Character Level Convolution Network for Text Classification

Project Id: 14

Team Name: Apriori

Gayatri Madduri (2019102043)

Kawshik Manikantan (2019111004)

Srividya Srigiri (2019102041)

Sai Akarsh (2019111017)

GitHub Repo Link:

```
https://github.com/KawshikManikantan/CREPE
```

Phase - 1 Progress:

Why Characters? Our Analysis:

- In Subword Algorithms, the increase in length of the input sequence in subword models will increase the computational complexity of the transformer which makes the model slower.
- Informal texts, variations in spellings are still challenging in subword models.
- Subword models which are limited to word-splitting operations are not a good approach in some cases like the languages with non-concatenative morphology and compounding morphology etc.
- In some algorithms, word splitting is based on whitespace or punctuation marks which is not suitable for languages that do not use space between words. These fixed pre-processing methods are not suggestable.
- Fixed vocabulary methods make us fixate with the same tokenizer and vocabulary.

- In some cases, infrequent vocabulary elements are not learned for good embedding.
- On the other hand, the character-based model will have good representations in the case of infrequent vocabulary using co-estimation of shared weights between words.

Baseline:

Bag-of-words

- Bag-of-words model is a simple way of extracting features from the text. Initially, we look for the vocabulary in the document and the occurrence of words in the documents needs to be scored. The scoring is done by counting the number of times each word appears in a document which we consider as the features of the model.
- For our model, we have considered the 50,000 most frequent words as the features. If a word other than these 50000 features is encountered, the word is assigned an all-zero feature vector.
- To perform the feature extraction step mentioned above, we use the function torch.utils.data.DataLoader. This function is helpful in performing the whole operation batch-wise and helps in working over large datasets such as dbpedia_14.
- The collate_fn parameter of the DataLoader function is employed to extract the features for the content in the training data. However, this requires tensor data, which in turn can only be obtained from a homogenous data of numbers.
- Hence, the 50000 most frequent words are extracted and the data is converted to indices. These indices are converted back to words in the collate function and the feature vectors are extracted using sklearn.feature_extraction.text.Countvectoriser().-
- These features are normalized using sklearn.preprocessing function.
- Logistic Regression is performed on the feature vectors thus formed to classify the data appropriately. The logistic regression model is built using nn.Linear. We build upon a basic feed-forward model to add useful utilities.

- Highly frequent words may not be domain-specific words. We solve this problem using TFIDF. In this approach, we re-scale the frequency of words by how often they appear in all documents known as inverse-document frequency.

Long-short term memory

This implementation uses a single-layer LSTM network based on word embeddings. The LSTM model is good at maintaining long and short-term memory which gives it a big advantage in text classification. We convert the sentences into a tensor and then feed it to the LSTM for predicting the class which is stored as a one-hot vector. This is used to train the model for many epochs. The results obtained are acceptable for such a simple setup and can easily be improved to give much better results.

- We chose the vector size for word embeddings as 25. This size is not too big for training and is not too small to lose sequential information.
- The sentences are tokenized into words which each are then embedded into a vector using a pre-trained word2vec model. We use padding during pre-processing since each sentence is of a different length.
- Then use the collate function for proper conversion to a invariable length 2-dimensional vector for each sentence and then store it in a pack sequence.
- Each column vector is sent in a single time stamp to get the final resultant hidden state vector for a sentence. Then that final vector is sent to a simple linear unit to reduce it into a vector whose dimension is equal to the number of classes. Then we apply softmax to get the class probability vector.
- We used a cross-entropy loss function to calculate the loss and then an Adam optimizer to update the weights from the calculated gradients.
- We send the sentence vectors and class vectors batch-wise during training and run it for multiple epochs.
- For testing or prediction, we do the same pre-processing and send it to the LSTM classifier. Then we apply argmax to the final softmax output to get the class of that sentence.

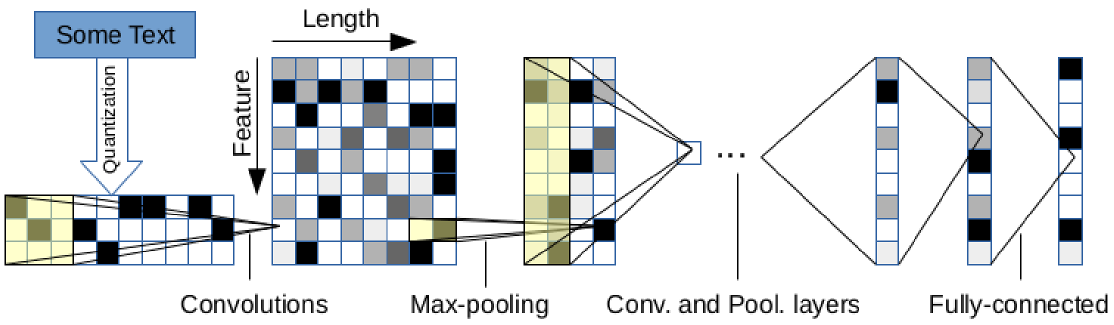
Paper Model:

Character-level CNN model

The model accepts a sequence of encoded characters as input and then quantizes that character sequence using one-hot encoding. The quantization order is backward which makes it easy in the case of a fully connected network. The model consists of 6 convolutional networks and 3 fully connected layers. The main components of the model are temporal convolution and temporal max-pooling which are defined as follows:

- Strided convolution : $h(y) = \sum_{x=1}^{k} f(x).g(y.d - x + c)$
- Max-pooling function : $h(y) = max_{x=1}^{k} g(y.d - x + c)$

  Where h_i and g_i are the input and output features respectively.



Graphical representation of the model

- The model was implemented in PyTorch framework on an NVIDIA 2080Ti GPU. PyTorch Lightning was used to have a modular approach and for its ease of logging.
- One hot-encoding was generated with a help of an alphabet dictionary.
- The entire model is implemented as a combination of two sequential models of convolution and a set of deep networks.
- A small change was made in the setup of the model, wherein the authors had decided to use a null vector for padding and for unknown characters. However, we have added a UNK token for such purposes and hence no row in the final embedding is null.
- The model was trained on the DBPedia dataset which consists of 560000 data points and creating the one-hot encoding for all the data points at once leads to

memory issues. Hence the conversion of categorical data to one-hot encoding was reserved to runtime.

- Results logged and graphed with the help of Lightning, Tensorboard, and torch metrics were analysed and the best model was selected.

Results:

Model	Accuracy	Precision	Recall	F1 score
Bag-of-words	90.32%	91%	90%	90%
LSTM	93%	94%	93%	93%
Character-level CNN	98%	98%	98%	98%

Preparation for the next phase:

Transformers vs CNNs:

- Temporal Sequence processing — Transformers tend to have a lot of advantages
- Transformers enable us to model long dependencies between input sequences and also supports parallelism. This includes an encoder and a decoder structure. Both the encoder and the decoder have self attention layers, linear layers and residual connection.
- Attention layers focus on the important parts of the data by fading away the unnecessary parts in the data and encode the relative position of different features.
- Self-attention is a weighted combination of all other word embeddings that is each component of a sequence stores the global information of the whole sequence.
- In case of CNNs to encode the relative positions of features we need large filters.

Final Model Architecture:

Canine:

Canine is a Transformer language model that operates directly on character sequences without any explicit tokenization. Input to this model is a sequence of uni-code characters. Canine model is the combination of down-sampling which reduces the length of the input sequence and a deep transformer stack that encodes the context of the text.

This model consists of 3 major components:

a) Vocabulary-free technique for text embedding.

b) Character level model includes down-sampling and up-sampling which make it more efficient.

c) Efficient means of performing masked language modelling.

Paper Architecture Description:

Given an input sequence of character embeddings $e \in \mathbb{R}^{n*d}$ with length n and dimensionality d: $Y_{seq} = UP(ENCODE(DOWN(e))$

Where $Y_{seq} \in \mathbb{R}^{n*d}$ is the final representation for sequence prediction task.

Pre-processing:

The input to the model is a set of unicode characters. In pre-processing, a sequence of characters is converted to code-point integer values by iterating over characters.

Character hash embedding:

The encoding of the codepoint integers is done by a hash embedding trick. Given a codepoint x_i, we will look for an embedding slice from each of B hash buckets each of which dimensionality d' = d/k such that hash function performs a lookup into its own embedding matrix.

$e_i = VecConcat(LOOKUP_k(H_k(x_i))$

Down-sampling:

In this step, we encode characters using a single layer block-wise local attention transformer, and then we do down-sampling using strided convolution to reduce the number of sequence positions.

$h_{init} = LocalTransformer_1(e)$

$h_{down} = StridedConv(h_{init}, r)$

Deep transformer stack:

After down-sampling, a deep transformer stack is applied to the down-sampled positions. This part of the model results in the new down-sampled representation.

$h'_{down} = Transformer_L(h_{down})$

$y_{cls} = [h'_{down}]_0$

Up-sampling:

We construct another representation that is useful in the prediction step by first concatenating the output of the original character transformer with the output of the deep transformer stack.

$h_{up} = Conv(VecConcat(h_{init}, h'_{down}), w)$

$y_{seq} = Transformer_1(h_{up})$

$y_{seq} \in \mathbb{R}^{n*d}$ is the final representation of the final sequence.

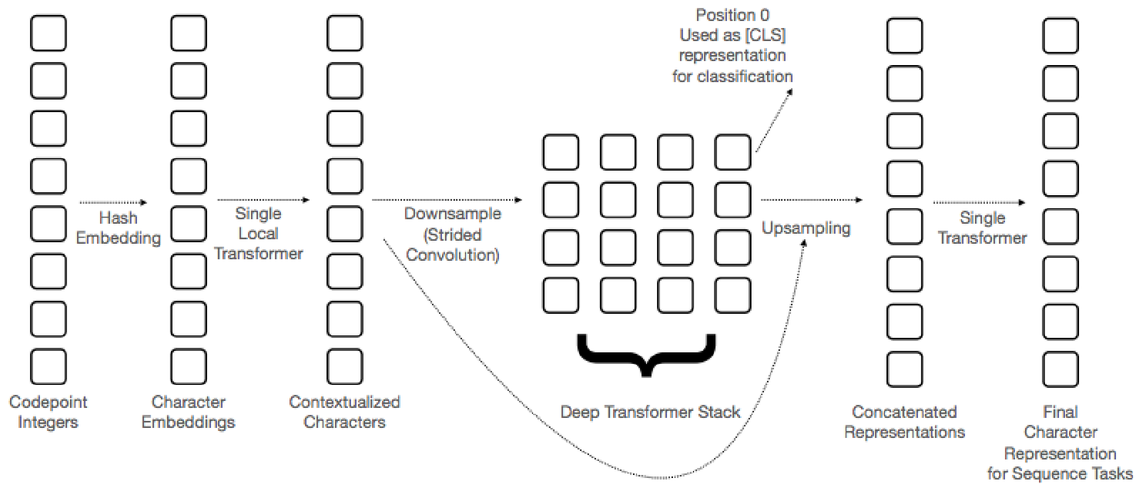


Figure 1: CANINE neural architecture.

Implementation Details:

Tokenization

Taking the input data and cleaning it by removing the unwanted characters in the data. Even though the actual model does not explicitly require a tokenizer as a part of preprocessing, we use the CANINE tokenizer to convert the sequence of characters into characters and then convert it to Uni-code code point.

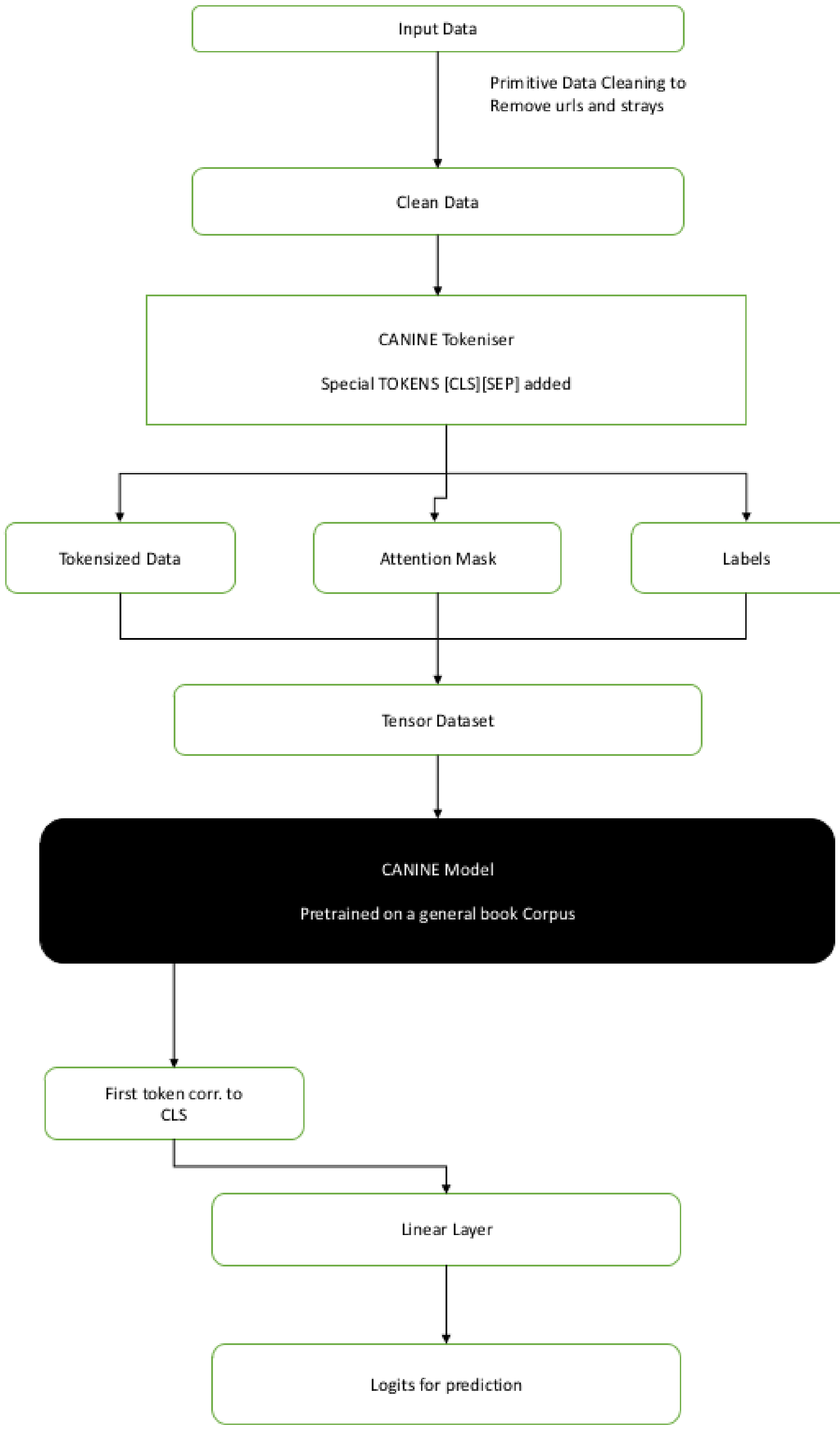
The Model

We divide the data set data into two parts. One for training the data and another for validation. Then the CANINE model is fine-tuned on the domain-specific data. This includes loading data, forward and backward propagation, and then updating the network parameters with an **AdamW** optimizer. For classification tasks, the model simply uses the zeroth element of the primary encoder, and a linear layer is added for classification tasks, and the predictions and errors are calculated. Parallelly, we validate the model to estimate the best model

Evaluation on test data

We test the finalized model with the test data and obtain the results. The hope is that transformers with their added advantage will perform much better than the given model.

The flow goes this way


Input Data
Primitive Data Cleaning to Remove urls and strays
Clean Data
CANINE Tokeniser
Special TOKENS [CLS][SEP] added
Tokensized Data
Attention Mask
Labels
Tensor Dataset
CANINE Model
Pretrained on a general book Corpus
First token corr. to CLS
Linear Layer
Logits for prediction