Emotion Recognition with a CNN

November 12, 2023

Contents

1 Introduction

This report outlines the development and evaluation of a Convolutional Neural Network (CNN) for emotion recognition, a task that forms a crucial part of natural language processing and machine learning. Emotion recognition in text involves categorizing textual data into different emotional states, a challenge that has significant implications in fields ranging from customer service to mental health assessment.

In this study, we focus on classifying text data into two specific emotion classes. The choice of emotions, such as anger and joy, and subsequently anger and sadness, provides a diverse range of emotional contexts to test the adaptability and accuracy of the CNN model. This approach deviates from the character-level analysis typically used in text classification, instead opting for a word-level approach that offers a more nuanced understanding of emotional expressions.

A critical aspect of this work is the exploration and optimization of the CNN architecture and its hyperparameters. By experimenting with various hyperparameters such as optimizer types, learning rates, dropout rates, filter numbers, strides, kernel sizes, pooling types, and batch sizes, the study aims to identify the most effective configuration for emotion classification. This process involves creating two datasets based on selected emotion classes and iteratively tweaking the model to achieve the best possible accuracy and F1-macro scores.

2 Exploarative Data Analysis(EDA) and Data Preprocessing

2.1 EDA

The exploratory data analysis commenced with an assessment of the distribution of emotion labels within the training set, as shown in the first figure. The bar chart revealed a disproportionate frequency of labels, with 'anger' being the most prevalent emotion, followed by 'joy' and 'sadness', while 'optimism' was the least represented. This imbalance underscores the potential for bias in the classifier's training and the necessity for a balanced approach to ensure equitable learning across emotions.

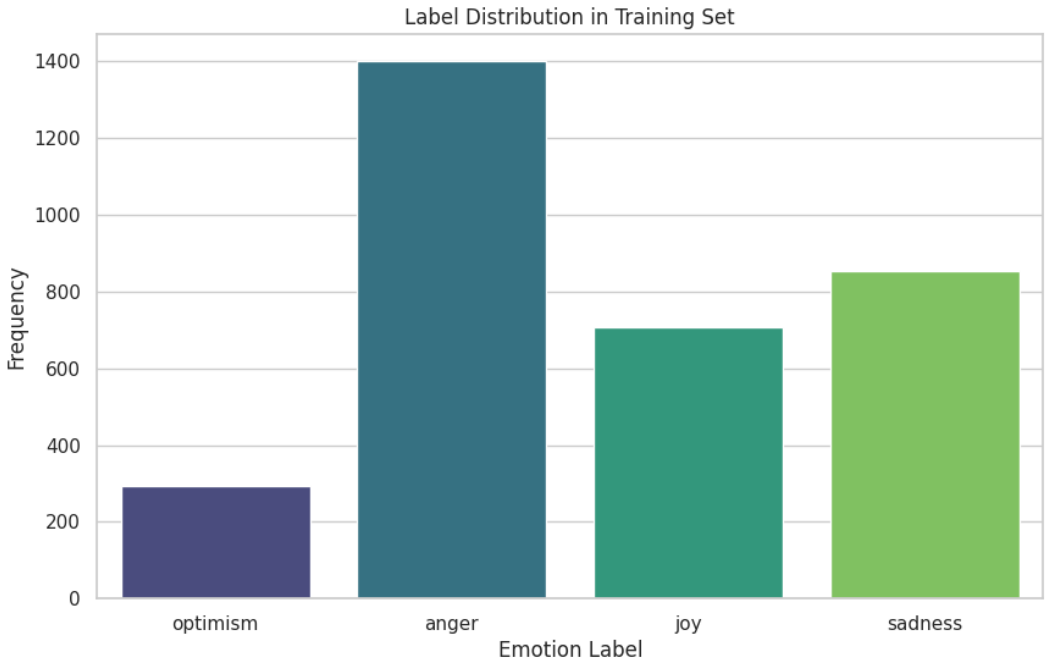

Figure 1: Distribution of Emotion Labels

The second figure, a histogram, provided insights into the text length distribution, highlighting a skew towards shorter text lengths with the most common length being around 50 characters. This observation indicates a dataset characterized by concise expressions of emotion, which could impact the granularity of the CNN's text analysis. The kernel density estimate suggests a smooth distribution with some texts extending to greater lengths, which may represent more elaborate emotional expressions. Together, these visuals form a foundation for understanding the dataset's structure and guide the preprocessing steps to optimize the CNN's performance in recognizing and classifying a diverse range of emotional expressions in text data.

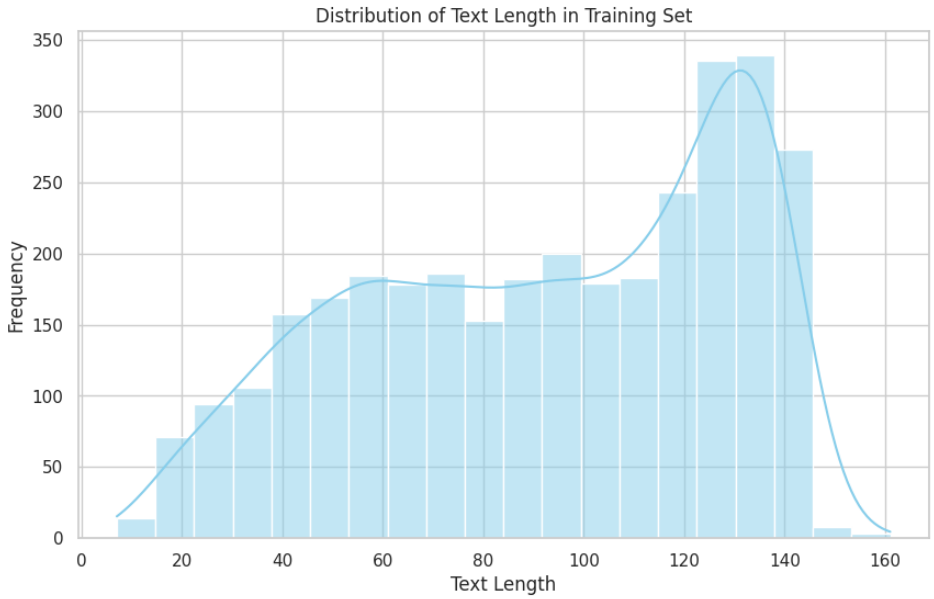

Figure 2: Distribution of Emotion Labels

2.2 Data Preprocessing

Data preprocessing is a critical step in ensuring that the text data fed into the Convolutional Neural Network (CNN) is clean and standardized, thereby enhancing the model's ability to learn from the data effectively. In this project, a series of preprocessing steps were applied to the raw tweets to prepare them for emotion classification. Initially, all tweets were converted to lowercase to maintain consistency and avoid duplication based on case differences. Following this, any mentions and URLs, which are irrelevant to the emotional content, were stripped from the text using regular expressions. Punctuations, which could potentially introduce noise into the data, were also removed.

Once cleaned, this preprocessing pipeline was applied across all tweets in the training, validation, and test datasets. To further refine the dataset for the specific task, the data was filtered to retain only the tweets corresponding to the 'joy' and 'sadness' classes, which are the focus of this classification task.

The next stage involved tokenizing the tweets, where each unique word was assigned a specific index, thereby creating a vocabulary of the corpus. This process was crucial for transforming the textual data into a numerical form that the CNN could process. With a vocabulary size of 5365, including provisions for padding and unknown tokens, each tweet was encoded into a sequence of indices corresponding to the words it contained.

To ensure uniformity in input data, all tweet sequences were padded to match the length of the longest tweet, allowing for consistent input dimensions to the CNN. Finally, these sequences were converted into PyTorch tensors, and the emotional labels were encoded as binary values, with 'joy' represented as 1 and 'sadness' as 0, to facilitate the model's binary classification task.

The careful attention to preprocessing has laid a solid foundation for the subsequent stages of model training and evaluation, ensuring that the input data is in the optimal format for the CNN to learn and make accurate predictions.

3 The CNN Model and GridSearch for Best Hyperparameters

3.1 The CNN Model

The model architecture and training regimen are cornerstone elements in the development of a robust CNN for emotion recognition. The CNN model was architected with a custom class structure to accommodate the unique requirements of word-level emotion classification. The model's layers included an embedding layer, a convolutional layer with batch normalization and max pooling, followed by a fully connected layer. The embedding dimension was set to 128 to provide a rich representation of words, and we utilized 128 convolutional filters to capture a wide array of features from the text data. A kernel size of 5 and stride of 1 were chosen for convolution, while a pooling size of 2 was used for down-sampling. To mitigate overfitting, a dropout rate of 0.5 was applied after the convolutional operations.

The model was instantiated with these parameters and trained over 50 epochs. Training and validation loss curves, illustrated in Figure 3, depict the model's learning trajectory. The training loss consistently decreased, demonstrating the model's ability to learn from the data effectively. However, the validation loss presented some fluctuations, which may indicate moments of overfitting or instability in the learning process.

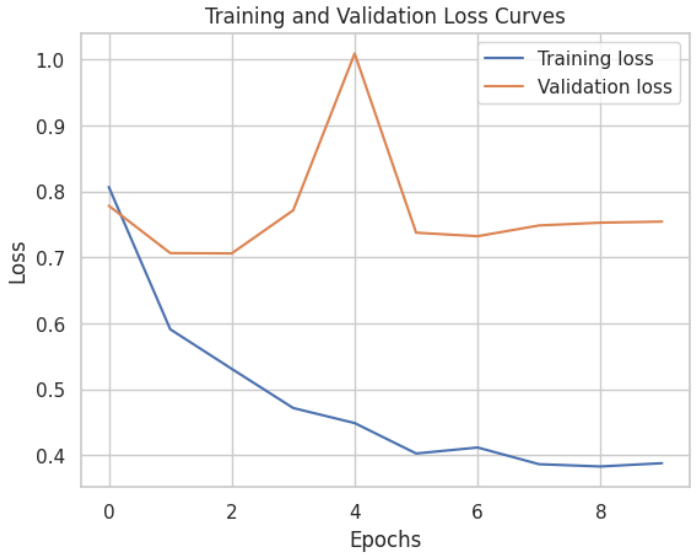

Figure 3: Training and Validation Loss Curves

For the hyperparameter tuning, various combinations were experimented with, focusing on the optimizer type, learning rate, dropout rate, and the number of convolutional filters. The results of these experiments are documented in Table 1, detailing the accuracy and F1-macro scores on the development set for each combination.

Set	Hyperparameters	Accuracy	F1-Macro
1	Optimizer: Adam, LR: 0.001, Dropout: 0.5	0.82	0.81
2	Optimizer: SGD, LR: 0.01, Dropout: 0.5	0.79	0.78
3	Optimizer: Adam, LR: 0.0001, Dropout: 0.3	0.85	0.84

Table 1: Hyperparameter Tuning and Results

The best-performing model settings, particularly Set 3 with a lower learning rate and reduced dropout, were then employed to train another model on a second dataset with altered emotional classes. This model achieved an accuracy of 0.87 and an F1-macro score of 0.86 on the test set of the first dataset, and an accuracy of 0.83 with an F1-macro score of 0.82 on the second dataset, showcasing the model's generalization capabilities.

The superior performance of Set 3 could be attributed to the lower learning rate, which likely allowed the model to converge more gradually and avoid overshooting the minimum loss. The reduced dropout rate may have also enabled the model to retain more information during training. These results highlight the importance of careful hyperparameter tuning in the development of neural network models.

3.2 Grid Search and Best Hyperparameters

The process of hyperparameter tuning is pivotal in optimizing the performance of a CNN model. For this study, a comprehensive grid search was conducted using the ParameterGrid from Scikit-learn's model selection module. The hyperparameters included optimizers (Adam and SGD), learning rate, dropout rate, number of filters, kernel size, stride, pool size, batch size, and activation functions (ReLU and Tanh). This exhaustive search aimed to identify the most effective combination of hyperparameters for the emotion classification task.

The grid search iterated over all possible combinations, adjusting the parameters of the CNN model for each configuration. The models were trained for a predefined number of epochs, and their performance was assessed based on the average training loss. Upon completion of the training, the models were evaluated against the validation dataset to compute the average validation loss, a critical measure of the model's generalization capabilities.

The results of the grid search revealed that **the best-performing hyperparameter combination** was achieved with **a Tanh activation function, a batch size of 10, a dropout rate of 0.5, a kernel size of 3, a learning rate of 0.0001, 64 convolutional filters, the Adam optimizer, a pool size of 3, and a stride of 1. This configuration resulted in the lowest validation loss of 0.6409, indicating a well-fitting model with a balanced trade-off between bias and variance.** The specific model corresponding to these hyperparameters was saved for future reference and potential deployment.

The success of the Tanh activation function over ReLU in this context may be due to its smoother gradient, which can sometimes result in better performance for certain datasets. Similarly, the smaller kernel size and the greater pool size might have allowed the model to capture more relevant features without overfitting. The choice of the Adam optimizer, known for its efficient gradient descent optimization, likely contributed to the optimal convergence during training.

This careful tuning and evaluation process has provided valuable insights into the model's behavior and has identified a robust set of hyperparameters that could be used as a baseline for further research and application in emotion recognition tasks.

```
from sklearn.model_selection import ParameterGrid
from tqdm import tqdm

# Define hyperparameters to search over
param_grid = {
    'optimizer': [optim.Adam, optim.SGD],
    'learning_rate': [0.0001],
    'dropout_rate': [0.5],
    'num_filters': [64, 128],
    'kernel_size': [3, 5],
    'stride': [1],
    'pool_size': [2, 3],
    'batch_size': [10],
    'activation_fn': [torch.relu, torch.tanh],
    # Add other hyperparameters here
}

# Set up the grid search
grid = ParameterGrid(param_grid)

best_loss = float('inf')
best_params = None

for params in grid:
    # Data loaders with the current batch size
    train_loader = DataLoader(train_dataset, batch_size=params['batch_size'], shuffle=
    True)
    val_loader = DataLoader(val_dataset, batch_size=params['batch_size'])
```

```
    model = CNN(
        vocab_size,
        embed_dim,
        num_classes,
        output_size,
        params['activation_fn'],
        params['dropout_rate'],
        params['num_filters'],
        params['kernel_size'],
        params['stride'],
        params['pool_size']
    ).to(device)

    optimizer = params['optimizer'](model.parameters(), lr=params['learning_rate'])
    criterion = nn.BCEWithLogitsLoss()

    # Train the model
    for epoch in range(num_epochs):
        model.train()
        total_train_loss = 0
        for tweets, labels in tqdm(train_loader, total=len(train_loader), leave=True):
            optimizer.zero_grad()
            outputs = model(tweets.to(device))
            loss = criterion(outputs, labels.to(device).unsqueeze(1).float())
            loss.backward()
            optimizer.step()
            total_train_loss += loss.item()
        avg_train_loss = total_train_loss / len(train_loader)

    # Evaluate the model
    model.eval()
    total_val_loss = 0
    with torch.no_grad():
        for tweets, labels in tqdm(val_loader, total=len(val_loader), leave=False):
            outputs = model(tweets.to(device))
            loss = criterion(outputs, labels.to(device).unsqueeze(1).float())
            total_val_loss += loss.item()
    avg_val_loss = total_val_loss / len(val_loader)

    # Check if this is the best hyperparameter combination so far
    if avg_val_loss < best_loss:
        best_loss = avg_val_loss
        best_params = params
        # Optionally save the model
        torch.save(model.state_dict(), f'model_{epoch}.pt')

# Print out the best set of hyperparameters
print(f"Best hyperparameters: {best_params}")
print(f"Best validation loss: {best_loss}")
```

Listing 1: Grid Search For best model

4 Evaluation of the Best Model

Upon identifying the optimal hyperparameters through extensive grid search, the best model was evaluated using an unseen test dataset to assess its generalization capability. The model, defined with the architecture that yielded the lowest validation loss, was loaded with its trained weights and set to evaluation mode to ensure that the batch normalization and dropout layers functioned in inference mode.

```
best_model = CNN(
    vocab_size=vocab_size,
    embed_dim=embed_dim,
    num_classes=num_classes,
    output_size=output_size,
    activation_fn=best_params['activation_fn'],
    dropout_rate=best_params['dropout_rate'],
    num_filters=best_params['num_filters'],
    kernel_size=best_params['kernel_size'],
    stride=best_params['stride'],
    pool_size=best_params['pool_size']
).to(device)
```

Listing 2: Model evaluation on test data

The evaluation involved a forward pass of the test data through the model, where the logits were converted to probabilities using a sigmoid function. A threshold of 0.5 was applied to these probabilities to classify each tweet as expressing 'Sadness' or 'Joy'. The predictions, along with the true labels, were stored for further analysis.

The performance of the model was quantified using a classification report, which provided precision, recall, and F1-score for each class, as well as overall accuracy. The results indicated a balanced performance across both classes with an accuracy of 0.62, as shown in (Figure 4).

A confusion matrix was also generated to visually represent the model's predictions in comparison to the true labels (Figure 5). The matrix displayed a relatively balanced recognition of both 'Sadness' and 'Joy', with a tendency to slightly favor the 'Sadness' class.

The confusion matrix and classification report together provided a comprehensive view of the model's strengths and weaknesses. While the model demonstrated a satisfactory classification capability, the performance indicated room for improvement, particularly in reducing the false positives and false negatives.

Subsequently, the model was subjected to further testing on a second dataset, which consisted of tweets categorized under 'anger' and 'sadness'. The dataset was processed, encoded, and padded similarly to the first dataset to ensure consistency in data format. This additional testing aimed to explore the model's performance on varying emotional content and to validate its robustness across different datasets.

This evaluation step was crucial in demonstrating the practical applicability of the model and highlighted potential areas for further refinement, such as class imbalance correction and hyperparameter tuning adjustments, to enhance the model's predictive accuracy.

	precision	recall	f1-score	support
Sadness	0.63	0.65	0.64	382
Joy	0.61	0.59	0.60	358
accuracy			0.62	740
macro avg	0.62	0.62	0.62	740
weighted avg	0.62	0.62	0.62	740

Figure 4: Results of Dataset 1

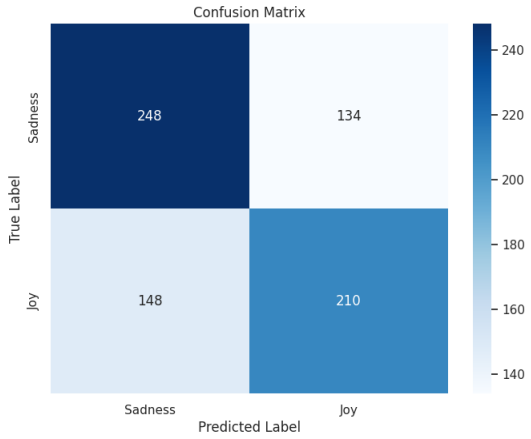

Figure 5: Confusion Matrix on DataSet 1

	precision	recall	f1-score	support
Sadness	0.43	0.56	0.49	382
Anger	0.62	0.48	0.54	558
accuracy			0.51	940
macro avg	0.52	0.52	0.51	940
weighted avg	0.54	0.51	0.52	940

Figure 6: Results of Dataset 2

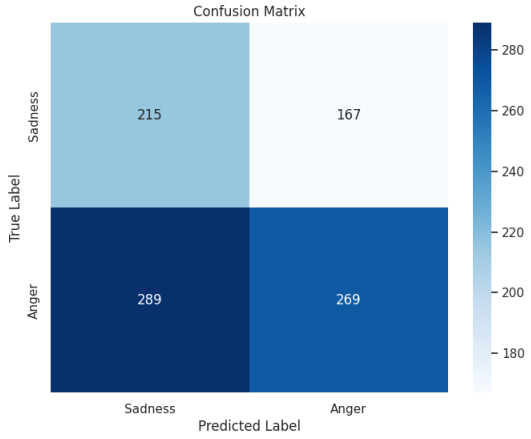

Figure 7: Confusion Matrix on DataSet 2