# Lecture Notes on Machine Learning Methods in Geosciences

Gerard T. Schuster with Yuqing Chen, Amr Ibrahim, Zongcai Feng,
Kai Lu, Zhaolun Liu, Shihang Feng, and Jing Li

October 27, 2018

# Contents

# Preface

These course notes present the theory and practice of solving geoscience problems with Machine Learning methods. Machine Learning is a field of data analysis methods that can be classified into three categories: supervised learning, unsupervised learning, and reinforcement learning. Our concentration is on the theory and practice of supervised learning methods in geosciences, with an emphasis on neural networks and deep learning. Much of the information on neural networks has been extracted from youtube videos, particularly those from Andrew Ng's excellent online learning course. The book *Pattern Recognition and Machine Learning* by Bishop is used for reference materials.

The first lecture provides a broad overview of Machine Learning (ML) methods classified into the three categories listed above. The next several lectures show how the single neuron model leads to the mathematical concept of a perceptron, the fundamental computational unit of a neural network. The perceptron model can be used for classifying the samples in a training set into different classes. This computational unit can also be used as the fundamental building block of the neural network and its broad applications in deep learning methods. Many case histories are used to demonstrate its usefulness in solving geoscience problems. The last set of lectures describe the theory and practice of the important classification methods of kernel machines and support vector machines.

In summary, the lecture notes cover the following topics.

1. Background: Optimization Theory
2. Background: Probability and Liklihood Theory
3. Introduction to Machine Learning
4. Classification with Perceptrons
5. Activation Functions
6. Linear Regression
7. Logistic Regression
8. Non-Linear Classification
9. Neural Networks: Theory

10. Neural Networks: MATLAB code and Examples
11. Neural Networks: Practical Implementation
12. Neural Networks: Deep Learning
13. Neural Networks: Case Histories
14. Neural Networks: Capacity, VC Dimension, and Dropout
15. Support Vector Machines
16. Kernel Machines
17. Unsupervised Learning Methods

MATLAB and Fortran labs are integrated throughout the text to deepen the reader's understanding of the concepts and their implementation. Such exercises are introduced early and geophysical applications are presented in almost every chapter.

## Acknowledgments

The author wishes to thank the support provided by the sponsors of the Utah Tomography and Modeling/Migration and CSIM consortiums. Strong support was also provided by King Abdullah University of Science and Technology who financially supported me while writing the final chapters.

I also thanks the following reviewers who provided very valuable edits:

# Abbreviations

- ABC - Absorbing boundary condition
- ADCIG - Angle-domain common image gather
- CAG - Common angle gather
- CG - Conjugate gradient
- CIG - Common image gather
- CMG - Common midpoint gather
- COG - Common offset gather
- CSG - Common shot gather
- DM - Diffraction-stack migration
- FD - Finite difference
- FCN - Fully connected neural network
- FWI - Full waveform inversion
- GOM - Gulf of Mexico
- KM - Kirchhoff migration
- LSM - Least squares migration
- LSRTM - Least squares reverse time migration
- MD - Migration deconvolution
- MVA - Migration velocity analysis
- NN - Neural network
- NMO - Normal moveout
- PDE - Partial differential equation
- PSTM - Prestack time migration

- QN - Quasi-Newton
- RTM - Reverse time migration
- SD - Steepest descent
- SPD - Symmetric positive definite
- SSP - Surface seismic profile
- QP - Quadratic programming
- VSP - Vertical seismic profile
- ZO - Zero offset
- WT - Wave equation traveltime tomography
- WTW - Wave equation traveltime and waveform tomography

# Chapter 1

# Introduction to Machine Learning in the Geosciences

The chapter provides a partial overview of machine learning methods and summarizes their historical development.

## 1.1 Introduction

Machine learning (ML) was generally defined by Arthur Samuel in 1950s as the following:

*[Machine learning is the] field of study that gives computers the ability to learn without being explicitly programmed.*

At that time the term *machine learning* was not in use, but this definition applies to the ML field today. Some researchers use the phrase "deep learning", but this typically refers to convolutional neural networks with many layers (Goodfellow et al., 2016).

Over the last 75 years machine mearning has undergone several metamorphisms. From the 1940s to early 1960s, cybernetics defined (Wiener, 1948) as "the scientific study of control and communication in the animal and the machine", was the hot research topic related to ML (see blue curve Figure 1.1) and is now used in a rather loose way to imply "control of any system using technology." (Wikipedia). That is, it is the scientific study of how humans, animals and machines control and communicate with each other. It was partly inspired in the early 1940s by the discovery of a biological model for theories of learning (McCulloch and Pitts, 1943; Hebb, 1949).

The neuron-function model of the brain was recast as a mathematical algorithm known as the perceptron introduced by Rosenblatt (1958). In its simplest form the $i^{th}$ input signal $x_i$ to a neuron is related to the output $y$ by

$$y = \sigma(\sum_i w_i x_i + b), \tag{1.1}$$

where $w_i$ is the weight to the input $x_i$, $b$ is a bias term and $\sigma(z) = 1/(1 + e^{-z})$ is the sigmoid function shown in Figure 1.2. In the original perceptron, the *activation* function

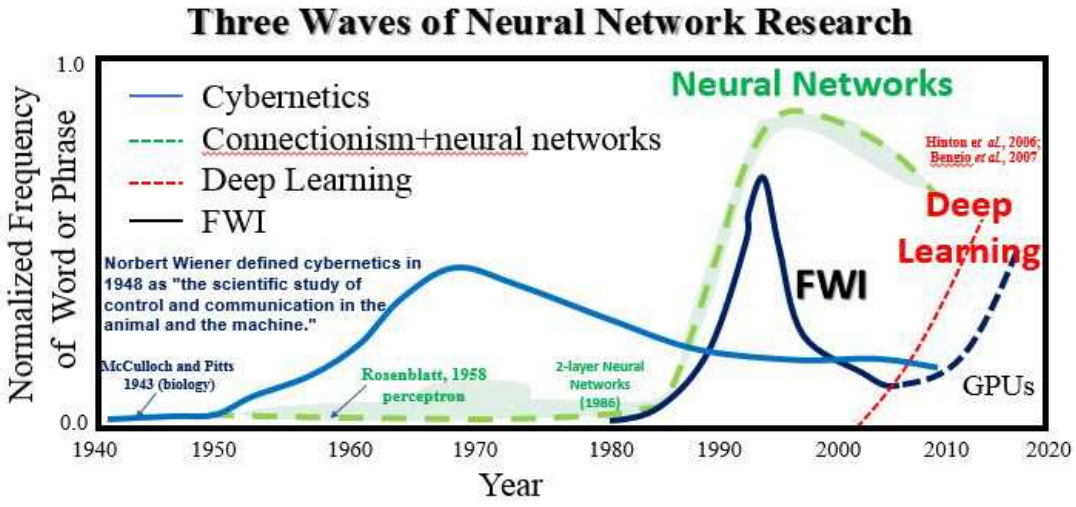


Figure 1.1: Two of the three historical waves of artificial neural nets research, as measured by the normalized frequency of the phrases cybernetics and connectionism or neural networks according to Google Books (Goodfellow et al., 2016). The third wave, deep learning, is schematically described by a rapid rise and does not correspond to the actual word count. For geophysicists, the schematic curve for FWI shows its rise, fall, and rebirth in the exploration community.

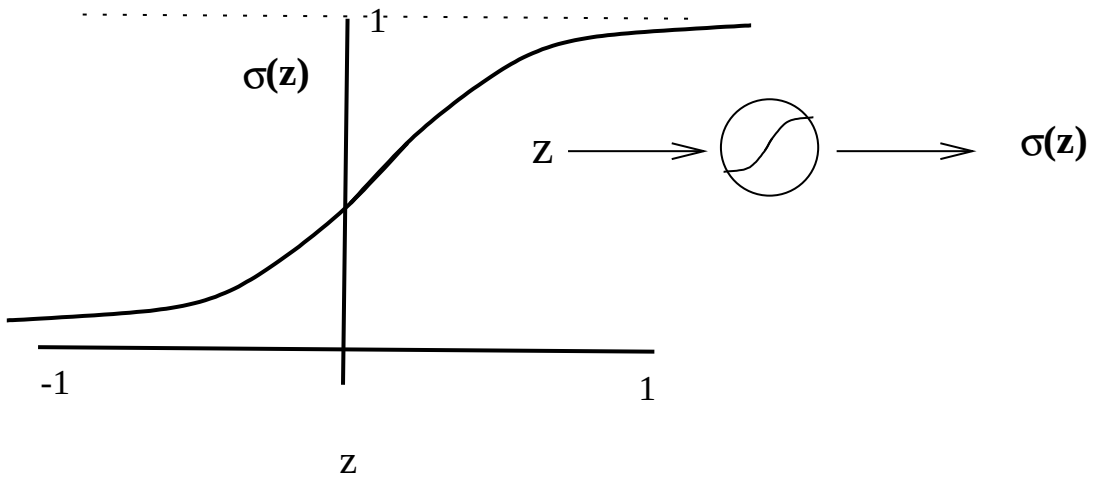


Figure 1.2: Sigmoid function plotted against its argument and its system diagram to the right.

Figure 1.3: Mark 1 computer for image recognition. The photograph on the left shows how the inputs were obtained using a simple camera system in which an input scene, in this case a printed character, was illuminated by powerful lights, and an image focussed onto a $20 \times 20$ array of cadmium sulphide photocells, giving a primitive 400 pixel image. The perceptron also had a patch board, shown in the middle photograph, which allowed different configurations of input features to be tried. Often these were wired up at random to demonstrate the ability of the perceptron to learn without the need for precise wiring, in contrast to a modern digital computer. The photograph on the right shows one of the racks of adaptive weights. Each weight was implemented using a rotary variable resistor, also called a potentiometer, driven by an electric motor thereby allowing the value of the weight to be adjusted automatically by the learning algorithm. Image and caption from Bishop (2006).

was the all-or-nothing[1] $\sigma(z) = sign(z)$ function, but was much later replaced by smoother almost-all-or-nothing functions such as the sigmoid $\sigma(z) = 1/(1+e^{-z})$ where the derivative exist everywhere.

According to Wikipedia: "The perceptron was intended to be a machine, rather than a program, and while its first implementation was in software for the IBM 704, it was subsequently implemented in custom-built hardware as the "Mark 1 perceptron" shown in Figure 1.3. This machine was designed for image recognition: it had an array of 400 photocells, randomly connected to the "neurons". Weights were encoded in potentiometers, and weight updates during learning were performed by electric motors. In a 1958 press conference organized by the US Navy, Rosenblatt made statements about the perceptron that caused a heated controversy among the fledgling AI community. The New York Times reported the perceptron to be *the embryo of an electronic computer that [the Navy] expects will be able to walk, talk, see, write, reproduce itself and be conscious of its existence.*".

From a simple biological point of view the perceptron model suggests that an input electrical signal $x$ will be transmitted by the neuron through the connecting synapse to the neighboring neuron if the input voltage exceeds a threshold value. The input values $x_i$ are similar to the electrical impulses that travel through the many synapses that feed into the neuron.

A system of many perceptrons came to be known as a neural network in the 1980s (see

[1]The *sign* function is also known as the Heavyside function.

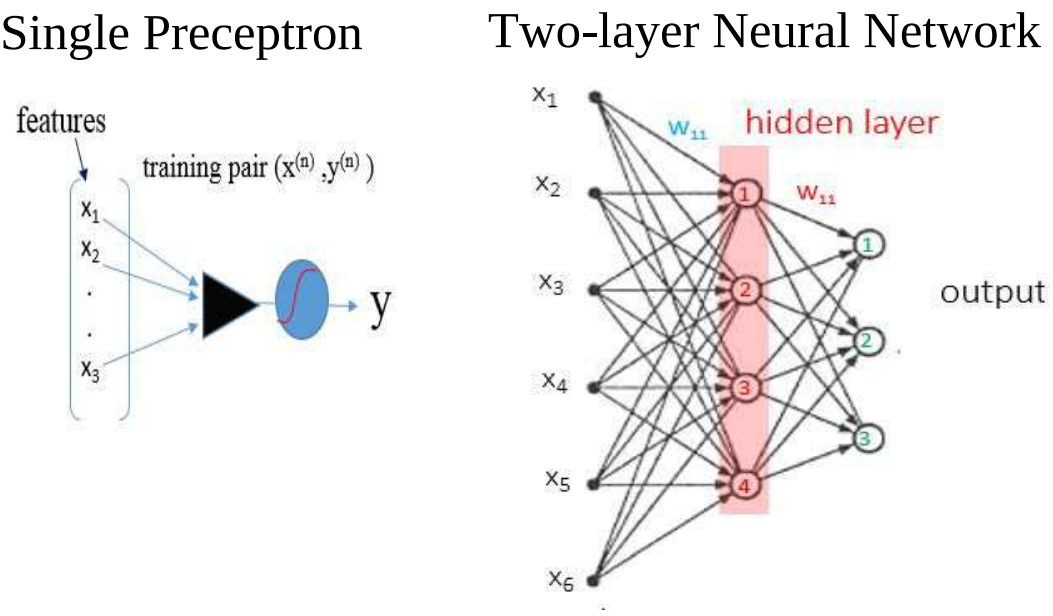


Figure 1.4: (Left) Single-layer perceptron and (right) two-layer neural network.

green-dashed curve Figure 1.1), which could be used to perform practical functions such as classification of images. Instead of one preceptron, the $N \times 1$ input signal $\mathbf{x}$ feeds into $M$ neurons so that equation 1.1 becomes

$$y_j = \sigma(\sum_{i=1}^{N} w_{ij}x_j + b_i), \tag{1.2}$$

where the $M \times 1$ output vector is $\mathbf{y}$, the weights $w_i \rightarrow w_{ij}$ become those associated with the $M \times N$ weight matrix $\mathbf{W}$. The activation function $\sigma$ acts on each element of the matrix-vector product $z_i = \sum_{i=1}^{N} w_{ij}x_j + b_i$ to return the $M \times 1$ output vector $\mathbf{y}$, which is also known as the activation vector $\mathbf{a}$ if it is a *hidden* layer and not the last column of nodes. A single perceptron with multiple inputs is shown on the left side of Figure 1.4. According to Goodfellow et al. (2016), connectionism is another name similar to that of artificial neural networks (ANN) used for ML in the 1980s and 1960s. The word connectionist refers to the speculation in the 1950's that "...the images of stimuli may never really be recorded at all, and that the central nervous system simply acts as an intricate switching network, where retention takes the form of new *connections*, or pathways, between centers of activity." (Rosenblatt, 1958).

Weighting the input values and summing them together to give the input to the non-linear activation function $\sigma$ defines the two basic building blocks of a neural network. The concatenation of each weighted sum followed by the application of the activation function forms one layer, and a sequence of such layers forms a multilayer neural network. An example is shown on the right side of Figure 1.4.

Interest in neural networks grew rapidly from the 1980s and peaked around the mid 1990s but started to fall off, partly because its limitations were revealed as being computa-

tionally too expensive to fulfill all of its early promises. It also suffered from the inability to theoretically predict its ability to perform well, including IBM's Deep Blue's success in beating a Chess grandmaster[2] in 1997. Around the same time attention was being refocused to mathematically tractable methods such as Support Vector Machines (Bishop, 2006).

This decline in interest suddenly changed in 1998 with the development of the convolutional neural networks (CNNs) by LeCun et al. (1998). Instead of requiring massive memory and computations to perform neural network computations, CNN replaced fully connected layers and large weight matrices with relatively small convolutional matrices. For equation 1.2 this means that the matrix-vector multiplication is replaced by a correlation of the input vector with a much smaller number of weights. For $M = N$, the memory and computational requirements are $O(N^2 n_f)$ for CNN compared to $O(N^4)$ for classical neural networks. Here, the input image for a grayscale picture is an $N \times N$ grid of intensity values and the number of convolutional filter coefficients is $n_f << N$. The CNN algorithm allowed for the development of large neural networks with many deep layers, and gave rise to re-branding of *neural network research* as *deep learning research* (see red-dashed curve Figure 1.1).

Deep learning has exploded in the level of interest in a wide variety of fields, and has become publicly prominent with its success in popular media companies such as Facebook and Google and its use with self-driving cars. A force multiplier is the development of fast cheap GPUs that can expedite the convolutional operations. CNN and its application to geoscience problems is the main focus of these lecture notes.

Coincidentally, the rise and fall and the rebirth of full waveform inversion (FWI) is schematically traced as the black curve in Figure 1.1. Part of the reason for its rebirth are two developments: 1) multiscale techniques that tend to avoid getting stuck in local minima and 2) the exponential increase in computing power.

## 1.2 Three Classes of Machine Learning

Machine learning methods can be classified into the three classes shown in Figure 1.5: supervised learning, unsupervised learning, and reinforcement learning. Researchers are currently exploring their potential applications in three categories of seismic exploration: reservoir analysis, seismic data processing and seismic interpretation. This book uses many seismic examples from the last two categories to highlight the different applications of supervised and supervised learning.

### 1.2.1 Supervised Learning

Supervised learning is a procedure for finding a model, for example finding the coefficients of $\mathbf{W}$, that predicts the output $\mathbf{y}$ from the input data $\mathbf{x}$. The procedure is denoted as supervised learning because $\mathbf{W}$ is found by using a large training set that consists of many examples $(\mathbf{x}^{(n)}, \mathbf{y}^{(n)})$ for $n \in [1, 2 \ldots N]$.

One use for supervised learning is for classifying input data into different categories. For example, the goal in Figure 1.5a is to create a neural network that can distinguish pictures

[2] *https://en.wikipedia.org/wiki/Deep_Blue_versus_Garry_Kasparov*

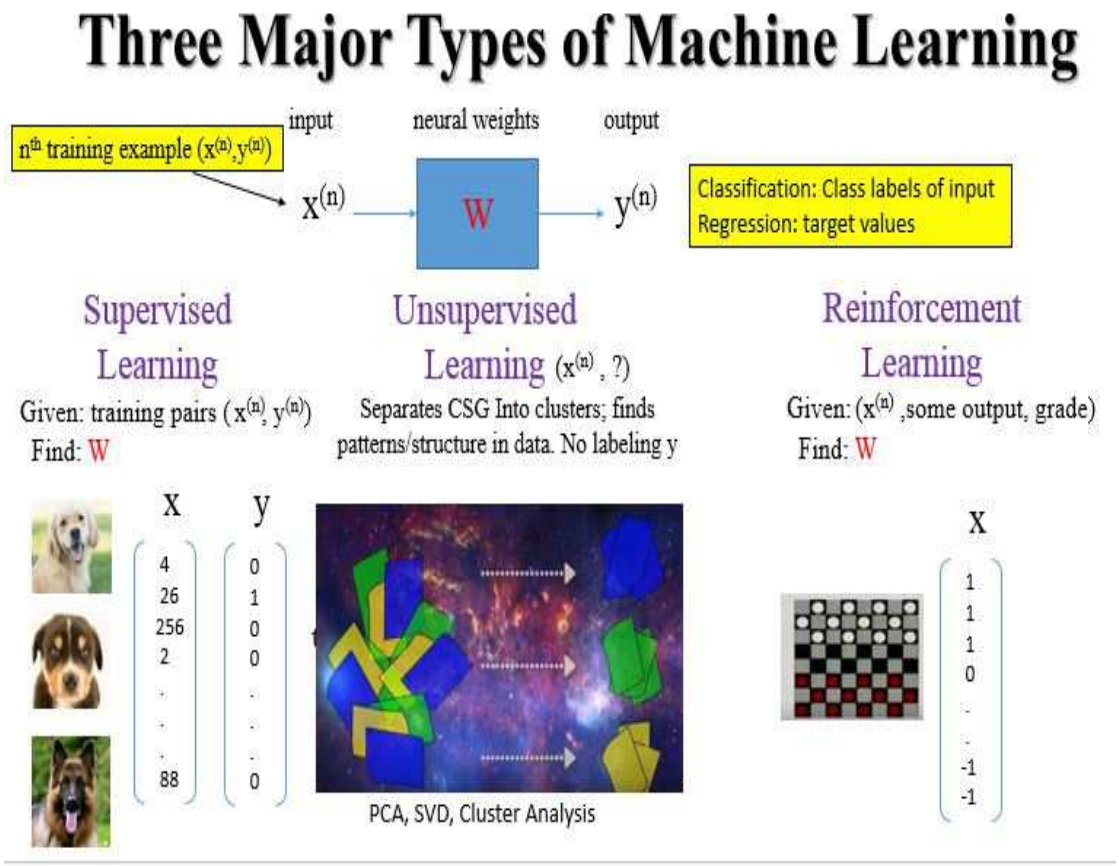


Figure 1.5: Three classes of machine learning.

of dogs from other types of animals. Here, the *training pairs* consist of dog images and the label $y = 1$ for a dog and the label $y = 0$ for not a dog. The $N \times N$ dog image is *flattened* into an $N^2 \times 1$ vector $\mathbf{x}^{(n)}$, and the training pair $(\mathbf{x}^{(n)}, \mathbf{y}^{(n)})$ consist of both the input $\mathbf{x}^{(n)}$ and target output $\mathbf{y}^{(n)}$ pairs. There are $N$ training pairs. The label vector $\mathbf{y}$ in this case is a scalar where $y = 1$ for a dog and $y = 0$ for not a dog.

Many pairs of training data are used to *train* the neural network so that an overdetermined system of equations is formed. The coefficients in $\mathbf{W}$ are found by an iterative optimization method (see Chapter 5). An accurate estimate of $\mathbf{W}$ will allow pictures of dogs that are outside the training set to be identified by the neural network.

Another example of supervised learning is shown in Figure 1.6 where thin sections of rocks are the input images (Patel and Chaterjee, 2016). The goal is to classify them according to the type of limestone. Instead on inputting the entire image of a thin section, they are skeletonized into smaller features, which is shown as a two part operation. Each colorized thin-section image is reduced to the histogram values of RGB colors. Then the second-order, third-order and fourth-order statistics of each of the three histograms are computed by calculating their variance, skewness, and kurtosis values. These are the values input into the neural network, and there are seven output classes. These classes correspond to the dominant type of geology in the image as determined by a geologist. A number of classified thin sections with class labels are used for training. The training estimates the optimal values of the coefficients in $\mathbf{W}$. A new set of thin sections are then used as input into the neural network, which then classifies these out-of-the-training set data. The results are shown in the lower right side of Figure 1.6. Comparison of the predicted classes and the actual classes show more than a 90% accuracy in labeling the data.

**Weakly Supervised and Semi-supervised learning.**

A subset of supervised learning is semi-supervised learning where their are only a few labeled data but there are lots of unlabeled data (Patel et al., 2016). In this case it is too expensive to label more than a few examples of data.

There is also weakly-supervised learning where each image is given a label but the goal is to find labels at the pixel level. Each image can contain features from each class, but it is given a label that denotes the dominant feature. For example, small localized images of seismic sections might be labeled as predominantly salt dome or a turbidite sequence, but they also contain other geological structures. Labeling each localized image as just one class is quickly done by an interpreter, but it ignores the detailed geology. Using a collection of these labeled images, a dictionary can be built up and be used to train a weakly supervised neural network to label each image pixel as the type of geology belonging to one of the classes. Thus, detailed geology at the pixel level can be provided by a machine-learning algorithm. A key goal of representation learning is to disentangle the factors of variation that contribute to the appearance of an image (Patel et al., 2016).

### 1.2.2 Unsupervised Learning

Unsupervised learning is when the training data do not specify the target vector $\mathbf{y}$ but recognizes that there are distinct patterns in the input data $\mathbf{x}$. The goal is to separate distinct patterns from one another. As an example, Figure 1.7a depicts a common shot

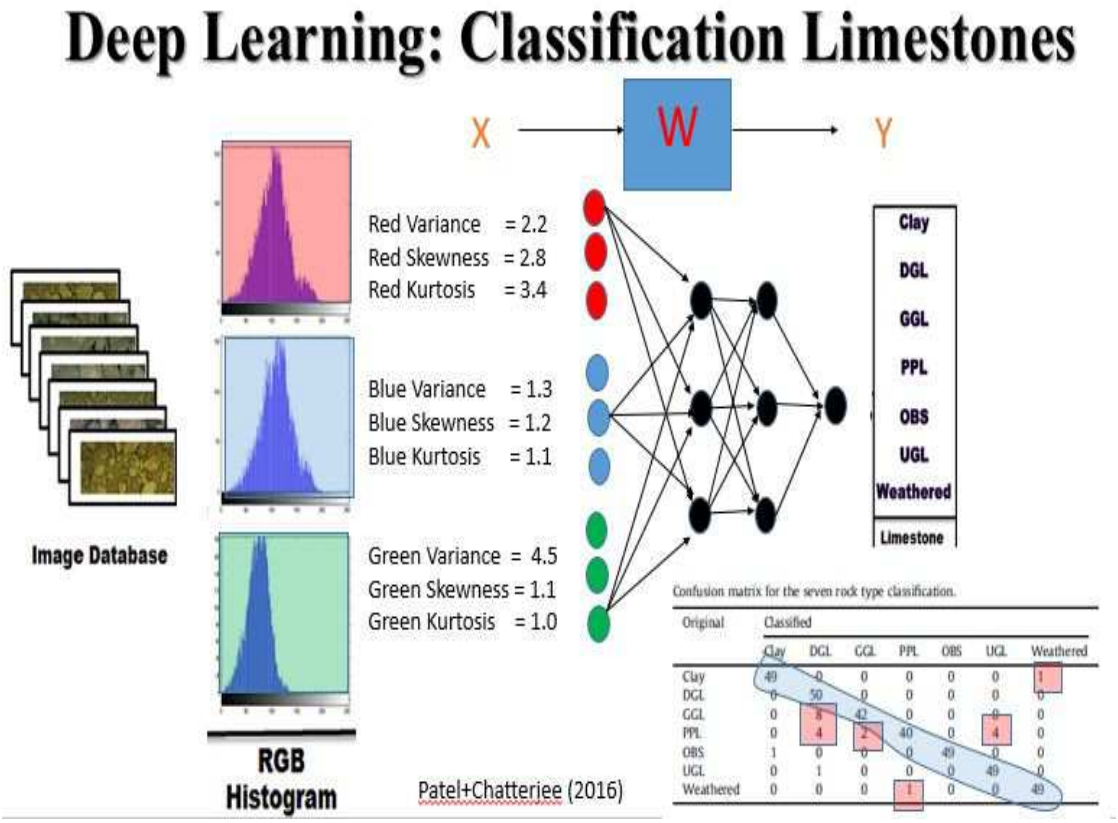


Figure 1.6: Neural layer network for classifying thin sections.

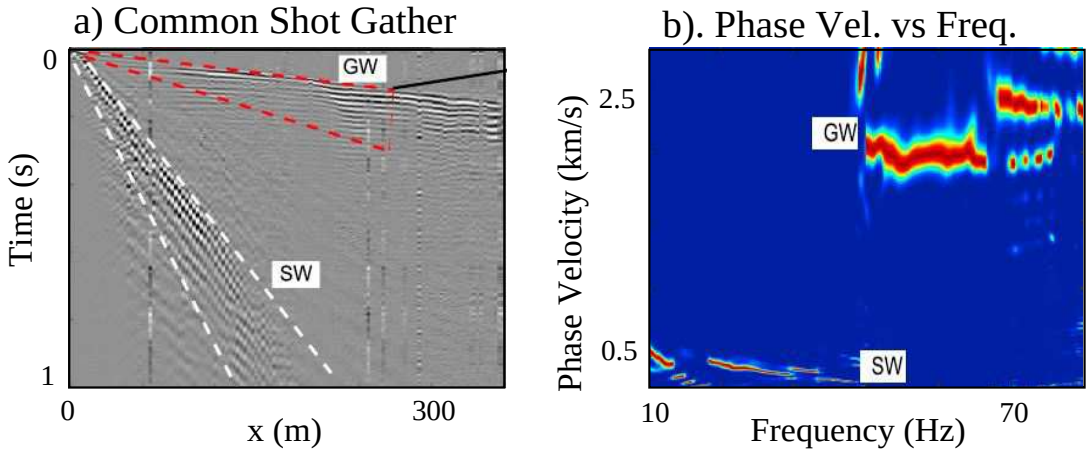


Figure 1.7: a). Common shot gather (CSG) of seismic data and b) phase velocity ($\omega/k$) versus frequency ($f = \omega/2\pi$) plot of the CSG. Here, the Rayleigh surface waves (SW) and P-wave guided waves (GW) are now well separated after a temporal Fourier transform and a Radon transform (Li et al., 2018).

gather of seismic data and its b) transform to phase-velocity and frequency space. The Rayleigh surface waves and P-wave guided waves (GW) are tangled together in the original data but are well separated in the $C - f$ domain because they each propagate with very different propagation velocities; here, $C$ is the phase velocity and $f$ denotes frequency. Other examples include the separation of signal from noise by principle component analysis (PCA) in Chapter **??** or by bandpass filtering.

### 1.2.3 Reinforcement Learning

Reinforcement learning is used to train a system to play a game. In his case, the output is not a labeled class but it is a reward for performing the desired behavior, such as winning a checkers or chess game. Training is performed by learning from chess games played by experts, and at some point the computer can play against itself to refine the strategies.

## 1.3 Summary

Machine learning and especially deep learning seem magical, but also frustrating because they sometimes appear to be theoretically impenetrable! The trend is to reduce the size of feature map with depth but increase the number of filters. Networks jump from convolutional to fully connected layers. What justifies these operations? Unfortunately, deep learning does not have a deep theoretical basis to help us understand its performance. It is a collection of algorithmic tricks that seem to usually work and are economically driven by its success with self-driving cars, Google, IBM, etc. But this seems to be a natural evolution of early scientific discoveries, make things intuitively work by trial and error, and then make efforts to understand its theoretical basis. This is no different than the technological developments of farming, mining, and other practical tools, including full waveform

inversion. Eventually a deeper understanding of the theoretical basis of deep learning will propel it forward to going beyond beyond brute-force interpolation.

# Chapter 2

# Data Prediction by Least Squares Inversion

The goal of inversion is to find the *model* $\mathbf{w}$ that *best predicts* the target data $\mathbf{t}$ from input data $\mathbf{X}$. Here, the governing system of equations is represented by a system of equations $\mathbf{Xw} = \mathbf{t}$ that are, typically, overdetermined, inconsistent and ill-conditioned. No exact solution exists so we seek the *optimal* model $\mathbf{w}^*$ that gives the minimum prediction error $\epsilon = \frac{1}{2}||\mathbf{Xw} - \mathbf{t}||^2$ under the $L_2$ norm. We show how regularization and preconditioners can be used to alleviate problems with ill-conditioning, inconsistency and non-linearity. The inordinate expense of using a direct method for solving a large system of equations forces us to adopt the iterative solution method known as gradient optimization.

For general geophysical inversion, the elements of the target-data vector $\mathbf{t}$ can take on any real-valued measurements, such as gravity readings, and the non-linear operator $\mathbf{X}$ is characterized by the Newtonian physics of, e.g., a partial differential equation. In contrast, classification by a neural network restricts the element values of $\mathbf{t}$ to be a restricted range of numbers that indicate the class of the input data. For example, the input data might be photographs of dogs and cats, and the classification problem is to classify the input as either a cat where $t_1 = 1$ and $t_2 = 0$ or a dog where $t_1 = 0$ and $t_2 = 1$. In this case the target vector is the $2 \times 1$ vector $\mathbf{t} = (t_1, t_2)^T$. There is typically no governing equation based on physics, instead the model $\mathbf{w}$ is extracted from patterns detected in a large amount of input samples $(\mathbf{X}, \mathbf{t})$.

## 2.1 Least Squares Inversion

The key ideas underlying least squares inversion will be explained by the simple example of determining the velocity of a motorcycle (see Figure 2.1a) as it travels down a gravel road of length 5 km. The procedure is to estimate the slowness $w = t/x$, i.e. the inverse speed, of the motorcycle by recording travel time $t$ as a function of travel distance $x$. The timing measurements (in units of seconds) are recorded every 1 km on a gravel road next to the Bonneville Salt Flats ("$https://en.wikipedia.org/wiki/Bonneville\_Salt\_Flats$") in Utah (see Figure 2.1a) . Five travel times and distances are recorded and stored in the $5 \times 1$ vectors $\mathbf{t}$ and $\mathbf{x}$, respectively. The combined data set $(\mathbf{x}, \mathbf{t})$ is known as a *sample* and

contains errors in the traveltimes. We might also repeat the experiment at different sites (for example, see Figure 2.1c) to give us a new sample $(\mathbf{x}', \mathbf{t}')$; the total ensemble of samples is known as the *training data*.

### 2.1.1 Overdetermined, Inconsistent, and Ill-conditioned Equations

The relationship that links $\mathbf{x}$ with $\mathbf{t}$ will be hypothesized, for now, to be a linear model where

$$xw = t \rightarrow \overbrace{\begin{bmatrix} x_{11} \\ x_{21} \\ x_{31} \\ x_{41} \\ x_{51} \end{bmatrix}}^{\mathbf{X}} \overbrace{\begin{pmatrix} w_1 \end{pmatrix}}^{\mathbf{w}} = \overbrace{\begin{pmatrix} t_1 \\ t_2 \\ t_3 \\ t_4 \\ t_5 \end{pmatrix}}^{\mathbf{t}=observed\ times} \rightarrow \begin{bmatrix} 0 \\ 1 \\ 2 \\ 3 \\ 4 \end{bmatrix} \begin{pmatrix} w \end{pmatrix} = \begin{pmatrix} 0 \\ 0.102 \\ 0.33 \\ 0.50 \\ 0.80 \end{pmatrix}, \qquad (2.1)$$

where $\mathbf{X}$ is the $5 \times 1$ input matrix of travel distances, $\mathbf{w}$ is the $1 \times 1$ slowness vector, and $\mathbf{t}$ is the $5 \times 1$ vector of recorded traveltimes. The element value $x_{ij}$ is equal to the distance traveled to the $i^{th}$ recording station with the motorcycle traveling at the $j^{th}$ slowness value. The distances (units of km) and times (units of hours) associated with the above equation are plotted in Figure 2.2a where $\mathbf{t}$ is assumed to be polluted with measurements errors, denoted as noise. The goal is to find the *best* estimate of the slowness $w$.

Equation 2.1 is an $M \times N$ system of equations where $M = 5$ is the number of equations and $N = 1$ is the number of unknown slowness values. This is an *overdetermined* system of equations because $M > N$. It is also an *inconsistent* set of equations because there is no common solution. For example, the solution $w = .102\ hr/km$ to the 2nd-row equation contradicts the solution to the 3rd-row solution $w = .33/2 = .165\ hr/km$.

If there are many solutions that can nearly satisfy the same equations then this is known as an *ill-conditioned system* of equations. For example, if the modeling equation is $w_1 + 10^{-3}w_2 = t$, then $\delta t/\delta w_2 = 10^{-3} \rightarrow \delta w_2 = 10^3 \delta t$, which says that a small change in the traveltime $\delta t$ will lead to a large change in the model $w_2$. As an example, small traveltime errors in the data will lead to unrealistically large changes in the model. The term $\delta t/\delta w_i$ is denoted as a *Frećhet derivative* that characterizes the sensitivity of the data $t$ to changes in the $i^{th}$ model parameter.

### 2.1.2 Least Squares Solution

There is no exact solution to equation 2.1 so we need to define a criterion for *best solution*. The one we will discuss for now is the least squares solution[1] that minimizes the *objective*

[1]The least squares solution is sometimes known as regression because it the least squares solution is the one that *regresses* to the average.

a) Road next to Bonneville Salt Flats

b) Suggestion


c) Great Gravel Road to Great Salt Lake

Figure 2.1: Pictures taken on author's motorcycle trip in Utah's West Desert with views of a) Bonnevile Salt Flat region, b) wildlife sign, and c) the Great Salt Lake.

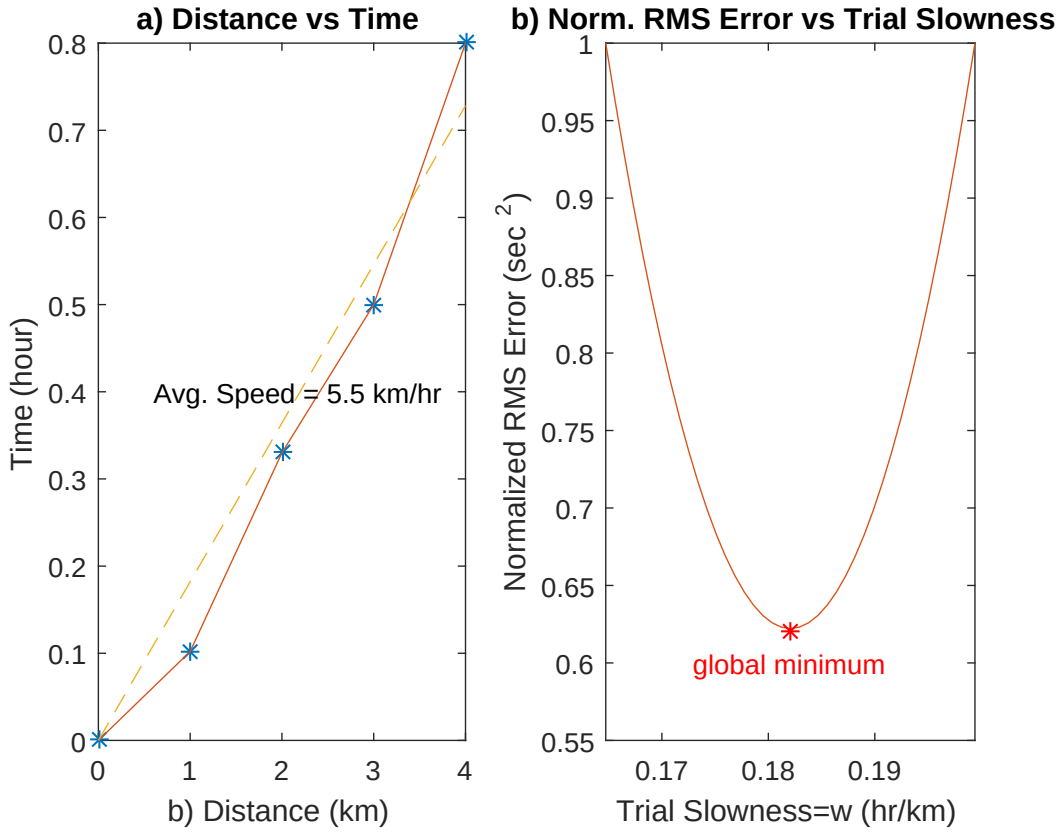


Figure 2.2: Plots for a) the observed $t(x)$ (blue stars) and b) misfit function $\epsilon$ for trial values of the slowness $w$. The dashed yellow line in a) represents the predicted times using the least squares estimate of the velocity and the red star in b) indicates the global minimum $\epsilon(\mathbf{w}^*)$ where $w^* = 1/5.5 = 0.182$ is the stationary point.

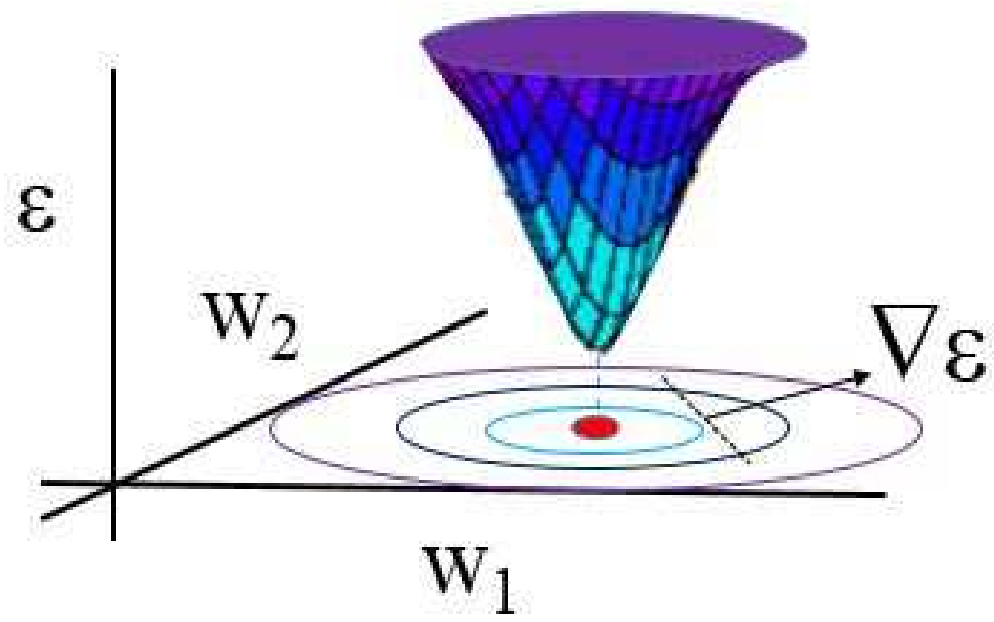


Figure 2.3: Misfit function and contours for $N = 2$, where the vertex defines the global minimum $\mathbf{w}^* = (w_1^*, w_2^*)$ at the red dot. The gradient $\nabla\epsilon = (\frac{\partial\epsilon}{\partial w_1}, \frac{\partial\epsilon}{\partial w_2})$ points along the steepest uphill direction and is perpendicular to the contour tangent (dashed line). See exercises 3-6.

*function*[2] $\epsilon$, the sum of the squared residuals:

$$\begin{aligned}
\epsilon &= \frac{1}{2}\overbrace{(\mathbf{Xw}-\mathbf{t})^T}^{\mathbf{r}^T}\overbrace{(\mathbf{Xw}-\mathbf{t})}^{\mathbf{r}}, \\
&= \frac{1}{2}\mathbf{w}^T\mathbf{X}^T\mathbf{Xw} - \mathbf{t}^T\mathbf{Xw} + \frac{1}{2}\mathbf{t}^T\mathbf{t} \\
&= \frac{1}{2}\sum_{i=1}^{M}(\overbrace{(\mathbf{Xw})_i - t_i)^2}^{r_i^2}, \qquad (2.2)
\end{aligned}$$

where $\epsilon$ is also denoted as a data misfit function. Here, $\mathbf{r}$ is the $M \times 1$ residual vector which is the difference between the predicted times $\mathbf{Xw}$ and measured times $\mathbf{t}$. For $N = 1$, $\epsilon$ plots out as the parabola in Figure 2.2b for the motorcycle data, for $N = 2$ it describes the error bowl in Figure 2.3, and for $N > 2$ it describes an $N - 1$-dimensional quadric surface.

The vertices of the parabola in Figure 2.2b and the error bowl in Figure 2.3 define the

[2]The machine learning community often denotes $\epsilon$ as the *loss function*, which can take many forms; $\epsilon$ is also known as a data misfit function if there is no regularization.

global minimum $w^*$ where the gradient $\nabla\epsilon =$ is zero. This zero-slope condition[3] is written as

$$\left.\frac{\partial\epsilon}{\partial w_k}\right|_{\mathbf{w}=\mathbf{w}^*} = 0, \quad \forall\ k, \tag{2.3}$$

where the $k^{th}$ component of the $N \times 1$ *misfit gradient* vector is explicitly written as

$$\begin{aligned}
\nabla\epsilon_k = \frac{\partial\epsilon}{\partial w_k} &= \sum_{i=1}^{M=5} r_i \frac{\partial r_i}{\partial w_k}, \\
&= \sum_{i=1}^{5} \overbrace{r_i}^{r_i=\sum_{n=1}^{N} x_{in}w_n - t_i} \underbrace{\frac{\partial(\sum_{n=1}^{N} x_{in}w_n - t_i)}{\partial w_k}}_{\partial r_i/\partial w_k = \sum_{n=1}^{N} x_{in}\partial w_n/\partial w_k = \sum_{n=1}^{N} x_{in}\delta_{nk} = x_{ik}}, \\
&= \overbrace{\sum_{i=1}^{5} x_{ik}(\sum_{n=1}^{N} x_{in}w_n - t_i)}^{(\mathbf{X}^T(\mathbf{X}\mathbf{w}-\mathbf{t}))_k},
\end{aligned} \tag{2.4}$$

and the *Kronecker delta function* is defined to be $\delta_{ik} = 1$ if $i = k$, otherwise $\delta_{ik} = 0$. Notice that the outermost summation $\sum_{i=1}^{5} x_{ik}r_i$ in the last line is over the row index $i$ of the element $x_{ik}$, so in matrix-vector notation this is equivalent to multiplying the transpose matrix $\mathbf{X}^T$ by the residual vector $\mathbf{r}$.

Setting the derivative in equation 2.4 to be zero $\forall\ k$ and rearranging gives the *normal equations*[4]

$$[\mathbf{X}^T\mathbf{X}]\mathbf{w} = \mathbf{X}^T\mathbf{t}. \tag{2.5}$$

The least squares solution to the normal equations is

$$\mathbf{w} = [\mathbf{X}^T\mathbf{X}]^{-1}\mathbf{X}^T\mathbf{t}, \tag{2.6}$$

which minimizes the sum of the squared residuals. See Box 2.1.1 for the least squares solution to equation 2.1.

[3]Equation 2.3 is considered to be a stationary condition because $\epsilon$ does not change much over a small range of $w$ at the bottom of the, for example, green curve in Figure 2.2.

[4]These are called normal equations because they can be multiplied by $\mathbf{w}^T$ and rearranged into the form $\mathbf{w}^T\mathbf{X}^T(\mathbf{X}\mathbf{w} - \mathbf{t}) = (\mathbf{X}\mathbf{w}, \mathbf{X}\mathbf{w} - \mathbf{t}) = 0$. This says that $\mathbf{w}^*$ gives the predicted traveltime vector $\mathbf{X}\mathbf{w}^*$ that is perpendicular to the residual vector $\mathbf{X}\mathbf{w}^* - \mathbf{t}$. Here, the parentheses indicate the inner product between two vectors.

**Box 2.1.1. Least Squares Solution to Equation 2.1**

For equation 2.1, there is only one unknown in $\mathbf{w} = (w)$ so that the stationary condition for equation 2.2 is

$$\frac{\partial \epsilon}{\partial w} = \sum_{i=1}^{5} x_{i1}(x_{i1}w - t_i) = 0. \tag{2.7}$$

Rearranging and solving for $w$ gives

$$w = \overbrace{\frac{1}{\sum_{i=1}^{5} x_{i1}^2}}^{[\mathbf{X}^T\mathbf{X}]^{-1}} \overbrace{\sum_{i=1}^{5} x_{i1}t_i}^{\mathbf{X}^T\mathbf{t}} = 0.182. \tag{2.8}$$

The stationary value $w^* = 0.182\ hr/km$ is at the global minimum indicated by the red star in Figure 2.2b.

### 2.1.3 Regularized Least Squares Solution

If there is a timing error $\delta t_i$ at each of the $i^{th}$ recording stations so that $t_i \rightarrow t_i^0 + \delta t_i$, then equation 2.8 becomes

$$w = \overbrace{\frac{\sum_{i=1}^{5} x_{i1}t_i^0}{\sum_{i=1}^{5} x_{i1}^2}}^{w_o} + \overbrace{\frac{\sum_{i=1}^{5} x_{i1}\delta t_i}{\sum_{i=1}^{5} x_{i1}^2}}^{\delta w}, \tag{2.9}$$

where $w_o$ is the true slowness and $\delta w$ is the slowness error caused by timing errors. If the distances $x_{i1}$ are very small[5] then the denominator in $\frac{1}{\sum x_{i1}^2}$ can be close to zero and so amplify timing errors $\delta t_i > x_{i1}$ into a large slowness error $|\delta w| = |\frac{\sum_{i=1}^{5} x_{i1}\delta t_i}{\sum_{i=1}^{5} x_{i1}^2}| >> 0$. To avoid this *near-zero-divide* instability we add a positive *regularization* term $\eta > 0$ to the denominator so that equation 2.8 becomes the regularized solution:

$$w = \frac{\sum_{i=1}^{5} x_{i1}t_i}{\sum_{i=1}^{5} x_{i1}^2 + \eta}, \tag{2.10}$$

where $\eta$ is typically set to be between 1% and 5% of the largest diagonal value of the matrix $\mathbf{X}^T\mathbf{X}$. This *regularization parameter* $\eta > 0$, also known as the *damping term*, increases the value of the denominator and thereby prevents an unstable near-zero divide and the strong amplification of data errors (Menke, 1984). The cost, however, is a loss of accuracy in the final solution. As discussed in Box 2.1.2, an unstable system of equations can be characterized by several properties: gentle slopes in the objective function $\frac{\partial \epsilon}{\partial w} \approx 0$ and large condition numbers associated with $\mathbf{X}^T\mathbf{X}$. See exercises 1-2.

[5]For example, assume that the five timing stations are spaced at 0.001 km intervals. In this case the timing errors can be very large compared to $x_{i1}$ and so generate enormous slowness errors.

**Box 2.1.2. Small Curvatures and Large Condition Numbers of $\mathbf{X}^T\mathbf{X} \rightarrow$ Unstable Solutions**

The green curves in Figure 2.4 depict a) stable and b) less-stable misfit functions based on the magnitude of $x_{i1}$ in $\mathbf{X}$. The larger value of $x_{i1} = 0.001$ in Figure 2.4a leads to the parabola with steeper sides (green line) compared to the gently dipping one in Figure 2.4b where $x_{i1} = 0.00011$. The flatter green parabola in b) means that there is a wider range of $w$ values, i.e. a more unstable solution, that can give almost the same misfit error as that at the global minimum (green star). Equivalently, this means that a small amount of data noise can lead to a large change in the model. This is an example of an ill-conditioned system of equations that forms the objective function.

If the curvature $\frac{\partial^2 \epsilon}{\partial w^2}$ is close to zero at $\mathbf{w}^*$:

$$\frac{\partial^2 \epsilon}{\partial w^2} = \lim_{\delta w \rightarrow 0} [\frac{\epsilon}{\partial w}\Big|_{w=w^*+\delta w} - \frac{\epsilon}{\partial w}\Big|_{w=w^*}] \approx 0, \tag{2.11}$$

then the slope $\frac{\partial \epsilon}{\partial w}$ hardly changes around the zero-slope point $w^*$. This means that there are many solutions that almost minimize the sum of squared residuals For equation 2.7 the curvature is

$$\frac{\partial^2 \epsilon}{\partial w^2} = \sum_{M=1}^{5} x_{i1}^2, \tag{2.12}$$

which confirms that $\frac{\partial^2 \epsilon}{\partial w^2} \approx 0$ when $|x_{i1}| \approx 0 \; \forall \; i$.

A general indicator of an unstable system of equations is that the condition number $|\lambda_{max}/\lambda_{min}|$ of the matrix $\mathbf{X}^T\mathbf{X}$ is large, where $\lambda_{max}$ ($\lambda_{min}$) is the maximum (minimum) eigenvalue of the matrix. Recall that the $i^{th}$ orthogonal eigenvector $\mathbf{x}_i$ of the symmetric positive definite matrix[6] $\mathbf{X}^T\mathbf{X}$ is defined as

$$\mathbf{X}^T\mathbf{X}\mathbf{x}_i = \lambda_i \mathbf{x}_i, \tag{2.13}$$

where $\lambda_i$ is the eigenvalue for the eigenvector $\mathbf{x}_i$. This means that the solution to the $N \times N$ normal equations 2.5 can be expanded into a sum of $N$ weighted eigenvectors

$$\mathbf{w} = \sum_{i=1}^{N} \beta_i \mathbf{x}_i. \tag{2.14}$$

The unknown coefficients $\beta_i$ can be found by inserting equation 2.14 into the normal equations $\mathbf{X}^T\mathbf{X}\mathbf{w} = \mathbf{X}^T\mathbf{t}$, taking its dot product with the $i^{th}$ eigenvector $\mathbf{x}_i$, and solving for $\beta_i$ to get

$$\beta_i = \frac{(\mathbf{x}_i, \mathbf{X}^T\mathbf{t})}{\lambda_i}. \tag{2.15}$$

This says that if there are small timing errors $\delta\mathbf{t}$ in $(\mathbf{x}_i, \mathbf{X}^T(\mathbf{t} + \delta\mathbf{t}))$ then they will strongly amplified if the $i^{th}$ eigenvalue $\lambda_i$ is small compared to errors associated with much larger eigenvalues. Thus, normal equations with condition numbers $|\lambda_{max}|/|\lambda_{min}|$ larger than about $10^4$ should be regularized.

The damping term $\eta$ in equation 2.10 was introduced in an ad hoc fashion to mitigate large errors in $w$ due to an unstable system of equations. A more rigorous derivation is obtained by defining the objective function to be the weighted sum of data misfit $\mathbf{r}^T\mathbf{r} = (\mathbf{Xw}-\mathbf{t})^T(\mathbf{Xw}-\mathbf{t})$ (green curves in Figure 2.4 ) and model penalty $\mathbf{w}^T\mathbf{w}$ (red curves in Figure 2.4) functions:

$$\begin{aligned}\epsilon &= \overbrace{\frac{1}{2}(\mathbf{Xw}-\mathbf{t})^T(\mathbf{Xw}-\mathbf{t})}^{data\ misfit=\frac{1}{2}\sum_i r_i^2} + \overbrace{\frac{1}{2}\eta\mathbf{w}^T\mathbf{w}}^{model\ penalty=\frac{1}{2}\eta\sum_i w_i^2}, \\ &= \frac{1}{2}\mathbf{w}^T[\mathbf{X}^T\mathbf{X}+\eta\mathbf{I}]\mathbf{w} - \mathbf{t}^T\mathbf{Xw} + \frac{1}{2}\mathbf{t}^T\mathbf{t}, \end{aligned} \tag{2.16}$$

which plots out as the blue curves in Figure 2.4. The derivative of equation 2.16 with respect to $w_k$ yields the $k^{th}$ component of the *misfit gradient*:

$$\begin{aligned}\frac{\partial\epsilon}{\partial w_k} &= \frac{1}{2}\frac{\partial\sum_i r_i^2}{\partial w_k} + \frac{\eta}{2}\sum_i \frac{\partial w_i^2}{\partial w_k}, \\ &= \overbrace{\sum_i x_{ik}r_i}^{(\mathbf{X}^T(\mathbf{Xw}-\mathbf{t}))_k} + \overbrace{\eta w_k}^{(\eta\mathbf{Iw})_k}, \end{aligned} \tag{2.17}$$

where $\mathbf{I}$ is the $N \times N$ identity matrix. Setting this gradient component to zero $\forall k \in [1, 2 \ldots N]$ and rearranging yields the *regularized normal equations*

$$[\mathbf{X}^T\mathbf{X}+\eta\mathbf{I}]\mathbf{w} = \mathbf{X}^T\mathbf{t}, \tag{2.18}$$

which is solved by the regularized least squares solution:

$$\mathbf{w} = [\mathbf{X}^T\mathbf{X}+\eta\mathbf{I}]^{-1}\mathbf{X}^T\mathbf{t}. \tag{2.19}$$

As a special case, setting the derivative in equation 2.17 to zero, renaming $w_k \rightarrow w$ and solving for $w$ gives equation 2.10.

The interpretation of equation 2.16 is that large values of $\eta$ will favor solutions $\mathbf{w}$ that are near the origin, but at the expense of a larger data misfit function $||\mathbf{r}||^2$. As an example, the global minimum $w^*$ (blue star) for the blue curve in Figure 2.4b is closer to the origin than that in Figure 2.4a because $\eta$ is 20% for b) and only 10% for a). However, if $\eta$ is not too large then the damped solution (blue star) is close enough to the undamped solution (green star) with an acceptable loss of precision.

### 2.1.4 Gradient of $\epsilon$

The gradient vector $\nabla\epsilon$ in Figure 2.3 points uphill along the steepest ascent direction. To prove this, define the vector

$$\mathbf{w} = \mathbf{w}_o + \Delta\mathbf{w}, \tag{2.20}$$

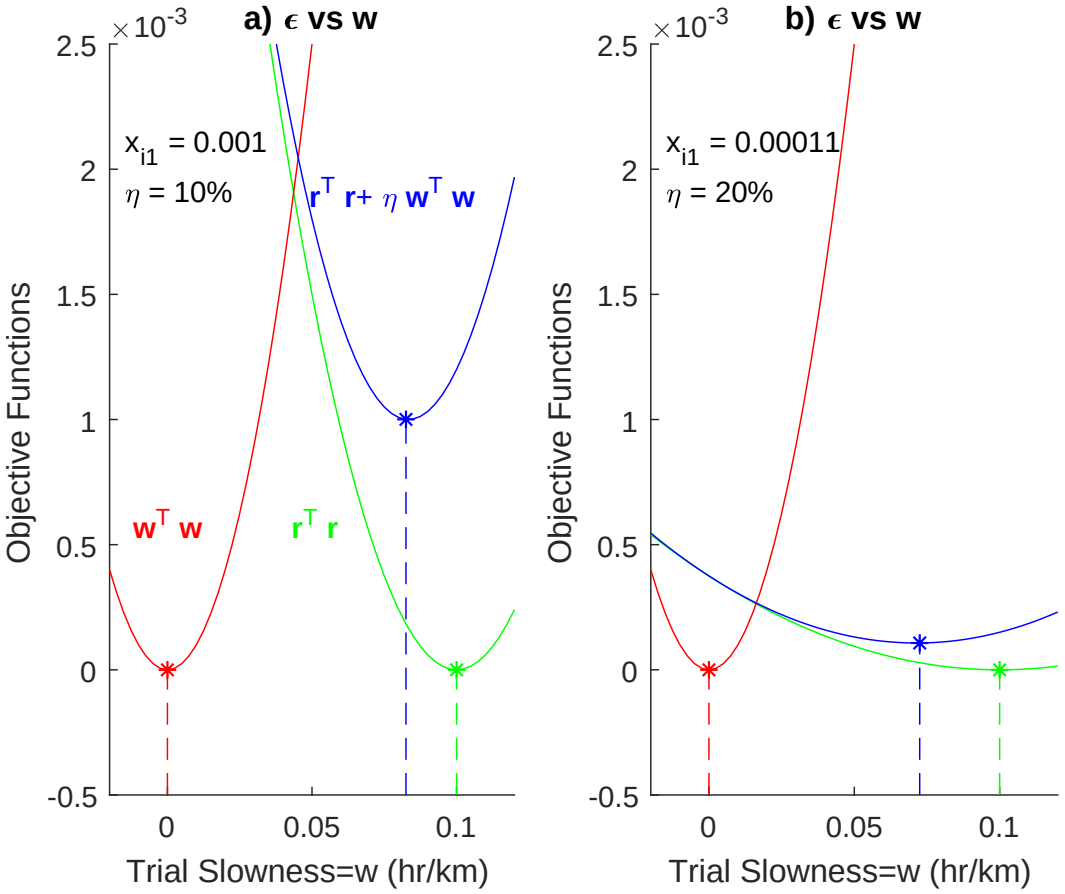


Figure 2.4: Values of the objective functions $\mathbf{r}^T\mathbf{r}$ (green curves), $\mathbf{w}^T\mathbf{w}$ (red curves) and $\mathbf{r}^T\mathbf{r}+\eta\mathbf{w}^T\mathbf{w}$ (blue curves) plotted against trial values of $w$ for the motorcycle equations 2.1. Here, there is no noise in the data so that $(\mathbf{X}^T\mathbf{w}-\mathbf{t})^T(\mathbf{X}^T\mathbf{w}-\mathbf{t})=0$ at the global minimum denoted by the green stars. In contrast, the minimizer $w^*$ for $(\mathbf{X}\mathbf{w}-\mathbf{t})^T(\mathbf{X}\mathbf{w}-\mathbf{t})+\eta\mathbf{w}^T\mathbf{w}$ is non-zero and is closer to $w=0$ for b) $\eta$ =20% compared to a) $\eta$ =10%. Larger values of $\eta$ reward solutions with smaller length $\mathbf{w}^T\mathbf{w}\approx 0$. In addition, the slope of the green parabola is gentler for b) smaller values of $\mathbf{x}_{i1}=.00011$ compared to larger values in a) $\mathbf{x}_{i1}=.001$ in a).

where $\Delta\mathbf{w}$ is a specified vector with very small magnitude. Expanding $\epsilon(\mathbf{w})$ in a Taylor series about $\mathbf{w}_o$, truncating after the first-order term in $\Delta\mathbf{w}$ and rearranging gives:

$$\epsilon(\mathbf{w}) - \epsilon(\mathbf{w}_o) \quad = \quad \nabla\epsilon(\mathbf{w})^T\Big|_{\mathbf{w}=\mathbf{w}_o}\Delta\mathbf{w}, \tag{2.21}$$

where $\nabla\epsilon = (\frac{\partial\epsilon}{\partial w_1}, \frac{\partial\epsilon}{\partial w_2}\ldots\frac{\partial\epsilon}{\partial w_N})^T$ and $\Delta\mathbf{w} = (w_1, w_2\ldots w_N)^T$. If $\Delta\mathbf{w}$ is pointing to a close neighboring point on the same contour that passes through $\mathbf{w}_o$ then $\epsilon(\mathbf{w}_o + \Delta\mathbf{w}) = \epsilon(\mathbf{w}_o)$, which implies $(\nabla\epsilon(\mathbf{w}_o + \Delta\mathbf{w}), \Delta\mathbf{w}) = 0$. This means that $\nabla\mathbf{w}$ is perpendicular to the contour's tangent vector at $\mathbf{w}_o$. Furthermore, if we redefine $\Delta\mathbf{w}$ to be a unit vector now pointing uphill and perpendicular to the contour at $\mathbf{w}_o$ then $\epsilon(\mathbf{w}) - \epsilon(\mathbf{w}_o) = \nabla\epsilon^T\Delta\mathbf{w} = |\nabla\epsilon| > 0$.

As illustrated in Figure 2.3, the gradient vector points uphill along the steepest ascent direction. Therefore, $-\nabla\epsilon$ points downhill along the steepest descent direction as shown in Figure 2.5. This compares to the regularized gradient in equation 2.17 which points more towards the origin in Figure 2.5 because it is a weighted sum of the misfit gradient and the penalty function $\mathbf{w}^T\mathbf{w}$, which points toward the origin. In fact, if $\eta >> 0$ then the penalty term dominates and $\nabla\epsilon$ points at the origin.

The Gauss-Newton arrow $\Delta\mathbf{w}$ in Figure 2.5 points toward the global minimum at $\mathbf{w}^*$ with

$$\Delta\mathbf{w} \quad = \quad [\mathbf{X}^T\mathbf{X}]^{-1}\mathbf{X}^T\Delta\mathbf{t}, \tag{2.22}$$

where $\Delta\mathbf{t} = \mathbf{t} - \mathbf{t}'$, $\Delta\mathbf{w} = \mathbf{w}^* - \mathbf{w}'$ and

$$\mathbf{X}\mathbf{w}' \quad = \quad \mathbf{t}'. \tag{2.23}$$

Equation 2.22 can be proven by recalling that the optimal $\mathbf{w}^*$ for the given traveltimes $\mathbf{t}$ satisfies $\mathbf{w}^* = [\mathbf{X}^T\mathbf{X}]^{-1}\mathbf{X}^T\mathbf{t}$. The sub-optimal model $\mathbf{w}'$ predicts the traveltimes $\mathbf{t}'$ that exactly satisfy equation 2.23 but these traveltimes are not the same as the recorded ones $\mathbf{t}$. Multiplying equation 2.23 by $\mathbf{X}^T$ and inverting gives

$$\mathbf{w}' \quad = \quad [\mathbf{X}^T\mathbf{X}]^{-1}\mathbf{X}^T\mathbf{t}'. \tag{2.24}$$

Subtracting $\mathbf{w}'$ from $\mathbf{w}^* = [\mathbf{X}^T\mathbf{X}]^{-1}\mathbf{X}^T\mathbf{t}$ gives the Gauss-Newton direction $\Delta\mathbf{w} = \mathbf{w}^* - \mathbf{w}'$ where

$$\begin{aligned}\Delta\mathbf{w} \quad &= \quad [\mathbf{X}^T\mathbf{X}]^{-1}\mathbf{X}^T\mathbf{t} - [\mathbf{X}^T\mathbf{X}]^{-1}\mathbf{X}^T\mathbf{t}', \\ &= \quad [\mathbf{X}^T\mathbf{X}]^{-1}\mathbf{X}^T\overbrace{(\mathbf{t} - \mathbf{t}')}^{\Delta t}.\end{aligned} \tag{2.25}$$

Substituting $\Delta\mathbf{w} = \mathbf{w}^* - \mathbf{w}$ into the above equation and rearranging gives the Gauss-Newton update formula for non-linear inversion:

$$\mathbf{w}^* \quad = \quad \mathbf{w}' + [\mathbf{X}^T\mathbf{X}]^{-1}\mathbf{X}^T\Delta\mathbf{t}. \tag{2.26}$$

This formula is the starting point for many gradient descent methods that iteratively search in downhill directions for the global minimum. See section 2.2.

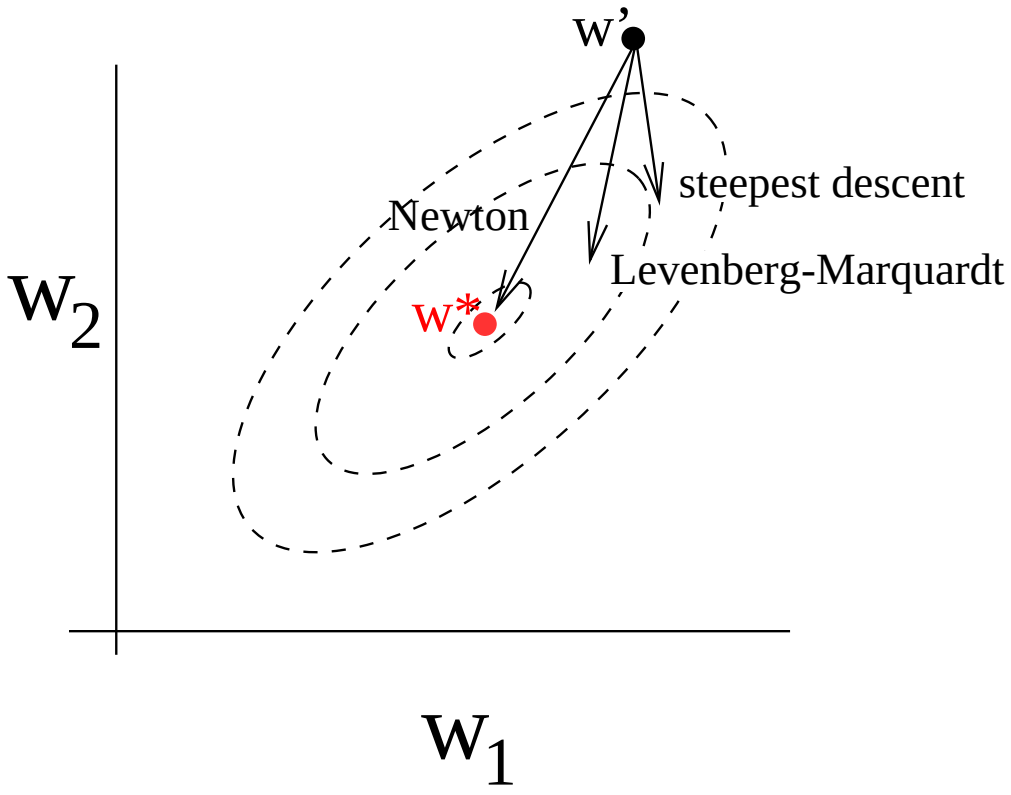


Figure 2.5: Elliptical contours associated with $\epsilon(\mathbf{w})$ and the gradient vectors associated with the Gauss-Newton, Levenberg-Marquardt, and steepest descent directions. Here, the Levenberg-Marquardt gradient is also known as the regularized gradient.

### 2.1.5 Preconditioning

If the number of unknowns $N = 2$ so that $\mathbf{w} = (w_1, w_2)$, then the objective function $\epsilon = 1/2||\mathbf{Xw} - \mathbf{t}||^2$ can be described by the second-order polynomial

$$\epsilon \quad = \quad aw_1^2 + bw_2^2 + cw_1w_2 + dw_1 + ew_2 + f, \tag{2.27}$$

where $(a, b, c, d, e, f)$ are functions of the input data. As a simple example, set $c = d = e = f = 0$ so equation 2.27 can be rewritten as

$$\epsilon = \begin{pmatrix} w_1 & w_2 \end{pmatrix} \begin{bmatrix} a & 0 \\ 0 & b \end{bmatrix} \begin{pmatrix} w_1 \\ w_2 \end{pmatrix}, \tag{2.28}$$

so that $\epsilon$ plots out as the concentric circles in Figure 2.6a for $a = b$, or as the ellipses in in Figure 2.6b for $(a, b) = (1, .01)$. If $a >> b$ then the ellipse become much taller than it is wide and leads to unstable solutions as discussed in Box 2.1.2. For example, a small value of $b$ says that $|\frac{\partial \epsilon}{\partial w_2}|$, $|\frac{\partial^2 \epsilon}{\partial w_2^2}| \approx 0$ so that large changes in $w_2$ lead to small changes in $\epsilon$. Another way of saying this is that small changes in $\delta t$ and, consequently, $\epsilon$ can lead to large changes in $w_2$. In contrast, the steep dip of the ellipse along the $w_1$ axis says that $|\partial \epsilon / \partial w_1| >> 0$ so that $w_1$ is not affected too much by small timing errors.

The $2 \times 2$ matrix $\mathbf{X}$ associated with equation 2.28 is given by

$$\mathbf{X} = \begin{bmatrix} a^{1/2} & 0 \\ 0 & b^{1/2} \end{bmatrix}. \tag{2.29}$$

Here, $\mathbf{X}$ is severely ill-conditioned if $|a^{1/2}| >>> |b^{1/2}|$ so that the associated ellipses of $\epsilon$ appear as long-narrow valleys. To transform these narrow valleys into rounder contours a diagonal *preconditioner matrix* $\mathbf{C}$

$$\mathbf{C} = \begin{bmatrix} a^{-1/2} & 0 \\ 0 & b^{-1/2} \end{bmatrix}. \tag{2.30}$$

can be multiplied by the original equations to give

$$\mathbf{CXw} \quad = \quad \mathbf{Ct}, \tag{2.31}$$

where $[\mathbf{C}]_{ij} = \frac{1}{[\mathbf{X}]_{ii}} \delta_{ij}$. In this case, $\mathbf{CX} = \mathbf{I}$ and has unit-valued eigenvalues $\lambda_1 = \lambda_2 = 1$ where the condition number $|\lambda_2|/|\lambda_1| = 1$. Thus, the associated objective function of $\frac{1}{2}||\mathbf{C}(\mathbf{Xw} - \mathbf{t})||^2$ plots out as round circles.

Normalizing each row of $\mathbf{X}$ by $\mathbf{CX}$ so it has about the same strength as the other ones is one type of preconditioning method[7] for accelerating convergence of iterative gradient methods. Preconditioning the data is also used in neural networks (see Chapter 6) where it is recommended that there should be the same number of data examples for any one type. A similar preconditioning is used for illumination compensation of seismic data (Rickett, 2003) where the weak strength of deep reflections is compensated to balance out the amplitudes of much stronger early arrivals recorded near the source.

[7]Using a diagonal preconditioner is sometimes known as scaling (Nocedal and Wright, 1999)

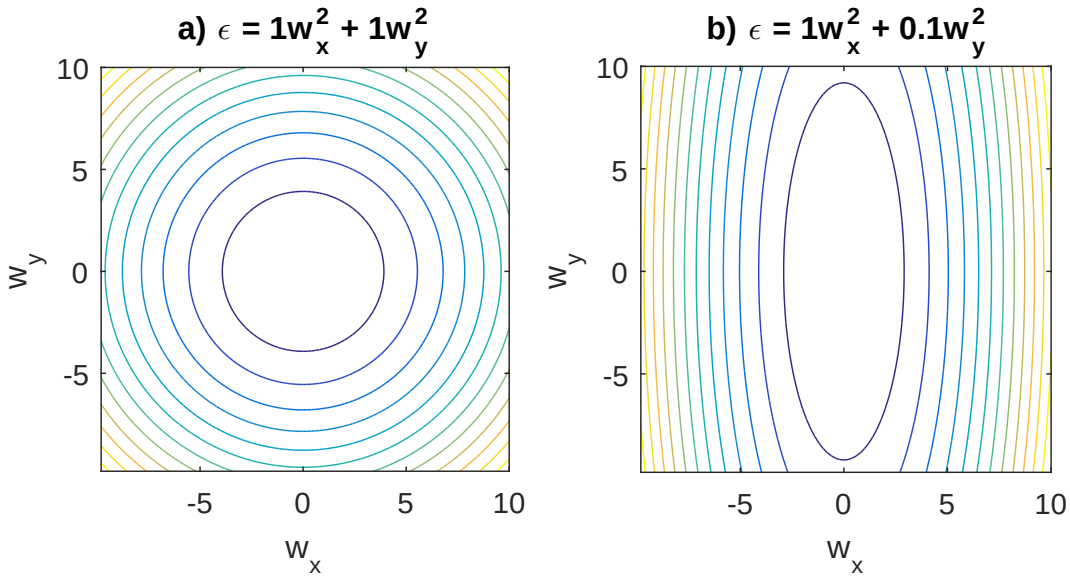


Figure 2.6: Contours of objective function $aw_1^2 + bw_2^2$ for a) $(a, b) = (1, 1)$ and b) $(a, b) = (1, 0.1)$.

If the system of equations is that for the normal equations then, in general, $\mathbf{C}$ is selected so that

$$\mathbf{C}[\mathbf{X}^T\mathbf{X}]\mathbf{C}^{-1}\mathbf{C}\mathbf{w} \quad = \quad \mathbf{C}\mathbf{X}^T\mathbf{t}, \tag{2.32}$$

where the eigenvalues of $\mathbf{C}[\mathbf{X}^T\mathbf{X}]\mathbf{C}^{-1}$ are more favorable for convergence. For example, construct $\mathbf{C}$ so that the condition number of $\mathbf{C}[\mathbf{X}^T\mathbf{X}]\mathbf{C}^{-1}$ is much smaller than that for $[\mathbf{X}^T\mathbf{X}]$. The condition number can also be lowered by clustering the eigenvalues of $\mathbf{C}[\mathbf{X}^T\mathbf{X}]\mathbf{C}^{-1}$ (Nocedal and Wright, 1999).

### 2.1.6 Overfitting Data

Equation 2.1 assumes that the simple one-dimensional model $\mathbf{w} = (w)$ can explain the data. The result is a system of equations that is inconsistent because of, presumably, timing errors. However, the timing expert might disagree and say the data errors are too small to account for deviations from the dashed line in Figure 2.2a: we must have used the wrong physics to explain the data! Instead of the linear model $t = x/v$, a quadratic model with polynomial order $P = 2$ in $x$ might be a better fit where

$$t \quad = \quad w_1 x + w_2 x^2. \tag{2.33}$$

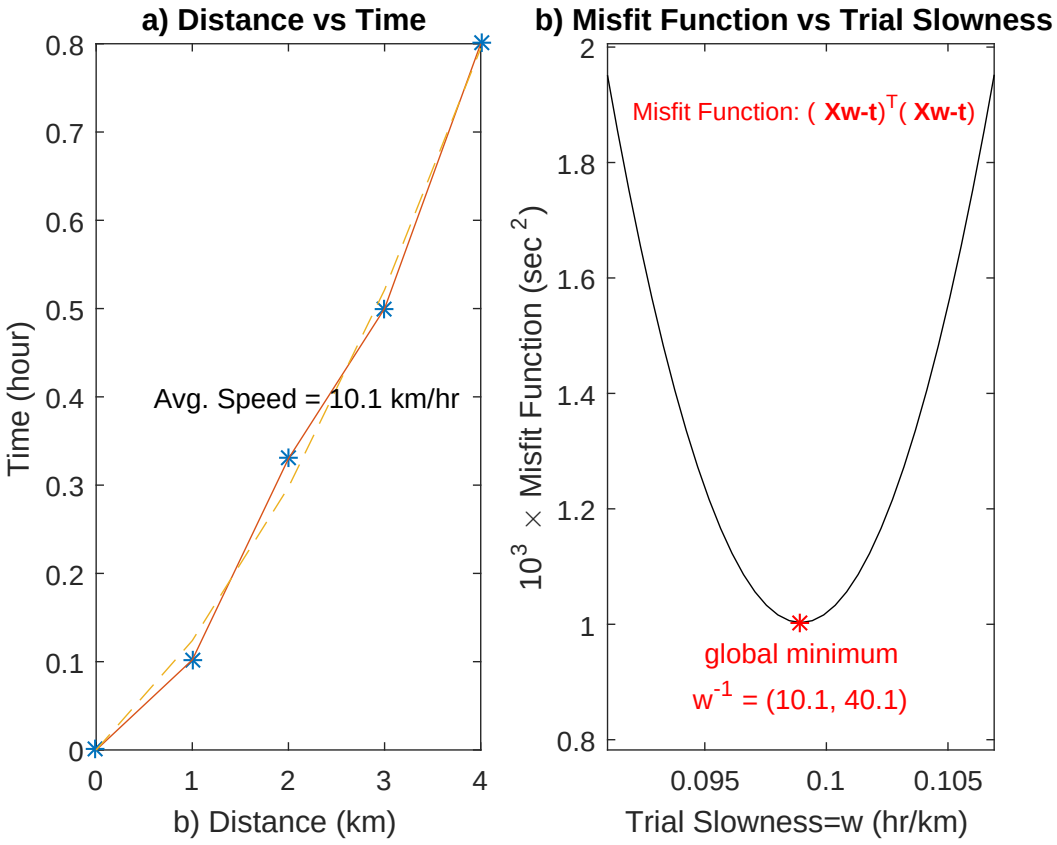


Figure 2.7: Same as Figure 2.2 except the quadratic equation $t = w_1 x + w_2 x^2$ is used to fit the motorcycle data. In this case, $\mathbf{w}^* = (0.099, 0.025) = (1/10.1,\ 1/40.1)$ and $\epsilon$ is only plotted along the slowness component $w_1$, with $w_2 = 0.025$.

In this case, the corresponding system of equations for the motorcycle data becomes

$$\overbrace{\begin{bmatrix} x_{11} & x_{11}^2 \\ x_{21} & x_{21}^2 \\ x_{31} & x_{31}^2 \\ x_{41} & x_{41}^2 \\ x_{51} & x_{51}^2 \end{bmatrix}}^{\mathbf{X}} \overbrace{\begin{pmatrix} w_1 \\ w_2 \end{pmatrix}}^{\mathbf{w}} = \overbrace{\begin{pmatrix} t_1 \\ t_2 \\ t_3 \\ t_4 \\ t_5 \end{pmatrix}}^{\mathbf{t}=observed\ times} \rightarrow \begin{bmatrix} 0 & 0 \\ 1 & 1^2 \\ 2 & 2^2 \\ 3 & 3^2 \\ 4 & 4^2 \end{bmatrix} \begin{pmatrix} w_1 \\ w_2 \end{pmatrix} = \begin{pmatrix} 0 \\ 0.102 \\ 0.33 \\ 0.50 \\ 0.80 \end{pmatrix}. \quad (2.34)$$

The intuitive motivation for this quadratic model is that the motorcycle speed increases towards the end of the road as it becomes more paved. Thus, the quadratic term $w_2 x^2$ dominates when $x$ is large near the end of the road.

As an example, Figure 2.7 depicts the results from fitting the quadratic equation 2.33 to the motorcycle data. Comparing Figures 2.2 and 2.7, the quadratic model gives the closest fit to the data in a) and has about 1/7 the misfit error of the linear model. This result suggests that the misfit error can be further decreased by increasing the order $P$ of the polynomial model. Unfortunately, increasing $P > M$ will yield an *underdetermined*

system of equations where there are more unknowns $P$ than the number $M$ of equations[8]. The least squares solution will then give a misfit error of zero, but at the cost of *overfitting* the data where the complexity of the model exceeds that of the noise-free data. In this case the higher-order polynomials are being fitted to the rapidly varying noise in the data.

Instead of a polynomial, one might suggest a high N-dimensional linear model:

$$t_i = \sum_{j=1}^{N} x_{ij} w_j, \tag{2.35}$$

where $\mathbf{w}$ is the $N \times 1$ model vector and $x_{ij}$ are unphysical weights specified by the mechanic. If these weights are specified to give rise to an $\mathbf{X}^T\mathbf{X}$ matrix that is invertible then this model will also overfit the data. For example, the two equations $w_1 = 1$ and $w_1 = 2$ are inconsistent and generate non-intersecting lines in the coordinate-space $(w_1, w_2)$. Increasing the number of unknowns to include $w_2$ so that the model equations are $w_1 = 1$ and $w_1 + .4w_2 = 2$ gives a consistent set of equations where the associated lines have a common intersection in $(w_1, w_2)$.

### Sanity Test for Overfitting

How can we determine if the complexity parameter $P$ of the proposed model is higher than that in the noiseless data? The machine learning learning community tries to answer this question by randomly dividing the entire training data set into several parts: *training data*, *test data* and *validation data*. One possibility is to use 90% of the original data for validation and determine the best model $\mathbf{w}$ and regularization parameters. The remaining 10% is used for testing. For the motorcycle example this would mean repeating the time trials many times to get a large set of training data.

Wikipedia makes the following claims (*https : //en.wikipedia.org/wiki/Training_test_and_validation_sets*
*Validation datasets can be used for regularization by early stopping: stop training when the error on the validation dataset increases, as this is a sign of overfitting to the training dataset. This simple procedure is complicated in practice by the fact that the validation dataset's error may fluctuate during training, producing multiple local minima. This complication has led to the creation of many ad-hoc rules for deciding when overfitting has truly begun. Finally, the test dataset is a dataset used to provide an unbiased evaluation of a final model fit on the training dataset.*

*Confusingly the terms test dataset and validation dataset are sometimes used with swapped meaning. As a result it has become commonplace to refer to the set used in iterative training as the test/validation set and the set that is used for hyperparameter tuning as the holdout set.*

As an example of tuning the architecture of the model, consider a model $\mathbf{w}$ with complexity $P$ that is first fitted to the validation data, and then it is tested against the holdout data. The misfit error for the holdout data is denoted as $\epsilon(P)$. This procedure is repeated except a model with different complexity value $P'$ is computed from the training data, and

[8] In fact, if $P = N - 1$ and a column of $1's$, also known as the bias vector, is appended to the matrix then its transpose is a Vandermonde matrix (Golub and van Loan, 1996), which is notoriously unstable for large values of $P$.

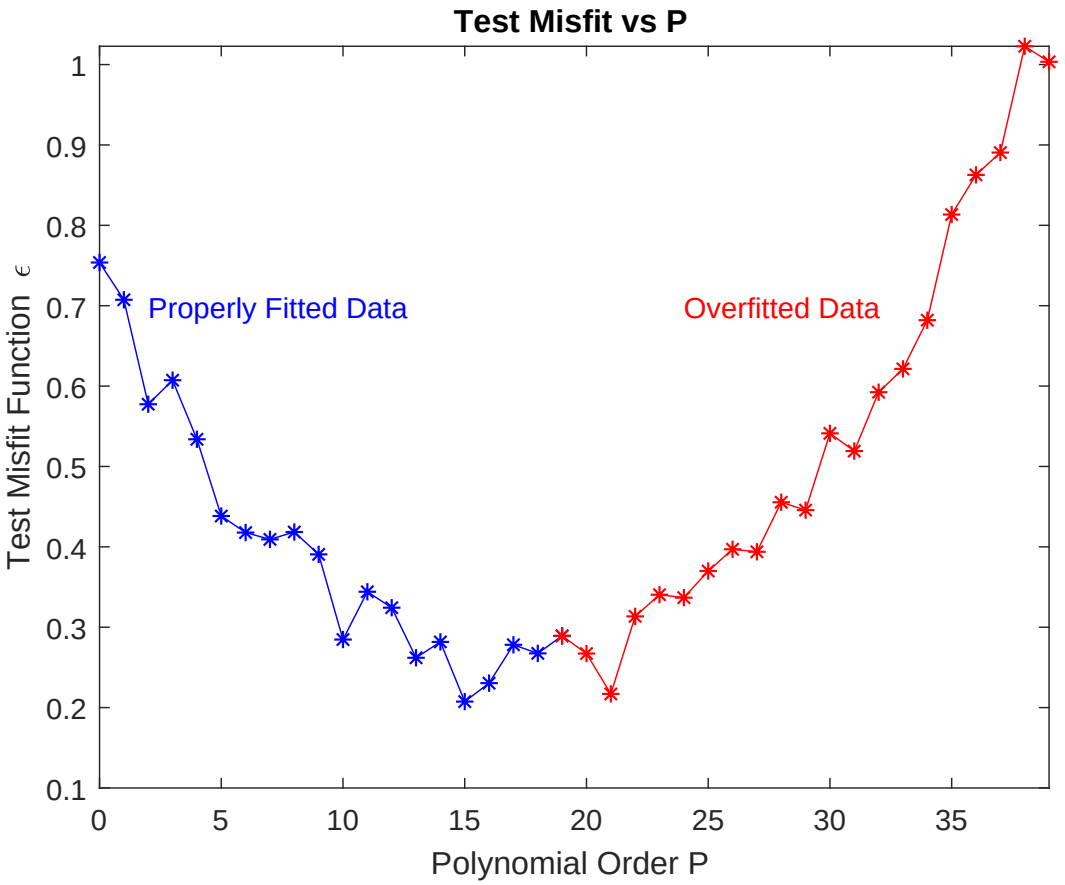


Figure 2.8: Ideal misfit error $\epsilon$ plotted against the order $P$ of the polynomial. The misfit values $\epsilon$ are for the holdout data using the parameters obtained from the distinct training dataset.

then tested against the holdout data to give $\epsilon(P')$. The misfit errors $\epsilon(P)$ are plotted as a function of the complexity parameter $P$, which is shown in Figure 2.8 as an idealized plot.

If the model complexity is less than that of the noiseless data then the misfit error should decrease as $P$ increases as long as the validation data is of the same type as the test data. A counterexample is if the training data were drawn from time trials of road motorcycles while those from the holdout data were drawn from time trials of all-terrain vehicles with four-wheel drive.

If the model complexity $P$ becomes greater than that of the noiseless data, then the optimal model $w^*$ obtained from the training data has largely been fitted to the noise in that data. This is realized on the rightside of the inflection in Figure 2.8 as the misfit error from the holdout data increases. A strategy for dividing up the data into different sets is also used for cross-validation studies (Golub and von Matt, 1997). See *https* : *//en.wikipedia.org/wiki/Training_test_and_validation_sets* for further details.

### 2.1.7 Inclusion of Bias Factor

Assume that the timing clock was not properly calibrated before the motorcycle trial so that the registered starting time consisted of an error of $\Delta\tau = 0.1$ hour for the $x = 0$ position. This time-shift error will be inherited by the times recorded at the other 4 stations. Unless this bias is corrected, the estimated velocity errors will be large.

To account for this timing *bias* the linear model in equation 2.1 should be changed to

$$t = w_0 + w_1 x, \tag{2.36}$$

where $w_0$ is the bias term. In this case the system of equations in equation 2.1 becomes

$$\overbrace{\begin{bmatrix} 1 & x_{11} \\ 1 & x_{21} \\ 1 & x_{31} \\ 1 & x_{41} \\ 1 & x_{51} \end{bmatrix}}^{\mathbf{X}} \overbrace{\begin{pmatrix} w_0 \\ w_1 \end{pmatrix}}^{\mathbf{w}} = \begin{pmatrix} t_1 \\ t_2 \\ t_3 \\ t_4 \\ t_5 \end{pmatrix}, \tag{2.37}$$

where the least squares solution $(w_0^*, w_1^*)$ can now be computed to accurately predict the data.

The importance of accurately estimating the bias term $w_0$ is similar to estimating an accurate starting model for seismic inversion. That is, cycle-skipping problems can be mitigated if an accurate low-wavenumber model is used to account for the bulk of an event's traveltime (Schuster, 2017). As we will discover in later chapters, almost all neural network models include a bias term in their mathematical formulations (Bishop, 2007).

## 2.2 Steepest Descent Optimization

Defining $\mathbf{w}^* \to \mathbf{w}^{(k+1)}$, $\mathbf{w}' \to \mathbf{w}^{(k)}$, $\mathbf{t}' = \mathbf{t}^{(k)}$, and $\Delta\boldsymbol{\tau}^{(k)} = \mathbf{t}^{(k)} - \mathbf{t}$ in equation 2.26 gives the Gauss-Newton iteration formula (Nocedal and Wright, 1999)

$$\mathbf{w}^{(k+1)} = \mathbf{w}^{(k)} - [\mathbf{X}^T\mathbf{X}]^{-1}\mathbf{X}^T\Delta\boldsymbol{\tau}^{(k)}, \tag{2.38}$$

where $\Delta\boldsymbol{\tau}^{(k)} = \mathbf{t}^{(k)} - \mathbf{t}$ is the data-residual vector at the $k^{th}$ iteration. For a linear system of equations with a well-conditioned $[\mathbf{X}^T\mathbf{X}]$, the global minimum should be reached in one iteration. However, the inverse to $[\mathbf{X}^T\mathbf{X}]$ is too expensive to compute for large data sets so its inverse is approximated by the diagonal matrix $[\mathbf{X}^T\mathbf{X}]^{-1}_{ij} \approx \frac{1}{[\mathbf{X}^T\mathbf{X}]^2_{ii}}\delta_{ij}$ to give the preconditioned steepest descent formula:

$$w_i^{(k+1)} = w_i^{(k)} - \alpha\frac{1}{[\mathbf{X}^T\mathbf{X}]^2_{ii}}(\mathbf{X}^T\Delta\boldsymbol{\tau}^{(k)})_i, \tag{2.39}$$

where $\alpha$ is a step length that can be numerically determined by a numerical line search (Gill, 1981) that $\mathbf{w}^{(k+1)}$ reduces the misfit error as much as possible[9]. Unlike the Gauss-Newton formula for linear systems, equation 2.39 does not typically give the right answer

[9]For a linear system of equations, there is an analytic formula for the step length, but it is typically not used for the non-linear problems addressed by the neural network method.

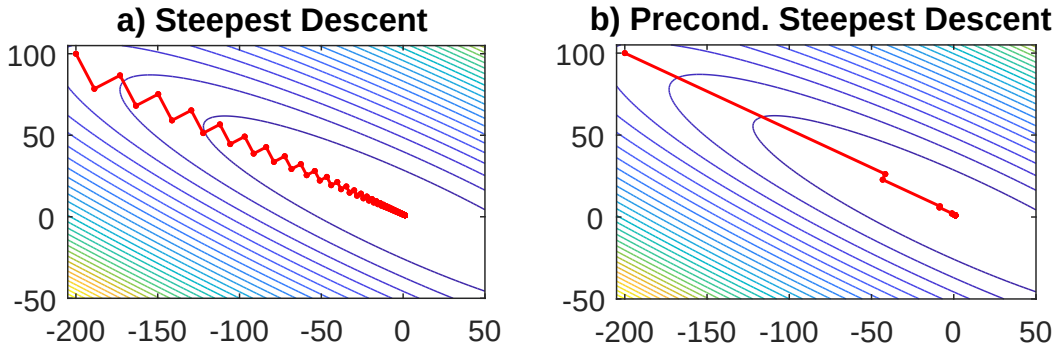


Figure 2.9: Iterative solutions to equation 2.41 by the a) steepest descent and b) preconditioned steepest descent methods.

for one *iteration*. Therefore this formula is used repeatedly to give updated solutions until convergence. The sequence of steps is as follows.

1. Specify $k = 0$ and a starting *guessed* solution $\mathbf{w}^{(0)}$. The starting solution is used to give the predicted traveltimes $\mathbf{w}^{(0)} = \mathbf{t}^{(0)}$ and the residual $\Delta\boldsymbol{\tau}^{(0)} = \mathbf{t}^{(0)} - \mathbf{t}$.

2. Equation 2.39 is used to estimate $\mathbf{w}^{(k+1)}$ after a suitable step length is computed. We will discuss step lengths in the next chapter, but for now conveniently set $\alpha = .01$

3. Compute the new predicted traveltime vector $\mathbf{w}^{(k+1)} = \mathbf{t}^{(k+1)}$ and the residual $\Delta\boldsymbol{\tau}^{(k+1)} = \mathbf{t}^{(k+1)} - \mathbf{t}$. If the length $||\mathbf{r}^{(k+1)}||$ of the residual vector falls below some specified limit, stop. Otherwise redefine $k := k + 1$ and repeat steps 2-3 until convergence.

If regularization and preconditioning are used then the preconditioned-regularized steepest descent formula is (see exercise 11):

$$w_i^{(k+1)} = w_i^{(k)} - \alpha \frac{1}{[\mathbf{X}^T\mathbf{X}]_{ii}^2}(\mathbf{X}^T\Delta\boldsymbol{\tau}^{(k)} + \eta\mathbf{w}^{(k)})_i, \tag{2.40}$$

where $\eta$ is the regularization parameter. The next chapter will examine the properties of the steepest descent method applied to a non-linear system of equations, which is the optimization technique for the neural network method.

Figure 2.9 presents the results of using both the steepest descent and preconditioned steepest descent methods in solving an overdetermined system of equations. In this case the system of equations is given by

$$\begin{bmatrix} 4 & 6 \\ 2 & 5 \\ 0 & 3 \\ 1 & 4 \end{bmatrix} \begin{pmatrix} w_1 \\ w_2 \end{pmatrix} \begin{pmatrix} 10 \\ 7 \\ 3 \\ 5 \end{pmatrix}, \tag{2.41}$$

and it is obvious that the preconditioned steepest descent is more than 5 times faster than the steepest descent method. A fragment of a MATLAB steepest descent code is below.

```
H=X'*X;
for i=1:100
     g=X'*(X*w-t);        % gradient
     alpha=g'*g/(g'*H*g); % step length
     w=w-alpha*g;         % model update
     r=X*w-t;             % residual
     If r*r <=.0001;i=100;end
end
```

where $\mathbf{H} = \mathbf{X}^T\mathbf{X}$ is the Hessian matrix and $\alpha = \mathbf{g}^T\mathbf{g}/(\mathbf{g}^T\mathbf{H}\mathbf{g})$ is the exact step length for an objective function that is exactly quadratic in the model parameters. This step-length formula is derived in Appendix 2.5 but is typically not used if the modeling parameters are not linearly related to the data, which is almost always the case for neural networks or seismic inversion.

## 2.3 Summary

Systems of equations $\mathbf{X}\mathbf{w} = \mathbf{t}$ can be ill-conditioned, inconsistent, and overdetermined. In these cases the regularized least squares solution $\mathbf{w}^*$ minimizes a weighted combination of data misfit $||\mathbf{X}\mathbf{w} - \mathbf{t}||^2$ and penalty $\eta||\mathbf{w}||^2$, where $\eta$ is the regularization parameter. A *bias column vector* is often appended to $\mathbf{X}$ to account for bulk shifts in the input data $\mathbf{t}$. The $\mathbf{w}^*$ solves the regularized normal system of equations in equation 2.16, which is now an $N \times N$ *consistent set of equations*. The solution avoids unstable models far from the origin, but at the loss of some precision for noiseless data. Another way to reduce the condition number is by applying the precondition matrix $\mathbf{C}$ to $\mathbf{X}^T\mathbf{X}$ such that $\mathbf{C}[\mathbf{X}^T\mathbf{X}$
$C^T$ is more well conditioned. For practical reasons, the scaling preconditioner matrix is used which is a diagonal matrix with the $i^{th}$ diagonal element equal to the reciprocal of $[\mathbf{X}^T\mathbf{X}]_{ii}$.

Overfitting can be an issue when the complexity of the hypothetical model becomes greater than that of the theoretical model. This can lead to the problem of the theoretical model explaining both the signal and the noise. One ad hoc remedy is to divide the training data into several data sets, and once the optimal parameters of the model are found on validation/testing data, determine their effectiveness of the holdout data.

The penalty function can be the misfit between an a priori model $\mathbf{w}_0$ and the estimated one, in which case it takes the form $||\mathbf{w} - \mathbf{w}_0||^2$. This is known as a model misfit function. The tradeoff between honoring the data misfit and penalty functions is determined by the value of the regularization parameter $\eta$. The penalty function can be defined to enforce certain characteristics of the model, such as the final model should smoothly vary in the spatial coordinates. In this case the penalty function can be, e.g., $\sum_i (\frac{\partial^n \mathbf{w}}{\partial x_i^n})^2$, where $n$ is the specified order of the spatial derivative.

We learned the meaning of the following terms used in the machine learning (ML) community.

- Ill-conditioned, inconsistent, and overdetermined system of equations.

- Objective function, misfit function, and penalty function. The objective function is often denoted as the *loss* function in the ML community.
- Training data, validation data, and testing data. The training data are divided into several data sets and used to decide which is the best model and regularization parameters should be used without overfitting the data. Details for implementing this anti-overfitting strategy will be given in the chapter in neural networks.
- Normal equations, least squares solution, preconditioning and regularized least squares solutions. Regularized least squares is sometimes denoted as damped least squares.
- The bias term is used to account for bulk shifts in the data and is almost always used for neural network methods.
- Iterative optimization methods, Gauss-Newton method for a linear system of equations, the steepest descent method, and the regularized and preconditioned steepest descent method.

For most geoscience problems, the number of unknowns is too large for a direct inverse to $[\mathbf{X}^T\mathbf{X} + \eta\mathbf{I}]$, which will cost $O(N^3)$ algebraic operations. The alternative is an iterative solution method that costs $O(KN^2)$, where the number of iterations is $K <<< N$, we hope! Regularized and preconditioned gradient optimization, which will be discussed in the next chapter, is the solution method used for neural networks.

## 2.4 Exercises

1. Why don't we allow the regularization parameter $\eta$ in equations 2.9 or 2.10 to be less than zero?
2. For noiseless data where $\delta t_i = 0$, derive the formula for the error $\delta w$ introduced by the damping parameter $\eta = .01x_{i1}$ in equation 2.10. Hint: Recall the approximation $\frac{1}{x^2+\eta} \approx \frac{1}{x^2}(1 - \frac{\eta}{x^2})$ for $\eta << x^2$.
3. Define $\mathbf{w} = \mathbf{w}_o + \alpha\hat{\mathbf{s}}$ where $\alpha$ is a scalar and $\hat{\mathbf{s}}$ is a specified unit vector. Show that $\frac{\partial f(\mathbf{w})}{\partial \alpha} = \hat{\mathbf{s}}^T \nabla f(\mathbf{w})$, which is known as a directional derivative. Explain why $\frac{\partial f(\mathbf{w})}{\partial \alpha}$ gives the slope of $f(\mathbf{w})$ in the direction $\hat{\mathbf{s}}$.
4. Assume a function $f(\mathbf{w})$ that can be approximated by a first-order Taylor series expanded about the point $\mathbf{w}_o$:

$$\begin{aligned} f(\mathbf{w}) - f(\mathbf{w}_o) &= \frac{\partial f(\mathbf{w}_o)}{\partial w_1}\delta w_1 + \frac{\partial f(\mathbf{w}_o)}{\partial w_2}\delta w_2, \\ &= \delta\mathbf{w}^T \nabla f(\mathbf{w}_o), \end{aligned} \tag{2.42}$$

   where $\mathbf{w} = (w_1, w_2)^T$, $\delta\mathbf{w}^T = (\delta w_1, \delta w_2)$ and the magnitude $|\delta\mathbf{w}|$ is very small. If $\mathbf{w}_o$ is on the contour where $f(\mathbf{w}_o) = cnst$ and $\mathbf{w} = \mathbf{w}_o + \delta\mathbf{w}$ is on the same contour then $f(\mathbf{w}) = f(\mathbf{w}_o)$ and $\delta\mathbf{w}$ is parallel to the tangent of the contour at $\mathbf{w}_o$. Therefore, the leftside of the above equation is zero if $\mathbf{w}$ and $\mathbf{w}_o$ are on the same contour. Explain

why this also says that the gradient $\nabla \mathbf{w} = (\frac{\partial f(\mathbf{w})}{\partial w_1}, \frac{\partial f(\mathbf{w})}{\partial w_2})^T$ is perpendicular to the tangent of this contour.

5. Prove that $\nabla f(\mathbf{w}_o)$ points toward increasing values of $f(\mathbf{w})$ by defining $\delta \mathbf{w}$ in equation 2.42 to be pointing uphill where $f(\mathbf{w} = \mathbf{w}_o + \delta \mathbf{w}) > f(\mathbf{w}_o)$. Hint: recall that two vectors point in similar directions if $(\mathbf{x}, \mathbf{y}) = |\mathbf{x}||\mathbf{y}| \cos\theta > 0$.

6. Prove that $\nabla f(\mathbf{w})$ is parallel to the direction of maximum increase in $f(\mathbf{w})$ at $\mathbf{w}$.

7. Define the second-order Taylor expansion of $f(x)$ about $\mathbf{w}_o$ as

$$f(\mathbf{w}) \quad = \quad f(\mathbf{w}_o) + \sum_{i=1}^{2} \frac{\partial f(\mathbf{w}_o)}{\partial w_j} \delta w_i + \frac{1}{2} \sum_{j=1}^{2} \sum_{i=1}^{2} \frac{\partial^2 f(\mathbf{w}_o)}{\partial w_i \partial w_j} \delta w_i \delta w_j, \tag{2.43}$$

and define $\nabla f(\mathbf{w}_o) = (\frac{\partial f(\mathbf{w}_o)}{\partial w_1}, \frac{\partial f(\mathbf{w}_o)}{\partial w_2})^T$. Show that equation 2.43 can be written as

$$f(\mathbf{w}) \quad = \quad f(\mathbf{w}_o) + \delta \mathbf{w}^T \nabla f(\mathbf{w}_o) + \frac{1}{2} \delta \mathbf{w}^T \mathbf{H} \delta \mathbf{w}, \tag{2.44}$$

where

$$\mathbf{H} = [\nabla \nabla^T] f(\mathbf{w}_o) = \begin{bmatrix} \frac{\partial^2 f}{\partial w_1^2} & \frac{\partial^2 f}{\partial w_1 \partial w_2} \\ \frac{\partial^2 f}{\partial w_1 \partial w_2} & \frac{\partial^2 f}{\partial w_2^2} \end{bmatrix}. \tag{2.45}$$

Show that $\hat{\mathbf{s}}^T \mathbf{H} \hat{\mathbf{s}}$ gives the curvature of $f(\mathbf{w})$ along the direction specified by $\hat{\mathbf{s}}$. Hint: Define $\mathbf{w} = \mathbf{w}_o + \alpha \hat{\mathbf{s}}$ so that

$$\frac{d^2 f}{d\alpha^2} \quad = \quad \frac{d}{d\alpha} \frac{df}{d\alpha} = \frac{d}{d\alpha} [(\nabla f)^T \hat{\mathbf{s}}] = \hat{\mathbf{s}}^T [\nabla (\nabla f)^T] \hat{\mathbf{s}} = \hat{\mathbf{s}}^T \overbrace{[\nabla \nabla^T f(\mathbf{w})]}^{Hessian} \hat{\mathbf{s}}. \tag{2.46}$$

8. Can there ever be more than one isolated minimum for a quadratic function? Prove your answer. Can there ever be more than one isolated minimum for a non-quadratic function?

9. Same question as the previous one except find the maximum and minimum curvatures of the Rosenbrock function instead of the slope. What is the curvature along the slope direction?

10. Prove that the gradient vector of $f(\mathbf{x})$ is perpendicular to the contour line tangent at $\mathbf{x}$ and points uphill. Hint: $f(\mathbf{x})$ does not change along a contour, so the directional derivative along the tangent $df(\mathbf{x})/dt \to 0$. This says that $df(\mathbf{x})/dt = d\mathbf{x}^T/dt \nabla f(\mathbf{x}) = 0$.

11. Starting from the regularized objective function, derive equation 2.40.

## 2.5 Appendix: Exact Step Length

The derivation of the step length for the exact line search is now given. The 1D line search problem is defined as finding the optimal value of the scalar $\alpha$ that minimizes $\epsilon(\mathbf{w}+\alpha\Delta\mathbf{w})$ for a fixed $\mathbf{w}$ and $\Delta\mathbf{w}$. Without loss of generality, we can set $\mathbf{w} = 0 + \alpha\Delta\mathbf{w}$ and $\eta = 0$ in equation 2.16 to get

$$\begin{aligned}\epsilon(\mathbf{w}+\alpha\Delta\mathbf{w}) &= \frac{1}{2}\alpha^2(\mathbf{w}+\Delta\mathbf{w})^T[\mathbf{X}^T\mathbf{X}](\mathbf{w}+\Delta\mathbf{w}) - \alpha\mathbf{t}^T\mathbf{X}(\mathbf{w}+\Delta\mathbf{w}) + \frac{1}{2}\mathbf{t}^T\mathbf{t}, \\ &= \frac{1}{2}\alpha^2\Delta\mathbf{w}^T[\mathbf{X}^T\mathbf{X}]\Delta\mathbf{w} + \frac{1}{2}\alpha^2\mathbf{w}^T[\mathbf{X}^T\mathbf{X}]\mathbf{w} + \alpha^2\Delta\mathbf{w}^T[\mathbf{X}^T\mathbf{X}]\mathbf{w} \\ &\quad -\mathbf{t}^T\mathbf{X}(\mathbf{w}+\Delta\mathbf{w}) + \frac{1}{2}\mathbf{t}^T\mathbf{t}. \end{aligned} \tag{2.47}$$

Setting $\eta = 0$ in equation 2.16 and subtracting it from equation 2.47 gives

$$\epsilon(\mathbf{w}+\alpha\Delta\mathbf{w}) - \epsilon(\mathbf{w}) = \frac{1}{2}\alpha^2\Delta\mathbf{w}^T[\mathbf{X}^T\mathbf{X}]\Delta\mathbf{w} + \alpha\Delta\mathbf{w}^T\mathbf{X}^T(\mathbf{X}\mathbf{w}-\mathbf{t}). \tag{2.48}$$

Differentiating the above equation with respect to $\alpha$ and setting the result equal to zero yields the stationary condition where $\epsilon(\mathbf{w}+\alpha\Delta\mathbf{w})$ achieves a minimum along the steepest descent direction $\Delta\mathbf{w}$:

$$\begin{aligned}\frac{\partial\epsilon(\mathbf{w}+\alpha\Delta\mathbf{w})}{\partial\alpha} &= \alpha\Delta\mathbf{w}^T\mathbf{H}\Delta\mathbf{w} + \Delta\mathbf{w}^T\mathbf{g}, \\ &= 0, \end{aligned} \tag{2.49}$$

where the $N \times N$ Hessian matrix is defined as $\mathbf{H} = \mathbf{X}^\mathbf{T}\mathbf{X}$ and the gradient is $\mathbf{g} = \mathbf{X}^T(\mathbf{X}\mathbf{w} - \mathbf{t})$. Solving for $\alpha$ gives the exact step length:

$$\alpha = \frac{-\mathbf{g}^T\Delta\mathbf{w}}{\Delta\mathbf{w}^T\mathbf{H}\Delta\mathbf{w}}. \tag{2.50}$$

This step-length calculation is exact if $\epsilon(\mathbf{w})$ is a quadratic functional. For the steepest descent method $\Delta\mathbf{w} = -\mathbf{g}$ so that

$$\alpha = \frac{\mathbf{g}^T\mathbf{g}}{\mathbf{g}^T\mathbf{H}\mathbf{g}}. \tag{2.51}$$

For non-linear problems, such as neural networks or full waveform inversion, this exact line search formula does not work well because it is suited only for linear problems where the data and model are linearly related. For non-linear problems a numerical line search method is used to find $\alpha$.

# Chapter 3

# Non-Linear Gradient Optimization

Many geophysical problems have a non-linear relationship between the data $\mathbf{t}$ and the actual earth model $\mathbf{w}$. This means that the misfit function is more wiggly than a quadratic function, leading to a misfit function with many local minima. Such holes can easily trap an iterative gradient method into a solution far from the global minimum. In this case non-linear gradient and multiscale optimization methods can be used to sometimes steer clear of such traps and converge to near the global minimum. This chapter now presents an overview of such methods applied to simple problems. Lessons learned can be used to help understand problems with the neural network method. Non-linear optimization is also the starting point for deriving the neural network method, as will be shown in the next Chapter.

## 3.1 Non-linear Gradient Optimization

The previous chapter mainly addressed the linear problem where the data $\mathbf{t}$ were linearly related to the model $\mathbf{w}$ by the modeling equation $\mathbf{Xw} = \mathbf{t}$, where the elements in $\mathbf{X}$ were independent of the model. However, many geophysical problems are non-linear where the $\mathbf{X}$ is a function of the model parameters. For the motorcycle time trials, the modeling equation 2.1 might be non-linear such that

$$xw^3 \quad = \quad t, \tag{3.1}$$

and the goals is to find the optimal value of $w$ that explains the inconsistent timing data. In this case equation 2.1 becomes

$$xw^3 = t \rightarrow \overbrace{\begin{bmatrix} w_1^2 x_{11} \\ w_1^2 x_{21} \\ w_1^2 x_{31} \\ w_1^2 x_{41} \\ w_1^2 x_{51} \end{bmatrix}}^{\mathbf{X}(\mathbf{w})} \overbrace{\begin{pmatrix} w_1 \end{pmatrix}}^{\mathbf{w}} = \overbrace{\begin{pmatrix} t_1 \\ t_2 \\ t_3 \\ t_4 \\ t_5 \end{pmatrix}}^{\mathbf{t}=observed\ times} , \tag{3.2}$$

where $\mathbf{X}(\mathbf{w})$ now depends quadratically on the model $\mathbf{w}$ we are inverting for.

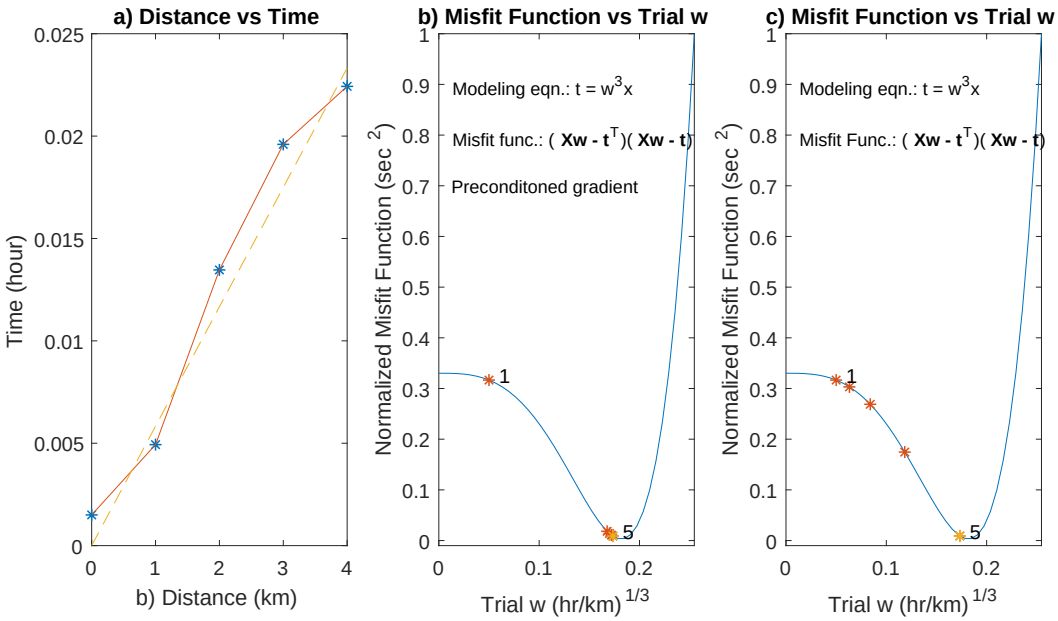


Figure 3.1: Same as Figure 2.2 except the modeling equation is $t = xw^3$ so that the objective function has more than one stationary point. The numbers and stars in b) and c) indicate the iteration numbers and associated misfit errors for solutions determined by the b) non-linear Newton and c) steepest descent methods.

For the motorcycle data, Figure 3.1a plots the recorded noisy times (blue stars) against $x$ for $w = 0.18$, and the dashed line is the predicted data computed with equation 3.1. The value $w = 0.18$ minimizes the misfit function[1] at the vertex of Figure 3.1b. Compared to the single minimum in Figure 2.1, the non-linear misfit function in Figure 3.1b has several minima, the one at $w = 0.018$ and the one at the flat portion of the curve near $w \approx 0$. We say that the one near the origin is a *local minimum* while the one at $\mathbf{w} = 0.18$ is the *global minimum*.

The global minimum in Figure 3.1b can be found by inspection, but how can we find it by the least squares solution $[\mathbf{X}^T\mathbf{X}]^{-1}\mathbf{X}^T\mathbf{t}$ when we need to know $\mathbf{w}$ in order to compute $\mathbf{X}(\mathbf{w})$ and $[\mathbf{X}^T\mathbf{X}]^{-1}$? This is like a short-tailed cat chasing its tail, eternally chasing but not catching.

Luckily, optimization can usually solve this *tail-catching* problem by sequentially computing a series of trial solutions $\mathbf{w}^{(k)}$ $k \in [1, 2, ..]$ where a new search direction $\Delta\mathbf{w}^{(k)} = \mathbf{w}^{(k+1)} - \mathbf{w}^{(k)}$ is defined at each point. This is known as a non-linear gradient optimization method where the search directions use the slope (and sometimes curvature) information of the misfit function at each point.

For example, the iterative non-linear Newton formula with regularization is obtained by by attaching the iteration index $(k)$ to $\mathbf{X}$ in equation 2.40 to give:

$$\mathbf{w}_i^{(k+1)} \quad = \quad \mathbf{w}_i^{(k)} - \alpha[\mathbf{X}^{(k)T}\mathbf{X}^{(k)}]^{-1}(\mathbf{X}^{(k)T}\Delta\boldsymbol{\tau}^{(k)} + \eta\mathbf{w}^{(k)})_i. \tag{3.3}$$

[1]There is a non-zero error at the vertex but it is too small to visible.

In this case the search direction $\Delta\mathbf{w}^{(k)} = \mathbf{w}^{(k+1)} - \mathbf{w}^{(k)}$ is parallel to $-[\mathbf{X}^{(k)T}\mathbf{X}^{(k)}]^{-1}(\mathbf{X}^{(k)T}\Delta\boldsymbol{\tau}^{(k)} + \eta\mathbf{w}^{(k)})$ at the $k^{th}$ point $\mathbf{w}^{(k)}$. The next solution is found by searching along $\Delta\mathbf{w}^{(k)}$ until the misfit function no longer decreases. When this happens at $\mathbf{w}^{(k+1)}$ the new value $\mathbf{x}^{(k+1)}$ is inserted into $\mathbf{X}(\mathbf{w}) \rightarrow \mathbf{X}^{(k+1)}$ and $\Delta\boldsymbol{\tau}^{(k+1)} = \mathbf{X}\tau^{(k+1)}$ is computed to give the updated search direction $-[\mathbf{X}^{(k+1)T}\mathbf{X}^{(k+1)}]^{-1}(\mathbf{X}^{(k+1)T}\Delta\boldsymbol{\tau}^{(k+1)} + \eta\mathbf{w}^{(k)})$. If $[\mathbf{X}^T\mathbf{X}]^{-1}$ is too expensive to compute then we have the preconditioned and regularized steepest descent formula:

$$\mathbf{w}_i^{(k+1)} = \mathbf{w}_i^{(k)} - \alpha \frac{1}{[\mathbf{X}^{(k)T}\mathbf{X}^{(k)}]_{ii}^2}(\mathbf{X}^{(k)T}\Delta\boldsymbol{\tau}^{(k)} + \eta\mathbf{w}^{(k)})_i, \tag{3.4}$$

which avoids calculating the inverse of $[\mathbf{X}^T\mathbf{X}]$.

In summary, the non-linear Newton gradient method is given in 4 steps.

1. Define a starting guess model $\mathbf{w}^{(0)}$ so that $\mathbf{X}^{(0)}$ and $\Delta\boldsymbol{\tau}^{(0)}$ can be computed. Set $k = 0$.

2. Use a numerical line search to find $\mathbf{w}^{(k+1)}$ in equation 3.3.

3. Use $\mathbf{w}^{(k+1)}$ to compute $\mathbf{X}^{(k+1)}$ and $\Delta\boldsymbol{\tau}^{(k+1)}$.

4. Set $k := k + 1$ and repeat steps 2-4 until $||\Delta\boldsymbol{\tau}^{(k)}||$ falls below some user specified threshold. A fragment of the MATLAB code for applying the Newton method to the motorcycle data is shown below.

```
w0=0.18                    % Actual model
x=[0 1 2 3 4 ]';
t=[0 .102 .33 .5 .8 ]';    % Observed noiseless data
e=.3*[.005 -.003 .006 .007 -.003]'; % Data errors
n=3;t=x*w0^n+e;            % noisy input data
w11(1)=.05;                % Starting solution
res     = (x*w11(1).^n-t); % Starting residual
for i=2:nit
    X       =  x*w11(i-1).^(n-1); % Matrix X=x*w^(n-1)
    adjoint =  X'*res;            % Compute adjoint X'(Xw-t)
    w11(i)  =  w11(i-1)-.05*inv(X'*X)*adjoint; % Newton update
    res     = (x*w11(i).^n-t);    %data residual
end
```

The above Newton algorithm is used to explain the motorcycle timing data plotted in Figure 3.1a. In this case we assume the cubic modeling equation in equation 3.2 and iteratively solve for $\mathbf{w}^{(k)}$ from the starting model $\mathbf{w}^{(1)} = 0.05$. Preconditioning in the form of $[\mathbf{X}^T\mathbf{X}]^{-1}$ is used and the step length is set to be 0.01 for the results in Figure 3.1b, and no preconditioning is used for the results in Figure 3.1c. The stars denote the misfit values for the iteration number denoted by the numbers next to each star.

$$\mathbf{w}_i^{(k+1)} = \mathbf{w}_i^{(k)} - \alpha \frac{1}{[\mathbf{X}^{(k)T}\mathbf{X}^{(k)}]_{ii}^2}(\mathbf{X}^{(k)T}\Delta\boldsymbol{\tau}^{(k)} + \eta\mathbf{w}^{(k)})_i. \tag{3.5}$$

## 3.2 Rigorous Derivation of the Newton Formula

The non-linear Newton formula was derived in a hand-waving manner from the linear Newton method for an overdetermined system of linear equations. We will now derive it from a more general point of view by assuming a smooth objective function $\epsilon(\mathbf{w})$ and expanding it in a Taylor's series about the starting point $\mathbf{w}_o$. This approach is much more general and less cumbersome than the approach in Chapter 2 that started from an $M \times N$ system of equations for a finite set of data.

Assume $\epsilon(\mathbf{w})$ is a smooth real function which can be accurately expanded about the starting point $\mathbf{w}_o$ with only three terms in the Taylor series:

$$\begin{aligned} \epsilon(\mathbf{w}_o + \Delta\mathbf{w}) &\approx \epsilon(\mathbf{w}_o) + \sum_{i=1}^{N} \frac{\partial \epsilon(\mathbf{w}_o)}{\partial w_i} \Delta w_i + \frac{1}{2} \sum_{i=1}^{N} \sum_{j=1}^{N} \frac{\partial^2 \epsilon(\mathbf{w}_o)}{\partial w_i \partial w_j} \Delta w_i \Delta w_j, \\ &= \epsilon(\mathbf{w}_o) + \mathbf{g}^T \Delta\mathbf{w} + \frac{1}{2} \Delta\mathbf{w}^T \nabla\nabla^T \epsilon(\mathbf{w}_o) \Delta\mathbf{w} + o(||\Delta\mathbf{w}||^3), \end{aligned} \tag{3.6}$$

where the $N \times 1$ gradient vector is explicitly written as

$$\nabla\epsilon(\mathbf{w}) := \begin{bmatrix} \dfrac{\partial \epsilon}{\partial w_1} \\ \cdot \\ \cdot \\ \cdot \\ \dfrac{\partial \epsilon}{\partial w_N} \end{bmatrix}, \tag{3.7}$$

and the $N \times N$ Hessian matrix $\mathbf{H}$ is defined as $\nabla\nabla^T\epsilon(\mathbf{w})$ with components

$$H_{ij} := [\nabla\nabla^T \epsilon(\mathbf{w})]_{ij} = \frac{\partial^2 \epsilon(\mathbf{w})}{\partial w_i \partial w_j}. \tag{3.8}$$

The matrix $\mathbf{H}$ is symmetric because $\frac{\partial^2 \epsilon(\mathbf{w})}{\partial w_i \partial w_j} = \frac{\partial^2 \epsilon(\mathbf{w})}{\partial w_j \partial w_i}$. It will be shown in the next section that the Hessian matrix contains information about the curvature or type of bumps associated with $f(\mathbf{x})$, while the negative gradient $-\nabla\epsilon(\mathbf{w})$ points in the steepest downhill direction at $\mathbf{w}$.

**Rosenbrock Function.** A geometrical interpretation of the gradient and the Hessian can be illustrated with the Rosenbrock function

$$f(\mathbf{x}) = 100(x_2 - x_1^2)^2 + (1 - x_1)^2, \tag{3.9}$$

a smooth but highly non-linear function plotted in Figure 3.2. In this case the gradient vector becomes

$$\nabla f(\mathbf{x}) = \mathbf{g} = \begin{bmatrix} \dfrac{\partial f}{\partial x_1} \\ \dfrac{\partial f}{\partial x_2} \end{bmatrix} = \begin{bmatrix} -400x_1(x_2 - x_1^2) - 2(1 - x_1) \\ 200(x_2 - x_1^2) \end{bmatrix}. \tag{3.10}$$

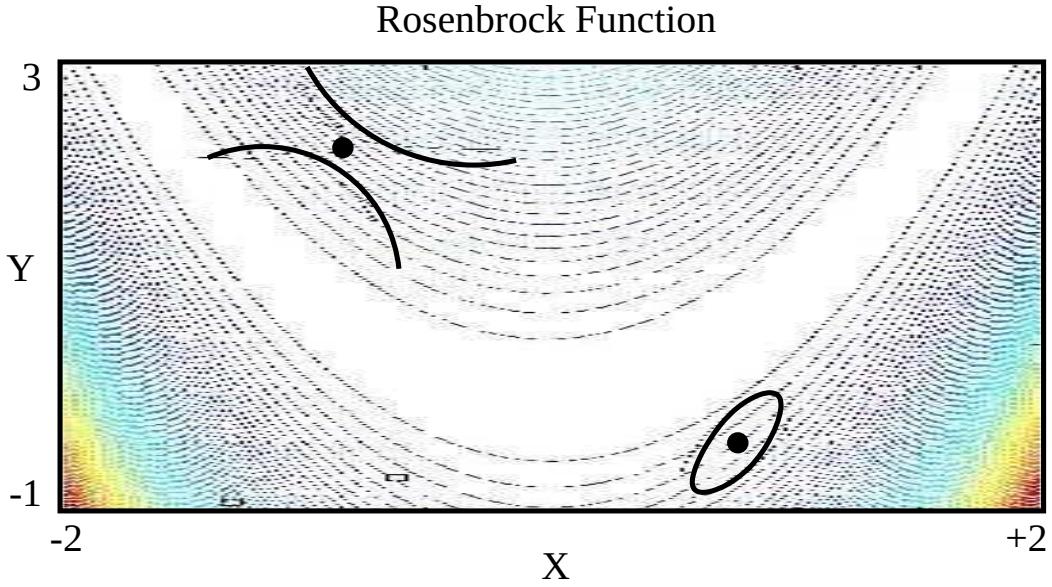


Figure 3.2: Contours of the Rosenbrock function along with icons (thick contours) indicating either a saddle-like or elongated ellipse geometry of $\mathbf{x}^T\mathbf{H}\mathbf{x}$ at the points indicated by solid circles (courtesy of Yunsong Huang). See exercise 1.

To assess the physical meaning of the gradient, we define the points on a line in multidimensional space by

$$\mathbf{x}(\alpha) \quad = \quad \mathbf{x}_o + \alpha\hat{\mathbf{s}}, \tag{3.11}$$

where $\hat{\mathbf{s}}$ is the unit vector parallel to the line and $\alpha$ is the scalar parameter that controls how far $\mathbf{x}$ is from the specified starting point $\mathbf{x}(\alpha = 0) = \mathbf{x}_o$. A directional derivative along the line direction $\hat{\mathbf{s}}$ in equation 3.11 is defined as $df/d\alpha$:

$$\frac{df(\mathbf{x})}{d\alpha} \quad = \quad \sum_i \overbrace{\frac{df(\mathbf{x})}{dx_i}}^{\nabla f(\mathbf{x})} \overbrace{\frac{dx_i}{d\alpha}}^{\hat{\mathbf{s}}} = \hat{\mathbf{s}}^T \nabla f(\mathbf{x}), \tag{3.12}$$

where $\hat{\mathbf{s}}^T\nabla f(\mathbf{x})$ is the projection of the gradient along the direction parallel to $\hat{\mathbf{s}}$. If $\hat{\mathbf{s}}$ is parallel to the contour tangent, then $f(\mathbf{x})$ does not change in this direction so $\hat{\mathbf{s}}^T\nabla f(\mathbf{x}) = 0$. This means that the gradient $\mathbf{g} = \nabla f(\mathbf{x})$ is perpendicular to the contour tangent, or parallel to the direction of steepest descent. For a 1D function, this gradient is also known as the slope: if it is positive then the uphill direction is to the right of the origin, otherwise it is to the left.

The second derivatives of $f(\mathbf{x})$ for the Rosenbrock function form the $2 \times 2$ Hessian

matrix:

$$\mathbf{H} = [\nabla\nabla^T]f(\mathbf{x}) = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1 \partial x_2} \\ \frac{\partial^2 f}{\partial x_1 \partial x_2} & \frac{\partial^2 f}{\partial x_2^2} \end{bmatrix} = \begin{bmatrix} 1200x_1^2 - 400x_2 + 2 & -400x_1 \\ -400x_1 & 200 \end{bmatrix}. \quad (3.13)$$

Unlike a quadratic function with elliptical contours and a constant Hessian matrix, equation 3.13 says that the curvature values in the Hessian matrix depend on $\mathbf{x}$ for $f(\mathbf{x})$ with polynomial order $> 2$.

Similar to defining $\hat{\mathbf{s}}^T\nabla f$ to be the slope of $f(\mathbf{x})$ along the line direction $\hat{\mathbf{s}}$, the second derivative or curvature of $f(\mathbf{x})$ along the same line is given by

$$\frac{d^2 f}{d\alpha^2} = \frac{d}{d\alpha}\frac{df}{d\alpha} = \frac{d}{d\alpha}[(\nabla f)^T\hat{\mathbf{s}}] = \hat{\mathbf{s}}^T[\nabla(\nabla f)^T]\hat{\mathbf{s}} = \hat{\mathbf{s}}^T\overbrace{[\nabla\nabla^T f(\mathbf{x})]}^{Hessian}\hat{\mathbf{s}}. \quad (3.14)$$

Therefore, $\hat{\mathbf{s}}^T\mathbf{H}\hat{\mathbf{s}}$ gives the curvature of $f(\mathbf{x})$ along the direction specified by $\hat{\mathbf{s}}$: that is, it can be used to find how quickly and where the slope is changing most rapidly.

## 3.3 MATLAB Examples of Newton's Method

Newton's methods are now used to solve some specific 1D and 2D non-linear functions. MATLAB codes are provided so the reader can explore finding optimal points for different non-linear functions.

**Example 3.3.1. 1D function**

For a 1D equation **??** becomes

$$\frac{\partial f(x_o)}{\partial x} = -\frac{\partial^2 f(x_o)}{\partial x^2}\Delta x, \quad (3.15)$$

which can be solved for $\Delta x$ to give

$$\Delta x = -\frac{\partial f(x_o)}{\partial x}/\frac{\partial^2 f(x_o)}{\partial x^2}. \quad (3.16)$$

The Newton iteration formula **??** for a 1D non-linear function reduces to

$$x^{(k+1)} = x^{(k)} - \alpha\frac{\partial f(x^{(k)})}{\partial x}/\frac{\partial^2 f(x^{(k)})}{\partial x^2}. \quad (3.17)$$

In general, the convergence rate of Newton's method depends on the topography, determined by the curvature and slope terms, of the function and how far the starting point is from the global minimum.

The MATLAB script which implements Newton's method for the 1D non-quadratic function $f(x) = ax^4 + x^2 - 2x$ is given below,

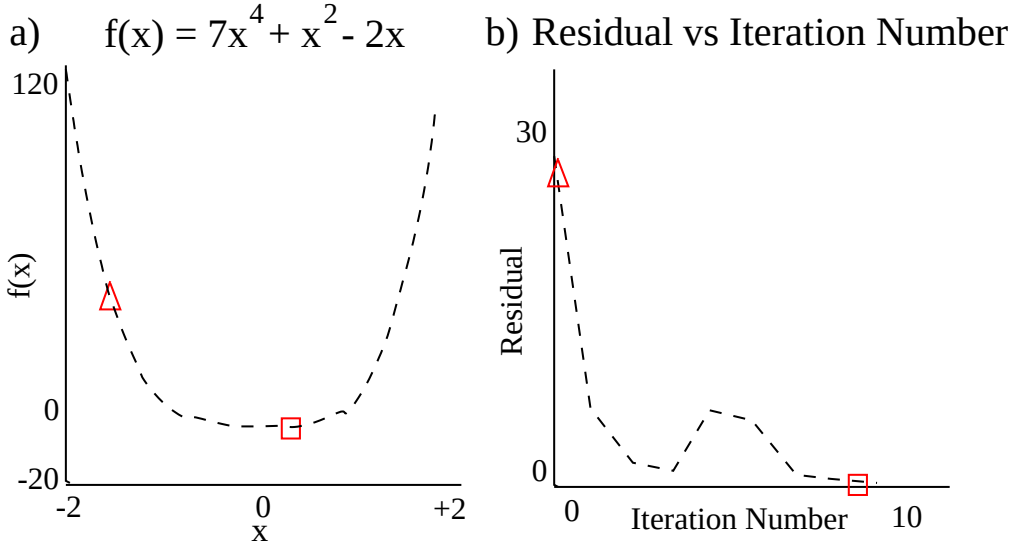


Figure 3.3: Plot of a) quartic function and b) residual vs iteration curve. The diamond (square) represents the starting (ending) model.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% 1D Newton method to find zeros of a quartic function
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
clear all;
a = 7;
subplot(121);x=[-2:.1:2];
plot(x,a*x.^4 + x.^2 - 2*x);
%
x    = -1.5;% starting point
f(1) = (a*x.^4 + x.^2 - 2*x);     % Quartic Function
xx(1)=x;
for it = 2:10                     % Start iterations
 f1prime = a*4*x.^3 + 2*x - 2;
 f2prime = a*12*x.^2 + 2;
 x       = x-f1prime/f2prime;     % Newton formula
 xx(it)  = x;
 f(it)   = (a*x.^4 + x.^2 - 2*x);
 residual(it-1) = abs(f(it)-f(it-1));
end
```

The values of $f(x^{(k)})$ are plotted against $x^{(k)}$ in Figure 3.3.

**Example 3.3.2. 2D Function**

The Rosenbrock function is plotted in Figure 3.2 and a MATLAB code for finding its minimum by Newton's method is below. Unlike a quadratic objective function, the curvature value depends on the location of $(x_1, x_2)$.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Iterative Newton method for finding minimizer
% of Rosenbrock function
% f=100.*(x2-x1.^2).^2+(1-x1).^2
% Minimizer point=(1,1)
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
iter = 10;
x1=-.9;x2=1*x1;
xx = zeros(2,iter);
xx(1,1) = x1;
xx(2,1) = x2;
x=[x1;x2];
%
for it = 2:iter % Begin non-linear iterations
g=[-400*x1*(x2-x1^2)-2*(1-x1); 200*(x2-x1^2)]; % 2x1 gradient vector
H=[1200*x1^2-400*x2+2 -400*x1; -400*x1 200]; % 2x2 Hessian matrix
x = x-inv(H)*g; % Non-linear Newton formula
xx(1,it) = x(1); % Update iterative solution
xx(2,it) = x(2);
x1 = x(1);
x2 = x(2);
end
xmx = max(xx(1,:))+.2; % Define plotting parameters
xmin = min(xx(1,:))-.2;
ymin0 = min(xx(2,:))-.2;
ymx = max(xx(2,:))+.2;
x = [xmin:.01:xmx];
y = [ymin0:.01:ymx];
[X,Y] = meshgrid(xmin:.01:xmx,ymin0:.01:ymx );
f = 100.*(Y-X.^2).^2+(1-X).^2; % Rosenbrock Function
imagesc(x,y,f);
hold on;
contour(X,Y,f,80,'w'); % Plot Rosenbrock function ith contours
hold off
hold on;


plot(x1,x2,'*r');
plot(xx(1,:),xx(2,:),'*-y');
plot(xx(1,iter),xx(2,iter),'r*');
hold off;
xlabel('X');
ylabel('Y');
title('f(X,Y)=100*(Y-X^2)^2+(1-X)^2 and Newton Iterates (Red * = Final)')
```

### 3.3.1 Inexact Newton Method

A variation of the non-linear steepest descent method is to use about five iterations of linear steepest descent where $\mathbf{X}$ is not updated, and then update $\mathbf{X}$ with the most recent estimate of $\mathbf{w}$. This pairing of 5 linear iterations and 1 non-linear iteration is repeated until convergence. Nocedal and Wright (1999) define this procedure as an inexact Newton method because it is similar to using a rough approximation to the inverse Hessian after 5 iterations of linear steepest descent. A similar strategy is used in full waveform inversion for exploration seismology except a linear conjugate gradient method is used (Schuster, 2017).

## 3.4 Multiscale Optimization

An iterative multiscale inversion strategy (Bunks et al., 1995) is often used in seismic inversion to mitigate the problem of getting stuck in a local minimum. The key idea is to start out the iterations with a coarse model and low-pass filtered data. This leads to a simpler objective function with fewer local minima. Iterate for 5 or so iterations until the

residual reduction has stalled. We are now, presumably, closer to the global minimum and the path to it is less cluttered with local minima. Later iterations refine the grid size of the model and incorporate higher frequencies into the data.

As examples, Figures 3.4a-3.4b depict a shot gather and its associated misfit function, respectively. The data were generated for a two-layer model and most of the events seen in the traces are reflection multiples within the top layer. Each trace $d(t)_i$ is quite complicated so that small changes in the upper-layer velocity will lead to significant time shifts in the data. Plotting the misfit function $\epsilon = 1/2 \sum_i \sum_t (d(t)_i - d(t)_i^{pred.})^2$ against trial values of the upper-layer velocity $V$ yields the bumpy misfit plot in Figure 3.4b. Any shift in the velocity that leads to a cycle shift such that the predicted and observed events are roughly in phase will lead to a smaller $\epsilon$ compared to when the predicted traces are mostly out of phase. Thus, the objective function is characterized by many local minima denoted by the red circles in Figure 3.4b.

To reduce the number of local minima the data or model should be simplified. One way to do this is to skeletonize both the data and model. In our case, we replace the waveform data with its traveltime picks in Figure 3.4c. These simple traveltimes are for the primary reflections and lead to the relatively simple misfit function seen in Figure 3.4d. In this case, an iterative gradient method will rapidly converge to a velocity model that is close to the true model. Another example for simplifying the data is to apply a low-pass filter to the data as shown in Figure 3.4e. The associated misfit function in Figure 3.4f is much simpler than the waveform misfit function in Figure 3.4b and so will have less tendency to getting stuck in a local minimum. Another example is in Figure 3.5 where the velocity model is refined every few iterations, and at every model refinement higher frequencies are added to the low-pass filtered data. In this case full waveform inversion is used to invert the seismic traces.

Reducing the complexity of the model and data to enhance convergence is also employed in the neural network community. Here they use the technique of max pooling (see Figure 3.6) where the number of model parameters is halved at one of the stages in the neural network. As will be shown in a numerical example in Chapter 5, this complexity reduction results in better convergence for the example presented.

## 3.5 Diagram for Matrix-Vector Multiplication

In practice, each iteration in gradient optimization typically requires multiplications of matrices and vectors. These are the same operations used for a neural network as illustrated in Figure 3.7. Here, two sets of model parameters are assumed: $w_{1i}$ for the red motorcycle at the top left and $w_{2i}$ for the gray motorcycle at the bottom left. Now there are a total of six unknowns compared to one unknown associated with the one-motorcycle equation **??**. These extra unknowns are introduced to account for more complexity in the model and data. These extra unknowns will demand many more equations of constraint with more time trials over different roads.

The diagrams for representing matrix-vector multiplication in Figure 3.7 is the same that is used in the neural network community. The first column of circles represent the input nodes for the data, the second column denotes the nodes for the neural network for the output elements of the matrix-vector multiplication. However, these output elements

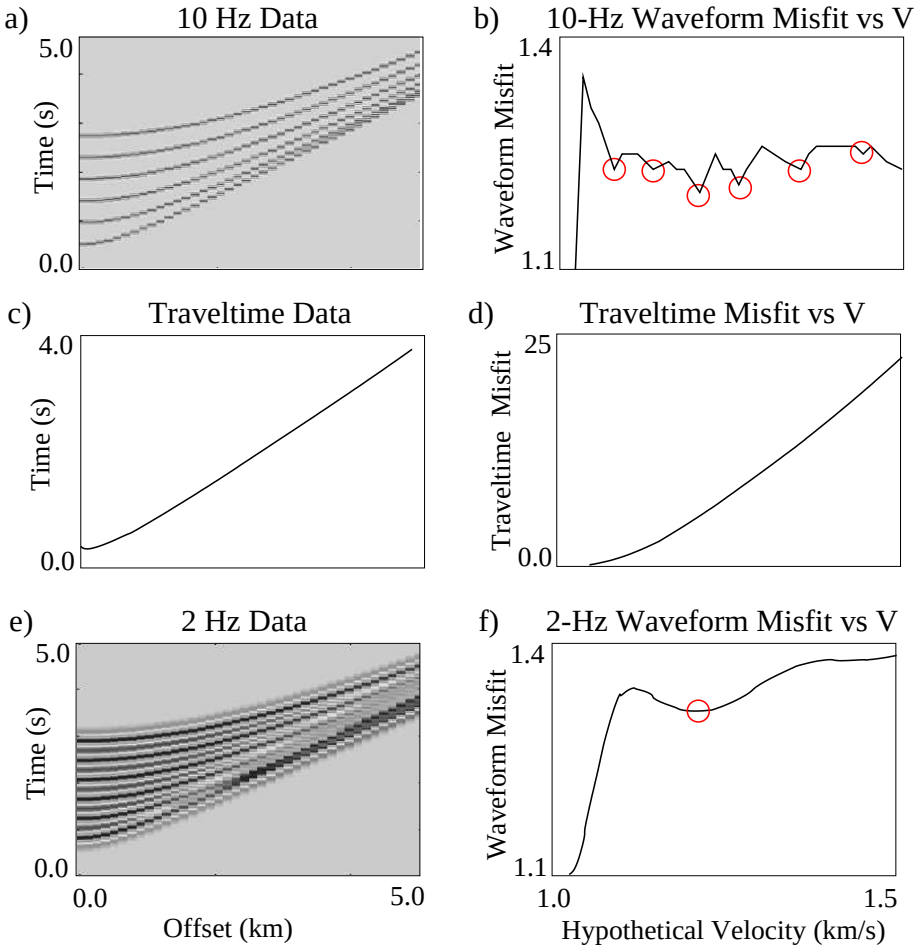


Figure 3.4: Synthetic reflection data in left column, and right column depicts associated plots of misfit functions vs trial velocity values of the first layer in the two-layer model. The correct value of $V$ is 1.0 $km/s$, where the misfit functions tend to zero. Notice the fewer local minima (red circles) for the 2-Hz data than the 10-Hz data.

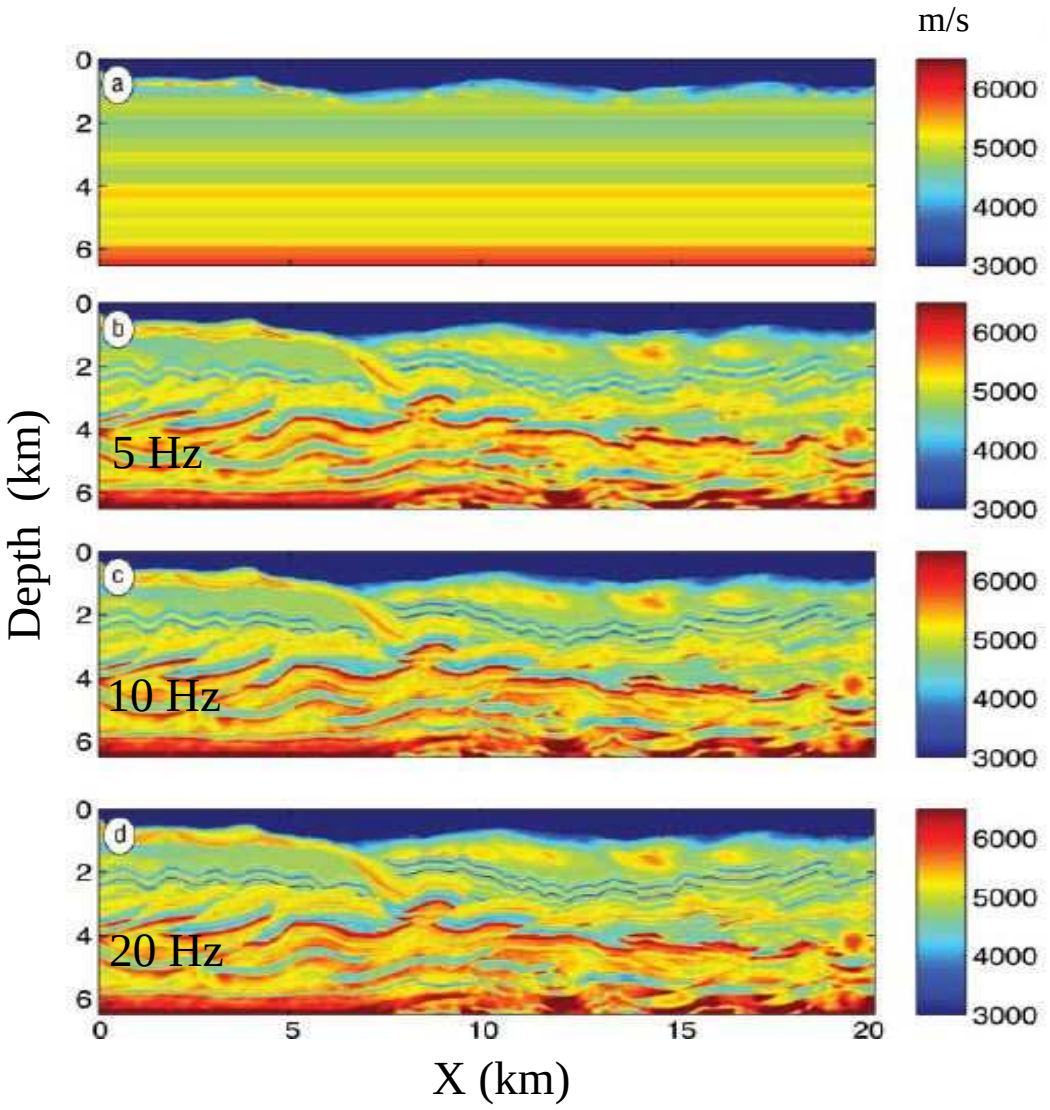


Figure 3.5: a) Initial velocity model. Waveform tomograms inverted from seismic data (204 shots with 1601 receivers per shot) with a peak frequency of b) 5 Hz, c) 10 Hz, and d) 20 Hz. Figure courtesy of Boonyasiriwat et al. (2009).

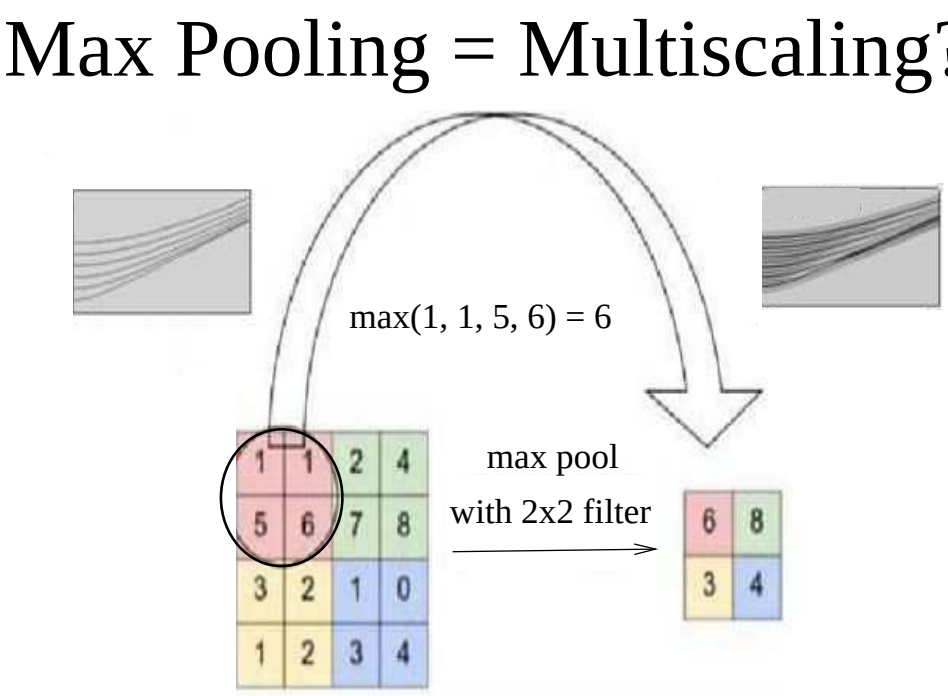


Figure 3.6: Detailed seismic data in the upper left are simplified by low-pass filtering to give the low-pass filtered shot gather in the upper right. The model is also simplified by reducing the number of pixels, which can lead to a simpler objective function and faster convergence. A similar procedure is used for reducing the number of model parameters in a neural network and is denoted as max pooling (see Chapters 5 and 6).

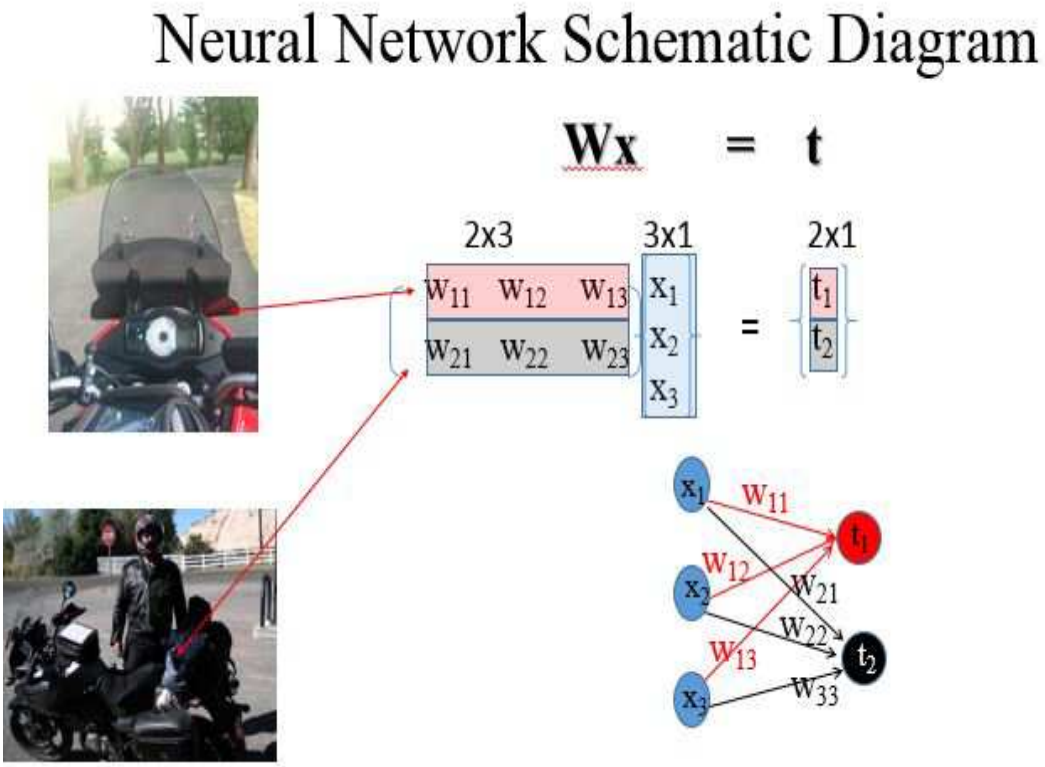


Figure 3.7: Two different types of unknown model parameters are now introduced: $w_{i1} \;\; i \in [1,2,3]$ for the red motorcycle and $w_{i2} \in [1,2,3]$ for the gray motorcycle. Instead of the $1 \times 1$ vector $\mathbf{w}$ of model parameters in equation 2.1, the model parameters are assembled into the $2 \times 3$ matrix where some new parameters have been added to spice up the story.

also undergo an all-or-almost-nothing activation operation briefly described in Chapter 1, with a more detailed explanation in Chapters 5 and 6.

## 3.6 Summary

When the data are non-linearly related to the model then non-linear optimization methods are used. Unlike the linear model $\mathbf{Xw} = \mathbf{t}$, the matrix $\mathbf{X}(\mathbf{w})$ for the non-linear problem is a function of the model parameters $\mathbf{w}$ so it must be updated after each iteration, i.e. $\mathbf{X} \rightarrow \mathbf{X}^{(k)}$. This introduces a new problem: convergence to a local minimum because the actual objective function is higher order than a quadratic polynomial in $w_i \;\; i \in [1, 2 \ldots N]$. The partial cure is multiscale optimization.

The starting point for non-linear gradient methods is to approximate the objective function as a Taylor series truncated after the second-order terms. This truncated series is a sum of 1) an inner product between the $N \times 1$ gradient vector and the search direction $\Delta\mathbf{w}$ and 2) the curvature term represented by a Hessian and its projection along the search direction vector. The solution that minimizes this sum leads to the iterative Newton formula for non-linear optimization.

## 3.7 Exercises

1. The sign and magnitude of the eigenvalue $\lambda_i$ of $\mathbf{H}$ determine the shape of $f(\mathbf{w})$ along the $i^{th}$ coordinate direction. For a 2D geometry this can be shown by setting $\mathbf{w} = \mathbf{w}^* + \alpha\mathbf{e}_1 + \beta\mathbf{e}_2$, where $\mathbf{e}_i$ is the $i^{th}$ orthonormal eigenvector of the symmetric matrix $\mathbf{H}$, $\mathbf{w}^*$ is the point where $f(\mathbf{w})$ is a minimum (i.e., $\mathbf{g}(\mathbf{w}^*) = 0$), and $\alpha$ and $\beta$ are scalars. Expanding $f(\mathbf{w})$ about the minimum point $\mathbf{w}^*$ so that equation 2.44 becomes

$$\begin{aligned} f(\mathbf{w}^* + \alpha\mathbf{e}_1 + \beta\mathbf{e}_2) &= f(\mathbf{w}^*) + [\alpha^2\mathbf{e}_1^T\mathbf{H}\mathbf{e}_1 + \beta^2\mathbf{e}_2^T\mathbf{H}\mathbf{e}_2]/2, \\ &= f(\mathbf{w}^*) + [\lambda_1\alpha^2\mathbf{e}_1^T\mathbf{e}_1 + \lambda_2\beta^2\mathbf{e}_2^T\mathbf{e}_2]/2, \\ &= f(\mathbf{x}^*) + [\lambda_1\alpha^2 + \lambda_2\beta^2]/2, \end{aligned} \tag{3.18}$$

   where the gradient term $\mathbf{g}^T\Delta\mathbf{w}$ is zero at the minimum point $\mathbf{w}^*$. The eigenvalues are positive if $\mathbf{H}$ is a SPD matrix, so any move along an eigenvector direction from $\mathbf{w}^*$ will increase the value of the function. Hence, $f(\mathbf{w})$ describes a bowl-like surface around the minimum point $\mathbf{w}^*$ (see left column of plots in Figure 3.8). Large positive eigenvalues suggest that small changes in position lead to large changes in $f(\mathbf{w})$ so that the bowl has steeply curving sides; conversely, small positive eigenvalues suggest a bowl with gently curving sides.

   If the Hessian is negative definite (i.e., $\mathbf{H}$ has only negative eigenvalues) then equation 3.18 says that any move from $\mathbf{w}^*$ along an eigenvector direction will decrease the function, i.e., $f(\mathbf{w})$ describes an inverted bowl about the *maximal* point $\mathbf{w}^*$. Show that an indefinite Hessian (both positive and negative eigenvalues) has the property that a move along one eigenvector direction will decrease the function value while a move along the other eigenvector direction will increase the function value. This latter surface describes the saddle depicted in 3.8f.

2. Plot the Rosenbrock function with the MATLAB commands:

```
xl=-1.25;xr=1.05;dx=.05; yl=-.3;yr=1.2;dy=.05; facx=1;facy=1;
[xx,yy]=meshgrid([xl:dx:xr]*facx,[yl:dy:yr]*facy);
%[xx,yy]=meshgrid([-0.05:0.005:0.45],[-0.06:0.005:0.12]);
mis=100*(yy - xx.^2).^2 + (1-xx).^2; contour(xx,yy,mis,252)
```

   Now plot the contours for $\partial f/\partial x_1$ and $\partial f/\partial x_2$. Is $f(\mathbf{x})$ a quadratic or non-quadratic function? Are the elements of the Hessian matrix, the components of the gradient, or the curvature along the $x_1$ direction functions of $(x_1, x_2)$? For an objective function that is strictly quadratic, are the elements of the Hessian and gradient a function of spatial position? Same question, except the objective function is a cubic polynomial.

3. Assume $f(\mathbf{x}) = x_1^2 + x_2^2 - 2x_1x_2$. Plot the contours of $f(\mathbf{x})$. Is $f(\mathbf{x})$ a quadratic or non-quadratic function? Are the elements of the Hessian matrix, the components of the gradient, or the curvature along the $x_1$ direction functions of $(x_1, x_2)$? For the above quadratic function, how many minima are there in the objective function? Identify the local and global minimum points for $f(\mathbf{x})$.

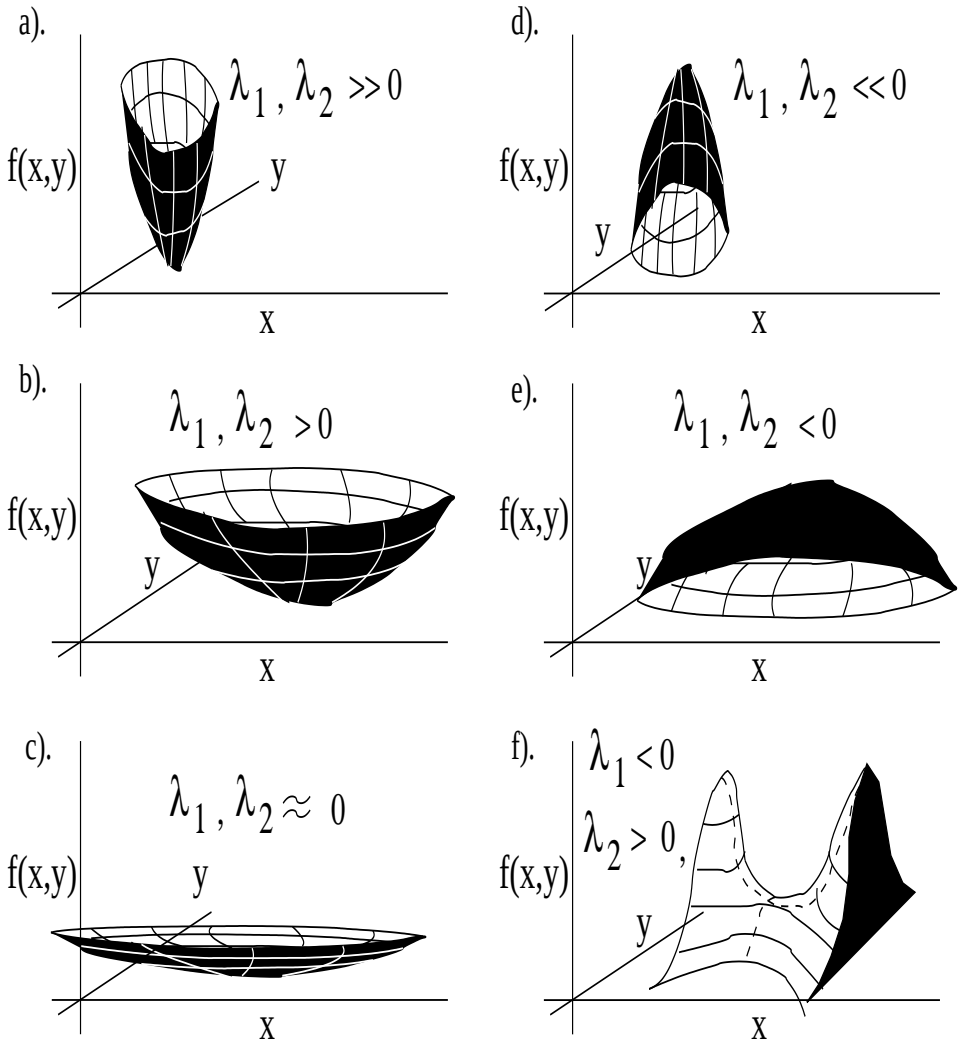


Figure 3.8: Plots of functions with different eigenvalues $\lambda_1$ and $\lambda_2$ for the $2 \times 2$ Hessian matrix. Left column of figures are associated $\lambda_1, \lambda_2 > 0$ while the right column corresponds to examples where at least one of the eigenvalues is negative.

4. Can there ever be more than one isolated minimum for a quadratic function? Prove your answer. Can there ever be more than one isolated minimum for a non-quadratic function?

5. Find the slope of the Rosenbrock function $f(\mathbf{x})$ at $\mathbf{x} = (2, 3)$ along the line direction parallel to the vector $(1, 1)$. Find the slope of $f(\mathbf{x})$ at $(2, 3)$ that is parallel to the vector orthogonal to the vector $(1, 1)$. Show work using the $2 \times 1$ gradient vector.

6. Same question as the previous one except find the maximum and minimum curvatures of the Rosenbrock function instead of the slope. What is the curvature along the slope direction? Is the greatest curvature always along the gradieent direction?

7. Show that a general quadratic function $f(\mathbf{x}) = (1/2)\mathbf{x}^T\mathbf{H}\mathbf{x} + \mathbf{g}^T\mathbf{x} + c$ where $\mathbf{H}$ is symmetric, can also be described as $f(\mathbf{x}) = (1/2)(\mathbf{x} - \mathbf{x}')^T\mathbf{H}(\mathbf{x} - \mathbf{x}') + c'$ where $\mathbf{H}\mathbf{x}' = -\mathbf{g}$ and $c' = c - (1/2)\mathbf{x}'^T\mathbf{H}\mathbf{x}'$. What is the geometrical meaning of this transformation?

8. Design the elements of a $2 \times 2$ Hessian matrix so all of the shapes shown in Figure 3.8 are obtained. Use a MATLAB plotting code similar to

```
[X,Y,Z] = peaks(30);
figure
surfc(X,Y,Z)
```

9. Starting from an objective function with multiple minima, devise a gradient descent method and multiscale strategy to mitigate getting stuck in a local minima. An example of an objective function with local minima is shown below.

```
[X Y Z]=peaks(100);
s=sqrt(X.^2+Y.^2);
surfc(cos(2*pi*s*2)+s);
```

One-dimensional and 2D examples are shown in Figure 3.9 along with the MATLAB script. To compute the element values of the gradient and Hessian you will have to use finite-difference approximations.

```
clear all
x = -1:.05:2;
y = -humps(x);n=length(y);
subplot(121)
plot(x,y)
xlabel('x')
ylabel('humps(x)')
grid on;subplot(122)
[X Y Z]=peaks(n);
yy=y'*y+10;
```

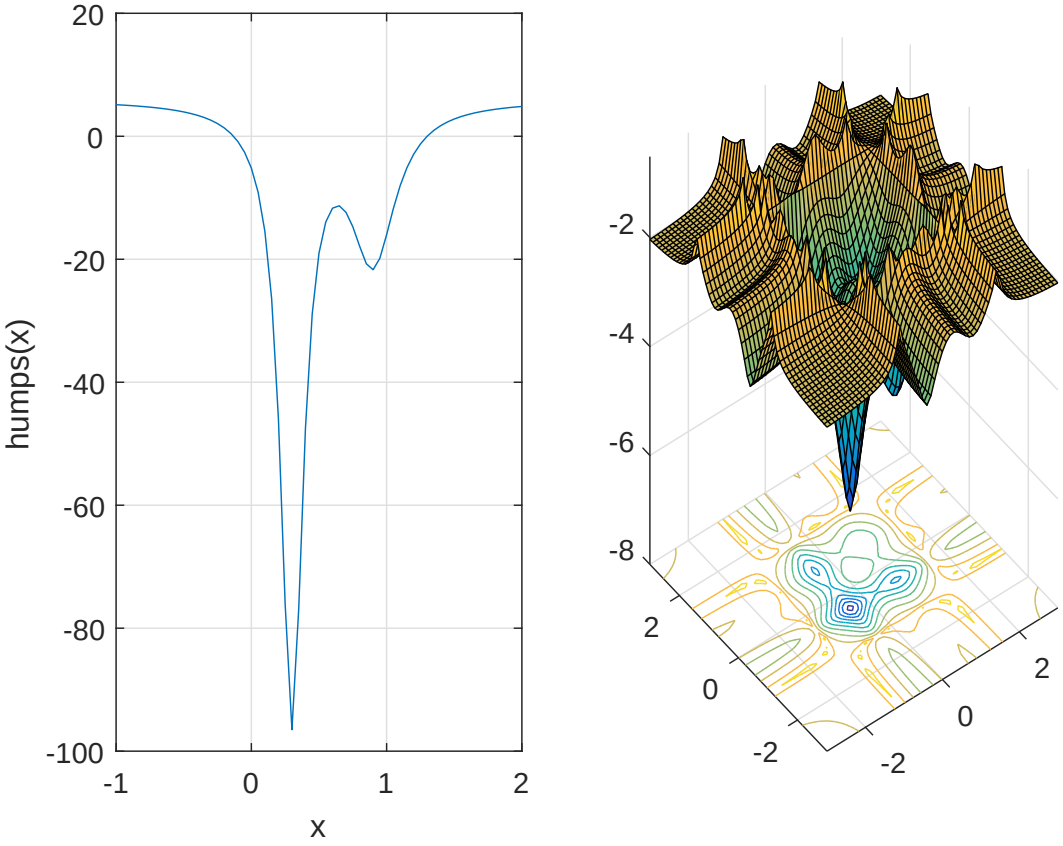


Figure 3.9: Objective functions with local minima.

```
yy=real((yy).^(.2));
surfc(X , Y, -yy)
```

# Chapter 4

# Introduction to Neural Networks

The previous two chapters sought to find the best model parameters in the matrix $\mathbf{W}$ that explains the data vector $\mathbf{t}$, where the elements of $\mathbf{t}$ can take on the value of any real number. The classification problem is similar, except the element values of the predicted vector $\mathbf{t}$ denote the *class* of the input $\mathbf{x}$, where the element values of a $1 \times 1$ vector $\mathbf{t} = t$ might be restricted to be either 1 or 0. For example, $t = 1$ indicates the class of red motorcycles and $t = 0$ indicates the class on non-red motorcycles.

We now introduce the method of neural networks, which is often used as a supervised machine learning method for classifying input images. Unlike the unsupervised methods of cluster analysis, the neural network method finds the optimal $\mathbf{W}$ from a large *training* set of classified data $[(\mathbf{x}^{(1)}, \mathbf{t}^{(1)}), (\mathbf{x}^{(2)}, \mathbf{t}^{(2)}), \ldots]$ such that, for example, $\sum_i ||\mathbf{W}\mathbf{x}^{(i)} - \mathbf{t}^{(i)}||_2$ is minimized. Here, $(\mathbf{x}^{(i)}, \mathbf{t}^{(i)})$ is the $i^{th}$ training *example* and it is denoted as a *supervised training pair* because the input vector $\mathbf{x}^{(i)}$ has been classified and its classification is coded into the sparse target vector $\mathbf{t}^{(i)}$. There are a large number of training examples so that the neural network can estimate the optimal $\mathbf{W}$ that can capture the hidden patterns in these training examples. Once $\mathbf{W}$ is *learned* it can be applied to a new set of data, sometimes denoted as the *holdout data*, which needs to be classified.

## 4.1 Neural Networks

The goal of neural networks is to estimate model parameters $\mathbf{W}$ that non-linearly explain the given training pair of vectors $(\mathbf{x}, \mathbf{t})$. The fundamental modeling equation of neural networks is defined as

$$a_i = g(z_i = \sum_{j=1}^{J} w_{ij} x_j). \tag{4.1}$$

where $a_i$ is the $i^{th}$ element of the $I \times 1$ predicted data vector $\mathbf{a}$. Here, $g(z_i)$ is known as the $i^{th}$ activation function that forms the $I \times 1$ activation vector $\mathbf{a}$ from the $\mathbf{J} \times 1$ input vector $\mathbf{z}$. Its role in the context of classification is to *squash* a wide range of input values $z_i$ into a tiny range of output values, e.g. $-1 \leq a_i \leq 1$. A series of these two elementary operations, matrix-vector multiplication followed by the squashing function, makes up the architecture of the neural network.

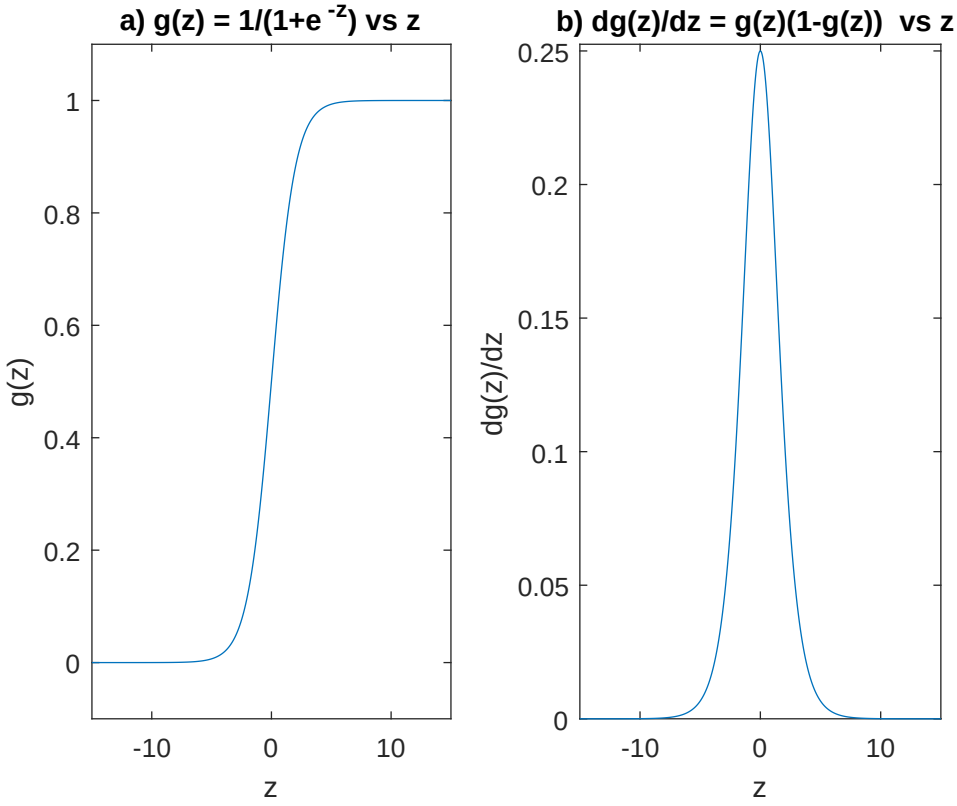


Figure 4.1: a) Sigmoid function and b) its derivative.

A typical shrinking function is the non-linear *sigmoid* threshold function[1] shown in Figure 4.2a and its derivative in Figure 4.2b:

$$g(z) = \frac{1}{1+e^{-z}} \quad and \quad \frac{\partial g(z)}{\partial z} = \frac{e^{-z}}{(1+e^{-z})^2} = g(z)[1-g(z)]. \tag{4.2}$$

This non-linear function is suited for binary classification problems because its *sparsified* output approximates an all-or-nothing function by returning the values 1 and 0 for, respectively, $z$ somewhat larger and somewhat smaller than 0. The output of the squashing function $g(z)$ resembles the activation of a brain's neuron: if the strength of an electrochemical impulse into a neuron is above a certain threshold, the neuron will transmit this information to a neighboring neuron.

The word *network* in *neural networks* indicates that the there is a series, i.e. network, of threshold-matrix-vector operations that are applied to $\mathbf{x}$, as depicted in the Figure 4.2 diagram. Each column of round nodes represents a *layer* and the number in the $i^{th}$ node of the $n^{th}$ layer is denoted as the activation element $a_i^{[}n]$ in equation **??**. The weights next to

[1]The sigmoid function is also known as the logistic function (Bishop, 2006) and is used as a probability density function for binary regression analysis. For example, the probability $P(h)$ for passing an exam increases with the number of hours $h$ studied. Thus, the chance for success is $P(h) = 1/(1+e^{-h})$ and that for failure is $1 - P(h)$.

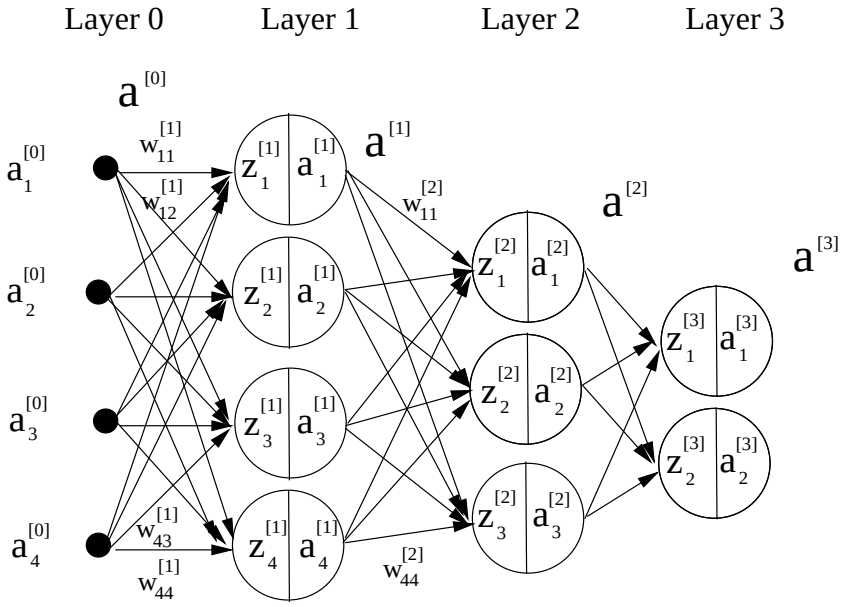


Figure 4.2: Notation for a three-layer, i.e. $N = 3$, neural network with four input nodes at the first layer. The activation output from the $i^{th}$ node in the $n^{th}$ layer is denoted as $a_i^{[n]} = g(z_i^{[n]})$, where $z_i^{[n]} = \sum_j w_{ij}^{[n]} a_j^{[n-1]}$. The input data $\mathbf{x}$ is the activation vector $\mathbf{a}^{[n]}$ for the $0^{th}$ layer and the predicted data are the activation vector $\mathbf{a}^{[3]}$ at the last layer. The layers between the first and last are denoted as hidden layers.

each arrow are the weights of the matrix elements $w_{ij}$, except we append the superscript $[n]$ to indicate the layer number so that equation 4.1 becomes

$$\begin{aligned} z_i^{[n]} &= \sum_{j=1}^{J} w_{ij}^{[n]} x_j^{[n]}, \quad i \in [1, 2 \dots I], \\ a_i &= g(\sum_{j=1}^{J} w_{ij} x_j). \end{aligned} \tag{4.3}$$

Here, the layer number $n$ is denoted as the superscript $[n]$ for both the input $z_i^{[n]}$ and its squashed output $a_i^{[n]} = g(z_i^{[n]})$.

The threshold function is non-linear with respect to the components of $\mathbf{w}$, so the neural network problem to-be-defined below is a non-linear problem[2]. Consequently, the iterative solution can get stuck in a local minimum and requires regularization and multiscale preconditioning such as *max pooling* discussed in the previous chapter.

[2]The operation $\mathbf{w}^T\mathbf{x}$ is a linear operation. However, $g(\mathbf{w}^T\mathbf{x})$ gives an output that is non-linearly related to the parameters $\mathbf{w}$ because it does not pass one of the simple linearity tests $g(\alpha z) = \alpha g(z)$, where $\alpha$ is a scalar.

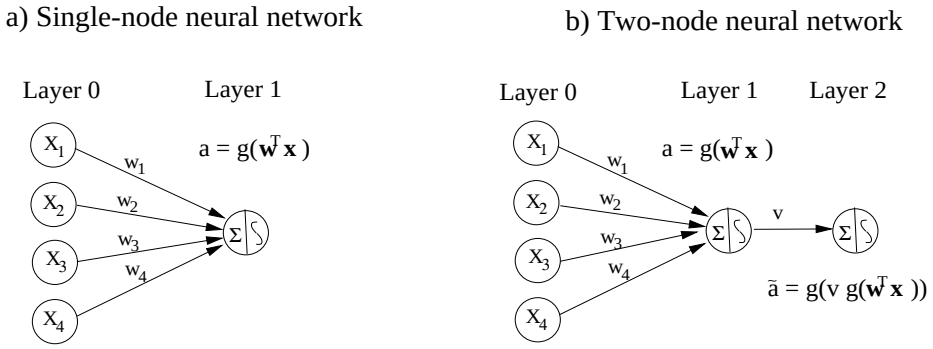


Figure 4.3: Diagrams of a) single-node and b) two-node neural networks.

### 4.1.1 Single-node Neural Network

Assume we are given $M$ training pairs $(\mathbf{x}^{(k)}, t^{(k)})$ of data, where $\mathbf{x}^{(k)}$ is a $N \times 1$ vector and $\mathbf{t}^{(i)} = t^{(k)}$ is the $1 \times 1$ vector with the element value that is either 1 or 0. We define the 1-node classification problem as the following.

$$\begin{aligned}
\textbf{Given:} \quad & (\mathbf{x}^{(k)}, t^{(k)}) \text{ and the model } \overbrace{g(\sum_{j=1}^{J} w_j x_j^{(k)})}^{predicted\ class} = \overbrace{t^{(k)}}^{obs.\ class}, \\
\textbf{Find:} \quad & \mathbf{w} := \arg\min_{\mathbf{w}} \ \tfrac{1}{2} \sum_{k=1}^{K} \overbrace{[g(\sum_{j=1}^{J} w_j x_j^{(k)}) - t^{(k)}]^2}^{g(\mathbf{w}^T \mathbf{x}^{(k)}) - t^{(k)} = \ r^{(k)}}, \\
\textbf{Solution:} \quad & w_i :== w_i - \alpha \frac{\partial \epsilon}{\partial w_i}, \qquad (4.4)
\end{aligned}$$

where $r^{(k)}$ is a scalar rather than a vector for the $k^{th}$ example because the target vector $\mathbf{t} = t$ for this one-node classification problem is a scalar. A diagram of the forward modeling operation is illustrated in Figure 4.3a, where the round node in the middle is divided into two parts: the left half indicates the inner product $\mathbf{z} = \mathbf{w}^T \mathbf{x}$ between $\mathbf{w}$ and the input vector $\mathbf{x}$, and the right indicates the operation of the squashing function on this. This is considered to be a one-layer single node neural network.

The optimal solution to equation 4.4 in the least-squares sense is found by defining the objective function as

$$\epsilon \ = \ \frac{1}{2} \sum_{k=1}^{K} (r^{(k)})^2 = \frac{1}{2} \sum_{k=1}^{K} [g(\overbrace{\mathbf{w}^T \mathbf{x}^{(k)}}^{z^{(k)}}) - t^{(k)}]^2, \qquad (4.5)$$

where the gradient is given by

$$\begin{aligned}\frac{\partial \epsilon}{\partial w_i} &= \sum_{k=1}^{K} r^{(k)} \frac{\partial r^{(k)}}{\partial w_i} \\ &= \sum_{k=1}^{K} (g(\mathbf{w}^T \mathbf{x}^{(k)}) - t^{(k)}) \frac{\partial g(z^{(k)})}{\partial z^{(k)}} \frac{\partial z^{(k)}}{\partial w_i}. \end{aligned} \tag{4.6}$$

Recognizing that $\frac{\partial z^{(k)}}{\partial w_i} = \frac{\partial \mathbf{w}^T \mathbf{x}^{(k)}}{\partial w_i} = x_i^{(k)}$ and substituting equation 4.2 into equation 4.6 gives the gradient for the non-linear steepest descent formula :

$$\mathbf{w}_i := \mathbf{w}_i - \alpha \sum_{k=1}^{K} \overbrace{g(z^{(k)})(1 - g(z^{(k)}) x_i^{(k)}}^{\frac{\partial r^{(k)}}{\partial w_i}} \overbrace{(g(\mathbf{w}^T \mathbf{x}^{(k)}) - t^{(k)})}^{r^{(k)} = k^{th}\ residual}, \tag{4.7}$$

where $\alpha$ is the step length and the computer science notation := redefines $w_i$ on the left side as the update to itself on the right-hand side of the equation. This avoids the introduction of an iteration index, so the fewer the better.

Equation 4.7 is similar to equation 2.4 in that the $i^{th}$ component of the gradient is a dot product between the residual vector and the derivative of the residual with respect to the $i^{th}$ model parameter. For the linearized motorcycle problem the derivative of the residual with respect to the $i^{th}$ model parameter is the $i^{th}$ row vector of the matrix $\mathbf{X}^T$. In contrast, $\partial r_i / \partial w_i$ for the non-linear neural network weights the components of $\mathbf{X}$ by $g(\mathbf{w}^T \mathbf{x})(1 - g(\mathbf{w}^T \mathbf{x}))$ because the modeling equation is non-linear. Thus, $g(\mathbf{w}^T \mathbf{x})(1 - g(\mathbf{w}^T \mathbf{x}))$ must be updated with the new estimate of $\mathbf{w}$ after each iteration in the non-linear steepest descent equation 3.5.

Equation 4.4 only contains one activation function that is applied to each input vector $\mathbf{x}^{(k)}$, so it is denoted as a single-node neural network. This model was initially developed in the context mathematically describing the function of neurons in animals (Rosenblatt, 1958; Hubel and Wiesel, 1962), and later developed into multi-nodal models. The single neuron equation for a single input vector is diagramed in Figure 4.3a, where the circular input nodes are connected to the activation node by solid lines. The weights on each solid line make up the elements of the weight vector $\mathbf{w}$. The two operations of the matrix-vector multiplication and thresholding are denoted by the symbols in the activation node. The two-node neural network is shown in Figure 4.3b, where the layer number is denoted by the superscript in square brackets.

The following MATLAB pseudo-code implements equation 4.7 as the solution to the 1-node neural network problem.

```
% x(N,M)  - input- M input feature vectors with size Nx1
% t(1,M)  - input- M input target vectors with size 1x1
%                  t= 1 or 0
% alpha   - input- step size
%
clear all
```

```
M=100;  % # of equations constraint
N=5;    % # of unknowns (w0, w1, wN)
x=zeros(N,M);t=zeros(M,1);tp=t;
[x,t,alpha,lim,beta,io]=Datain(M,N,x,t); % Create training data
nit=10000;res=zeros(nit,1);
x=[ones(1,M);x];  N=N+1;w=rand(N,1);% start model, Bias Inclusion:
                                    % Append 1st row of x with 1's
%% ----------------------------------------
for k=2:nit         % Looping over gradient iterations
    [grad,res]=gradient(M,N,x,t,w,beta,res,k,io);
    alpha=.01*sqrt(grad'*grad);
    w=w-alpha*grad;  % Gradient descent update
    if res(k)<=lim;k=nit;end
end
Display(M,res,nit,w,x,t,beta,io)
```

and the MATLAB code for the gradient is below.

```
function [grad,res]=gradient(M,N,x,t,w,beta,res,k,io);
% M  - input- # data examples
% N  - input- # unknowns in layer
% t  - input- # observed data
%io=0- input- cross-entropy gradient
grad=zeros(N,1);grade=grad;
residual=zeros(M,1);
for m=1:M        % Loop over M data examples in training set
  z=w'*x(:,m);
  g=1/(1+exp(-z*beta));
  dg=beta*g*(1-g);
  residual(m)=g-t(m);              % residual by Zongcai
  grad0  = dg*x(:,m)*residual(m);  % Stochastic gradient
  grad1  = x(:,m)*residual(m);     % Stochastic cross-entropy gradient
  grad   = grad+grad0;             % Batch gradient
  grade  = grade+grad1;            % Batch gradient
end
res(k)=sqrt(residual'*residual)/M;
```

In this code the input training pairs consisted of $(\mathbf{x}, t)$ where $\mathbf{x}$ is a $5 \times 1$ vector with elements restricted to be either 1 or 0. The output is the binary classification of $t = 1$ if $\mathbf{x}$ contains just a single 1, otherwise $t = 0$. One hundred training examples are used to train the network to give the results shown in Figure 4.4. Here, the predicted values of $t^{pred.} = g(\mathbf{w}^T\mathbf{x})$ never get to be equal to 1 as plotted in Figure 4.4b. Thus one might consider this a failure. However, the correct classifications can be determined by noticing that any predicted classification above the threshold value of 0.4 could be considered as the class denoted as $t = 1$. With this threshold constraint there is almost 100% accuracy in the predicted class shown in Figure 4.4c. Here, the actual class is a red star and the

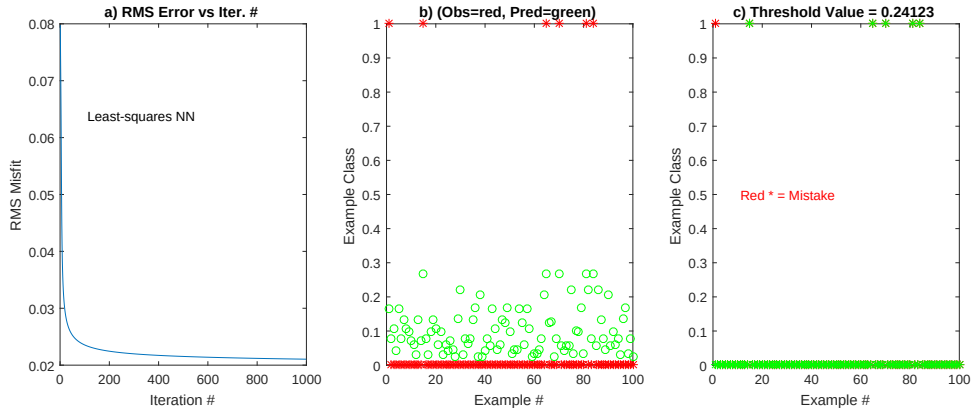


Figure 4.4: a) RMS vs iteration number, b) predicted (green) versus actual (red) classes after 10000 iterations, and c) classes predicted after assigning $t = 1$ to green stars above the threshold of 0.24 in b).

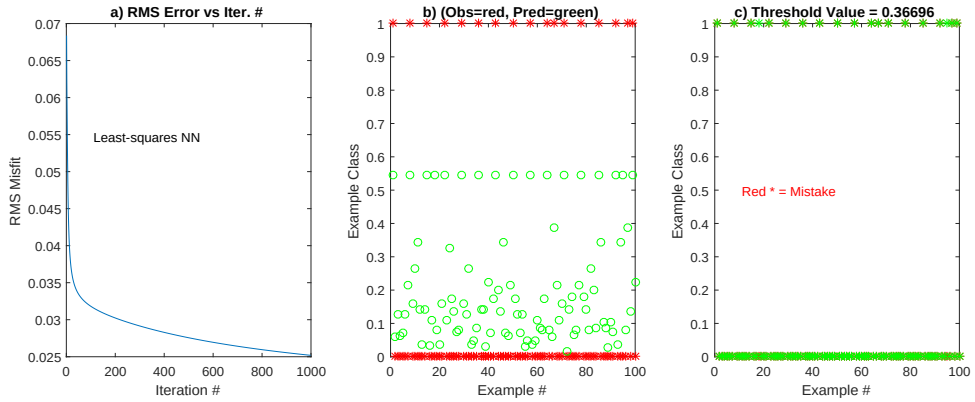


Figure 4.5: Same as Figure 4.4 except the input examples are almost evenly balanced between those with $t = 0$ and $t = 1$. Here the threshold value is 0.36 to get c).

predicted class (after the threshold constraint) is a green star printed over the red stars. If the predictions are correct then the red stars should be completely hidden by green stars, which is largely the case.

To expedite convergence, we must balance out the number of $t = 1$ samples with the number of $t = 0$ examples, otherwise the system of equations can become ill-conditioned as was pointed out in earlier chapters. An unbalanced training set was used for the Figure 4.4 tests and so resulted in a weak separation between the $t = 0$ and $t = 1$ classes in Figure 4.4b. To remedy this problem another test was conducted except the number of $t = 1$ examples were about the same number as the $t = 0$ samples. The results are shown in Figure 4.5. and show a much better separation between the $t = 1$ and $t = 0$ examples.

### 4.1.2 One-node Neural Network with Cross-Entropy Objective Function

Instead of the least squares objective function , the neural network community often uses the cross-entropy objective function for solving the binary classification problem. Empirical evidence suggests it has better convergence properties compared to the least squares approach.

The starting point for the cross-entropy method ($https://rdipietro.github.io/friendly-intro-to-cross-entropy-loss/$) is to assume that the sigmoid function $0 \leq g(z) \leq 1$ can be used as the liklihood function for the binary random variable $t$:

$$P(t|\mathbf{w},\mathbf{x}) \quad = \quad g(\mathbf{w}^T\mathbf{x})^t(1-g(\mathbf{w}^T\mathbf{x}))^{1-t}, \tag{4.8}$$

which says that, given $\mathbf{w}$ and $\mathbf{x}$, the probability for $t=1$ is $g(\mathbf{w}^T\mathbf{x})$, and for $t=0$ it is $1-g(\mathbf{w}^T\mathbf{x})$. Finding the optimal $\mathbf{w}$ that maximizes $P(t|\mathbf{w},\mathbf{x})$ is known as the maximum liklihood solution. Equation 4.8 is in too clumsy of a form so we take the natural logarithm of its negative to get the cross-entropy error function (Bishop, 2006):

$$\epsilon \quad = \quad -t\ln g(\mathbf{w}^T\mathbf{x}) - (1-t)\ln(1-g(\mathbf{w}^T\mathbf{x})). \tag{4.9}$$

Since the logarithm is a monotonic function then the optimal $\mathbf{w}$ that maximizes the liklihood function in equation 4.8 also minimizes $\epsilon$ in equation 4.9.

If we have $K$ training pairs $(\mathbf{x}^{(k)}, t^{(k)})$ then the liklihood of $K$ training pairs having the specific outcome of $(t^{(1)}, t^{(2)} \ldots t^{(K)})$ is the product the individual liklihood functions. Taking the negative logarithm of this product of liklihood functions gives a summation of cross-entropy error functions:

$$\epsilon \quad = \quad -[\sum_{k=1}^{K} t^{(k)}\ln g(\mathbf{w}^T\mathbf{x}^{(k)}) + (1-t^{(k)})\ln(1-g(\mathbf{w}^T\mathbf{x}^{(k)})\ )]. \tag{4.10}$$

For $K=1$, the cross-entropy error function is plotted in Figure 4.6b for $t^{(1)}=1$ and $t^{(1)}=0$. Here, the objective function rises much more steeply than the squared error misfit function in Figure 4.6a for wrong estimates of the solution. This means that the cross entropy misfit function penalizes wrong solutions much more than the squared error one does so that iterative solutions might converge more quickly with the cross-entropy objective function.

The gradient for the cross entropy function is similar to that in equations 4.7 for the least squares objective function, except

$$\begin{aligned}
\frac{\partial\epsilon}{\partial w_i} \quad &= \quad -\sum_{k=1}^{K}\left(\frac{t^{(k)}}{g(\mathbf{w}^T\mathbf{x}^{(k)})} - \frac{1-t^{(k)}}{1-g(\mathbf{w}^T\mathbf{x}^{(k)})}\right)\overbrace{g(\mathbf{w}^T\mathbf{x}^{(k)})(1-g(\mathbf{w}^T\mathbf{x}^{(k)}))x_i^{(k)}}^{eqn.\ 4.2:\ \frac{\partial g}{\partial w_i}}, \\
&= \quad \sum_{k=1}^{K}(\overbrace{g(\mathbf{w}^T\mathbf{x}^{(k)})}^{pred.} - \overbrace{t^{(k)}}^{obs.})x_i^{(k)}.
\end{aligned} \tag{4.11}$$

In this case the predicted data ($g(\mathbf{w}^T\mathbf{x}^{(k)})$ is a squashed version of the original prediction $\mathbf{w}^T\mathbf{x}^{(k)}$.

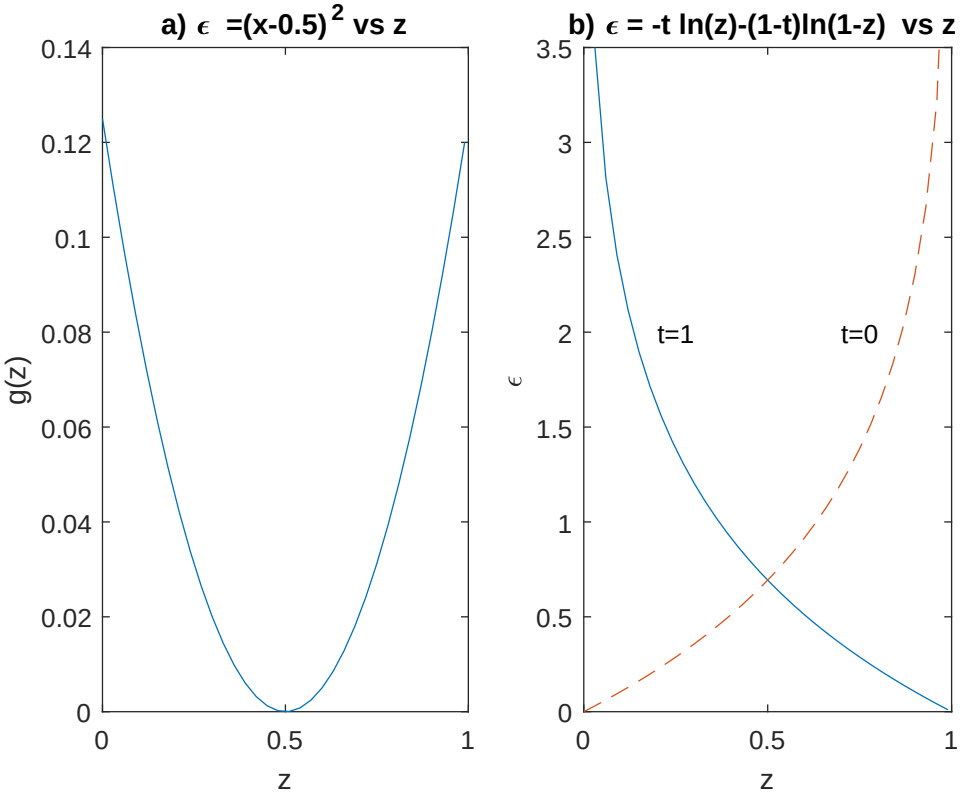


Figure 4.6: Objective functions plotted for a) $L_2$ misfit function $\frac{1}{2}(0.5 - z)^2$ and b) cross-entropy function $\ln tg(z) + (1 - t)\ln[1 - g(z)\ ]$ .

### 4.1.3 Two-node Neural Network

The two-node neural network in Figure 4.3b feeds the weighted output $\theta$

$$\theta = g(\mathbf{w}^T\mathbf{x})v, \tag{4.12}$$

of the single-node neural network in Figure 4.3a into another squashing function to get the class prediction and its derivative:

$$g(\theta) = 1/(1+e^{-\theta}); \qquad \frac{\partial g(\theta)}{\partial \theta} = g(\theta)(1-g(\theta)), \tag{4.13}$$

where the superscript $k$ will be mercifully silent for both $\theta = g(\mathbf{w}^T\mathbf{x}^{(k)})v$ and $z = \mathbf{w}^T\mathbf{x}^{(k)}$. Here, we assume that $g(\theta)$ is a sigmoid function and $v$ is the scalar weight associated with layer two. The least squares misfit function is given by

$$\begin{aligned} \epsilon &= \frac{1}{2}\sum_{k=1}^{K}[\overbrace{g(\theta) - t^{(k)}}^{r^{(k)}}]^2, \\ &= \frac{1}{2}\sum_{k=1}^{K}[1/(1+e^{-\overbrace{g(\mathbf{w}^T\mathbf{x}^{(k)})v}^{\theta}}) - t^{(k)}]^2. \end{aligned} \tag{4.14}$$

where there are two sets of unknowns, the weights $w_i$ in the first layer and $v$ for the second layer. Two gradients now need to be computed, a gradient for the components $w_i$ and one for $v$.

**Gradient wrt $w_i$: $\dfrac{\partial \epsilon}{\partial w_i}$**

The summand in equation 4.14 says that the residual is a function of $\theta = vg(z)$, which in turn is a function of $z = \mathbf{w}^T\mathbf{x}^{(k)}$. Taking the derivative of a function that is a function of another function demands the chain rule. As a simple example, if $f := f(g(h(t)))$ then $\frac{\partial f}{\partial t} = \frac{\partial f}{\partial g}\frac{\partial g}{\partial h}\frac{\partial h}{\partial t}$. We will now use this chain rule to derive the formulas for the gradients of $\epsilon$. In fact, we can use the chain rule recursively if we already labored to compute $\frac{\partial f}{\partial h} = \frac{\partial f}{\partial g}\frac{\partial g}{\partial h}$ at one stage of our calculations, then we can recycle this computation to efficiently compute the derivative $\frac{\partial f}{\partial t} = \frac{\partial f}{\partial h}\frac{\partial h}{\partial t}$. This is an example of left-to-right recursion, which will be called backward recursion.

The formula for $\dfrac{\partial \epsilon}{\partial w_i}$ is

$$\frac{\partial \epsilon}{\partial w_i} = \sum_{k=1}^{K} r^{(k)}\frac{g(\theta)}{\partial w_i}. \tag{4.15}$$

The chain rule says that

$$\frac{\partial g(\theta)}{\partial w_i} = \frac{\partial g(\theta)}{\partial \theta}\frac{\partial \theta}{\partial z}\frac{\partial z}{\partial w_i} = \overbrace{g(\theta)(1-g(\theta))}^{\text{eqn. 4.13: } \frac{\partial g(\theta)}{\partial \theta}}\overbrace{g(z)(1-g(z))v}^{\text{eqn. 4.17: } \frac{\partial \theta}{\partial z}}\overbrace{x_i}^{\frac{\partial z}{\partial w_i}}, \tag{4.16}$$

where

$$z = \mathbf{w}^T\mathbf{x}^{(k)}; \frac{\partial z}{\partial w_i} = x_i; \quad \theta = g(\overbrace{\mathbf{w}^T\mathbf{x}^{(k)}}^{z})v; \quad \frac{\partial \theta}{\partial v} = g(z); \; \frac{\partial \theta}{\partial z} = g(z)(1-g(z))v. \quad (4.17)$$

Plugging equation 4.16 into equation 4.15 gives

$$\frac{\partial \epsilon}{\partial w_i} = \sum_{k=1}^{K} r^{(k)} g(\theta)(1-g(\theta))g(z)(1-g(z))v x_i^{(k)}. \quad (4.18)$$

**Gradient wrt $v$: $\frac{\partial \epsilon}{\partial v}$**

Taking the derivative of $\epsilon$ in equation 4.14 wrt $v$ gives

$$\begin{aligned} \frac{\partial \epsilon}{\partial v} &= \sum_{k=1}^{K} r^{(k)} \frac{\partial r^{(k)}}{\partial g(\theta)} \frac{\partial g(\theta)}{\partial \theta} \overbrace{\frac{\partial \theta}{\partial v}}^{eqn.\ 4.17}, \\ &= \sum_{k=1}^{K} r^{(k)} \overbrace{g(\theta)(1-g(\theta))g(z)}^{\frac{\partial r^{(k)}}{\partial v}}. \end{aligned} \quad (4.19)$$

Plugging $\frac{\partial r^{(k)}}{\partial v} = g(\theta)(1-g(\theta))g(z)$ from equation 4.19 into equation 4.18 gives

$$\frac{\partial \epsilon}{\partial w_i} = \sum_{k=1}^{K} r^{(k)} \frac{\partial r^{(k)}}{\partial v}(1-g(z))v x_i^{(k)}, \quad (4.20)$$

which says that the gradient $\partial r^{(k)}/\partial v$ of the residual computed at a higher-order layer can be reused for the gradient calculation at the next lower-order layer. This is called backward recursion because we are reusing the calculation of the higher-order gradient for the gradient computation at the next lower-order layer. Chapter 5 derives the recursive back-propagation formula for a general neural network, which results in a significant savings in computational costs.

### 4.1.4 Multiple-node and Multiple-layer Neural Network

In general, the one-node neural network cannot emulate complex relationships between the input feature vectors and the target vectors (Bishop, 2006). However, multi-node and multi-layer neural networks have been found to be universal approximators (Hornik et al., 1989; Hornik, 1991; Ripley, 1996).

Therefore we need to increase the complexity of the neural network by incorporating more nodes and layers into the network (see Figure 4.2). This means that we will have to use a more general notation that includes identification of layer numbers and node numbers. As a simple example, Figure 4.2 depicts a neural network with three layers, an input layer

and several nodes per layer. In the previous chapters we denoted the input or as the $N \times 1$ feature vector $\mathbf{x}$ and the last output as the vector $\mathbf{t}$; each element of $\mathbf{x}$ occupied a different input node in this vertical layer of $N$ nodes. We now denote this starting vertical layer of nodes as the $0^{th}$ layer, $\mathbf{x} \rightarrow \mathbf{a}^{[0]}$ and the last $N^{th}$ column of output nodes as $\mathbf{t} \rightarrow \mathbf{a}^{[N]}$.

To summarize, the output activation function from the $i^{th}$ node at the $n^{th}$ layer is denoted by

$$a_i^{[n]} = g(z_i^{[n]}) \qquad \rightarrow activation\ function\ output\ at\ i^{th}\ node\ in\ layer\ n, \tag{4.21}$$

where

$$z_i^{[n]} = \sum_j w_{ij}^{[n]} a_i^{[n-1]} \qquad \rightarrow weighted\ sum\ inputs\ at\ i^{th}\ node\ in\ layer\ n. \tag{4.22}$$

For convenience the bias term is absorbed into the weights so that the input feature vector is $\mathbf{a}^{[0]} = [1,\ x_1, x_2, ..x_M]$; a bias term is included for most of the layers. The activation function is denoted as $g(z_i^{[n]})$ for the input scalar $z_i^{[n]}$ at the $i^{th}$ node. Here, $\mathbf{W^{[n]}}$ denotes the matrix that contains the filter weights associated with the $n^{th}$ layer. The last layer in the neural network is denoted as the $N^{th}$ layer.

The neural network problem can now be succinctly defined as finding the weights $w_{ij}^{[n]}$ that minimize the objective function:

$$\epsilon = 1/2 \sum_i (\overbrace{a_i^{[N]}}^{predicted} - \overbrace{y_i}^{observed})^2 + regularization\ term, \tag{4.23}$$

where $y_i$ is the observed output at the $i^{th}$ node and $a_i^{[N]}$ is the predicted output at the $i^{th}$ node of the $N^{th}$ layer. Previously we denoted the target vector as $\mathbf{t}$ but we now denote it as $\mathbf{a}^{[N]}$.

The simplified[3] update formula for iterative steepest descent without regularization is given by

$$\begin{aligned} &for\ j = 1 : niter \\ &\quad for\ n = 1 : N \\ &\qquad\qquad w_{ij}^{[n]} := w_{ij}^{[n]} - \alpha \frac{\partial \epsilon}{\partial w_{ij}^{[n]}}, \\ &\quad end \\ &end \end{aligned} \tag{4.24}$$

where := indicates that the variable $w_{ij}^{[n]}$ is reset to the value given on the righthand side and $\alpha$ is the step length. The above iterative formula appears simple, but the next chapter will derive the gradient terms for a multi-node multi-layer neural network.

[3]To reduce notational tyranny, we exclude notation for different training examples and assume just one example. The input layer in Figure 4.2 where the features are inserted is not enumerated as a layer because it is devoid of weights.

## 4.2 Multinomial Classifiers

The logistic function $g(z) = 1/(1 + e^{-z})$ was used to squash down a wide range of numbers $-\infty < z < \infty$ into a small range of numbers between 0 and 1. This function was also used to define the liklihood function for a binary classifier, which is the starting point for the cross-entropy error function as the objective function. If there are $C > 2$ classes of input objects, then we need a multinomial classifier that can classify the objects into $C$ classes. This can be done by recognizing that the standard logistic function for binary classification can be recast as the probability density function (pdf):

$$P(z) = \frac{e^z}{(e^z+1)} = \frac{1}{(1+e^{-z})}, \tag{4.25}$$

so that for binary classification $P(z = z_1) + P(z = z_2) = 1$ where $z_1 = 0$ and $z_2 = 1$ and $P(z) \leq 1$. Therefore, the pdf can be expressed as

$$P(z = z_k) = \frac{e^{z_k}}{(\sum_{i=1}^{2} e^{z_i})}, \tag{4.26}$$

which is the *soft-max* function used at the output layer with a single node with only binary classification. This says that if $P(z = z_1) + P(z = z_2) = 1$ Therefore the last layer of the neural network has $C$ nodes, and the $k^{th}$ node takes the input $z_k$ from the previous layer and applies the soft-max operation to get

$$y_k = \frac{e^{z_k}}{\sum_{i=1}^{C} e^{z_i}}, \tag{4.27}$$

which takes on the values between 0 and 1. This is the squasher used for the last node of a neural network that employs the cross-entropy error function.

## 4.3 ReLu Activation Function

The sigmoid function was often used as the activation function for neural networks in the 1990s. Empirical experience now shows a preference for other activation functions, especially the *rectified linear unit* $ReLu(z)$ function:

$$ReLu(z) = \begin{cases} 0 & \text{if } z < 0 \\ z & \text{if } z \geq 0 \end{cases} \tag{4.28}$$

The derivative $\frac{\partial ReLu(z)}{\partial z} = 0$ for $z < 0$ and $\frac{\partial ReLu(z)}{\partial z} = 1$ if $z > 0$. The benefit of using $ReLu(z)$ as an activation function is often a significantly faster convergence rate.

## 4.4 Summary

The equations for solving a neural network are presented, and details for implementing it with a one-layer network are presented. MATLAB codes are provided and should be used to get a feel for how different parameters such as damping, iteration number, step length, and input data affect the final result. The much more complex equations for finding the gradient of a general neural network are presented in the next chapter.

## 4.5 Exercises

1. A linear operator $A(z)$ has the property that $A(\alpha z) = \alpha A(z)$. Show that $g(\alpha z) = \alpha g(z)$ is not true for $g(z)$ being the sigmoid function.

2. The Datain.m and Dipslay.m codes are given below. Use these codes in conjunction with the one presented in the text to test the performance of a single neural network code that classifies $5 \times 1$ vectors $\mathbf{x}$ whose element values are either 1 or 0. The goal is to train a neural network that can recognize an input vector $\mathbf{x}$ where there is only a single 1 in one of its elements.

   - Compare the performance of the least squares algorithm against that of the cross-entropy algorithm. Which has a faster convergence rate.
   - The number of data examples that are classified as 1 should be about the same as the ones that are classified as 0, otherwise the system of equations will be ill-conditioned and convergence will slow down. Numerically demonstrate this fact.

```
        function [x,t,alpha,lim,beta,io]=Datain(M,N,x,t)
for i=1:M
    x(:,i)=round(rand(N,1));
end
for i=1:M
    if sum(x(:,i))==0;t(i)=1;end
end

% Now we are biasing the data set to be about 1/2
% of x(:,:) to only have a single 1 in the 4x1 vector
% so about 1/2 data are classed at t=1
for i=1:100:M
    x(:,i)=0;
    x(1,i)=1; t(i)=1;
 %   x(3,i+49)=1; t(i+49)=1;
end

alpha=.001;         % step size
lim=.00001;         % stopping criterion
 beta=1;io=1;  % io=0 cross-entropy
```

   and the Display.m code is below.

```
function Display(M,res,nit,w,x,t,beta,io)
tp=t*0;tpp=tp;
thresh=.34;
for m=1:M
```

```
 tp(m)=w'*x(:,m);
  tp(m)=1./(1+exp(-tp(m)*beta)); %corrected by Zongcai
if tp(m)>=thresh;tpp(m)=1;end
if tp(m)<thresh;tpp(m)=0;end
end
subplot(131);plot(res(2:nit));
title('a) RMS Error vs Iter. #')
if io==0;text(nit/9,max(res)*.8,'Cross-entropy NN');end
if io==1;text(nit/9,max(res)*.8,'Least-squares NN');end
ylabel('RMS Misfit');xlabel('Iteration #')
subplot(132);plot([1:M],t,'r*',[1:M],tp,'go')
ylabel('Example Class');xlabel('Example #')
title('b) (Obs=red, Pred=green)')
[xx,thresh]=ginput(1)
for m=1:M
 tp(m)=w'*x(:,m);
  tp(m)=1./(1+exp(-tp(m)*beta)); %corrected by Zongcai
if tp(m)>=thresh;tpp(m)=1;end
if tp(m)<thresh;tpp(m)=0;end
end
subplot(133);plot([1:M],t,'r*',[1:M],tpp,'g*')
ylabel('Example Class');xlabel('Example #')
title(['c) Threshold Value = ',num2str(thresh)])
text(M/9,.5,'Red * = Mistake','color','red')
```

3. Create a large training set and tune the parameters (such as step length, starting model etc) to give reasonably good convergence and a small error rate. Now take a holdout data set that was not used for training and use the trained coefficients $\mathbf{w}$ to classify each example in the holdout data set. Is the error rate similar to that of the training data? Did you overfit the data? Was your training set sufficiently diverse to account for the characteristics of the holdout data? if you overfitted, what are the remedies.

4. Sprinkle noise into your training data (i.e., deliberately misclassify some of the training data). Repeat the previous exercise.

5. Compare the two-node MATLAB algorithm with the single-node algorithm. Which one is more tolerant to complexity and noise in the training examples.

6. What is the result if the starting model is $\mathbf{w}^{(0)} = 0$? What is the result if the starting model is $\mathbf{w}^{(0)} = (1111)^T$? Show results.

7. In exercise 2, how do you determine if you are stuck in a local minima? Hint: try different starting models. Show results.

# Chapter 5

# Multilayer Neural Networks

The recursive equations for both the feed-forward and backward-propagation operations are derived for a multi-layered neural network. For each iteration, the feed-forward and gradient computations cost about $O(N^2M)$ algebraic operations for $N$ layers, each one associated with an $M \times M$ weight matrix. The workflow and pseudo-MATLAB codes are presented for implementing the multi-layer neural network algorithm.

## 5.1 Introduction

A neural network consists of a sequence of computational nodes arranged in distinct layers (see Figure 4.2). Each layer is a vertical sequence of computational nodes, where we assume $N$ layers in the neural network and each layer can have a different number of nodes. The input nodes are not counted as a network layer because there is no computational processing at these nodes.

There are two operations required for computing the gradient in equation 4.24: feed-forward and back-propagation. The next two sections derive the formulas for these two operations so they can be computed in an efficient recursive fashion.

## 5.2 Feed-forward Operation

The feed-forward operation of the neural network inputs the features $a_i^{[0]}$ from the starting $0^{th}$ layer and produces the predicted target value $a_i^{[N]}$ at the last $N^{th}$ layer:

$$a_i^{[N]} = g(\overbrace{\sum_{j=1}^{\mu_N} w_{ij}^{[N]} a_j^{[N-1]}}^{z_i^{[N]}}(\overbrace{\sum_{k=1}^{\mu_{N-1}} w_{jk}^{[N-1]} a_k^{[N-2]}}^{z_j^{[N-1]}}(.... a_m^{[1]}(\overbrace{\sum_{n=1}^{\mu_1} w_{mn}^{[1]} a_n^{[0]}}^{z_m^{[1]}})))), \tag{5.1}$$

where $\mu_i$ is the number of nodes in the $i^{th}$ layer. Here, the variable $z_i^{[k]}$ and output parameters $a_i^{[k]} = g(z_i^{[k]})$ of the $k^{th}$ layer can be assembled into the $\mu_k \times 1$ vectors $\mathbf{z}^{[k]}$ and $\mathbf{a}^{[k]} = g(\mathbf{z}^{[k]})$, respectively. The activation function $g(\ )$ is applied to each element of $\mathbf{a}^{[k]}$ so that $g(\mathbf{z}^{[n]})$ is a vector of the same dimension as $\mathbf{z}^{[n]}$.

The computation of the components of the $\mu_n \times 1$ vector $\mathbf{a}^{[n]}$ can be recursively computed from the components of $\mathbf{a}^{[n-1]}$ in the lower-order $(n-1)^{th}$ layer. This recursion can lead to a significant gain in efficient computations as seen in the pseudo-MATLAB code below.

$$
\begin{array}{l}
load\ \mathbf{a}^{[0]}; \\
\quad for \quad n = 1 : N \\
\qquad load\ \mathbf{W}^{[n]} \\
\qquad \mathbf{z}^{[n]} = \mathbf{W}^{[n]} \mathbf{a}^{[n-1]} \\
\qquad \mathbf{a}^{[n]} = g(\mathbf{z}^{[n]}) \\
\quad end
\end{array}
$$

where the matrix coefficients $w_{kl}^{[n]}$ for the $n^{th}$ layer are the elements of the $\mu_n \times \mu_{n-1}$ matrix $\mathbf{W}^{[n]}$ and the $\mu_n \times 1$ vector $\mathbf{z}^{[n]}$ contains the components $z_i^{[n]}$ in equation 4.22. For convenience the bias terms $b_i^{[n]}$ for the $n^{th}$ layer are implicitly represented in the first column of the matrix $\mathbf{W}^{[n]}$. Assuming that all of the layers have the same number $M$ of nodes, then each matrix-vector product/layer costs $O(M^2)$ operations. Thus, the computational count for computing the predicted target vector $\mathbf{a}^{[n]}$ is $O(M^2 N)$. Without using this recursive procedure, the computational count is $O(M^2 N^2)$ to compute the activation vector $\mathbf{a}^{[n]}$ at each layer.

## 5.3 Back-propagation Operation

The gradient term in equation 4.24 can be recursively computed *backward* by updating $w_{ij}^{[N-1]}$ from the higher-order terms in the $N^{th}$ layer. Unlike forward propagation where recursion computes the higher-order activation term from the previous-order one, back-propagation recursively computes lower-order terms from the higher-order ones. Similar to forward propagation, the computational count for recursively computing gradients at all of the layers is $O(M^2 N)$.

### 5.3.1 Formula for $\partial\epsilon/\partial w_{ij}^{[N]}$

The gradient of the misfit function in equation 4.24 for the weights in the $N^{th}$ layer is

$$
\frac{\partial \epsilon}{\partial w_{jk}^{[N]}} = \sum_{i=1}^{\mu_N} \overbrace{(a_i^{[N]} - y_i)}^{residual} \frac{\partial a_i^{[N]}}{\partial w_{jk}^{[N]}}, \tag{5.2}
$$

where we set $\lambda = 0$ for pedagogical simplicity. From the chain rule and equations 4.21-4.22 we have

$$
\frac{\partial a_i^{[N]}}{\partial w_{jk}^{[N]}} = \frac{\partial a_i^{[N]}}{\partial z_i^{[N]}} \overbrace{\frac{\partial z_i^{[N]}}{\partial w_{jk}^{[N]}}}^{\partial \sum_p w_{ip}^{[N]} a_p^{[N-1]}} = g(z_j^{[N]})' a_k^{[N-1]} \delta_{ij}, \tag{5.3}
$$

and

$$\frac{\partial a_m^{[N]}}{\partial a_i^{[N-1]}} = \frac{\partial g(z_m^{[N]})}{\partial z_m^{[N]}} \frac{\partial z_m^{[N]}}{\partial a_i^{[N-1]}} \quad = \quad g(z_m^{[N]})' w_{mi}^{[N]}, \tag{5.4}$$

where $g(z_j^{[N]})' = \frac{\partial g(z_j^{[N]})}{\partial z_j^{[N]}}$ and $\delta_{ij}$ represents the Kronecker delta function. Plugging equation 5.3 into equation 5.2 gives the gradient wrt $w_{jk}^{[N]}$ in the $N^{th}$ layer:

$$\frac{\partial \epsilon}{\partial w_{jk}^{[N]}} \quad = \quad \overbrace{(a_j^{[N]} - y_j) g(z_j^{[N]})'}^{\delta_j^{[N]}} a_k^{[N-1]}, \tag{5.5}$$

where $\delta_j^{[N]}$ is the $j^{th}$ component of the weighted residual that vector multiplies the "forward" field value $a_k^{[N-1]}$ at the $k^{th}$ node in the $(N-1)^{th}$ layer. If $g(z)$ is a sigmoid function then $g(z)' = g(z)(1 - g(z))$. Other types of activation functions will provide different formulas for the derivative.

Can we recursively use the term $\delta_j^{[N]}$ in the formula for the calculation of the next lower-order gradient? The answer is yes as the next section demonstrates.

### 5.3.2 Formula for $\partial\epsilon/\partial w_{ij}^{[N-1]}$

To derive the expression for $\partial\epsilon/\partial w_{ij}^{[N-1]}$, recall the multidimensional chain rule where $f(a_1(t), a_2(t), a_3(t)...a_M(t))$ is a regular function of $M$ smooth functions $a_i(t)$. In this case the derivative $\partial f(t)\partial t$ can be expressed as

$$\frac{\partial f(a_1^{[N]}, a_2^{[N]}, ...a_M^{[N]})}{\partial t} \quad = \quad \sum_{m=1}^{M} \frac{\partial f(a_1^{[N]}, a_2^{[N]}, \dots a_M^{[N]})}{\partial a_m^{[N]}} \frac{\partial a_m^{[N]}}{\partial t}, \tag{5.6}$$

where we append the superscript $[N]$ to the $a_i$ variables. Setting $t \to a_i^{[N-1]}$, this equation says that, for forward propagation of the feature values, a perturbation in $a_i^{[N-1]}$ will affect the values of the outputs $a_i^{[N]}$ in the next highest-order layer. Since this perturbation $t$ will also affect the misfit function $\epsilon(a_1^{[N]}, a_2^{[N]}, ...a_{\mu_n}^{[N]})$, then according to the chain rule in equation 5.6 the gradient w/r to the weights in the $N-1$ layer becomes

$$\begin{aligned}
\frac{\partial \epsilon}{\partial w_{jk}^{[N-1]}} \quad &= \quad \sum_{m=1}^{\mu_N} \overbrace{\frac{\partial \epsilon}{\partial a_m^{[N]}}}^{(a_m^{[N]} - y_m)} \sum_{i=1}^{\mu_{N-1}} \overbrace{\frac{\partial a_m^{[N]}}{\partial a_i^{[N-1]}}}^{\text{eq. 5.4: } g(z_m^{[N]})' w_{mi}^{[N]}} \overbrace{\frac{\partial a_i^{[N-1]}}{\partial w_{jk}^{[N-1]}}}^{\text{eq. 5.3: } g(z_j^{[N-1]})' a_k^{[N-2]} \delta_{ij}}, \\
&= \quad \sum_{m=1}^{\mu_N} \overbrace{(a_m^{[N]} - y_m)}^{\mathbf{r}^{[N]}} g(z_m^{[N]})' g(z_j^{[N-1]})' a_k^{[N-2]} \sum_{i=1}^{\mu_{N-1}} w_{mi}^{[N]} \delta_{ij}, \\
&= \quad \overbrace{g(z_j^{[N-1]})'}^{\mathbf{dg}^{[N-1]}} \overbrace{a_k^{[N-2]}}^{\mathbf{a}^{[N-2]}} \left( \overbrace{\sum_{m=1}^{\mu_N} w_{mj}^{[N]}}^{\mathbf{W}^{[N]T}} \overbrace{g(z_m^{[N]})'}^{\mathbf{dg}^{[N]}} \overbrace{(a_m^{[N]} - y_m)}^{\mathbf{r}^{[N]}} \right).
\end{aligned} \tag{5.7}$$

Here, the role of the $\{\mathbf{W}^{[N]^T}\mathbf{dg}^{[N]}.*\mathbf{r}^{[N]}\}_j$ term is to back-propagate the weighted residual vector from the $N^{th}$ layer to the $j^{th}$ node in the $(N-1)^{th}$ layer. Note that $\mathbf{dg}.*\mathbf{a}$ indicates an element-by-element MATLAB multiplication of the two vectors and the summation associated with the matrix-vector multiplication is over the first subscript $m$ of $w_{mj}^{[N]}$. Thus, we use the transpose symbol $T$ in the superscript of $\mathbf{W}^{[N]}$.

### 5.3.3 Formula for $\partial\epsilon/\partial w_{ij}^{[N-2]}$

The previous section derived the gradient formula for the $N-1$ hidden layer next to the output layer $N$. We now derive the gradient $\partial\epsilon/\partial w_{ij}^{[N-2]}$ for the hidden layer surrounded only by hidden layers. In this case the objective function will be a function of the activation parameters in two neighboring links of the chain of layers. This means we need a two-link chain rule for $f = f(g(h(t)))$, where $f(g)$ is a function of $g$ and $g(h)$ is a function of $h(t)$. Therefore, the single-link chain rule in equation 5.6 can be extended to the two-link chain rule where $a_i(t) \to a_i(b_1(t), b_2(t) \dots b_M(t))$:

$$\frac{\partial f(a_1, a_2, \dots a_M)}{\partial t} = \sum_{m=1}^{M}\sum_{n=1}^{M}\frac{\partial f(a_1, a_2, \dots a_M)}{\partial a_m}\frac{\partial a_m}{\partial b_n}\frac{\partial b_n}{\partial t}. \tag{5.8}$$

Setting $a_i \to a_i^{[N]}$, $b_i \to a_i^{[N-1]}$, $f \to \epsilon$ and $t \to a_i^{[N-2]}$ gives the formula for $\frac{\partial\epsilon}{\partial w_{jk}^{[N-2]}}$:

$$\begin{aligned}
\frac{\partial\epsilon}{\partial w_{jk}^{[N-2]}} &= \sum_{i=1}^{\mu_{N-2}}\frac{\partial\epsilon}{\partial a_i^{[N-2]}}\frac{\partial a_i^{[N-2]}}{\partial w_{jk}^{[N-2]}}, \\
&= \sum_{m=1}^{\mu_N}\overbrace{\frac{\partial\epsilon}{\partial a_m^{[N]}}}^{(a_m^{[N]}-y_m)}\sum_{n=1}^{\mu_{N-1}}\overbrace{\frac{\partial a_m^{[N]}}{\partial a_n^{[N-1]}}}^{g(z_m^{[N]})'w_{mn}^{[N]}}\sum_{i=1}^{\mu_{N-2}}\overbrace{\frac{\partial a_n^{[N-1]}}{\partial a_i^{[N-2]}}}^{g(z_n^{[N-1]})'w_{ni}^{[N-1]}}\overbrace{\frac{\partial a_i^{[N-2]}}{\partial w_{jk}^{[N-2]}}}^{g(z_j^{[N-2]})'a_k^{[N-3]}\delta_{ij}} \\
&= \overbrace{g(z_j^{[N-2]})'}^{\mathbf{dg}^{[N-2]}}\overbrace{a_k^{[N-3]}}^{\mathbf{a}^{[N-3]}}\left(\overbrace{\sum_{n=1}^{\mu_{N-1}}w_{nj}^{[N-1]}}^{\mathbf{W}^{[N-1]^T}}\overbrace{g(z_n^{[N-1]})'}^{\mathbf{dg}^{[N-1]}}\right)\left(\overbrace{\sum_{m=1}^{\mu_N}w_{mn}^{[N]}}^{\mathbf{W}^{[N]^T}}\overbrace{g(z_m^{[N]})'}^{\mathbf{dg}^{[N]}}\overbrace{(a_m^{[N]}-y_m)}^{\mathbf{r}^{[N]}}\right),
\end{aligned} \tag{5.9}$$

where the boldface variables are also seen in equation 5.7 and $g(z_m^{[N]})'(a_m^{[N]} - y_m)$ is the residual component in equation 5.5.

Each layer of the neural network has the same type of structure, so the formula for the gradient in the $P^{th}$ layer, where $1 < P < N-2$, is given by

$$\frac{\partial\epsilon}{\partial w_{jk}^{[P]}} = g(z_j^{[P]})'a_k^{[P-1]}\left(\mathbf{V}^{[P+1]}\mathbf{V}^{[P+2]}\dots\mathbf{V}^{[N-1]}\mathbf{V}^{[N]}\mathbf{r}^{[N]}\right)_j, \tag{5.10}$$

where

$$(\mathbf{V}^{[P]})_{ij} = w_{ij}^{[P]}g(z_i^{[P]})', \quad i \in [1, 2 \dots \mu_P],\ j \in [1, 2 \dots \mu_{P-1}], \tag{5.11}$$

The numerical implementation of equation 5.10 is ripe for recursion. Unlike the recursive algorithm of forward propagation that starts at layer 1, backward recursion starts from the $N^{th}$ layer to get $\mathbf{V}^{[N]}\mathbf{r}^{[N]}$, and then recursively computes the next lower-order terms until reaching the layer of interest. This is known as backward propagation of the residual.

The pseudo-code for recursively computing the $M \times M$ gradient matrix $\mathbf{d}\epsilon$ at the $(J-1)^{th}$ layer is given below. Here we assume the bias values are absorbed into the matrix $\mathbf{W}^{[N]}$ and $N > J > 1$, where $J$ is a positive integer. Assume $\mathbf{W}^{[i]}$, $\mathbf{a}^{[i]}$, $\mathbf{dg}^{[i]}$ are the $M \times M$ weight, $M \times 1$ activation and $M \times 1$ gradient $\frac{\partial g}{\partial z}$ matrices, respectively, that have been pre-computed for all $N$ layers. For notational simplicity we assume that the number of nodes in each layer is $M$.

$$
\begin{aligned}
Load\ (\mathbf{a}^{[0]},\ \mathbf{a}^{[1]} \ldots \mathbf{a}^{[N]}),\ \mathbf{y},\ (\mathbf{dg}^{[0]},\ \mathbf{dg}^{[1]} \ldots \mathbf{g}^{[N]}),\ (\mathbf{W}^{[0]},\ \mathbf{W}^{[1]} \ldots \mathbf{W}^{[N]}) \\
\mathbf{in} = \mathbf{a}^{[N]} - \mathbf{y} \\
for \quad i = N : -1 : J \\
\mathbf{in} = \mathbf{dg}^{[i]} .* \mathbf{in} \\
\mathbf{in} = \mathbf{W'}^{[i]} * \mathbf{in} \\
end \\
\mathbf{in} = \mathbf{in} .* \mathbf{dg}^{[i-1]} \\
d\epsilon(j,k) = in(j) * a(k)^{[i-2]}
\end{aligned}
$$

Each gradient calculation for a layer's weights will cost about $O(M^2)$ algebraic operations per iteration because of the matrix-vector product. If the number of layers is $N$ then the total cost will be about $O(M^2 N)$ algebraic operations for computing the weight gradients for all the layers.

## 5.4 MATLAB Code

The MATLAB code for the fully-connected neural network can be derived from the above pseudo-code. There are two principal portions of the code, forward and backward propagation in *gradientnn.m* and the main code *NNode.m*. Below is the major fragment from *NNode.m*:

```
%  To train your neural network, we will now use steepest decent and line search

fprintf('\nTraining Neural Network... \n')

for k=2:nit     % Looping over iterations
    alpha=1;
    %gradientnn ouput: grad =gradient and res(k) = objective function value
    [grad,res(k)]=gradientnn(M,x,t',ww,layer_size,obj_option,act_option,lambda);

    for ilayer=1:layer_num-1
        ww{ilayer}=ww{ilayer}-alpha*grad{ilayer};   %update the weights
```

```
    end
% Start line search
    [~,res1]=gradientnn(M,x,t',ww,layer_size,obj_option,act_option,lambda);
    while (res1>res(k)) && (alpha>0.00001)
        alpha=alpha*0.5;
        for ilayer=1:layer_num-1
            ww{ilayer}=ww{ilayer}-alpha*grad{ilayer};
        end
        [~,res1]=gradientnn(M,x,t',ww,layer_size,obj_option,act_option,lambda);
    end
    %-------------------------------------------------------------------------
    %kk=kk+1
end
```

A fragment of the pseudo-code for forward propagation modeling in *gradientmm.m* is shown below

```
% Do forward propagation
for iter =1:layer_num-1                  % N =  # of layers

    aa{iter} = [ones(1, M); aa{iter}];     % ones(1, m) is for bias
    zz{iter}=ww{iter}*aa{iter};            % z[n]=W[n]a[n-1]
    %%%----- a[n]=g(z[n])------%%%
if iter ==layer_num-1                      % output layer sigmoid
 [aa{iter+1},~] = activation(zz{iter},1);
 else
 [aa{iter+1},~] = activation(zz{iter},act_option);
end

 penalize =penalize+ sum(sum(ww{iter}.^ 2)); % Bias regularization
end
% final output for forward propagation predicted target
t_pred = aa{layer_num};

% calculate the objective function and the derivative of objective
% function with respect to t_pred using likelihood or L2 type
% obj_option=1 for L2, 2 for likelihood

[res,in]= misfit( t_pred,t,1/M,obj_option);

res = res + (lambda/(2*M)) * penalize; % add regularization
```

and the back-propagation MATLAB code is below

```
% Implement back-propagation algorithm to compute the gradients
%  written according to Back-propagation Operation
```

```
% Implement back-propagation

for iter=layer_num-1:-1:1
    %'sigmoidgrad' means compute the gradient of the sigmoid function
    if iter ==layer_num-1 %output layer = sigmoid
        [~,dg]=activation(zz{iter},1);
    else
        [~,dg]=activation(zz{iter},act_option); % in=dg[i].*in
    end

  in=dg.*in;                                  % in=backward field, A=forward field

  grad{iter}=in*aa{iter}'+(lambda/M)*ww{iter};% de(j,k)=in*(a[i-2])^T
    % update  gradient with respect to next weights
  in=ww{iter}'*in;  % in=W'[i]*in
  in = in(2:end, :);
end

end
```

The activation function *activation.m* is given by

```
function [ g,dg ] = activation( z,type )
%z  -  input- for activation function
%type: objective function type: 1 for sigmoid, 2 for ReLU
% g - output- value of the activation function
% dg- output- gradient of g with respective to z

if type==1 % for sigmoid
    g = 1.0 ./ (1.0 + exp(-z));
    dg = g .* (1 - g); %Compute the gradient of the sigmoid function
elseif type==2 % for ReLU
    g=z*0.0; dg=z*0.0;
    g(z>0)=z(z>0);
    g(z<=0)=z(z<=0)*0.0;
    dg(z>0)=1.0;
    dg(z<=0)=0.0;
else
    display('You entered the wrong type for the activation function');
    stop
end
end
```

A vectorized version of the back-propagation and forward propagations algorithms are in Appendix 5.9.

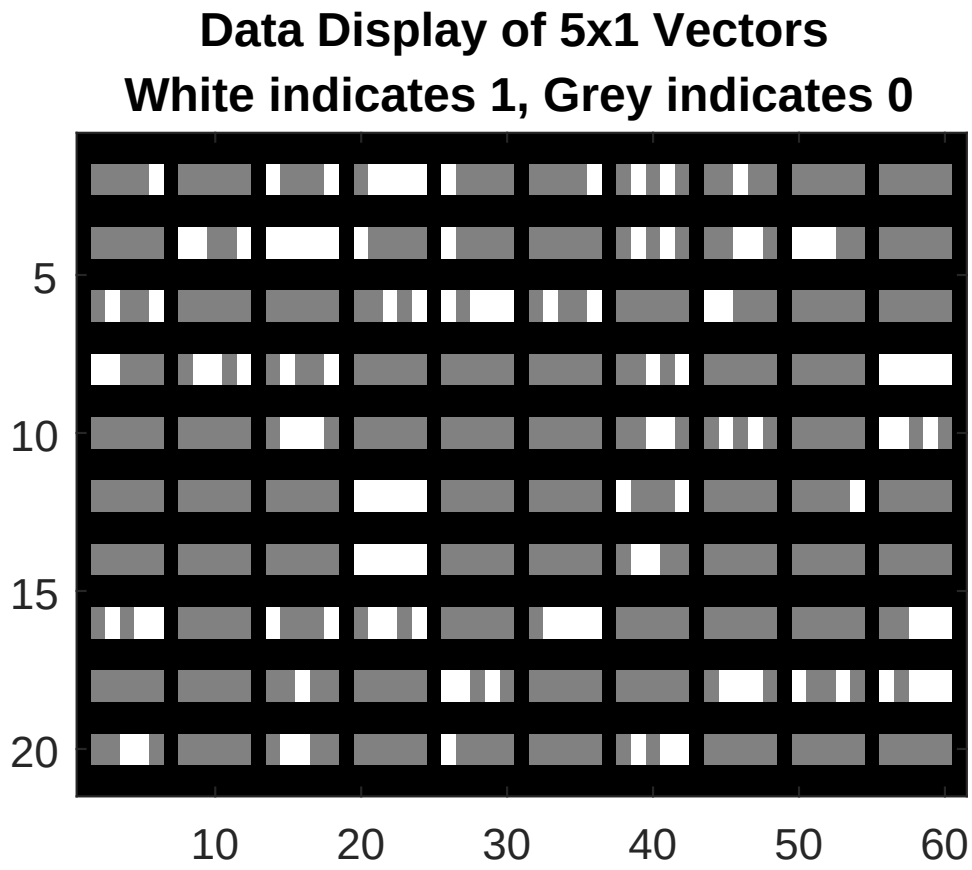


Figure 5.1: Examples of $5 \times 1$ vectors with element values equal to either 1 (gray) or 0 (white). Approximately half the vectors only contain 0 values.

## 5.5 Numerical Examples

As a test example, the input data were $5 \times 1$ vectors where the vector-element values were either 1s or 0s. Figure 5.1 graphically depicts a large number of vector examples, where gray (white) means an element value of 1 (0). The goal is to train a network so that it can classify any vector which only has 0 values as class 1, otherwise it is class 0. Using one-hundred training examples and a 4-layer neural network (we don't count the input layer) gives the results shown in Figure **??**. The top row of figures shows the results for a neural network with an L2 loss function and a sigmoid activation function, while the bottom row is for a ReLu activation function and a cross-entropy loss function. It is obvious that the ReLU and cross-entropy network provides the fastest convergence and most accurate results.

Figure 5.3 is similar to Figure **??** except for different combinations of loss and activation functions. It appears that the ReLu and the cross-entropy loss function in Figure **??**c- **??**d gives the best results in terms of accuracy and fast convergence rate.

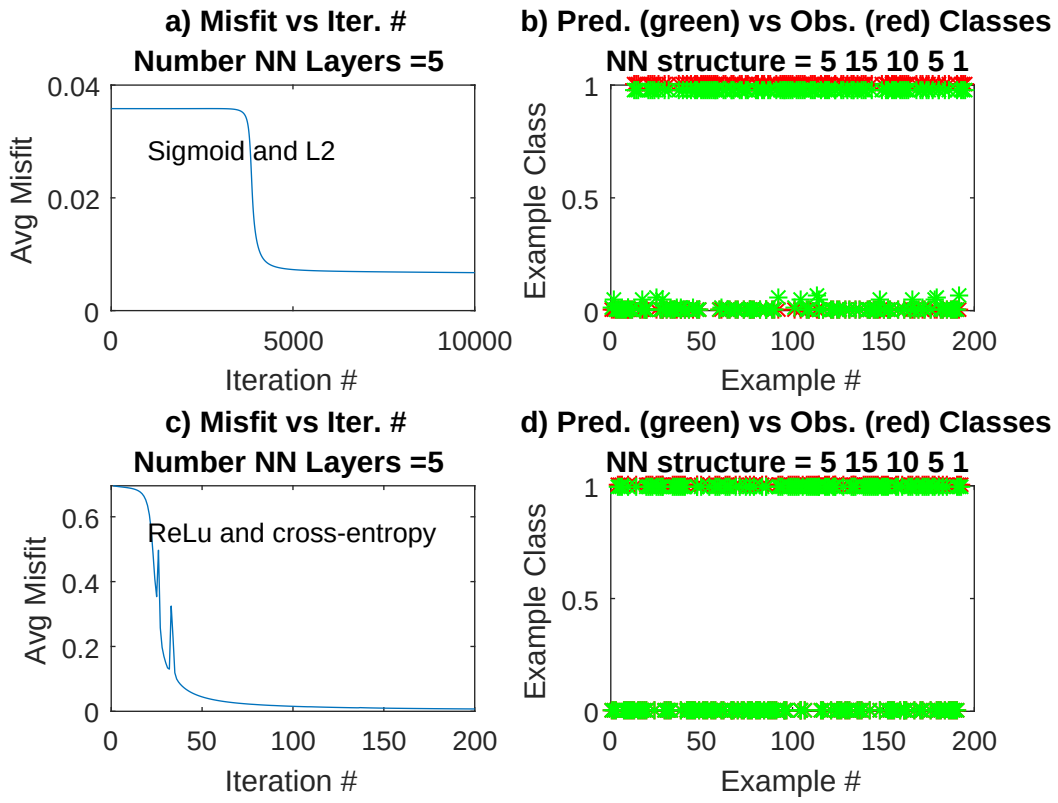


Figure 5.2: Convergence plots and comparison of predicted (green) and observed (red) classes of $5 \times 1$ input vectors for a 5-layer neural network. The 5 layers have 5, 15, 10, 5 and 1 nodes, starting from the input layer with 5 nodes. The top row is for the network with a sigmoid activation function and L2 loss function, while the bottom row is for a ReLu activation function and cross-entropy loss function. If only red stars appeared in the right column of figures then no classification errors were encountered.

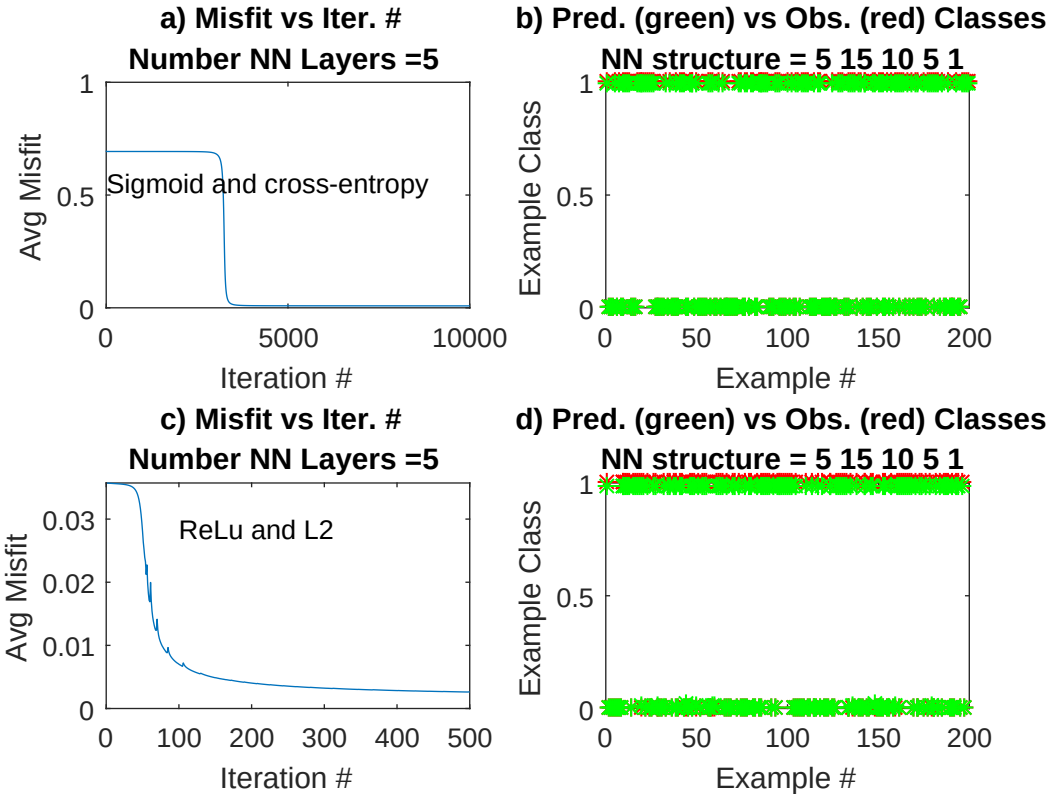


Figure 5.3: Same as previous figure except different combinations of activation and loss functions.

## 5.6 Summary

The recursive equations are derived for the forward propagation and back-propagation operations of a neural network. These operations can be used to find the weights $w_{ij}^{[n]}$ in each layer that minimize the objective function. The recursive nature of these equations provide for an efficient steepest descent procedure that requires $O(M^2N)$ computation per iteration, where $M$ is the number of nodes per layer and $N$ is the number of layers. Here, we assume that the number of nodes $M$ is the same for each layer. Numerical examples are provided that empirically demonstrate the superior performance of the ReLu+cross-entropy network over the other types of networks.

## 5.7 Exercises

1. Derive the formula for the three-link chain rule.

2. Derive the formula for the weight gradient at the $N = 12$ layer.

3. Show that equation 5.7 reduces to equation 4.16 for the two-layer neural network.

4. Write the pseudo-MATLAB code where the bias vector and its gradient are explicitly computed.

5. Derive the steepest descent formula for the cross-entropy objective function.

6. Write the pseudo-MATLAB code where regularization is used.

7. Write the pseudo-MATLAB code where the input is a batch matrix that represents the training set.

8. Write the pseudo-MATLAB code where the number of nodes per layer is different from one another.

## 5.8 Computational Labs

1. Go to Chapter Fully Connected Neural network lab in *Lab*1*index.html* and run the Fully Connected Neural network Lab.

## 5.9 Appendix: Vectorized Steepest Descent Formula for Neural Networks

Assume a network architecture where there are $M$ nodes per layer. The vectorized stochastic steepest descent formula for computing $w_{ij}^{[n]}$ for each layer is given by the following MATLAB

pseudo-code.

*Load* $\mathbf{a}^{[0]}$, $\mathbf{y}$; *Initialize* $\alpha$, *matrices* $\mathbf{W}^{[i]}$ *for* $i = [1, 2, \ldots N]$

*for* $m = 1 : N$

$\mathbf{a}^{[m]} = forw(\mathbf{a}^{[0]}, \mathbf{W}^{[1]} \ldots \mathbf{W}^{[N]})$ *Compute predicted data* $\mathbf{a}^{[N]}$

$\mathbf{dg}^{[m]} = deriv(\mathbf{a}^{[m]})$ *Compute predicted derivative* $\mathbf{dg}^{[m]}$

*end*

*for* $n = 1 : niter$ *Iterate steepest descent*

$\mathbf{in} = \mathbf{a}^{[N]} - \mathbf{y}$ *Residual*

*for* $i = N : -1 : 1$ *Compute* $\mathbf{W}^{[i]}$ *at each layer*

$\mathbf{in} = \mathbf{dg}^{[i]} .* \mathbf{in}$

$\mathbf{d\epsilon} = (\mathbf{in} .* \mathbf{dg}^{[i-1]}) * \mathbf{a}^{[i-1]T}$

$\mathbf{in} = \mathbf{W}'^{[i]} * \mathbf{in}$

$\mathbf{W}^{[i]} := \mathbf{W}^{[i]} - \alpha \mathbf{d\epsilon}$

*end*

*for* $m = 1 : N$ *Compute new prediction* $\mathbf{a}^{[N]}$

$\mathbf{a}^{[m]} = forw(\mathbf{a}^{[0]}, \mathbf{W}^{[1]}, \ldots \mathbf{W}^{[N]})$

$\mathbf{dg}^{[m]} = deriv(\mathbf{a}^{[m]})$

*end*

*end*

The above code is vectorized because we have almost entirely eliminated the explicit use of *for loops*, which can speed up computations by more than order-of-magnitude. If the input data are a batch of data then this can be accommodated by an outer loop over the different training examples. However, this is inefficient so a vectorized version can be obtained by replacing the input $M \times 1$ data vector $\mathbf{a}^{[0]} \rightarrow \mathbf{A}^{[0]}$, where the $M \times P$ batch-data matrix has $P$ columns, each one a different training example of input values. Similarly, the $M \times 1$ target vector $\mathbf{y}$ can be transformed into a $M \times P$ target matrix. This assumes that there are $M$ nodes per layer.

# Chapter 6

# Convolutional Neural Networks

The theory and practice of convolutional neural networks (CNNs) is introduced where the fully-connected layers are replaced by convolutional layers. In a convolutional neural network the summation in $a_i = g(\sum_{j=1}^{M} w_{ij}a_j)$, is replaced by the weighted sum $a_i = g(\sum_{j=1}^{M'} \hat{w}_{i-j}a_j)$ over a limited number $M' << M$ of nodes in the layer, where $\hat{w}_{i-j} = w_{ij}$. Thus the weights $\hat{w}_i$ are spatially invariant and reused at every input node. Therefore, the weight $w_{ij}$ in the $O(M \times M)$ $\mathbf{W}^{[\mathbf{n}]}$ matrix is replaced by the convolutional weight $w_{ij} \to w_{i-j}$ with a memory requirement of $O(M)$. This leads to a significant reduction in storage and computational requirements. Moreover, the local nature of the "convolution" operation appears to be consistent with the local identification of objects by the brain's neurons in the visual cortex.

## 6.1 Introduction

The problem with the classical neural network is that it leads to enormous computational and memory demands for input data of reasonable size. For example, if images of cats are to be distinguished from those of other animals, then the training set might consist of $1024 \times 1024$ photos of cats with $O(10^6)$ pixels/photo. For a fully-connected layer, this means that there are $10^6$ nodes in the first layer, which will lead to a $10^6 \times 10^6$ $\mathbf{W}^{[1]}$ weight matrix. Thus, the storage and computational demands for large sized input vectors can be prohibitive and discourage the use of neural networks in all but the largest computers.

Another problem with feeding in an entire image into a fully-connected layer (FCL) is overfitting. In the previous example, all of the intensity values of a $1024 \times 1024$ image are weighted and summed as an input value into any one node. Thus a FCL will lead to more than a million unknowns with just one layer. This suggests that the conventional neural network might lead to an undetermined set of equations that can almost exactly explain the training data, but do very poorly on data outside the training set. This type of neural network overfits the data.

To mitigate overfitting and reduce the storage and computational demands, convolutional neural networks are now in widespread use today. Instead of feeding forward all of the input data into each node, only a small localized portion is fed into each node. This localized portion of data is weighted by the same weights for any node. This type of neural

network is known as a convolutional neural network (CNN) and is the most popular form of supervised learning since the early 2000's.

CNN can trace its roots back to the biology experiments in the 1950-1960s where it was discovered that a cat's visual cortex greatly responded to certain orientations of an object. Figure 6.1 shows how a cat's visual neurons are most responsive to horizontal or oblique orientations of a bar that it views. Thus, a cluster of neurons in the cat's brain was suited to significantly responding to the view of a small orientation range of a long skinny object. This is similar to the actions of a *localized* dip filter that only lights up for certain dips of an object.

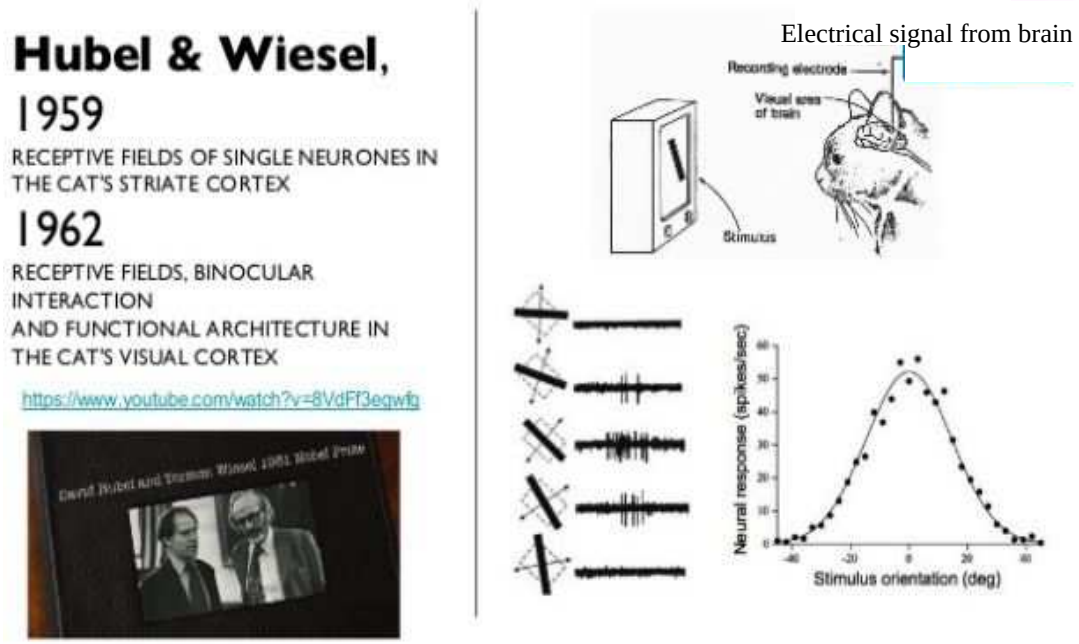


Figure 6.1: Cat and graph of its neural visual response for different orientations of the black bar. Figure extracted from Youtube video of Deep Learning for Computer Vision (Andrej Karpathy, OpenAI).

In the 1980s a spatially invariant neural-network architecture was inspired by the feline visual processing system. Fukushima (1980) proposed a spatially invariant operation, the summation of a localized set of input values with neural weights, as a working hypothesis for some neural mechanisms of a cat's visual pattern recognition. This is a classical neural network model except the number of convolutional weights is much less than the number of input nodes and the weights are spatial invariant. That is, $\hat{w}_{i-j} = w_{ij}$. Thus, the input $z_i$ to $i^{th}$ node can be mathematically computed as a correlation of a small patch of input image values with a small set of spatially invariant weights. Instead of feeding forward all of the input data into each node, only a small localized portion, with the same weights, is fed into each node.

More precisely, the fully-connected neural network input value at the $i^{th}$ node is given

by

$$z_i^{[n]} = \sum_{j=1}^{M} w_{ij}^{[n]} a_j^{[n-1]}, \tag{6.1}$$

while $z_i^{[n]}$ for the convolutional network illustrated in Figure 6.2 is

$$z_i^{[n]} = \sum_{j \in B} \hat{w}_{i+j}^{[n]} a_j^{[n-1]}. \tag{6.2}$$

Here, $w_j \; for \; j \in [1, 2, 3 \ldots M']$, $M' << M$, and $B$ is a small set of integers that describe the available weights $\hat{w}_j$. Thus, CNN has a computational and storage requirement of $O(M)$ compared to $O(M^2)$ for a fully-connected neural network. Moreover, the local nature of the "convolution" operation appears to be consistent with the local identification of objects by the brain's neurons in the visual cortex.

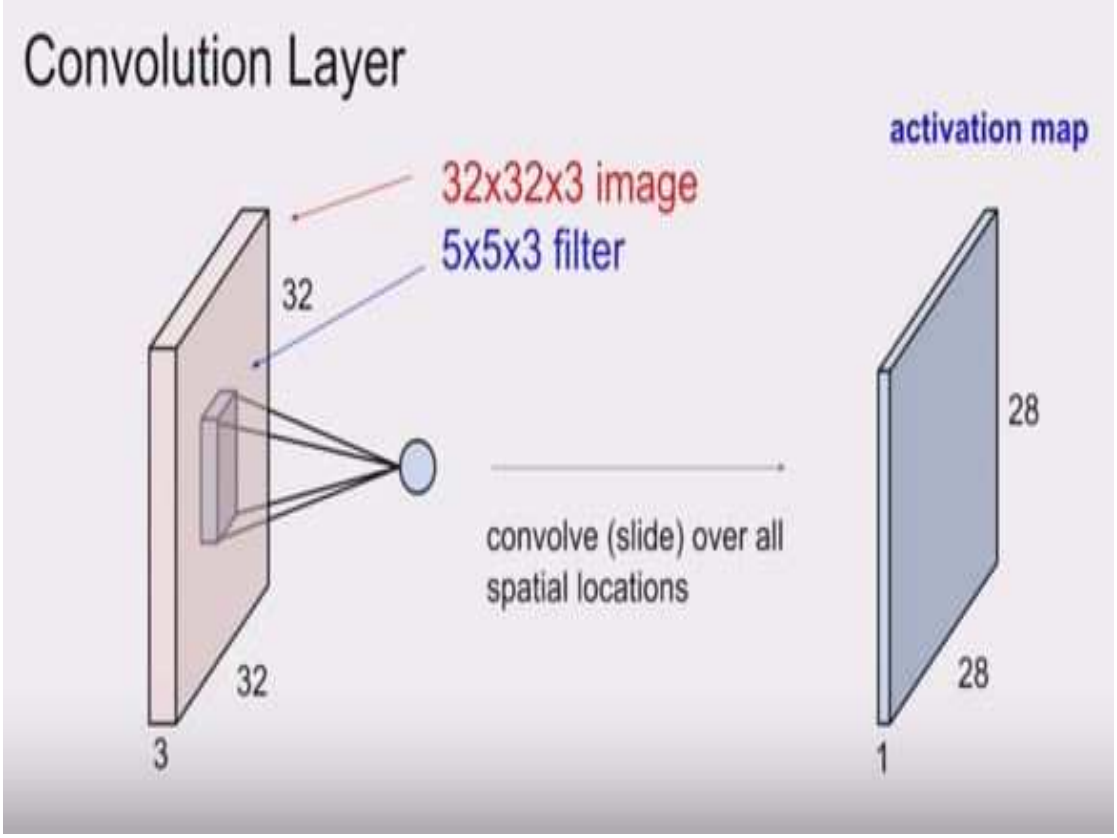


Figure 6.2: Convolution where the input is the $32 \times 32 \times 3$ image and the convolutional filter has dimension $5 \times 5 \times 3$. The output $z_i$ at the $i^{th}$ node is computed by taking the dot product of the weights and the activation values $z_i = \sum_{j \in B} \hat{w}_{i-j} a_j$, where $B$ is the set of indices associated with the small gray window on the left. Figure extracted from Youtube video of Deep Learning for Computer Vision (Andrej Karpathy, OpenAI).

The earliest gradient-based learning of a convolutional network was presented by LeCun et al. (1998). Its architecture is illustrated in Figure 6.3 and depicts the image of the letter $A$ as the input data. The intensities of the pixel values $a_i$ over a small patch of $A$ centered at the $i^{th}$ pixel are multiplied by the convolutional weights and summed to give the input $z_i = \sum_j \hat{w}_{i-j} a_i$ to the $i^th$ node of the next layer. These gray patch is shifted over by one pixel and the dot product operation is repeated with the same neural weights. There are three images depicted in the C1 layer, which means that there are 6 different types of filters. In this case, the index for the $k^{th}$ filter might be designated as $\hat{w}_i^{[n](k)}$ for $k \in [1, 2 \ldots 6]$. The reduction is storage and computational requirements is significantly less than that for

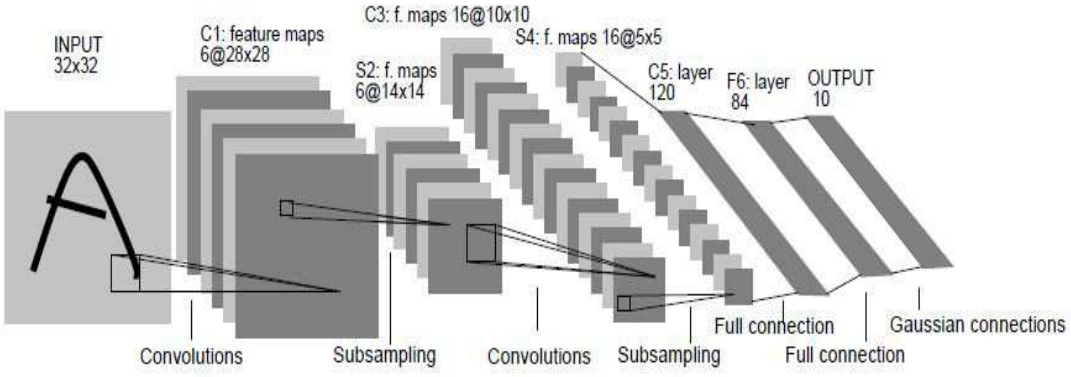


Figure 6.3: LeNet5 CNN architecture (LeCun et al. 1998). Each plane is a feature map.

fully-connected network, which is sometimes designated the classical neural network.

This chapter describes the technical characteristics of the CNN used in the past and some of the ones now being used in 2018. The basis for using different architectures does not currently have a theoretical justification, but it seems to evolve over time to latest CNN architectures that have significantly reduced testing errors and sped up convergence. As an example, Figure 6.4 depicts the speedups over the last decade for different CNN architectures. The trend is deeper CNN networks with larger numbers of layers and more accurate networks.

## 6.2 Building Blocks of CNN

There are five building blocks that comprise a typical CNN architecture are shown in Figure 6.5: convolution layer, filter, pooling layer, FCC, and Soft-Max Layer.

### 6.2.1 Convolution Layer

Assume the input image of the gray-scale robot is represented by a $32 \times 32$ matrix of intensity values. The filter size for the first convolution layer in Figure 6.5 is a $5 \times 5 \times 1$ filter for the values of $w_i^{[n]}$ in equation 6.2. In this case there is only one channel for the input image. Taking the dot product of this $5 \times 5 \times 1$ filter with different $5 \times 5 \times 1$ patches of the input $A$ image gives rise to one of the four $5 \times 5 \times 1$ feature maps. Each FM one has size $28 \times 28$

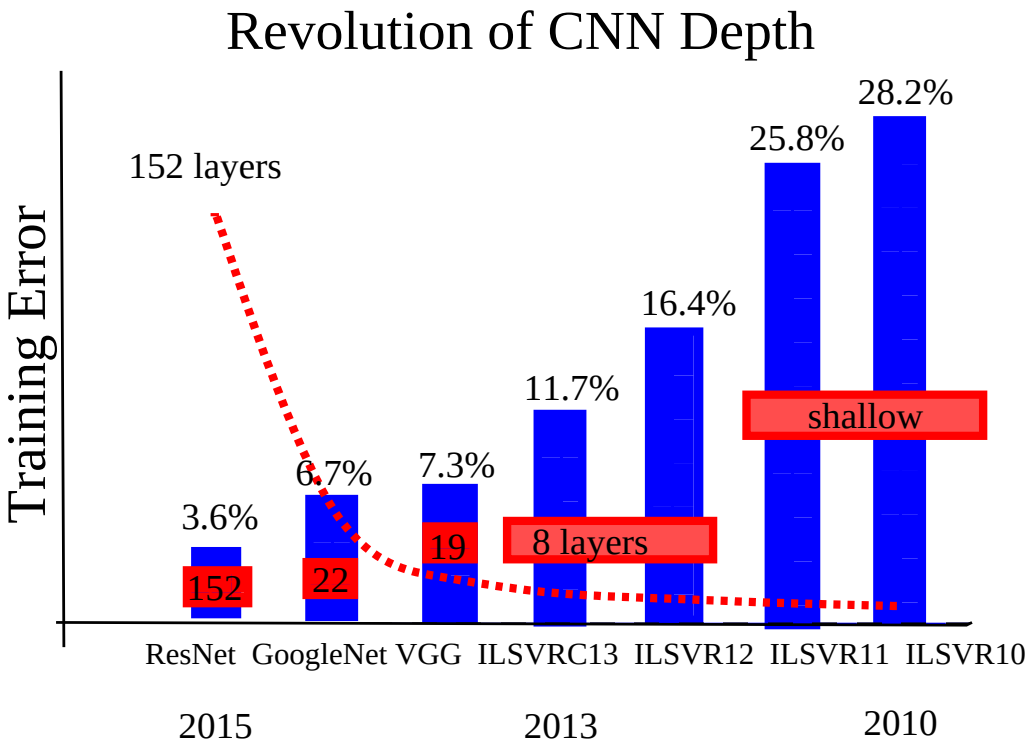


Figure 6.4: Training error vs calendar years for different CNN architectures developed over the last decade. Figure adapted from Youtube video of Deep Learning for Computer Vision (Andrej Karpathy, OpenAI).

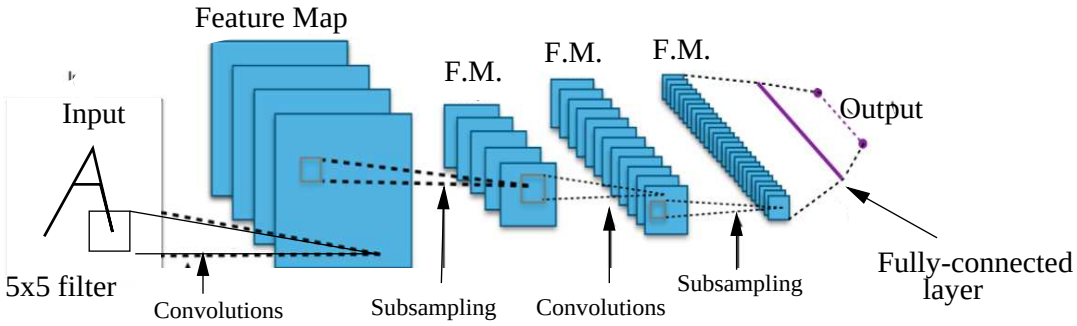


Figure 6.5: Schematic of building blocks in a CNN where the input is a $32 \times 32$ image of the letter $A$. Figure from *en.wikipedia.org/wiki/Convolutional.neural.network*.

and was created with a different $5 \times 5 \times 1$ filter. In this case we say that the output of layer 1 has four channels, each with a $28 \times 28$ image. The reduced size of each FM results from the fact that the filter is 5 pixels wide and tall, so that the filter can only be slid 28 pixels

across or down the image before part of it falls outside the input image.

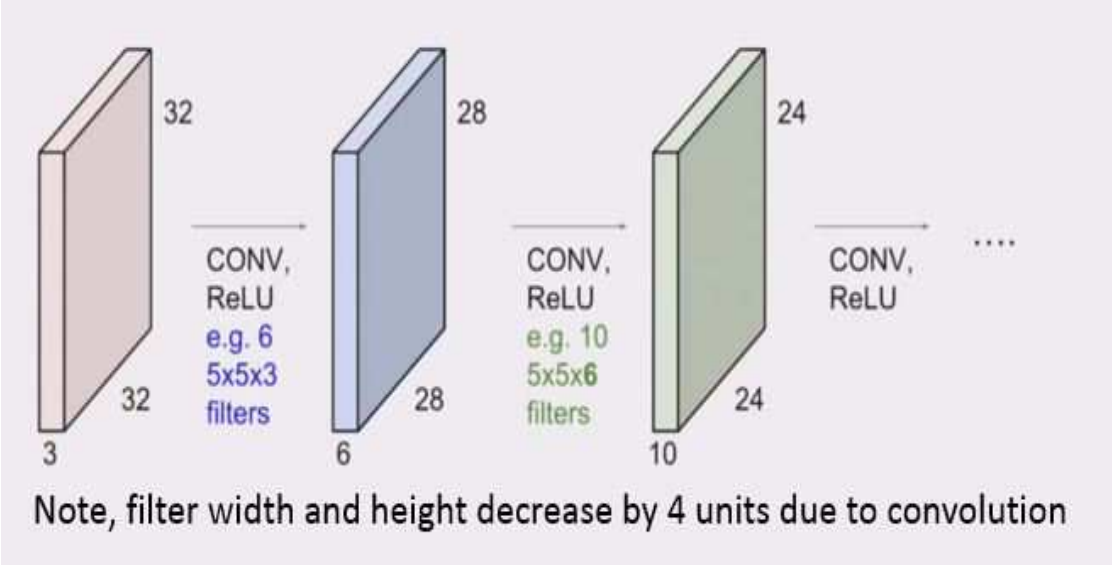


Figure 6.6: Example of reduction in size of FMs by sequential convolutions with $5 \times 5$ filters with stride $n_s = 1$. In this example the first layer has three FMs, the second one has 6 because 2 unique filters are used for each FM. The evolution of CNN architectures shows that the number of FMs per layer typically increase the deeper into the CNN. Figure adapted from Youtube video of Deep Learning for Computer Vision (Andrej Karpathy, OpenAI).

For the second layer associated with convolution, the filter will have a third dimension equal to the number of channels in the previous layer. For example, the tan layer in Figure 6.6 consists of three channels so the convolution filter will be of dimension $32 \times 32 \times 3$. In this example there are 6 different $5 \times 5 \times 3$ filters to give the blue stack of six images, each with dimension $28 \times 28$. To get the green images ten $5 \times 5 \times 6$ filters are convolved with the blue images.

In general, if the input image has size $n \times n \times n_{nch}$ then the filter size is $n_f \times n_f \times n_{ch}$ and the output feature map dimension is $n_c \times n_c$. Here,

$$n_c = \frac{n - n_f + 1}{n_s} + n_p, \tag{6.3}$$

and $n_p$ is the width and height of the zero-padding added to the feature map. Here, $n_{ch}$ is the number of channels, $n \times n$ is the dimension of the input image, and $n_s$ is the stride of the filter which is equal to the number of pixels the filter is shifted between each dot product of the dot-product operation. In the Figure 6.5 example $n_s = 1$, $n = 32$, $n_p = 1$ and $n_f = 32$ to give $n_c = 28$.

### 6.2.2 Activation Functions

Most activation functions use the rectified linear unit (ReLU) defined as

$$g(z_i) \quad = \quad max(0, z_i), \tag{6.4}$$

which eliminates negative values of the input $z$ and gives has the same value of $dg(z_i)/dz$ for any $z_i > 0$. The steepest descent backpropagation formula contains the factor $dg(z_i)/dz_i$ so its value will determine the type of residuals that will be used to updates the weights. Thus the $dg(z)/dz$ term for the ReLU function weights the residuals with equal magnitude to find the correct adjustment of the weights $w_i$ that minimize the objective function.

In comparison, the sigmoid activation function has the derivative $dg(z)/dz = g(z)(1 - g(z)) \approx 0$ unless $z_i \approx 0$. Since the backprojected residual is multiplied by $dg(z_i)/dz_i$ at the $i^{th}$ node, then updating occurs only when the input $z_i$ is nearly equal to zero. This means that $z_i$ terms much larger than zero are ignored. Thus convergence can slow down if the gradinet is mostly zero as evidenced by the observation that deep-layer CNNs with sigmoid or tanh activation functions stall after a certain number of iterations, even though the residual is far from zero. To include all non-zero residuals in the update of weights the ReLU function is now preferred over most activation functions.

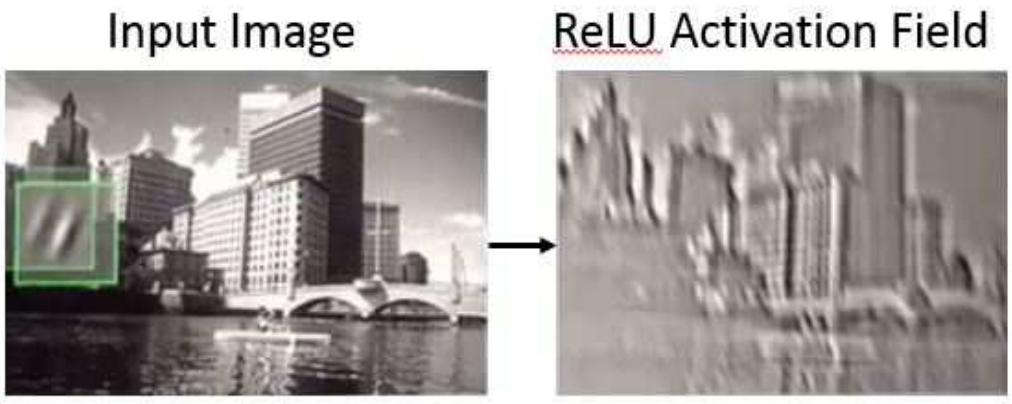


Figure 6.7: (Left) Input image and (right) activation image after convolution of the small filter (see green patch on left) with the input image and application of the ReLU activation function to the output. In this example, a "same" convolution was used to output the same sized-sized image after convolution. Notice that all the values in the activation image are positive. Figure adapted from Youtube video of Deep Learning for Computer Vision (Andrej Karpathy, OpenAI).

Figure 6.8 compares ReLU vs a sigmoid activation function for learning in a CNN with 9 layers. Obviously the ReLU learns the fastest for this test

A variation ReLU is Leaky ReLU as shown in Figure 6.9. Here $\alpha$ is a small number and Leaky ReLU reduces to standard ReLU if $\alpha = 0$. Leaky ReLU accelerates convergence in some cases.

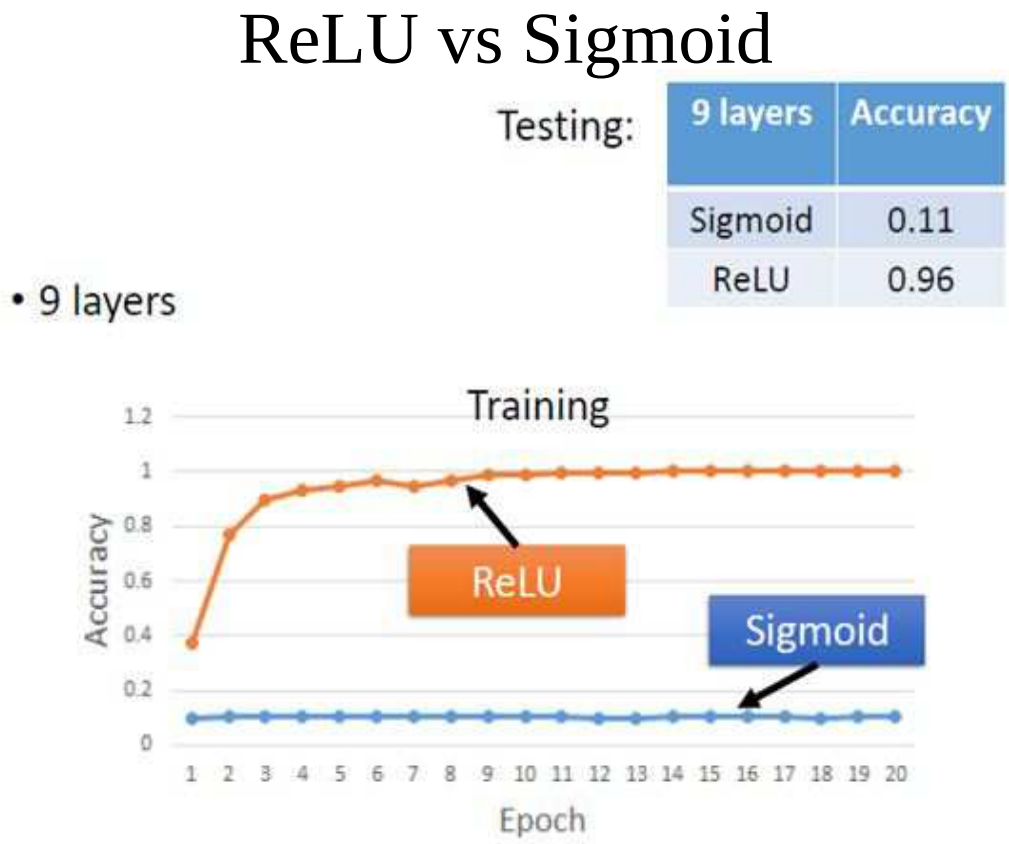


Figure 6.8: Comparison of accuracy of CCN learning using ReLu compared against sigmoid activation functions. Figure adapted from Hung-yi Lee PPT.

### 6.2.3 Feature Maps

There are four feature maps (FMs) just after the leftmost convolution in Figure 6.5 that result from applying four different $5 \times 5$ filters $\mathbf{w}^{(i)}$ (for $i \in [1, 2, 3, 4]$) to the input image. The role of each filter is to, hopefully, decompose the original image into four different images, each one associated with one of the four local features associated with the filters. For example, if $\mathbf{w}^{(1)}$ and $\mathbf{w}^{(2)}$ are the matrices

$$\mathbf{w}^{(1)} = \begin{bmatrix} 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{bmatrix}; \quad \mathbf{w}^{(2)} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}, \tag{6.5}$$

then the FM for $\mathbf{w}^{(1)}$ would be largely composed of the vertical parts of the letter $A$ while the FM for $\mathbf{w}^{(2)}$ would consist of the horizontal parts. These filters act as *matching* filters and extract parts of the image that are similar to the pattern in the filter.

# Leaky ReLU

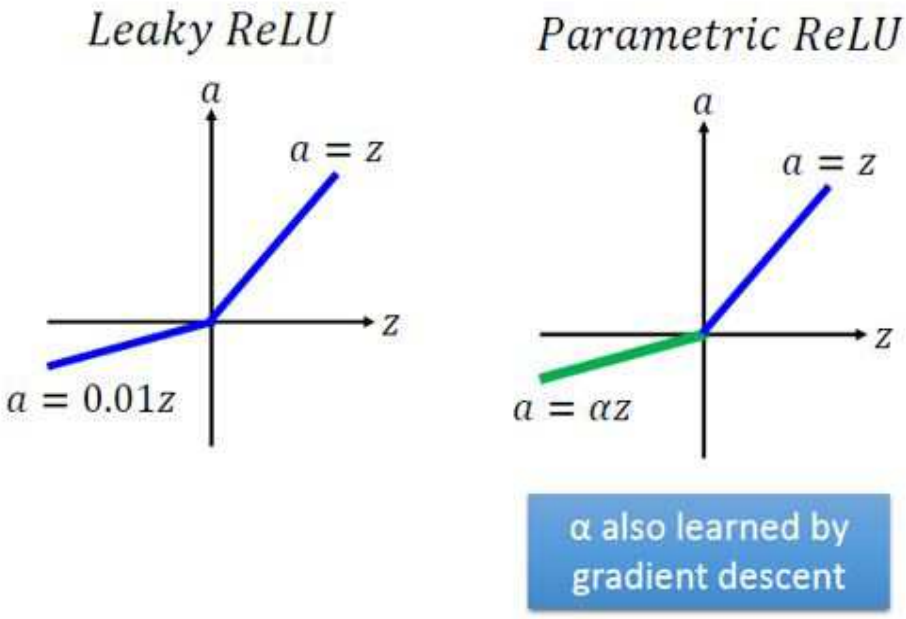


Figure 6.9: Leaky ReLU on left and the parameter value $\alpha$ can be determined by gradient descent on the right. Figure adapted from Hung-yi Lee PPT.

The one puzzle is that the backpropagation algorithm determines the patterns in these filters $\mathbf{w}^{(i)}$so it is a mystery as to which patterns will emerge in the FM. What is not mysterious is that these patterns are for small features in the original image because the filters are usually very small and localized. They are equivalent to high-wavenumber pass filters.

### 6.2.4 Pooling Layer

The pooling layer is created by a subsampling operation that reduces the size of the input map by an integer multiple. For example, the leftmost pooling or subsampling operation in Figure 6.5 reduces the $28 \times 28 \times 4$ FMs to $14 \times 14 \times 4$ FMs. After pooling, the activation function is applied to each sample in the pixel to give four $14 \times 14$ activation images $\mathbf{a}^{[1](i)}$, where $i$ denotes the $i^{th}$ activation image in layer 1. Note, the layer number is in square brackets and the channel number is in the rounded superscript[1].

There are several pooling strategies such as replacing the intensity value at a pixel by

[1]We often eliminate an abundance of superscript or subscripts when trying to quickly get across a key idea. For example, no channel superscripts are in equations 6.1 or 6.2.

the average value of its neighbors. The most widely used one is the maximum as illustrated in Figure 6.10. Here, the maximum value of four neighbors replaces the intensity value of the central point. The $2 \times 2$ patch is slid over by two pixels, i.e. a stride of 2, and the max-pool process is repeated. In this example, the $4 \times 4$ patch of pixels is reduced to a $2 \times 2$ patch. Typically, a pooling filter no larger than $2 \times 2$ is used, otherwise high-wavenumber

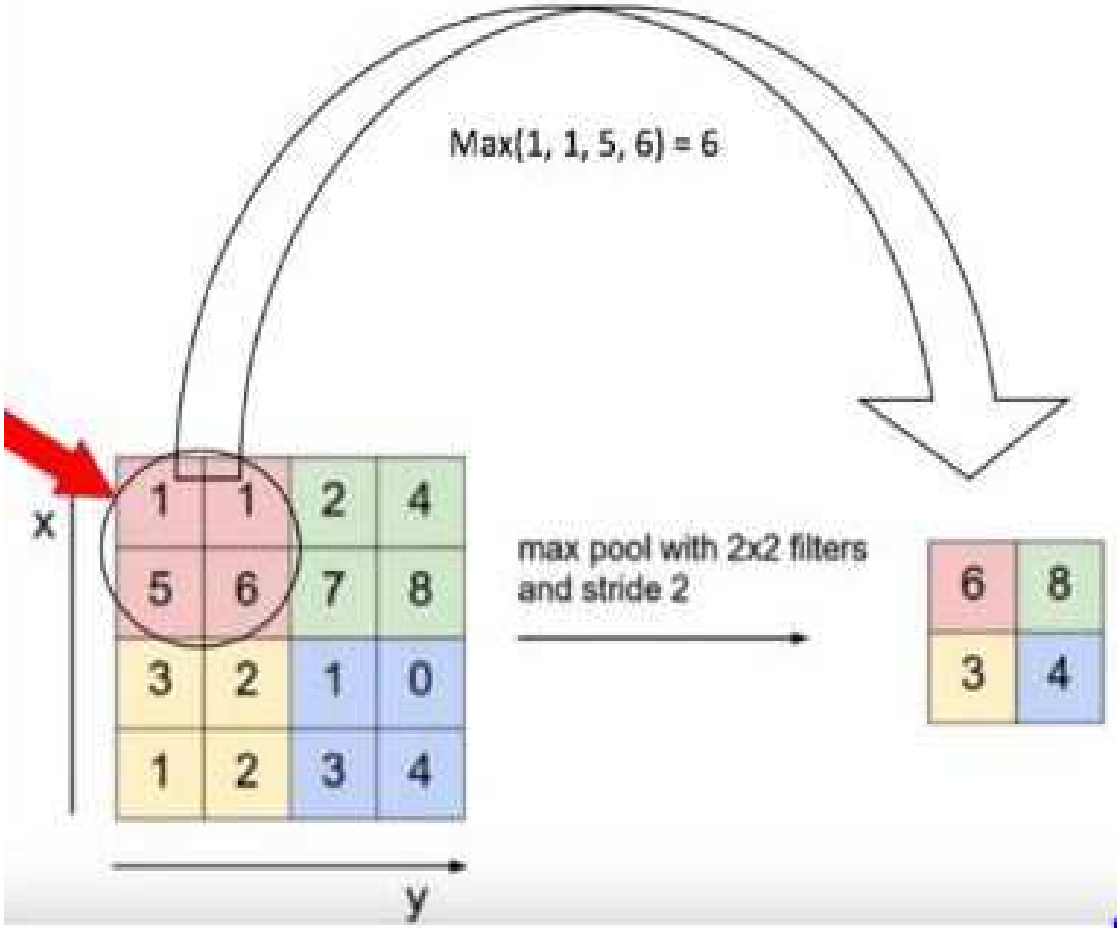


Figure 6.10: Example of max-pooling applied to a $4 \times 4$ image to reduce it to a $2 \times 2$ image. This size reduction leads to greater computational efficiency. According to Karpathy and others, CNN architectures should move way from subsampling. Figure adapted from Youtube video of Deep Learning for Computer Vision (Andrej Karpathy, OpenAI).

information gets lost.

The combination of the operations *convolution* − *activation* and *pooling* is considered one layer. This sequence of layers is repeated with different hyperparameters for each layer until we get to the last few layers. The hyperparameters are the design parameters, such as number of filters, filter size, number of layers, that don't get updated in the iterative backprojection operations. In contrast, the weights $w_{i-j}^{[n]}$ get updated every iteration and are denoted as parameters.

### 6.2.5 Fully-Connected Layer

The last few layers combine all the local FMs into a $fully - connected$ image by concatenating all of the vectors associated with each FM into one tall vector. For example, if there are two $3 \times 3$ FMs then each one can be transformed into a $9 \times 1$ vector, and these two vectors are concatenated to form the *tall* $18 \times 1$ vector. This tall vector is then used as input into a fully-connected layer. A fully-connected layer combines all other parts of the input image into every node of the next layer. It can bring together many local features to form a much larger image.

### 6.2.6 Soft-Max Layer

In image classification there might there might be dozens of classes, not just two as in binary classification. In this case a soft-max activation function $g(\mathbf{z})$ can be used in the output layer such that

$$g(\mathbf{z}_i) = \frac{e^{z_i}}{\sum_{i=1}^{M} e^{z_i}}, \tag{6.6}$$

where the summation is over all the $M$ nodes in the last output layer. The typical output vector will typically consist of positive numbers between 0 and 1 and can be interpreted as the probability of the input image being in one of the $M$ classes. Each of the $M$ output nodes is associated with a unique class. The one with the highest probability is selected as the class of the input image.

### 6.2.7 Loss Function

Which loss function should be used? For binary classification the cross-entropy loss function

$$\epsilon = \sum_{i=1}^{M} y_i \ln y_i, \tag{6.7}$$

over the mean square error (MSE) loss function

$$\epsilon = 1/2 \sum_{i=1}^{M} (i_i - y_i^{obs})^2. \tag{6.8}$$

is preferred (see Figure 6.11). Cross entropy has the fastest convergence rate as depicted in the example shown in Figure 6.12.

### 6.2.8 Dropout Regularization

Too many fully-connected layers can lead to overfitting. Thus, a regularization method such as damping is needed to stabilize the solution. However, overfitting does always occur because with some deep networks the data cannot be fitted well even with many layers (see Figure 6.13). There are many regularization methods, some of them are listed below.

1. More training sets? longer computation

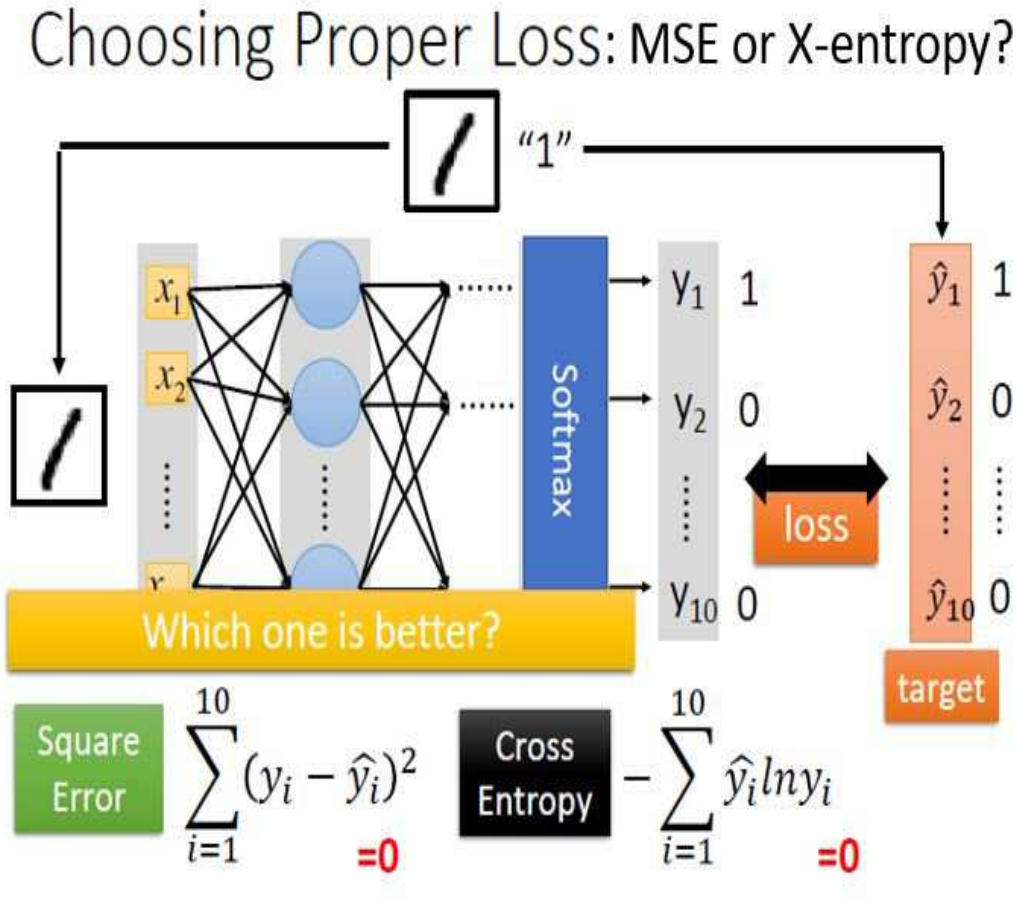


Figure 6.11: Which loss function is preferred? Figure adapted from Hung-yi Lee PPT.

2. Early stopping
3. Marquardt damping: large weights are penalized or constrained
4. Bootstrap: classify different subsets of the training data, and fit a model which is based on these subsets.
5. Limiting the number of hidden layers and units
6. Dropout: A successful way to prevent overfitting is to perform a dropout. Here units are randomly removed from the neural network, which can also be seen as a form of adding noise to the network. p=0.5and removing units in the input layer withp=0.2

A popular regularization method is dropout regularization (Nitish et al., 2014). As indicated in the wikipedia page *At each training stage, individual nodes are either "dropped out" of the net with probability* $1-p$ *or kept with probability* $p$*, so that a reduced network is left; incoming and outgoing edges to a dropped-out node are also removed. Only the reduced network is trained on the data in that stage. The removed nodes are then reinserted into the network with their original weights.*

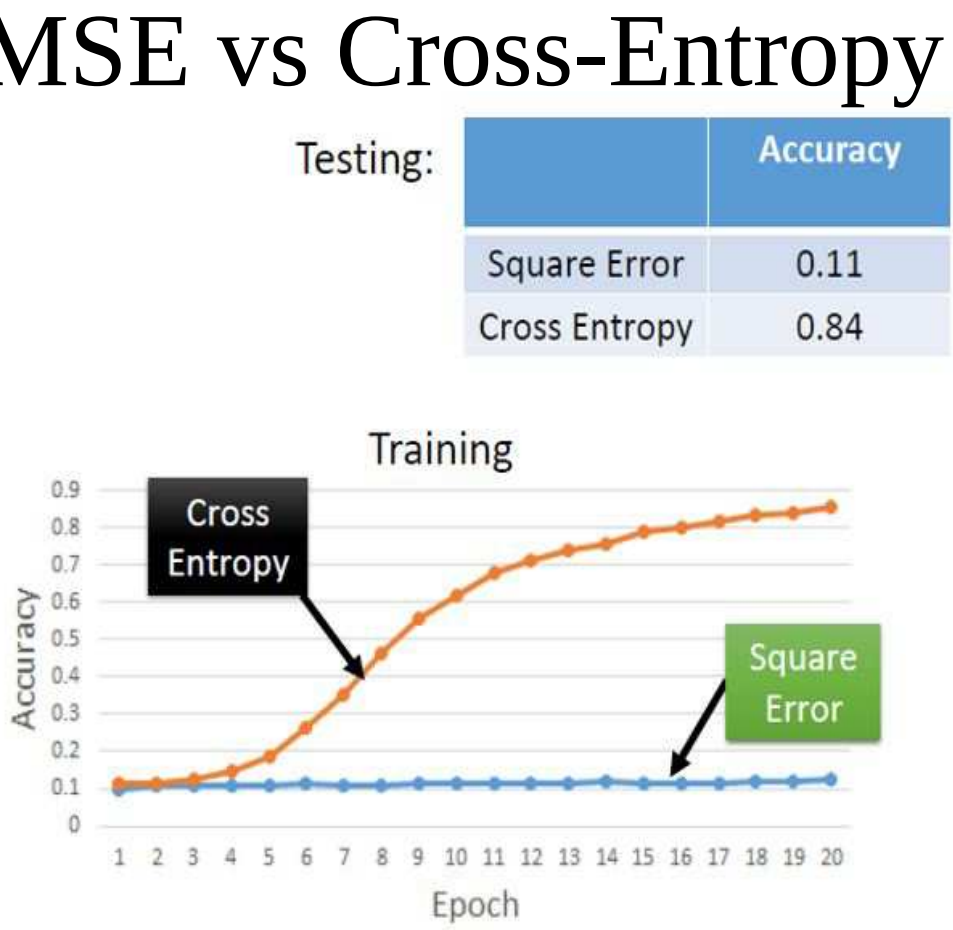


Figure 6.12: Accuracy vs iteration number for MSE and cross-entropy loss functions. Figure adapted from Hung-yi Lee PPT.

*In the training stages, the probability that a hidden node will be dropped is usually* 0.5*; for input nodes, this should be much lower, intuitively because information is directly lost when input nodes are ignored.*

*At testing time after training has finished, we would ideally like to find a sample average of all possible* $2^n$ *dropped-out networks; unfortunately this is unfeasible for large values of* $n$*. However, we can find an approximation by using the full network with each node's output weighted by a factor of* $p$*, so the expected value of the output of any node is the same as in the training stages. This is the biggest contribution of the dropout method: although it effectively generates* $2^n$ *neural nets, and as such allows for model combination, at test time only a single network needs to be tested.*

*By avoiding training all nodes on all training data, dropout decreases overfitting. The method also significantly improves training speed. This makes model combination practical, even for deep neural nets. The technique seems to reduce node interactions, leading them to learn more robust features that better generalize to new data.*

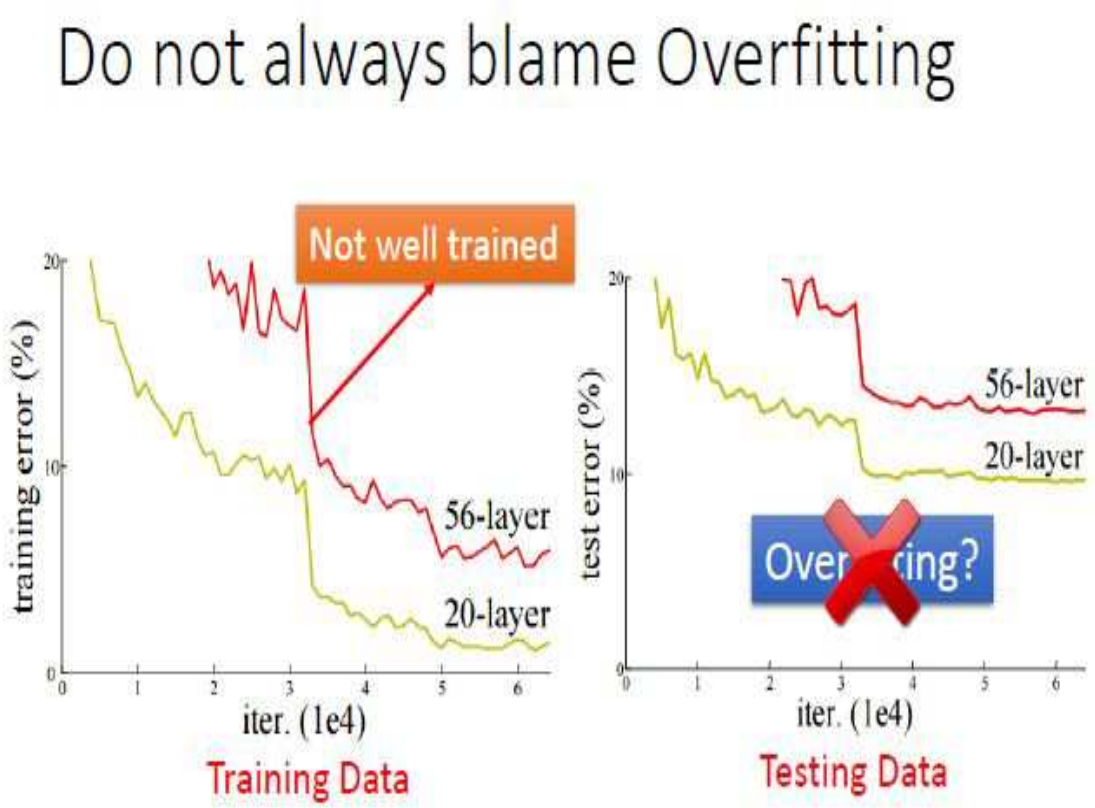


Figure 6.13: Even with deep networks with many unknowns overfitting does not always occur. The deeper the network for standard architectures, the more likely the gradient tends to vanish so the convergence stalls. ResNet tends to fix this problem. Figure adapted from Hung-yi Lee PPT.

### 6.2.9 DropConnect Regularization

Again, wikipedia saya the following about DropConnect regularization: DropConnect (Perez, 2013) is the generalization of dropout in which each connection, rather than each output unit, can be dropped with probability $1 - p$. Each unit thus receives input from a random subset of units in the previous layer.

DropConnect is similar to dropout as it introduces dynamic sparsity within the model, but differs in that the sparsity is on the weights, rather than the output vectors of a layer. In other words, the fully connected layer with DropConnect becomes a sparsely connected layer in which the connections are chosen at random during the training stage.

### 6.2.10 Local Response Normalization(LRN) Regularization

Local Response Normalization(LRN) type of layer turns out to be useful when using neurons with unbounded activations (e.g. rectified linear neurons), because it permits the detection

of high-frequency features with a big neuron response, while damping responses that are uniformly large in a local neighborhood. See $https://stats.stackexchange.com/questions/145768/importance-of-local-response-normalization-in-cnn$.

To excite all the neurons equally in a layer LRN is used. There are some reports that claim it is not used very much anymore. However its definition is below.

**Original Formula:** For every particular position $(x, y)$ and kernel $i$that corresponds to a single 'pixel' output we apply a 'filter', that incorporates information about outputs of other $n$ kernels applied to the same position. This regularization is applied before activation function.

$$b^i_{xy} = \frac{a^i_{xy}}{(k + \alpha \sum_{max(0,i-n/2)}^{min(N-1,i+n/2)} (a^i_{xy})^2)^\beta} \tag{6.9}$$

where $b^i_{xy}$ is the regularized output for kernel i at position $(x, y)$, where $a^i_{xy}$ is the source output for kernel i applied at position $(x, y)$, $N$ is the total number of kernels, $n$ is the size of the normalization neighborhood, and $\alpha$, $\beta$, $k$ and $n$ are hyperparameters.

In practice two approaches can be used:

1. Within Channel. Normalize over local neighborhood of a single channel (corresponding to a single convolutional filter). In other words, divide response of a single channel of a single pixel according to output values of the same neuron for pixels nearby.

2. Across Channels. For a single pixel normalize values of every channel according to values of all channels for the same pixel.

LRN was used more often during the days of early convets like LeNet-5. Current implementation of GoogLeNet (Inception) in Caffe often uses LRN in connection with pooling techniques, but it seems to be done for the sake of just having it. Neither original Inception/GoogLeNet (here) nor any of the following versions mention LRN in any way. Also, TensorFlow implementation of Inception (provided and updated by the team of original authors) networks does not use LRN despite it being available.

Many types of normalization layers have been proposed for use in ConvNet architectures, sometimes with the intentions of implementing inhibition schemes observed in the biological brain. However, these layers have recently fallen out of favor because in practice their contribution has been shown to be minimal, if any.

### 6.2.11 Mini-Batch

The preferred approach to finding the weights is to divide up the training set into mini-batches of training examples. The starting weights are randomized, and the updated weights are found by one iteration for one mini-batch of input data (see Figure 6.14). That is, the gradient is formed from

$$\nabla\epsilon = \nabla(l^1 + l^{31}...) \tag{6.10}$$

where $l^i$ represents the loss function in Figure 6.14. These updated weights are used as the starting weights for the next mini-batch and this process is repeated until all the mini-batches have been used. This is known as one epoch. In the next epoch the data are

resorted into non-overlapping mini-batches and the weights are updated for the next epoch.

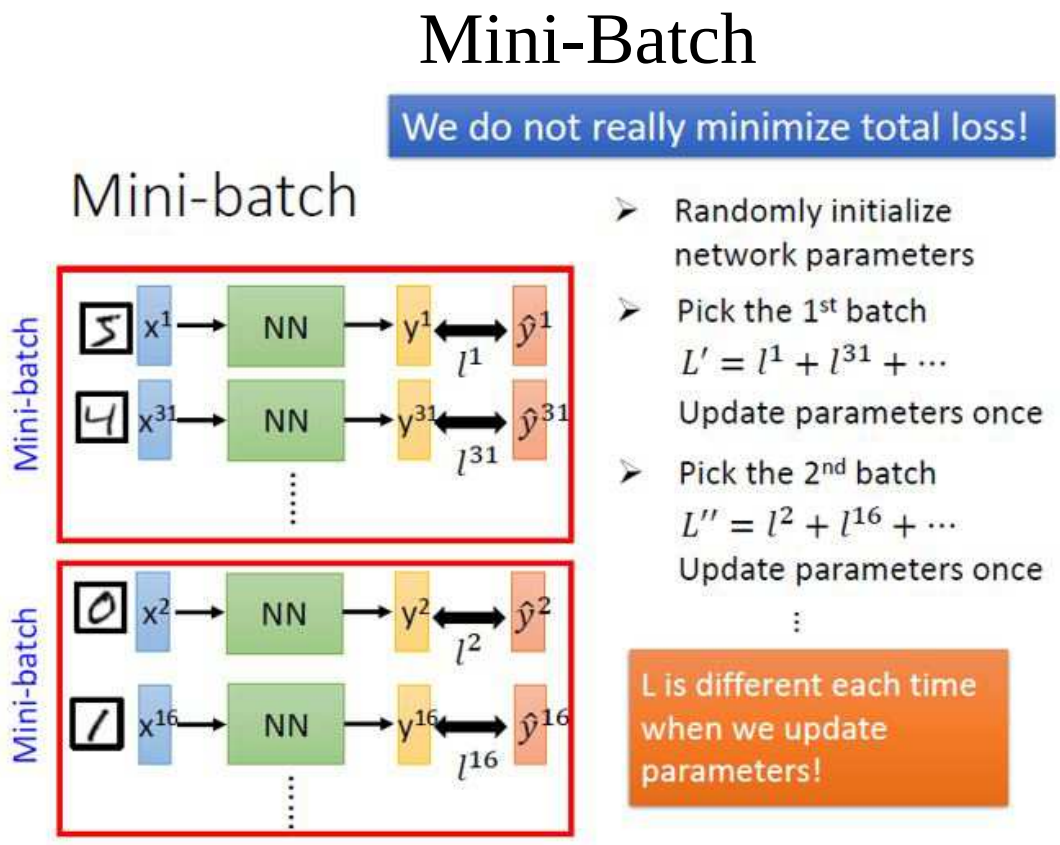


Figure 6.14: Weights are updated by one pass through each mini-batch of data in the training set. It is recommended that the training examples are reshuffled after each epoch. Figure adapted from Hung-yi Lee PPT.

### 6.2.12 Step Length

The step length $\alpha$ in the steepest descent formula is also known as the learning rate. Typically the learning rate is large for the early iterations (e.g., $\alpha = .0.5$) and then decrease $\alpha$ after several epochs. One formula suggested by Hung-yi Lee is

$$\alpha^k = \frac{\alpha}{\sqrt{k+1}}, \tag{6.11}$$

where $k$ is the iteration index for different epochs.

Another strategy is to replace $\sqrt{k+1}$ by $\sum_{i=1}^{k} g_i^2$ where $g_i$ is the gradient for the $i^{th}$ iteration. According to Hung-yi Lee this is known as John Duchi's Adagrad. In this case smaller gradients get greater learning rates in order to increase their importance. Of course we should also use momentum as discussed in an earlier chapter.

In summary the training strategy is summarized in Figure 6.15. We first train in the training set and then test the validity of the weights on a non-overlapping validation set. The validation sets should have the same character as the original training set and usually it is about 20% in size.

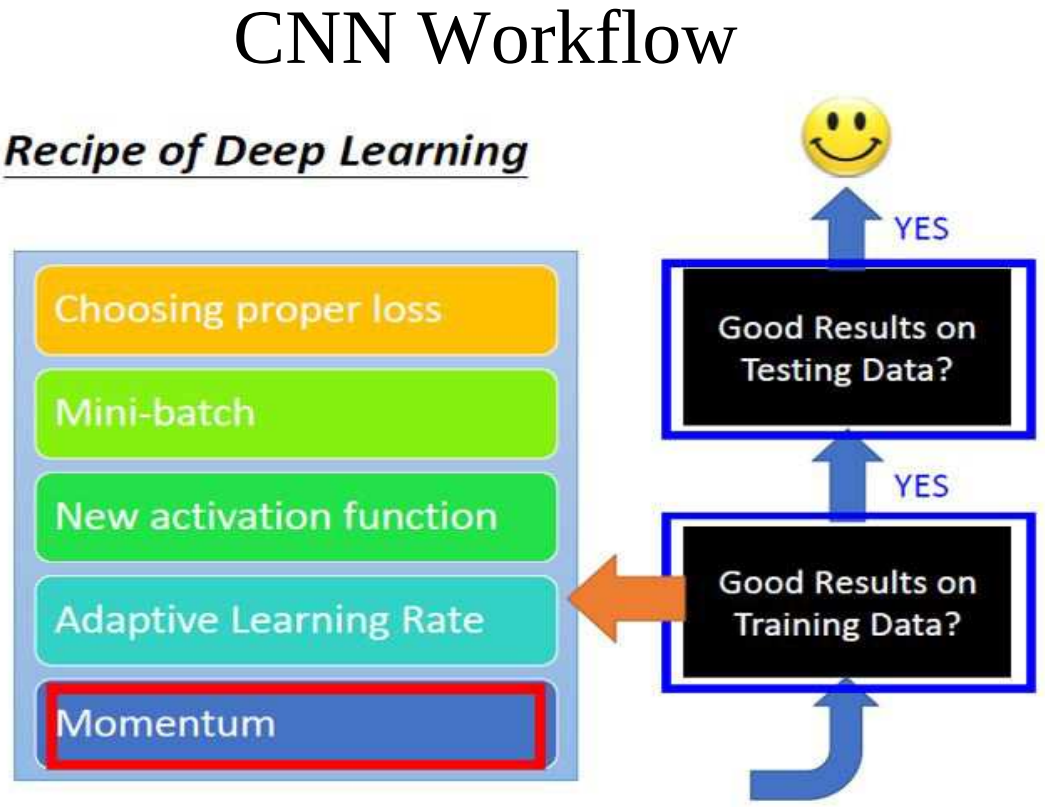


Figure 6.15: Weights are updated by one pass through each mini-batch of data in the training set. It is recommended that the training examples are reshuffled after each epoch. Figure adapted from Hung-yi Lee PPT.

## 6.3 Architectures of CNN

There are several CNN architectures that are most popular today and their performances are graphed in Figure 6.4. The VGG net is quite appealing from a programming point of view because it uses the same number of nodes in each layer. These different architectures appear to have evolved by trial and error, where some intuition is used to help improve the design. No mathematical basis appears to justify their usage.

### 6.3.1 AlexNet

The AlexNet architecture is shown in Figure 6.16. Alexnet is one of the pioneer Deep Neural Networks which aim to classify images. It was developed by Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton and won an Image classification Challenge (ILSVRC) in 2012 by a large margin. At that time the other competing algorithms were not based on Deep Learning. Now, and since then, they almost all are. This net had a huge impact on the domain and most of following nets were more or less based on its architecture. Alexnet (Figure 6.16) is composed of 5 convolutional layers (C1 to C5 on schema) followed by two fully connected (FC6 and FC7), and a final softmax output layer (FC8). It was initially trained to recognize 1000 different objects.

The intuition behind this net is that each convolutional layer learns a more detailed representation of the images (feature map) than the previous one. For example the first layer is able to recognize very simple forms or colors, and the last one more complex forms such as full faces for instance

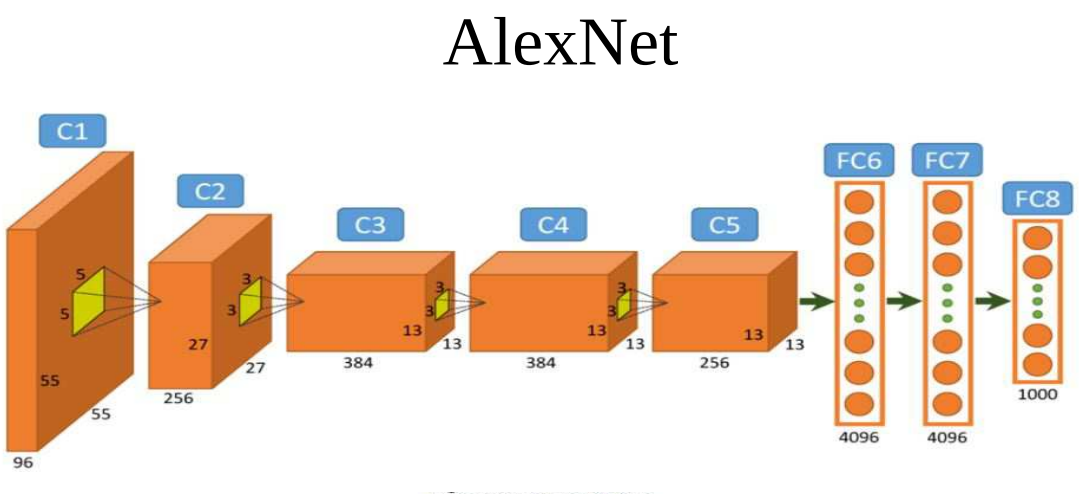


Figure 6.16: AlexNet architecture. Figure and text from *https* : *//www.saagie.com/blog/object − detection − part*1.

### 6.3.2 ZFNet

The ZFNet architecture is shown in Figure 6.17. ZFNet has the same global architecture as Alexnet, that is to say 5 convolutional layers, two fully connected layers and an output softmax one. The differences are, for example, better sized convolutional kernels.

### 6.3.3 VGGNet

The pleasing feature of VGGNet in Figure 6.18 is that the filter size ($3 \times 3 \times n_c$ and down-sampling fraction stays the same for every layer. Figure 6.4 shows that it also outperforms earlier architectures such as the LeCUN one shown in Figure 6.3.

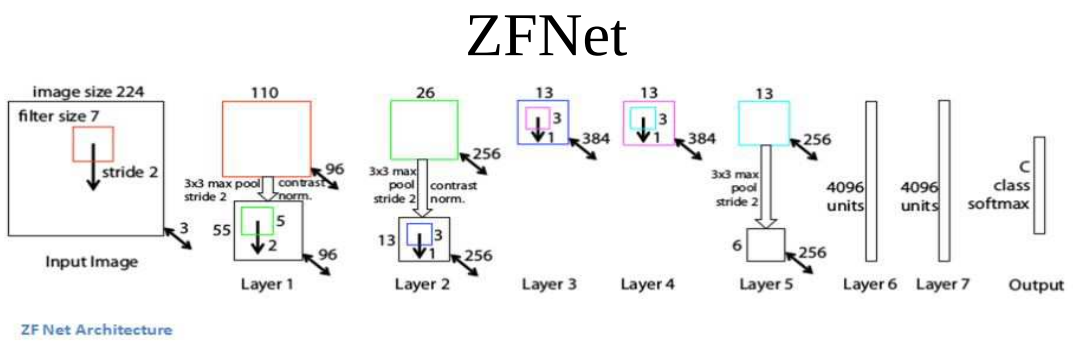


Figure 6.17: ZFNet architecture. Figure and text from *https* : *//www.saagie.com/blog/object − detection − part*1.

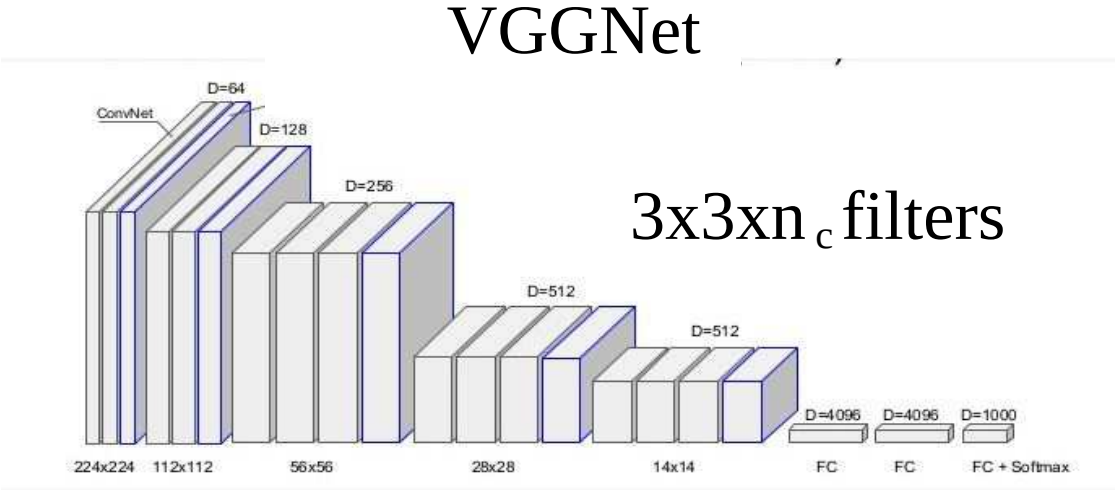


Figure 6.18: Architecture of VGGnet where $D$ is the number of channels and the $x-z$ size of each feature map decreases the deeper into the CNN.

A schedule for a VGG CNN is depicted in Figure 6.19 where the filter $x-z$ size stays the same for every layer. However, the number of channels or filter dimension in the channel dimension doubles after every pooling. The input image has a dimension of $224 \times 224 \times 3 = O(150K)$. There are 64 filters in the first layer so there are 64 feature maps in the first layer, each of them with a dimension of $224 \times 224$. The filter size is $3 \times 3 \times \times 3$ for each FM so that the total number of unknowns in the 1st-layer is $(3*3*3)*64 = 1,728$, not counting biases. The 64 FMs requires a total storage of $224*224*64 = O(3\ 10^6)$ variables. Another convolution layer is used with the filter size of $3 \times 3 \times \times 64$ with $O(36K)$ unknowns. Then pooling is applied to halve the size of the FM. The number of channels doubles after every pooling.

VGG is a very deep and simple net. In the most common version, it has 16 layers (The blue pooling layers aren't counted on the schema). However the global architecture is very similar to the Alexnet one. Actually the Alexnet convolutionnal layers are here represented

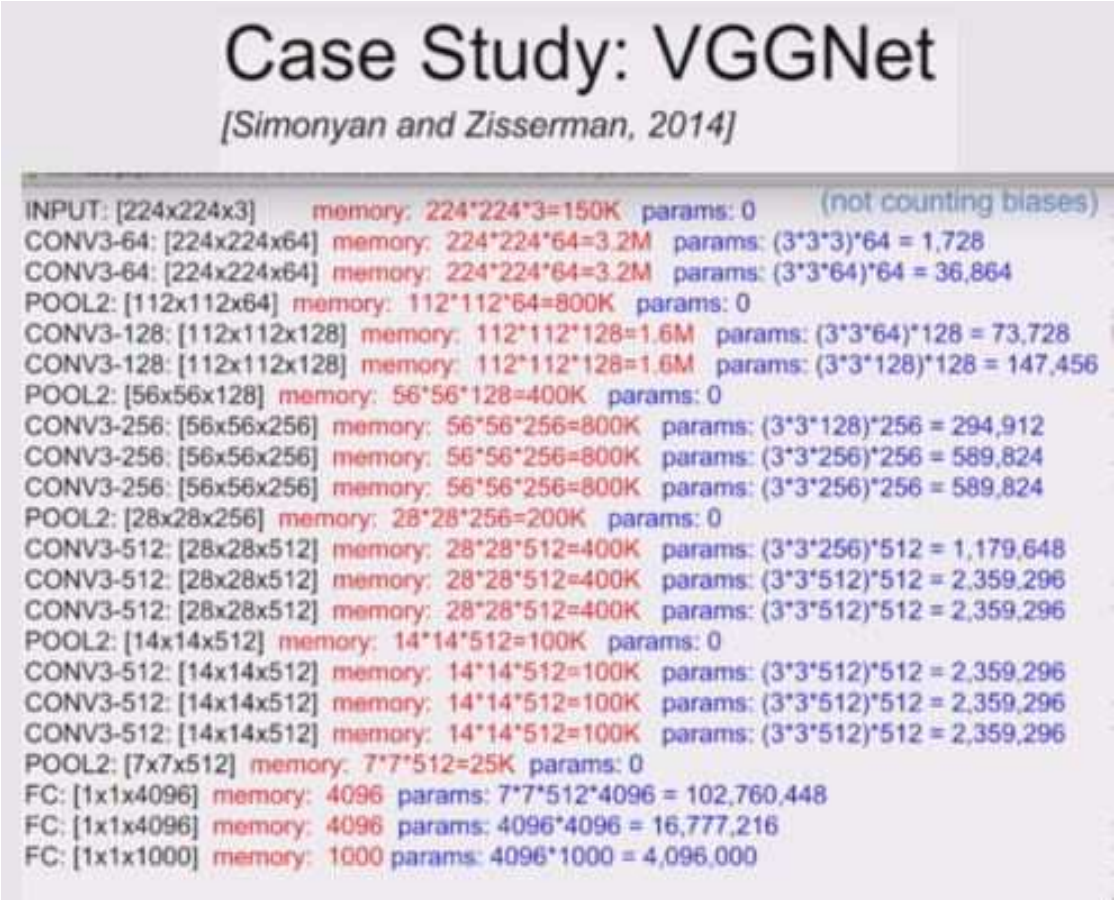


Figure 6.19: Hyperparameter values for a VGG CNN. The input image has a dimension of $224 \times 224 \times 3 = O(150K)$. Figure adapted from Youtube video of Deep Learning for Computer Vision (Andrej Karpathy, OpenAI).

by two or three following convolutional layers. Another difference is that each convolutional layers has a 3x3 kernel unlike the other nets thats have different sized kernels for each layer.

### 6.3.4 GoogleNet

The Google CNN architecture is depicted in Figure 6.20 and has an inception layer with several convolutions with different filters and a pooling layer. It won the ILSVRC contest for 2014 by having the lowest percent errors for the testing set.

### 6.3.5 ResNet

ResNet is the 2016 ILSVRC contest winner with an error rate of less than 5%. Its architecture is shown in Figure 6.21, where the innovation is that it adds an identity operator to the output of an activation function. Assume the $i^{th}$ input is $z_i^{[n]}$ and the $i^{th}$ output of a layer's standard operations is $a_i^{[n]}$. The ResNet architecture says the $i^{th}$ output is $a_i^{[n]} + z_i^{[n]}$.

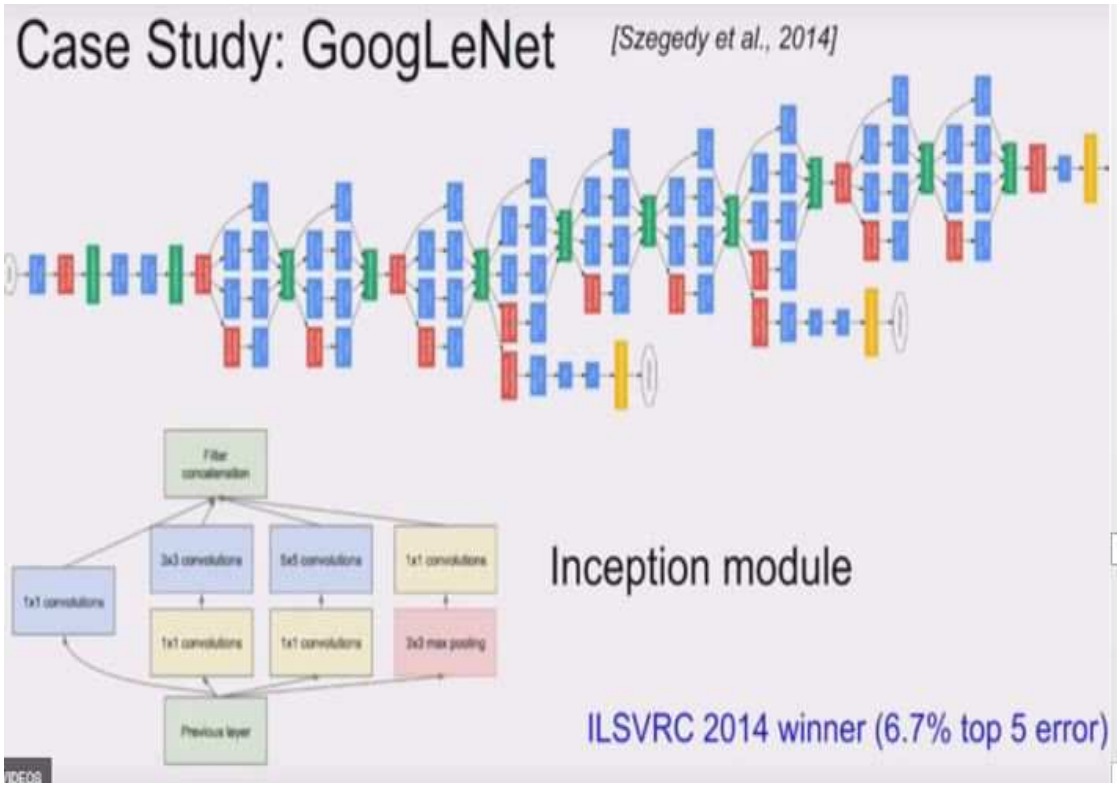


Figure 6.20: Architecture for GoogLeNet CNN. The inception module contains several convolution operations with different filters. Figure adapted from Youtube video of Deep Learning for Computer Vision (Andrej Karpathy, OpenAI).

This can be modified by saying the $i^{th}$ output is $a_i^{[n+x]} + z_i^{[n]}$, where $x$ is some integer larger than 1.

ResNet-152 introduced the concept of residual learning in which the subtraction of feature is learned from the input of that layer by using shortcut connections (directly connecting input of $n^{th}$ layer to some $(n+x)^{th}$ layer, which is shown as curved arrow). It has proven that the residual learning can improve the performance of model training, especially when the model has deep network with more than 20 layers, and also revolve the problem of degrading accuracy in deep networks.

Which CNN architecture should one use and should one try to innovate with new ones? The answer according to Hinton, Karpathy and Ng is to "not be a hero". Use one of the latest ones that work best with the ILSVRC contest, perhaps delete some parts. Karpathy says to play with the regularization strength and dropout rates.

## 6.4 Deep Learning Software and Youtube Classes

Karpathy points out a number of CNN software packages, and recommends Keras as the easiest to use, similar to Python. It is high level language based on the Torch/Tensorflow software which is for pros. Once you master the Torch language, you can develop your

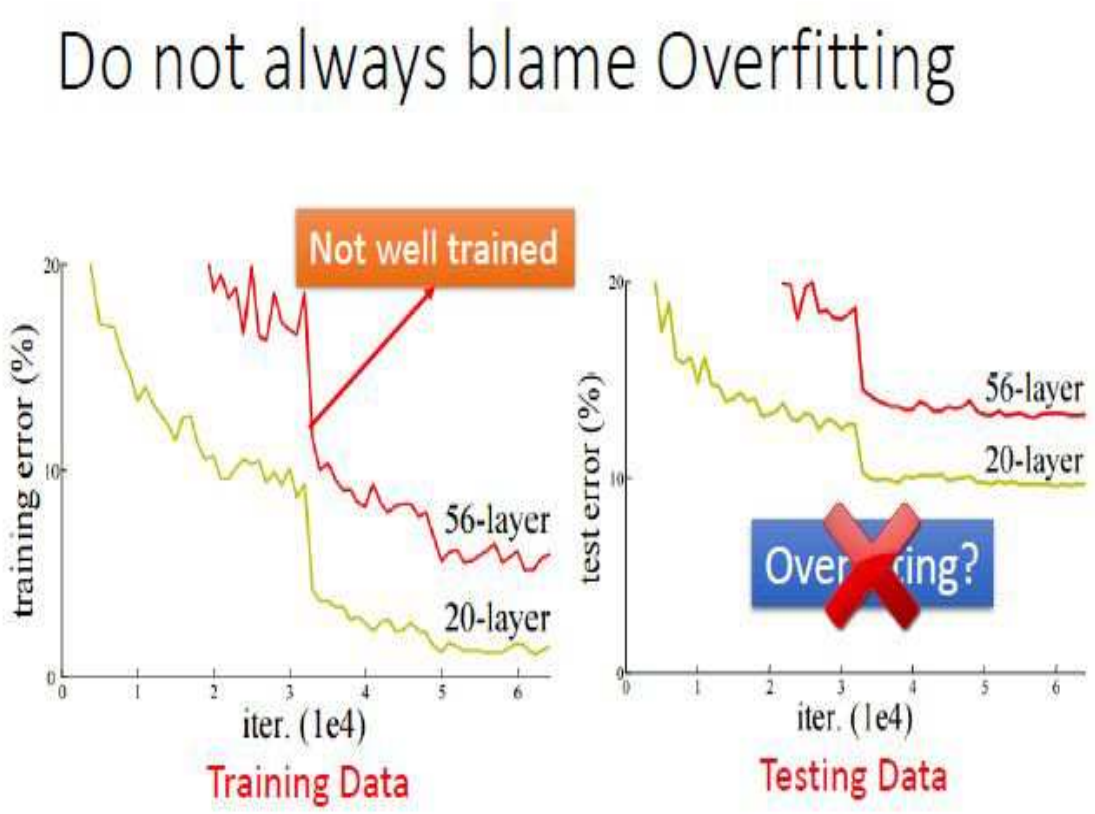


Figure 6.21: Architectures for (left) VGG and (middle) ResNet CNN.

own special architectures. For some reason he doesn't list Cafe as one of his top three favorites, but he notes the nice features of 1). Run a script to convert data, 2). define CNN architecture, 3). define silver, and 4). train with pretrained weights. Now I understand, you don't train you simply feedforward! He doesn't list Theano or Lasagne as his favorites.

What about MATLAB? More details of his course are at cs231n.stanford.edu, he seems like an excellent lecturer. I also recommend Andrew Ng's Youtube videos on Deep Learning.

If you want to start your own company, the advice in 2016 is to buy NVIDIA DGX-1 (P100 GPUs) or NVIDIA DIGITS DevBox (Titan X GPUs). Karpathy claims the cloud does not offer good GPU services yet.

## 6.5 Seismic Fault Interpretation by CNN

Xiong et al. (2018) used a CNN system to identify faults in seismic data. Unlike previous attempts that used features extracted from seismic data as input (Zhang et al., 2014), Xiong et al. (2018) used three slices from the migration image as the input (see Figure 6.22) and used an attribute image as the output image (see Figure **??**). In the attribute image they identified the faults and labeled them as such. Xiong et al.'s (2018) schematic for the CNN

architecture is shown in middle of Figure 6.22. Some examples from the training set are given in Figure 6.23.

The target output values in the training set at each point $O$ can be can be interreted as faults or not by interpreters or auto-picking algorithms. In their paper, they classified eight different 3D seismic image cubes of real data using a skeletonized-seismic-coherence-based auto-picking method (Qi et al., 2017). After annotation, every point in the training cubes is labeled as fault or non-fault using a threshold strategy: the points with skeletonized coherence value larger than a predefined threshold is classified as fault, others are non-fault.

Before the input of three slices is fed into the network, it is normalized by subtracting the mean and dividing by the standard deviation. Similar to the classical CIFAR-10 classification problem (Krizhevsky and Hinton, 2009; Google, 2017), the network consists mainly of two convolutional (Conv) layers and two fully-connected (FC) layers followed by a softmax classifier, which gives the label prediction output (Bishop, 2006). The Conv layers both have 64 filters with sizes of $5 \times 5$. There are 384 and 192 feature maps in the two FC layers, respectively. After every Conv layer and FC layer, they applied rectified linear activation (ReLU). Max-pooling with both size and stride being 22 and local response normalization (LRN) are used after both Conv layers. The final softmax classifier produces a probability indicating the likelihood of a fault being presented in the center point of the input. The prediction of the label is then generated by applying a threshold (0.5) to the output likelihood values, such that the one above the threshold would be considered to be a fault location, while those below would not.

The network is trained from scratch by initializing the weights of all the layers as in the CIFAR-10 tutorial of Tensorflow (Google, 2017; Abadi et al., 2016). A gradient descent optimizer is implemented in Tensorflow with the default parameters and the initial learning rate, i.e. step length, being 0.1. The training samples are randomly shuffled before they are fed into the network for every epoch, which is a critical step to obtain good network performance. The model is saved when the error (or loss) function

$$\epsilon \quad = \quad \frac{1}{n} \sum_{i=1}^{n} \ln \; p(Y = y_i | \mathbf{x}_i) \tag{6.12}$$

reaches a plateau during the optimization process, and it is then validated with the validation dataset. The saved model achieved classification accuracies of about 73% on both the training and validation datasets.

The weights for the trained model are applied to the testing cube comprising a 3D seismic image volume of real data. The cube size is $1000 \times 655 \times 1083$. The top, front and side panels are showing time section ($T = 312$), inline section ($Y = 423$) and cross-line section ($X = 417$), respectively. Locations of the sections are marked by the black lines in the figure. One of the eight cubes is randomly selected for validation and will not be seen by the network during the training process, while the other seven cubes are used to generate the training dataset. The training dataset is constructed by randomly selecting 50000 points from each cube labeled as faults (which is approximately 0.2% of all fault points in the cubes), and another 50000 points labeled non-fault. The number of non-fault points is much larger than fault points in the cubes. For this real data cube, the CNN obtains about 74% classification accuracy; close to that obtained for training and validation datasets.

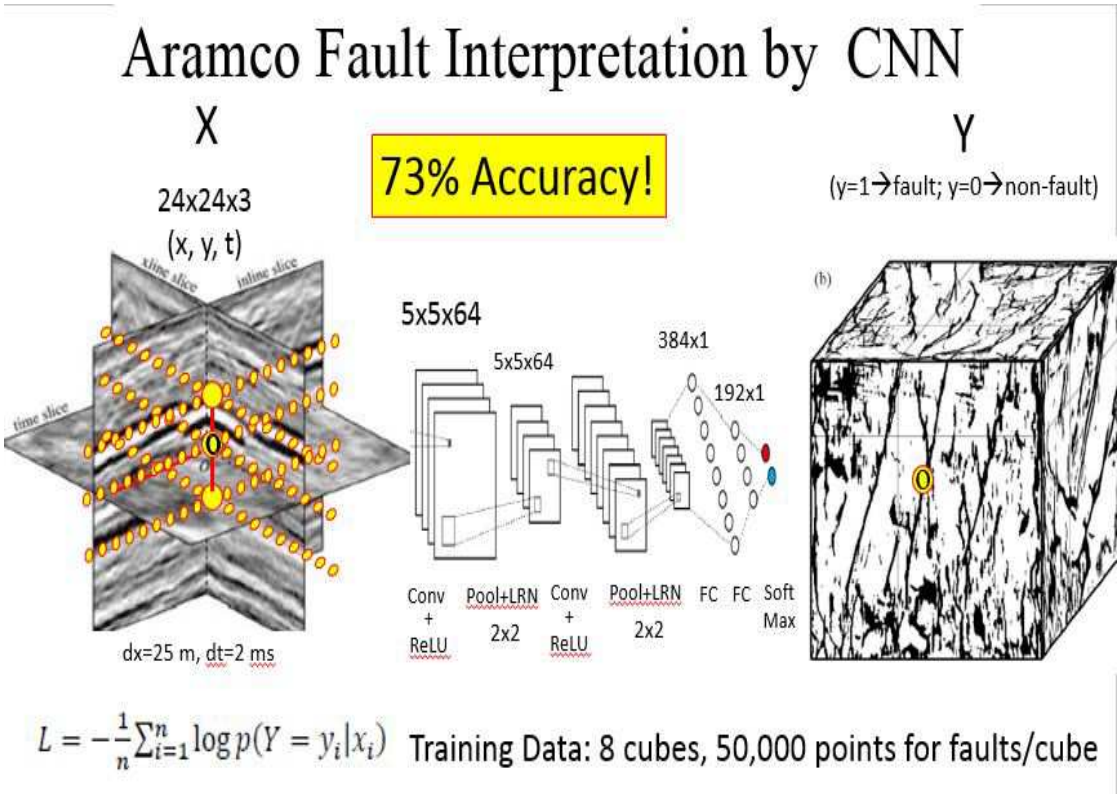


Figure 6.22: Three slices on the left are taken from a 3D migration cube and used as the input data to the CNN. The output is the point $y_i$ at $o$ on the right which is taken to be either a fault ($y_i = 1$) or not ($y_i = 0$). The $24 \times 24 \times 3$ filter is shifted over to a new position $o'$, and the dot product of the filter with the image points is computed again. Repeating this for all input points gives the first feature map. Note, the filter is not shifted up or down and only shifted along the lateral directions for this training example. A new training example will have the center point of the filter shifted up or down. Figure adapted from Xiong et al. (2018).

Figure 6.23 shows some representative fault and non-fault samples randomly selected from the training dataset. While it is not easy for human eyes to distinguish every single fault and non-fault sample, we can see a systematic difference between the two classes. Non-fault samples show more continuous seismic events. A validation dataset of the same size is constructed in the same way as the training dataset, which is used to monitor the training process and determine the termination of training.

The trained model is applied to the testing cube comprising a 3D seismic image volume of real data as shown in Figure 6.24. The cube size is $1000 \times 655 \times 1083$. The top, front and side panels are showing time section (T=312), inline section (Y=423) and cross-line section (X=417), respectively. Locations of the sections are marked by the black lines in the figure. For this real data cube, the CNN obtains about 74% classification accuracy; close to that obtained for training and validation datasets.

The fault probability cube output by the CNN is shown in Figure 6.24a along with results

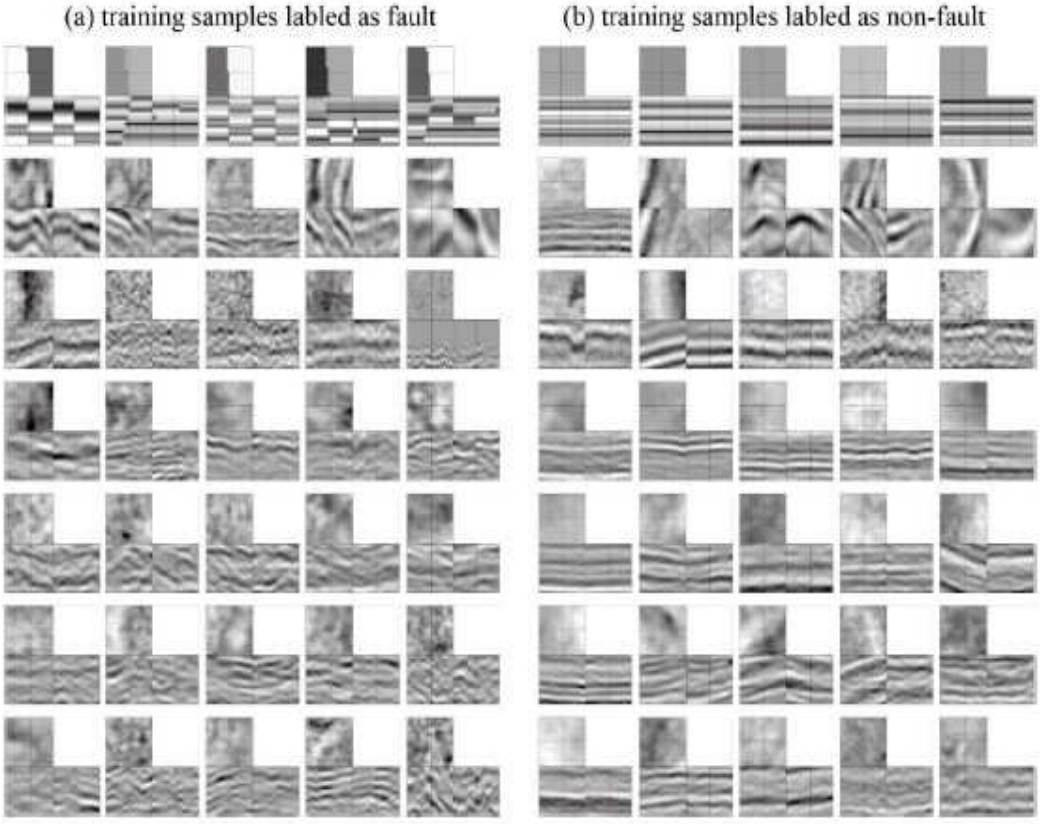


Figure 6.23: Samples labeled as (a) fault and (b) non-fault. All samples are randomly selected from the training dataset, except the first row showing synthetic samples. Each sample is composed of front, side and top panels which are inline, cross-line and time slices, respectively. The size of each slice is $24 \times 24$. The spatial grid size is 25 meters while the vertical time sampling interval is either 0.002 or 0.004 seconds for different cubes. Figure adapted from YXiong et al. (2018).

from a seismic coherence cube (Figure 6.24b). Seismic coherence is a well-known and widely used attribute to highlight discontinuities in seismic image (Bahorich and Farmer, 1995). As we can be seen in Figures 6.24 and 6.25, the CNN results show a higher resolution compared to the coherence volume. The seismic faults as well as channels are clearer in the CNN results. Recalling that the fault probability calculation is made independently for different points, the clear continuous outlines of discontinuities in the fault probability images show that the trained network performs robustly in the presence of noise.

For the test with the 3D image cube of size $1000 \times 655 \times 1083$, it took about 2.5 hours to obtain the fault probability result using a computer cluster with 20 nodes (40 CPU cores in each node). So this method is still practical even without possible improvements by reducing duplicate calculations in adjacent locations.

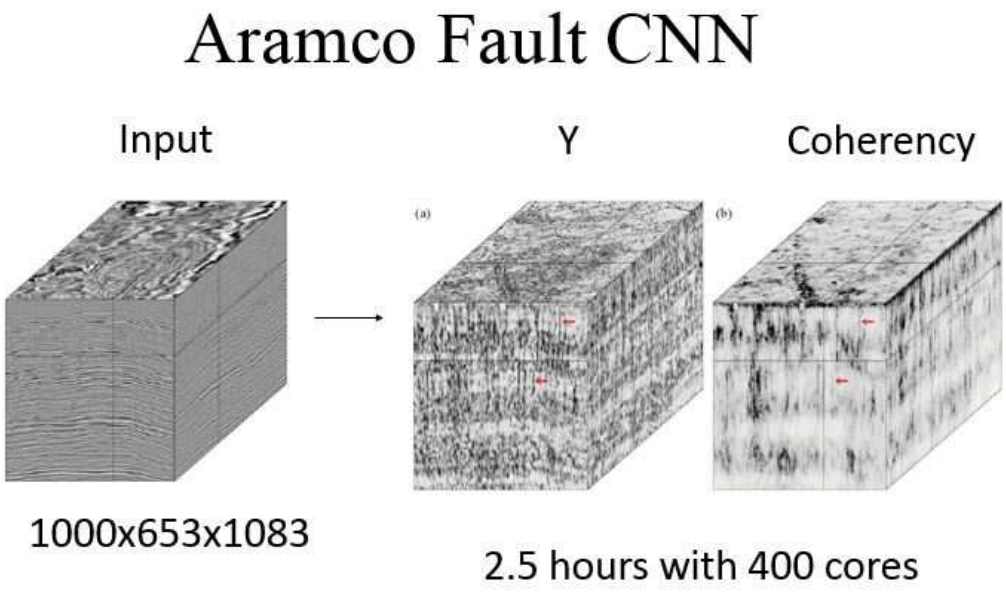


Figure 6.24: Real data example showing the fault probability cube from the CNN prediction, compared with the corresponding coherence cube. The cube size is $1000 \times 655 \times 1083$. The spatial grid size is 25 meters while the vertical time sampling interval is 0.002 seconds. Figure adapted from Xiong et al. (2018).

## 6.6 Summary

The characteristics of a CNN are described. Its main advantage over classical neural networks is that its convolutional nature leads to a significant increase in computational efficiency by more than several orders of magnitude. It also more closely mimicks the actual processing of the brain's neurons in processing visual information in the brain. The CNN uses a weighted sum of local pixel values to give the input the $j^{th}$ node of the next layer. The set of $N$ weights $w_i \ for \ i \in [1, 2 \in N]$ are shared for computing the input into any node of the next layer.

An example is presented for using a CNN to identify faults in a migration image. The result is a more accurate identification of faults than provided by the interpretation of a coherency cube. This is a promising result and gives optimism that CNN's will be a useful tool for many aspects of exploration geophysics.

## 6.7 Exercises

1. Describe how a migration deconvolution filter might be computed using CNN. Describe the effects of using convolution filters less than a wavelength or much larger than a wavelength.

2. Describe how a Hessian inverse might be obtained by computing $m = L^T d$, $Lm = \tilde{d}$, and $\tilde{m} = L^T \tilde{d}$, so that $\tilde{m} = L^T L m$. In this show how to design a CNN so that the

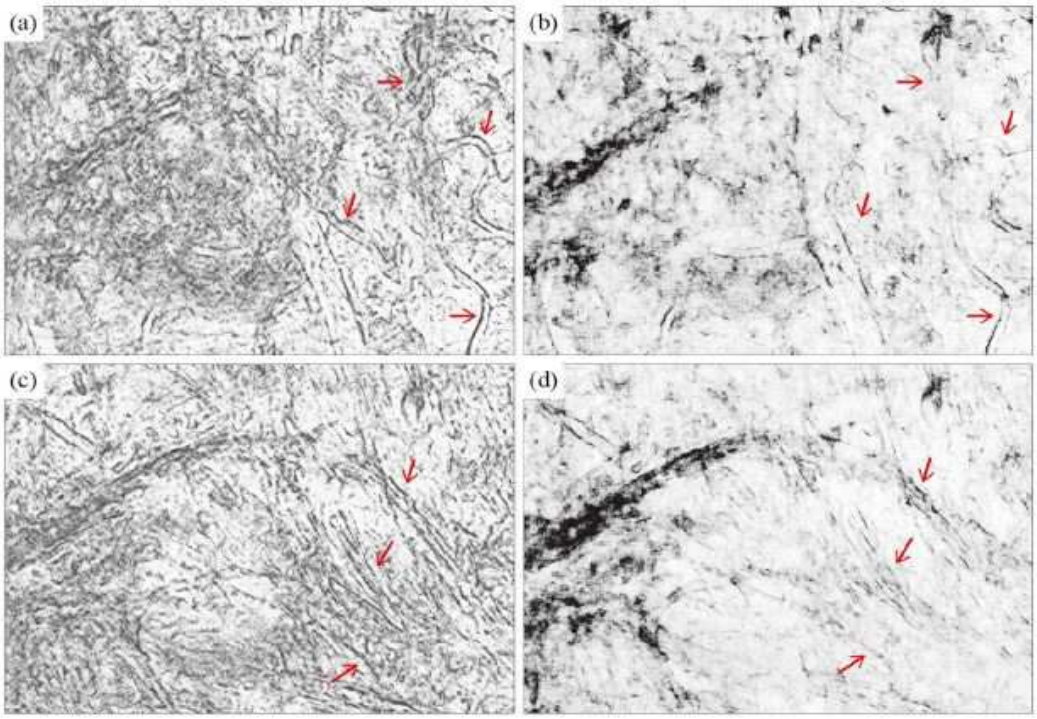


Figure 6.25: Time slices of fault probability [(a) and (c)] from CNN prediction, compared with seismic coherence [(b) and (d)], at two different time slices, $T = 312$ (top row) and $T = 387$ (bottom row), respectively. The section size is $655 \times 1083$. The spatial grid size is 25 meters. Figure adapted from YXiong et al. (2018).

weights for an inverse Hessian are obtained such that $m = [L^T L]^{-1}\tilde{m}$. Is this the same inverse Hessian computed in exercise 1? Explain.

3. The first layer of a CNN supposedly decomp[oses an image into smaller components associated with edges oriented at certain angles. How can one use local damped regularization and local slant stacking to encourage the filters in the first layer to output FMs with edges oriented in different directions?

4. Too large of a filter means that the high wavenumber data have been lost. Develop a multiscale CNN that first finds weights that fit the low-wavenumber features of the input data, and then finds the weights for the high-wavenumber features. Explain why this might be important for speeding up convergence as is done in a multigrid method. Would it be useful to have a low-pass filter or high-pass filter in each FM to reduce noise?

5. Write the pseudo-MATLAB code for dropout regularization.

6. Sketch the outline for developing CNN as a perfect absorbing boundary condition near the sides of the finite-difference computational grid. Should the constraint $||Lp = 0||^2$

be included in the objective function, where $Lp$ represents the acoustic wave equation. Test your CNN algorithm on analytical solutions to the acoustic wave equation in a homogeneous 2D medium.

7. How can FWI be combined with CNN to improve the performance of FWI below salt?

# Chapter 7

# Wave Equation Inversion and Neural Networks

We compare the full waveform inversion (FWI), skeletonized wave equation inversion (SWI), and supervised Machine Learning (ML) algorithms with one another. For velocity inversion the advantage of SWI over FWI is it is more robust and has less of a tendency in getting stuck at local minima. This is because SWI only needs to explain the kinematic information in the seismograms, which is less demanding than FWI's difficult task of explaining all of the wiggles in every arrival. The disadvantage of SWI is that it provides a tomogram with theoretically less resolution than the ideal FWI tomogram. In this case, the SWI tomogram can be used as an excellent starting model for FWI. SWI is similar to supervised Machine Learning in that both use skeletonized representations of the original data. Simpler input data lead to simpler misfit functions characterized by quicker convergence to useful solutions. I show how a hybrid ML+SWI method and the implicit function theorem can be used to extract almost any skeletal feature in the data and invert it using the wave equation. This assumes that the skeletal data are sensitive to variations in the model parameter of interest.

## 7.1 Introduction

Full waveform inversion is very ambitious, it seeks an earth model $\mathbf{m}$ that explains every complicated wiggle in a large set of very wiggly seismograms denoted by $\mathbf{d}$ (Tarantola, 1987). A basic assumption is that the forward modeling operator $\mathbf{L}$ (see Figure 7.1a) is a "good enough" representation of the actual physics of wave propagation in the Earth. Such ambitions and assumptions can sometimes lead to getting stuck in local minima, slow convergence and tomograms with unacceptable errors. To partly mitigate these problems a multiscale strategy can be employed. As an alternative, we present wave equation inversion of skeletonized data. Here, the large complicated data set is reduced to skeletonized data $\tilde{\mathbf{d}}$ which are much less complicated yet still contain essential information about the model parameters of interest. This means that the skeletonized data lead to a less complicated objective function that does not contain so many local minima associated with the one for FWI.

To invert the skeletonized data, skeletonized wave equation inversion (SWI) uses nu-

merical solutions to the wave equation to update the model by back-propagation of the weighted skeletonized data residual (Luo and Schuster, 1991; Lu et al., 2017). The result is a largely robust inversion algorithm free of layered media or high-frequency approximations. The main benefits are that the inversion algorithm is less prone to getting stuck in a local minimum, but at the cost of a reduction in model resolution. However, the inverted tomogram can be used as the starting model for FWI because it is likely to be sufficiently close to the actual earth model.

Recent examples (Lu et al., 2017) of skeletonized wave equation inversion include the following cases. 1). **Inversion of $Q(\mathbf{x})$ from diving waves and surface waves**. Here the skeletonized data $\tilde{\mathbf{d}}$ are the peak frequencies associated with the arrivals of diving waves or surface waves, and the output model is $Q(\mathbf{x})$. 2). **$\mathbf{V(x)}$ inversion by migration velocity analysis.** The skeletonized data $\tilde{\mathbf{d}}$ are the depth residuals or static shifts associated with plane-wave migration images in the common image gather domain. 3). **Inversion of $V(\mathbf{x})$ from the dispersion curves of surface waves.** Here, the input skeletonized data $\tilde{\mathbf{d}}$ are the dispersion curves of surface waves in the $k-\omega$ domain and the output model is the shear velocity for Rayleigh waves, Love waves, and the P-velocity model for near-surface guided waves.

The skeletonization strategy of SWI is similar to that of neural networks, where a complex data set is often reduced to its *essential features* (Bishop, 2006). These skeletonized *"features"* are input into the system of neural network layers depicted in Figure 7.1c. Here, the original data can be very complex but complexity can be reduced by an unsupervised learning algorithm such as a principle component analysis, singular value decomposition (Bishop, 2006) or some statistical measure of important features (Patel and Chatterjee, 2016). In principle, these less complex features are still rich in information about the model. As an example, Figure 7.2 depicts the workflow for classifying rock types using a supervised learning method (Patel and Chatterjee, 2016). Here the original input data consist of images $\mathbf{d}$ of thin sections and their classification $\mathbf{m}$, each one classified as a type of limestone by a geologist. For example, if there are three types of classified rocks then the $3\times 1$ target model vector $\mathbf{m}=(m_1,\ m_2,\ m_3)$ might be assigned $\mathbf{m}=(1,0,0)$ if it is a pure limestone, $\mathbf{m}=(0,1,0)$ if it is a weathered rock, and $\mathbf{m}=(0,0,1)$ if it is a shale-limestone. In practice, the assignment is over a range of classification numbers and $\mathbf{m}$ is a $7\times 1$ vector according to Figure 7.2.

To simplify the input data, the thin-slice images are skeletonized into histograms of red, blue, and green colors. These histograms are further skeletonized into second-order (variance), 3rd-order (skewness), and 4th-order (kurtosis) statistical parameters for each color histogram. The examples from this skeletonized training set are denoted as $(\tilde{\mathbf{d}},\mathbf{m})$, where for each image the input is a $9\times 1$ vector of statistical values from the three histograms. For pedagogical simplicity we avoid the superscript notation that denotes the $i^{th}$ example from the training set. After sufficient training with $N$ training examples, the coefficients of the neural network are inverted for and used to classify thin sections from out of the training set. In their case history, Patel and Chatterjee achieved more than a 90% accuracy in the classification of thin sections taken from outside the training set.

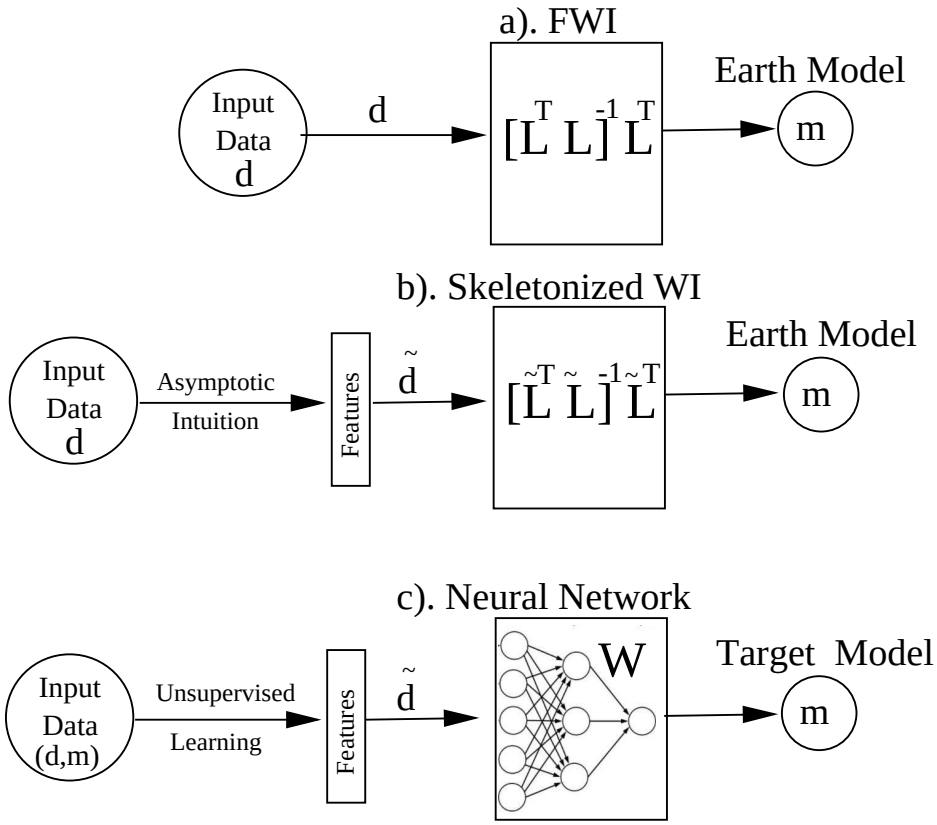


Figure 7.1: System diagrams for a) FWI, b) SWI, and c). a neural network. The neural network and SWI are similar in that the input data are skeletonized features of the raw data strongly influenced by small changes in the model. However, the neural network is first trained to get an estimate of $\mathbf{W}$, which is then used to estimate the target model $\mathbf{m}$ from untrained data.

## 7.2 Theory for Wave Equation Inversion of Skeletonized Data

The starting point for wave equation inversion of skeletonized data is to define an objective function

$$\epsilon \;=\; \frac{1}{2}\sum_{j=1}^{N}(\tilde{d}_j - \tilde{d}_j^{obs})^2, \tag{7.1}$$

where $d_j$ is the predicted data for $j\epsilon[1,2,..N]$ and the *obs* superscript denotes the observed data. Different norms can be employed, and different objective functions can be used such as a correlation objective function. For simplicity of exposition, we do not include regularization or preconditioning terms and assume that $d_j$ is always real.

An iterative gradient descent method can be used to find the model parameters that

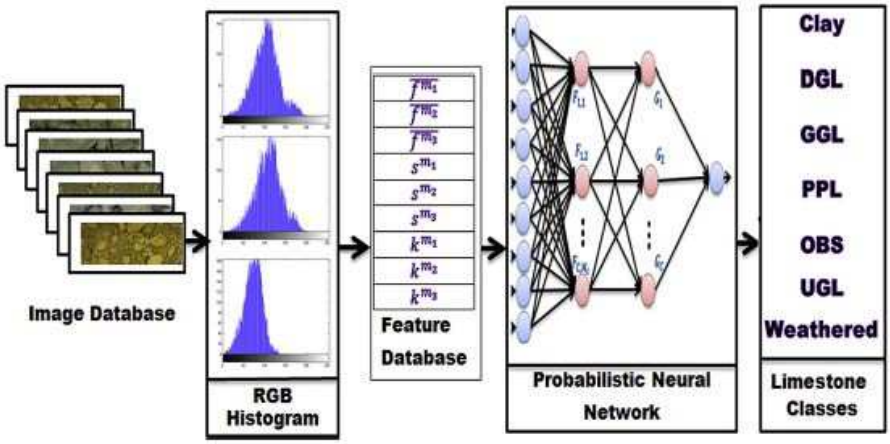


Figure 7.2: System diagram for the probabilistic neural network for classifying thin sections (from Patel and Chatterjee, 2016).

best explain the data in the least squares sense:

$$m_i^{(k+1)} = m_i^{(k)} - \alpha \sum_{j=1}^{N} \overbrace{(\tilde{d}_j - \tilde{d}_j^{obs})}^{r_j = residual} \overbrace{\frac{\partial \tilde{d}_j}{\partial m_k}}^{Fre'chet}, \tag{7.2}$$

where $\alpha$ is the step length and $m_i^{(k)}$ is the model parameter of interest in the $i^{th}$ cell at the $k^{th}$ iteration. If the skeletal data value $d_i$ is not explicitly contained in the wave equation, e.g. reflection traveltimes are not explicit variables in the wave equation, then a connective function $\Phi(d_j, m_k)$ is required to get a formula for the Fréchet derivative $\frac{\partial \tilde{d}_j}{\partial m_k}$. For example, Luo and Schuster (1991) proposed the crosscorrelation between the observed and predicted pressure traces for wave equation traveltime inversion. Lu et al. (2017) presents examples where the correlation is between the observed and predicted magnitude spectra of dispersion curves from surface waves. There are many other examples, some of which are described in Lu et al. (2017). Once the connective function is properly defined then the implicit function theorem (Luo and Schuster, 1991) can be used to define the formula for the Fréchet derivative:

$$\frac{\partial \tilde{d}_j}{\partial m_k} = A \frac{\partial P_j}{\partial m_k}, \tag{7.3}$$

where $A$ is a scalar that acts as a normalization term and $P_j$ is defined a fundamental field variable that explicitly appears in the wave equation. Fortunately, the formula for the Fréchet derivative $\frac{\partial P_j}{\partial m_k}$ of a fundamental field variable is straightforward to derive by the adjoint-state method. For acoustic data, the field variable is the pressure field recorded at the surface and the formula for its Fréchet derivative is well known. If more than one type of unknown is inverted for the multidimensional implicit function theorem can be used to compute the Fréchet derivatives for each type of model parameter (Lu et al., 2017). Note,

if $d_i \rightarrow P_i$ and $m_i$ denotes the velocity value in the $i^{th}$ cell then the update formula in equation 7.1 is that for FWI.

### 7.2.1 Feature Extraction

The most important step in SWI is the identification of skeletonized features that are simple and also greatly influenced by changes in the model parameter of interest. Towards this goal we look to the past where theoreticians identified skeletal data and derived asymptotic formulas that connected the skeletal data with the model parameter of interest. An obvious example is that of asymptotic ray-based tomography where, under the high-frequency approximation, traveltimes can be quickly generated by tracing rays through a sufficiently smooth model (Aki and Richards, 2002). If the physics of the problem is well understood, then the identification of the skeletonized data can be straightforward. If the physics between skeletal data and the model parameters is not well understood, then the sensitivity of the skeletal Frećhet derivative can be estimated by a combination of intuition and numerical sensitivity tests.

Can we go one step further by asking a Machine Learning algorithm to identify an important skeletal data set. Once identified, these skeletonized data can be inverted by a SWI method. For example, Figure 7.3 suggests a hybrid Machine Learning and SWI algorithm where an unsupervised machine learning method can be used to extract simplified but important features in the data set. Then the SWI method and the implicit function theorem can be used to discover the skeletal Frećhet derivative to be used in the update formula in equation 7.2. One possibility is that the skeletonized data can be obtained by principle component analysis (PCA) and the connective function is the spatial crosscorrelation between the observed and predicted PCA images. These PCA images can be localized in space. Can a general machine learning algorithm be used to identify the connection between skeletal features which are most sensitive to the model parameters of interest? This might be possible using a suitable training set obtained with numerically simulated data in realistic models.

We can go beyond skeletal data and define skeletal models as well. But this is another story.

## 7.3 Conclusions

We compared FWI, SWI and a supervised Machine Learning algorithm with one another. The advantage of SWI over FWI is it is more robust but provides less resolution. Thus, the SWI model can be used as the starting model for FWI. SWI is similar to supervised Machine Learning in that both use skeletonized representations of the original data. The next step is to combine the elements of SWI with Machine Learning to produce a Newtonian Machine Learning algorithm that employs Machine Learning, Newton's laws and solutions to the wave equation to invert for the model parameters from the skeletonized features of a complicated data set.

1. Derive the formula for the three-link chain rule.

2. Derive the formula for weight gradient at the $N = 12$ layer.

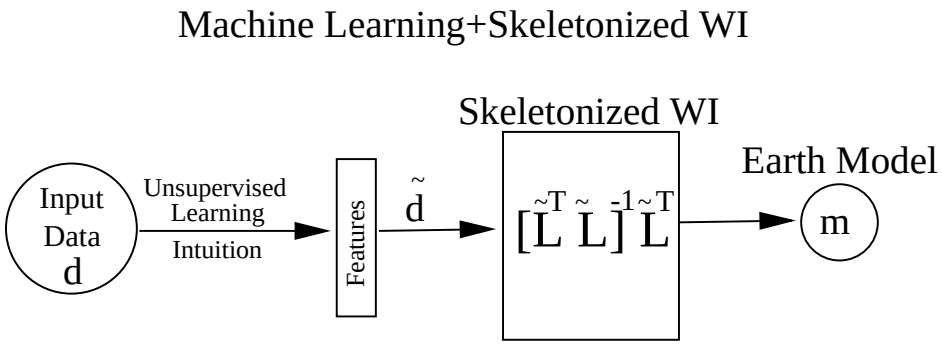


Figure 7.3: System diagram for a hybrid Machine Learning and SWI algorithm. Here, the skeletonized data $\tilde{d}$ can be obtained by, for example, a principle component analysis (PCA), where the connective function is the crosscorrelation between the observed and predicted PCA images.Machine Learning combined with WI gives rise to a Newtonian Machine Learning algorithm, where the optimal skeletonized features are learned from the data and then inverted by numerical solutions to the wave equation.

3. Write the pseudo-MATLAB code where the bias vector and its gradient are explicitly computed.

4. Derive the steepest descent formula where the objective function is the one where the gradient is the cross-entropy term.

5. Write the pseudo-MATLAB code where regularization is used.

6. Write the pseudo-MATLAB code where the input is a batch matrix that represents the training set.

7. Write the pseudo-MATLAB code where the number of nodes per layer is different from one another.

# Chapter 8

# Support Vector Machines

A support learning machine (SVM) is a supervised learning method that classifies *binary* data into two classes (Cortes and Vapnik, 1995). It has the same purpose as logistic regression in classifying data, but SVM is sometimes preferred because it can sometime provide faster convergence and more robustness in the presence of classification errors. It can also be generalized to classifying data that has more than two classes (Bishop, 2007) and its popularity has been growing over the last two decades (see Figure 8.1).

## 8.1 Introduction

In the supervised learning problem of classification we are given a set of training data consisting of $N$ feature vectors

$$(\mathbf{x}^{(i)}, y^{(i)}), \tag{8.1}$$

where $\mathbf{x}^{(i)}$ is the $i^{th}$ feature vector with dimension $D \times 1$ and $y^{(i)} = \pm 1$ is the binary target data value for classification. For a set of two-dimensional feature vectors $\mathbf{x}^{(i)}$, the goal of SVM is to find the *best* line that separates the two classes of data, denoted as +'s and -'s, in Figure 8.2. Here, the *best* line is the dashed black one with the fattest margin width $2d$ and is equidistant between the solid black lines that are parallel to one another. This dashed line, also known as the *decision line* or *decision boundary*, is mathematically characterized by the normal vector $\mathbf{w}$ and intercept value $b$, where any vector $\mathbf{x}$ on the decision line satisfies $w_1 x_1 + w_2 x_2 + b = 0$. The black solid lines intersect the circled margin points $\mathbf{x}^{(i)}$ known as *support vectors* for the specified index values $i$. These support vectors define the points with the closest perpendicular distance to the decision boundary.

A less than optimal *decision line* is represented by the red dashed line in Figure 8.2 that also separates the two classes of data. Here, the margin thickness is skinnier than the one associated with the black lines. The skinnier the margin thickness, the easier it is to misclassify data subject to errors in the training pair $(\mathbf{x}^{(i)}, y^{(i)})$.

Once the $\mathbf{w}$ is found after sufficient training, the SVM classifies new points $\tilde{\mathbf{x}}^{(i)}$ that are out of the training set. Figure 8.2 depicts points in a two-dimensional space, but points in a $D$-dimensional space can be separated by a $D-1$ dimensional hyperplane. In this case the point $\mathbf{x}$ on the hyperplane satisfies $\mathbf{w} \cdot \mathbf{x} + b = w_1 x_1 + w_2 x_2 + \ldots + w_d x_D + b = 0$.

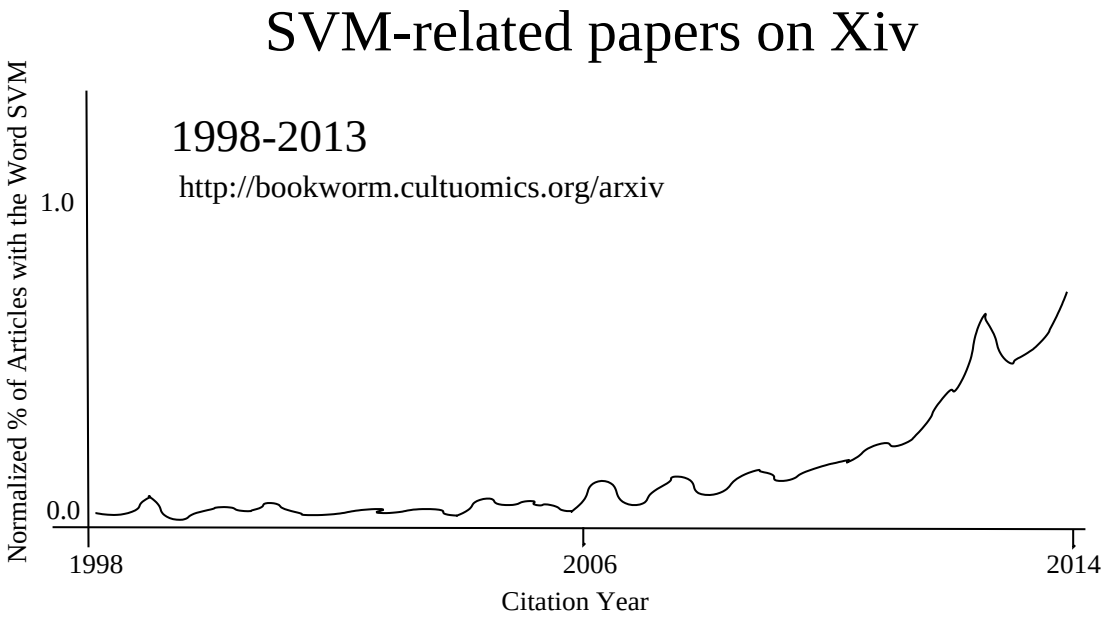


Figure 8.1: Citations in books for the word *support vector machines* plotted against calendar year.

The advantage of SVM over classification by logistic regression is that the optimal SVM line has the thickest margin, which is not necessarily the case with logistic regression when the points are linearly separable. The SVM procedure is very popular as a classification method, as evidenced by its citations in Figure 8.1 and some of the examples shown in this book. It is still one of the most effective classification methods, and can be adapted for non-linear classification.

### 8.1.1 SVM Applications

Many fields of science and engineering use SVM for classification, which include the following.

1. SVMs are helpful in text and hypertext categorization.

2. Classification of images can also be performed using SVMs. According to Wikipedia ($https://en.wikipedia.org/wiki/Support\_vector\_machine$) "experimental results show that SVMs achieve significantly higher search accuracy than traditional query refinement schemes after just three to four rounds of relevance feedback. This is also true of image segmentation systems, including those using a modified version SVM that uses the privileged approach as suggested by Vapnik (DeCoste, 2002).".

3. Hand-written characters can be recognized using SVM (DeCoste, 2002)..

4. According to Wikipedia ($https://en.wikipedia.org/wiki/Support\_vector\_machine$) "The SVM algorithm has been widely applied in the biological and other sciences. They have been used to classify proteins with up to 90% of the compounds classified

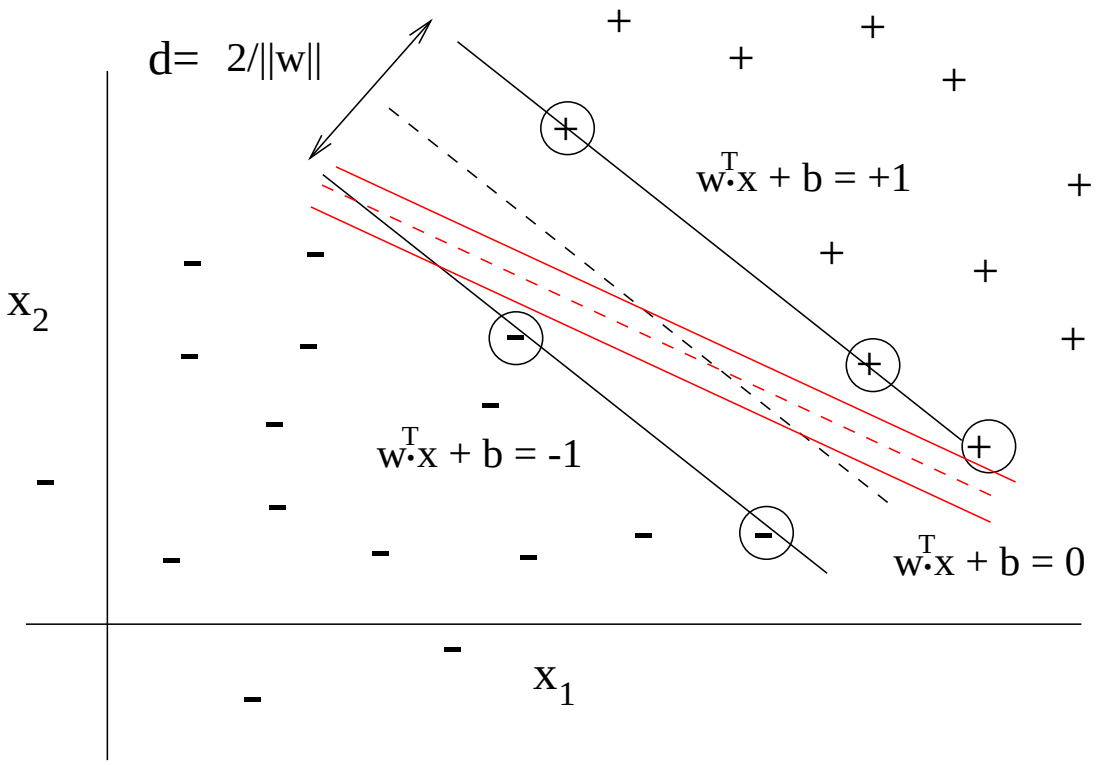


Figure 8.2: The points $\mathbf{x}^{(i)}$ denoted by the class symbols $\pm$ are separable by a decision line. The decision line with the thickest margin is the optimal one estimated by SVM and the circled margin points $\mathbf{x}^{(i)}$ are known as *support vectors*.

correctly. Permutation tests based on SVM weights have been suggested as a mechanism for interpretation of SVM models (Gaokar and Davatzikos, 2013). Support vector machine weights have also been used to interpret SVM models in the past. Support vector machine models can be used to identify features in the biological sciences.".

## 8.2 Linear SVM Theory

Define the $N$ feature vectors $(\mathbf{x}^{(n)}, y^{(n)})$ for $n \in [1, 2 \ldots N]$, with the binary classification of $y^{(n)} = \pm 1$. We assume that the set of points $(\mathbf{x}^{(n)}, y^{(n)})$ are linearly separable so that there exists lines that separate the positive and negative classes from one another in Figure 8.2. SVM seeks the decision line with the fattest margin thickness. This margin thickness can be defined by first defining the perpendicular distance $\tilde{d}$ between $\mathbf{x}$ on the decision plane in Figure 8.3 and any other point $\mathbf{x}^{(n)}$ as

$$\begin{aligned} \tilde{d} &= \frac{1}{||\mathbf{w}||}|\mathbf{w}^T \cdot (\mathbf{x}^{(n)} - \mathbf{x})|, \\ &= \frac{1}{||\mathbf{w}||}|\mathbf{w}^T \cdot \mathbf{x}^{(n)} + b - \overbrace{(\mathbf{w}^T \cdot \mathbf{x} + b)}^{= 0 \text{ for } \mathbf{x} \text{ on margin line}}|, \\ &= \frac{1}{||\mathbf{w}||}|\mathbf{w}^T \cdot \mathbf{x}^{(n)} + b|, \end{aligned} \tag{8.2}$$

where $\hat{\mathbf{w}} = \mathbf{w}/||\mathbf{w}||$ is the unit vector in Figure 8.3. Note, if $\mathbf{x}^{(n)}$ is on the decision line then $\mathbf{w}^T \cdot \mathbf{x}^{(n)} + b = 0$. If $\mathbf{x}^{(n)}$ is on the same side of the decision plane that $\mathbf{w}$ is pointing towards then $\mathbf{w}^T \cdot \mathbf{x}^{(n)} + b > 0$. If $\mathbf{x}^{(n)}$ is on the other side then $\mathbf{w}^T \cdot \mathbf{x}^{(n)} + b < 0$.

The annoying absolute value operator in equation 8.2 can be eliminated by replacing it with the product of the target value $y^{(n)}$ and $\mathbf{w}^T \cdot \mathbf{x}^{(n)} + b$:

$$\tilde{d} = \frac{1}{||\mathbf{w}||} y^{(n)}(\mathbf{w}^T \cdot \mathbf{x}^{(n)} + b), \tag{8.3}$$

which is always positive if $\mathbf{x}^{(n)}$ is correctly classified as the value of $y^{(n)}$. For example, if $\mathbf{w}$ in Figure 8.2 points to the right then the product $y^{(n)}(\mathbf{w}^T \cdot \mathbf{x}^{(n)} + b) > 0$ is positive for $\mathbf{x}^{(n)}$ to the right of the decision line. This is because $y^{(n)} = +1$ and $(\mathbf{w}^T \cdot \mathbf{x}^{(n)} + b) > 0$ are also positive. For points $\mathbf{x}^{(n)}$ to the left of the decision line $y^{(n)} = -1$, then the value $y^{(n)}(\mathbf{w}^T \cdot \mathbf{x}^{(n)} + b) > 0$ is positive because both $(\mathbf{w}^T \cdot \mathbf{x}^{(n)} + b) < 0$ and $y^{(n)} = -1$ are negative.

For a given $\mathbf{w}$ and $b$ that cleanly separates the data, the margin thickness is given by the perpendicular distance between the decision boundary and the closest point in the data set that satisfies the following conditions:

$$\frac{1}{||\mathbf{w}||} \min_{\mathbf{x}^{(n)}}[y^{(n)}(\mathbf{w}^T \cdot \mathbf{x}^{(n)} + b)]$$

$$\text{subject to } \; y^{(n)}(\mathbf{w}^T \cdot \mathbf{x}^{(n)} + b) > 0 \quad \forall \, \mathbf{x}^{(n)}, \tag{8.4}$$

where the inequality constraint can be described as the *clean-separation* condition.

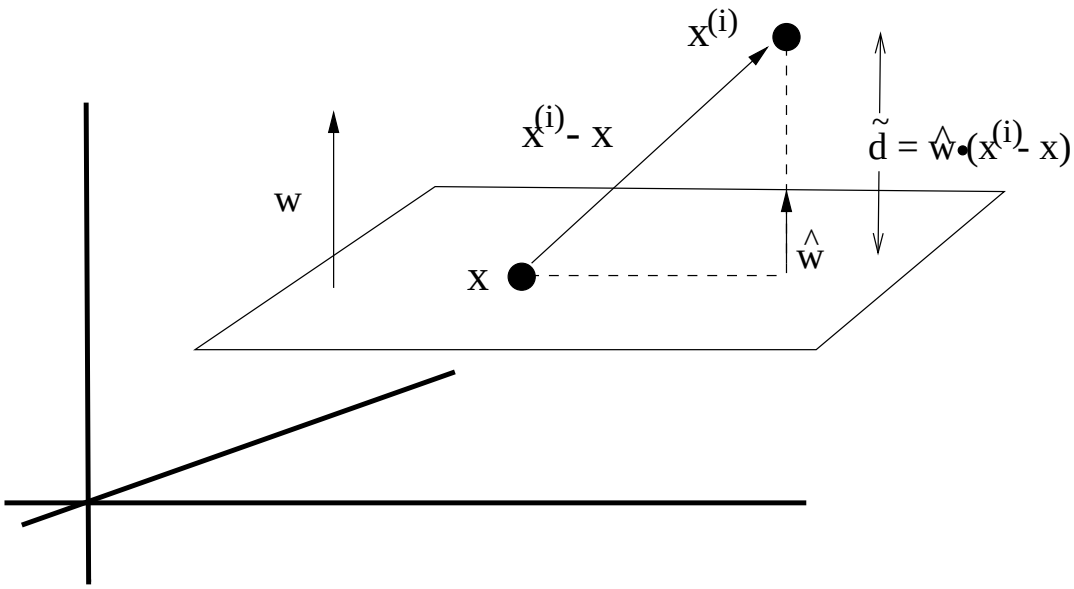


Figure 8.3: The perpendicular distance between $\mathbf{x}$ on the plane defined by $\hat{\mathbf{w}}^T \cdot \mathbf{x} + b = 0$ and $\mathbf{x}^{(i)}$ is given by $\tilde{d} = |\hat{\mathbf{w}}^T \cdot (\mathbf{x}^{(i)} - \mathbf{x})|$, where $\hat{\mathbf{w}}$ is the unit vector perpendicular to the decision plane.

As illustrated in Figure 8.2, there are many different sets of $(\mathbf{w}, b)$ that satisfy the above conditions. However, there is only one set of $(\mathbf{w}, b)$ that satisfies the following conditions:

$$arg \max_{\mathbf{w},\ b} \{\ \frac{1}{||\mathbf{w}||} \min_{\mathbf{x}^{(n)}} [y^{(n)}(\mathbf{w}^T \cdot \mathbf{x}^{(n)} + b)]\ \}, \tag{8.5}$$

$$\text{subject to}\ \ y^{(n)}(\mathbf{w}^T \cdot \mathbf{x}^{(n)} + b) > 0 \quad \forall\ \mathbf{x}^{(n)}. \tag{8.6}$$

where the $1/||\mathbf{w}||$ term is outside the $\min_{\mathbf{x}(n)}$ operation because it does not depend on the point $\mathbf{x}^{(n)}$. This optimal set $(\mathbf{w}, b)$ and the support vectors on the margin boundary describe the SVM slab with the thickest width.

The ratio $\frac{1}{||\mathbf{w}||} \min_{\mathbf{x}^{(n)}} [y^{(n)}(\mathbf{w}^T \cdot \mathbf{x}^{(n)} + b)]\ \}$ in equation 8.5 is the same value if the elements of $(\mathbf{w}, b)$ are rescaled as $\alpha \times (\mathbf{w}, b)$, where $\alpha$ is a scalar. We choose the scalar $\alpha$ so that the transformation $(\mathbf{w}, b) := \alpha(\mathbf{w}, b)$ transforms the inequality constraint in equation 8.6 to be

$$y^{(n)}(\mathbf{w}^T \cdot \mathbf{x}^{(n)} + b) \geq 1 \quad \forall\ \mathbf{x}^{(n)}, \tag{8.7}$$

where

$$y^{(n)}(\mathbf{w}^T \cdot \mathbf{x}^{(n)} + b) \ \ = \ \ 1, \quad \forall\ \mathbf{x}^{(n)*}, \tag{8.8}$$

for $\mathbf{x}^{(n)*}$ defined as a support vector on the margin boundary. There will be at least one support vector at each of the two margin boundaries.

The vectors $\mathbf{x}^{(n)}$ in equation 8.5 that minimize the numerator $\min_{\mathbf{x}^{(n)}} [y^{(n)}(\mathbf{w}^T \cdot \mathbf{x}^{(n)} + b)$ are the support vectors $\mathbf{x}^{(n)*}$ on the margin boundaries. Therefore, equation 8.8 says that

the numerator in equation 8.5 can be replaced by the value 1 so that the optimization problem defined by equations 8.7-8.8 can be redefined as finding $(\mathbf{w}, b)$ such that

$$arg \max_{\mathbf{w},\, b} \{ \frac{1}{||\mathbf{w}||} \},$$

$$\text{subject to } \; y^{(n)}(\mathbf{w}^T \cdot \mathbf{x}^{(n)} + b) \geq 1 \quad \forall \; \mathbf{x}^{(n)}. \tag{8.9}$$

It is algorithmically convenient to transform this maximization problem into a minimization problem by replacing $1/||\mathbf{w}||$ with $1/2||\mathbf{w}||^2$:

$$arg \min_{\mathbf{w},b} \quad d = \frac{1}{2}||\mathbf{w}||^2,$$

$$subject\ to\ y^{(n)}(\mathbf{w}^T \cdot \mathbf{x}^{(n)} + b) \geq 1 \quad \forall \; \mathbf{x}^{(n)}. \tag{8.10}$$

The inequality demands that $(\mathbf{w}, b)$ cleanly separates the training set $(\mathbf{x}^{(n)}, b)$ and the minimization equation is satisfied by the decision boundary with the shortest inverse thickness $1/d = ||\mathbf{w}||$ of the margin. Shortest inverse thickness is, of course, equivalent to maximum thickness of the margin. A numerical solution by, e.g. quadratic programming (Nocedal and Wright, 1999; Press et al., 2007), will give the optimal values of $(\mathbf{w}, b)$ that have the fattest margin in Figure 8.2. We can treat quadratic programming as a black box, where the algorithmic details are given in Nocedal and Wright (1999).

## 8.3 Nonlinear SVM

A set of training data might not be linearly separable, such as the data points shown in Figure 8.4a. In this case, a line cannot separate the different classes of data but higher-order polynomials might describe the separable boundary. To determine this non-linear separation boundary we use a non-linear transformation $z_i = \phi(\mathbf{x})_i$ of the coordinates in $\mathbf{x}$ to add extra dimensions to the solution space, just as we did for the motorcycle model in equation 2.34.

As an example, the two classes of data in Figure 8.4a can be separated by the dashed red circle. This decision boundary can be described by the transformation from 2D feature coordinates $(x_1, x_2)$ to the 3D space of $(z_1, z_2, z_3)$ defined by

$$(x_1, x_2) \quad \rightarrow \quad (z_1, z_2, z_3) = (\overbrace{x_1}^{\phi(\mathbf{x})_1}, \overbrace{x_2}^{\phi(\mathbf{x})_2}, \overbrace{\sqrt{x_1^2 + x_2^2}}^{\phi(\mathbf{x})_3}). \tag{8.11}$$

A hyperplane in the transformed space is described by the $3 \times 1$ normal vector $\tilde{\mathbf{w}} = (\tilde{w}_1, \tilde{w}_2, \tilde{w}_3)^T$ and the bias term $\tilde{b}$. Figure 8.4b depicts the data plotted in this transformed coordinate system, which are now separated by the boundary defined by $z_3 = 3$. Once the coordinates $(z_1^b, z_2^b, z_3^b)$ of the boundary are determined from $(\tilde{\mathbf{w}}, \tilde{b})$, then the associated decision boundary in the original space $(x_1, x_2)$ can be found by the inverse mapping from $\mathbf{z}$ to $\mathbf{x}$. For convenience we did not include the affine coordinate in this transformation, but it can easily be included.

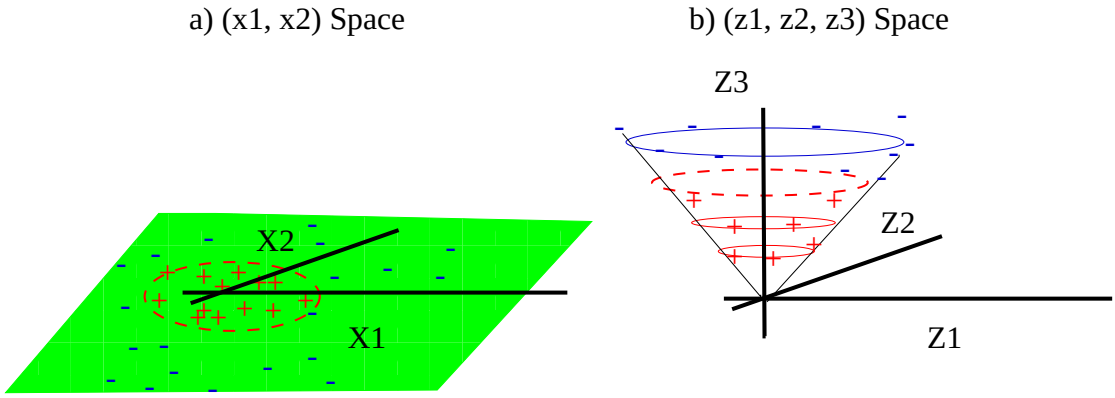


Figure 8.4: Data with a binary classification are plotted in a) the original 2D and b) transformed 3D spaces using equation 8.11. The blue symbols above the plane $z_3 = 3$ in b) are cleanly separated from the red symbols below it.

Therefore, a non-linear SVM defines a non-linear transformation such that the new feature coordinates are $z_i = \phi(\mathbf{x})_i$ and we seek the solution $(\tilde{\mathbf{w}}, \tilde{b})$ to the following optimization problem with inequality constraints.

**Box 8.3.1. Primal Optimization Problem with Inequality Constraints**

$$(\tilde{w}^*, \tilde{b}^*) = arg\min_{\tilde{\mathbf{w}}, \tilde{b}} \quad d = \frac{1}{2}||\tilde{\mathbf{w}}||^2,$$

$$subject\ to\ y^{(n)}(\tilde{\mathbf{w}}^T \cdot \mathbf{z}^{(n)} + \tilde{b}) \geq 1. \tag{8.12}$$

where the linear inequality constraint is with respect to the z-coordinates and the tilde indicates the parameters associated with the hyperplane in the z-space. The optimal values of $(\tilde{\mathbf{w}}^*, \tilde{b}^*)$ define the hyperplane with the fattest margin that cleanly separates the transformed training data in z-space.

As in the original problem, a quadratic programming method can be used to find the solution.

For the example described by equation 8.11, the inverse mapping is straightforward because $x_1 = z_1^b$ and $x_2 = z_2^b$ so that the boundary in the $\mathbf{x}$ coordinates is described by the equation $\sqrt{x_1^2 + x_2^2} = 3$ for a circle. In general, the inverse map is more complicated and an iterative gradient-search method can be used to find the mapping $\mathbf{z}^b \rightarrow \mathbf{x}^b$. For example, a steepest descent method can be used to find $\mathbf{x}^b$ from $\mathbf{z}^b = (z_1^b, z_2^b, z_3^b \ldots z_I^b)$ where the objective function $\epsilon$ is

$$\epsilon = 1/2 \sum_{b=1}^{B} \sum_{i=1}^{I} ||z_i^b - \phi(\mathbf{x}^b)_i||^2. \tag{8.13}$$

The steepest descent algorithm is then used to find the coordinates of $\mathbf{x}^b$ along the boundary defined by the $B$ boundary points $\mathbf{z}^b$. Here there are $B$ boundary points and $I$ is the dimension of the coordinates in the z-space.

The penalty in going to a higher dimension is that it can increase the computational cost by a significant amount. For example, if the feature vector is a $D \times 1 = 10^5 \times 1$ vector and the non-linear transform increases the dimension by a factor of four, then the number of unknowns is also increased by a factor of four. For quadratic programming the solution cost is $O(4^3 D^3)$ algebraic operations for increasing $D$ by a factor of four. For $D$ very large, this extra cost can be reduced by solving the dual Lagrangian problem as discussed in the next section.

## 8.4 Primal and Dual Solutions

The constrained optimization problem with inequality constraints in equation 8.10 or equation 8.12 is defined as the primal problem. Its numerical solution requires about $O(D^3)$ algebraic operations, where $D$ is the dimension of the $D \times 1$ vector $\mathbf{w}$. This can be expensive for $D >>> 1$ so the alternative is to solve the dual problem.

**Box 8.4.1. Dual Optimization Problem with Inequality Constraints**

The dual problem is defined as finding $\mathbf{w}$, $b$ and $\boldsymbol{\alpha}$ that maximize the Lagrangian $L(\mathbf{w}, b, \boldsymbol{\alpha})$:

$$L(\mathbf{w}, b, \boldsymbol{\alpha}) = \frac{1}{2}\mathbf{w}^T \cdot \mathbf{w} - \sum_{n=1}^{N} \alpha_n (\overbrace{y^{(n)}(\mathbf{w}^T \cdot \mathbf{x}^{(n)} + b)}^{d_n} - 1),$$

$$\text{subject to } \alpha_n \geq 0 \text{ for } n = 1, 2 \ldots N, \tag{8.14}$$

for $\alpha_n > 0$ and $L$ is the Lagrangian function discussed in Appendix 8.12. Here, $N$ is the number of inequality constraints, $\boldsymbol{\alpha}_n$ is the Lagrange multiplier for the $n^{th}$ inequality equation and $\boldsymbol{\alpha} = (\alpha_1, \alpha_2 \ldots \alpha_N)^T$ is the $N \times 1$ Lagrange multiplier vector. We will show that this approach only requires finding $N$ unknowns, the number of inequality constraints in the $\mathbf{x}$ space; for SVM the number of inequality constraints is equal to the number of feature vectors in a mini-batch of data. This can result in a significant reduction in computational costs if $N$ is much less than the number of elements in the $D \times 1$ vector $\mathbf{w}$.

The starting point for computing the solution to equation 8.14 is to find $\mathbf{w}$ and $b$ in terms of the $\alpha_n$. The resulting Lagrangian is denoted as the reduced Lagrangian. Once this is done we can pass the problem to a quadratic programming code to solve for the $N$ values of $\alpha_i$.

We have replaced the primal problem with inequality constraints in equation **??** with a similar one in equation 8.14, so what have we gained? Nothing is gained until we eliminate the $\mathbf{w}$ and $b$ in terms of the variables $\alpha_n$. This can be done by setting the gradients of the

Lagrangian $L$ w.r.t. to $w_i$ and $b$ to zero:

$$\frac{\partial L}{\partial w_i} = w_i - \sum_{n=1}^{N} \alpha_n y^{(n)} x_i^{(n)} = 0 \rightarrow \qquad \mathbf{w} = \sum_{n=1}^{N} \alpha_n y^{(n)} \mathbf{x}^{(n)}, \tag{8.15}$$

and

$$\frac{\partial L}{\partial b} = \sum_{n=1}^{N} \alpha_n y^{(n)} = 0. \tag{8.16}$$

Inserting equation 8.16 into equation 8.14 gives the transformed Lagrangian:

$$\begin{aligned} L(\mathbf{w}, b, \boldsymbol{\alpha}) &= \frac{1}{2}\mathbf{w}^T \cdot \mathbf{w} - \sum_{n=1}^{N} \alpha_n (y^{(n)}(\mathbf{w}^T \cdot \mathbf{x}^{(n)} + b) - 1), \\ &= \frac{1}{2}\mathbf{w}^T \cdot \mathbf{w} - \sum_{n=1}^{N} \alpha_n y^{(n)} \mathbf{w}^T \cdot \mathbf{x}^{(n)} + \sum_{n=1}^{N} \alpha_n - \overbrace{b \sum_{n=1}^{N} \alpha_n y^{(n)}}^{\boldsymbol{\alpha}^T \mathbf{y} = 0 \text{ by eq. } 8.16}, \\ &= \sum_{n=1}^{N} \alpha_n + \frac{1}{2}\mathbf{w}^T \cdot \mathbf{w} - \sum_{n=1}^{N} \alpha_n y^{(n)} \mathbf{w}^T \cdot \mathbf{x}^{(n)}, \end{aligned} \tag{8.17}$$

where $b$ is now eliminated.

**Box 8.4.2. Reduced Dual Optimization Problem with Inequality Constraints**

The $\mathbf{w}$ in equation 8.17 can be eliminated by plugging $\mathbf{w} = \sum_{n=1}^{N} \alpha_n y^{(n)} \mathbf{x}^{(n)}$ from equation 8.15 into equation 8.17 to get the *reduced* Lagrangian:

$$\begin{aligned} L(\mathbf{w}, b, \boldsymbol{\alpha}) &= \sum_{n=1}^{N} \alpha_n - \frac{1}{2} \sum_{n=1}^{N} \sum_{m=1}^{N} y^{(n)} y^{(m)} \alpha_n \alpha_m \mathbf{x}^{(n)T} \cdot \mathbf{x}^{(m)}, \\ \alpha_n \geq 0; \qquad & \boldsymbol{\alpha}^T \mathbf{y} = 0, \end{aligned} \tag{8.18}$$

where the only unknown are the $N$ variables $\boldsymbol{\alpha}_n$ that maximize $L(\mathbf{w}, b, \boldsymbol{\alpha})$ subject to the inequality constraints. Here, $N$ is also the number of feature vectors in the training set which can be small if the original training data set is broken up into mini-batches. In this case, solving the reduced dual optimization problem is much more efficient if the dimension of the $D \times 1$ feature vector is huge compared to the number $N$ of feature vectors in a mini-batch.

Multiplying the reduced lagrangian by $-1$ transforms it to a minimization problem w.r.t.

$\alpha_i$:

$$\min_{\boldsymbol{\alpha}} \frac{1}{2}\boldsymbol{\alpha}^T \overbrace{\begin{bmatrix} y^{(1)}y^{(1)}\mathbf{x}^{(1)T}\cdot\mathbf{x}^{(1)} & y^{(1)}y^{(2)}\mathbf{x}^{(1)T}\cdot\mathbf{x}^{(2)} & \dots & y^{(1)}y^{(N)}\mathbf{x}^{(1)T}\cdot\mathbf{x}^{(N)} \\ y^{(1)}y^{(2)}\mathbf{x}^{(2)T}\cdot\mathbf{x}^{(1)} & y^{(2)}y^{(2)}\mathbf{x}^{(2)T}\cdot\mathbf{x}^{(2)} & \dots & y^{(2)}y^{(N)}\mathbf{x}^{(2)T}\cdot\mathbf{x}^{(N)} \\ \dots & \dots & \dots & \dots \\ y^{(N)}y^{(1)}\mathbf{x}^{(N)T}\cdot\mathbf{x}^{(1)} & y^{(N)}y^{(2)}\mathbf{x}^{(N)T}\cdot\mathbf{x}^{(2)} & \dots & y^{(N)}y^{(N)}\mathbf{x}^{(N)T}\cdot\mathbf{x}^{(N)} \end{bmatrix}}^{quadratic\ coefficients} \boldsymbol{\alpha} + (-\mathbf{1}^T)\boldsymbol{\alpha} \tag{8.19}$$

$$\text{subject to} \qquad \mathbf{y}^T\cdot\boldsymbol{\alpha} = 0 \ \text{ and } \alpha_n \geq 0 \ \forall n. \tag{8.20}$$

The $N \times N$ matrix of dot products is known as the Gram matrix, where the optimal solution is found by a quadratic programming method with a computational cost as high as $O(N^3)$ for inverting this matrix.

If the non-linear SVM method is used then we are seeking the $(\tilde{\mathbf{w}}, \tilde{b})$ that gives rise to the same type of Gram matrix seen in equation 8.19, except $\mathbf{x}^{(n)T}\cdot\mathbf{x}^{(m)}$ is replaced by $\mathbf{z}^{(m)T}\cdot\mathbf{z}^{(m)}$:

$$\min_{\boldsymbol{\alpha}} \frac{1}{2}\boldsymbol{\alpha}^T \overbrace{\begin{bmatrix} y^{(1)}y^{(1)}\mathbf{z}^{(1)T}\cdot\mathbf{z}^{(1)} & y^{(1)}y^{(2)}\mathbf{z}^{(1)T}\cdot\mathbf{z}^{(2)} & \dots & y^{(1)}y^{(N)}\mathbf{z}^{(1)T}\cdot\mathbf{z}^{(N)} \\ y^{(1)}y^{(2)}\mathbf{z}^{(2)T}\cdot\mathbf{z}^{(1)} & y^{(2)}y^{(2)}\mathbf{z}^{(2)T}\cdot\mathbf{z}^{(2)} & \dots & y^{(2)}y^{(N)}\mathbf{z}^{(2)T}\cdot\mathbf{z}^{(N)} \\ \dots & \dots & \dots & \dots \\ y^{(N)}y^{(1)}\mathbf{z}^{(N)T}\cdot\mathbf{z}^{(1)} & y^{(N)}y^{(2)}\mathbf{z}^{(N)T}\cdot\mathbf{z}^{(2)} & \dots & y^{(N)}y^{(N)}\mathbf{z}^{(N)T}\cdot\mathbf{z}^{(N)} \end{bmatrix}}^{quadratic\ coefficients} \boldsymbol{\alpha} + (-\mathbf{1}^T)\boldsymbol{\alpha} \tag{8.21}$$

$$\text{subject to} \qquad \mathbf{y}^T\cdot\boldsymbol{\alpha} = 0 \ \text{ and } \alpha_i \geq 0. \tag{8.22}$$

Here, we have to construct $\mathbf{z}^{(m)}$ and then use them to compute their dot products. However, this is impractical if $\mathbf{z}^{(m)}$ is of large or infinite dimension. The remedy is to specify the analytical form of the dot product as a kernel $k(\mathbf{x}, \mathbf{x}') = \mathbf{z}(\mathbf{x})^T\cdot\mathbf{z}(\mathbf{x}')$, as will be shown in the next section.

## 8.5 Kernel Methods

The problem with the non-linear SVM method is that the choice of a very high-order transformation $z_i = \phi(\mathbf{x})$ will increase the number of unknowns and therefore increase the computational cost. It also can lead to overfitting of the training data if a highly oscillatory polynomial is selected.

To overcome these two problems we can reformulate the non-linear SVM problem in the dual space by specifying an infinite dimensional kernel function $k(\mathbf{x}, \mathbf{x}') = \mathbf{z}(\mathbf{x})^T\cdot\mathbf{z}(\mathbf{x}')$. In

this case, equation 8.21 becomes the Gram matrix:

$$\min_{\boldsymbol{\alpha}} \frac{1}{2}\boldsymbol{\alpha}^T \begin{bmatrix} y^{(1)}y^{(1)}k(\mathbf{x}^{(1)},\mathbf{x}^{(1)}) & y^{(1)}y^{(2)}k(\mathbf{x}^{(1)},\mathbf{x}^{(2)}) & \dots & y^{(1)}y^{(N)}k(\mathbf{x}^{(1)},\mathbf{x}^{(N)}) \\ y^{(2)}y^{(1)}k(\mathbf{x}^{(2)},\mathbf{x}^{(1)}) & y^{(2)}y^{(2)}k(\mathbf{x}^{(2)},\mathbf{x}^{(2)}) & \dots & y^{(2)}y^{(N)}k(\mathbf{x}^{(2)},\mathbf{x}^{(N)}) \\ \dots & \dots & \dots & \dots \\ y^{(N)}y^{(1)}k(\mathbf{x}^{(N)},\mathbf{x}^{(1)}) & y^{(N)}y^{(2)}k(\mathbf{x}^{(N)},\mathbf{x}^{(2)}) & \dots & y^{(N)}y^{(N)}k(\mathbf{x}^{(N)},\mathbf{x}^{(N)}) \end{bmatrix} \boldsymbol{\alpha} + (-\mathbf{1}^T)\boldsymbol{\alpha}. \tag{8.23}$$

The stationary conditions in equation 8.17 become

$$\frac{\partial L}{\partial \tilde{w}_i} = \tilde{w}_i - \sum_{n=1}^{N_z} \alpha_n y^{(n)} z_i^{(n)} = 0 \rightarrow \qquad \tilde{\mathbf{w}} = \sum_{n=1}^{N_z} \alpha_n y^{(n)} \mathbf{z}^{(n)}, \tag{8.24}$$

and

$$\frac{\partial L}{\partial \tilde{b}} = \sum_{n=1}^{N_z} \alpha_n y^{(n)} = 0. \tag{8.25}$$

Here $\tilde{b}$ and $\tilde{\mathbf{w}}$ define, respectively, the bias and $D_z \times 1$ hyperplane normal vector in the z-space.

The main advantage of explicitly specifying only the kernel $k(\mathbf{x},\mathbf{x}')$ in equation 8.27 rather than $\mathbf{z}(\mathbf{x})^T \cdot \mathbf{z}(\mathbf{x}')$ in equation 8.21 is that an infinite-dimensional vector $\mathbf{z}$ and its dot product $\mathbf{z}(\mathbf{x})^T \cdot \mathbf{z}(\mathbf{x}')$ cannot be practically computed. However, if $k(\mathbf{x},\mathbf{x}')$ is analytically specified as a, e.g. Gaussian kernel $k(\mathbf{x},\mathbf{x}') = e^{-||\mathbf{x}-\mathbf{x}'||^2}$, then it can easily be computed for any specified pair of vectors $\mathbf{x}$ and $\mathbf{x}'$. Can $k(\mathbf{x},\mathbf{x}')$ be constructed as a dot product of $\mathbf{z}(\mathbf{x}')^T \cdot \mathbf{z}(\mathbf{x})$? As an example, take the scalar case where

$$\begin{aligned} k(x',x) &= e^{-(x'-x)^2}, \\ &= e^{-x^2}e^{-x'^2}e^{-2xx'}, \\ &= e^{-x^2}e^{-x'^2}\sum_k^\infty \frac{2x^k x'^k}{k!}, \\ &= e^{-x^2}e^{-x'^2}\overbrace{(1+2xx'+x^2x'^2+\dots)}^{Taylor\ series}, \\ &= (\overbrace{e^{-x^2}(1,\sqrt{2}x,x^2,\dots)}^{\mathbf{z}(x)},\overbrace{e^{-x'^2}(1,\sqrt{2}x',x'^2,\dots)}^{\mathbf{z}(x')})^T. \end{aligned} \tag{8.26}$$

Here, the infinite dimensional vector $\mathbf{z}(x)$ and its inner product with $\mathbf{z}(x')$ cannot be practically computed, but the computation of the kernel $k(\mathbf{x},\mathbf{x}') = e^{-||\mathbf{x}-\mathbf{x}'||^2}$ is trivial.

The kernel $k(\mathbf{x},\mathbf{x}')$ must satisfy two properties in order to be used as a substitute for $\mathbf{z}(\mathbf{x}') \cdot \mathbf{z}(\mathbf{x})$ in equation 8.21:

- $k(\mathbf{x},\mathbf{x}')$ must be symmetric such that $k(\mathbf{x},\mathbf{x}') = k(\mathbf{x}',\mathbf{x})$.

- The Gram matrix must be positive semi-definite. Symmetry for the Gaussian kernel is proven by inspection and the positive semi-definite property can be empirically tested by using an example matrix.

**Box 8.5.1. SVM Algorithm**

In summary, the non-linear SVM algorithm with an infinite-dimensional kernel is the following.

1. Specify the functional form of the kernel, e.g. $k(\mathbf{x}^{(n)}, \mathbf{x}^{(m)}) = e^{-\beta||\mathbf{x}^{(n)}-\mathbf{x}^{(m)}||^2}$, and compute the $N \times N$ Gram matrix in equation 8.24. Here, $\beta$ is a constant specified by the user where large values of $\beta$ indicate that the main influence of this basis function is only around points $\mathbf{x}^{(n)}$ and $\mathbf{x}^{(m)}$ that are close neighbors to one another. Use quadratic programming (QP) or the Lagrangian barrier method (Nocedal and Wright, 1999) method to solve for $\alpha_n \ \forall n$. If the number of feature vectors is huge, then the entire data set is broken up into small mini-batches and the non-linear optimization method is used to inexpensively invert the mini-batch of data to find $(\mathbf{w}, b)$; this is computationally inexpensive because the number of mini-batch feature vectors is small. This solution is then the starting model for inverting the next mini-batch of data, and the procedure is repeated until all the mini-batches are used.

2. Define the feature vector $\mathbf{x}$ so that its predicted classification value $y$ is given by $sign(\tilde{\mathbf{w}} \cdot \mathbf{z}(\mathbf{x}) + \tilde{b})$. Substituting $\tilde{\mathbf{w}}$ from equation 8.24 gives the predicted classification value:

$$\begin{aligned} y &= sign(\sum_{n=1}^{N} \alpha_n y^{(n)} \mathbf{z}(\mathbf{x})^T \cdot \mathbf{z}(\mathbf{x})^{(n)} + \tilde{b}), \\ &= sign(\sum_{n \in B_{support}} \alpha_n y^{(n)} k(\mathbf{x}, \mathbf{x}^{(n)}) + \tilde{b}), \end{aligned} \tag{8.27}$$

   where $B_{support}$ is the set of indices associated with the support vectors where $\alpha_i \neq 0$.

   The value of $\tilde{b}$ in equation 8.27 is obtained by noting that the support vector $\mathbf{z}(\mathbf{x}_i)$ for $i \in B_{support}$ is associated with the non-zero Lagrange multiplier $\alpha_n \neq 0$ that satisfies the constraint equation

$$\begin{aligned} y^{(i)} &= \tilde{\mathbf{w}}^T \cdot \mathbf{z}(\mathbf{x}^{(i)}) + \tilde{b}, \\ &= \sum_{n \in B_{support}} \alpha_n y^{(n)} \mathbf{z}(\mathbf{x}^{(n)})^T \cdot \mathbf{z}(\mathbf{x}^{(i)}) + \tilde{b} \quad \text{for } i \in B_{support}, \end{aligned} \tag{8.28}$$

   where equation 8.24 is used for $\tilde{\mathbf{w}}$. Substituting $k(\mathbf{x}^{(n)}, \mathbf{x}^{(i)}) = \mathbf{z}(\mathbf{x}^{(n)})^T \cdot \mathbf{z}(\mathbf{x}^{(i)})$ and equation 8.24 for $\mathbf{w}$ gives

$$\sum_{n \in B_{support}} \alpha_n y^{(n)} k(\mathbf{x}^{(i)}, \mathbf{x}^{(n)}) + \tilde{b} = y^{(i)}. \tag{8.29}$$

   This equation is now solved for in terms of $\tilde{b}$. To get a more robust estimate of $\tilde{b}$, compute the average of the different support vectors to get $\bar{b}$:

$$\bar{b} = \frac{1}{N_{support}} \sum_{i \in B_{suport}} [y^{(i)} - \sum_{n \in B_{support}} \alpha_n y^{(n)} k(\mathbf{x}^{(i)}, \mathbf{x}^{(n)})], \tag{8.30}$$

   where $N_{support}$ is the number of support vectors in the dual space where $\alpha_i \neq 0$.

3. All of the feature vectors in the validation set can now be classified with equations 8.27 and 8.30. An example is given in Figure 8.5.

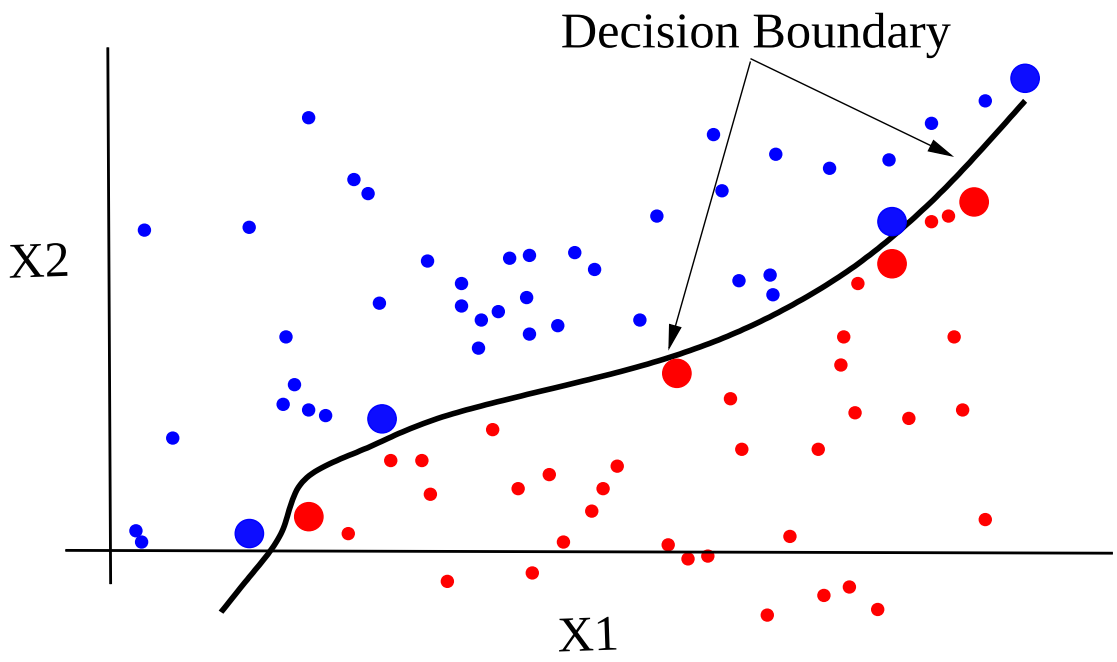


Figure 8.5: Feature points in $(x_1, x_2)$ space with the black-decision boundary computed by the Gaussian-kernel method in the dual space. The support points are the large filled circles and the input examples are classified as either blue or red points. Image redrawn from the Youtube video of Dr. AbuMostafa.

## 8.6 Numerical Examples

Recall that seismic migration is an imaging method that identifies the geometries and strengths of subsurface reflectors from seismic data; the final result is the migration image $m(\mathbf{x})$. For example, seismic data recorded on the top horizontal surface were migrated to give the large amplitude values of $m(\mathbf{x})$ in Figure 8.6a, which denote the locations of reflecting interfaces. Some of the large amplitude values are artifacts from the migration method and do not indicate an actual reflector. A simple description of seismic migration is in Appendix 8.14 and more details are in Yilmaz (2001). We will now use the SVM method to denoise migration images computed from synthetic seismic data.

The goal is to use SVM to distinguish noise from the yellow signal in the Figure 8.6a migration image. The elliptical features are typically due to aliasing artifacts in the migration image and can be identified as noise by a trained interpreter. The SVM can then be trained to distinguish noise from signal in the classified images, and then it can be used to identify noise in the rest of the data. Once identified, a suitable filtering method can be used to suppress the noisy artifacts in the migration image.

To train the SVM, the first step is to define a small $5 \times 5$ window centered at the $i^{th}$ pixel.

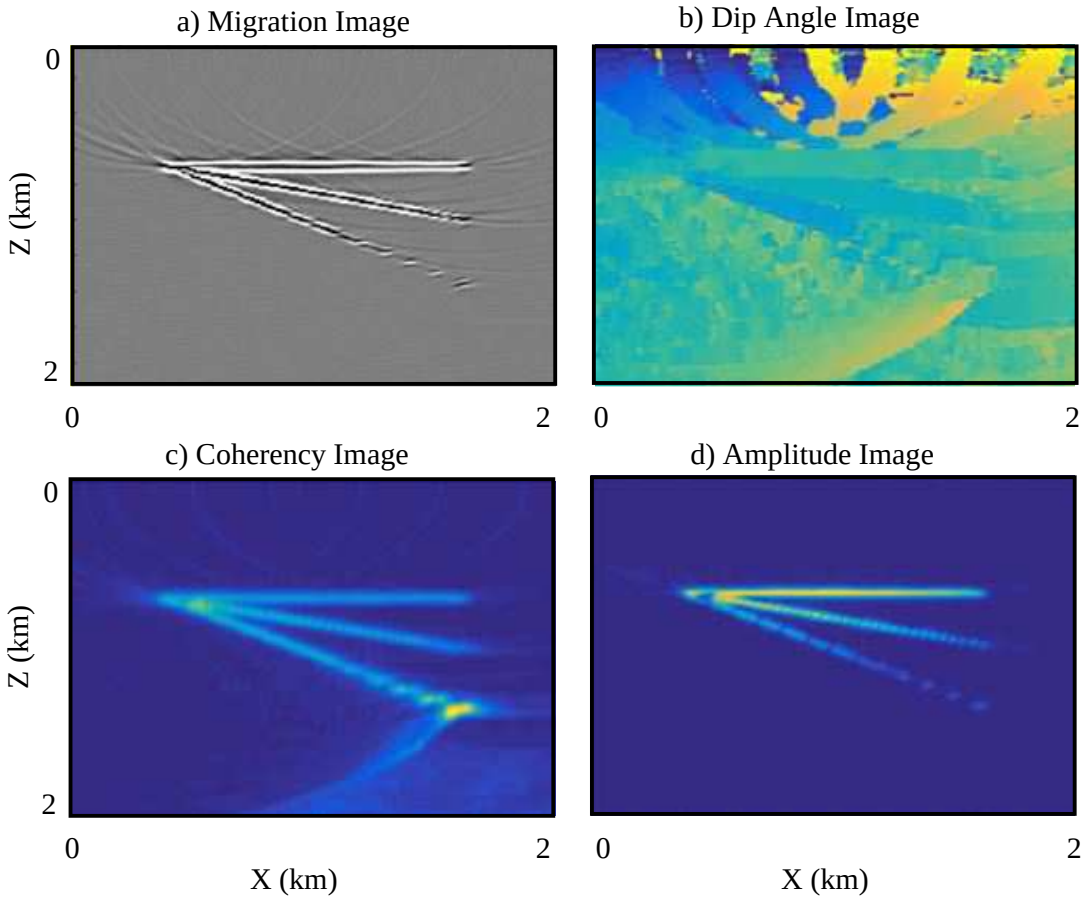


Figure 8.6: Images of the a) raw migration section, b) dominant dip angle at each pixel, c) migration coherency, and d) amplitudes. The yellow areas in the Figure 8.7c migration image a) correspond to the label of signal ($y = 1$) and the other areas are labeled noise ($y = -1$). Images courtesy of Yuqing Chen.

Three skeletonized characteristics are then computed for the image in this small window: dominant dip angle, coherency and maximum amplitude as described in Appendix 8.14. The user classifies each point in the training migration image as either noise or signal. The computer computes the three skeletal characteristics $(x_1^{(i)}, x_2^{(i)}, x_3^{(i)})$ at each point of $m(\mathbf{x})$ and assigns the classification value $y^{(i)}$. Therefore, the input feature vectors are the $4 \times 1$ vectors $(x_1^{(i)}, x_2^{(i)}, x_3^{(i)}, y^{(i)})$. For the migration examples, only 70 image points are selected for training from the $201 \times 201$ migration image $m(\mathbf{x})$, which means that only 0.17 % of the original migration image for training.

What should we use for the skeletal characteristics of $m(\mathbf{x})$ that can be used to distinguish noise from signal? The answer is to use intuition and experience to identify the most significant characteristics that distinguish noise from signal. For example, we recognize that migration artifacts should have weak coherency $\phi_i$, large dip angles $\theta_i$, and weak amplitudes $A_i$ at the $i^{th}$ pixel, while the reflection signal should mostly be characterized by strong coherency, modest dip angles, and strong amplitudes. As an example, Figure 8.6 plots the $\phi_i$ values at all pixels where weak coherency is indicated by the deep-blue color. Therefore we use $(\theta_i, \phi_i, A^{(i)}, y^{(i)})$ as the skeletonized characteristics of the migration image to distinguish noise from signal. Other characteristics might be more effective and should be used if needed for more accuracy.

The input training set is now defined as $(\mathbf{x}^{(i)}, y^{(i)}) = (\theta_i, \phi_i, A^{(i)}, y^{(i)})$ for 70 points in the migration image in Figure 8.6a. Each pixel in the training migration images is manually classified as either signal $(+1)$ or noise $(-1)$. The SVM procedure is then used to find the $3 \times 1$ vector $\mathbf{w} = (w_1, w_2, w_3)$ and the bias $b$ that cleanly separate the labeled noise and signal. This generates a hyperplane in feature space as shown in Figure 8.7a, where the yellow colored plane defines the decision hyperplane in 3D. Once the $\mathbf{w}$ and $b$ are found then the other parts of the migration image (i.e., out-of-the-training-set data) are automatically labeled by determining the sign of $\mathbf{w} \cdot \mathbf{x}^{(n)} + b$. If the amplitude of the automatically labeled image is noise then that amplitude can later be filtered by some method.

The automatically labeled features are shown in Figure 8.7c, where yellow indicates signal and deep blue indicates noise. In comparison, when regularization is included in the SVM method the surface is shown in Figure 8.7b, with the resulting labeled signal denoted by the yellow portions. As a comparison, logistic regression with orders $n = 1$ and $n = 2$ are used to classify the same training set used with SVM. See the diagram on the left of Figure 8.8 for the architecture of the logistic regression. In this case $n = 1$ means the input is a $3 \times 1$ feature vector $(x_1, x_2, x_3)$ while $n = 2$ means that the input is the $9 \times 1$ vector: $(x_1, x_2, x_3, x_1^2, x_2^2, x_3^2, x_1x2, x_1x_3, x_2x_3)$. The results are shown in Figure 8.9, where there is not much of a difference in any of the images.

These results are to be compared to those in Figure 8.10 where the features are classified by a fully connected neural network with two layers and 15 nodes in the hidden layer. The neural network appears to provide a similar classification of image points compared to logistic regression, even though it misclassified a data point.

Finally, a realistic migration image is used as the input data to get the three features of maximum dip angle, coherence, and amplitude for each pixel. The signals in the images are colored yellow in Figure 8.11 as classified by a) two-node logistic regression, b) a two-layer neural network, c) SVM with a Gaussian kernel without regularization, and d) SVM with a Gaussian kernel with regularization. The SVM image w/o regularization in Figure 8.11c

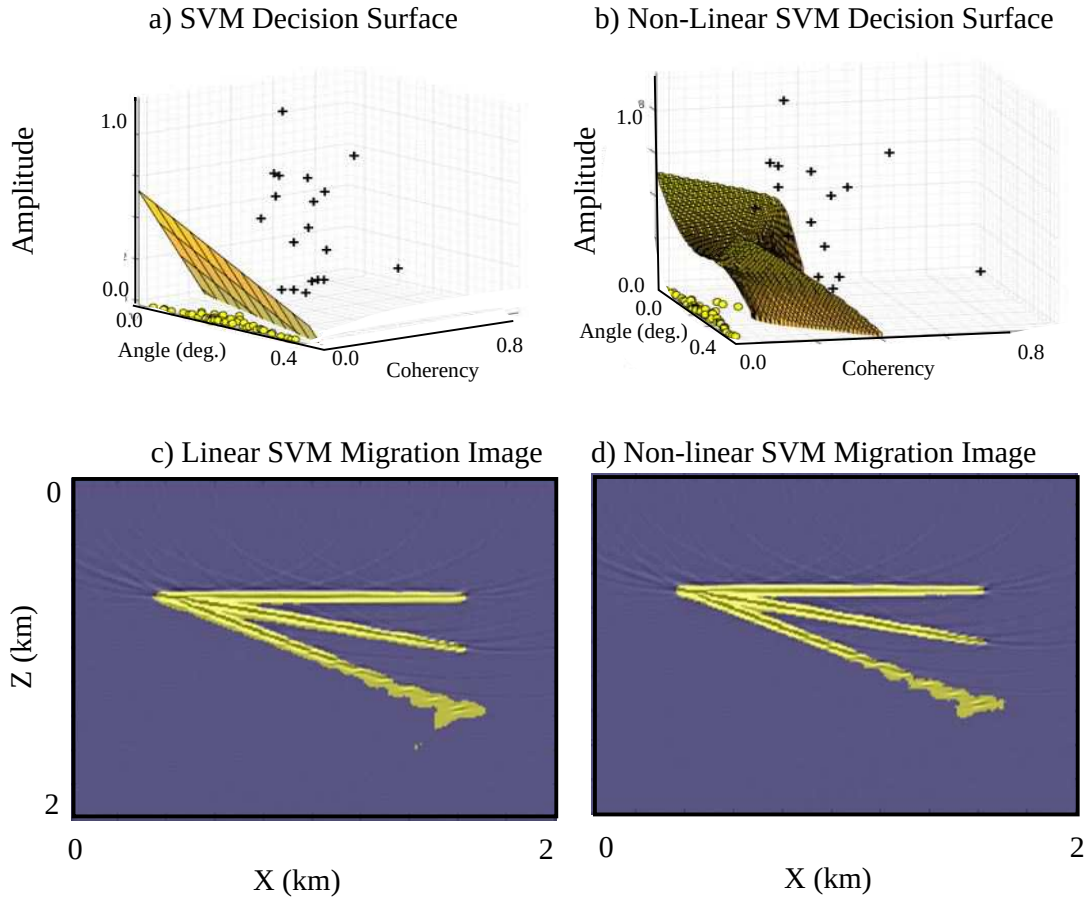


Figure 8.7: Decision boundaries computed by a) linear SVM and b) non-linear SVM with regularization. The corresponding migration images are directly below the feature plots. Here, the yellow colors denote signal and the black colors denote noise. The amplitudes classified as noise are suppressed by muting and the resulting migration images are shown in c) and d).

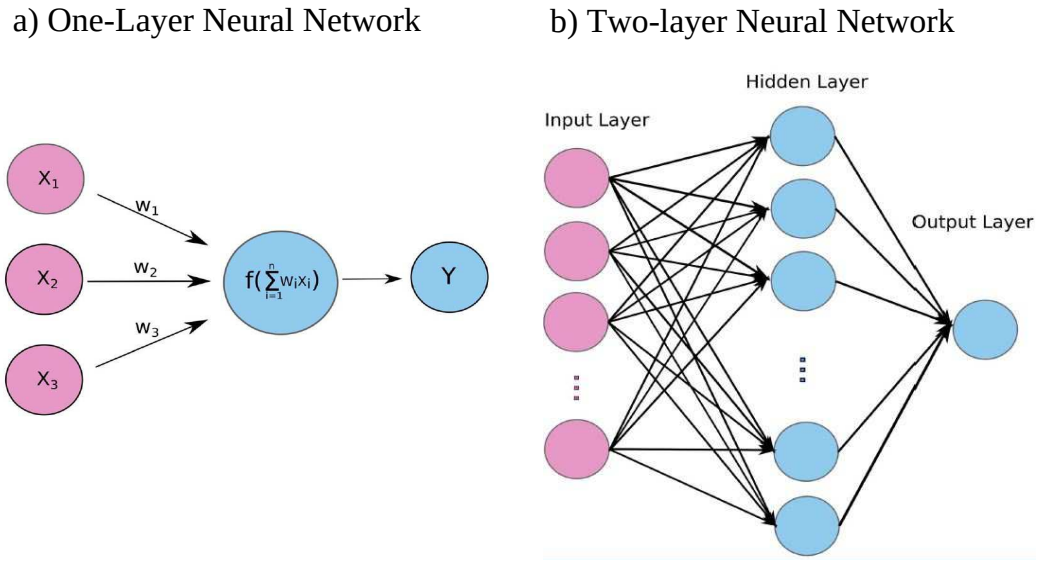


Figure 8.8: Architectural diagrams for a) one- and b) two-layer neural networks. The one-layer one-node network diagram is for the logistic network and the two-layer multi-node architecture is for the neural network used in the examples. Images courtesy of Yuqing Chen.

is judged to be the best result because it picks out most of the steeply dipping interfaces in the red box. This suggests that the regularization strategy should be improved to not exclude steep dips.

## 8.7 Multiclass SVM

Sometimes the data need to be divided into more than two types of data, in which case a multiclass SVM method can be used. In this case the *one-versus-the-rest* approach of Vapnik (1998) is often used. In this method if there are $K$ classes of data, one of the data types, denoted as the $k = 1$ class, is assigned one binary value by the SVM method and the rest of the data are assigned the other one. This procedure is repeated for the $k = 2$ class, and then again one at a time until all the classes are separated.

The problem with this approach is that the data are imbalanced, with fewer data in the specified class than the rest of the data. This can lead to an imbalanced gradient that prefers reducing the residuals for the rest of the class compared to that for the specified class. This can lead to both slow convergence and a large percentage error in the predicted classifications. Balancing the gradient, also known as preconditioning (Schuster, 2017) in full waveform inversion, can be a partial remedy where the weight of the specified class has $+1$ while the other data have a weight of $-1/(K-1)$. If the number of points in each class differs significantly from one another then each point in a class is divided by the number of points in that class.

Sometimes data points are assigned by the *one-versus-the-rest* SVM to more than one class, which is an error in classification. This contradiction is sometimes resolved by as-

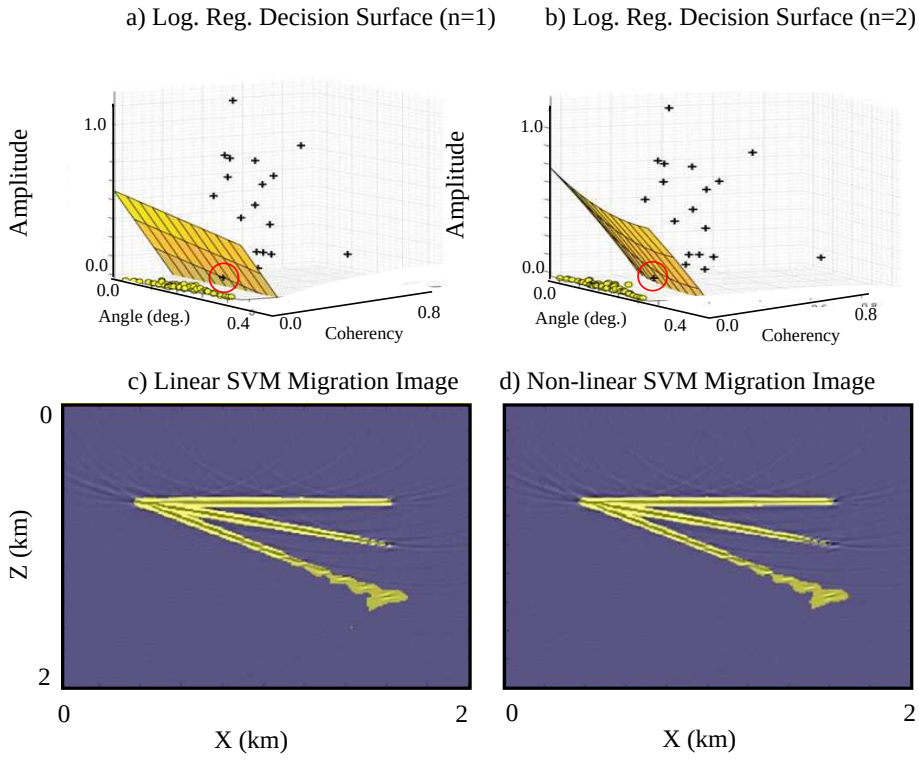


Figure 8.9: Same as Figure 8.7 except logistic regression with a)-c) order $n = 1$ and b)-d) $n = 2$ is used for classification of the features. Notice the misclassified point denoted by a red circle. Images courtesy of Yuqing Chen.

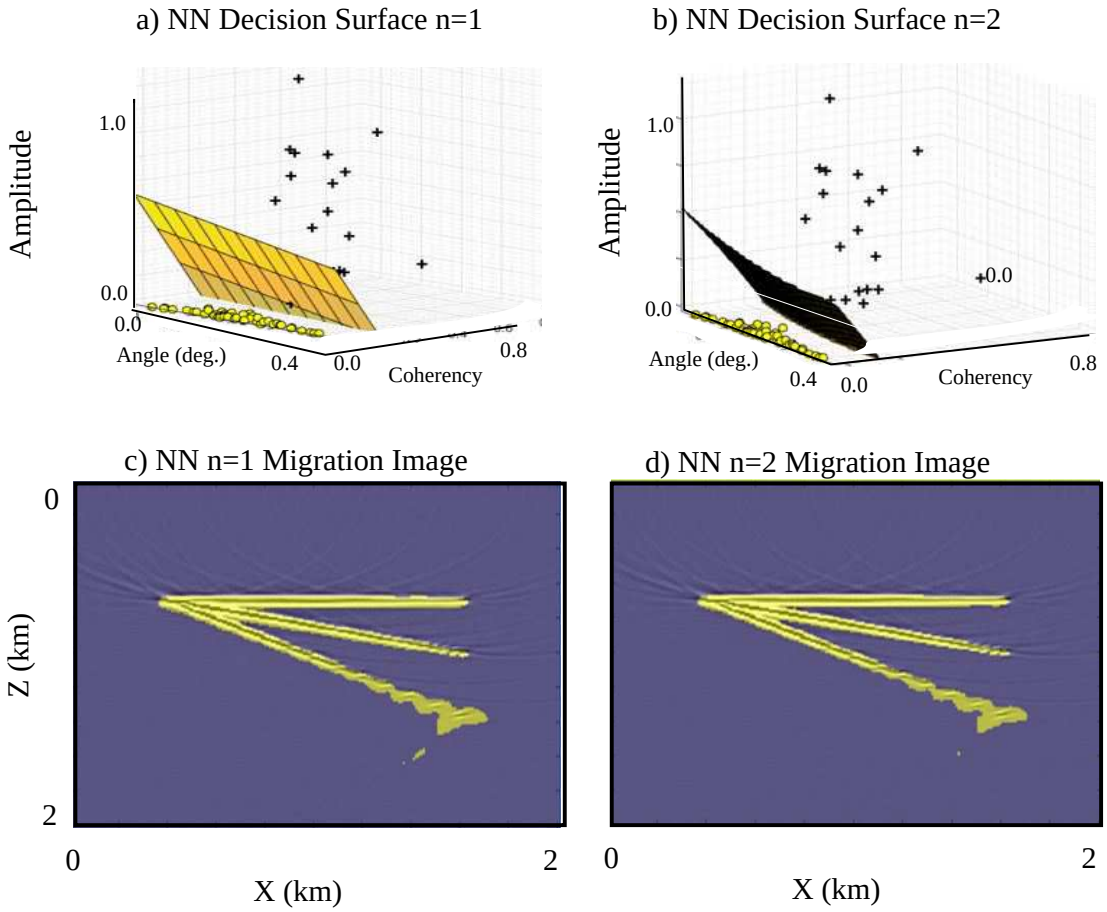


Figure 8.10: Same as Figure 8.7 except a neural network with a)-c) n=1 and b)-d) n=2 layers are used for classification. Images courtesy of Yuqing Chen.

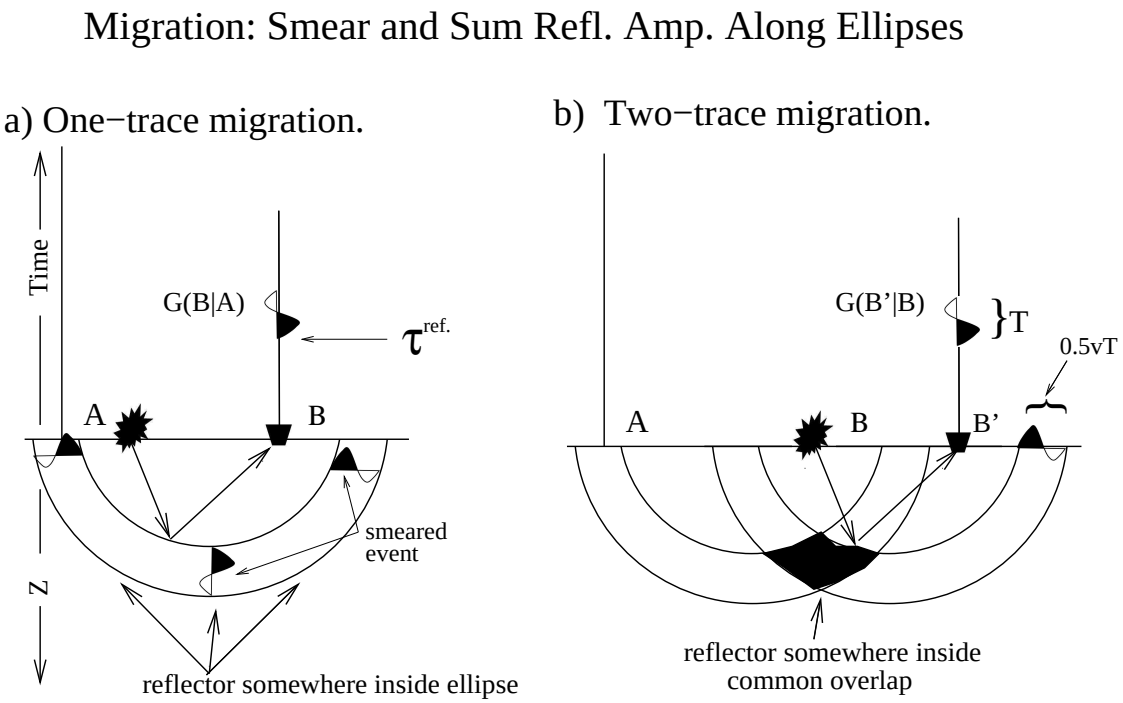


Figure 8.11: Migration images classified as signal (yellow) by a) logistic regression of order two, b) a neural network with 2 layers and 15 nodes in the hidden layer, c) SVM with a Gaussian kernel w/o regularization and d) SVM with a Gaussian kernel with regularization. Images courtesy of Yuqing Chen.

signing the misclassified point to the class with the largest value of the predicted target function $y$.

A number of other heuristic approaches to multiclass SVM are described in Bishop (2007), but he states that the *all-versus-one* approach is the one that is most widely used today. Research into better multiclass SVM methods is still an ongoing effort.

## 8.8 Soft-Margin SVM

The linear SVM assumes that the data are linearly separable with the consequence that the QP solution to the dual problem will give a *hard-margin* decision boundary. This is sometimes referred to as *hard-margin* SVM. However, data are often polluted with errors in misclassification and/or feature coordinates. To account for these errors *soft-margin* SVM adds a regularization function $\gamma(\mathbf{x}^{(n)})$ to the SVM objective function $\epsilon$, where

$$\gamma(\mathbf{x}^{(n)}) \quad = \quad max(0, 1 - y^{(n)}(\mathbf{w} \cdot \mathbf{x}^{(n)} - b)). \tag{8.31}$$

This regularization term, also known as a penalty function, is denoted as the hinge-loss function in the ML community.

If the data are properly classified then $1 - y^{(n)}(\mathbf{w} \cdot \mathbf{x}^{(n)} - b) \leq 0$ so that $\gamma(\mathbf{x}^{(n)}) = 0$; hence, $\epsilon$ collapses to the hard-margin objective function. However, misclassified data points $\mathbf{x}^{(n)}$ lead to progressively larger penalty functions the farther away they are from the correct side of the margin boundary. For example if $\mathbf{x}^{(n)}$ is on the actual decision boundary then $y^{(n)}(\mathbf{w} \cdot \mathbf{x}^{(n)} - b) = 0$ so that the penalty value is $\gamma(\mathbf{x}^{(n)}) = 1$. Therefore the regularized SVM objective function is

$$\epsilon \quad = \quad \lambda||\mathbf{w}||^2 + \frac{1}{N}\sum_{n=1}^{N} \max(0, 1 - y^{(n)}(\mathbf{w} \cdot \mathbf{x}^{(n)} - b)), \tag{8.32}$$

where the parameter $\lambda$ determines the tradeoff between maximizing the margin width and ensuring that $\mathbf{x}^{(n)}$ is on the correct side of the margin boundary. For a large enough value of $\lambda > 0$ the decision boundary is the same as that for unregularized SVM. In this case the margin thickness can be quite skinny if there are slight errors in the feature coordinates. However, the margin thickness can get larger in the presence of such errors if $\lambda$ becomes smaller. A concern is that strong outliers might unduly influence the final solution.

## 8.9 Practical Issues for Implementing SVM

For migration denoising, the three skeletonized characteristics had different units. Therefore, one feature characteristic might consist of numbers that are much bigger or smaller than the other two feature characteristics. This will lead to a scaling problem so that the resulting matrix for QP can be ill-conditioned. In addition, a wide range of examples should be used in order to characterize any type of example outside the training set. If not enough examples are available for training then data augmentation can be used.

### 8.9.1 Scaling

Each of the feature values need to be normalized before they are input into the SVM algorithm. This scaling can take the form of dividing each feature $x_i$ at the $i^{th}$ pixel by its standard deviation:

$$\hat{x}_i = \frac{x_i - \bar{x}}{\sigma}, \tag{8.33}$$

where the data are also demeaned by subtracting the mean value $\bar{x}$ from the feature value $x_i$. An alternative is the mean normalization formula:

$$\hat{x}_i = \frac{x_i - \bar{x}}{max(x_i) - min(x_i)}, \tag{8.34}$$

where $max$ and $min$ take the maximum and minimum values of the feature values for all examples. This will provide an input data set with feature values varying between 1 and $-1$ with a zero mean.

### 8.9.2 Data Augmentation

Data augmentation is used to increase the size the training data. Inexpensive data augmentation methods are the following.

1. Image flip. The images are flipped, i.e. reflected, along different axes to create new images with different orientations. Flipping along the vertical or horizontal axes is the easiest augmentation method.

2. Image rotation. Images are rotated by a rotation transform to create new images.

3. Scaling. Images are either magnified or demagnified to create new images. Care must be taken to not distort images too much.

4. Croping. Crop images so that the main features are still present but the background is reduced in size.

5. Additive Gaussian noise. Add Gaussian noise to images to emulate actual noise in input images.

## 8.10 Summary

The linear SVM is a binary classification method that finds $(\mathbf{w}, b)$ which gives the fattest margin distance in a $D-1$ hyperplane. This defines the optimization problem with linear inequality constraints, also known as the primal problem. The optimal $(\mathbf{w}, b)$ can be computed by a quadratic programming method. If the dimension of the $D \times 1$ feature vector is huge compared to the number $N$ of feature vectors in a mini-batch, then solving the reduced dual problem is the only way to go.

If the data are not separable by a hyperplane, then a higher-order SVM surface can be found by applying a non-linear transformation $\mathbf{z}_i = \phi(\mathbf{x})_i$ to the original feature vectors

$\mathbf{x}$, where the new $D_z \times 1$ feature vector is $\mathbf{z}$ and $D_z > D$. To reduce the computational costs, the primal SVM problem is transformed into the dual Lagrangian problem, where the number of unknowns $D$ is the same as that for original linear problem.

Kernel methods are used to avoid the problems of highly oscillatory basis functions with a non-linear transform. The first step is to specify the functional form of a kernel $k(\mathbf{x}, \mathbf{x}') = \mathbf{z}(\mathbf{x})^T \cdot \mathbf{z}(\mathbf{x}')$, where there is no explicit specification of $\mathbf{z}$. The solution to the reduced Lagrangian is then obtained by a QP method, and the validation data can now be classified.

## 8.11 Exercises

1. Form the Lagrangian for the dual problem of soft-margin SVM in equation 8.32. Write down the equations for the stationarity conditions of $L(\boldsymbol{\alpha}, \mathbf{w}, b)$. The derivative of the *max* operator cannot be strictly defined. Replace it with a suitable approximation that can be differentiated.

2. Reformulate the objective to be $\epsilon = 1/2||\mathbf{w}||^2 + C\sum_{n=1}^{N} \eta_n$ with the constraint

$$\mathbf{w} \cdot \mathbf{x}^{(n)} + b - 1 \quad \geq \quad \eta_n, \tag{8.35}$$

   where $\eta_n \geq 0$. Here, $C$ is a specified tradeoff parameter. What are the KKT conditions? See section 8.13 for the definition of the KKT conditions.

3. Derive the reduced Lagrangian for the above problem and state the constraints. See page 333 in Bishop (2007).

4. Derive the gradient for the misfit function in equation 8.13. Write a pseudo-MATLAB code for finding $\mathbf{x}^b$ from $\mathbf{z}^b$ by a steepest descent method.

## 8.12 Appendix: Defining the Dual Problem with a Lagrangian

Finding the optimal $\mathbf{x}$ that minimizes or maximizes a quadratic functional $f(\mathbf{x})$ subject to $N$ inequality constraints $g(\mathbf{x})^{(n)} \geq 0$ can be recast as the solution to the dual problem. The objective function for the dual problem is formed by adding $N$ weighted inequality constraints onto $f(\mathbf{x})$ to form what is known as the Lagrangian: $L = 1/2 f(\mathbf{x}) - \sum_n \alpha_n g(\mathbf{x})^{(n)}$ subject to the simpler inequality constraints $\alpha_n \geq 0$. There is an unknown weight $\alpha_n$, also known as the Lagrange multiplier, for each constraint so that there are $N$ more unknowns to be determined. This dual problem is sometimes computationally less expensive to solve than the *primal* problem, as discussed in section 8.4. We now give a heuristic and intuitive derivation of a Lagrangian function for one inequality constraint, i.e. $n = 1$.

Consider the following minimization problem with one inequality constraint:

$$\begin{array}{ll} \underset{\mathbf{x}}{\text{minimize}} & f(\mathbf{x}) \\ \text{subject to} & g(\mathbf{x}) \geq 0, \end{array}. \tag{8.36}$$

Figure 8.12a shows a 2D example where $f(\mathbf{x})$ describes the red elliptical contours and $g(\mathbf{x}) = \sqrt{x_1^2 + x_2^2}$ describes a circle for the inequality constraint. In this case all points

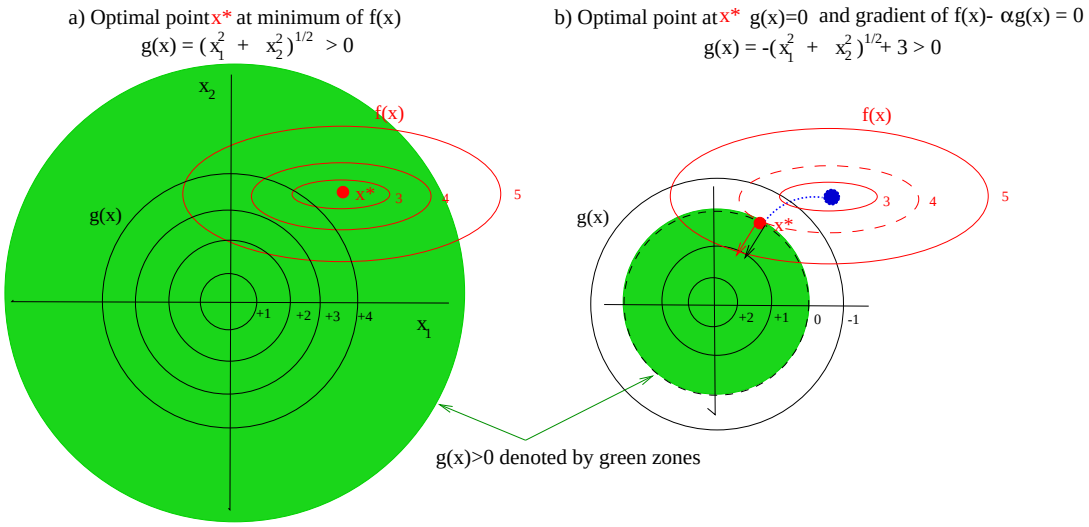


Figure 8.12: Red and black contours for the functions $f(\mathbf{x})$ and $g(\mathbf{x})$, respectively; the green areas define the points $\mathbf{x} \in V_o$ that satisfy the inequality constraint $g(\mathbf{x}) \geq 0$. In a) the constraint $g(\mathbf{x}) = \sqrt{x_1^2 + x_2^2} \geq 0$ is satisfied everywhere so the minimum of $f(\mathbf{x})$ is at the origin. In b), $g(\mathbf{x}) = -\sqrt{x_1^2 + x_2^2} + 3 \geq 0$ so the feasible solution points $\mathbf{x}$ are inside the small green disk. The points $\mathbf{x} \in B_o$ on the dashed circular boundary satisfy $g(\mathbf{x}) = 0$, and the optimal point where $L$ is stationary and a maximum is at the "kiss" point $\mathbf{x}^*$ on $B_o$ where $\nabla L(\mathbf{x}, \alpha) = \nabla f(\mathbf{x}) - \alpha \nabla g(\mathbf{x}) = 0$. The points along the dashed blue line are defined to be the ones where the elliptical and circular contours "kiss" each other, i.e. they satisfy the stationary condition $\nabla f(\mathbf{x}) - \alpha \nabla g(x) = 0$ where $\alpha = |\nabla f(\mathbf{x})|/|\nabla g(x)|$.

in the infinitely-extended green region in Figure 8.12a satisfy the inequality constraint $g(\mathbf{x}) \geq 0$ so that the solution to equation 8.36 is at the center point $\mathbf{x}^*$ of the red ellipse. An unconstrained steepest descent method can then be used to search for $\mathbf{x}$ that satisfies $\nabla f(\mathbf{x}) = 0$.

If $g(\mathbf{x}) = -\sqrt{x_1^2 + x_2^2} + 3 \geq 0$ then the feasible points $\mathbf{x}$ are only in the small green disk in Figure 8.12b. Which of these feasible points minimizes $f(\mathbf{x})$? It is geometrically clear that the dashed red ellipse with the contour value $f(\mathbf{x}) = 4$ just *kisses* the boundary of the green disk at $\mathbf{x}^*$. No other ellipse contour with a *smaller* value intersects the green region. Therefore, $\mathbf{x}^*$ must be the solution that minimizes $f(\mathbf{x})$ and also satisfies the inequality constraint in equation 8.36. As discussed below, $L$ achieves a maximum at this stationary point.

Notice that the circular contour at $\mathbf{x}^*$ in Figure 8.12b is tangent to the red dashed contour of the ellipse. This means that the normal vectors $\nabla g(\mathbf{x})$ and $\nabla f(\mathbf{x})$ at $\mathbf{x}^*$ are parallel[1] to one another at the minimization point $\mathbf{x}^*$. This suggests that we can define a new function $L$ that is the weighted sum

$$L(\mathbf{x}, \alpha) \quad = \quad f(\mathbf{x}) - \alpha g(\mathbf{x}), \tag{8.37}$$

[1]The values of the ellipse and circle are both increasing to the left at $\mathbf{x}^*$. Hence, the gradients are parallel to one another, not anti-parallel.

which has the property of being stationary at the *kissing* point $\mathbf{x}^*$:

$$\begin{aligned} \nabla L(\mathbf{x},\alpha)|_{\mathbf{x}=\mathbf{x}^*} &= [\nabla f(\mathbf{x}) - \alpha \nabla g(\mathbf{x})]_{\mathbf{x}=\mathbf{x}^*}, \\ &= 0, \end{aligned} \tag{8.38}$$

if $\alpha = |\nabla f|/|\nabla g| \geq 0$. The weight $\alpha \geq 0$ because the gradients of both functions are parallel and have the same sign at $\mathbf{x}^*$, so $\alpha \geq 0$ to satisfy equation 8.38. In the third quadrant of b), the gradients $\nabla f$ and $\nabla g$ at their kissing points are *anti-parallel* so $\alpha \geq 0$ excludes the stationarity condition in equation 8.38.

Equation 8.38 is a necessary but not sufficient condition for $\mathbf{x}^*$ to be a minimizer of $f(\mathbf{x})$. For example, the dashed blue curve in Figure 8.12b is denoted as the *kiss* curve where the circular and elliptical contours just *kiss* one another so that $\nabla L(\mathbf{x},\alpha)|_{\mathbf{x}=kiss\ pts} = 0$ for $\alpha = |\nabla f|/|\nabla g|$. These *kiss* points along the blue curve can be continued into the green disk.

So we need another mathematical condition in addition to equation 8.38 to pick out the optimal *kiss* point $\mathbf{x}^*$ at the dashed black boundary. This new condition is that, for all the kiss points on the blue *kiss* curve[2] (and its extension into the green zone), the optimal $\mathbf{x}^*$ is where $L(\mathbf{x},\alpha)$ is maximum. This can be proven by noting that the values of $\alpha = |\nabla f|/|\nabla g|$ vary continuously on the "kiss" curve, so that differentiating equation 8.37 w/r to $\alpha$ gives

$$\frac{\partial L}{\partial \alpha} = -g(\mathbf{x}). \tag{8.39}$$

At the boundary point $\mathbf{x}^*$ we have $\frac{\partial L}{\partial \alpha}|_{\mathbf{x}=\mathbf{x}^*} = -g(\mathbf{x}^*) = 0$, and at points just to the left of $\mathbf{x}^*$ the slope is negative and to the right the slope is positive. This is the condition for $L(\mathbf{x},\alpha)$ to be a maximum at $\mathbf{x}^*$ along the kiss curve. Therefore, the two conditions that are necessary and sufficient for $\mathbf{x}^*$ to minimize $f(\mathbf{x})$ subject to the inequality constraint are

$$\begin{aligned} \underset{\mathbf{x},\boldsymbol{\alpha}}{\text{maximize}} \quad & f(\mathbf{x}) - \alpha g(\mathbf{x}), \\ \text{subject to} \quad & \nabla f(\mathbf{x}) - \alpha \nabla g(\mathbf{x}) = 0 \\ & \alpha \geq 0, \end{aligned} \tag{8.40}$$

The next section shows a 1D example for solving the Lagrangian dual problem.

### 8.12.1 Simple Example of a Dual Solution

For a simple 1D convex minimization problem with inequality constraints, the optimization problem is defined as

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & f(x) \\ \text{subject to} \quad & g(x) \geq 0, \end{aligned} \quad . \tag{8.41}$$

For the example $f(x) = x^2$ and one inequality constraint, equation 8.41 becomes

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & f(x) = x^2 \\ \text{subject to} \quad & g(x) = 0.7x - 0.14 \geq 0, \end{aligned} \tag{8.42}$$

[2]The blue kiss curve is defined by the *kissing* points where $\nabla L = 0$ for $\alpha = |\nabla f|/|\nabla g|$

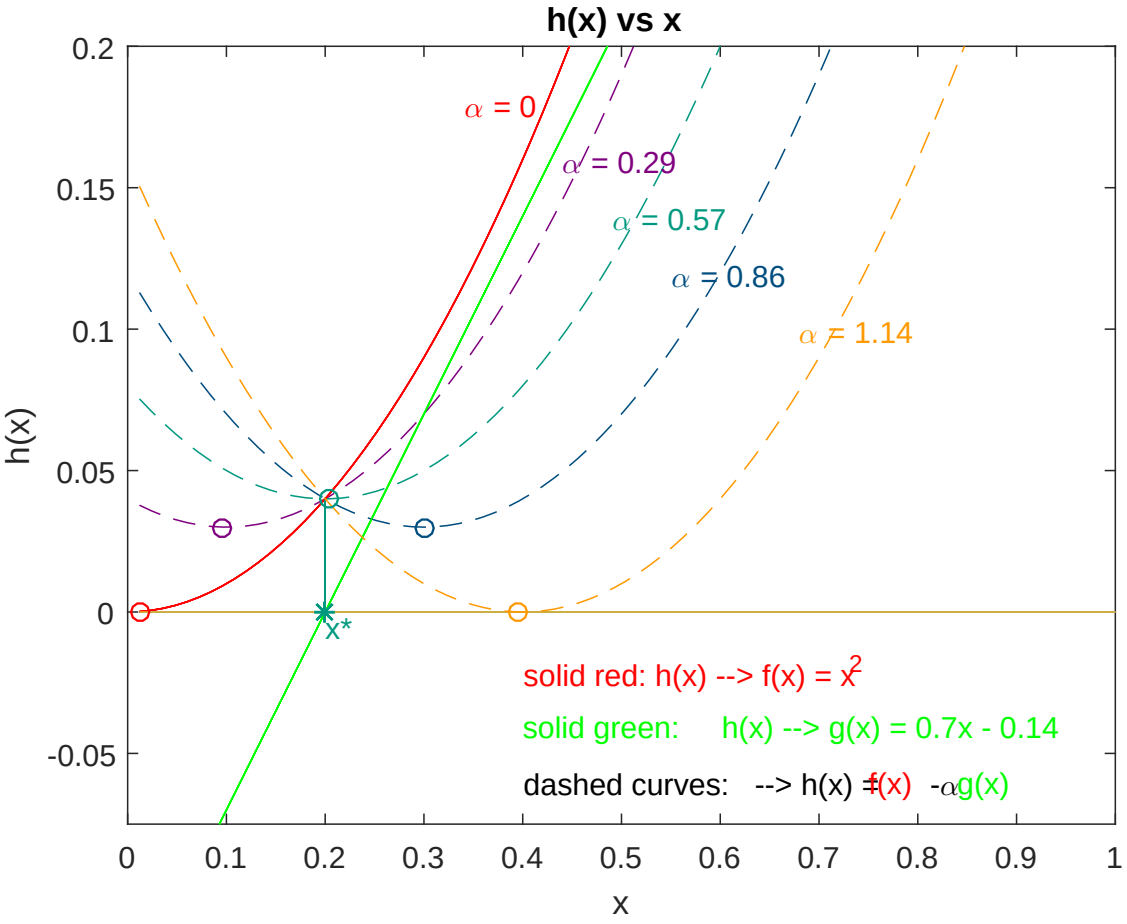


Figure 8.13: Plots of $f(x) = x^2$, $g(x) = 0.7x - 0.14$, and $f(x) - \alpha g(x)$, where $\mathbf{x}^*$ is the optimal solution that minimizes $L$ subject to the inequality constraint. The Lagrangian $L(x, \alpha) = f(x) - \alpha g(x)$ is plotted as dashed lines for various values of $\alpha$.

where $f(x) = x^2$, $g(x) = 0.7x - 0.14$, and $h(x) = f(x) - \alpha g(x)$ are plotted in Figure 8.13. The inequality constraint is $g(x) = 0.7x - 0.14 \geq 0$ so that the feasible points are for $x \geq 0.2$. It is obvious that $x^* = 0.2$ at the green star minimizes $f(x)$ for the feasible points $x \geq 0.2$. In contrast, the global minimum of $f(x)$ is at $x = 0$ to the left of $x^*$, but $x = 0$ will violate the inequality constraint $x \geq 0.2$.

An alternative to the above optimization problem is to find $x$ that minimizes the Lagrangian

$$\begin{array}{ll} \underset{x}{\text{minimize}} & L(x, \alpha) = f(x) - \alpha g(x), \\ \text{subject to} & \alpha > 0, \end{array} \quad . \tag{8.43}$$

and also maximizes $L(x, \alpha)$ w.r.t. $\alpha$, where $\alpha$ is the Lagrange multiplier. These two sta-

tionarity conditions are expressed as

$$\begin{aligned} \frac{\partial L}{\partial x} &= \frac{\partial f}{\partial x} - \alpha \frac{\partial g}{\partial x} = 0 \\ \frac{\partial L}{\partial \alpha} &= -g(x) = 0 \qquad \text{subject to } \alpha > 0. \end{aligned} \tag{8.44}$$

For the equation 8.42 example,

$$L(x, \alpha) = x^2 - \alpha(0.7x - 0.14), \tag{8.45}$$

so that equation 8.44 becomes

$$\frac{\partial L}{\partial x} = \frac{\partial x^2}{\partial x} - \alpha \frac{\partial (0.7x - 0.14)}{\partial x} = 2x - 0.7\alpha = 0 \quad \rightarrow \quad \alpha = x/0.35, \tag{8.46}$$

$$\frac{\partial L}{\partial \alpha} = -0.7x + 0.14 = 0 \quad \rightarrow \quad x = 0.2. \tag{8.47}$$

Plugging equation 8.47 into equation 8.46 gives $\alpha = 0.2/0.35 = 0.57$, which is associated with the dashed blue curve $L(x, \alpha = 0.57)$ plotted in Figure 8.13. This value of $\alpha = 0.57 > 0$ satisfies the positivity constraint and also balances out the gradients of $df/dx$ and $dg/dx$ so that the sum $\partial L/\partial x = \partial f/\partial x - \alpha \partial g/\partial x = 0$ is stationary at $x^* = 0.2$.

At each value of $x$ there is a unique value of $\alpha = x/0.35$ (see equation 8.46) that allows $\partial L/\partial x = 0$. Setting $\alpha \rightarrow \alpha(x) = x/0.35$ from equation 8.46 says that $L(x, \alpha(x))$ is a maximum at $x^* = 0.2$ because it satisfies the maximality conditions: $\partial L/\partial \alpha = 0$ for $x = 0.2$ and its second derivative

$$\begin{aligned} \frac{\partial^2 L(x = 0.35 * \alpha, \alpha)}{\partial \alpha^2} &= \frac{\partial^2 (0.1225\alpha^2 - 0.245\alpha^2 + 0.049\alpha)}{\partial \alpha^2}, \\ &= -.245 \end{aligned} \tag{8.48}$$

is negative[3] Thus, the values of $L(x, \alpha(x))$ plotted against $x$ intersect the circled points in Figure 8.13 where $L(x, \alpha(x))$ is a maximum at $x^* = 0.2$.

## 8.13 Karush-Kuhn-Tucker Conditions

If $\mathbf{x}$ is in the *inactive* region where $g(\mathbf{x}) > 0$ then the inequality constraint $g(\mathbf{x}) \geq 0$ is denoted as inactive and $\alpha = 0$ for the solution to the Lagrangian problem. This constraint is not needed when $\mathbf{x}$ is in the inactive region so it makes sense to set $\lambda = 0$ because it leads to a smaller Lagrangian, i.e.

$$\mathbf{x} \in \text{inactive region}; \quad [f(\mathbf{x}) - \alpha g(\mathbf{x})]_{\ \alpha > 0} \ > \ [f(\mathbf{x}) - \alpha g(\mathbf{x})]_{\alpha = 0}. \tag{8.49}$$

However, if $\mathbf{x}$ is on the boundary defined by $g(\mathbf{x}) = 0$ then the constraint is active and $\lambda > 0$.

[3]The negative second-derivative is an alternative to the first-derivative maximum conditions we used earlier.

Assigning $\alpha$ to be either 0 or $> 0$ is compactly summarized by the condition $\alpha g(\mathbf{x}) = 0$, where $\alpha > 0$ for $\mathbf{x}$ on the boundary or $\alpha = 0$ for $\mathbf{x}$ in an inactive region where $g(\mathbf{x}) > 0$. Thus, this leads to three constraints for the Lagrangian problem known as the Karush-Kuhn-Tucker (KKT) conditions:

$$\begin{aligned} g(\mathbf{x}) &\geq 0, \\ \alpha &\geq 0, \\ \alpha g(\mathbf{x}) &= 0. \end{aligned} \tag{8.50}$$

If there are $N$ inequality constraint such that $g(\mathbf{x})^{(n)} \geq \; for \; n = [1, 2, \ldots N]$ then there are $N$ Lagrange multipliers with $\alpha_n \geq 0$ and $g(\mathbf{x})^{(n)}\alpha_n = 0$ for $n = [1, 2, \ldots N]$. See Nocedal and Wright (1999) for more details.

## 8.14 Appendix: Diffraction Stack Migration

Integral equation migration is a popular seismic imaging method (Yilmaz, 2001) used in the oil industry. It can be described as summing the weighted reflection data along pseudo-hyperbolas associated with each trial image point at $\mathbf{x}$ and placing that summation number at $\mathbf{x}$ to give the migration image $m(\mathbf{x})$. This summation is described by the diffraction-stack migration formula

$$\begin{aligned} m(\mathbf{x}) &= \sum_A \sum_B \sum_\omega [i\omega D(\mathbf{B}|\mathbf{A}) e^{-i\omega(\tau_{Ax}+\tau_{xB})}], \\ &= \sum_A \sum_B \dot{d}(\mathbf{B}, \tau_{Ax} + \tau_{xB}|\mathbf{A}), \end{aligned} \tag{8.51}$$

where $D(\mathbf{B}|\mathbf{A})$ is the frequency-domain representation of the trace in Figure 8.14, its inverse Fourier transform is given by $d(\mathbf{B}, t|\mathbf{A})$, and the summations run over the indices for the source at $A$ and the receiver at $\mathbf{B}$. For a homogeneous medium having velocity $c$ the time for waves to propagate from the source at $\mathbf{A}$ down to the scatterer at $\mathbf{x}_0$ and back up to the receiver at $\mathbf{B}$ is given by

$$\tau_{Ax} + \tau_{xB} = \frac{\sqrt{(x_A - x_0)^2 + (z_A - z_0)^2} + \sqrt{(x_B - x_0)^2 + (z_B - z_0)^2}}{c}, \tag{8.52}$$

where $z_A = z_B = 0$ for sources and receivers on a horizontal datum. This two-way traveltime equation describes a hyperbola in $x_B - time$ coordinates for any image point at $\mathbf{x}$ and a source fixed at $\mathbf{A}$; the apex of the hyperbola is centered over the point scatterer at $\mathbf{x}_o$. If the reflection time on the leftside of equation 8.52 is fixed at a specific time then the values of $\mathbf{x}$ that satisfy this traveltime equation form an ellipse with the foci at $\mathbf{A}$ and $\mathbf{B}$, as depicted in Figure 8.14.

The observed data for a single scatterer can be approximated by

$$D(\mathbf{B}|\mathbf{A}) = e^{-i\omega(\tau_{Ax_o}+\tau_{x_oB})}, \tag{8.53}$$

where geometric spreading and the reflection coefficient are conveniently ignored. Substituting equation 8.53 into equation 8.51 gives

$$m(\mathbf{x}) = \sum_A \sum_B \sum_\omega [i\omega e^{-i\omega(\tau_{Ax}+\tau_{xB}-\tau_{Ax_o}-\tau_{x_oB})}]. \tag{8.54}$$

If the trial image point at $\mathbf{x}$ coincides with the actual scatterer at $\mathbf{x}_o$ then the phases cancel one another exactly and the summation over the sources at $\mathbf{A}$ and receivers at $\mathbf{B}$ is completely coherent and sums to a very large number to be placed at $\mathbf{x}_o$. On the other hand, if $\mathbf{x}$ is far from the actual scatterer then the predicted hyperbola will not match the actual hyperbola of recorded events. The consequence is that the summation in equation 8.54 will be largely incoherent and sum to a small number to be placed at the trial image point at $\mathbf{x} \neq \mathbf{x}_0$. data along.

This type of imaging is also known as diffraction-stack migration (Yilmaz, 2001). For a single source and single receiver the above equation becomes

$$m(\mathbf{x}) \quad = \quad \dot{d}(B, \tau_{Ax} + \tau_{xB}|A). \tag{8.55}$$

In a 2D homogeneous medium, the reflection traveltime $\tau^{ref.} = \tau_{Ax} + \tau_{xB}$ associated with the trial scattering point at $\mathbf{x}$ can be expressed as

$$\tau^{ref.} \quad = \quad [\sqrt{(x_A - x)^2 + (z_A - z)^2} + \sqrt{(x_B - x)^2 + (z_B - z)^2}]/v, \tag{8.56}$$

which, for $\tau^{ref.} = cnst$, describes an ellipse in model-space coordinates $(x, z)$ with the foci at $\mathbf{A} = (x_A, z_A)$ and $\mathbf{B} = (x_B, z_B)$. It follows from equation 8.55 that the reflectivity image $m(\mathbf{x})$ is approximated by smearing the trace amplitude at time $\tau^{ref.}$ over the ellipse described by equation 8.56 in model space.

Smearing the seismic amplitude over an ellipse is shown in Figure 8.14a; here, the temporal interval $T$ of the trace's source wavelet determines the thickness $c/T$ of the fat ellipse in $(x, z)$ space. Somewhere along this ellipse is a scatterer that gave rise to the event at time $\tau^{ref.}$ for the receiver at $\mathbf{d}$. The scatterer's location can be better estimated by stacking (see equation 8.51) the "smears" from other traces into the model, as illustrated in Figure 8.14b. Figure 8.15 gives an example of migration of poststack data (where the source is at the receiver) for a homogeneous velocity model with 6 scattering points. The lateral spatial resolution of the image becomes worse with increasing depth.

## 8.15 Dip Angles, Coherency and Amplitudes of a Migration Image

Dip angle, coherency and amplitude of a migration image are skeletonized characteristics that can be used to possibly distinguish noise from signal. We now describe the computation of each feature.

To compute these three skeletonized characteristics we first define a $5 \times 5$ window centered at the $i^{th}$ pixel in the migration image. A $5x5$ window is used because it is about one wavelength in size. To compute the dip angle in this window we apply a local slant stack (Yilmaz, 2001) to the windowed data centered at the $i^{th}$ pixel to get the dip angle that has the dominant energy; this dip angle $\theta_i$ is assigned to the $i^{th}$ pixel. Slant stacking consists of summing the amplitudes of the data that intercept a line with a specified dip angle. This dip angle is iteratively incremented by about 15 degrees and the dip angle with the largest summed amplitude is assigned to be $\theta_i$.

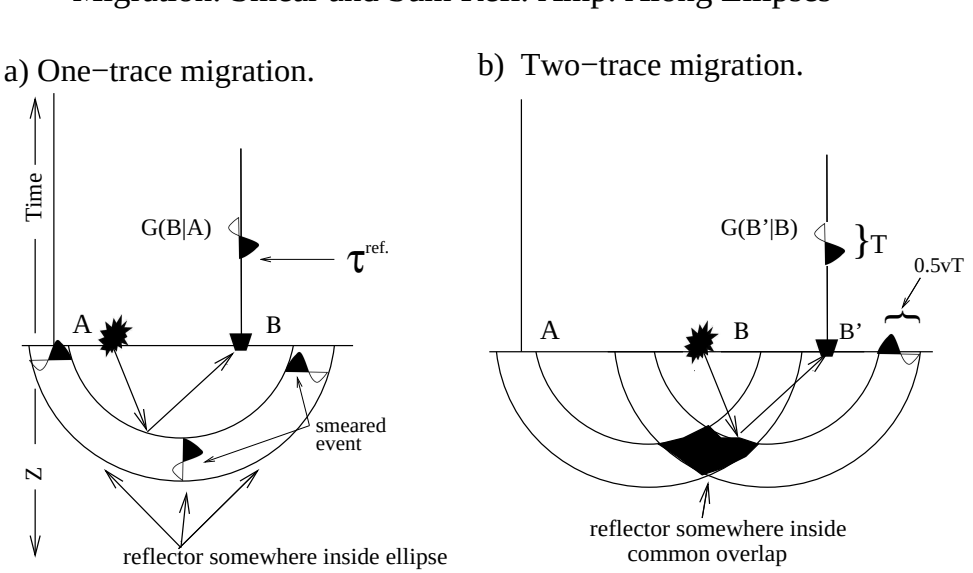


Figure 8.14: Migration is the smearing of trace amplitudes along the appropriate fat ellipses in $(x, z)$ for each source-receiver pair $\mathbf{A} - \mathbf{B}$ (Claerbout, 1992). Migration of two traces in b) has better spatial resolution than migrating just one trace in a), and the thickness of each fat ellipse is $c/T$, where $T$ is the dominant period of the source wavelet.

The window is shifted over by one pixel to the $i+1$ pixel and the slant stack procedure is repeated to give $\theta_{i+1}$. Repeating this procedure for all the training set pixels in Figure 8.6a gives the dip angle image shown in Figure 8.6b. In practice, only 70 pixels are in the actual training set.

The coherency $\phi_i$ at the $i^{th}$ pixel of Figure 8.6a is computed by forming a common image gather (CIG) or a common angle gather (CAG) from the prestack migration images (Yilmaz, 2001). If the migration velocity is accurate then summing along the offset axis (or angle axis if the CAG is used) should give a strong coherency value $\phi_i$ at the $i^{th}$ pixel. Repeating this procedure for all the pixels in Figure 8.6a gives the coherency image shown in Figure 8.6c.

The amplitude image in Figure 8.6d is obtained by computing the Hilbert transform of a trace in the migration image (Yilmaz, 2001). Then the envelope of the migration trace is computed in the window centered at the $i^{th}$ pixel and its maximum amplitude gives $A_i$.

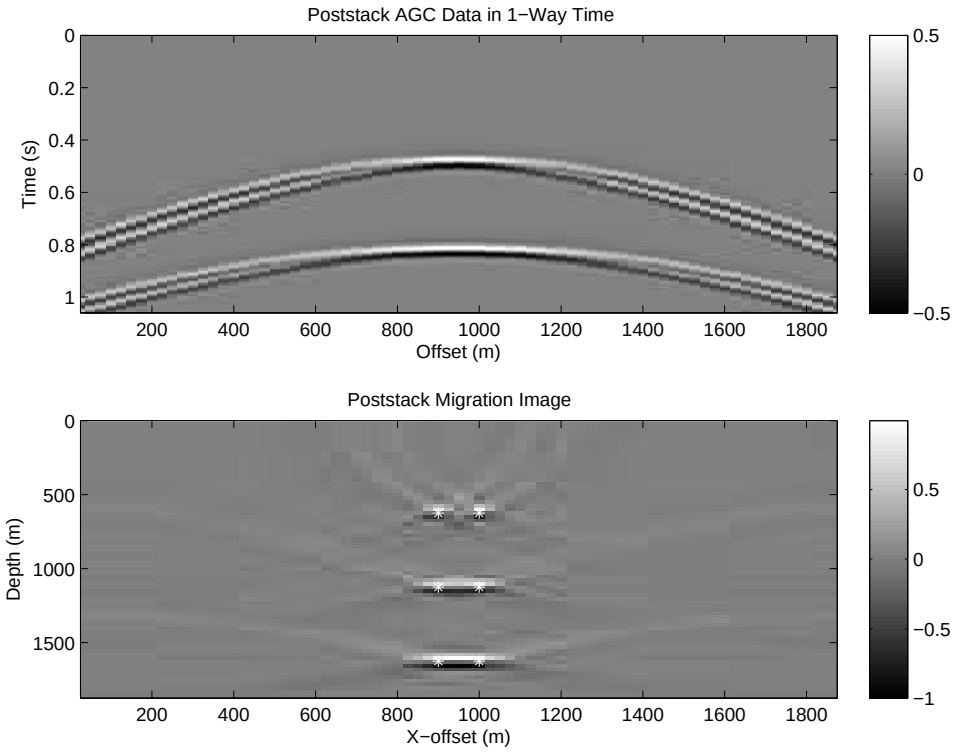


Figure 8.15: (Top) Poststack data and (bottom) migration image where there are 6 point scatterers indicated by the white stars. See MATLAB codes in Appendix 5 to duplicate this figure. Note, the spatial resolution of the image becomes better with decreasing depth of the scatterers.

# References


Aki, K. and P. Richards, 2002, Quantitative Seismology: University Science Books.

Ball, G. and Hall, D., 1965, ISODATA, a novel method of data analysis and pattern classification: Technical report NTIS, AD 699616. Stanford Research Institute, Stanford, CA.

Barghout, L., 2015, Spatial-Taxon information granules as used in iterative fuzzy-decision-making for image segmentation: Granular Computing and Decision-Making, Springer International Publishing, 285–318.

Bishop, C., 2006, Pattern Recognition and Machine Learning: Springer Press.

Boonyasiriwat, C., P. Valasek, P. Routh, B. Macy, W. Cao, and G. T. Schuster, 2009, A multiscale method for time-domain waveform tomography: Geophysics **74**, WCC59–WCC68.

Boyd, S. and L. Vandenberghe, 2004, Convex Optimization, Cambridge University Press. p. 216.

Bunks, C., F. Saleck, S. Zaleski, and G. Chavent, 1995, Multiscale seismic waveform inversion: Geophysics, **60**, 1457–1473.

Chen, Y., 2018, K-means clustering for picking stacking velocity curves in semblance panels: CSIM midyear report.

Claerbout, J. F., 1992, Earth Soundings Analysis: Processing vs Inversion: Blackwell Scientific Inc.

Coates, A., H. Lee, and A. Ng, 2011, An analysis of single-layer networks in unsupervised feature learning: Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics, PMLR, 215–223.

Cortes, C., and V. Vapnik, 1995, Support-vector networks: Machine Learning, 20, 273–297.

DeCoste, D., 2002, Training Invariant Support vector machines: Machine Learning. 46, 161–190.

Gaonkar, B. and Davatzikos, C., 2013, Analytic estimation of statistical significance maps for support vector machine based multi-variate image analysis and classification: Neuroimage, 78, 270-283.

Geoffrion, A. M., 1971, Duality in nonlinear programming: A simplified applications-oriented development: SIAM Review, **13**, 1–37.

Fukushima, K., 1980, Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position: Biol. Cybernetics, **36**, 193–202.

Gill, P., W. Murray, and M. Wright, 1981, Practical Optimization: Academic Press Inc.

Golub, G. and C. van Loan, 1996, Matrix Computations (4th edition): John Hopkins University Press.

Golub, G. and U. von Matt, 1997, Generalized cross-validation for large-scale problems: Journal of Computational and Graphical Statistics, **6**, 1–34.

Goodfellow, I., Y. Benigo, and A. Courville, 2016, Deep Learning, MIT Press.

Google, 2017, Tensorflow tutorials: convolutional neural networks, $https://www.tensorflow.org/tutorials/deep-cnn$, August 2017.

Hotelling, H., 1936,. Relations between two sets of variates: Biometrika, **28**, 321–377.

Hornik, K., 1991, Approximation capabilities of multilayer feedforward networks: Neural Networks, **4(2)**, 251-257.

Hornik, K., M. Stinchcombe, and H. White, 1989, Multilayer feedforward networks are universal approximators: Neural Networks, **2(5)**, 359-366.

Hubel, D. and T. Wiesel, 1962, Receptive fields, binocular interaction and functional architecture in Cat's visual cortex: J. Physiol., **160**, 106–154.

Jain, A., 2010, Data clustering: 50 years beyond K-means: Pattern Recognition, **31**, 651–666.

Jolliffe I.T., 2002, Principal Component Analysis: Springer Series in Statistics, 2nd ed., Springer, NY, 428 pages.

Krizhevsky, A., and G. Hinton, 2009, Learning multiple layers of features from tiny images: M.S. thesis, University of Toronto.

Krizhevsky, A., I. Sutskever, and G. E. Hinton, 2012, Imagenet classification with deep convolutional neural networks: Advances in neural information processing systems, 25, no. 2, 1097–1105.

Li, J., S. Hanafy and G.T. Schuster, 2018, Wave equation dispersion inversion of guided P waves in a waveguide of arbitrary geometry: J. of Geophys. Research: Solid Earth, **123**, 10.1029/2018JB016127.

Lines, L. R. and S. Treitel, 1984, Tutorial, review of least-squares inversion and its application to geophysical problems: Geophysical Prospecting, **32**, 159–186.

LeCun, Y., L. Bottou, Y. Bengio, and P. Haffner, 1998, Gradient-based learning applied to document recognition: Proceedings of the IEEE.

Lloyd, S., 1982. Least squares quantization in PCM: IEEE Trans. Inform. Theory 28, 129–137. Originally as an unpublished Bell laboratories Technical Note (1957).

Lu, K., J. Li, B. Guo, L. Fu, and G. Schuster, 2017, Tutorial for wave-equation inversion of skeletonized data: Interpretation, SO1–SO10.

Luo, Y., and G. T. Schuster, 1991a, Wave equation inversion of skeletonized geophysical data: Geophysical Journal International, 105, 289–294.

Luo, Y. and G. T. Schuster, 1991b, Wave equation traveltime inversion: Geophysics, **56**, 645–653.

MacQueen, J., 1967. Some methods for classification and analysis of multivariate observations: In *Fifth Berkeley Symposium on Mathematics. Statistics and Probability*, University of California Press, 281–297.

Menke, W., 1984, Geophysical Data Analysis: Discrete Inverse Theory: Academic Press Inc., NY, NY.

Nitish, N., C. G. Hinton, A. Krizhevsky, I. Sutskever, R. Salakhutdinov, 2014, Dropout: A Simple Way to Prevent Neural Networks from overfitting: Journal of Machine Learning Research, **15**, 1929–1958.

Nocedal, J. and S. Wright, 1999, Numerical Optimization: Springer Verlag Inc.

Patel, A. B., T. Nguyen, R. Baraniuk, 2016, A probabilistic framework for deep learning: 2016, arXiv:1612.01936P.

Patel, A. and S. Chatterjee, 2016, Computer vision-based limestone rock-type classification using probabilistic neural network: Geoscience Frontiers, 7, 53–60.

Pearson, K., 1901, On lines and planes of closest fit to systems of points in space: Philosophical Magazine, **2 (11)**, 559–572.

Perez, C., 2013, Regularization of neural networks using DropConnect: ICML and JMLR, W and CP., jmlr.org.

Press, W., S. Teukolsky, W. Vetterling, and B. Flannery, 2007, Numerical Recipes: The Art of Scientific Computing: Cambridge University Press Inc.

Qi, J., G. Machado, and K. Marfurt, 2017, A workflow to skeletonize faults and stratigraphic features: Geophysics, **82**, no. 4, O57–O70.

Rickett, J., 2003, Illumination-based normalization for wave-equation depth migration: Geophysics, **68**, 1371–1379.

Ripley, B. D., 1996, Pattern Recognition and Neural Networks: Cambridge University Press.

Rosenblatt, F., 1958, The Perceptron: A probabilistic model for information storage and organization in the brain: Cornell Aeronautical Laboratory, Psychological Review, v65, No. 6, 386–408. doi:10.1037/h0042519.

Rosenblatt, F., 1962, Principles of Neurodynamics: Washington, DC:Spartan Books.

Steinhaus, H., 1956, Sur la division des corp materiels en parties: Bull. Acad. Polon. Sci. IV (C1.III), 801–804.

Sun, Y. and G. T. Schuster, 1992, Hierarchic optimizations for smoothing and cooperative inversion: SEG Technical Program Expanded Abstracts, 745–748.

Tarantola, A., 1987, Inverse Problem Theory Methods for Data Fitting and Model Parameter Estimation: Elsevier Science Publication.

Wiener, N., 1948, Cybernetics, or Control and Communication in the Animal and the Machine. Cambridge: MIT Press.

Vapnik, V. N., 1998, Statistical Learning Theory: Wiley.

Xiong, W., X. Ji, Y. Ma1, Y. Wang, N. Ben-Hassan and Y. Luo, 2018, Seismic fault detection with convolutional neural network: Geophysics (in press).

Yilmaz, O., 2001, Seismic Data Analysis: SEG Press Book.

Zhang, C., C. Frogner, M. Araya-Polo, and D. Hohl, 2014, Machine-learning based auto-

mated fault detection in seismic traces: 76th Conference and Exhibition, EAGE, Extended Abstract. doi: 10.3997/2214–4609.20141500.