Algorithmic Analysis of Convolutional Neural Network for Text Classification (Emotions Dataset)

Do Viet Long - ICT 2440037

May 31, 2025

Abstract

This report provides a comprehensive algorithmic analysis and implementation review of a Convolutional Neural Network (CNN) designed for text classification, specifically emotion detection from textual data. The project utilizes an emotions dataset (likely the "Emotions Dataset for NLP" from Kaggle) and employs the TensorFlow/Keras framework for model development. The methodology encompasses several key stages: data loading and preprocessing, including text tokenization and sequence padding; data balancing by undersampling majority classes and removing minority classes; a multi-branch CNN architecture incorporating embedding layers, 1D convolutional layers with different filter configurations, global max pooling, and dense layers for classification; model compilation with the Adamax optimizer and categorical cross-entropy loss; and finally, model training and evaluation using metrics such as accuracy, precision, recall, and confusion matrices. The model reportedly achieves high accuracy (around 95%), indicating the effectiveness of the chosen CNN approach for this NLP task.

1 Introduction

Text classification is a fundamental task in Natural Language Processing (NLP) with applications ranging from sentiment analysis to topic categorization and emotion detection. This project aims to develop a deep learning model, specifically a Convolutional Neural Network (CNN), to classify text into predefined emotion categories (e.g., joy, sadness, anger, fear). CNNs, traditionally known for their success in computer vision, have also proven effective for text by capturing local contextual features (n-grams) through 1D convolutions over word embeddings.

The implementation is carried out using TensorFlow/Keras. This report will dissect the algorithmic components and their code realization, including:

- **Data Acquisition and Preprocessing**: Loading the text dataset, handling class labels, and preparing the text for a CNN model through tokenization and padding.
- **Data Balancing**: Addressing potential class imbalances in the dataset to ensure fair model training.
- **CNN Model Architecture**: Detailing the multi-branch CNN structure, including embedding layers, 1D convolutional layers, pooling mechanisms, and fully connected layers.
- **Training Procedure**: Outlining the model compilation, choice of optimizer and loss function, and the training process.
- **Evaluation Metrics**: Discussing how the model's performance is assessed.

2 Algorithmic Analysis and Source Code Implementation

2.1 Environment Setup and Utilized Libraries

The project leverages several Python libraries standard for NLP and deep learning tasks:

- **Pandas**: For loading and manipulating structured data, likely reading `train.txt`, `val.txt`, and `test.txt` files.
- **NumPy**: For numerical operations, especially array manipulations.
- **Matplotlib & Seaborn**: For data visualization, such as plotting label distributions and model performance metrics (e.g., confusion matrix, training history).
- **Scikit-learn** (`sklearn.metrics`, `sklearn.preprocessing.LabelEncoder`): For evaluating model performance (classification report, confusion matrix) and for encoding categorical labels into numerical format.
- **TensorFlow/Keras** (`tensorflow.keras.preprocessing.text.Tokenizer`, `tensorflow.keras.preprocessing.sequence.pad_sequences`, `tensorflow.keras.layers`, `tensorflow.keras.models`, `tensorflow.keras.optimizers`, `tensorflow.keras.metrics`, `keras.utils.to_categorical`): The core deep learning framework used for:
  - Text preprocessing (`Tokenizer`, `pad_sequences`).
  - Building the CNN model (various layers like `Embedding`, `Conv1D`, `GlobalMaxPooling1D`, `Dense`, `Dropout`, `BatchNormalization`, `Concatenate`).
  - Defining and compiling the model (`Model`, `Adamax` optimizer, `categorical_crossentropy` loss).
  - Utility functions (`to_categorical` for one-hot encoding labels).

2.2 Data Preparation: Algorithm and Implementation

Data Loading and Initial Exploration: The dataset is loaded from text files (`train`, `validation`, `test` sets), likely containing sentences and their corresponding emotion labels (e.g., `sadness`, `anger`, `love`, `surprise`, `fear`, `joy`). The code performs an initial exploration by displaying unique labels and their value counts, often visualized using a pie chart to identify class distribution.

Data Balancing: The notebook indicates an initial imbalance in the dataset. To address this, the implementation involves:

- **Removing Minority Classes:** Labels like `love` and `surprise` (which have the lowest counts) are removed from the dataset.
- **Undersampling Majority Classes:** For the remaining classes (`joy`, `sadness`, `fear`, `anger`), a fixed number of samples (e.g., 2200 for `joy` and `sadness`) or all available samples for smaller majority classes are taken to create a more balanced training set. This is done using `DataFrame.sample(n=..., random_state=...)`.

The sampled DataFrames for each class are concatenated, and the final DataFrame is shuffled. This balancing step is crucial for preventing the model from being biased towards over-represented classes. Similar balancing is applied to the validation and test sets.

Label Encoding: The categorical emotion labels (strings) are converted into numerical representations using `sklearn.preprocessing.LabelEncoder`. The `fit_transform` method is applied on the training labels, and `transform` is used on validation and test labels.

Text Tokenization and Sequencing:

- **Tokenizer Initialization:** `tensorflow.keras.preprocessing.text.Tokenizer(num_word` is initialized, where `max_words` (e.g., 10000) defines the maximum number of most frequent words to keep in the vocabulary.

- **Fitting Tokenizer:** The tokenizer is fitted on the training text data (`tokenizer.fit_on_texts(tr_text)`), which builds the word index (vocabulary).
- **Text to Sequences:** The texts (train, validation, test) are converted into sequences of integers (`tokenizer.texts_to_sequences(...)`), where each integer represents a word in the vocabulary.

Sequence Padding: Since CNNs require inputs of uniform length, the integer sequences are padded (or truncated) to a `maxlen` (e.g., 50) using `tensorflow.keras.preprocessing.sequence.pad_sequences`. This ensures all input sequences have the same length.

One-Hot Encoding Labels: The numerical labels are converted into a one-hot encoded format suitable for categorical cross-entropy loss using `keras.utils.to_categorical`. After balancing, the number of classes would be 4 (`joy`, `sadness`, `fear`, `anger`).

2.3 CNN Model Architecture: Implementation with Keras Functional API

The model employs a multi-branch CNN architecture, likely to capture features using different n-gram filter sizes or convolutional configurations simultaneously.

Hyperparameters:

- `max_words`: Size of the vocabulary (e.g., 10000).
- `max_len`: Length of input sequences (e.g., 50).
- `embedding_dim`: Dimensionality of the word embedding vectors (e.g., 32).

Embedding Layer: `Embedding(max_words, embedding_dim, input_length=max_len)` is the first layer in each branch. It maps each word index in the input sequence to a dense vector of `embedding_dim` dimensions. This layer learns word representations during training.

Convolutional Branches (e.g., Branch 1, Branch 2): The notebook snippet shows two identical branches, but in practice, these could have different `Conv1D` filter sizes (e.g., kernel sizes of 3, 4, 5 to capture tri-grams, quad-grams, etc.). Each branch has the following structure:

- **Input:** Takes the output of the Embedding layer.
- `Conv1D(filters=64, kernel_size=3, padding='same', activation='relu')`: A 1D convolutional layer with 64 filters and a kernel size of 3. `padding='same'` ensures the output sequence length is the same as the input. ReLU activation introduces non-linearity.
- `BatchNormalization()`: Normalizes the activations of the previous layer, which can help stabilize and accelerate training.
- `ReLU()`: An additional ReLU activation after batch normalization.
- `Dropout(0.5)`: A dropout layer that randomly sets 50% of input units to 0 during training, a regularization technique to prevent overfitting.
- `GlobalMaxPooling1D()`: Performs max pooling over the entire output of the convolutional layer for each filter. This reduces each feature map to a single number, effectively capturing the most important feature detected by that filter across the sequence.

Concatenation Layer: The outputs (feature vectors) from the `GlobalMaxPooling1D` layers of all branches are concatenated using `Concatenate()([branch1.output, branch2.output, ...])`. This combines the features learned by different convolutional configurations. For example, if two branches with 64 features each are used, the concatenated output will have 128 features.

Fully Connected Layers (Classifier Head):

- `Dense(128, activation='relu')(concatenated)`: A dense (fully connected) layer with 128 units and ReLU activation, processing the concatenated features.
- `Dropout(0.3)(hid_layer)`: Another dropout layer for regularization.
- `Dense(num_classes, activation='softmax')(dropout)`: The output layer with `num_classes` (e.g., 4 after balancing) units and softmax activation. Softmax outputs a probability distribution over the classes.

Model Definition: The model is created using the Keras Functional API: `Model(inputs=[branch1.input, branch2.input, ...], outputs=output_layer)`. The inputs would be a list of input tensors, one for each branch if they process the same input sequence in parallel. The snippet shows `[branch1.input, branch2.input]`, implying the same input sequence is fed to both branches.

2.4 Training Algorithm and Implementation

Model Compilation: Before training, the model is compiled with the following configuration:

- `optimizer='adamax'`: The Adamax optimizer is chosen, a variant of the Adam optimizer that is sometimes more stable for certain tasks.
- `loss='categorical_crossentropy'`: Suitable for multi-class classification with one-hot encoded labels.
- `metrics=['accuracy', Precision(), Recall()]`: Accuracy, precision, and recall are tracked during training and evaluation to monitor performance.

Model Training (`model.fit`): The model is trained using the `fit` method with the following setup:

- **Inputs:** `[tr_x, tr_x]` — since the model has two input layers for the two branches, both processing the same input sequence.
- **Outputs:** `tr_y` — the one-hot encoded training labels.
- **epochs:** Number of training iterations over the entire dataset (e.g., 25).
- **batch_size:** Number of samples per gradient update (e.g., 256).
- **validation_data:** `([val_x, val_x], val_y)` — used for evaluating the model on the validation set after each epoch.

The training history, including metrics such as loss, accuracy, precision, and recall for both the training and validation sets, is stored. This history can later be used for visualizing performance trends over epochs.

2.5 Evaluation and Result Visualization

Model Evaluation (`model.evaluate`): The trained model is evaluated on both the training set and the test set to obtain final performance metrics:

- `(loss, accuracy, precision, recall) = model.evaluate([ts_x, ts_x], ts_y)`

Result Visualization:

- **Training History Plots:** Using `matplotlib`, the following plots are generated:
  - Training loss vs. validation loss per epoch.
  - Training accuracy vs. validation accuracy per epoch.
  - Training precision vs. validation precision per epoch.
  - Training recall vs. validation recall per epoch.

Best epoch points (e.g., lowest validation loss, highest validation accuracy) are often highlighted on these plots to indicate the optimal training performance.

- **Confusion Matrix:**
  - Predictions are made on the test set: `y_pred = np.argmax(model.predict([ts_x, ts_x]), axis=1)`.
  - True labels `y_true` are obtained by applying `argmax` on `ts_y` or using the original encoded labels.
  - `sklearn.metrics.confusion_matrix(y_true, y_pred)` computes the confusion matrix.
  - `seaborn`'s `heatmap` is used to visualize the confusion matrix, with annotations showing counts and class labels.
- **Classification Report:**
  - `sklearn.metrics.classification_report(y_true, y_pred)` provides precision, recall, F1-score, and support for each class, offering a detailed breakdown of the model's classification performance.

2.6 Prediction Function and Model Saving

Model Saving:
The trained Keras model is saved to an HDF5 file using:

- `model.save('nlp.h5')`: Saves the complete model architecture, weights, and optimizer state.
- The tokenizer object is saved using `pickle`:
- `pickle.dump(tokenizer, tokenizer_file)`: This is essential as the tokenizer is required for preprocessing new text during inference to ensure consistency.

Prediction Function (`predict`): A dedicated function is defined to handle new text predictions. It operates as follows:

- Loads the saved Keras model from the HDF5 file.
- Loads the pickled tokenizer from disk.
- Preprocesses the input text using the loaded tokenizer:
  - Converts text to sequences using `texts_to_sequences`.
  - Pads the sequences using `pad_sequences` to match the model's expected input length.
- Uses `model.predict()` to obtain prediction probabilities.
- Maps the predicted probabilities to corresponding emotion labels.
- Visualizes the prediction probabilities for each emotion class using a horizontal bar chart created with `matplotlib`, providing an intuitive view of the model's confidence for each emotion.

3 Algorithmic Results and Model Performance

The notebook title and evaluation output suggest that the model achieves high accuracy, approximately 95% on the test set after balancing and removing some classes.

Key Performance Metrics:

- **Accuracy:** The primary metric, indicating the overall correctness of the model's classifications across all classes.
- **Precision & Recall:** These metrics provide a more detailed view of performance, especially on a per-class basis.
  - High precision indicates the model produces few false positives.
  - High recall indicates the model misses few true instances (few false negatives).

- **Loss Curves:** The training and validation loss curves ideally show both decreasing trends and convergence, with only a small gap between them, which suggests good generalization and minimal overfitting.

- **Confusion Matrix:** The confusion matrix highlights which emotions are well-distinguished by the model and which ones tend to be confused. This can inform further refinement or targeted improvements.

Model Insights:
The multi-branch CNN architecture, combined with learned word embeddings, enables the model to capture and leverage relevant features from the text for effective emotion classification. Notably, the data balancing process is a critical step that likely contributed significantly to achieving fair and high performance across the considered emotion classes, ensuring the model does not become biased toward over-represented categories.

4 Discussion of Algorithmic Choices and Code Implementation

Effectiveness of CNN for Text:
1D CNNs are effective for text classification because they can learn hierarchical features representing n-grams of varying lengths. The `Conv1D` layers act as feature detectors for local patterns within sequences, while `GlobalMaxPooling1D` selects the most salient features, summarizing each feature map into a single value.

Multi-Branch Architecture:
Using multiple convolutional branches (even if identical in the presented snippet, they could be varied with different kernel sizes) allows the model to learn different types of features or patterns at different scales simultaneously. Concatenating these diverse features provides a richer, more informative representation for the final classifier.

Embedding Layer:
The embedding layer is a crucial component, as it transforms sparse, high-dimensional word indices into lower-dimensional dense vectors that capture semantic relationships between words. Training this layer jointly with the rest of the network allows the embeddings to be specialized for the specific task of emotion detection.

Data Balancing:
The decision to remove minority classes (such as `love` and `surprise`) and undersample majority classes (`joy`, `sadness`) is a pragmatic approach to addressing class imbalance. While this reduces the dataset size and potentially discards some valuable information, it helps prevent the model from becoming overly biased towards frequent classes and often leads to improved per-class performance on the retained categories.

Hyperparameters:
Key hyperparameters such as `max_words`, `max_len`, `embedding_dim`, number of filters, kernel sizes, dropout rates, batch size, and number of epochs all play a critical role in the model's performance. The reported 95% accuracy suggests that the chosen parameters were well-suited for this task.

Limitations:

- **Loss of Information:** Removing classes and undersampling reduces the dataset and may discard nuanced patterns present in the original data.

- **Fixed Sequence Length:** Padding or truncating sequences to a fixed `max_len` means that very long texts lose information, while very short texts are artificially extended, which may not be optimal.

- **Contextual Understanding:** While CNNs capture local context effectively, they may struggle to capture long-range dependencies in text compared to architectures like Recurrent Neural Networks (RNNs) or Transformers.

Potential Future Work:

- **Advanced Architectures:** Exploring RNNs (such as LSTMs or GRUs), Transformers (like BERT), or hybrid CNN-RNN models for enhanced contextual understanding.
- **Pre-trained Embeddings:** Utilizing pre-trained word embeddings such as Word2Vec, GloVe, or FastText as initial weights for the embedding layer, which can be especially advantageous when working with smaller datasets.
- **Hyperparameter Optimization:** Conducting systematic searches (e.g., grid search, random search, or Bayesian optimization) for optimal hyperparameter settings.
- **Handling Imbalance Differently:** Investigating alternative techniques such as weighted loss functions or sophisticated oversampling methods (e.g., SMOTE for text data) instead of removing classes.
- **Attention Mechanisms:** Incorporating attention mechanisms into the CNN framework to allow the model to focus on the most relevant parts of the input text.

5 Conclusion

The project successfully demonstrates the application of a multi-branch Convolutional Neural Network using TensorFlow/Keras for emotion classification from text, achieving a high reported accuracy of 95%.

Key algorithmic steps include careful data preprocessing through tokenization and padding, the application of a crucial data balancing strategy, and the design of a well-structured CNN architecture that leverages word embeddings and 1D convolutions to extract meaningful features from textual input.

The implementation provides a clear and practical example of building an effective text classifier. Detailed evaluation using various metrics and visualizations confirms the model's proficiency and robustness. While there are certainly areas for future exploration — such as experimenting with more advanced architectures or employing sophisticated data handling techniques — the current approach serves as a strong and reliable baseline. It effectively highlights the capabilities of CNNs in natural language processing tasks when combined with appropriate and thoughtful data preparation.