NANYANG TECHNOLOGICAL UNIVERSITY
SINGAPORE

AI6126 DEEP NEURAL NETWORK FOR NATURAL LANGUAGE PROCESSING

ASSIGNMENT 1: DEEP LEARNING MODELS FOR SENTIMENT CLASSIFICATION

School of Computer Science and Engineering

Programme: MSAI

Date: 12 March 2024

Authored By: Tan Jie Heng Alfred (G2304193L)

Contents

1. Introduction

1.1 Sentiment Classification

In this project, we aim to train models for a sentiment classification task. The dataset which we work on is the iMDB movie review, and the sentiment classification task requires the machine to classify a given review (i.e., document) as either positive (1) or negative (0). Therefore, this is a binary classification task, where our dataset consists of textual data. There are many model architectures that could take in textual data. Here, we will explore the use of recurrent neural network (RNN), feedforward network (FFN), convolutional neural network (CNN), and long short-term memory network (LSTM), for our classification task.

1.2 Data Preparation and Evaluation

In this section, we briefly outline how the data is being processed. The iMDB movie review dataset could be downloaded from `torch.text.datasets`, which has been split into training and test sets. The documents in the dataset are then tokenized with the `spaCy` tokenizer. For the training set, it is further split into `train` and `valid` sets, where the latter validation set is typically used for hyperparameter selection. We then use the `train` set to build our vocabulary, clipping it at a maximum of 25,000 words. Finally, when the data are split into minibatches, each document is padded up to the longest document in its mini batch – the padding is not consistent across the entire dataset.

2. Recurrent Neural Network

2.1 Randomly Initialized Embedding

The RNN architecture allows for the processing of sequential data. In order to work with textual data, each token (i.e., `string`) has to first be converted to a vector (i.e., `float`), in a process called token embedding. In this section, we experiment with the randomly initialized embedding using `nn.Embedding`. Specifically, the tokens in the vocabulary are being projected onto a vector space (of dimension `embedding_dim`), by the weights in `nn.Embedding`, which are initialized from $\mathcal{N}(0,1)$. Here, we set `embedding_dim = 100`.

Next, we feed these vector embeddings sequentially into our RNN. This is implemented using `nn.RNN`, and we set the dimension of each hidden state at time t, h_t, to be `hidden_dim = 256`. For our classification task, we take the final hidden state, h_T, and feed it through a fully connected layer (implemented using `nn.Linear`) to get a single output logit, $\hat{y}$. Then, to train the binary classification model, we aim to minimize the binary cross entropy loss (implemented using `nn.BCEWithLogitsLoss`), with the final output logit, $\hat{y}$, as our prediction. This loss function will be used for all the subsequent experiments.

As we train the model on the `train` set, we concurrently evaluate it on the validation set (i.e., `valid`). Then, after training completion, we choose our final model to be parametrized by the weights corresponding to the lowest validation loss. As the model would not have been trained on the validation set, it acts as a proxy to how well the model performs on unseen data. Finally, to evaluate the performance of the model, we will evaluate the model against the holdout test set and calculate the test set accuracy. Accuracy is defined to be the number of correct predictions over the total number of data instances in the (test) set.

In this section, we will present both the test accuracy and test loss of our implemented RNN (see Appendix (a)), against different optimizers and number of epochs of training. In particular, we trained the RNN using Stochastic Gradient Descent (i.e., `optim.SGD`), Adam (i.e., `optim.Adam`), and Adagrad (i.e., `optim.Adagrad`) optimizers, for 5, 10, 20, and 50 epochs. For each of the optimizer, we set the learning rate to be 10^{-3}. The results are shown in Table 1 below.

From Table 1, we observe that the test loss and accuracy seem to improve with increasing number of epochs. This is not unexpected as the model is able to train for more iterations and therefore better able to fit the training set. Ideally, this allows the model to also better model the underlying data distribution. On the other hand, with a fixed number of epochs, Adagrad seems to consistently produce the best results, slightly edging above Adam, while SGD always produces the worst. This is likely because both Adagrad and Adam optimizers include terms that account for historic gradients, therefore having stabler weight updates during training. Moving forward, we will use Adam as our optimizer, with learning rate of 10^{-3}, and train each of the models for 50 epochs.

	Epoch = 5	Epoch = 10	Epoch = 20	Epoch = 50
Stochastic Gradient Descent (SGD)	52.87 % (0.690)	53.95% (0.688)	55.49% (0.684)	**56.82%** **(0.676)**
Adam	61.13% (0.657)	62.91% (0.705)	67.28% (0.607)	**69.61%** **(0.652)**
Adagrad	62.07% (0.655)	66.44% (0.618)	69.02% (0.591)	**74.66%** **(0.523)**

Table 1: Accuracy (Loss) of RNN with randomly initialized embedding evaluated on test set.

2.2 Pre-trained Word2Vec Embedding

In this section, we experiment with the embedding layer. In particular, we use a pre-trained Word2Vec embedding layer, which was previously trained on Google News, instead of a randomly initialized one as seen in Section 2.1. This pre-trained embedding layer (henceforth called W2V) embeds each token as a vector of 300 dimensions. Because W2V was trained on a different but much larger dataset, we have to compare the words embedded within W2V with the ones in our vocabulary. As it turns out, there are out-of-vocabulary (OOV) tokens, which appear in our training dataset but not in the Google News. This means that there are no corresponding embedding vectors within W2V for these OOV tokens.

We devised three ways (see Appendix, (b) – (d)) to deal with these OOV tokens:

1. Map these words to the embedding vector of the string `'unk'` in W2V, which we assumed to be the placeholder similar to the `'<unk>'` token in our vocabulary.
2. Sample the OOV word embeddings from $\mathcal{N}(\boldsymbol{\mu}_{\text{W2V}}, \boldsymbol{\sigma}^2_{\text{W2V}})$, where $\boldsymbol{\mu}_{\text{W2V}}$ and $\boldsymbol{\sigma}^2_{\text{W2V}}$ refer to the mean and variance of all the word embeddings within W2V.
3. Sample the OOV word embeddings from $\mathcal{N}(\boldsymbol{\mu}_{\text{vocab}}, \boldsymbol{\sigma}^2_{\text{vocab}})$, where $\boldsymbol{\mu}_{\text{vocab}}$ and $\boldsymbol{\sigma}^2_{\text{vocab}}$ refer to the mean and variance of all the matched word embeddings within our vocabulary.

Altogether, we replaced the embedding of the in-vocabulary words with their corresponding embeddings in W2V and experimented with the three methods of sampling the OOV word embeddings – these form our pre-trained embedding layer. Additionally, we experimented with 'freezing' and 'unfreezing' the weights of our embedding layer, controlling whether the embedding could be further fine-tuned during training. We report the test accuracy and loss of our experiments in Table 2.

	Method 1	Method 2	Method 3
Embedding weights 'frozen'	71.21% (0.582)	65.80% (0.620)	**73.48%** **(0.559)**
Embedding weights 'unfrozen'	76.18% (0.568)	**77.55%** **(0.511)**	76.39% (0.537)

Table 2: Accuracy (Loss) of RNN with W2V embedding evaluated on test set.

We can see that by allowing the embedding layer for all three methods to further fine-tune, the performance of the model increases. This is to be expected, since the embedding layer would learn a better representation (i.e., word embedding) that models the distribution of our data (i.e., movie reviews). This better representation could be used more meaningfully for our downstream classification task. Interestingly, they all outperformed the trained, randomly initialized embedding used in Section 2.1, which yielded a test accuracy of 69.61%. This suggests that some semantics captured by W2V on the larger Google News dataset are meaningful for our classification task.

However, if we decide to not fine-tune the pre-trained embedding layer, the performance of our model depends on the method in which we sample the OOV word embeddings. Specifically, by replacing OOV word embeddings using method 2, we get an appreciably worse result as compared to either method 1 or 3. This could be explained by the difference in domain for which the data are being observed – W2V was pre-trained on Google News which may not contain semantics relevant to iMDB movie reviews. This means that the OOV words are distributed significantly differently than most of the words in Google News and sampling them from $\mathcal{N}(\boldsymbol{\mu}_{\text{W2V}}, \boldsymbol{\sigma}^2_{\text{W2V}})$ builds a very poor representation of our underlying distribution, hence affecting the downstream classification task.

Along the same vein, when we sample from the space induced by the in-vocabulary word embeddings (i.e., $\mathcal{N}(\boldsymbol{\mu}_{\text{vocab}}, \boldsymbol{\sigma}^2_{\text{vocab}})$), the resulting OOV word embeddings would capture semantics relevant to these in-

vocabulary words, thereby capturing semantics relevant to our training set. On the other hand, some of these OOV words may have been classified as `'unk'`, similar to `'<unk>'` in our dataset. Because the pre-trained W2V has already captured the semantics of the tokens within the word embeddings, the semantics of our OOV words could have been captured by the embedding vector of `'unk`. Hence, both methods 1 and 2 provide better initial representations of our dataset, by leveraging on the semantics captured by W2V, thereby providing better results without further fine-tuning of the embedding layer.

3. Other Models

Finally, we experimented with other models, specifically, FFN, CNN and LSTM. For all the models, we used a randomly initialized embedding layer, with `embedding_dim = 100`, followed by a final linear prediction layer. In this section, we first describe how each of the architecture is implemented before tabulating and analysing the results of the models.

3.1 Feedforward Network

We experimented with three FFN architectures: FFN-1, which is a one hidden-layer network of dimension 500; FFN-2, which is a two hidden-layer dimension of 500 followed by 300; and FFN-3, which is a three hidden-layer network with dimensions 500, 300 and 200 respectively (see Appendix, (e) – (g)). We applied the FFN independently on each of the word embeddings, then we performed a mean-pooling on the output of the final hidden layer, before feeding the resulting feature into the final linear prediction layer. The mean-pooling is performed across all the features in the sentence. This is done to 'consolidate' the semantics of all the features in the sentence. Another benefit of mean pooling is that it reduces the number of parameters and hence computational cost, as opposed to concatenating all the word embeddings in a sentence before feeding into the FFN,

3.2 Convolutional Neural Network

We experimented with a CNN (see Appendix (h)) that produces three feature maps of sizes 1, 2, and 3. Each feature map has dimension `hidden_dim` (i.e number of channels). Then, we perform max pooling on each of the feature map across their spatial dimension, before concatenating the three feature maps to form a final feature map of dimension `3*hidden_dim`. This final feature map will be used as input into the final linear prediction layer. The architecture is akin to finding the most "important" n-gram ($n = 1, 2, 3$), where the filter size and weights determine the value of n and 'importance' metric respectively, while the max-pooling chooses only the n-gram of highest 'importance'.

3.3 LSTM

Finally, we experimented with two LSTM models: A uni-directional (i.e., LSTM) and a bi-directional one (i.e., Bi-LSTM). The former would capture semantics by only considering each sequence (i.e., sentence) from left-to-right, whereas the latter provides a second direction from right-to-left. Because of this, the usual LSTM will only have one final hidden state, $h_T^{(1)}$, but the Bi-LSTM would have two hidden states, $h_{T,\text{forward}}^{(2)}$ and $h_{T,\text{backward}}^{(2)}$, each corresponding to one direction. Hence, for the usual LSTM, we use $h_T^{(1)}$ as our input into the final linear prediction layer, whereas for Bi-LSTM, we use $h = \text{concatenate}\left[h_{T,\text{left}}^{(2)}, h_{T,\text{right}}^{(2)}\right]$ as the input into the final linear prediction layer instead (see Appendix (i) & (j)). Similar to RNN, we set the `hidden_dim` of $h_T^{(1)}$, $h_{T,\text{left}}^{(2)}$, and $h_{T,\text{right}}^{(2)}$ to be 256.

3.4 Results and Analysis

We present the results of all the models evaluated on the test set in Table 3 below, including the initial RNN. We denote CNN-1 and CNN-10 as the CNN models with `hidden_dim = 1` and `10` respectively.

Model	RNN	FFN-1	FFN-2	FFN-3	CNN-1	CNN-10	LSTM	Bi-LSTM
Accuracy (Loss)	69.61% (0.652)	**86.87% (0.322)**	86.25% (0.332)	86.44% (0.330)	76.32% (0.496)	84.80% (0.352)	83.97% (0.370)	85.47% (0.368)

Table 3: Accuracy (Loss) of different models evaluated on test set.

Surprisingly, the FFN architecture outperformed all the other models. Equally surprisingly, RNN, which usually performs well on sequential data, performed the worst, despite having roughly the same number of parameters

as FFN-1, at roughly 2.6M trainable parameters. Interestingly, with a similar value of `hidden_dim`, the LSTM significantly outperformed the RNN, despite only having slightly more trainable parameters of 2.9M. This suggests that the classification task relies on long-range dependency, afforded by LSTM but not RNN, for accurate predictions. One possible reason for this is the presence of stop words, which are noise that do not meaningfully impact the prediction. Intersperse them frequently in a sentence, the RNN would tend to lose 'memory' of the earlier semantically significant words, as the sentence gets longer and noisier, resulting in diminishing gradients. Therefore, these noisy inputs make it harder for the RNN to learn the sentiment pattern. On the same note, because of the gating mechanism of LSTM, the model could learn to ignore these noisy inputs and have more capacity to 'remember' the more important tokens (i.e., encoded in the hidden state) and make better prediction.

On the other hand, by applying FFN independently on each word followed by mean-pooling, we do not have a sequential recency bias like RNN does. The FFN likely attempts to further encode the word embeddings in a higher-dimensional space for our classification task. That is, semantically significant word embeddings would be encoded such that they contribute more to the predictions than the noisy word embeddings. Through this lens, we can see that a single hidden layer of dimensionality 500 is sufficient to encode the word embeddings for our downstream classification task, explaining the insignificant differences between FFN-1, FFN-2, and FFN-3. Furthermore, the mean-pooling allows all the encoding to be aggregated for the final classification task, despite not having any positional information. In this sense, FFN has access to global (average) sentiment information to make a more informed prediction.

Next, notice that CNN-1 significantly outperformed RNN, but still fell short of the FFNs. Unlike RNN, it does not suffer from sequential recency bias, and could choose the most 'important' n-gram at any part of the sentence for the classification task. However, this window-based method of CNN does not allow it to access any global information to extract sentiment pattern. This is further exacerbated by the presence of noisy inputs, which could interfere with the calculation of n-gram importance. Fortunately, the increase in `hidden_dim` would increase the number of 'important' n-gram, (for each n), each defined by different metrics (i.e., defined by the kernel weights). Although this does not guarantee that all global information could be accessed, more information across the sentence could be picked by the different filters, giving the model more features for the downstream classification task. This could explain the improvement in performance from CNN-1 to CNN-10.

Finally, there is an increase, albeit small, in model performance when we switch from LSTM to Bi-LSTM. This suggests that the classification task, hence the latent sentiment pattern, is largely agnostic of sentence direction or more generally the structure of the sentence. This could also suggest why the FFNs, which did not encode any positional information, performed so well in this classification task. It is worth noting, however, that FFNs tend to not perform well in other tasks that require positional information, such as language modelling. Furthermore, with additional pre-processing, such as word normalization and stop word removal, many of the models are likely to perform better, as the noise within the dataset are being suppressed.

4. Conclusion

In this project, we conducted several experiments with different optimizers and different architectures. Although some of the results are surprising, upon further analysis, they could be explained by the way the data was being handled. This sheds light on the importance of proper data handling and preparation, highlighting the importance of certain pre-processing steps that could potentially amplify the performance of many models.

5. Appendix: Code Snippets

(a) RNN with randomly initialized embedding.

```
import torch.nn as nn

class RNN(nn.Module):
    def __init__(self, input_dim, embedding_dim, hidden_dim, output_dim):

        super().__init__()

        self.embedding = nn.Embedding(input_dim, embedding_dim)

        self.rnn = nn.RNN(embedding_dim, hidden_dim)

        self.fc = nn.Linear(hidden_dim, output_dim)

    def forward(self, text, text_lengths):
        #text = [sent len, batch size]

        embedded = self.embedding(text)
        #embedded = [sent len, batch size, emb dim]

        output, hidden = self.rnn(embedded)
        #output = [sent len, batch size, hid dim]
        #hidden = [1, batch size, hid dim]


        #assert torch.equal(output[-1,:,:], hidden.squeeze(0))

        return self.fc(hidden.squeeze(0))
```

```
INPUT_DIM = len(TEXT.vocab)
EMBEDDING_DIM = 100
HIDDEN_DIM = 256
OUTPUT_DIM = 1

model = RNN(INPUT_DIM, EMBEDDING_DIM, HIDDEN_DIM, OUTPUT_DIM)
```

(b) Method 1 – Replacing OOV words with `'unk'`

```
#Match TEXT and W2V Embedding
#Non-matching word to unk

filtered_embedding = []
missed_word = 0
for i in range(len(TEXT.vocab)):
  text = TEXT.vocab.itos[i]

  if text in wv_model.key_to_index:
    text_embedding = wv_model[text]
    filtered_embedding.append(text_embedding)
  else:
    missed_word += 1
    text_embedding = wv_model['unk']
    filtered_embedding.append(text_embedding)
    continue

print(f"missed words = {missed_word}; found words = {len(filtered_embedding)}")


#Convert embedding to tensor
from torch.nn import Embedding
import numpy as np

embedding_weights = torch.FloatTensor(np.array(filtered_embedding))
#embedding = Embedding.from_pretrained(embedding_weights).to(device)
embedding = Embedding.from_pretrained(embedding_weights, freeze = False).to(device)
```

(c) Method 2 – Sample OOV embeddings from $\mathcal{N}(\boldsymbol{\mu}_{\mathrm{W2V}}, \boldsymbol{\sigma}^2_{\mathrm{W2V}})$

```
#Match TEXT and W2V Embedding
#Non-matching word to mean and std w2v embedding

import numpy as np
from torch.nn import Embedding

embeddings = np.array([wv_model[wv_model.index_to_key[i]] for i in range(len(wv_model))])
mean = np.mean(embeddings, axis=0)
std = np.std(embeddings, axis=0)

# Pre-process OOV embeddings
oov_embeddings = {}
for text in TEXT.vocab.itos:
    if text not in wv_model.key_to_index:
        oov_embeddings[text] = np.random.normal(loc=mean, scale=std, size=300)

filtered_embedding = np.empty((len(TEXT.vocab), 300))

for i in range(len(TEXT.vocab)):
    text = TEXT.vocab.itos[i]

    if text in wv_model.key_to_index:
        text_embedding = wv_model[text]
    else:
        text_embedding = oov_embeddings[text]

    filtered_embedding[i] = text_embedding

embedding_weights = torch.FloatTensor(filtered_embedding)
embedding = Embedding.from_pretrained(embedding_weights).to(device) #Frozen weight
#embedding = Embedding.from_pretrained(embedding_weights, freeze = False).to(device) #Unfrozen weight
```

(d) Method 3 – Sample OOV embeddings from $\mathcal{N}(\boldsymbol{\mu}_{\text{vocab}}, \boldsymbol{\sigma}^2_{\text{vocab}})$

```
#OOV words sampled from mean and std of vocab embedding
import numpy as np
from torch.nn import Embedding

available_embeddings = []
for i in range(len(TEXT.vocab)):
    text = TEXT.vocab.itos[i]

    if text in wv_model.key_to_index:
        available_embeddings.append(wv_model[text])

available_embeddings = np.array(available_embeddings)
mean = np.mean(available_embeddings, axis = 0)
std = np.std(available_embeddings, axis = 0 )

oov_embeddings = {}
for text in TEXT.vocab.itos:
    if text not in wv_model.key_to_index:
        oov_embeddings[text] = np.random.normal(loc=mean, scale=std, size=300)

filtered_embedding = np.empty((len(TEXT.vocab), 300))

for i in range(len(TEXT.vocab)):
    text = TEXT.vocab.itos[i]

    if text in wv_model.key_to_index:
        text_embedding = wv_model[text]
    else:
        text_embedding = oov_embeddings[text]

    filtered_embedding[i] = text_embedding

embedding_weights = torch.FloatTensor(filtered_embedding)
embedding = Embedding.from_pretrained(embedding_weights).to(device) #Frozen weight
```

(e) FFN-1: 1-layer Feedforward Network

```
#One-layer FFN: hidden dim = 500
import torch.nn as nn

class FFN1(nn.Module):
    def __init__(self, input_dim, embedding_dim, hidden_dim, output_dim):

        super(FFN1, self).__init__()

        self.embedding = nn.Embedding(input_dim, embedding_dim)

        self.output_dim = output_dim

        self.fc1 = nn.Linear(embedding_dim, hidden_dim)

        self.output = nn.Linear(hidden_dim, output_dim)

    def forward(self, text, text_lengths):
        #text = [sent len, batch size]

        embedded = self.embedding(text)
        #embedded = [sent len, batch size, emb dim]

        x = self.fc1(embedded)

        x = nn.functional.relu(x) #ReLU activation

        x = torch.mean(x, dim=0)

        output = self.output(x)

        #assert torch.equal(output[-1,:,:], x.squeeze(0))

        return output.squeeze()
```

(f) FFN-2: Two-layer Feedforward Network

```python
#Two-layer FFN: hidden dim = 500, 300
import torch.nn as nn

class FFN2(nn.Module):
    def __init__(self, input_dim, embedding_dim, hidden_dim1, hidden_dim2, output_dim):

        super(FFN2, self).__init__()

        self.embedding = nn.Embedding(input_dim, embedding_dim)

        self.output_dim = output_dim

        self.fc1 = nn.Linear(embedding_dim, hidden_dim1)

        self.fc2 = nn.Linear(hidden_dim1, hidden_dim2)

        self.output = nn.Linear(hidden_dim2, output_dim)

    def forward(self, text, text_lengths):
        #text = [sent len, batch size]

        embedded = self.embedding(text)
        #embedded = [sent len, batch size, emb dim]

        x = self.fc1(embedded)
        x = nn.functional.relu(x) #ReLU activation

        x = self.fc2(x)
        x = nn.functional.relu(x)

        x = torch.mean(x, dim=0)


        output = self.output(x)

        #assert torch.equal(output[-1,:,:], x.squeeze(0))

        return output.squeeze()
```

(g) FFN-3: Three-layer Feedforward Network

```python
#Three-layer FFN: hidden dim = 500, 300, 200
import torch.nn as nn

class FFN3(nn.Module):
    def __init__(self, input_dim, embedding_dim, hidden_dim1, hidden_dim2, hidden_dim3, output_dim):

        super(FFN3, self).__init__()

        self.embedding = nn.Embedding(input_dim, embedding_dim)

        self.output_dim = output_dim

        self.fc1 = nn.Linear(embedding_dim, hidden_dim1)

        self.fc2 = nn.Linear(hidden_dim1, hidden_dim2)

        self.fc3 = nn.Linear(hidden_dim2, hidden_dim3)

        self.output = nn.Linear(hidden_dim3, output_dim)

    def forward(self, text, text_lengths):
        #text = [sent len, batch size]

        embedded = self.embedding(text)
        #embedded = [sent len, batch size, emb dim]

        x = self.fc1(embedded)
        x = nn.functional.relu(x) #ReLU activation

        x = self.fc2(x)
        x = nn.functional.relu(x)

        x = self.fc3(x)
        x = nn.functional.relu(x)

        x = torch.mean(x, dim=0)

        output = self.output(x)

        #assert torch.equal(output[-1,:,:], x.squeeze(0))

        return output.squeeze()
```

(h) CNN: Convolutional Neural Network (`hidden_dim` is a hyperparameter)

```
#CNN using Conv2d
import torch.nn as nn

class CNN(nn.Module):
  def __init__(self, input_dim, embedding_dim, hidden_dim, output_dim):
    super(CNN, self).__init__()
    self.embedding = nn.Embedding(input_dim, embedding_dim)
    self.conv1 = nn.Conv2d(1, hidden_dim, kernel_size=(1, embedding_dim))
    self.conv2 = nn.Conv2d(1, hidden_dim, kernel_size=(2, embedding_dim))
    self.conv3 = nn.Conv2d(1, hidden_dim, kernel_size=(3, embedding_dim))
    self.relu = nn.ReLU()
    self.pool = nn.AdaptiveMaxPool2d(1)
    self.output = nn.Linear(hidden_dim * 3, output_dim)


  def forward(self, text, text_lengths):

    embedded = self.embedding(text)
    embedded = embedded.permute(1,0,2)
    embedded = embedded.unsqueeze(1)  # Add a channel dimension (required for Conv2d)

    conv1 = self.relu(self.conv1(embedded))
    conv2 = self.relu(self.conv2(embedded))
    conv3 = self.relu(self.conv3(embedded))


    pooled1 = self.pool(conv1)
    pooled2 = self.pool(conv2)
    pooled3 = self.pool(conv3)


    features = torch.cat([pooled1, pooled2, pooled3], dim=1).squeeze()
    output = self.output(features)

    return output.squeeze()
```

(i) LSTM: Long Short-Term Memory Network

```
#LSTM
import torch.nn as nn
class LSTM(nn.Module):
    def __init__(self, input_dim, embedding_dim, hidden_dim, output_dim):
        super(LSTM, self).__init__()
        self.embedding = nn.Embedding(input_dim, embedding_dim)
        self.lstm = nn.LSTM(embedding_dim, hidden_dim)
        self.output = nn.Linear(hidden_dim, output_dim)


    def forward(self, text, text_lengths):
        embedded = self.embedding(text)
        _, (hidden,cell) = self.lstm(embedded) #no need for activation because of gating mechanism
        output = self.output(hidden)


        return output.squeeze()
```

(j) BiLSTM: Bi-directional LSTM

```
#BILSTM
import torch.nn as nn
class BiLSTM(nn.Module):
    def __init__(self, input_dim, embedding_dim, hidden_dim, output_dim):
        super(BiLSTM, self).__init__()
        self.embedding = nn.Embedding(input_dim, embedding_dim)
        self.bilstm = nn.LSTM(embedding_dim, hidden_dim, bidirectional= True)
        self.output = nn.Linear(hidden_dim*2, output_dim)


    def forward(self, text, text_lengths):
        embedded = self.embedding(text)
        _, (hidden,cell) = self.bilstm(embedded) #no need for activation because of gating mechanism
        hidden = torch.concat((hidden[0,:,:], hidden[1,:,:]), dim = 1)       #2 hidden states, concat them together along batch size
        #print('hidden:' , hidden.shape)

        output = self.output(hidden)
        #print('output: ', output.shape)


        return output.squeeze()
```