COMPGI19 Assignment 3 Report

Reece Doyle **Navid Hallajian** **Kapileshwar Syamasundar** **Oliver Tun**

1 Gradient Checking

Using the approximation instead of backpropagation means we lose intermediate derivatives resulting in recalculating derivatives and wasting computation.

Consider that we instead use backpropogation but use the formula for approximating the gradient as opposed to the calculated derivative. The derivative of the model is calculated at every block comprising it, and each of these blocks will introduce an error. Calculating the model's gradient multiplies these gradients together (as in chain rule), thus potentially introducing errors significant enough to change the classification.

2 Sum-of-Word-Vectors Model

2.1 Blocks

1. Dot product

$$\frac{\partial \mathbf{x} \cdot \mathbf{y}}{\partial \mathbf{x}} = \mathbf{y}$$

$$\frac{\partial \mathbf{x} \cdot \mathbf{y}}{\partial \mathbf{y}} = \mathbf{x}$$

2. Sigmoid

$$\frac{\partial \text{sigmoid}(x)}{\partial x} = \text{sigmoid}(x)(1 - \text{sigmoid}(x))$$

3. Negative log-likelihood

$$\frac{\partial \mathcal{L}(f_\theta(x), y)}{\partial f_\theta(x)} = \frac{-y}{f_\theta(x)} + \frac{1-y}{1-f_\theta(x)}$$

4. Regularisation

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

2.2 Model

We get an error of 7.042×10^{-11} for the sum of words vector model.

2.3 Grid Search and Evaluation

2.4 Loss Analysis

2.5 t-SNE Visualisation

We round the values of the predictions to get a binary classification. Blue data points represent a positive classification and orange is negative.

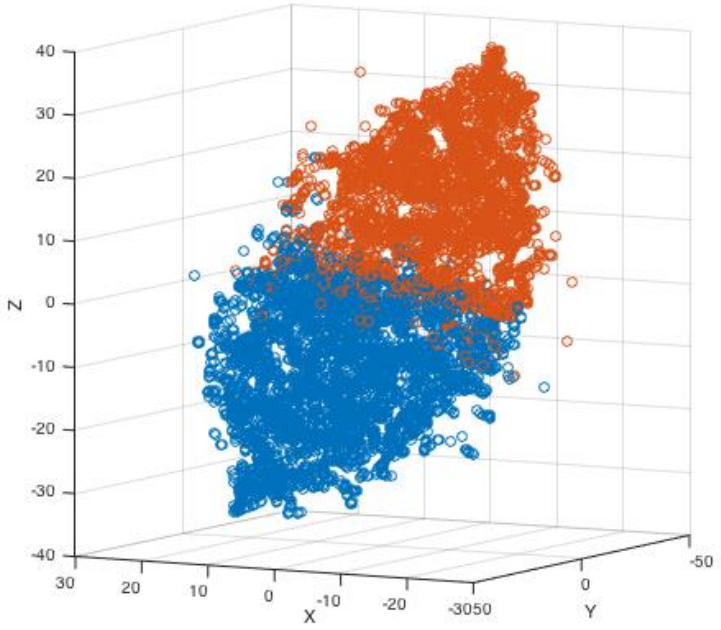

Figure 1: t-SNE visualisation.

3 Recurrent Neural Networks

3.1 Blocks

1. Matrix-vector multiplication

$$\mathbf{z}\frac{\partial \mathbf{W}\mathbf{x}}{\partial \mathbf{x}} = \mathbf{W}^{\mathrm{T}}\mathbf{z}$$

$$\mathbf{z}\frac{\partial \mathbf{W}\mathbf{x}}{\partial \mathbf{W}} = \mathbf{z} \otimes \mathbf{x}$$

2. tanh

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

3.2 Model

We get an error of 6.301×10^{-11} for the RNN model.

3.3 Grid Search, Initialisation, Evaluation and Comparison

4 Sky's the Limit

4.1 Dropout regularisation

Dropout regularisation works by dropping connections between nodes of the neural network during training time, changing the topology of the neural network between each epoch. After training is completed, the neural network reconnects all of the nodes as normal.

Dropout regularisation reduces the amount of overtraining experienced by the neural network and prevents co-adaptation from occurring. It does this by having the approximate effect of training an exponentially large amount of smaller and more focused neural nets, combining them into a larger neural net during test time.

The implementation involved adding to the Blocks class by implementing dropout blocks. Each dropout block is given a probability and randomly generates a double between 0 and 1 on each update function. If the randomly generated value is greater than the probability value given, the dropout block will return a zero vector when the forward function is called and perform no actions when the backward function is called. Otherwise, the dropout block will act transparently and pass values between nodes as if it did not exist.

During testing time, the weights from the dropout model are copied into a normal SumOfVectors model (no dropout blocks) which is used for evaluation. Using the optimal parameters that we obtained for the original SumOfVectors model, we achieve an accuracy of 77.1% on the development set.

4.2 Multiplication-of-Word-Vectors Model

This model was implemented by judging the score of a sentence by taking the element-wise product of the outputs produced by the nodes as opposed to summing them as performed by the sum-of-word-vectors model.

Upon comparison of the two models, it appeared that the sum model performed more effectively than the multiplicative model over a period of 10 epochs and 100 epochs. The accuracy on the development set provided by the multiplication model was 59.25%, lower than what we achieved for sum-of-word-vectors.

4.3 Long Short-Term Memory RNN (LSTM)

4.3.1 Blocks

1. Concat

   This block represents the vertical concatenation of two vectors to one vector. The backwards gradient is calculated by splitting the upstream gradient into two vectors that are the same length as the two arguments.

2. Per-element sigmoid

   This block applies the sigmoid function to each element of a vector to produce a new vector. The derivative for each element is equivalent to the derivative given for sigmoid in 2.1.2.

3. Element-wise vector multiplication

   This block takes two vector blocks as arguments and applies element-wise multiplication, similar to the Hadamard product of matrices. Note that in the derivative given below, $\odot$ is the element-wise multiplication operator.

$$\mathbf{z}\frac{\partial \mathbf{x} \odot \mathbf{y}}{\partial \mathbf{x}} = \mathbf{z} \odot \mathbf{y}$$

4.3.2 Model

LSTMs are capable of learning long-term dependencies. One problem with standard RNNs is that with each new word, the impact of a prior word on the current hidden state becomes increasingly diluted, meaning it can be hard to learn dependencies between distant words in a tweet. One example in which it is useful to learn such a dependency is the tweet

> I really don't, and I'm not just saying this because it's a great example and I couldn't think of anything else, like Scala.

In this example, it is useful to remember that *don't* occurred before *like* when conducting sentiment analysis. The LSTM uses an additional cell state to which it can write information about the word vectors and hidden weights to help form these dependencies.

4.4 Loading pre-trained word2vec vectors

When we encounter a word for the first time, we generate its vector representation as a set of random values. wor2vec is a tool that is able to create representative word embeddings given a corpus. Our intuition is that representative embeddings will cause our model to converge to the optimum development accuracy faster since it would not need as many epochs to learn word embeddings as compared to a random embedding.

word2vec has several models that can achieve this result, but we decided to use the skip-gram model. This model considers the probability $p(c|w)$ where c is a context and w is a word.

To use word2vec, we used a Python interface over the original C library to pass in our training data and generate a CSV file with words and their vector representations. We read this file into our model's lookup table.

Using word2vec on our sum-of-word-vectors model resulted in almost the same results.