

DL Assignment4

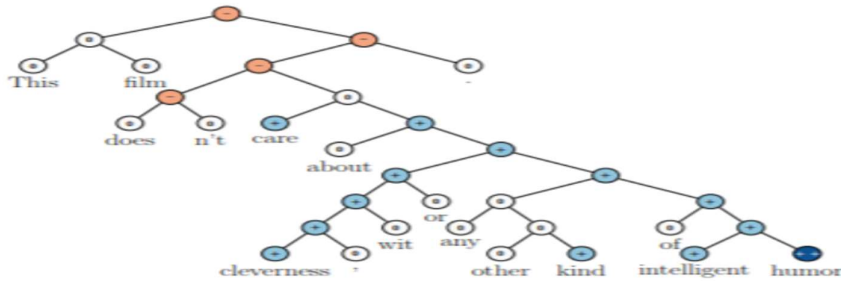
Akshara B

211AI012

Dataset:

The NLP task I chose is sentiment analysis. The dataset I used is Stanford Sentiment Treebank – 5 (SST5). This is a dataset used for sentiment analysis. The SST-5 dataset consists of 11,855 single sentences extracted from movie reviews. It includes a total of 215,154 unique phrases from parse trees. Each phrase is labeled as either negative (0), somewhat negative(1), neutral(2), somewhat positive(3), or positive(4). The highest accuracy achieved on this dataset till date was around 69%. So the models created by me are close in terms of benchmark

The dataset structure:



This dataset is used for both tasks 1 and task 2

Link: <https://huggingface.co/datasets/gimmaru/SetFit-sst5>

<https://www.kaggle.com/datasets/haoshaoyang/sst5-data>

kaggle colab notebook:

code can be found here –

<https://www.kaggle.com/code/akshara211ai012/notebook8cad284ef2>

Task 1: creating RNN, Ni-RNN, LSTM, Bi-LSTM models and usinf them for NLP task of fine-grained dentiment classification.

Base code (same for all architectures)

Initially we divided our dataset into 80% train 20% test

▷

```
flatten = lambda l: [item for sublist in l for item in sublist]
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

HIDDEN_SIZE = 50 #30
ROOT_ONLY = False
BATCH_SIZE = 80
EPOCH = 5
LR = 0.001
print(device)
```

cuda

First, we install all the necessary libraries and define hypeparameters. I am fixing epochs as 5 batch size as and hidden layer size as 50. Batch size as 80 and learning rate as 0.001

Next we define a function `get_batch` that shuffles the training data and loads `train_data` in the size of `BATCH_SIZE` parameter.

```
def getBatch(batch_size, train_data):
    random.shuffle(train_data)
    sindex = 0
    eindex = batch_size
    while eindex < len(train_data):
        batch = train_data[sindex: eindex]
        temp = eindex
        eindex = eindex + batch_size
        sindex = temp
        yield batch

    if eindex >= len(train_data):
        batch = train_data[sindex:]
        yield batch
```

PARSING

Then we create node and tree classes to represent our data in a tree form. Node` and `Tree`, which together facilitate the creation and manipulation of tree structures.

```
In [4]: class Node: # a node in the tree
    def __init__(self, label, word=None):
        self.label = label
        self.word = word
        self.parent = None # reference to parent
        self.left = None # reference to left child
        self.right = None # reference to right child
        # true if I am a leaf (could have probably derived this from if I have
        # a word)
        self.isLeaf = False
        # true if we have finished performing forwardprop on this node (note,
        # there are many ways to implement the recursion.. some might not
        # require this flag)

    def __str__(self):
        if self.isLeaf:
            return '{0}:{1}'.format(self.word, self.label)
        return '({0} <- [{1}:{2}] -> {3})'.format(self.left, self.word, self.label, self.right)
```

```

def __str__(self):
    if self.isLeaf:
        return '{0}:{1}'.format(self.word, self.label)
    return '{0} <- [{1}:{2}] -> {3}'.format(self.left, self.word, self.label, self.right)

class Tree:

    def __init__(self, treeString, openChar='(', closeChar=')'):
        tokens = []
        self.open = '('
        self.close = ')'
        for toks in treeString.strip().split():
            tokens += list(toks)
        self.root = self.parse(tokens)
        # get list of labels as obtained through a post-order traversal
        self.labels = get_labels(self.root)
        self.num_words = len(self.labels)

    def parse(self, tokens, parent=None):
        assert tokens[0] == self.open, "Malformed tree"
        assert tokens[-1] == self.close, "Malformed tree"

```

```

def parse(self, tokens, parent=None):
    assert tokens[0] == self.open, "Malformed tree"
    assert tokens[-1] == self.close, "Malformed tree"

    split = 2 # position after open and label
    countOpen = countClose = 0

    if tokens[split] == self.open:
        countOpen += 1
        split += 1
    # Find where left child and right child split
    while countOpen != countClose:
        if tokens[split] == self.open:
            countOpen += 1
        if tokens[split] == self.close:
            countClose += 1
        split += 1

    # New node
    node = Node(int(tokens[1])) # zero index labels

    node.parent = parent

```

```

        split += 1

    # New node
    node = Node(int(tokens[1])) # zero index labels

    node.parent = parent

    # leaf Node
    if countOpen == 0:
        node.word = ''.join(tokens[2: -1]).lower() # lower case?
        node.isLeaf = True
        return node

    node.left = self.parse(tokens[2: split], parent=node)
    node.right = self.parse(tokens[split: -1], parent=node)

    return node

def get_words(self):
    leaves = getLeaves(self.root)
    words = [node.word for node in leaves]
    return words

```

1. Node Class:

- The `Node` class represents a single node in a tree. It contains the following attributes:


- `label`: The label associated with the node.
- `word`: The word associated with the node (only applicable to leaf nodes).
- `parent`: A reference to the parent node.
- `left`: A reference to the left child node.
- `right`: A reference to the right child node.
- `isLeaf`: A boolean flag indicating whether the node is a leaf node or not.

- The `__str__` method provides a string representation of the node. If the node is a leaf, it returns a string containing the word and label enclosed in square brackets. Otherwise, it returns a string representing the node and its children in a tree-like format.

2. Tree Class:

- The `Tree` class represents a tree structure composed of nodes. It has the following attributes:
 - `root`: A reference to the root node of the tree.
 - `labels`: A list containing the labels of all nodes in the tree, obtained through a post-order traversal.
 - `num_words`: The total number of words in the tree.
- The `__init__` method initializes a tree object from a string representation of the tree. It parses the string and constructs the tree recursively using the `parse` method.
- The `parse` method recursively builds the tree structure from a list of tokens representing the tree string. It handles cases where the tree is malformed or contains nested nodes.
- The `get_words` method returns a list of words present in the leaf nodes of the tree.

After this we define three important functions.



```
def get_labels(node):
    if node is None:
        return []
    return get_labels(node.left) + get_labels(node.right) + [node.label]

def getLeaves(node):
    if node is None:
        return []
    if node.isLeaf:
        return [node]
    else:
        return getLeaves(node.left) + getLeaves(node.right)

def loadTrees(data='train'):
    file = '/content/data/data/%s.txt' % data
    print("Loading %s trees.." % data)
    with open(file, 'r', encoding='utf-8') as fid:
        trees = [Tree(l) for l in fid.readlines()]

    return trees
```

✓ 0s completed at 4:27 PM

1. get_labels(node):

- This function takes a `node` as input and recursively traverses the tree to collect the labels of all nodes. If the input `node` is `None`, indicating an empty

subtree, it returns an empty list. Otherwise, it recursively calls itself on the left and right child nodes of the current node (``node.left`` and ``node.right``) and concatenates the lists of labels obtained from the left and right subtrees with the label of the current node (``node.label``) Finally, it returns the list of labels.

2. `getLeaves(node)`:

- This function takes a ``node`` as input and recursively traverses the tree to collect all leaf nodes. If the input ``node`` is ``None``, indicating an empty subtree, it returns an empty list. If the input ``node`` is a leaf node (i.e., ``node.isLeaf`` is ``True``), it returns a list containing only that leaf node. Otherwise, it recursively calls itself on the left and right child nodes of the current node (``node.left`` and ``node.right``) and concatenates the lists of leaf nodes obtained from the left and right subtrees. Finally, it returns the list of leaf nodes.

3. `loadTrees(data='train')`:

- This function loads tree data from a file specified by the ``data`` parameter. It assumes that the tree data is stored in a text file where each line represents a tree in a specific format. It opens the file in read mode and reads each line using a ``fid`` file object. Each line, it creates a ``Tree`` object by passing the line to the ``Tree`` constructor, resulting in a list of ``Tree`` objects representing the trees in the file. Finally, it returns the list of ``Tree`` objects representing the loaded trees.

After this we define our model class and architecture depending on subproblem.

After defining model. We create train and test functions.

```

import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix
import seaborn as sns

def train(model, lr, epochs):
    RESCHEDULED = False
    loss_function = nn.CrossEntropyLoss()
    optimizer = optim.Adam(model.parameters(), lr=lr)

    # Initialize lists to store loss and accuracy values
    loss_values = []
    accuracy_values = []
    epoch_losses = []

    for epoch in range(epochs):
        losses = []

        if RESCHEDULED == False and epoch == epochs // 2:
            lr *= 0.1
            optimizer = optim.Adam(model.parameters(), lr=lr, weight_decay=1e-5)
            RESCHEDULED = True

        for i, batch in enumerate(getBatch(BATCH_SIZE, train_data)):
            model.zero_grad()
            preds = model(batch, ROOT_ONLY)

            selected_values = preds[:, :2]

            preds = selected_values.reshape(-1)

            if ROOT_ONLY:
                labels = [tree.labels[-1] for tree in batch]

```

```

def test(model):
    test_data = loadTrees('test')
    accuracy = 0
    num_node = 0

    for test in test_data:
        model.zero_grad()
        preds = model(test, ROOT_ONLY)
        labels = test.labels[-1:] if ROOT_ONLY else test.labels
        for pred, label in zip(preds.max(1)[1].data.tolist(), labels):
            num_node += 1
            if pred == label:
                accuracy += 1
    print('Test Acc: ', accuracy / num_node * 100)

```

For training we initialise the model and run it as `train(model,LR,EPOCH)` and for testing we run `test(model)`

1. RNN

The following is model architecture.


```

class RNN(nn.Module):

    def __init__(self, word2index, hidden_size, output_size):
        super(RNN, self).__init__()

        self.word2index = word2index
        self.embed = nn.Embedding(len(word2index), hidden_size)
        self.rnn = nn.RNN(hidden_size, hidden_size)
        self.fc = nn.Linear(hidden_size, output_size)

    def forward(self, input):
        embedded = self.embed(input)
        output, _ = self.rnn(embedded)
        output = self.fc(output[-1]) # taking the last output
        return F.log_softmax(output, dim=1)

```

The rest of the code remains same.

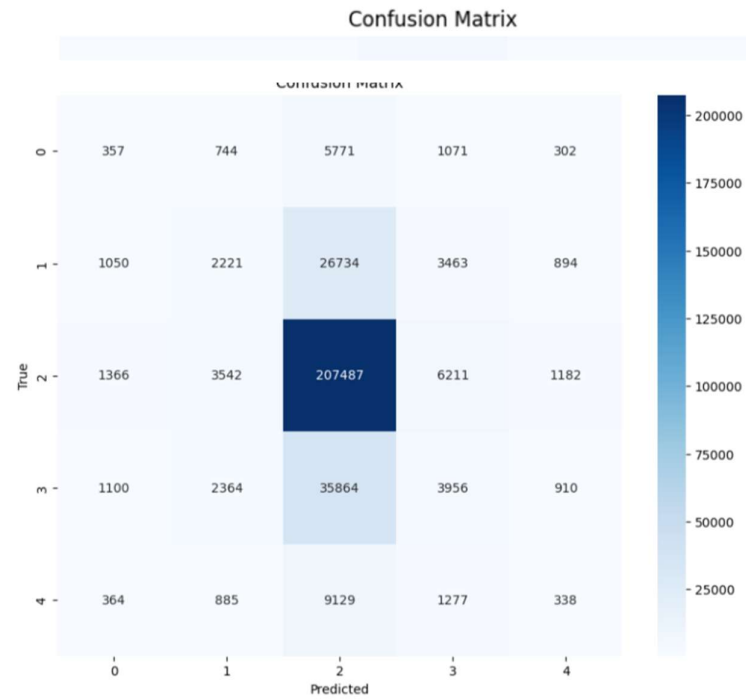
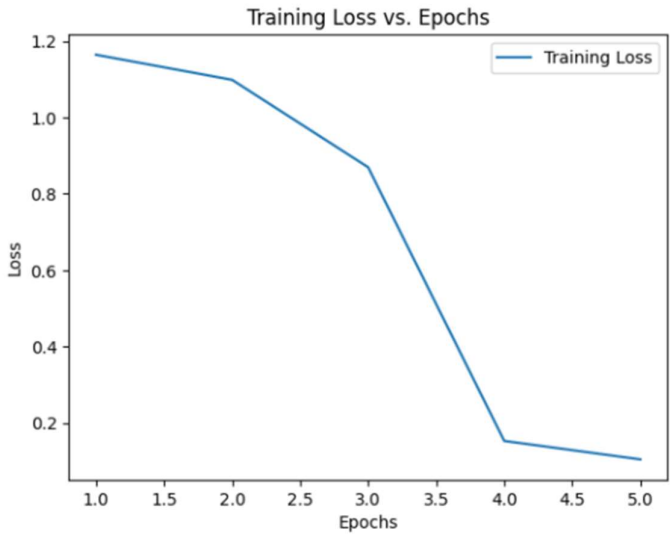
This model is a basic Recurrent Neural Network (RNN) designed for sequence-to-sequence learning tasks.

1. Initialization: It takes parameters `word2index`, `hidden_size`, and `output_size`. It initializes the embedding layer (`embed`) with an embedding matrix for word representation. It also initializes the RNN layer (`rnn`) with a hidden size specified by `hidden_size`. The RNN layer is a basic RNN cell that takes input of size `hidden_size` and produces output of size `hidden_size`. Finally, it initializes the fully connected layer (`fc`) which maps the output of the RNN to the output size specified by `output_size`.

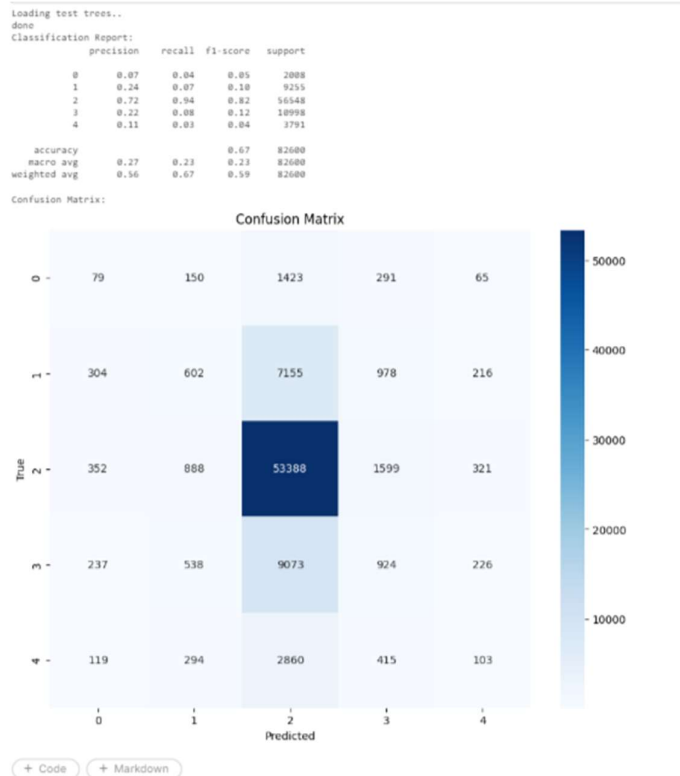
2. Forward Pass: The `forward` method defines the forward pass computation of the RNN model. It takes an input sequence (`input`) as input. First, it embeds the input sequence using the embedding layer (`embed`). Then, it passes the embedded sequence through the RNN layer (`rnn`). The RNN layer processes the sequence iteratively, updating its hidden state at each time step. The output of the RNN layer is a sequence of hidden states. In this implementation, only the hidden state corresponding to the last time step is used. This hidden state is passed through the fully connected layer (`fc`) to produce the final output. The output is then passed through a log softmax activation function along dimension 1 (`dim=1`) to obtain the output probabilities.

Results:

```
[1/5] mean_loss : 1.16
[2/5] mean_loss : 1.10
[3/5] mean_loss : 0.87
[4/5] mean_loss : 0.15
[5/5] mean_loss : 0.10
```



training accuracy 0.6728534568808031



The training accuracy is 0.67 for both train and test sets. It doesn't outperform recursive NN. It seems like Recursive NN is better suitable for this task rather than RNN. Class 2 has highest f1score of 0.82 but classes 1 and 3 have the least. There is high variation in distribution of f1scores among classes leading to low overall accuracy.

2. Bidirectional RNN

```
class biRNN(nn.Module):
    def __init__(self, word2index, hidden_size, output_size):
        super(biRNN, self).__init__()

        self.word2index = word2index
        self.embed = nn.Embedding(len(word2index), hidden_size)
        self.rnn = nn.RNN(hidden_size, hidden_size, bidirectional=True)
        self.fc = nn.Linear(hidden_size, output_size)

    def forward(self, input):
        embedded = self.embed(input)
        output, _ = self.rnn(embedded)
        output = self.fc(output[-1]) # taking the last output
        return F.log_softmax(output, dim=1)
```

rest of the code remains the same.

This model is a bidirectional Recurrent Neural Network (RNN) designed for sequence-to-sequence learning tasks.

1. Initialization: It takes parameters ``word2index``, ``hidden_size``, and ``output_size``. It initializes the embedding layer (``embed``) with an embedding matrix for word representation. The RNN layer (``rnn``) is initialized with a hidden size specified by ``hidden_size`` and set to be bidirectional (``bidirectional=True``). This means that the RNN cell processes input sequences in both forward and backward directions, allowing it to capture information from past and future contexts. The fully connected layer (``fc``) maps the output of the RNN to the output size specified by ``output_size``.

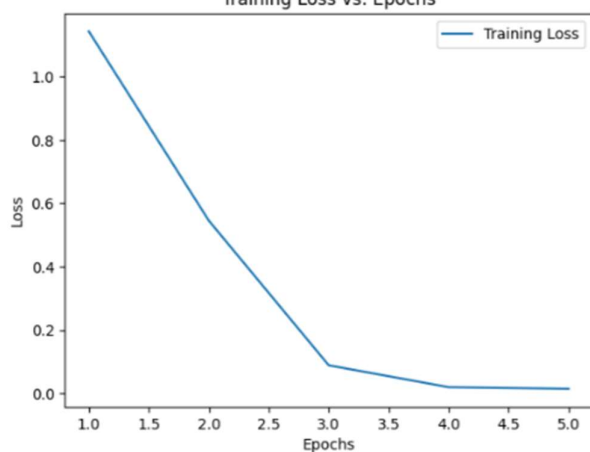
2. Forward Pass: The ``forward`` method defines the forward pass computation of the bidirectional RNN model. It takes an input sequence (``input``) as input. First, it embeds the input sequence using the embedding layer (``embed``). Then, it passes the embedded sequence through the bidirectional RNN layer (``rnn``). The RNN layer processes the sequence iteratively in both forward and backward directions, updating its hidden states at each time step. The output of the RNN layer is a sequence of hidden states for each direction. In this implementation, only the hidden states corresponding to the last time step are used. These hidden states are concatenated and passed through the fully connected layer (``fc``) to produce the final output. The output is then passed through a log softmax activation function along dimension 1 (``dim=1``) to obtain the output probabilities.

this bidirectional RNN model captures bidirectional dependencies in input sequences by processing them in both forward and backward directions. It enhances the model's ability to capture long-range dependencies and improves performance in tasks such as sequence labeling and sequence classification.

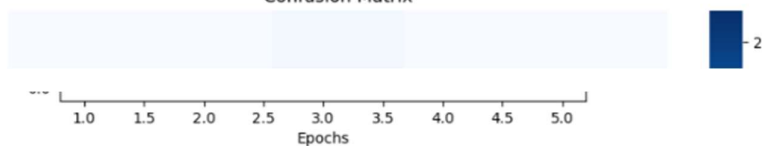
Results

```
[1/5] mean_loss : 1.14
[2/5] mean_loss : 0.55
[3/5] mean_loss : 0.09
[4/5] mean_loss : 0.02
[5/5] mean_loss : 0.01
```

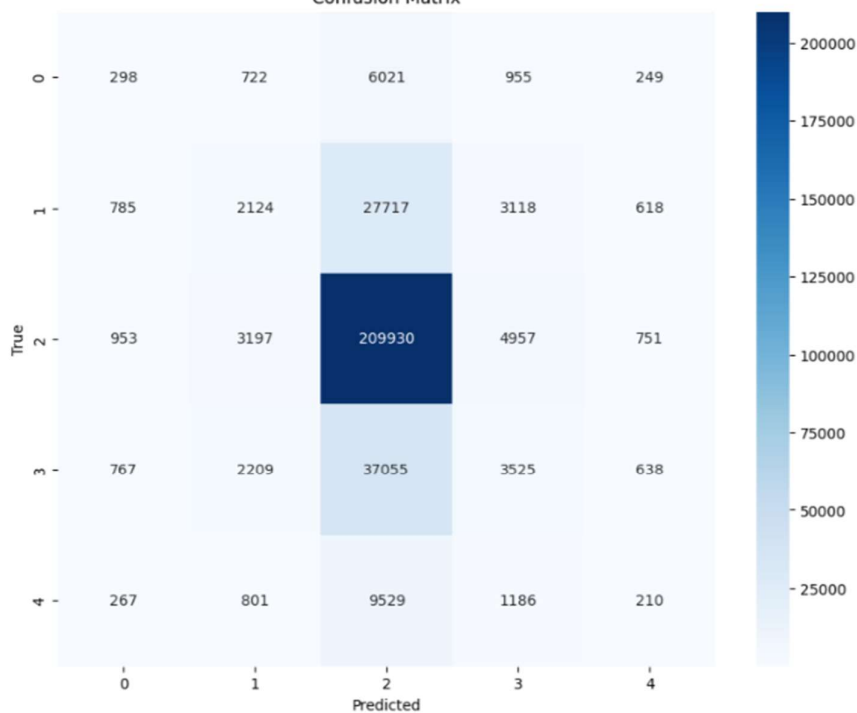
Training Loss vs. Epochs



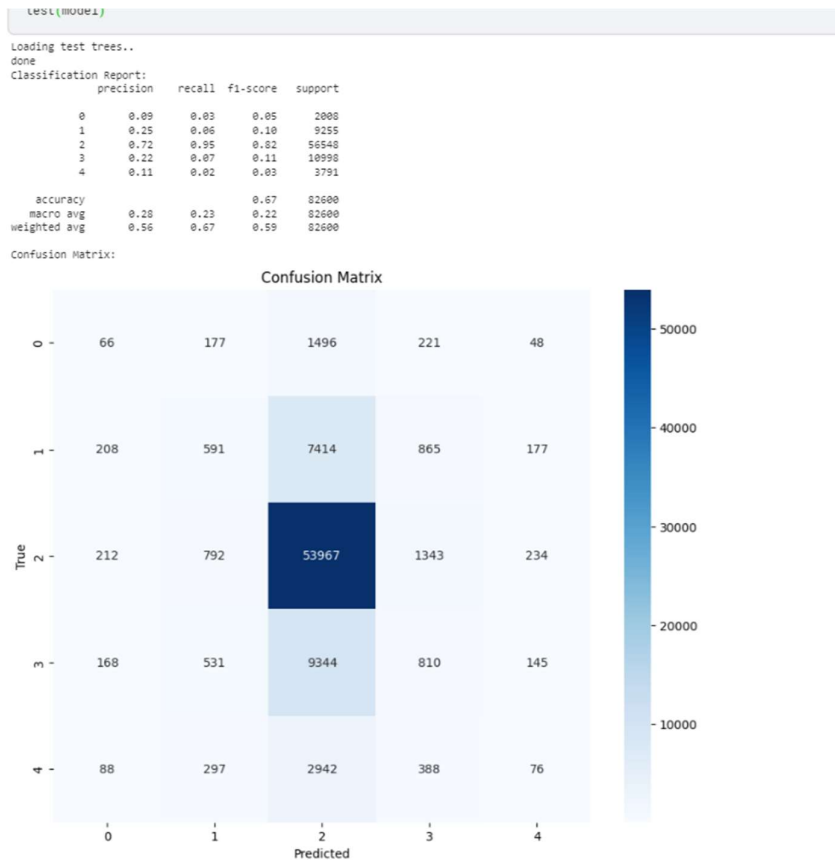
Confusion Matrix



Confusion Matrix



training accuracy 0.6782774921370323



As we can see the test and train accuracy were both 0.67. the Bidirectional RNN was no different than RNN in terms of performance. Both performed just alright. Maybe a basic RNN model isn't sufficient to achieve better accuracies for this dataset. Hybrid architectures should be used. F1scores of classes 1 and 3 are underperforming

3. LSTM

This is the model. Rest of the code remains same as explained in task2.

```

index2word = {v: k for k, v in word2index.items()}
class LSTM(nn.Module):
    def __init__(self, word2index, hidden_size, output_size):
        super(LSTM, self).__init__()

        self.word2index = word2index
        self.embed = nn.Embedding(len(word2index), hidden_size)
        self.lstm = nn.LSTM(hidden_size, hidden_size)
        self.fc = nn.Linear(hidden_size, output_size)

    def forward(self, input):
        embedded = self.embed(input)
        output, _ = self.lstm(embedded)
        output = self.fc(output[-1]) # taking the last output
        return F.log_softmax(output, dim=1)

```

This model is an implementation of a Long Short-Term Memory (LSTM) network for sequence-to-sequence learning tasks.

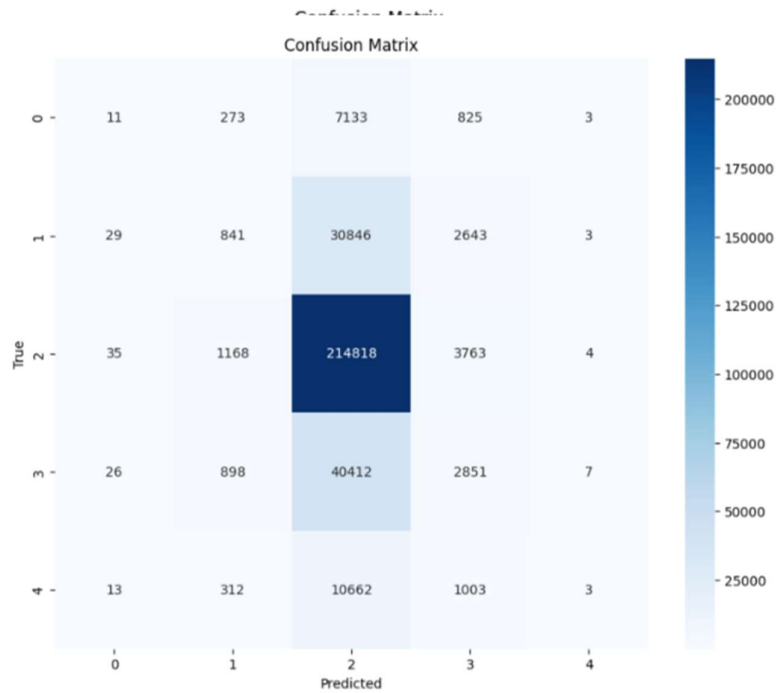
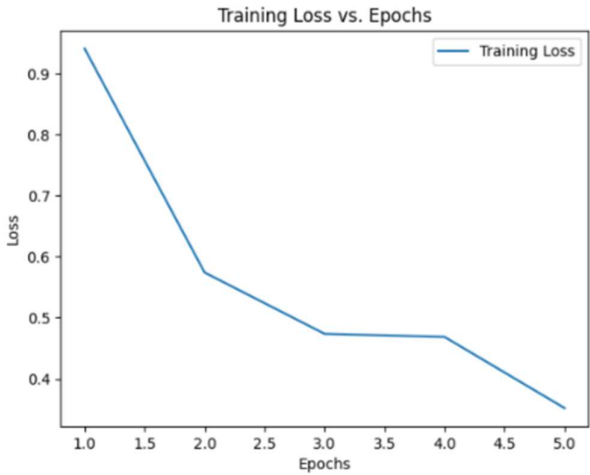
1. Initialization: In the constructor `__init__`, the `LSTM` class initializes the model. It takes parameters `word2index`, `hidden_size`, and `output_size`. It initializes the embedding layer (`embed`) with an embedding matrix for word representation. The LSTM layer (`lstm`) is initialized with a hidden size specified by `hidden_size`. The fully connected layer (`fc`) maps the output of the LSTM to the output size specified by `output_size`.

2. Forward Pass: The `forward` method defines the forward pass computation of the LSTM model. It takes an input sequence (`input`) as input. First, it embeds the input sequence using the embedding layer (`embed`). Then, it passes the embedded sequence through the LSTM layer (`lstm`). The LSTM layer processes the sequence iteratively, updating its hidden states and cell states at each time step. The output of the LSTM layer is a sequence of hidden states for each time step. In this implementation, only the hidden state corresponding to the last time step is used. This hidden state is passed through the fully connected layer (`fc`) to produce the final output. The output is then passed through a log softmax activation function along dimension 1 (`dim=1`) to obtain the output probabilities.

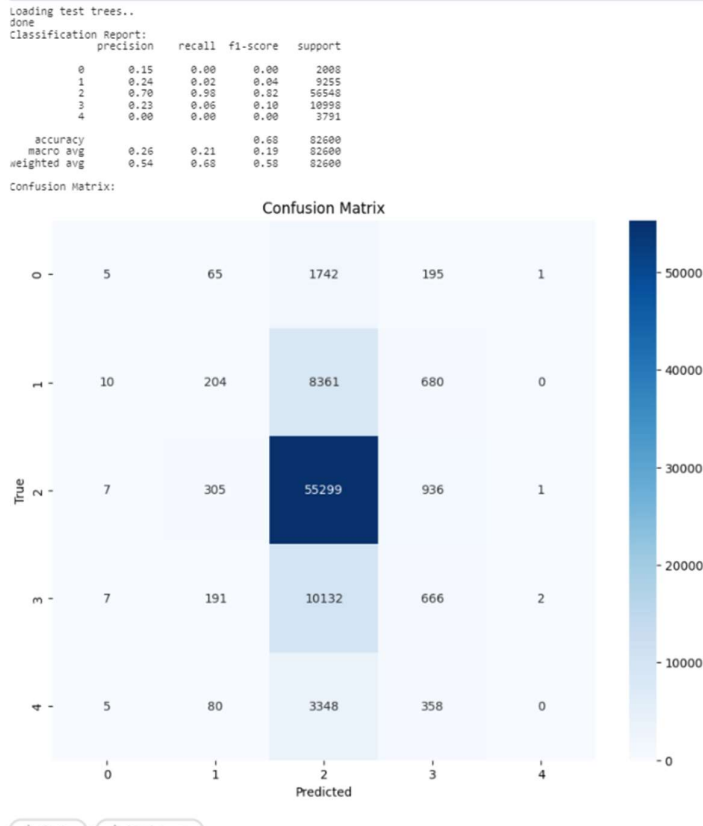
this LSTM model is capable of capturing long-range dependencies and handling vanishing/exploding gradient problems better than simple RNNs. It is known to perform better than RNN in almost every NLP task.

Results

[1/5] mean_loss : 0.94
[2/5] mean_loss : 0.57
[3/5] mean_loss : 0.47
[4/5] mean_loss : 0.47
[5/5] mean_loss : 0.35



training accuracy 0.6859270140811471



The train accuracy is 0.685 and test accuracy is 0.68. this is close to recursive NN (0.687). This model performs better than RNN as expected. Class2 has the highest f1score of 0.81

4. Bidirectional LSTM

The model is given below. The rest of the code is same as task 1

```

class biLSTM(nn.Module):
    def __init__(self, word2index, hidden_size, output_size):
        super(biLSTM, self).__init__()

        self.word2index = word2index
        self.embed = nn.Embedding(len(word2index), hidden_size)
        self.lstm = nn.LSTM(hidden_size, hidden_size, bidirectional=True)
        self.fc = nn.Linear(hidden_size * 2, output_size)

    def forward(self, input):
        embedded = self.embed(input)
        output, _ = self.lstm(embedded)
        output = self.fc(output[-1]) # taking the last output
        return F.log_softmax(output, dim=1)

def train(model, lr, epochs):

```

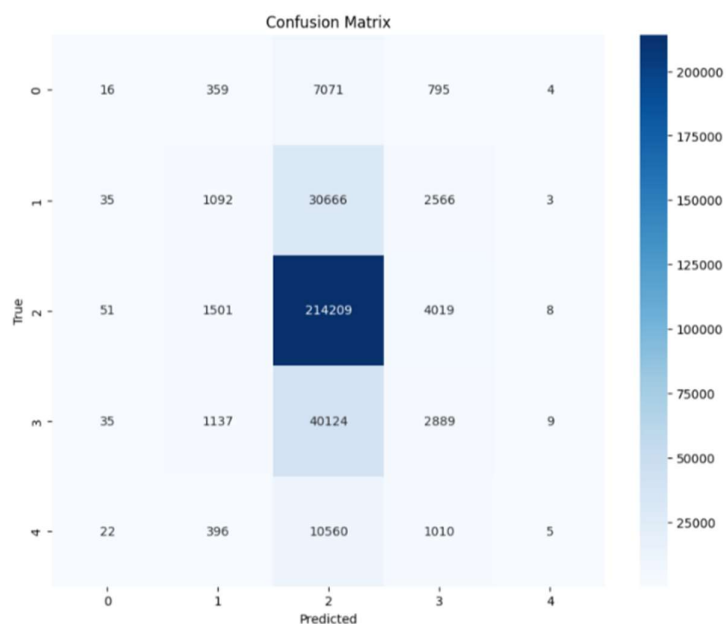
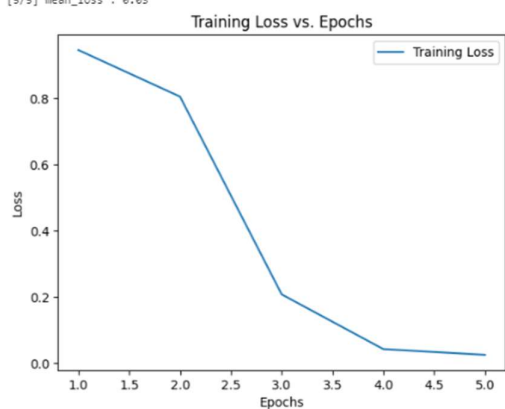
1. Initialization: to other neural network models, it takes parameters such as `word2index`, `hidden_size`, and `output_size`. It initializes an embedding layer (`embed`) to represent words in a continuous vector space. The bidirectional LSTM layer (`lstm`) is initialized with a hidden size specified by `hidden_size`. The bidirectional LSTM processes the input sequence in both forward and backward directions, providing richer context representation. The fully connected layer (`fc`) maps the concatenated output of both directions of the LSTM to the output size specified by `output_size`.

2. Forward Pass: The `forward` method defines the forward pass computation of the biLSTM model. It takes an input sequence (`input`) as input. First, it embeds the input sequence using the embedding layer (`embed`). Then, it passes the embedded sequence through the bidirectional LSTM layer (`lstm`). The bidirectional LSTM processes the sequence in both forward and backward directions, capturing both past and future contexts for each time step. The output of the LSTM layer is a sequence of hidden states for each time step, concatenated from both directions. In this implementation, only the hidden state corresponding to the last time step is used. This concatenated hidden state is passed through the fully connected layer (`fc`) to produce the final output. The output is then passed through a log softmax activation function along dimension 1 (`dim=1`) to obtain the output probabilities.

this biLSTM model is capable of capturing bidirectional dependencies in the input sequence. This is more suitable for tasks where context information from both past and future is crucial, like our task of sentiment analysis

Results:

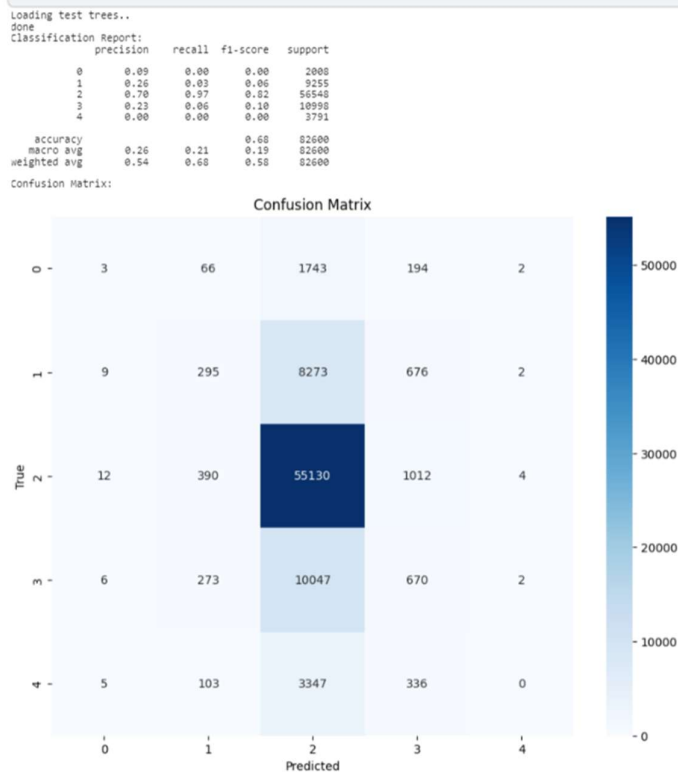
```
[1/5] mean_loss : 0.95
[2/5] mean_loss : 0.81
[3/5] mean_loss : 0.21
[4/5] mean_loss : 0.04
[5/5] mean_loss : 0.03
```



training accuracy 0.6849445354728139

[+ Code](#)

[+ Markdown](#)



This model gave accuracy of 0.68 on train set and 0.69 on test set. This model performed the best for our task and outperformed even RecNN. Still it suffers the shortcoming of other classes being misclassified as class 2. Hence low precision, recall and f1 score. Classes 0 and 4 have the least f1scores.

TASK2: Creating a Bidirectional recursive NN and recursive NN architecture from scratch.

1. Bi-recNN

Next Is the important part where we define Bi-RecNN architecture.

```

class BiRNTN(nn.Module):
    def __init__(self, word2index, hidden_size, output_size):
        super(BiRNTN, self).__init__()

        self.word2index = word2index
        self.embed = nn.Embedding(len(word2index), hidden_size)
        self.V = nn.ParameterList(
            [nn.Parameter(torch.randn(hidden_size * 2, hidden_size * 2)) for _ in range(hidden_size)])
        self.W = nn.Parameter(torch.randn(hidden_size * 2, hidden_size))
        self.b = nn.Parameter(torch.randn(1, hidden_size))
        self.W_out = nn.Linear(hidden_size * 2, output_size)
        self.hidden_size=hidden_size

    def init_weight(self):
        nn.init.xavier_uniform_(self.embed.state_dict()['weight'])
        nn.init.xavier_uniform_(self.W_out.state_dict()['weight'])
        for param in self.V.parameters():
            nn.init.xavier_uniform_(param)
        nn.init.xavier_uniform_(self.W)
        self.b.data.fill_(0)

    def tree_propagation(self, node):
        recursive_tensor = OrderedDict()
        current = None
        if node.isLeaf:
            tensor = Variable(torch.tensor([self.word2index[node.word]], dtype=torch.long, device=device)) \
                if node.word in self.word2index.keys() \
                else Variable(torch.tensor([self.word2index['<UNK>']], dtype=torch.long, device=device))
            current = self.embed(tensor)
        else:
            recursive_tensor.update(self.tree_propagation(node.left))
            recursive_tensor.update(self.tree_propagation(node.right))

```

```

        else:
            recursive_tensor.update(self.tree_propagation(node.left))
            recursive_tensor.update(self.tree_propagation(node.right))

            concatenated = torch.cat([recursive_tensor[node.left], recursive_tensor[node.right]], 1)
            xVx = []
            for i, v in enumerate(self.V):
                xVx.append(torch.matmul(torch.matmul(concatened, v), concatenated.transpose(0, 1)))

            xVx = torch.cat(xVx, 1)
            Wx = torch.matmul(concatened, self.W)

            current = F.tanh(xVx + Wx + self.b)
            recursive_tensor[node] = current
            return recursive_tensor

    def forward(self, Trees, root_only=False):
        propagated = []
        if not isinstance(Trees, list):
            Trees = [Trees]

        for Tree in Trees:
            recursive_tensor = self.tree_propagation(Tree.root)
            if root_only:
                recursive_tensor = recursive_tensor[Tree.root]
                propagated.append(recursive_tensor)
            else:
                recursive_tensor = [tensor for node, tensor in recursive_tensor.items()]
                propagated.extend(recursive_tensor)

        propagated = torch.cat(propagated)

```

```
propagated = torch.cat(propagated)

# Reshape propagated tensor to match the expected input size of W_out
propagated = propagated.view(-1, self.hidden_size * 2)

return F.log_softmax(self.W_out(propagated), 1)
```

Summary:

1. Initialization and Parameters:

- The `BiRNTN` class inherits from `nn.Module`, and is initialised with these parameters.
 - `word2index`: A dictionary mapping words to their corresponding indices in the vocabulary.
 - `hidden_size`: The dimensionality of the hidden states in the network.
 - `output_size`: The dimensionality of the output.
- The parameters include:
 - `embed`: An embedding layer to convert word indices into dense vectors of fixed size (`hidden_size`).
 - `V`: A list of tensor parameters used for interaction between child nodes during tree propagation.
 - `W`: A tensor parameter for the linear transformation applied to the concatenated child node representations.
 - `b`: A tensor parameter for the bias term.
 - `W_out`: A linear layer to produce the final output.

2. Weight Initialization:

- The `init_weight` method initializes the weights of the embedding layer, output layer, and all tensor parameters using Xavier initialization. This ensures that the initial weights are suitable for training the network effectively.

3. Tree Propagation:

- The `tree_propagation`` method recursively propagates information through the tree structure.
- It takes a `node`` as input and returns a dictionary (`recursive_tensor``) containing the representations of all nodes in the subtree rooted at the input `node``.
- If the input `node`` is a leaf node, its word representation is obtained from the embedding layer. If the word is not in the vocabulary, a default representation (`<UNK>``) is used.
- If the input `node`` is an internal node, the method recursively computes representations for its left and right child nodes and concatenates them.
- The concatenated representation is then transformed using tensor parameters `V`` and `W``, along with a bias term `b``, and passed through a hyperbolic tangent (tanh) activation function to compute the current node's representation.

4. Forward Pass:

- The `forward`` method takes a single tree or a list of trees (`Trees``) as input.
- It calls the `tree_propagation`` method to compute representations for all nodes in each tree.
- Depending on the value of the `root_only`` parameter, it either selects only the root node representations or includes representations for all nodes in the output.
- The concatenated representations are reshaped to match the expected input size of the output layer (`W_out``), and the softmax activation function is applied to produce the final output probabilities for classification.

5. Bidirectional Aspect:

- The bidirectionality in this architecture comes from the recursive computation of node representations. When computing the representation of

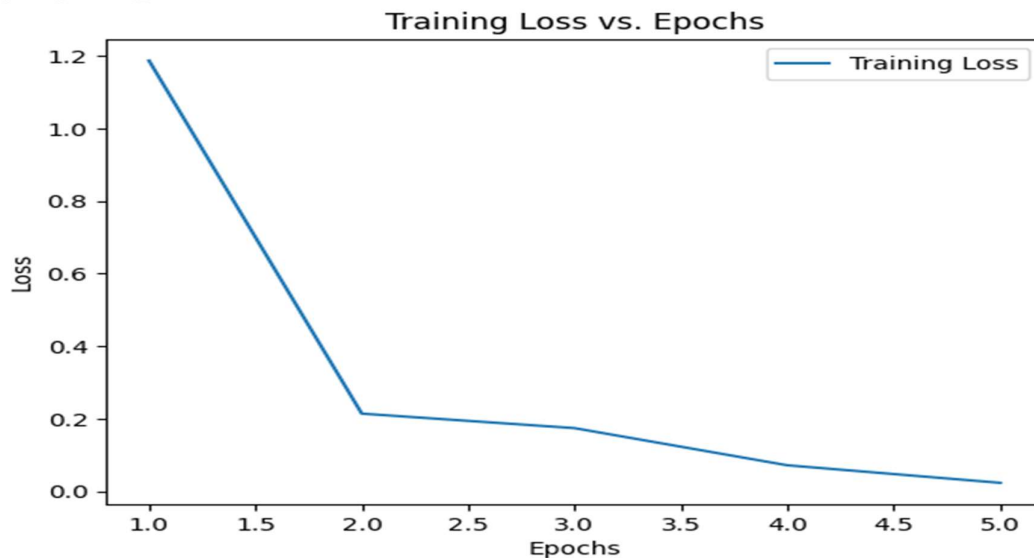
an internal node, information from both its left and right child nodes is combined.

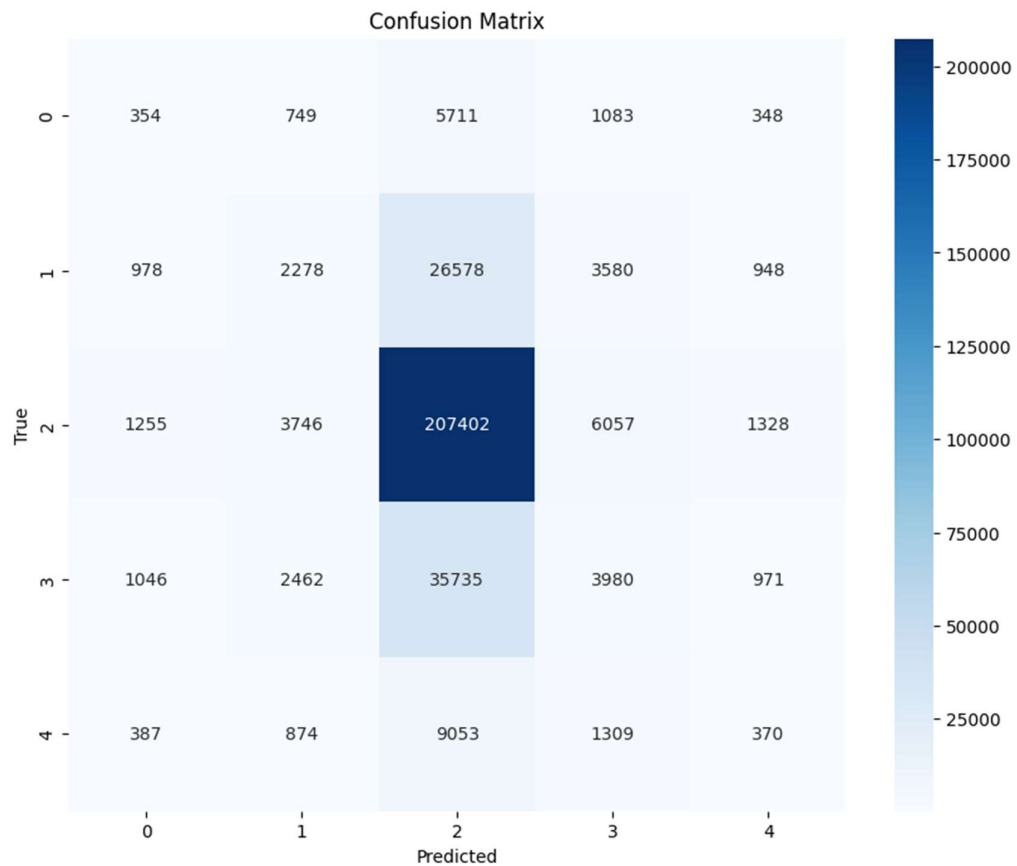
- By recursively propagating information from leaf nodes up to the root, the network captures bidirectional dependencies in the tree structure. This allows the network to consider both left-to-right and right-to-left information flow during tree traversal.

Now we train our models. While training we get the epochs vs loss curve and training accuracy. While testing we get the classification report and confusion matrix.

Results:

```
[1/5] mean_loss : 1.19  
[2/5] mean_loss : 0.21  
[3/5] mean_loss : 0.17  
[4/5] mean_loss : 0.07  
[5/5] mean_loss : 0.02
```





This is the confusion matrix for training data. The training accuracy is 0.672

training accuracy 0.672931929613098

+ Code + Markdown

[32]:

```
test(model)
```

For the test set

```
test(model)
```

Loading test trees..

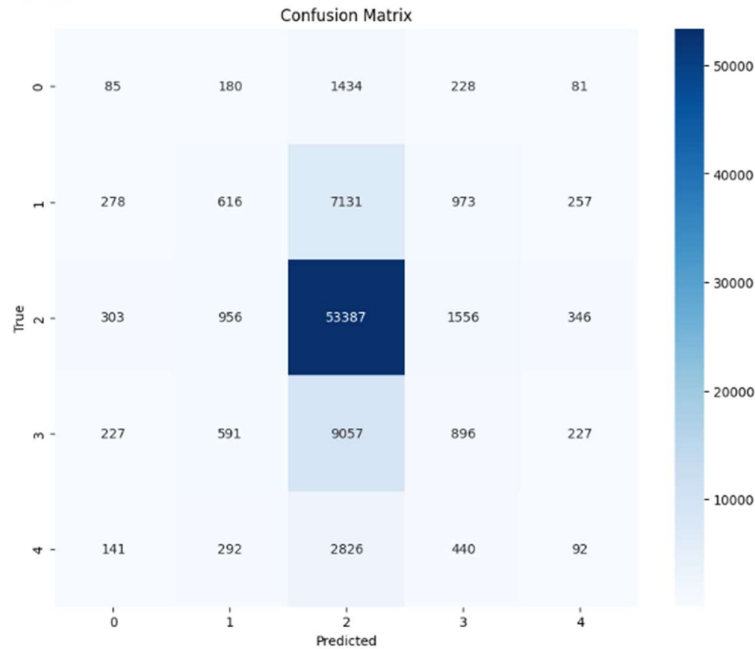
done

Classification Report:

	precision	recall	f1-score	support
0	0.08	0.04	0.06	2008
1	0.23	0.07	0.10	9255
2	0.72	0.94	0.82	56548
3	0.22	0.08	0.12	10998
4	0.09	0.02	0.04	3791

accuracy			0.67	82600
macro avg	0.27	0.23	0.23	82600
weighted avg	0.56	0.67	0.59	82600

Confusion Matrix:



Accuracy is 0.67. we see that the model performs decently compared to the benchmark. Class 2 i.e., the 'neutral' class has highest accuracy. Other classes are being misclassified as class 2 hence low precision. Here f1score is the least for class1 (0.01) and highest for class2(0.8)

2. Recursive NN

The base code only the model part is changed.

```

class RNTN(nn.Module):

    def __init__(self, word2index, hidden_size, output_size):
        super(RNTN, self).__init__()

        self.word2index = word2index
        self.embed = nn.Embedding(len(word2index), hidden_size)
        # self.V = nn.ModuleList([nn.Linear(hidden_size*2,hidden_size*2) for _ in range(hidden_size)])
        # self.W = nn.Linear(hidden_size*2,hidden_size)
        self.V = nn.ParameterList(
            [nn.Parameter(torch.randn(hidden_size * 2, hidden_size * 2)) for _ in range(hidden_size)] # Tensor
        self.W = nn.Parameter(torch.randn(hidden_size * 2, hidden_size))
        self.b = nn.Parameter(torch.randn(1, hidden_size))
        # self.W_out = nn.Parameter(torch.randn(hidden_size,output_size))
        self.W_out = nn.Linear(hidden_size, output_size)

    def init_weight(self):
        nn.init.xavier_uniform_(self.embed.state_dict()['weight'])
        nn.init.xavier_uniform_(self.W_out.state_dict()['weight'])
        for param in self.V.parameters():
            nn.init.xavier_uniform_(param)
        nn.init.xavier_uniform_(self.W)
        self.b.data.fill_(0)

        # nn.init.xavier_uniform(self.W_out)

    def tree_propagation(self, node):

        recursive_tensor = OrderedDict()
        current = None
        if node.isLeaf:
            tensor = Variable(torch.tensor([self.word2index[node.word]], dtype=torch.long, device=device)) \
                if node.word in self.word2index.keys() \
                else Variable(torch.tensor([self.word2index['<UNK>']], dtype=torch.long, device=device))
            current = self.embed(tensor) # 1xD
        else:
            recursive_tensor[node.word] = self.W_out(self.V[recursive_tensor[node.word]] + self.b)

```

```

def tree_propagation(self, node):
    recursive_tensor = OrderedDict()
    current = None
    if node.isLeaf:
        tensor = Variable(torch.tensor([self.word2index[node.word]], dtype=torch.long, device=device)) \
            if node.word in self.word2index.keys() \
            else Variable(torch.tensor([self.word2index['<UNK>']], dtype=torch.long, device=device))
        current = self.embed(tensor) # 1xD
    else:
        recursive_tensor.update(self.tree_propagation(node.left))
        recursive_tensor.update(self.tree_propagation(node.right))

        concated = torch.cat([recursive_tensor[node.left], recursive_tensor[node.right]], 1) # 1x2D
        xvX = []
        for i, v in enumerate(self.V):
            # xvX.append(torch.matmul(v(concated), concated.transpose(0,1)))
            xvX.append(torch.matmul(torch.matmul(concated, v), concated.transpose(0, 1)))

        xvX = torch.cat(xvX, 1) # 1xD
        # Wx = self.W(concated)
        Wx = torch.matmul(concated, self.W) # 1xD

        current = F.tanh(xvX + Wx + self.b) # 1xD
        recursive_tensor[node] = current
    return recursive_tensor

def forward(self, Trees, root_only=False):
    propagated = []
    if not isinstance(Trees, list):
        Trees = [Trees]

    for Tree in Trees:
        recursive_tensor = self.tree_propagation(Tree.root)
        if root_only:
            recursive_tensor = recursive_tensor[Tree.root]
            propagated.append(recursive_tensor)
        else:
            recursive_tensor = [tensor for node, tensor in recursive_tensor.items()]
            propagated.extend(recursive_tensor)

    propagated = torch.cat(propagated) # (num_of_node in batch, D)

    # return F.log_softmax(propagated.matmul(self.W_out))
    return F.log_softmax(self.W_out(propagated), 1)

```

This model is called Recursive Neural Tensor Network (RNTN). It's a type of neural network architecture designed to handle tree-structured data, such as parse trees that we are using.

1. Initialization: It takes parameters `word2index`, `hidden_size`, and `output_size`. It initializes the embedding layer (`embed`) with an embedding matrix for word representation. It also initializes the parameters `V`, `W`, `b`, and `W_out` which are the learnable parameters of the model.
2. Weight Initialization: The `init_weight` method initializes the weights of the embedding layer (`embed`), the output layer (`W_out`), and the tensor parameters (`V`, `W`) using Xavier initialization. Xavier initialization is a method used to initialize weights in a neural network to ensure that the weights are initialized neither too large nor too small.

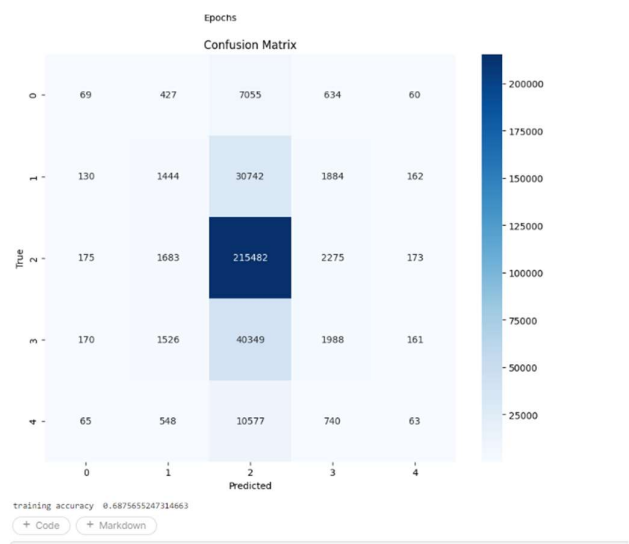
3. Tree Propagation: The `tree_propagation` method performs the recursive computation over the tree structure. It traverses the tree in a recursive manner, updating the tensors associated with each node based on the concatenation of tensors from its children nodes. It computes the output of each node using tensor operations involving parameters `V`, `W`, and `b`.

4. Forward Pass: The `forward` method takes a list of trees (`Trees`) as input and computes the output of the model. It iterates through each tree in the input list, computes the propagated tensor for each node using `tree_propagation`, and concatenates the tensors. If `root_only` is set to `True`, it returns only the output tensor corresponding to the root node; otherwise, it returns the output tensors for all nodes in the batch.

5. Output Layer: The output of the model is passed through a linear layer (`W_out`) followed by a log softmax activation function to produce the final output probabilities.

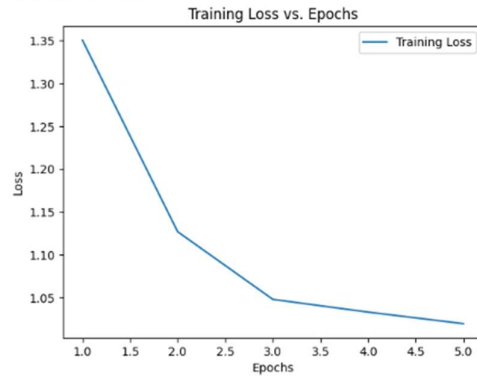
the RNTN model processes tree-structured data by recursively aggregating information from child nodes to parent nodes, capturing hierarchical relationships present in the data. It leverages tensor operations and learnable parameters to capture complex interactions between features at different levels of the tree.

Results:



```
[14]: #Train a model
train(model, LR, EPOCH)

/usr/local/lib/python3.10/site-packages/tqdm/autor.py:21: TqdmWarning: IProgress not found. Please u
from .autonotebook import tqdm as notebook_tqdm
[1/5] mean_loss : 1.35
[2/5] mean_loss : 1.13
[3/5] mean_loss : 1.09
[4/5] mean_loss : 1.03
[5/5] mean_loss : 1.02
```



Confusion Matrix

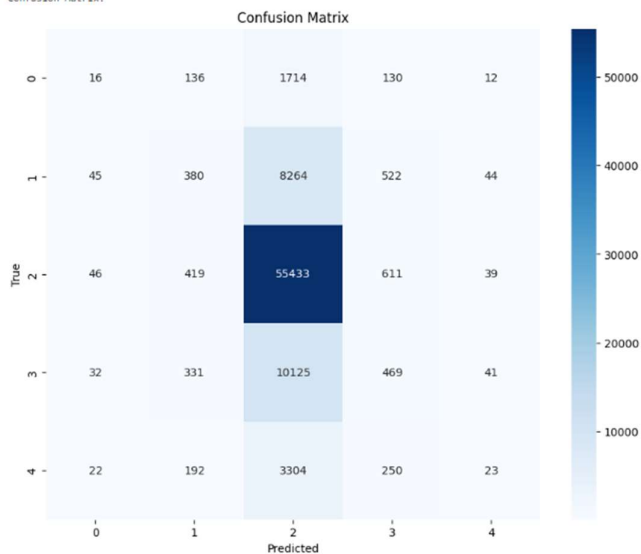
The training accuracy is 0.68 and higher than Bi-recNN. Still the model suffers the same problem as Bi-recNN many other classes get misclassified as class2 and class 2 has highest accuracy and f1 score of 0.82. classes 0 and 4 have the least of 0.01

```
[22]: # test a model
test(model)
```

Loading test trees..
done

	precision	recall	f1-score	support
0	0.10	0.01	0.01	2008
1	0.26	0.04	0.07	9255
2	0.70	0.08	0.82	56548
3	0.24	0.04	0.07	10998
4	0.14	0.01	0.01	3791
accuracy			0.68	82600
macro avg	0.29	0.22	0.20	82600
weighted avg	0.55	0.68	0.58	82600

Confusion Matrix:



The test accuracy is also 0.68. recall 0.68 and precision 0.55 are also low. Class 2 has the highest classwise f1 score and classes 0 and 4 have the least.

Overall Analysis:

The highest accuracy for this task was achieved using Bi-LSTM followed by recursive NN. The least performing model was RNN. All the models suffer the same problem of overestimation bias towards class 2. This might be due to high number of training sample of class 2. Since we are labelling phrases we need to note that most of the phrases are neutral. Lets say we have a sentence “(((I had) a (bad dream) today)”. This sentence has 4 phrases and 3 of them convey neutral feelings except one. This happens in almost all spoken sentences. Undersampling or oversampling is not an option here because datasets are trees. So we need to investigate further techniques on how to eliminate this bias and make our model perform better. But considering the benchmark for this dataset(0.69) our models performed decent and achieved good overall results.