Assignment 3:
Sentence Representations for Sentiment Analysis on Twitter

Tim Rocktäschel, Sebastian Riedel

Due: 1st February 2016

January 1, 2016

1 Introduction

In this assignment we will investigate representation learning for sentiment analysis of tweets. To this end, we will build our own simple deep learning framework and learn task-specific word and sentence representations.

2 Rules

You are expected to work in **teams of up to 4 people**. You will write one joint report, and you are free to allocate responsibility as you see fit.

To get a feeling for "low-level" NLP implementation, for this assignment you are not allowed to use ANY existing NLP packages, outside of the code provided and the `stat-nlp-book` code. The general UCL plagiarism rules apply.

2.1 Report and Format

Your submission needs to consist of a report, in which you will summarise what you did and how you addressed the posed questions, and the code you hacked (filled in code stubs).

You will use the NIPS[1] style files for writing your reports. You can find out more about these styles on `https://nips.cc/Conferences/2015/PaperInformation/StyleFiles`. There are LaTeX, Word and RTF templates available. Your submission should be 6-8 pages maximum, with maximum 2 additional pages for references. Images do not count against the maximum. Rules written here trump all the non-formatting instructions in the NIPS style files. If something is not clear regarding the style and instructions for the report, please post your question to Moodle. You need to submit a camera-ready version (not anonymised!) in PDF format, strictly.

2.2 Submission

Please upload your submission to Moodle as a zip file, containing: your report in PDF format (including the names of all members of your team); your code in a zip file; **and your predictions for the provided test set**. The code you send needs to be compilable, as we will execute it on the test set.

3 Background

Advances in deep learning had a substantial impact on the field of natural language processing in recent years. At the heart of these methods lie computational frameworks that automatically differentiate parametrized functions and learn these functions using continuous optimization. In this assignment we are building (a light variant) of such a framework and apply it to the task of classifying the sentiment of twitter messages. In contrast to the previous assignment, we are not assuming any linguistic structure in these tweets apart from a sequence of symbols. Thus, we will not hand-craft any features but instead learn word representations (also called embeddings), i.e., fixed-length dense vectors. Furthermore, we seek to compose these word vectors to sentence vectors that capture the semantics of the tweet sufficient for classifying whether the sentiment of the tweet is positive or negative. Specifically, we will train a logistic regression classifier only based on the sentence vector.

[1] Conference and Workshop on Neural Information Processing Systems (NIPS) is one of the top conferences in Machine Learning

More formally, our models will be expressed as $f_\theta(x_1, \ldots, x_N)$ where $(x_1, \ldots, x_N)$ is the sequence of words in a tweet and θ are the parameters of the model. Given a loss function $\mathcal{L}$ (e.g. negative log-likelihood) and the target class y of a tweet (true/1 for positive sentiment, false/0 for negative sentiment), we will use stochastic gradient descent to learn the parameters of the model. To this end, we need to be able to calculate $\frac{\partial \mathcal{L}(f_\theta(x_1,\ldots,x_N),y)}{\partial \theta}$, i.e, the gradient of the parameters with respect to the loss. As our models will become quite complex and we want to investigate a variety of models, it becomes infeasible to hard-code these gradients. Instead, we are going to employ backpropagation to automatically calculate the gradients of the parameters. Specifically, for every sub-function g_θ (e.g. vector addition, sigmoid etc.) that we want to use in f_θ, we need to be able to calculate the output of that function given inputs and current parameters (forward pass) and calculate the gradient of the inputs to the function and its parameters based on an upstream gradient ∂z (backward pass). This is illustrated in Figure 1. Once we have implemented the forward and backward pass for a set of functions, we can compose these functions in many ways to build powerful models.

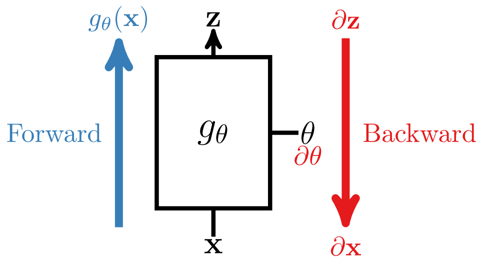

Figure 1: Methods of a Block for backpropagation.

3.1 Provided Code

3.1.1 Blocks

We already provide you with a scaffold for backpropagation in `Block.scala`. In this file you will find the trait `Block[P]`. Every sub-function that we want to use in our models will extend this trait. The type parameter `P` in `Block[P]` enables us to extend functions with different output domains from this trait. For instance, the result of a summation of two Doubles is a Double, hence a `Block` representing this operation is `Block[Double]`. In contrast, a `Block` representing the summation of vectors is a `Block[Vector]`. Every `Block` has to provide three functions:

forward This simply applies the function represented by the `Block`. Furthermore, this function will cache its result in the variable `output` as we might frequently access it in the backward pass.

backward Given an upstream `gradient`, backward calculates the gradient of its parameters and inputs. If the `Block` is a `ParamBlock` (see below), the gradient is accumulated (added to) a `gradParam` member; otherwise we call backward on the inputs of the `Block` and pass the calculated grardient as upstream gradient to these inputs. Note that backward assumes that forward has been called first!

update If the block is a `ParamBlock` this function should update the parameters `param` based on the accumulated gradients in `gradParam` and then reset `gradParam` (i.e. set to zero); otherwise, we call update on the inputs of the `Block`.

3.1.2 Constants and Parameters

We provide two trivial blocks representing constant doubles and vectors (see `DoubleConstant` and `VectorConstant`). The only functionality they provide is to return their constant value when forward is called. In contrast to constants, parameters are trainable and hence their return value of forward changes during training. Every parameter has to extend the provided trait `ParamBlock[P]`.

3.1.3 Models

We provide a trait `Model` for models that you will develop in this assignment in `Model.scala`. Every model needs to implement the following functions:

wordToVector maps a word String to its vector representation.

wordVectorsToSentenceVector composes a sequence of word vectors to a sentence vector.

scoreSentence provides a score between 0.0 and 1.0 for a sentence vector indicating whether the sentence is of positive (1.0) or negative (0.0) sentiment.

regularizer operates on the vector and matrix parameters of a model.

Furthermore, the following functions are already implemented but depend on the correct implementation of the functions above:

loss composes a `Block` that represents the training loss.

predict Given a sequence of words, this functions predicts whether the tweet is of positive sentiment.

3.2 Data and Evaluation

The data can be found in `./data/assignment3` in your project directory. You can access this corpus via `SentimentAnalysisCorpus.scala`. **WARNING**: The corpus we give you contains real tweets. This means there are a lot of spam links to potentially malicious websites. Don't click on them unless you know what you are doing! For evaluation we provide a function in `Evaluation.scala` that takes a trained model and the name of a corpus ("train", "dev" or "debug") and calculates the accuracy on that corpus. Together with your report and code, we ask you to take your best model and predict the sentiment on a test set that we provide. **In your zip file, provide a file named `predictions.txt` where every line corresponds to the tweet in the same line in `test.tsv` and shows the prediction for that tweet (1 for positive sentiment, 0 for negative sentiment)!**

4 Problems

4.1 Warm-up [0pts]

We are using Breeze (`https://github.com/scalanlp/breeze`) as linear algebra library for this assignment. Make yourself familiar with the functionality Breeze provides. Specifically, have a look at the linear algebra cheat sheet (`https://github.com/scalanlp/breeze/wiki/Linear-Algebra-Cheat-Sheet`), the quickstart guide (`https://github.com/scalanlp/breeze/wiki/Quickstart`) and the worksheet we provide (`Breeze.sc`).

4.2 Problem 1: Gradient Checking [3pts]

Before implementing blocks it is a good idea to think of ways to automatically test whether our implementations of the backward pass is correct. We can use gradient checking to approximate the true gradient numerically and then test whether our implementation gives the same result up to some error bound (for an introduction to gradient checking see `http://cs231n.github.io/neural-networks-3/#gradcheck`). Specifically, given a function $g_\theta(x)$ that depends on an input vector x we can approximate $\frac{\partial g_\theta(x)}{\partial x_i}$ as follows:

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

where $+_i$ and $-_i$ represent the addition to and subtraction from the ith component respectively.

We provide a partial implementation of gradient checking in `GradientChecker.scala`. You will find a function `wiggledForward` which given an index i and a value γ calculates $g_\theta(x +_i \gamma)$. Your task is to complete this implementation by calculating the expected gradient. Make sure to debug and test this implementation thoroughly as once you have gradient checking working you can test the correctness of your subsequent block implementations yourself.

Since we can approximate the gradient with the formula above up to an arbitrary precision we could simply use it for training our models instead of backpropagation. Explain why this would be a bad idea. In addition, report the average deviation from the approximated gradient of a model/block of your choice.

4.3 Problem 2: Sum-of-Word-Vectors Model [40pts]

The first model we will implement is a sum-of-word-vectors model, which is also called bag-of-word-vectors as it does not consider the order in which words appear:

$$f_\theta(x_1, ..., x_N) = \text{sigmoid}\left(\mathbf{w} \cdot \sum_i \mathbf{v}_{\mathrm{x}_i}\right)$$

where $\mathbf{v}_{\mathrm{x}_i}$ is the vector representing the ith word in the sequence, $\mathbf{w}$ is a global weight vector and $\cdot$ denotes the vector dot product.

4.3.1 Blocks

Vector Parameter [4pts] Before we can train any model, we need to implement a trainable parameter block. Complete the implementation of `VectorParam`. This block will be used for $\mathbf{w}$ and $\mathbf{v}_{\mathrm{x}_i}$. **A useful optimization trick is gradient clipping: before we update the parameters we make sure that the gradient for every component in the vector lies in the range $(-clip, clip)$ where $clip \in \mathbb{R}$ (use the function `breeze.linalg.clip` to achieve this).** Implementation hint: `:+=`, `:-=` and `:*=` are useful Breeze in-place operations.

Vector Sum [3pts] Complete the implementation of the block `Sum`, which represents vector summation. This block takes an arbitrary long sequence of vectors and sums them up.

Dot Product [4pts] Complete the implementation of `Dot`, representing the dot product $\mathbf{x} \cdot \mathbf{y}$ of two vectors. Write the solution to $\mathbf{z}\frac{\partial \mathbf{x}\cdot\mathbf{y}}{\partial \mathbf{x}}$ and $\mathbf{z}\frac{\partial \mathbf{x}\cdot\mathbf{y}}{\partial \mathbf{y}}$ in your report ($\mathbf{z}$ is the upstream gradient from a block that uses the dot product).

Sigmoid [3pts] Complete the implementation of the block `Sigmoid`, which represents the application of the sigmoid function to a scalar value, and report the solution to $z\frac{\partial \text{sigmoid}(x)}{\partial x}$.

Negative Log-Likelihood [4pts] As training loss we will use negative log-likelihood:

$$\mathcal{L}(f_\theta(x), y) = -y * \log\left(f_\theta(x)\right) - (1-y) * \log\left(1 - f_\theta(x)\right)$$

where $y \in \{0, 1\}$ is a training target. Complete the implementation of `NegativeLogLikelihoodLoss` and report the solution to $\frac{\partial \mathcal{L}(f(x),y)}{\partial f(x)}$.

ℓ_2 Regularization [2pts] In many applications we need to constrain the model complexity to prevent overfitting on the training corpus and to avoid exploding weights. We will use ℓ_2 regularization on the parameters θ of our model:

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

Complete the implementation of `L2Regularization` and report the solution to $\frac{\partial \lambda * \frac{1}{2}\|\theta\|_2^2}{\partial \theta}$. Implementation hint: you can convert matrices to vectors using the Breeze method `.toDenseVector`.

4.3.2 Model [5pts]

At this point we are ready to compose the sum-of-word-vectors model from the building blocks above. If you haven't done this already, this is the time to use the gradient checker to make sure the implementation of all your blocks is correct.[2] Complete the implementation of `SumOfWordVectorsModel`. Implementation hints: (i) You can use `LookupTable.addTrainableWordVector` to add vectors for words to `vectorParams`. (ii) You can retrieve parameters using `vectorParams(NAME)`. Test the whole model using gradient checking and report the difference to the estimated gradient.

[2]Or more rather: your forward and backward passes are consistent.

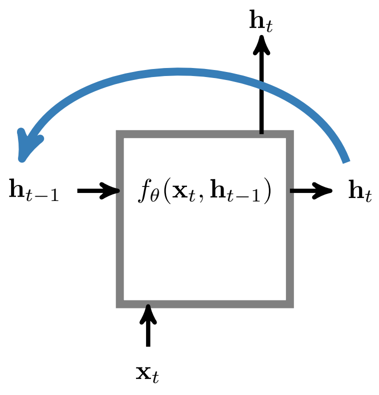

Figure 2: Schematic Recurrent Neural Network.

4.3.3 Stochastic Gradient Descent [2pts]

Complete the code of `StochasticGradientDescentLearner`. Hint: `forward`, `backward` and `update` are your friends.

4.3.4 Grid Search and Evaluation [6pts]

In `Main.scala` we provide some code that shows how to use the model and training code. We provide an `epochHook` that prints the accuracy on the training and dev set after every epoch (i.e. after all training examples have been processed). Perform a grid search over the hyperparameters (dimension of word vectors, regularization strength and learning rate). Report the best configuration and discuss your results.

Recommendation: In case you get exploding weights (usually indicated by a NaN loss or weight), you might need to use a higher regularizer and lower learning rate.

4.3.5 Loss Analysis [3pts]

Analyze how the training loss and the accuracy on the training and dev set change from epoch to epoch. To this end, show a plot with these measures on the y-axis and the number of epochs on the x-axis. Does your model suffer from over- or underfitting?

4.3.6 t-SNE Visualization [4pts]

Visualize the trained word representations by projecting them into two- or three-dimensional space using tSNE (`https://lvdmaaten.github.io/tsne/`) and plotting them using a library of your choice. Color the words based on how much they contribute to positive or negative sentiment using the global weight vector $\mathbf{w}$.

4.4 Problem 3: Recurrent Neural Networks [31pts]

Next we will implement a recurrent neural network (see Figure 2) that allows us to take word order into account. Given an input word vector $\mathbf{x}_t$ at position t and a hidden state at the previous step $\mathbf{h}_{t-1}$, the next state can be caculated as:

$$\mathbf{h}_t = tanh(\mathbf{W}^h \mathbf{h}_{t-1} + \mathbf{W}^x \mathbf{x}_t + \mathbf{b})$$

where $tanh$ is the element-wise application of the tanh function, $W^x \in \mathbb{R}^{H \times I}$ and $W^h \in \mathbb{R}^{H \times H}$ are weight matrices and $b \in \mathbb{R}^H$ is a bias vector. H denotes the dimension of the hidden vectors and I the dimension of the input word vectors. As sentence representation we use the last hidden vector h_N, i.e., the hidden state after all words have been processed.

4.4.1 Blocks

To express this model, we need to implement various new blocks.

Matrix Parameter [3pts] First, we need a matrix parameter for the weight matrices parameterizing the recurrence. Complete the implementation of `MatrixParam`. **As for `VectorParam` use gradient clipping to constrain the gradient to lie in a predefined range.**

Matrix-Vector Multiplication [4pts] Complete the implementation of `Mul`, which represents the matrix-vector multiplication, and report the solution to $\mathbf{z}\frac{\partial \mathbf{Wx}}{\partial \mathbf{x}}$ and $\mathbf{z}\frac{\partial \mathbf{Wx}}{\partial \mathbf{W}}$. Implementation hint: you can use the implementation of the outer product of two vectors (`outer` is provided in `package.scala`).

Tanh [3pts] Complete the implementation of `Tanh`, the element-wise application of the tanh function to the components of a vector input, and report the solution to $\mathbf{z}\frac{\partial tanh(\mathbf{x})}{\partial \mathbf{x}}$.

4.4.2 Model [10pts]

Complete the implementation of `RecurrentNeuralNetworkModel`. Implementation hint: you can use `foldLeft` with a start-state vector parameter $\mathbf{h}_0$. Test the whole model using gradient checking and report the difference to the estimated gradient.

4.4.3 Grid Search, Initialization, Evaluation and Comparison [10pts]

Perform a small grid search over the hyperparameters (dimension of word vectors, regularization strength and learning rate). In addition, experiment with different initializations of the parameter matrices (you can use the method `set` of `ParamBlock` to initialize a parameter vector/matrix to a specific vector/matrix). Explain your choice of initialization, report the best configuration and discuss your results. Compare them to the results obtained from the sum-of-word-vectors model.

Recommendations: it is often a good idea to first see if your model can fit a small training set (like the dev or debug corpus we provided). If your model already fails achieving accuracies close to 100% on the debug set, something is most likely wrong. Furthermore, initialization of the matrices defining the recurrence can make a difference.

4.5 Problem 4: Sky's the Limit [up to 27pts]

Implement the best model for sentiment analysis on Twitter using the Blocks framework. Explain your modeling choices. You might want to implement additional blocks. Some ideas you could try (feel free to do something completely different though!):

- Dropout regularization
- Multiplication-of-Word-Vectors Model
- Element-wise sigmoid
- Vector-concatenation and split
- Gated Recurrent Unit RNN (GRU)
- Long Short-Term Memory RNN (LSTM)
- Loading pre-trained word2vec vectors (`https://code.google.com/p/word2vec/`)
- Learning a representation of the unknown word
- ...

We are eager to see how your model performs on the test set. Along with the best configuration of the sum-of-word-vectors or RNN model, also provide predictions on the test set for your own model in a file "predictions_own.txt" using the same format mentioned in Section 3.2. Aside from good performance on the test set, we are mostly interested in your thoughts and modeling choices that you write about in your report, so explain why you did what you did :)