} &= \sum_{t=1}^{n} (\hat{\boldsymbol{y}}^{(t)} - \boldsymbol{y}^{(t)})\end{aligned} \tag{5.16}$$

5.5 Multilayer perceptron (MLP)

5.5.1 Forward

We first define the computational graph for a 2-layer MLP, our architecture of choice.

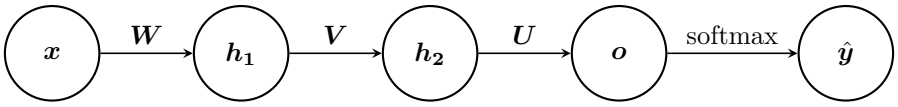


Figure 5: MLP computational graph

Similar to the RNN, the input is defined by $\boldsymbol{x}$, the predicted label $\hat{\boldsymbol{y}}$, and the target labels $\boldsymbol{y}$. The loss function $\mathcal{L}$ is again the cross-entropy loss.

The rest of the graph is defined as such:

$$\begin{aligned} \boldsymbol{a}_1 &= \boldsymbol{W}\boldsymbol{x} + \boldsymbol{b} \\ \boldsymbol{h}_1 &= \text{ReLU}(\boldsymbol{a}_1) \\ \boldsymbol{a}_2 &= \boldsymbol{V}\boldsymbol{h}_1 + \boldsymbol{c} \\ \boldsymbol{h}_2 &= \text{ReLU}(\boldsymbol{a}_1) \\ \boldsymbol{o} &= \boldsymbol{U}\boldsymbol{h}_2 + \boldsymbol{d} \\ \hat{\boldsymbol{y}} &= \text{softmax}(\boldsymbol{o}) \end{aligned} \tag{5.17}$$

5.5.2 Backward

The equations for backpropagation of the MLP are defined similar to the RNN.

Output layer

$$\begin{aligned} \frac{\partial \mathcal{L}}{\partial \boldsymbol{o}} &= \text{softmax}(\boldsymbol{o} - \boldsymbol{y}) \\ &= \hat{\boldsymbol{y}} - \boldsymbol{y} \end{aligned} \tag{5.18}$$

$$\begin{aligned} \frac{\partial \mathcal{L}}{\partial \boldsymbol{d}} &= \frac{\partial \mathcal{L}}{\partial \boldsymbol{o}} \frac{\partial \boldsymbol{o}}{\partial \boldsymbol{d}} \\ &= \frac{\partial \mathcal{L}}{\partial \boldsymbol{o}} \end{aligned} \tag{5.19}$$

$$\begin{aligned} \frac{\partial \mathcal{L}}{\partial \boldsymbol{U}} &= \frac{\partial \mathcal{L}}{\partial \boldsymbol{o}} \frac{\partial \boldsymbol{o}}{\partial \boldsymbol{U}} \\ &= \frac{\partial \mathcal{L}}{\partial \boldsymbol{o}} \, \boldsymbol{h}_2^T \end{aligned} \tag{5.20}$$

Second hidden layer

$$\begin{aligned} \frac{\partial \mathcal{L}}{\partial \boldsymbol{h}_2} &= \frac{\partial \mathcal{L}}{\partial \boldsymbol{o}} \frac{\partial \boldsymbol{o}}{\partial \boldsymbol{h}_2} \\ &= \boldsymbol{U}^T \frac{\partial \mathcal{L}}{\partial \boldsymbol{o}} \end{aligned} \tag{5.21}$$

$$\begin{aligned} \frac{\partial \mathcal{L}}{\partial \boldsymbol{a}_2} &= \frac{\partial \mathcal{L}}{\partial \boldsymbol{h}_2} \frac{\partial \boldsymbol{h}_2}{\partial \boldsymbol{a}_2} \\ \left(\frac{\partial \boldsymbol{h}_2}{\partial \boldsymbol{a}_2} \right)_i &= \begin{cases} 1, (\boldsymbol{a}_2)_i \geq 0 \\ 0, (\boldsymbol{a}_2)_i < 0 \end{cases} \end{aligned} \tag{5.22}$$

$$\begin{aligned}\frac{\partial \mathcal{L}}{\partial \boldsymbol{c}} &= \frac{\partial \mathcal{L}}{\partial \boldsymbol{a}_2}\frac{\partial \boldsymbol{a}_2}{\partial \boldsymbol{c}} \\ &= \frac{\partial \mathcal{L}}{\partial \boldsymbol{a}_2}\end{aligned} \tag{5.23}$$

$$\begin{aligned}\frac{\partial \mathcal{L}}{\partial \boldsymbol{V}} &= \frac{\partial \mathcal{L}}{\partial \boldsymbol{a}_2}\frac{\partial \boldsymbol{a}_2}{\partial \boldsymbol{V}} \\ &= \frac{\partial \mathcal{L}}{\partial \boldsymbol{a}_2}\,\boldsymbol{h}_1^T\end{aligned} \tag{5.24}$$

First hidden layer

$$\begin{aligned}\frac{\partial \mathcal{L}}{\partial \boldsymbol{h}_1} &= \frac{\partial \mathcal{L}}{\partial \boldsymbol{a}_2}\frac{\partial \boldsymbol{a}_2}{\partial \boldsymbol{h}_1} \\ &= \boldsymbol{V}^T\,\frac{\partial \mathcal{L}}{\partial \boldsymbol{a}_2}\end{aligned} \tag{5.25}$$

$$\begin{aligned}\frac{\partial \mathcal{L}}{\partial \boldsymbol{a}_1} &= \frac{\partial \mathcal{L}}{\partial \boldsymbol{h}_1}\frac{\partial \boldsymbol{h}_1}{\partial \boldsymbol{a}_1} \\ \left(\frac{\partial \boldsymbol{h}_1}{\partial \boldsymbol{a}_1}\right)_i &= \begin{cases}1, (\boldsymbol{a}_1)_i \geq 0 \\ 0, (\boldsymbol{a}_1)_i < 0\end{cases}\end{aligned} \tag{5.26}$$

$$\begin{aligned}\frac{\partial \mathcal{L}}{\partial \boldsymbol{b}} &= \frac{\partial \mathcal{L}}{\partial \boldsymbol{a}_1}\frac{\partial \boldsymbol{a}_1}{\partial \boldsymbol{b}} \\ &= \frac{\partial \mathcal{L}}{\partial \boldsymbol{a}_1}\end{aligned} \tag{5.27}$$

$$\begin{aligned}\frac{\partial \mathcal{L}}{\partial \boldsymbol{W}} &= \frac{\partial \mathcal{L}}{\partial \boldsymbol{a}_1}\frac{\partial \boldsymbol{a}_1}{\partial \boldsymbol{W}} \\ &= \frac{\partial \mathcal{L}}{\partial \boldsymbol{a}_1}\,\boldsymbol{x}^T\end{aligned} \tag{5.28}$$

5.6 Training

5.6.1 Weight initialization

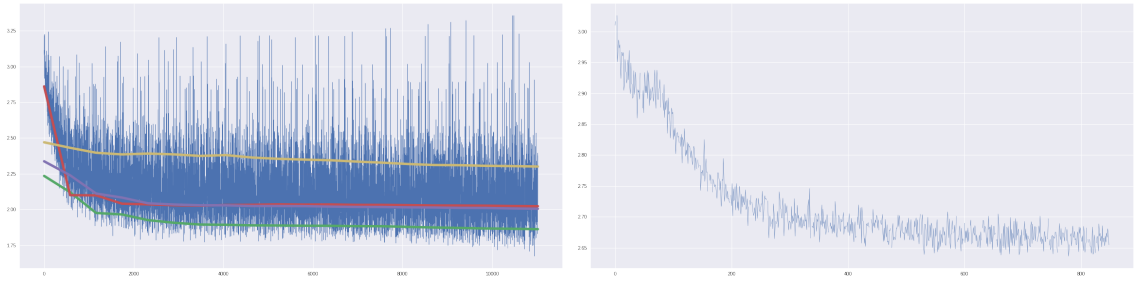
All weights were initialized with values sampled from a Gaussian distribution $\mathcal{N} \sim \{0, 0.1^2\}$.
All biases were initialized with a constant value 0.1.

```
W = np.random.normal(0, 0.1, size=[300, 128])
b = np.ones(shape=[1, 128]) * 0.1
```

5.6.2 Batch updates

The weight updates were done in batches. b sentences were randomly chosen, where b is a hyperparameter. The gradients of each sentence were first independently obtained. Then, the mean of those gradients were taken and applied to the weights.

This helped to smooth the loss decrease over training. We compare the value of the training loss over time before and after implementing batch updates in Figure 6.



(a) Before batch updates

(b) After batch updates

Figure 6: Training loss against epochs

5.6.3 Optimizers

Stochastic gradient descent
SGD was the initial choice of optimizer due to its ease of implementation. However, it was shown to perform poorly on class imbalance without strong regularization.

Adagrad
Adagrad was then used in place of SGD. This yielded highly favourable results.

```
cum_grad = [np.zeros_like(weight) for weight in model]
for i in range(n):
    loss, grad = backward(*model, x=x, y=y)

    # gradient accumulation
    for cum_weight, weight_update in zip(cum_grad, grad):
        cum_weight += np.square(weight_update)

    # gradient update
    for weight, cum_weight, weight_update in zip(model, cum_grad, grad):
        weight -= (lr / (np.sqrt(cum_weight) + 1e-6)) * weight_update
```

5.6.4 Regularization

We add L_2 regularization to the weights $\boldsymbol{U}, \boldsymbol{W}, \boldsymbol{V}$.
This adds an additional term to their gradients, e.g. $\lambda \|\boldsymbol{U}\|_1$, where λ is the regularization penalty coefficient.

The regularization term helped to deal with the class imbalance in the labels during training of the RNN. The class imbalance is illustrated in Figure 7.

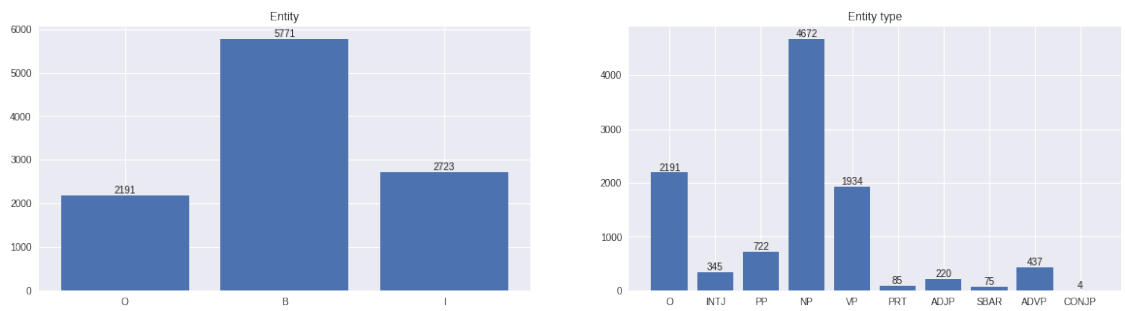


Figure 7: Distribution of labels in FR dataset

In addition, we make use of dropout [4]. We add a dropout after the first hidden layer to help with overfitting issues. The probability of dropping out a neuron is a hyperparameter that we tune using grid search as described in Section 5.6.6.

5.6.5 Class weighting

The class imbalance as illustrated in Figure 7 caused the model to over-predict a particular class, resulting in very recall.

In order to correct the class imbalance, the gradient function was modified to include class weights w:

$$\frac{\partial \mathcal{L}^{(t)}}{\partial \boldsymbol{o}^{(t)}} = (\hat{\boldsymbol{y}}^{(t)} - \boldsymbol{y}^{(t)}) \circ w$$

where $\circ$ is the element-wise multiplication between the two vectors.

However, by introducing class weights, it was found that the precision of the results drastically worsened. As such, class weighting was not used.

5.6.6 Hyperparameter optimization

In order to determine the optimal hyperparameters for this task, we perform a grid search. We enumerate over the following hyperparameters and the possible values:

Hyperparameter	Possible Values
Batch Size	[8, 16, 32, 64]
Learning Rate	[1e-2, 1e-3, 1e-4]
Hidden Dimension	[32, 128, 256, 512]
Dropout	[0.0, 0.1, 0.2, 0.3]

Table 1: Hyperparameter Selection

After performing the above search, we selected the best results based on the validation set. For the EN dataset, the optimal parameters based on the validation set are a batch size of 16, learning rate of $1e-2$, hidden dimension of 512, and dropout of 0.2. For the FR dataset, the optimal parameters are a batch size of 64, learning rate of $1e-2$, hidden dimension of 512, and dropout of 0.2. The results for the two models are presented in Table 3.

5.7 Results

We compare the results on the EN and FR datasets between the RNN and MLP method.

5.7.1 RNN

Dataset	**F-Score**	
	Entity	**Entity Type**
EN	0.572	0.268
FR	0.136	0.094

Table 2: RNN Results

5.7.2 MLP

Dataset	F-Score	
	Entity	**Entity Type**
EN	0.723	0.631
FR	0.683	0.472

Table 3: MLP Results

5.8 Evaluation

In general, we see that the MLP performs significantly better than the RNN on all metrics. This is likely due to the vanishing gradients problem of the RNN, as some of the sentences may be quite long. Additionally, the FR dataset is quite poorly labeled, with most of the words being designated a label of "O". This may point to the downsides of a sequence based model, which may overfit to the poor structure in the sentences. We attribute the performance gain of the MLP for this task due to its high fitting power. With a hidden dimension size of 512 neurons, it is able to fit the training set very well.

References

[1] T. Mikolov, I. Sutskever, K. Chen, G. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," *CoRR*, vol. abs/1310.4546, 2013.

[2] J. Pennington, R. Socher, and C. D. Manning, "Glove: Global vectors for word representation," in *Empirical Methods in Natural Language Processing (EMNLP)*, pp. 1532–1543, 2014.

[3] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[4] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: a simple way to prevent neural networks from overfitting," *The Journal of Machine Learning Research*, vol. 15, no. 1, pp. 1929–1958, 2014.