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

1	2
2	0
3	1

1	0	anger
2	1	joy
3	2	optimism
4	3	sadness

Dataset Preprocessing:

Dataset Cleaning:

We first clean the dataset as follows:

1. Decoded all the ascii values.
2. Removed all the twitter user tags eg: @user, @john, etc.
3. Removed all the words which are hashtags eg: #itiswhatitis, etc.
4. Removed all the punctuations eg: "!", ":", etc.
5. Removed all the stopwords from the sentences eg: "what", "your", etc.
6. Applied Lemmatization eg: "process" and "processing" should be the same word.

Dataset Splitting:

We first divided the above data into two datasets:

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

Dataset Balancing:

After splitting the dataset into two datasets for respective classifiers, we balanced the labelled classes. We upsampled the minority class by randomly replicating instances of the minority class.

Dataset Processing:

We processed the datasets to prepare them for training. 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:

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.

Given below is detailed architecture of CNN implemented for emotion recognition classifier:

1. GloVe embedding of variable dimension (dimension tuned during hyperparameter tuning)
2. A variable block of CNN which includes conv1d, batch normalisation, Relu activation function, max pooling 1 dimensional. This block length is variable and tuned during hyperparameter tuning. All the hyperparameters in each block were also tuned during hyperparameter tuning.
3. A fully connected layer was added with dropouts to avoid overfitting. Dropout value was kept variable and tuned during hyperparameter tuning.

CNN Network Architecture

```
import torchtext.vocab as vocab
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])
            )
            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 f1-macro and used the best model for testing. We used 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:

Table showing hyperparameter tuning for Anger and Joy Classifier

Model Name	Embedding dimension	Learning rate	Dropouts	Weight Decay	Filter Sizes	Number of Filters

A	300	0.0001	0.5	0.00001	[3, 4, 5]	[100, 100, 100]
B	300	0.001	0.2	0.000001	[3, 3, 4, 5]	[100, 100, 100, 100]
C	**300**	**0.0001**	**0.5**	**0.001**	**[3, 3, 4, 4, 5]**	**[100, 100, 100, 100, 100]**
D	50	0.0001	0.2	0.00001	[3, 4, 5]	[150, 150, 150]

Epochs = 50, Batch size = 64, Optimizer = SGD are the best performing hyperparameters and therefore all the noted hyperparameters in the table mentioned above were trained using these values.

Table showing metrics on dev set of Anger and Joy Classifier (on last epoch)

Model Name	Training Accuracy	Validation Accuracy	Training F1-macro	Validation F1-macro
A	0.6114	0.5312	0.6114	0.5294
B	0.9600	0.5844	0.9600	0.5840
C	0.6136	**0.6031**	0.6136	**0.6014**
D	0.5800	0.5656	0.5800	0.5610

Based on the above experiments, we selected model C as the best model because

1. Model is not overfitting like model B.
2. Model has the best Validation Accuracy and Validation F1-macro scores among all other models.

Therefore, we used the setting of Model C to train another Emotion Recognition Model on **Anger and Sadness dataset.**

Table showing metrics on dev set of Anger and Sadness Classifier (on last epoch)

Model Name	Training	Validation	Training	Validation

	Accuracy	Accuracy	F1-macro	F1-macro
E	0.5950	0.4969	0.5950	0.4876

Table showing metrics on Test set on both datasets

Dataset	Test Accuracy	Test F1-macro
Anger and Joy Dataset (Model C)	0.5484	0.5483
Anger and Sadness Dataset (Model E)	0.5582	0.5582

There can be a variety of reasons why specific combinations of hyperparameters resulted in the best performing model. We observed from the example of Model B that our model was overfitting. In order to avoid that, we increase weight decay and dropout value. Therefore, a dropout value of 0.5 and weight decay of 0.001 worked the best in our situation. Secondly, we observed that on increasing learning rate, our model is overfitting even more and convergence was unstable. Therefore, we kept a relatively smaller learning rate. In order to increase the accuracy and F1-macro, we increased the number of filters and number of filter sizes i.e. a deeper network. Having a deeper network with 5 conv1d layers helped in increasing accuracy and F1-macro of Model.

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. Dataset provided was not clean. Lots of measures have been taken to clean and balance the dataset.
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.