AIL721: Deep Learning (Assignment - 3)

Prof. James Arambam

Harshit Joshi (2023MAS7141)

1 Problem Statement

Given a multi-category classification dataset

- Design a neural architecture that combines CNN and LSTM (or its variants) to solve the multi-category classification problem. You may explore whether the use of a self-attention mechanism at the output and/or dynamic meta-embedding at the input improves the performance of the neural architecture.
- Implement a Transformer-based text encoding model. You are expected to analyze its performance with respect to the number of encoder blocks and the effect of including or excluding positional embeddings.

2 Intro

Of the two reference papers cited, [1] and [2], [1] is a later and improved version of [2].
Table 1 in [1] shows the following:

- C-BiLSTM outperforms C-LSTM; however, standard LSTM performs slightly better than Bi-LSTM.
- The C-LSTM model with word embeddings performs better than both C-LSTM and C-BiLSTM without embeddings.

Model	Accuracy
textCNN	0.8157
LSTM	0.8497
Bi-LSTM	0.8485
C-LSTM	0.8566
C-BiLSTM	0.8601
C-LSTM with word embedding	0.8653

Table1 Model Accuracy Comparison

3 Word Embeddings

We have to create custom word embeddings which I've done as follows

3.1 HyperParameters

```
EMBEDDING_DIM = 300
BATCH_SIZE = 1026
EPOCHS = 30
LEARNING_RATE = 0.001
WINDOW_SIZE = 2
```

3.2 Preprocessing

```
sentences = df['Text'].tolist()
```

To tokenize the sentences I've used nltk

```
tokenized = [nltk.word_tokenize(sentence.lower()) for sentence in sentences]
```

Then I converted this into the vocabulary for our text corpus using `Counter`, I've also accounted for `<PAD>` and `<UNK>`

```
words = [word for sentence in tokenized_sentences for word in sentence]
word_counts = Counter(words)
vocab = {'<PAD>': 0, '<UNK>': 1}
vocab.update({word: i+2 for i, (word, _) in enumerate(word_counts.items())})
vocab_size = len(vocab)
```

As due to padding each context window will have the same size I've used

```
    torch.backends.cudnn.benchmark = True
```

for optimization

For computing context target pairs I've used

```
def compute_pairs(tokenized, WINDOW_SIZE, vocab):
    contexts = []
    targets = []
    for sentence in tokenized:
        for i in range(len(sentence)):
            start = max(0, i - WINDOW_SIZE)
            end = min(len(sentence), i + WINDOW_SIZE + 1)
            context = sentence[start:i] + sentence[i+1:end]
            context = context + ['<PAD>'] * (2 * WINDOW_SIZE - len(context))
            contexts.append([vocab.get(w, 1) for w in context])
            targets.append(vocab.get(sentence[i], 1))
    return torch.LongTensor(contexts), torch.LongTensor(targets)

contexts, targets = compute_pairs(tokenized, WINDOW_SIZE, vocab)
```

This

- Computes context and targets
- Pads if necessary
- Converts them to one hot encoded vectors according to our vocab using `<UNK>` for unseen words
- Converts them to torch tensors

3.3 DataSet and DataLoader Prep

I've tried optimizing the dataloader using `persistent_workers`, this doesnt destruct the workers for each epoch, a small optimization but optimization none the less.

```
dataset = TensorDataset(contexts, targets)
dataloader = DataLoader(
    dataset,
    batch_size=BATCH_SIZE,
    shuffle=True,
    num_workers=4,
    persistent_workers=True
)
```

The CBOW Model is as follows

```
class CBOW(nn.Module):
def __init__(self, v, e):
    super(CBOW, self).__init__()
    self.embeddings = nn.Embedding(v, e)
    self.linear = nn.Linear(e, v)

def forward(self, contexts):
    embeds = self.embeddings(contexts)
    embedcontexts = torch.mean(embeds, dim=1)
    out = self.linear(embedcontexts)
    return out
```

```
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
print(device)
model = CBOW(vocab_size, EMBEDDING_DIM).to(device)
criterion = nn.CrossEntropyLoss().to(device)
optimizer = optim.AdamW(model.parameters(), lr=LEARNING_RATE)

for epoch in range(EPOCHS):
    s = time.time()
    total_loss = 0

    for contexts, targets in dataloader:
        contexts = contexts.to(device, non_blocking=True)
        targets = targets.to(device, non_blocking=True)
        optimizer.zero_grad()
        outputs = model(contexts)
        loss = criterion(outputs, targets)
        loss.backward()
        optimizer.step()
        total_loss += loss.item()
    print(f"epoch {epoch} loss: {total_loss:.4f} time: {time.time() - s:.2f}s")
```

```
epoch 0 loss: 4374.2630 time: 16.88s
epoch 1 loss: 3541.0087 time: 16.67s
epoch 2 loss: 3205.5845 time: 16.76s
epoch 3 loss: 2956.2404 time: 17.11s
epoch 4 loss: 2754.9316 time: 17.01s
epoch 5 loss: 2587.6526 time: 16.94s
epoch 6 loss: 2447.7446 time: 17.48s
epoch 7 loss: 2331.2527 time: 16.66s
epoch 8 loss: 2230.9283 time: 16.67s
epoch 9 loss: 2142.7751 time: 16.67s
epoch 10 loss: 2064.1184 time: 17.72s
epoch 11 loss: 1992.7908 time: 16.87s
epoch 12 loss: 1927.2852 time: 16.56s
epoch 13 loss: 1867.3167 time: 16.56s
epoch 14 loss: 1811.8594 time: 17.17s
epoch 15 loss: 1760.1489 time: 17.02s
epoch 16 loss: 1712.1375 time: 16.74s
epoch 17 loss: 1666.8987 time: 16.68s
epoch 18 loss: 1624.6513 time: 17.33s
epoch 19 loss: 1585.0258 time: 16.88s
epoch 20 loss: 1547.7626 time: 16.72s
epoch 21 loss: 1512.6084 time: 16.86s
epoch 22 loss: 1479.5837 time: 16.95s
epoch 23 loss: 1448.3616 time: 16.91s
epoch 24 loss: 1419.0331 time: 16.86s
epoch 25 loss: 1391.3648 time: 16.71s
epoch 26 loss: 1365.1977 time: 17.18s
epoch 27 loss: 1340.4237 time: 17.09s
epoch 28 loss: 1317.1080 time: 17.00s
epoch 29 loss: 1295.0736 time: 16.60s
```

4 C-LSTM with Word Embeddings

We can now use these custom word embeddings to apply the C-LSTM model as proposed in [1]
As the paper is pretty vague about the dimensions I've taken values I saw fit

4.1 Hyperparameters

```
EMBEDDING_DIM = embedding_matrix.shape[1]
NUM_CLASSES = len(df['Category'].unique())
BATCH_SIZE = 64
EPOCHS = 10
LSTM_HIDDEN_DIM = 128
DENSE_EMBED_DIM = 128
CONV_OUT_DIM = 100
FUSION_DIM = 256
```

here embedding_matrix is extracted from our CBOW Model as follows

```
embedding_matrix = model.embeddings.weight.data.cpu().numpy()
```

4.2 Preprocessing and Data Prep

From the paper we can pad/truncate each sequence to match the size with the max length sentence in dataset

```
MAX_LEN = max(len(sequence) for sequence in sequences)
MAX_LEN
```

```
tokenized = [nltk.word_tokenize(text.lower()) for text in df['Text']]
sequences = [[vocab.get(word, 1) for word in seq] for seq in tokenized]
padded = torch.zeros((len(sequences), MAX_LEN), dtype=torch.long)
for i, seq in enumerate(sequences):
    length = min(len(seq), MAX_LEN)
    padded[i,:length] = torch.tensor(seq[:length])
```

for outputs ive used Label Encoder from `sklearn`

```
le = LabelEncoder()
labels = le.fit_transform(df['Category'])
```

The Dataset is as follows

```
class NewsDataset(Dataset):
    def __init__(self, sequences, labels):
        self.sequences = sequences
        self.labels = labels

    def __len__(self):
        return len(self.sequences)

    def __getitem__(self, idx):
        return self.sequences[idx], self.labels[idx]
```

To test our models later on i am splitting the dataset to train and test,

```
dataset_train = NewsDataset(train_sequences, train_labels)
dataset_test = NewsDataset(test_sequences, test_labels)

train_loader = DataLoader(dataset_train, batch_size=BATCH_SIZE, shuffle=True)
test_loader = DataLoader(dataset_test, batch_size=BATCH_SIZE, shuffle=False)
```

The model is implemented as follows

```python
class CLSTM_WE(nn.Module):
    def __init__(self, embedding_matrix, num_classes):
        super().__init__()

        self.v, self.e = embedding_matrix.shape

        self.embedding = nn.Embedding.from_pretrained(
            torch.FloatTensor(embedding_matrix),
            freeze=False
        )
        self.conv3 = nn.Conv1d(self.e, CONV_OUT_DIM, kernel_size=3)
        self.conv4 = nn.Conv1d(self.e, CONV_OUT_DIM, kernel_size=4)
        self.conv5 = nn.Conv1d(self.e, CONV_OUT_DIM, kernel_size=5)
        self.cnn_dropout = nn.Dropout(0.5)

        self.lstm = nn.LSTM(self.e, LSTM_HIDDEN_DIM, bidirectional=False, batch_first=True)
        self.lstm_dropout = nn.Dropout(0.5)

        self.embed_dense = nn.Linear(self.e, DENSE_EMBED_DIM)

        self.fc1 = nn.Linear(CONV_OUT_DIM*3 + LSTM_HIDDEN_DIM + DENSE_EMBED_DIM, FUSION_DIM)
        self.fc2 = nn.Linear(FUSION_DIM, num_classes)
        self.dropout = nn.Dropout(0.5)

    def forward(self, x):

        print(x.shape)

        x_embed = self.embedding(x)
        print(x_embed.shape)

        x_cnn = x_embed.permute(0, 2, 1)
        c3 = torch.relu(self.conv3(x_cnn)).max(dim=2)[0]
        c4 = torch.relu(self.conv4(x_cnn)).max(dim=2)[0]
        c5 = torch.relu(self.conv5(x_cnn)).max(dim=2)[0]
        cnn_out = torch.cat([c3, c4, c5], dim=1)
        cnn_out = self.cnn_dropout(cnn_out)

        lstm_out, _ = self.lstm(x_embed)
        lstm_out = lstm_out.mean(dim=1)
        lstm_out = self.lstm_dropout(lstm_out)

        embed_out = x_embed.mean(dim=1)
        embed_out = self.embed_dense(embed_out)

        combined = torch.cat([cnn_out, lstm_out, embed_out], dim=1)
        out = self.dropout(torch.relu(self.fc1(combined)))
        out = self.fc2(out)

        return out
```

We have 3 parts to our model

- raw embeddings
- cnn blocks
- lstm hidden states

We start with one-hot vectors (index in our vocab) and encode them using our `embedding_matrix`, these emebeddings are then passed onto c3, c4, c5 (conv blocks with kernel = 3, 4, 5), we apply `maxpool` to all these conv blocks and then `concat` them to a 300 dim vector, dropout is subsequently applied, For our LSTM block we use a simple LSTM block with hidden state dimension 128, we `averag_pool` over the sequential dimension to obtain a single vector for a whole sequence, For our embeddings we use a dense Linear layer to map them into 128 dim embeddings
These three outputs are then concatenated and connected to a Linear layer to shape 256, which is followed by another Layer to `num_classes`
Loss and Optimizer used is

```python
model = CLSTM_WE(embedding_matrix, NUM_CLASSES).to(device)
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)
```

I will use the standard 10 epochs for each model

```python
import time
for epoch in range(EPOCHS):
    model.train()
    total_loss = 0
    correct = 0
    total = 0
    s = time.time()
    for inputs, labels in train_loader:
        inputs, labels = inputs.to(device), labels.to(device)

        optimizer.zero_grad()
        outputs = model(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        total_loss += loss.item()
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

    print(f'epoch {epoch}, loss: {total_loss/len(train_loader):.4f} time: {time.time()-s:.2f} acc:
                                                        {correct/total:.4f}')
```

Results:

```
epoch 0, loss: 1.7013 time: 14.82 acc: 0.2273
epoch 1, loss: 1.4439 time: 6.15 acc: 0.4237
epoch 2, loss: 1.1029 time: 6.26 acc: 0.5646
epoch 3, loss: 0.7198 time: 6.24 acc: 0.7383
epoch 4, loss: 0.4687 time: 6.32 acc: 0.8356
epoch 5, loss: 0.2957 time: 6.34 acc: 0.9002
epoch 6, loss: 0.2109 time: 6.40 acc: 0.9279
epoch 7, loss: 0.1399 time: 6.43 acc: 0.9572
epoch 8, loss: 0.1388 time: 6.46 acc: 0.9547
epoch 9, loss: 0.0808 time: 6.51 acc: 0.9748
```

On test dataset the accuracy is

```
test_acc: 0.9497
```

5 C-BiLSTM with Word Embeddings

Here I am using Bi-LSTM instead of LSTM, the rest of the code is pretty much the same

```
class CLSTM_WE(nn.Module):
    def __init__(self, embedding_matrix, num_classes):
        super().__init__()

        self.v, self.e = embedding_matrix.shape

        self.embedding = nn.Embedding.from_pretrained(
            torch.FloatTensor(embedding_matrix),
            freeze=False
        )
        self.conv3 = nn.Conv1d(self.e, CONV_OUT_DIM, kernel_size=3)
        self.conv4 = nn.Conv1d(self.e, CONV_OUT_DIM, kernel_size=4)
        self.conv5 = nn.Conv1d(self.e, CONV_OUT_DIM, kernel_size=5)
        self.cnn_dropout = nn.Dropout(0.5)

        self.lstm = nn.LSTM(self.e, LSTM_HIDDEN_DIM, bidirectional=True, batch_first=True)
        self.lstm_dropout = nn.Dropout(0.5)

        self.embed_dense = nn.Linear(self.e, DENSE_EMBED_DIM)

        self.fc1 = nn.Linear(CONV_OUT_DIM*3 + 2*LSTM_HIDDEN_DIM + DENSE_EMBED_DIM, FUSION_DIM)
        self.fc2 = nn.Linear(FUSION_DIM, num_classes)
        self.dropout = nn.Dropout(0.5)

    def forward(self, x):

        print(x.shape)

        x_embed = self.embedding(x)
        print(x_embed.shape)

        x_cnn = x_embed.permute(0, 2, 1)
        c3 = torch.relu(self.conv3(x_cnn)).max(dim=2)[0]
        c4 = torch.relu(self.conv4(x_cnn)).max(dim=2)[0]
        c5 = torch.relu(self.conv5(x_cnn)).max(dim=2)[0]
        cnn_out = torch.cat([c3, c4, c5], dim=1)
        cnn_out = self.cnn_dropout(cnn_out)

        lstm_out, _ = self.lstm(x_embed)
        lstm_out = lstm_out.mean(dim=1)
        lstm_out = self.lstm_dropout(lstm_out)

        embed_out = x_embed.mean(dim=1)
        embed_out = self.embed_dense(embed_out)

        combined = torch.cat([cnn_out, lstm_out, embed_out], dim=1)
        out = self.dropout(torch.relu(self.fc1(combined)))
        out = self.fc2(out)

        return out
```

results are

```
epoch 0, loss: 1.6750 time: 8.51 acc: 0.2441
epoch 1, loss: 1.4227 time: 8.54 acc: 0.3935
epoch 2, loss: 0.9840 time: 8.70 acc: 0.6267
epoch 3, loss: 0.6536 time: 8.68 acc: 0.7651
epoch 4, loss: 0.4404 time: 8.82 acc: 0.8322
epoch 5, loss: 0.2663 time: 8.94 acc: 0.9144
epoch 6, loss: 0.1857 time: 9.04 acc: 0.9362
epoch 7, loss: 0.1425 time: 9.07 acc: 0.9530
epoch 8, loss: 0.1053 time: 9.04 acc: 0.9690
epoch 9, loss: 0.0879 time: 8.92 acc: 0.9748
```

```
test_acc: 0.9564
```

So I will be preferring C-BiLSTM henceforth in this paper

6 C-BiLSTM with Word Embeddings and Self-Attention

In this section I've used Self-Attention at the output to better understand the relationships among the three types/branches of outputs (raw embeddings, lstm, cnn) Unlike as in the earlier models we were concatenating the three outputs and then fusing them using a dense layer here we project all the branch outputs to a common dimension (`ATTN_DIM` = 100), these representations are then stacked which will make the tokens of shape $(64, 3, 100)$, these tokens are then passed to **`self_attn`** layer as Query, Key, Values to capture the interdependencies, **`self_attn`** layer embeds these tokens and then averages over the 0th dimn $(3 \rightarrow 1)$, the aggregated vector is then passed to fully connected layer after dropout for classification.

```
ATTN_DIM = 100

class CBiLSTM_WE_SA(nn.Module):
    def __init__(self, embedding_matrix, num_classes):
        super().__init__()
        self.v, self.e = embedding_matrix.shape

        self.embedding = nn.Embedding.from_pretrained(
            torch.FloatTensor(embedding_matrix),
            freeze=False
        )

        self.conv3 = nn.Conv1d(self.e, CONV_OUT_DIM, kernel_size=3)
        self.conv4 = nn.Conv1d(self.e, CONV_OUT_DIM, kernel_size=4)
        self.conv5 = nn.Conv1d(self.e, CONV_OUT_DIM, kernel_size=5)
        self.cnn_dropout = nn.Dropout(0.5)

        self.lstm = nn.LSTM(self.e, LSTM_HIDDEN_DIM, bidirectional=True, batch_first=True)
        self.lstm_dropout = nn.Dropout(0.5)

        self.embed_dense = nn.Linear(self.e, DENSE_EMBED_DIM)

        self.attn_dim = ATTN_DIM
        self.proj_cnn = nn.Linear(CONV_OUT_DIM * 3, self.attn_dim)
        self.proj_lstm = nn.Linear(2 * LSTM_HIDDEN_DIM, self.attn_dim)
        self.proj_embed = nn.Linear(DENSE_EMBED_DIM, self.attn_dim)

        self.self_attn = nn.MultiheadAttention(embed_dim=self.attn_dim, num_heads=4)

        self.fc2 = nn.Linear(self.attn_dim, num_classes)
        self.dropout = nn.Dropout(0.5)

    def forward(self, x):
        x_embed = self.embedding(x)

        x_cnn = x_embed.permute(0, 2, 1)
        c3 = torch.relu(self.conv3(x_cnn)).max(dim=2)[0]
        c4 = torch.relu(self.conv4(x_cnn)).max(dim=2)[0]
        c5 = torch.relu(self.conv5(x_cnn)).max(dim=2)[0]
        cnn_out = torch.cat([c3, c4, c5], dim=1)
        cnn_out = self.cnn_dropout(cnn_out)

        lstm_out, _ = self.lstm(x_embed)
        lstm_out = lstm_out.mean(dim=1)
        lstm_out = self.lstm_dropout(lstm_out)

        embed_out = x_embed.mean(dim=1)
        embed_out = self.embed_dense(embed_out)

        cnn_token = self.proj_cnn(cnn_out)
        lstm_token = self.proj_lstm(lstm_out)
        embed_token = self.proj_embed(embed_out)

        tokens = torch.stack([cnn_token, lstm_token, embed_token], dim=1)

        tokens = tokens.permute(1, 0, 2)
        attn_output, _ = self.self_attn(tokens, tokens, tokens)

        aggregated = attn_output.mean(dim=0)
        out = self.dropout(aggregated)
        out = self.fc2(out)

        return out
```

The results for this model are

```
epoch 0, loss: 1.5852 time: 18.56 acc: 0.2693
epoch 1, loss: 1.1371 time: 8.32 acc: 0.5487
epoch 2, loss: 0.6258 time: 8.34 acc: 0.7718
epoch 3, loss: 0.3503 time: 8.42 acc: 0.8867
epoch 4, loss: 0.1969 time: 8.51 acc: 0.9295
epoch 5, loss: 0.1552 time: 8.49 acc: 0.9446
epoch 6, loss: 0.1226 time: 8.43 acc: 0.9622
epoch 7, loss: 0.1016 time: 8.36 acc: 0.9622
epoch 8, loss: 0.0770 time: 8.31 acc: 0.9757
epoch 9, loss: 0.0438 time: 8.37 acc: 0.9815
```

```
test_acc: 0.9698
```

This is a huge improvement over the baseline C-BiLSTM_WE model

7 C-BiLSTM with Dynamic Meta Word Embeddings

For this section, Ive combined two different Word Embeddings, one from the matrix we've used till now and the other from Google' s official Word2Vec CBOW embeddings.

```
import gensim.downloader as api

model = api.load("word2vec-google-news-300")

embedding_dim = model.vector_size

embedding_matrix2 = np.zeros((len(vocab), embedding_dim))

for word, idx in vocab.items():
    if word in model.key_to_index:
        embedding_matrix2[idx] = model[word]
    else:
        embedding_matrix2[idx] = np.random.normal(scale=0.6, size=(embedding_dim, ))

print("Embedding matrix shape:", embedding_matrix2.shape)
```

I've used random values so that the words not in word2vec corpus (incase there are) wont be zeros and thus wont be indistinguishable from `<PAD>` and `<UNK>` in our custom model

```
    class CBiLSTM_WE_ME(nn.Module):
    def __init__(self, embedding_matrix1, embedding_matrix2, num_classes):
        super().__init__()
        self.v1, self.e = embedding_matrix1.shape
        self.v2, _ = embedding_matrix2.shape

        self.embedding1 = nn.Embedding.from_pretrained(
            torch.FloatTensor(embedding_matrix1),
            freeze=False
        )
        self.embedding2 = nn.Embedding.from_pretrained(
            torch.FloatTensor(embedding_matrix2),
            freeze=False
        )

        self.gate1 = nn.Linear(self.e, 1)
        self.gate2 = nn.Linear(self.e, 1)

        self.conv3 = nn.Conv1d(self.e, CONV_OUT_DIM, kernel_size=3)
        self.conv4 = nn.Conv1d(self.e, CONV_OUT_DIM, kernel_size=4)
        self.conv5 = nn.Conv1d(self.e, CONV_OUT_DIM, kernel_size=5)
        self.cnn_dropout = nn.Dropout(0.5)

        self.lstm = nn.LSTM(self.e, LSTM_HIDDEN_DIM, bidirectional=True, batch_first=True)
        self.lstm_dropout = nn.Dropout(0.5)

        self.embed_dense = nn.Linear(self.e, DENSE_EMBED_DIM)

        self.fc1 = nn.Linear(CONV_OUT_DIM * 3 + 2 * LSTM_HIDDEN_DIM + DENSE_EMBED_DIM, FUSION_DIM)
        self.fc2 = nn.Linear(FUSION_DIM, num_classes)
        self.dropout = nn.Dropout(0.5)

    def forward(self, x):

        emb1 = self.embedding1(x)
        emb2 = self.embedding2(x)

        gate1 = self.gate1(emb1)
        gate2 = self.gate2(emb2)
        gates = torch.cat([gate1, gate2], dim=2)
        weights = torch.softmax(gates, dim=2)

        meta_emb = weights[:, :, 0:1] * emb1 + weights[:, :, 1:2] * emb2

        x_cnn = meta_emb.permute(0, 2, 1)
        c3 = torch.relu(self.conv3(x_cnn)).max(dim=2)[0]
        c4 = torch.relu(self.conv4(x_cnn)).max(dim=2)[0]
        c5 = torch.relu(self.conv5(x_cnn)).max(dim=2)[0]
        cnn_out = torch.cat([c3, c4, c5], dim=1)
        cnn_out = self.cnn_dropout(cnn_out)

        lstm_out, _ = self.lstm(meta_emb)
        lstm_out = lstm_out.mean(dim=1)
        lstm_out = self.lstm_dropout(lstm_out)

        embed_out = meta_emb.mean(dim=1)
        embed_out = self.embed_dense(embed_out)

        combined = torch.cat([cnn_out, lstm_out, embed_out], dim=1)
        out = self.dropout(torch.relu(self.fc1(combined)))
        out = self.fc2(out)
        return out
```

Here, to combine both the different embeddings, we take a weighted average of both. This is done using:

$$\text{meta\_emb} = w_1 \cdot \text{emb}_1 + w_2 \cdot \text{emb}_2$$

where $w_1 = \text{softmax}(\text{gate}_1, [\text{gate}_1, \text{gate}_2])$, and gate_i is calculated by mapping emb_i to a single neuron.

The results

```
epoch 0, loss: 1.6233 time: 20.19 acc: 0.2483
epoch 1, loss: 1.3379 time: 9.27 acc: 0.4530
epoch 2, loss: 0.8365 time: 9.37 acc: 0.7039
epoch 3, loss: 0.5183 time: 9.43 acc: 0.8129
epoch 4, loss: 0.2948 time: 9.43 acc: 0.9086
epoch 5, loss: 0.1840 time: 9.41 acc: 0.9471
epoch 6, loss: 0.1272 time: 9.37 acc: 0.9513
epoch 7, loss: 0.0956 time: 9.31 acc: 0.9706
epoch 8, loss: 0.0681 time: 9.28 acc: 0.9782
epoch 9, loss: 0.0503 time: 9.29 acc: 0.9883
```

```
test_acc: 0.9631
```

This also is an improvement over the baseline model

8 Combining ME and SA

As both outperform the baseline model by a good margin, I tried combining both the approaches

```
class CLSTM_WE_ME_SA(nn.Module):
    def __init__(self, embedding_matrix1, embedding_matrix2, num_classes):
        super().__init__()

        self.v1, self.e = embedding_matrix1.shape
        self.v2, _ = embedding_matrix2.shape

        self.embedding1 = nn.Embedding.from_pretrained(
            torch.FloatTensor(embedding_matrix1), freeze=False
        )
        self.embedding2 = nn.Embedding.from_pretrained(
            torch.FloatTensor(embedding_matrix2), freeze=False
        )
        self.gate1 = nn.Linear(self.e, 1)
        self.gate2 = nn.Linear(self.e, 1)

        self.conv3 = nn.Conv1d(self.e, CONV_OUT_DIM, kernel_size=3)
        self.conv4 = nn.Conv1d(self.e, CONV_OUT_DIM, kernel_size=4)
        self.conv5 = nn.Conv1d(self.e, CONV_OUT_DIM, kernel_size=5)
        self.cnn_dropout = nn.Dropout(0.5)

        self.lstm = nn.LSTM(self.e, LSTM_HIDDEN_DIM, bidirectional=True, batch_first=True)
        self.lstm_dropout = nn.Dropout(0.5)

        self.embed_dense = nn.Linear(self.e, DENSE_EMBED_DIM)

        self.attn_dim = FUSION_DIM
        self.proj_cnn = nn.Linear(CONV_OUT_DIM * 3, self.attn_dim)
        self.proj_lstm = nn.Linear(2 * LSTM_HIDDEN_DIM, self.attn_dim)
        self.proj_embed = nn.Linear(DENSE_EMBED_DIM, self.attn_dim)

        self.self_attn = nn.MultiheadAttention(embed_dim=self.attn_dim, num_heads=4)

        self.fc2 = nn.Linear(self.attn_dim, num_classes)
        self.dropout = nn.Dropout(0.5)
    def forward(self, x):

        emb1 = self.embedding1(x)
        emb2 = self.embedding2(x)

        gate1 = self.gate1(emb1)
        gate2 = self.gate2(emb2)
        gates = torch.cat([gate1, gate2], dim=2)
        weights = torch.softmax(gates, dim=2)
        meta_emb = weights[:, :, 0:1] * emb1 + weights[:, :, 1:2] * emb2

        x_cnn = meta_emb.permute(0, 2, 1)
        c3 = torch.relu(self.conv3(x_cnn)).max(dim=2)[0]
        c4 = torch.relu(self.conv4(x_cnn)).max(dim=2)[0]
        c5 = torch.relu(self.conv5(x_cnn)).max(dim=2)[0]
        cnn_out = torch.cat([c3, c4, c5], dim=1)
        cnn_out = self.cnn_dropout(cnn_out)

        lstm_out, _ = self.lstm(meta_emb)
        lstm_out = lstm_out.mean(dim=1)
        lstm_out = self.lstm_dropout(lstm_out)

        embed_out = meta_emb.mean(dim=1)
        embed_out = self.embed_dense(embed_out)

        cnn_token = self.proj_cnn(cnn_out)
        lstm_token = self.proj_lstm(lstm_out)
        embed_token = self.proj_embed(embed_out)

        tokens = torch.stack([cnn_token, lstm_token, embed_token], dim=1)
        tokens = tokens.permute(1, 0, 2)

        attn_output, _ = self.self_attn(tokens, tokens, tokens)
        aggregated = attn_output.mean(dim=0)

        out = self.dropout(aggregated)
        out = self.fc2(out)
        return out
```

The results are a bit dissapointing:

```
epoch 0, loss: 1.5235 time: 20.74 acc: 0.2978
epoch 1, loss: 0.7981 time: 9.77 acc: 0.6988
epoch 2, loss: 0.3563 time: 9.80 acc: 0.8876
epoch 3, loss: 0.2082 time: 10.00 acc: 0.9413
epoch 4, loss: 0.1252 time: 10.10 acc: 0.9488
epoch 5, loss: 0.1137 time: 10.10 acc: 0.9555
epoch 6, loss: 0.1002 time: 9.98 acc: 0.9715
epoch 7, loss: 0.0872 time: 9.90 acc: 0.9757
epoch 8, loss: 0.0540 time: 9.82 acc: 0.9849
epoch 9, loss: 0.0586 time: 9.85 acc: 0.9773
```

```
test_acc: 0.9564
```

However I decided to give this model another try, so I trained it for yet another epoch

```
epoch 0, loss: 0.0481 time: 9.85 acc: 0.9891
```

```
test_acc: 0.9631
```

So now I will be sticking to only Self-Attention as it performs the best on the test set

9 Using Transformers

Here I've used Transformer Encoder on my embeddings (with Self Attention at the output layers)

```
ATTN_DIM = 100

class CBiLSTM_WE_SA_T(nn.Module):
    def __init__(self, embedding_matrix, num_classes):
        super().__init__()
        self.v, self.e = embedding_matrix.shape

        self.embedding = nn.Embedding.from_pretrained(
            torch.FloatTensor(embedding_matrix),
            freeze=False
        )

        T_enc = nn.TransformerEncoderLayer(
            d_model=self.e,
            nhead=4,
            dim_feedforward=2048
        )
        self.trans_enc = nn.TransformerEncoder(T_enc, num_layers=1)
        self.conv3 = nn.Conv1d(self.e, CONV_OUT_DIM, kernel_size=3)
        self.conv4 = nn.Conv1d(self.e, CONV_OUT_DIM, kernel_size=4)
        self.conv5 = nn.Conv1d(self.e, CONV_OUT_DIM, kernel_size=5)
        self.cnn_dropout = nn.Dropout(0.5)

        self.lstm = nn.LSTM(self.e, LSTM_HIDDEN_DIM, bidirectional=True, batch_first=True)
        self.lstm_dropout = nn.Dropout(0.5)

        self.embed_dense = nn.Linear(self.e, DENSE_EMBED_DIM)

        self.attn_dim = ATTN_DIM
        self.proj_cnn = nn.Linear(CONV_OUT_DIM * 3, self.attn_dim)
        self.proj_lstm = nn.Linear(2 * LSTM_HIDDEN_DIM, self.attn_dim)
        self.proj_embed = nn.Linear(DENSE_EMBED_DIM, self.attn_dim)

        self.self_attn = nn.MultiheadAttention(embed_dim=self.attn_dim, num_heads=4)

        self.fc2 = nn.Linear(self.attn_dim, num_classes)
        self.dropout = nn.Dropout(0.5)

    def forward(self, x):
        x_embed = self.embedding(x)
        x_enc = self.trans_enc(x_embed.permute(1, 0, 2)).permute(1, 0, 2)

        x_cnn = x_enc.permute(0, 2, 1)
        c3 = torch.relu(self.conv3(x_cnn)).max(dim=2)[0]
        c4 = torch.relu(self.conv4(x_cnn)).max(dim=2)[0]
        c5 = torch.relu(self.conv5(x_cnn)).max(dim=2)[0]
        cnn_out = torch.cat([c3, c4, c5], dim=1)
        cnn_out = self.cnn_dropout(cnn_out)

        lstm_out, _ = self.lstm(x_enc)
        lstm_out = lstm_out.mean(dim=1)
        lstm_out = self.lstm_dropout(lstm_out)

        embed_out = x_enc.mean(dim=1)
        embed_out = self.embed_dense(embed_out)

        cnn_token = self.proj_cnn(cnn_out)
        lstm_token = self.proj_lstm(lstm_out)
        embed_token = self.proj_embed(embed_out)

        tokens = torch.stack([cnn_token, lstm_token, embed_token], dim=1)

        tokens = tokens.permute(1, 0, 2)
        attn_output, _ = self.self_attn(tokens, tokens, tokens)

        aggregated = attn_output.mean(dim=0)
        out = self.dropout(aggregated)
        out = self.fc2(out)

        return out
```

The results are

```
epoch 0, loss: 1.6248 time: 74.92 acc: 0.2156
epoch 1, loss: 1.4452 time: 71.81 acc: 0.3238
epoch 2, loss: 1.0107 time: 71.99 acc: 0.5789
epoch 3, loss: 0.5512 time: 72.18 acc: 0.8037
epoch 4, loss: 0.3083 time: 72.09 acc: 0.9111
epoch 5, loss: 0.1879 time: 72.02 acc: 0.9438
epoch 6, loss: 0.1299 time: 72.13 acc: 0.9681
epoch 7, loss: 0.1397 time: 72.12 acc: 0.9648
epoch 8, loss: 0.1218 time: 71.96 acc: 0.9639
epoch 9, loss: 0.1880 time: 71.90 acc: 0.9614
```

```
Test Accuracy: 0.9060
```

This model widely under-delivers, even when compared to our skeletal C-LSTM-WE Model
Not just that it is computationally much more expensive than the rest.

10 Using Positional Embeddings

Transformers have no inherent sense of word order, I use positional embeddings (sinosuidal fixed) before the transformer encoding block to adress this, these positional embeddings are added to our original embeddings

This allows the self-attention mechanism to make use of the relative and absolute positions of tokens in the sequence.

```
ATTN_DIM = 100

import torch
import torch.nn as nn
import math

class CBiLSTM_WE_SA_T_PE(nn.Module):
    def __init__(self, embedding_matrix, num_classes):
        super().__init__()
        self.v, self.e = embedding_matrix.shape

        self.embedding = nn.Embedding.from_pretrained(
            torch.FloatTensor(embedding_matrix),
            freeze=False
        )

        position = torch.arange(MAX_LEN).unsqueeze(1)
        div_term = torch.exp(torch.arange(0, self.e, 2) * (-math.log(10000.0) / self.e))
        pe = torch.zeros(MAX_LEN, 1, self.e)
        pe[:, 0, 0::2] = torch.sin(position * div_term)
        pe[:, 0, 1::2] = torch.cos(position * div_term)
        self.register_buffer('pe', pe)

        T_enc = nn.TransformerEncoderLayer(
            d_model=self.e,
            nhead=4,
            dim_feedforward=2048
        )
        self.trans_enc = nn.TransformerEncoder(T_enc, num_layers=1)

        self.conv3 = nn.Conv1d(self.e, CONV_OUT_DIM, kernel_size=3)
        self.conv4 = nn.Conv1d(self.e, CONV_OUT_DIM, kernel_size=4)
        self.conv5 = nn.Conv1d(self.e, CONV_OUT_DIM, kernel_size=5)
        self.cnn_dropout = nn.Dropout(0.5)

        self.lstm = nn.LSTM(self.e, LSTM_HIDDEN_DIM, bidirectional=True, batch_first=True)
        self.lstm_dropout = nn.Dropout(0.5)

        self.embed_dense = nn.Linear(self.e, DENSE_EMBED_DIM)

        self.attn_dim = ATTN_DIM
        self.proj_cnn = nn.Linear(CONV_OUT_DIM * 3, self.attn_dim)
        self.proj_lstm = nn.Linear(2 * LSTM_HIDDEN_DIM, self.attn_dim)
        self.proj_embed = nn.Linear(DENSE_EMBED_DIM, self.attn_dim)
        self.self_attn = nn.MultiheadAttention(embed_dim=self.attn_dim, num_heads=4)

        self.fc2 = nn.Linear(self.attn_dim, num_classes)
        self.dropout = nn.Dropout(0.5)

    def forward(self, x):

        x_embed = self.embedding(x)
        x_embed_permuted = x_embed.permute(1, 0, 2)
        seq_len = x_embed_permuted.size(0)

        x_embedpos = x_embed_permuted + self.pe[:seq_len]

        x_enc = self.trans_enc(x_embedpos)

        x_enc = x_enc.permute(1, 0, 2)

        x_cnn = x_enc.permute(0, 2, 1)
        c3 = torch.relu(self.conv3(x_cnn)).max(dim=2)[0]
        c4 = torch.relu(self.conv4(x_cnn)).max(dim=2)[0]
        c5 = torch.relu(self.conv5(x_cnn)).max(dim=2)[0]
        cnn_out = torch.cat([c3, c4, c5], dim=1)
        cnn_out = self.cnn_dropout(cnn_out)

        lstm_out, _ = self.lstm(x_enc)
        lstm_out = lstm_out.mean(dim=1)
        lstm_out = self.lstm_dropout(lstm_out)

        embed_out = x_enc.mean(dim=1)
        embed_out = self.embed_dense(embed_out)

        cnn_token = self.proj_cnn(cnn_out)
        lstm_token = self.proj_lstm(lstm_out)
        embed_token = self.proj_embed(embed_out)

        tokens = torch.stack([cnn_token, lstm_token, embed_token], dim=1)
        tokens = tokens.permute(1, 0, 2)
        attn_output, _ = self.self_attn(tokens, tokens, tokens)

        aggregated = attn_output.mean(dim=0)
        out = self.dropout(aggregated)
        out = self.fc2(out)

        return out
```

The results are

```
epoch 0, loss: 1.6185 time: 76.17 acc: 0.2391
epoch 1, loss: 1.3244 time: 74.76 acc: 0.4312
epoch 2, loss: 0.9060 time: 76.02 acc: 0.6602
epoch 3, loss: 0.6128 time: 76.88 acc: 0.7945
epoch 4, loss: 0.4445 time: 77.02 acc: 0.8624
epoch 5, loss: 0.3372 time: 76.93 acc: 0.8935
epoch 6, loss: 0.3436 time: 76.91 acc: 0.9018
epoch 7, loss: 0.2683 time: 76.69 acc: 0.9253
epoch 8, loss: 0.2393 time: 76.63 acc: 0.9329
epoch 9, loss: 0.2159 time: 76.73 acc: 0.9379
```

```
Test Accuracy: 0.8758
```

So Positional Embeddings didn't really help us here, will be ignoring them from now.

11 Using 5 Transformer Blocks instead of 1

This part is pretty much the same as the Transformer one except here

```
self.trans_enc = nn.TransformerEncoder(T_enc, num_layers=5)
```

`num_layers` is 5
The results are

```
epoch 0, loss: 1.6309 time: 341.04 acc: 0.2156
epoch 1, loss: 1.6203 time: 345.31 acc: 0.2122
epoch 2, loss: 1.6135 time: 345.80 acc: 0.2139
epoch 3, loss: 1.6105 time: 345.22 acc: 0.2500
epoch 4, loss: 1.6137 time: 345.45 acc: 0.2164
epoch 5, loss: 1.6067 time: 345.34 acc: 0.2248
epoch 6, loss: 1.6136 time: 345.60 acc: 0.2072
epoch 7, loss: 1.6102 time: 345.10 acc: 0.2282
epoch 8, loss: 1.6095 time: 344.77 acc: 0.2198
epoch 9, loss: 1.6065 time: 344.60 acc: 0.2232
```

```
Test Accuracy: 0.2248
```

This was computationally the heaviest model and it produced the worst result. I couldn't really figure out if there was something wrong on my behalf as the accuracy oscillates widely, my initial thought was something regarding the optimizer, however I've used the same optimizer as I did with the 1 Transformer Classifier also I just changed a single parameter to 5 so it should've been alright.
Training this took me about an hour, so I couldn't really gather the courage to run it again.

12 Final Architecture

I've used C-BiLSTM with Word Embeddings and Self-Attention as my final choice of model as it performed the best on testset. The f1 score achieved on `TestLabels.csv` was however just `0.2059` which was highly de-motivating.

13 References

[1] Minyong Shi, Kaixiang Wang, and Chunfang Li. A c-lstm with word embedding model for news text classification. In 2019 IEEE/ACIS 18th International Conference on Computer and Information Science (ICIS), pages 253–257, 2019.
[2] Chunting Zhou, Chonglin Sun, Zhiyuan Liu, and Francis C. M. Lau. A c-lstm neural network for text classification, 2015.