Minor in AI

Sequence Modelling

NLP Pipeline

March 11, 2025

1 Problem with RNNs

Recurrent Neural Networks (RNNs) process text sequentially, meaning they predict outputs one word at a time based only on previous words. However, some tasks, like Named Entity Recognition (NER), require looking at future words to make correct classifications. This limitation is evident in the given example.

Sentence 1:

Teddy Roosevelt was the president of USA.

Teddy	Roosevelt	was	the	president	of	USA.
1	1	0	0	0	0	1

Here, "Teddy Roosevelt" refers to a **person**, and both words are correctly labeled as named entities (1 for "Teddy" and "Roosevelt"). The entity can be recognized by seeing the second word ("Roosevelt"), confirming that "Teddy" in this case is a name.

Sentence 2:

Teddy Bears are for Sale in Big Bazaar.

Teddy	Bears	are	for	Sale	in	Big Bazaar.
0	0	0	0	0	0	1

Here, "Teddy" does not refer to a **person**; instead, "Teddy Bears" is a noun phrase referring to stuffed toys. However, if the model sees only "Teddy" initially, it may mistakenly classify it as a person. The correct classification (0 for "Teddy" and "Bears") can only be determined if the model looks ahead to see "Bears".

Key Issue with RNNs

- RNNs process words in order and lack the ability to see **future words** while making predictions.
- In **Sentence 2**, an RNN might wrongly classify "Teddy" as a named entity because it does not know that "Bears" follows.
- This limitation arises because RNNs work with **only past information**, making them ineffective in cases where future context is crucial.

To resolve this issue, models like Bidirectional RNNs (BiRNNs) or Transformers (e.g., BERT) are used, as they incorporate both past and future context when making predictions.

The process of training a neural network involves updating its weights using backpropagation. This document explains how weight updates happen using the given diagram.

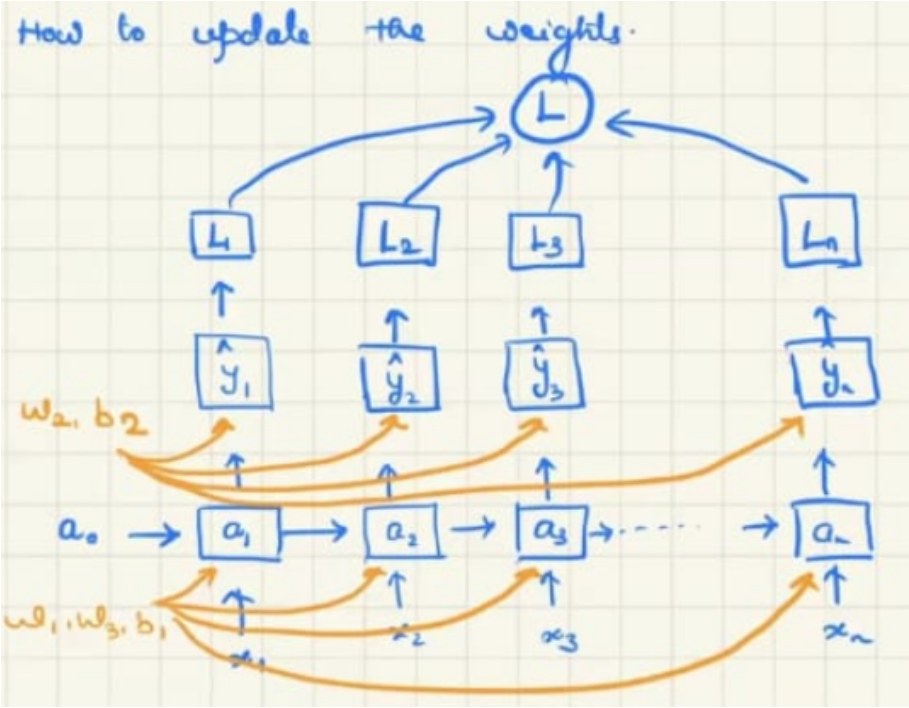

Figure 1: Training Workflow

2 How to update the weights?

2.1 Feedforward Pass

During the feedforward pass:

- The input vector $x = (x_1, x_2, \ldots, x_n)$ propagates through the network.
- Each neuron activation is computed as:

$$a_i = f(W_i a_{i-1} + b_i)$$

  where f is the activation function.

- The final layer produces an output $\hat{y}$, which is compared with the true value y.

2.2 Loss Computation

The error between the predicted output $\hat{y}$ and the true label y is calculated using a loss function L, such as Mean Squared Error (MSE) or Cross-Entropy Loss.

2.3 Backpropagation (Gradient Calculation)

The loss is propagated backward through the network:

- Compute the gradient of the loss with respect to the weights:

$$\frac{\partial L}{\partial W_i}$$

- Update the weights using gradient descent:

$$W_i \leftarrow W_i - \eta \frac{\partial L}{\partial W_i}$$

  where η is the learning rate.

This process ensures that the model learns optimal weights, improving its accuracy over multiple iterations.

3 Binary Cross-Entropy Loss – `BCELoss`

Binary Cross-Entropy (BCE) is a widely used loss function for binary classification tasks. It measures the difference between two probability distributions, often used in logistic regression and neural networks for classification.

3.1 Mathematical Formulation

Given a dataset with N samples, where each sample has:

- True label $y_i \in \{0, 1\}$
- Predicted probability $\hat{y}_i$ (output of a sigmoid function)

The Binary Cross-Entropy (BCE) Loss is defined as:

$$L = -\frac{1}{N} \sum_{i=1}^{N} [y_i \log \hat{y}_i + (1 - y_i) \log(1 - \hat{y}_i)]$$

Explanation

- If $y_i = 1$, the loss simplifies to $-\log(\hat{y}_i)$, meaning we penalize low predicted probabilities for the positive class.
- If $y_i = 0$, the loss simplifies to $-\log(1 - \hat{y}_i)$, meaning we penalize high predicted probabilities for the negative class.
- The logarithm ensures that predictions close to the true label have a lower penalty, whereas incorrect predictions are penalized heavily.

3.2 Implementation in Python

The BCE loss function is implemented in deep learning frameworks such as TensorFlow and PyTorch:

3.2.1 PyTorch Implementation

```
import torch.nn as nn

bce_loss = nn.BCELoss()
```

3.2.2 TensorFlow/Keras Implementation

```
from tensorflow.keras.losses import BinaryCrossentropy

bce_loss = BinaryCrossentropy()
```

3.3 Derivative of BCE

To update the weights during backpropagation, we compute the derivative of BCE with respect to $\hat{y}_i$:

$$\frac{\partial L}{\partial \hat{y}_i} = -\frac{y_i}{\hat{y}_i} + \frac{(1 - y_i)}{(1 - \hat{y}_i)}$$

Binary Cross-Entropy is an effective loss function for binary classification, ensuring better learning by penalizing incorrect predictions heavily while rewarding correct ones.

4 NLP Pipeline

This document explains the text preprocessing steps using Python's `nltk` (Natural Language Toolkit). We demonstrate key techniques such as:

- Tokenization (splitting text into words and sentences)
- Part-of-Speech (POS) tagging
- Text normalization (lowercasing, stemming, lemmatization)
- Stopword and punctuation removal

4.1 Step 1: Loading the Input

We begin by reading the text file `The_Time_Machine.txt`.

```
# Step 1: Loading the input
with open("The_Time_Machine.txt", "r", encoding="utf-8") as f:
    lines = f.read()

# Display the first 100 characters
lines[:100]
```

4.2 Step 2: Tokenization

Tokenization involves breaking down text into individual words and sentences.

```
# Step 2: Tokenization
import nltk
nltk.download("punkt_tab")
from nltk.tokenize import word_tokenize, sent_tokenize

# Sentence and word tokenization
sentences = sent_tokenize(lines)
words = word_tokenize(lines)

```

```
print(len(sentences))  # Number of sentences
print(len(words))      # Number of words
```

```
[nltk_data] Downloading package punkt_tab to /root/nltk_data...
[nltk_data]   Unzipping tokenizers/punkt_tab.zip.
1945
41001
```

```
# Display first three sentences
for i in range(3):
    print(sentences[i])
    print("--------------")
```

```
The Project Gutenberg eBook of The Time Machine

This ebook is for the use of anyone anywhere in the United States and
most other parts of the world at no cost and with almost no restrictions
whatsoever.
--------------
You may copy it, give it away or re-use it under the terms
of the Project Gutenberg License included with this ebook or online
at www.gutenberg.org.
--------------
If you are not located in the United States,
you will have to check the laws of the country where you are located
before using this eBook.
--------------
```

```
# Display first 50 words
for i in range(50):
    print(words[i])
    print("----------")
```

```
The
----------
Project
----------
Gutenberg
----------
eBook
----------
of
----------
The
----------
Time
----------
Machine
----------
```

```
This
----------
ebook
----------
is
----------
for
----------
the
----------
use
----------
of
----------
anyone
----------
anywhere
----------
in
----------
the
----------
United
----------
States
----------
and
----------
most
----------
other
----------
parts
----------
of
----------
the
----------
world
----------
at
----------
no
----------
cost
----------
and
----------
```

```
with
----------
almost
----------
no
----------
restrictions
----------
whatsoever
----------
.
----------
You
----------
may
----------
copy
----------
it
----------
,
----------
give
----------
it
----------
away
----------
or
----------
re-use
----------
it
----------
under
----------
```

4.3 Step 3: Part-of-Speech (POS) Tagging

POS tagging assigns grammatical categories (noun, verb, adjective, etc.) to each word.

```
# Step 3: Part of Speech tagging
nltk.download("averaged_perceptron_tagger_eng")
nltk.pos_tag(words[10:25])  # POS tagging for a sample of words
```

```
[nltk_data] Downloading package averaged_perceptron_tagger_eng to
[nltk_data]     /root/nltk_data...
[nltk_data]   Package averaged_perceptron_tagger_eng is already up-to-
[nltk_data]       date!
```

```
[('is', 'VBZ'),
 ('for', 'IN'),
 ('the', 'DT'),
 ('use', 'NN'),
 ('of', 'IN'),
 ('anyone', 'NN'),
 ('anywhere', 'RB'),
 ('in', 'IN'),
 ('the', 'DT'),
 ('United', 'NNP'),
 ('States', 'NNPS'),
 ('and', 'CC'),
 ('most', 'JJS'),
 ('other', 'JJ'),
 ('parts', 'NNS')]
```

4.4 Step 4: Text Normalization

Text normalization involves:

- Lowercasing (converting all letters to lower case)
- Stemming (reducing words to their root forms, sometimes non-meaningful)
- Lemmatization (reducing words to their dictionary form)

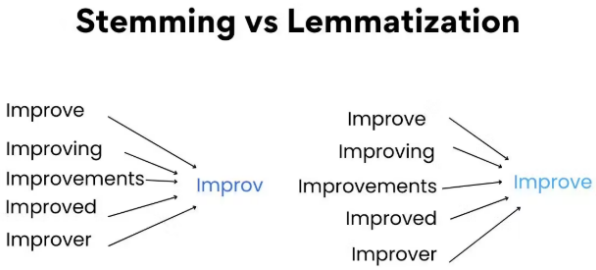

Figure 2: Stemming vs Lemmatization P.C. : Turing

Word	Stemming	Lemmatization
information	inform	information
informative	inform	informative
computers	comput	computer
feet	feet	foot

Figure 3: Few Examples P.C. : StudyML

4.4.1 Lowercasing

```
# Lowercasing the text
lower_text = lines.lower()
print(lines[25:35])       # Before lowercasing
print(lower_text[25:35])  # After lowercasing
```

```
ook of The
ook of the
```

4.4.2 Stemming

Stemming reduces words to their base form but may produce non-meaningful words.

```
from nltk.stem import PorterStemmer

ps = PorterStemmer()
stemmed_words = [ps.stem(word) for word in words]

print(words[35:45])         # Original words
print(stemmed_words[35:45]) # Stemmed words
```

```
['restrictions', 'whatsoever', '.', 'You', 'may', 'copy', 'it', ',', 'give', 'it']
['restrict', 'whatsoev', '.', 'you', 'may', 'copi', 'it', ',', 'give', 'it']
```

4.4.3 Lemmatization

Lemmatization provides dictionary-based word forms, preserving meaning.

```
nltk.download("wordnet")
from nltk.stem import WordNetLemmatizer

lemmatizer = WordNetLemmatizer()
lemmatized_words = [lemmatizer.lemmatize(word) for word in stemmed_words
    ]

# Display words where lemmatization changed the stemmed form
for i in range(len(stemmed_words)):
    if stemmed_words[i] != lemmatized_words[i]:
        print(stemmed_words[i], lemmatized_words[i])
```

```
us u
pass pas
us u
us u
as a
us u
as a
as a
as a
as a
laps lap
as a
```

```
as a
as a
as a
us u
pass pas
as a
as a
feet foot
as a
us u
us u
us u
as a
us u
```

⋮

```
[nltk_data] Downloading package wordnet to /root/nltk_data...
[nltk_data]   Package wordnet is already up-to-date!
```

4.5 Step 5: Stopwords and Punctuation Removal

Stopwords (common words like *the, is, in*) and punctuation are removed to enhance text quality.

```python
from nltk.corpus import stopwords
nltk.download("stopwords")

stop_words = set(stopwords.words("english"))

# Remove stopwords
filtered_words = [word for word in lemmatized_words if word.lower() not
    in stop_words]

print(len(lemmatized_words))  # Before filtering
print(len(filtered_words))    # After filtering
```

```
41001
23768
[nltk_data] Downloading package stopwords to /root/nltk_data...
[nltk_data]   Unzipping corpora/stopwords.zip.
```

Key Takeaways

- **Binary Cross-Entropy (BCE) Loss**: Used for binary classification, it penalizes incorrect predictions using the formula:

$$L = -\frac{1}{N}\sum_{i=1}^{N}\left[y_i \log(\hat{y}_i) + (1 - y_i)\log(1 - \hat{y}_i)\right]$$

  ensuring stable training with a small ϵ to avoid $\log(0)$ errors.

- **Training the Network**: Involves forward propagation to compute loss and backpropagation to update weights using optimizers like SGD or Adam, with techniques such as dropout and L2 regularization for stability.
- **Problems with RNN in Sequence Modeling**: Struggles with long-term dependencies due to vanishing/exploding gradients, has inefficient parallelization, and limited memory retention; LSTMs and GRUs mitigate these issues.
- **NLP Pipeline**: Includes text preprocessing (tokenization, stemming, stopword removal), feature extraction (TF-IDF, Word2Vec), and model training using deep learning architectures like LSTMs and transformers.

Google Collab Link : Code Here!!!