UCLA
UCLA Electronic Theses and Dissertations

Title
Empirical Study on the Effect of Zero-Padding in Text Classification with CNN

Permalink
https://escholarship.org/uc/item/7bc9c7jp

Author
Cheng, Henry

Publication Date
2020

Peer reviewed|Thesis/dissertation

eScholarship.org Powered by the California Digital Library
University of California

UNIVERSITY OF CALIFORNIA

Los Angeles

Empirical Study on the Effect of Zero-Padding in Text Classification with CNN

A thesis submitted in partial satisfaction

of the requirements for the degree

Master of Applied Statistics

by

Henry Jen-Hao Cheng

2020

© Copyright by
Henry Jen-Hao Cheng
2020

ABSTRACT OF THE THESIS

Empirical Study on the Effect
of Zero-Padding in Text Classification
with CNN

by

Henry Jen-Hao Cheng
Master of in Applied Statistics
University of California, Los Angeles, 2020
Professor Yingnian Wu, Chair

In CNN-based text classification tasks, where a CNN model is trained on top of pre-trained word vectors, padding is applied to ensure the input dimension is consistent, which is a requirement for CNN architecture. Traditionally, there are no set rules on how padding should be applied and padding is usually applied to the bottom of the text to achieve uniform length. Borrowing from the idea in computer vision, we show that there is no significant difference between applying zero-padding to the bottom of text embeddings and to both sides of the text embeddings.

The thesis of Henry Jen-Hao Cheng is approved.

Nicolas Christou

Rick Paik Schoenberg

Yingnian Wu, Committee Chair

University of California, Los Angeles

2020

TABLE OF CONTENTS

1 Introduction 1

2 Related Work 4

3 Model 7

3.1 Word Vectors 7

3.2 Padding 9

3.3 Multilayer Perceptron 9

3.4 CNN - simple 11

3.5 CNN 12

3.6 CNN - deep 13

4 Datasets and Experimental Setup 15

4.1 Dataset 17

4.1.1 AG News 17

4.1.2 Amazon Review Polarity 17

4.1.3 Amazon Review Full 18

4.1.4 DBPedia Ontology 18

4.1.5 Yelp Review Polarity 19

4.1.6 Yelp Review Full 19

4.2 Hyperparameters and Training 20

5 Results and Discussions 21

6 Conclusion **26**

7 Code **28**

7.1 Word2Vec Model Training 28

7.1.1 Configuration 28

7.1.2 Main 28

7.2 Model Training 32

7.2.1 Configuration 32

7.2.2 Main 33

7.2.3 Net Architecture Classes 39

7.2.4 Miscellaneous Functions 45

7.3 Model Evaluation 51

7.3.1 Main 51

References **55**

LIST OF FIGURES

1.1 zero-padding applied to image 3

3.1 A simple CBOW model with only one word in the context 8

3.2 Multilayer Perceptron Architecture 10

3.3 CNN -simple Architecture 11

3.4 CNN Architecture 12

3.5 CNN - deep Architecture and one unit of Convolutional Block 14

LIST OF TABLES

4.1 Summary Statistics of Dataset 15

4.2 Examples of text samples and their labels 16

5.1 Accuracy of 3-star reviews 24

5.2 Error of Model by Padding Method 25

CHAPTER 1

Introduction

The goal of text classification is to assign labels to a body of text, which is useful in applications such as question answering, sentiment analysis, document categorization, spam detection and many more. In deep-learning based text classification techniques, specifically the one using Convolutional neural network (CNN) trained on top of word embeddings such as CBOW, text are projected from 1-of-V (where V is the vocabulary size) onto a lower dimension space where words having close semantic meanings are close in euclidean or cosine distance. The word vectors are then used as input to train CNN, which performs the text classification tasks.

CNN uses convolving layers that apply filters to local features as its feature extraction process. The idea is popular in computer vision and has been shown to perform well in NLP tasks, including text classification. Kim [1] had shown that a simple CNN with one convolutional layer on top of word vectors can achieve excellent results, whereas Conneau et. al [2] showed that even better performance can be achieved by increasing depth of the CNN.

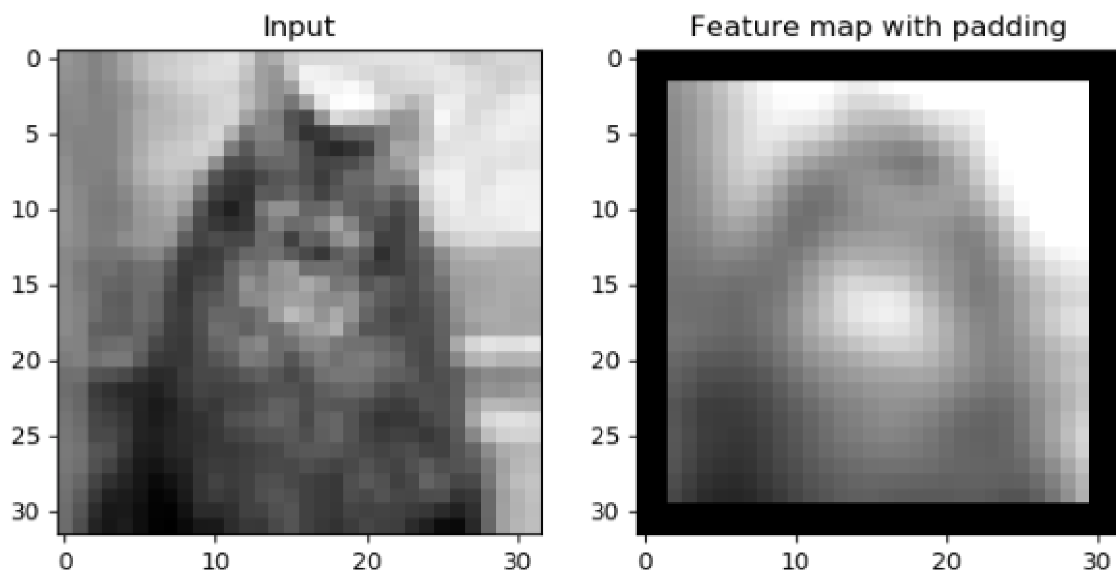
One problem arises with CNN, both in computer vision and NLP, is that the input dimension can be of different values. A CNN takes fixed dimension, n x m x c, as input and has to perform padding where vectors, often with value of zero, are appended to ensure the input image or text fits the input dimension n x m. In the case of computer vision, images are mostly in rectangular shape; padding is often done by appending zero-vectors symmetrically to the width and/or length dimension, as shown in figure 1.1. In NLP, word

embedding ensures the m dimension to be fulfilled as part of the embedding process and therefore only the n dimension needs to be enforced, where n dimension is the length of the text "block" or the input data. Padding for NLP has been traditionally done by adding zero-vectors to the bottom of the text to lengthen the word vectors until its length reaches n.

In this paper, we perform empirical study to compare symmetric padding and bottom-only padding in the context of CNN text classification. We trained word2vec embedding on 2 datasets over 3 neural network models. Our datasets include one with smaller paragraphs while having multiple classes and one with larger paragraphs and two classes. The neural network models consist of one multilayer perceptron model, which serves as a baseline, and two CNN models with various depths. In addition to symmetric and bottom-only padding, random padding where zero-vectors are inserted randomly into the text is also used to show symmetric and bottom-only padding has effect on classification accuracy. We then compare the model performances and found that both padding methods achieve similar performances.

Code is available at: https://github.com/henryjcheng/text_padding

Figure 1.1: zero-padding applied to image



CHAPTER 2

Related Work

Text classification that utilizes a two stage approach, feature extraction and classification, is a well-researched area. In Kim's work, a shallow network is introduced that uses word2vec pre-trained on 100 billion words from Google News for feature extraction, followed by shallow network consisting of a convolutional layer, a max pooling over time layer, and a fully connected layer with dropout and softmax for classification. The network adopts dropout on the penultimate layer and l_2-norms on the weight vectors to prevent overfitting. Kim uses 6 datasets to verify the results.

Kalchbrenner et. al [3] described a sentence modeling architecture called DCNN using CNN with dynamic k-max pooling, which is a global pooling operation over linear sequences. The DCNN includes 5 convolutional layers, starting with wide convolution over projected sentence matrix, dynamic k-max pooling, another layer of wide convolution, a folding layer followed by k-max pooling layer, and finally a fully connected layer. K-max pooling different from max pooling by returning k maximum values in the subsampling operations rather than a single maximum value. K can be dynamically chosen by making k a function of other aspects, such as sentence length. DCNN is evaluated on four different experiments, each varying by tasks performed, and achieved excellent result.

Conneau et. al introduced VDCNN architecture inspired by VGG and ResNet. VDCNN performs feature extraction at character level then passes the embedding through a deep neural architecture for text classification. The architecture starts with a mapping layer to map characters to their corresponding vectors, a convolutional layer followed by temporal

"convolutional blocks" where each block consists of a convolution layer, batch norm layer, and a ReLu activation, a k-max pooling layer, and three fully-connected layers. A memory reduction technique is also implemented by doubling feature space from two convolution blocks, then halve the temporal resolution, or output dimension, with a pooling layer. VDCNN is evaluated on 8 datasets, with 2 being a variation of other datasets that are also used in the evaluation.

Le et. al [4] showed that while a deep network performs well using character-level embedding, a shallow and wide network can achieve even better performance using word-level embedding. The paper compared Kim's shallow and wide network with DenseNet [5], which is introduced by Huang et. al for image classification task. DenseNet uses the same memory reduction technique as VDCNN, where temporal convolution doubles the feature space followed by a pooling layer that halves the output dimension. In addition, skip-connections is introduced to allow the gradient to back-propagate deeper in the network and eliminated issues such as vanishing gradients. With the potential issue of skip-connection negatively affect the information flow in the model, dense connection is implemented which allows to create direct connections from any layer to all subsequent layers. Le et. al compared shallow-and-wide and DenseNet architectures at character-level embedding and word-level embedding over 5 datasets.

Mahdi [6] compared image resizing with zero-padding and interpolation (scaling) where zero-padding is applied around the images to expand their dimension to fit AlexNet [7], a CNN based architecture with 5 convolutional layers followed by 3 fully-connected layers, and showed that zero-padding has no effect on classification accuracy but considerably reduced training time. The model performances are recorded on two datasets, Tiny Imagenet and Visual Domain Decathlon, with no pre-processing other than resizing with padding or scaling.

In summary, many architectures have been proposed on CNN-based text classification method where an embedding stage extracts features on character, word, or other forms of text units, and a classification stage where CNN is trained on top of the embedding

layer to perform classification tasks. In such modeling technique, padding is applied after embedding to keep the input dimension constant, which is a requirement for CNN. The difference is padding methods are not systematically studied and therefore, this study is devoted to exploring that.

CHAPTER 3

Model

We use word2vec embedding as feature extraction, padding to the maximum sentence length, and 3 separate neural network models for classification tasks.

3.1 Word Vectors

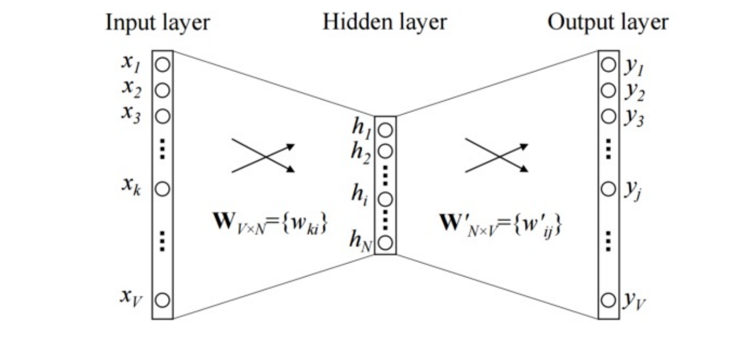
Word2Vec is a shallow, two-layer neural networks trained to convert words into vectors where words with close semantics are close in the cosine distance of the generated vector space. Training neural network using word vectors as inputs has been shown to greatly improve model performances.

Word2Vec starts with a one hot encoded vector of size V, where V is the vocabulary size of the input corpus and $x_i \in V, 1 \leq i \leq V$ represents the unique word or text unit in vocabulary. The model uses the one-hot encoded vectors as input layer to train a hidden layer with N neurons, where the value of N is specified (in our case, 50). Softmax is then applied to the output layer which also has dimension V.

In continuous bag-of-words (CBOW) architecture, a word is left out and the model tries to predict the target words using the surrounding words. The output layer of the model, in this case, represents the probability of each word being the left out word. A simple example of CBOW architecture is shown in figure 3.1. The Skip-Gram architecture uses the inverse of CBOW, where target words are used to predict surrounding context. In our experiment, we use CBOW to train our word vectors due to its training efficiency and not much drop in

accuracy. Once the training is completed, the weights used for input layer to hidden layer is extracted as a lookup table for embedding.

Figure 3.1: A simple CBOW model with only one word in the context



source: https://towardsdatascience.com/introduction-to-word-embedding-and-word2vec-652d0c2060fa

3.2 Padding

Padding here refers to concatenating vector(s) to the word vectors. The padding vector shares the same dimension as the word vectors, and can have specified or random values for each element. In our case, we use zero value for our padding vectors to perform bottom and symmetric padding.

For bottom padding, the operation takes the form of

$$x_l \oplus \nu_{n-l}$$

where $\oplus$ is the concatenating operation, ν is the zero vector, l is the text length, and n is the required CNN input dimension. The bottom padding operation essentially adds zero vectors to the bottom of the word embeddings until it reaches length n. For symmetric padding, the operation takes the form of

$$\nu_{\lfloor \frac{n-l}{2} \rfloor} \oplus x_l \oplus \nu_{\lceil \frac{n-l}{2} \rceil}$$

The padding operation adds zero vectors to the top and bottom of the word vectors until the entire embedding unit has length n.

3.3 Multilayer Perceptron

This is our baseline model for comparison against other CNN-based models. The multilayer perceptron model (MLP), as shown in figure 3.2, starts with the lookup table to map words to its corresponding embedding vectors, a flattening layer as the first fully connected layers, and two more fully connected layers with the last layer having dimension equals to class size. A max operation is applied to determine the class label.

Let $\boldsymbol{x}_b \in \mathbb{R}^{n \times m}$ be the b-th mini batch of word embedding vectors (with padding applied), then $\underset{n \times m \times 1}{\boldsymbol{x}_i} \in \mathbb{R}^m$ corresponds to the i-th input in $\boldsymbol{x}_b$; here n is the paragraph length in addition to padding, m is the dimension of embedding vectors, and 1 is the channel size.

With f defined as the flattening operation, the flattening transformation is therefore:

$$\underset{(n\times m\times 1)\times 1}{\boldsymbol{x'}_i} = f(\underset{n\times m\times 1}{\boldsymbol{x}_i})$$

The feedforward transformation can be represented as:

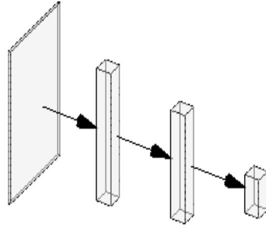
$$\boldsymbol{y}_i = g(\boldsymbol{w}^T \cdot \boldsymbol{x'}_i + b)$$

where g represents the ReLu activation function, $\underset{(n\times m\times 1)\times d}{w}$ is the weight matrix, d is the output dimension, and b is the bias term.

Feedforward transformation is carried out twice in our setting and call the result $\boldsymbol{y} \in \mathbb{R}^c$ with c representing the class size. A max function is applied to determine the class label:

$$\hat{y} = max\{\boldsymbol{y}\} \in \mathbb{R}$$

Figure 3.2: Multilayer Perceptron Architecture



3.4 CNN - simple

This is our vanilla version of CNN architecture which uses one layer of convolution with ReLu activation function, followed by a max pooling layer, followed by the multilayer perceptron model introduced earlier. The kernel size is 4 for both the convolutional layer and max pooling layer. The 3 feedforward layers in MLP has output dimensions of 120, 50, and c, respectively, where c represents the number of classes in the dataset.

Let $x_i \in \mathbb{R}^m$ be the m-dimensional word vector corresponding to the i-th word in a paragraph. Convolution operation applying a filter $\boldsymbol{w} \in \mathbb{R}^{hm}$ to h-grams $x_{i:i+h-1}$ can be represented as:

$$c_i = g(\boldsymbol{w} \cdot \boldsymbol{x}_{i:i+h-1} + b)$$

Here g is the non-linear ReLu function and $b \in \mathbb{R}$ is the bias term. This window is applied to each possible window of words in the paragraph $\{\boldsymbol{x}_{1:h}, \boldsymbol{x}_{2:h+1}, ..., \boldsymbol{x}_{n-h+1:n}\}$ to produce feature map $\boldsymbol{c} \in \mathbb{R}^{n-h+1}$. Max pooling is then applied to extract the maximum value from the feature map:

$$\hat{c} = max\{\boldsymbol{c}\}$$

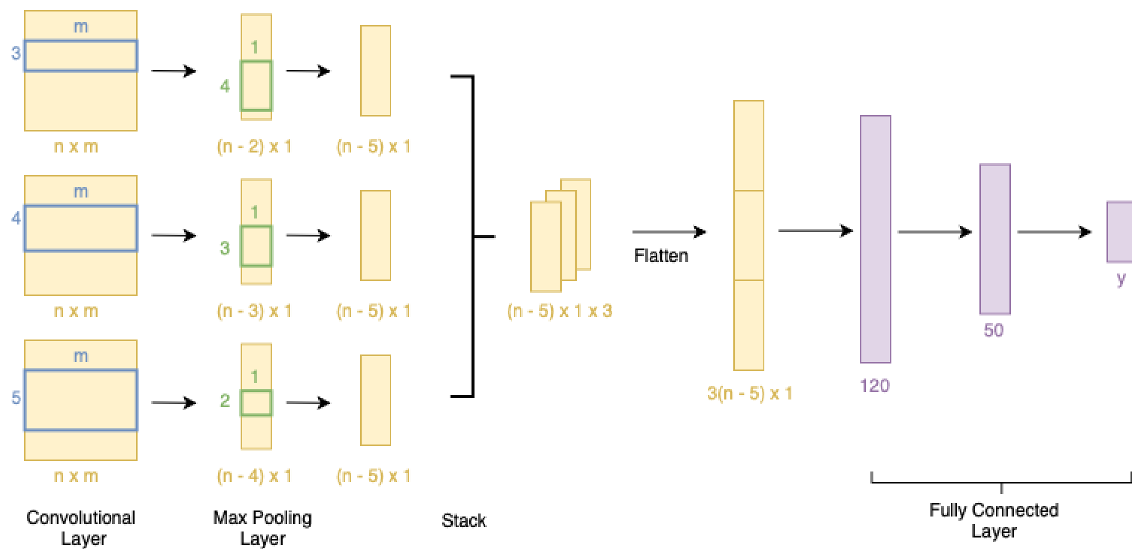
In figure 3.3, we use "4, Conv, 1" to denote convolutional layer with kernel size 4 and output dimension 1 and "fc(120, 50), ReLu" to denote fully connected layer with input dimension 120, output dimension 50, with ReLu activation function.

Figure 3.3: CNN -simple Architecture

3.5 CNN

This is our "shallow" CNN architecture that's similar to Kim's model. The model, as shown in figure 3.4, consists of 1 layer of convolution with 3 channels; the channels have kernel size of 3, 4, 5, respectively. ReLu activation is then applied, followed by max pooling with decreasing filter size for each channel to ensure the output dimensions are consistent. Here we replicate the input $\boldsymbol{x}$ 3 times and stack them to create $\boldsymbol{x}'$ with 3 identical channels. h or the filter size of $\{3, 4, 5\}$ are applied to channel 1, 2, 3 respectively. Max pooling layer is then applied with filter size $\{4, 3, 2\}$ to ensure the output dimension of $\hat{c}$ is consistent for each channel. $\hat{c}$ is then passed through the MLP model introduced earlier.

Figure 3.4: CNN Architecture

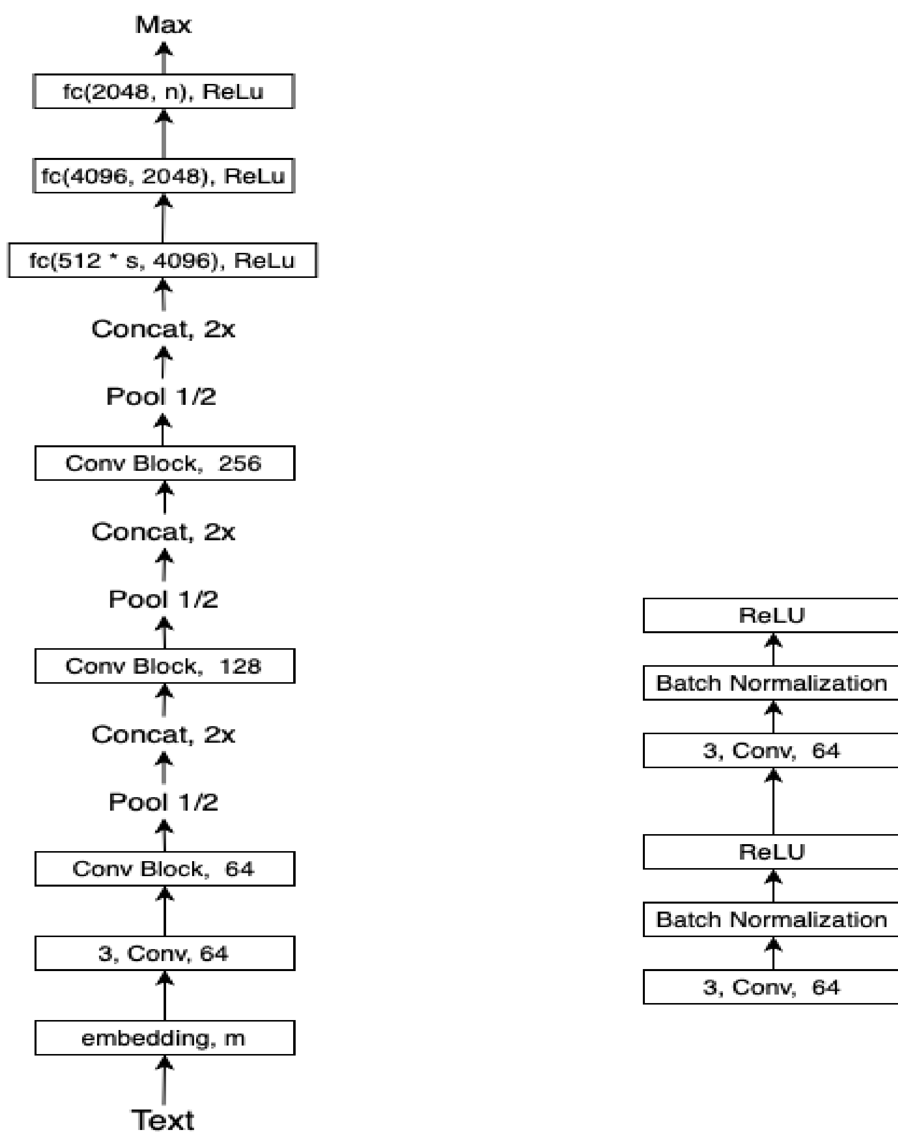


3.6 CNN - deep

Our “deep” model borrows the architecture from Conneau et. al, with the exception of using embedding at word-level instead of character-level, as well as omitting k-max pooling. The model starts with an embedding layer that maps words to their corresponding embedding vectors, followed by a convolutional layer with dimension of 64, followed by 3 convolutional blocks, and finally, the multilayer perceptron introduced in previous paragraphs. The convolutional block is the exact replica in VDCNN with two convolutional layers, each with batch norm and ReLu applied, and a pooling layer. The memory reduction trick is also applied in our model.

In figure 3.4, we use ”Conv Block, 64” to signify convolutional block with output dimension of 64, ”3, Conv, 64” to signify a convolutional layer with kernel size of 3 and output dimension of 64, and ”fc(4096, 2048)” to signify fully-connected layer with input dimension of 4096 and output dimension of 2048.

Figure 3.5: CNN - deep Architecture and one unit of Convolutional Block



CHAPTER 4

Datasets and Experimental Setup

We use six datasets for our experiment. The summary statistics for the dataset is in table 4.1.

For datasets with large text lengths, such as Amazon Review and Yelp Review, 50000 records are sampled, with each class having similar proportions. The sampling was required due to limited RAM. The word2vec embedding, however, are all trained using the entire dataset.

Table 4.1: Summary Statistics of Dataset

Dataset	**Class Size**	**Avg. Text Length**	**Train Set Size**	**Vocab Size**	**Test Set Size**
AG News	4	193	120000	104195	7600
Amazon Review Polarity	2	84	3600000	308293	400000
Amazon Review Full	5	86	3000000	278564	650000
DBpedia Ontology	14	48	560000	140361	70000
Yelp Review Polarity	2	726	560000	108236	38000
Yelp Review Full	5	153	650000	119256	50000

Table 4.2: Examples of text samples and their labels

Dataset	Class	Sample
AG News	1	Reuters - Venezuelans turned out early and in large numbers on Sunday to vote in a historic referendum that will either remove left-wing President [...]
Amazon Polarity	2	I'm reading a lot of reviews saying that this is the best 'game soundtrack' and I figured that I'd write a review to disagree a bit. [...]
Amazon Full	5	I hope a lot of people hear this cd. We need more strong and positive vibes like this. Great vocals, fresh tunes, cross-cultural happiness. Her blues is from the gut. [...]
DBpedia Ontology	1	Schwan-STABILO is a German maker of pens for writing colouring and cosmetics as well as markers and highlighters for office use. It is the world's largest manufacturer of highlighter pens Stabilo Boss.
Yelp Polarity	1	I'm writing this review to give you a heads up before you see this Doctor. The office staff and administration are very unprofessional. I left a message with multiple people regarding my bill [...]
Yelp Full	4	Got a letter in the mail last week that said Dr. Goldberg is moving to Arizona to take a new position there in June. He will be missed very much. [...]

4.1 Dataset

4.1.1 AG News

News articles in the 4 largest classes (world, sports, business, sci/tech) on AG's corpus of news articles on the web. Each class is equally represented in the training and testing dataset. Classification involves identifying the class for each article.

AG News are medium in text lengths relative to other datasets, which has the advantage of giving enough information without occupying significant memory resources. Distinctive words also exists in each category, which is extremely helpful in categorization tasks. For example, country-specific words such as "Venezuelans" or "S.Koreans" are present in the "world" category and sports-specific words such as "freestyle" or "Olympics" are present in the "sports" category.

Conneau et. al achieved error of 8.67 using VDCNN architecture using character-level embedding. Given that we are restricted in computing power and performing embedding at word-level, we should expect slightly worse performance from all our models.

4.1.2 Amazon Review Polarity

34,686,770 Amazon reviews from 6,643,669 users on 2,441,053 products, from the Stanford Network Analysis Project (SNAP). This subset contains 1,800,000 training samples and 200,000 testing samples in each polarity sentiment.

Within the "positive" class, dataset contains words such as "intrigued", "awesome", or "excellent". The "negative" class tends to have words such as "horrible". The challenge in this dataset is that sentiments may not be expressed in only one words, which word-level embedding will have a hard time capturing. For example, "stay away from" has a negative connotation but since embedding is done separately for each word, when none of the words in itself is particularly negative, the connotation will not be captured. Other issues such as

irony or contexts are also not captured by word-level embedding method.

Conneau et. al achieved error of 4.28 for this particular task. Given that we had to randomly sample the dataset so the training process fits into memory, we are expecting a much worse performance than what Conneau et. al was able to achieve.

4.1.3 Amazon Review Full

34,686,770 Amazon reviews from 6,643,669 users on 2,441,053 products, from the Stanford Network Analysis Project (SNAP). This full dataset contains 600,000 training samples and 130,000 testing samples in each class. Each class represents the "stars" the product received for the review, with "starts" ranging from 1 star to 5 stars.

Amazon Review Full dataset contains the same challenges the polarity dataset presents. In addition, the sentiment level also needs to be captured; that is, the model needs to be able to distinguish between a 1-star review and a 2-star review. The model also needs to capture "neutral" reviews, or the 3-star ones.

Conneau et. al achieved error of 37. Given that we had to randomly sample the dataset so the training process fits into memory, we are again expecting a much worse performance. In fact, since the training dataset for each class is much smaller, chances are our model won't be able to capture the relationships well. We may therefore observe similar performances across all models as well as all padding methods.

4.1.4 DBPedia Ontology

40,000 training samples and 5,000 testing samples from 14 non-overlapping classes from DBpedia 2014. Each class represents an "infobox" on Wikipedia, which can be categories such as book, music, history, or food and drinks, ...etc. The infobox can be viewed as a short summary for the subject on the Wikipedia page.

DBPedia Ontology presents a different type of challenge, which is a larger number of

classes. The classes are also general concepts that can contain a large variety of texts. However, since class-specific words are more of less present in each class, our word-level embedding should be expected to perform well on this dataset.

Conneau et. al achieved error of 1.29. We should expect our models to have similar performances.

4.1.5 Yelp Review Polarity

1,569,264 samples from the Yelp Dataset Challenge 2015. This subset has 280,000 training samples and 19,000 test samples in each polarity.

Yelp Review Polarity is another sentiment analysis task that has challenges similar to Amazon Review Polarity. In addition to the semantic challenges, Yelp reviews are in its nature lengthier than Amazon product reviews since Yelp reviews tend to be describing scenarios that had happened at restaurants or stores.

Conneau et. al achieved error of 4.28. With word-embedding having trouble capturing sentiments, we should expect much worse performances.

4.1.6 Yelp Review Full

1,569,264 samples from the Yelp Dataset Challenge 2015. This full dataset has 130,000 training samples and 10,000 testing samples in each star.

Yelp Review Full presents similar challenges to Amazon Review Full. In addition, Yelp reviews are much longer in length, which contains more information but also consumes more memory in training.

Conneau et. al achieved error of 35.28. We are expecting much worse performances since a sub-sample is used to fit data into memory. We're also expecting model to fail in capturing relationship between text and stars; therefore, all of our models should have similar performances across different padding methods.

4.2 Hyperparameters and Training

For all our datasets, we use: rectified linear units (ReLu) as activation function, features map of 50 to start, mini-batch size of 32, and 50 epochs with early stopping. The models are trained to minimize cross-entropy loss using stochastic gradient descent with learning rate of .001 and momentum of 0.9.

Our Word2Vec has dimension of 50 and uses the continuous bag-of-words (CBOW) architecture. The parameters are chosen to reduce training time and memory space requirement. Both the context window and subsampling frequency are set to 5. Words that are not in the vocabulary of the custom trained Word2Vec model are dropped.

CHAPTER 5

Results and Discussions

The result of our models with different padding methods are listed in table 5.2. As expected, padding by inserting random zero-vectors has the highest error among the three methods. **In general, we did not notice much difference in accuracy between bottom padding and symmetric padding.** However, when the CNN model is deep, bottom-padding tends to outperform both-side padding (5 out of 6 in our experiment).

We also observed that a simple multilayer perceptron achieves similar result as the relative more complicated CNN models. In one instance, DBPedia Ontology, multilayer perceptron model even outperforms our CNN model and only slightly worse than the CNN - deep model. This is different from Conneau et. al's conclusion where better performance can be achieved by adding layers to neural network model.

From our experiment, we also made the following discoveries:

1. word2vec and CNN is a good feature learner

In many cases, we were able to make accurate classifications even when texts has semantics relating to multiple classes. One example in our AG News dataset is as follows:

```
AP - If Hurricane Charley had struck three years ago,
President Bush's tour through the wreckage of this
coastal city would have been just the sort of post-disaster
visit that other presidents have made to the scenes of
storms, earthquakes, floods and fires.
```

The correct class for this article is ”world”, but the article is not too far from the ”science” category. In fact, 2 out of our 8 models (excluding random padding models) predicted ”science” category, while 5 models correctly identified this article as ”world” category. In addition to having science-related terms, this article also lacks the more common world-related words such as specific country names. Yet, most of our CNN models were able to pick up the right category.

However, the model can struggle when not enough information is given. In the following example,

```
Descriptions of urban afflictions and images of giant mosquitoes and
cockroaches to convey a sense of how Houston is nevertheless
beloved by many residents.
```

all but one model struggled to identify this as ”business” category. There is an even spread between ”science” and ”sports” for the incorrect answers. This may be due to the fact that words such as ”mosquitoes” and ”cockroaches” are present for the ”science” category, as well as the word ”Houston” for the ”sports” category.

2. Neutral Feeling is hard to be captured

In Yelp and Amazon review dataset, we tried to predict the number of stars associated with the review. Among the different star counts, 3-star is consider to be having a neutral feeling towards the business or the product. All of our models struggled to learn the patterns for neutral reviews. In fact, after inspecting, we believe human would also have a hard time identifying 3-star reviews.

In the following example in Yelp Review Full dataset,

```
not thrilling, not disappointing.  RB is average across the board.
Try the Hazelnut Crusted Chicken.
```

only 1 out of 8 model classified it as a 3-star. Most model predicted 4-star for this review.

A 4-star seems to be reasonable since one can argue the overall tone is slightly positive.

In another example,

`The Korean tacos are not half bad, and the kid really likes the burgers.`

2 out of 8 models correctly identified this as a 3-star, while the rest believe the review is either a 4-star or 5-star. 4 to 5 stars seems to be reasonable considering the phrase "really likes" exists.

The above two examples are related to the difficulty of quantifying satisfaction; that is, certain individual may view an experience as neutral when other's view it as positive. Another type of challenge in identifying neutral review is when both positive and negative experiences exist and cancel each other out, leaving the reviewer feeling neutral. For example,

`Drink prices are high but the dancing is fun!`

Our models assigned 4 to 5 stars to this review when the reviewer felt neutral about the experiences. This review is, again, hard to be determined even by human.

Table 5.1 shows the accuracy of predicting 3-star reviews. We can see the performance is low across all models and padding methods.

3. Negating words and sarcasms are hard to capture

Negating words such as "not" turns the sentiment of a word from positive to negative and vice versa. Sarcasm also carries the same effect but done so in an even more subtle way. Our models had a hard learning and identifying the correct sentiment when negating words or sarcasm is present in the text.

In the following example,

```
Overpriced, salty and overrated!!!
Why this place is so popular I will never understand.
```

Table 5.1: Accuracy of 3-star reviews

Type		**Yelp**	**Amazon**
Overall		2.28%	1.77%
By model	MP	1.58%	0.85%
	CNN - simple	1.27%	1.01%
	CNN	4.28%	1.38%
	CNN - deep	1.99%	3.84%
By padding method	bottom	2.27%	2.09%
	symmetric	2.29%	1.45%

the sentiment is clearly negative, but only 1 of our models correctly identify the sentiment. It is likely that the models saw the phrase "this place is so popular" and classified the review as positive, despite the reviewer's intent was to be sarcastic.

Table 5.2: Error of Model by Padding Method

Dataset	**Model**	**Bottom**	**Symmetric**	**Random**
AG News	Multilayer Perceptron	15.34	**14.93**	19.3
avg. text length 193	CNN - simple	32.36	**29.99**	31.17
vocab size 104195	CNN	15.33	**14.76**	19.59
class size 4	CNN - deep	**13.32**	20.58	19.62
Amazon Review Polarity	Multilayer Perceptron	**23.60**	24.12	27.71
avg. text length 84	CNN - simple	**22.76**	24.02	30.18
vocab size 308293	CNN	**19.15**	19.37	19.60
class size 2	CNN - deep	**24.54**	26.58	29.47
Amazon Review Full	Multilayer Perceptron	**65.73**	66.51	68.60
avg. text length 86	CNN - simple	68.10	**67.11**	72.91
vocab size 278564	CNN	63.90	**63.59**	64.68
class size 5	CNN - deep	**64.29**	79	71.14
BDPedia Ontology	Multilayer Perceptron	**7.11**	10.07	17.30
avg. text length 48	CNN - simple	42.36	41.55	**41.21**
vocab size 140361	CNN	**13.82**	14.37	16.34
class size 14	CNN - deep	**6.17**	15.78	18.69
Yelp Review Polarity	Multilayer Perceptron	**14.94**	15.41	16.31
avg. text length 726	CNN - simple	**13.18**	**13.18**	22.26
vocab size 108236	CNN	9.61	**9.58**	10.69
class size 2	CNN - deep	9.93	**9.47**	10.14
Yelp Review Full	Multilayer Perceptron	**58.42**	58.77	62.16
avg. text length 153	CNN - simple	**64.16**	65.26	69.94
vocab size 119256	CNN	54.87	**55.79**	56.32
class size 5	CNN - deep	**53.03**	58.87	72.60

CHAPTER 6

Conclusion

We used 6 different datasets, each with various text length, vocabulary size, as well as class size, to train their own customized word2vec embedding model, then use the embedding vectors as input to train 4 neural network model, each model having different complexity and depth. The neural network models are all trained using the same hyperparameters, except epoch which is adjusted by early-stopping to prevent overfitting.

We have shown that for CNN-based text classification, where traditionally padding is applied to the bottom of the text, there is no significant difference in applying zero-padding to the bottom of the text versus padding symmetrically. This is expected because in image recognition, zero-padding was introduced to bring minimal effect while addressing the dimension difference issue. While the convolutional outputs may be different for each padding method, the max pooling layer ensures that the feature extracted to be similar if not the same. That said, we did observe that if the model is deep, bottom padding seems to slightly outperform symmetric padding. This is more likely to due to randomness or our setup unintentionally favoring bottom-padding method. Further study should be conducted to validate if such effect exists by comparing the performances on more datasets.

In our experiment, we used the same hyperparameters and training criteria across all models. In the future, to address the issue of our setup favoring a certain type of model-padding method combination, we should train each combination to its best ability (such as achieving testing error less than a chosen value) then compare their performances. We also failed to train model that captured the relationship between words and neutral responses, as

well as between sarcasms and negative semantics; thus, we were not able to draw conclusion from those datasets. Similar experiment should be carried out with models properly trained to identify neutral responses or sarcasms and compare whether padding methods have effect in those types of text classification tasks.

Lastly, we focused on the padding methods for pre-trained word embedding on top of convolutional neural network. Future research can be done to explore if the same conclusion holds for other types of architecture that have word embedding component such as LSTM or transformers.

CHAPTER 7

Code

7.1 Word2Vec Model Training

7.1.1 Configuration

```
# Config file for train_w2v.py

[PATH]
data_path = ../../data/yelp_review_polarity/train.csv
model_save_path = ../../model/w2v

[MODEL]
dataset = yelp_review_polarity
model_name = yelp_review_polarity.model
embedding_dimension = 50
min_frequency = 5
```

7.1.2 Main

```
import os
import sys
import time
```

```
import pandas as pd
import multiprocessing
import configparser
from gensim.models import Word2Vec
from nltk.tokenize import word_tokenize
## first time uinsg nltk, uncomment the following 2 lines
# import nltk
# nltk.download('punkt')

def train_w2v(df, emb_dim, min_count):
    """
    This function is the main function that trains
    word2vec model from given text.
    The column containing text needs to
    have column name as 'text', eg. df['text']

    df: pandas DataFrame comtaining at least
    one column named 'text'
    emb_dim: embedding dimension, dimension
    of the word2vec model
    min_count: minimum frequency count of
    word in the word2vec model
    """
    print('Start training word2vec...')
    time0 = time.time()

    # tokenize
```

```
    print('\tTokenization...')
    df['text_token'] = df['text'].apply(lambda x: word_tokenize(x))

    # train model
    print('\tTrain word2vec model...')
    print(f'\t\t- dimension: {emb_dim}')
    print(f'\t\t- min_count: {min_count}')
    w2v = Word2Vec(df['text_token'].tolist(),
                   size=emb_dim,
                   window=5,
                   min_count=min_count,
                   negative=15,
                   iter=10,
                   workers=multiprocessing.cpu_count())

    time_diff = round(time.time() - time0, 2)
    print(f'Training complete.    Time elapsed: {time_diff}')

    return w2v



if __name__ == "__main__":
    config = configparser.ConfigParser()
    config.read('w2v.cfg')

    # setting up parameters
```

```
data_path = config['PATH']['data_path']
model_save_path = config['PATH']['model_save_path']

dataset = config['MODEL']['dataset']
model_name = config['MODEL']['model_name']
emb_dim = int(config['MODEL']['embedding_dimension'])
min_freq = int(config['MODEL']['min_frequency'])

# preprocessing
if dataset == 'ag_news':
    df = pd.read_csv(data_path)
    df_text = df[['Description']]\
                                .reset_index(drop=True)\
                        .rename(columns={'Description':'text'})
if dataset == 'yelp_review_polarity':
    df = pd.read_csv(data_path, header=0,
                                          names=['label', 'text'])
    df_text = df[['text']].reset_index(drop=True)
else:
    print(f'Dataset: {dataset} not recognized.')

# train model
w2v = train_w2v(df_text, emb_dim, min_freq)

# save trained model
w2v.save(os.path.join(model_save_path, model_name))
```

7.2 Model Training

7.2.1 Configuration

```
[PATH]
train_data_path = ../data/ag_news/train.csv
test_data_path = ../data/ag_news/test.csv
w2v_path = ../model/w2v/ag_news.model
model_save_path = ../model/cnn/ag_news
model_name = MP_bottom_test
dataset = ag_news

[MODEL_PARAMETERS]
# sample:          if True, sample 5000 records
# model_type:      takes value: MP, CNN, CNN_kim, CNN_deep
# emb_dim:         embedding dimesnion for word2vec
# pad_method:      padding method, takes value:
#                               bottom, bothside, random

sample = False
model_type = MP
emb_dim = 50
pad_method = bottom
batch_size = 32
shuffle = True
epoch = 5
```

7.2.2 Main

```
"""
This module contains code to train model.
The module takes input from model.cfg file.
"""
import os
import time
import random
import pandas as pd
import configparser
from nltk.tokenize import word_tokenize
from gensim.models import Word2Vec

import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import TensorDataset, DataLoader

from utility import zero_padding, model_loader, vocab_clean_up

## 0. setting up parameter
config = configparser.ConfigParser()
config.read('model.cfg')

## PATH
data_path = config['PATH']['train_data_path']
w2v_path = config['PATH']['w2v_path']
```

```
model_save_path = config['PATH']['model_save_path']
model_name = config['PATH']['model_name']
dataset = config['PATH']['dataset']

## MODEL_PARAMETERS
sample = config['MODEL_PARAMETERS'].getboolean('sample')
model_type = config['MODEL_PARAMETERS']['model_type']
emb_dim = int(config['MODEL_PARAMETERS']['emb_dim'])
pad_method = config['MODEL_PARAMETERS']['pad_method']

batch_size = int(config['MODEL_PARAMETERS']['batch_size'])
shuffle = config['MODEL_PARAMETERS'].getboolean('shuffle')
epoch = int(config['MODEL_PARAMETERS']['epoch'])

## 1. load dataset
if sample:
    nrows = 5000
else:
    nrows = None

if dataset == 'ag_news':
    df = pd.read_csv(data_path, nrows=nrows)
    df['Class Index'] = df['Class Index'].replace(4, 0)
    df = df.rename(columns={'Class Index':'label'})
    df['text_token'] = df['Description'\
                    .apply(lambda x: word_tokenize(x))
elif dataset == 'yelp_review_polarity':
```

```
    nrows=50000     # so it fits into 32Gb RAM
    df = pd.read_csv(data_path, nrows=nrows,
        names=['label', 'text'])
    df['label'] = df['label'].replace(2, 0)
    df['text_token'] = df['text'\
        .apply(lambda x: word_tokenize(x))
else:
    print(f'Dataset: {dataset} is not recognized.')

## 2. apply tokenization and embedding
w2v = Word2Vec.load(w2v_path)
df['text_token'] = df['text_token']\
        .apply(lambda x: vocab_clean_up(x, w2v))

# clean up rows with empty embedding
df['text_length'] = df['text_token'].apply(lambda x: len(x))
df = df[df['text_length'] > 0].reset_index(drop=True)
# reduce footprint
if dataset == 'ag_news':
    df = df
elif dataset == 'yelp_review_polarity':
    df = df[['label', 'text_token', 'text_length']]\
        .reset_index(drop=True)
else:
    print(f'Dataset: {dataset} is not recognized.')

df['embedding'] = df['text_token'].apply(lambda x: w2v[x])
```

```
## 3. zero pad to max length
if dataset == 'ag_news':
    max_length = 245
elif dataset == 'yelp_review_polarity':
    max_length = 1500
else:
    print(f'Dataset: {dataset} is not recognized.')

print(f'sample is {sample},
        training size: {df.shape[0]},    max_length: {max_length}')

df['embedding'] = df['embedding']\
        .apply(lambda x: zero_padding(x,
                max_length, emb_dim, pad_method))

## 4. load nn architecture
net = model_loader(model_type, dataset)

# define loss function and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(net.parameters(), lr=0.001, momentum=0.9)

if continuous_train:
    print('\nTraining from checkpoint.')
    checkpoint = torch.load(model_checkpoint_path)
    net.load_state_dict(checkpoint['state_dict'])
```

```
    epoch = epochs_left

# train on GPU
device = torch.device("cuda:0"
        if torch.cuda.is_available() else 'cpu')
print(f'\ndevice: {device}')

net.to(device)

## 5. create training pipeline
tensor_x = torch.tensor(df['embedding'].tolist())
tensor_y = torch.tensor(df['label'].tolist(), dtype=torch.long)

data_train = TensorDataset(tensor_x, tensor_y)
loader_train = DataLoader(data_train,
        batch_size=batch_size, shuffle=shuffle)

## 6. train and save model
save_every_epoch = False
for run in range(epoch):
    running_loss = 0.0
    print(f'\nepoch {run + 1}')
    time0_epoch = time.time()

    for i, data in enumerate(loader_train):
        # get the inputs; data is a list of [inputs, labels]
        inputs, labels = data[0].to(device), data[1].to(device)
```

```
        if model_type != 'MP':
            inputs = inputs.unsqueeze(1)
            # reshape by add 1 to num_channel
            # (parameter: batch_size, num_channel, height, width)

        # zero the parameter gradients
        optimizer.zero_grad()

        # forward + backward + optimize
        outputs = net(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        # print statistics
        running_loss += loss.item()
        if i and i % 200 == 0:
            print(f'\tbatch {i}    loss: {running_loss/200}')
            running_loss = 0.0

    time_diff_epoch = round(time.time() - time0_epoch, 2)
    print(f'\tTime elapsed: {time_diff_epoch}')

    if save_every_epoch:
        model_save_path_full = os.path\
                .join(model_save_path, model_name_temp)
        torch.save(net.state_dict(), model_save_path_full)
```

```
# save model checkpoint for re-training purposes
state = {'state_dict': net.state_dict()}
model_name_temp = model_name + '_checkpoint.pth'
model_checkpt_path_full = os.path\
        .join(model_save_path, 'checkpoint', model_name_temp)
torch.save(state, model_checkpt_path_full)

# save full model
model_name_temp = model_name + '.pth'
model_save_path_full = os.path\
        .join(model_save_path, model_name_temp)
torch.save(net.state_dict(), model_save_path_full)

print('\nProcess complete.')
```

7.2.3 Net Architecture Classes

```
"""
This module contains neural network classes
"""
import torch
import torch.nn as nn
import torch.nn.functional as F

class multilayer_perceptron(nn.Module):
    def __init__(self, dataset):
```

```
        if dataset == 'ag_news':
            self.fc1_in = 245 * 50 * 1
            self.fc1_out = 120          # same as fc2_in
            self.fc2_out = 50           # same as fc3_in
            self.fc3_out = 4
            # same as number of classes
        elif dataset == 'yelp_review_polarity':
            print('space holder')
        else:
            raise ValueError(f'Dataset: {dataset} not recognized.')

        super(multilayer_perceptron, self).__init__()
        self.fc1 = nn.Linear(self.fc1_in, self.fc1_out)
        self.fc2 = nn.Linear(self.fc1_out, self.fc2_out)
        self.fc3 = nn.Linear(self.fc2_out, self.fc3_out)

    def forward(self, x):
        x = x.view(-1, self.fc1_in)
        # token length, w2v embedding dimension, channel
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x

class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
```

```
        self.conv1 = nn.Conv2d(1, 1, (4, 50))
        # input channel, output channel, kernel size
        self.pool = nn.MaxPool2d(kernel_size=(4, 1), stride=(1, 1))
        self.fc1 = nn.Linear(239 * 1, 120)
        # 120 chosen randomly (< input dimension)
        self.fc2 = nn.Linear(120, 50)
        # 50 chosen randomly (< 50)
        self.fc3 = nn.Linear(50, 4)
        # 4 = number of classes

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = x.view(-1, 239 * 1)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x

class CNN_kim(nn.Module):
    def __init__(self):
        super(CNN_kim, self).__init__()
        self.conv1_a = nn.Conv2d(1, 1, (3, 50))
        # channel 1 of conv, with kernel=3
        self.conv1_b = nn.Conv2d(1, 1, (4, 50))
        # channel 2 of conv, with kernel=4
        self.conv1_c = nn.Conv2d(1, 1, (5, 50))
        # channel 3 of conv, with kernel=5
```

```
        self.pool = nn.MaxPool2d(kernel_size=(4, 1),
                 stride=(1, 1))
        self.pool_b = nn.MaxPool2d(kernel_size=(3, 1),
                stride=(1, 1))
        self.pool_c = nn.MaxPool2d(kernel_size=(2, 1),
                stride=(1, 1))
        self.fc1 = nn.Linear(240 * 3, 120)
        self.fc2 = nn.Linear(120, 50)
        self.fc3 = nn.Linear(50, 4)

    def forward(self, x):
        x0 = x
        x = self.pool(F.relu(self.conv1_a(x)))
        y = self.pool_b(F.relu(self.conv1_b(x0)))
        z = self.pool_c(F.relu(self.conv1_c(x0)))
        x = x.view(-1, 240 * 1)
        y = y.view(-1, 240 * 1)
        z = z.view(-1, 240 * 1)
        x = torch.cat((x, y, z), dim=1)
        # combine results from three conv
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x

class CNN_deep(nn.Module):
    def __init__(self):
```

```
        super(CNN_deep, self).__init__()
        self.conv1 = nn.Conv2d(1, 64, (3, 50))

        # Conv block 1
        self.conv2 = nn.Conv2d(64, 64, (3, 1))
        self.conv2_bn = nn.BatchNorm2d(64)
        self.pool2 = nn.MaxPool2d(kernel_size=(120, 1),
                stride=(1, 1))  # halve the dimension

        # Conv block 2
        self.conv3 = nn.Conv2d(128, 128, (3, 1))
        self.conv3_bn = nn.BatchNorm2d(128)
        self.pool3 = nn.MaxPool2d(kernel_size=(59, 1),
                stride=(1, 1))    # halve the dimension

        # Conv block 3
        self.conv4 = nn.Conv2d(256, 256, (3, 1))
        self.conv4_bn = nn.BatchNorm2d(256)
        self.pool4 = nn.MaxPool2d(kernel_size=(28, 1),
                stride=(1, 1))   # halve the dimension

        self.fc1 = nn.Linear(27 * 1 * 512, 4096)
        self.fc2 = nn.Linear(4096, 2048)
        self.fc3 = nn.Linear(2048, 4)

    def forward(self, x):
        x = self.conv1(x)
```

```
# Conv block 1
x = F.relu(self.conv2_bn(self.conv2(x)))
x = F.relu(self.conv2_bn(self.conv2(x)))
x = self.pool2(x)
x = torch.cat((x, x), dim=1)
# doubling the feature space

# Conv block 2
x = F.relu(self.conv3_bn(self.conv3(x)))
x = F.relu(self.conv3_bn(self.conv3(x)))
x = self.pool3(x)
x = torch.cat((x, x), dim=1)
# doubling the feature space

# Conv block 3
x = F.relu(self.conv4_bn(self.conv4(x)))
x = F.relu(self.conv4_bn(self.conv4(x)))
x = self.pool4(x)
x = torch.cat((x, x), dim=1)
# doubling the feature space

x = x.view(-1, 27 * 1 * 512)
x = F.relu(self.fc1(x))
x = F.relu(self.fc2(x))
x = self.fc3(x)
return x
```

7.2.4 Miscellaneous Functions

```
"""
This module contains misc. function used
in train_model.py
"""
import random
import numpy as np
import torch
import nets


def zero_padding(list_to_pad, max_length,
        pad_dimension, pad_method='bottom'):
    """
    This function takes a list and add list of zeros until
    max_length is reached.
    The number of zeroes in added list is determined
    by pad_dimension, which is the
    same as the dimension of the word2vec model.

    There are three modes available:
        bottom - zero vectors are added to the bottom/right
        side of the embedding bothside - zero vectors are
        added to both side of the embedding random -
        zero vectors's positions are randomly inserted
        into the embedding

    This function is intended to handle one list
```

```
only so it can be passed
into a dataframe as a lambda function.
"""
# find number of padding vector needed
num_pad = max_length - len(list_to_pad)

# create zero vector based on pad_dimension
vector_pad = np.asarray([0] * pad_dimension, dtype=np.float32)
vector_pad = [vector_pad]
# convert to list of np.ndarray so we can append together

if pad_method == 'bottom':
    iteration = 0
    while iteration < num_pad:
        list_to_pad = np.append(list_to_pad,
        vector_pad, axis=0)
        iteration += 1

elif pad_method == 'bothside':
    num_each_side = int(num_pad/2)
    iteration = 0
    list_each_side = np.empty((0, pad_dimension),
            dtype=np.float32)
    while iteration < num_each_side:
        list_each_side = np.append(list_each_side,
        vector_pad, axis=0)
        iteration += 1
```

```
        list_to_pad = np.append(list_each_side,
                list_to_pad, axis=0)
        list_to_pad = np.append(list_to_pad,
                list_each_side, axis=0)

        # add one more pad to the right side when odd
        # number of padding vector
        if num_pad%2 == 1:
            list_to_pad = np.append(list_to_pad,
                vector_pad, axis=0)

    elif pad_method == 'random':
        position_random = random.sample(range(0, max_length-1),
         num_pad)
        index_list_to_pad = 0
        list_temp = np.empty((0, pad_dimension), dtype=np.float32)
        for position in range(max_length):
            if position in position_random:
                vector_to_append = vector_pad
            else:
                vector_to_append = [list_to_pad[index_list_to_pad]]
                index_list_to_pad += 1

            list_temp = np.append(list_temp,
                vector_to_append, axis=0)
        list_to_pad = list_temp
```

```
    else:
        raise ValueError(f'{pad_method} is not
                a valid padding method.')

    return list_to_pad


def evaluate_accuracy(loader_test, net, classes, model_type):
    """
    This function takes pytorch data loader,
    pytorch class for NN,
    and a tuple of class labels to calculate accuracy
    at macro and class levels
    """
    correct = 0
    total = 0
    with torch.no_grad():
        for data in loader_test:
            text, labels = data
            if model_type != 'MP':
                text = text.unsqueeze(1)
                # reshape text to add 1 channel

            outputs = net(text)
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()
```

```
    print(f'\nAccuracy: {100 * correct/total}%')

    class_correct = list(0. for i in range(len(classes)))
    class_total = list(0. for i in range(len(classes)))
    with torch.no_grad():
        for batch, data in enumerate(loader_test):
            text, labels = data
            if model_type != 'MP':
                text = text.unsqueeze(1)
                # reshape text to add 1 channel

            outputs = net(text)
            _, predicted = torch.max(outputs, 1)
            c = (predicted == labels).squeeze()

            for i in range(len(labels)):
                label = labels[i]
                class_correct[label] += c[i].item()
                class_total[label] += 1

    for i in range(4):
        print('Accuracy of class %5s : %2d %%' % (
            classes[i], 100 *
                class_correct[i] / (class_total[i] + .000001)))

def model_loader(model_type, dataset):
```

```
    """
    This function loads model from net.py
    """
    if model_type == 'MP':
        net = nets.multilayer_perceptron(dataset)
    elif model_type == 'CNN':
        net = nets.CNN(dataset)
    elif model_type == 'CNN_kim':
        net = nets.CNN_kim(dataset)
    elif model_type == 'CNN_deep':
        net = nets.CNN_deep(dataset)
    else:
        raise ValueError(f'\nmodel_type:
                {model_type} is not recognized.')

    return net

def vocab_clean_up(token_list, w2v):
    """
    This function removes words not in w2v vocab list.
    Meant to be used in lambda function.
    eg. df['text_clean'] = df['text'].apply(lambda x:
    vocab_clean_up(x, w2v))
    """
    vocab_list = w2v.wv
    temp_list = []
```

```
    for token in token_list:
        if token in vocab_list:
            temp_list.append(token)

    return temp_list

if __name__ == "__main__":
    pass
```

7.3 Model Evaluation

7.3.1 Main

```
"""
This module contains code to evaluate model against test set.
The module takes input from model.cfg file.
"""
import os
import random
import pandas as pd
import configparser
from nltk.tokenize import word_tokenize
from gensim.models import Word2Vec

import torch
import torch.nn as nn
import torch.optim as optim
```

```
from torch.utils.data import TensorDataset, DataLoader

from utility import zero_padding, evaluate_accuracy,
model_loader, vocab_clean_up
from nets import multilayer_perceptron, CNN, CNN_kim, CNN_deep

## 0. setting up parameter
config = configparser.ConfigParser()
config.read('model.cfg')

## PATH
data_path = config['PATH']['test_data_path']
w2v_path = config['PATH']['w2v_path']
model_save_path = config['PATH']['model_save_path']
model_name = config['PATH']['model_name']
dataset = config['PATH']['dataset']

## MODEL_PARAMETERS
model_type = config['MODEL_PARAMETERS']['model_type']
emb_dim = int(config['MODEL_PARAMETERS']['emb_dim'])
pad_method = config['MODEL_PARAMETERS']['pad_method']

classes = ('0', '1', '2', '3')

## 1. load dataset
df = pd.read_csv(data_path)
# convert class 4 to class 0
```

```
df['Class Index'] = df['Class Index'].replace(4, 0)
print(df['Class Index'].value_counts())

## 2. apply tokenization and embedding
df['text_token'] = df['Description']\
.apply(lambda x: word_tokenize(x))

w2v = Word2Vec.load(w2v_path)
df['text_token'] = df['text_token']\
.apply(lambda x: vocab_clean_up(x, w2v))
df['embedding'] = df['text_token'].apply(lambda x: w2v[x])

## 3. zero pad to max length
df['text_length'] = df['text_token'].apply(lambda x: len(x))
#max_length = max(df['text_length'])
max_length = 245    # specify max length from train set

print(f'max_length: {max_length}')

df['embedding'] = df['embedding']\
.apply(lambda x: zero_padding(x, max_length,
        emb_dim, pad_method))

tensor_x = torch.tensor(df['embedding'].tolist())
tensor_y = torch.tensor(df['Class Index'].tolist(),
dtype=torch.long)
```

```
data_test= TensorDataset(tensor_x, tensor_y) # create your datset
loader_test = DataLoader(data_test, batch_size=32, shuffle=False)
# create your dataloader

dataiter = iter(loader_test)
text, labels = dataiter.next()

# load model
net = model_loader(model_type, dataset)
model_name_temp = model_name + '.pth'
model_save_path_full = os.path.join(model_save_path,
model_name_temp)

net.load_state_dict(torch.load(model_save_path_full))
net.eval()

evaluate_accuracy(loader_test, net, classes, model_type)
```

REFERENCES

[1] Yoon Kim *Convolutional neural networks for sentence classification*. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, page 1746-1751, Doha, Qatar. Association for Computational Linguistics, 2014

[2] A. Conneau, H. Schwenk, L. Barrault, and Y. LeCun *Very deep convolutional networks for text classification*. arXiv preprint arXiv:1606.01781, 2016

[3] N. Kalchbrenner, E. Grefenstette, and P. Blunsom *A Convolutional Neural Network for Modeling Sentences*. arXiv preprint arXiv:1404.2188, 2014

[4] H. Le, C. Cerisara, A. Denis *Do Convolutional Networks need to be Deep for Text Classification?*. arXiv preprint arXiv:1707.04108, 2017

[5] G. Huang, L. Zhuang, L. Maaten, K. Q. Weinberger *Densely Connected Convolutional Networks*. arXiv preprint arXiv:1608.06993, 2016

[6] Hashemi, M. Enlarging smaller images before inputting into convolutional neural network: zero-padding vs. interpolation. J Big Data 6, 98 (2019). https://doi.org/10.1186/s40537-019-0263-7, 2019

[7] A. Krizhevsky, I. Sutskever, and G. Hinton. Imagenet classification with deep convolutional neural networks. In *NIPS*, 2012