LSTM Text Classification Using Pytorch

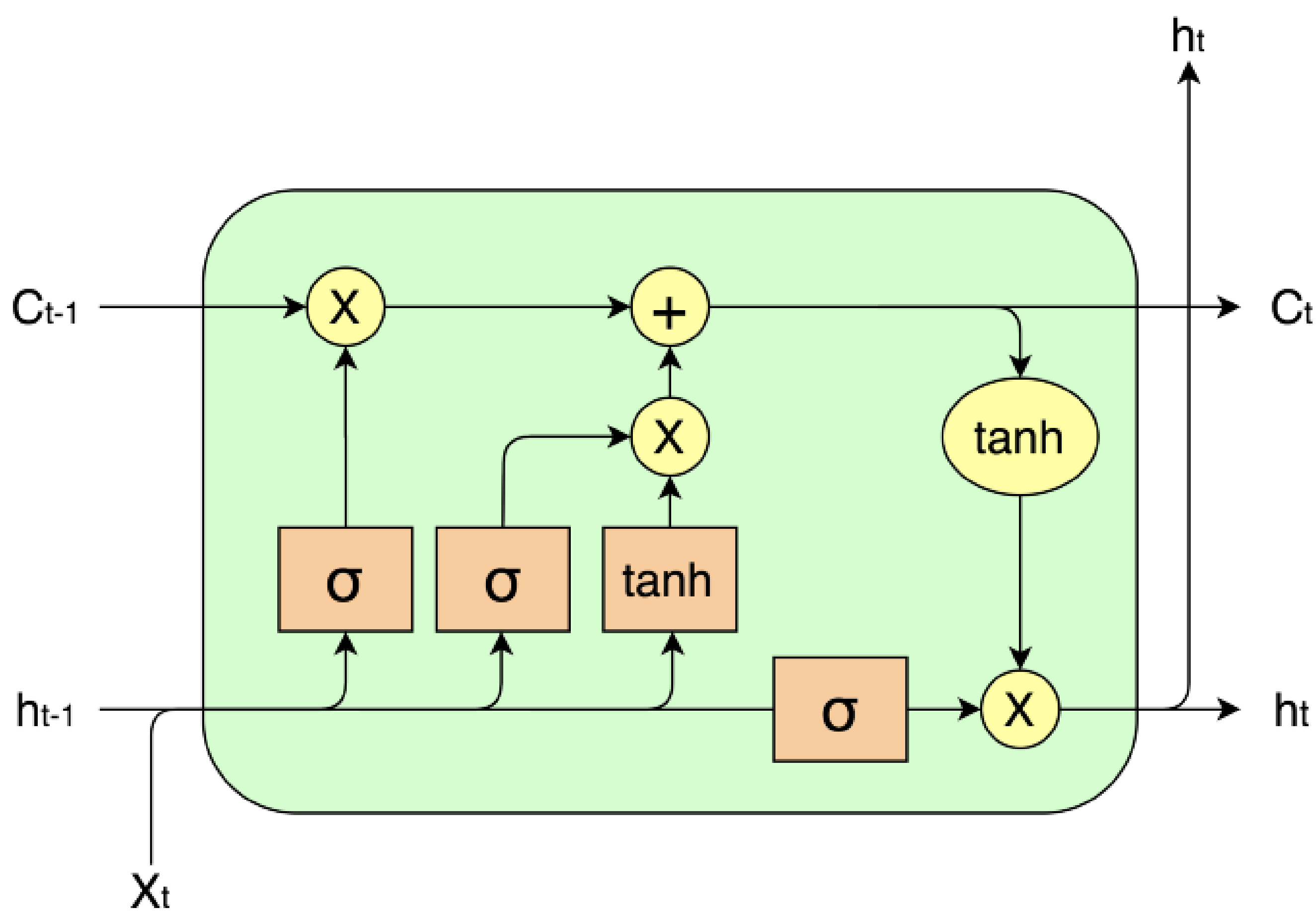
A step-by-step guide teaching you how to build a bidirectional LSTM in Pytorch!

Raymond Cheng Jul 1 · 5 min read ★

Intro

Welcome to this tutorial! This tutorial will teach you how to build a bidirectional **LSTM** for text classification in just a few minutes. If you haven't already checked out my previous article on **BERT Text Classification**, this tutorial contains similar code with that one but contains some modifications to support LSTM. This article also gives explanations on how I preprocessed the dataset used in both articles, which is **the REAL and FAKE News Dataset** from Kaggle.

First of all, what is an LSTM and why do we use it? LSTM stands for **Long Short-Term Memory Network**, which belongs to a larger category of neural networks called **Recurrent Neural Network (RNN)**. Its main advantage over the vanilla RNN is that it is better capable of handling long term dependencies through its sophisticated architecture that includes three different gates: input gate, output gate, and the forget gate. The three gates operate together to decide what information to remember and what to forget in the LSTM cell over an arbitrary time.



Now, we have a bit more understanding of LSTM, let's focus on how to implement it for text classification. The tutorial is divided into the following steps:

1. Preprocess Dataset
2. Importing Libraries
3. Load Dataset
4. Build Model
5. Training
6. Evaluation

Before we dive right into the tutorial, here is where you can access the code in this article:

- **Preprocessing of Fake News Dataset**
- **LSTM Text Classification Google Colab**

Step 1: Preprocess Dataset

The raw dataset looks like the following:

	Unnamed: 0	title	text	label
0	8476	You Can Smell Hillary's Fear	Daniel Greenfield, a Shillman Journalism Fello...	FAKE
1	10294	Watch The Exact Moment Paul Ryan Committed Pol...	Google Pinterest Digg Linkedin Reddit Stumbleu...	FAKE
2	3608	Kerry to go to Paris in gesture of sympathy	U.S. Secretary of State John F. Kerry said Mon...	REAL
3	10142	Bernie supporters on Twitter erupt in anger ag...	— Kaydee King (@KaydeeKing) November 9, 2016 T...	FAKE
4	875	The Battle of New York: Why This Primary Matters	It's primary day in New York and front-runners...	REAL
...	...	...	...	...
6330	4490	State Department says it can't find emails fro...	The State Department told the Republican Natio...	REAL
6331	8062	The 'P' in PBS Should Stand for 'Plutocratic' ...	The 'P' in PBS Should Stand for 'Plutocratic' ...	FAKE
6332	8622	Anti-Trump Protesters Are Tools of the Oligarc...	Anti-Trump Protesters Are Tools of the Oligar...	FAKE
6333	4021	In Ethiopia, Obama seeks progress on peace, se...	ADDIS ABABA, Ethiopia —President Obama convene...	REAL
6334	4330	Jeb Bush Is Suddenly Attacking Trump. Here's W...	Jeb Bush Is Suddenly Attacking Trump. Here's W...	REAL

6335 rows × 4 columns

Dataset Overview

```
import pandas as pd
from sklearn.model_selection import train_test_split

raw_data_path = '/content/drive/My Drive/lstm/Data/news.csv'
destination_folder = '/content/drive/My Drive/lstm/Data'

train_test_ratio = 0.10
train_valid_ratio = 0.80

first_n_words = 200

def trim_string(x):
    x = x.split(maxsplit=first_n_words)
    x = ' '.join(x[:first_n_words])
    return x
```

libraries_variables.py hosted with ❤ by **GitHub** **view raw**

For preprocessing, we import Pandas and Sklearn and define some variables for path, training validation and test ratio, as well as the `trim_string` function which will be used to cut each sentence to the first `first_n_words` words. Trimming the samples in a dataset is not necessary but it enables faster training for heavier models and is normally enough to predict the outcome.

```
# Read raw data
df_raw = pd.read_csv(raw_data_path)

# Prepare columns
df_raw['label'] = (df_raw['label'] == 'FAKE').astype('int')
df_raw['titletext'] = df_raw['title'] + ". " + df_raw['text']
df_raw = df_raw.reindex(columns=['label', 'title', 'text', 'titletext'])

# Drop rows with empty text
df_raw.drop( df_raw[df_raw.text.str.len() < 5].index, inplace=True)

# Trim text and titletext to first_n_words
df_raw['text'] = df_raw['text'].apply(trim_string)
df_raw['titletext'] = df_raw['titletext'].apply(trim_string)

# Split according to label
```

```
# Train-test split
df_real_full_train, df_real_test = train_test_split(df_real, train_size = train_test_rat
df_fake_full_train, df_fake_test = train_test_split(df_fake, train_size = train_test_rat

# Train-valid split
df_real_train, df_real_valid = train_test_split(df_real_full_train, train_size = train_v
df_fake_train, df_fake_valid = train_test_split(df_fake_full_train, train_size = train_v

# Concatenate splits of different labels
df_train = pd.concat([df_real_train, df_fake_train], ignore_index=True, sort=False)
df_valid = pd.concat([df_real_valid, df_fake_valid], ignore_index=True, sort=False)
df_test = pd.concat([df_real_test, df_fake_test], ignore_index=True, sort=False)

# Write preprocessed data
df_train.to_csv(destination_folder + '/train.csv', index=False)
df_valid.to_csv(destination_folder + '/valid.csv', index=False)
df_test.to_csv(destination_folder + '/test.csv', index=False)
```

preprocess.py hosted with ❤ by **GitHub** **view raw**

Next, we convert *REAL* to 0 and *FAKE* to 1, concatenate *title* and *text* to form a new column *titletext* (we use both the title and text to decide the outcome), drop rows with empty text, trim each sample to the `first_n_words` , and split the dataset according to `train_test_ratio` and `train_valid_ratio` . We save the resulting dataframes into *.csv* files, getting *train.csv*, *valid.csv*, and *test.csv*.

Step 2: Importing Libraries

```
# Libraries

import matplotlib.pyplot as plt
import pandas as pd
import torch

# Preliminaries

from torchtext.data import Field, TabularDataset, BucketIterator

# Models
```

```
from torch.nn.utils.rnn import pack_padded_sequence, pad_packed_sequence

# Training

import torch.optim as optim

# Evaluation

from sklearn.metrics import accuracy_score, classification_report, confusion_matrix
import seaborn as sns
```

imports.py hosted with ❤ by GitHub view raw

We import Pytorch for model construction, torchText for loading data, matplotlib for plotting, and sklearn for evaluation.

Step 3: Load Dataset

```
# Fields

label_field = Field(sequential=False, use_vocab=False, batch_first=True, dtype=torch.flo
text_field = Field(tokenize='spacy', lower=True, include_lengths=True, batch_first=True)
fields = [('label', label_field), ('title', text_field), ('text', text_field), ('titlete

# TabularDataset

train, valid, test = TabularDataset.splits(path=source_folder, train='train.csv', valida
                                           format='CSV', fields=fields, skip_header=True

# Iterators

train_iter = BucketIterator(train, batch_size=32, sort_key=lambda x: len(x.text),
                            device=device, sort=True, sort_within_batch=True)
valid_iter = BucketIterator(valid, batch_size=32, sort_key=lambda x: len(x.text),
                            device=device, sort=True, sort_within_batch=True)
test_iter = BucketIterator(test, batch_size=32, sort_key=lambda x: len(x.text),
                            device=device, sort=True, sort_within_batch=True)

# Vocabulary

text_field.build_vocab(train, min_freq=3)
```

First, we use torchText to create a label field for the *label* in our dataset and a text field for the *title*, *text*, and *titletext*. We then build a TabularDataset by pointing it to the path containing the *train.csv*, *valid.csv*, and *test.csv* dataset files. We create the train, valid, and test iterators that load the data, and finally, build the vocabulary using the train iterator (counting only the tokens with a minimum frequency of 3).

Step 4: Build Model

```
class LSTM(nn.Module):

    def __init__(self, dimension=128):
        super(LSTM, self).__init__()

        self.embedding = nn.Embedding(len(text_field.vocab), 300)
        self.dimension = dimension
        self.lstm = nn.LSTM(input_size=300,
                            hidden_size=dimension,
                            num_layers=1,
                            batch_first=True,
                            bidirectional=True)
        self.drop = nn.Dropout(p=0.5)

        self.fc = nn.Linear(2*dimension, 1)

    def forward(self, text, text_len):

        text_emb = self.embedding(text)

        packed_input = pack_padded_sequence(text_emb, text_len, batch_first=True, enforc
        packed_output, _ = self.lstm(packed_input)
        output, _ = pad_packed_sequence(packed_output, batch_first=True)

        out_forward = output[range(len(output)), text_len - 1, :self.dimension]
        out_reverse = output[:, 0, self.dimension:]
        out_reduced = torch.cat((out_forward, out_reverse), 1)
        text_fea = self.drop(out_reduced)

        text_fea = self.fc(text_fea)
        text_fea = torch.squeeze(text_fea, 1)
```

```
        return text_out
```

model.py hosted with ❤ by GitHub view raw

We construct the LSTM class that inherits from the *nn.Module*. Inside the LSTM, we construct an Embedding layer, followed by a bi-LSTM layer, and ending with a fully connected linear layer. In the *forward* function, we pass the text IDs through the embedding layer to get the embeddings, pass it through the LSTM accommodating variable-length sequences, learn from both directions, pass it through the fully connected linear layer, and finally *sigmoid* to get the probability of the sequences belonging to FAKE (being 1).

Step 5: Training

```
# Save and Load Functions

def save_checkpoint(save_path, model, optimizer, valid_loss):

    if save_path == None:
        return

    state_dict = {'model_state_dict': model.state_dict(),
                  'optimizer_state_dict': optimizer.state_dict(),
                  'valid_loss': valid_loss}

    torch.save(state_dict, save_path)
    print(f'Model saved to ==> {save_path}')


def load_checkpoint(load_path, model, optimizer):

    if load_path==None:
        return

    state_dict = torch.load(load_path, map_location=device)
    print(f'Model loaded from <== {load_path}')

    model.load_state_dict(state_dict['model_state_dict'])
    optimizer.load_state_dict(state_dict['optimizer_state_dict'])
```

```python

def save_metrics(save_path, train_loss_list, valid_loss_list, global_steps_list):

    if save_path == None:
        return

    state_dict = {'train_loss_list': train_loss_list,
                  'valid_loss_list': valid_loss_list,
                  'global_steps_list': global_steps_list}

    torch.save(state_dict, save_path)
    print(f'Model saved to ==> {save_path}')


def load_metrics(load_path):

    if load_path==None:
        return

    state_dict = torch.load(load_path, map_location=device)
    print(f'Model loaded from <== {load_path}')

    return state_dict['train_loss_list'], state_dict['valid_loss_list'], state_dict['glo
```

save_load.py hosted with ❤ by GitHub view raw

Before training, we build save and load functions for checkpoints and metrics. For checkpoints, the model parameters and optimizer are saved; for metrics, the train loss, valid loss, and global steps are saved so diagrams can be easily reconstructed later.

```python
# Training Function

def train(model,
          optimizer,
          criterion = nn.BCELoss(),
          train_loader = train_iter,
          valid_loader = valid_iter,
          num_epochs = 5,
          eval_every = len(train_iter) // 2,
          file_path = destination_folder,
```

```
    # initialize running values
    running_loss = 0.0
    valid_running_loss = 0.0
    global_step = 0
    train_loss_list = []
    valid_loss_list = []
    global_steps_list = []

    # training loop
    model.train()
    for epoch in range(num_epochs):
        for (labels, (title, title_len), (text, text_len), (titletext, titletext_len)),
            labels = labels.to(device)
            titletext = titletext.to(device)
            titletext_len = titletext_len.to(device)
            output = model(titletext, titletext_len)

            loss = criterion(output, labels)
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()

            # update running values
            running_loss += loss.item()
            global_step += 1

            # evaluation step
            if global_step % eval_every == 0:
                model.eval()
                with torch.no_grad():
                  # validation loop
                  for (labels, (title, title_len), (text, text_len), (titletext, titlete
                      labels = labels.to(device)
                      titletext = titletext.to(device)
                      titletext_len = titletext_len.to(device)
                      output = model(titletext, titletext_len)

                      loss = criterion(output, labels)
                      valid_running_loss += loss.item()

                # evaluation
                average_train_loss = running_loss / eval_every
```

```
                global_steps_list.append(global_step)

                # resetting running values
                running_loss = 0.0
                valid_running_loss = 0.0
                model.train()

                # print progress
                print('Epoch [{}/{}], Step [{}/{}], Train Loss: {:.4f}, Valid Loss: {:.4
                      .format(epoch+1, num_epochs, global_step, num_epochs*len(train_loa
                              average_train_loss, average_valid_loss))

                # checkpoint
                if best_valid_loss > average_valid_loss:
                    best_valid_loss = average_valid_loss
                    save_checkpoint(file_path + '/model.pt', model, optimizer, best_vali
                    save_metrics(file_path + '/metrics.pt', train_loss_list, valid_loss_

    save_metrics(file_path + '/metrics.pt', train_loss_list, valid_loss_list, global_ste
    print('Finished Training!')


model = LSTM().to(device)
optimizer = optim.Adam(model.parameters(), lr=0.001)

train(model=model, optimizer=optimizer, num_epochs=10)
```

training.py hosted with ❤ by **GitHub** **view raw**

We train the LSTM with 10 epochs and save the checkpoint and metrics whenever a hyperparameter setting achieves the best (lowest) validation loss. Here is the output during training:

```
Epoch [1/10], Step [8/160], Train Loss: 0.6933, Valid Loss: 0.6729
Model saved to ==> /content/drive/My Drive/lstm/Model/model.pt
Model saved to ==> /content/drive/My Drive/lstm/Model/metrics.pt
Epoch [1/10], Step [16/160], Train Loss: 0.6914, Valid Loss: 0.6588
Model saved to ==> /content/drive/My Drive/lstm/Model/model.pt
Model saved to ==> /content/drive/My Drive/lstm/Model/metrics.pt
Epoch [2/10], Step [24/160], Train Loss: 0.5762, Valid Loss: 0.6457
Model saved to ==> /content/drive/My Drive/lstm/Model/model.pt
Model saved to ==> /content/drive/My Drive/lstm/Model/metrics.pt
Epoch [2/10], Step [32/160], Train Loss: 0.5932, Valid Loss: 0.6317
Model saved to ==> /content/drive/My Drive/lstm/Model/model.pt
```

```
Model saved to ==> /content/drive/My Drive/lstm/Model/model.pt
Model saved to ==> /content/drive/My Drive/lstm/Model/metrics.pt
Epoch [4/10], Step [56/160], Train Loss: 0.3592, Valid Loss: 0.5966
Model saved to ==> /content/drive/My Drive/lstm/Model/model.pt
Model saved to ==> /content/drive/My Drive/lstm/Model/metrics.pt
Epoch [4/10], Step [64/160], Train Loss: 0.3295, Valid Loss: 0.5513
Model saved to ==> /content/drive/My Drive/lstm/Model/model.pt
Model saved to ==> /content/drive/My Drive/lstm/Model/metrics.pt
Epoch [5/10], Step [72/160], Train Loss: 0.2058, Valid Loss: 0.4844
Model saved to ==> /content/drive/My Drive/lstm/Model/model.pt
Model saved to ==> /content/drive/My Drive/lstm/Model/metrics.pt
Epoch [5/10], Step [80/160], Train Loss: 0.1669, Valid Loss: 0.5313
Epoch [6/10], Step [88/160], Train Loss: 0.1015, Valid Loss: 0.4999
Epoch [6/10], Step [96/160], Train Loss: 0.0744, Valid Loss: 0.4997
Epoch [7/10], Step [104/160], Train Loss: 0.0420, Valid Loss: 0.6247
Epoch [7/10], Step [112/160], Train Loss: 0.0488, Valid Loss: 0.4529
Model saved to ==> /content/drive/My Drive/lstm/Model/model.pt
Model saved to ==> /content/drive/My Drive/lstm/Model/metrics.pt
Epoch [8/10], Step [120/160], Train Loss: 0.0272, Valid Loss: 1.0449
Epoch [8/10], Step [128/160], Train Loss: 0.0533, Valid Loss: 0.4725
Epoch [9/10], Step [136/160], Train Loss: 0.0546, Valid Loss: 0.6471
Epoch [9/10], Step [144/160], Train Loss: 0.0397, Valid Loss: 0.5033
Epoch [10/10], Step [152/160], Train Loss: 0.0308, Valid Loss: 0.5138
Epoch [10/10], Step [160/160], Train Loss: 0.0223, Valid Loss: 0.5254
Model saved to ==> /content/drive/My Drive/lstm/Model/metrics.pt
Finished Training!
```

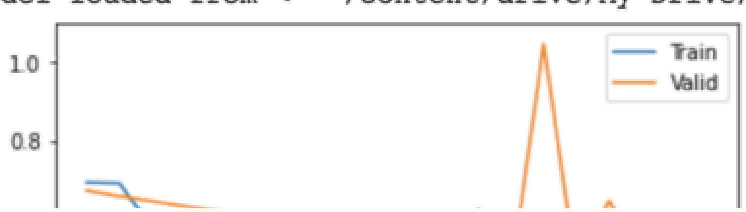
The whole training process was fast on Google Colab. It took less than two minutes to train!

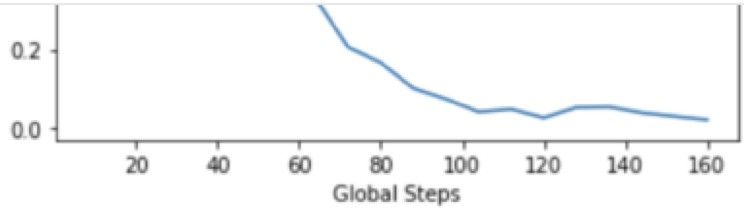
```python
train_loss_list, valid_loss_list, global_steps_list = load_metrics(destination_folder + '
plt.plot(global_steps_list, train_loss_list, label='Train')
plt.plot(global_steps_list, valid_loss_list, label='Valid')
plt.xlabel('Global Steps')
plt.ylabel('Loss')
plt.legend()
plt.show()
```

visualization.py hosted with ❤ by **GitHub** **view raw**

Once we finished training, we can load the metrics previously saved and output a diagram showing the training loss and validation loss throughout time.

```
Model loaded from <== /content/drive/My Drive/lstm/Model/metrics.pt
```





Step 6: Evaluation

```
# Evaluation Function

def evaluate(model, test_loader, version='title', threshold=0.5):
    y_pred = []
    y_true = []

    model.eval()
    with torch.no_grad():
        for (labels, (title, title_len), (text, text_len), (titletext, titletext_len)),
            labels = labels.to(device)
            titletext = titletext.to(device)
            titletext_len = titletext_len.to(device)
            output = model(titletext, titletext_len)

            output = (output > threshold).int()
            y_pred.extend(output.tolist())
            y_true.extend(labels.tolist())

    print('Classification Report:')
    print(classification_report(y_true, y_pred, labels=[1,0], digits=4))

    cm = confusion_matrix(y_true, y_pred, labels=[1,0])
    ax= plt.subplot()
    sns.heatmap(cm, annot=True, ax = ax, cmap='Blues', fmt="d")

    ax.set_title('Confusion Matrix')

    ax.set_xlabel('Predicted Labels')
    ax.set_ylabel('True Labels')

    ax.xaxis.set_ticklabels(['FAKE', 'REAL'])
    ax.yaxis.set_ticklabels(['FAKE', 'REAL'])

```

```
optimizer = optim.Adam(best_model.parameters(), lr=0.001)

load_checkpoint(destination_folder + '/model.pt', best_model, optimizer)
evaluate(best_model, test_iter)
```

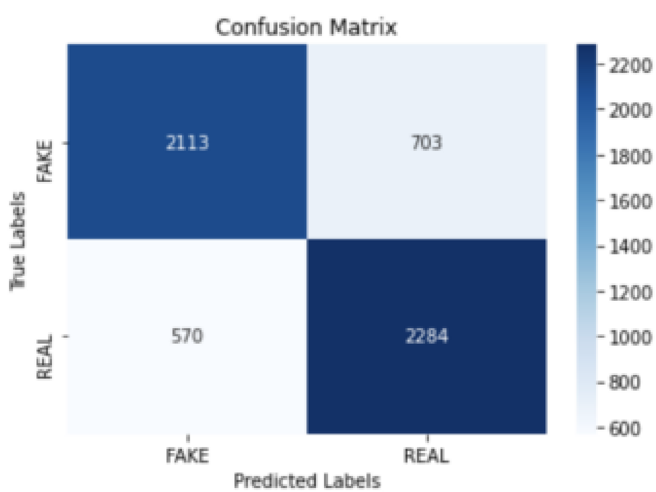
evaluation.py hosted with ❤ by GitHub view raw

Finally for evaluation, we pick the best model previously saved and evaluate it against our test dataset. We use a default threshold of 0.5 to decide when to classify a sample as FAKE. If the model output is greater than 0.5, we classify that news as FAKE; otherwise, REAL. We output the classification report indicating the precision, recall, and F1-score for each class, as well as the overall accuracy. We also output the confusion matrix.

```
Model loaded from <== /content/drive/My Drive/lstm/Model/model.pt
Classification Report:
              precision    recall  f1-score   support

           1     0.7876    0.7504    0.7685      2816
           0     0.7646    0.8003    0.7821      2854

    accuracy                         0.7755      5670
   macro avg     0.7761    0.7753    0.7753      5670
weighted avg     0.7760    0.7755    0.7753      5670
```



We can see that with a one-layer bi-LSTM, we can achieve an accuracy of 77.53% on the fake news detection task.

Conclusion

accuracy for fake news detection but still has room to improve. If you want a more competitive performance, check out my previous article on **BERT Text Classification**!

BERT Text Classification Using Pytorch

Text classification is a common task in NLP. We apply BERT, a popular Transformer model, on fake news detection using…

towardsdatascience.com

If you want to learn more about modern NLP and deep learning, make sure to follow me for updates on upcoming articles :)

References

[1] S. Hochreiter, J. Schmidhuber, Long Short-Term Memory (1997), Neural Computation