Mohamed Khider University of Biskra

Faculty of Sciences and Technology

Department of Electrical engineering

MASTER'S THESIS

Electrical Engineering
Telecommunications
Networks and Telecommunications

Ref. :

Submitted and defended by:

Touhami Moussa

On : 24-6-2021

Handwritten Digits Recognition Using Neural Networks

Board of Examiners :

Pr.	A-K OUAFI	MAA	University of Biskra	President
Dr.	S MEDOUAKH	MCA	University of Biskra	Examiner
Pr.	Baarir Zine Eddine	PR	University of Biskra	Supervisor

Academic Year : 2020 – 2021

Dedication

First and foremost, praises and thanks to Allah, the Almighty. I want to thank my Parents for their unconditional love and support, I also want to acknowledge my sincere gratitude to the people on the Internet who are dedicating their time and efforts to make all kinds of scientific information accessible for free, these people are truly making the world a better place. Finally Special thanks to Pr. Dr. Baarir Zine Eddine.

Abstract

In this research I explore the theoretical aspects and the working machinery of neural networks by solving the classic example of handwritten digits recognition. I will start with their simplest form with no added frills known as Multi-Layer Perceptron (MLP), which will be a necessary prerequisite for understanding any of the more powerful modern variants, like Convolution Neural Networks which are explored as well in this research. I describe the main algorithms and techniques used by MLP, and other sophisticated variants. And I will focus on the intuition for how neural networks can behave intelligently, and what we mean by learning and how it occurs exactly. The adopted strategies, in the learning and in the testing phase, lead to high performances in terms of recognition rate and speed. We trained and tested the neural network using the MNIST handwritten digits database. Obviously, the choice of neural networks over other classification methods comes down to their performance.

Since the title of the research suggests handwritten digits recognition, I will dedicate the first chapter to exposes the need for handwritten recognition and Optical Character Recognition (OCR) systems, and the steps required for these systems along with in depth look to some of the preprocessing steps, namely slant removal, and skew correction.

In the second chapter I describe the key concepts of feed-forward neural networks, starting with their origins, how these structures can behave intelligently, and how learning occurs and the rules that govern it.

And as for our learning algorithm the Stochastic-Gradient-Descent (SGD) is discussed in the third chapter. I will dive in its details, and derive the formulas for updating the network parameters, so learning occurs. Then I will focus on the back-propagation algorithm, and provide a full step-by-step walk through on how it allows us to calculate the gradient efficiently.

The fourth chapter is dedicated toward exploring various optimization and regularization techniques, to improve the performance of our model.

The final chapter is dedicated toward exploring Convolutional Neural Networks (CNN) as an alternative for Fully-connected NN, I will go back to the back-propagation and derive its formulas in CNN, then I explore Depth-wise-separable convolution as an alternative for the standard one. Finally, I go back to an important aspect of NNs, which is the activation function and consider other better activation functions.

Keywords

Handwritten recognition, Neural networks, Back-Propagation, Regularization, CNN

Table of contents

Table of Figures

General Introduction

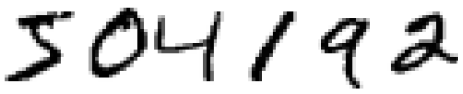
Due to increased usage of digital technologies in all sectors and in almost all day-to-day activities to store and pass information, Handwriting character recognition has become a popular subject of research. Handwriting remains relevant, because people still want to have Handwriting copies converted into electronic copies that can be communicated and stored electronically [9]. Wikipedia defines handwriting recognition as: "the ability of a computer to receive and interpret intelligible handwritten input from sources such as paper documents, photographs, touch-screens, and other devices.".

In the last two decades, considerable progress has been made in the field of handwriting recognition. This progress is due to the many works in this field and also the availability of international standards databases for handwriting that allowed the researchers to report in a credible way the performance of their approaches in this field, with the possibility of comparing them with other approaches which they use the same bases [10]

This research aims to report the development of a Handwriting digits recognition system. The development is based on an **Artificial Neural Network**, which is a field of study in **Artificial Intelligence**. Different techniques and methods are used to develop a Handwriting character recognition system. However, few of them focus on neural networks. The use of neural networks for recognizing Handwriting characters is more efficient and robust compared with other computing techniques. The research also outlines the methodology, design, and architecture of the Handwriting digits recognition system and testing and results of the system development. *The aim is to demonstrate the effectiveness of neural networks for Handwriting character recognition.* [9]

Problem

Consider the example of recognizing handwritten digits, illustrated in Figure bellow.:



Most people effortlessly recognize those digits as 504192. That ease is deceptive. Recognizing handwritten digits isn't easy. Rather, we humans are astoundingly good at making sense of what our eyes show us. But nearly all that work is done unconsciously. And so, we don't usually appreciate how tough a problem our visual systems solve. The difficulty of visual pattern recognition becomes apparent if you attempt to write a computer program to recognize digits like those above [1]. Images (audio as well) are a type unstructured data, which means they don't have a defined structure, but rather their structure is implicit, and it has been much harder for computers to make sense of this type of data. In fact, we as human being are very good at understanding unstructured data.

The goal is to build a system that takes 28×28-pixel image of a digits as input, and predict the corresponding digit. This is a nontrivial problem due to the wide variability of handwriting. It could be tackled using handcrafted rules or heuristics for distinguishing the digits based on the shapes of the strokes, simple intuitions about how we recognize shapes - "a 9 has a loop at the top, and a vertical stroke in the bottom right" - but in practice such an approach leads to a proliferation of rules and of exceptions to the rules and so on, and invariably gives poor results. turn out to be not so simple to express algorithmically. When you try to make such rules precise, you quickly get lost in a morass of exceptions and caveats and special cases. It seems hopeless. [1,3]

Maybe for task like these we better off using the same mechanism our visual system uses. Neural networks approach the problem in a different way. The idea is to take a large number of handwritten digits, known as training examples, and then develop a system which can learn from those training examples. In other words, the neural network uses the examples to automatically infer rules for recognizing handwritten digits. Furthermore, by increasing the number of training examples, the network can learn more about handwriting, and so improve its accuracy.

Research Objectives and organization

Objectives

Even though the title of this research suggests that the focus, is only about developing a system that uses Neural Networks, that is able to recognize handwritten digits. But the main goal of this research is aimed toward understanding the working machinery of neural networks in a deeper level. Because eventually, the gained knowledge can be applied to any real-world practical problem, not just for handwritten digits recognition, or pattern recognition in general.

Research Questions

This research is aimed to answer the following questions:

- Why a network like a structure allows for an intelligent behavior?
- How learning in neural networks occurs?
- How can we improve the performance of the neural network?

Organization

My research is divided into five chapters.

1. The first one, is dedicated toward exposing the need to handwritten character recognition in general. And dive a bit into the field of Optical Character recognition (OCR), discussing the different steps involved.
2. The second chapter, focuses on developing the key concepts of neural networks, starting from the main building block (the neuron), then discussing their architecture, and their working machinery that allow for such structure to behave intelligently.
3. The third chapter is devoted toward exploring the core algorithm that allows learning to happen, which are the stochastic gradient descent and backpropagation.
4. The fourth chapter explores different techniques to improve the performance of NNs.
5. The final chapter, dive into Convolution Neural Network.

Chapter One: Handwritten Recognition and OCR

Introduction

Handwriting recognition, also known as handwriting OCR, is a subfield of OCR (Optical character recognition) technology, that allows you to extract any printed or handwritten text from images, such as photos of street signs and products, as well as from documents—invoices, bills, financial reports, articles, and more. Handwriting characters remain complex since different individuals have different handwriting styles.[9]

Optical Character Recognition (OCR)

As the technology evolves, emergence of Optical Character Recognition (OCR) for both printed and handwritten documents of any language is obvious.

Optical character recognition is the electronic conversion of images of typed, handwritten or printed text into machine-encoded text, whether from a scanned document, a photo of a document, a scene-photo (for example the text on signs and billboards in a landscape photo) or from subtitle text superimposed on an image (for example from a television broadcast). It is widely used as a form of information entry from printed paper data records, whether passport documents, invoices, bank statements, computerized receipts, business cards, mail, printouts of static-data, or any suitable documentation. It is a common method of digitizing printed texts so that they can be electronically edited, searched, stored more compactly, displayed on-line, and used in machine processes such as cognitive computing, machine translation, (extracted) text-to-speech, key data and text mining. OCR is a field of research in pattern recognition, artificial intelligence and computer vision. [11]

Early versions needed to be trained with images of each character, and worked on one font at a time. Advanced systems capable of producing a high degree of recognition accuracy for most fonts are now common, and with support for a variety of digital image file format inputs. Some systems are capable of reproducing formatted output that closely approximates the original page including images, columns, and other non-textual components. [11]

OCR systems usually involves multiple steps, some of them are mandatory, and others are not, some are considered to be pre-processing, preparing the data to be classified and determining the presented text content.

Different steps of OCR

Various approaches have been presented which deal with the recognition of handwritten script. Some work by segmenting the word and then performing character recognition, other by avoiding the segmentation stage. However, whatever the application and the methodology is, the preprocessing stage is necessary both for the segmentation procedure as well as for the character recognition, since skewed or slanted words may cause considerable difficulties in both tasks.[17]

Pre-processing

The raw data is subjected to a number of preliminary processing steps to make it usable in the descriptive stages of character analysis [11]. Pre-processing aims to produce data that are easy for the OCR systems to operate accurately. Pre-processing might also be performed in order to speed up computation. For example, if the goal is real-time face detection in a high-resolution video stream, the computer must handle huge numbers of pixels per second, and presenting these directly to a complex pattern recognition algorithm may be computationally infeasible. Instead, the aim is to find useful features that are fast to compute, and yet that also preserve useful discriminatory information enabling faces to be distinguished from non-faces. These features are then used as the inputs to the pattern recognition algorithm. [3]

The main pre-processing steps are:

Binarization: Convert an image from color or greyscale to black-and-white (called a "binary image" because there are two colors). The task of binarization is performed as a simple way of separating the text (or any other desired image component) from the background. Because binarization provides sharper and clearer contours of various objects present in the image. This feature extraction improves the learning of AI models. In the old days binarization was important for sending faxes.

A colored image consists of 3 channels (Red, Green and Blue) with values ranging from 0 to 255. A grey scale image consists only of one channel, representing the pixels intensity.

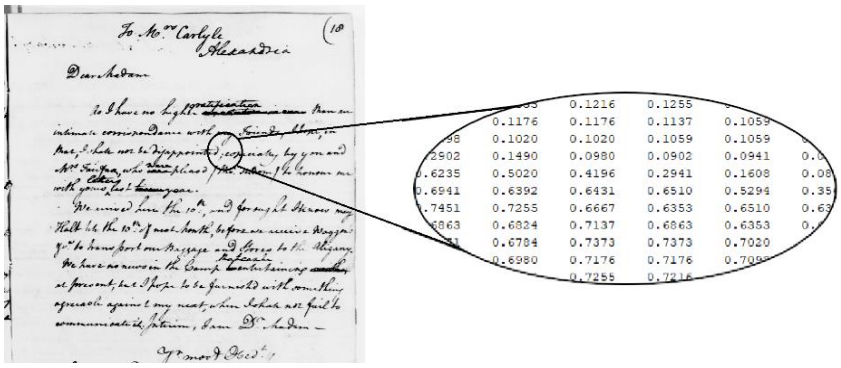


Figure 1- 1: Grayscale image representation

At first the image is converted into greyscale, then a threshold gets applied:

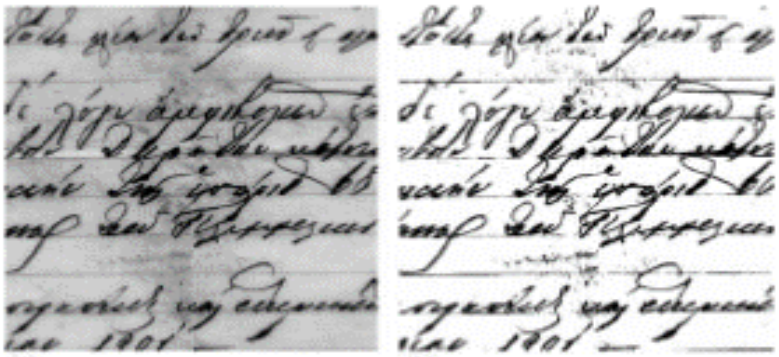


Figure 1- 2: Image binarization [11]

Note: in our work, all the image are greyscale, 0 represent black, and 1 represent white, and all values in between. The difference between greyscale and black-and-white is not so significant for our purpose.

Noise Reduction: Remove positive and negative spots, smoothing edges.

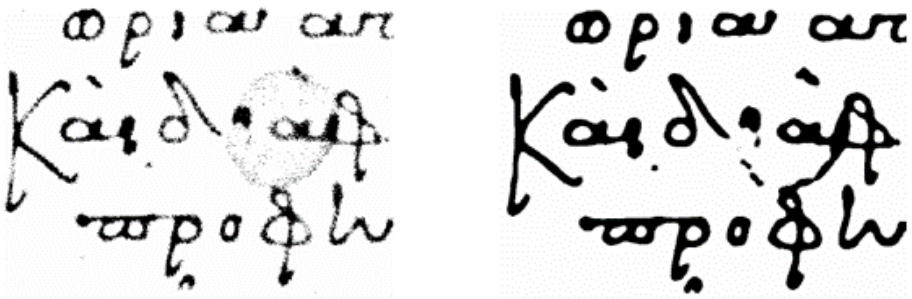


Figure 1- 3: Noise Reduction example [11]

Skew Correction: If the document was not aligned properly when scanned, it may need to be tilted a few degrees clockwise or counterclockwise in order to make lines of text perfectly horizontal or vertical. Main approaches for skew detection include correlation, projection profiles, Hough transform.

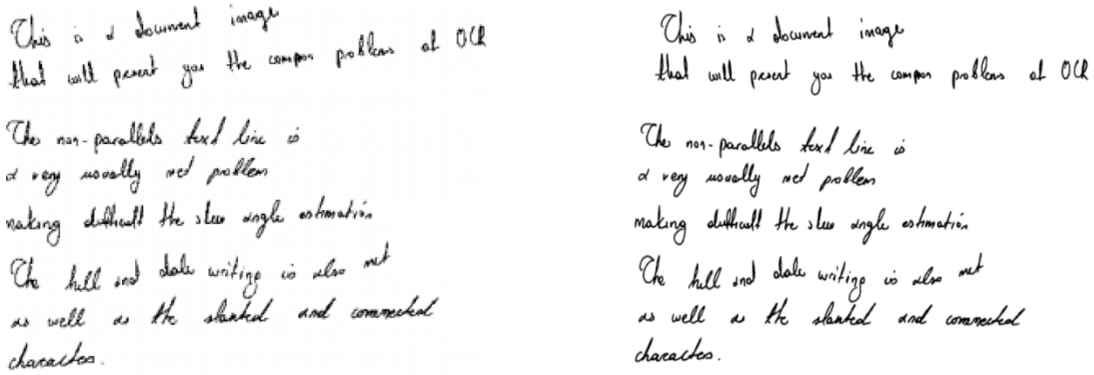


Figure 1- 4: Skew Correction example [11]

Slant Removal: The slant of handwritten texts varies from user to user. Slant removal methods are used to normalize the all characters to a standard form. Popular deslanting techniques are:

- Projection Profiles
- Calculation of the average angle of near-vertical elements
- Bozinovic–Shrihari Method (BSM)

Figure 1- 5: Slant removal example [11]

some of the slant removal techniques will be discussed in details bellow.

Segmentation

Splits a large image into smaller regions of interest, generating candidate regions for recognition. Segmentation algorithms often attempt to split a document into pieces: pages into lines, lines into words, and words into characters [12]

- Text Line Detection (Hough Transform, projections, smearing)

- Word Extraction (vertical projections, connected component analysis)

The major challenge in segmentation involves removal of background noise. The other problems involve gray levels, noisy background, multiple skew levels, variable font size and different styles of writing.[18]

Types of Segmentation

- **Explicit Segmentation:** In explicit approaches one tries to identify the smallest possible word segments (primitive segments) that may be smaller than letters, but surely cannot be segmented further. Later in the recognition process these primitive segments are assembled into letters based on input from the character recognizer. The advantage of the first strategy is that it is robust and quite straightforward, but is not very flexible. [11]

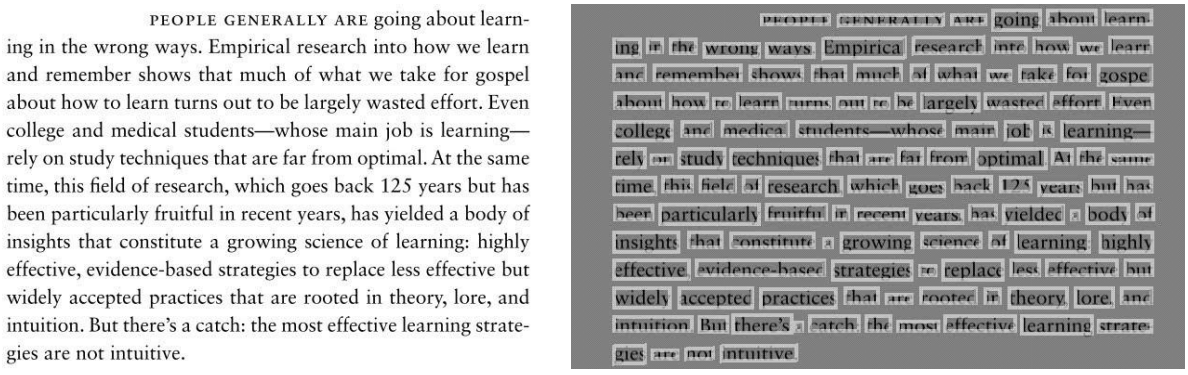


Figure 1- 6: Explicit Segmentation example

Implicit Segmentation: In implicit approaches the words are recognized entirely without segmenting them into letters. This is most effective and viable only when the set of possible words is small and known in advance, such as the recognition of bank checks and postal address. [11]

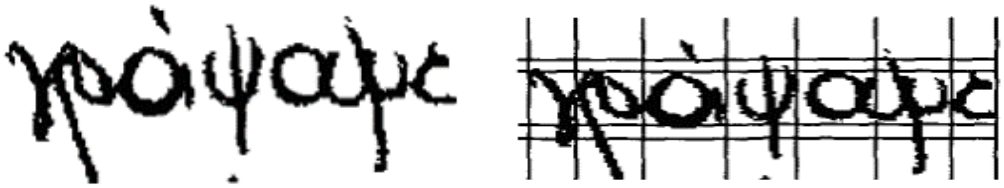


Figure 1- 7: Implicit Segmentation example [11]

Feature Extraction

For most practical applications, the original input variables are typically preprocessed to transform them into some new space of variables where, it is hoped, the pattern recognition problem will be easier to solve. For instance, in the digit recognition problem, the images of the digits are typically translated and scaled so that each digit is contained within a box of a fixed size. This greatly reduces the variability within each digit class, because the location and scale of all the digits are now the same, which makes it much easier for a subsequent pattern recognition algorithm to distinguish between the different classes. This pre-processing stage is sometimes also called feature extraction. Note that new test data must be pre-processed using the same steps as the training data. [3]

In feature extraction stage each character is represented as a feature vector, which becomes its identity. The major goal of feature extraction is to extract a set of features, which maximizes the recognition rate with the least number of elements. Due to the nature of handwriting with its high degree of variability and imprecision obtaining these features, is a difficult task. Feature extraction methods are based on 3 types of features: [11]

Statistical

Statistical features describe a pattern in terms of a set of characteristic measurements extracted from the pattern. [12]

Zoning: The character image is divided into NxM zones. From each zone features are extracted to form the feature vector. The goal of zoning is to obtain the local characteristics instead of global characteristics [11]

By finding density features: The number of foreground pixels, or the normalized number of foreground pixels, in each cell is considered a feature. Darker squares indicate higher density of zone pixels.[11]

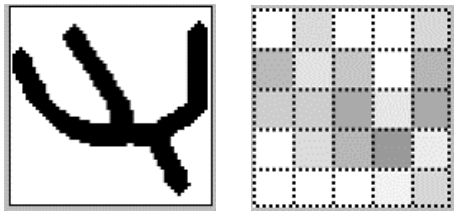


Figure 1- 8: Density Features [11]

By finding directional features (Chain Codes): One of the simplest transformations is representing the skeleton or contour of a pattern as a chain of direction codes. They are used to represent a boundary by a connected sequence of straight-line segments. As an edge is traced from its beginning point to the end point, the direction that must be taken to move from one pixel to the next is given by the number in the 8-chain code [11, 12]. The 8-chain code is as follows for the corresponding directions:

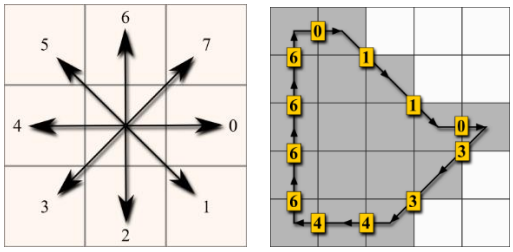


Figure 1- 9 Directional Features [11]

Projection Histograms: Projection histograms count the number of pixels in each column and row of a character image. Projection histograms can separate characters such as “m” and “n”.

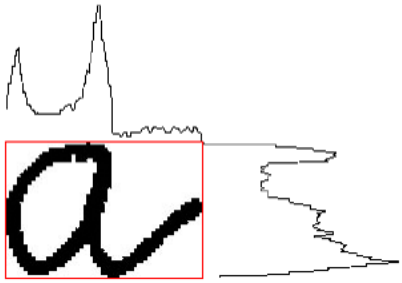


Figure 1- 10: Vertical and Horizontal Projection Histograms [11]

Profiles: The profile counts the number of pixels (distance) between the bounding box of the character image and the edge of the character. The profiles describe well the external shapes of characters and allow to distinguish between a great number of letters, such as “p” and “q”. [11]

Figure 1- 11: Profiles [11]

Structural

- Characters can be represented by structural features with high tolerance to distortions and style variations. This type of representation may also encode some knowledge about the structure of the object or may provide some knowledge as to what sort of components make up that object. Structural features are based on topological and geometrical properties of the character, such as aspect ratio, cross points, loops, branch points, strokes and their directions, inflection between two points, horizontal curves at top or bottom, etc.[11]
- Global transformations and moments The Fourier Transform (FT) of the contour of the image is calculated. Since the first n coefficients of the FT can be used in order to reconstruct the contour, then these n coefficients are considered to be a n-dimesional feature vector that represents the character. Central, Zenrike moments that make the process of recognizing an object scale, translation, and rotation invariant. The original image can be completely reconstructed from the moment coefficients. [11]

Classification: For classification any of k-Nearest Neighbor (k-NN), Bayes Classifier, Neural Networks (NN), Hidden Markov Models (HMM), Support Vector Machines (SVM), etc. can be used. There is no such thing as the "best classifier". The use of classifier depends on many factors, such as available training set, number of free parameters etc. But Neural networks proven themselves to be the most reliable and performant among them all. [11]

Post-processing

OCR accuracy can be increased if the output is constrained by a lexicon – a list of words that are allowed to occur in a document. This might be, for example, all the words in the English language, or a more technical lexicon for a specific field. This technique can be problematic if the document contains words not in the lexicon, like proper nouns. [11]

Slant removal in details

Slanted text has been demonstrated to be a salient feature of handwriting. [14] In particular, the slanted characters slope either from right to left or vice versa. Moreover, different deviations may appear not only within a text but also within a single word. [13]

Slanted characters constitute a common feature of any natural language with a Latin-style alphabet (e.g., English, Modern Greek, etc.). For example, the percentage of slanted writing in IAM-DB [1] database of English reaches 77% while the corresponding percentage in GRUHD, a database of Modern Greek (1000 writers are included), approaches 59%. Its estimation is a necessary preprocessing task in many document image processing systems in order to improve the required training. [13, 14]

Consequently, a robust optical character recognition (OCR) system has to be able to cope with slanted characters. In handwriting, slant removal is a necessary component of the text normalization procedure in systems that perform recognition (e.g., optical character recognition (OCR) [1] or word-spotting), After ideal slant removal processing, the text should appear with the vertical stokes parallel to the perpendicular axis of the page. Due to its importance, many researchers have already developed techniques for slant removal. [13, 14]

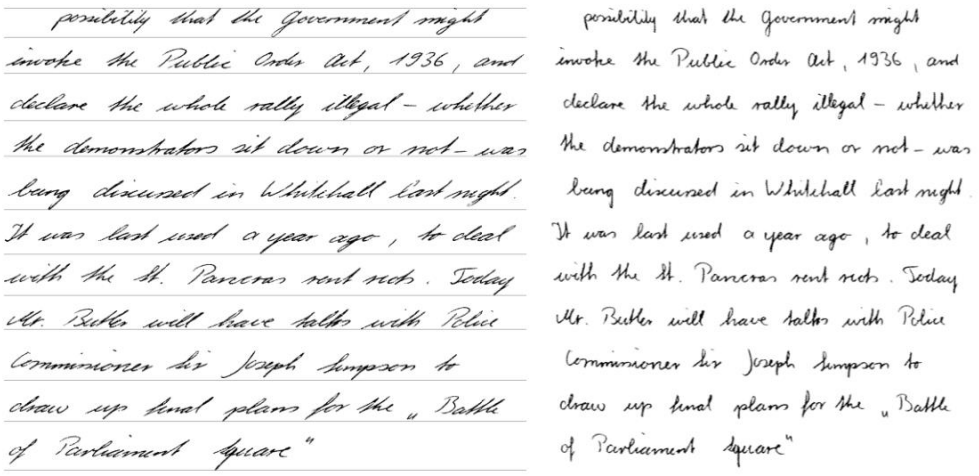


Figure 1- 12: Slant removal [14]

To a further extent, the two most important problems that may arise from the existence of slanted characters, in regard to an OCR system, are the following:

- If character segmentation procedure, that produce vertical segment boundaries between characters (e.g., based on histograms) is required, this could result in defectively segmented characters as well as in noisy segments.
- Both the computational cost of the training procedure and accuracy of the recognition stage would be affected in a negative way. Indeed, concerning a single character, the amount of the training data required for covering as many slant angles as possible is substantial. Moreover, the classification of a given character into the correct class is much harder since the set of possible classes is now bigger. [13]

Therefore, the majority of recent OCR systems contain a preprocessing stage dealing with slant correction. This stage is usually located before the segmentation module, if it exists, or just before the recognition stage otherwise.

Slant estimation and removal techniques

The available techniques may be divided into three categories:

- Techniques that estimate the slant by averaging the angles of the near-vertical strokes.
- Techniques that analyze projection histograms and detect the slant based on a pre-defined criterion (e.g., a parameter maximization or minimization).

- Techniques that are based on the statistics of chain-coded contours. [14]

Considering the application these techniques can handle, they can be further classified into:

1. Uniform slant estimation and removal techniques: deal with uniform slant all over the text.
2. Non-uniform slant correction techniques: they handle the characters apart and deal with the existence of several slants, simultaneously. [14]

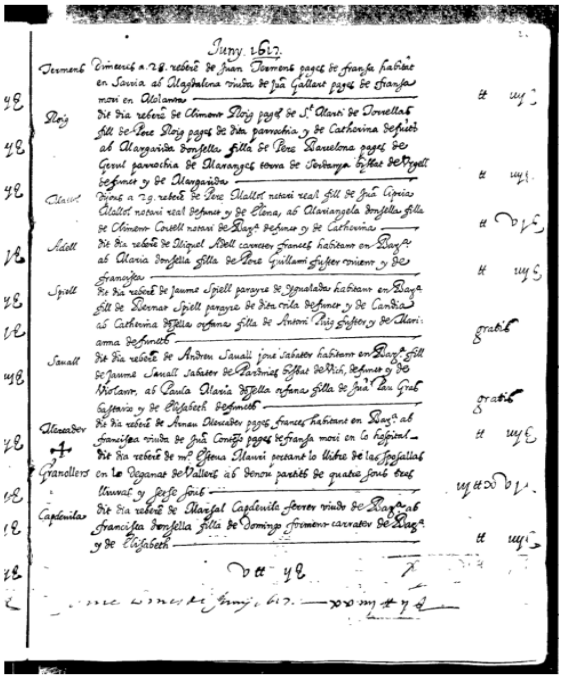


Figure 1- 13: A document from the Barcelona historical, handwritten marriages database (BH2M) [14]

In Figure 1- 12, an example of the slant removal algorithm described in [9], is presented. The image is from the IAM Handwriting Database (IAM-DB), and the application of the algorithm requires image segmentation into text lines Figure 1- 12, horizontal stripes). For this example, text line segmentation could succeed since text lines are spaced enough.[14] It is not the case for the document image shown in **Error! Reference source not found.** (17th century) w hich includes touching ascenders and descenders and noise in the inter-line space.

This next method for slant estimation is based on a combination of the projection profile technique and the Wigner-Ville distribution. Moreover, it uses a simple and fast shearing

transformation technique. And it's character independent and can easily be adapted in order to satisfy the requirements of any OCR system. [13]

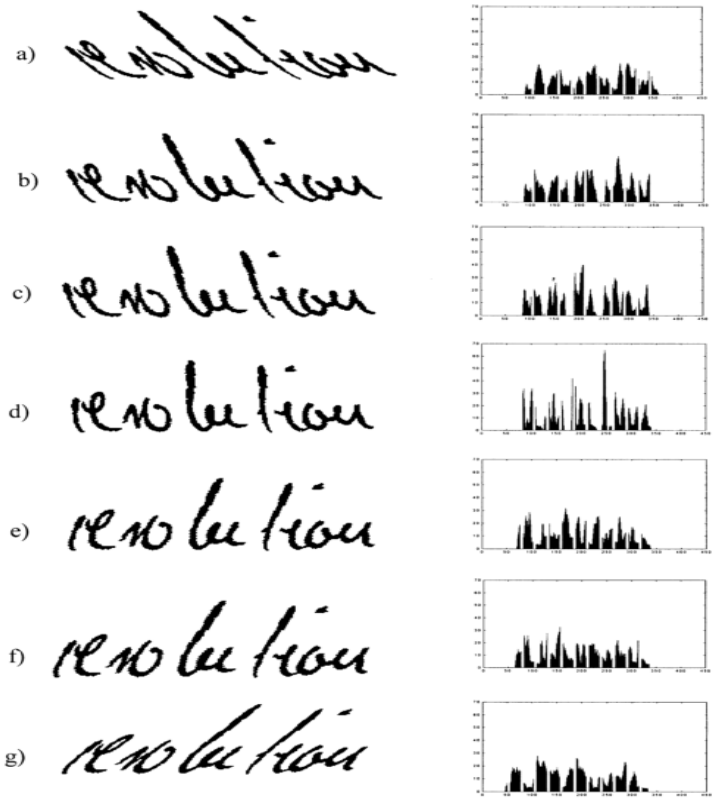


Figure 1- 14: Vertical projection profiles of a word with various slants [13]

The slant removal algorithm

The vertical projection profile of a non-slanted word presents the most dips between the characters, even if they are connected, and the highest peaks at the main body of the characters. The latter is much more evident when ascenders and descenders are included. In the case of slanted words, the otherwise vertical strokes of the characters cover now the intra-character gaps. Hence, the dips of the histogram are less deep, while the peaks are smoother. The alternations of the vertical projection profile of certain words are more intense when it is non-slanted than slanted.[13]

In Figure 1- 14, the vertical projection profiles for different slants of the same word are shown.

The proposed algorithm for slant estimation and correction for a given word, consists of six steps:

1. The word image is artificially slanted to both left and right for different slant angles. The maximum slant angle is 45°, the slant angle step depends on the height of the word image.
2. For each of the extracted word images, the vertical histogram is calculated.
3. The WVD is calculated for all the above histograms.
4. The curves of maximum intensity of the WVDs are extracted.
5. The curve of maximum intensity with the greatest peak, corresponding to the histogram with the most intense alternations, is selected.
6. The corresponding word image is selected as the most non-slanted word.

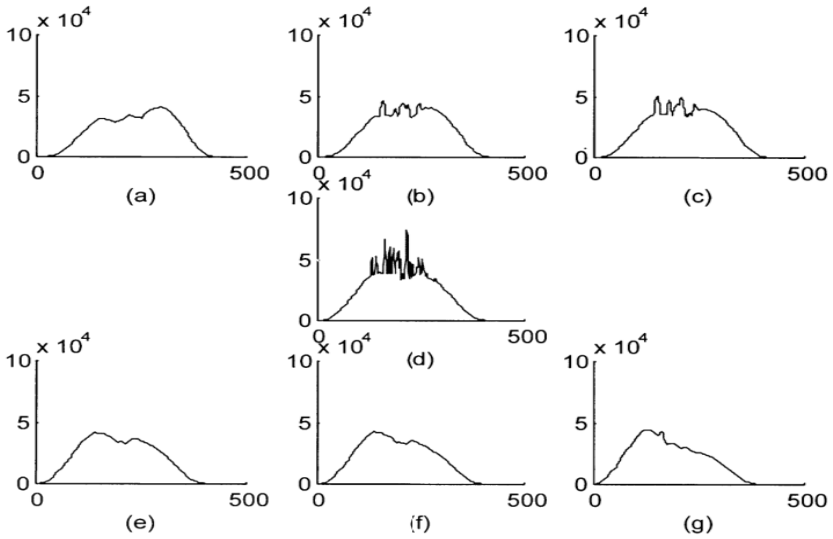


Figure 1- 15: Curves of maximum intensity that corresponds to the histograms of Figure 14 [13]

Slanting a word image

In order to slant a word image to right or left we follow the procedure below. The word image is segmented in equally wide horizontal zones. The lowest zone is considered to be the base. The zone above the base is shifted one pixel to the right or to the left. The next zone (if it exists) is shifted two pixels to the same direction, etc. [13]

Thus, each pixel p (x, y) of the image is shifted to the point (x', y'): $x' = x + i,\ -Z < i < Z, 0 < Z \leq h$, and $y' = y$

where successive zones are identified by successive ordinals Z and h the height of the image. The corresponding slant θ will be: $tan(\theta) = (Z - 1)/h$

The more the zones, the greater the slant angle. The maximum slant angle corresponds to one-pixel-wide zones (i.e., when the number of zones is equal to the height h of the word image in pixels). In this case, the higher zone is shifted by (h-1) pixels and the corresponding slant angle θ is the maximum one and it can be calculated as: $tan(\theta) = (h-1)/h \approx 1 => \theta = 45°$. [13]

Figure 1- 16 illustrate the above procedure, the gradual slanting of a vertical stroke

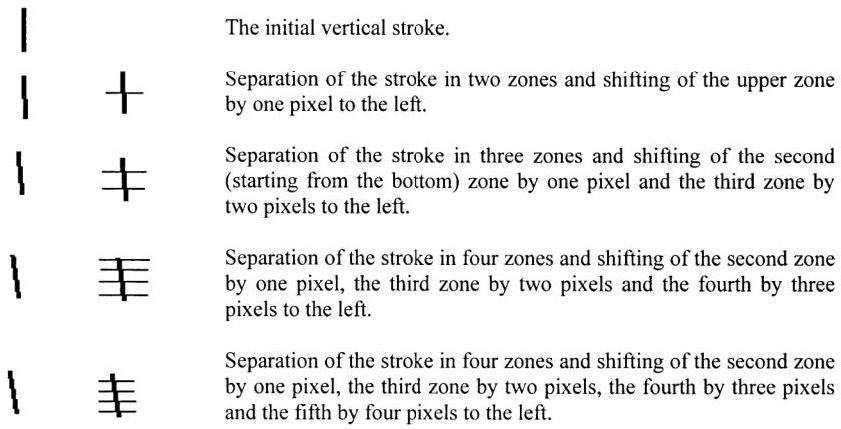


Figure 1- 16: Gradual Slanting of A Vertical Stroke [13]

Notice that the words produced by that technique are very natural (as can be seen in Figure 1- 17). A maximum slant angle of ±45° covers the vast majority of handwritings. Shifting each zone by more than one pixel would increase the maximum slant angle. However, it could cause undesirable disconnections between adjacent zones producing an unnatural outcome. [13]

The slant removal Algorithm and its results

The presented slant removal algorithm in [13], has been tested on a collection of word images taken from the IAM-DB and the GRUHD databases comprising English and Modern Greek unconstrained handwriting, respectively. In more detail, more than 1500 word images were used taken from approximately 500 different writers, selected randomly. The word segmentation was performed automatically based on an OCR preprocessing system.

Fig. 8. An example of IAM-DB document image (a) as inserted into the system and (b) after the slant removing.

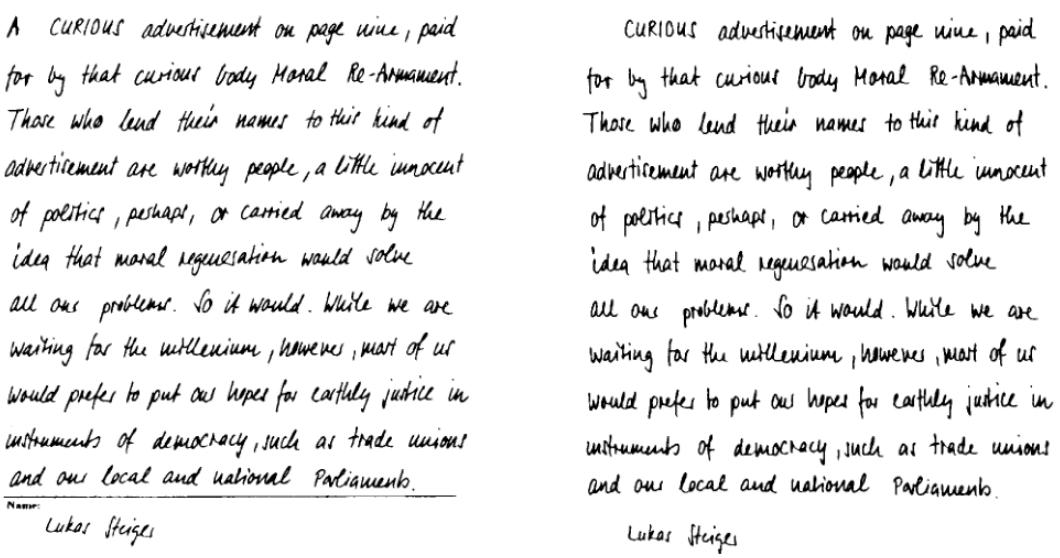


Figure 1- 17: An example of IAM-DB document image (a) as inserted into the system and (b) after the slant removing. [13]

The evaluation of a slant removal method is difficult since the selection of the most appropriate result very often falls under subjective judgments (especially in case of dealing with variant-slanted words). Moreover, a slant removal algorithm can indirectly be evaluated by taking into account the improvement it provides to an existing OCR system [13, 15]

***Note**: It is worth noting that the already non-slanted words are not affected in a negative way by applying this algorithm whether the characters are connected or not.*

The presented algorithm can incorporate into most character recognition system, The system, often includes other modules, namely skew angle estimation and correction, printed-handwritten text discrimination, line segmentation, word segmentation, character segmentation and recognition.

***Note**: The proposed technique could be applied to words or directly to the text line images resulted from line segmentation. However, the uneven valleys of the vertical histogram, i.e. wide valleys between words narrow valley between characters, could give confusing results in some cases. This is the reason that we preferred to use part of the text lines instead.*

skew correction in details

Many times, in OCR we have to deal with documents that suffers from some degrees of skew. Skew angle detection is an important component of any Optical Character Recognition

(OCR) and document analysis system. [16] The majority of both segmentation and character recognition algorithms are sensitive to the orientation of the word. Furthermore, the skewed words are very often found in handwritten text. Even in the case of correctly oriented pages, the handwritten words could present smaller or larger skews [17]. Skew angle is the angle that the text lines of the recorded digital document make with the horizontal direction. [16].

The three classes of skew estimation techniques that are based on:

- Projection profile
- Component nearest neighbor clustering
- Hough transform

The horizontal (vertical) projection profile (Hou, 1983) is a histogram of the number of black pixels along horizontal (vertical) scan lines. For a script with horizontal text lines the horizontal projection profile will have peaks at text line positions and troughs at positions in-between successive text lines. To determine the skew of a document, the projection profile is computed at a number of angles and for each angle a measure of difference of peak and trough height is made. The maximum difference corresponds to the best alignment with the text line direction which, in turn, determines the skew angle. [16] Some papers describe an approach, to determine the profile that represent the maximum difference, where the WVDs are, then, calculated and the maximum intensity of each histogram is marked. Finally, the angle, whose projection presents the maximum intensity, is selected as the most appropriate and the word is corrected. [17]

Baird (1987) proposed a modification for quick convergence of this iterative approach, Akiyama and Hagita (1990) described an approach where the document is partitioned into vertical strips. The horizontal projection profiles are calculated for each strip and from the correlation of the profiles of the neighboring strips the skew angle is determined. This method is fast but less accurate. Pavlidis and Zhau (1992) proposed a method based on vertical projection profile of horizontal strips which works well if the skew angle is small.[16]

Hashizume et al. (1986) proposed nearest neighbor clustering to skew detection. He found all the connected components in the document and for each component computed the

direction of its nearest neighbor. A histogram of the direction angle are computed, the peak of which indicates the document skew angle. O'Gorman (1993) generalized the approach in so called 'docstrum' analysis. [16]

Hough transform has been used by Srihari and Govindaraju (1989) for skew detection. The basic method consists of mapping points in Cartesian space (x,y) to sinusoidal curves in (ρ, θ) space via the transformation

$$\rho = x \cos\theta + y \sin\theta \quad (1)$$

Each time a sinusoidal curve intersects another at a particular value of p and θ, the likelihood increases that a line corresponding to that (p,θ) coordinate value is present in the original image. An accumulator array is used to count the number of intersection at various p and θ values. The skew is then determined by the θ values corresponding to the highest number of counts in the accumulator array.[16]

Conclusion

In this chapter, we introduced Handwritten Recognition and OCR, and the need of such systems, and then we presented the necessary components and steps of the system. Then we dived into some of the algorithms appropriate for the preprocessing stage of OCR systems, one that corrects the skew angle of words, while the other removes the slant from handwritten words, if it exists. Theses algorithms should be employed in an OCR system immediately after the word segmentation procedure and before the character segmentation stage, if it exists, or before the feature extraction stage if character segmentation is skipped. They could also be used in of a character segmentation system as a preprocessing stage. In both cases, the use of theses algorithms improves the accuracy of the OCR system.

Chapter Two: Feed-Forward Neural Networks

Introduction: What is a neural network?

The term ‘neural network’ has its origins in attempts to find mathematical representations of information processing in biological systems (McCulloch and Pitts, 1943; Widrow and Hoff, 1960; Rosenblatt, 1962; Rumelhart et al., 1986). Artificial neural networks (or simply neural networks) are, as their name indicates, computational networks which attempt to simulate, in a gross manner, the decision process in networks of nerve cell (neurons) of the biological (human or animal) central nervous system. [3,8]

Fundamentals of Biological Neural Networks

The biological neural network consists of nerve cells (neurons) as in Figure 2- 1, which are interconnected as in Figure 2- 2. The cell body of the neuron, which includes the neuron’s nucleus is where most of the neural “computation” takes place. [8]

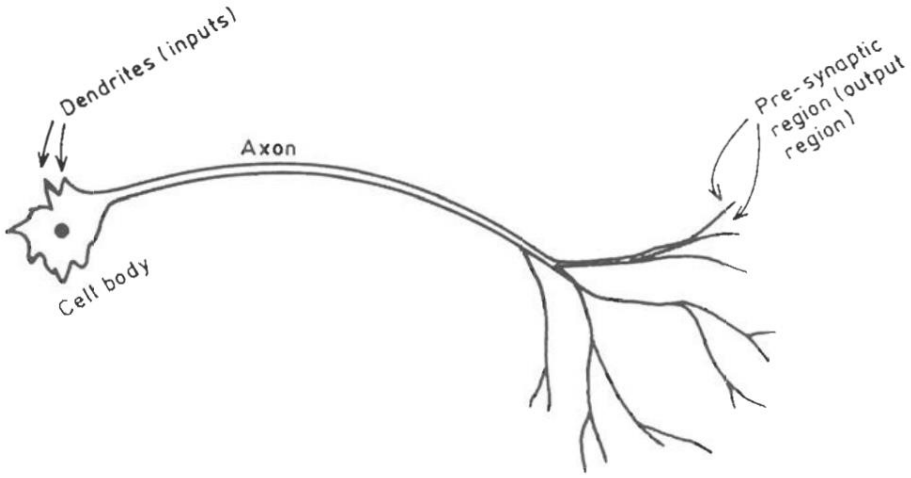


Figure 2- 1: A biological neural cell (neuron). [8]

Neural activity passes from one neuron to another in terms of electrical triggers which travel from one cell to the other down the neuron’s axon, by means of an electro-chemical process of voltage-gated ion exchange along the axon and of diffusion of neurotransmitter molecules through the membrane over the synaptic gap (Figure 2- 2). The axon can be viewed as a connection wire. However, the mechanism of signal flow is not via electrical conduction but via charge exchange that is transported by diffusion of ions. This transportation process moves along the neuron’s cell, down the axon and then through synaptic junctions at the end of the axon via a very narrow synaptic space to the dendrites and/or soma of the next neuron at an average rate of 3 m/sec.[8]

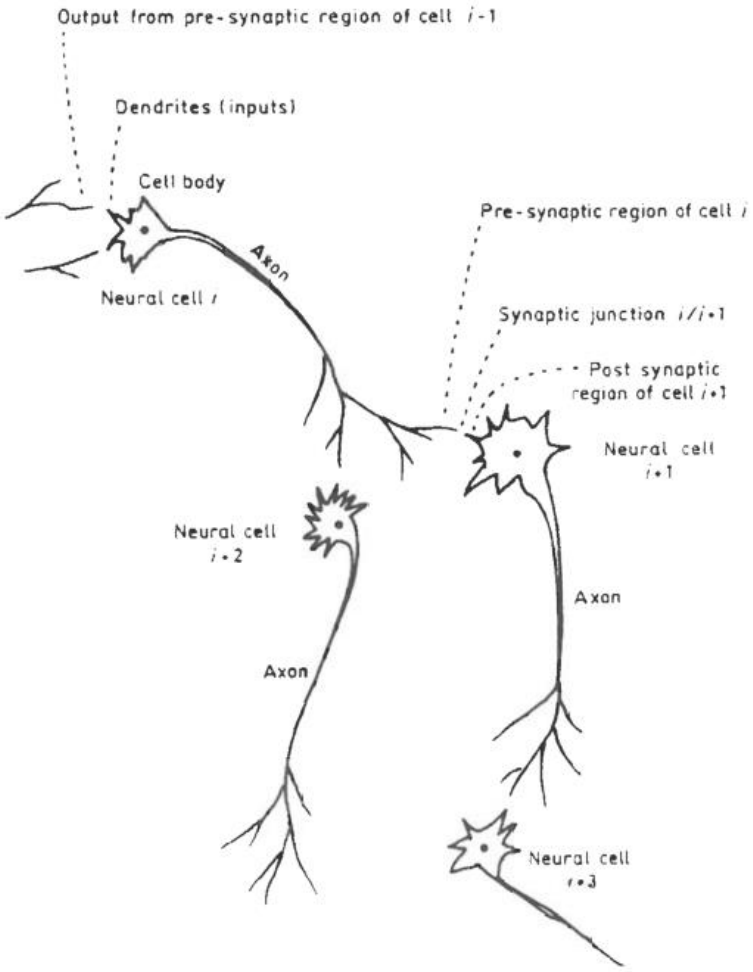


Figure 2- 2: Interconnection of biological neural nets [8]

It is important to note that not all interconnections are equally weighted. Some have a higher priority (a higher weight) than others. Also, some are excitatory and some are inhibitory (serving to block transmission of a message). These differences are affected by differences in chemistry and by the existence of chemical transmitters and modulating substances inside and near the neurons, the axons and in the synaptic junction. This nature of interconnection between neurons and weighting of messages is also fundamental to artificial neural networks (ANNs). [8]

A simple analog of the neural element of (Figure 2- 3 a) is as in (Figure 2- 3 b). In that analog, which is the common building block (neuron) of every artificial neural network, we observe the differences in weighting of messages at the various interconnections (synapses) as mentioned above. Analogs of cell body, dendrite, axon and synaptic junction of the biological neuron of (Figure 2- 3 a) are indicated in the appropriate parts of (Figure 2- 3 b). The biological network of (Figure 2- 3 c) thus becomes the network of (Figure 2- 3 d).

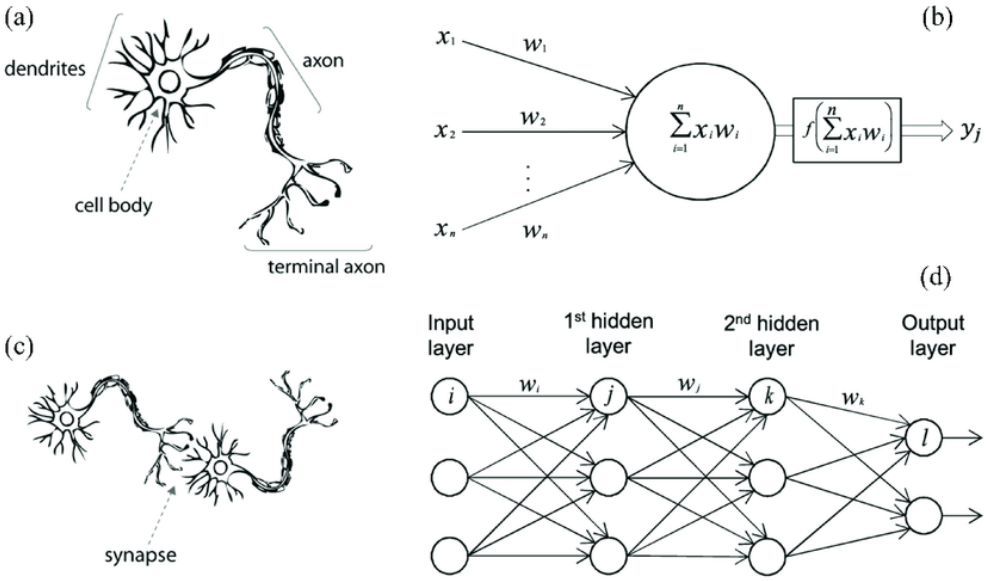


Figure 2- 3: Schematic analog of a biological neural cell and neural network. [24]

Why spending some time talking about the biological NNs? when we are just grossly borrowing their working machinery. Arguably, is because the Artificial NNs are inspired by their biological counterpart so it's important to have a generic idea of what's going on there, but what's more important for us is an idea that is usually phrased as "Neurons that fire together wire together", which is an important aspect for Artificial NNs as well, as soon it will become apparent.

The Way NNs Operate

The way the network operates, activations in one layer determine the activations of the next layer, and of course the heart of the network as an information processing mechanism comes down to exactly how those activations from one layer bring about activations in the next layer. It's meant to be loosely analogous to how in biological networks of neurons some groups of neurons firing cause certain others to fire.

For instance let's assume that we have a network that has already been trained to recognize digits, it means if you feed in an image lighting up all 784 neurons of the input layer according to the brightness of each pixel in the image that pattern of activations causes some very specific pattern in the next layer to activate, which causes some pattern in the one after it, which finally gives some pattern in the output layer and the brightest neuron of that output layer is the network's choice sort of speak for what digit this image represents.

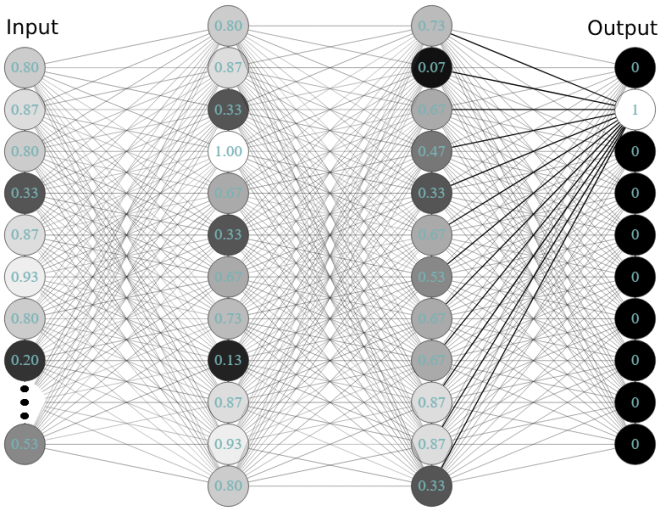


Figure 3- 1 feed-forward activation mechanism in NNs

Why it's reasonable to expect a layered structure to behave intelligently?

Now, let's argue about why it's even reasonable to expect a layered structure like this to behave intelligently. What are we expecting here? what is the best hope for what those middle layers might be doing? Well, when your eyes recognize digits, we piece together various components a '9' has a loop up top and a line on the right, an '8' also has a loop up top but it's paired with another loop down low, a '4' basically breaks down into three specific lines and things like that.

In a perfect world we might hope that each neuron in the second-to-last layer corresponds with one of these sub components, that anytime you feed in an image with say a loop up top like a nine or an eight there's some specific neuron whose activation is going to be close to one. The hope would be that any generally loopy pattern towards the top, sets off a specific neuron in the second-to-last layer, that way going from the third layer to the last one, just requires learning which combination of sub components corresponds to which digits.

For neuron in the first layer to be able to detect small edges that we hope are used by neurons in the layers after to compose more complex patterns, what we'll do is assign a weight* to each one of the connections between our neuron and the neurons from the first layer, these weights are just numbers, then take all those activations from the first layer and compute their weighted sum according to these weights.

Recognizing a loop can also break down into subproblems, one reasonable way to do this would be to first recognize the various little edges that make it up, similarly a long line like the kind you might see in the digits 1 or 4 or 7, well that's really just a long edge or maybe you think of it as a certain pattern of several smaller edges, so maybe our hope is that each neuron in the second layer of the network corresponds with the various relevant little edges, maybe when an image like this one comes in it lights up all of the neurons associated with around eight to ten

specific little edges, which in turn lights up the neurons associated with the upper loop and a long vertical line and those light up the neuron associated with a nine. Whether or not this is what our final network actually does is another question, one that I'll come back to once we see how to train the network, but this is a hope that we might have, a sort of goal with the layered structure like this, moreover you can imagine how being able to detect edges and patterns like this would be really useful for other image recognition tasks and even beyond image recognition there are all sorts of intelligent things you might want to do that break down into layers of abstraction, parsing speech for example involves taking raw audio and picking out distinct sounds which combine to make certain syllables which combine to form words which combine to make up phrases and more abstract thoughts … etc.

Now each neuron in the second layer will be connected to every neuron in the first (input) layer, if you organize these weights as an image (2D-matrix) you will be able to see what each neuron is detecting. For instance, if we are designing the network manually to detect an arch in a specific region in the input image, we might want to set the weights covering the region that correspond to the pattern we want to detect to one (positive weights), and set all other weights to 0. If we want to be able to detect edge, we can set the outer region. So that the output will be maximized when the input contains exactly the pattern (white pixels) corresponding to the region where we did set the weights to ones, and all the other pixels are zeros.

Perceptrons

Perceptrons are a type of artificial neuron, were developed in the 1950s and 1960s by the scientist Frank Rosenblatt, inspired by earlier work by Warren McCulloch and Walter Pitts. Today, it's more common to use other models of artificial neurons - in this book, and in much modern work on neural networks, the main neuron model used is one called the sigmoid neuron [1] (another type of neurons that has been adopted in recent networks is called ReLU). We'll get to sigmoid neurons shortly. But to understand why sigmoid neurons are defined the way they are, it's worth taking the time to first understand perceptrons.

The Perceptron, which you can think of it as a node (function or unit) that takes in one or more binary inputs, x_1, x_2 ..., and produces a single binary output:

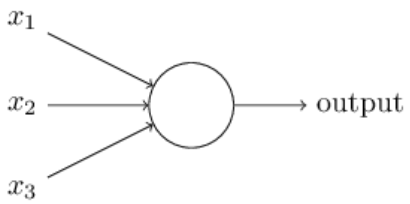


Figure 2- 4: Neuron structure [1]

The example shown above displays a perceptron with three inputs, x_1, x_2, x_3, and one output. In general, it could have more or fewer inputs. The Perceptron's cell's output differs from

the summation of its weighted inputs, by the activation operation of the cell's body, just as the output of the biological cell differs from the weighted sum of its input. The activation operation is in terms of an activation function f(z_i,), which is a nonlinear function yielding the ith cell's output a_i. Rosenblatt proposed a simple rule to compute the output. He introduced **weights**, $w_1, w_1,\ldots$, *real numbers expressing the importance of the respective inputs to the output.* The neuron's output, 0 or 1, is determined by whether the weighted sum $\sum_j \mathrm{w}_j \mathrm{x}_j$ is less than or greater than some **threshold** value. Just like the weights, the threshold is a real number which is a parameter of the neuron. To put it in more precise algebraic terms [8,1,5]:

$$\text{Output} = \begin{cases} 0 \ if \ \Sigma_j w_j x_j \leq threshould \\ 1 \ if \ \Sigma_j w_j x_j > threshould \end{cases} \quad (2)$$

That's the basic mathematical model. A way you can think about the perceptron is that it's a device that makes decisions by weighing up evidence [1]. To give you example. Which is not a super realistic, but it's easy to understand, and more importantly will illustrate the point, and we'll soon get to more realistic examples. Suppose you've heard that there's going to be a Music festival in your city this weekend. You like music, and you are trying to decide whether or not to go to the festival. You might make your decision by weighing up three factors:

- Is the weather good?
- Is your friend willing to accompany you?
- Is the festival near public transit? (You don't own a car).

these three factors can be represented by corresponding binary variables $\mathrm{x}_1, \mathrm{x}_2$ and x_3. For instance, we'd have $\mathrm{x}_1 = 1$ if the weather is good, and $\mathrm{x}_1 = 0$ if the weather is bad. Similarly, $\mathrm{x}_2 = 1$ if your friend wants to go, and $\mathrm{x}_2 = 0$ if not. And similarly, again for x3 and public transit.

We can represent these three factors by corresponding binary variables $\mathrm{x}_1, \mathrm{x}_2$ and x_3. For instance, we'd have $\mathrm{x}_1 = 1$ if the weather is good, and $\mathrm{x}_1 = 0$ if the weather is bad. Similarly, $\mathrm{x}_2 = 1$ if your friend wants to go, and $\mathrm{x}_2 = 1$ if not. And similarly, again for x_3 and public transit.

Now, let's assume that you love music so much that you are ready to go, regardless of whether your friend accompanies you or not. But you really hate bad weather and there is no way you can attend the festival if the weather is bad. You can use perceptrons to model that sort of decision-making. One way to accomplish this is to select a weight $w_1 = 6$ for the weather, and $w_2 = 2$ and $w_3 = 2$ for the other conditions. The larger value of w_1 indicates that the weather matters a lot to you, much more than whether your friend joins you, or the nearness of public transit. Finally, suppose you choose a threshold of 5 for the perceptron. With these choices, the perceptron implements the desired decision-making model, outputting 1 whenever the weather is

good, and 0 whenever the weather is bad. It makes no difference to the output whether your friend wants to go, or whether public transit is nearby.

By changing the weights and the threshold, we can obtain different models for decision-making. For instance, suppose a threshold of 3 is chosen instead. Then the perceptron would decide that you should go to the festival whenever the weather was good or when both the festival was near public transit and your friend was willing to join you. Put another way, it would be a different decision-making model. Lowering the threshold means that you are more likely to attend the festival.

The condition $\Sigma_j w_j x_j > threshould$ is cumbersome, and we can make two notational changes to simplify it. The first change is to write $\Sigma_j w_j x_j$ as a dot product, $w \cdot x \equiv \Sigma_j w_j x_j$, where w and x are vectors, whose components are the weights and inputs, respectively. The second change is to move the threshold to the other side of the inequality, and to replace it by what's known as the perceptron's bias, b≡-threshold. Using the bias instead of the threshold, the perceptron rule can be rewritten:

$$\text{Output} = \begin{cases} 0 \; if \;\; w.x + b \leq 0 \\ 1 \; if \;\; w.x + b > 0 \end{cases} \qquad (3)$$

You can think of the bias as a measure of how easy it is to get the perceptron to output a 1. For a perceptron with a really big bias, it's extremely easy for the perceptron to output a 1. But if the bias is very negative, then it's difficult for the perceptron to output a 1.

The perceptron algorithm:

The perceptron of Rosenblatt (1962) is an example of a linear discriminant model, which occupies an important place in the history of pattern recognition algorithms [3]. As a very general overview, the step function meant to mimic a neuron in the brain, either “firing” or not - like an on-off switch. In programming, an on-off switch as a function would be called a step function because it looks like a step if we graph it [4].

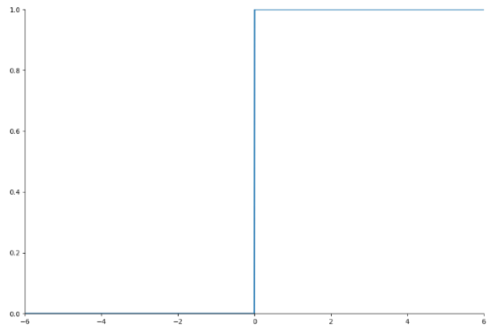


Figure 2- 5: Graph of the step function

Note: The activation function f is also known as a squashing function. It keeps the cell's output between certain limits as is the case in the biological neuron [8]. Different functions $f(z_i)$ are in use, all of which have the above limiting property. The most common activation function is the sigmoid function which is a continuously differentiable function [8]

Obviously, the perceptron isn't a complete model of human decision-making! But what the example illustrates is how a perceptron can weigh up different kinds of evidence in order to make decisions. And it should seem plausible that a complex network of perceptrons could make quite subtle decisions:[1]

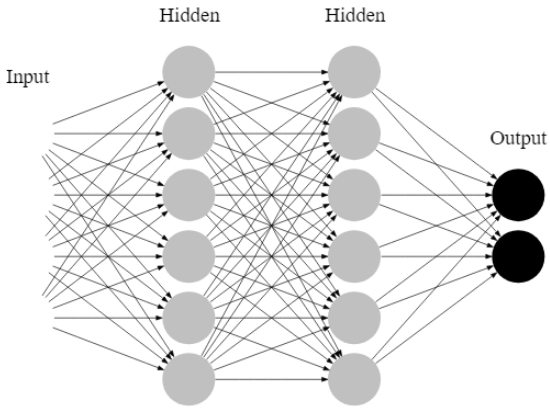


Figure 2- 6: Simple Network Demonstrating the Decision-Making Machinery

In this network, the first column of perceptrons - what we'll call the first layer of perceptrons - is making very simple decisions, by weighing the input evidence. The perceptrons in the second layer are making a decision by weighing up the results from the first layer of decision-making. In this way a perceptron in the second layer can make a decision at a more complex and more abstract level than perceptrons in the first layer. And even more complex decisions can be made by the perceptron in the third layer. In this way, a many-layer network of perceptrons can engage in sophisticated decision making. [1]

It turns out that we can devise learning algorithms which can automatically tune the weights and biases of a network of artificial neurons. This tuning happens in response to external stimuli, without direct intervention by a programmer [1]. Instead of explicitly specifying the exact values for the weights and biases, our neural networks can simply learn to solve problems.

Sigmoid neurons

The question that poses itself now, is how can we come up with such algorithms for a neural network? and how this algorithm will work exactly and based on what? But before we talk about the exact algorithm and the algebraic form of the functions we will use for this algorithm, we need to think at more abstract and high level how this algorithm allows the network to learn,

in another word how this algorithm change the wight and biases of the network in order to optimize its behavior.

To tackle this problem, suppose we have a network of perceptrons that we'd like to use to learn to solve some problem. For example, the inputs to the network might be the raw pixel data from a scanned, handwritten image of a digit. And we'd like the network to learn weights and biases so that the output from the network correctly classifies the digit. To see how learning might work, suppose we make a small change in some weight (or bias) in the network. What we'd like is for this small change in weight to cause only a small corresponding change in the output from the network. As we'll see in a moment, this property will make learning possible. Schematically, here's what we want:

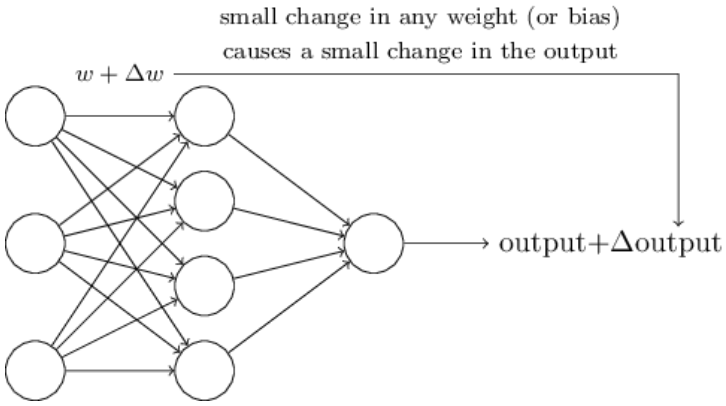


Figure 2- 7: Weights Output Relationship [1]

If it were true that a small change in a weight (or bias) causes only a small change in output, then we could use this fact to modify the weights and biases to get our network to behave more in the manner we want. Because what we need eventually, is network in which small change in the weight and biases causes only small change in the output. So, we can tune those weights and biases till we land on good settings for our network to correctly classify numbers.

But the problem is that this isn't what happens when our network contains perceptrons. It turns out that, a small change in the weights or bias of any single perceptron in the network can sometimes cause the output of that perceptron to completely flip, say from 0 to 1 (because the perceptrons have a binary behavior, either 0 or 1). That flip may then cause the behavior of the rest of the network to completely change in some very complicated way. That makes it difficult to see how to gradually modify the weights and biases so that the network gets closer to the desired behavior.

Introducing Sigmoid neurons

To overcome this issue, we need to use a different type of artificial neuron, known as **sigmoid neuron**. Sigmoid neurons are similar to perceptrons, in way that they take multiple

inputs and weights and biases to produce an output, but modified so that small changes in their weights and bias cause only a small change in their output. That's the crucial fact which will allow a network of sigmoid neurons to learn.

Unlike perceptron neuron, in which the inputs and outputs are binary either 0 or 1, the inputs for the sigmoid neuron can be any real value, and its output is a value between 0 and 1. So, for instance, 0.356… is a valid input for a sigmoid neuron. The exact value of the output of the sigmoid neuron is calculated by summing up the product of the weight and input and adding up the bias and then feeding the result to the sigmoid function $\sigma(\mathrm{w} \cdot \mathrm{x} + \mathrm{b})$, and that is surprisingly enough where the sigmoid neuron gets its name.

Note: σ is sometimes referred to as the **Logistic Function**, and this new class of neurons called **Logistic Neurons**. It's useful to remember this terminology, since these terms are used by many people working with neural nets. However, we'll stick with the sigmoid terminology.

$$\sigma(\mathrm{z}) \equiv \frac{1}{1 + e^{-\mathrm{z}}} \tag{4}$$

To put it all a little more explicitly, the output of a sigmoid neuron with inputs $x_1, x_2, \ldots$, weights $w_1, w_2, \ldots$, and bias b is

$$\sigma(z) \equiv \frac{1}{1 + exp(-\Sigma_j w_j x_j + b)} \tag{5}$$

The algebraic form of the sigmoid function is very different to perceptrons. and the difference in the value of their inputs and output (binary vs real-number), may expand your view of their difference, but honestly, they are not that different my friend. The continuity in the input and the output of the sigmoid neuron, is just a feature that we sought in the sigmoid at the first place.

To understand the similarity to the perceptron model, suppose $\mathrm{z} \equiv \mathrm{w} \cdot \mathrm{x} + \mathrm{b}$ is a large positive number. Then $e^{-\mathrm{z}} \approx 0$ and so $\sigma(\mathrm{z}) \approx 1$. In other words, when $\mathrm{z} = \mathrm{w} \cdot \mathrm{x} + \mathrm{b}$ is large and positive, the output from the sigmoid neuron is approximately 1, just as it would have been for a perceptron. Suppose on the other hand that $\mathrm{z} = \mathrm{w} \cdot \mathrm{x} + \mathrm{b}$ is very negative. Then $\mathrm{e}^{-\mathrm{z}} \to \infty$, and $\sigma(\mathrm{z}) \approx 0$. So, when $\mathrm{z} = \mathrm{w} \cdot \mathrm{x} + \mathrm{b}$ is very negative, the behavior of a sigmoid neuron also closely approximates a perceptron. It's only when $\mathrm{w} \cdot \mathrm{x} + \mathrm{b}$ is of modest size that there's much deviation from the perceptron model.

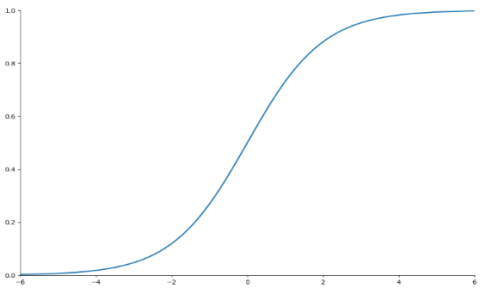


Figure 2- 8: Sigmoid function

In fact, the exact form of σ isn't so important - what really matters is the shape of the function. The smoothness of the σ function that is the crucial fact, not its detailed form. The smoothness of σ means that small changes Δw_j in the weights and Δb in the bias will produce a small change in the output from the neuron. In fact, calculus tells us that $\Delta output$ is well approximated by" [1]

$$\Delta output \approx \sum_j \frac{\partial output}{\partial w_j} \Delta w_j + \frac{\partial output}{\partial b} \Delta b \qquad (6)$$

$\partial output/\partial w_j$ and $\partial output/\partial b$ denote partial derivatives of the output with respect to w_j and b, respectively.

The function as whole is saying something very simple, which is the change in the output $\Delta output$, is s a linear function of the changes Δw_j and Δb in the weights and bias. This linearity makes it easy to choose small changes in the weights and biases to achieve any desired small change in the output. So, while sigmoid neurons have much of the same qualitative behavior as perceptrons, they make it much easier to figure out how changing the weights and biases will change the output.[1]

A concern that may arise when using the sigmoid function, is how we should interpret it output. Obviously, when using perceptron neurons, the output is either 0 or 1, so the answer is clear, either True or False, but with sigmoid neurons the case is different though. They can have as output any real number between 0 and 1, so values such as 0.173… and 0.689… are legitimate outputs. This can be useful, for example, if we want to use the output value to represent the average intensity of the pixels in an image input to a neural network. However sometimes it can be a nuisance. Suppose we want the output from the network to indicate either "the input image is a 9" or "the input image is not a 9". Obviously, it'd be easiest to do this if the output was a 0 or a 1, as in a perceptron. But in practice we can set up a convention to deal with this, for example, by deciding to interpret any output of at least 0.5 as indicating a "9", and any output less than 0.5 as indicating "not a 9". I'll always explicitly state when we're using such a convention, so it shouldn't cause any confusion.

Training data, MNIST

Now that we have a design for our neural network, how can it learn to recognize digits? The first thing we'll need is a data set to learn from - a so-called training data set. We'll use the MNIST data set, which contains tens of thousands of scanned images of handwritten digits, together with their correct classifications. MNIST's name comes from the fact that it is a modified subset of two data sets collected by NIST, the United States' National Institute of Standards and Technology. Here's a few images from MNIST [1]:

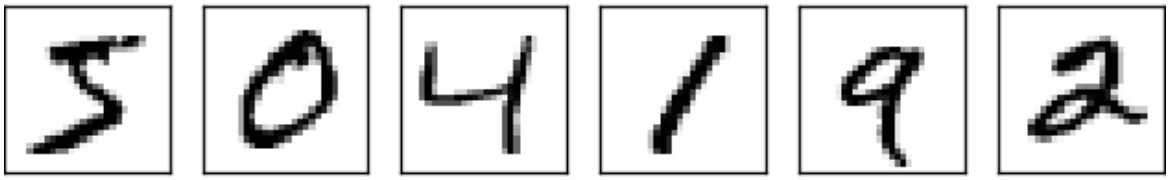


Figure 2- 9: Samples images from MNIST [1]

The MNIST data comes in two parts. The first part contains 60,000 images to be used as training data. The second part of the MNIST data set is 10,000 images to be used as test data. Both sets, are 28 by 28 greyscale images. We'll use the test data to evaluate how well our neural network has learned to recognize digits. To make this a good test of performance, the test data was taken from a different set of 250 people than the original training data (albeit still a group split between Census Bureau employees and high school students). This helps give us confidence that our system can recognize digits from people whose writing it didn't see during training.

Neural Network’s Architecture

In this section we will work our way up to build our neural network that is able to do good job at classifying handwritten digits. but before we do that, we need to set a common ground and agree on the terminologies that we will use to describe the different parts of the network, throughout this section and the rest of the work.

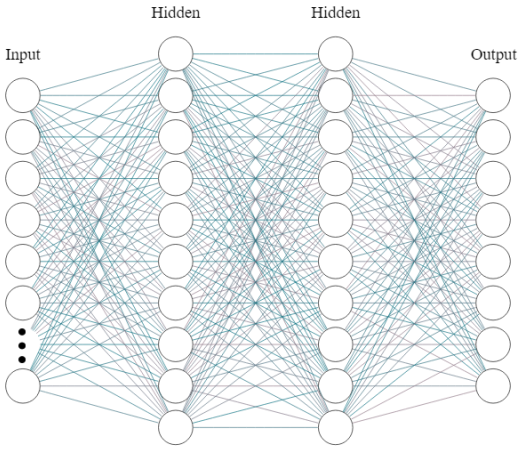


Figure 2- 10: Neural Network With 2 Hidden Layers

As mentioned earlier, the leftmost layer in this network is called the input layer, and the neurons within the layer are called input neurons. The rightmost or output layer contains the output neurons. The middle layer is called a hidden layer, since the neurons in this layer are neither inputs nor outputs.

Note: Multiple layer networks are sometimes called **Multi-Layer Perceptrons** or **MLPs**, despite being made up of sigmoid neurons, not perceptrons.

The challenge of designing the input and output layers of the network typically depends on the network's task and the type of input and output. and it's often a straightforward.

While the design of the input and output layers of a neural network is often straightforward, there can be quite an art to the design of the hidden layers. In particular, it's not possible to sum up the design process for the hidden layers with a few simple rules of thumb. Instead, neural networks researchers have developed many design heuristics for the hidden layers, which help people get the behavior they want out of their nets. [1]

The type of the neural network, we have been discussing so far, in which the output of one layer is used as an input for the next layer, are known as **Feed Forward Neural Networks**. Which mean there are no feed-back loops present in the network. There other artificial neural network architectures that employs the idea of feed-back loops, namely **Recurrent Neural Networks** (RNN) [1, 3]. The key feature of recurrent neural networks is that information loops back in the network. This gives recurrent neural networks a type of memory they can use to better understand sequential data. A popular type of recurrent neural network is the long short-term memory (LSTM) network, which allows for information to loop backward in the network.[2]

A simple network to classify handwritten digits

We can split the problem of recognizing handwritten digits into two sub-problems. First, we'd like a way of breaking an image containing many digits into a sequence of separate images, each containing a single digit. We humans solve this segmentation problem with ease, but it's challenging for a computer program to correctly break up the image. Once the image has been segmented, the program then needs to classify each individual digit. We'll focus on writing a program to solve the second problem, that is, classifying individual digits [1, 9].

For our network that we will use, needs at least three layers, the input, the output layer, and at least one hidden layer that will solve the problem of recognizing handwritten digits.

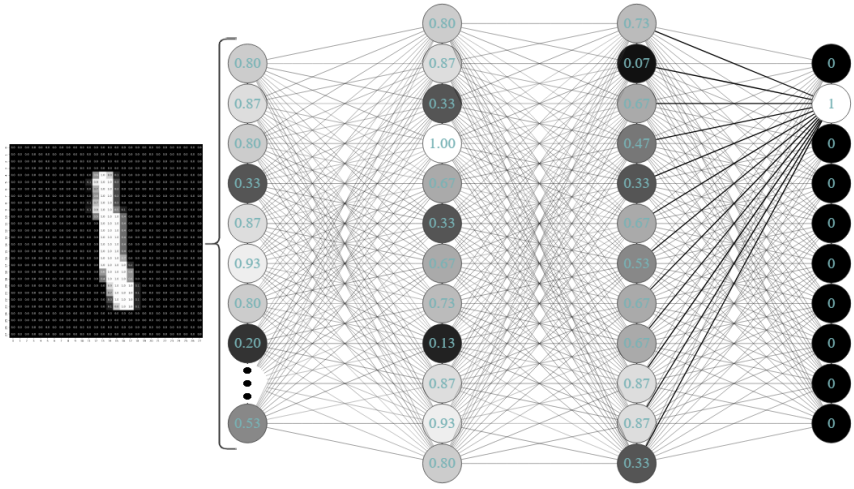


Figure 2- 11: Feed-forward Neural Network

The input layer of the network contains neurons encoding the values of the input pixels. As discussed in the next section, our training data for the network will consist of many 28 by 28-pixel images of scanned handwritten digits, and so the input layer contains 784=28×28 neurons. For simplicity I've omitted most of the 784 input neurons in the diagram above. The input pixels are greyscale, with a value of 0.0 representing white, a value of 1.0 representing black, and in between values representing gradually darkening shades of grey.[1]

The second layer of the network is a hidden layer. We denote the number of neurons in this hidden layer by n, and we'll experiment with different values for n. The example shown illustrates a small hidden layer, containing just n=15 neurons. [1]

The output layer of the network contains 10 neurons. If the first neuron fires, i.e., has an output ≈1, then that will indicate that the network thinks the digit is a 0. If the second neuron fires then that will indicate that the network thinks the digit is a 1. And so on. [1]

Conclusion

Neural Networks intelligence is an accumulation of multifactor, the main one being the nonlinearity of the activation function, it is basically the most important aspect of NNs, and what gives its power, and what allows the neural network to be flexible. The second most important aspect of is the layered structure, as discussed before, a pattern of activations in the previous layer cause a very specific pattern of activation in the next layer.

Learning with Gradient Descent

Introduction

What set Machine learning different than the classic methods for solving problems, is the learning part. Machine learning is all about devising systems that are able to learn from the training data to perform a certain task. In this chapter I discuss the main algorithms that are used for learning in neural networks and deep neural networks.

Cost (error) function

Now, we a have both, a design for our neural network, and the training data, agreed upon, and before we move on to the learning part. We need to figure out a way to measure the error of the model, the obvious way of doing that, is by computing the mean squared error (MSE) of the network, that is the average of the squared difference between the predicted and target values of all neurons in the output layer – that is why it's called the error function.

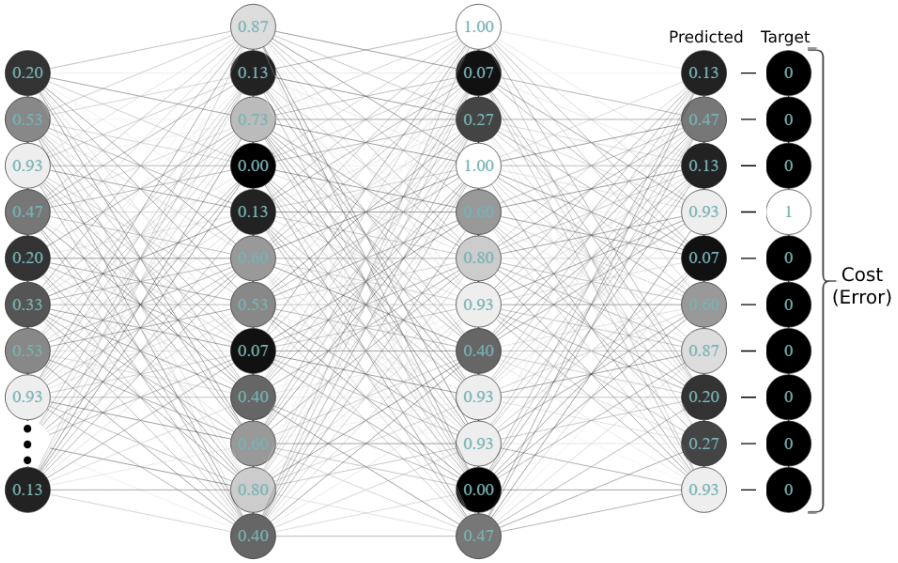


Figure 3- 2 Measuring the cost (error) of the NN

The reason we square the differences (aka residuals), is because this will produce a convex function, which is easy to work with and reason about. In another word, by measuring the error on the test data, we devise a mean that allows us to optimize the performance by minimizing the objective function. A way to tell the model how well or bad it is doing on the classification task, so it better understands how to optimize its weights and biases to do a better job, or put differently, learn what are the most optimal weights and biases. For that we will use the MSE Cost function:

$$C(w,b) = \frac{1}{2n}\sum_{x}\|y(x) - a\|^2 \quad (7)$$

where w and b denote, the collection of weights and biases for the network respectively, n is the total number of training inputs, a is 10-dimensional vector that represent the correct answer sort of speak for the current input x. The notation ‖v‖ just denotes the usual length function (magnitude) for a vector v. It's common to refer to the function C as the Quadratic Cost Function, or the loss function and it's also sometimes known as the mean squared error or just MSE. [1]

Why are we dividing by 2?

Well, it is simple. It is included for later convenience, due to the fact that when you take the derivative of the cost function, that 2 in the power get canceled with the 1/2 multiplier, thus the derivation is cleaner. These techniques are widely used in math in order "To make the derivations mathematically more convenient". You can simply remove the multiplier, and expect the same result [3].

Two features of any cost function

Inspecting the form of the quadratic cost function, we see that:

a) $C(w,b)$ is non-negative, since every term in the sum is non-negative.
b) The cost $C(w,b)$ becomes small, i.e., $C(w,b) \approx 0$ precisely when $y(x)$ is approximately equal to the output, a, for all training inputs, x.

In fact, these two features are the two-primary requirements in any cost (error) function. And we see that MSE fulfill them, later in the next chapter in the optimization section we will consider using other cost (error) function.

So, our training algorithm has done a good job if it can find weights and biases so that $C(w,b) \approx 0$. By contrast, it's not doing so well when $C(w,b)$ is large - that would mean that y(x) is not close to the output a for a large number of inputs. So, the aim of our training algorithm will be to minimize the cost $C(w,b)$as a function of the weights and biases. In other word, we want to find a set of weights and biases which make the cost as small as possible. We'll do that using an algorithm known as gradient descent. [1, 2]

Gradient Descent

Going back to the main goal that we have in mind, which is to find the collection of weights and biases for the network that minimize the cost function, which is a well posed problem if we ignore most of the distracting details, just focusing on the fact that the cost function is just a positive real valued function, with many inputs, that we want to minimize.

For simplicity, let's assume that the cost (error) function is a function with two variables (inputs), the graph of the function is a 3D valley , imagine a ball rolling down the slope of the valley, let's think about what happens when we move the ball a small amount Δv_1 in the v_1 direction, and a small amount Δv_2 in the v_2 direction. Calculus tells us that C changes as follows:

$$\Delta C \approx \frac{\partial C}{\partial v_1}\Delta v_1 + \frac{\partial C}{\partial v_2}\Delta v_2 \tag{8}$$

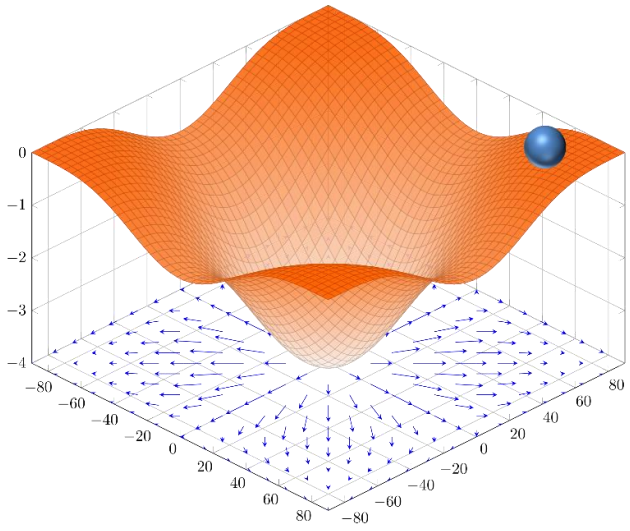


Figure 3- 3: 3D graph of two variables function with the its gradient vector field

We're going to find a way of choosing ΔV_1 and ΔV_2 so as to make ΔC negative; i.e., we'll choose them so the ball is rolling down into the valley. To figure out how to make such a choice it helps to define ΔV to be the vector of changes in v, $\Delta V \equiv (\Delta V_1, \Delta V_2)^T$, where T is the transpose operation, turning row vectors into column vectors.

The gradient of C is the vector of partial derivatives, $(\partial C/\partial v_1, \partial C/\partial v_2)^T$. We denote the gradient vector by ΔC, i.e.:

$$\nabla C \equiv \left(\frac{\partial C}{\partial v_1}, \frac{\partial C}{\partial v_2}\right)^T \tag{9}$$

With these definitions, the expression (10) for ΔC can be rewritten as

$$\Delta C \approx \nabla C \cdot \Delta v \tag{10}$$

∇C relates changes in v to changes in C. But what's cool about this equation is that it lets us see how to choose Δv so as to make ΔC negative

$$\Delta v = -\eta \nabla C \tag{11}$$

where η is a small, positive parameter (known as the learning rate). Then equation (12) tells us that $\Delta C \approx -\eta \nabla C \cdot \nabla C = -\eta \parallel \nabla C \parallel^2$. Because $\parallel \nabla C \parallel^2 \geq 0$, this guarantees that ΔC≤0, i.e., C will always decrease, never increase, if we change v according to the prescription in (13). Then we use equation (10) to compute the value of Δv, to change v.

$$v \rightarrow v' = v - \eta \nabla C \tag{12}$$

Then we'll use this update rule again, to make another move. If we keep doing this, over and over, we'll keep decreasing C until - we hope - we reach a global minimum.

To make gradient descent work correctly, we need to choose the learning rate η to be small enough that Equation (12), is a good approximation. If we don't, we might end up with $\Delta C > 0$, which obviously would not be good! At the same time, we don't want η to be too small, since that will make the changes Δv tiny, and thus the gradient descent algorithm will work very slowly. *In practical implementations, η is often varied so that Equation (9) remains a good approximation.*[1], and this is known as adaptive learning rate.

I've explained gradient descent when C is a function of just two variables. But, in fact, everything works just as well even when C is a function of many more variables. Suppose in particular that C is a function of m variables, $v_1, \ldots, v_m$. Then the change ΔC in C produced by a small change $\Delta v = (\Delta v_1, \ldots, \Delta v_m)^T$ is

$$\Delta C \approx \nabla C \cdot \Delta v \tag{13}$$

where the gradient ∇C is the vector

$$\Delta C \equiv \left(\frac{\partial C}{\partial v_1}, \ldots, \frac{\partial C}{\partial v_m}\right)^T \tag{14}$$

Just as for the two-variables case, we can choose

$$\Delta v - \eta \nabla C \tag{15}$$

and we're guaranteed that our (approximate) expression (12) for ΔC will be negative. This gives us a way of following the gradient to a minimum, even when C is a function of many variables, by repeatedly applying the update rule

$$v \rightarrow v' = v - \eta \nabla C \tag{16}$$

You can think of this update rule as defining the gradient descent algorithm. It gives us a way of repeatedly changing the position v in order to find a minimum of the function C. [1]

How can we apply gradient descent to learn in a neural network? The idea is to use gradient descent to find the weights (w_k) and biases (b_l) which minimize the cost in Equation (9). To see how this works, let's restate the gradient descent update rule, with the weights and biases replacing the variables (v_j) In other words, our "position" now has components (w_k) and (b_l), and the gradient vector (∇C) has corresponding components ($\partial C/\partial w_k$) and ($\partial C/\partial b_l$). Writing out the gradient descent update rule in terms of components, we have

$$w_k \rightarrow w'_k = w_k - \eta \frac{\partial C}{\partial w_k} \tag{17}$$

$$b_l \rightarrow b'_l = b_l - \eta \frac{\partial C}{\partial b_l} \tag{18}$$

By repeatedly applying this update rule we can "roll down the hill", and hopefully find a minimum of the cost function. In other words, this is a rule which can be used to learn in a neural network.[1]

Contrasting the 3 Variation of Gradient Descent:

Batch Gradient Descent

The implementation of the gradient descent I've just described, is known as batch gradient descent. It calculates the error for each example in the training dataset, but only updates the model after all training examples have been evaluated - Yes, the name is misleading a bit since we use the term "batch" mainly in "mini-batch" to refer to small set of the training set, and

the term epoch to refer to the entire training set, thus, it should make more since if it was named "Epoch Gradient Descent" but this term is never used.

- Upsides:
  - Computationally efficient, since we only perform model update once.
  - Deferring the update of model's parameters to the end, allows for parallel implementations of the algorithm.
  - Averaging the gradient over the entire training set means less variation, avoiding local minima.
- Downsides:
  - As the size of the training set and samples increase, this method won't be feasible, since it requires that the entire training set to be readily accessible in memory.
  - As the size of the training set and samples increase, this method become inefficient, since it is required keep the gradient of each sample in memory, (or write it somewhere in the disk which extremely slow) and after all the training set is exhausted, accumulate and the gradient of all sample and apply it.
  - No feedback of the performance, since this method won't have any way of providing how the optimization is doing.
  - Error information are averaged across the all training data.

The main problem with this implementation is, it's slow. If we inspect the cost function in Equation (9), we see that we need to average the cost for the entire training data. In practice, to compute the gradient (∇C) we need to compute the gradients (∇C x) separately for each training input, (x), and then average them, $\nabla C = 1/n \sum_x \nabla Cx$ [1]. And when the number of training input is large as it is the case for most situation in Machine Learning this can take a long time, and learning thus occurs slowly.

Stochastic Gradient Descent

An idea called **stochastic gradient descent** can be used to speed up learning. The idea is to estimate the gradient ∇C by computing ∇Cx for a small sample of randomly chosen training inputs. By averaging over this small sample, it turns out that we can quickly get a good estimate of the true gradient ∇C, and this helps speed up gradient descent, and thus learning.

To make these ideas more precise, stochastic gradient descent works by randomly picking out a small number m of randomly chosen training inputs. We'll label those random training inputs $X_1, X_2, \ldots, X_m$, and refer to them as a mini-batch. Provided the sample size m is large enough we expect that the average value of the $\nabla C x_j$ will be roughly equal to the average over all ∇C_x, that is,

$$\frac{\sum_{j=1}^{m} \nabla C_{X_j}}{m} \approx \frac{\sum_x \nabla C_x}{n} = \nabla C \tag{19}$$

where the second sum is over the entire set of training data. Swapping sides, we get

$$\frac{\sum_{j=1}^{m} \nabla C_{X_j}}{m} \approx \frac{\sum_x \nabla C_x}{n} = \nabla C \tag{20}$$

confirming that we can estimate the overall gradient by computing gradients just for the randomly chosen mini-batch.

To connect this explicitly to learning in neural networks, suppose w_k and b_l denote the weights and biases in our neural network. Then stochastic gradient descent works by picking out a randomly chosen mini-batch of training inputs, and training with those,

$$w_k \rightarrow w'_k = w_k - \frac{\eta}{m} \sum_j \frac{\partial C_{X_j}}{\partial w_k} \tag{21}$$

$$b_k \rightarrow b'_k = b_l - \frac{\eta}{m} \sum_j \frac{\partial C_{X_j}}{\partial b_l} \tag{22}$$

where the sums are over all the training examples X_j in the current mini-batch. Then we pick out another randomly chosen mini-batch and train with those. And so on, until we've exhausted the training inputs, which is said to complete an epoch of training. At that point we start over with a new training epoch.

- Upsides:
  - Averaging the gradient of a mini-batch results robust convergence avoiding local minima.
  - Computationally efficient, since it only updates the model after each mini-batch.
  - Efficient use of memory, since we don't need to have all the training set, in memory.

- Almost Immediate insight into the performance of the model and the rate of improvement.

➢ Downsides:

- Additional hyper-parameter 'Mini-batch Size'

Online Gradient Descent:

Online Gradient Descent, is a variation of the gradient descent algorithm that calculates the error and updates the model for each example in the training dataset.

➢ **Upsides**:

- Simple
- Immediate insight into the performance of the model and the rate of improvement.
- Frequent model update, can speed up learning in some cases.

➢ **Downsides**:

- Slow, since the model update is computationally expensive, thus take significantly longer to train.
- Unstable gradient, since each training sample have completely different gradient, which may cause the model error to jump around (have a higher variance over training epochs).

The Mini-Batch Size (commonly referred to just as Batch Size):

- Small values result a faster converges at the cost of noise in the training process.
- Large values result a slower converges with accurate estimates of the error gradient.

What value should we use? It's usually recommended that the batch-size is chosen is a power of two, that is: 16, 32, 64, 128 or even 256. The reason why is to fit the CPU (or GPU) hardware architecture when vectorizing the computation, so that all mini-batch samples gradient is computed in vectorized manner (single step of computation), allowing for fast efficient training. And 32 seems to be the commonly recommended value by many papers;

[35] writes "[batch size] is typically chosen between 1 and a few hundreds, e.g. [batch size] = 32 is a good default value, with values above 10 taking advantage of the speedup of matrix-matrix products over matrix-vector products.". And [34] also suggest the same batch-size, "The presented results confirm that using small batch sizes achieves the best training stability and generalization performance, for a given computational cost, across a wide range of experiments. In all cases the best results have been obtained with batch sizes m = 32 or smaller, often as small as m = 2 or m = 4."

Back-Propagation:

Now, we discussed the gradient descent, the question that arise is how do we calculate the gradient. So, in this next section, I start by taking a deep dive to how we can calculate the gradient descent using an algorithm known as the Back-propagation. But before we do so we need to agree on some ground rules, like how we will refer to elements in the network using notation, and some basic linear algebra concepts. Then, we will run the code and test its performance. Finally, we will turn our focus into some optimization techniques that will help improve the performance

linear algebra:

Linear algebra plays an extremely important role in Machine Learning, and it's necessary to be familiar with its fundamental concepts. So, in the next section I will take a quick look at some of these concepts.

- Vectors: A vector is an array of numbers. The numbers are arranged in order. We can identify each individual number by its index in that ordering. We can think of vectors as identifying points in space, with each element giving the coordinate along a different axis.
- Matrices: A matrix is a 2-D array of numbers, so each element is identified by two indices instead of just one. If a real-valued matrix A has a height of m and a width of n, then we say that $A \in R^{m \times n}$.
- The Transpose of The Matrix: can be thought of as a mirror image across the main diagonal. where the column vectors become rows vectors, in the resulting matrix, and vice-versa. We denote the transpose of a matrix A as A^T, and it is defined such that $(A^T)_{i,j} = A_{j,i}$.
- Tensors: are a generalization of matrices, since matrices are 2-D arrays, and often we need higher dimensional arrays, for this we use the term Tensor.

Matrices and Vectors addition and subtraction:

Vectors, matrices and tensors of the same length, can be added together (or subtracted), the only requirement is that, they need to be of the same dimension and same size.

Matrices and Vectors Multiplication:

One of the most important operations involving matrices is multiplication of two matrices. The matrix product of matrices A and B is a third matrix C. In order for this product to

be defined, A must have the same number of columns as B has rows. If A is of shape m × n and B is of shape n × p, then C is of shape m × p. The product operation is defined by:

$$C_{i,j} = \sum_k A_{i,k} B_{k,j} \tag{23}$$

The dot product:

The dot product between two vectors x and y of the same dimensionality is the matrix product of the transpose of the first vector matrix multiplied with the second vector, $x^T y$, result a scaler.

The Hadamard product:

The backpropagation algorithm is based on common linear algebraic operations - things like vector addition, multiplying a vector by a matrix, and so on. But one of the operations is a little less commonly used. In particular, suppose and are two vectors of the same dimension. Then we use s ⊙ t to denote the elementwise product of the two vectors. Thus, the component of the Hadamard product of two vectors (should be of the same size), is the product of each element from one vector with the corresponding element from the other vector. As an example,

$$\begin{bmatrix}1\\2\end{bmatrix} \odot \begin{bmatrix}3\\4\end{bmatrix} = \begin{bmatrix}1 \cdot 3\\2 \cdot 4\end{bmatrix} = \begin{bmatrix}3\\8\end{bmatrix} \tag{24}$$

Matrix-Based Approach for Computing the Output from a Neural Network

Let's begin with a notation which lets us refer to biases and activation in the network in an unambiguous way. we will use b_j^l for the bias of the j^{th} neuron in the l^{th} layer. And we use a_j^l for the activation of the j^{th} neuron in the l^{th} layer.

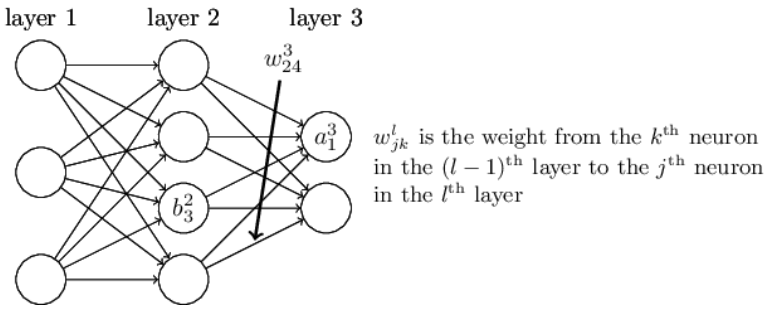


Figure 3- 4: Weights, Biases and Activations Indexing [1]

We use a similar notation for the network's weights w_{jk}^{l} to denote the weight for the connection from the k^{th} neuron in the $(l-1)^{th}$ layer to the j^{th} neuron in the l^{th} layer. So, for instance, the diagram shows the weight on a connection from the fourth neuron in the second layer to the second neuron in the third layer of a network.

Note: The weights connecting all neurons form the previous layer, to the neuron in the next layer are represented with a row in the weight's matrix (as demonstrated below). This is the convention I followed, throughout this research and the code implementation. Different books and libraries may do this differently.

Figure 1 Visual representation of how the weight matrix looks like

This ordering of the j and k indices may seem strange - surely, it'd make more sense to swap the and indices around? The big advantage of using this ordering is that it makes the activation a_j^l of the j^{th} neuron in the l^{th} layer related to the activation in the $(l-1)^{th}$ layer by the equation:

$$a_j^l = \sigma\left(\sum_k w_{jk}^l a^{(l-1)} + b_j^l\right) \tag{25}$$

where the sum is over all neurons in the $(l-1)^{th}$, layer. To rewrite this expression in a matrix form we define a weight matrix w^l for each layer, l. The entries of the weight matrix w^l are just the weights connecting to the l^{th} layer of neurons, that is, the entry in the j^{th} row and k^{th} column is w_{jk}^l . Similarly, for each layer we define a bias vector, b^l. You can probably guess

how this works - the components of the bias vector are just the values b_j^l, one component for each neuron in the l^{th} layer. And finally, we define an activation vector a^l whose components are the activations.

The last ingredient we need to rewrite Equation (27) in a matrix form, is the idea of vectorizing a function such as σ. We use the obvious notation σ(a) to denote this kind of elementwise application of a function. With these notations in mind, Equation (27), can be rewritten in the beautiful and compact vectorized form:

$$a^l = \sigma\left(w^l a^{l-1} + b^l\right) \tag{26}$$

This expression gives us a much more global way of thinking about how the activations in one layer relate to activations in the previous layer: we just apply the weight matrix to the activations, then add the bias vector, and finally apply the activation function*, to compute a^l, we compute the intermediate quantity $z^l = w^l a^{(l-1)} + b^l$ along the way. This quantity turns out to be useful enough to be worth naming: we call z^l the weighted input to the neurons in layer. We'll make considerable use of the weighted z^l input later in the chapter.

Back-Propagation:

When the model is a feedforward neural network, the gradient of the cost function can be computed economically with an algorithm called backpropagation algorithm [Rumelhart et al. 1986; Werbos 1974[6]. The backpropagation algorithm was originally introduced in the 1970s, but its importance wasn't fully appreciated until a famous 1986 paper by David Rumelhart, Geoffrey Hinton, and Ronald Williams. That paper describes several neural networks where backpropagation works far faster than earlier approaches to learning, making it possible to use neural nets to solve problems which had previously been insoluble. Today, the backpropagation algorithm is the workhorse of learning in neural networks.[1] Actually, backpropagation is not a training algorithm, but an ingredient in a training procedure.[6]

Simplest Neural Network Backpropagation:

To get started with the backpropagation, I will start with a super simple network that consist only of two neurons the input (in the input layer) and output neuron (in the output layer) connected with one weighed connection **and no added bias**

The output neuron takes the output of the input neuron as an input and multiplies it with the weight with no added bias, all scalar values. In another word, the activation is simply the net input times the weight and there is no non-linearity involved. We randomly initialize the weight to 0.8 and we have a simple training data set, consisting of an input 1.5 and desired of 0.5, we would like this network to produce the value of 0.5.

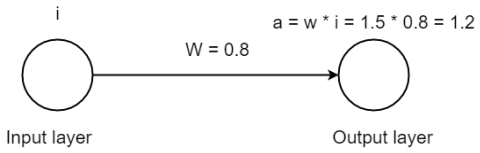


Figure 3- 5: A network with a single neuron

With the current weight however, you can see that it actually produces 1.2, we need to define the error for the network to be able to train itself we use the MSE as our cost function, $C = (a - y)^2$, so we simply take the difference between the network current output and the actual desired output, squared.

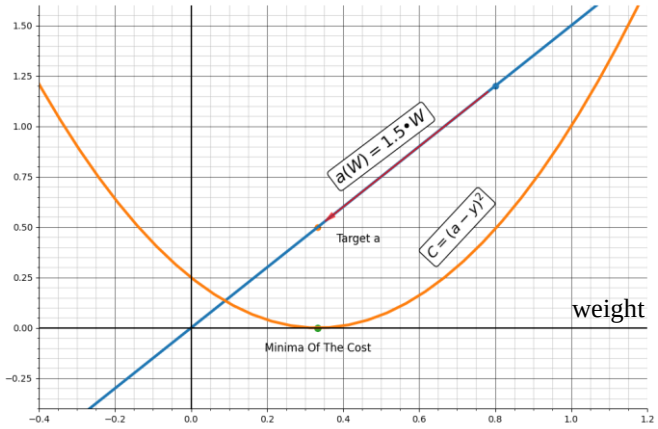


Figure 3- 6: The output activation and the cost (error) function

looking at the activation as a function of the weight, we can see that where is it currently it outputs 1.2, and in order to bring it down to where we want it to be, which is 0.5 for the activation, the weight has to be just above 0.3. Now to formalize this desire of ours, we bring in the error function which when plotted looks like a parabola and you can clearly see in in the figure above.

The Backpropagation algorithm is a way to calculate the gradient that we use to minimize the error by descending along the cost function. It looks at the slope at each given point and figures out which way is down, so we need to know the slope or the derivative of the cost function which is simple to calculate.

$$\frac{\partial C}{\partial a} = 2(a - y) \tag{27}$$

However, the only thing we can change in the network is the weight W, so we need an expression for the rate of change of the cost function with respect to W and for that we need the chain rule of differentiation, which states that if there is a function of a function the derivative of that function is the derivative of the inner function times the derivative of the outer function, this should be familiar to most of you.

$$\frac{\partial f(g(x))}{\partial x} = g'(x)f'(g(x)) \tag{28}$$

All right so what we're trying to find out is how much can we improve the situation, or how much can we reduce the error by adjusting W, and we obtained this by finding out how much W affects the output activation and how much the output activation effects the error function, these two derivatives of course are very simple to obtain, $\frac{\partial C}{\partial w} = \frac{\partial C}{\partial a}\frac{\partial a}{\partial z}$, where $\frac{\partial C}{\partial a} = 2(a - y)$, $\frac{\partial a}{\partial w} = i = 1.5$. When we multiply the two terms together, $\partial C/\partial w = 1.5 * 2(a - y) = 3a - 3y = 4.5w - 3.6$, we get a first-order polynomial, which is plotted bellow, and this tells us the direction in which we need to change W.

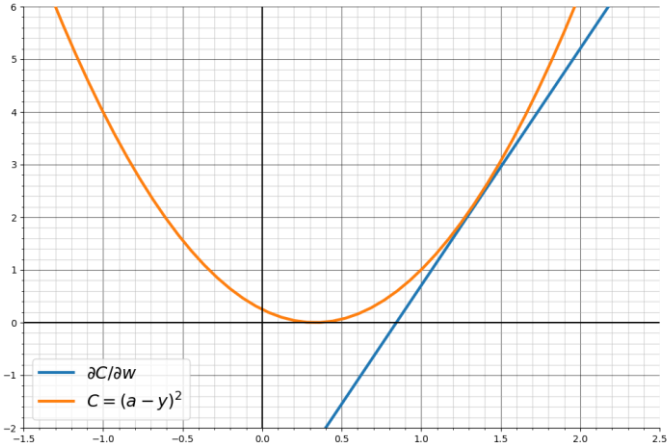


Figure 3- 7: partial derivation of the cost with respect to the weight

So, on right side of the minima of the cost function, the gradient is positive, we need to deduct or reduce W by a positive number so we're actually going down, and if the weight was less than 0.333 (on the other side of cost function) the gradient is negative so we need to deduce from W a negative number in other words we're increasing W, that makes sense because we're always converging towards this minimum of the cost function.

Now we have all the parts we need to actually learn and train the model, we are going to reduce the weight or modify the weight proportionately to the gradient, we also use the learning rate to define by how much exactly we are going to bring the effect of the gradient into our variables.

$$W = W_0 - r\frac{\partial C}{\partial w} = W_0 - r\,\frac{\partial C}{\partial a}\frac{\partial a}{\partial z} = W_0 - r\;2(a-y)\;1.5 = W_0 - 0.1\;(4.5\,W_0 - 3.6)$$

So, in a simple expression we deduct from the old W_0, the rate of change of C with respect to W, multiplied by the learning rate. And in this case, we have 0.1 as our learning rate, $4.5 \times W_0 - 3.6$ is our gradient, and we are ready to start calculating numbers. So, as you remember our weight was 0.8 when we put that value in expression above, and calculate this the value of this expression it comes out to 0.59 and we iterate this, we start converging towards a value in this case 3.333...

So that's how backpropagation works, we have now trained this model and the optimal weight for this model is 0.33333. This is also very easy to generalize to several layers, now that we understand how the chain rule of differentiation is used in back propagation. Previously we looked at a network with no hidden layers, only an input and output layer, but if this was a network with any number of hidden layers, the latter example would be the last layers, in a network with multiple layers.

$$-r\;\frac{\partial C}{\partial a^L}\frac{\partial a^L}{\partial w^L}$$

***Note**: the L in indicate the activation of the last layer (aka the output layer).*

Now on the next level of back propagating, we use the same terms, from previous layers, but we add another factor which is how much does the weight W of the next layer (propagating

backward) changes the error (or the cost function), and so on, we go one layer after another as far as we have to go.

$$-r \frac{\partial C}{\partial a^L} \frac{\partial a^L}{\partial a^{L-1}} \frac{\partial a^{L-1}}{\partial a^{L-2}} \dots \frac{\partial a^{L-n}}{\partial w^{L-n}} \tag{29}$$

In intuitive terms, what this means is the higher the activation behind the weight in the current layer, the more significance this weight has for the network, and therefore we change it proportionately to the activation from the previous layer. Similarly, the weights of the path from the current neuron all the way to the output activation, this tells us how significant the change in that neuron, is for this given output, and if that path is clear meaning the strength of the weights along the path is strong then the strength of our adjustment is also going to be great.

Training a Neural Network with Backpropagation

In the single-layer neural network, the training process is relatively straightforward because the error (or loss function) can be computed as a direct function of the weights, which allows easy gradient computation.

In the case of multi-layer networks, the problem is that the loss is a complicated composition function of the weights in earlier layers. The backpropagation algorithm leverages the chain rule of differential calculus, which computes the error gradients in terms of summations of local-gradient products over the various paths from a node to the output. Although this summation has an exponential number of components (paths), one can compute it efficiently using dynamic programming. [7]

Backpropagation's two phases:

The backpropagation algorithm is a direct application of dynamic programming. It contains two main phases, referred to as the forward and backward phases, respectively. The forward phase is required to compute the output values and the local derivatives at various nodes, and the backward phase is required to accumulate the products of these local values over all paths from the node to the output:[7]

- **Forward phase:** In this phase, the inputs for a training instance are fed into the neural network. This results in a forward cascade of computations across the layers, using the

current set of weights. The final predicted output can be compared to that of the training instance and the derivative of the loss function with respect to the output is computed. The derivative of this loss now needs to be computed with respect to the weights in all layers in the backwards phase.[7]

- **Backward phase:** The main goal of the backward phase is to learn the gradient of the loss function with respect to the different weights by using the chain rule of differential calculus. These gradients are used to update the weights. Since these gradients are learned in the backward direction, starting from the output node, this learning process is referred to as the backward phase.[7]

Backpropagation's four fundamental equations

Backpropagation is about understanding how changing the weights and biases in a network changes the cost function. But to compute those, we first introduce an intermediate quantity, δ_j^l , which we call the error in the j^{th} neuron in the l^{th} layer. Where the error δ_j^l is defined by:

$$\delta_j^l = \frac{\partial C}{\partial z_j^l} \tag{30}$$

And we use δ^l to denote the vector of errors associated with layer l.

What is the error δ all about and why computing it?

The error δ_j^l quantity, is a measure for how the change in one neuron changes the cost function. You might wonder why we are not considering the change in the cost function with respect to the output activation of the neuron instead of the intermediary quantities z, in fact things will work exactly the same if you do that. It turns out that by computing the error, we can easily compute the rate of the change of the cost function with respect to the bias or the weights of that neuron, starting from the same error expression.

$$\frac{\partial C}{\partial b_j^l} = \frac{\partial C}{\partial z_j^l}\frac{\partial z_j^l}{\partial b_j^l} = \delta_j^l \frac{\partial z_j^l}{\partial b_j^l} \tag{31}$$

$$\frac{\partial C}{\partial w_{jk}^l} = \frac{\partial C}{\partial z_j^l}\frac{\partial z_j^l}{\partial w_{jk}^l} = \delta_j^l \frac{\partial z_j^l}{\partial w_{jk}^l} \tag{32}$$

- An equation for the error in the output layer

$$\delta_j^L = \frac{\partial C}{\partial z_j^L} = \frac{\partial C}{\partial a_j^L}\frac{\partial a_j^L}{\partial z_j^L} = \frac{\partial C}{\partial a_j^L}\sigma'(z_j^L) \quad (33)$$

(BP1)

***Note**: The exact form of $\partial C/\partial a$ will, of course, depend on the form of the cost function. For instance, if we are using the quadratic cost function $C = \frac{1}{2}\sum_j (y_j^L - a_j)^2$ then $\partial C/\partial a_j^L = (a_j^L - y_j)$, which obviously is easily computable. [1]*

Now, we need to write the BP1 in the matrix-based form we want for backpropagation, we write the partial derivation of the cost function with respect to a (the activation of the output layer) as $\nabla_a C$ a vector whose components are the partial derivatives $\partial C/\partial a$, or the gradient of the cost with respect to a. As an example, in the case of the quadratic cost we have $\nabla_a C = (a^l - y)$ and so the fully matrix-based form of (BP1) becomes

$$\delta^L = (a^L - y) \odot \sigma'(z_j^L) \quad (34)$$

(BP1a)

- An equation for the error in terms of the error in the next layer, δ^{L+1}:

$$\delta^l = \left(\left(w^{l+1}\right)^T \delta^{l+1}\right) \odot \sigma'(z_j^l) \quad (35)$$

(BP2)

Where $\left(w^{(l+1)}\right)^T$ is the transpose of the weight matrix $w^{(l+1)}$ for the $(l+1)^{th}$ layer. When we apply the transpose to the weight matrix, $\left(w^{(l+1)}\right)^T$ – remember, this is because we used a convention where each row of the weight matrix represent the connection of the previous layer's neurons with a neuron in the next layer $(l+1)^{th}$ - intuitively this can be thought like moving the error (δ^{l+1}) of the $(l+1)^{th}$ layer, backward through the network, giving us some sort of measure of the error at the output of the l^{th} layer.

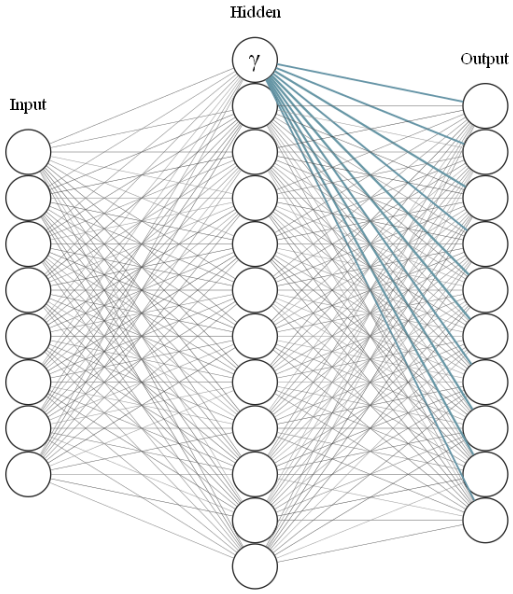


Figure 3- 8 Back-propagating the error

*Note: In the figure above (*Figure 3- 8*), I am only highlighting the back-propagation process for a single neuron, but the errors for all neurons of the same layer are computed in a single step. That's the beauty of the back-propagation, and why we use at the first place, it allows for efficient computation of the gradient.*

By combining (BP2) with (BP1) we can compute the error δ^l for any layer in the network. We start by using (BP1) to compute δ^L, then apply Equation (BP2) to compute δ^{L-1}, then Equation (BP2) again to compute δ^{L-2}, and so on, all the way back through the network.

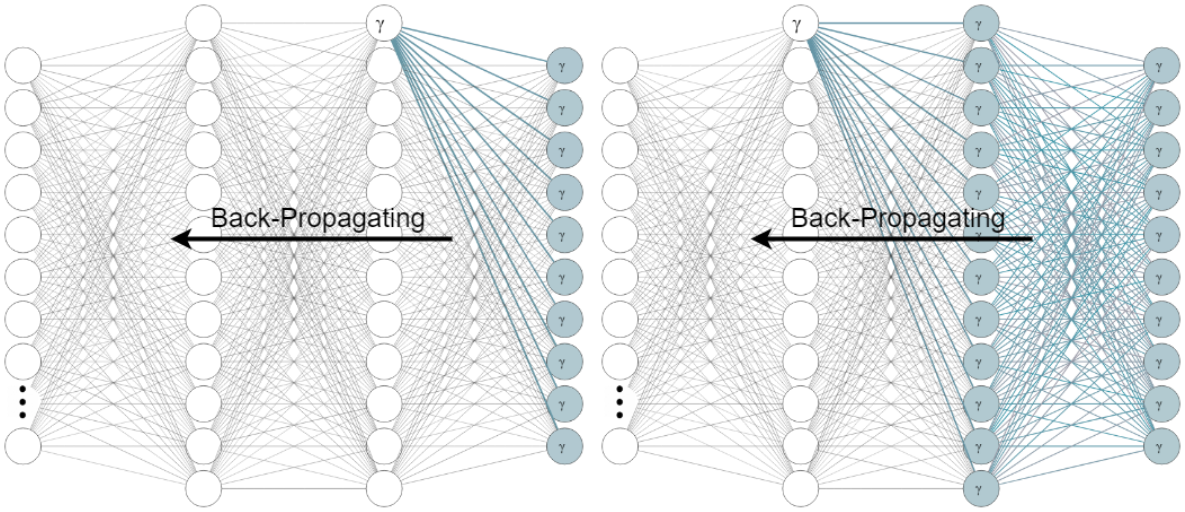


Figure 3- 9 Illustration of the back-propagation process through a 3 layers network.

- An equation for the rate of change of the cost with respect to any bias:

$$\frac{\partial C}{\partial b_j^l} = \delta_j^l \tag{36}$$

(BP3)

That is, the error δ_j^l is exactly equal to the rate of change $\partial C/\partial b_j^l$, and that's because when the intermediary value z: $z^l = w^l a^{(l-1)} + b^l$, differentiated with b. we get 1

$$\frac{\partial z_j^l}{\partial b_j^l} = 1 \tag{37}$$

- An equation for the rate of change of the cost with respect to any weight in the network:

$$\frac{\partial C}{\partial w_{jk}^l} = a_k^{l-1}\delta_j^l \tag{38}$$

(BP4)

This tells us how to compute the partial derivatives $\partial C/\partial w_{jk}^L$ in terms of the quantities δ^l and a^{l-1}, which we already know how to compute.

To give a brief explanation to why the expression above is true. We already know how to compute the error for any layer (for any neuron) in the network $\delta_j^l = \partial C/\partial z_j^l$ one small foot left is to compute the change of the input value z of the neuron, with respect to the change in its weights,

$$\frac{\partial C}{\partial w_{jk}^l} = \frac{\partial C}{\partial z_j^l}\frac{\partial z_j^l}{\partial w_{jk}^l} \tag{39}$$

where:

$$\frac{\partial z^l}{\partial w^l} = \frac{\partial\left(w^l a^{l-1} + b^l\right)}{\partial w^l} = a^{l-1} \tag{40}$$

***Note**: the σ function becomes very flat when $\sigma\left(z_j^L\right)$ is approximately 0 or 1. When this occurs, we will have $\sigma'\left(z_j^L\right) \approx 0$. So, the weight in the final layer will learn slowly if the output neuron is either low activation (≈0) or high activation (≈1). In this case it's common to say the output neuron has saturated and, as a result, the weight has stopped learning (or is learning slowly) – a common problem in machine learning that we will come back to later. Similar remarks hold also for the biases of output neuron. We can also obtain similar insights for earlier*

layers. In particular, note the $\sigma'(z^l)$ term in (BP2). This means that δ_j^l is likely to get small if the neuron is near saturation. And this, in turn, means that any weights input to a saturated neuron will learn slowly.*

Summing up, the four fundamental equations behind backpropagation:

$$\delta^L = (a^L - y) \odot \sigma'(z_j^L) \quad \text{(BP1)}$$

$$\delta^l = \left((w^{l+1})^T \delta^{l+1}\right) \odot \sigma'(z_j^l) \quad \text{(BP2)}$$

$$\frac{\partial C}{\partial b_j^l} = \delta_j^l \quad \text{(BP3)}$$

$$\frac{\partial C}{\partial w_{jk}^l} = a_k^{l-1}\delta_j^l \quad \text{(BP4)}$$

Chapter four: Implementation and Optimization

Introduction

Much of the recent progress made in image classification research can be credited to training procedure refinements, such as changes in data augmentations and optimization methods [28]. In this chapter we will examine a collection of such refinements and empirically evaluate their impact on the final model accuracy through ablation study. I will start with a brief description about the code implementation.

The hyper-parameter of the network

Most machine learning algorithms have several settings that we can use to control the behavior of the learning algorithm. These settings are called hyper-parameter, in order to distinguish them from the parameters (weights and biases) learned by our learning algorithm. things like, the learning rate, number of epochs, (each epoch represents a single iteration over the entire training set), batch-size (the number of samples after which we update the network parameters), etc. The values of the hyper-parameters are not adapted by the learning algorithm itself (though we can design a nested learning procedure where one learning algorithm learns the best hyper-parameter for another learning algorithm) [2]. And of course, we need to make specific choices for them.

Implementing the neural network:

The code is implemented in Python, using nothing but NumPy (which is the essential package for numerical computing in Python) and it works as follows. In each epoch, it starts by randomly shuffling the training data, and then partitions it into mini-batches of the appropriate size. Then for each mini-batch we apply a single step of gradient descent. Which updates the network weights and biases according to a single iteration of gradient descent. The gradient descent iterates over the mini-batch samples and invokes the backpropagation, to calculate the gradient of the cost function, for each training sample.

Weight and Biases Initialization

The biases and weights in the Network are all initialized randomly, using Gaussian distributions with mean 0 and standard deviation 1. This random initialization gives our Stochastic Gradient Descent (SGD) algorithm a place to start from. Later we'll find better ways of initializing the weights and biases, but this will do for now.

Running the code

The question we are faced with right now is, what values should we use for the hyper-parameters? Well, this is tricky question, and to keep things simple for now, a simple and a common approach to land on good values for these hyper-parameters, is to hold off a part of the training data (which is referred to as the validation data) and train the network on different

values, and simply pick the ones that yield the best results. Why using the validation data for choosing the hyper-parameters instead of the training data, is something I will come back to later when we talk about overfitting. So, for now to keep things simple, I've gone ahead and pick arbitrary values that gave me the best result.

Starting with a network with one hidden layer, with 30 neurons, we will train for 30 epochs, and for our learning rate, we will start with 5. The final classification accuracy the network lands on, is 95.29%, which encouraging, considering that we did not spend any effort what so ever telling it what patterns or shapes to look for in the training data.

Optimization

Up to now, we have been working with the vanilla form of the neural network with no added twists, even though this gave us impressive results, (up to 95% accuracy) in its basic form, but we should not stop there, we should thrive for higher classification accuracy, thus developing optimization techniques to improve the performance of the network. Because there are situations, where the network will perform poorly in its basic form. So deep and through understanding of the machineries that govern the learning process, is required to combat these issues.

The techniques we'll develop in this next part include: a better choice of cost function, known as the cross-entropy cost function; four so-called "regularization" methods (L1 and L2 regularization, dropout, and artificial expansion of the training data), which make our networks better at generalizing beyond the training data; a better method for initializing the weights in the network [1].

Optimization techniques

Better cost (error) function

An important aspect of the design of a neural network is the choice of the cost function [2].

The cross-entropy cost function

Why do we need a different cost function?

As human being, we learn quickly when we're decisively wrong, in another word we tend correct quickly when we find out that we are very wrong. Ideally, we hope and expect that our neural networks have the same property, but unfortunately it turns out that's not the case with current implementation.

To see why, let's consider a network with only one neuron, and we want to train the neuron so, when the input is 1, we want the output to be 0. So, we initialize the weight and bias randomly, to 0.6 and 0.9 respectively. The initial output from the neuron is 0.82. Now we let the network learn the correct weight and bias, iterating over the same training input. We will plot the graph of the cost function in a bit, but before we do so, lets initialize the weight and bias to the

same different value now, which is 2. In this case the initial output is 0.98 which is worse than the first case which was 0.82, with initial weight and bias 0.6 and 0.9 respectively. We let the network learn again the correct weight and bias.

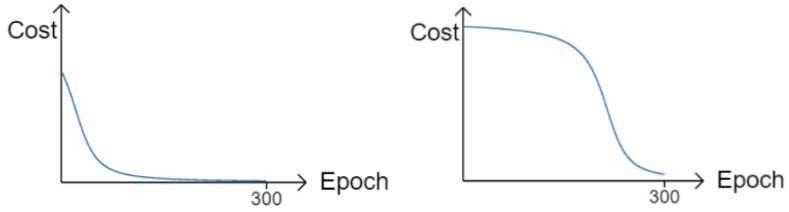


Figure 4 - 1 : Right neuron started in a saturated region which causes the learning to slow down

Looking at the graph of the cost function for both situations. We see that the artificial neuron doesn't behave anything like human learning, in another word, the neuron doesn't have the same feature of quick convergence toward the correct answer in the case where it's very wrong. What's more, it turns out that this behavior occurs not just in this super simple network with one neuron, it also occurs with large networks too.

To understand the origin of the problem, we need to inspect the expression of the gradient because it's the one part in the code that is responsible for learning. Remember that the gradient is the partial derivation of the cost function, we have: $C = \frac{(y-a)^2}{2}$. where a is the neuron's output when the training input x=1 is used, and y=0 is the corresponding desired output

$$\frac{\partial C}{\partial w} = (a - y)\sigma'(z)x = a\sigma'(z) \tag{41}$$

$$\frac{\partial C}{\partial b} = (a - y)\sigma'(z) = a\sigma'(z) \tag{42}$$

where I have substituted x=1 and y=0.

Looking at the expression of the gradient of the cost function with respect to the weight and bias, we see that both are the same, and both are proportional rate of change of the sigmoid. Recalling the shape of the graph of the Sigmoid function **Error! Reference source not found.**. W e see that it gets very flat when the output is large either positive of negative value, so the sigmoid prime tends to be very small near those regions.

This is the origin of the learning slowdown. What's more, as we shall see a little later, the learning slowdown occurs for essentially the same reason in more general neural networks, not just in this super basic example.

Introducing the cross-entropy cost function

It turns out that we can solve this problem by using different cost function, known as the cross-entropy. Which avoids the problem of learning slowing-down.

$$C = -\frac{1}{n}\sum_{x}[y \ln a + (1-y)\ln(1-a)] \tag{43}$$

where n is the total number of items of training data, the sum is over all training inputs, x, and y is the corresponding desired output, a is the network predicted output.

Since the cross-entropy expression is a sum of logarithmic terms, and both logarithm's inputs are in the range between 0 and 1; and there is a minus sign out the front of the sum, this means the cross-entropy is a positive real valued function so it achieves the first requirement (a). When the network output is the close to the desired output we see the cross-entropy will go to 0, which corresponds to the second requirement (b).

To see how the cross-entropy limit the problem of slow-learning, we need to inspect the expression of the gradient of the cost function, the way we did before with the quadratic-cost function. starting with the partial derivation of the cost with respect to the weight.

writing a as σ(z)

$$C = -\frac{1}{n}\sum_{x}[y \ln a + (1-y)\ln(1-a)] \tag{44}$$

$$\frac{\partial C}{\partial w_j} = -\frac{1}{n}\sum_{x}\left(\frac{y}{\sigma(z)} - \frac{(1-y)}{1-\sigma(z)}\right)\frac{\sigma(z)}{\partial w_j} \tag{45}$$

$$= -\frac{1}{n}\sum_{x}\left(\frac{y}{\sigma(z)} - \frac{(1-y)}{1-\sigma(z)}\right)\sigma'(z)x_j \tag{46}$$

Putting everything over a common denominator and simplifying this becomes:

$$\frac{\partial C}{\partial w_j} = -\frac{1}{n}\sum_{x}\frac{\sigma'(z)x_j}{\sigma(z)\big(1-\sigma(z)\big)}\,(\sigma(z)-y) \tag{47}$$

Using the definition of the sigmoid function, $\sigma(z) = 1/(1+e^{-z})$, and the sigmoid prime function $\sigma'(z) = \sigma(z)\big(1-\sigma(z)\big)$, we get:

$$\frac{\partial C}{\partial w_j} = -\frac{1}{n}\sum_{x} x_j(\sigma(z)-y) \tag{48}$$

This expression tells us that the rate at which the weight learns is controlled by σ(z)−y, i.e., by the error in the output**. The larger the error, the faster the neuron will learn**. The cross-entropy was specially chosen to have just this property.

In a similar way, we can compute the partial derivative for the bias.

$$\frac{\partial C}{\partial b_j} = -\frac{1}{n}\sum_{x}(\sigma(z)-y) \tag{49}$$

Again, this avoids the learning slowdown caused by the σ′(z) term, in the case of quadratic-cost.

Verifying that the cross-entropy actually works.

Running the same test, we did before with quadratic-cost function, using the same values as before for both situations, and plotting the output of the cost function.

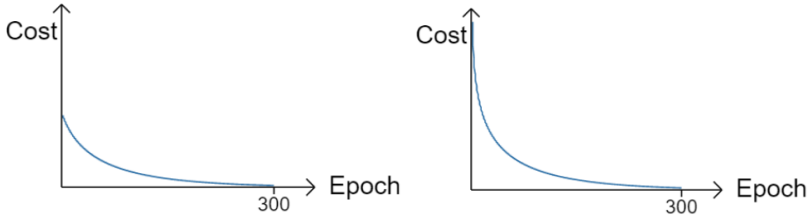


Figure 4 - 2 Saturation effect disappears with the use of the cross-entropy

This time the neuron learned quickly, just as we hoped.

Weight Initialization:

When we create a network, we have to make a definitive choice for the weights and biases. Up to now, we've been initializing them randomly using Gaussian distribution with mean of 0, and standard deviation of 1. Even though this approach served us well, it was quite ad hoc. It turns out that we can do quite a bit better than initializing with normalized Gaussians.

To understand why this method of parameter initialization, is not the best choice, we will consider a neuron in the first hidden layer, which gets its input from the input layer, where the weighted sum of that neuron is $z = \sum_j w_j x_j + b_j$, we will consider that half of the input neurons have value of 1 and the other half are 0. Thus, z is itself distributed as a Gaussian with mean zero and standard deviation $\sqrt{(n_{in}/2)}$, where n is the number of neurons in the input layer.

Plotting z, we see that is has a very broad Gaussian distribution, this means that z has relatively equal possibility to be any value between, $-\sqrt{(n_{in}/2)}$ and $\sqrt{(n_{in}/2)}$:

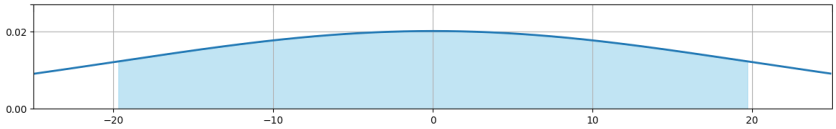


Figure 3- 10: Distribution of z with the Old method of weight initialization

Recalling the shape of the graph of the sigmoid function again, we know that the sigmoid tends to get flat when the input tends to be large either positive or negative value. which means that our hidden neuron would saturate, consequently, making small changes in the weights will make only minuscule changes in the activation of our hidden neuron, thus, barely affect the rest of the neurons in the network at all, and we'll see a correspondingly minuscule change in the cost function. As a result, those weights will only learn very slowly.

Now, the line of thinking is that, we want to have probability of z peaked around 0, so that the values of z, is relatively small centered around 0, thus the sigmoid neuron doesn't start in the saturated region. Then we shall initialize those weights as Gaussian random variables with mean 0 and standard deviation $1/\sqrt{n_{in}}$. That is, we'll squash the Gaussians down, making it less likely that our neuron will saturate.

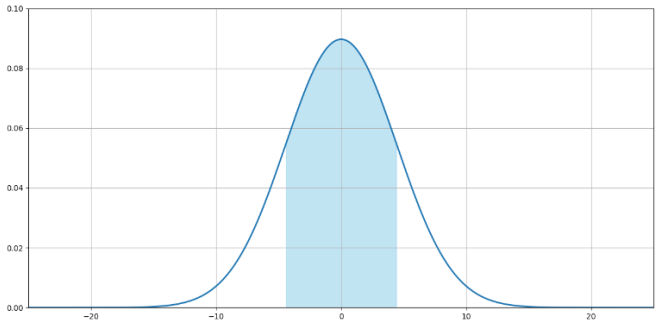


Figure 4 - 3 Distribution of z with the New method of weight initialization

We'll continue to initialize the biases the way we did before, as Gaussian random variables with a mean of 0 and a standard deviation of 1. The reason why, is because it is a single value - we only have one bias in each neuron - and its value will be between -1 and 1, which doesn't affect the weighted sum 'z' by much.

Comparing the old and new approaches of weight initialization:

We will use the same parameter as we have used before for both networks, 30 and 100 hidden neurons in each run respectively, a mini-batch size of 10, and the cross-entropy cost function, *a regularization parameter* $\lambda=5.0$. For the learning rate we will use $\eta=0.1$ because it turns out that it makes the results a little more easily visible when plotted.

Now, training the network using both the old and the new method of weight initialization, and plotting the classification accuracy side by side.

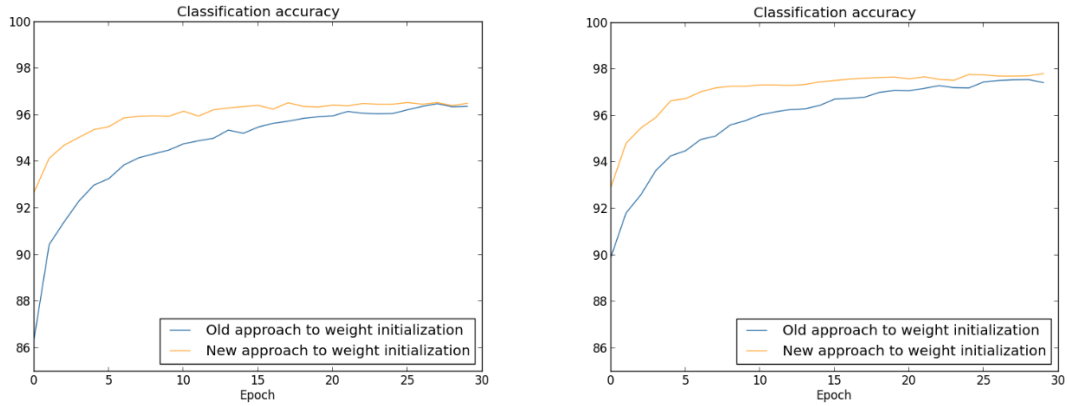


Figure 4 - 4 Classification Accuracy of the old and new method of weight initialization, the left figure corresponds to the 30 neurons network, the right figure corresponds to 100 neurons network

The final classification accuracy is almost exactly the same in both method in the two networks. But the new initialization technique brings us there much, much faster. the new method of weight initialization lands us on a slightly better classification accuracy in 100 neurons network.

Generalization:

Capacity, Overfitting and Underfitting and regularization

The central challenge in machine learning is that we must perform well on new, previously unseen inputs—not just those on which our model was trained. The ability to perform well on previously unobserved inputs is called generalization [2]

Typically, when training a machine learning model, we have access to a training set, we can compute some error measure on the training set called the training error, and we reduce this training error. So far, what we have described is simply an optimization problem. What separates machine learning from optimization is that we want the generalization error, also called the test error, to be low as well. The generalization error is defined as the expected value of the error on a new input.[2].

The two factors for improving the performance of a machine learning:

- Make the training error small.
- Make the gap between training and test error small.

These two factors correspond to the two central challenges in machine learning: underfitting and overfitting. [2]

- If the neural network has too many parameters (it is said to be over-parameterized), it will be too “flexible,” so that its output will fit very accurately all points of the training set

(including the noise present in these points), but it will provide meaningless responses in situations that are not present in the training set. That is known as overfitting.[6]

- By contrast, a neural network with too few parameters will not be complex enough to match the complexity of the (unknown) regression function, so that it will not be able to learn the training data. [6]

We can control whether a model is more likely to overfit or underfit by altering its capacity. Informally, a model's capacity is its ability to fit a wide variety of functions. Models with low capacity may struggle to fit the training set. Models with high capacity can overfit by memorizing properties of the training set that do not serve them well on the test set.[2]

Note: Machine learning algorithms will generally perform best when their capacity is appropriate for the true complexity of the task they need to perform, and the amount of training data they are provided with. Models with insufficient capacity are unable to solve complex tasks. Models with high capacity can solve complex tasks, but when their capacity is higher than needed to solve the present task, they may overfit.[2]

Regularization and overfitting avoidance is generally achieved by penalizing complexity of models or networks. In regularization, the training error and the complexity penalty should be of related functional forms.[5]

Overfitting:

A model with a large number of free parameters can describe an amazingly wide range of phenomena. Even if such a model agrees well with the available data, that doesn't make it a good model. It may just mean there's enough freedom in the model that it can describe almost any data set of the given size, without capturing any genuine insights into the underlying phenomenon. When that happens, the model will work well for the existing data, but will fail to generalize to new situations. The true test of a model is its ability to make predictions in situations it hasn't been exposed to before.

Constructing bad generalization model

To see the effect of overfitting, we will construct a situation where our network does a bad job generalizing to new situations. We'll use our 30 hidden neuron network. But we won't train the network using all 50,000 MNIST training images. Instead, we'll use just the first 1,000 training images. Using that restricted set will make the problem with generalization much more evident. We'll train in a similar way to before, using the cross-entropy cost function, with a learning rate of $\eta=0.5$ and a mini-batch size of 10. However, we'll train for 400 epochs, a somewhat larger number than before, because we're not using as many training examples. Tracking the cost changes as the network learns*

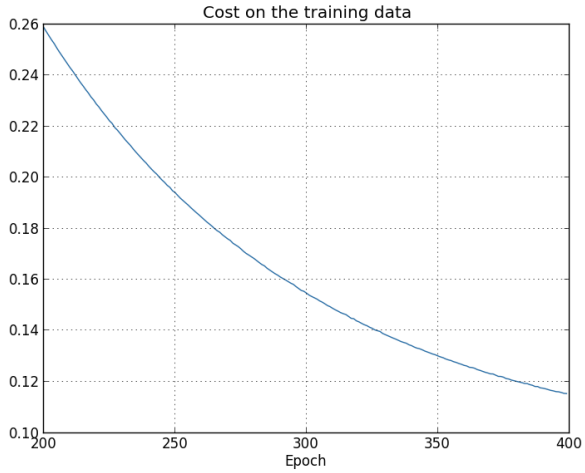


Figure 4 - 5 tracking the cost on the training data throughout the learning process.

Looking at the graph above this looks encouraging, showing a smooth decrease in the cost, just as we expect. Let's now look at how the classification accuracy on the test data changes over time:

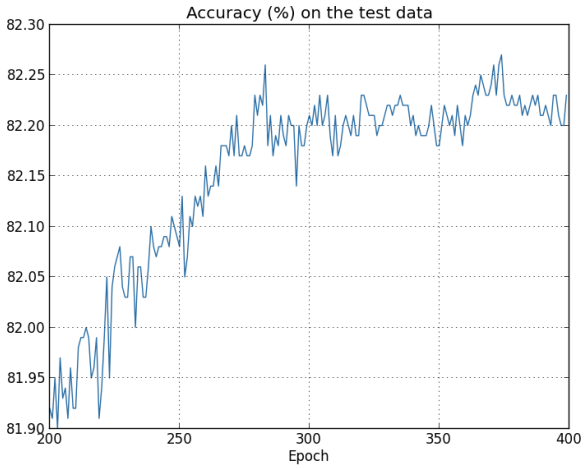


Figure 4 - 6 classification accuracy on the test data

The accuracy rises to just under 82 percent. Finally, at around epoch 280 the classification accuracy pretty much stops improving. Contrast this with the earlier graph, where the cost associated to the training data continues to smoothly drop. If we just look at that cost, it appears that our model is still getting "better". But the test accuracy results show the improvement is an illusion. What our network learns after epoch 280 no longer generalizes to the test data. And so it's not useful learning. We say the network is overfitting or overtraining beyond epoch 280.

Another sign of overfitting may be seen in the classification accuracy on the training data (which I didn't include), the accuracy rises all the way up to 100 percent. That is, our network correctly classifies all 1,000 training images! Meanwhile, our test accuracy tops out at just 82.27 percent. So, our network really is learning about peculiarities of the training set, not just recognizing digits in general. It's almost as though our network is merely memorizing the training set, without understanding digits well enough to generalize to the test set.

Overfitting is a major problem in neural networks. This is especially true in modern networks, which often have very large numbers of weights and biases. To train effectively, we need a way of detecting when overfitting is going on, so we don't overtrain. The obvious way to detect overfitting is to use the approach above, keeping track of accuracy on the test data as our network trains. If we see that the accuracy on the test data is no longer improving, then we should stop training. This strategy is called early stopping.

Looking at weather Increasing the amount of training data reduce overfitting or not:

Now, training the network using the full 50 thousands MNIST training image. We'll keep all the other parameters the same (30 hidden neurons, learning rate 0.5, mini-batch size of 10),

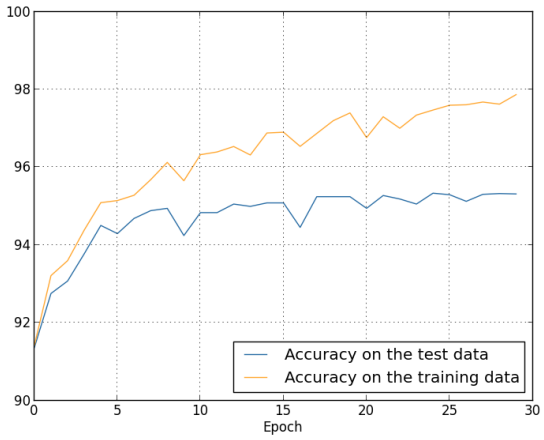


Figure 4 - 7 comparing the classification accuracy on both the training data and the test data.

As you can see, the accuracy on the test and training data remains much closer together than when we were using 1,000 training examples. In particular, the best classification accuracy of 97.86 percent on the training data is only 2.53 percent higher than the 95.33 percent on the test data. That's compared to the 17.73 percent gap we had earlier! Overfitting is still going on, but it's been greatly reduced. Our network is generalizing much better from the training data to the test data.

In general, one of the best ways of reducing overfitting is to increase the size of the training data. With enough training data it is difficult for even a very large network to overfit.

Unfortunately, training data can be expensive or difficult to acquire, so this is not always a practical option.

Regularization:

Another approach for avoiding overfitting, is to reduce the size of our network. However, large networks have the potential to be more powerful than small networks, and so this is an option we'd only adopt reluctantly [1]. We must remember that while simpler functions are more likely to generalize (to have a small gap between training and test error) we must still choose a sufficiently complex hypothesis to achieve low training error. Typically, training error decreases until it asymptotes to the minimum possible error value as model capacity increases (assuming the error measure has a minimum value).[2]

L2 (Weight Decay) Regularization

Fortunately, there are other techniques which can reduce overfitting, even when we have a fixed network and fixed training data. These are known as regularization techniques. In this section I describe one of the most commonly used regularization techniques, a technique sometimes known as weight decay or L2 (or weight decay) regularization. The idea of L2 regularization is to add an extra term to the cost function, a term called the regularization term. $\lambda/(2n)\sum_j w^2$, which is the sum of the squares of all the weights in the network, scaled by a factor λ/2n, where λ>0 is known as the regularization parameter, and n is, as usual, the size of our training set.

Here's the regularized cross-entropy:

$$C = -\frac{1}{n}\sum_{xj}\left[y_j \ln a_j^L + (1-y_j)\ln(1-a_j^L)\right]\frac{\lambda}{2n}\sum_w w^2 \qquad (50)$$

It's worth noting that the regularization term doesn't include the biases. It's also worth noting that the regularization term, can be used with other cost functions as well, such as quadratic-cost function. In general case we write:

$$C = C_0 + \frac{\lambda}{2n}\sum_w w^2 \qquad (51)$$

where C_0 is the original unregularized cost function.

Intuitively, the effect of regularization is to make it so the network prefers to learn small weights. Large weights will only be allowed if they considerably improve the first part of the cost function.

Let's see how regularization changes the performance of our neural network, in the case where we used only 1,000 training images. We'll use a network with 30 hidden neurons, a mini-

batch size of 10, a learning rate of 0.5, and the cross-entropy cost function. However, this time we'll use a regularization parameter of λ=0.1.

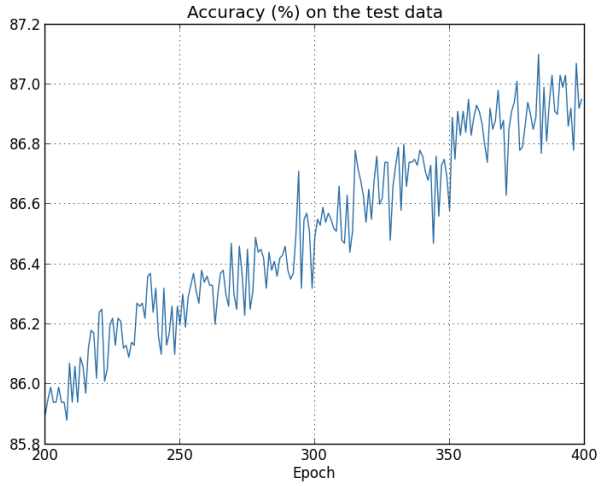


Figure 4 - 8 Classification accuracy on the test data, with only 1000 training image

The cost on the training data decreases over the whole time, much as it did in the earlier, unregularized case. But this time the accuracy on the test_data continues to increase for the entire 400 epochs.

Clearly, the use of regularization has suppressed overfitting. What's more, the accuracy is considerably higher, with a peak classification accuracy of 87.1 percent, compared to the peak of 82.27 percent obtained in the unregularized case. Indeed, we could almost certainly get considerably better results by continuing to train past 400 epochs. It seems that, empirically, regularization is causing our network to generalize better, and considerably reducing the effects of overfitting.

Now, returning to our real model, with 50,000 training sample, we've seen already that overfitting is much less of a problem with the full 50,000 images. To see whether that regularization can improve the situation any further. Using the hyper-parameters, the same as before - 30 epochs, learning rate 0.5, mini-batch size of 10. However, we need to modify the regularization parameter. The reason is because the size n of the training set has changed from n=1,000 to n=50,000, and this changes the weight decay factor $1 - \eta\lambda/n$. If we continued to use λ=0.1 that would mean much less weight decay, and thus much less of a regularization effect. We compensate by changing to λ=5.0.

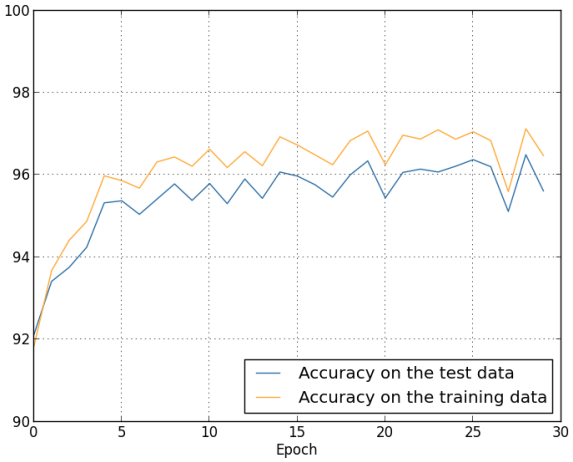


Figure 4 - 9 classification accuracy of the 30 neurons model, trained with the full training set regularized with L2

We can notice two observations, First, our classification accuracy on the test data is up, from 95.49 percent when running unregularized, to 96.49 percent. That's a big improvement. Second, we can see that the gap between results on the training and test data is much narrower than before.

Let's see how regularization changes the performance of the neural network with 100 neurons in the hidden layer, remember that we are using the cross-entropy cost function. Using the same hyper-parameters as in the previous example with 30 neurons.

The final result is a classification accuracy of 97.92 percent on the validation data. That's a big jump from the 30 hidden neuron case. In fact, tuning just a little more, to run for 60 epochs at η=0.1 and λ=5.0 we break the 98 percent barrier, achieving 98.04 percent classification accuracy on the validation data.

Other regularization techniques:

There are many regularization techniques other than L2 regularization. In this section I briefly describe three other approaches to reducing overfitting: L1 regularization, dropout, and artificially increasing the training set size.

L1 regularization:

In this approach we modify the unregularized cost function by adding the sum of the absolute values of the weights:

$$C = C0 + \frac{\lambda}{n}\sum_{w}|w| \qquad (52)$$

Intuitively, this is similar to L2 regularization, penalizing large weights, and tending to make the network prefer small weights. Of course, the L1 regularization term isn't the same as

the L2 regularization term, and so we shouldn't expect to get exactly the same behavior. To see how L1 regularization differ from L2, we should inspect the expression of the partial derivatives of the cost function,

$$\frac{\partial C}{\partial w} = \frac{\partial C_0}{\partial w} + \frac{\lambda}{n} sgn(w) \tag{53}$$

where sgn(w) is the sign of w, that is, +1 if w is positive, and −1 if w is negative. Using this expression, we can easily modify backpropagation to do stochastic gradient descent using L1 regularization. The resulting update rule for an L1 regularized network is:

$$w \to w' = w - \frac{\eta\lambda}{n} sgn(w) - \eta \frac{\partial C_0}{\partial w} \tag{54}$$

Compare that to the update rule for L2 regularization:

$$w \to w' = w \left(1 - \frac{\eta\lambda}{n}\right) - \frac{\eta}{m} \frac{\partial C_x}{\partial w} \tag{55}$$

In both expressions the effect of regularization is to shrink the weights. This accords with our intuition that both kinds of regularization penalize large weights. But the way the weights shrink is different. In L1 regularization, the weights shrink by a constant amount toward 0. In L2 regularization, the weights shrink by an amount which is proportional to w. And so, when a particular weight has a large magnitude, |w|, L1 regularization shrinks the weight much less than L2 regularization does. By contrast, when |w| is small, L1 regularization shrinks the weight much more than L2 regularization. The net result is that L1 regularization tends to concentrate the weight of the network in a relatively small number of high-importance connections, while the other weights are driven toward zero.

Dropout:

Dropout is by far the most interesting technique for regularization and it is a radically different than the other regularization techniques discussed so far. Unlike L1 and L2 regularization, dropout doesn't rely on modifying the cost function. Instead, in dropout we modify the network itself. It works by disabling a random subset of the network's neuron throughout the learning process for each mini-batch.

Suppose we have a training input x and corresponding desired output y. Ordinarily, we'd train by forward-propagating x through the network, and then backpropagating to determine the contribution to the gradient. With dropout, this process is modified. We start by randomly (and temporarily) deleting "dropping out" half the hidden neurons in the network, while leaving the input and output neurons untouched - some networks drop more than halve the neuron still obtain consistently better performance -. We forward-propagate the input x through the modified network, and then backpropagate the result, also through the modified network. [1]. The neurons

which are “dropped out” in this way do not contribute to the forward pass and do not participate in back-propagation. So, every time an input is presented, the neural network samples a different architecture, but all these architectures share weights. [22]

At test time, when we actually run the full network that means that twice as many hidden neurons will be active. Of course, those weights and biases will have been learned under conditions in which half the hidden neurons were dropped out - Assuming we used 50% drop ration of neurons during the training. To compensate for that, we use all the neurons but multiply their outputs by 0.5. [22,20]

The two heuristics for why dropout might help combat overfitting

There are two distinct heuristics for why dorpout might combat the problem of overfitting and improve performance.

The first one, which is seemingly the most prevailing, has been discussed in many papers, and machine learning books. Which is the model combination view:

Model combination nearly always improves the performance of machine learning methods. With large neural networks, however, the obvious idea of averaging the outputs of many separately trained nets is prohibitively expensive, for big neural networks that already take several days to train. Combining several models is most helpful when the individual models are different from each other and in order to make neural net models different, they should either have different architectures or be trained on different data. Training many different architectures is hard because finding optimal hyperparameters for each architecture is a daunting task and training each large network requires a lot of computation. Moreover, large networks normally require large amounts of training data and there may not be enough data available to train different networks on different subsets of the data. Even if one was able to train many different large networks, using them all at test time is infeasible in applications where it is important to respond quickly. [20,22]

Dropout is a technique that addresses both these issues. It is a very efficient version of model combination that only costs about a factor of two during training. It prevents overfitting and provides a way of approximately combining exponentially many different neural network architectures efficiently [20,22].

•The second heuristic, promotes the idea that the absence of different neurons throughout the learning process, forces the network to learn redundant information from different paths, as it was described in [22]: ‘This technique reduces complex co-adaptations of neurons, since a neuron cannot rely on the presence of particular other neurons. It is, therefore, forced to learn more robust features that are useful in conjunction with many different random subsets of the other neurons’.

This prevents complex co-adaptations in which a feature detector is only helpful in the context of several other specific feature detectors. Instead, each neuron learns to detect a feature that is generally helpful for producing the correct answer given the combinatorically large variety of internal contexts in which it must operate.[21]

Using dropout with other regularization techniques

When using dropout as a regularization technique along with L2 (weight decay), the expression for the L2 changes a bit, since we don't have all the neurons active in both the forward or the backward pass. We use the standard, stochastic gradient descent procedure for training the dropout neural networks on mini-batches of training cases, but we modify the penalty term that is normally used to prevent the weights from growing too large. Instead of penalizing the squared length (L2 norm) of the whole weight vector, we set an upper bound on the L2 norm of the incoming weight vector for each individual hidden unit.[21]

The original paper introducing the technique "Improving neural networks by preventing co-adaptation of feature detectors" by Geoffrey Hinton, Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov (2012), applied it to many different tasks. For us, it's of particular interest that they applied dropout to MNIST digits classification, using a vanilla feedforward neural network along lines similar to those we've been considering. The paper noted that the best result anyone had achieved up to that point using such an architecture was 98.4 percent classification accuracy on the test set. They improved that to 98.7 percent accuracy using a combination of dropout and a modified form of L2 regularization. Similarly impressive results have been obtained for many other tasks, including problems in image and speech recognition, and natural language processing. Dropout has been especially useful in training large, deep networks, where the problem of overfitting is often acute. [1]

Details for dropout finetuning

Apart from training a neural network starting from random weights, dropout can also be used to finetune pretrained models. We found that finetuning a model using dropout with a small learning rate can give much better performance than standard backpropagation finetuning. [21]

The researchers behind the [21] paper, run an experiment to fine tune a pre-trained 'Deep Belief Nets', and 'Deep Boltzmann Machines' model using dropout, where they used Dropout rate: 50% for hidden units and 20% for visible units, on both model. While standard back propagation gave about 118 errors, dropout decreased the errors to about 92, for Deep Belief Nets model. they were able to get a mean of about 79 errors with dropout whereas usual finetuning gives about 94 errors, with Deep Boltzmann Machines model. In both cases the dropout technique improved the classification accuracy dramatically.

Artificially expanding the training data:

As we saw before, increasing the size of the training data set, greatly reduce overfitting and improved the classification accuracy. Unfortunately, it can be expensive, and so is not

always possible in practice. However, there's another idea which can work nearly as well, and that's to artificially expand the training data. [1,22, 23]

Data Augmentation (expansion): is the easiest and the most common regularization scheme that artificially inflates the data-set by using label preserving transformations to add more invariant examples [23,22]. Recently there has been extensive use of generic data augmentation to improve Convolutional Neural Network (CNN) task performance. [23]

Generic DA is a set of computationally inexpensive methods, previously used to reduce overfitting in training a CNN for the ImageNet Large-Scale Visual Recognition Challenge (ILSVRC), and achieved state-of the-art results at the time. This augmentation scheme consists of Geometric and Photometric transformations [23]. Generally these transformations, allow transformed images to be produced from the original images with very little computation, so the transformed images do not need to be stored on disk. And they can be run on the CPU while the network is training on the GPU, So these data augmentation schemes are, in effect, computationally free.[22] Some of the transformations used for image are: rotation, slanting, horizontal reflection (reflection across the y-axis) (horizontal reflection doesn't apply for text data). Affine distortions greatly improved our results on the MNIST database. However, our best results were obtained when we considered elastic deformations.[19]

Looking at some of the result in [19] paper. One of the NN architectures they considered was along similar lines to what we've been using, a feedforward network with 800 hidden neurons and using the cross-entropy cost function. Running the network with the standard MNIST training data they achieved a classification accuracy of 98.4 percent on their test set. But then they expanded the training data, using rotations, translations, and skewing. By training on the expanded data set they increased their network's accuracy to 98.9 percent. They also experimented with what they called "elastic distortions" a special type of image distortion intended to emulate the random oscillations found in hand muscles. By using the elastic distortions to expand the data they achieved an even higher accuracy, 99.3 percent. Effectively, they were broadening the experience of their network by exposing it to the sort of variations that are found in real handwriting.

Conclusion:

NN models tends to perform best when their size fit the training data, too flexible model (model with high capacity aka high variance) can easily overfit, which cause the model's performance to deteriorate for generalizing to new unseen data. Regularization techniques such as L1, L2 and dropout can easily be implemented to prevent this problem. Data augmentation is another effective technique for combating this problem as well.

Exploiting the spatial structure of unstructured data

Introduction: Structured vs Unstructured data

Structured data is data that has been predefined and formatted to a set structure before being placed in data storage, which is often referred to as schema-on-write. The best example of structured data is the relational database: the data has been formatted into precisely defined fields, such as credit card numbers or address, in order to be easily queried with SQL. Unstructured data on other hand, is data stored in its native format and not processed, which is known as schema-on-read. It comes in a myriad of file formats, audio (speech, environmental, music), images (digital cameras, medical imaging, satellite imagery), Industrial and IoT sensor data, email, social media posts and chats.

Now, the definition we use when we talk about data for Machine learning, is relatively similar, except that we usually make the distinction between the two classes of data, by saying structured data is one that is stored in tabular like format (rows and columns), and reordering it based on different feature doesn't affect its meaning. Unstructured data (images, audio) on the other hand, is one that loses its meaning when it gets reordered.

Why making this distinction?

Historically it has been much harder for computers to make sense of unstructured data compared to structured data. In fact, we as human being are very good at understanding unstructured data, I am talking about acoustic and visual ques (audio and images), and so one of the most exciting things about the rise of neural networks is that thanks to neural networks computers are now much better at interpreting unstructured data as well. And this creates opportunities for many new exciting applications that use speech recognition image recognition natural language processing, much more than was possible even just ten years ago. And as a personal statement I think we, human have a natural empathy to understanding unstructured data, you might hear about NNs successes on unstructured data more in the media because it's just cool when something we built (computer) learn to interpret and recognize the content of these types of unstructured data that we as human naturally good at. When a computer recognizes a cat, we all like that, because we all know what that means.

Up to this point we have only been discussing fully connected feed-forward neural networks, and the model we have developed in the previous chapter did very well on the task of handwritten image classification, since we've obtained a classification accuracy up to 98 %, with a relatively simple model, and that's of course after employing some optimization (better cost function and better weight initialization approach), and regularization techniques. And we can summarize its working machinery in few lines: A vector is received as input and is multiplied with a matrix to produce an output, to which a bias vector is usually added - this first part is known as affine transformation - before passing the result through a non-linearity. This is applicable to any type of input, not just to image data, a sound clip or an unordered collection of

features: whatever their dimensionality, their representation can always be FLATTENED into a vector before the transformation [30]

Images, sound clips and many other similar kinds of data have an intrinsic structure, despite that they belong to unstructured data class. More formally, they share these important properties:

- They are stored as multi-dimensional arrays.
- They feature one or more axes for which ordering matters (e.g., width and height axes for an image, time axis for a sound clip).
- One axis, called the channel axis, is used to access different views of the data (e.g., the red, green and blue channels of a color image, or the left and right channels of a stereo audio track).

Convolutional Neural Networks (CNNs)

It turns out that fully connected feed-forward networks are not the best choice for these types of unstructured data that has an implicit structure, since they do not take into account the implicit spatial structure present in the input data. In fact, all the axes are treated in the same way and the topological information is not exploited. For instance, they treat input pixels which are far apart and close together on exactly the same footing. Such concepts of spatial structure must instead be inferred from the training data. What if, instead of starting with a NN architecture that is tabula rasa (agnostic to the spatial structure), we start with a NN architecture that take into account the spatial structure. Convolutional Neural Networks, use a special architecture which is particularly well-adapted for tasks like pattern recognition and classification. You can think of CNN, as being a NN that has some type of specialization for being able to pick up or detect pattern and make sense of them [1-3, 30].

Convolutional Neural Networks or CNNs, are a specialized kind of neural network for processing data that has a known, grid-like topology. Examples include time-series data, which can be thought of as a 1D grid taking samples at regular time intervals, and image data, which can be thought of as a 2D grid of pixels [2]. Although CNNs have been used as early as the nineties to solve character recognition tasks (Le Cun et al., 1997), their current widespread application is due to much more recent work, when a deep CNN was used to beat state-of-the-art in the ImageNet image classification challenge (Krizhevsky et al., 2012) [30].

The Convolution Operation:

As the name “convolutional neural network” implies CNN employs the convolution mathematical operation - In fact, they use a closely related operation known as Cross-correlation but still call it convolution as we will see shortly. Convolution is a specialized kind of linear

operation. Convolutional networks are simply neural networks that use convolution in place of general matrix multiplication in at least one of their layers [2]

A convolution is an operation on two functions of a real-valued argument, it is defined as the integral of the product of the two functions after one is reversed and shifted. The integral is evaluated for all values of shift. It is often notated with an asterisk ($*$) operator, where:

$$s(t) = (f * g)(t) = \int_{-\infty}^{\infty} f(\tau)g(t-\tau)d\tau = \int_{-\infty}^{\infty} f(t-\tau)g(\tau)d\tau \qquad (56)$$

Usually, when we work with data on a computer, data will be discretized. So, the discrete version of the convolution operation:

$$s(n) = (f * g)[n] = \sum_{m=-\infty}^{\infty} f[m]g[n-m] = \sum_{m=-\infty}^{\infty} f[n-m]g[m] \qquad (57)$$

Note: In CNN terminology, the first argument (in this example, the function f) to the convolution is often referred to as the input and the second argument (in this example, the function g) as the filter (kernel). The output is sometimes referred to as the feature map.

Algebraically, the convolution is the sum of Hadamard product (aka the element-wise product) of the kernel parameters with the corresponding values of the input.

From a practical perspective, convolution can be seen as the measure of overlap between two signals as one slides over the other as demonstrated in (fig below). The output can be thought as the blend of the two signals.

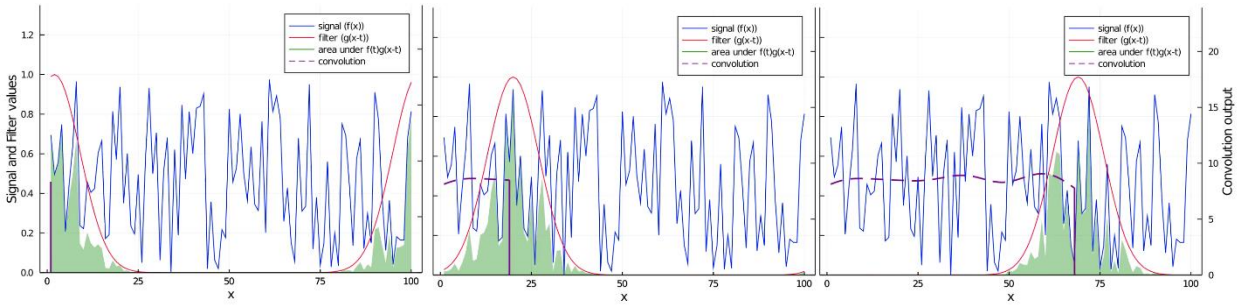


Figure 5 - 1 One-dimensional convolution of a Gaussian filter with a random signal

In the above figure, the output of the convolution operation doesn't resemble the original signal. In fact, it doesn't look anything like the original signal, that's because we applied a relatively wide Gaussian filter, so that the input signal got heavily filtered. Because, the signal above is a one-dimensional time series data, filtering in the situation means filtering out the high frequencies, but if the input was two-dimensions like image data, the effect would be a blurring effect, if the filter was two-dimensional Gaussian filter.

Cross-correlation

Similar operation to the convolution is Cross-correlation (denoted by a star $\star$), where the only difference is, one of the functions is first flipped about the y-axis before performing the standard convolution, so that the cross-correlation of f and g is f(x)*g(-x). Note, when using cross-correlation we lose the commutativity property, that's:

$$s(t) = (f \star g)(t) = \int_{-\infty}^{\infty} f(\tau)g(t+\tau)d\tau \neq \int_{-\infty}^{\infty} f(t+\tau)g(\tau)d\tau \quad (58)$$

Note: the convolution operation that I've just demonstrated above is heavily used in many fields such: digital signal processing, image processing, engineering, physics, computer vision and differential equations just to name a few. The main difference between the applications of the convolution in these fields and in machine learning is that, the filters (or kernels) parameters are learned by the network during the training phase.

2D Convolution

In machine learning applications, the input is usually a multidimensional array of data and the kernel is usually a multidimensional array of parameters that are adapted by the learning algorithm. And we often use convolutions over more than one axis at a time. For example, if we use a two-dimensional image I as our input, we probably also want to use a two-dimensional kernel

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_m I(m,n)I(i-m, j-n) \quad (59)$$

Convolution in any number of dimensions is commutative just like in one-dimension, meaning we can equivalently write: $S(i,j) = (I * K)(i,j) = \sum_m \sum_m I(i-m, j-n)K(m,n)$

While the commutative Property is mathematically handy, it is usually not an important property for a neural network implementation. Instead, most NN libraries implement the cross-correlation and call it convolution, which is the same as convolution but without flipping the kernel.

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_m I(i+m, j+n)K(m,n) \quad (60)$$

CNNs Core Components:

The CNN architecture is somehow reminiscent to the organization of the Visual Cortex. Individual neurons respond to stimuli, only in a restricted region of the visual field known as the Receptive Field. A collection of such fields overlap to cover the entire visual area. Here are the fundamental components in any Convolutional Neural Network:

- Local Receptive Fields: a small region of the input image (a little window on the input), covered by the kernel. The local receptive field (kernel, or filter) moves to the right with a certain Stride Value till it parses the complete width. Moving on, it hops down to the

beginning (left) of the image with the same Stride Value and repeats the process until the entire image is traversed (Fig.).

- Shared Weights, Shared Biases: all the neurons in the feature map (output of the convolutional layer), will use the same collection of weights and biases (same filter). This means that all the neurons in the first hidden layer detect exactly the same feature*, just at different locations in the input image.

Note: think of the feature detected by a neuron as the kind of input pattern that will cause the neuron to activate: it might be an edge or an arch in the image, or some other type of shape.

For this reason, we sometimes call the map from the input layer to the hidden layer a feature map. We call the weights defining the feature map the shared weights. And we call the bias defining the feature map in this way the shared bias. The shared weights and bias are often said to define a kernel or filter.

- Filter (aka Kernel): is the matrix of the weights + bias, that will be applied to the local receptive field, to produce the output of the convolutional layer.

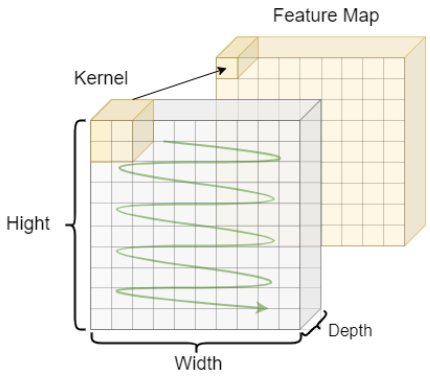


Figure 5 - 2 Standard convolution

Note: in the above figure, I am only showing one filter (producing one feature map), in fact all CNNs use more than one filter in each layer, where each filter learns different feature, the more filters the more features the CNN can learn.

Now, when performing the convolution operation there are several parameters to choose from, that affect the output of the operation, theses parameters are:

- **Kernel Size:** larger kernels reduce the size of the input.
- **Stride**: steps to move the kernel when convolving, large stride yields smaller feature map

- **Padding**: adds columns and rows of zeroes to the original input before convolving, to keep the spatial sizes constant after convolution, and make a more use of the pixels on the edges, there are two types of padding:
    - **Valid**: no padding
    - **Same**: add as many extra pixels to the sides of the input to produce a feature map with the same size as the input.

Note: These parameters are an extra layer of complexity for CNN. Yes, flexibility is often a good thing to have, but, since you need to make a definitive choice for theses parameters and don't always have a good idea of where to start. Having these additional parameters on top of the typical hyper-parameters (model size, type of activation function, cost function, batch size, learning rate ...) are what makes CNNs hard to work with.

Now, I am not going to go over how to pick theses parameters since they are usually task dependent, and the handwritten digits doesn't greatly exhibit the problem of translation invariance because all the digits are centered, and size-normalized. But just like we did with the other hyper-parameters, we can always use the validation set to land on values that yield best results.

Here's how to compute the size of the feature map:

$$Height = \frac{Height(input) + 2 \times padding - Height(kernel)}{stride} + 1$$

$$Width = \frac{Width(input) + 2 \times padding - Width(kernel)}{stride} + 1$$

Note: A side effect of convolution is reducing the size of input, basically the size of the feature map yielded by the convolution operation is usually less than the input, most often this effect is not problem, because we want to have a smaller and efficient representation of the input any way, but you can always prevent that by using same padding.

Pooling

In addition to discrete convolutions themselves, pooling operations make up another important building block in CNNs. Pooling operations reduce the size of feature maps by using some function to summarize subregions, such as taking the average or the maximum value. It works by sliding a kernel across the input and feeding the content of the window to a pooling function, much like the convolution, in some sense.

Pooling Layers: usually used immediately after convolutional layers to simplify the information in the output from the convolutional layer. Normally, there is pooling layer for each

feature map that prepares a condensed feature map, for instance each unit in the pooling layer, may summarize a region of say (2×2) in the previous layer [1].

There are two types of Pooling: Max Pooling and Average Pooling (aka L2 pooling). Max Pooling returns the maximum value from the portion of the image covered by the Kernel. On the other hand, Average Pooling returns the average of all the values (square root of the sum, to be prices) from the portion of the image covered by the Kernel.

We can think of max-pooling as a way for the network to ask whether a given feature is found anywhere in a region of the image. It then throws away the exact positional information. The intuition is that once a feature has been found, its exact location isn't as important as its rough location relative to other features [1].

Max Pooling can also be thought as performing Noise Suppressing along with dimensionality reduction. Where, it discards the noise (activations) of nearby neurons and just returns the most dominant feature in the kernel. On the other hand, Average Pooling simply performs dimensionality reduction. Thus, Max Pooling is the preferred option for pooling.

The Convolutional and the Pooling layer are usually counted as a single layer in CNN. Depending on the size and the complexities of the input, the number of such layers may need to increase.

Putting it all together

Yes, convolutional layers enable the model to extract features present in the input, but to make the final prediction, the model needs to put the pieces of the puzzles together, and for that, one or more fully connected layers are usually used. Where the output of the final convolutional layer is flattened before being fed to those fully connected layers.

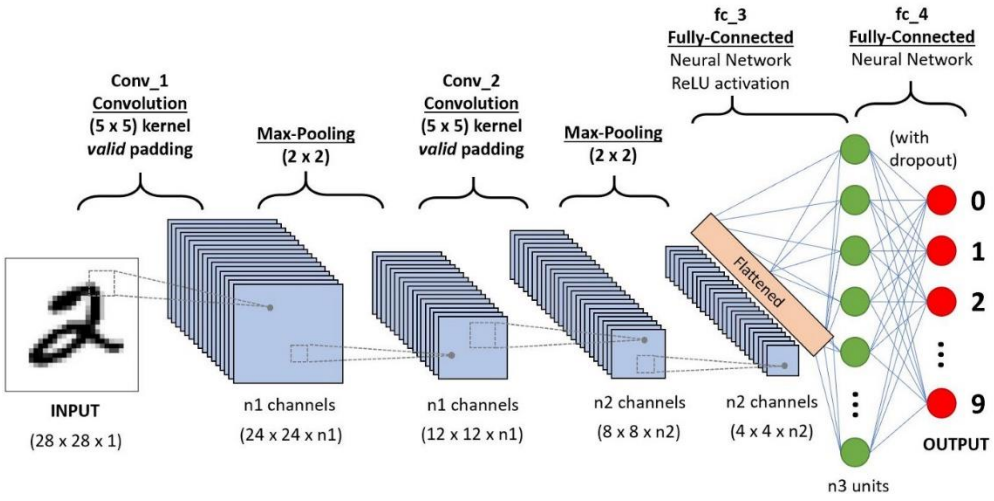


Figure 5 - 3 A typical CNN architecture for classifying Handwritten digits

Using CNN for classifying handwritten digits

Now, it's not fair or even reasonable to contrast Dense NN to CNN, since they both have completely different architecture, I mean what size and hyper-parameters should you choose for an objective comparison of the performance difference? but just for the sake of exhibiting the superiority of CNN on unstructured, I've gone ahead and plot the performance of two models one Dense NN with a single hidden layer with 100 neurons, and the second is a CNN model with two convolutional layer ($20\times5\times5$ and $10\times3\times3$ kernel respectively) followed by a max pooling layer each, finally a softmax output layer.

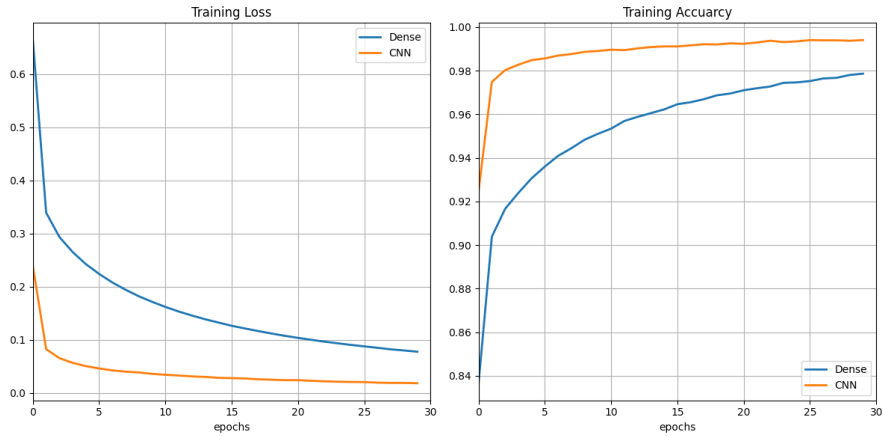


Figure 5 - 4 CNN vs Dense NN Loss and classification accuracy on the training data

As you can see in (Figure 5 - 4), CNN performance is definitely superior to Dense NNs, it yields both lower error and better classification accuracy on both training data as well as the validation data, converging much faster. The final classification accuracy on the test data after 30 epochs using CNN is 98.77%, with an error (loss) value equal to 0.0409. Using Dense NNs on the other hand, we got an accuracy of 96.93 % with error (loss) value equal to 0.0977.

Back-propagation in CNN

This is a bit of an overkill to derive the full formulas for the back-propagation algorithm in CCN, but since the aim of this research to demystify the theoretical details of the Neural networks, I am going to include it anyway.

Note: this derivation assumes that the cost function is MSE (Quadratic cost) function.

Forward-Pass

We'll call:

- $a^0_{j,k}, 0 \leq j,k \leq 27$, the input activations;

- $w^1_{l,m}, 0 \leq l, m \leq 4$, the shared weights for the convolutional layer;
- b^1, the shared bias for the convolutional layer;
- $z^1_{j,k} 0 \leq j, k \leq 23$, the weighted input to neuron (j, k) (line j, column k) in the conv layer:

$$z^1_{j,k} = b^1 + \sum_{l=0}^{4} \sum_{m=0}^{4} w^1_{l,m} a^0_{j+l,k+m}$$

$a^1_{j,k}, 0 \leq j, k \leq 23$, the activation of neuron (j, k) in the convolutional layer:

$$a^1_{j,k} = \sigma\left(z^1_{j,k}\right)$$

$a^2_{j,k}, 0 \leq j, k \leq 11$, the activation of neuron (j, k) in the max-pooling layer:

$$a^2_{j,k} = max\left(a^1_{2j,2k}, a^1_{2j,2k+1}, a^1_{2j+1,2k}, a^1_{2j+1,2k+1}\right)$$

So, neuron $(\boldsymbol{j}, \boldsymbol{k})$ in the convolutional layer will contribute to the computation of the max for neuron $\left(\left\lfloor \frac{j}{2} \right\rfloor, \left\lfloor \frac{k}{2} \right\rfloor\right)$.

Note: the symbol $\left\lfloor \frac{\mathbf{j}}{\mathbf{2}} \right\rfloor$, indicate the floor function of $\frac{\mathbf{j}}{\mathbf{2}}$.

Note that the max-pooling layer doesn't have any weights, biases, or weighted inputs!

$\boldsymbol{w^3_{l;j,k}}\ \mathbf{0} \leq \boldsymbol{j, k} \leq \mathbf{11}, \mathbf{0} \leq \boldsymbol{l} \leq \mathbf{9}$, the weight of the connection between neuron $(\boldsymbol{j}, \boldsymbol{k})$ in the max-pooling layer and neuron $\boldsymbol{l}$ in the output layer;

$\boldsymbol{b^3_l}\mathbf{0} \leq \boldsymbol{l} \leq 9$, the bias of neuron $\boldsymbol{l}$ in the output layer;

$\boldsymbol{z^3_l}\mathbf{0} \leq \boldsymbol{l} \leq \mathbf{9}$, the weighted input of neuron $\boldsymbol{l}$ in the output layer:

$$z^3_l = b^3_l + \sum_{0 \leq j \leq 11} \sum_{0 \leq k \leq 11} w^3_{l;j,k} a^2_{j,k}$$

- $a^3_l 0 \leq l \leq 9$, the output activation of neuron l in the output layer:

$$a^3_l = \sigma(z^3_l)$$

Back-propagation four equations

Now for comparison, here are equations BP1 - BP4 for regular fully connected networks:

***Note**: **BP2** is written using conventional matrix multiplication, unlike before where we used Hadamard product $\boldsymbol{BP2} = \delta^l_j = ((w^{l+1})^T\ \delta^{l+1}) \odot\ \sigma'\left(z^l_j\right)$:*

BP1: $$\delta_j^L = \frac{\partial C}{\partial z_j^L} = \frac{\partial C}{\partial a_j^L}\frac{\partial a_j^L}{\partial z_j^L} = \frac{\partial C}{\partial a_j^L}\sigma'(z_j^L)$$

BP2: $$\delta_j^l = \frac{\partial C}{\partial z_j^l} = \sum_k \frac{\partial C}{\partial z_k^{l+1}}\frac{\partial z_k^{l+1}}{\partial a_j^l}\frac{\partial a_j^l}{\partial z_j^l} = \sum_k \delta_k^{l+1} w_{kj}^{l+1}\sigma'(z_j^l)$$

BP3: $$\frac{\partial C}{\partial b_j^l} = \frac{\partial C}{\partial z_j^l}\frac{\partial z_j^l}{\partial b_j^l} = \delta_j^l \times 1$$

BP4: $$\frac{\partial C}{\partial w_{jk}^l} = \frac{\partial C}{\partial z_j^l}\frac{\partial z_j^l}{\partial w_{jk}^l} = \delta_j^l a_k^{l-1}$$

Let's look at each equation in turn, with our new network architecture.

- **BP1:** The last layer following the previous network architecture, we see that the derivation of BP1 remains correct. Therefore, BP1 doesn't change.
- **BP2:** since the max-pooling layer doesn't have any weighted inputs, we'll just have to compute $\delta_{j,k}^1$.

$$\delta_{j,k}^1 = \frac{\partial C}{\partial z_{j,k}^1}$$

$$= \sum_{l=0}^{9} \frac{\partial C}{\partial z_l^3}\frac{\partial z_l^3}{\partial z_{j,k}^1}$$

$$= \sum_{l=0}^{9} \delta_l^3 \frac{\partial z_l^3}{\partial a_{j',k'}^2}\frac{\partial a_{j',k'}^2}{\partial z_{j,k}^1}$$

with $j' = \left\lfloor \frac{j}{2} \right\rfloor, \mathrm{k}' = \left\lfloor \frac{k}{2} \right\rfloor$

$a_{j',k'}^2$ being the only activation in the max-pooling layer affected by $z_{j,k}^1$.

$$= \sum_{l=0}^{9} \delta_l^3 w_{l;j',k'}^3 \frac{\partial a_{j',k'}^2}{\partial z_{j,k}^1}$$

$$= \sum_{l=0}^{9} \delta_l^3 w_{l;j',k'}^3 \frac{\partial a_{j',k'}^2}{\partial a_{j,k}^1}\frac{\partial a_{j,k}^1}{\partial z_{j,k}^1}$$

$$= \sum_{l=0}^{9} \delta_l^3 w_{l;j',k'}^3 \frac{\partial a_{j',k'}^2}{\partial a_{j,k}^1} \sigma'\left(z_{j,k}^1\right)$$

Now since $a_{j',k'}^2 = \max\left(a_{2j',2k'}^1, a_{2j',2k'+1}^1, a_{2j'+1,2k'}^1, a_{2j'+1,2k'+1}^1\right)$ and we're talking about infinitesimal changes, we have:

$$\frac{\partial a_{j',k'}^2}{\partial a_{j,k}^1} = \begin{cases} 0, & if\ a_{\{j,k\}}^1 \neq max\left(a_{\{2j',2k'\}}^1, a_{\{2j',2k'+1\}}^1, a_{\{2j'+1,2k'\}}^1, a_{\{2j'+1,2k'+1\}}^1\right) \\ 1, & if\ a_{\{j,k\}}^1 = max\left(a_{\{2j',2k'\}}^1, a_{\{2j',2k'+1\}}^1, a_{\{2j'+1,2k'\}}^1, a_{\{2j'+1,2k'+1\}}^1\right) \end{cases}$$

This is because $a_{j,k}^1$ only affects $a_{j',k'}^2$ if $a_{j,k}^1$ is the maximum activation in its local pooling field. In this case, we have $a_{j',k'}^2 = a_{j,k}^1$, so $\frac{\partial a_{j',k'}^2}{\partial a_{j,k}^1}$ =1.

And so to conclude the derivation of our new BP2:

$$\delta_{j,k}^1 = \begin{cases} 0, & if\ a_{\{j,k\}}^1 \neq max\left(a_{\{2j',2k'\}}^1, a_{\{2j',2k'+1\}}^1, a_{\{2j'+1,2k'\}}^1, a_{\{2j'+1,2k'+1\}}^1\right) \\ \sum_{l=0}^{9} \delta_l^3 w_{l;j',k'}^3 \sigma'\left(z_{j,k}^1\right), & if\ a_{\{j,k\}}^1 = max\left(a_{\{2j',2k'\}}^1, a_{\{2j',2k'+1\}}^1, a_{\{2j'+1,2k'\}}^1, a_{\{2j'+1,2k'+1\}}^1\right) \end{cases}$$

- **BP3**: we consider two cases:
  - $\frac{\partial C}{\partial b_l^3} = \delta_l^3$ as the third layer respects the previous architecture (the derivation still works);
  - $\frac{\partial C}{\partial b^1}$. This one is different, since the bias b^1 is shared for all neurons in the convolutional layer.

We have:

$$\frac{\partial C}{\partial b^1} = \sum_{0 \le j,k \le 23} \frac{\partial C}{\partial z_{j,k}^1} \frac{\partial z_{j,k}^1}{\partial b^1}$$

$$= \sum_{0 \le j,k \le 23} \delta_{j,k}^1 \frac{\partial z_{j,k}^1}{\partial b^1}$$

$$= \sum_{0 \le j,k \le 23} \delta_{j,k}^1 \qquad \text{as } z_{j,k}^1 = b^1 + \sum_{l=0}^{4} \sum_{m=0}^{4} w_{l,m}^1 a_{j+l,k+m}^0$$

- BP4:

- $\frac{\partial C}{\partial w^3_{l;j,k}} = a^2_{j,k}\delta^3_l$ since, again, the derivation still works for the third layer;
- $\frac{\partial C}{\partial w^1_{l,m}}, 0 \le l,m \le 4$. These 25 weights are shared, and each of them is used in the computation of the weighted input of each neuron in the convolutional layer:

$$\frac{\partial C}{\partial w^1_{l,m}} = \sum_{0\le j,k\le 23} \frac{\partial C}{\partial z^1_{j,k}} \frac{\partial z^1_{j,k}}{\partial w^1_{l,m}}$$

$$= \sum_{0\le j,k\le 23} \delta^1_{j,k} \frac{\partial z^1_{j,k}}{\partial w^1_{l,m}}$$

$$= \sum_{0\le j,k\le 23} \delta^1_{j,k} a^0_{j+l,k+m} \qquad \text{as } z^1_{j,k} = b^1 + \sum_{l=0}^{4}\sum_{m=0}^{4} w^1_{l,m} a^0_{j+l,k+m}$$

We can also express this formula using the convolution operation (cross-correlation, really)

$$= \delta^1 \star a^0 \quad \text{as } z^1_{j,k} = b^1 + w^1 \star a^0$$

Depth-wise Separable Convolution - A faster convolution!

The standard convolution operation is relatively slow to perform, thus training and testing CNN model is much slower than fully connected NN one. However, we can speed this up with an alternative method Depth wise Separable Convolution (DSC).

In this next section, I will discuss the computational cost of the convolution operation for a single training input, where I will compute the number of multiplications performed (which is a direct measure of performance). Then I will contrast it with the number of multiplications in a Fully connected NN. After that, I will discuss an alternative less expensive type of convolution, known as Depth wise Separable Convolution (DSC).

The Standard Convolution.

let's first very quickly go over the basics of the standard convolution on an input volume. Consider an input volume F, of shape DF×DF×M where DF is the width and height of the input volume and M is the number of input channels, if a color image was an input, then M would be equal to 3 for the RGB channels. We apply convolution on a kernel K of shape DK×DK×M this will give us an output of shape DG×DG×1, if we apply n such kernels on the input then we get an output volume G of shape DG×DG×N (Fig below).

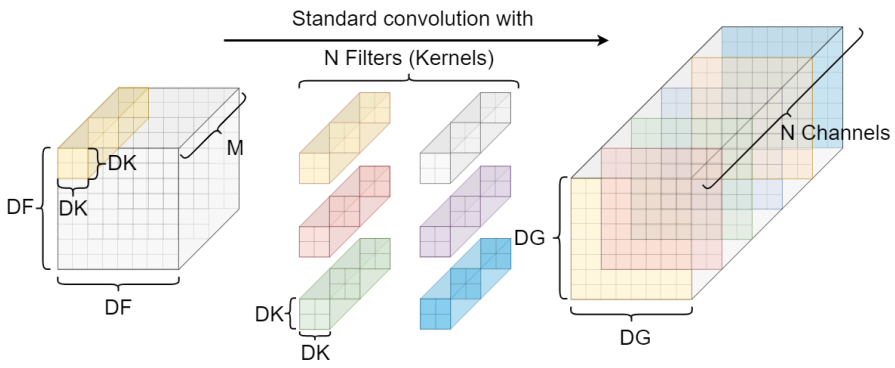


Figure 5 - 5 Standard convolution with N filters

The computational cost of the standard convolution

Now, we should be concerned with the cost of this convolution operation. Yes, the convolutional layers unleash a superpower to Neural networks, but as it is the case for almost everything, there's always a BUT, so, let's take a look at that. We can measure the computation required for convolution by taking a look at the number of multiplications required. But, why is that? it's because multiplication is an expensive operation relative to addition, so let's determine the number of multiplications for one convolution operation. The number of multiplications is the number of elements in that kernel so that would be $DK \times DK \times M$ multiplications, but we slide this kernel over the input we perform DG (size of the feature map) convolutions along the width, and DG convolutions along the height and hence $DG \times DG$ convolutions over all, so the number of multiplications in the convolution of one kernel over the entire input F is $DG^2 \times DK^2 \times M$ now this is for just one kernel, but if we have N such kernels which makes the absolute total number of multiplications become $DG^2 \times DK^2 \times M \times N$ multiplications.

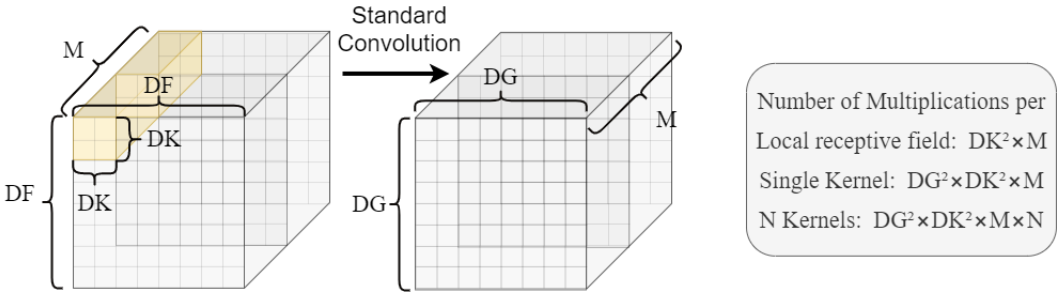


Figure 5 - 6 Number of multiplications in the standard convolution

Depth wise separable convolution operation.

Let's now take a look at Depth wise Separable Convolutions (DSC). In standard convolution the application of filters across all input channels and the combination of these values are done in a single step. DSC on the other hand breaks us down into two parts, the first is Depth wise Convolution (DC) that is it performs the filtering stage, and then Point wise

Convolution (PC) which performs the combining stage. Let's get into some details here, depth wise convolution applies convolution to a single input channel at a time, this is different from the standard convolution that applies convolution to all channels, let us take the same input volume F to understand this process F has a shape DF×DF×M, where DF is the width and height of the input volume and M is the number of input channels, like I mentioned before.

For depth wise convolution we use filters or kernels K of shape DK×DK×1, here DK is the width and height of the square kernel and it has a depth of 1 because this convolution is only applied to a channel unlike standard convolution which is applied throughout the entire depth, and since we apply one kernel to a single input channel, we require M such DK×DK×1 kernels over the entire input volume F.

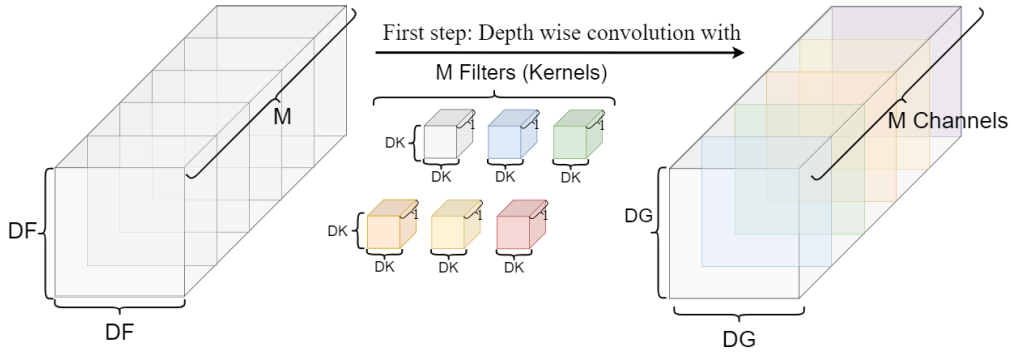


Figure 5 - 7 First step of Depth wise Separable Convolution

For each of these M convolutions we end up with an output DG×DG×1 in shape, now stacking these outputs together we have an output volume of G which is of shape DG×DG×M, this is the end of the first phase that is the end of depth wise convolution.

Now this is succeeded by point wise convolution point wise convolution involves performing the linear combination of each of these layers, here the input is the volume of shape DG×DG×M, the filter KPC has a shape 1×1×M, this is basically a 1×1 convolution operation over all M layers. The output will thus have the same input width and height as the input DG×DG for each filter assuming that we want to use N such filters the output volume becomes DG×DG×N.

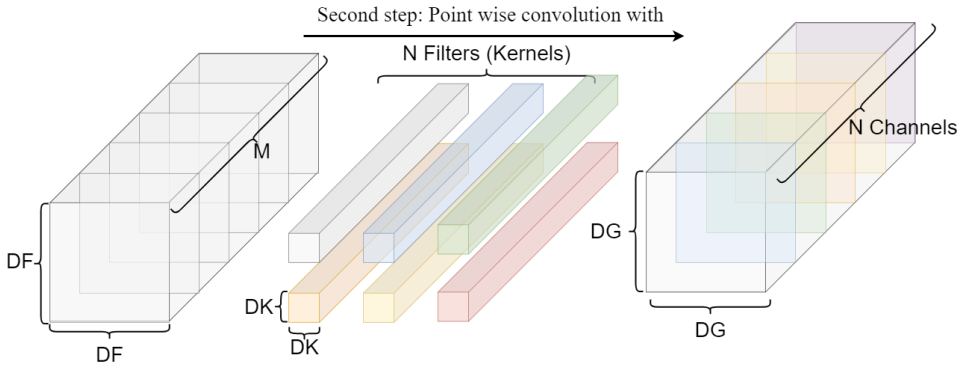


Figure 5 - 8 Second step of Depth wise Separable Convolution

Depth wise separable convolution complexity.

Now let's take a look at the complexity of this convolution, we can split this into two parts as we have two phases, first we compute the number of multiplications in depth wise convolution, so here the kernels have a shape DK×DK×1, so the number of multiplications on one convolution operation is all DK×DK, or DK^2, when applied over the entire input channel this convolution is performed DG×DG number of times, so the number of multiplications for the kernel over the input channel becomes $DG^2 \times DK^2$. Now such multiplications are applied over all M input channels, for each channel we have a different kernel and hence the total number of multiplications in the first phase, that is depth wise convolution is $M \times DG^2 \times DK^2$.

Next, we compute the number of multiplications in the second phase that is point wise convolution, here the kernels have a shape 1×1×M where m is the depth of the input volume and hence the number of multiplications for one instance of convolution is M, this is applied to the entire output of the first phase which has a width and height of DG, so the total number of multiplications for this kernel is DG×DG×M, so for some N kernels will have DG×DG×M×N multiplications. Thus, the total number of multiplications is the sum of multiplications in the depth wise convolution stage, plus the number of multiplications in the point-wise convolution stage.

Total Number of Multiplications = $M \times DG^2 \times DK^2 + DG^2 \times M \times N = M \times DG^2 (DK^2+N)$

Now we compare the standard convolution with depth wise convolution we get the ratio as:

$$\frac{N.Muts\ in\ DepthwiseSeparableConvolution}{N.Muts\ in\ StandardConvolution} = \frac{M \times DG^2(DK^2 + N)}{M \times DG^2 \times DK^2 \times N} = \frac{DK^2 + N}{DK^2 \times N} = \frac{1}{N} + \frac{1}{DK^2}$$

This formula above tells us, the larger the number of kernel and the kernel size, the more efficient Depth-wise convolution is, compared to standard convolution. To put this into perspective of how effective DSC is, let us take an example, so consider the output feature volume n of 512 and a kernel of size 4 that's DK=4. Plugging these values into the relation we get 0.0644, in other words standard convolution has more than 15 times more the number of multiplications as that of depth wise separable convolution, this is a lot of computing power.

We can also quickly compare the number of parameters in both convolutions in standard convolution each kernel has DK×DK×M learn about parameters since there are N such kernels there are $N \times M \times DK^2$ parameters in DSC will split this once again into two parts, in the depth wise convolution phase we use M kernels of shape DK×DK in point wise convolution we use n kernels of shape 1×1×M so the total is $M \times DK^2 + M \times N$, or we can just take M common, taking the ratio we get the same ratio as we did for computational power required.

$$\frac{N.ParametersDepthwiseSeparableConvolution}{N.ParametersStandardConvolution} = \frac{M \times (DK^2 + N)}{M \times DK^2 \times N} = \frac{DK^2 + N}{DK^2 \times N} = \frac{1}{N} + \frac{1}{DK^2}$$

For instance, here is a table that contrasts the standard and depth-wise separable convolution

	Standard convolution	Depth-wise separable convolution
N. of parameters first conv layer	340	56
N. of parameters second conv layer	1810	390
Total number of parameters	4,660	2,956
Total number of multiplications In the convolutional layers	380,000	24,400
Time elapsed after 30 epochs	1581.94 seconds	1273.60 seconds

Where depth wise has been used?

Now, we understood exactly what depth wise convolution is, and also its computation power with respect to the traditional standard convolution, but where exactly has this been used? well there are some very interesting papers here, the first is on multi model neural networks these are networks designed to solve multiple problems using a single network a multi model network has four parts, the first is modality Nets to convert different input types to a universal internal representation then we have an encoder to process inputs we have a mixer to encode inputs with previous outputs and we have a decoder to generate outputs. A fundamental component of each of these parts is depth wise separable convolution it works effectively in such large networks.

Next up, we have Xception (Franc¸ois Chollet from Google) a CNN architecture based entirely on depth wise separable convolutional layers, it has shown the state-of-the-art performance on large datasets like Google's JFT image dataset, which is a repository of 350 million images with 17,000 class labels, to put this into perspective the popular image net took 3 days to Train however to Train even a subset of this JFT data set it took a month and it didn't even converge. In fact, it would have approximately taken about three months to converge. This paper has pushed convolution neural networks to use DSC as a default convolution.

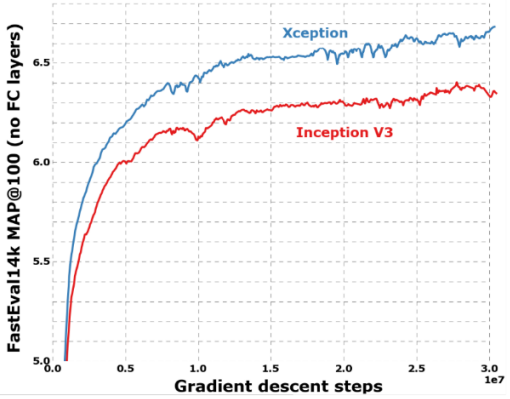


Figure 5 - 9 Training profile on JFT, without fully-connected layers

Up third we have mobile Nets a neural network architecture that strives to minimize latency of smaller scale networks so that computer vision applications run well on mobile devices. Mobile Nets used DSC in its 28-layer architecture. This paper compares the performance of mobile nets with fully connected layers versus DSC layers. It turns out the accuracy on image net only drops

Model	ImageNet Accuracy	Million Mult-Adds	Million Parameters
Conv MobileNet	71.7%	4866	29.3
MobileNet	70.6%	569	4.2

Figure 5 - 10 Depthwise Separable vs Full Convolution MobileNet [Mobilenets]

a 1% while using significantly less number of parameters, from twenty nine point three million the number of parameters it's down to just 4.2 million. We can see the Mult-Adds which is a direct measure of computation has also significantly decreased for DSC mobile Nets.

Employing Depth-wise Separable Convolution

We already compared the accuracy performance of CNNs that use the standard convolution to fully connected networks. So, no need to go back to the part, we know that CNNs are superior. And, we also contrasted the computational complexity of standard convolution to depth wise separable convolution. Now I want to compare the accuracy performance of CNNs that use the standard convolution to CNNs that use the Depth-wise Separable Convolution. For that, I will use the same model size (two convolution layers) with the same everything for both CNNs.

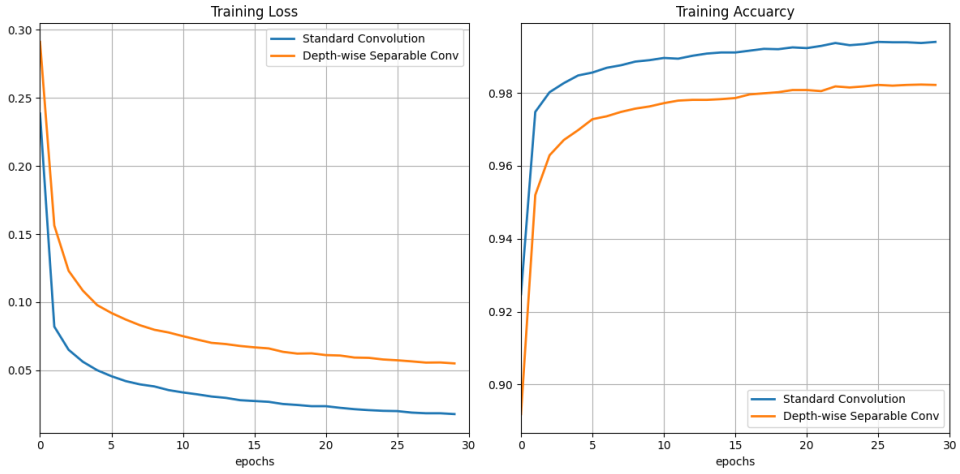


Figure 5 - 11 Standard versus Depth-wise separable convolution performance

Obviously standard convolution yields higher accuracy and lower loss on the training data (as well as the validation data, which I did not include), but the difference seems large because I am only zooming up close, for instance both types of convolution passed the 98% accuracy, and the depth-wise separable convolution is 1.12% less than the standard one, which is acceptable considering we went from 4,660 to 2,956 parameters in the convolution layers, and the training time is significantly smaller. The final classification accuracy on the test data, in fact tells us that the DSC model is better where its classification accuracy is 97.88 % compared to 98.77 % in the standard convolution. But that I guess due to the model started to overfit, because I didn't use any regularization technique

Other Activation functions

Up to now, we have only used Sigmoid as our activation function, let's turn our focus now to this part of the network and investigate the performance of different types of activation

functions, since it's pretty important aspect of network design. The activation function does not only affect the forward pass, but the backward pass as well during the back-propagation, for this reason we need to care not only about its output, but its derivative as well.

There are many activation functions used in modern networks, we will consider the most commonly used ones, namely, Tanh and ReLU (and its derivatives). Now, we have built a considerable familiarity with the Sigmoid function, after investigating two problems, first, the learning slowdown in the output layer when using the MSE (quadratic) cost function. Second, the weight initialization. That's great, we are ready to employ different activation functions.

Another function that almost always works better than the sigmoid is the Tanh (or the hyperbolic tangent) function, defined as:

$$tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} - e^{-x}} \tag{61}$$

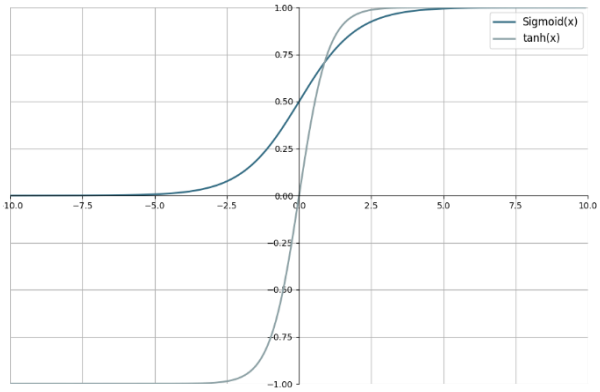


Figure 5 - 12 Tanh vs Sigmoid function

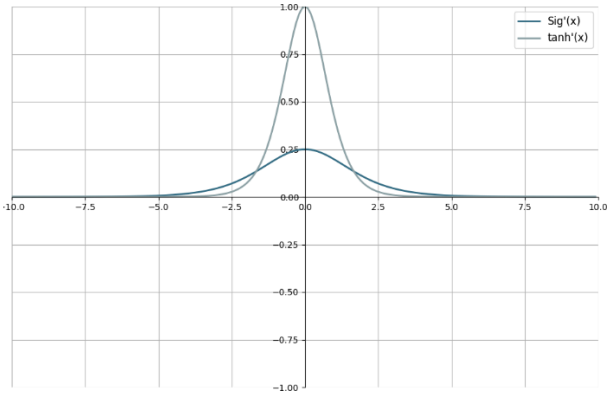


Figure 5 - 13 Tanh and sigmoid derivative

Looking at the graph of the tanh (Figure 5 - 12), unlike sigmoid tanh's output goes from -1 to 1, (so it usually said that is, scaled and shifted version of the sigmoid), but what's more importantly is its derivative graph (Figure 5 - 13), we see that it has derivative of 1, (much higher than the sigmoid) around 0, this will result a faster convergence (fast training).

LeCun et al. write: "Convergence is usually faster if the average of each input variable over the training set is close to zero. To see this, consider the extreme case where all the inputs are positive. Weights to a particular node in the first weight layer are updated by an amount proportional to δ_x where δ is the (scalar) error at that node and is the input vector [...]. When all of the components of an input vector are positive, all of the updates of weights that feed into a node will be the same sign (i.e. sign(δ)). As a result, these weights can only all decrease or all increase together for a given input pattern. Thus, if a weight vector must change direction, it can only do so by zigzagging which is inefficient and thus very slow. [...] Sigmoids that are

symmetric about the origin are preferred for the same reason that inputs should be normalized, namely, because they are more likely to produce outputs (which are inputs to the next layer) that are on average close to zero."

Problem with squashing function

Now, even though tanh works almost always better than sigmoid, it as well is a saturating function, meaning when the input is large (positive or negative) its output will be squeezed to 1 or -1, thus resulting 0 (or near 0) derivative. This leads to a common issue in deep networks known as vanishing and exploding gradient, something which I don't intend to get into it in detail, because we are not considering very deep network in this work (three layers top), and this topic is well explored. Anyway, we've already touched the surface of this issue when we discussed two optimization problems, the idea of learning slowdown in the output layer, and the weight initialization - which was due to the saturation of the sigmoid. The basic idea is, in very large network when back-propagating, the error (derivative) of each layer gets either smaller and smaller or larger, thus causing the learning to either slow down or completely stop.

Rectified Linear Unit family

To solve this problem and produce a fast convergence, a non-saturating activation function like the ReLU (or one of its derivatives) is used [38-41].

Note: The ReLU family, are piecewise functions, meaning that they are defined for different range of input with different function.

- ReLU (Rectified Linear Unit): is a max function given by $f(x) = max(0, x)$ where x is the input. It was first used in Restricted Boltzmann Machines (Nair & Hinton, 2010) [37].

$$\text{ReLU(x)} = \begin{cases} 1, & if\ x \geq 0 \\ 0, & if\ x < 0 \end{cases} \tag{62}$$

- Leaky ReLU: was first introduced in acoustic model (Maas et al., 2013) [38], defined as:

$$\text{Leaky ReLU(x)} = \begin{cases} x, & if\ x \geq 0 \\ x/a_i, & if\ x < 0 \end{cases} \tag{63}$$

  where a_i is a fixed parameter in $range(1, +\infty)$. In original paper, the authors suggest to set a_i to a large number like 100.

- Parametric ReLU: is the same as leaky ReLU with the exception that a_i is learned in the training via back propagation. It was proposed by (He et al., 2015). The authors reported its performance is much better than ReLU in large scale image classification task.
- Randomized Leaky ReLU: is the randomized version of leaky ReLU. It is first proposed and used in Kaggle NDSB Competition. The highlight of RReLU is that in training process, a_{ji} is a random number sampled from a uniform distribution U(l, u) [36]. Formally, we have:

$$\text{Leaky ReLU(x)} = \begin{cases} x, & if\ x \geq 0 \\ a_i x, & if\ x < 0 \end{cases} \quad (64)$$

where $a_{ji} \sim U(l,u), l < u$ and $l,u \in [0,1)$

In the test phase, we take average of all the a_{ji} in training as in the method of dropout (Srivastava et al., 2014) , and thus set a_{ij} to $(l+u)/2$ to get a deterministic result. Suggested by the NDSB competition winner, a_{ji} is sampled from U(3, 8). [36]

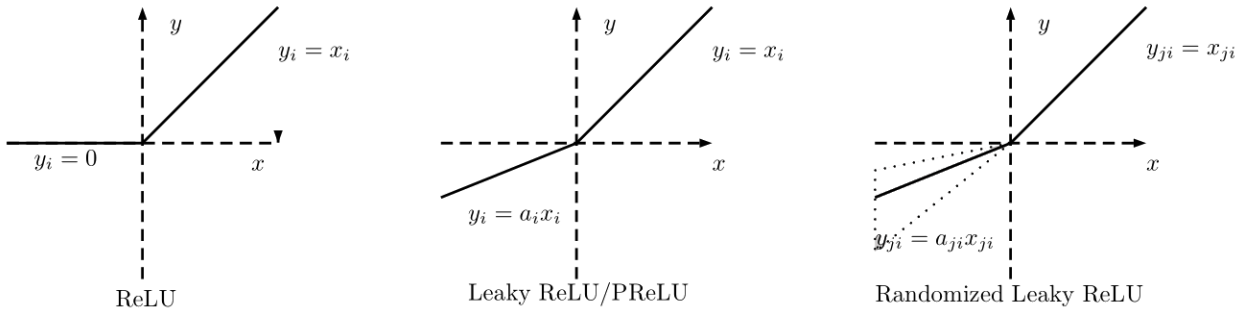


Figure 2 ReLU activation functions family [36]

ReLU family are also usually define using min and max functions:

- $ReLU(x) = max(0, x_i)$
- $LeakyReLU(x) = max(0, x_i) + a_i min(0, x_i)$: where a_i predefined.
- $PrametricLeakyReLU(x) = max(0, x_i) + a_i min(0, x_i)$: where a_i is learned.
- $RandomizeLeakyReLU(x) = max(0, x_i) + a_i min(0, x_i)$: where a_i is random.

Note: The ReLU and its derivatives, are not differentiable at zero, but it's usually implemented so that, the derivative is equal to 1, at zero.

Advantages of using ReLU (Or one of its derivatives):

ReLU have become the standard activation function in NNs and here is why:

- ReLU activations do not face gradient vanishing problem as with sigmoid and tanh function.
- ReLU activations do not require any exponential computation (such as those required in sigmoid or tanh activations). This ensures faster training than sigmoids due to less numerical computation.
- ReLU activations overfit more easily than sigmoids, this sets them up nicely to be used in combination with dropout, a technique to avoid overfitting.

ReLU and its derivatives in use

Benchmarking ReLU against sigmoid in CNNs

I will benchmark the performance of the sigmoid against the ReLU and we will make inferences relying on experiments [36] as well, we will make our conclusions about the rest of ReLU derivatives. I've gone ahead and train two CNN models using the same model and hyper-parameters except one uses sigmoid, and the other uses ReLU activation function.

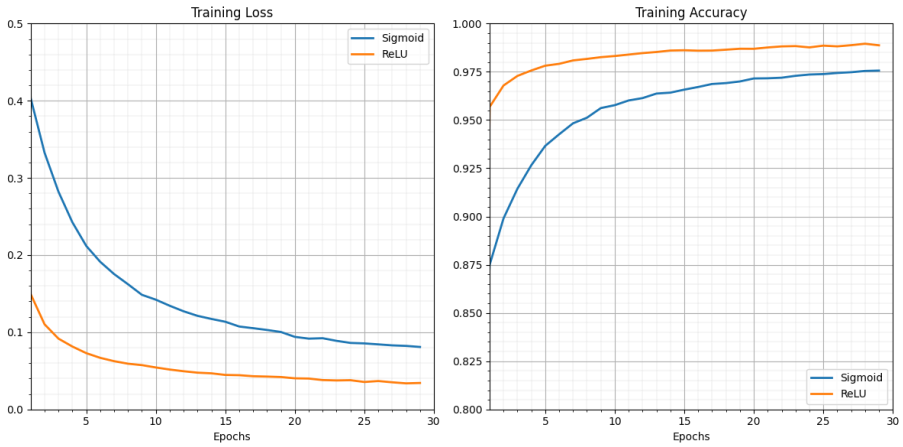


Figure 5 - 14 Loss and classification accuracy on the training data for two CNNs with single convolutional layer followed by a max pooling layer, finally a softmax layer, one using sigmoid, the other using ReLU as an activation fucntion

ReLU performance is definitely superior to the sigmoid, it yields both lower error and better classification accuracy, converging much faster. The final classification accuracy on the test data, using ReLU, 98.85%, with a loss: 0.0350, using sigmoid on the other hand, we got an accuracy of 97.57 % with a loss: 0.0563. For completeness, here is the loss and accuracy on the validation data, as you can see the ReLU outperform the sigmoid on the validation set, just like it did on training set.

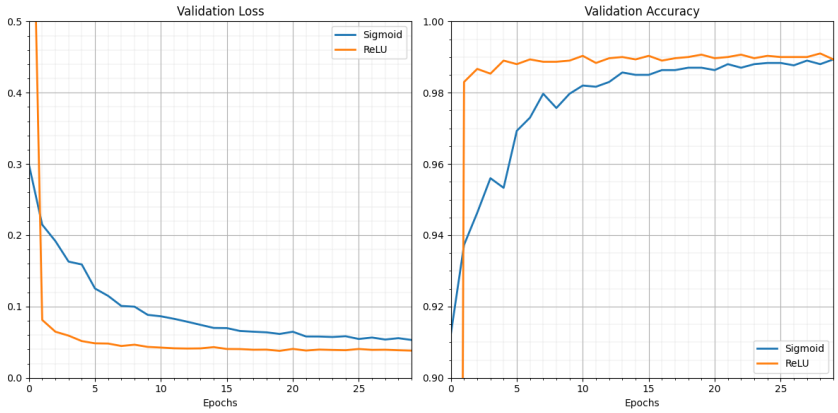


Figure 5 - 15 16 Loss and classification accuracy on the validation data for two CNNs with single convolutional layer followed by a max pooling layer, finally a softmax layer, one using sigmoid, the other using ReLU as an activation fucntion

[36] experiments, suggest that incorporating a nonzero slope for negative part in rectified activation units could consistently improve the results. That's because of the negative input the ReLU has derivative (gradient) of 0, thus the network won't learn when the input of the neuron is negative. And they write "Based on our experiment, we conclude on small dataset, Leaky ReLU and its variants are consistently better than ReLU in convolutional neural networks. RReLU is favorable due to its randomness in training which reduces the risk of overfitting."

Conclusion

Fully-connected NNs are agnostic to the spatial structure, they treat input pixels which are far apart and close together on exactly the same footing. Convolution neural networks on the other hand, are specialized type of NNs that are able to harness the intrinsic spatial structure present in unstructured data, and are well adapted to the translation invariance.

Depth-wise Separable Convolution decreases the computation and number of parameters when compared to standard convolution, second, is that DSC is a combination of depth wise convolution followed by a point wise convolution, depth wise convolution is the filtering step and point wise convolution can be thought of as the combination step. Finally, they have been successfully implemented in neural network architectures like multi model networks exception and mobile nets.

Despite its depth, one of the key characteristics of modern deep learning system is to use non-saturated activation function (e.g. ReLU) to replace its saturated counterpart (e.g. sigmoid, tanh). The advantage of using non-saturated activation function lies in two aspects: The first is to solve the so called "exploding/vanishing gradient". The second is to accelerate the convergence speed [36]. LeakyReLU almost always performs better than ReLU, PReLU can add another dimension of flexibility and allow the network to be more powerful. RReLU minimize overfitting, since ReLU and LeakyReLU tend to overfitt easily, overfitting is not a problem in modern network when we have enough data, since we can always dropout.

General Conclusion

Artificial Intelligence has been witnessing a monumental growth in bridging the gap between the capabilities of humans and machines. Machine learning is a discipline in AI that is all about devising systems and techniques that allow the computer to learn how to perform certain task by inferring rules from the training examples. In this research we focused on the theoretical aspect of neural networks, which is the most successful sub field of machine learning, that encompasses the huge branch in AI, that is deep learning.

Handwritten recognition systems are essential part of our life digitization process, that make passing and storing data whole a lot easier. Data pre-processing is an important aspect of theses system pipelines, to prepare the data to be fed to recognition systems for a robust performance. Feature extraction are as well as an important aspect of OCR system for harnessing the most useful aspect about the input data. Post-processing techniques might enhance the performance and accuracy even further.

The two factors that allow Neural Networks to behave intelligently are, first, the non-linearity of the activation function, second, the layered structure, a specific pattern of activations in the previous layer cause a very specific pattern of activation in the next layer, till we reach the output layer which will be network predicted output of what the target might be.

The gradient descent is the most straight forward way of minimizing the cost function, and the back-propagation is a super-efficient and the most common algorithm for calculating it.

Flexibility is often a good thing, but in the case of neural network a too flexible model (model with high capacity aka high variance) can easily overfit to the training data and start memorizing the details and peculiarities of the training dataset rather than just learning the general pattern, which cause the model's performance to deteriorate for generalizing to new unseen data. Regularization techniques such as L1, L2 and dropout can easily be implemented to prevent this problem. Data augmentation is another effective technique for combating this problem as well.

Convolution neural networks are specialized type of NNs that are able to harness the intrinsic spatial structure in unstructured data, by learning filters that are used to detect patterns in the data, and are well adapted to the translation invariance.

Learning slow-down is a reoccurring problem in NNs when using squashing function such as sigmoid or tanh (although tanh is much better than sigmoid), therefore non-saturating function (ReLU or one of its derivatives) is always preferable especially in deep networks, for avoiding this problem resulting faster convergence and mitigating the problem of Vanishing-Gradient as well.

References

[1]: "Neural Networks and Deep Learning", by Michael Nielsen
[2]: "Deep Learning" by Ian Goodfellow, Yoshua Bengio, Aaron Courville
[3]: "Pattern Recognition and Machine Learning", by Christopher Bishop
[4]: "Neural Networks A Systematic Introduction " by Raúl Rojas
[5]: "Pattern Classification", by Richard O. Duda, Peter E. Hart, David G. Stork
[6]: "Neural Networks Methodology and Applications" by Gérard Dreyfus
[7]: "Neural Networks and Deep Learning" by Charu C. Aggarwal
[8]: "Principles of Artificial Neural Networks" by Daniel Graupe
[9]: "Handwriting Recognition Using Artificial Intelligence Neural Network and Image Processing", by S Aqab, MU Tariq
[10]: "Arabic Online Handwriting Recognition Using Neural Network" by Abdelkarim Mars and Georges Antoniadis
[11]: [vlabs.iitb.ac.in/vlabs-dev/labs_local/machine_learning/labs/exp11/theory.php]
[12]: "Guide to OCR for Arabic Scripts" by Sargur N. Srihari, Gregory Ball (auth.), Volker Märgner
[13]: "Slant Estimation Algorithm for OCR Systems", by E. Kavallieratou, N. Fakotakis and G. Kokkinakis
[14]: "Slant Removal Technique for Historical Document Images" 2018, by Ergina Kavallieratou, Laurence Likforman-Sulem and Nikos Vasilopoulos
[15]: "Knowledge-based intelligent techniques in character recognition" by Lakhmi C. Jain
[16]: "An Improved Document Skew Angle Estimation Technique" by U. Pal, B.B. Chaudhuri
[17]: "Optical Character Recognition"
[18]: "Character Segmentation and Skew Correction for Handwritten Devanagari Scripts":
[19]: "Best Practices for Convolutional Neural Networks Applied to Visual Document Analysis" by PY Simard, D Steinkraus, JC Platt
[20]: "Dropout: A Simple Way to Prevent Neural Networks from Overfitting" 2014 by N Srivastava, G Hinton, A Krizhevsky
[21]: "Improving Neural Networks by Preventing Co-Adaptation of Feature Detectors" 2012 by GE Hinton, N Srivastava, A Krizhevsky
[22]: "Imagenet Classification with Deep Convolutional Neural Networks" NIPS 2012, by Alex Krizhevsky, Ilya Sutskever, Geoffrey E. Hinton
[23]: "Improving Deep Learning with Generic Data Augmentation" 2017 by Luke Taylor, Geoff Nitschke
[24]: "Using a Data Driven Approach to Predict Waves Generated" by Gravity Driven Mass Flows
[25]: "Efficient Backpropagation" 1998 by Y. lecun, L. Bottou, Genevieve B. Grr
[26]: "Backpropagation applied to handwritten zip code recognition" 1989 by Y. LeCun, B. Boser, JS. Denker, D Henderson,
[27]: "An introduction to Neural Networks", by Ben Krose, Patrick van der Smagt

[28]: "Bag of Tricks for Image Classification with Convolutional Neural Networks" 2018, by Tong He, Zhi Zhang, Hang Zhang, Zhongyue Zhang, Junyuan Xie, Mu Li,
[29]: "Cross-Entropy vs. Squared Error Training: a Theoretical and Experimental Comparison" 2013 by Pavel Golik, Patrick Doetsch, Hermann Ney
[30]: "A guide to convolution arithmetic for deep learning" 2016 by V. Dumoulin, F. Visin
[31]: "One Model to Learn Them All" 2017 by L. Kaiser, A.N. Gomez, N. Shazeer, A. Vaswani
[32]: "Xception: Deep Learning with Depthwise Separable Convolutions" 2017 by Franc¸ois Chollet
[33]: "Mobilenets: Efficient convolutional neural networks for mobile vision applications" 2017 by AG Howard, M Zhu, B Chen, D Kalenichenko
[34]: "Revisiting Small Batch Training for Deep Neural Networks", 2018 by Dominic Masters, Carlo Luschi
[35]: "Practical recommendations for gradient-based training of deep architectures", 2012 by Yoshua Bengio
[36]: "Empirical Evaluation of Rectified Activations in Convolution Network" 2015 by B. Xu, N. Wang, T. Chen, Mu Li
[37]: "Rectified linear units improve restricted Boltzmann machines." In ICML, pp. 807–814, 2010 by Nair, Vinod and Hinton, Geoffrey E.
[38]: "Rectifier nonlinearities improve neural net- work acoustic models" In ICML, volume 30, 2013, by Maas, Andrew L, Hannun, Awni Y, and Ng, An- drew Y.
[39]: “Delving deep into rectifiers: Surpassing human-level performance on imagenet classification.”, arXiv preprint arXiv:1502.01852, 2015 by He, Kaiming, Zhang, Xiangyu, Ren, Shaoqing, and Sun, Jian.