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 thesis supervisor Dr Seth Flaxman who always had great insights and ideas. I am gratefully indebted to his valuable comments on this thesis. I also want to express my profound gratitude to my parents for giving me the opportunity to study in Oxford, and for their unfaltering support throughout my years of study.

Deep Sentiment Analysis on Tumblr

Anthony Hu

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

Abstract

We propose a novel approach to Sentiment Analysis using Deep Neural Networks combining Visual Recognition and Natural Language Processing. Our approach leverages Tumblr posts containing images and text to predict the emotional state, given by a Tumblr tag, of users. Deep convolutional layers extract relevant features of the images and high-dimensional word embedding followed by a recurrent layer process the textual information to accurately infer the emotion of a given Tumblr post. We demonstrate that the network architecture, named Deep Sentiment, learns meaningful relations between visual data and language as it vastly outperforms models using a single modality. We then show that Deep Sentiment can also be adapted to generate images and text representative of an emotion.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Sentiment analysis has been an active area of research in the past few years, especially on the readily available Twitter data, e.g. Bollen et al. [2] who investigated the impact of collective mood states on stock market or Flaxman et al. [1] who analysed day-of-week population well-being.

Contrary to Twitter, Tumblr's posts are not limited to 140 characters, allowing more expressiveness, and are not centered on the textual content but on the image content instead. A Tumblr post will almost always be an image with some text accompanying the latter. Pictures have become prevalent on social media and characterising them could enable the understanding of billions of users.

We propose a novel method to uncover the emotional state of an individual posting on social media. The ground truth emotion will be extracted from the tags, considered as the 'self-reported' emotion of the user. Our model incorporates both text and image and we aim to 'read' them to be able to understand the emotional content they imply about the user. Concretely, the Deep Sentiment model associates the features learned by the two modalities as follows:

- We fine-tune a pre-trained Deep Convolutional Neural Network, named Inception [3], trained on the ImageNet dataset to our specific task of emotion inferring.

- We project the text in a rich high-dimensional space with a word representation learned by Word2Vec [4], that was trained on billions of words. The word

vectors then go through a Recurrent Neural Network which preserves the word order and captures the semantics of human language.

- A fully-connected layer combines the information in the two modalities and a final softmax output layer gives the probability distribution of the emotional state of the user.

We will also see that Deep Sentiment can be rearranged to generate Tumblr posts expressing one of the learned emotion.

Chapter 2

Natural Language Processing

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

Figure 2.1: Which emotion is it?

It's unclear whether the user wants to convey sadness or disgust. Only by reading the text "Me when I see a couple expressing their affection in physical ways in public", we can finally conclude that the person was *disgusted*. The text is extremely informative and is usually crucial to accurately infer the expressed emotion.

Neural networks only accept numbers as inputs, therefore the text has to be converted beforehand. A very successful way to capture the meaning of textual information is by using word embedding.

2.1 Word Embedding

One way to transform text into numbers would be to use a dictionary that maps each word in a vocabulary (containing all the words of every Tumblr posts) to an integer. Then, we could transform any word into an one-hot vector – a vector of of the same size as 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 of that representation is 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 sparsity problem is specific to text as in contrary, 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.

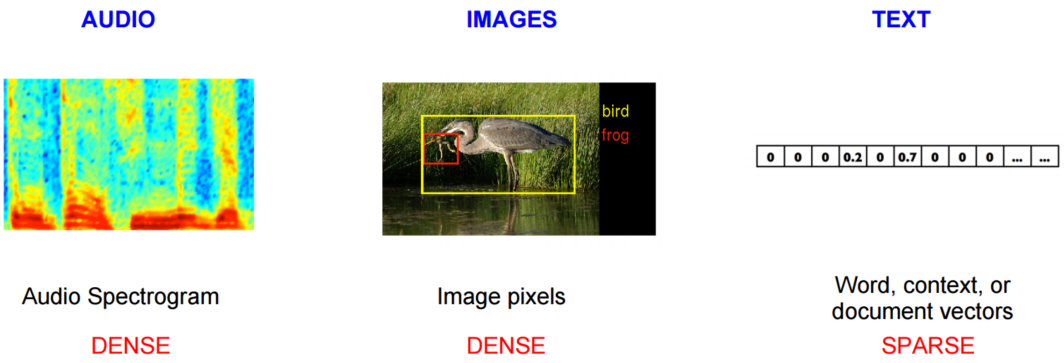


Figure 2.2: Data sparsity in text [17]

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 to 'tree' as it is to '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 preserves semantic relationships. Therefore, what the model has learned about dogs can be used when faced with a cat.

2.1.1 Word2Vec Overview

The Word2Vec model by Mikolov et al. [4] is an efficient implementation of word embedding that learned the high-dimensional representation of words with a *fake task*. The idea is to train a neural network with a single hidden layer on a task, but we call the task *fake* as the network will never be used: we're only after the weights of the hidden layer that will actually be the word representation.

Now to explain the fake task, suppose you have a sentence: "the ants in the garden". We can break that sentence in (context, target) pairs where the contexts are the words surrounding the target word. For example, if we take a context with a window of size 1, we get the following pairs: ([the, in], ants), ([ants, the], in), ([in, garden], the). We 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 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, we 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) and (ants, in). The Continuous Bag-of-Words model smooth over the distributional information by using the whole context, and works well on smaller datasets, but by breaking (context, target) pairs 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 Skip-Gram 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:

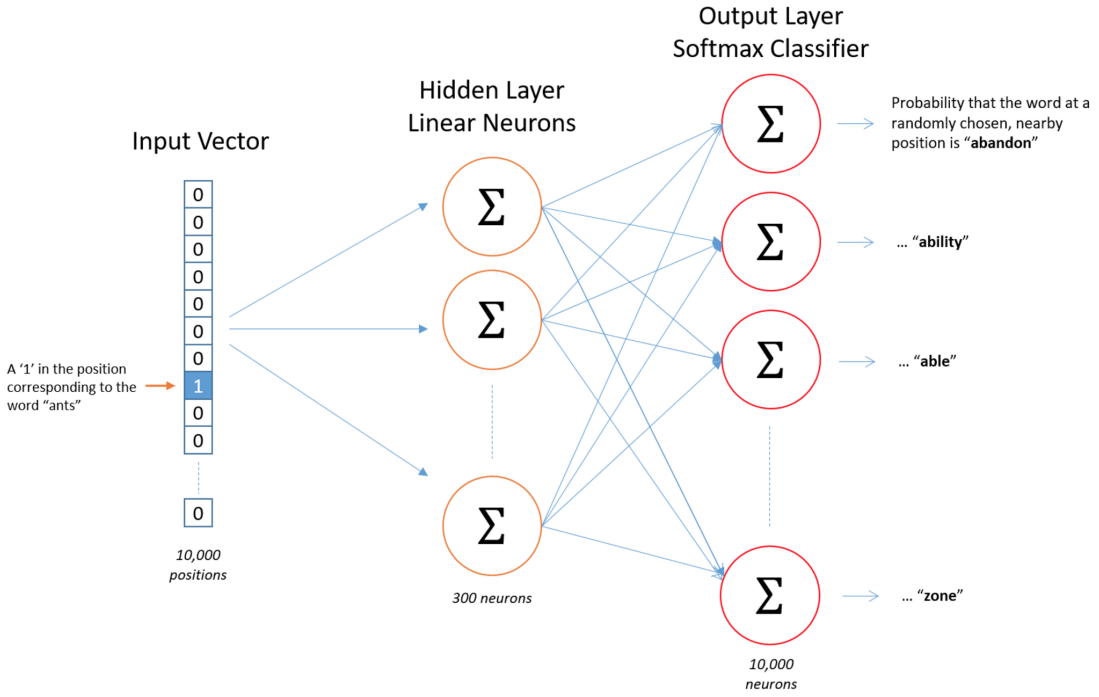


Figure 2.3: Skip-Gram model architecture [18], with a hidden layer of size 300

There is no activation function in the hidden layer, but output neurons go through softmax to obtain a probability distribution of the context word. The weights of the hidden layer matrix give the embedding as when we multiply a 1×10,000 one-hot vector with a 10,000×300 matrix (the hidden layer weights) we 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: Word embedding origin [18]

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

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

With $\mathrm{score}(w_c, w_t)$ 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 distributions. 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, e.g. 'apples' and 'apple'.

2.2 Training Word2Vec

Training the network described above involves, with a vocabulary size of 10,000: 10,000×300=3,000,000 parameters for the hidden layer and 300×10,000=3,000,000 for the output layer. And in fact, the actual Word2Vec model contain 3M words: the number of parameters is thus 2×3,000,000×300=1,800,000,000. That's a huge neural network that will need a really large training set to train those parameters, which is not feasible as is. Word2Vec uses a few tricks to obtain word representation of high quality at a reduced computational cost:

1. **Negative sampling**, to dramatically reduce the training time complexity.
2. **Word pairs and phrases**, to add frequently ocurring group of words in the vocabulary.
3. **Subsampling of frequent words**, to downsize the effects of words occurring too often.

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 word, 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' (words we want the network to output a 0 for) to update the weights for. The paper by T. Mikolov et al. [19] states that 5-20 negative samples for small datasets and 2-5 for large datasets achieve good results.

More specifically, the negative examples are sampled 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 the V other words (V been the vocabulary size):

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

If we select 5 negative samples, then in the output layer those 5 words and the context word will be updated. As they each have 300 parameters in the output layer (the embedding size), only 6×300=1,800 parameters will be updated among the 0.9B parameters in the output layer: or in other words, 0.0002% of the parameters, which considerably speed up the training.

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 representation of the input word 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 in the output layer would be updated. We'd like to instead use an approximation with a loss cheaper to compute.

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 the unigram distribution Q, called the noise distribution, that we'll discriminate from the context word 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 example ($y = 1$) 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:

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

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:

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

We still haven't given 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:

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

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, as $P(y=0|w_c,w_t) = 1 - P(y=1|w_c,w_t)$:

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

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 [20] actually set the parameter to 1 as they report that it doesn't affect the performance. This statement was bolstered by Zoph et al. [21] who found that this 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:

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

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 [20] proved that as the number of noise samples k increases, the gradient of the NCE goes toward the gradient of the softmax function. In their paper, they also state that 25 samples are enough to match the performance of the softmax, and with an increased speed of a factor 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 needs 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 [19]. 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:

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

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:

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

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 consecutive 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.

- $|W|$ the training set size, in order to make the threshold more independent of the training size.

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 there are 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) [note that in Skip-Gram, (target, context) pairs can also be made of words that have incomplete context, that is no left or no right context word such as the target word 'garden' that only have a left context word 'the']. The word 'the' doesn't help a lot at understanding the context of the words 'ant' and 'garden' as it appears in virtually every noun. Furthermore, 'the' will have far more training instances than are actually needed to get a quality vector representation of that word.

To address these problems, subsampling is used: each word in the corpus has a probability to be deleted relative to its frequency. Therefore, if we have a window size of 10 and the word 'the' is deleted, then this word will not appear in the context of the remaining words. Also, we now have 10 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.

Figure 2.6 shows how the probability to keep a word decreases with word frequency:

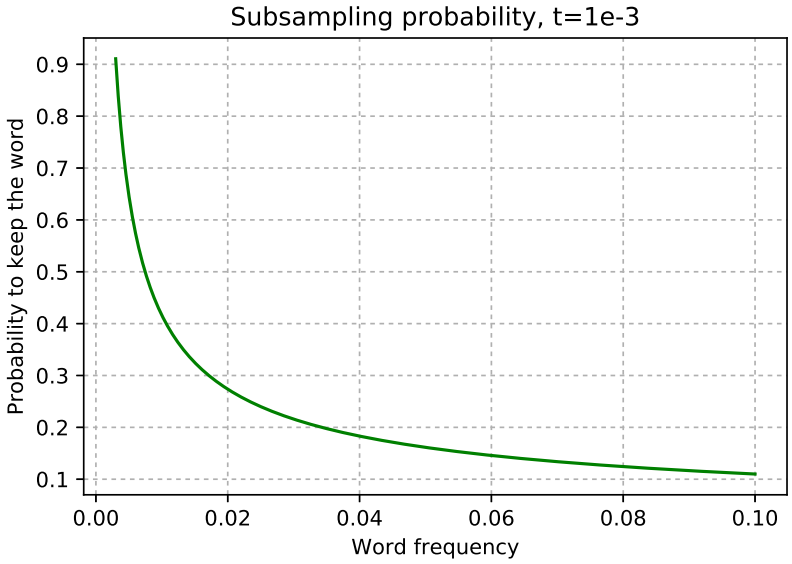


Figure 2.6: Subsampling probability graph

Note that the formula (2.6) is different from the one in [19], but we decided to keep (2.6) as it was used in the actual C implementation of Word2Vec. Note that if $t = 1e{-}3$

- $P(w_i) = 1.0$ for $f_i \leq 0.0026$, meaning that subsampling will only affect words that represent more than 0.26% of the words.
- $P(w_i) = 0.5$ for $f_i = 0.0075$, any word representing 0.75% of the words will have a fifty-fifty chance of being dropped.

2.2.5 Word2Vec Pre-Trained Model

For our task of emotion prediction, we will be using a pre-trained model to convert the text in Tumblr posts into rich high-dimensional vectors. The Word2Vec model was trained on Google News dataset (aggregation of news from all around the world, including blog posts which should be similar to Tumblr content) containing about 100 billion words. Each word in the vocabulary (3 million words) are embedded into a 300 dimensional vector.

The Skip-Gram model was trained using the following parameters:

- A context window of size 10.
- 5 negative examples in Negative Sampling.
- In Word Pairs and Phrases, the discounting factor δ is set to 100, and the threshold to 100. The number of passes could not be found unfortunately.
- A subsampling threshold of 1e−3.

2.3 Application to Tumblr Data

Each post in the dataset does not necessarily contain the same number of words. Even after embedding each word, the input will be of variable size and most learning algorithm expect a fixed-sized input. To solve that problem, we can simply average across the number of words. The information loss is still minimal as the features come from a high-dimensional space [23].

However note that the word order is completely lost. Human language relies on the word order to communicate as for example the word *change* can be both a noun and a verb, and negation such as 'not entertained' can only be understood if 'not' directly precedes the verb.

The order information can be preserved using Recurrent Neural Networks, our next chapter.

Chapter 3

Recurrent Neural Networks

Recurrent Neural Networks (RNN) can be used in a wide variety of tasks as they can work with inputs and outputs of variable size as shown in Figure 3.1.

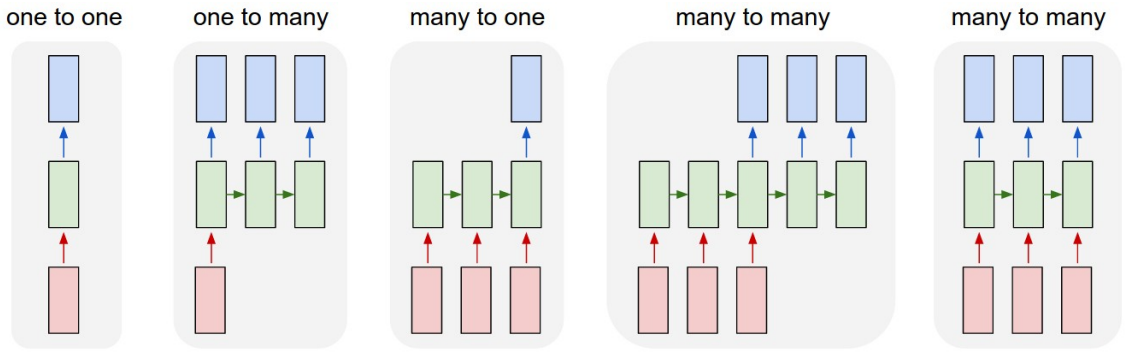


Figure 3.1: The applications of Recurrent Neural Networks [24]

1. **One to one**: A vanilla neural network with fixed-sized input and fixed-sized output.
2. **One to many**: An RNN with sequence output, e.g. image captioning: generate a sentence describing an input image.
3. **Many to one**: An RNN with sequence input, e.g. sentiment analysis: infer the emotion of a given sentence.
4. **Many to many (1)**: Sequence input and sequence output, e.g.: machine translation: output a sentence in Spanish given a sentence in English.
5. **Many to many (2)**: Synced sequence input and output, e.g.: video classification: label each frame of a video.

3.1 Vanilla RNN

Traditional neural networks do not have any memory as each input of the network are independent. Recurrent neural networks use loops to make information persist:

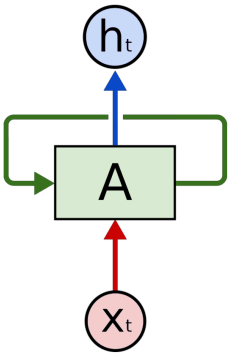


Figure 3.2: A vanilla recurrent neural network [25]

Given an input x_t and a layer A, the output h_t is fed again to A during the next step along with the next input x_{t+1}. This becomes clearer when we unroll the RNN:

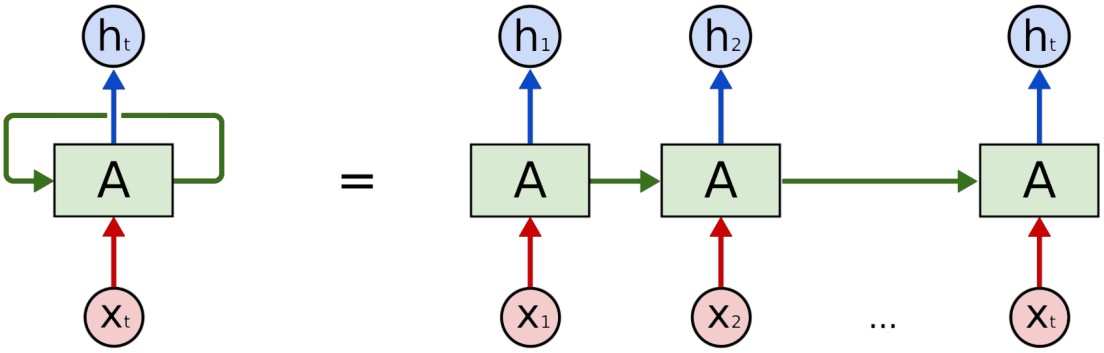


Figure 3.3: An unrolled vanilla recurrent neural network [25]

Basically, if we denote the function of layer A by f:

$$h_t = f(h_{t-1}, x_t) \tag{3.1}$$

Suppose the input $x_t \in \mathbb{R}^D$ and the layer A contains H neurons. Denote $W_x \in \mathbb{R}^{D \times H}$ the weights of the input, and $W_h \in \mathbb{R}^{H \times H}$ the weights of the hidden state.

Then the explicit expression of f is simply:

$$h_t = \tanh(W_x^T x_t + W_h^T h_{t-1}) \tag{3.2}$$

With $t \in \mathbb{N}^*$ and h_0 a randomly initialised vector. The expression of h_t is similar to the neurons in a traditional neural network with a tanh (hyperbolic tangent, see Figure 3.4) activation function and two inputs instead of one.

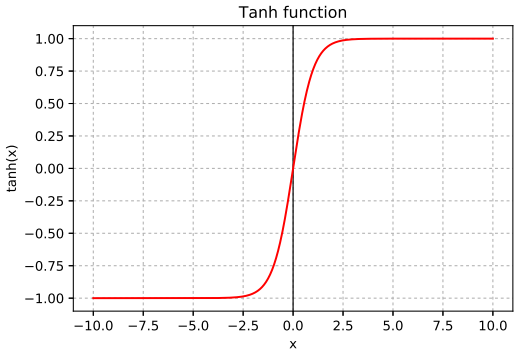


Figure 3.4: Another activation function: tanh

The vanilla RNN is a good progress towards learning from sequential inputs but is actually difficult to train as we'll see shortly.

3.2 On the Difficulty of Training Vanilla RNNs

In order to train a recurrent neural network, we unroll the network for T steps (chosen depending on what kind of dependency we want to learn and also on memory limitation) then backpropagate through the unrolled network.

Concretely, if we ignore the inputs x_t and consider that $h_t = \tanh(a_t)$ with $a_t = W_h^T h_{t-1}$, the gradients of the hidden states are computed as: for t=1..T:

- $\frac{\partial L}{\partial a_t} = 1 - \tanh^2(\frac{\partial L}{\partial h_t})$
- $\frac{\partial L}{\partial h_{t-1}} = W_h \frac{\partial L}{\partial a_t}$

Note that we also ignored the gradient coming through the output neurons and basically only backpropagated through the the green neurons in Figure 3.3. Now

suppose that W_h is a scalar, then as the gradient $\frac{\partial L}{\partial h_{t-1}}$ is repeatedly multiplied by W_h, if $W_h > 1$ the gradient diverge and if $W_h < 1$ the gradient goes to zero. Similarly in the general case $W_h \in \mathbb{R}^{H \times H}$, let us denote by $\lambda_{\max}$ the largest eigenvalue of W_h then Pascanu et al. proved in [27] that:

- If $\lambda_{\max} > 1$, the gradients will explode.
- If $\lambda_{\max} < 1$, the gradients will vanish.

To prevent the gradients from exploding, Pascanu et al. proposed gradient clipping: if the norm of the gradient exceeds a threshold, then simply clip the gradient. If we denote by g the gradient, then more formally, if $\|g\| >$ threshold, then set g to $\frac{\text{threshold}}{\|g\|} g$.

To address the vanishing gradient problem in recurrent neural networks, we use a variant of Vanilla RNN called Long-Short Term Memory (LSTM) [26] that does not have vanishing gradients.

3.3 Long Short-Term Memory

The vanilla RNN is hard to train as the gradients often either vanish or explode. LSTMs solve this problem by using a gating mechanism.

They keep track of the hidden state vector $h_t \in \mathbb{R}^H$ but also of a cell state vector $c_t \in \mathbb{R}^H$, we'll shortly explain. First, we compute the activation vector $a \in \mathbb{R}^{4H}$ that is 4 times bigger than vanilla RNN, which is necessary to remember long and short-term features.

$$a = W_x^T x + W_h^T h \tag{3.3}$$

with $x \in \mathbb{R}^D$, the input, $h \in \mathbb{R}^H$, the hidden state, $W_x \in \mathbb{R}^{D \times 4H}$ and $W_h \in \mathbb{R}^{H \times 4H}$.

Then this activation vector a is split into 4 vectors: $a_i, a_f, a_o, a_g \in \mathbb{R}^H$, where a_i is made up of the first H elements of a, a_f the H next, etc. The so-called *gates*: the input gate $i \in \mathbb{R}^H$, the forget gate $f \in \mathbb{R}^H$, the output gate $o \in \mathbb{R}^H$ and the block input $g \in \mathbb{R}^H$ are computed as:

$$i = \sigma(a_i) \qquad f = \sigma(a_f) \qquad o = \sigma(a_o) \qquad g = \tanh(a_g) \tag{3.4}$$

where σ is the sigmoid function.

Finally, we can compute the new cell state and the new hidden state as follow:

$$c_t = i \odot g + f \odot c_{t-1} \qquad h_t = o \odot \tanh(c_t) \tag{3.5}$$

where $\odot$ is the elementwise product of vectors. c_t is the sum of:

- The new information g weighted by how much we want to add that new information with gate i (remember that the sigmoid function has values between 0 and 1).
- What we knew before, c_{t-1}, weighted by how much we want to forget long-term information with gate f.

That cell state then goes through a tanh activation function (just like in the vanilla RNN) and is multiplied by the output gate o that decide how much information to let through the next state.

LSTMs therefore manage to easily remember long-term dependencies, as well as short-term ones, thus its name.

Also, note that the activation function in recurrent neural networks is tanh and not ReLU. ReLU was originally introduced to replace tanh because of the vanishing gradient problem. However, in the case of recurrent networks, LSTMs are built to not have vanishing gradients, which makes ReLU unnecessary.

3.4 LSTM for Sentiment Analysis

We'll be using the following architecture:

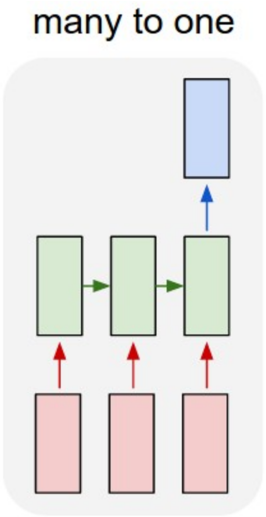


Figure 3.5: Many to one architecture

Each post will be broken down into a sequence of words and then fed to the LSTM that will infer the emotion of the user. On a more technical note, the vector of words will be represented by a list of ids from the Word2Vec vocabulary say $[3, 20, 1, 49, 6]$. To account for shorter posts, we'll have to zero-pad the vector – the id 0 will actually be associated with a word token <PAD> – like $[3, 20, 1, 49, 6, 0, 0, ..., 0]$. For longer posts, we'll only keep the 200 first words. The model is:

- Each word is embedded into a vector of dimension 300.
- An LSTM layer of size 512.
- An output layer of size 6.

The network was trained with:

- 10,000 training steps
- Mini-batch of size 128
- Adam optimizer with an initial learning rate of 0.01
- Learning rate decay of $\frac{1}{2}$ every 1000 steps
- LSTM unrolled for 200 words
- Gradient clipping with a maximum norm of 5.0

The training loss is displayed in Figure 3.6:

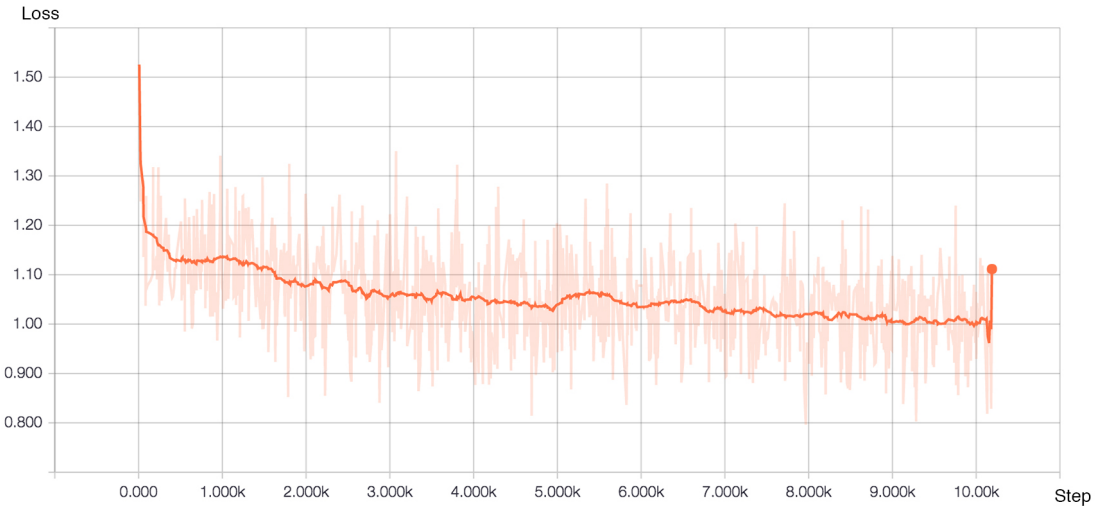


Figure 3.6: LSTM model loss

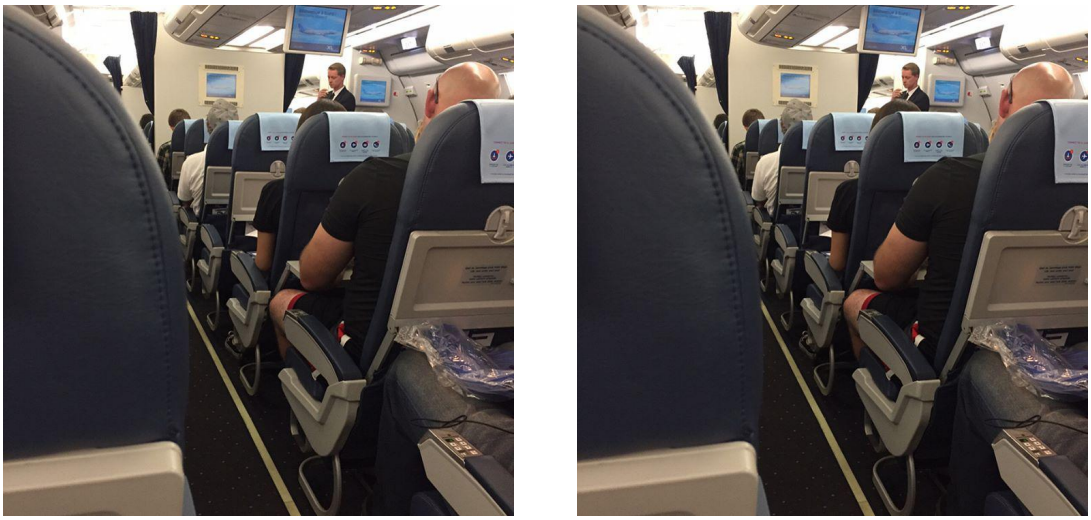
This model achieves **63%** accuracy on the train set and **61%** accuracy on the test set, which is an improvement compared to the image model. Let us finally dive into Deep Sentiment which makes use of both image and text.

Chapter 4

Deep Sentiment

Real-world information oftentimes come in several modalities. For instance, in speech recognition, humans integrate audio and visual information to understand speech, as was demonstrated by the McGurk effect [28]. Separating what we see from what we hear seems like an easy task, but in an experiment conducted by McGurk, the subjects who were listening to a */ba/* sound with a visual */ga/* actually reported they were hearing a */da/*. This is uncanny as even if you know the actual sound is a */ba/*, you cannot stop your brain from interpreting it as a */da/*.

Likewise, an image almost always come with a text as different interpretation can arise when a textual context is not provided, as shown in Figure 4.1:



(a) "Planes might just be the most frightening thing ever." **scared**

(b) "I hate it when people are taking too much space on planes." **angry**

Figure 4.1: Different meanings with different captions.

Exploiting both visual and textual information is therefore key to understand the user's emotional state. Deep Sentiment is the name of the deep neural network incorporating visual recognition and text analysis.

4.1 The Architecture

Deep Sentiment builds on the models we have seen before as shown in Figure 4.2:

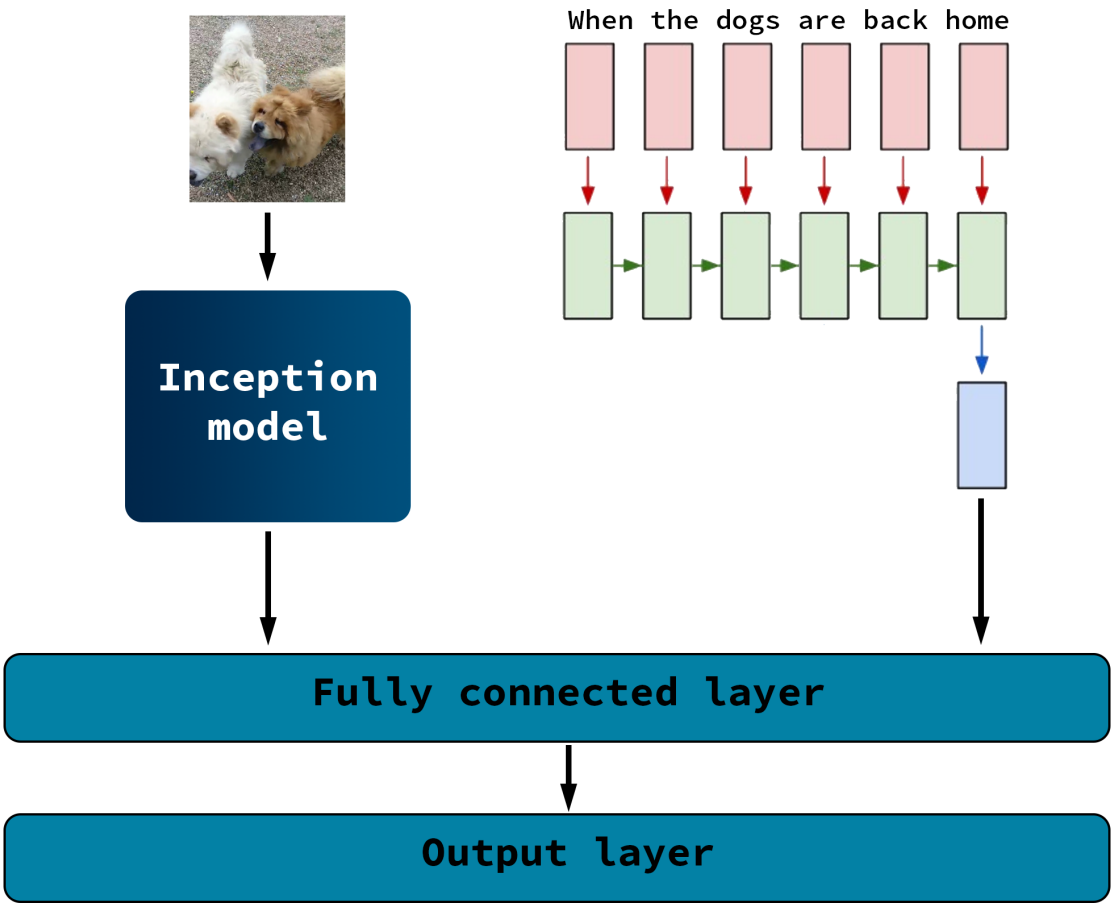


Figure 4.2: Deep Sentiment architecture

1. The image go through the pre-trained Inception model that will extract features from the images: more precisely with 128 neurons in the last Inception layer.

2. The text is embedded in a high-dimensional space with Word2Vec and will be fed to an LSTM with a 2048 neurons output layer.

3. The two outputs are concatenated to form the 'Fully connected layer' in the graph with 128 + 2048 = 2176 neurons.

4. The final layer contains 6 neurons one for each basic emotion.

5. Softmax is applied to the final layer to give the probability distribution of the emotional state of the user.

4.2 Results

Deep Sentiment was trained with:

- 10,000 steps
- Mini-batch size of 32
- Adam optimizer with initial learning rate of 1e−3
- Learning rate decay of $\frac{1}{2}$ every 1000 steps

The training process of the Inception fine-tuning was monitored with Tensorboard:

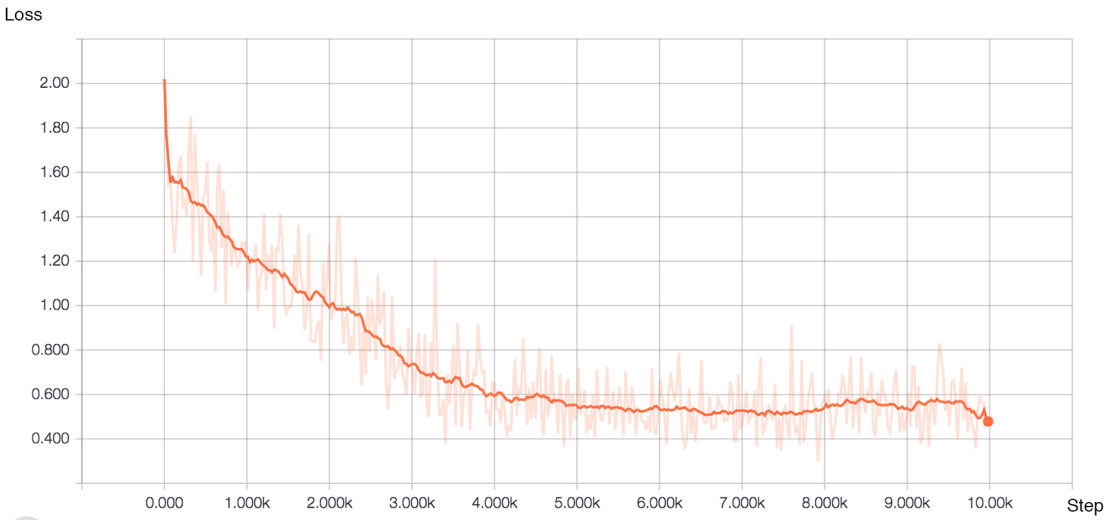


Figure 4.3: Loss function of Deep Sentiment

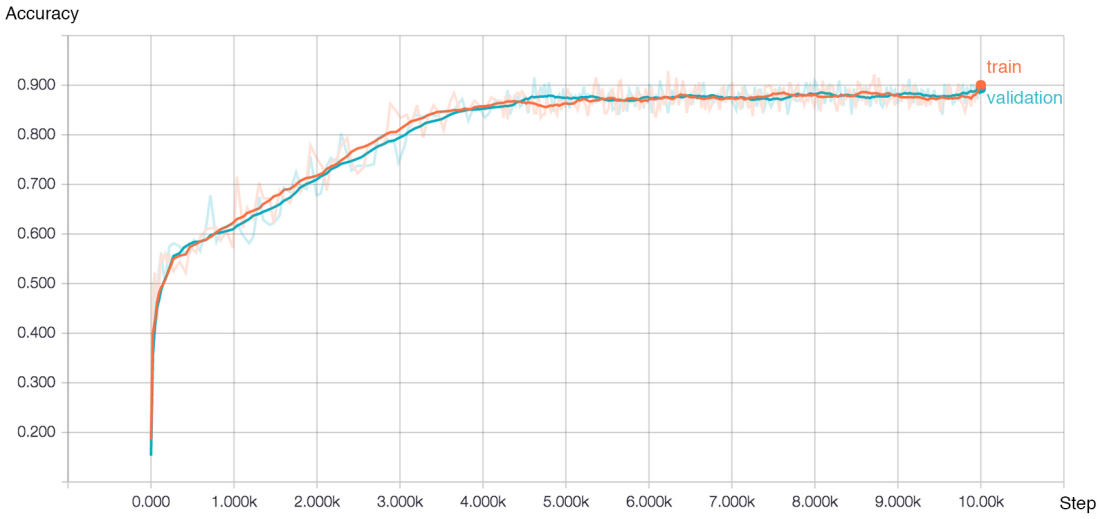


Figure 4.4: Train/validation accuracies of Deep Sentiment

This model combining text and image vastly outperforms the algorithms only using those elements separately with 90% train accuracy and 89% test accuracy. This shows that just like a human being, a neural network needs both visual and textual information to determine the emotion conveyed by a post. The synergy between visual recognition and natural language processing is impressive as shown in the comparison table 4.1.

	Loss	**Train accuracy**	**Test accuracy**
Random guessing	-	24%	24%
Inception fine-tuned	1.61	48%	42%
LSTM word embedding	1.01	61%	63%
Deep Sentiment	**0.48**	**90%**	**89%**

Table 4.1: Models results

4.3 Emotion visualisation

We can generate an image that maximises the score of a certain emotion by performing gradient ascent on a randomly initialised image [29].

More concretely, let I be an image and y be a target emotion. Let us denote by $s_y(I)$ the score of class y for the image I, that is one of the six neurons right before the softmax layer. We want to generate an image with a high score for emotion y by solving the problem:

$$I^* = \arg\max_I \; s_y(I) - R(I) \tag{4.1}$$

with $R(I)$ a regulariser that contains both explicit and implicit regularisation we will describe shortly.

Note that we're maximising the unnormalised class scores $s_y(I)$ and not the probabilities returned by the softmax: $\frac{s_y(I)}{\sum_c s_c(I)}$. The reason is that maximising the softmax probabilities can be achieved by minimising the scores of the other emotions. Instead, we want to make sure the optimisation concentrates on the emotion we want to visualise.

4.3.1 Regularisation

The explicit regulariser is the L_2 decay: $R(I) = \lambda \|I\|_2^2$ that prevents extreme pixel values from dominating the generated image. Those pixel values do not occur naturally in real images and are not useful for visualisation.

The implicit regularisations are: [30]

1. **Gaussian blur:** Gradient ascent tends to produce image with high frequency information. What are frequencies in images? To put it simply, each image is made of various frequencies: start with the average colour (low frequency) and slowly add higher frequencies wavelengths to build the details of the image.

   An image with high frequency information causes high activations but are not realistic nor interpretable as shown by Nguyen et al. [31]. High frequency information are penalised using a Gaussian blur step on image I :

GaussianBlur$(I, \theta_{\text{blur}})$ with θ_{blur} the standard deviation of the Gaussian kernel used in the blur step. Blurring an image is computationally expensive and as such, we're only blurring every $\theta_{\text{blur-every}}$ steps.

2. **Pixel clipping:** After performing L_2 decay and Gaussian blur, that suppress high amplitude and high frequency information, we're left with images with pixel values that are small and smooth. However, each pixel will still be non-zero and contribute a little bit to the gradient. We want to discard the contribution of unimportant pixels and focus only on the main object. That can be done by setting pixels with small norm (over the red, green, blue channels) to zero. The threshold $\theta_{\text{small-norm}}$ for the norm is set to be a percentile of all pixel norms in the image.

4.3.2 Generated images

We performed gradient ascent on a randomly initialised image using the following parameters:

- L_2 regularisation parameter: $\lambda = 0.001$.
- In Gaussian blur, $\theta_{\text{blur}} = 0.5$ and $\theta_{\text{blur-every}} = 10$.
- In pixel clipping, $\theta_{\text{small-norm}}$ is the norm of the 10^{th} percentile.
- 500 gradient updates.

Maximising over the emotion 'happy' yielded:

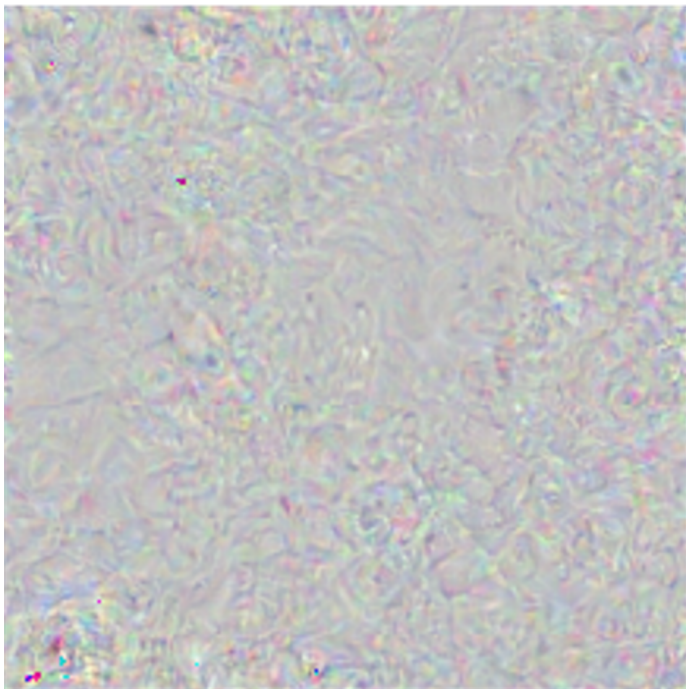


Figure 4.5: Generated image maximising happiness

Happy posts usually contained pictures of people and in the image gradient we can spot atleast two faces, but that might be because humans are especially good at spotting faces, even when there are none.

4.4 Generate text posts

We can tweak Deep Sentiment to instead make the neural network generate text by feeding an image. The network will be trained to predict the next word of the text as shown in Figure 4.6:

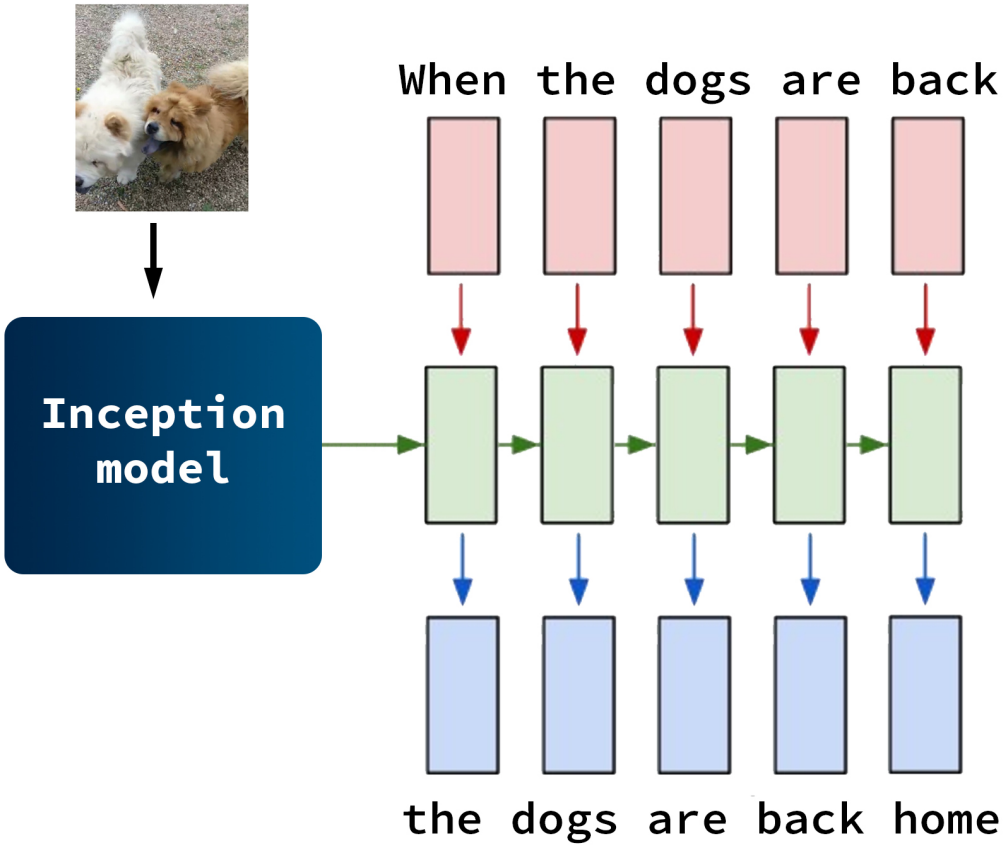


Figure 4.6: Deep Sentiment for text generation

Need to add word embedding in graph

The input words are embedded with Word2Vec and then fed to a one layer LSTM with 512 neurons. The initial hidden state h_0 of the LSTM is the output of the Inception network. If we denote by C the number of emotions, the loss J_{GEN} is the sum of the cross-entropy loss of the different predictions $\hat{y}_1, \hat{y}_2, ..., \hat{y}_T \in [0,1]^C$ of the true labels $y_1, y_2, ..., y_T \in \{0,1\}^C$ (one-hot encoding):

$$J_{GEN} = \sum_{t=1}^{T} \text{cross entropy}(y_t, \hat{y}_t) \tag{4.2}$$

with the cross-entropy loss given by: $\text{cross entropy}(y_t, \hat{y}_t) = -y_t \log(\hat{y}_t)$, therefore:

$$J_{GEN} = -\sum_{t=1}^{T} y_t \log(\hat{y}_t) \tag{4.3}$$

The network was trained using only 'happy' posts. In order to make batches, each post had a fixed length of 50 words (shorter posts were filled with a special token).

For generation, the network was fed with an image and a random first word in the dictionary. The output would then be a probability distribution of the next word that we would sample from and feed again to the network.

Here is an example of a generated post using this image (which was not in the training set:

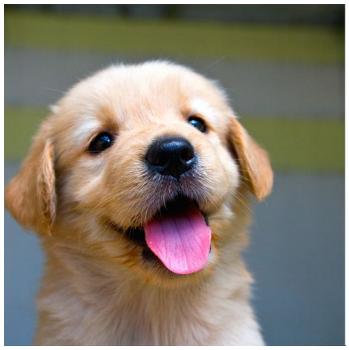


Figure 4.7: Image used to generate post

> *Need that just can have a relax months and my perfect film ! #goodlife #chill #fashion.*

That wouldn't be something a puppy would necessary say but it sounds almost human-like. There are some grammar mistakes such as 'a relax months' and unrelated hashtags '#fashion' but the Tumblr spirit is there.

Chapter 5

Conclusions

Deep Sentiment is able to determine the emotional state of Tumblr users with high accuracy by combining the textual and visual information.

On images, fine-tuning the Inception network managed to extract useful features at a reduced computational cost as convolutional neural networks are known to be tedious to train. On text, projecting words into a high dimensional space added semantic understanding of the words that could be effectively used in the recurrent layer to capture the meaning of the sentences.

The synergy between text and image is quite formidable given the jump in accuracy: from about 40-60% to 90%. At a larger scale, this algorithm could be applied population wise in order to have a real-time emotion trend during important events for example.

The deep neural network can also be rearranged to generate brand new Tumblr text according to a given emotion. Changing the model to instead accept characters instead of words could better learn the specific way of writing in blogs.

Likewise, we could try to make the model learn to generate an image given a few blog sentences. The network would try to create the image that most closely match the text it was given. For instance, a post talking about engagement and rings could produce an image with hands with engagement rings.

Lastly, an interesting experiment to make would be to try to manually label Tumblr posts in order to assess whether Deep Sentiment beats human performance.

Bibliography

[1] S. Flaxman and K. Kassam, On #agony and #ecstasy: Potential and pitfalls of linguistic sentiment analysis. In preparation, 2016.

[2] J. Bollen, H. Mao, X.-J. Zeng, Twitter mood predicts the stock market. In *Journal of Computational Science*, 2011.

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

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

[5] P. Ekman, An Argument for Basic Emotions. In *Cognitive and Emotion*, 1992.

[6] 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

[7] D. H. Huble and T. N. Wiesel, Receptive fields and functional architecture of monkey striate cortex. In *Journal of Physiology (London)*, 1968.

[8] Convolution images, M. Gorner, Tensorflow and Deep Learning without a PhD. Presentation at *Google Cloud Next*, 2017.
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.

[9] V. Nair and G. E. Hinton, Rectified Linear Units Improve Restricted Boltzmann Machines. In *ICML*, 2010.

[10] Max pooling image, P. Velickovic, Deep learning for complete beginners: convolutional neural networks with keras, 2017.
https://cambridgespark.com/content/tutorials/convolutional-neural-networks-with-keras/index.html

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

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

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

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

[15] D. Hebb, in his book The Organization of Behavior, 1949.

[16] Video explaning Inception Module, Udacity, 2016.
https://www.youtube.com/watch?v=VxhSouuSZDY

[17] Word2Vec tutorial, Tensorflow, 2017.
https://www.tensorflow.org/tutorials/word2vec

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

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

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

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

[22] Word2Vec pre-trained model, Google, 2013. https://code.google.com/archive/p/word2vec/

[23] S. Flaxman et al., Who Supported Obama in 2012? Ecological Inference through Distribution Regression. In *KDD*, 2015.

[24] A. Karpathy, The Unreasonable Effectiveness of Recurrent Neural Networks, 2015. http://karpathy.github.io/2015/05/21/rnn-effectiveness/

[25] C. Olah, Understanding LSTM Networks, 2015. http://colah.github.io/posts/2015-08-Understanding-LSTMs/

[26] S. Hochreiter and J. Schmidhuber, Long Short-Term Memory. In *Neural Computation*, 1997.

[27] R. Pascanu et al., On the difficulty of training recurrent neural networks. In *ICML*, 2013.

[28] H. McGurk and J. MacDonald, Hearing lips and seeing voices. In *Nature*, 1976.

[29] K. Simonyan, A. Vedaldi, and A. Zisserman, Deep Inside Convolutional Networks: Visualising Image Classification Models and Saliency Maps. In *ICLR*, 2014.

[30] J. Yosinski et al., Understanding Neural Networks Through Deep Visualization. In *ICML* 2015.

[31] A. Nguyen et al., Deep Neural Networks are Easily Fooled: High Confidence Predictions for Unrecognizable Images. In *CVPR*, 2015.