Deep Sentiment Analysis on Tumblr

Anthony Hu

A dissertation submitted in partial fulfilment
of the requirements for the degree of
Master of Science in Applied Statistics

Department of Statistics
University of Oxford
Oxford, United Kingdom

September 2017

Declaration

The work in this thesis is based on research carried out at the Department of Statistics, University of Oxford. No part of this thesis has been submitted elsewhere for any other degree or qualification and it is all my own work unless referenced to the contrary in the text.

Acknowledgements

I would like to thank my supervisor Seth Flaxman who always had great insights and ideas. I would also like to thank my parents for giving me the opportunity to study here in Oxford, and for their unfaltering support.

Deep Sentiment Analysis on Tumblr

Anthony Hu

Submitted for the degree of Master of Science in Applied Statistics
September 2017

Abstract

This thesis proposes a novel approach to sentiment analysis using deep neural networks on both image and text. Deep convolutional layers extract relevant features on Tumblr photos and word embedding summarise the textual information to accurately infer the emotion of the post.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Chapter 2

Natural Language Processing

Even as a human being, it can be difficult to guess the expressed emotion by looking only at the Tumblr Image without reading the text as shown by Figure 2.1

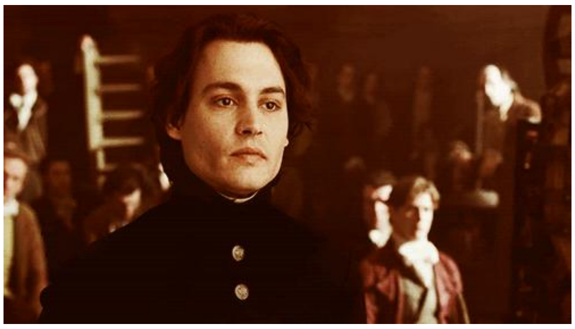


Figure 2.1: Which emotion is it?

With only the image, it's unclear whether the user is sad or disgusted. Only be reading the text "Me when I see a couple expressing their affection in physical ways in public?, you can finally conclude that the emotion conveyed was: *disgusted*. The text is extremely informative and is usually crucial to accurately infer the emotion. Neural networks only work with numbers, therefore the text has to be converted into raw numbers. A very successful way to capture the meaning of a sentence is by using word embedding.

2.1 Word Embedding

One way to map the text into numbers would be to use a dictionary that maps each word in the vocabulary (containing all the words of every Tumblr posts) to an integer. Then, you could transform any word into an one-hot vector, a vector of size the number of words in the vocabulary, with a 1 in the position of the word and 0s elsewhere. A sentence could then be encoded as a sum of vectors, that can be normalised by some distance (L^2 for instance). A major drawback is that this will cause data sparsity as the vocabulary size can be huge. For example, the number of 5-word sentences with a vocabulary size of 1000, is $1000^5 = 10^{15}$. This problem is specific to text as image and audio processing systems train on rich high-dimensional data (pixel intensities for images and spectral densities for audio), as shown by Figure 2.2.

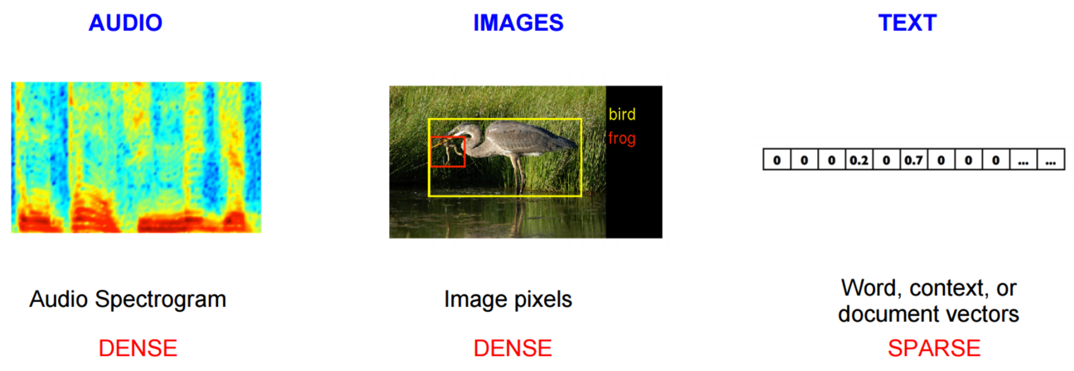


Figure 2.2: Data sparsity in text [11]

Most learning algorithms rely on the local smoothness hypothesis, that is, similar training instances are spatially close. This hypothesis clearly doesn't hold with one-hot encoding as 'dog' is as close as 'tree' as it is with 'cat'. Ideally, you would like to transform the word 'dog' in a space so that it's closer to 'cat' than it is to 'tree'. That's exactly how word embedding works: every word is projected into a highly dimensional space that keep semantic relationships. Therefore, what the model has learned about dogs can be used when a cat is encountered.

2.1.1 Word2Vec Overview

The Word2Vec model by Mikolov et al. [10] is an efficient implementation of word embedding and works as follow: Suppose you have a sentence: 'the ants in the garden'. You can break that sentence in (context, target) pairs where the context are the words surrounding the target word. For example, if you take a context with a window of 1, you get the follow pairs: ([the, in], ants), ([ants, the], in), ([in, garden], the) [we omitted the pairs where the context wasn't of size 2]. You will then train a model to predict the target word given the context, and the weights of the model will give the word embedding (this will become clearer shortly). This model is called the Continuous Bag-of-Words model, and Word2Vec also comes in another flavor called the Skip-Gram model.

In the Skip-Gram model, you will predict the context given the target word, creating more pairs as for instance ([the, in], ants) are split into two training instances: (ants, the), (ants, in). The Continuous Bag-of-Words model smooth over the the distributional information by using the whole context, and works well on smaller datasets, but by breaking the (context, target) pair into more observations, the Skip-Gram model tends to perform better on larger datasets, and that's the model we will stick on from now on.

2.1.2 Skip-Gram Model

The model is a neural network with one hidden layer and its objective is to predict the context word given the target. The input of the network will be a target word represented as a one-hot vector, of size say 10,000 (that's the vocabulary size) and the output will also be a vector of size 10,000 giving the probability distribution of the context word. Here is the architecture of the model:

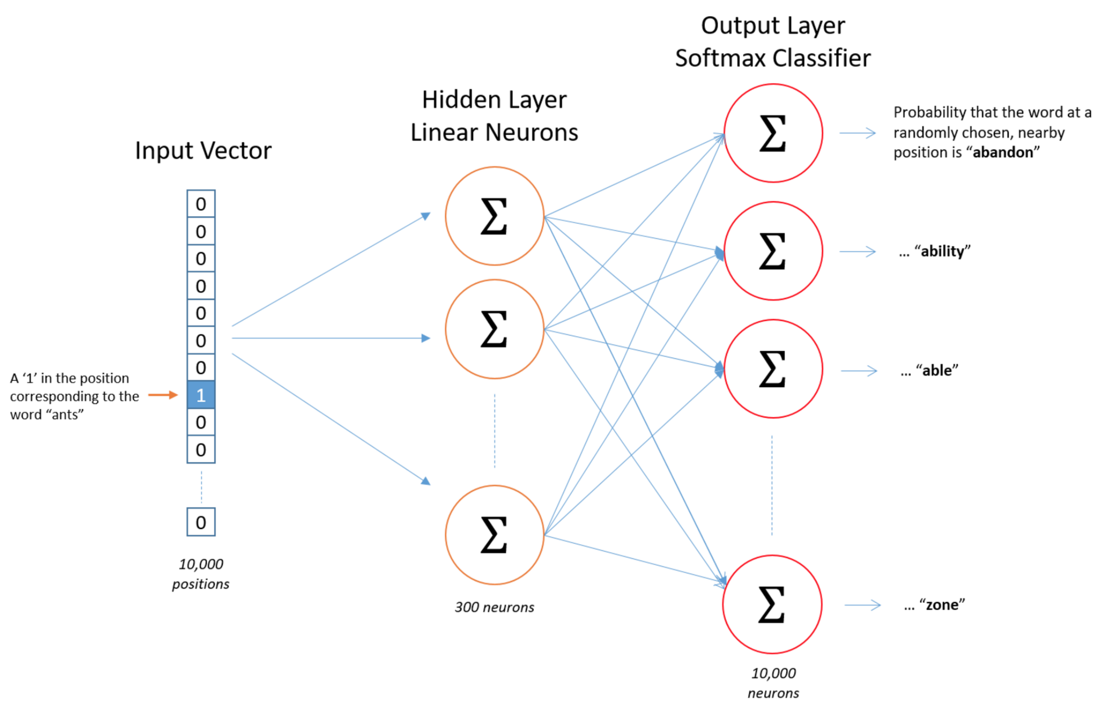


Figure 2.3: Skip-Gram model architecture [12]

There is no activation function in the hidden layer, but output neurons use softmax. The weights of the hidden layer matrix give the embedding as when you multiply a 1×10 000 one-hot vector with a 10 000×300 matrix you select the row of size 1 × 300 corresponding to the high-dimensional representation of that word:

$$[0 \quad 0 \quad 0 \quad 1 \quad 0] \times \begin{bmatrix} 17 & 24 & 1 \\ 23 & 5 & 7 \\ 4 & 6 & 13 \\ 10 & 12 & 19 \\ 11 & 18 & 25 \end{bmatrix} = [10 \quad 12 \quad 19]$$

Figure 2.4: Where the embedding comes from [12]

To learn those weights, the network is minimising the cross-entropy loss given a (target word, context word) = (w_t, w_c):

$$\begin{aligned} \mathrm{J_{CE}}(w_t, w_c) &= -\log P(w_c|w_t) \\ &= -\log \{\mathrm{softmax}(w_c, w_t)\} \\ &= -\log \left(\frac{\exp\{\mathrm{score}(w_c, w_t)\}}{\sum_{\text{Word w' in Vocab}} \exp\{\mathrm{score}(w_c, w')\}} \right) \end{aligned} \tag{2.1}$$

With score(w_c, w_t) being the element in position w_c in the output layer (right before the softmax activation function).

2.1.3 Intuition

In this model, two words that have similar contexts should output similar probability distribution. One way to produce that is to simply learn a similar word embedding for these two words. Therefore, words with similar context will have a similar vector representation which is exactly what we wanted.

Which words have similar contexts? Synonyms are a good example: ‘brave’ and ‘fearless’ are two words that must appear in similar contexts. The same applies for words that are related such as ‘physics’ and ‘thermodynamics’ or words with the same stem ‘apples’ and ’apple’.

2.2 Training the Model

Training the network described above involves, with a vocabulary size of 10 000: 10 000×300=3M parameters for the hidden layer and 300×10 000=3M for the output layer. In fact, the actual Word2Vec model contain 3M words, the number of parameters is thus 2×3M×300=1.8B. That’s a huge neural network that will need a really large training set to train those parameters, which is not feasible without a few tweaks.

2.2.1 Negative Sampling

When training the model with gradient descent, each backward pass will update all the parameters of the model. Negative sampling addresses this problem by only updating a fraction of the parameters.

For a given (target, context word) pair, we want the output of the model to be 1 on the context word and 0s for all the other words. With negative sampling, we’ll instead randomly select a small subset of ‘negative samples’ (a word we want the network to output a 0 for) to update the weights for. The paper by T. Mikolov et al. [13] states that 5-20 negative samples for small datasets and 2-5 for large datasets achieve good results.

More specifically, the negative samples are selected using a 'unigram distribution' with more frequent words more likely to be selected. The probability to select a word w_i is simply its frequency f_i to the power 3/4 (chosen empirically) divided by the sum of weights of all the other words:

$$P(w_i) = \frac{f_i^{3/4}}{\sum_{j=1}^{V} f_j^{3/4}} \tag{2.2}$$

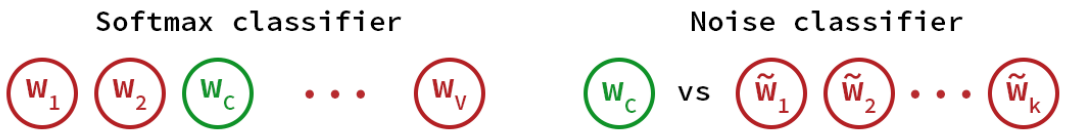
If we select 5 negative samples, then in the output layer those 5 words and the context word will be updated, and they each have 300 parameters (the embedding size) meaning that only 1800 parameters will be updated among the 0.9B parameters in the output layer, that's only 0.0002% of the parameters.

In the hidden layer, only the weights of the input word (300 parameters) will be updated, but that's always the case regardless of negative sampling as the one-hot vector zero-out every weight in the hidden layer that does not belong to the input word.

2.2.2 Negative Sampling, with Maths

The loss function (2.1) has a normalising denominator that is expensive to compute, and is the direct cause of why all the parameters would be updated. We'd like to instead use an approximation with a loss cheaper to compute. This approximation is only used when training as during inference, the softmax function has to be computed to obtain a proper probability distribution.

Instead of discriminating the context word w_c from all the other words in the vocabulary, we'll sample k words $\tilde{w}_1, \tilde{w}_2, ..., \tilde{w}_k$ from a noise distribution Q, the unigram distribution, that we'll discriminate from w_c, as shown in Figure 2.5.



(a) Softmax loss (b) Negative Sampling loss

Figure 2.5: Comparison of the two loss functions

The new binary classification task has w_c as positive examples ($y = 1$) and all the noise samples $\tilde{w}_1, \tilde{w}_2, ..., \tilde{w}_k$ as negative examples $y = 0$. The loss function, with a pair (w_t, w_c), is:

$$\mathrm{J}_{\mathrm{NEG}}(w_t, w_c) = -[\log\, P(y = 1|w_c, w_t) + k\mathbb{E}_{\tilde{w}\sim Q}[\log\, P(y = 0|\tilde{w}, w_t]] \tag{2.3}$$

Calculating the expectation $\mathbb{E}_{\tilde{w}\sim Q}$ would involve summing over all the vocabulary to compute the probability normalising constant, which is exactly what we wanted to avoid in the first place. That's why we will instead use a Monte Carlo approximation with our noise samples:

$$\begin{aligned}\mathrm{J}_{\mathrm{NEG}}(w_t, w_c) &= -\left[\log\, P(y = 1|w_c, w_t) + k\sum_{j=1}^{k}\frac{1}{k}\log\, P(y = 0|\tilde{w}_j, w_t]\right] \\ &= -\left[\log\, P(y = 1|w_c, w_t) + \sum_{j=1}^{k}\log\, P(y = 0|\tilde{w}_j, w_t)\right] \end{aligned} \tag{2.4}$$

We still haven't a proper derivation of the probability $P(y = 1|w_c, w_t)$. Note that for each context word w_c given its target word w_t, we're generating k noise samples from a distribution Q. There are two distributions at stake: the distribution P_{train} of the context word w_c given w_t which is simply the softmax computed earlier:

$$P_{\text{train}}(w_c|w_t) = \text{softmax}(w_c, w_t) \tag{2.5}$$

And the unigram distribution Q to sample the noise:

$$Q(w) = \frac{f_i^{3/4}}{\sum_{j=1}^{V} f_j^{3/4}} \tag{2.6}$$

The probability to obtain a positive example is simply a weighted probability of seeing an example from P_{train}:

$$\begin{aligned} P(y=1|w_c,w_t) &= \frac{\frac{1}{k+1}P_{\text{train}}(w_c|w_t)}{\frac{1}{k+1}P_{\text{train}}(w_c|w_t)+\frac{k}{k+1}Q(w_c)} \\ &= \frac{P_{\text{train}}(w_c|w_t)}{P_{\text{train}}(w_c|w_t)+kQ(w_c)} \end{aligned} \tag{2.7}$$

Therefore we also have:

$$P(y=0|w_c,w_t) = \frac{kQ(w_c)}{P_{\text{train}}(w_c|w_t)+kQ(w_c)} \tag{2.8}$$

Computing $P_{\text{train}}(w_c|w_t)$ remains expensive as it involves the softmax normalising factor $Z(w_c)$:

$$\begin{aligned} P_{\text{train}}(w_c|w_t) &= \frac{\exp\{\text{score}(w_c,w_t)\}}{\sum_{\text{Word w' in Vocab}}\exp\{\text{score}(w_c,w')\}} \\ &= \frac{\exp\{\text{score}(w_c,w_t)\}}{Z(w_c)} \end{aligned} \tag{2.9}$$

The trick to avoid computing $Z(w_c)$ is to simply consider it as another parameter the model has to learn. Mnih and Teh [14] actually set the parameter to 1 as they report that it doesn't affect the performance. This statement was bolstered by Zoph et al. [15] who found that the parameter was close to 1 with a low variance. Setting $Z(w_c)$ to 1 gives:

$$P(y=1|w_c,w_t) = \frac{\exp\{\text{score}(w_c,w_t)\}}{\exp\{\text{score}(w_c,w_t)\}+kQ(w_c)} \tag{2.10}$$

The loss function becomes:

$$\mathrm{J}_{\text{NEG}}(w_t,w_c) = -\left[\log\frac{\exp\{\text{score}(w_c,w_t)\}}{\exp\{\text{score}(w_c,w_t)\}+kQ(w_c)}+\sum_{j=1}^{k}\log\frac{kQ(w_c)}{\exp\{\text{score}(w_c,w_t)\}+kQ(w_c)}\right] \tag{2.11}$$

Actually, this loss function is not quite Negative Sampling, but instead what we call Noise Contrastive Estimation (NCE), Negative Sampling has a further simplification we'll discuss shortly. Mnih and Teh [14] proved that as the number of noise samples k increases, the gradient of the derivative of the NCE goes toward the softmax function. In their paper, they also state that 25 samples are enough to match the performance of the softmax, but with an increased speed of 45.

The remaining expensive term to compute in the loss function is $kQ(w_c)$, as it involves computing the unigram distribution over all the vocabulary. This isn't as expensive as the normalising factor $Z(w_c)$ as the noise distribution only need to be computed once and stored in a matrix during the whole training. However, in Negative Sampling, this most expensive term $kQ(w_c)$ is set to 1 [13]. This is actually true when $k = V$ and Q is an uniform distribution. Now $P(y = 1|w_c, w_t)$ is actually a sigmoid function:

$$P(y = 1|w_c, w_t) = \frac{1}{1 + \exp\{-\text{score}(w_c, w_t)\}} \tag{2.12}$$

Giving the following final loss function:

$$\mathrm{J}_{\mathrm{NEG}}(w_t, w_c) = -\left[\log \frac{1}{1 + \exp\{-\text{score}(w_c, w_t)\}} + \sum_{j=1}^{k} \log \frac{1}{1 + \exp\{\text{score}(w_c, w_t)\}}\right] \tag{2.13}$$

2.2.3 Word Pairs and Phrases

'Times Higher Education' has a different meaning than 'times', 'higher' and 'education' taken separately, it's therefore sensible to add that kind of phrases in the vocabulary. Ideally, we would like to also not add word pairs such as 'that is' or 'and are' to the vocabulary as they make more sense being separated. We want to group words that are frequent together but infrequent in general, to do so, we use the following scoring function of two words w_i and w_j:

$$\text{score}(w_i, w_j) = \frac{\text{count}(w_i w_j) - \delta}{\text{count}(w_i) \times \text{count}(w_j)} \tag{2.14}$$

With:

- count(w_i) the number of time the word w_i appear in the corpus (all the training data).

- count($w_i w_j$) the number of occurrence of both w_i and w_j in the corpus.
- δ a discounting coefficient to prevent too many phrases made up of very infrequent words.

The pairs (w_i, w_j) with a score above a threshold are then added to the vocabulary. Several passes on the training data are made to make longer phrases such as 'Times Higher Education' (usually 2-4 passes with a decreasing threshold value after each pass).

2.2.4 Subsampling of Frequent Words

If we look again at the example 'the ants in the garden', the training instances will contain (ants, the) and (garden, the). The word 'the' doesn't help a lot at understanding the usual context of the words 'ant' and 'garden' as it appears in virtually every noun. Furthermore, 'the' will have far too many training instances to get a decent vector representation of the word.

To address these problems, subsampling was used: each word in the corpus has a probability to be deleted relative to its frequency. Therefore, if we have a window size of 15 and the word 'the' is deleted, then this word will not appear in the context of the remaining words. Also, we now have 15 times fewer training instances containing 'the'.

The probability of keeping a word w_i with frequency f_i is:

$$P(w_i) = \frac{t}{f_i} + \sqrt{\frac{t}{f_i}} \tag{2.15}$$

With t a parameter controlling how aggressive subsampling is, smaller value of t means more subsampling. The recommended value of t is 1e−5.

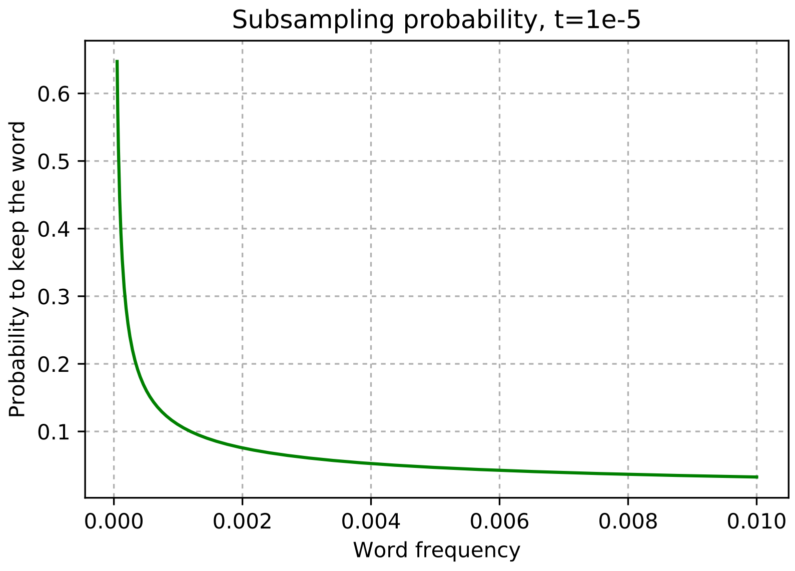


Figure 2.6: Subsampling probability graph

Note that the formula (2.13) is different from the one in [13], but we decided to keep (2.13) as it was used in the actual C implementation of Word2Vec. Note that:

- $P(w_i) = 1.0$ for $f_i \leq 2.6\text{e}{-5}$, meaning that subsampling will only affect words that represent more than 0.0026% of the words.
- $P(w_i) = 0.5$ for $f_i = 7.5\text{e}{-5}$, any word representing 0.0075% of the words will have a fifty-fifty chance of being dropped.

Chapter 3

Recurrent Neural Networks for text generation

3.1 Section 1

Chapter 4

Useful maths in LaTeX

4.1 Equations related

$$\frac{\partial u_1}{\partial t} = \Delta w_1 \quad \text{in } \ \Omega, t > 0, \tag{4.1}$$

$$\frac{\partial u_2}{\partial t} = \Delta w_2 \quad \text{in } \ \Omega, t > 0, \tag{4.2}$$

where

$$w_1 = \frac{\delta F(u_1, u_2)}{\delta u_1}, \tag{4.3}$$

$$w_2 = \frac{\delta F(u_1, u_2)}{\delta u_2}, \tag{4.4}$$

$$\begin{aligned} F(u_1, u_2) &= b_1 u_1^4 - a_1 u_1^2 + c_1 |\nabla u_1|^2 \\ &\quad + b_2 u_2^4 - a_2 u_2^2 + c_2 |\nabla u_2|^2 \\ &\quad + D \left(u_1 + \sqrt{\frac{a_1}{2b_1}} \right)^2 \left(u_2 + \sqrt{\frac{a_2}{2b_2}} \right)^2 . \end{aligned} \tag{4.5}$$

$$U_1^n = \sum_{i=1}^{J} U_{1,i}^n \eta_i, \quad W_1^n = \sum_{i=1}^{J} W_{1,i}^n \eta_i, \tag{4.6}$$

$$U_2^n = \sum_{i=1}^{J} U_{2,i}^n \eta_i, \quad W_2^n = \sum_{i=1}^{J} W_{2,i}^n \eta_i, \tag{4.7}$$

We also use the following notation, for $1 \leq q < \infty$,

$$L^q(0,T;W^{m,p}(\Omega)) := \left\{ \eta(x,t) : \ \eta(\cdot,t) \in W^{m,p}(\Omega), \int_0^T \|\eta(\cdot,t)\|_{m,p}^q \, dt < \infty \right\},$$
$$L^\infty(0,T;W^{m,p}(\Omega)) := \left\{ \eta(x,t) : \eta(\cdot,t) \in W^{m,p}(\Omega), \ \operatorname*{ess\,sup}_{t\in(0,T)} \|\eta(\cdot,t)\|_{m,p} < \infty \right\},$$

Cases

$$|v|_{0,r} \leq C|v|_{0,p}^{1-\mu}\|v\|_{m,p}^{\mu}, \qquad \text{holds for } r \in \begin{cases} [p,\infty] & \text{if } m - \frac{d}{p} > 0, \\ [p,\infty) & \text{if } m - \frac{d}{p} = 0, \\ [p, -\frac{d}{m-d/p}] & \text{if } m - \frac{d}{p} < 0. \end{cases} \tag{4.8}$$

4.2 Writing

Lemma 4.2.1 Let $u, v, \eta \in H^1(\Omega)$, $f = u - v$, $g = u^m v^{n-m}$, $m, n = 0, 1, 2$, and $n - m \geq 0$. Then for $d = 1, 2, 3$,

$$\left| \int_\Omega fg\eta dx \right| \leq C|u-v|_0 \ \|u\|_1^m \ \|v\|_1^{n-m} \ \|\eta\|_1. \tag{4.9}$$

Proof: Note that using the Cauchy-Schwarz inequality we have

$$|(u)^m v^{n-m}|_{0,p} \leq \begin{cases} |u|_{0,2mp}^m \ |v|_{0,2(n-m)p}^{(n-m)} & \text{for } \ n-m \neq 0, \ \text{ and } \ m \neq 0, \\ |u|_{0,mp}^m \ \text{ or } \ |v|_{0,(n-m)p}^{(n-m)} & \text{for } \ m = 0, \ \text{ or } \ n-m = 0 \ \text{ respectively.} \end{cases}$$

Noting the generalise Hölder inequality and the result above we have

$$\begin{aligned} \left| \int_\Omega fg\eta dx \right| &\leq |u-v|_0 \ |u^m v^{n-m}|_{0,3} \ |\eta|_{0,6}, \\ &\leq |u-v|_0 \ \ |\eta|_{0,6} \begin{cases} |u|_{0,6}^2 & \text{for } \ m = 2, \\ |u|_{0,6} \ |v|_{0,6} & \text{for } \ m = 1, \\ |v|_{0,6}^2 & \text{for } \ m = 0, \end{cases} \\ &\leq C|u-v|_0 \ \|u\|_1^m \ \|v\|_1^{n-m} \ \|\eta\|_1, \end{aligned}$$

where we have noted (4.8) to obtain the last inequality. This ends the proof. $\square$

We consider the problem:

(**P**) Find $\{u_i, w_i\}$ such that $u_i \in H^1(0,T;(H^1(\Omega))') \cap L^\infty(0,T;H^1(\Omega))$ for *a.e.* $t \in (0,T)$, $w_i \in L^2(0,T;H^1(\Omega))$

$$\left\langle \frac{\partial u_1}{\partial t}, \eta \right\rangle$$

Chapter 5

Conclusions

Bibliography

[1] **Tumblr photos:**
http://fordosjulius.tumblr.com/post/161996729297/just-relax-with-amazing-view-ocean-and
http://ybacony.tumblr.com/post/161878010606/on-a-plane-bitchessss-we-about-to-head-out
https://little-sleepingkitten.tumblr.com/post/161996340361/its-okay-to-be-upset-its-okay-to-not-always-be
http://shydragon327.tumblr.com/post/161929701863/tensions-were-high-this-caturday
https://beardytheshank.tumblr.com/post/161087141680/which-tea-peppermint-tea-what-is-your-favorite
https://idreamtofflying.tumblr.com/post/161651437343/me-when-i-see-a-couple-expressing-their-affection

[2] **Convolution images:** M. Gorner, Tensorflow and Deep Learning without a PhD, Presentation at *Google Cloud Next '17*
https://docs.google.com/presentation/d/1TVixw6ItiZ8igjp6U17tcgoFrLSaHWQmMOwjlgQY9co/pub?slide=id.g1245051c73_0_2184
The slide on the convolutional neural network was adapted to our architecture.

[3] **Max pooling:** Cambridge Spark
https://cambridgespark.com/content/tutorials/convolutional-neural-networks-with-keras/index.html

[4] A. Krizhevsky, I. Sutskever and G. Hinton, ImageNet Classification with Deep Convolutional Neural Networks. In *NIPS*, 2012.

[5] C. Szegedy et al., Going deeper with convolutions. In *CVPR*, 2015.

[6] K. He et al., Deep Residual Learning for Image Recognition. In *CVPR*, 2016 .

[7] A. Karpathy, L. Fei-Fei, J. Johnson, Transfer Learning. In *Stanford CS231n Convolutional Neural Networks for Visual Recognition*, 2016.

[8] S. Arora et al., Provable Bounds for Learning Some Deep Representations. In *ICML*, 2014.

[9] Video explaning Inception Module, https://www.youtube.com/watch?v=VxhSouuSZDY.

[10] T. Mikolov et al., Efficient Estimation of Word Representations in Vector Space. In *ICLR*, 2013.

[11] TensorFlow, Word2Vec tutorial, https://www.tensorflow.org/tutorials/word2vec.

[12] C. McCormick, Word2Vec Tutorial - The Skip-Gram Model, http://mccormickml.com/2016/04/19/word2vec-tutorial-the-skip-gram-model/.

[13] T. Mikolov et al., Distributed Representations of Words and Phrases and their Compositionality. In *NIPS*, 2013.

[14] A. Mnih and Y. W. Teh, A fast and simple algorithm for training neural probabilistic language models. In *ICML*, 2012.

[15] B. Zoph et al., Simple, Fast Noise-Contrastive Estimation for Large RNN Vocabularies. In NAACL, 2016.

Appendix A

Basic and Auxiliary Results

A.1 Basic Results