Supervised and Semi-Supervised Text Categorization using LSTM for Region Embeddings

A replication

Fatemeh Aarabi
Arthur Jaques
Marcel Klehr

Contents

Contents **1**

Introduction **2**

Background **2**

Glossary **2**

Convolutions, Convolutional layer 2

Word embedding (Word2Vec) 2

One-hot encoding 3

Recurrent Neural Network (RNN) 3

LSTM 3

GRU 4

Model and experiment **4**

Task 4

Original model architecture 4

Data set 5

Tested model architectures 7

Training 9

Testing 12

Results and Evaluation **12**

Discussion 13

Bibliography **13**

Introduction

Supervised text classification tasks using neural networks have traditionally been carried out in the style of computer vision models using a number of one-dimensional convolutional layers (see CNN) operating on word embeddings. The paper at hand by Johnson and Zhang (Johnson and Zhang, 2016), deviates from this approach in several notable ways, yielding a reported improvement in learning time and accuracy. In this work we describe their work as well as the conceptual design and implementation of their model for supervised text classification and replicate their results in an exemplary training run on the IMDB dataset.

Background

Algorithmic document classification and specifically text classification ancompasses a wide range of problems such as genre classification, language identification, spam filtering, sentiment analysis etc. At the heart of the problem lies the realization that humans would like to categorize texts, i.e. written or transcribed speech acts, without having to read them. Notable approaches in this endeavor have been tf-idf, which uses simple term frequency, Naive Bayes classifiers, which make use of Bayesian probability networks, K-nearest neighbour classification, decision trees, as well as full natural language processing approaches.

Glossary

Convolutions, Convolutional layer

Convolutional layers have traditionally been used for computer vision tasks. They segment their input tensors into smaller equally-sized, potentially overlapping chunks and apply a variety of kernels using a sliding dot-product. The outputs of all kernel applications are typically fed into a max-pooling or average-pooling layer, which summarize multiple output values into a single value to reduce dimensions and thus computation. (Heaton, 2018, p. 326)

Word embedding (Word2Vec)

A traditional preprocessing procedure for natural language tasks is the embedding of words into a vector space prior to training and prediction. This is achieved by training a separate deep model to approximate an identity function utilizing dimensionality reducing and expanding layers. After training, the output of the dimensionality reduction layers is then used as a word embedding into a vector space. (Bengio *et al.*, no date, pp. 137–186)

One-hot encoding

In contrast to word embeddings, one-hot encodings are the simplest possible embedding of words into a vector space. Each word has its own dimension and the resulting word vectors have only one active dimension, all other dimensions being set to zero, thus coining the name ‘one-hot’. (Harris and Harris, 2010, p. 129)

Recurrent Neural Network (RNN)

In the past Recurrent Neural Networks (RNNs) have been widely used in the place of traditional neural networks in modelling the human language. The appeal of RNNs is how they are able to connect previous information to the present task: Instead of simply passing on information to the next layer as in a feed-forward network, recurrent layers use their own output as an additional input in the next step. However in practice, simple recurrent layers are not capable of handling long-term dependencies, which is an important factor when modelling the human language. This problem is addressed with the Long short-term memory (LSTM) architecture.

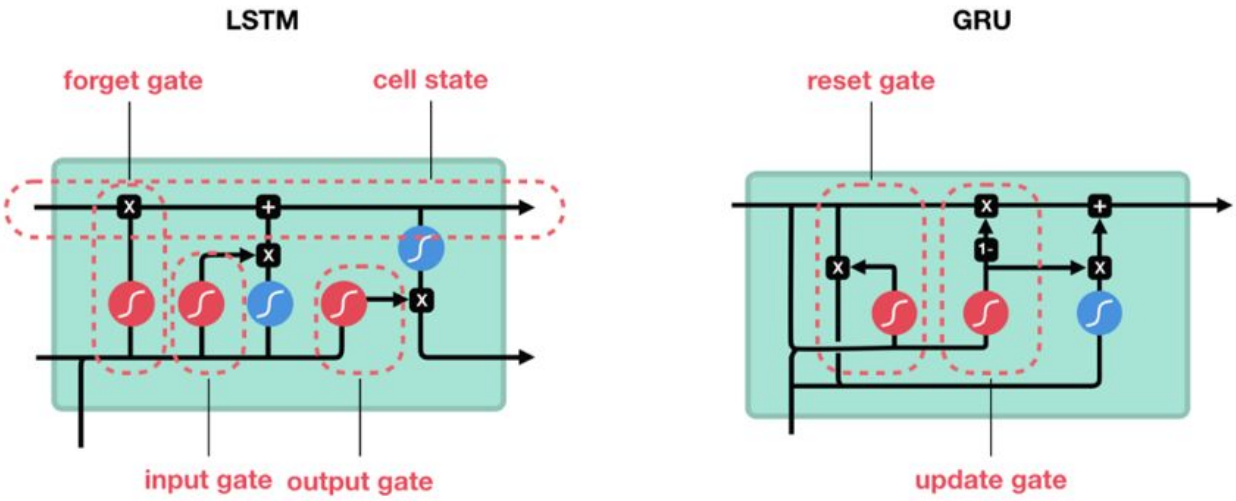

Figure 1: Long short-term memory and Gated recurrent unit architecture

LSTM

LSTM networks are a special kind of RNN capable of learning long-term dependencies. The core idea of the LSTM is the cell state where information is removed and added, carefully regulated by structures called gates. There are three gates: the forget gate, the input gate and the output gate. The forget gate takes the current input and the previous hidden state and regulates which information in the cell state is to be removed. The input gate decides what new information will be stored in the cell state from the current input and previous hidden state. The output gate creates a candidate for the hidden state from the cell state, regulating which information of the new candidate for the hidden state can pass and it updates the hidden state

by filtering the candidate for the hidden state. (Hochreiter and Schmidhuber, 1997) (*Understanding LSTM Networks -- colah's blog*, no date)

GRU

The Gated recurrent unit (GRU) is a newer generation of RNNs similar to LSTMs. In GRUs the cell state is removed and the hidden state is used as a means of transferring information and consists of only two gates: the reset gate and the update gate. The reset gate decides how to combine the current input and the previous hidden state and how much of the past information to forget. The update gate is similar to the forget and input gate of an LSTM combined. From the current input and previous hidden state, it decides what information to remove and what new information to add. GRU has fewer tensors; therefore, it is faster to train than the LSTM. (Britz, 2015) (Nguyen, 2018)

Model and experiment

Task

As neural networks have been most popular initially in the realm of computer vision, the first attempts at text classification models borrowed heavily from computer vision architectures, namely convolutional layers. However, the traditional approach to text classification using word embeddings and convolutions has several drawbacks. Word embeddings are costly to produce, yet seem to add little benefit over one-hot encodings of the same text when comparing the resulting model accuracy. Additionally, convolutional neural networks only accept a fixed input size, which imposes the need to cut longer texts into smaller chunks, potentially reducing the resulting accuracy due to lost long-term dependencies. A too small size of the applied convolution windows may also cause missed long-term dependencies in the text; on the other hand, the size of convolution windows must be kept small to allow for generalization and avoid overfitting.

Johnson and Zhang introduce a novel architecture for solving the text classification problem, which reportedly produces remarkably better results with an increased learning accuracy. The general idea remains nonetheless the same: regions of the text, maintained in a strict temporal order, are encoded in earlier layers (region embedding), and the output of these first layers is passed to later deep layers.

Original model architecture

Johnson and Zhang rely only on one-hot encodings of the input texts, which significantly reduces learning time. Instead of feeding these encodings into a convolutional feed-forward

network, the input series are fed to two parallel LSTM layers (or modified versions thereof, such as GRU), once forward and once backward. The advantage of a recurrent architecture here is the possibility of obtaining sliding region embeddings, of variable, fuzzy length. The outputs of all LSTM steps, aka. region embeddings, are then fed into a max pooling layer for each of the two parallel LSTM layers, which identify the embeddings significant for the task. A dropout layer is added before the top layer to limit overfitting. A final dense layer produces a classification prediction using soft-max activation, combining the output of the two LSTM-pooling stages. This approach renders the chopping of input texts for classification unnecessary and replaces rigid convolutions with fuzzy region embeddings, thus improving overall performance by allowing the model to pick up on long term dependencies in the texts.

Data set

The training data set used in this project was the IMDB dataset, available in the keras datasets module and consisting of English-language movie review texts labelled with a concise positive or negative ruling on the movie, based on the score awarded by the author. The supervised learning task is thus a simple binary classification task.

As can be seen below, only the 30’000 most common words were kept, to prevent overfitting, which can easily happen when using one-hot encodings, and to keep the vectors at a reasonable size.

```
from tensorflow.keras.datasets import imdb

# take only 30000 most frequent words
vocab_size = 30000

(x_train, y_train), (x_test, y_test) = imdb.load_data(
   path='imdb.npz',
   num_words=vocab_size
)
```

Snippet 1: Loading the IMDB data set

The dataset was further prepared by splitting the training dataset into training and validation data, chopping of the training data into overlapping chunks of 100 words and batching it into batches of 64 samples.

```
# prepare dataset for training
(train_dataset, samples, batch_size, x_validation, y_validation, x_test,
y_test) = prepare_dataset(
```

```
   x_train, y_train, x_test, y_test, validation_size=1000, batch_size=128,
chop_size=100, overlapping_chops=True)
import random

# prepare the dataset by train/validation splitting, chopping of the training
data and batching
def prepare_dataset (x_train, y_train, x_test, y_test, validation_size=1000,
                     batch_size=128, chop_size=100, overlapping_chops=False):

   # initialize lists
   x_validation = []
   y_validation = []
   # tranform into lists to allow index deletion
   x_train = list(x_train)
   y_train = list(y_train)

   # pick random examples to use as validation dataset
   for i in range(validation_size):
       index = random.randint(0, len(x_train)-1)
       x_validation.append(x_train[index])
       del x_train[index]
       y_validation.append(y_train[index])
       del y_train[index]

   x_train = np.array(x_train)
   y_train = np.array(y_train)

   # chop training examples into fixed length texts
   # (to improve training speed and allow for batching in tensorflow)
   x_slices = []
   y_slices = []
   for i in range(len(x_train)):
       s = [x_train[i][j:j+chop_size] for j in range(0,
len(x_train[i])-chop_size+1, chop_size)]
       x_slices += s
       y_slices += [y_train[i]]*len(s) # 🚂

   # overlapping chops augment the data and allow for learning other temporal
dependencies,
   # cut out by simple chopping.
   # We decided to simply add chops starting from half the chop size, thus
creating double so much data
   if (overlapping_chops):
       s = [x_train[i][j+int(chop_size/2):j+int(chop_size/2)+chop_size]
```

```
            for j in range(0, len(x_train[i])-chop_size+1-int(chop_size/2)-1,
chop_size)]
        x_slices += s
        y_slices += [y_train[i]]*len(s) # 🚂


    # shuffle and batch the training dataset
    train_dataset = tf.data.Dataset.from_tensor_slices((x_slices,
y_slices)).shuffle(

buffer_size=len(x_slices)).batch(batch_size)

    samples = len(x_slices)

    return ((train_dataset, samples, batch_size, x_validation, y_validation,
x_test, y_test))
```

Snippet 2: Preparing the data set

Tested model architectures

We compared different model architectures.
The general model described by Johnson & Zhang uses region embedding by max-pooling over a certain number of outputs of the LSTM layer (the number of regions is the hyperparameter k). The results of the authors showed that the best performance on the IMDB dataset is reached by using k=1, i.e. max-pooling over the whole LSTM layer (the text is one single big region, encoded by a single output). We tried using different values for k.
Johnson & Zhang used a simplified LSTM model, removing both input and output gates. We tried using both LSTM cells (original definition) and GRU cells (simpler, slightly less time and memory consuming).

Code snippet 3 shows the one-hot GRU model using max-pooling over the whole text (k=1).

```
class oh2GRUp_whole_text (Layer):

    def __init__(self, units=256, nb_classes=2):
        super(oh2GRUp_whole_text, self).__init__()

        # forward GRU layer
        self.forward = GRU(units, return_sequences=True)
        # backwards GRU layer
        self.backward = GRU(units, return_sequences=True, go_backwards=True)
```

```
        # concatenation
        self.concat = Concatenate(axis=1)
        # dropout
        self.dropout = Dropout(0.4)
        # top dense layer
        self.dense = Dense(nb_classes, activation='softmax')


    def call(self, x):
        y1 = self.forward(x)
        y2 = self.backward(x)

        # total maxpooling since k=1
        maxForward = tf.reduce_max(y1, axis=1)
        maxBackward = tf.reduce_max(y2, axis=1)

        concatenated = self.concat([maxForward, maxBackward])
        return self.dense(self.dropout(concatenated))
```

Snippet 3: GRU-based model with k=1 max-pooling

Code snippet 4 shows the one-hot GRU model using regional max-pooling (k>1).

```
class oh2GRUp_k_regions(Layer):

    def __init__(self, k, units=128, nb_classes=2):
        super(oh2GRUp_k_regions, self).__init__()

        self.nb_classes = nb_classes
        self.k = k

        self.forward = GRU(units, return_sequences=True)
        self.backward = GRU(units, return_sequences=True, go_backwards=True)
        self.concat = Concatenate(axis=1)
        self.dropout = Dropout(0.4)
        self.dense = Dense(self.nb_classes, activation='softmax')


    def call(self, x):
        y1 = self.forward(x)
        y2 = self.backward(x)

        # split text regions
        y1 = [tf.split (y, [int(y1.shape[1]/(self.k))]*int(self.k) +
[y1.shape[1]-int(y1.shape[1]/self.k)*int(self.k)],
```

```
                        axis=0)[:-1] for y in y1]
        # regional maxpooling (region embedding)
        maxForward = tf.reshape(tf.reduce_max(y1, axis=2), shape=(len(y1),-1))

        # split text regions
        y2 = [tf.split (y, [int(y2.shape[1]/(self.k))]*int(self.k) +
[y2.shape[1]-int(y2.shape[1]/self.k)*int(self.k)],
                        axis=0)[:-1] for y in y2]
        # regional maxpooling (region embedding)
        maxBackward = tf.reshape(tf.reduce_max(y2, axis=2), shape=(len(y2), -1))

        concatenated = self.concat([maxForward, maxBackward])
        return self.dense(self.dropout(concatenated))
```

Snippet 4: GRU-based model with k>1 max-pooling

Training

For training, the training texts were chopped into segments of fixed size (100 words, see above). This enables significantly faster training times, since the training data can be batched (TensorFlow requires a rectangular input for batching), and parallelization can speed up the training time. Batching of texts of variable size, not immediately possible in TensorFlow, would be inefficient as the speed boost allowed by parallelization would be significantly reduced by the presence of some longer texts.

Additionally, we used overlapping text chunks for training. On the one hand this allows joining together words that would have been separated by non-overlapping chunking, losing potential semantic connections, and on the other hand augments the training data, which allows for generalization in a more meaningful way than possibly by merely adding more training epochs. However, Johnson and Zhang report that the improvement gained from using overlapping chunks is very small.

To evaluate the performance of the trained model on new data, we used periodic performance assessments on a validation dataset of 1000 samples. The size was kept small since the predictions were calculated without chopping, which made batching impossible and slowed down the process significantly.

The model contains different parameters, which were tuned to find the best-performing variant:

- K: the number of regions of text to be embedded
- The number of units in the LSTM layer
- The type of LSTM cell (we tested both on classic LSTM and GRU cells)

Additionally, the usual hyperparameters had to be tuned:

- Learning rate
- Batch size (for training)
- Optimizer (we used Adam)

The convergence turned out to be really quick: thus, we always kept the number of epochs small: 3-4 epochs when training without overlapping chops, and only 2 epochs when training with overlapping chunks.

```
import time

# train by one-hotting the input data

def train_one_hot (model, train_dataset, samples, batch_size, vocab_size,
x_validation, y_validation, nb_classes, learning_rate=0.002, epochs=3):

   optimizer = tf.keras.optimizers.Adam(learning_rate=learning_rate)
   cce_loss = tf.keras.losses.CategoricalCrossentropy()

   step = 0
   total_steps = (samples / batch_size) * epochs
   train_losses = []
   train_accuracies = []
   train_steps = []
   validation_accuracies = []
   progress = tqdm(total=total_steps)
   training_time = 0

   def results():
       plt.figure()
       line1, = plt.plot(train_steps, train_losses)
       plt.xlabel('training steps')
       plt.ylabel('loss')
       plt.show()

       plt.figure()
       line1, = plt.plot(train_steps, train_accuracies)
       plt.xlabel('training steps')
       plt.ylabel('training accuracy')
       plt.show()

       plt.figure()
       line1, = plt.plot(range(len(validation_accuracies)),
validation_accuracies)
       plt.xlabel('training steps')
       plt.ylabel('validation accuracy')
       plt.show()

   for epoch in range(epochs):
```

```
        for (x,t) in train_dataset:
            # Turn the labels into one-hot vectors.
            x = tf.one_hot(x, depth=vocab_size)
            t = tf.one_hot(t, depth=nb_classes)

            # Perform a training step.
            with tf.GradientTape() as tape:
                output = model(x)
                loss = cce_loss(t, output)
                gradients = tape.gradient(loss, model.trainable_variables)

            # Apply gradients.
            optimizer.apply_gradients(zip(gradients, model.trainable_variables))

            # Calculate the training accuracy
            accuracy = np.sum(np.argmax(t, axis=1) == np.argmax(output, axis=1))
/ t.shape[0]

            # Store loss
            train_losses.append(loss)

            train_accuracies.append(accuracy)

            train_steps.append(step)

            # evaluate accuracy on validation data
            if step % 100 == 0:
                if step!=0:
                    time_end = time.time()
                    training_time += time_end - time_start
                validation_accuracies.append(test_model(x_validation,
y_validation, model, vocab_size))
                results()
                time_start = time.time()

            progress.update(1)

            step += 1

    progress.close()

    return (validation_accuracies, training_time)
```

Snippet 5: Training

Testing

Testing was done on a held out portion of the IMDB dataset. We tried testing on both full-text and chunked data, with the same chunk size as used in training. As expected, the performance is significantly better when using full texts (variable length), indicating that the LSTM (respectively, GRU) cells generalize well to variable lengths.

```
def test_model (x_test, y_test, model, vocab_size):

   test_accuracies = []

   for (x,t) in zip(x_test, y_test):
       # Turn the labels into one-hot vectors.
       X = tf.expand_dims(tf.one_hot(x, depth=vocab_size), axis=0)
       T = tf.expand_dims(tf.one_hot(t, depth=2), axis=0)
       output = model(X)
       accuracy = np.sum(np.argmax(T, axis=1) == np.argmax(output, axis=1))
       test_accuracies.append(accuracy)

   print ("Test accuracy = {}".format(np.mean(np.array(test_accuracies))))
   return np.mean(np.array(test_accuracies))
```

Snippet 6: Testing

Results and Evaluation

Model	Validation accuracy	Testing accuracy
GRU with k=5 pooling, no dropout, batch_size=64, 128x2 units, chunk size=100	87.9 %	87.0 %
GRU with k=1 pooling, no dropout, batch_size=128, 256x2 units, chunk size=100	89.0 %	87.2 %
GRU with k=1, dropout=0.1, batch_size=64, 256x2 units, chunk size=50	91.0 %	87.8 %
GRU with k=1, dropout=0.1, batch_size=128, 512x2 units, chunk size=50	91.4 %	87.9 %

Discussion

Johnson and Zhang report error rates around 7%. With an error rate of 12-13%, we clearly weren't able to achieve the same performance in our setup. The reason for this may have been our use of slightly different implementations of recurrent layers for the region embedding, or the resource limitations of our computing platform Google Colaboratory, which made training with large batch sizes and many units in the recurrent layers impossible.

Bibliography

Bengio, Y. *et al.* (no date) 'Neural Probabilistic Language Models', *Innovations in Machine Learning*, pp. 137–186. doi: 10.1007/10985687_6.

Britz, D. (2015) *Recurrent Neural Network Tutorial, Part 4 – Implementing a GRU/LSTM RNN with Python and Theano*, *WildML*. Available at: http://www.wildml.com/2015/10/recurrent-neural-network-tutorial-part-4-implementing-a-grulstm-rnn-with-python-and-theano/ (Accessed: 15 April 2020).

Harris, D. and Harris, S. (2010) *Digital Design and Computer Architecture*. Morgan Kaufmann.

Heaton, J. (2018) 'Ian Goodfellow, Yoshua Bengio, and Aaron Courville: Deep learning', *Genetic Programming and Evolvable Machines*, pp. 305–307. doi: 10.1007/s10710-017-9314-z.

Hochreiter, S. and Schmidhuber, J. (1997) 'Long Short-Term Memory', *Neural Computation*, pp. 1735–1780. doi: 10.1162/neco.1997.9.8.1735.

Johnson, R. and Zhang, T. (2016) 'Supervised and Semi-Supervised Text Categorization using LSTM for Region Embeddings'. Available at: http://arxiv.org/abs/1602.02373 (Accessed: 15 April 2020).

Nguyen, M. (2018) *Illustrated Guide to LSTM's and GRU's: A step by step explanation*, *Medium*. Towards Data Science. Available at: https://towardsdatascience.com/illustrated-guide-to-lstms-and-gru-s-a-step-by-step-explanation-44e9eb85bf21 (Accessed: 15 April 2020).

Understanding LSTM Networks -- colah's blog (no date). Available at: http://colah.github.io/posts/2015-08-Understanding-LSTMs/ (Accessed: 15 April 2020).