NLP Assignment 3: Sentence Representations for Sentiment Analysis on Twitter

Victor Yip
15032476
Department of Computer Science
UCL
victor.yip.15@ucl.ac.uk

Tak Loon Ng Chan
15021013
Department of Computer Science
UCL
tak.chan.15@ucl.ac.uk

Agus Nur Hidayat
15054859
Department of Computer Science
UCL
agus.hidayat.15@ucl.ac.uk

Miroslav Krl
15082813
Department of Computer Science
UCL
miroslav.kral.15@ucl.ac.uk

Abstract

The aim of this work is to address and discuss problems of the third assignment of the course COMPGI19 Statistical NLP at University College London. This assignment relates to Deep learning and main task was to implement deep learning model that predict sentiment of tweets.

1 Problem 1: Gradient Checking

Gradient Checker makes sure **backward propagation(BP)** works properly by comparing the gradient returned by BP with the expected gradient. The expected gradient is calculated by:

$$\frac{\partial g_\theta}{\partial x_i} \approx \frac{g_\theta(x_i + \epsilon) - g_\theta(x_i - \epsilon)}{2 * \epsilon}$$

We approximated the gradient by calculating two new outputs of the forward function, based on inputs from a small increment and decrement to the original x. Since we can get the approximation of the gradient with the formula above, logically, we can use it to train our models instead of using BP. However, this is computationally expensive as forward function g has to be calculated three times at each node - once in **forward propagation (FP)** for $g_\theta(x_i)$, and twice for $g_\theta(x_i + \epsilon)$, $g_\theta(x_i - \epsilon)$. We will see in the proper BP implementation later, g is calculated once only in FP, and that result is **reused** in subsequent gradient calculation in BP.

Table 1: Average deviation from the approximated gradient of block implementations

BLOCK	AVERAGE DEVIATION
Sigmoid	1.6150E-10
Tanh	1.2838E-10

2 Problem 2: Sum-of-Word-Vectors Model

2.1 Blocks

2.1.1 Vector Parameter

It is a trainable vector parameter that takes two inputs, vector dimension and gradient clip range. It will initialize the value of variable **param** with random values and set variable **gradParam** into zero vector. There are five methods in this class. Method **forward** stores the value of **param** into variable **output** and using it as the return value. Method **backward** accumulates the gradient input into variable **gradParam**. Method **resetGradient** reassigns the value of variable **gradParam** into zero. Method **update** clips the gradient with input clip range and updates **param** by subtracting itself with the multiplication of clipped gradient by input **learning rate (LR)**. Method **initialize** set random values for **param** using **gaussian** sampling function.

2.1.2 Vector Sum

It is the summation of vectors that takes sequence of vectors as the inputs. There are three methods in this class. Method *forward* accumulates the value of FP of all input vectors into variable **output**. During the first initialization of **output**, forward().copy is called instead of forward because original vector that came from the input to the function should not be modified since vector is object in Scala. Method forward.copy() will create another vector with the same values as the original one. Then, the rest of will be summed up over this new vector. Method *backward* calls BP of all input vectors with upstream gradient as the input. Method *update* calls update of all input vectors with input LR.

2.1.3 Dot Product

It is the dot multiplication between two vectors that takes two vectors **x** and **y** as the inputs. There are three methods in this class. Method *forward* stores the result of dot multiplication between **x** and **y** into variable **output.** In method *backward*, the derivative of **x** dot **y** w.r.t. **x** is **y**and the derivative of **x** dot **y** w.r.t. **y** is **x**. Thus, BP of **x** is called using the multiplication of upstream gradient by the output of **y.**. Next, BP of **y** is called using the multiplication of upstream gradient by the output of **x.** Ina mathematical notation, derivation of dot product can be described as follows:

$$z\frac{\partial x \cdot y}{\partial x} = zy$$
$$z\frac{\partial x \cdot y}{\partial x} = zx$$

Method *update* calls update of **x** and **y** with input LR.

2.1.4 Sigmoid

It is the sigmoid function $g(x)$ that takes one scalar value x as an input. Below is the formula of $g(x)$:

$$g(x) = \frac{1}{1+e^{-x}}$$

There are three methods in this class. Method *forward* stores the result of **g** into variable **output**. Method *backward* calls BP of **x** using the multiplication of upstream gradient by the derivative of **g** w.r.t. **x**. Below is the derivative of **g(x)**:

$$z\,\frac{\partial g(x)}{\partial x} = z\,\frac{\partial \frac{1}{1+e^{-x}}}{\partial x} = z\;g(x)(1-g(x)$$

Method *update* updates parameter of x with input LR.

2.1.5 Negative Log-Likelihood

It is the negative log-likelihood loss function $L(f(x), y)$ that takes two inputs, an actual value **y** and a prediction **f(x)**. This value can be either 0 (FALSE) or 1 (TRUE). Below is the formula of $L(f(x), y)$:

$$L(f_\theta(x), y) = -y * log(f_\theta(x)) - (1 - y) * log(1 - f_\theta(x))$$

Method*forward*: storing the result of $L(f(x), y)$ into variable **output**. Method *backward* calls BP of $f(x)$ using multiplication of upstream gradient by the derivative of $L(f(x), y)$ with respect to $f(x)$. Below is the derivative of $L(f(x), y)$:

$$\frac{\partial L(f(x), y)}{\partial f(x)} = \frac{y - 1}{f(x) - 1} - \frac{y}{f(x)}$$

Method *update* updates parameter of $f(x)$ using input LR.

2.1.6 l_2 Regularization

It is the regularization $R(\theta)$ with the norm value equals to 2 that takes two inputs, a regularization strength and a sequence of vectors or matrices. Below is the formula to calculate $R(\theta)$:

$$R(\theta) = \lambda * \frac{1}{2} * \|\theta\|_2^2$$

θ is the sequence of vectors or matrices of parameters (list of words representations concatenated with global weight). λ is the regularization strength. There are three methods in this class. Method *forward*implements the regularization formula described above and returns sum of the losses. It calculates loos for each argument separately and sum in the end. Method *backward* calculates gradient of regularization function $R(\theta)$ w.r.t. θ. Formula bellow capture this derivation.

$$\frac{\partial R(\theta)}{\partial \theta} = \frac{\partial \lambda * \frac{1}{2} * \|\theta\|_2^2}{\partial \theta} = \lambda * \theta$$

Method *update* updates the values of θ with input LR.

2.2 Model

2.2.1 Sum-of-Word Vectors

The model takes two inputs, dimension of the word vector representation and the regularization strength. There are 1 variable and 4 methods inside the model. Variable *vectorParams* initializes the LookUp table that will be used to keep track of the word representations. A global parameter vector *w* is added into the LookUp Table in the beginning. Method *wordToVector* retrieves the Vector representation of a word. It will check whether the vector representation is available in the LookUp table or not. If it's available, then it will call it using the word as the key. If it's not available, then it will store it into the LookUp table and return the value of the vector representation. Method *wordVectorsToSentenceVector* puts a sequence of words into sentence by accumulating every word vector representation into a new sentence vector representation. Method *scoreSentence* predicts the sentence's score using sigmoid function. Methods *regularizer* concatenate the global parameter vector *w* with a sequence of word vectors and then use it together with regularization strength as inputs for calling L2Regularization. It also inherits method *loss* from class Model that accumulates the error from negative log-likelihood loss function and regularizer. Below is the result of Gradient Checking for this model:

Model inputs: dimension = 10; regularization strength=0.1;

Table 2: Results of gradient checking

METHOD	INPUTS	AVERAGE ERRORS
scoreSentence	Sentence Vector	2.52E-10
wordVectorsToSentenceVector	4 word-vectors	1.02E-10, 1.08E-10, 1.02E-10, 0, 1.02E-10
regularizer	A sequence of word-vectors	1.24E-10
loss	A sequence of words, target	6.92E-10, 8.75E-10, 7.82E-10, 4.12E-10, 8.62E-10

2.2.2 Stochastic Gradient Descent

Stochastic gradient descent is one type of gradient descent algorithms. The goal is to retrieve parameters that will cause the loss function to return the minimum error rate. Derivation:

$$\begin{aligned}\frac{\partial}{\partial\theta_j}J(\theta) &= \frac{\partial}{\partial\theta_j}\frac{1}{2}(h_\theta(x)-y)^2\\ &= 2*\frac{1}{2}(h_\theta(x)-y)*\frac{\partial}{\partial\theta_j}(h_\theta(x)-y)\\ &= (h_\theta(x)-y)*\frac{\partial}{\partial\theta_j}(\sum_{i=0}^{n}\theta_i x_i - y)\\ &= (h_\theta(x)-y)x_j\end{aligned}$$

We improved our implementation to take advantage of the loss function we built in Model. Initially, we were calling generic blocks, vectorParam, dot and sigmoid and their respective forward and backward functions. It had resulted in not only much more complicated codes, but also significantly longer runtime as well as codes that were difficult to debug.

2.3 Grid Search and Evaluation

Grid search is exhaustive search method that allows us to optimize hyper-parameters of the model. Hyper-parameters of our models, in our case are learning rate, value of regularization parameter and dimension of weight vectors. The way how grid search works is that we prepared arrays of possible values of the hyper-parameters and then train and evaluate models with all possible combination of the possible parameters. Combination that returns maximal value of **development set accuracy** is considered as a optimal.

In case of SumOfWordVectors model, the parameters that yield best dev set accuracy of 78% are following:

$$\begin{aligned}learningRate &= 0.01\\ vectorDimension &= 10\\ regularizationStrength &= 0.1\\ epochs &= 23\end{aligned}$$

We also used regularization by ”early stopping” because the best dev. accuracy is reached with smaller number of epochs then default 100.

2.4 Loss Analysis

Loss and also train and development accuracy depends on the epoch. To analyse loss and accuracy, we trained model with best parameters founded by grid search and plot dependency of loss and accuracy on number of epochs. On the upper plot of Figure 1 below we can see that curves from development set and training set are different. While curve for development set grows until epoch number 23 when reaches its maximum of 78%, curve of train set steadily and reached accuracy of approximately 95% after 100 epochs. Based on this plot we can also conclude that this model suffer from over-fitting. However, to the epoch number 23 (the epoch, that ensure best dev. accuracy) the phenomena of over-fitting is not so significant.

Figure 1 captures dependency of accumulated loss on number of epochs. Accumulated loss decreases in every single epoch.

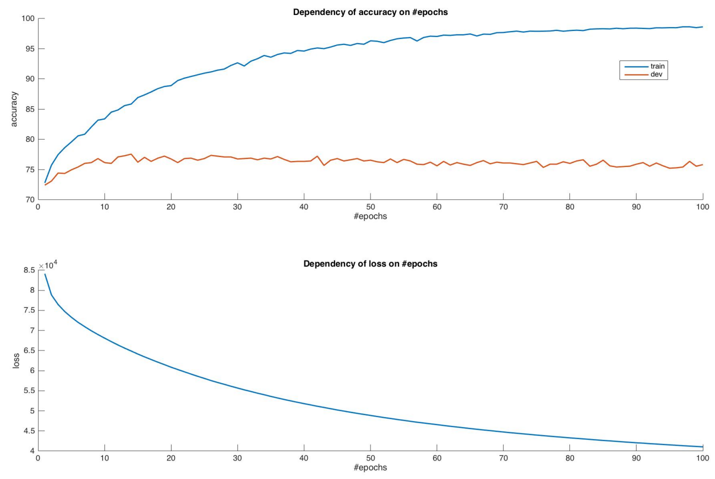

Figure 1: Dependency of loos, train and develepment accuracy on number of epochs

2.5 t-SNE Visualization

Figure 2 captures 100 randomly selected words from the whole data set and their position in the 2-dimensional vector space. For each of these words was also calculated score (we assumed the sentence is made up from a single word and the score was calculated for this single-word sentence) and the color of the markers in plot shows how positive or negative sentiment particular word has. We can clearly see that negative words are located close to each other and also positive ones make clusters. We can also see that the calculated score is mostly reasonable, beyond negative words belongs for example 'good-by' or 'sad' and beyond positive ones 'Uhm.)' for instance. However, even in this 100 words sample we can find suprising insights - word 'wife' or 'Everton' are considered as strongly negative.

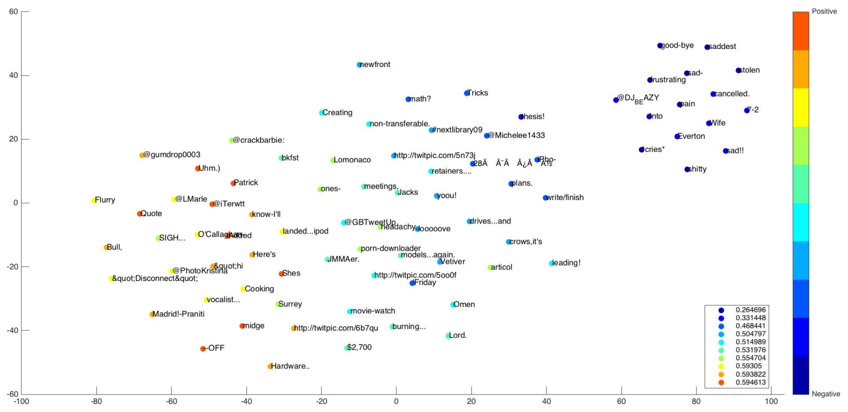

Figure 2: t-SNE Visualization of sample words

3 Problem 3: Recurrent Neural Networks

3.1 Blocks

3.1.1 Matrix Parameter

It is the matrix parameter that takes three inputs, first and second dimension of the matrix, and the range to clip the gradient. It will initialize the value of variable **param** with random values and set variable **gradParam** into zero matrix. There are five methods in this class. Method *forward* stores the value of **param** into variable **output** and uses it as the return value. Method *backward* accumulates the gradient input into variable **gradParam**. Method *resetGradient* reassigns the value of **gradParam** to zero. Method *update* clips the gradient with clip input range and updates the **param** by subtracting itself with the multiplication of clipped gradient by input LR. Method *initialize* returns random values for **param** with *gaussian* sampling function.

3.1.2 Matrix-Vector Multiplication

This is a class representation of the multiplication between a matrix and a vector that takes one matrix M and one vector v as the inputs. There are three methods in this class. Method *forward* stores the result of multiplication between M and v into variable **output**. In method *backward*, the derivative of $M * v$ w.r.t M is v and the derivative of $M * v$ w.r.t v is M. Thus, BP of M is called using the outer product between the upstream gradient and the output of v. Next,BP of v is called using the multiplication between the transpose of output of M and the upstream gradient. This is captured in equation bellow:

$$z\,\frac{\partial M\ *\ v}{\partial v} = v\ *\ M$$
$$z\,\frac{\partial M\ *\ v}{\partial M} = v\ *\ v$$

Method *update* calls update of x and y with input LR.

3.1.3 Tanh

It is the tanh function $t(x)$ that takes one vector $f(x)$ as an input. Below is the formula of $t(x)$:

$$tanh(x) = \frac{2}{1 + e^{-2x}} - 1$$

There are three methods in this class. Method *forward* stores the result of $tanh(x)$ into variable **output**. Method *backward* calls BP of x using the multiplication of upstream gradient by the derivative of $tanh(x)$ w.r.t.x. Below is the derivative of $tanh(x)$:

$$z\,\frac{\partial tanh(x)}{\partial x} = z\,1 - tanh(x)^2$$

Method *update* calls update of x with input LR.

3.2 Model

The model takes four inputs, dimension of the word vector representation, dimension of the hidden state vector, regularization strength for the word vectors and global parameter vector *w* and the regularization strength for the transition matrices. There are 2 variables and 4 methods inside the model. Variable *vectorParams* initializes the LookUp table that will be used to keep track of the word representations. Vector *w*, *h0*, and *b* with hidden state size as the dimension are added into the LookUp Table. Value of vector *b* is set into 1.0. There is also a map *matrixParams* that stores two transition matrices *Wh* (with dimension = wordVector x hiddenState) and *Wx* (with dimension = hiddenState x hiddenState). Method *wordToVector* retrieves the Vector representation of a particular word. Firstly, it will check whether the vector representation is available in the lookup table or not. If it's available, then it will call it using the word as the key. If it's not available, then it will store it into the lookup table and return the value of the vector representation. Method *wordVectorsToSentenceVector* is based on the formula below:

$$h_t = tanh(W^h h_{t-1} + W^x x_t + b)$$

It puts a sequence of words into sentence by calling tanh function iteratively. In the first iteration, it will call tanh with input accumulation of: multiplication between matrix *Wh* and vector *h0*; multiplication between matrix *Wx* and word vector at index 0; and vector parameter *b*. Within the next iteration **i**, it will call tanh with input accumulation of: multiplication between matrix *Wh* and previous tanh result; multiplication between matrix *Wx* and word vector at index **i**; and vector parameter *b*. Method *scoreSentence* predicts the score of the sentence using sigmoid activation function. Methods *regularizer* calls L2Regularization twice. The first one with input vector regularization strength and concatenation of global parameter w and the word vectors. The second one with input matrix regularization strength, and two matrices (*Wh* and *Wx*). It also inherits method *loss* from class Model that accumulates the error from regularized negative log-likelihood loss function. Below is the result of Gradient Checking for this model:

Model inputs: dimension = 10; size of hidden state = 10; vector regularization strength=0.1; matrix regularization strength = 0.1;

Table 3: Results of gradient checking in RNN

METHOD	INPUTS	AVERAGE ERRORS
scoreSentence	Sentence Vector	1.55E-10
wordToSentenceVector	4 word-vectors	3.39E-10, 2.24E-10, 2.27E-10, 2.82E-10, 2.28E-10, 0, 2.19E-10, 2.72E-09, 2.50E-09
regularizer	A sequence of word-vectors	9.46E-11, 1.72E-10, 1.11E-10, 1.52E-10
loss	A sequence of words, target	4.08E-10, 5.82E-10, 4.38E-10, 4.18E-10, 3.45E-10, 4.70E-10, 3.66E-10, 6.11E-09, 5.48E-09

3.3 Grid Search, Initialization, Evaluation and Comparison

The algorithm of grid search for Problem3 is similar to the one in Problem2. The only difference is the hyper-parameters. In this case, the hyper-parameters of the model are learning rate, regularization strength for both word vectors and transition matrices, the dimension of the word vectors, and the dimension of the hidden state vectors. Combination that returns maximal value of **development set accuracy** is considered as the best ones.

For each of the hyper-parameters we tried 4 to 5 different values. Find bellow values we used in grid search.

$$\begin{aligned} learningRate &= [0.1, 1e-2, 1e-3, 1e-4] \\ vectorDimension &= [10, 15, 20, 25, 30] \\ hiddenStateDimension &= [10, 20, 30] \\ vectorRegularizationStrength &= [0.1, 1e-2, 1e-3, 1e-4, 1e-5] \\ matrixRegularizationStrength &= [0.1, 1e-2, 1e-3, 1e-4] \\ epochs &= 30 \end{aligned}$$

With RNN model, the parameters that yield best dev set accuracy of 77,4% are following:

$$\begin{aligned} learningRate &= 0.001 \\ vectorDimension &= 25 \\ hiddenStateDimension &= 30 \\ vectorRegularizationStrength &= 1e-4 \\ matrixRegularizationStrength &= 1e-4 \\ epochs &= 24 \end{aligned}$$

Hyper-parameters are similar to the one in SumOfWordVectors model, we also used regularization by ”early stopping”.

We also have tried different strategies how to initialize weight matrices as well. We tried to multiply values of random initial weights by $\frac{2}{\sqrt[2]{n}}$ where n is dimensionality of word vector representation. In math notation, ij element of matrix W was initialize as follows:

$$W_{ij} = \frac{rand}{\sqrt[2]{n}}$$

Random function $rand$ sample from Gaussian distribution with $\mu = 0$. This type of initialisation is generally can be concern as optimal because initial weights are small enough and ”and empirically improves the rate of convergence” [1].Different initialization of weight matrices affect the slope of accuracy curve, however, the best score remains almost unchanged.

In comparison with SumOfWordVector, RNN model gives almost the same results. However, this ’low’ accuracy can be also caused by difficult process of training and optimizing performance of RNN model.

4 Problem 4: Dropout, Pre-Trained word vectors and Sum-Mulplication of word vectors models

For problem 4, we tried 3 different models. Unfortunately, none of them worked as we expected. We first tried to implement Dropout in order to avoid over fit in the RNN model (Srivastava, Hinton, Krizhevsky, Sutskever, & Salakhutdinov, 2014). During training, the forward pass set to 0 elements in the sentence vector (last call of Tanh). We used 0.2 as the dropout probability. A dropout vector was generated in the Dropout block to remember which elements were set to 0. As back propagation, element wise multiplication was applied to the upstream gradient by using the dropout vector.

During test time, the forward pass returned the sentence vector multiplied by (1 - dropout probability). The back propagation multiplied the upstream gradient by (1 - dropout probability). We ran the RNN with Dropout regularization, but the dev accuracy never went over 60

Next, we tried pre-trained word vectors on the Sum of Word vectors and RNN models. The idea was that a better word to vector algorithm could give us better accuracies. Google’s Word2Vec was used to generate the word vectors on the training corpus (Google, 2013). We set the pre-trained word vectors as fixed in the LookupTable and trained them. On both models, the accuracies on dev started around 70, climbed to 71-72 in the next 1 or 2 epochs and fell back again to 70. We saw this pattern again and again as more epochs were ran.

Finally, we tested with the sum and multiplication of word vectors. As pointed out by Mitchell, multiplication of word vectors allows the model to scale up relevant elements in the word vectors, rather than just to sum them up. Multiplication has the side effect of missing all the contributions of the i elements in the word vectors if in any of them has a value of 0. Thus, we combined multiplication and sum to get the benefit of multiplication and to avoid the problem of 0 in the elements of the word vectors (Mitchell & Lapata, 2008). During test time, this model did not perform better than the Sum of Word vectors model.

Reference

[1] karpathy@cs.stanford.edu (2016) CS231n Convolutional neural networks for visual recognition. Available at: http://cs231n.github.io/neural-networks-2/ (Accessed: 29 January 2016).

[2] Google. (2013, 07 30). Word2vec. Retrieved from Google Code: https://code.google.com/archive/p/word2vec/

[3] Mitchell, J., & Lapata, M. (2008). Vector-based Models of Semantic Composition. ACL-08: HLT (p. 236). Columbus: Association for Computational Linguistics.

[4] Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. (2014). Dropout: A Simple Way to Prevent Neural Networks from Overfitting. Journal of Machine Learning Research, 1929.