Journal of Big Data

RESEARCH

Open Access

Improving deep neural network design with new text data representations

Joseph D. Prusa* and Taghi M. Khoshgoftaar

*Correspondence:
jprusa@fau.edu
Florida Atlantic University, 777 Glades Road, Boca Raton, FL, USA

Abstract

Using traditional machine learning approaches, there is no single feature engineering solution for all text mining and learning tasks. Thus, researchers must determine and implement the best feature engineering approach for each text classification task; however, deep learning allows us to skip this step by extracting and learning high-level features automatically from low-level text representations. Convolutional neural networks, a popular type of neural network for deep learning, have been shown to be effective at performing feature extraction and classification for many domains including text. Recently, it was demonstrated that convolutional neural networks can be used to train classifiers from character-level representations of text. This approach achieved superior performance compared to classifiers trained on word-level text representations, likely due to the use of character-level representations preserving more information from the data. Training neural networks from character level data requires a large volume of instances; however, the large volume of training data and model complexity makes training these networks a slow and computationally expensive task. In this paper, we propose a new method of creating character-level representations of text to reduce the computational costs associated with training a deep convolutional neural network. We demonstrate that our method of character embedding greatly reduces training time and memory use, while significantly improving classification performance. Additionally, we show that our proposed embedding can be used with padded convolutional layers to enable the use of current convolutional network architectures, while still facilitating faster training and higher performance than the previous approach for learning from character-level text.

Keywords: Deep learning, Big data, Text mining, Sentiment, Convolutional neural networks, Character embedding, GPU computing

Background

Many current application domains of machine learning and artificial intelligence involve knowledge discovery from text, such as sentiment analysis, document ontology and spam detection. Humans have years of experience and training with language, enabling them to understand complicated, nuanced text passages with relative ease. Text classifiers attempt to emulate or replicate this knowledge so that computers can discriminate between concepts encountered in text. Learning high-level concepts from text, such as those found in many applications of text classification, is a difficult task due to the many challenges associated with text mining and classification. A common approach to this machine learning task is to find a way to represent text, such as human-engineered features extracted from the text, then using this representation with training data to train a classifier. In the current text classification paradigm, human researchers create an approach for extracting features from text by looking at words, phrases, parts of speech, other morphological features, employing a lexicon, and mapping relations between words to determine concept families and semantic similarities. These different types of features may be used separately or combined into a single feature space in an attempt to build a representation capturing the complexity of language. Unfortunately, these approaches are often language or even task dependant [1] and may contain less information than the original text. Thus, this approach may yield feature engineering that is only applicable to a narrow range of text classification tasks.

One of the most commonly used method of feature engineering in text is the bag-of-word approach. A word vector is constructed from the data by parsing the data (text documents), identifying the words used throughout all documents and making a vector with entries representing each word, then ascribing values to this vector for each instance or document using word presence of frequency. While this approach has been demonstrated to be effective, it is far from ideal. Word placement within a sentence is not preserved and contextual indicators that may change the meaning of a word are absent from the final feature space. Additionally, a very large number of words can be identified with many being infrequent or unique to a single document leading to a high dimensional, sparse feature space. This increases computational costs and can degrade performance due to overfitting. While high dimensionality may be addressed with feature selection techniques [2], this approach to text understanding eliminates the majority of information that humans use when reading and writing text.

More complicated approaches have been devised and combined (or used in place of) the bag-of-words approach, such as grouping words by semantic concepts [3], the addition of part of speech tags [4], detecting grammatical patterns [5], and word ontology driven natural language processing (NLP) [6]; however, these feature spaces still contain less information than the original text as the full structure and context of the text is not preserved. Additionally, as features may be designed with key domain knowledge, more complicated feature engineering methodologies result in a text understanding solution that is less versatile and may be restricted to a single text domain. While additional information from the text can be turned into features, the burden of engineering features from text still falls upon the human researcher. A researcher implements a feature engineering methodology to extract features they believe will be useful; however, this is ultimately an educated guess. The resulting feature space is an incomplete representation of the text and may be missing valuable information that may not be easily identifiable to the researcher.

Deep neural networks provide an alternative approach for text mining tasks and feature extraction. High-level features can be learned automatically, allowing for the removal of human bias in feature engineering and the preservation of more information as the original data can be used for training. As features are learned as part of the training process of a deep neural network, researchers are not required to provide any specialized domain knowledge to the network making this family of approaches language and task independent. Instead, large volumes (potentially petabytes) of data are leveraged to train through repeated example.

Convolutional neural networks (CNN) are a family of neural networks that have been shown to be among the best solutions for training computers in tasks of computer vision. Krizhevsky et al. [7] developed a deep CNN that outperformed all previously existing approaches for image object recognition on the ImageNet dataset, due to its ability to detect high-level, abstract features for image detection. Additionally, they have recently been demonstrated to be able to learn high-level text concepts from character-level representations of text [1] in a manner similar to how they learn features from and can classify images. CNNs do not need any prior knowledge of the data to train a classifier as complex features can be learned automatically when training the network. This makes them well suited to text classification, since they do not need any prior knowledge of language. Starting from raw, character-level text data, abstract language concepts, including words, syntax, grammar and semantic similarities, are learned automatically. Zhang et al. [1] demonstrated using deep CNNs with character-level data representations of text outperforms approaches using higher level human generated text representations such as bag-of-words. As the network is trained from raw text no information is lost in constructing features. Also, since the network can be trained from character-level text representations, data dimensionality is less of an issue as there are less commonly used characters compared to words.

In this paper, we investigate the use of our new embedding for character-level representation of text classification tasks for use with CNNs and network design considerations due to the adoption of this embedding approach. First, we explain our new character embedding approach and demonstrate that it greatly reduces memory use and network training time due to greatly reducing the size of the initial input received by the network. We show that it outperforms the previous character embedding for the task of binary tweet sentiment classification, i.e. determining if tweets convey positive or negative sentiment. We also show that our character embedding can employ a larger alphabet at little to no additional cost, further enhancing performance, as training time scales logarithmically with alphabet size instead of linearly. Furthermore, we also explore network design implications due to using our new embedding. Namely, our embedding results in matrix representations of text instances where one dimension is much greater than the other. In this scenario, convolutional layers with padding are required to allow deeper networks to be trained.

This paper demonstrates the effectiveness of a new character embedding designed for training CNNs and explores how neural network design may be impacted by its adoption. To the best of our knowledge, text classification using character-level representations and deep neural networks has been previously investigated by only one research group [1], and we are the first to propose a more compact character embedding and to investigate its implications on neural network design. We show that our character embedding greatly reduces computational costs and training time, and improves classification performance. Additionally, we also show that using padded convolutional layers allows our embedding to be used with networks of arbitrary depth and the use of padding does not negate the benefits of our character embedding. Thus, our proposed character embedding can be adapted to any big data domain where high-level understanding of text is required, such as sentiment analysis, webpage ontology and topic classification.

The remainder of this paper is organized as follows. "Related works" section provides an overview of related work in text mining and deep learning. "Character-level text representations" section describes our newly created embedding for character-level representation. "Convolutional neural network design" section describes how convolutional neural networks work and design considerations that must be made on account of our new embedding. "General methodology" section provides details on the experimental methodology used to train and evaluate networks. "Experimental results" section presents results for our embedding and the use of padded layers. Finally, conclusions and future work are contained in "Conclusion" section.

Related works

Deep learning has been used for many text mining tasks including various non-classification natural language processing tasks, aiding in feature extraction prior to applying a separate machine learning algorithm, and end-to-end classification on word-level text representations. More recently, it has been shown that learning from character-level text representations may allow better performing classifiers to be trained [1]. The remainder of this section provides details on applications and current state of deep learning in text mining.

One of deep learning's most popular applications in text mining is to perform semantic indexing of text. In semantic indexing, relationships between words and/or phrases are found and a condensed feature space of abstracted data representations can be generated. As an example of this, Google's Word2vec provides an automated means of extracting semantic representations from big data. Word2vec uses a large-scale text corpus as input and produces vector representations of words by constructing a vocabulary and learning how to describe words outside the vocabulary from those inside it [8]. Word2vec has been combined with big data tools to produce a publicy available dictionary from a 100 million word text corpus from Google News [9] and provides a 300-dimensional vector representation of each word. However, bag-of-words has been shown to outperform Word2vec, in some instances such as Amazon review polarity and Yahoo answer topic prediction, despite its simplicity and high dimensionality [1].

Long short term memory (LSTM) networks are a popular deep neural network design for learning tasks with sequential data. This makes them a logical choice for text classification tasks as the data can be represented as a sequence of of words of characters. Using a LSTM network has been demonstrated to be effective for a wide range of text classification tasks, including sentiment, and can outperform traditional learners such as SVM [10]. While LSTMs can produce models with high performance, they are also slow to train and prone to over-fitting [11]. Due to their sensitivity to network architecture changes and hyper-parameter tuning, training a LSTM network can be considerably more challenging than using a CNN. Additionally, LSTMs are slower to train than CNNs as there is no equivalent to NVIDIA's CuDNN library [12] to greatly accelerate the training of LSTMs.

CNNs have been used to perform a variety of natural language processing tasks, including part-of-speech tagging, named entity extraction, identification of semantic roles, and linking semantically similar words [13]. More recently, CNNs have been used for end-to-end discriminative text classification tasks involving the identification of high-level concepts prevalent throughout an entire document; however, most implementations are built on top of existing feature engineering and extraction methodologies. Kim [14], demonstrated training a CNN on top of static, pre-made word vectors performs well for sentence classification tasks. Additionally, a network with a single convolutional layer performed better than any traditional learner, such as Multinomial Naïve Bayes and support vector machine (SVM), on their benchmarking datasets. CNNs have also been used to aid feature extraction from text. Poria et al. [15] built higher level features from textual data, represented by a 306 dimensional vector consisting of a word vector and part of speech values, by training a CNN. They used output of the penultimate fully connected layer to create features to use with other classifiers and found the features produced by the network allowed better classifiers to be trained and using a different learner for classification, such as SVM, performed better than relying on the CNN's final output layer for classification.

Recently, Zhang and LeCun [1] proposed a novel approach for text learning tasks where they were able to use CNNs to train classifiers by representing text in an image like, character-level fashion. This enabled them to train a deep convolutional neural network for text classification tasks involving high-level concepts from scratch with no prior feature engineering or extraction. They accomplished this by employing 1-of-m embedding, where each character is represented by a vector of size m, where m is the number of characters in their alphabet. Each character in a text instance was represented as a character vector and the instance as a sequence of character vectors. When visualized, a braille like output is generated. Using this embedding, they fed their data into a network with six 1D temporal convolutional layers and three fully connected layers. Their results showed training a deep CNN from character-level data outperformed networks trained on data with features generated with bag-of-words, bag-of-centroids and the deep learning approach Word2Vec.

We build on the work of Zhang and LeCun [1] by providing a new character embedding methodology for text classification with CNNs, demonstrate that our new character embedding greatly reduces training time, memory use and improves classification performance. Additionally, we demonstrate any restrictions to network design imposed by this embedding approach can be alleviated through the use of padded convolutional layers.

Character-level text representations

While no feature extraction needs to be performed, as we are teaching our CNNs to read from scratch, character-level data must still be represented in a manner that is acceptable as input for our networks. For this purpose we propose a new character embedding offering greatly reduced memory consumption and training time. We benchmark our approach against the embedding used by Zhang and LeCun [1]. We also investigate using different sized character alphabets.

Zhang and LeCun use character-level text representations to train CNNs by employing 1-of-*m* embedding. With this embedding, each character is represented as an *m* sized vector where all values are zero except the entry corresponding to the position in the alphabet of the found character (which has a value of 1). Each instance is then represented by a sequence of *m* sized vectors with length *l*, where *m* is the alphabet size and *l* is the length of the character sequences. The sequence length, *l*, is a pre-defined constant which is chosen to suit the specific classification task as it is determined by the document character length. The algorithm for this embedding is presented in Algorithm 1 . Text encoded in this fashion generated a sparsely populated matrix, as seen by our visualizations of 1-of-*m* embedding in Fig. 1a, b. In these figures, each character is a column and the full image is created by binding the sequence of characters together to create an $m \times l$ image. Zhang and LeCun used an alphabet of 70 characters ($m = 70$). This alphabet includes all 26 lowercase English letters, 10 digits, the new line command and 33 other characters [1].

Characters not contained in this alphabet are represented as a vector of zeroes. This embedding turns each character into a vector of length 70 with one non-zero entry (one black pixel in the column). For example, "a" would be 1000...0 (a 1 followed by 69 0s). Zhang and LeCun note sequences of characters encoded in this manner are visually similar to braille.

We also chose to use the larger UTF-8 alphabet[1] which contains 256 distinct characters ($m = 256$). Using this alphabet preserves far more information from our raw text data than the 70 character alphabet of Zhang and LeCun. Using the UTF-8 alphabet allows us to distinguish between lowercase and uppercase letters, something that is not done with the 70 character alphabet. This is advantageous since capitalization has been found to be beneficial for the detection of sentiment in tweets [5].

Algorithm 1 1-of-m encoding algorithm

```
1: procedure 1-OF-M-EMBEDDING
2:     maxlen = maximum character length $\forall$ instances
3:     a = Alphabet length
4:     for $i \in$ training instances do
5:         T = sequence of characters $\in i$
6:         m = $a \times maxlen$ array of zeros
7:         for j in range 0 to length($T$) do
8:             $u$ = alphabet position of $T[j]$
9:             $m[u, j] = 1$
```

Algorithm 2 Log-m embedding algorithm

```
1: procedure LOG-M-EMBEDDING
2:     maxlen = maximum character length $\forall$ instances
3:     for $i \in$ training instances do
4:         T = sequence of characters $\in i$
5:         m = $8 \times maxlen$ array of zeros
6:         for j in range 0 to length($T$) do
7:             $u$ = UTF8 representation of $T[j]$
8:             $b$ = binary array representation of u
9:             for $k$ in range 0 to length(b) do
10:                $m[k, j] = b[k]$
```

[1] UTF-8 includes unicode characters U+0000 to U+00FF. For the complete UTF-8 alphabet see: http://www.utf8-chartable.de/.

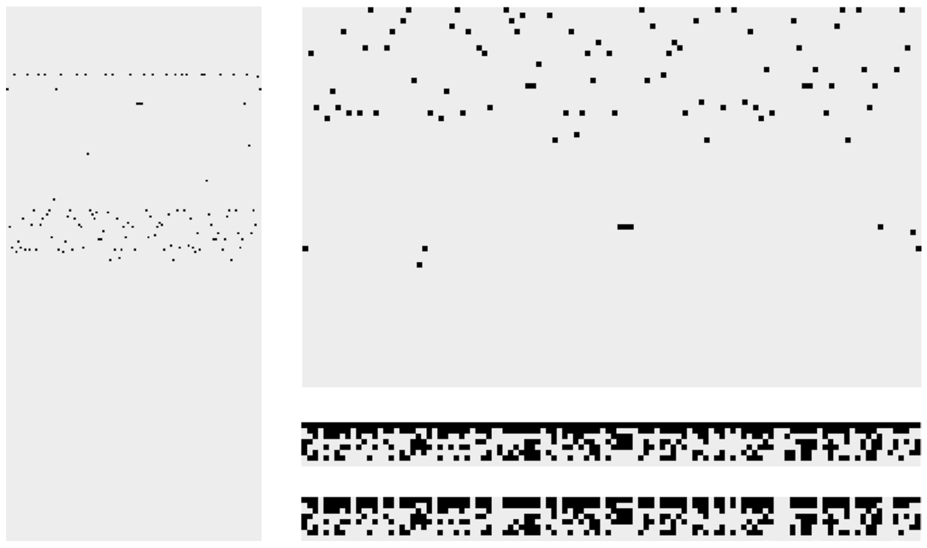

Fig. 1 Visualization of the four character embeddings and alphabet size combinations for a randomly selected tweet. *Black* pixels indicate a value of 1 while *grey* represent 0. *Left* 1-of-25. *Top–right* 1-of-70. *Middle–right* Log-256. *Bottom–right* Log-70

Our proposed embedding is inspired by 1-of-m embedding, but achieves a much smaller, denser representation reducing memory use, network input layer size and training time. The premise behind our representation is that instead of having a single non-zero value in our character embedding vector, we can have multiple non-zero values. Our new approach functions as follows. Each character in our chosen alphabet is given an integer value. Characters in our instances are then replaced by the corresponding integer. We then find the equivalent binary representation of a character's integer value, then turn this sequence into a vector of 0s and 1s. The exact embedding process for each instance is provided in Algorithm 2. Using this approach, an alphabet with n bits can be represented with n-dimensional character vectors. Thus, as the 256 unicode characters found in UTF-8 can be represented as an 8-bit integer, we can use a 8-dimensional vector to represent all characters in the UTF-8 alphabet. For example, within the UTF-8 alphabet, the letter "a" has an integer value of 97. This is equal to the binary representation "01100001" which we then turn into the 8-dimensional vector (0,1,1,0,0,0,0,1). Using 1-of-m embedding with the UTF-8 alphabet each character vector would be 256-dimensional.

Our new embedding can also be visualized and doing so creates braille-like results; however, it is far denser than Zhang and LeCun's 1-of-m embedding and its vertical dimension is far smaller, as seen in Fig. 1. Applying our approach to any alphabet size, such as the 70 used by Zhang and Chen, reduces character vector size from m to $log_2(m)$. Thus, using 256 characters requires a vector of size 8, while using 70 requires a vector of size 7. From this, we see that greatly increasing alphabet size has minimal impact on the size of character vectors using our embedding approach. As it reduces vector size from m to $log_2(m)$ we choose to call it log-m embedding. In our experiments comparing 1-of-m embedding and log-m embedding, we chose alphabets of 70 and 256 characters and denote the four approaches as log-70, log-256, 1-of-256 and 1-of-70 for the remainder of the paper.

Notably, the resulting vector for each character with this embedding is 1/16th the size of the one produced by 1-of-m embedding for the UTF-8 alphabet since it only requires a vector of length $log_2(m)$ instead of length m. Thus, the resulting matrix representations of each instance contain fewer values and requires less memory. Additionally, as the input layer of a CNN must match the dimension of instances from the input data a smaller input layer with less neurons can be employed. Thus, using log-m embedding results in a network containing far less total neurons, for the same depth and layer parameters, as 1-of-m embedding, greatly reducing training costs.

Convolutional neural network design

Convolutional neural networks consist of sparsely connected convolutional layers followed by fully connected dense layers (these dense layers are equivalent to a multilayer perceptron neural network). Convolutional layers are sparsely connected. Neurons in these layers are connected to a small region of the previous layer known as the receptive field, instead of the entire previous layer as is found in a dense layer. The most common receptor field sizes are small, such as 3×3 or 5×5 neurons. By being sparsely connected CNNs view and learn local correlations. The convolutional layers in the network are followed by several densely connected layers with the last layer containing one neuron for each possible classification outcome.

Several key hyper parameters control the behavior of the convolutional layers and number of neurons in the final network including number of filters, stride and padding. Neurons in each layer learn a weight vector to create a feature map. These weights are determined by applying numerous filters, which have a small receptive field size and are convolved across the dimensions of the data to create an activation map of the filter. Multiple filters can be applied so that more features can be extracted by connecting multiple neurons to each region of the previous layer. Stride is the distance between receptive fields on the input volume. Padding refers to adding a border of zeroes around the input volume. Typically, an nxn instance would be reduced in dimension by the application of a convolutional layer when output to the subsequent layer. For example, a 3×3 convolution would reduce an $n \times n$ instance to $(n-2) \times (n-2)$. This occurs because the convolution can not be moved to the extreme edges of the instance as it would have null inputs. Adding padding can eliminate this size reduction as it allows filters to be applied to the edges of the original input volume. Training iterations (epochs) employs gradient decent to learn network parameters. As each convolutional layer is a view of the neurons of the previous layer, stacking layers allows a larger region of the initial input to be viewable by neurons in deeper layers. For example two 3×3 layers will result in a network that has 5×5 views as each layer transforms 3×3 views to a single output neuron.

Convolutional layers are also accompanied by an activation function. We use a rectified linear units (ReLU) layer as it promotes nonlinear responses [16]. We also strategically place an additional type of layer, the max pooling layer, between convolutional layers. The max pooling layer conducts non-linear down sampling by eliminating non-maximal values, thus reducing computation for upper layers as many can be discarded

[17]. We use a pool size of 2×2, which halves both dimensions of the input field for our next layer.

For our study, we created two sets of CNNs. The first set consists of three networks designed for comparison of our different data representations. Three networks were required as we have three very different input dimensions, $8 \times l$, $70 \times l$, and $256 \times l$, where l is the maximum instance length, and three networks were designed in an investigation of how our new embedding might impact network design. In our first set of networks, each network consists of three convolutional layers followed by three fully connected layers. Our convolutional layers used a 3×3 receptive fields (filter size), with a stride of 1 and no padding, followed by three fully connected layers to match the networks used by Zhang and Lecun [1]. We used 32 filters based on preliminary experimentation.

When using 1-of-m embedding, a 2×2 max pooling layer is placed after every convolutional layer (3 in total), while only a single max pooling layer is included in the network for log-m embedding. Due to having an input volume of $8 \times l$, less max pooling layers could be used with for log-m embedding. Since each max pooling layer halves both dimensions of the input for the next layer and stacking three 3×3 convolutional layers creates an effective receptive field of 7×7, max pooling could not be performed on the $8 \times l$ input volume generated by log-m embedding between each convolutional layer. Only one max pooling layer could be used in this network and it is placed between the convolutional and fully connected layers. When using log-70 embedding, we padded the character vectors with an extra zero to make them 8 dimensional instead of 7 so that the same network architecture could be used for both log-m approaches.

The first two fully-connected layers consist of 512 neurons. Each is followed by a dropout layers, with $p = 0.5$, that helps avoid overfitting [18]. Since we are performing binary sentiment classification, the final fully-connected layer contains two neurons using the softmax logistic function [19]. We use binary cross entropy [20] to create our loss function to evaluate the network using gradient descent after each epoch. Parameters and architecture for our networks are provided in Table 1.

As noted above, when using log-m embedding for our two alphabet sizes, we have an input volume of $8 \times l$. Without padding, each convolutional layer reduces the dimension by 2. Thus, after three layers we are looking at an $(8 - 3 * 2) \times (l - 3 * 2)$ or $2 \times (l - 6)$ volume. Since we have a dimension of 2 in one direction, no further convolutional layers with 3×3 filters can be applied. This sets a limit of three for the maximum number of 3×3 convolutional layers that can be used with this embedding. By using padding, this limit can be removed. Since padding adds a border of zeroes around our input volume to prevent its dimension from being reduced by each convolutional layer, we no longer have a limit to the number of layers employed. Thus, we can design a network similar to AlexNet [7], one of the highest performing network architectures for image classification. This architecture consists of stacking three 3×3 convolutional layers, followed by a 2×2 max pooling layer. Again, our small dimension (8) imposes a limit on the number of layer stacks we can have due to max pooling. After our first max pooling layer, we have a $4 \times (l/2)$ volume, after the second we have a $2 \times (l/4)$ volume and no further 3×3 convolutions can be performed. Thus, for the deepest network, we can construct an architecture consisting of three convolutional layers followed by a max pooling layer,

Table 1 Neural network parameters for networks used to test character embeddings and padded vs. non-padded convolutonal layers

1-ofm vs. log(m) embedding						
	log-m			**1-of-m**		
Convolutional layer	**Number of filters**	**Filter**	**Pool**	**Number of filters**	**Filter**	**Pool**
1	32	3 × 3	–	32	3 × 3	2 × 2
2	32	3 × 3	–	32	3 × 3	2 × 2
3	32	3 × 3	2 × 2	32	3 × 3	2 × 2
Dense layer		**Number of neurons**			**Number of neurons**	
1		512			1024	
2		512			1024	
3		2			2	
Padded vs. non-padded						
	Deep padded			**Padded/non-padded**		
Convolutional layer	**Number of filters**	**Filter**	**Pool**	**Number of filters**	**Filter**	**Pool**
1	128	3 × 3	–	128	3 × 3	–
2	128	3 × 3	–	128	3 × 3	–
3	128	3 × 3	2 × 2	128	3 × 3	2 × 2
4	128	3 × 3	–	–	–	–
5	128	3 × 3	–	–	–	–
6	128	3 × 3	2 × 2	–	–	–
Dense layer		**Number of neurons**			**Number of neurons**	
1		512			1024	
2		512			1024	
3		2			2	

three more convolutional layers, a second max pooling layer, then the fully connected layers.

Using additional convolutional layers should improve classifier performance as higher-level features will be extracted. As we are curious about the benefits of using padded layers to build deeper networks on data where one dimension of input volume is small, comparing a six layer network against a three layer network is unfair. Thus, we designed two networks, one using padding and one with no padding, that are otherwise equal. Each network consists of three 3×3 convolutonal layers, 128 filters on each layer, followed by a 2×2 max pooling layer, then two fully connected layers with 1024 neurons and a final output layer of two neurons. Parameters for the convolutional layers are provided in Table 1.

General methodology

This section provides details on our data, system environment and experimental design for evaluating our character embedding and the impact of padding in convolutional layers.

Data set

We use instances from the sentiment 140 corpus [21] as training, validation and test data. For evaluation of different character embedding, we created our dataset by randomly sampling (without replacement) 50,000 positive and 50,000 negative instances to create a balanced dataset of 100,000 instances. The corpus contains 1.6 million instances with positive or negative class labels and was generated by collecting and labeling tweets using emoticons. While using additional instances would result in better classifiers, this number was sufficient to train an effective classifier and demonstrate the differences between embeddings, while being small enough to train numerous models. When investigating padding, we elected to use the entire corpus (1.6 million tweets).

Albeit each tweet should be 140 characters or less, our longest tweet from the 100,000 randomly sampled tweets was 182 characters in length so we chose $l = 182$ and appended zero vectors to pad short tweets have the correct length. When using all tweets max length was found to be 374 and l was set to be 374. Tweets should be limited to 140 characters; however as the corpus is encoded with UTF-8 embedding, characters outside the range of UTF-8 are depicted as a string of characters resulting in the observed text lengths. Furthermore, we choose l to be defined by the longest document in our data (182 for experiments on embedding, 374 for experiments with padding), since our text is short. For a text corpus with longer documents and greater variation in document length, it may be more appropriate to select an l that is shorter than the longest document to avoid excessive padding, truncating the few longer instances as needed.

When training and evaluating our networks, we split our data into three partitions. First, it was split into training (90%) and test (10%). Test data is never seen by the network until after it has finished training as is used to perform a final performance evaluation. Training data was further split, with 20% set aside for validation of each epoch to optimize the network through back propagation with gradient decent. The remaining training data is used to train the parameters of the network. Every time a network is trained, the instances selected for each partition are randomized. Thus, every experimental run trains a different network. Tables 2 and 3 provide full details for the size of each partition and the resulting shape of our data for experiments on embedding and padding.

Experimental design, training and validation

For testing embedding, each network was trained for 100 epochs with experiments repeated 20 times to eliminate bias, due to chance split when creating training, validation and test partitions. This allows us to conduct tests for statistically significant differences in classification performance between the different character embedding methods. However, 1-of-m embedding with 256 characters was only trained once to allow comparisons of training time due to its computational resource requirements. For our experiments on using padded convolutional layers, each network was trained for 10 epochs and experiments were not repeated due to computational cost, since this experiment is a demonstration that padding enables our proposed character embedding to be used with deep networks while still offering faster training than 1-of-m embedding with un-padded layers.

Table 2 Training, validation and test partition sizes for data

Experiment	Partition	Num. of instances	% of data
Embedding	Full	100,000	100
	Training	72,000	72
	Validation	12,960	18
	Test	1296	10
Padding	Full	1600,000	100
	Training	1152,000	72
	Validation	207,360	18
	Test	20,736	10

Table 3 Dimension of each instance's using the four different character embeddings.

Experiment	Embedding	Dimension
Embedding	log-70	8×182
	log-256	8×182
	1-of-70	70×182
	1-of-256	256×182
Padding	log-256	8×374

We present classification results using accuracy as our performance metric. Accuracy is defined by the number of correctly identified instances divided by the total number of instances when our trained model is applied to the test data partition. This is an appropriate metric as the data is balanced and both classes are of equal importance.

Experiments were conducted on a cluster node with 20 Intel Xeon Cores, 128 GB of RAM and a NVIDIA Tesla K20 GPU accelerator running Scientific Linux 6.5. Networks were constructed using Theano 0.8.0 [22, 23] with NVIDIA CUDA 7.5 [24], Lasagne 0.2.dev1 [25] and the NVIDIA CuDNNv4 library [12] and compiled using GCC 5.2.0 and NVCC with the flag –use_fast_math.

Experimental results

Embedding approaches

Results, presented in Table 4, are the average of 20 runs using randomly generated training, validation and test partitions for each run. Average accuracy and average training time per epoch are displayed for each embedding approach. The results with the highest average accuracy and fastest training time are highlighted in bold The highest accuracy was achieved using log-256 embedding, followed by log-70 embedding. 1-of-70 has very similar accuracy compared to using 1-of-256 embedding.

Both log-m with an alphabet of 70 or 256 allow the network to be trained approximately 4.85× faster than 1-of-70 and 5.76× faster than 1-of-256. It is not surprising that log-m embedding enables the network to be trained considerably faster than 1-of-*m* embedding as a much smaller input layer is used leading to a network with far less parameters. Additionally, changing training batch size could lead to further reduction in training time. In our tests we did not optimize the batch size for different representations; however, as data has a smaller memory footprint using log-m embedding, training

Table 4 Results for 1-of-70, 1-of-256, log-70 and log-256

Embedding	Alphabet size	Time per Epoch	Accuracy
log-m	70	*13.5s*	73.106
	256	*13.5s*	*74.945*
1-of-m	70	65.5s	69.845
	256	77.7s	69.748

Best results in italic

batch sizes can be increased which could reduce I/O operations between the CPU and GPU.

Training time for networks using 1-of-256 encoded data is only slightly longer than 1-of-70, indicating that training time is not linearly proportional to the size of input, since we should observe it taking approximately three times as long to train if the relationship was linear. This is most likely due to the sparsity of 1-of-256 embedding and the efficiency of the CUDNN library for performing convolutions on sparse matrices.

Differences between character embeddings, excluding 1-of-256, were statistically tested using a one factor ANalysis Of VAriance (ANOVA) and a Tukey Honestly Significant Difference (HSD) tests [26]. The ANOVA test, Table 5, shows there are significant differences between character embedding approaches as p-value is less than 0.05. Table 6 presents the results of the Tukey HSD test and shows that both log-m embeddings are significantly better than 1-of-70; however, there is no significant difference between our two alphabet sizes.

Impact of padding convolutions

For our experiments comparing padded vs. non-padded convolutional layers, the full sentiment140 corpus was used. Thus, classifier accuracy is higher than was observed when testing the four embedding approaches. Results, presented in Table 7, show that there is little difference in accuracy between using padded vs non-padded convolutional layers (0.2%); however, doubling the number of layers result in 2.2–2.4% increase in classification accuracy. Thus, using padding to enable deeper networks to be trained is beneficial.

Comparing training time, it is observed that using padding results in the network taking 2.72$\times$ as long to train despite achieving similar classification performance. The increase in training time is due to the use of padding causing the input volume for subsequent layers to not be reduced by each convolutional layer. When using no padding, dimension is reduced by 2 for each of the three convolutional layers, then halved by the max pooling layer. This results in the input's volume of 8×374 being reduced to 1×184 before the fully connected layers. When using padding, only max pooling provides any

Table 5 ANOVA results for 1-of-70, log-70 and log-256 embedding

Factor	Df	Sum Sq	Mean sq	F value	Pr (>F)
Embedding	2	266.85	133.42	14.00	0.0000
Residuals	57	543.34	9.53		

Table 6 HSD results for 1-of-70, log-70 and log-256 embedding

Rank	Data_rep	Group	Average accuracy	stdev
1	log-256	a	74.945	2.455
2	log-70	a	73.106	4.712
3	1-of-70	b	69.844	0.603

Table 7 Results comparing padded vs. non-padded CNNs

Padding	Conv layers	Training time	Test loss	Accuracy
No	3	25.219s	0.39671	82.316
Yes	3	68.719s	0.39559	82.099
Yes	6	82.196s	0.35322	*84.519*

Best results in italic

dimension reduction, thus an input volume for the fully connected layers is 4×187 and requires approximately four times as many neural connections.

Adding an additional three convolutional layers and one max pooling layer slows training by a factor of 1.20. This is a far smaller increase in training time than switching from non-padded to padded convolutions. The presence of a second max pooling layer prevents training time from greatly increasing by offsetting the cost of calculating weights for the additional convolutional layers and associated filters through reducing input volume for the fully connected layers by an additional factor of 2. Compared to the unpadded 3 layer network, the padded 6 convolutional layer network takes $3.26\times$ longer to train for a 2.2% increase in performance.

Conclusion

Convolutional Neural Networks have been shown to be effective for text mining tasks including feature extraction and classification, and recently have been used to enable a classifier to be trained from character-level text data due to their ability to automatically identify and extract high-level concepts and features from text. Training from character-level data has the advantage of requiring no feature engineering or pre-training. Additionally, this approach has been shown to outperform training networks from data represented by previously extracted features [1]. In this study, we devised a new method of character embedding to facilitate faster training of deep convolutional neural networks from character-level representations of text data, discuss how it changes network architecture and investigate using padding to building deep networks when using this type of character embedding.

We present and demonstrate our new character embedding and show that it performs better than previous approaches. In our first set of experiments, we train networks using our new embedding approach and a previously defined method of embedding, 1-of-*m*, and measured the accuracy and training time of each approach. The results demonstrate our new character embedding approach, denoted as log-m, greatly reduces training time by 4.85 to $5.75\times$ and reduces memory use by up to a factor of 16. Furthermore, using log-m embedding resulted in significantly better classification performance compared to 1-of-*m* embedding.

When comparing padded vs. non-padded convolutional layers, we found little difference in classification performance; however, using padding to enable creation of deeper networks increased training time by 2.72×, and using padding while doubling the number of convolutional layers increased training time by 3.26×.

Though padding increases training time, it allowed us to build a network with six convolutional layers resulting in a 2.2–2.4% increase in classification accuracy. We observed using six convolutional layers required less than 20% more time to train than the padded CNN with three layers. Additionally, the reduction in training time when using log-m embedding more than offsets the cost of padding. Thus, the combination of log-m embedding with padded convolutional layers allows networks using popular deep CNN architectures, such as AlexNet, to be trained with higher classification accuracy in less time than 1-of-*m* embedding. Additionally, these results should apply to more recent networks architectures, such as the inception model [27], as the use of padding enables arbitrarily deep networks to be constructed from any input layer size.

While our experiments were conducted on tweet sentiment data, our approach can be applied to all text classification tasks. 1-of-m encoding has already been demonstrated to work in multiple text classification domains [1]. Future work should continue to investigate network design when using log-m embedding and performance of on longer text documents should be investigated. Also, while padding allows deep networks to be constructed with a variety of architectures, it does increase training time. Mixing padded and non-padded layers may help improve training time. Alternatively, non-square convolutional filters and max pooling layers could be implemented to allow deeper networks to be trained without relying on padding and should be investigated. Experiments should be extended to include text in other languages, since our proposed embedding should be far more efficient than 1-of-m embedding or word embeddings when using text with thousands of unique characters, such as Chinese.

Authors' contributions

JDP performed the primary literature review, experimentation and analysis for this work, and also drafted themanuscript. TMK worked with JDP to develop the article's framework and focus. TMK introduced this topic toJDP, and to complete and finalize this work. Both authors read and approved the final manuscript.

Acknowledgement

We acknowledge partial support by the NSF (CNS-1427536). Opinions, findings, conclusions, or recommendationsin this material are the authors' and do not reect the views of the NSF.

Competing interests

The authors declare that they have no competing interests.

Received: 4 November 2016 Accepted: 6 March 2017
Published online: 28 March 2017

References

1. Zhang X, LeCun Y. Text understanding from scratch. 2015. arXiv preprint arXiv:1502.01710.
2. Prusa JD, Khoshgoftaar TM, Dittman DJ. Impact of feature selection techniques for tweet sentiment classification. In: Proceedings of the 28th International FLAIRS Conference; 2015. p. 299–304.
3. Saif H, He Y, Alani H. Semantic sentiment analysis of twitter. In: The semantic web–ISWC. Berlin: Springer; 2012. p. 508–24.
4. Kouloumpis E, Wilson T, Moore J. Twitter sentiment analysis: the good the bad and the omg!. ICWSM. 2011;11:538–41.
5. Agarwal A, Xie B, Vovsha I, Rambow O, Passonneau R. Sentiment analysis of twitter data. In: Proceedings of the workshop on languages in social media. Association for Computational Linguistics; 2011. p. 30–8.

6. Trim C. NLP-driven ontology modeling. 2012. http://www.ibm.com/developerworks/community/blogs/nlp/entry/nlp_driven_ontology_modeling8?lang=en.
7. Krizhevsky A, Sutskever I, Hinton GE. Imagenet classification with deep convolutional neural networks. Adv Neural Info Proc Syst. 2012: 1097–105.
8. Najafabadi MM, Villanustre F, Khoshgoftaar TM, Seliya N, Wald R, Muharemagic E. Deep learning applications and challenges in big data analytics. J Big Data. 2015;2(1):1–21.
9. Mikolov T, Chen K, Corrado G, Dean J. Efficient estimation of word representations in vector space. 2013. arXiv preprint arXiv:1301.3781.
10. Tang D, Qin B, Liu T. Document modeling with gated recurrent neural network for sentiment classification. In: Proceedings of the 2015 conference on empirical methods in natural language processing; 2015. p. 1422–432.
11. Graves A. Neural networks. In: Supervised sequence labelling with recurrent neural networks. Berlin: Springer. p. 15–35.
12. Chetlur S, Woolley C, Vandermersch P, Cohen J, Tran J, Catanzaro B, Shelhamer E. cudnn: efficient primitives for deep learning. 2014. arXiv preprint arXiv:1410.0759.
13. Collobert R, Weston J. A unified architecture for natural language processing: deep neural networks with multitask learning. In: Proceedings of the 25th international conference on machine learning. New York city: ACM; 2008. p. 160–67.
14. Kim Y. Convolutional neural networks for sentence classification. 2014. arXiv preprint arXiv:1408.5882.
15. Poria S, Cambria E, Gelbukh A. Deep convolutional neural network textual features and multiple kernel learning for utterance-level multimodal sentiment analysis. In: Proceedings of EMNLP; 2015. p. 2539–544.
16. Nair V, Hinton GE. Rectified linear units improve restricted boltzmann machines. In: Proceedings of the 27th international conference on machine learning: ICML-10; 2010. p. 807–14.
17. Boureau YL, Ponce J, LeCun Y. A theoretical analysis of feature pooling in visual recognition. In: Proceedings of the 27th international conference on machine learning: ICML-10; 2010. p. 111–18.
18. Srivastava N, Hinton G, Krizhevsky A, Sutskever I, Salakhutdinov R. Dropout: a simple way to prevent neural networks from overfitting. J Mach Learn Res. 2014;15(1):1929–58.
19. Bridle JS. Probabilistic interpretation of feedforward classification network outputs, with relationships to statistical pattern recognition. In: Neurocomputing. Berlin: Springer; 1990. p. 227–36.
20. Goodman, J.: Classes for fast maximum entropy training. In: Proceedings IEEE international conference on acoustics, speech, and signal processing, vol. 1. Piscataway: IEEE. p. 561–64.
21. Go A, Bhayani R, Huang L. Twitter sentiment classification using distant supervision. CS224N Project Report: Stanford; 2009. p. 1–12.
22. Bastien F, Lamblin P, Pascanu R, Bergstra J, Goodfellow IJ, Bergeron A, Bouchard N, Bengio Y. Theano: new features and speed improvements. Deep learning and unsupervised feature learning NIPS 2012 Workshop. 2012.
23. Bergstra J, Breuleux O, Bastien F, Lamblin P, Pascanu R, Desjardins G, Turian J, Warde-Farley D, Bengio Y. Theano: a CPU and GPU math expression compiler. In: Proceedings of the Python for scientific computing conference: SciPy; 2010. Oral Presentation.
24. Nickolls J, Buck I, Garland M, Skadron K. Scalable parallel programming with cuda. Queue. 2008;6(2):40–53.
25. Dieleman S, Schlüter J, Raffel C, Olson E, Sønderby SK, Nouri D, Maturana D, Thoma M, Battenberg E, Kelly J, Fauw JD, Heilman M, diogo149, McFee B, Weideman H, takacsg84, peterderivaz, Jon, instagibbs, Rasul DK, CongLiu, Britefury, Degrave J. Lasagne: first release. 2015. doi:10.5281/zenodo.27878.
26. Berenson ML, Goldstein M, Levine D. Intermediate statistical methods and applications: a computer package approach. 2nd ed. Upper Saddle River: Prentice Hall; 1983.
27. Szegedy C, Liu W, Jia Y, Sermanet P, Reed S, Anguelov D, Erhan D, Vanhoucke V, Rabinovich A. Going deeper with convolutions. In: Proceedings of the IEEE Conference on computer vision and pattern recognition. 2015. p. 1–9.