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 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 high-dimensional word embedding followed by a recurrent layer process the textual information to accurately infer the emotion of the post. The network architecture, named Deep Sentiment, can also be adapted to generate images and text given 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.

http://www.ifp.illinois.edu/ jyang29/papers/AAAI15-sentiment.pdf

We propose a novel method to uncover the emotional 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.

Chapter 2

Visual Recognition

The pictures are valuable to accurately determine the emotion of the user. For instance, happy photos might contain sunny landscapes and sandy beaches while sad pictures might contain darker colors. To analyse the images, we'll use convolutional neural networks, which achieve state-of-the-art performances in many visual recognition tasks. First we'll explain how they work and then we'll dive into the architecture we've used for deep sentiment analysis.

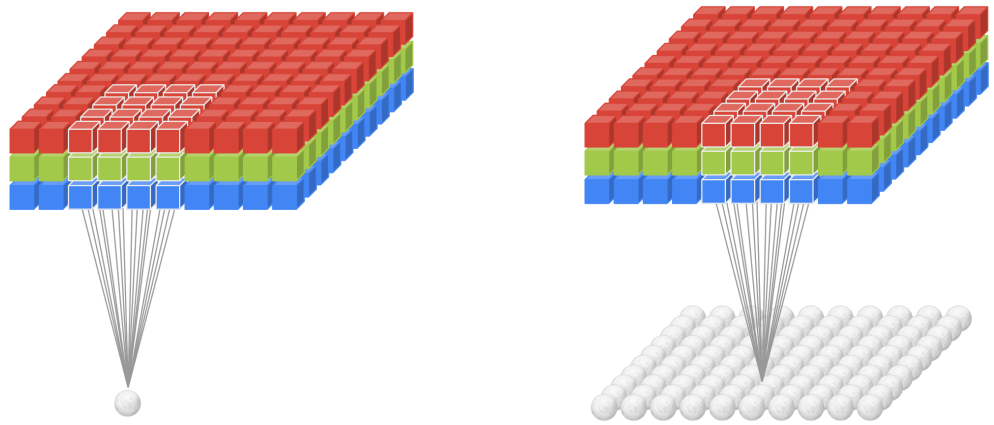
2.1 Convolutional Neural Networks

Convolutional neural networks, often called ConvNets, were inspired by the work of Hubel and Wiesel [**?**] on the human visual cortex. They found that our visual cortex operates as a complex arrangement of cells, where each cell is receptive to a small region of the visual field and is called a *receptive field*.

2.1.1 Convolutional Layer

Take an image of dimension $(h, w, 3)$ with h the height, w the width and *3* representing the number of channels – red, blue and green. If you simply flatten that image and transform it into a vector of size $h \times w \times 3$ and feed it to a neural network, you'll get poor results as you've thrown away all the spatial information. Convolutions extract that spatial information and work the following way:

- Each convolution is described by a filter F of size $(f, f, 3)$, f usually being equal to 3, 5, or 7.
- Position the filter on the upper left of the image and element-wise multiply the $f \times f \times 3$ chunk of image with the filter, then sum those numbers to obtain a 'neuron'.
- Slide across the image, one pixel at a time horizontally and vertically, and repeat the previous operation (see Figure 2.1).



(a) One operation of convolution

(b) The result of the convolution

Figure 2.1: A convolution, where each neuron is a 'receptive field' [6]

By sliding through the image, you will get a new matrix of dimension $(h_{new}, w_{new}, 1)$, with $h_{new} = h - f + 1$ and $w_{new} = w - f + 1$. However, we usually don't want to reduce the size of our input image that fast, as we want to stack several convolutions. To ensure that the image has the same size after each convolution, zero-padding is used: we add p zeros to the borders of the input image to preserve the spatial size of the input. (illustration needed, before and after zero-padding) With zero-padding, h_{new} becomes: $h_{new} = h + 2p - f + 1$, and we want h_{new} to be equal to h:

$$h + 2p - f + 1 = h \tag{2.1}$$

Solving (2.1) gives $p = \frac{f-1}{2}$. Besides, zeros are used instead of any other number because you want the filter to activate on the pixels of the image only, therefore, setting

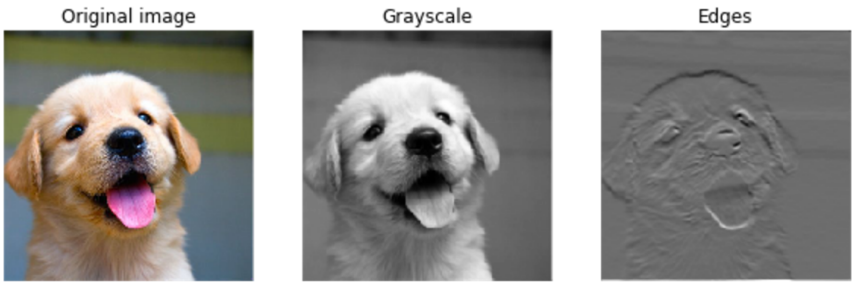


Figure 2.2: Examples of convolution

the added border to zeros ensure that the resulting neuron will not be influenced by the border.

A convolution extracts information about the image such as edges or blotches of some color (Figure 2.2). The grayscale and edges filters were hardcoded but in a ConvNet setting, the weights of the filter F are learned through optimising a loss function – in our case, a metric measuring how accurate the predictions of the emotions are. The network will learn weights that will detect features that will be most relevant to our specific task.

A convolution also has a **depth** parameter d: simply repeat the operation described above d times with d independent filters of the same size $(f, f, 3)$, to create a new tensor of dimension (h, w, d).

We can then apply convolutions on that new tensor. First layers will detect simple features such as edges or aggregation of colors, and deeper layers might recognise more complex features such as faces.

2.1.2 ReLU Layer

Stacking convolutions is nice, but as it is, we are only creating features that are linearly dependent on the input pixels: we could in fact replace all the convolutions with a single matrix multiplication. In order to learn more interesting functions, we have to add non-linearities – that is to say transforming the tensor (h, w, d) with a non-linear function. Historically, the popular choice was the sigmoid function defined as:

$$\text{sigmoid}(x) = \frac{1}{1 + e^{-x}} \tag{2.2}$$

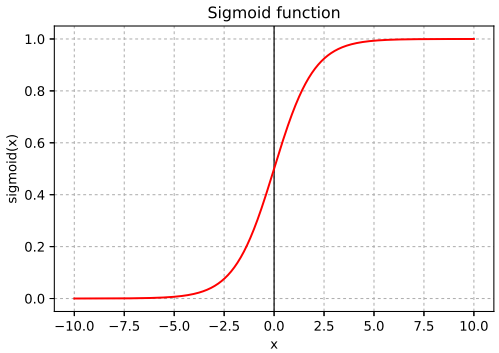


Figure 2.3: Sigmoid function

The sigmoid function is the simplest function to have values between 0 and 1, mimicking the biological neurons 'firing' in reaction to their inputs. However, when the network is learning to minimise a loss function through backpropagation, the gradients tend to vanish to zero as the sigmoid's derivative goes to zero for values that are highly negative or positive. The most popular choice of non-linearity is now the Rectified Linear Unit (ReLU) [7] defined as:

$$\text{ReLU}(x) = \max(0, x) \tag{2.3}$$

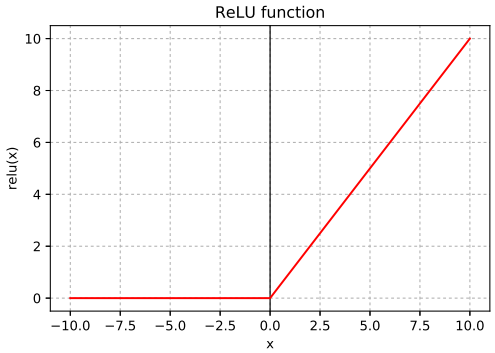


Figure 2.4: ReLU function

The ReLU's gradient is non-saturating for highly excited neurons which turns out to be a nice property to learn faster. In the network, each layer of convolution is followed by a ReLU layer, that simply applies the function $\max(0, x)$ to each neuron.

2.1.3 Pooling Layer

There is a lot of spatial redundancy in an image, we don't need all the pixels to be able to identify what's in a picture. For example we can perfectly identify the animal in Figure 2.5 by reducing the number of pixels by two.

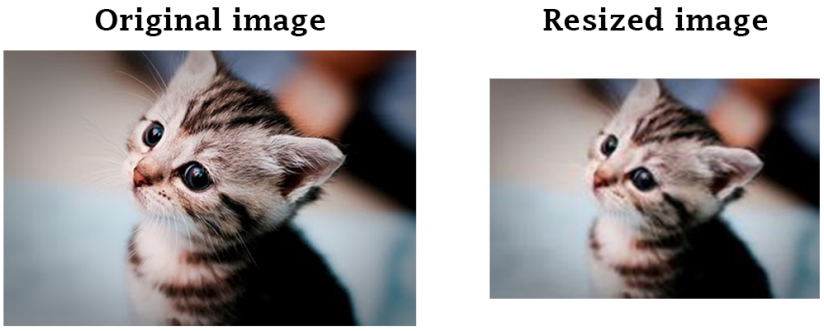


Figure 2.5: A kitten, and the same kitten with half the pixels

The same idea applies to convolved images, we might not need all the neurons that were created. The pooling operation downsamples the image in the following way:

- Pick a channel among the d ones.
- Start at the top-left 2x2 square of the image and take the max.
- Repeat by sliding through the image vertically and horizontally with a stride/step of 2 (see Figure 2.6).

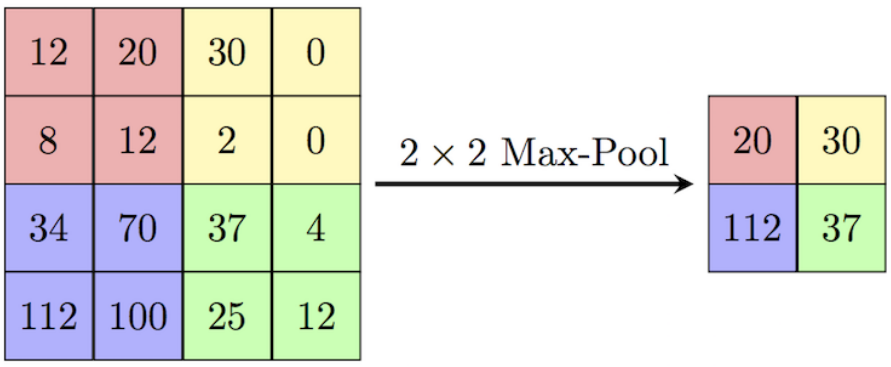


Figure 2.6: Max pooling [8]

After applying max pooling to each channel, the resulting image dimension is $(\frac{h}{2}, \frac{w}{2}, d)$ and we have discarded 75% of the neurons (as in each max-pool operation, we only keep the maximum neuron among the fours), effectively reducing the number of parameters and controlling overfitting.

One could wonder why max-pooling and not average pooling (taking the mean value of the four neurons). The convolutions allow us to see if a certain feature is in the image when a neuron fires, and we only want to know if that feature is there in a certain region. Therefore taking the max of the four neurons is sufficient to know whether that feature is there or not in that particular region.

In practice, after a few iterations of convolutions, inserting pooling layers in-between convolutional layers might be a good idea to control the spatial complexity of the network.

2.1.4 An Example of Convolutional Network

Here is an example of a convolutional neural network with an input image of size $(224, 224, 3)$:

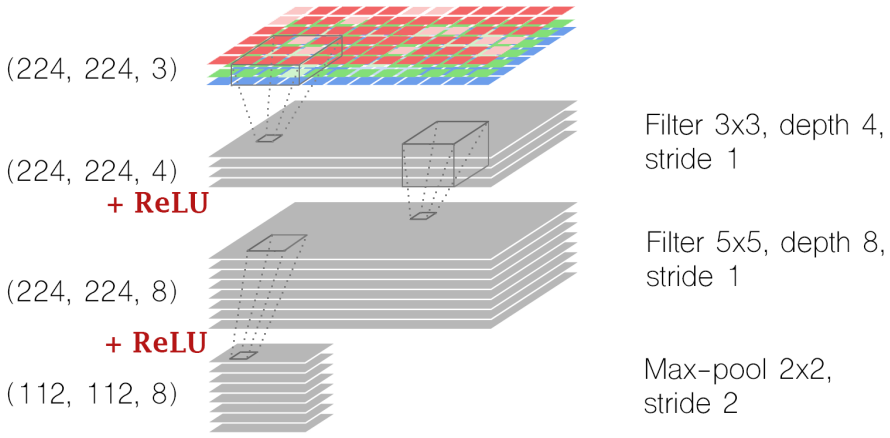


Figure 2.7: The architecture of a convolutional neural network [6]

- A first convolution with a filter of size 3×3 is applied, with depth 4, stride 1 and zero-padding of 1.
- A ReLU layer.

- A second convolution with a filter of size 5×5 is applied, with depth 8, stride 1 and zero-padding of 2.
- A ReLU layer.
- Max-pooling of size 2×2 with stride 2, reducing the height and width by 2.

After the last operation, the neurons are reshaped into a vector that can be fed to the traditional fully connected layers of neural networks.

2.2 Deep Convolutional Networks

Best results are achieved using deep convolutional networks, that is to say by stacking many layers of convolutions/ReLU/max-pool. But what exactly is 'many'? Let us have a look at the main Computer Vision competition: ImageNet Large Scale Visual Recognition (ILSVR).

1. **AlexNet** [9]: The first popular convolutional network, developed by Alex Krizhevsky, Ilya Sutskever and Geoffrey Hinton, that outperformed the other competitors at ILSVR 2012 by a large margin: top-5 error of 16% compared to the runner-up with 26%. AlexNet has 5 convolutional layers (followed by ReLU), 3 max-pool layers and 3 fully-connected layers, producing a 8-layer deep network (not counting the max-pooling as it doesn't have any parameters).

2. **GoogLeNet** (also known as Inception) [10]: The the winner of ILSVR 2014 with a top-5 error of 6.7% . This 22-layer architecture used the 'Inception Module' (that will be described shortly) which allowed to drastically reduce the number of parameters: from 60M for AlexNet to 4M for GoogLeNet.

3. **ResNet** [11]: The winner of ILSVR 2015 with a top-5 error of 3.6% thanks to an astonishing 152 layers convolutional network. This architecture features 'skip connections' allowing this ultra-deep network achieve such results.

2.3 Transfer Learning

Training a convolutional network from scratch can be difficult as a large amount of data is needed and plenty of different architectures and hyperparameters need to be tried before finding a decent model. To circumvent that issue, we can take advantage of the pre-trained models available that learned to recognize images through near 1.2M training examples from the ImageNet dataset, and a deep architecture that took weeks to train on multiple GPUs.

More specifically, the pre-trained networks learned to recognise features on a picture in order to classify the latter among the 1000 classes on the ImageNet dataset. Those features are combined in the final output layer (of size 1000, each neuron being a class probability). Suppose that instead of classifying an image into 1000 classes we want to label it according to 6 different emotions (happy, sad, angry, scared, surprised, disgusted). The same features can be combined in a different way to let the network take a decision about what is the emotion of the image.

The process described above is called *Transfer Learning*: we chop off the last layer of the network and add our own layer given how many classes we have. We then freeze the weights of the other layers and only backpropagate through the newly created layer when training the network on our examples. If we have enough data, we can unfreeze more higher-level layers and backpropagate through them.

Earlier features of ConvNets contain more generic features (such as edges or color blobs) that can be used for any task, while later features become more specific to the details of the classes present in the dataset. For example in ImageNet, there are many dog breeds and the later representational power might be used to distinguish those [12]. We will be using Google's Inception network and fine-tune through the last 3 layers. (that number is subject to change)

2.4 Google Inception Network

2.4.1 Motivation

After AlexNet proved that convolutional networks outperformed traditional machine learning models, the trend to achieve even better results was to build wider (more units per layer) and deeper (more layers) networks and to add dropout to address overfitting. However, bigger networks are more expensive to train (more parameters) and could not be usable for real-time prediction if a forward-pass takes more than a second for instance. Moreover, if the increased capacity is not used efficiently, for example if most added weights are close to zero, then the extra depth and width will be completely wasted.

To address this problem, we could replace the fully connected layers by sparse ones, even inside convolutions [10]. Not only would it mimic more closely biological systems, but it would also have more theoretical ground thanks to the work of Arora et al. [13]. They proved that if the probability distribution of a dataset can be represented by a large, very sparse neural network, then it's possible to build the optimal network layer after layer by clustering highly correlated neurons of the preceding layer. By clustering, they mean regrouping neurons into a single entity, by assigning weights to the neurons, that will be fed to the next layer. This process also resonates well with the Hebbian principle *neurons that fire together, wire together* [14].

Current networks architectures are not using sparse layers as the libraries are heavily optimised for dense matrix multiplication. Training sparse layers would incur considerable overhead (lookups, cache) that are not handled by today's computing infrastructures. However, the Inception module is an elegant solution to add more expressiveness to a network while keeping the number of parameters low.

2.4.2 Inception Module

At each layer, we would normally be facing the dilemma of choosing between 1×1, 3×3, 5×5 convolution or max-pooling:

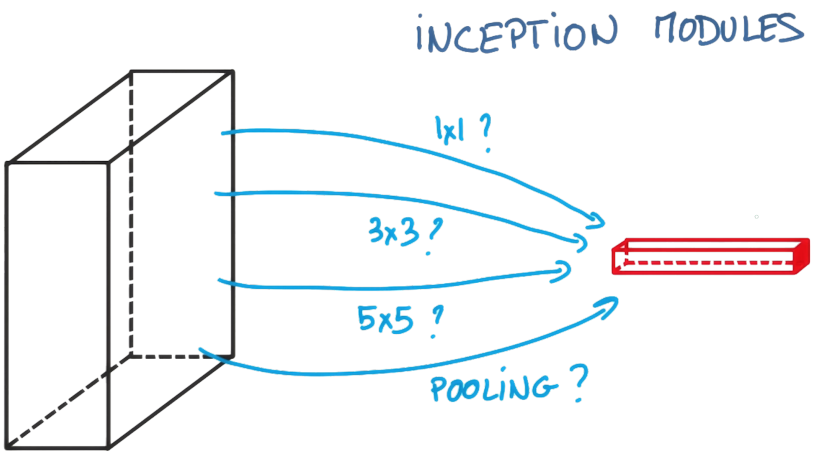


Figure 2.8: Which layer to choose? [15]

In the Inception module, we perform all of the above operations and let the network decide for us. Each operation is done in parallel before being concatenated and fed to the next layer. This allows to capture both local features via small convolutions and more high-level features via large convolutions.

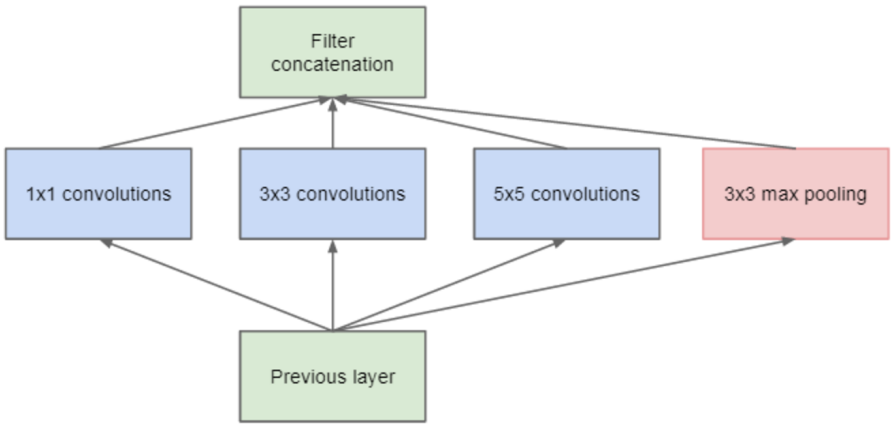


Figure 2.9: Naive Inception module [10]

As it is, the Inception module seems to rather increase the number of parameters! But this is actually the naive implementation of the module. One key component are the 1×1 convolutions, which behave like clustering the highly correlated neurons, before the large convolutions. Let's take an example to understand: suppose at an arbitrary layer, your input is size $(14, 14, 480)$.

- A **5×5 convolution, depth 48**: requires $(14^2)(480)(5^2)(48) = 112,896,000$

operations (supposing stride 1 and zero-padding).

- An **1×1 convolution, depth 16, followed by a 5×5 convolution, depth 48:** requires $[(14^2)(480)(1^2)(16)] + [(14^2)(16)(5^2)(48)] = 5,268,480$ operations.

The second operation with 1×1 convolutions is more than twenty times faster! The number of parameters is also reduced by twenty as the reduction factor is the same (to get the actual number of parameters, you only need to divide the above by 14^2). The final Inception module is the following:

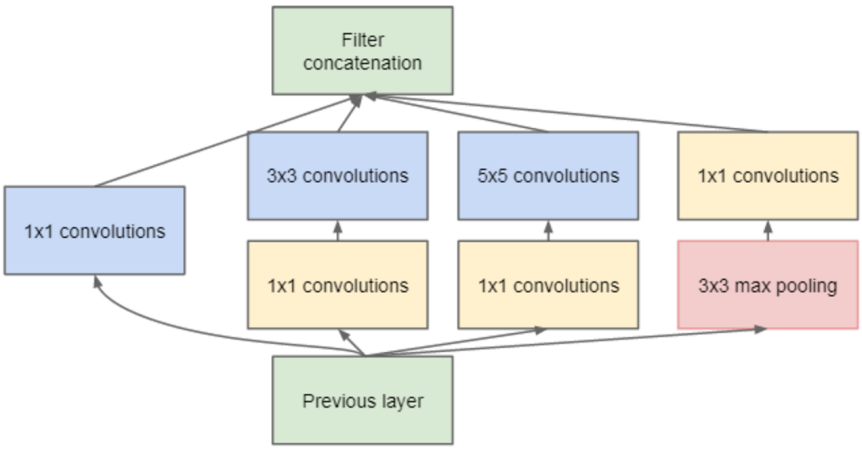


Figure 2.10: Inception module [10]

2.4.3 GoogleNet

The complete GoogleNet architecture is:

type	patch size/ stride	output size	depth	#1×1	#3×3 reduce	#3×3	#5×5 reduce	#5×5	pool proj	params	ops
convolution	7×7/2	112×112×64	1							2.7K	34M
max pool	3×3/2	56×56×64	0								
convolution	3×3/1	56×56×192	2		64	192				112K	360M
max pool	3×3/2	28×28×192	0								
inception (3a)		28×28×256	2	64	96	128	16	32	32	159K	128M
inception (3b)		28×28×480	2	128	128	192	32	96	64	380K	304M
max pool	3×3/2	14×14×480	0								
inception (4a)		14×14×512	2	192	96	208	16	48	64	364K	73M
inception (4b)		14×14×512	2	160	112	224	24	64	64	437K	88M
inception (4c)		14×14×512	2	128	128	256	24	64	64	463K	100M
inception (4d)		14×14×528	2	112	144	288	32	64	64	580K	119M
inception (4e)		14×14×832	2	256	160	320	32	128	128	840K	170M
max pool	3×3/2	7×7×832	0								
inception (5a)		7×7×832	2	256	160	320	32	128	128	1072K	54M
inception (5b)		7×7×1024	2	384	192	384	48	128	128	1388K	71M
avg pool	7×7/1	1×1×1024	0								
dropout (40%)		1×1×1024	0								
linear		1×1×1000	1							1000K	1M
softmax		1×1×1000	0								

Figure 2.11: GoogleNet architecture [10]

The '#3 × 3 reduce' and '#5 × 5 reduce' refer to the dimensionality reduction with the 1×1 convolution. The 'pool proj' refers to the depth of the 1×1 convolution following the 3 × 3 max-pool with stride 1 and zero-padding 1.

In our model, we got rid of the last linear layer and replaced it with another linear layer of size $(1, 1, 6)$ for the 6 different emotions.

2.5 Results

Are the images enough to accurately determine the emotion conveyed by the user? We tried several machine learning models on the raw images, that were resized to a fixed size $(224, 224, 3)$.

- **Softmax regression**: with L_2 regularisation of 0.01.
- **Random Forest Classification**: with 1000 trees and max depth of 5.
- **5 layers Convolutional Neural Network**: (each convolution is with stride 1, zero-padding to keep image size and followed by ReLU)
  - 3×3 convolution, depth 8
  - 3×3 convolution, depth 16
  - 2×2 max-pooling with stride 2
  - 3×3 convolution, depth 32
  - 2×2 max-pooling with stride 2
  - Fully connected layer
  - Fully connected layer
- **Inception fine-tuned**: As described in 3.1.6, we retrained the final layer of the Inception model with:
  - 20 epochs (1 epoch = 1 full sampling of the training data)
  - Mini-batch size of 32
  - Adam optimizer with learning rate 1e−6

After the preprocessing described in Section 2, the dataset contains 295,508 posts that we split as 80% train set and 20% test set. The accuracy is the fraction of correctly classified images. These accuracies have to be compared to a baseline: random guessing. Actually this guessing will include the prior probabilities of the classes:

	happiness	**sadness**	**anger**	**surprise**	**fear**	**disgust**
Prior proba.	0.32	0.22	0.19	0.03	0.22	0.02

Table 2.1: Prior probabilities of the classes

And the results:

	Train accuracy	**Test accuracy**
Random guessing	0.24	0.24
Softmax regression	0.41	0.34
Random forest	0.60	0.44
Inception fine-tuned	0.48	0.42

Table 2.2: Comparison of models using raw images

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

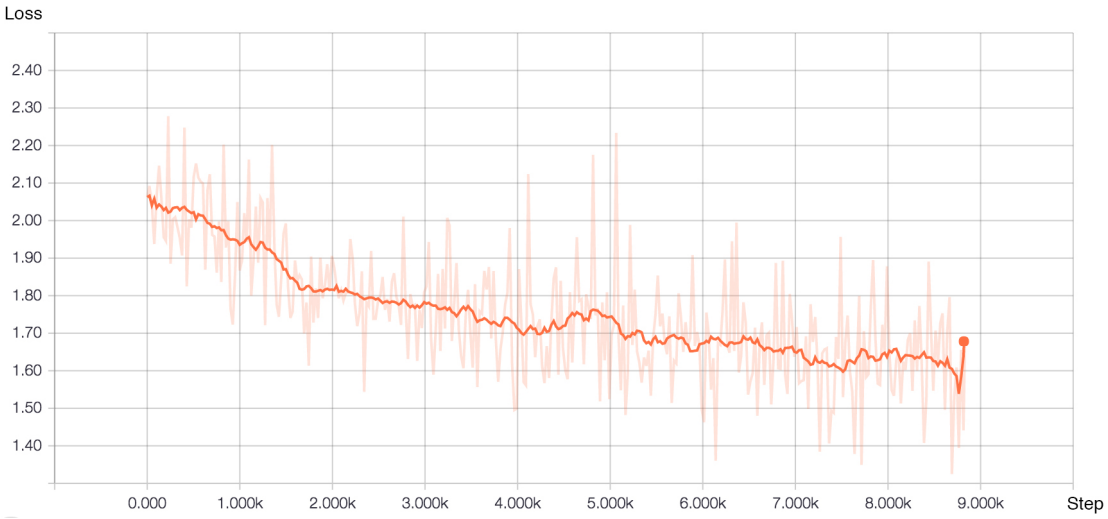


Figure 2.12: Loss function of the Inception fine-tuned model

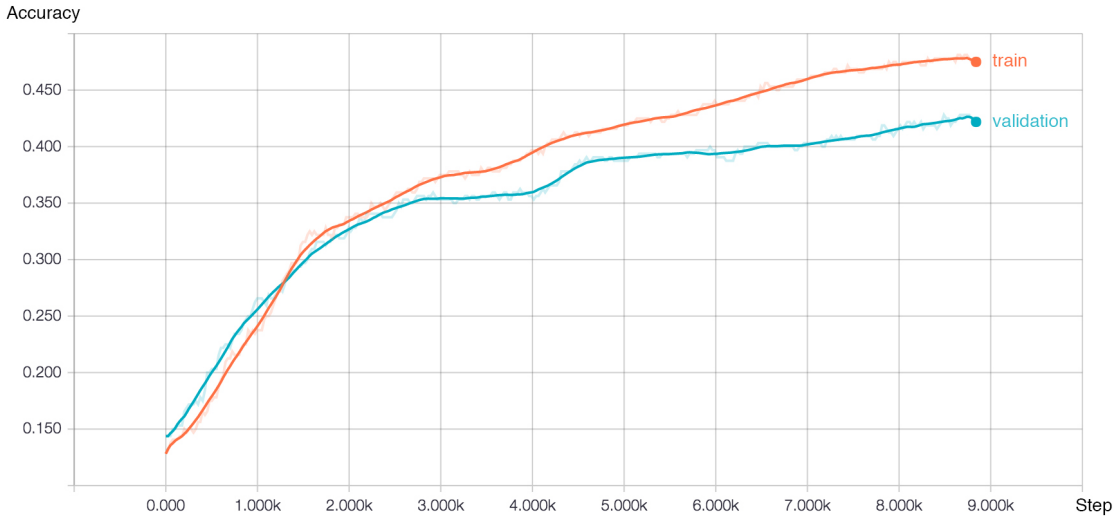


Figure 2.13: Train/Validation accuracies of the Inception fine-tuned model

The results are quite satisfactory as even if some images are easy to read:

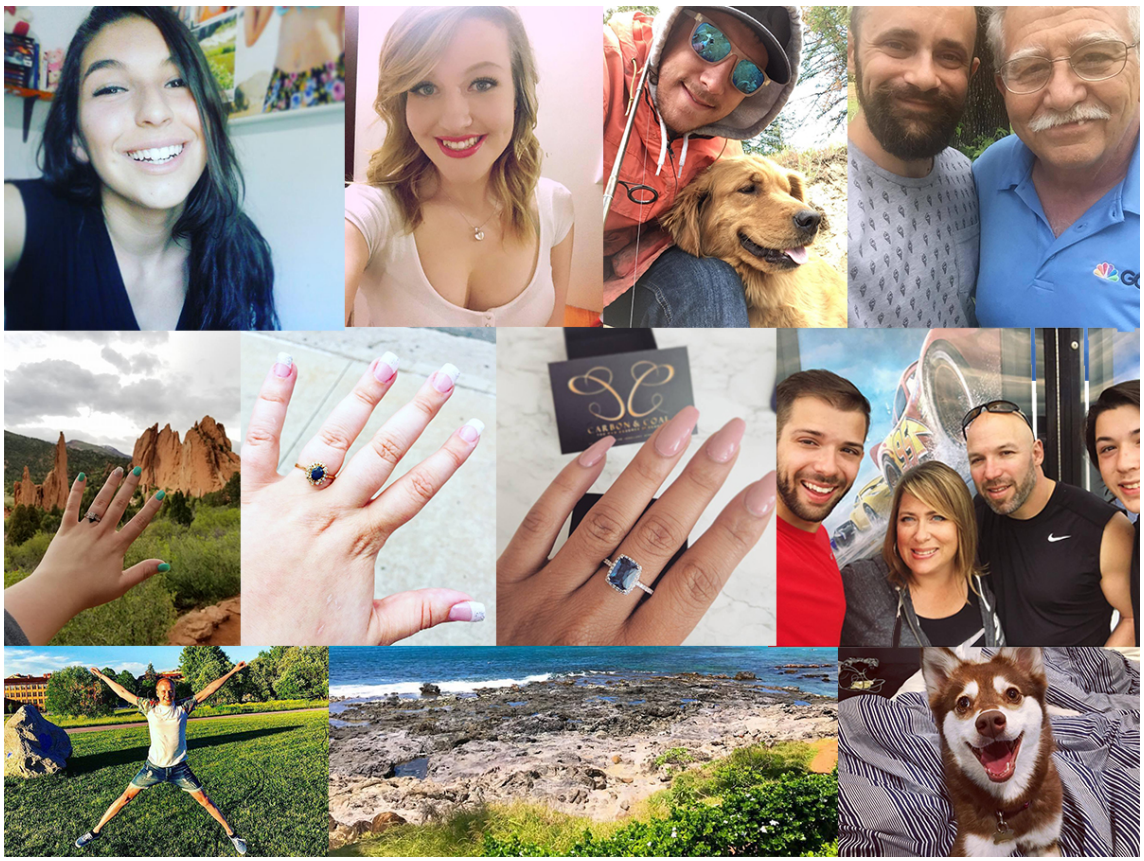


Figure 2.14: Easy happy images

Some other images are more challenging:

Figure 2.15: Hard happy images

- visualise classified images with confidence, misclassified images

Chapter 3

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 3.1

Figure 3.1: Which emotion is it?

It's unclear whether the user is sad or disgusted. Only by 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 is: *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.

3.1 Word Embedding

One way to map 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 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 3.2.

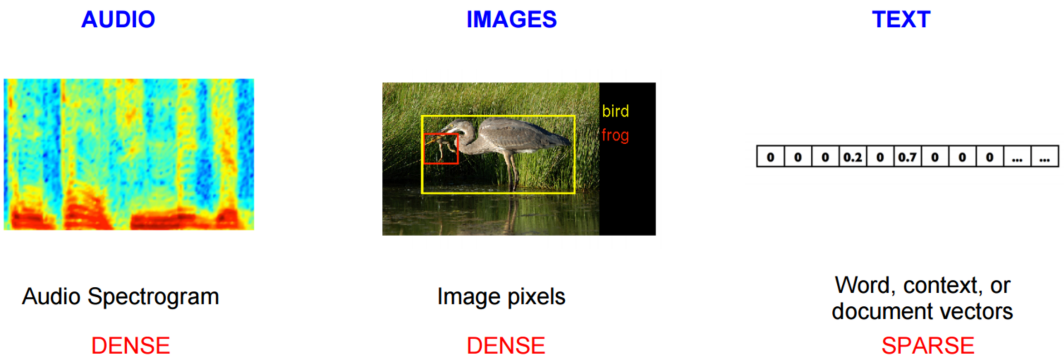


Figure 3.2: Data sparsity in text [17]

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 keep semantic relationships. Therefore, what the model has learned about dogs can be used when faced with a cat.

3.1.1 Word2Vec Overview

The Word2Vec model by Mikolov et al. [16] is an efficient implementation of word embedding and works as follow: Suppose you have a sentence: "the ants in the garden". We can break that sentence in (context, target) pairs where the context are the words surrounding the target word. For example, if we take a context with a window of 1, we get the following pairs: ([the, in], ants), ([ants, the], in), ([in, garden], the) [we omitted the pairs where the context wasn't of size 2]. 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 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, 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), (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 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.

3.1.2 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:

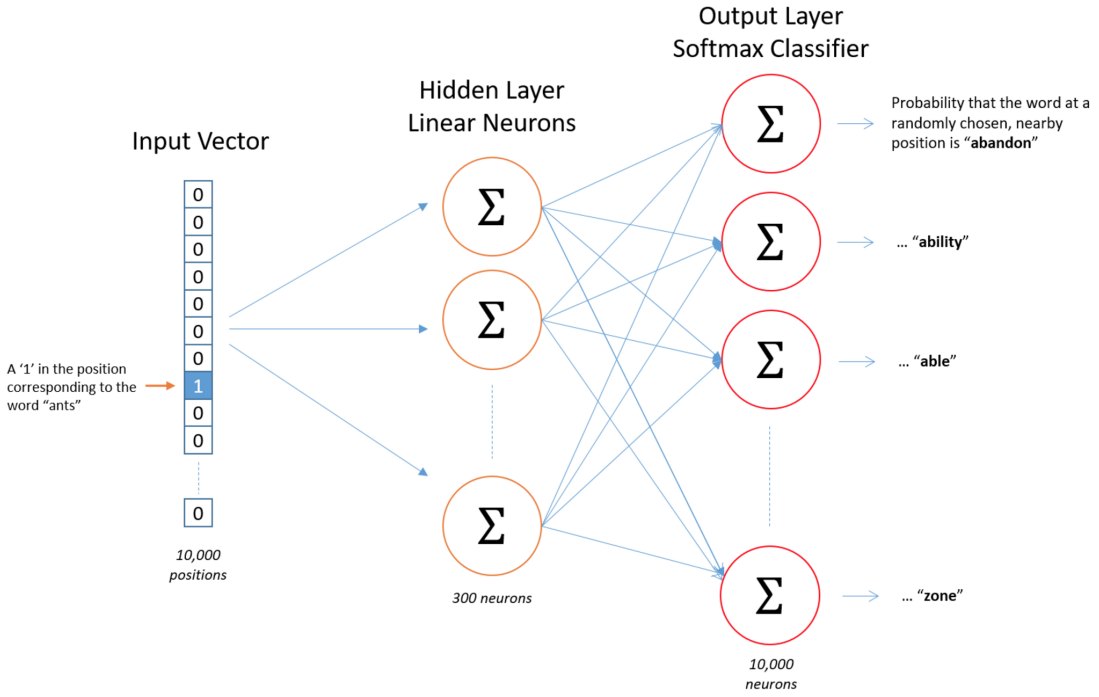


Figure 3.3: Skip-Gram model architecture [18]

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:

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

Figure 3.4: Where the embedding comes from [18]

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

$$\begin{aligned} \mathrm{J_{CE}} &= -\log P(w_c|w_t) \\ &= -\log \{\mathrm{softmax}(w_c, w_t)\} \\ &= -\log \left(\frac{\exp\{\mathrm{score}(w_c, w_t)\}}{\sum_{\text{Word w' in Vocab}} \exp\{\mathrm{score}(w_c, w')\}} \right) \end{aligned} \tag{3.1}$$

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

3.1.3 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'.

3.2 Training the Model

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. 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 without a few tweaks:

1. Negative sampling
2. Word pairs and phrases
3. Subsampling of frequent words

3.2.1 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. [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 all the other words:

$$P(w_i) = \frac{f_i^{3/4}}{\sum_{j=1}^{V} f_j^{3/4}} \tag{3.2}$$

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 of the input word zero-out every weight in the hidden layer that does not belong to the input word.

3.2.2 Negative Sampling, with Maths

The loss function (3.1) 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. 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 3.5.

(a) Softmax loss (b) Negative Sampling loss

Figure 3.5: 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:

$$\mathrm{J}_{\mathrm{NEG}} = -[\log\, P(y = 1|w_c, w_t) + k\mathbb{E}_{\tilde{w}\sim Q}[\log\, P(y = 0|\tilde{w}, w_t]] \tag{3.3}$$

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:

$$\begin{aligned} \mathrm{J}_{\mathrm{NEG}} &= -\left[\log\, P(y=1|w_c,w_t) + k\sum_{j=1}^{k}\frac{1}{k}\log\, P(y=0|\tilde{w}_j,w_t]\right] \\ &= -\left[\log\, P(y=1|w_c,w_t) + \sum_{j=1}^{k}\log\, P(y=0|\tilde{w}_j,w_t)\right] \end{aligned} \tag{3.4}$$

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:

$$P_{\text{train}}(w_c|w_t) = \text{softmax}(w_c,w_t) \tag{3.5}$$

And the unigram distribution Q to sample the noise:

$$Q(w_i) = \frac{f_i^{3/4}}{\sum_{j=1}^{V} f_j^{3/4}} \tag{3.6}$$

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

$$\begin{aligned} P(y=1|w_c,w_t) &= \frac{\frac{1}{k+1}P_{\text{train}}(w_c|w_t)}{\frac{1}{k+1}P_{\text{train}}(w_c|w_t) + \frac{k}{k+1}Q(w_c)} \\ &= \frac{P_{\text{train}}(w_c|w_t)}{P_{\text{train}}(w_c|w_t) + kQ(w_c)} \end{aligned} \tag{3.7}$$

Therefore we also have:

$$P(y=0|w_c,w_t) = \frac{kQ(w_c)}{P_{\text{train}}(w_c|w_t) + kQ(w_c)} \tag{3.8}$$

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

factor $Z(w_c)$:

$$\begin{aligned} P_{\text{train}}(w_c|w_t) &= \frac{\exp\{\text{score}(w_c, w_t)\}}{\sum_{\text{Word w' in Vocab}} \exp\{\text{score}(w_c, w')\}} \\ &= \frac{\exp\{\text{score}(w_c, w_t)\}}{Z(w_c)} \end{aligned} \tag{3.9}$$

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 the parameter was close to 1 with a low variance. Setting $Z(w_c)$ to 1 gives:

$$P(y = 1|w_c, w_t) = \frac{\exp\{\text{score}(w_c, w_t)\}}{\exp\{\text{score}(w_c, w_t)\} + kQ(w_c)} \tag{3.10}$$

The loss function becomes:

$$\text{J}_{\text{NEG}} = -\left[\log \frac{\exp\{\text{score}(w_c, w_t)\}}{\exp\{\text{score}(w_c, w_t)\} + kQ(w_c)} + \sum_{j=1}^{k} \log \frac{kQ(\tilde{w}_j)}{\exp\{\text{score}(\tilde{w}_j, w_t)\} + kQ(\tilde{w}_j)}\right] \tag{3.11}$$

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 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 [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:

$$P(y = 1|w_c, w_t) = \frac{1}{1 + \exp\{-\text{score}(w_c, w_t)\}} \tag{3.12}$$

Giving the following final loss function:

$$\text{J}_{\text{NEG}} = -\left[\log \frac{1}{1 + \exp\{-\text{score}(w_c, w_t)\}} + \sum_{j=1}^{k} \log \frac{1}{1 + \exp\{\text{score}(\tilde{w}_j, w_t)\}}\right] \tag{3.13}$$

3.2.3 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:

$$\text{score}(w_i, w_j) = \frac{\text{count}(w_i w_j) - \delta}{\text{count}(w_i) \times \text{count}(w_j)} \times |W| \tag{3.14}$$

With:

- $\text{count}(w_i)$ the number of time the word w_i appear in the corpus (all the training data).
- $\text{count}(w_i w_j)$ 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.
- $|W|$ 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 2-4 passes with a decreasing threshold value after each pass).

3.2.4 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 vector representation of that 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:

$$P(w_i) = \frac{t}{f_i} + \sqrt{\frac{t}{f_i}} \tag{3.15}$$

With t a parameter controlling how aggressive subsampling is, smaller value of t means more subsampling.

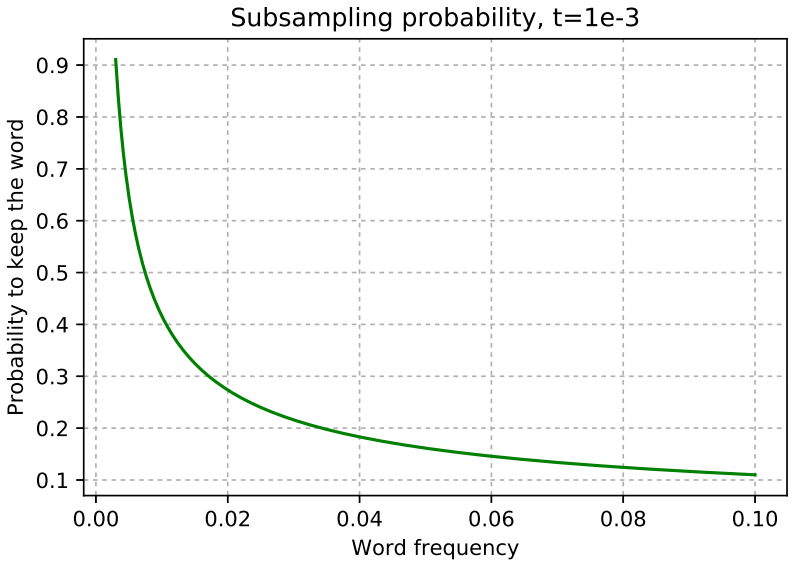


Figure 3.6: Subsampling probability graph

Note that the formula (3.15) is different from the one in [19], but we decided to

keep (3.15) 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.

3.2.5 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 model was trained on Google News dataset (aggregation of news from all around the world, including blog posts) containing about 100 billion words. Each word in the vocabulary (3 million words) are embedded into a 300 dimensional vector.

The skip-gram model had the following parameters:

- Context window size of 10.
- 5 negative examples for Negative Sampling.
- In Word Pairs and Phrases, δ is equal to 100, and the threshold is set to 100. The number of passes was not found [need to look].
- Subsampling threshold of $1e{-}3$.

3.3 Results

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] [Elaborate on mean embedding].

The network architecture is straight-forward as the output softmax layer directly follows the mean embedding [illustration required]. This model achieves **55%** accuracy on the test set. This is not bad at all, but note that the word order is completely lost! The order can be preserved using Recurrent Neural Networks, which is our next section.

Chapter 4

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.

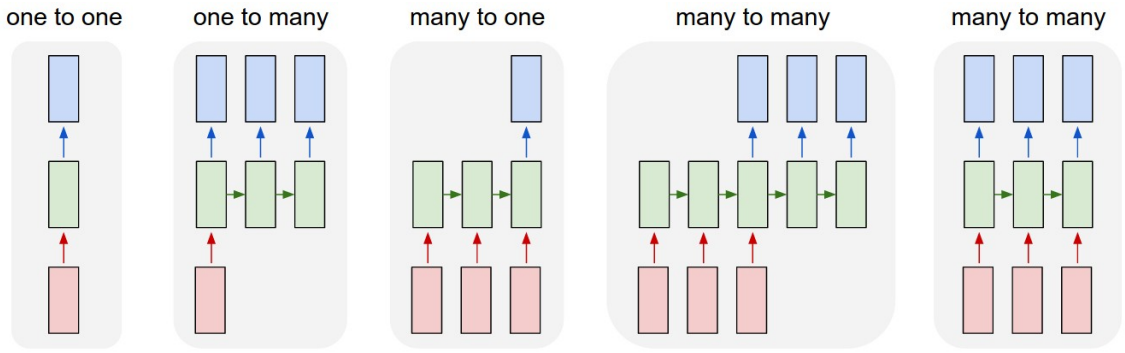


Figure 4.1: The applications of Recurrent Neural Networks [24]

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.

4.1 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 as shown on Figure (4.2).

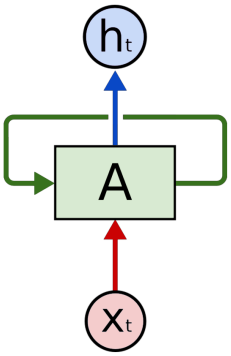


Figure 4.2: A vanilla recurrent neural network [25]

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

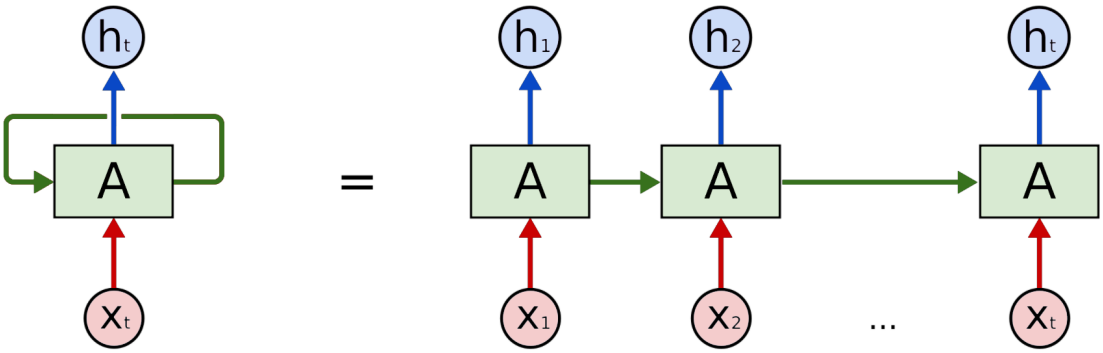


Figure 4.3: An unrolled vanilla recurrent neural network [25]

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

$$h_t = f(h_{t-1}, x_t) \tag{4.1}$$

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.

Usually f is simply:

$$h_t = \tanh(W_x x_t + W_h h_{t-1}) \tag{4.2}$$

Just like the layer traditional neural network with a tanh (hyperbolic tangent) activation function and two inputs instead of one.

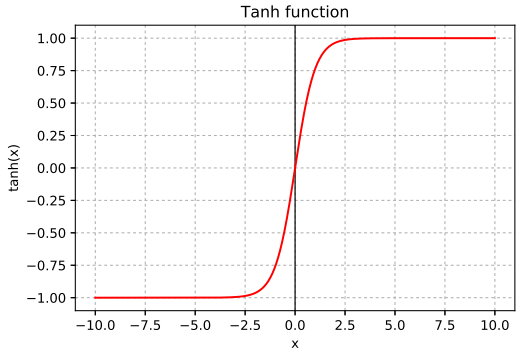


Figure 4.4: Another activation function: tanh

This vanilla RNN is a good step towards learning from sequential inputs but most people use a variant called Long-Short Term Memory (LSTM) that work extremely well on many different tasks.

4.2 Long-Short Term Memory Networks

The vanilla RNN is hard to train as the gradients often either vanish or explode. That's because during backpropagation, the weights W_x and W_h go through repeated matrix multiplication [detail the maths] than can either make them go to zero or diverge. 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.

$$a = W_x x + W_h h \tag{4.3}$$

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: $a_i, a_f, a_o, a_g \in \mathbb{R}^H$, where a_i is made up of the first H elements of a, a_f the H next, etc. Now we compute 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 $g \in \mathbb{R}^H$ as:

$$i = \sigma(a_i) \qquad f = \sigma(a_f) \qquad o = \sigma(a_o) \qquad g = \tanh(a_g) \tag{4.4}$$

where σ is the sigmoid function.

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

$$c_t = i \odot g + f \odot c_{t-1} \qquad h_t = o \odot \tanh(c_t) \tag{4.5}$$

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 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 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 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 a tanh and not a 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.

4.3 LSTM for Sentiment Analysis

We'll be using the following architecture:

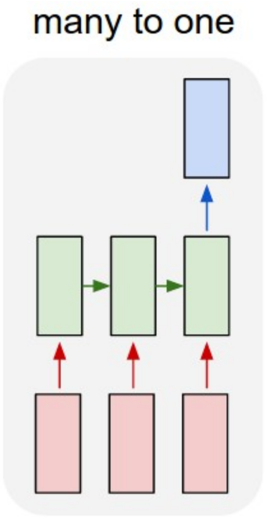


Figure 4.5: Many to one architecture

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> – as $[3, 20, 1, 49, 6, 0, 0, ..., 0]$. The LSTM will then give the prediction by stopping before the zero-padding.

4.4 Results

(upcoming)

Chapter 5

Deep Sentiment

Deep Sentiment is the name of the deep neural network architecture merging both visual recognition and text analysis.

5.1 The Architecture

An illustration is worth a thousand words:

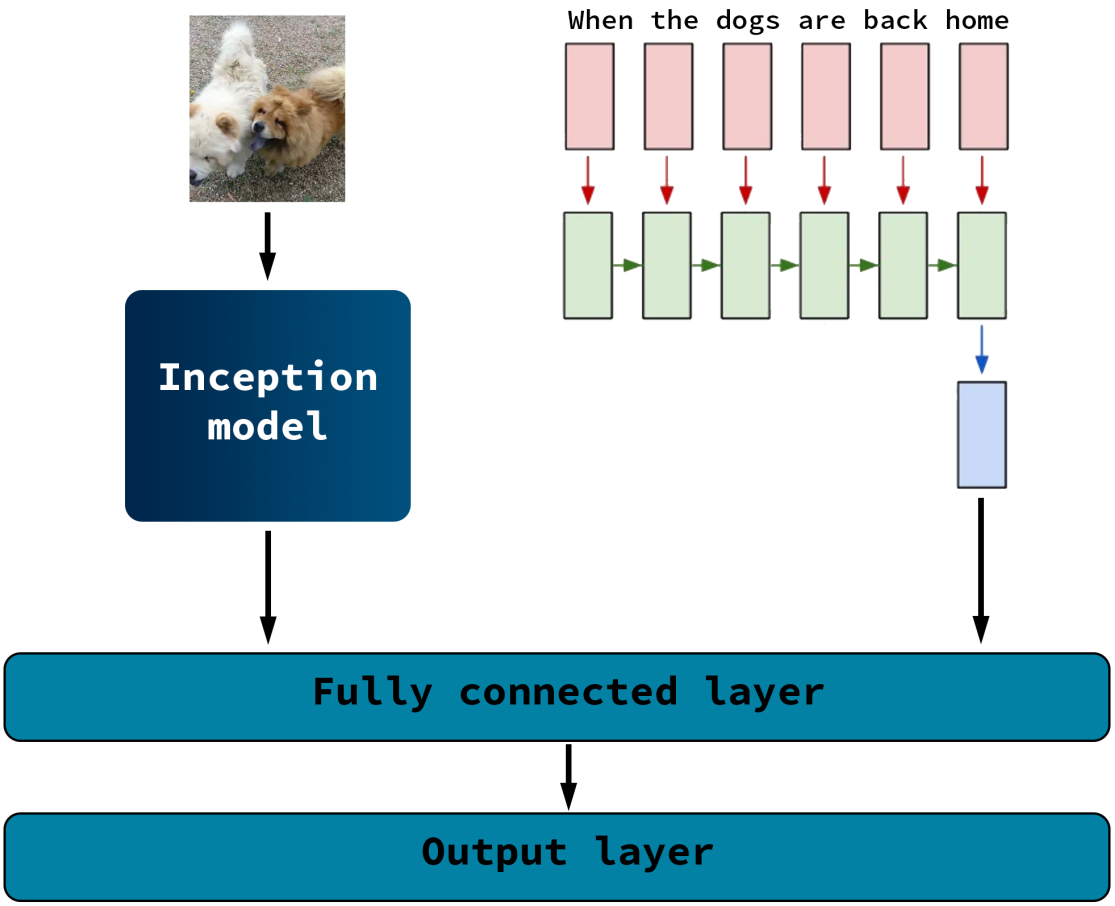


Figure 5.1: Deep Sentiment architecture

1. The image will go through the pre-trained Inception model that will extract features from the images.

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

3. The two outputs will be combined in a fully-connected layer.

4. The final layer will give the probability distribution of the emotion of the post.

Chapter 6

Generation of Tumblr Posts

This chapter will be about image and text generation.

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

[4] 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

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

[6] 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.

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

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

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

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

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

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

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

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

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

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

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

[18] C. McCormick, Word2Vec Tutorial - The Skip-Gram Model, 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/