Machine Learning for NLP - 1

Exercise - 3

Goal:

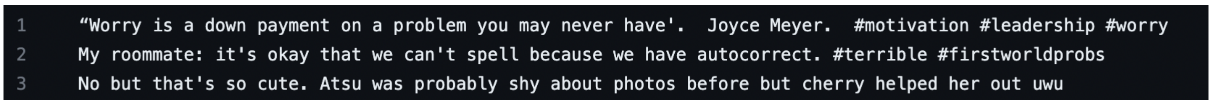
The goal was to implement emotion recognition classifiers in PyTorch with Convolutional Neural Networks. Two emotion recognition binary classifiers are implemented in this exercise:

1. Anger or Joy classifier
2. Anger or Sadness classifier

Dataset:

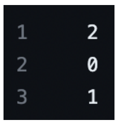
For training, validation, and testing purposes we used the Tweeteval dataset which can be found here: https://github.com/cardiffnlp/tweeteval

Example Data:

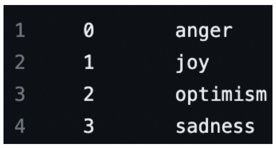


```
“Worry is a down payment on a problem you may never have'.  Joyce Meyer.  #motivation #leadership #worry
My roommate: it's okay that we can't spell because we have autocorrect. #terrible #firstworldprobs
No but that's so cute. Atsu was probably shy about photos before but cherry helped her out uwu
```

Example Labels and Labels Mapping:



```
2
0
1
```



```
0       anger
1       joy
2       optimism
3       sadness
```

Dataset Preprocessing:

We first divided the above data into two datasets:

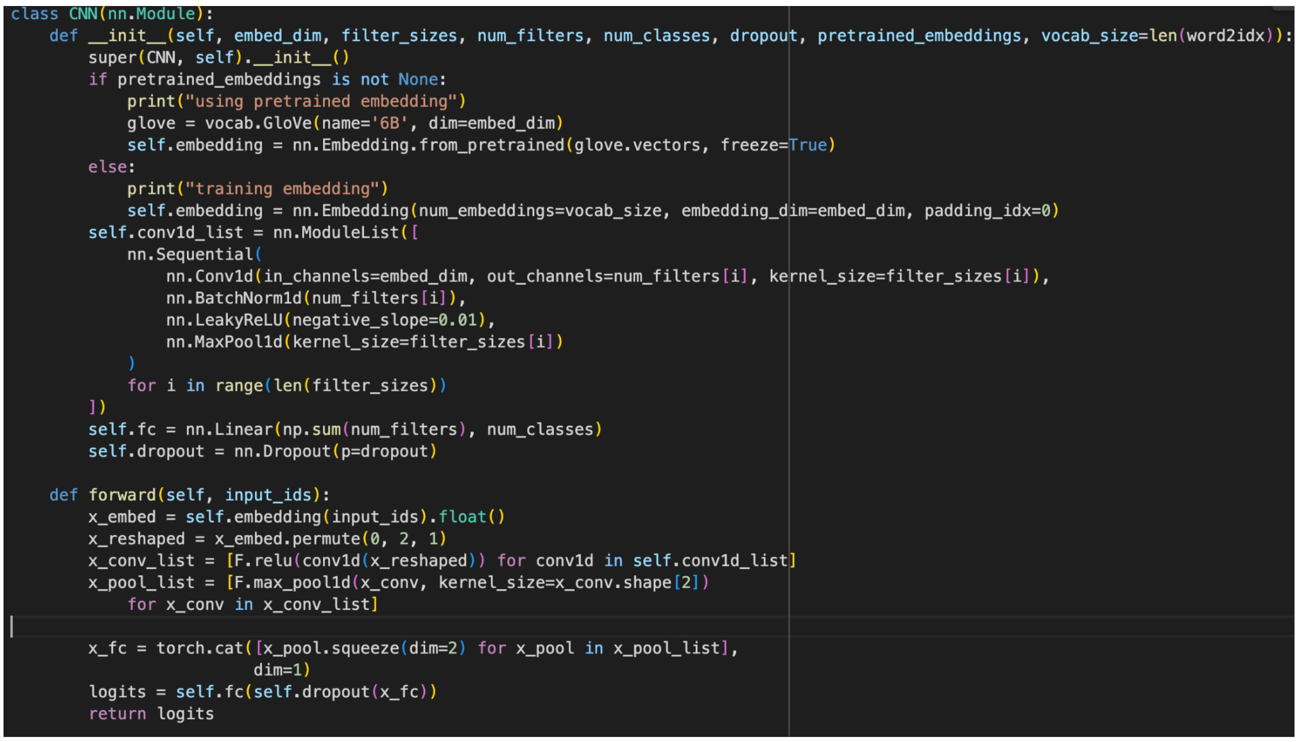
1. Anger and Joy dataset
2. Anger and Sadness dataset

We applied preprocessing to both the dataset. Steps followed are as follows:

1. Tokenization: Splitted the text in words/tokens
2. Unique ids: We assign unique id to each tokens
3. Padding: We applied padding to pad each sentences to maximum length
4. Data split: Since our dataset was already splitted **no** data splitted was required

Network Architecture:

Shown below is the CNN architecture used to train the emotion recognition model. We used pre-trained word embeddings from torchtext.vocab named GloVe. We kept embedding dimension variables so that we can tune it along with other hyperparameters.



```
class CNN(nn.Module):
    def __init__(self, embed_dim, filter_sizes, num_filters, num_classes, dropout, pretrained_embeddings, vocab_size=len(word2idx)):
        super(CNN, self).__init__()
        if pretrained_embeddings is not None:
            print("using pretrained embedding")
            glove = vocab.GloVe(name='6B', dim=embed_dim)
            self.embedding = nn.Embedding.from_pretrained(glove.vectors, freeze=True)
        else:
            print("training embedding")
            self.embedding = nn.Embedding(num_embeddings=vocab_size, embedding_dim=embed_dim, padding_idx=0)
        self.conv1d_list = nn.ModuleList([
            nn.Sequential(
                nn.Conv1d(in_channels=embed_dim, out_channels=num_filters[i], kernel_size=filter_sizes[i]),
                nn.BatchNorm1d(num_filters[i]),
                nn.LeakyReLU(negative_slope=0.01),
                nn.MaxPool1d(kernel_size=filter_sizes[i])
            )
            for i in range(len(filter_sizes))
        ])
        self.fc = nn.Linear(np.sum(num_filters), num_classes)
        self.dropout = nn.Dropout(p=dropout)

    def forward(self, input_ids):
        x_embed = self.embedding(input_ids).float()
        x_reshaped = x_embed.permute(0, 2, 1)
        x_conv_list = [F.relu(conv1d(x_reshaped)) for conv1d in self.conv1d_list]
        x_pool_list = [F.max_pool1d(x_conv, kernel_size=x_conv.shape[2])
            for x_conv in x_conv_list]

        x_fc = torch.cat([x_pool.squeeze(dim=2) for x_pool in x_pool_list],
                         dim=1)
        logits = self.fc(self.dropout(x_fc))
        return logits
```

Training and Results:

For training, we used SGD optimizer alongside weight decay due to the constraints of a relatively small dataset. Given the risk of overfitting, we favoured SGD, a simpler optimizer compared to Adam. Additionally, the incorporation of weight decay was essential to mitigate overfitting issues.

We employed the Cross Entropy Loss during training, as our task involved binary classification, and Cross Entropy Loss aligns perfectly with this specific classification scenario.

While training we saved the best model using validation accuracy and used the same model for testing. We used loss, accuracy, and F1-macro scores to evaluate the performance of our model.

We first selected Anger and Joy dataset and found the optimal model architecture and training regime for our CNN classifier. We experimented with the model on several hyperparameters. Following are some better performing combinations:

Model Name	Embedding dimension	Learning rate	Dropouts	Weight Decay	Filter Sizes	Number of Filters
A	**300**	**0.0001**	**0.2**	**0.00001**	**[3, 3, 4, 5]**	**[100, 100, 100, 100]**
B	300	0.0001	0.5	0.00001	[3, 4, 5]	[100, 100, 100]
C	300	0.001	0.2	0.000001	[3, 3, 4, 5]	[100, 100, 100, 100]
D	50	0.0001	0.2	0.00001	[3, 3, 4, 5]	[150, 150, 150, 150]
E	50	0.0001	0.2	0.00001	[3, 4, 5]	[150, 150, 150]

Following are Accuracies and F1-macro on development dataset of Anger and Joy dataset:

Model Name	Training Accuracy	Validation Accuracy	Training F1-macro	Validation F1-macro
A	**0.65**	**0.61**	**0.57**	**0.43**
B	0.63	0.61	0.56	0.4
C	0.86	0.61	0.84	0.39
D	0.63	0.61	0.55	0.41
E	0.63	0.6	0.53	0.38

Based on the experiments shown above, we selected Model **A** with its corresponding hyperparameters because unlike Model C, it is not overfitting, and has better Validation Metrics than all other Models. We used Model **A** setting to train another Emotion Recognition Model on Anger and Sadness dataset. Following are accuracies and F1-macro on test sets of 1) Anger and Joy dataset and 2) Anger and Sadness dataset.

Dataset	Test Accuracy	Test F1-macro
Anger and Joy Dataset	0.61	0.43
Anger and Sadness Dataset	0.59	0.43

There can be a variety of reasons why specific combinations of hyperparameters resulted in the best performing model. The identification of overfitting in our model was apparent, with the training accuracy significantly surpassing the validation accuracy. To mitigate this, we introduced regularisation through weight decays in the stochastic gradient descent (SGD) optimizer. The selection of a weight decay value of 0.00001 was deliberate, aiming to strike a balance that minimises overfitting. Additionally, our observation revealed that deeper neural networks, characterised by an increased number of layers, tend to exhibit superior performance. Hence, the enhanced performance of Model A, featuring four conv1d layers, could be attributed to its deeper architecture.

Limitations and Challenges:

1. As previously noted, our model experienced overfitting, prompting the implementation of various preventive measures, including the introduction of weight decays and dropouts.
2. The inadequacy of the provided dataset could be a contributing factor to the observed overfitting phenomenon.
3. Moreover, our ability to fine-tune hyperparameters, particularly during multiple iterations necessitated by the overfitting challenge, was constrained by limited GPU access. This constraint led to the exhaustive utilisation of resources allocated under the free Google Colab account.