Recurrent Neural Networks-Based Sentiment Analysis of Amazon Reviews: A Comparative Study of Vanilla and Gated Implementations

Tanishq Ahuja (24074033)

June 10, 2025

Contents

1 Introduction

1.1 Goal

The goal of this project is to build an RNN (Vanilla and LSTM/GRU) to predict sentiment polarity from product review text. Before that, the raw data from the dataset (Amazon reviews) needs to go through conventional NLP Preprocessing, Padding and, Tokenization

1.2 Dataset

Amazon reviews Dataset on Kaggle

- The dataset has two files, train and test with 3.6M and 400k rows respectively
- Each row has 3 columns
  - polarity (1 for negative, 2 for positive)
  - title (review heading)
  - text (review body)

1.3 Overview of the approach

1. Clean the data by filling missing cells, then sampling the data keeping in mind both class balance and review length as it is an important factor.
2. Pre-process the data to handle the two text columns, title and review. Deal with language intricacies like punctuations, contractions, and different casing and put IDs to the datapoints.
3. Tokenize the data based on the vocabulary of pre-trained embedding (`GloVe` for this task)
4. Prepare Training Loaders and Datsets to correctly pad the data and supply tokens, labels, ids and lengths (of the review) to the NN with the help of the `collate_fn`.
5. Creating a NN having 3 main layers (details in the relevant section)
   - **Frozen Embedding layer**
   - **Recurrent Layer**
   - **Dense Linear Layer**
6. Using a threshold to convert the final probability to a predicted label.

2 Data Management

2.1 Data Cleaning

Let's start by importing the training the data and checking NA values

```python
import pandas as pd
df = pd.read_csv("train.csv", names=["rating", "title", "review"])
df.isna().sum()
```

NA Values We can see that there are 207 missing title cells. We fill them with empty strings.

Review length Since, the sequence length is an important factor in Recurrent architectures, we will create a column for that.

Label Mapping The dataset has labels 1 and 2, which are not conventional for Binary classification so we will map them to 0 and 1.

```python
def clean_df(df):
    df["title"] = df["title"].fillna("")
    df["review_length"] = df["review"].apply(len)
    df["rating"] -= 1
    return df

df = clean_df(df)
```

2.2 Dataset Sampling

For this part, I took help of an LLM
The dataset is quite huge (3.6M rows), so it is necessary we sample some part of it for initial training and hyperparameter tuning. It is not ideal to use the entire dataset as it would require a lot of compute resources and time for hit and trial during the initial stages of model writing.
For the final model, we can use the entire dataset if enough compute (specifically RAM) is available.

```python
def sample(df, number_per_class):
    bins = df['review_length'].quantile([0, 0.2, 0.4, 0.6, 0.8,
    ↪  1.0]).to_numpy()
    bins[0] = 0
    bins[-1] += 1
    bins
    labels = np.linspace(0, 4, 5)
    df['length_bin'] = pd.cut(df['review_length'], bins=bins, labels=labels)

    # Stratified sampling: group by class and bin
    grouped = df.groupby(['rating', 'length_bin'])

    # Decide total size per class
    sample_size_per_class = number_per_class  # or whatever you want
    final_samples = []

    for label in df['rating'].unique():
        group = df[df['rating'] == label]
        bin_counts = group['length_bin'].value_counts(normalize=True)

        for bin_label, frac in bin_counts.items():
            n = int(sample_size_per_class * frac)
            subset = grouped.get_group((label, bin_label))

            if len(subset) < n:
                n = len(subset)  # safeguard
            sampled = subset.sample(n=n, random_state=42)
            final_samples.append(sampled)

    # Combine
    sampled_df = pd.concat(final_samples).sample(frac=1,
    ↪   random_state=42).reset_index(drop=True)
    return sampled_df
```

3 Preprocessing

3.1 Preprocessing function

```python
def preprocess(title, review):
    text = title + " : " + review
    text = text.lower()
    text = contractions.fix(text)
    text = re.sub(r'&', r'and', text)
    text = re.sub(r"[^a-z0-9!?.,:' ]+", "", text)
    text = re.sub(r'([!?.,])\1{1,}', r'\1', text)
    #separating punctuation from words for better tokenization later.
    text = re.sub(r'([!?.,])', r' \1 ', text)
    text = re.sub(r'([ ])\1{1,}', r'\1', text)
    return text
```

Title and Review combination Since both the title and review are important for judging the sentiment of the review, both are combined into one text. The separator ":" is used in hope that the model learns through different examples that this acts as a separator between title and review.

Handling Contractions The Reviews are often written in an informal sense and consist of a variety of contractions (e.g. isn't, would've). To deal with this, the python module `contractions` was used.

Handling Punctuation Relevant punctuation symbols (`,`, `.`, `?`, `!`) can add meaning to the context of the review and help with better sentiment judgement (confusion, excitement, disappointment etc.). Since, `GloVe` has separate embeddings for these symbols, we need to separate them from the words and use them as a token of their own. Some special characters like "`&`" were explicitly replaced with relevant english words like "`and`"

IDs for the datapoints Later during testing phase, we would be required to analyze which specific review triggered a false-positive and we would require the actual readable text and not tokens or word embeddings. So, the IDs act as a connecting bridge between the two. We can simply log the IDs of the datapoints that were misclassified and use those IDs to get the text.

```
df = sample(original_df, 50000)
df['id'] = range(1, len(df) + 1)
df["text"] = df.apply(lambda row: preprocess(row["title"], row["review"]),
↪   axis=1)
df.drop(["title", "review", "review_length", "length_bin"], inplace=True,
↪   axis=1)
```

4 Word Embeddings and Tokenization

4.1 Embedding Sources

As previously mentioned, for this project I used pre-trained `GloVe` word embeddings (specifically the dataset of 400k words trained on wikipedia for 100 dimensional embedding vector). It can be found here.

The source file was imported and parsed to create two dictionaries, one a vocbulary map for converting words to integral tokens and the other to conver the tokens to their respective embeddings.

```
dim = 100
embed_file = f"glove.6B/glove.6B.{dim}d.txt"
embeds = {}
vocab_map = {}
#out of vocabulary vector
embeds[0] = np.zeros(dim)
#padding index vector
embeds[1] = np.zeros(dim)

with open(embed_file, "r", encoding='utf-8') as f:
    i = 2
    for l in f:
        l_split = l.split()
        word = l_split[0]
        vector = np.asarray(l_split[1:], "float32")
        vocab_map[word] = i
        embeds[i] = vector
        i += 1
```

The `i=0` index of these dictionaries is reserved for the out of vocabulary words (the `<UNK>` token). While the `i=1` is reserved for padding token (`<PAD>`). The padding index is going to be `1`.

After this, a tensor having all of the embeddings was created for use in the embeddings layer. This tensor can also be save locally for future use.

```python
embed_tensor = torch.stack([torch.tensor(v, dtype=torch.float32) for v in
↪   embeds.values()])
torch.save(embed_tensor, "embed_tensor.pt")
```

4.2 Tokenization Strategy

Since, I used pretrained `GloVe` embeddings, the tokenization was simply done based on word separator " " (space)

Handling Out-of-Vocabulary Words Since, the `GloVe` vocabulary used is only 400k words long, the model is obviously going to encounter words that are out of vocabulary. To deal with them, I just use a zero vector. But, strategies like tokenizing subparts of the word (BPE, WordPiece etc.) exist to create a morphological embedding out of the unknown word.

4.3 Padding and `collate_fn`

```python
def tokenize(text):
    tokens = [vocab_map.get(word, 0) for word in text.split()]
    return torch.tensor(tokens,dtype=torch.long)

df["tokens"] = df["text"].apply(tokenize)
df["length"] = df["tokens"].apply(len)
```

For this project I tried to implement dynamic padding based on batches. Basically, each datapoint is padded to the size of the longest sample in the current batch. This is done using PyTorch's `torch.nn.utils.rnn.pad_sequence` and `collate_fn` attribute from the `torch.utils.data.DataLoader` class.

```python
from torch.nn.utils.rnn import pad_sequence

def collate_fn(batch):
    tokens, labels, lengths, ids = zip(*batch)
    lengths = torch.tensor(lengths)
    ids = torch.tensor(ids)
    padded_tokens = pad_sequence(tokens, batch_first=True, padding_value=1)
    labels = torch.tensor(labels, dtype=torch.float32).reshape(-1, 1)
    return padded_tokens, labels, lengths, ids
```

The function takes in a batch of samples (extracted by the `DataLoader` mentioned in the Training section of this document). Unzips them to get tokens, labels, lengths and ids of all the samples, pads the tokens using `pad_sequence`, converts them to respective tensors and returns. These tensors will be used in the training loop and will be fed to the model.

5 Sentiment Analysis

5.1 Model

I have used PyTorch's `torch.nn.Module` as the base class to create my Sentiment analysis Model.

5.1.1 `__init__`

```python
import torch.nn as nn
from torch.nn.utils.rnn import pack_padded_sequence

class SentimentNN(nn.Module):
    def __init__(self, input_size, hidden_size, embed_tensor, lstm=True):
        super(SentimentNN, self).__init__()
        self.s_i = input_size
        self.s_h = hidden_size
        self.lstm = lstm

        self.embed_layer = nn.Embedding.from_pretrained(embed_tensor,
        ↪  padding_idx=1)
        self.embed_layer.weight.requires_grad = False
        if self.lstm:
            self.rnn = nn.LSTM(self.s_i, self.s_h, batch_first=True)
        else:
            self.rnn = nn.RNN(self.s_i, self.s_h, batch_first=True)
        self.dense = nn.Linear(self.s_h, 1)

    ...
```

As mentioned in the overview, the model consists of three main layers.

- **Frozen Embedding layer** converts tokens to word vectors using pretrained `GloVe` embeddings on the fly. Made use of `torch.nn.Embedding.from_pretrained`. The embed tensor previously created is also passed.
- **Recurrent Layer** can be vanilla RNN or LSTM based on the value of flag during `__init__` call. Made use of `torch.nn.RNN / LSTM`
- **Dense Linear Layer** with sigmoid activation to give final prediction of the sentiment. Takes in the last hidden state of the RNN/LSTM to produce probability. Made use of `torch.nn.Linear`. This linear layer can also be replaced with a group of layers having different number of neurons. This may help in even more generalization and introduce more non-linearity in the model.

5.1.2 `forward`

```
    ...

    def forward(self, x, lengths):
        vec = self.embed_layer(x)
        packed_input = pack_padded_sequence(vec, lengths, batch_first=True,
        ↪  enforce_sorted=False)
        if self.lstm:
            output, (hn, cn) = self.rnn(packed_input)
            last_hidden = hn[-1]
        else:
            output, hn = self.rnn(packed_input)
            last_hidden = hn[-1]
        logits = self.dense(last_hidden)
        y_hat = torch.sigmoid(logits)
        return y_hat
```

The forward function takes in the review tokens as a sequence (x) and the actual unpadded number of the tokens (length).

- First, the tokens are converted to their respective embeddings with the help of the embedding layer. Then these sequence of embeddings are processed using `pack_padded_sequence` which also takes in the original length. This function is very important for tasks involving variable length sequences as our own. A naive approach to this would have been to send even the ¡PAD¿ tokens to the RNN/LSTM. But, this would create noise in the actual model predictions and result in lower scores.
- Then, this packed input is send to the recurrent layer (RNN / LSTM) out of which the last hidden state is extracted. This state is supposed to convey the entire context and sentiment of the whole review.
- Finally, this is used as input to the dense layer out of which probabilities (`y_hat`) are generated using sigmoid. These probabilities are returned.

5.2 Training

5.2.1 Dataset and Loader

```
from torch.utils.data import DataLoader

df_train = df.sample(frac=0.8, random_state=42)

training_dataset = list(zip(df_train["tokens"].to_list(),
↪  df_train["rating"].to_list(), df_train["length"].to_list(),
↪  df_train["id"].to_list())) #similarly val_dataset is created
training_loader = DataLoader(training_dataset, batch_size=3000,
↪  collate_fn=collate_fn) #similarly val_loader is created
```

As mentioned before, the dataset created contains not only the tokens and the labels but also, the unpadded length of tokens (to be used by the recurrent layer).

5.2.2 Train and Validate

Let's have a look at the training loop.

```python
from tqdm import tqdm #this is for a visual progressbar through epochs

def train(model, training_loader, lr, epochs, threshold=0.5,
↪  val_loader=None):
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    model.to(device)
    model.train()

    optimizer = torch.optim.Adam(model.parameters(), lr=lr)
    loss_fn = nn.BCELoss()

    losses = np.zeros(epochs)
    val_losses = np.zeros(epochs)
    val_f1 = np.zeros(epochs)
    for e in tqdm(range(epochs), leave=False):
        model.train()
        loss_epoch = 0
        for X_batch, y_batch, lengths, ids in training_loader:
            X_batch = X_batch.to(device)
            y_batch = y_batch.to(device)

            optimizer.zero_grad()
            y_hat = model(X_batch, lengths)
            loss = loss_fn(y_hat, y_batch)
            loss.backward()
            optimizer.step()

            loss_epoch += loss.item() * X_batch.shape[0]
        losses[e] = loss_epoch / len(training_loader.dataset)

        if val_loader:
            val_losses[e], val_f1[e] = validation_metrics(model, val_loader,
             ↪  loss_fn, threshold, device)
    return losses, val_losses, val_f1
```

Some specifications of the training are as follows:

- I have used the ADAM optimizer for a faster convergence of the loss function.
- Since the task is Binary Classification, the loss function used is Binary Cross Entropy Loss. As there is no class imbalance this time, so class sensitive weights were not required.
- With the help of a validation set, after each epoch, the loss and F1-scores are calculated (function in the ipynb file). Training loss after each epoch is also stored.

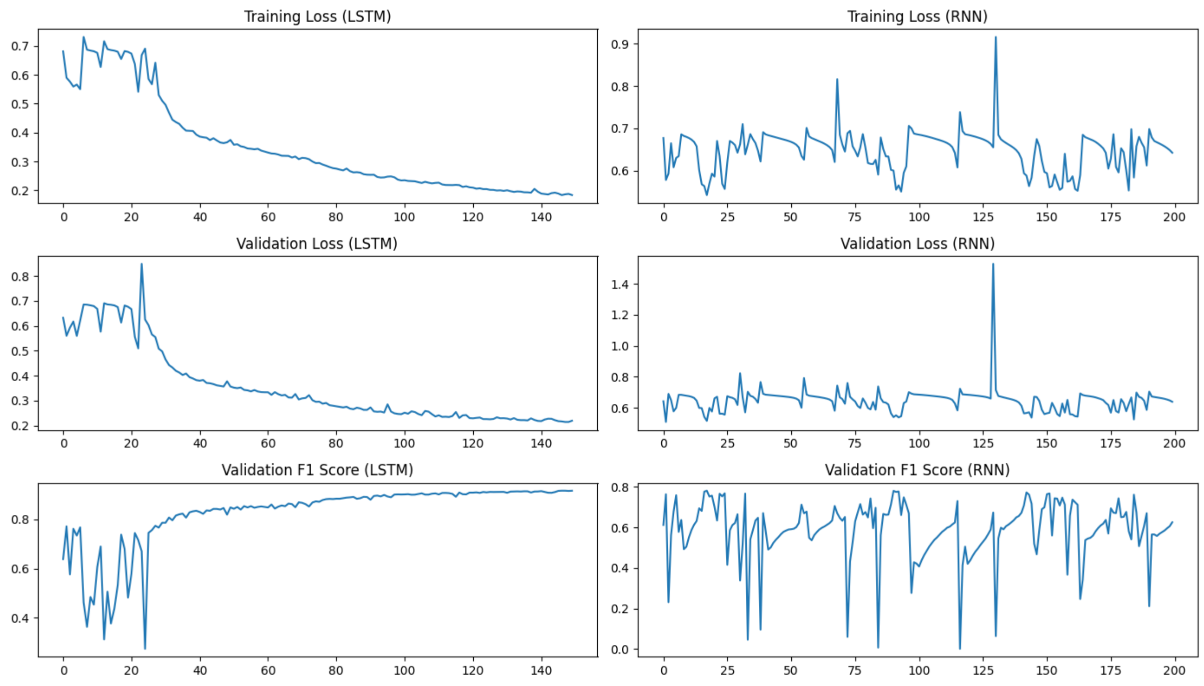

Figure 1: Training metrics of LSTM (left) and RNN (right)

```
lstm_sentiment = SentimentNN(dim, 128, embed_tensor)
train_loss, val_loss, val_f1 = train(lstm_sentiment, training_loader,
↪   0.0005, epochs=150, threshold=0.5, val_loader=val_loader)

rnn_sentiment = SentimentNN(dim, 128, embed_tensor, lstm=False)
train_loss_rnn, val_loss_rnn, val_f1_rnn = train(rnn_sentiment,
↪   training_loader, 0.0005, epochs=200, threshold=0.5,
↪   val_loader=val_loader)
```

5.2.3 Save Model Locally

Since, this model is computationally expensive and the dataset is massive, it is recommended to train the best possible model once and store its metrics and weight for future analysis and testing.

```
lstm_sentiment.to("cpu")
model = {
    "metrics": (train_loss, val_loss, val_f1),
    "model": lstm_sentiment.state_dict()
}
torch.save(model, "lstm_sentiment.pt")
lstm_sentiment.to("cuda")
```

5.3 Testing and Results

5.3.1 F1-Score, Accuracy, ROC-AUC

The function used for testing and the test loader can be found in the ipynb file if required

F1-Score	Accuracy	ROC-AUC
0.9122	91.13	0.97

Table 1: Performance Metrics of the LSTM Model

5.3.2 Confusion Matrix

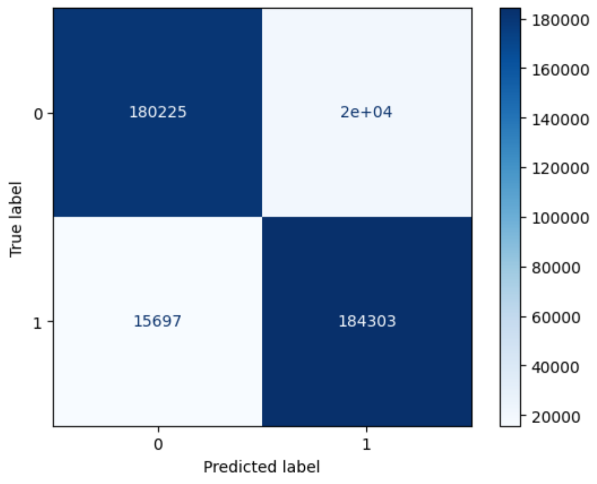

Figure 2: Confusion Matrix for the LSTM Model

5.3.3 ROC Curve

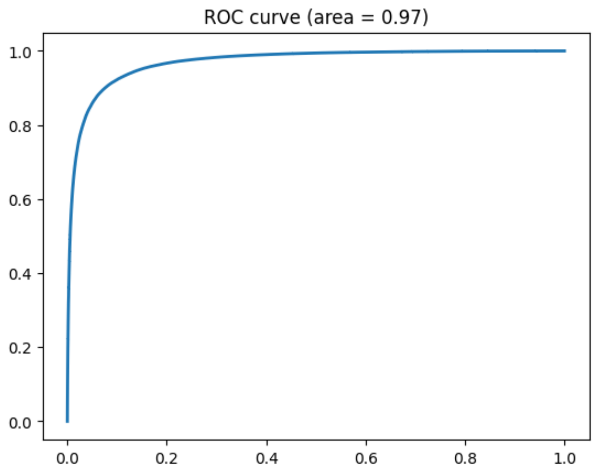

Figure 3: ROC Curve for the LSTM Model

6 Further Analysis

6.1 Minimal Clue Challenge

6.1.1 False Positives

- **Sarcasm and Tone Shift**
  "*great iron . died after 1 yr , . : this was a great iron . lots of steam , large water tank , and no water leakage . all translated into very easy ironing . sadly , the iron died after a little after 1 yr . black and decker just does not build durable irons . stay away from this one unless you are ok with a short lived appliance .* "

Label	Prediction	Probability
0	1	0.576

  Analysis The review begins with strong praise but ends with disappointment, introducing sarcasm and tone shift. So, the LSTM model is misled by the initial positive language. Although the model does not show a lot of positive feedback, it is still confused with almost neutral but slightly positive sentiment.

  Solution Training on sarcastic datasets with negative labels. So that the model gets to know that praise followed by contradiction is not positive.

- **Overweighing Positive Phrases**
  "*maybe i was wrong , the pundits are right , this device is basically perfect : i bought this hoping to speed upload of music and also comedy that i have on various media cassette , reeltoreel , lp and painstakingly recorded onto standard minidisc using a minidisc deck je630 . the range of features is very impressive . i particularly liked the ability to choose bit rate for conversion when transferring via sonic stage to himd . this enabled me to store 42 12 hour radio comedy programs goon shows on a single md . i had some problems which i reported previously , and which report i now unreservedly withdraw . something is not 100 right for me but most probably it is not the mzrh1 . no i was not contacted by anyone at sony , this is merely in the interests of public transparency and honesty . i would caution all users to make sure that they check the sound quality of what is being uploaded and saved . too bad , i cannot give it 5 stars any more , the software would not let me alter that for the review .* "

Label	Prediction	Probability
0	1	0.879

  Analysis The model latched onto strong positive words like "*basically perfect*", "*very impressive*". But, the overall sentiment is cautious. Also, the reviewer does not explicitly write bad phrases as strongly as good phrases are written. It is also possible that because of the noise produced by OOV words like "*reeltoreel*", "*je630*" etc. caused a rise in positive feedback. The model also failed to track shift in phrases. But an interesting point to be noted is that the reviewer themselves say that they want to change the review but the software is not letting them alter it. So, it could just be a case of mislabelling too.

6.1.2 False Negatives

- **Negation of Negative word Unrecognized**

  "*album of the year . : although some people may refer to this album as a disappointment , i think it is one of the best albums i have heard . it it so much more developed than afi's previous albums and you can tell just by one listen to the album . the hard work has obviously paid off because there is more texture in the songs and , of course also the trademark afi lyrics and chants . i do not think i will ever get sick of this album !* "

Label	Prediction	Probability
1	0	0.292

  Analysis The model sees the emotionally heavy negative word "disappointment" and fails to realize that it is being disagreed with, and not expressed. The phrase "some people may refer to..." is too subtle for the model to consider this "negation of disappointment" as praising. In short, this is a confusion arised due to confusion in negation. Upon further research, it was found out that this problem is called *scope-of-negation failure.* It is talked about in detail in Making Language Models Robust Against Negation by University of Arizona

- **Confusion due to Context Unawareness**

  "*see for yourself : i could see that my cousin lost 10 lbs in couple weeks , not only that i have seen it w my own eyes that his fat belly's gone . oh yeah , he is lab rat and i ordered one .* "

Label	Prediction	Probability
1	0	0.355

  Analysis This is a context dependent review, where sentiment analysis requires more input (visual cues "*see*"" and context awareness). The product success i.e fat loss is not connected with positive sentiment due to use of negative words like "*loss*". Lack of external world knowledge results in misclassification in this case.