Some Applications of Data Science

A thesis presented By

Talha Aslam

CIIT/SP20-BSM-038/ISB

Bachelor of Science in Mathematics

COMSATS University Islamabad

Department of Mathematics

Jan 2024

COMSATS University Islamabad
Departent of Mathematics

Some Applications of Data Science

A Thesis Presented to

COMSATS University Islamabad

In partial fufilment
of the requirement for the degree of

Bachelor of Science in Mathematics

Submitted By

Talha Aslam
CIIT/SP20-BSM-038/ISB

Declaration

I, Talha Aslam, registration number CIIT/SP20-BSM-038/ISB, hereby declare that his thesis entitled ”Some Applications of Data Science” and the work presented in it is my own. I confirm that this work was done wholly or mainly while in candidature for a research degree at this University and where I have consulted that published work of others, this is always clearly attributed. I have acknowledged all main sources of help. No portion of the work presented in this report has been submitted in the support of any other degree or qualification of this or any other University or Institute of learning, if it is found I shall stand responsible.

Signature : ____________________

Talha Aslam

SP20-BSM-038/ISB

Some Applications of Data Science

An Undergraduate Thesis submitted to the Department of Maths as partial fulfilment for the award of Degree Bachelor of Science Mathematics.

Name	Registration Number
Talha Aslam	CIIT/SP20-BSM-038/ISB

Supervisor:
Dr. Salman Amin Malik
Associate Professor
Department of Mathematics
COMSATS University Islamabad

Talha Aslam
SP20-BSM-038/ISB

Final Approval

This thesis titled

Some Applications
of Data Science

By

Talha Aslam

CIIT/SP20-BSM-038/ISB

has been approved for

COMSATS University Islamabad

Supervisor: [illegible]

Dr. Salman Amin Malik
Associate Professor
Department of Maths
COMSATS University Islamabad

External Examiner: [illegible]

Dr. Masood Khan

Head of Department: [illegible]

Prof. Dr. Shams-ul-Islam
Department of Mathematics
COMSATS University Islamabad

Dedication

This work is dedicated to
My beloved Parents, Teachers,
Friends and Everyone who was
always there for me when
I needed them.

Acknowledgements

In the Name of **Allah Almighty**—the Most Compassionate, Most Merciful. The creator of universe and everything it contains. All Praise Belongs to him only, Who gave me vision and courage to accomplish this work successfully.

Prophet Muhammad (PBUH), who builds the height and standard of the character for all over the world and guided his ”Ummah” to seek knowledge from cradle to grave.

A research project is very difficult to be accomplished alone in any field. The contributions of many people have made it possible. I would like to extend my appreciation especially to the following:

I would like to express my deepest gratitude to **Dr. Salman Amin Malik**, my supervisor, for their guidance throughout the journey of my research. His insights and expertise have been invaluable in shaping both this thesis and my academic development.This thesis would not have reached its fruition without his guidance.

I would also like to thank HOD **Prof. Dr. Shams-ul-Islam** and all respected **teachers and professors** whose knowledge, guidance, and passion for education have been instrumental in shaping my academic journey.

I extend my heartfelt gratitude to my **parents**, who have been my constant source of love, motivation, and support. All my achievement, successes and accomplishment would not have been possible without them.

I would like to express my gratitude to **My brother and sister**, for their support throughout the years. Thank you! for being my companions and confidantes.

I would like to express my gratitude to my friends, class mates and fellow research worker Haider Rasool who helped me a lot and guided me whenever I needed it.

To all those mentioned above, and to anyone else who has contributed to my thesis in any way, I offer my deepest thanks.

Talha Aslam
SP20-BSM-038

Abstract

This thesis aims to provide some applications in the field of data science, i.e., *Optimization and Deep Learning.*

For **Optimization** two types of optimizations problems has been considered. Firstly, linear and quadratic programming has been discussed then we considered the game theory. In order to solve the optimization problems duality and saddle points are key ideas, which has been used and explained in this project. The Lagrange multipliers techniques has been used for constrained optimization problems. The gradient descent method and its analog in case of random process, i.e., stochastic gradient descent method has been considered to have minimum of certain physical problems.

Deep learning begins with the architecture of a neural net. An input layer is connected to hidden layers and finally to the output layer. For the training data, input vectors v are known. Also correct outputs are known (often $\boldsymbol{w}$ is the correct classification of $\boldsymbol{v}$). **We optimize the weights $\boldsymbol{w}$ in the learning function F so that $\boldsymbol{F(x, v)}$ is close to $\boldsymbol{w}$ for almost every training input $\boldsymbol{v}$.** Then F is applied to *test data*, drawn from the same population as the training data. If F learned what it needs, the test error will also be low. A neural net defeated the world champion at Go. The function F is often *piecewise linear.* Every neuron on every hidden layer also has a nonlinear "activation function". The ramp function **ReLU**$(\boldsymbol{x})$ is now the overwhelming favorite. There is a growing world of expertise in designing the layers that make up F$(\boldsymbol{x}, \boldsymbol{v})$. We start with *fully connected* layers-all neurons on layer n connected to all neurons on layer $n+1$. Often CNN's are better-*Convolutional neural nets* repeat the same weights around all pixels in an image: a very important construction. A *pooling layer* reduces the dimension. *Dropout* randomly leaves out neurons. *Batch normalization* resets the mean and variance. All these steps create a function that closely matches the training data. Then $F(\boldsymbol{x}, \boldsymbol{v})$ is ready to use.

Contents

Chapter 1

Optimization

The goal of optimization is to minimize a function $F(x_1, ..., x_n)$–often with many variables. This subject must begin with the most important equation of calculus: Derivative = Zero at the minimum point $\boldsymbol{x*}$. With n variables, F has n partial derivatives $\partial F/\partial x_i$. If there are no "constraints" that x must satisfy, we have n equations $\partial F/\partial x_i = 0$ for n unknowns $x_1^*, ..., x_n^*$.

At the same time, there are often conditions that the vector $\boldsymbol{x}$ must satisfy. These **constraints on x** could be equations $\boldsymbol{Ax = b}$ or inequalities $x \geq 0$. The constraints will enter the equations through Lagrange multipliers $\lambda_1, ..., \lambda_m$. Now we have $m + n$ unknowns ($x's$ and $\lambda's$) and $m + n$ equations (derivatives = 0). So this subject combines linear algebra and multivariable calculus. We are often in high dimensions.

This introduction ends with key facts of calculus: the approximation of $\boldsymbol{F(x+\Delta x)}$ by $\boldsymbol{F(x) + \Delta x^T \nabla F + \frac{1}{2} x^T H x}$. Please see that important page.

Evidently this is part of mathematics. Yet optimization has its own ideas and certainly its own algorithms. It is not always presented as an essential course in the mathematics department. But departments in the social sciences and the physical sciences— economics, finance, psychology, sociology and every field of engineering–use and teach this subject because they need it.[1]

We have organized this chapter to emphasize the key ideas of optimization:

1. The central importance of **convexity**, which replaces linearity. Convexity involves second derivatives—the graph of $F(x)$ will bend upwards. A big

question computationally is whether we can find and use all those second partial derivatives $\partial^2 F/\partial x_i \partial x_j$. The choice is between **"Newton methods"** that use them and **"gradient methods"** that don't: second-order methods or first-order methods.

Generally neural networks for deep learning involve very many unknowns. Then gradient methods (first order) are chosen. *Often F is not convex!* The last sections of this chapter describe those important algorithms— they move along the gradient (the vector of first derivatives) toward the minimum of $F(x)$.

2. The meaning of **Lagrange multipliers,** which build the constraints into the equation *derivative = zero*. Most importantly, those multipliers give the **derivatives of the cost with respect to the constraints**. They are the Greeks of mathematical finance.

3. The classical problems **LP, QP, SDP** of "mathematical programming". The unknowns are vectors or matrices. The inequalities ask for nonnegative vectors $\boldsymbol{x} \geq \mathbf{0}$ or positive semidefinite matrices $\boldsymbol{X} \geq \mathbf{0}$. Each minimization problem has a **dual problem**—a *maximization*. The multipliers in one problem become the unknowns in the dual problem. They both appear in a **2-person game**.

4. First-order algorithms begin with **gradient descent**. The derivative of the cost in the search direction is negative. The choice of direction to move and how far to move this is the art of computational optimization. You will see crucial decisions to be made, like adding "momentum" to descend more quickly.

   Levenberg-Marquardt combines gradient descent with Newton's method. The idea is to get near a* with a first order method and then converge quickly with (almost) second order. This is a favorite for nonlinear least squares.

5. **Stochastic gradient descent.** In neural networks, the function to minimize is a sum of many terms-the losses in all samples in the training data. The learning function F depends on the "weights". Computing its gradient is expensive. So each step learns only a **minibatch** of B training samples—chosen randomly or "stochastically". One step accounts for a part of the

data but not all. We *hope and expect that the part is reasonably typical of the whole.*

Stochastic gradient descent—often with speedup terms from **"ADAM"** to account for earlier step directions–has become the workhorse of deep learning. The partial derivatives we need from F are computed by **backpropagation**. This key idea will be explained separately in the final chapter of the book.

Like so much of applied mathematics, optimization has a discrete form and a continuous form. Our unknowns are vectors and our constraints involve matrices. For the calculus of variations the unknowns are functions and the constraints involve integrals. The vector equation "first derivative = zero" becomes the Euler-Lagrange differential equation "first variation = zero".

That is a parallel (and continuous) world of optimization but we won't go there.

1.0.1 The Expression "argmin"

The minimum of the function $F(x) = (x-1)^2$ is zero: $\min F(x) = 0$. That tells us how low the graph of F goes. But it does not tell us which number x^* gives the minimum. In optimization, that "argument" x^* is the number we usually solve for. The minimizing x for $F = (x-1)^2$ is $x^* = \arg\min F(x) = 1$.

$$\boxed{\textbf{argmin F(x)} = \text{value(s) of x where F reaches its minimum.}}$$

For strictly convex functions, argmin $F(x)$ is **one point $\boldsymbol{x}^*$**: an isolated minimum.

1.0.2 Multivariable Calculus

Machine learning involves functions $F(x_1, ..., x_n)$ of many variables. We need basic facts about the first and second derivatives of F. These are "partial derivatives" when $n > 1$.

The important facts are in equations (1) − (2) − (3).I don't believe you need a whole course (too much about integrals) to use these facts in optimizing a deep learning function $F(x)$.

$$\boxed{\begin{array}{cc} \textbf{One Function } \boldsymbol{F} & \\ \textbf{One variable } \boldsymbol{x} & \boldsymbol{F(x+\Delta x) \approx F(x) + \Delta x \frac{dF}{dx}(x) + \frac{1}{2}(\Delta x)^2 \frac{d^2F}{dx^2}} \end{array}} \tag{1}$$

This is the beginning of a Taylor series—and we don't often go beyond that second-order term. The first terms $F(x) + (\Delta x)(dF/dx)$ give a first order approximation to $F(x + \Delta x)$, using information at x. Then the Δx^2 term makes it a second order approximation.

The Δx term gives a point on the tangent line—tangent to the graph of *F(x)*. The $(\Delta x)^2$ term moves from the tangent line to the "tangent parabola". The function F will be **convex**—its slope increases and its graph bends upward, as in $y = x^2$—when the second derivative of *F(x)* is positive: $\boldsymbol{d^2F/dx^2 > 0}$. Equation (2) lifts (1) into n dimensions.

$$\boxed{\begin{array}{cc} \textbf{One Function F} & \\ \textbf{One variable} & \boldsymbol{F(x+\Delta x) \approx F(x) + (\Delta x)^T \nabla F + \frac{1}{2}(\Delta x)^T H (\Delta x)} \\ \boldsymbol{x_1 to x_n} & \end{array}} \tag{2}$$

This is the important formula! **The vector ∇F is the gradient of F**—the column vector of n partial derivatives $\partial F/\partial x_1$ to $\partial F/\partial x_n$. **The matrix H is the Hessian matrix**. H is the symmetric matrix of second derivatives $H_{ij} = \partial^2 F/\partial x_i \partial x_j = \partial^2 F/\partial x_j \partial x_i$.

The graph of $y = F(1, ..., n)$ is now a surface in $(n + 1)$–dimensional space. The tangent line becomes a tangent plane at x. When the second derivative matrix H is positive definite, F is a *strictly convex function*: it stays above its tangents. A convex function F has a **minimum** at $\boldsymbol{x^*}$ if $\boldsymbol{f = \nabla F(x^*) = 0}$: n equations for $\boldsymbol{x^*}$.

Sometimes we meet m different functions $f_1(x)$ to $f_m(x)$: a vector function $\boldsymbol{f}$:

$$\begin{array}{l} \boldsymbol{m} \textbf{ functions } \boldsymbol{f} = (\boldsymbol{f_1},...,\boldsymbol{f_m}) \\ \boldsymbol{n} \textbf{ variables } \boldsymbol{x} \ = (\boldsymbol{x_1},...,\boldsymbol{x_n}) \end{array} \qquad \boldsymbol{f}(\boldsymbol{x}+\boldsymbol{\Delta x}) \approx \boldsymbol{f}(\boldsymbol{x}) + \boldsymbol{J}(\boldsymbol{x})\boldsymbol{\Delta x} \tag{3}$$

The symbol $\boldsymbol{J(x)}$ represents the m by n **Jacobian matrixof f(x)** at the point x. The m rows of J contain the gradient vectors of the m functions $f_1(x)$ to $f_m(x)$.

$$\textbf{Jacobian matrix } \boldsymbol{J} = \begin{bmatrix} (\boldsymbol{\nabla f_1})^T \\ \cdot \\ \cdot \\ (\boldsymbol{\nabla f_m})^T \end{bmatrix} = \begin{bmatrix} \frac{\partial \boldsymbol{f_1}}{\partial \boldsymbol{x_1}} & \cdot\ \cdot\ \cdot & \frac{\partial \boldsymbol{f_1}}{\partial \boldsymbol{x_n}} \\ \cdot & & \cdot \\ \cdot & & \cdot \\ \frac{\partial \boldsymbol{f_m}}{\partial \boldsymbol{x_1}} & \cdot\ \cdot\ \cdot & \frac{\partial \boldsymbol{f_m}}{\partial \boldsymbol{x_n}} \end{bmatrix} \tag{4}$$

The **Hessian matrix H is the Jacobian J of the gradient $\boldsymbol{f} = \boldsymbol{\nabla F}$!** The determinant of J (when $m = n$) appears in n-dimensional integrals. It is the r in the area formula $\int\int r dr d\theta$.

1.1 Minimum Problems: Convexity and Newton's Method

This part will focus on problems of minimization, for functions $F(x)$ with many variables: $F(\boldsymbol{x}) = F(x_1, ..., x_n)$. There will often be constraints on the vectors x:

Linear constraints	**Ax =b**	(the set of these x is convex)
Inequality constraints	$\boldsymbol{x} \geq \boldsymbol{0}$	(the set of these x is convex)
Integer constraints	**Each $\boldsymbol{x_i}$ is 0 or 1**	(the set of these x is not convex)

The problem statement could be unstructured or highly structured. Then the algorithms to find the minimizing $\boldsymbol{x}$ range from very general to very specific. Here are examples:

Unstructured	Minimize $F(\boldsymbol{x})$ for vectors $\boldsymbol{x}$ in a subset K of $\boldsymbol{R}^n$
Structured	Minimize quadratic cost$F(\boldsymbol{x}) = \frac{1}{2}\boldsymbol{x}^T S\boldsymbol{x}$constrained by$A\boldsymbol{x} = \boldsymbol{b}$
	Minimize linear cost$F(\boldsymbol{x}) = c^T\boldsymbol{x}$constrained by $A\boldsymbol{x} = \boldsymbol{x}$ and $\boldsymbol{x} \geq \boldsymbol{0}$
	Minimize with a binary constraint: each x_i is 0 or 1

We cannot go far with those problems until we recognize the crucial role of **convexity**. We hope the function $F(\boldsymbol{x})$ is convex. We hope the constraint set K is convex. In the theory of optimization, we have too live without linearity. **Convexity** will take control when linearity is lost. Here is convexity for a function $F(\boldsymbol{x})$ and a constraint set K:

K is a convex set	IF $\boldsymbol{x}$ and $\boldsymbol{y}$ are in K, so is the line from $\boldsymbol{x}$ to $\boldsymbol{y}$
F is a convex function	The set of points on and above the graph of F is convex
F is smooth and convex	$F(\boldsymbol{x}) \geq F(\boldsymbol{y}) + (\boldsymbol{\Delta F}(\boldsymbol{y}), \boldsymbol{x} - \boldsymbol{y})$

That last inequality says that the graph of a convex F *stays above its tangent lines.*

A triangle in the plane is certainly a convex set in $\boldsymbol{R}^2$. What about the union of two triangles? Right now I can see only two ways for the union to be convex:

1) One triangle contains the other triangle. The union is the bigger triangle.

2) They share a complete side and their union has no inward-pointing angles.

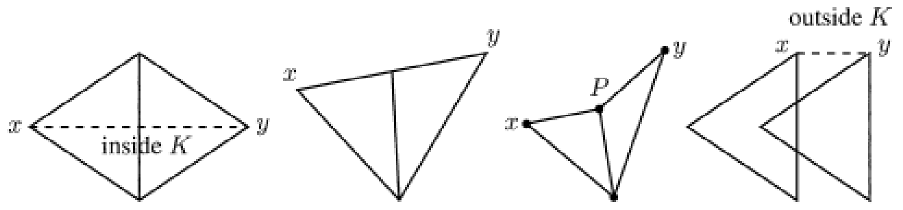


Figure 1.1: Two convex sets and two non-convex sets in $\boldsymbol{R}^2$: Inward-pointing at P.

For functions F there is a direct way to define convexity. Look at all points $p\boldsymbol{x} + (1-p)\boldsymbol{y}$ between $\boldsymbol{x}$ and $\boldsymbol{y}$. The graph of F stays on or goes below a straight line graph,

$$\boxed{\boldsymbol{F} \textbf{ is convex} \quad \boldsymbol{F(px + (1-p)y) \leq pF(x) + (1-p)F(y)} \text{ for } 0 < p < 1} \tag{1}$$

For a **strictly convex** function, this holds with strict inequality (replace $\leq$ by $<$). Then the graph of F goes strictly below the chord that connects the point $\boldsymbol{x}$, $F(\boldsymbol{x})$ to $\boldsymbol{y}$, $F(\boldsymbol{y})$.

Interesting that the graph stays above its tangent lines and below its chords. Here are three examples of convex functions. Only F_2 is strictly convex:

$$F_1 = ax + b \quad F_2 = x^2 (\text{but not } -x^2) \quad F_3 = \max\,(F_1, F_2)$$

The convexity of that function F_3 is an important fact. This is where linearity fails but convexity succeeds! The maximum of two or more linear functions is rarely linear. But **the maximum $\mathbf{F(x)}$ of two or more convex functions $\boldsymbol{F_i(x)}$** is always convex. For any $z = p\boldsymbol{x} + (1-p)\boldsymbol{y}$ between $\boldsymbol{x}$ and $\boldsymbol{y}$, each function F_i has

$$\boxed{F_i(z) \leq pF_i(\boldsymbol{x}) + (1-p)F_i(\boldsymbol{y}) \leq pF(x) + (1-p)F(\boldsymbol{y})} \tag{2}$$

This is true for each i. Then $F(z) = maxF_i(z) \leq pF(x) + (1-p)F(y)$, as required.

The maximum of any family of convex functions (in particular any family of linear functions) is convex. Suppose we have *all linear functions that stay below a convex function F.* Then the maximum of those linear functions below F is exactly equal to F.

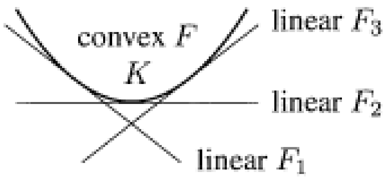


Figure 1.2: **A convex function $\boldsymbol{F}$ is the maximum of all its tangent functions.**

A convex set K is the intersection of all half-spaces that contain it. And the *intersection* of any family of convex sets is convex. But the *union* of convex sets is not always convex.

Similarly the *maximum* of any family of convex functions will be convex. But the *minimum* of two convex functions is generally not convex—it can have a ”double well”.

Here are two useful facts about matrices, based on pos def + pos def = pos def:

The set of $positive\ definite$ n by n matrices is convex.

The set of $positive\ semidefinite$ n by n matrices is convex.

The first set is ”open”. The second set is ”closed”. It contains its semidefinite limit points.

1.1.1 The Second Derivative Matrix

An ordinary function $f(x)$ is convex if $d^2f/dx^2 \geq 0$. Reason: The slope df/dx is increasing. *The curve bends upward* (like the parabola $f = x^2$ with second derivative = 2). The extension to n variables involves the n by n matrix H (x) of second derivatives. If $F(x)$ is a smooth function then there is an almost perfect test for convexity:

$\boldsymbol{F}(x_1, ..., x_n)$ is convex if and only if its second derivative matrix $H(\boldsymbol{x})$ is **positive semidefinite at all x**. That **Hessian matrix** is symmetric because $\partial^2 F/\partial x_i \partial x_j = \partial^2 F/\partial x_j \partial x_i$. The function $\boldsymbol{F}$ is **strictly** convex if $H(\boldsymbol{x})$ is **positive definite at all $\boldsymbol{x}$.**

$$H(\boldsymbol{x}) = \begin{bmatrix} \partial^2 F/\partial x_1^2 & \partial^2 F/\partial x_1 \partial x_2 & . \\ \partial^2 F/\partial x_2 \partial x_1 & \partial^2 F/\partial x_2^2 & . \\ . & . & . \end{bmatrix} H_{ij} = \frac{\partial^2 F}{\partial x_i \partial x_j} = H_{ji}$$

A linear function $F = \boldsymbol{c}^T\boldsymbol{x}$ is convex (but not strictly convex). Above its graph is a half space: flat boundary. Its second derivative matrix is $H = 0$ (very semidefinite).

A quadratic $F = \frac{1}{2}\boldsymbol{x}^T S\boldsymbol{x}$ has gradient $S\boldsymbol{x}$. Its symmetric second derivative matrix is S. Above its graph is a bowl, when S is positive definite. This function F is strictly convex.

1.1.2 Convexity prevents two local minima

We minimize a convex function $F(\boldsymbol{x})$ for $\boldsymbol{x}$ in a convex set K. That double convexity has a favorable effect: *There will not be two isolated solutions.* If $\boldsymbol{x}$ and $\boldsymbol{y}$ are in K and they give the same minimum, then all points z on the line between them are also in K and give that minimum. Convexity avoids the truly dangerous situation when F has its minimum value at an unknown number of separate points in K.

This contribution of convexity is already clear for ordinary functions $F(x)$ with one variable x. Here is the graph of a non-convex function with minima at x and y and z.

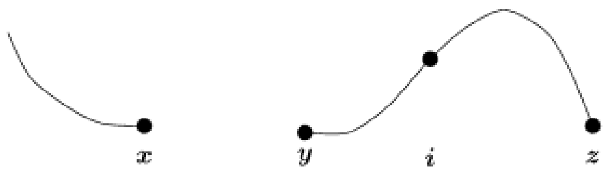


Figure 1.3

F is not convex. It is concave after the inflection point i, where $\partial^2 F/\partial x^2$ goes negative. And F is not defined on a convex set K (because of the gap between x and y). To fix both problems, we could connect x to y by a straight line, and end the graph at i.

For a convex problem to have multiple solutions x and y, the interval between them must be filled with solutions. Never two isolated minima, usually just a single point. **The set of minimizing points æ in a convex problem is convex**.

1.1.3 The l^1 and l^2 and l^∞ Norms of x

Norms $F(x) = ||x||$ are convex functions of x. The unit ball where $||x|| \leq 1$ is a convex set K of vectors x. That first sentence is exactly the triangle inequality:

Convexity of $||x|| \quad ||px + (1-p)y|| \leq p||x|| + (1-p)||y||$

There are three favorite vector norms $\boldsymbol{l^1}$,$\boldsymbol{l^2}$,$\boldsymbol{l^\infty}$. We draw the unit balls $||x|| \leq 1$ in $\boldsymbol{R}^2$:

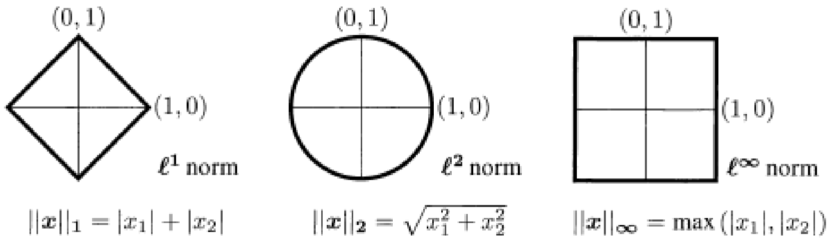

(0, 1)
(1, 0)
ℓ^1 norm
$||x||_1 = |x_1| + |x_2|$
(0, 1)
(1, 0)
ℓ^2 norm
$||x||_2 = \sqrt{x_1^2 + x_2^2}$
(0, 1)
(1, 0)
ℓ^∞ norm
$||x||_\infty = \max(|x_1|, |x_2|)$

Figure 1.4: For all norms,the convex "unit ball" where $||\boldsymbol{x}|| \leq \mathbf{1}$ is centered at $\boldsymbol{x} = \mathbf{0}$.

1.1.4 Newton's Method

We are looking for the point $\boldsymbol{x}^*$ where $F(x)$ has a minimum and its gradient $\nabla F(x^*)$ is the zero vector. We have reached a nearby point x_k . We aim to move to a new point x_{k+1} that is closer than x_k to $x^* = argminF(x)$. What is a suitable step $x_{k+1} - x_k$ to reach that new point x_{k+1}?

Calculus provides an answer. Near our current point x_k, the gradient $\boldsymbol{\nabla F}$ is often well estimated by using its first derivatives—which are the second derivatives of $F(x)$. Those second derivatives $\partial^2 F/\partial x_i \partial x_j$, are in the Hessian matrix H:

$$\boldsymbol{\nabla F}(\boldsymbol{x}_{k+1}) \approx \boldsymbol{\nabla F}(\boldsymbol{x}_k) + H(\boldsymbol{x}_k)(\boldsymbol{x}_{k+1} - \boldsymbol{x}_k). \tag{3}$$

We want that left hand side to be zero. So the natural choice for x_{k+1} comes when the *right side is zero*: we have n linear equations for the step $\Delta x_k = x_{k+1} - x_k$;

$$\boxed{\textbf{Newton's Method} \quad H(\boldsymbol{x}_k)(\Delta \boldsymbol{x}_k) = -\boldsymbol{\nabla F}(\boldsymbol{x}_k) \quad \text{and} \quad \boldsymbol{x}_{k+1} = \boldsymbol{x}_k + \Delta \boldsymbol{x}_k} \tag{4}$$

Newton's method is also producing the minimizer of a quadratic function built from F and its derivatives ∇F and its second derivatives H at the point x_k

$$\boldsymbol{x}_{k+1} \text{ minimizes } F(\boldsymbol{x}_k) + \boldsymbol{\nabla F}(\boldsymbol{x}_k)^T(\boldsymbol{x} - \boldsymbol{x}_k) + (\boldsymbol{x} - \boldsymbol{x}_k)^T H(\boldsymbol{x}_k)(\boldsymbol{x} - \boldsymbol{x}_k). \tag{5}$$

Newton's method is *second order*. It uses second derivatives (in H). There will still be an error in the new x_{k+1}. But that error is proportional to the **square** of the error in x_k:

$$\boxed{\textbf{Quadratic convergence} \quad ||\boldsymbol{x}_{k+1} - \boldsymbol{x}^*|| \le C||\boldsymbol{x}_k - \boldsymbol{x}^*||^2} \tag{6}$$

If $\boldsymbol{x}_k$ is close to $\boldsymbol{x}^*$, then $\boldsymbol{x}_{k+1}$ will be much closer. An example is the computation of $\boldsymbol{x}^* = \sqrt{4} = 2$ in one dimension. Newton is solving $\boldsymbol{x^2 - 4 = 0}$:

$$\textbf{Minimize} \quad \boldsymbol{F(x) = \tfrac{1}{3}x^3 - 4x} \quad \textbf{with } \boldsymbol{\nabla F(x) = x^2 - 4} \quad \textbf{and } \boldsymbol{H(x) = 2x}$$

One step of Newton's method : $H(x_k)(\Delta x_k) = 2x_k(x_{k+1} - x_k) = -x_k^2 + 4$.

Then $2x_k x_{k+1} = x_k^2 + 4$. So newton chooses $\boldsymbol{x_{k+1} = \frac{1}{2}(x_k + \frac{4}{x_k})}$

Guess the square root, divide into 4, and average the two numbers. We can start from 2.5:

$$x_o = \mathbf{2.5} \quad x_1 = \mathbf{2.05} \quad x_2 = \mathbf{2.0006} \quad x_3 = \mathbf{2.000000009}$$

The wrong decimal is twice as far out at each step. **The error $\boldsymbol{x_k - 2}$ is squared:**

$\boxed{x_{k+1} - 2 = \frac{1}{2}(x_k + \frac{4}{x_k}) - 2 = \frac{1}{2x_k}(x_k - 2)^2}$ $\boxed{||\boldsymbol{x} + k + 1 - \boldsymbol{x}^*|| \approx \frac{1}{4}||\boldsymbol{x}_k - \boldsymbol{x}^*||^2}$

Squaring the error explains the speed of Newton's method—provided x_k is close.

How well does Newton's method work in practice? At the start, $\boldsymbol{x}_o$ may not be close to $\boldsymbol{x}^*$. We cannot trust the second derivative at x_o to be useful. So we compute the Newton step $\Delta\boldsymbol{x}_o = \boldsymbol{x}_1 - \boldsymbol{x}_o$, but then we allow **backtracking**:

Choose $\alpha < \frac{1}{2}$ and $\beta < 1$ and reduce the step $\Delta\boldsymbol{x}$ by the factor β until we

know that the new $\boldsymbol{x}_{k+1} = \boldsymbol{x}_k + t\Delta\boldsymbol{x}$ is producing a sufficient drop in $F(x)$:

Reduce t until the drop in $\boldsymbol{F}$ satisfies

$$F(\boldsymbol{x}_k + t\boldsymbol{\Delta x}) \leq F(\boldsymbol{x}_k) + \alpha t \boldsymbol{\nabla F}^T \boldsymbol{\Delta x} \tag{7}$$

We return to backtracking in next section. It is a safety step in any search direction, to know a safe choice of $\boldsymbol{x}_{k+1}$ after the direction from $\boldsymbol{x}_k$ has been set. Or we can fix a small stepsize—this is the hyperparameter–and skip the search in training a large neural network.

Summary Newton's method is eventually fast, because it uses the second derivatives of $F(\boldsymbol{x})$. But those can be too expensive to compute–especially in high dimensions. $Quasi-Newton$ methods in early Section allow the Hessian H to be built gradually from information gained as the algorithm continues. Often neural networks are simply too large to use H. *Gradient descent* is the algorithm of choice. The next two pages describe a compromise that gets better near $\boldsymbol{x}^*$.

1.2 Lagrange Multipliers = Derivatives of the Cost

Unstructured problems deal with convex functions $F(x)$ on convex sets K. This section starts with highly structured problems, to see how Lagrange multipliers deal with constraints. We want to bring out the meaning of the multipliers $\lambda_1, ..., \lambda_m$. After introducing them and using them, it is a big mistake to discard them.

Our first example is in two dimensions. The function F is quadratic. The set K is linear.

$$\textbf{Minimize } \boldsymbol{F(x) = x_1^2 + x_2^2} \textbf{ on the line } \boldsymbol{K : a_1x_1 + a_2x_2 = b}$$

On the line K, we are looking for the point that is nearest to $(0, 0)$. The cost $F(x)$ is distance squared. In Figure given below, the constraint line is **tangent to the circle** at the winning point $\boldsymbol{x}^* = (x_1^*, x_2*)$. We discover this from simple calculus, *after we bring the constraint equation* $a_1x_1 + a_2x_2 = b$ *into the function* $F = x_1^2 + x_2^2$.

This was Lagrange's beautiful idea.

Multiply $\boldsymbol{a_1x_1 + a_2x_2 - b}$ by an unknown multiplier $\boldsymbol{\lambda}$ and add it to $\boldsymbol{F(x)}$Lagrangian $\boldsymbol{L(x, \lambda)}$ $= F(x) + \lambda(a_1x_1 + a_2x_2 - b)$

$$= \boldsymbol{x_1^2 + x_2^2 + \lambda(a_1x_1 + a_2x_2 - b)}$$

Set the derivatives $\boldsymbol{\partial L/\partial x_1}$ and $\boldsymbol{\partial l/\partial \lambda}$ to zero.

Solve those three equations for $\boldsymbol{x_1, x_2, \lambda}$

(1)

$$\boldsymbol{\partial L/\partial x_1 = 2x_1 + \lambda a_1 = 0} \tag{2a}$$

$$\boldsymbol{\partial L/\partial x_2 = 2x_2 + \lambda a_2 = 0} \tag{2b}$$

$$\boldsymbol{\partial L/\partial \lambda = a_1x_1 + a_2x_2 - b = 0} \ \text{(the constraint!)} \tag{2c}$$

The first equations give $x_1 = -\frac{1}{2}\lambda a_1$ and $x_2 = -\frac{1}{2}\lambda a_2$. Substitute into $a_1x_1 + a_2x_2 = b$:

$$-\frac{1}{2}\lambda a_1^2 - \frac{1}{2}\lambda a_2^2 = b \text{ and } \boldsymbol{\lambda = \frac{-2b}{a_1^2 + a_2^2}} \tag{3}$$

Substituting λ into (2a) and (2b) reveals the closest point (x_1^*, x_2^*) and the minimum cost $(x_1^*)^2 + (x_2^*)^2$:

$$\boldsymbol{x_1^*} = -\tfrac{1}{2}\lambda a_1 = \tfrac{\boldsymbol{a_1 b}}{\boldsymbol{a_1^2+a_2^2}} \quad \boldsymbol{x_2^*} = -\tfrac{1}{2}\lambda a_2 = \tfrac{\boldsymbol{a_2 b}}{\boldsymbol{a_1^2+a_2^2}} \quad \boldsymbol{(x_1^*)}^2 + \boldsymbol{(x_2^*)}^2 = \tfrac{\boldsymbol{b^2}}{\boldsymbol{a_1^2+a_2^2}}$$

The derivative of the minimum cost with respect to the constraint level b is minus the Lagrange multiplier:

$$\boxed{\boldsymbol{\tfrac{d}{db}}\left(\tfrac{\boldsymbol{b^2}}{\boldsymbol{a_1^2+a_2^2}}\right) = \tfrac{\boldsymbol{2b}}{\boldsymbol{a_1^2+a_2^2}} = \boldsymbol{-\lambda}} \tag{4}$$

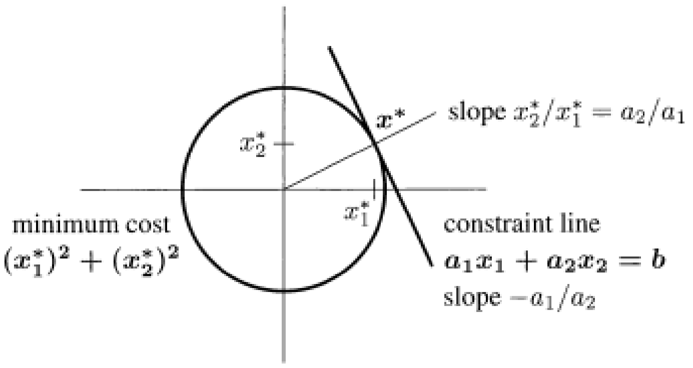

$\boldsymbol{x}^*$
slope $x_2^*/x_1^* = a_2/a_1$
x_2^*
x_1^*
minimum cost
$(\boldsymbol{x_1^*})^2 + (\boldsymbol{x_2^*})^2$
constraint line
$\boldsymbol{a_1 x_1 + a_2 x_2 = b}$
slope $-a_1/a_2$

Figure 1.5: The constraint line is tangent to the minimum cost circle at the solution $\boldsymbol{x}^*$

1.2.1 Minimizing a Quadratic with Linear Constraints

We will move that example from the plane $\boldsymbol{R}^2$ to the space $\boldsymbol{R}^n$. Instead of one constraint on x we have bmm**constraints**$\boldsymbol{A^T x = b}$. **The matrix** $\boldsymbol{A^T}$ will be $\boldsymbol{m}$ by $\boldsymbol{n}$. There will be m Lagrange multipliers $\lambda_1, ..., \lambda_m$: one for each constraint. The cost function $F(\boldsymbol{x}) = \frac{1}{2}\boldsymbol{x}^T S \boldsymbol{x}$ allows any symmetric positive definite matrix S.

Problem : Minimize

$$\boldsymbol{F} = \frac{\mathbf{1}}{\mathbf{2}}\boldsymbol{x^T S x} \ \ \textbf{subject to} \ \ \boldsymbol{A^T x} = \boldsymbol{b}. \tag{5}$$

With m constraints there will be m Lagrange multipliers $\boldsymbol{\lambda} = (\lambda_1, ..., \lambda_m)$. They build the constraints $A^T\boldsymbol{x} = \boldsymbol{b}$ into the Lagrangian $L(\boldsymbol{x}, \boldsymbol{\lambda}) = \frac{1}{2}\boldsymbol{x^T S x} + \boldsymbol{\lambda}^T(A^T\boldsymbol{x} - \boldsymbol{b})$. The $n+m$ derivatives of L give $n+m$ equations for a vector $\boldsymbol{x}$ in $\boldsymbol{R}^n$ and $\boldsymbol{\lambda}$ in $\boldsymbol{R}^m$:

$$\boxed{\begin{array}{ll} \text{x-derivatives of L:} & \text{S}\boldsymbol{x} + A\boldsymbol{\lambda} = \mathbf{0} \\ \lambda\text{-derivatives of L:} & A^T\boldsymbol{x} = \boldsymbol{b} \end{array}} \tag{6}$$

The first equations give $\boldsymbol{x} = -S^{-1}A\boldsymbol{\lambda}$. Then second equations give $-A^T S^{-1} A\boldsymbol{\lambda} = \boldsymbol{b}$. This determines the optimal $\boldsymbol{\lambda}^*$ and therefore the optimal $\boldsymbol{x}^*$:

$$\textbf{Solution } \boldsymbol{\lambda}^*, \boldsymbol{x}^* \qquad \boldsymbol{\lambda}^* = -(A^T S^{-1} A)^{-1}\boldsymbol{b} \qquad \boldsymbol{x}^* = S^{-1}A(A^T S^{-1} A)^{-1}\boldsymbol{b} \tag{7}$$

Minimum cost $F^* = \frac{1}{2}(\boldsymbol{x}^*)^T S\boldsymbol{x}^* = \frac{1}{2}\boldsymbol{b}^T(A^T S^{-1}A)^{-1}A^T S^{-1} S S^{-1} A(A^T S^{-1} A)^{-1}\boldsymbol{b}$

This simplifies a lot!

$$\boxed{\begin{array}{c} \textbf{Minimum cost F}^* = \frac{\mathbf{1}}{\mathbf{2}}\boldsymbol{b^T(A^T S^{-1} A)^{-1} b} \\ \textbf{Gradient of cost } \frac{\partial \boldsymbol{F}^*}{\partial \boldsymbol{b}} = \boldsymbol{(A^T S^{-1} A)^{-1} b} = -\boldsymbol{\lambda}^* \end{array}} \tag{8}$$

This is truly a model problem. When the constraint changes to inequality $A^T\boldsymbol{x} \leq \boldsymbol{b}$, the multipliers become $\lambda_i \geq 0$ and the problem becomes harder.

May I return to the "saddle point matrix" or "KKT matrix" in equation (6):

$$\boldsymbol{M}\begin{bmatrix}\boldsymbol{x}\\ \boldsymbol{\lambda}\end{bmatrix} = \begin{bmatrix}\boldsymbol{S} & \boldsymbol{A}\\ \boldsymbol{A^T} & \mathbf{0}\end{bmatrix}\begin{bmatrix}\boldsymbol{x}\\ \boldsymbol{\lambda}\end{bmatrix} = \begin{bmatrix}\mathbf{0}\\ \boldsymbol{b}\end{bmatrix} \tag{9}$$

That matridx M is not positive definite or negative definite. Suppose you multiply the first block row $[SA]$ by $A^T S^{-1}$ to get $[A^T A^T S^{-1} A]$. Subtract from the second block row to see a zero block:

$$\begin{bmatrix}\boldsymbol{S} & \boldsymbol{A}\\ \mathbf{0} & -\boldsymbol{A^T S^{-1} A}\end{bmatrix}\begin{bmatrix}\boldsymbol{x}\\ \boldsymbol{\lambda}\end{bmatrix} = \begin{bmatrix}\mathbf{0}\\ \boldsymbol{b}\end{bmatrix} \tag{10}$$

This is just elimination on the 2 by 2 block matrix M. That new block in the $2,2$ position is called the **Schur complement**(named after the greatest linear algebraist of all time).

We reached the same equation $-A^TS^{-1}A\boldsymbol{\lambda} = \boldsymbol{b}$ as before. Elimination is just an organized way to solve linear equations. The first n pivots were positive because S is **positive** definite. Now there will be m negative pivots because $-A^TS^{-1}A$ is **negative** definite. This is the unmistakable sign of a **saddle point in the Lagrangian $\boldsymbol{L(x,\lambda)}$**.

That function $L = \frac{1}{2}\boldsymbol{x^TSx} + \boldsymbol{\lambda^T}(\boldsymbol{A^Tx - b})$ is **convex in x and concave in $\boldsymbol{\lambda}$!**

1.2.2 Minimax = Maximum

There is more to learn from this problem. The $\boldsymbol{x}$-derivative of L and the $\boldsymbol{\lambda}$-derivative of L were set to zero in equation (6). We solved those two equations for $\boldsymbol{x}^*$ and $\boldsymbol{\lambda}^*$. That pair ($\boldsymbol{x}^*$,$\boldsymbol{\lambda}^*$) is a saddle point of L in equation (7). By solving (7) we found the minimum cost and its derivative in (8).

Suppose we separate this into two problems: a minimum and maximum problem. First minimize $L(\boldsymbol{x,\lambda})$ for each fixed λ. The minimizing $\boldsymbol{x}^*$ depends on $\boldsymbol{\lambda}$. Then find the $\boldsymbol{\lambda}^8$ that maximizes $L(\boldsymbol{x}^*(\boldsymbol{\lambda}),\boldsymbol{\lambda})$.

Minimize L at $\boldsymbol{x}^* = -S^{-1}A\boldsymbol{\lambda}$ At that point $\boldsymbol{x}^*$, min $L = -\frac{1}{2}\boldsymbol{\lambda^T}A^TS^{-1}A\boldsymbol{\lambda} - \boldsymbol{\lambda^Tb}$

Maximize that minimum $\boldsymbol{\lambda}^* = -(A^TS^{-1}A)^{-1}\boldsymbol{b}$ gives $L = \frac{1}{2}\boldsymbol{b^T}(A^TS^{-1}A)^{-1}\boldsymbol{b}$

$$\boxed{\max_{\boldsymbol{\lambda}} \quad \min_{\boldsymbol{x}} \quad L = \frac{1}{2}\boldsymbol{b}^T(A^TS^{-1}A)\boldsymbol{b}}$$

This *maximum* was $\boldsymbol{x}$ first and $\boldsymbol{\lambda}$ second. The reverse order is *minimax*: $\boldsymbol{\lambda}$ first, $\boldsymbol{x}$ second.

The maximum over $\boldsymbol{\lambda}$ of $L(\boldsymbol{x},\boldsymbol{\lambda}) = \frac{1}{2}\boldsymbol{x}^T\boldsymbol{Sx}+\boldsymbol{\lambda}^T(A^T\boldsymbol{x}-\boldsymbol{b})$ is $\begin{cases} +\infty \text{ if } A^T\boldsymbol{x} \neq \boldsymbol{b} \\ \frac{1}{2}\boldsymbol{x^T Sx} \text{ if } A^T\boldsymbol{x} = \boldsymbol{b} \end{cases}$

The minimum over $\boldsymbol{x}$ of that maximum over $\boldsymbol{\lambda}$ is our answer $\frac{1}{2}\boldsymbol{b}^T(A^TS^{-1}A)^{-1}\boldsymbol{b}$.

$$\boxed{\max_{\boldsymbol{x}} \quad \min_{\boldsymbol{\lambda}} \quad L = \frac{1}{2}\boldsymbol{b}^T(A^TS^{-1}A)^{-1}\boldsymbol{b}}$$

At the saddle point $(\mathbf{x}^*, \boldsymbol{\lambda}^*)$ **we have** $\frac{\partial \boldsymbol{L}}{\partial \boldsymbol{x}} = \frac{\partial \boldsymbol{L}}{\partial \boldsymbol{\lambda}} = \mathbf{0}$ **and** $\max_{\boldsymbol{\lambda}} \min_{\boldsymbol{x}} \boldsymbol{L} = \min_{\boldsymbol{x}} \max_{\boldsymbol{\lambda}} \boldsymbol{L}$.

1.2.3 Dual problems in Science and Engineering

Minimizing a quadratic $\frac{1}{2}x^TSx$ with a linear constraint $A^T\boldsymbol{x} = \boldsymbol{b}$ is not just an abstract exercise. It is the central problem in physical applied mathematics–when a linear differential equation is made discrete. Here are two major examples.

1 Network equations for electrical circuits

Unknowns: Voltages at nodes, currents along edges

Equations: Kirchhoff's Laws and Ohm's Law

Matrices: ATS-1A is the conductance matrix.

2 Finite element method for structures

Unknowns: Displacements at nodes, stresses in the structure

Equations: Balance of forces and stress-strain relations

Matrices: $A^TS^{-1}A$ is the **stiffness matrix**.

The full list would extend to every field of engineering. The stiffness matrix and the conductance matrix are symmetric positive definite. Normally the constraints are equations and not inequalities. Then mathematics offers three approaches to the modeling of the physical problem:

(i) Linear equations with the stiffness matrix or conductance matrix or system matrix

(ii) Minimization with currents or stresses as the unknowns $\boldsymbol{x}$

(iii) Maximization with voltages or displacements as the unknowns λ

In the end, the linear equations (i) are the popular choice. We reduce equation (9) to equation (10). Those network equations account for Kirchhoff and Ohm together. The structure equations account for force balance and properties of the material. All electrical and mechanical laws are built into the final system.

For problems of fluid flow, that system of equations is often in its saddle point form. The unknowns $\boldsymbol{x}$ and $\boldsymbol{\lambda}$ are velocities and pressures. The numerical analysis is well described in *Finite Elements* and *Fast Iterative Solvers by* Elman, Silvester, and Wathen.

For network equations and finite element equations leading to conductance matrices and stiffness matrices A^TCA, one reference is my textbook on *Computational Science and Engineering.*

In statistics and least squares (linear regression), the matrix $A^T\sum^{-1}A$ includes $\sum$ = covariance matrix. We divide by variances $\sum^2$ to whiten the noise.

For nonlinear problems, the energy is no longer a quadratic $\frac{1}{2}x^TSx$. *Geometric non- linearities* appear in the matrix A. *Material nonlinearities* (usually simpler) appear in the matrix C. Large displacements and large stresses are a typical source of nonlinearity.

1.3 Linear Programming, Game Theory and Duality

This section is about highly structured optimization problems. Linear programming comes first-linear cost and linear constraints (including inequalities). It was also historically first, when Dantzig invented the simplex algorithm to find the optimal solution. Our approach here will be to see the ”duality” between a minimum problem and a maximum– *two linear programs that are solved at the same time.*

An inequality constraint $\boldsymbol{x_k} \geq 0$ has two states—active and inactive. If the minimizing solution ends up with $x_k^* > 0$, then that requirement was inactive–it didn't change anything. Its Lagrange multiplier will have $x_k^* = 0$. The minimum cost is not affected by that constraint on x_k. But if the constraint $x_k > 0$. But if the constraint $x_k \geq 0$ actively forces the best $\boldsymbol{x}^*$ to have $x_k^* = 0$, then the multiplier will have $\lambda_k^* > 0$. So the optimality condition is $\boldsymbol{x_k^* \lambda_k} = 0$ for each k.

One more point about linear programming. It solves all **2-person zero sum games**. Profit to one player is loss to the other player. The optimal strategies produce a saddle point.

Inequality constraints are still present in quadratic programming (**QP**) and semidefinite programming (**SDP**). The constraints in SDP involve symmetric matrices. The inequality $S \geq 0$ means that the matrix is positive semidefinite (or definite). If the best S is actually positive definite, then the constraint $S \geq 0$ was not active and the Lagrange multiplier (now also a matrix) will be zero.

1.3.1 Linear Programming

Linear programming starts with a cost vector $\boldsymbol{c} = (c_1, ..., c_n)$. The problem is to minimize the cost $F(\boldsymbol{x}) = c_1x_1, ... + c_nx_n = \boldsymbol{c^T x}$. The constraint are m linear equations $\boldsymbol{Ax = b}$ and n inequalities $x_1 \geq 0, ..., x_n \geq 0$. We just write $\boldsymbol{x} \geq 0$ to include all n components:

$$\textbf{Linear Program} \quad \boxed{\textbf{Minimize } \boldsymbol{c^T x} \textbf{ subject to } \boldsymbol{Ax = b} \textbf{ and } \boldsymbol{x \geq o}} \quad (1)$$

If A is 1 by 3, $\boldsymbol{Ax = b}$ gives a plane like $x_1 + x_2 + 2x_3 = 4$ in 3-dimensional space. That plane will be chopped off by the constraints $x_1 \geq 0, x_2 \geq 0, x_3 \geq 0$. This leaves a triangle on a plane, with corners at $(x_1, x_2, x_3) = (4, 0, 0)$ and $(0, 0, 2)$. Our problem is to find the point $\boldsymbol{x}^*$ in this triangle that minimizes the cost $c^T x$.

Because the cost is linear, **its minimum will be reached at one of those corners**. Linear programming has to find that minimum cost corner. Computing all corners is exponentially impractical when m and n are large. So the *simplex method* finds one starting corner that satisfies $\boldsymbol{Ax = b}$ and $\boldsymbol{x \geq 0}$. Then it moves along an edge of the constraint set K to another (lower cost) corner. The cost $\boldsymbol{c^T x}$ drops at every step.

It is a linear algebra problem to find the steepest edge and the next corner (where that edge ends). The simplex method repeats this step many times, from corner to corner.

New starting corner, new steepest edge, new lower cost corner in the simplex method. In practice the number of steps is polynomial (but in theory it could be exponential).

Our interest here is to identify the **dual problem**–a maximum problem for $\boldsymbol{y}$ in $\boldsymbol{R}^m$.

It is standard to use $\boldsymbol{y}$ instead of $\boldsymbol{\lambda}$ for the dual unknowns–the Lagrange multipliers.

$$\textbf{Dual Problem} \quad \boxed{\textbf{Maximize } \boldsymbol{y^T b} \textbf{ subject to } \boldsymbol{A^T y \le c}} \tag{2}$$

This is another linear program for the simplex method to solve. It has the same inputs $\boldsymbol{A, b, c}$ as before. When the matrix $\boldsymbol{A}$ is m by n, the matrix A^T is n by m. So $A^T y \le c$ has n constraints. A beautiful fact: $\boldsymbol{y^T b}$ in the maximum problem is never larger than $\boldsymbol{c^T x}$ in the minimum problem.

Weak Duality

$$\boldsymbol{y^T b = y^T (Ax) = (A^T y)^T x \le c^T x} \quad \textbf{Maximum (2)} \le \textbf{minimum (1)}$$

Maximizing pushes $\boldsymbol{y^T}$ upward. Minimizing pushes $\boldsymbol{c^T x}$ downward. The great duality theorem (**the minimax theorem**) says that they meet at the best $\boldsymbol{x}^*$ and the best $\boldsymbol{y}^*$

$$\boxed{\textbf{Duality} \qquad \textbf{The maximum of } \boldsymbol{y^T b} \textbf{ equals the minimum of } \boldsymbol{c^T x}.}$$

The simplex method will solve both problems at once. For many years that method had no competition. Now this has changed. Newer algorithms go directly *through* the set allowed $x's$ instead of traveling around its edges (from corner to corner).

Interior point methods are competitive because they can use calculus to achieve steepest descent.

The situation right now is that either method could win–along the edges or inside.

1.3.2 Max Flow-min Cut

Here is a special linear program. The matrix A will be the incidence matrix of a graph. That means that flow in equals flow out at every node. Each edge of the graph has a *capacity*M_j–which the flow y_j along that edge cannot exceed.

The maximum problem is to send the greatest possible flow from the source node s to the sink node t. This flow is returned from t to s on a special edge with unlimited capacity—drawn on the next page. The constraints on y are Kirchhoff's Current Law $\boldsymbol{A^T y} = 0$ and the capacity bounds $|y_j| \leq M_j$ on each edge of the graph. The beauty of this example is that you can solve it by common sense (for a small graph). In the process, you discover and solve the dual minimum problem, which is **min cut**.

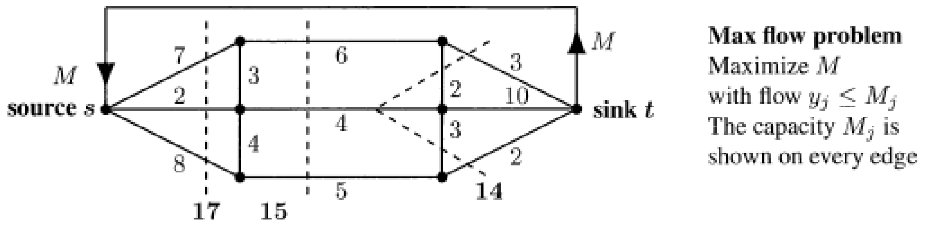


Figure 1.6: The max flow M is bounded by the capacity of any cut (dotted line). *By duality, the capacity of the* ***minimum cut*** *equals the* ***maximum flow:*** $M = \mathbf{14}$

Begin by sending flow out of the source. The three edges from s have capacity $7 + 2 + 8 = \mathbf{17}$. Is there a tighter bound than $M \leq 17$?

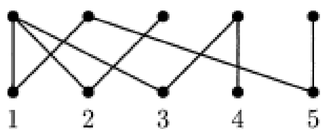
Yes, a cut through the three middle edges only has capacity $6 + 4 + 5 = \mathbf{15}$. Therefore 17 cannot reach the sink. Is there a tighter bound than $M \leq 15$?

Yes, a cut through five later edges only has capacity $3 + 2 + 4 + 3 + 2 = \mathbf{14}$. The total flow M cannot exceed 14. Is that flow of 14 achievable and is this the tightest cut?

Yes, this is the min cut (*it is an l1 problem!*) and by duality 14 is the max flow.

Wikipedia shows a list of faster and faster algorithms to solve this important problem. It has many applications. If the capacities M_j are integers, the optimal flows y_j, are integers. Normally integer programs are extra difficult, but not here.

A special max flow problem has all capacities $M_j = 1$ or 0. The graph is **bipartite** (all edges go from a node in part 1 to a node in part 2). We are matching people in part 1 to jobs in part 2 (at most one person per job and one job per person). Then the maximum matching is $M = max$ flow in the graph = max number of assignments.



This bipartite graph allows a perfect matching: $M = 5$. Remove the edge from 2 down to 1. Now only $M = 4$ assignments are possible, because 2 and 5 will only be qualified for one job (5).

For bipartite graphs, max flow = min cut is König's theorem and Hall's marriage theorem.

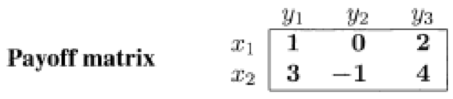
Figure 1.7

1.3.3 Two Person Games

Games with three or more players are very difficult to solve. Groups of players can combine against the others, and those alliances are unstable. New teams will often form. It took John Nash to make good progress, leading to his Nobel Prize (in economics!). But two-person zero-sum games were completely solved by von Neumann. We will see their close connection to linear programming and duality.

The players are X and Y. There is a payoff matrix A. At every turn, player X chooses a row of A and player Y chooses a column. The number in that row and column of A is the payoff, and then the players take another turn.

To match with linear programming. I will make the payment fo from player X to Y. Then X wants to minimize and Y wants to maximize. Here is a very small payoff matrix. It has two rows for X to choose and three columns for Y.



Payoff matrix

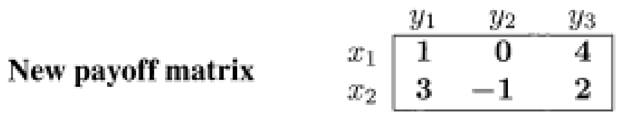
$$\begin{array}{c|ccc|} & y_1 & y_2 & y_3 \\ \hline x_1 & \mathbf{1} & \mathbf{0} & \mathbf{2} \\ x_2 & \mathbf{3} & \mathbf{-1} & \mathbf{4} \\ \hline \end{array}$$

Y likes those large numbers in column 3. X sees that the smallest number in that column is 2 (in row 1). Both players have no reason to move from this simple strategy of column 3 for Y and row 1 for X. The payoff of 2 is from X to Y:

2 is smallest in its column and largest in its row

This is a saddle point. Y cannot expect to win more than 2. X cannot expect to lose less than 2. Every play of the game will be the same because no player has an incentive to change. The optimal strategies $\boldsymbol{x}^*$ and $\boldsymbol{y}^*$ are clear: row 1 for X and column 3 for Y.

But a change in column 3 will require new thinking by both players.



New payoff matrix

$$\begin{array}{c|ccc|} & y_1 & y_2 & y_3 \\ \hline x_1 & \mathbf{1} & \mathbf{0} & \mathbf{4} \\ x_2 & \mathbf{3} & \mathbf{-1} & \mathbf{2} \\ \hline \end{array}$$

X likes those small and favorable numbers in column 2. But Y will never choose that column. Column 3 looks best (biggest) for Y, and X should counter by choosing row 2 (to avoid paying 4). But then column 1 becomes better than column 3 for Y, because winning 3 in column 1 is better than winning 2.

You are seeing that Y still wants column 3 but must go sometimes to column 1. Sim- ilarly X must have a *mixed strategy*: choose rows 1 and 2 with probabilities $x_1 and x_2$. The choice at each turn must be unpredictable, or the other player will take advantage. So the decision for X is two probabilities $x_1 \geq 0$ and $x_2 \geq 0$ that add to $x_1 + x_2 = 1$. The payoff matrix has a new row from this mixed strategy:

row 1	1	0	4
row 2	3	-1	2
$\boldsymbol{x_1}$**(row 1)** $+ \boldsymbol{x_2}$**(row 2)**	$\boldsymbol{x_1 + 3x_2}$	$\boldsymbol{-x_2}$	$\boldsymbol{4x_1 + 2x_2}$

X will choose fractions x_1 and x_2 to make the worst (*largest*) payoff as small as possible. Remembering $x_2 = 1 - x_1$, this will happen when the two largest payoffs are equal:

$\boldsymbol{x_1 + 3x_2 = 4x_1 + 2x_2}$ **means** $\boldsymbol{x_1 + 3(1 - x_1) = 4x_1 + 2(1 - x_1)}$

That equation gives $\boldsymbol{x_1^* = \frac{1}{4}}$ **and** $\boldsymbol{x_1^* = \frac{3}{4}}$. **The new mixed row is** $\boldsymbol{2.5, -.75, 2.5.}$

Similarly Y will choose columns 1,2,3 with probabilities y_1, y, y_3. Again they add to 1. That mixed strategy combines the three columns of A into a new column for Y.

column 1	column 2	column 3	**mix 1, 2, 3**
1	0	4	$\boldsymbol{y_1 + 4y_3}$
3	-1	2	$\boldsymbol{3y_1 - y_2 + 2y_3}$

Y will choose the fractions $y_1 + y_2 + y_3 = 1$ to make the worst (*smallest*) payoff as large as possible. That happens when $y_2 = 0$ and $y_3 = 1 - y_1$. The two mixed payoffs are equal:

$y_1 + 4(1 - y_1) = 3y_1 + 2(1 - y_1)$ gives $-3y_1 + 4 = y_1 + 2$ and $\boldsymbol{y_1^* = y_3^* = \frac{1}{2}}$ **The new mixed columns has 2.5 in both components. These optimal strategies identify 2.5 as the value of the game.** With the mixed strategy $x_1^* = \frac{1}{4}$ and $x_2^* = \frac{3}{4}$, Player X can guarantee to pay no more than 2.5. Player Y can guarantee to receive no less than 2.5. We have found the saddle point (best mixed strategies, with minimax payoff from X = maximum payoff to Y = **2.5**) of this two-person game.

	y_1	y_2	y_3	$\frac{1}{2}$**col 1** + $\frac{1}{2}$**col 2**
row 1	1	0	4	**2.5**
row 2	3	−1	2	**2.5**
$\frac{1}{4}$ **row 1** + $\frac{3}{4}$**row 2**	**2.5**	−.75	**2.5**	

Von Neumann's **minimax theorem** for games gives a solution for every payoff matrix. It is equivalent to the duality theorem *min* $\boldsymbol{c^T x}$ = *max* $\boldsymbol{y^T b}$ for linear programming.

1.4 Gradient Descent Toward the Minimum

This section of the course is about a fundamental problem: **Minimize a function** $\boldsymbol{f(x_1, ..., x_n)}$. Calculus teaches us that all the first derivatives $\partial f / \partial x_i$, are zero at the minimum (when f is smooth). If we have $n = 20$ unknowns (a small number in deep learning) then minimizing one function f produces 20 equations $\partial f / \partial x_i = 0$. *"Gradient descent" uses the derivatives $\partial f / \partial x_i$ to find a direction that reduces f($\boldsymbol{x}$).* The steepest direction, in which $f(\boldsymbol{x})$ decreases fastest, is given by the gradient $\boldsymbol{-\nabla f}$ **:**

$$\boxed{\textbf{Gradient descent} \quad \boldsymbol{x_{k+1} = x_k - s_k \nabla f(x_k)}} \qquad (1)$$

The symbol $\boldsymbol{\nabla f}$ represents the vector of n partial derivatives of f : its **gradient**. So (1) is a vector equation for each step $k = 1, 2, 3, ...$ and s_k is the *stepsize* or the *learning rate*. We hope to move toward the point $\boldsymbol{x}^*$ where the graph of $f(\boldsymbol{x})$ hits bottom.

We are willing to assume for now that 20 first derivatives exist and can be computed. We are not willing to assume that those 20 functions also have 20 convenient derivatives $\partial / \partial x_j (\partial f / \partial x_i)$. Those are the 210 **second derivatives of** f-which go into a 20 by 20 symmetric matrix H. (Symmetry reduces $n^2 = 400$ to $\frac{1}{2}n^2 + \frac{1}{2}n = 210$ computations.) The second derivatives would be very useful extra information, but in many problems we have to go without.

You should know that 20 first derivatives and 210 second derivatives don't multiply the computing cost by 20 and 210. The neat idea of **automatic differentiation**–rediscovered and extended as **backpropagation** in machine learning—makes those cost factors much smaller in practice. This idea is described in early section.

Return for a moment to equation (1). The step $-s_k \boldsymbol{\nabla f(x_k)}$ includes a minus sign (to descend) and a factor s_k (to control the the stepsize) and the gradient vector $\boldsymbol{\nabla f}$ (containing the first derivatives of f computed at the current point $\boldsymbol{x_k}$). A lot of thought and computational experience has gone into the choice of stepsize and search direction. We start with the main facts from calculus about derivatives and gradient vectors $\boldsymbol{\nabla f}$.

1.4.1 The Derivative of $f(x)$: $n = 1$

The derivative of $f(x)$ involves a *limit*—this is the key difference between calculus and algebra. We are comparing the values of f at two nearby points x and $x + \Delta x$, as Δx approaches zero. More accurately, we are watching the slope $\Delta f/\Delta x$ between two points on the graph of $f(x)$:

$$\textbf{Derivative of } \boldsymbol{f} \textbf{ at } \boldsymbol{x} \quad \boldsymbol{\frac{df}{dx}} = \textbf{limit of } \left[\boldsymbol{\frac{f(x+\Delta x)-f(x)}{\Delta x}}\right] \tag{2}$$

This is a forward difference when Δx is positive and a backward difference when $\Delta x < 0$. When we have the same limit from both sides, that number is the slope of the graph at x.

The ramp function $\text{ReLU}(x) = f(x) = \max(0, a)$ is heavily involved in deep learning in next chapter . It has unequal slopes $\mathbf{1}$ to the right and $\mathbf{0}$ to the left of $x = 0$. So the derivative df/dx does not exist at that corner point in the graph. For $n = 1$, df/dx is the gradient $\boldsymbol{\nabla f}$.

$$\text{ReLU} = \begin{matrix} \boldsymbol{x} \text{ for } x \geq 0 \\ \mathbf{0} \text{ for } x \leq 0 \end{matrix} \qquad \textbf{Slope } \frac{\Delta f}{\Delta x} = \frac{f(0+\Delta x)-f(0)}{\Delta x} = \begin{matrix} \boldsymbol{\Delta x/\Delta x = 1} \text{ if } \Delta x > 0 \\ \boldsymbol{0/\Delta x = 0} \text{ if} < 0 \end{matrix}$$

For the smooth function $\boldsymbol{f(x) = x^2}$, the ratio $\Delta f/\Delta x$ will safely approach the derivative df/dx from both sides. But the approach could be slow (just first order). Look again at the point $x = 0$, where the true derivative $df/dx = 2x$ is now zero:

The ratio $\frac{\Delta f}{\Delta x}$ at $x = 0$ is $\frac{f(\Delta x)-f(0)}{\Delta x} = \frac{(\Delta x)^2-0}{\Delta x} = \boldsymbol{\Delta x}$ Then limit = slope = 0.

In this case and in almost all cases, we get a better ratio (closer to the limiting slope df/dx) by averaging the **forward difference** (where $\Delta x > 0$) with the **backward difference** (where $\Delta x < 0$). The average is the more accurate **centered difference.**

$$\textbf{centered at } \boldsymbol{x} \qquad \frac{1}{2}\left[\frac{f(x+\Delta x)-f(x)}{\Delta x} + \frac{f(x-\Delta x)-f(x)}{-\Delta x} = \boldsymbol{\frac{f(x+\Delta x)-f(x-\Delta x)}{2\Delta x}}\right]$$

For the example $f(x) = x^2$ this centering will produce the exact derivative $df/dx = 2x$. In the picture we are averaging plus and minus slopes to get the correct slope 0 at $x = 0$. For all smooth functions, the centered differences reduce the error to size $(\Delta x)^2$. This is a big improvement over the error of size Δx for uncentered differences $f(x + \Delta x) - f(x)$.

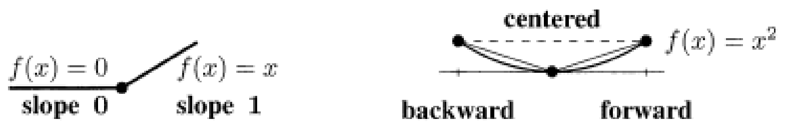


Figure 1.8: ReLU function = ramp from deep learning. centered slope of $f = x^2$ is exact.

Most finite difference approximations are centered for extra accuracy. But we are still dividing by a small number $2\Delta x$. And for a multivariable function $F(x_1, x_2, ..., x_n)$ we will need ratios $\Delta F/\Delta x_i$, in n different directions—possibly for large n. Those ratios approximate the n partial derivatives that go into the **gradient vector grad $\boldsymbol{F} = \boldsymbol{\nabla F}$**.

$$\boxed{\textbf{The gradient of } \boldsymbol{F(x_1, ..., x_n)} \textbf{ is column vector } \boldsymbol{\nabla F = \left(\frac{\partial F}{\partial x_1}, ..., \frac{\partial F}{\partial x_n}\right)}}$$

Its components are the n partial derivatives of F. $\boldsymbol{\nabla} F$ points in the steepest direction.

Examples 1-3 will show the value of vector notation ($\boldsymbol{\nabla F}$is always a column vector).

Example 1 For a constant vector $\boldsymbol{a} = (a_1, ..., a_n), F(\boldsymbol{x}) = \boldsymbol{a}^T\boldsymbol{x}$ has gradient $\boldsymbol{\nabla F = a}$

The partial derivatives of $F = a_1x_1 + ... + a_nx_n$ are the numbers $\partial F/\partial x_k = a_k$

Example 2 For a symmetric matrix S, the gradient pf $F(\boldsymbol{x}) = \boldsymbol{x}^T\boldsymbol{Sx}$ is $\boldsymbol{\nabla F = 2Sx}$. To see this, write out the function $F(x_1, x_2)$ when $n = 2$. The matrix S is 2 by 2:

$$F = \begin{bmatrix} \mathrm{x}_1 & \mathrm{x}_2 \end{bmatrix} \begin{bmatrix} a & b \\ b & c \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{matrix} ax_1^2 + cx_2^2 \\ +2bx_1x_2 \end{matrix} \quad \begin{bmatrix} \partial f/\partial x_1 \\ \partial f/\partial x_2 \end{bmatrix} = 2 \begin{bmatrix} ax_1 + bx_2 \\ bx_1 + cx_2 \end{bmatrix}$$

$$= \boldsymbol{2S} \begin{bmatrix} \boldsymbol{x_1} \\ \boldsymbol{x_2} \end{bmatrix}$$

Example 3 For a positive definite symmetric S, the minimum of a quadratic $F(\boldsymbol{x}) = \boldsymbol{\frac{1}{2}x}^T\boldsymbol{Sx} - \boldsymbol{a}^T\boldsymbol{x}$

$$\boxed{\boldsymbol{\nabla F} = \begin{bmatrix} \boldsymbol{\partial F/\partial x_1} \\ \boldsymbol{\cdot} \\ \boldsymbol{\cdot} \\ \boldsymbol{\partial F/\partial x_n} \end{bmatrix} = \boldsymbol{Sx - a = 0} \text{ at } \boldsymbol{x^* = S^{-1}a = argminF}} \tag{3}$$

As always, **that notation *argmin F* stands for the point $\boldsymbol{x}^*$ where the minimum of $F(\boldsymbol{x}) = \boldsymbol{\frac{1}{2}x}^T\boldsymbol{Sx} - \boldsymbol{a}^T\boldsymbol{x}$ is reached.** Often we are moved interested in this minimizing $\boldsymbol{x}^*$ than in the actual minimum value $F_{min} = F(x^*)$ at that point:

$$\boldsymbol{F_{min}} \ is \ \tfrac{1}{2}(S^{-1}a)^TS(S^{-1}a) - a^T(S^{-1}a) = \tfrac{1}{2}a^TS^{-1}a - a^TS^{-1}a = \boldsymbol{-\tfrac{1}{2}a^TS^{-1}a}$$

The graph of F is a bowl passing through zero at $\boldsymbol{x} = \mathbf{0}$ and dipping to its minimum at $\boldsymbol{x}^*$.

Example 4 The **determinant** $F(x) = det\boldsymbol{X}$ is a function of all n^2 variables x_{ij}. In the formula for $det\boldsymbol{X}$, each x_{ij} along a row is multiplied by its "cofactor" C_{ij}. This cofactor is a determinant of size $n-1$, using all rows of $\boldsymbol{X}$ except row i and all columns except column j–and multiplied by $(-1)^{i+j}$:

The partial derivatives $\frac{\partial(detX)}{\partial x_{ij}} = C_{ij}$ in the matrix of cofactor of X give $\boldsymbol{\nabla F}$.

Example 5 The **logarithm of the determinant** is a most remarkable function:

$\boldsymbol{L(X) = log(detX)}$ has partial derivatives $\frac{\partial L}{\partial x_{ij}} = \frac{C_{ij}}{det\boldsymbol{X}} = \boldsymbol{j, i}$ **entry of** $\boldsymbol{X^{-1}}$

The chain rule for $L = \log F$ is give as $(\partial L/\partial F)(\partial F/\partial x_{ij}) = (1/F)(\partial F/\partial x_{ij}) = (1/\det X)C_{ij}$. Then this ratio of cofactors to determinant gives the j, i entries of the inverse matrix X^{-1}.

It is neat that X^{-1} contains the n^2 first derivatives of $L = logdetX$. The second derivatives of L are remarkable too. We have n^2 variables x_{ij} and n^2 first derivatives in $\boldsymbol{\nabla L} = (X^{-1})^T$. This means n^4 second derivatives! What is amazing is that the matrix of second derivatives is **negative definite** when $X = S$ is symmetric positive definite. So we reverse the sign of L: **positive definite second derivatives** $\longleftrightarrow$ **convex function**.

- log (det S) is a convex function of the entries of the positive definite matrix S.

1.4.2 The Geometry of the Gradient Vector ∇f

Start with a function $f(x, y)$. It has $n = 2$ variables. Its gradient is $\boldsymbol{\nabla f} = (\frac{\boldsymbol{\partial f}}{\boldsymbol{\partial x}}, \frac{\boldsymbol{\partial f}}{\boldsymbol{\partial y}})$. This vector changes length as we move the point x, y where the derivatives are computed:

$\boldsymbol{\nabla f} = (\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y})$ **Length** $= ||\boldsymbol{\nabla f}|| = \sqrt{(\frac{\partial f}{\partial x})^2 + (\frac{\partial f}{\partial y})^2} =$ steepest slope of f

That length $||\boldsymbol{\nabla f}||$ tells us the steepness of the graph of $z = f(x, y)$. The graph is normally a curved surface—like a mountain or a valley in xyz space, At each point there is a slope $\partial f/\partial x$ in the x-direction and a slope $\partial f/\partial y$ in the y-direction. **The steepest slope is in the direction of $\boldsymbol{\nabla f} = \boldsymbol{gradf}$. The magnitude of that steepest slope is** $||\boldsymbol{\nabla f}||$.

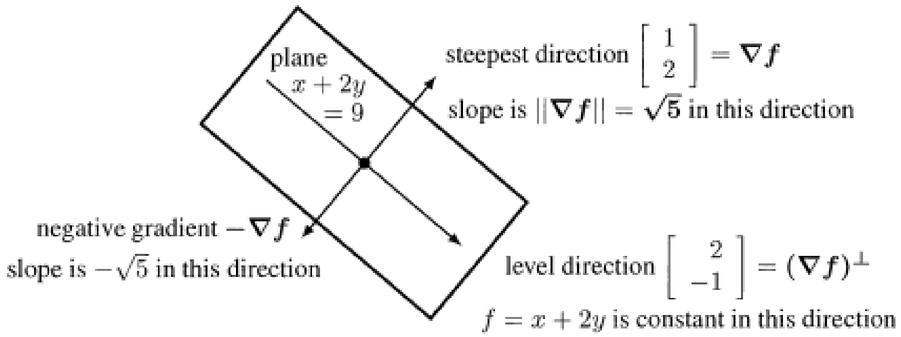

plane
$x + 2y = 9$
steepest direction $\begin{bmatrix} 1 \\ 2 \end{bmatrix} = \boldsymbol{\nabla f}$
slope is $||\boldsymbol{\nabla f}|| = \sqrt{5}$ in this direction
negative gradient $-\boldsymbol{\nabla f}$
slope is $-\sqrt{5}$ in this direction
level direction $\begin{bmatrix} 2 \\ -1 \end{bmatrix} = (\boldsymbol{\nabla f})^{\perp}$
$f = x + 2y$ is constant in this direction

Figure 1.9: The negative gradient $-\boldsymbol{\nabla f}$ gives the direction of steepest descent.

For the nonlinear function $f(x, y) = ax^2 + by^2$, the gradient is $\boldsymbol{\nabla f} = \begin{bmatrix} 2ax \\ 2by \end{bmatrix}$. That tells us the steepest direction, changing from point to point. We are on a curved surface (a bowl opening upward). The bottom of the bowl is at $x = y = 0$ where the gradient vector is zero. The slope in the steepest direction is $||\boldsymbol{\nabla f}||$. At the minimum, $\boldsymbol{\nabla f} = (2ax, 2by) = (0, 0)$ and *slope = zero.*

The level direction has $\boldsymbol{z = ax^2 + by^2}$ = constant height. That plane z = constant cuts through the bowl in a level curve. In this example the level curve $ax^2 + by^2 = c$ is an ellipse. The direction of the ellipse (level direction) is perpendicular to the gradient vector (steepest direction). But there is a serious difficulty for steepest descent:

The steepest direction changes as you go down! The gradient doesn't point to the bottom!

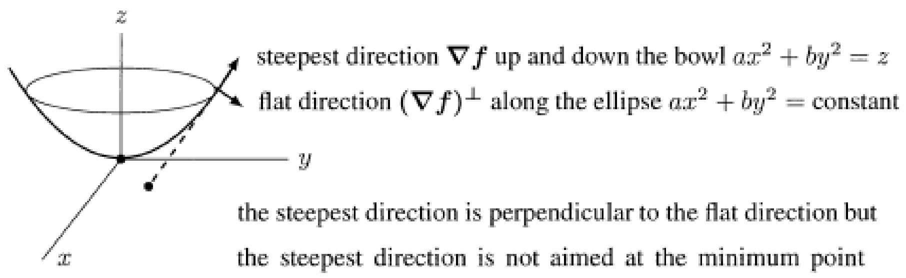

z
steepest direction ∇f up and down the bowl $ax^2 + by^2 = z$
flat direction $(\nabla f)^{\perp}$ along the ellipse $ax^2 + by^2 =$ constant
y
the steepest direction is perpendicular to the flat direction but
x
the steepest direction is not aimed at the minimum point

Figure 1.10: **Steepest descent moves down the bowl in the gradient direction** $\begin{bmatrix} \boldsymbol{-2ax} \\ \boldsymbol{-2by} \end{bmatrix}$

Let me repeat. At the point x_o, y_o the gradient direction for $f = ax^2 + by^2$ is along $\boldsymbol{\nabla f} = (2ax_o, 2by_o)$. The steepest line through x_o, y_o is $2ax_o(y - y_o) = 2by_o(x - x_o)$. But then the lowest point $(x, y) = (0, 0)$ does not lie on the line! **We will not find that minimum point in one step of "gradient descent". The steepest direction does not lead to the bottom of the bowl**–except when $b = a$ and the bowl is circular.

Water changes direction as it goes down a mountain. Sooner or later, we must change direction too. In practice we keep going in the gradient direction and stop when our cost function f is not decreasing quickly. At that point Step 1 ends and we recompute the gradient $\boldsymbol{\nabla f}$. This gives a new descent direction for Step 2.

1.4.3 An Important Example with Zig-Zag

The example $f(x, y) = \boldsymbol{\frac{1}{2}(x^2 + by^2)}$ is extremely useful for $0 < b \le 1$. Its gradient $\boldsymbol{\nabla f}$ has two components $\partial f/\partial x = \boldsymbol{x}$ and $\partial f/\partial y = \boldsymbol{by}$. The minimum value of f is zero. That minimum is reached at the point $(x^*, y^*) = (0, 0)$. Best of all, steepest descent with exact line search produces a simple formula for each point (x_k, y_k) in the slow progress down the bowl toward $(0, 0)$. Starting from $(x_o, y_o) = (b, 1)$ we find these points:

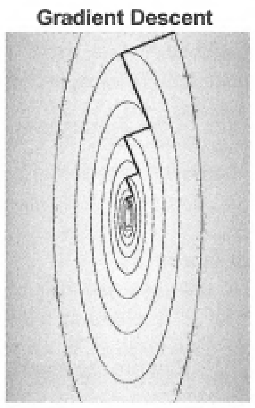
$$\boxed{\boldsymbol{x_k = b\left(\tfrac{b-1}{b+1}\right)^k \quad y_k = \left(\tfrac{1-b}{1+b}\right)^k \quad f(x_k, y_k) = \left(\tfrac{1-b}{1+b}\right)^{2k} f(x_o, y_o)}} \qquad (4)$$

If $b = 1$, you see immediate success in one step. The point (x_1, y_1) is $(0, 0)$. The bowl is perfectly circular with $f = \frac{1}{2}(x^2+y^2)$. The negative gradient direction goes exactly through $(0, 0)$. Then the first step of gradient descent finds that correct minimizing point where $f = 0$.

The real purpose of this example is seen when b is small. The crucial ratio in equation (4) is $r = (b-1)/(b+1)$. For $b = \frac{1}{10}$ this ratio is $r = -9/11$. For $b = \frac{1}{100}$ the ratio is $-99/101$. The ratio is approaching -1 and the progress toward $(0, 0)$ has virtually stopped when b is very small.

Figure given below shows the frustrating zig-zag pattern of the steps toward $(0, 0)$. Every step is short and progress is very slow. This is a case where the stepsize s_k in $\boldsymbol{x}_{k+1} = \boldsymbol{x}_k - s_k \boldsymbol{\nabla f}(\boldsymbol{x}_k)$ was exactly chosen to minimize f (an exact line search). But the direction of $\boldsymbol{-\nabla f}$, even if steepest, is pointing far from the final answer $(x^*, y^*) = (0, 0)$.

The bowl has become a narrow valley when b is small, and we are uselessly crossing the valley instead of moving down the valley to the bottom.



The first descent step starts out perpendicular to the level set. As it crosses through lower level sets, the function $f(x, y)$ is decreasing. **Eventually its path is tangent to a level set L.** Descent has stopped. Going further will increase f. The first step ends. The next step is perpendicular to L. So the zig-zag path took a 90° turn.

Figure 1.11: Slow convergence on a zig-zag path to the minimum of $f = x^2 + by^2$.

For b close to 1, this gradient descent is faster. First-order convergence means that the distance to $(x^*, y^*) = (0, 0)$ is reduced by the constant factor $(1-b)/(1+b)$ at every step. The following analysis will show that linear convergence extends to all strongly convex functions f–first when each line search is exact, and then (more realistically) when the search at each step is close to exact.

Machine learning often uses **stochastic gradient descent**. The next section will describe this variation (especially useful when n is large and $\boldsymbol{x}$ has many components). And we recall that **Newton's method** uses second derivatives to produce quadratic convergence—the reduction factor $(1-b)/(1+b)$ drops to zero and the error is squared at every step. (Our model problem of minimizing a quadratic $\frac{1}{2}\boldsymbol{x}^T\boldsymbol{S}\boldsymbol{x}$ is solved in one step.) This is a gold standard that approximation algorithms don't often reach in practice.

1.4.4 Convergence Analysis for Steepest Descent

On this page we are following the presentation by Boyd and Vandenberghe in *Convex Optimization* (published by Cambridge University Press). From the specific choice of $f(x,y) = \frac{1}{2}(x^2+by^2)$, we move to any strongly convex $f(\boldsymbol{x})$ in n dimensions. Convexity is tested by the positive definiteness of the symmetric matrix $H = \boldsymbol{\nabla^2 f}$ of second derivatives (the Hessian matrix). In one dimension this is the number d^2f/dx^2:

Strongly convex **$\boldsymbol{m > 0}$** $\quad \boldsymbol{H_{ij}} = \frac{\boldsymbol{\partial^2 f}}{\boldsymbol{\partial x_i \partial x_j}}$ has eigenvalue between $\boldsymbol{m \le \lambda \le M}$ at all $\boldsymbol{x}$

The quadratic $f = \frac{1}{2}(x^2+by^2)$ has second derivatives 1 and b. The mixed derivative $\partial^2 f/\partial x \partial y$ is zero. So the matrix is diagonal and its two eigenvalues are $m = b$ and $M = 1$. We will now see that **the ratio $\boldsymbol{m/M}$ controls the speed of steepest descent**.

The gradient descent step is $x_k + 1 = x_k - s\boldsymbol{\nabla f_k}$. We estimate f by its Taylor series:

$$f(\boldsymbol{x}_{k+1}) \le f(\boldsymbol{x}_k) + \boldsymbol{\nabla f}^T(\boldsymbol{x}_{k+1}-\boldsymbol{x}_k) + \frac{M}{2}||\boldsymbol{x}_{k+1}-\boldsymbol{x}_k||^2 \tag{5}$$

$$= f(\boldsymbol{x}_k) - s||\boldsymbol{\nabla f}||^2 + \frac{Ms^2}{2}||\boldsymbol{\nabla f}||^2 \tag{6}$$

The best s minimizes the left side (exact line search). The minimum of the right side is at $s = 1/M$. Substituting that number for s, the next point $\boldsymbol{x}_{k+1}$ has

$$f(\boldsymbol{x}_{k+1}) \le f(\boldsymbol{x}_k) - \frac{1}{2m}||\boldsymbol{\nabla f(x_k)}||^2 \tag{7}$$

A parallel argument uses m instead of M to reverse the inequality sign in (5).

$$f(\boldsymbol{x}^*) \geq f(\boldsymbol{x}_k) - \frac{1}{2m}||\boldsymbol{\nabla f(x_k)}||^2 \tag{8}$$

Multiply (7) by M and (8) by m and subtract to remove $||\boldsymbol{\nabla f(x_k)}||^2$. Rewrite the result as

$$\textbf{Steady drop in } \boldsymbol{f} \quad \boxed{f(\boldsymbol{x}_{k+1}) - f(\boldsymbol{x}^*) \leq (1 - \tfrac{m}{M})(f(\boldsymbol{x}_k) - f(\boldsymbol{x}^*))} \tag{9}$$

This says that every step reduces the height above the bottom of the valley by at least $c = 1 - \frac{m}{M}$. That is **linear convergence: very slow when $\boldsymbol{b = m/M}$ is small**.

Our zig-zag example had m =b and $M = 1$. The estimate (9) guarantees that the height $f(\boldsymbol{x}_k)$ above $f(\boldsymbol{x}^*) = 0$ is reduced by at least $1 - b$. The exact formula in that totally computable problem produced the reduction factor $(1 - b)^2/(1 + b)^2$. When b is small this is about $1 - 4b$. So the actual improvement was only 4 times better than the rough estimate $1 - b$ in (9). **This gives us considerable faith that (9) is realistic**.

1.4.5 Momentum and the Path of a Heavy Ball

The slow zig-zag path of steepest descent is a real problem. We have to improve it. Our model example $f = \frac{1}{2}(x^2 + by^2)$ has only two variables x, y and its second derivative matrix H is diagonal–constant entries $f_{xx} = 1$ and $f_{yy} = b$. But it shows the zig-zag problem very clearly when $\boldsymbol{b = \lambda_m in/\lambda_m ax = m/M}$ **is small**.

Key idea: Zig-zag would not happen for a heavy ball rolling downhill. Its momentum carries it through the narrow valley–bumping the sides but moving mostly forward. So we **add momentum with coefficient $\boldsymbol{\beta}$ to the gradient** (Polyak's important idea). This gives one of the most convenient and powerful ideas in deep learning.

The direction $\boldsymbol{z}_k$ of the new step remembers the previous direction $z_k - 1$.

$$\textbf{Descent with momentum} \quad \boxed{\boldsymbol{x_{k+1} = x_k - s z_k} \text{ with } \boldsymbol{z_k = \nabla f(x_k) + \beta z_{k-1}}} \tag{10}$$

Now we have two coefficients to choose–the stepsize $\boldsymbol{s}$ and also $\boldsymbol{\beta}$. Most important, **the step to $\boldsymbol{x}_{k+1}$ in equation (10) involves** z_{k-1}. Momentum has turned a one-step method (gradient descent) into a two-step method. To get back to one step, we have to rewrite equation (10) as **two coupled equations** (one vector equation) for the state at time $k+1$:

$$\begin{array}{c}\textbf{Descent with}\\ \textbf{momentum}\end{array} \quad \boxed{\begin{array}{ll} \boldsymbol{x}_{k+1} & = \boldsymbol{x}_k - \boldsymbol{s}\boldsymbol{z}_k \\ \boldsymbol{z}_{k+1} - \boldsymbol{\nabla f}(\boldsymbol{x}_{k+1}) & = \beta z_k \end{array}} \tag{11}$$

With those two equations, we have reached a one-step method. This is exactly like reducing a single second order differential equation to a system of two first order equations. Second order reduces to first order when dy/dt becomes a second unknown along with y.

$$\begin{array}{l}\textbf{2nd order equation}\\ \textbf{1st order system}\end{array} \quad \frac{d^2y}{dt^2} + b\frac{dy}{dt} + ky = 0 \text{ becomes } \frac{d}{dt}\begin{bmatrix} y \\ dy/dt \end{bmatrix}$$

$$= \begin{bmatrix} 0 & 1 \\ -k & -b \end{bmatrix} \begin{bmatrix} y \\ dy/dt \end{bmatrix}$$

Interesting that this b is damping the motion while β adds momentum to encourage it.

1.4.6 The Quadratic Model

When $f(\boldsymbol{x}) = \frac{1}{2}\boldsymbol{x}^T\boldsymbol{S}\boldsymbol{x}$ is quadratic, its gradient $\boldsymbol{\nabla f} = \boldsymbol{S}\boldsymbol{x}$ is linear. This is the model problem to understand: S is symmetric positive definite and $\boldsymbol{\nabla f}\boldsymbol{x}_{k+1}$ becomes $S\boldsymbol{x}_{k+1}$ in equation (11). Our 2 by 2 supermodel is included, when the matrix S is diagonal with entries 1 and b. For a bigger matrix S, you will see that its largest and smallest eigenvalues determine the best choices for β and the stepsize s–so the 2 by 2 case actually contains the essence of the whole problem.

To follow the steps of accelerated descent, we track each eigenvector of S. Suppose $S\boldsymbol{q} = \lambda\boldsymbol{q}$ and $\boldsymbol{x}_k = c_k\boldsymbol{q}$ and $z_k = d_k\boldsymbol{q}$ and $\boldsymbol{\nabla f}_k = S\boldsymbol{x}_k = \lambda c_k\boldsymbol{q}$. Then our equation (11) connects the numbers c_k and d_k at step k to c_{k+1} and d_{k+1} at step $k+1$.

Following the eigenvector $\boldsymbol{q}$ which is given as follows:

$$\begin{aligned} c_{k+1} &= c_k - sd_k \\ -\lambda c_{k+1} + d_{k+1} &= \beta d_k \end{aligned} \qquad \begin{bmatrix} \mathbf{1} & \mathbf{0} \\ -\boldsymbol{\lambda} & \mathbf{1} \end{bmatrix} \begin{bmatrix} c_{k+1} \\ d_{k+1} \end{bmatrix} = \begin{bmatrix} \mathbf{1} & -\boldsymbol{s} \\ 0 & \boldsymbol{\beta} \end{bmatrix} \begin{bmatrix} c_k \\ d_k \end{bmatrix} \tag{12}$$

Finally we invert the first matrix ($-\lambda$ becomes $+\lambda$) to see each descent step clearly:

Descent step multiplies by $\boldsymbol{R}$ which is given as follow:

$$\boxed{\begin{bmatrix} c_{k+1} \\ d_{k+1} \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ \lambda & 1 \end{bmatrix} \begin{bmatrix} 1 & -s \\ 0 & \beta \end{bmatrix} \begin{bmatrix} c_k \\ d_k \end{bmatrix} = \begin{bmatrix} \mathbf{1} & -\boldsymbol{s} \\ \boldsymbol{\lambda} & \boldsymbol{\beta} - \boldsymbol{\lambda s} \end{bmatrix} \begin{bmatrix} c_k \\ d_k \end{bmatrix} = \boldsymbol{R} \begin{bmatrix} c_k \\ d_k \end{bmatrix}} \tag{13}$$

After k steps the starting vector is multiplied by R^k. For fast convergence to zero (which is the minimum of $f = \frac{1}{2}\boldsymbol{x}^T\boldsymbol{S}\boldsymbol{x}$) we want both eigenvalues e_1 and e_2 of R to be as small as possible. Clearly those eigenvalues of R depend on the eigenvalue λ of S. That eigenvalue λ could be anywhere between $\lambda_{min}(S)$ and $\lambda_{max}(S)$. Our problem is:

Choose $\boldsymbol{s}$ and $\boldsymbol{\beta}$ to minimize $\quad \mathbf{max} \Big[|\boldsymbol{e_1}(\boldsymbol{\lambda})|, |\boldsymbol{e_2}(\boldsymbol{\lambda})|\Big]$ for $\quad \lambda_{min}(S) \le \lambda \le \lambda_{max}(S)$ (14)

It seems a miracle that this problem has a beautiful solution. The optimal s and β are

$$\boxed{s = \left(\frac{2}{\sqrt{\lambda_{max}}+\sqrt{\lambda_{min}}}\right)^2 \quad \text{and} \quad \beta = \left(\frac{\sqrt{\lambda_{max}}-\sqrt{\lambda_{min}}}{\sqrt{\lambda_{max}}+\sqrt{\lambda_{min}}}\right)^2} \tag{15}$$

Think of the 2 by 2 supermodel, when S has eigenvalues $\lambda_{max} = 1$ and $\lambda_{min} = b$:

$$s = \left(\tfrac{2}{1+\sqrt{b}}\right)^2 \qquad \text{and} \qquad \beta = \left(\tfrac{1-\sqrt{b}}{1+\sqrt{b}}\right)^2 \tag{16}$$

These choices of stepsize and momentum give a convergence rate that looks like the rate equation (4) for ordinary steepest descent (no momentum). But there is a crucial difference: $\boldsymbol{b}$ **is replaced by** $\sqrt{\boldsymbol{b}}$.

Ordinary descent factor	**Accelerated descent factor**	
$\left(\dfrac{1-b}{1+b}\right)^2$	$\left(\dfrac{1-\sqrt{b}}{1+\sqrt{b}}\right)^2$	(17)

So similar but so different. The real test comes when b is very small. Then the ordinary descent factor is essentially $1 - 4b$, very close to 1. The accelerated descent factor is essentially $1 - 4\sqrt{b}$, *much further from 1.*

To emphasize the improvement that momentum brings, suppose $b = 1/100$. Then $\sqrt{b} = 1/10$ (ten times larger than b). The convergence factors in equation (17) are

Steepest descent $\left(\frac{.99}{1.01}\right)^2 = \mathbf{.96}$ **Accelerated descent** $\left(\frac{.9}{1.1}\right)^2 = \mathbf{.67}$

Ten steps of ordinary descent multiply the starting error by 0.67. This is matched by a single momentum step. Ten steps with the momentum term multiply the error by 0.018.

Notice that $\lambda_{max}/\lambda_{min} = 1/b = \kappa$ is the **condition number of** $\boldsymbol{S}$. This controls everything. For the non-quadratic problems studied next, the condition number is still the key. That number κ becomes L/μ as you will see.

[2]

Next Section on stochastic gradient descent returns to these papers for this message: Adaptive methods can possibly converge to undesirable weights in deep learning. Gradient descent from $\boldsymbol{x}_o = 0$ (and **SGD**) finds the minimum norm solution to least squares.[3]

Those adaptive methods (variants of *Adam*) are popular. The formula for $\boldsymbol{x}_{k+1}$ that stopped at $\boldsymbol{x}_{k-1}$ will go much further back–to include all earlier points starting at $\boldsymbol{x}_o$. In many problems that leads to faster training. As with momentum, *memory can help.*

When there are more weights to determine than samples to use (underdetermined problems), we can have multiple minimum points for $f(\boldsymbol{x})$ and multiple solutions to $\boldsymbol{\nabla f} = \mathbf{0}$. A crucial question in next Section is whether improved adaptive methods find good solutions.[4]

1.5 Stochastic Gradient Descent and ADAM

Gradient descent is fundamental in training a deep neural network. It is based on a step of the form $\boldsymbol{x}_{k+1} = \boldsymbol{x}_k - s_k \boldsymbol{\nabla} L(\boldsymbol{x}_k)$. That step should lead us downhill toward the point $\boldsymbol{x}^*$ where the loss function $L(\boldsymbol{x})$ is minimized for the test data $\boldsymbol{v}$. But for large networks with many samples in the training set, this algorithm (as it stands) is not successful! [5]

It is important to recognize two different problems with classical steepest descent:

1. Computing $\boldsymbol{\nabla} L$ at every descent step–the derivatives of the total loss L with respect to all the weights $\boldsymbol{x}$ in the network–is too expensive. That total loss adds the individual losses $l(\boldsymbol{x}, \boldsymbol{v}_i)$ *for every sample* $\boldsymbol{v}_i$ *in the training set*—potentially millions of separate losses are computed and added in every computation of $\boldsymbol{L}$.
2. The number of weights is even larger. So $\boldsymbol{\nabla_x L} = \mathbf{0}$ for many different choices $\boldsymbol{x}^*$ of the weights. **Some of those choices can give poor results on unseen test data**. The learning function $\boldsymbol{F}$ can fail to "generalize". But **stochastic gradient descent (SGD)** does find weights $\boldsymbol{x}^*$ that generalize—weights that will succeed on unseen vectors v from a similar population.

Stochastic gradient descent uses only a "minibatch" of the training data at each step. B samples will be chosen randomly. Replacing the full batch of all the training data by a minibatch changes $L(\boldsymbol{x}) = \frac{1}{n}\sum l_i(\boldsymbol{x})$ to a sum of only B losses. This resolves both difficulties at once. The success of deep learning rests on these two facts:

1. Computing $\boldsymbol{\nabla} l_i$ by backpropagation on B samples is much faster. Often $B = 1$.
2. The stochastic algorithm produces weights $\boldsymbol{x}^*$ that also succeed on unseen data.

The first point is clear. The calculation per step is greatly reduced. The second point is a miracle. Generalizing well to new data is a gift that researchers work hard to explain.

We can describe the big picture of weight optimization in a large neural network. The weights are determined by the training data. That data often consists of thousands of samples. We know the "$features$" of each sample—maybe its height and weight, or its shape and color, or the number of nouns and verbs and commas (for a sample of text). Those features go into a vector v for each sample. We use a minibatch of samples.

And for each sample in the training set, we know if it is "a cat or a dog"–or if the text is "poetry or prose". We look for a **learning function $\boldsymbol{F}$** that assigns good weights. Then for v in a similar population, F outputs the correct classification "cat" or "poetry".

We use this function F for unidentified **test data**. The features of the test data are the new inputs v. The output from F will be the correct (?) classification—provided the function has learned the training data in a way that generalizes.

Here is a remarkable observation from experience. *We don't want to fit the training data too perfectly.* That would often be **overfitting**. The function F becomes oversensitive. It memorized everything but it hasn't learned anything. **Generalization by SGD** is the ability to give the correct classification for unseen test data v, based on the weights $\boldsymbol{x}$ that were learned from the training data.

I compare overfitting with choosing a polynomial of degree 60 that fits exactly to 61 data points. Its 61 coefficients a_0 to a_60 will perfectly learn the data. But that high degree polynomial will oscillate wildly between the data points. For test data at a nearby point, the perfect-fit polynomial gives a *completely wrong answer.*

So a fundamental strategy in training a neural network (which means finding a function that learns from the training data and generalizes well to test data) is **early stopping**. Machine learning needs to know when to quit! Possibly this is true of human learning too.

1.5.1 The Loss Function and the learning Function

Now we establish the optimization problem that the network will solve. We need to define the "loss" $\boldsymbol{L}(\boldsymbol{x})$ that our function will (approximately) minimize. This is the sum of the errors in classifying each of the training data vectors v. And we need to describe the form of the learning function $\boldsymbol{F}$ that classifies each data vector v.

At the beginning of machine learning the function F was *linear*–a severe limitation. Now F is certainly nonlinear. Just the inclusion of one particular nonlinear function at each neuron in each layer has made a dramatic difference. It has turned out that with thousands of samples, the function F can be correctly trained.

It is the processing power of the computer that makes for fast operations on the data. In particular, we depend on the speed of GPU's (the Graphical Processing Units that were originally developed for computer games). They make deep learning possible.

We first choose a loss function to minimize. Then we describe stochastic gradient descent. The gradient is determined by the network architecture–the "feedforward" steps whose weights we will optimize. Our goal in this section is to find *optimization algorithms that apply to very large problems.* Then Next Chapter will describe how the architecture and the algorithms have made the learning functions successful.

Here are three loss functions—cross-entropy is a favorite for neural nets. Section 4 in will describe the advantages of cross-entropy loss over square loss (as in least squares).

1. **Square loss** $\quad L(\boldsymbol{x}) = \frac{1}{N}\sum_1^N ||F(\boldsymbol{x},\boldsymbol{v}_i) - \text{true}||^2$: sum over the training samples $\boldsymbol{v}_i$
2. **hinge loss** $\quad L(\boldsymbol{x}) = \frac{1}{N}\sum_1^N \textbf{max}(0, 1 - tF(\boldsymbol{x}))$ for **classification** $t = 1$ or -1
3. **Cross-entropy loss** $\quad L(\boldsymbol{x}) = -\frac{1}{N}\sum_1^N [y_i log \widehat{y}_i + (1 - y_i) log(1 - \widehat{y}_i)]$ for $y_i = 0$ or 1

Cross-entropy loss or ”logistic loss” is preferred for *logistic regression* (with two choices only). The true label $y_i = 0$ or 1 could be -1 or 1 ($\widehat{y}_i$ is a computed label).

For a minibatch of size B, replace N by B. And choose the B samples randomly.

1.5.2 Stochastic Descent Using One Sample Per Step

To simplify, suppose each minibatch contains only one sample $\boldsymbol{v}_k$ (so $B = 1$). That sample is chosen randomly. The theory of stochastic descent usually assumes that the sample is replaced after use–in principle the sample could be chosen again at step $k + 1$. But replacement is expensive compared to starting with a random ordering of the samples. In practice, we often omit replacement and work through samples in a random order.

Each pass through the training data is **one epoch** of the descent algorithm. Ordinary gradient descent computes one epoch per step (batch mode). Stochastic gradient descent needs many steps (for minibatches). The online advice is to choose B $<$ 32.

Stochastic descent began with a seminal paper of Herbert Robbins and Sutton Monro in the *Annals of Mathematical Statistics* **22** (1951) 400-407: A Stochastic

Approximation Method. Their goal was a fast method that converges to $\boldsymbol{x}^*$ in probability:

To prove Prob $(||\boldsymbol{x}_k - \boldsymbol{x}^*|| > \epsilon)$ approaches zero as $k \to \infty$.

Stochastic descent is more sensitive to the stepsizes s_k than full gradient descent. If we randomly choose sample v_i at step k, then the kth descent step is familiar:

$$\boxed{\boldsymbol{x}_{k+1} = \boldsymbol{x}_k - s_k \boldsymbol{\nabla_x l}(\boldsymbol{x}_i, \boldsymbol{v}_i)}$$
$$\boxed{\boldsymbol{\nabla_x l} = \text{derivative of the loss term from sample } \boldsymbol{v}_i}$$

We are doing much less work per step (B inputs instead of all inputs from the training set). But we do not necessarily converge more slowly. A typical feature of stochastic gradient descent is **"semi-convergence"**: fast convergence at the start.

Early steps of SGD often converge more quickly than GD toward the solution $\boldsymbol{x}^*$.

This is highly desirable for deep learning. Section 4 showed zig-zag for full batch mode. This was improved by adding momentum from the previous step (which we may also do for SGD). Another improvement frequently comes by using **adaptive methods** like some variation of ADAM. Adaptive methods look further back than momentum–now all previous descent directions are remembered and used. Those come later in this section.

Here we pause to look at semi-convergence: Fast start by stochastic gradient descent. We admit immediately that later iterations of SGD are frequently erratic. **Convergence at the start changes to large oscillations near the solution**. Figure given below will show this. One response is to stop early. And thereby we avoid overfitting the data.

In the following example, the solution $\boldsymbol{x}^8$ is in a specific interval I. If the current approximation $\boldsymbol{x}_k$ is outside I, the next approximation $\boldsymbol{x}_{k+1}$ is closer to I (or inside I). That gives semiconvergence–a good start. *But eventually the x bounce around inside I.*

1.5.3 Fast Convergence at the Start: Least Squares with $n = 1$

We learned from Suvrit Sra that the simplest example is the best. The vector $\boldsymbol{x}$ has only one component x. The ith loss is $l_i = \frac{1}{2}(a_i x - b_i)^2$ with $a_i > 0$. The gradient of l_i is its derivative $a_i(a_i x - b_i)$. It is zero and l_i is minimized at $\boldsymbol{x} = \boldsymbol{b_i}/\boldsymbol{a_i}$. The total loss over all N samples is $L(x) = \frac{1}{2N}\sum(a_i x - b_i)^2$: Least squares with N equations, 1 unknown.

$$\text{The equation to solve is } \boldsymbol{\nabla L} = \frac{1}{2N}\sum_{1}^{N}(a_i x - b_i) = 0.\text{The solution } \boldsymbol{x}^* = \frac{\sum \boldsymbol{a_i b_i}}{\sum \boldsymbol{a_i^2}} \tag{1}$$

Important If B/A is the largest ratio b_i/a_i, then the true solution $\boldsymbol{x}^*$ **is below** $\boldsymbol{B/A}$

$$\text{all } \frac{\boldsymbol{b_i}}{\boldsymbol{a_i}} \le \frac{\boldsymbol{B}}{\boldsymbol{A}} \quad Aa_ib_i \le Ba_i^2 \quad A(\sum a_ib_i) \le B(\sum a_i^2) \;\; \boldsymbol{x}^* = \frac{\sum a_ib_i}{\sum a_i^2} \le \frac{\boldsymbol{B}}{\boldsymbol{A}} \tag{2}$$

Similarly $\boldsymbol{x}^*$ is above the smallest ratio $\boldsymbol{\beta/\alpha}$. Conclusion: If $\boldsymbol{x}_k$ is outside the interval $\boldsymbol{I}$ from β/α to B/A, then the kth gradient descent step will move *toward that interval* $\boldsymbol{I}$ *containing* $\boldsymbol{x}^*$. Here is what we can expect from stochastic gradient descent:

> **If $\boldsymbol{x_k}$ is outside $\boldsymbol{I}$, then $\boldsymbol{x_{k+1}}$ moves towards the interval $\boldsymbol{\beta/\alpha \le x \le B/A}$**
> **If $\boldsymbol{x_k}$ is inside $\boldsymbol{I}$, then so is $\boldsymbol{x_{k+1}}$. The iterations can bounce** around inside $\boldsymbol{I}$.

A typical sequence $\boldsymbol{x}\boldsymbol{x}_1, \boldsymbol{x}_2, ...$ from minimizing $||A\boldsymbol{x} - \boldsymbol{b}||^2$ by stochastic gradient descent is graphed in Figure given below. *You see the fast start and the oscillating finish.* This behavior is a perfect signal to think about early stopping or averaging when the oscillations start.

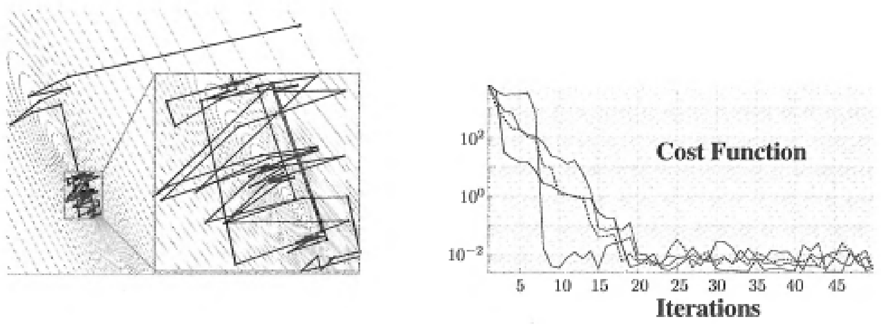


Figure 1.12: The left figure shows a trajectory of stochastic gradient decent with two unknowns. The early iterations succeed but later iterations oscillate (as shown in the inset). On the right, the quadratic cost function decreases quickly at first and then fluctuates instead of converging. The four paths start from the same x_o with random choices of i in equation (3). The condition number of the 40 by 2 matrix A is only 8.6.

1.5.4 Randomized Kaczmarz is Stochastic Gradient Descent for $Ax = b$

$$\boxed{\textbf{Kaczmarz for } \boldsymbol{Ax = b} \textbf{ with random } \boldsymbol{i(k)} \qquad \boldsymbol{x}_{k+1} = \boldsymbol{x}_k + \frac{\boldsymbol{b}_i - \boldsymbol{a}_i^T \boldsymbol{x}_k}{||\boldsymbol{a}_i||^2}\boldsymbol{a}_i} \qquad (3)$$

We are randomly selecting row i of A at step k. We are adjusting $\boldsymbol{x}_{k+1}$ to solve equation i in $A\boldsymbol{x} = \boldsymbol{b}$. (Multiply equation (3) by $\boldsymbol{a}_i^T$ to verify that $\boldsymbol{a}_i^T \boldsymbol{x}_{k+1} = b_i$. This is equation i in $A\boldsymbol{x} = \boldsymbol{b}$.) Geometrically, $\boldsymbol{x}_{k+1}$ is the projection of $\boldsymbol{x}_k$ onto one of the hyperplanes $\boldsymbol{a}_i^T \boldsymbol{x} = b_i$ that meet at $\boldsymbol{x}^* = A^{-1}\boldsymbol{b}$.

This algorithm resisted a close analysis for many years. The equations $\boldsymbol{a}_1^T \boldsymbol{x} = b_1, \boldsymbol{a}_2^T = b_2 ...$ were taken in cyclic order with step $s = 1$. Then Strohmer and Vershynin proved fast convergence for random Kaczmarz. They used SGD with *norm-squared sam- pling (importance sampling)*: Choose row i of A with probability p_i proportional to $||\boldsymbol{a}_i^T||2$.

The previous page described the Kaczmarz iterations for $A\boldsymbol{x} = \boldsymbol{b}$ when A was N by 1. The sequence $x_0, x_1, x_2, ...$ moved toward the interval I. The least squares solution x^* was in that interval. For an N by K matrix A, we expect the K by 1 vectors $\boldsymbol{x}_i$ to move into a K-dimensional box around $\boldsymbol{x}^*$. Figure above showed this for $K = 2$.

The next page will present numerical experiments for stochastic gradient descent:

A variant of random Kaczmarz was developed by Gower and Richtarik, with no less than six equivalent randomized interpretations. Here are references that connect the many variants from the original by Kaczmarz in the 1937 Bulletin de l'Académie Polonaise.[6][7][8]

1.5.5 Random Kaczmarz and Iterated Projections

Suppose $A\boldsymbol{x}^* = \boldsymbol{b}$. A typical step of random Kaczmarz projects the current error $\boldsymbol{x}_k - \boldsymbol{x}^*$ onto the hyperplane $\boldsymbol{x}_i^T\boldsymbol{x} = \boldsymbol{b}_i$. Here i is chosen randomly at step k (often with importance sampling using probabilities proportional to $||a_i||^2$). To see that projection matrix $\boldsymbol{a}_i\boldsymbol{a}_i^T/\boldsymbol{a}_i^T\boldsymbol{a}_i$, substitute $b_i = \boldsymbol{a}_i^T\boldsymbol{x}^*$ into the update step (3):

$$\boldsymbol{x}_{k+1} - \boldsymbol{x}^* = \boldsymbol{x}_k - \boldsymbol{x}^* + \frac{b_i - \boldsymbol{a}_i^T\boldsymbol{x}_k}{||\boldsymbol{a}_i||^2}\boldsymbol{a}_i = (\boldsymbol{x}_{\boldsymbol{k}} - \boldsymbol{x}^*) - \frac{\boldsymbol{a}_i\boldsymbol{a}_i^T}{\boldsymbol{a}_i^T\boldsymbol{a}_i}(\boldsymbol{x}_k - \boldsymbol{x}^*) \tag{4}$$

Orthogonal projection never increases length. The error can only decrease. The error norm $||\boldsymbol{x}_k - \boldsymbol{x}^*||$ decreases steadily, even if the cost function $||A\boldsymbol{x}_k - \boldsymbol{b}||$ does not. *But convergence is usually slow!* Strohmer-Vershynin estimate the expected error:

$$\mathbf{E}\left[||\boldsymbol{x}_k - \boldsymbol{x}^*||^2\right] \leq \left(1 - \tfrac{1}{c^2}\right)^k ||\boldsymbol{x}_0 - \boldsymbol{x}^*||^2, c = \text{condition number of } A. \tag{5}$$

This is slow compared to gradient descent (there c^2 is replaced by c, and then $\sqrt{c}$ with momentum in I.4). But (5) is independent of the size of A: attractive for large problems.

The theory of alternating projections was initiated by von Neumann (in Hilbert space). See books and papers by Bauschke-Borwein, Escalante-Raydan, Diaconis, Xu,...

Our experiments converge slowly! The 100 by 10 matrix A is random with $c \approx 400$. The figures show random Kaczmarz for 600,000 steps. We measure convergence by the angle θ_k between $\boldsymbol{x}_k - \boldsymbol{x}^*$ and the row $\boldsymbol{a}_i$ chosen at step k. The error equation (4) is

$$||\boldsymbol{x}_{k+1} - \boldsymbol{x}^*||^2 = (1 - \cos^2\theta_k)||\boldsymbol{x}_k - \boldsymbol{x}^*||^2 \tag{6}$$

The graph shows that those numbers $1 - cos^2\theta_k$ are very close to 1: **slow convergence**. But the second graph confirms that convergence does occur. The Strohmer-Vershynin bound (5) becomes $\boldsymbol{E}[cos^2\theta_k] \geq 1/c^2$. Our example matrix has $1/c^2 \approx 10^{-5}$ and often $cos^2\theta_k \approx 2.10_{-5}$, confirming that bound.

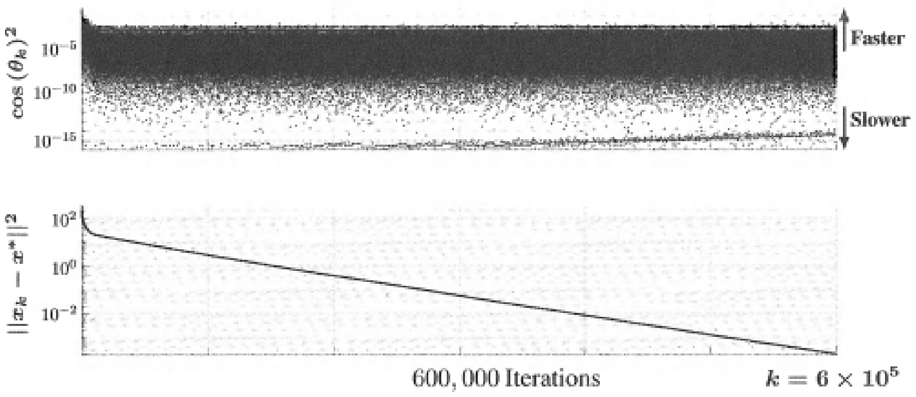


Figure 1.13: Convergence of the squared error for random Kaczmarz. Equation (6) with $1 - cos^2\theta_k$ close to $1 - 10^{-5}$ produces the slow convergence in the lower graph.

1.5.6 Convergence in Expectation

For a stochastic algorithm like SGD, we need a convergence proof that accounts for randomness–in the assumptions and also in the conclusions. Suvrit Sra provided us with such a proof, and we reproduce it here. The function $f(\boldsymbol{x})$ is a sum $\frac{1}{n}\sum f_i(\boldsymbol{x})$ of n terms. The sampling chooses $i(k)$ at step k uniformly from the numbers 1 to n (with replacement!) and the stepsize is $s = \text{constant}/\sqrt{T}$. First come assumptions on $f(\boldsymbol{x})$ and $\boldsymbol{\nabla f}(\boldsymbol{x})$, followed by a standard requirement (no bias) for the random sampling.

1. **Lipschitz smoothness of $\boldsymbol{\nabla f}(\boldsymbol{x})$** $\quad ||\boldsymbol{\nabla f}(\boldsymbol{x}) - \boldsymbol{\nabla f}(\boldsymbol{y})|| \leq L||\boldsymbol{x} - \boldsymbol{y}||$
2. **Bounded gradients** $\quad ||\boldsymbol{\nabla f}_{i(k)}(\boldsymbol{x})|| \leq G$

3. **Unbiased stochastic gradients** $\quad \boldsymbol{E}[\boldsymbol{\nabla f}_{i(k)} - \boldsymbol{\nabla f}(\boldsymbol{x})] = 0$

From Assumption 1 it follows that

$$f(\boldsymbol{x}_{k+1}) \leq f(\boldsymbol{x}_k) + (\boldsymbol{\nabla f}(\boldsymbol{x}_k), \boldsymbol{x}_{k+1} - \boldsymbol{x}_k) + \tfrac{1}{2}Ls^2||\boldsymbol{\nabla f}_{i(k)}(\boldsymbol{x}_k)||^2$$

$$f(\boldsymbol{x}_{k+1}) \leq f(\boldsymbol{x}_k) + (\boldsymbol{\nabla f}(\boldsymbol{x}_k), -s\boldsymbol{\nabla f}_{i(k)}(\boldsymbol{x}_k)) + \tfrac{1}{2}Ls^2||\boldsymbol{\nabla f}_{i(k)}(\boldsymbol{x}_k)||^2$$

now take expectations of both sides and use Assumptions 2-5:

$$\boldsymbol{E}[f(\boldsymbol{x}_{k+1})] \leq \boldsymbol{E}[f(\boldsymbol{x}_k)] - s\boldsymbol{E}[||\boldsymbol{\nabla f}(\boldsymbol{x}_k)||^2] + \tfrac{1}{2}Ls^2G^2$$

$$\Rightarrow \boldsymbol{E}[||\boldsymbol{\nabla f}(\boldsymbol{x}_k)||^2] \leq \frac{1}{s}\boldsymbol{E}[f(\boldsymbol{x}_k) - f(\boldsymbol{x}_{k+1})] + \frac{1}{2}Ls^2G^2 \tag{7}$$

Choose the stepsize $s = c/\sqrt{T}$, and add up (7) from $k = 1$ to T. The sum telescopes:

$$\frac{1}{T}\sum_{K=1}^{T} \boldsymbol{E}[||\boldsymbol{\nabla f}(\boldsymbol{x}_k)||^2] \leq \frac{1}{\sqrt{T}}\left(\tfrac{f(\boldsymbol{x}_1)-f(\boldsymbol{x}^*)}{c} + \tfrac{Lc}{2}G^2\right) = \frac{C}{\sqrt{T}} \tag{8}$$

Here $f(\boldsymbol{x}^*)$ is the global minimum. The smallest term in (8) is below the average:

$$1 \leq \min_k \leq T \;\; \boldsymbol{E}[||\boldsymbol{\nabla f}(\boldsymbol{x_k})||^2] \leq \boldsymbol{C/\sqrt{T}} \tag{9}$$

The conclusion of Sra's theorem is convergence in expectation at a sublinear rate.

1.5.7 Weight Averaging Inside SGD

The idea of **averaging the outputs** from several steps of stochastic gradient descent looks promising. The learning rate (stepsize) can be constant or cyclical over each group of outputs. Gordon Wilson et al have named this method *Stochastic Weight Averaging* (SWA). They emphasize that this gives promising results for training deep networks, with better generalization and almost no overhead. It seems natural and effective.

P. Izmailov, D. Podoprikhin, T. Garipov, D. Vetrov, A. Gordon Wilson, *Averaging weights leads to wider optima and better generalization*, arXiv: 1803.05407.

1.5.8 Adaptive Methods Using Earlier Gradients

For faster convergence of gradient descent and stochastic gradient descent, adaptive methods have been a major direction. The idea is to *use gradients from earlier steps.* That "memory" guides the choice of search direction $\boldsymbol{D}$ and the all-important stepsize s. We are searching for the vector $\boldsymbol{x}^*$ that minimizes a specified loss function $L(\boldsymbol{x})$. In the step from $\boldsymbol{x}_k$ to $\boldsymbol{x}_{k+1}$, we are free to choose $\boldsymbol{D}_k$ and s_k:

$$\boldsymbol{D}_k = \boldsymbol{D}(\boldsymbol{\nabla L}_k, \boldsymbol{\nabla L}_{k-1}, ..., \boldsymbol{\nabla L}_0) \quad \text{and} \quad s_k = s(\boldsymbol{\nabla L}_k, \boldsymbol{\nabla L}_{k-1}, ..., \boldsymbol{\nabla L}_0) \tag{10}$$

For a standard SGD iteration, $\boldsymbol{D}_k$ depends only on the current gradient $\boldsymbol{\nabla L_k}$ (and s_k might be $s/\sqrt{k}$). That gradient $\boldsymbol{\nabla L_k}(\boldsymbol{x}_k, B)$ is evaluated only on a random minibatch B of the test data. Now, deep networks often have the option of using some or all of the earlier gradients (computed on earlier random minibatches):

$$\boxed{\textbf{Adaptive Stochastic Gradient Descent} \qquad \boldsymbol{x}_{k+1} = \boldsymbol{x}_k - s_k \boldsymbol{D}_k} \tag{11}$$

Success or failure will depend on $\boldsymbol{D}_k$ and s_k. The first adaptive method (called ADAGRAD) chose the usual search direction $\boldsymbol{D}_k = \boldsymbol{\nabla L}(\boldsymbol{x}_k)$ but computed the stepsize from all previous gradients [Duchi-Hazan-Singer]:

$$\textbf{ADAGRAD stepsize} \qquad s_k = \left(\frac{\alpha}{\sqrt{k}}\right) \left[\frac{1}{k}\text{diag}\left(\textstyle\sum_1^k ||\boldsymbol{\nabla L_i}||^2\right)\right]^{1/2} \tag{12}$$

$\alpha/\sqrt{k}$ is a typical decreasing stepsize in proving convergence of stochastic descent. It is often omitted when it slows down convergence in practice. The "memory factor" in (12) led to real gains in convergence speed. Those gains made adaptive methods a focus for more development.

Exponential moving averages in ADAM [Kingma-Ba] have become the favorites. Unlike (12), recent gradients $\boldsymbol{\nabla L}$ have greater weight than earlier gradients in both s_k and the step direction $\boldsymbol{D}_k$. The exponential weights in $\boldsymbol{D}$ and s come from $\delta < 1$ and $\beta < 1$:

$$\boldsymbol{D}_k = (1-\delta)\sum_{i=1}^{k}\delta^{k-i}\boldsymbol{\nabla} L(\boldsymbol{x}_i) \quad s_k = \left(\frac{\alpha}{\sqrt{k}}\right)\left[(1-\beta)\text{diag}\textstyle\sum_{i=1}^{k}\beta^{k-i}||\boldsymbol{\nabla L}(\boldsymbol{x}_i)||^2\right]^{1/2} \tag{13}$$

Typical values are $8\delta = 0.9$ and $\beta = 0.999$. Small values of δ and β will effectively kill off the moving memory and lose the advantages of adaptive methods for convergence speed. That speed is important in the total cost of gradient descent! The look-back formula for the direction $\boldsymbol{D}_k$ is like including momentum in the heavy ball method of Section 4.

The actual computation of $\boldsymbol{D}_k$ and s_k will be a recursive combination of old and new:

$$\boxed{\boldsymbol{D}_k = \delta\boldsymbol{D}_{k-1} + (1-\delta)\boldsymbol{\nabla L}(\boldsymbol{x}_k) \quad s_k^2 = \beta s_{k-1}^2 + (1-\beta)||\boldsymbol{\nabla L}(\boldsymbol{x}_k)||^2} \tag{14}$$

This adaptive method clearly produce fast convergence.

ADAM is highly popular in practice (this is written in 2018). But several authors have pointed out its defects. Hardt, Recht, and Singer constructed examples to show that its limit x_∞ for the weights in deep learning could be problematic: Convergence may fail or (worse) the limiting weights may *generalize poorly* in applications to unseen test data.

Equations (13)-(14) follow the recent conference paper of Reddi, Kale, and Kumar. Those authors prove non-convergence of ADAM, with simple examples in which *the stepsize s_k increases in time*–an undesired outcome. In their example, ADAM takes the wrong direction twice and the right direction once in every three steps. The exponential decay scales down that good step and overall the stepsizes s_k do not decrease. A large β (near 1) is needed and used, but there are always convex optimization problems on which ADAM will fail. The *idea is still good.*

One approach is to use an increasing minibatch size B. The NIPS 2018 paper proposes a new adaptive algorithm YOGI, which better controls the learning rate (the stepsize). Compared with ADAM, a key change is to an additive update; other steps are unchanged. At this moment, experiments are showing improved results with YOGI.

And after fast convergence to weights that nearly solve $\boldsymbol{\nabla l}(\boldsymbol{x}) = \mathbf{0}$ there is still the crucial issue: **Why do those weights generalize well to unseen test data?**[9] [10] [11] [12]

We end this chapter by emphasizing: Stochastic gradient descent is now the leading method to find weights $\boldsymbol{x}$ that minimize the loss $L(\boldsymbol{x})$ and solve $\boldsymbol{\nabla l}(\boldsymbol{x}^*) = 0$, Those weights from SGD normally succeed on unseen test data.

1.5.9 Generalization: Why is Deep Learning So Effective

We end Chapter I–and connect to Chapter II–with a short discussion of a central question for deep learning. The issue here is **generalization**. This refers to the behavior of a neural network on test data that it has not seen. If we construct a function $F(\boldsymbol{x}, \boldsymbol{v})$ that successfully classifies the known training data v, will F continue to give correct results when v is outside the training set?

The answer must lie in the stochastic gradient descent algorithm that chooses weights. Those weights $\boldsymbol{x}$ minimize a loss function $L(\boldsymbol{x}, \boldsymbol{v})$ over the training data. The question is: **Why do the computed weights do so well on the test data?**

Often we have more free parameters in $\boldsymbol{x}$ than data in v. In that case we can expect many sets of weights (many vectors $\boldsymbol{x}$) to be equally accurate on the training set. Those weights could be good or bad. They could generalize well or poorly. Our algorithm chooses a particular $\boldsymbol{x}$ and applies those weights to new data v_{test}.

An unusual experiment produced unexpectedly positive results. The components of each input vector v were randomly shuffled. So the individual features represented by v suddenly had no meaning. Nevertheless the deep neural net learned those randomized samples. The learning function $F(\boldsymbol{x}, \boldsymbol{v})$ still classified the test data correctly. Of course $\boldsymbol{F}$ could not succeed with unseen data, when the components of v are reordered.

It is a common feature of optimization that smooth functions are easier to approximate than irregular functions. But here, for completely randomized input vectors, stochastic gradient descent needed only three times as many epochs (triple the number of iterations) to learn the training data. This random labeling of the training samples (the experiment has become famous) is described in arXiv: 1611.03530.

1.5.10 Kaczmarz Method in Tomographic Imaging (Ct)

A key property of Kaczmarz is its quick success in early iterations. This is called *semiconvergence* in tomography (where solving $A\boldsymbol{x} = \boldsymbol{b}$ constructs a CT image, and the method produces a regularized solution when the data is noisy). Quick semi-convergence for noisy data is an excellent property for such a simple method. The first steps all approach the correct interval from α/β to A/B (for one scalar unknown). But inside that interval, Kaczmarz jumps around unimpressively.

We are entering here the enormous topic of ill-conditioned inverse problems (see the books of P. C. Hansen). In this book we can do no more than open the door.

Chapter 2

Learning from Data

This part of the book is a great adventure–hopefully for the reader, certainly for the author, and it involves the whole science of thought and intelligence. You could call it Machine Learning (ML) or Artificial Intelligence (AI). Human intelligence created it (but we don't fully understand what we have done). Out of some combination of ideas and failures, attempting at first to imitate the neurons in the brain, a successful approach has emerged to finding **patterns in data**.

What is important to understand about deep learning is that those data-fitting computations, of almost unprecedented size, are often heavily underdetermined. There are a great many points in the training data, but there are far more weights to be computed in a deep network. The art of deep learning is to find, among many possible solutions, one that will **generalize to new data.**

It is a remarkable observation that learning on deep neural nets with many weights leads to a successful tradeoff: F is accurate on the training set *and* the unseen test set. This is the good outcome from minibatch gradient descent with momentum and the hyperparameters from Section 4 (including stepsize selection and early stopping).

This chapter is organized in an irregular order. **Deep learning comes first**. Earlier models like Support Vector Machines and Kernel Methods are briefly described in II.5. The order is anhistorical, and the reader will know why. Neural nets have become the primary architecture for the most interesting (and the most difficult) problems of machine learning. That multi-layer architecture often succeeds, but by no means always! This book has been preparing for deep learning and we simply give it first place.

Sections 1-2 describe the learning function $\boldsymbol{F}(\boldsymbol{x}, \boldsymbol{v})$ for *fully connected nets and convolutional nets.* The training data is given by a set of feature vectors $\boldsymbol{v}$. The weights that allow $\boldsymbol{F}$ to classify that data are in the vector $\boldsymbol{x}$. To optimize $\boldsymbol{F}$, gradient descent needs its derivatives $\boldsymbol{\partial F/\partial x}$. The weights $\boldsymbol{x}$ are the matrices $A_1, ..., A_L$, and bias vectors $\boldsymbol{b}_1, ..., \boldsymbol{b}_L$ that take the sample data $\boldsymbol{v} = \boldsymbol{v}_o$ to the output $\boldsymbol{w} = \boldsymbol{v}_L$.

Formulas for $\partial F/\partial A$ and $\partial F/\partial \boldsymbol{b}$ are not difficult. Those formulas are useful to see. But real codes use *automatic differentiation (AD) for backpropagation* (Section 3). Each hidden layer with its optimized weights learns more about data and the population from which it comes–in order to classify new and unseen data from the same population.

2.0.1 The functions Deep Learning

Suppose one of the digits $0, 1, ..., 9$ is drawn in a square. How does a person recognize which digit it is? That neuroscience question is not answered here. How can a computer recognize which digit it is? This is a machine learning question. Probably both answers begin with the same idea: *Learn from examples.*

So we start with M different images (the training set). An image will be a set of p small pixels or a vector $\boldsymbol{v} = (v_1, ..., v_p)$. The component v_i tells us the "grayscale" of the ith pixel in the image: how dark or light it is. So we have M images each with p features: M vectors $\boldsymbol{v}$ in p-dimensional space. For every $\boldsymbol{v}$ in that training set we know the digit it represents.

In a way, we know a function. We have M inputs in $\boldsymbol{R}^p$ each with an output from 0 to 9. But we don't have a "rule". We are helpless with a new input. Machine learning proposes to create a rule that succeeds on (most of) the training images. But "succeed" means much more than that: The rule should give the correct digit for a much wider set of test images, taken from the same population. This essential requirement is called *generalization*.

What form shall the rule take? Here we meet the fundamental question. Our first answer might be: $F(\boldsymbol{v})$ could be a linear function from $\boldsymbol{R}^p$ to $\boldsymbol{R}^1 0$ (a 10 by p matrix). The 10 outputs would be probabilities of the numbers 0 to 9. We would have $10p$ entries and M training samples to get mostly right.

The difficulty is: Linearity is far too limited. Artistically, two zeros could make an 8. 1 and 0 could combine into a handwritten 9 or possibly 6. Images don't add. In recognizing faces instead of numbers, we will need a lot of pixels–and the input-output rule is nowhere near linear.

Artificial intelligence languished for a generation, waiting for new ideas. There is no claim that the absolutely best class of functions has now been found. That class needs to allow a great many parameters (called weights). And it must remain feasible to compute all those weights (in a reasonable time) from knowledge of the training set.

The choice that has succeeded beyond expectation–and has turned shallow learning into deep learning is *Continuous Piecewise Linear (CPL) functions*. **Linear** for simplicity, **continuous** to model an unknown but reasonable rule, and **piecewise** to achieve the nonlinearity that is an absolute requirement for real images and data.

This leaves the crucial question of computability. What parameters will quickly describe a large family of CPL functions? Linear finite elements start with a triangular mesh. But specifying many individual nodes in $\boldsymbol{R}^P$ is expensive. Much better if those nodes are the *intersections* of a smaller number of lines (or hyperplanes). Please know that a regular grid is too simple.

Here is a first construction of a piecewise linear function of the data vector $\boldsymbol{v}$. Choose a matrix A_1 and vector by $\boldsymbol{b}_1$. Then set to zero (this is the nonlinear step) all negative components of $A_1\boldsymbol{v}+\boldsymbol{b}_1$. Then multiply by a matrix A_2 to produce 10 outputs in $\boldsymbol{w} = F(\boldsymbol{v}) = A_2(A_1\boldsymbol{v} + \boldsymbol{b}_1)_+$. That vector $(A_1\boldsymbol{v} + \boldsymbol{b}_1)_+$ forms a "hidden layer" between the input $\boldsymbol{v}$ and the output $\boldsymbol{w}$.

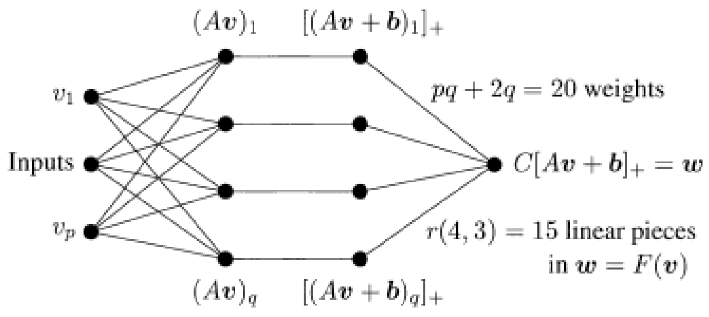

$(A\boldsymbol{v})_1$
$[(A\boldsymbol{v}+\boldsymbol{b})_1]_+$
v_1
$pq + 2q = 20$ weights
Inputs
$C[A\boldsymbol{v}+\boldsymbol{b}]_+ = \boldsymbol{w}$
v_p
$r(4,3) = 15$ linear pieces in $\boldsymbol{w} = F(\boldsymbol{v})$
$(A\boldsymbol{v})_q$
$[(A\boldsymbol{v}+\boldsymbol{b})_q]_+$

Figure 2.1

Actually the nonlinear function called **ReLU** $(x) = x_+ = \max\ (x, 0)$ was originally smoothed into a logistic curve like $1/(1 + e^{-x})$. It was reasonable to think that continuous derivatives would help in optimizing the weights $A_1, \boldsymbol{b}_1, A_2$. That proved to be wrong.

The graph of each component of $(A_1\boldsymbol{v} + \boldsymbol{b}_1)_+$ has two halfplanes (one is flat, from the zeros where $A_1\boldsymbol{v} + \boldsymbol{b}_1$ is negative). If A_1 is q by p, the input space $\boldsymbol{R}^p$ is sliced by q hyperplanes into r pieces. We can count those pieces! This measures the "expressivity" of the overall function $F(\boldsymbol{v})$. The formula from combinatorics uses the binomial coefficients (see Section 1):

$$r(q,p) = \binom{q}{0} + \binom{q}{1} + ... + \binom{q}{p}$$

This number gives an impression of the graph of F. But our function is not yet sufficiently expressive, and one more idea is needed.

Here is the indispensable ingredient in the learning function F. The best way to create complex functions from simple functions is by **composition**. Each F_i is linear (or affine) followed by the nonlinear **ReLU** : $F_i(\boldsymbol{v}) = (A_i\boldsymbol{v} + \boldsymbol{b}_i)_+$. Their composition is $F(\boldsymbol{v}) = F_L(F_{L-1}(...F_2(F_1(\boldsymbol{v}))))$. We now have $L - 1$ hidden layers before the final output layer. The network becomes deeper as L increases. That depth can grow quickly for convolutional nets (with banded Toeplitz matrices A).

The great optimization problem of deep learning is to compute weights A_i and $\boldsymbol{b}_i$ that will make the outputs $F(\boldsymbol{v})$ nearly correct—close to the digit $w(\boldsymbol{v})$ that the image $\boldsymbol{v}$ represents. This problem of minimizing some measure of $F(\boldsymbol{v}) - w(\boldsymbol{v})$ is solved by following a gradient downhill. The gradient of this complicated function is computed by *backpropagation*–the workhorse of deep learning that executes the chain rule.[13]

A historic competition in 2012 was to identify the 1.2 million images collected in ImageNet. The breakthrough neural network in AlexNet had 60 million weights. Its accuracy (after 5 days of stochastic gradient descent) cut in half the next best error rate. Deep learning had arrived.

Our goal here was to identify continuous piecewise linear functions as powerful approximators. That family is also convenient–closed under addition and maximization and composition. The magic is that the learning function $F(A_i, \boldsymbol{b}_i, \boldsymbol{v})$ gives accurate results on images $\boldsymbol{v}$ that F has never seen.

This two-page essay was written for SIAM News (December 2018).

2.0.2 Bias vs. Variance: Underfit vs. Overfit

A **training set** contains N vectors $\boldsymbol{v}_1, ..., \boldsymbol{v}_N$ with m components each (the m features of each sample). For each of those N points in $\boldsymbol{R}^m$, we are given a value y_i. We assume there is an unknown function $f(\boldsymbol{x})$ so that $y_i = f(\boldsymbol{x}_i) + \epsilon_i$, where the noise ϵ has zero mean and variance σ^2. That is the function $f(\boldsymbol{x})$ that our algorithms try to learn.

Our learning algorithm actually finds a function $F(\boldsymbol{x})$ close to $f(\boldsymbol{x})$. For example, F from our learning algorithm could be linear (not that great) or piecewise linear (much better)–this depends on the algorithm we use. We fervently hope that $F(\boldsymbol{x})$ will be close to the correct $f(\boldsymbol{x})$ **not only on the training samples but also for later test samples**.

The warning is often repeated, and always the same: **Don't overfit the data**. The option is there, to reproduce all known observations. It is more important to prevent large swings in the learning function (which is built from the weights). This function is going to be applied to new data. Implicitly or explicitly, we need to **regularize** this function F.

Ordinarily, we regularize by adding a penalty term like $\lambda||\boldsymbol{x}||$ to the function that we are minimizing. This gives a smoother and more stable solution as the minimum point. For deep learning problems this isn't always necessary! We don't fully understand why steepest descent or stochastic steepest descent will find a near minimum that generalizes well to unseen test data–with no penalty term. Perhaps the success comes from following this rule: *Stop the minimization early before you overfit*.

If F does poorly on training samples with large error (**bias**), that is $underfitting$

If F does well on the training samples but not well on test samples, that's $overfitting$

This is the **bias-variance tradeoff**. high bias from underfitting, high variance from overfitting. Suppose we scale f and F so that $\mathrm{E}[F(\boldsymbol{x})] = 1$.

$$\textbf{Bias} = \mathrm{E}[f(\boldsymbol{x}) - F(\boldsymbol{x})] \quad \textbf{Variance}= \mathrm{E}[(F(\boldsymbol{x}))^2] - (\mathrm{E}[F(\boldsymbol{x})])^2$$

We are forced into this tradeoff by following the given identity $(\text{Bias})^2$+(Variance) +$(\text{Noise})^2$:

$$\mathrm{E}[(y - F(\boldsymbol{x}))^2] = (\mathrm{E}[f(\boldsymbol{x}) - F(\boldsymbol{x})])^2 + \mathrm{E}[(F(\boldsymbol{x}))^2] - (\mathrm{E}[F(\boldsymbol{x})])^2 + \mathrm{E}[(y - f(\boldsymbol{x}))^2]$$

Again, *bias* comes from allowing less freedom and using fewer parameters (weights). Variance is large when we provide too much freedom and too many parameters for F. Then the learned function F can be super-accurate on the training set but out of control on an unseen test set. **Overfitting produces an *F* that does not generalize.**

Here are links to six sites that support codes for machine learning:

1. Caffe : arXiv:1408.5093
2. Keras : http://Keras.io
3. MatConvNet : www.vlfeat.org/matconvnet
4. Theano : arXiv: 1605.02688
5. Torch : torch.ch
6. TensorFlow : www.tensorflow.org

2.1 The Construction of Deep Neural Networks

Deep neural networks have evolved into a major force in machine learning. Step by step, the structure of the network has become more resilient and powerful– and more easily adapted to new applications. One way to begin is to describe essential pieces in the structure. Those pieces come together into a **learning function $\boldsymbol{F}(\boldsymbol{x}, \boldsymbol{v})$** *with weights $\boldsymbol{x}$ that capture information from the training data* $\boldsymbol{v}$–to prepare for use with new test data.

Here are important steps in creating that function F:

1 Key operation **Composition $\boldsymbol{F} = \boldsymbol{F_3}(\boldsymbol{F_2}(\boldsymbol{F_1}(\boldsymbol{x}, \boldsymbol{v})))$**

2 Key rule **Chain rule for $\boldsymbol{x}$-derivatives of $\boldsymbol{F}$**

3 Key algorithm **Stochastic gradient descent to find the best weights $\boldsymbol{x}$**

4 Key subroutine **Backpropagation to execute the chain rule**

5 Key nonlinearity **ReLU$(\boldsymbol{y})$ = max$(\boldsymbol{y}, \mathbf{0})$ = ramp function**

Our first step is to describe the pieces $F_1, F_2, F_3, ...$ for one layer of neurons at a time. The weights $\boldsymbol{x}$ that connect the layers $\boldsymbol{v}$ are optimized in creating $\boldsymbol{F}$. The vector $\boldsymbol{v} = \boldsymbol{v}_o$ comes from the training set, and the function F_k produces the vector $\boldsymbol{v}_k$ at layer k. The whole success is to build the power of F from those pieces F_k in equation (1).

2.1.1 F_k is a Piecewise Linear Function of v_{k-1}

The input to F_k is a vector $\boldsymbol{v}_{k-1}$ of length N_{k-1}. The output is a vector $\boldsymbol{v}_k$ of length N_k, ready for input to F_{k+1}. This function F_k hs two parts, first linear and then nonlinear:

1. The linear part of F_k yields $A_k\boldsymbol{v}_{k-1} + \boldsymbol{b}_k$ (that bias vector $\boldsymbol{b}_k$ makes this "affine")
2. A fixed nonlinear function like ReLU is apllied to *each component* of $A_k\boldsymbol{v}_{k-1}+\boldsymbol{b}_k$

$$\boxed{\boldsymbol{v}_k = F_k(\boldsymbol{v}_{k-1}) = \text{ReLU }(A_k\boldsymbol{v}_{k-1} + \boldsymbol{b}_k)} \tag{1}$$

The training data for each sample is in a feature vector $\boldsymbol{v}_o$. The matrix $A-k$ has shape N_k by N_{k-1}. The column vector $\boldsymbol{b}_k$ has N_k components. **These $\boldsymbol{A_k}$ and $\boldsymbol{b_k}$ are weights** constructed by the optimization algorithm. Frequently *stochastic gradient descent* computes optimal weights $\boldsymbol{x} = (A_1, \boldsymbol{b}_1, ..., A_L, b_L)$ in the central computation of deep learning. It relies on backpropagation to find the $\boldsymbol{x}$-derivatives of F, to solve $\boldsymbol{\nabla F = 0}$.

The activation function $\textbf{ReLU}(y) = \max(y, 0)$ gives flexibility and adaptability. Linear steps alone were of limited power and ultimately they were unsuccessful.

ReLU is applied to every "neuron" in every internal layer. There are N_k neurons in layer k, containing the N_k outputs from $A_k\boldsymbol{v}_{k-1}+\boldsymbol{b}_k$. Notice that ReLU itself is continuous and piecewise linear, as its graph shows. (The graph is just a ramp with slopes 0 and 1. Its derivative is the usual step function.) When we choose ReLU, the composite function $F = F_L(F_2(F_1(\boldsymbol{x}, \boldsymbol{v})))$ has an important and attractive property:

$$\boxed{\textbf{The learning function } \boldsymbol{F} \textbf{ is continuous and piecewise linear in } \boldsymbol{v}\textbf{.}}$$

2.1.2 One Internal Layer ($L = 2$)

Suppose we have measured $m = 3$ features of one sample point in the training set. Those features are the 3 components of the input vector $\boldsymbol{v} = \boldsymbol{v}_o$. Then the first function F_1 in chain multiplies $\boldsymbol{v}_o$ by matrix A_1 and adds an offset vector $\boldsymbol{b}_1$ (bias vector). If A_1 is 4 by 3 and vector $\boldsymbol{b}_1$ is 4 by 1, we have 4 components of $A_o\boldsymbol{v}_o + \boldsymbol{b}_o$.

That step found 4 combinations of the 3 original features in $\boldsymbol{v} = \boldsymbol{v}_o$. The 12 weights in matrix A_1 were optimized over many feature vectors $\boldsymbol{v}_o$ in the training set, to choose a 4 by 3 matrix (and a 4 by 1 bias vector) that would find 4 insightful combinations.

The final step to reach $\boldsymbol{v}_1$ is to apply the nonlinear ”activation function” to each of the 4 components of $A_1\boldsymbol{v}_o + \boldsymbol{b}_1$. Historically, the graph of that nonlinear function was often given by a smooth ”$S - curve$”. Particular choices then and now are in Figure given below.

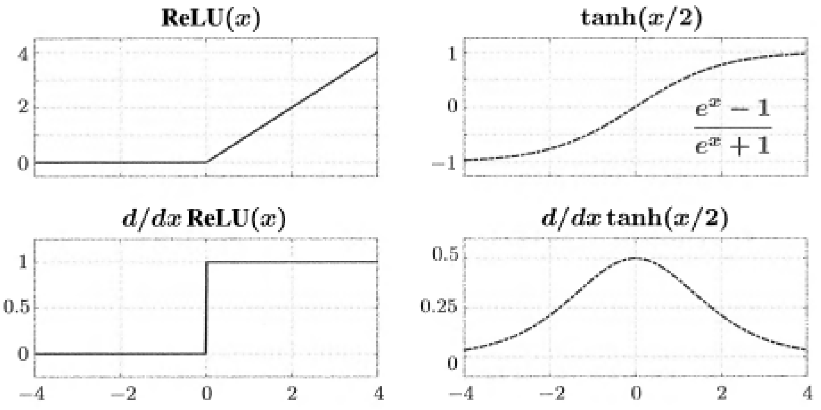


Figure 2.2: The **Re**ctified **L**inear **U**nit and a sigmoid option for nonlinearity.

Previously it was thought that a sudden change of slope would be dangerous and possibly unstable. But large scale numerical experiments indicated otherwise! A better result was achieved by the **ramp function** ReLU$(y) = \max(y, 0)$. We will work with ReLU:

$$\boxed{\begin{array}{c}\textbf{Substitute } \boldsymbol{A_1 v_o} + \boldsymbol{b_1} \textbf{ into ReLU to find } \boldsymbol{v}_1 \\ (\boldsymbol{v_1})_k = \max((A_1\boldsymbol{v_o} + \boldsymbol{b}_1)_k, 0).\end{array}} \tag{2}$$

Now we have the components of $\boldsymbol{v}_1$ at the four "neurons" in layer 1. The input layer held the three components of this particular sample of training data. We may have thousands or millions of samples. The optimization algorithm found A_1 and $\boldsymbol{b}_1$, possibly by stochastic gradient descent using backpropagation to compute gradients of the overall loss.

Suppose our neural net is shallow instead of deep. It only has this first layer of 4 neurons. Then the final step will multiply the 4-component vector $\boldsymbol{v}_1$ by a 1 by 4 matrix A_2 (a row vector). It can add a single number $\boldsymbol{b}_2$ to reach the value $\boldsymbol{v}_2 = A_2\boldsymbol{v}_1 + \boldsymbol{b}_2$. The nonlinear function ReLU is not applied to the output.

$$\boxed{\begin{array}{l}\text{Overall we compute } \boldsymbol{v}_2 = F(\boldsymbol{x}, \boldsymbol{v_o}) \text{ for each feature vector } \boldsymbol{v}_o \text{ in training set.}\\ \text{The steps are } \boldsymbol{v}_2 = A_2\boldsymbol{v_1} + \boldsymbol{b}_2 = A_2(\text{ReLU}(A_1\boldsymbol{v}_o + \boldsymbol{b}_1)) + \boldsymbol{b}_2 = F(\boldsymbol{x}, \boldsymbol{v_o}).\end{array}} \quad (3)$$

The goal in optimizing $\boldsymbol{x} = A_1, \boldsymbol{b}_1, A_2, \boldsymbol{b}_2$ is that the output values $\boldsymbol{v}_l = \boldsymbol{v}_2$ at last layer $l = 2$ should correctly capture the important features of training data $\boldsymbol{v}_o$.

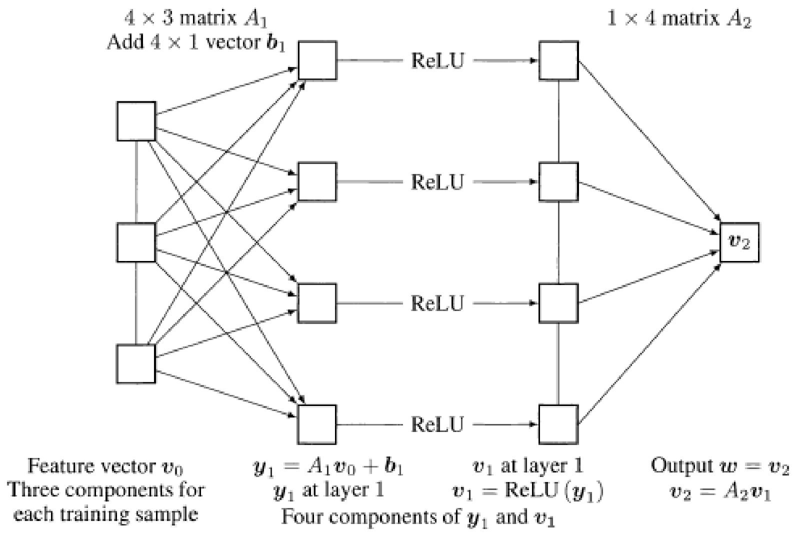

4×3 matrix A_1
Add 4×1 vector $\boldsymbol{b}_1$
ReLU
ReLU
ReLU
ReLU
1×4 matrix A_2
$\boldsymbol{v}_2$
Feature vector $\boldsymbol{v}_0$
Three components for each training sample
$\boldsymbol{y}_1 = A_1\boldsymbol{v}_0 + \boldsymbol{b}_1$
$\boldsymbol{y}_1$ at layer 1
Four components of $\boldsymbol{y}_1$ and $\boldsymbol{v}_1$
$\boldsymbol{v}_1$ at layer 1
$\boldsymbol{v}_1 = \text{ReLU}(\boldsymbol{y}_1)$
Output $\boldsymbol{w} = \boldsymbol{v}_2$
$\boldsymbol{v}_2 = A_2\boldsymbol{v}_1$

Figure 2.3: A feed-forward neural net with 4 neurons on **one internal layer.** The output $\boldsymbol{v}_2$ (plus or minus) classifies the input $\boldsymbol{v}_o$ (dog or cat). Then $\boldsymbol{v}_2$ is a composite measure of the 3-component feature vector $\boldsymbol{v}_o$. This net has 20 weights in A_k, and b_k.

For a **classification problem** each sample $\boldsymbol{v}_o$ of the training data is assigned **1 or -1**. We want the output $\boldsymbol{v}_2$ to have that correct sign (most of the time). For a **regression problem** we use the numerical value (**not just the sign**) of $\boldsymbol{v}_2$. We do not choose enough weights A_k and $\boldsymbol{b}_k$ to get every sample correct. And we do not necessarily want to! That would probably be **overfitting the training data**. It could give erratic results when F is applied to new and unknown test data.

Depending on our choice of loss function $L(x, \boldsymbol{v}_2)$ to minimize, this problem can be like least squares or entropy minimization. We are choosing $\boldsymbol{x}$ = weight matrices A_k and bias vectors $\boldsymbol{b}_k$ to minimize L. Those two loss functions—square loss and cross-entropy loss—are compared in Section 4.

Our hope is that the **function $\boldsymbol{F}$ has "learned" the data.** This is machine learning. We don't want to choose so many weights in $\boldsymbol{x}$ that every input sample is sure to be correctly classified. *That is not learning.* That is simply fitting (overfitting) the data.

We want a balance where the function F has learned what is important in recognizing *dog versus cat or identifying an oncoming car versus a turning car.*

Machine learning doesn't aim to capture every detail of the numbers 0, 1, 2...,9. It just aims to capture enough information to decide correctly *which number it is.*

2.1.3 The Initial Weights x_o in Gradient Descent

The architecture in a neural net decides the form of the learning function $F(\boldsymbol{x}, \boldsymbol{v})$. The training data goes into $\boldsymbol{v}$. Then we initialize the weights $\boldsymbol{x}$ in the matrices A and vectors $\boldsymbol{b}$. From those initial weights $\boldsymbol{x}_o$, the optimization algorithm (normally a form of gradient descent) computes weights $\boldsymbol{x}_1$ and $\boldsymbol{x}_2$ and onward, aiming to minimize the total loss.

The question is: *What weights $\boldsymbol{x}_o$ to start with?* Choosing $\boldsymbol{x}_o = 0$ would be a disaster. Poor initialization is an important cause of failure in deep learning. A proper choice of the net and the initial $\boldsymbol{x}_o$ has random (and independent) weights that meet two requirements:

1. $\boldsymbol{x}_o$ has a carefully chosen variance σ^2.

2. The hidden layers in the neural net have enough neurons (not too narrow).

Hanin and Rolnick show that the initial variance σ^2 controls the mean of the computed weights. The layer widths control the variance of the weights. The key point is this: *Many-layered depth can reduce the loss on the training set. But if σ^2 is wrong or width is sacrificed, then gradient descent can lose control of the weights. They can explode to infinity or implode to zero.*

The danger controlled by the variance σ^2 of $\boldsymbol{x}_o$ is exponentially large or exponentially small weights. The good choice is $\sigma^2 = 2/\text{fan-in}$. The fan-in is the maximum number of inputs to neurons (Above Figure has fan-in = 4 at the output). The initialization "He uniform" in Keras makes this choice of σ^2.

The danger from narrow hidden layers is exponentially large variance of $\boldsymbol{x}$ for deep nets. If layer j has n_j, neurons, the quantity to control is the sum of 1/(layer widths n_j).

Looking ahead, convolutional nets (ConvNets) and residual networks (ResNets) can be very deep. Exploding or vanishing weights is a constant danger. Ideas from physics (mean field theory) have become powerful tools to explain and also avoid these dangers. Pennington and coauthors proposed a way to stay on the edge between fast growth and decay, even for 10,000 layers. A key is to use orthogonal transformations: Exactly as in matrix multiplication $Q_1Q_2Q_3$, orthogonality leaves the size unchanged.

For ConvNets, fan-in becomes the number of features times the kernel size (and not the full size of A). For ResNets, a correct σ^2 normally removes both dangers. very deep networks can produce very impressive learning

The key point : Deep learning can go wrong if it doesn't start right.

2.1.4 Stride and Subsampling

Those words represent two ways to achieve the same goal: *Reduce the dimension.* Suppose we start with a 1D signal of length 128. We want to filter that signal– multiply that vector by a weight matrix A. We also want to reduce the length to

64. Here are two ways to reach that goal.

In two steps Multiply the 128-component vector v by A, and then discard the odd-numbered components of the output. This is filtering followed by subsampling. The output is $(\downarrow 2)\ A\boldsymbol{v}$.

In one step Discard the odd-numbered rows of the matrix A. The new matrix A_2 becomes short and wide: 64 rows and 128 columns. The "**stride**" of the filter is now 2. Now multiply the 128-component vector $\boldsymbol{v}$ by A_2. Then $A_2\boldsymbol{v}$ is the same as $(\downarrow 2)\ A\boldsymbol{v}$. A stride of 3 would keep every third component.

Certainly the one-step striding method is more efficient. If the stride is 4, the dimension is divided by 4. In two dimensions (for images) it is reduced by 16.

The two-step method makes clear that half or three-fourths of the information is lost. Here is a way to reduce the dimension from 128 to 64 as before, but to run less risk of destroying important information: *Max-pooling.*

2.1.5 Max-pooling

Multiply the 128-component vector $\boldsymbol{v}$ by A, as before. Then from each even-odd pair of outputs like $(A\boldsymbol{v})_2$ and $(A\boldsymbol{v})_3$, *keep the maximum.* Please notice right away: Max-pooling is simple and fast, **but taking the maximum is not a linear operation**. It is a sensible route to dimension reduction, pure and simple.

For an image (a 2-dimensional signal) we might use max-pooling over every 2 by 2 square of pixels. Each dimension is reduced by 2, The image dimension is reduced by 4. This speeds up the training, when the number of neurons on a hidden layer is divided by 4.

Normally a max-pooling step is given its own separate place in the overall architecture of the neural net. Thus a part of that architecture might look like this:

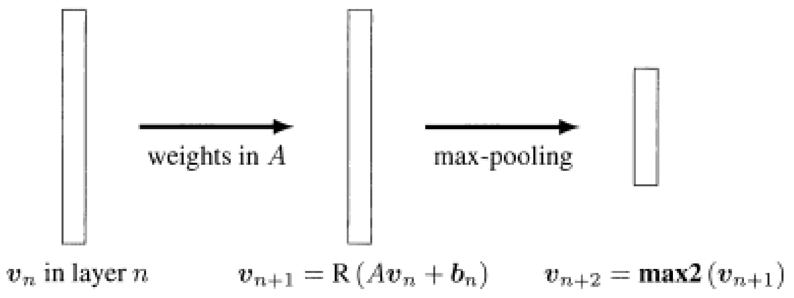

weights in A
max-pooling
v_n in layer n
$v_{n+1} = \mathrm{R}(Av_n + b_n)$
$v_{n+2} = \mathbf{max2}(v_{n+1})$

Figure 2.4

Dimension reduction has another important advantage, in addition to reducing the computation. *Pooling also reduces the possibility of overfitting.* Average pooling would keep the *average* of the numbers in each pool : now the pooling layer is linear.

2.1.6 The Graph of the Learning Function $F(v)$

The graph of $F(\boldsymbol{v})$ is a surface made up of many, many flat pieces—they are planes or hyperplanes that fit together along all the folds where ReLU produced a change of slope. This is like origami except that this graph has flat pieces going to infinity. And the graph might not be in $\boldsymbol{R}^3$—the feature vector $\boldsymbol{v} = \boldsymbol{v}_o$ has $N_o = m$ components.

Part of the *mathematics of deep learning* is to estimate the number of flat pieces and to visualize how they fit into one piecewise linear surface. That estimate comes after an example of a neural net with one internal layer. Each feature vector $\boldsymbol{v}_o$ contains m measurements like height, weight, age of a sample in the training set.

In the example, F had three inputs in $\boldsymbol{v}_o$ and one output $\boldsymbol{v}_2$. Its graph will be a piecewise flat surface in 4-dimensional space. The height of the graph is $\boldsymbol{v}_2 = F(\boldsymbol{v}_o)$, over the point $\boldsymbol{v}_o$ in 3-dimensional space. Limitations of space in the book (and severe limitations of imagination in the author) prevent us from drawing that graph in $\boldsymbol{R}^4$. Nevertheless we can try to count the flat pieces, based on 3 inputs and 4 neurons and 1 output.

Note **1** With only $m = 2$ inputs (2 features for each training sample) the graph of F

is a surface in 3D. We can and will make an attempt to describe it.

Note **2** You actually see points on the graph of F when you run examples on

playground.tensorflow.org. This is a very instructive website.

That website offers four options for the training set of points v_o. You choose the number of layers and neurons. Please choose the ReLU activation function! Then the program counts epochs as gradient descent optimizes the weights. (An *epoch sees* all samples on average once.) If you have allowed enough layers and neurons to correctly classify the blue and orange training samples, you will see a polygon separating them. **That polygon shows where $\boldsymbol{F = 0}$** . It is the cross-section of the graph of $z = F(\boldsymbol{v})$ at height $z = 0$.

That polygon separating blue from orange (or *plus* from *minus*: this is classification) is the analog of a separating hyperplane in a Support Vector Machine. If we were limited to linear functions and a straight line between a blue ball and an orange ring around it, separation would be impossible. But for the deep learning function F this is not difficult...

We will discuss experiments on this **playground.tensorflow** site in the Problem Set.

Important Note: Fully Connected versus Convolutional

We don't want to mislead the reader. Those "fully connected" nets are often not the most effective. If the weights around one pixel in an image can be repeated around all pixels (why not?), then one row of A is all we need. The row can assign zero weights to faraway pixels. Local convolutional neural nets (CNN's) are the subject of Section 2.

You will see that the count grows exponentially with the number of neurons and layers. That is a useful insight into the power of deep learning. We badly need insight because the size and depth of the neural network make it difficult to visualize in full detail.

2.1.7 Counting Flat Pieces in the Graph: One Internal Layer

It is easy to count entries in the weight matrices A_k and the bias vectors $\boldsymbol{b}_k$. Those numbers determine the function F. But it is far more interesting to count the number of flat pieces in the graph of F. This number measures the **expressivity** of the neural network. $F(\boldsymbol{x}, \boldsymbol{v})$ is a more complicated function than we fully understand (at least so far). The system is deciding and acting on its own, without explicit approval of its "thinking". For driverless cars we will see the consequences fairly soon.

Suppose $\boldsymbol{v}_o$ has m components and $A_1\boldsymbol{v}_o+\boldsymbol{b}_1$ has N components. We have N functions of $\boldsymbol{v}_o$. Each of those linear functions is zero along a hyperplane (dimension $m-1$) in $\boldsymbol{R}^m$. When we apply ReLU to that linear function it becomes piecewise linear, with a fold along that hyperplane. On one side of the fold its graph is sloping, on the other side the function changes from negative to zero.

Then the next matrix A_2 combines those N piecewise linear functions of v_o, so we now have folds along *N different hyperplanes in $\boldsymbol{R}^m$*. This describes each piecewise linear component of the next layer $A_2(\text{ReLU}(A_1\boldsymbol{v}_o + \boldsymbol{b}_1))$ in the typical case.

You could think of N straight folds in the plane (the folds are actually along N hyper- planes in m-dimensional space). The first fold separates the plane in two pieces. The next fold from ReLU will leave us with four pieces. The third fold is more difficult to visualize, but the following figure shows that there are seven (*not eight*) pieces.

In combinatorial theory, we have a **hyperplane arrangement**–and a theorem of Tom Zaslavsky counts the pieces. The proof is presented in Richard Stanley's great textbook on *Enumerative Combinatorics (2001).* But that theorem is more complicated than we need, because it allows the fold lines to meet in all possible ways. Our task is simpler because we assume that the fold lines are in "general position"$-m+1$ folds don't meet. For this case we now apply the neat counting argument given by Raghu, Poole, Kleinberg, Gangul, and Dickstein: *On the Expressive Power of Deep Neural Networks*, arXiv: 1606.05336v6: See also The Number of Response Regions by Pascanu, Montufar, and Bengio on arXiv 1312.6098.

Theorem For v in $\boldsymbol{R}^m$, suppose the graph of $F(\mathbf{v})$ has folds along N hyperplanes $H_1, ..., H_N$. Those come from N linear equations $\boldsymbol{a}_i^T\boldsymbol{v}+b_i = 0$, in other words from ReLU at N neurons. Then the number of linear pieces of F and regions bounded by the N hyperplanes is $r(N, m)$:

$$\boldsymbol{r(N,m)} = \sum_{i=0}^{m} \binom{\boldsymbol{N}}{\boldsymbol{i}} = \binom{\boldsymbol{N}}{\boldsymbol{0}} + \binom{\boldsymbol{N}}{\boldsymbol{1}} + ... + \binom{\boldsymbol{N}}{\boldsymbol{m}} \tag{4}$$

These binomial coefficients are

$$\binom{\boldsymbol{N}}{\boldsymbol{i}} = \frac{N!}{i!(N-1)!} \text{ with } 0! = 1 \text{ and } \binom{\boldsymbol{N}}{\boldsymbol{0}} = 1 \text{ and} \binom{\boldsymbol{N}}{\boldsymbol{i}} = 0 \text{ for } i > N$$

Example The function $F(x, y, z) =$ ReLU (x)+ReLU (y) +ReLU (z) has 3 folds along the 3 planes $x = 0$, $y = 0$, $z = 0$. Those planes divide $\boldsymbol{R}^3$ into $r(3, 3) = 8$ pieces where $F = x + y + z$ and $x + z$ and x and 0 (and 4 more). Adding ReLU $(x + y + z - 1)$ gives a fourth fold and $r(4, 3) = 15$ pieces of $\boldsymbol{R}^3$. Not 16 because the new fold plane $x + y + z = 1$ does not meet the 8th original piece where $x < 0, y < 0, z < 0$.

George Polya's famous YouTube video *Let Us Teach Guessing* cut a cake by 5 planes. He helps the class to find $r(5, 3) = 26$ pieces. Formula (4) allows m-dimensional cakes.

One hyperplane in $\boldsymbol{R}^m$ produces $\binom{1}{0} + \binom{1}{1} = 2$ regions. And $N = 2$ hyperplanes will produce $r(2, m) = 1 + 2 + 1 = 4$ regions provided $m > 1$. When $m = 1$ we have two folds in a line, which only separates the line into $r(2, 1) = 3$ pieces.

The count r of linear pieces will follow from the recursive formula

$$\boxed{r(N, m) = r(N - 1, m) + r(N - 1, m - 1).} \tag{5}$$

To understand that recursion, start with $N-1$ hyperplanes in $\boldsymbol{R}^m$ and $r(N-1,m)$ regions. *Add one more hyperplane H (dimension m - 1).* The established $N-1$ hyperplanes cut H into $r(N-1,m-1)$ regions. Each of those pieces of H divides one existing region into two, adding $r(N-1,m-1)$ regions to the original $r(N-1,m)$; see Figure given below. So the recursion is correct, and we now apply equation (5) to compute $r(N,m)$.

The count starts at $r(1,0) = r(0,1) = \mathbf{1}$. Then (4) is proved by induction on $N+m$:

$$\begin{aligned} r(N-1,m)+r(N-1,m-1) &= \sum_0^m \binom{N-1}{i} + \sum_0^{m-1}\binom{N-1}{i} \\ &= \binom{N-1}{0} + \sum_0^{m-1}\left[\binom{N-1}{i}+\binom{N-1}{i+1}\right] \\ &= \binom{N}{0} + \sum_0^{m-1}\binom{N}{i+1} \\ &= \sum_0^m \binom{\boldsymbol{N}}{\boldsymbol{i}} \end{aligned} \tag{6}$$

The two terms in brackets (second line) became one term because of a useful identity:

$\binom{N-1}{i} + \binom{N-1}{i+1} = \binom{N}{i+1}$ and the induction is complete.

Mike Giles made that presentation clearer, and he suggested Figure below show's the effect of the last hyperplane H. There are $r = 2^N$ linear pieces of $F(\boldsymbol{v})$ for $N \leq m$ and $r \approx N^m/m!$ pieces for $N >> m$, when the hidden layers has many neurons.

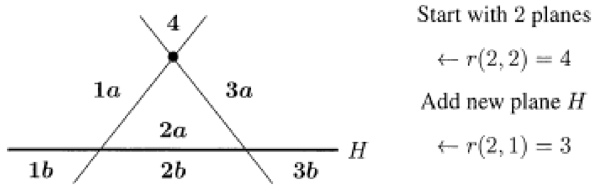


Figure 2.5: The $r(2, 1) = 3$ pieces of H create 3 new regions. Then the count becomes $r(3, 2) = 4 + 37$ flat regions in the continuous piecewise linear surface $\boldsymbol{v}_2 = F(\boldsymbol{v}_o)$. A fourth fold will cross all 3 existing folds and create 4 new regions, so $r(4, 2) = 11$.

2.1.8 Flat Pieces of $F(v)$ with more Hidden Layers

Counting the linear pieces of$F(\boldsymbol{v})$ is much harder with 2 internal layers in the network. Again v_o and v_1 have m and N_1 components. Now $A_1\boldsymbol{v}_1 + \boldsymbol{b}_1$ will have N_2 components before ReLU. Each one is like the function F for one layer, described above. Then appli- cation of ReLU will create new folds in its graph. Those folds are along the lines where a component of $A_1\boldsymbol{v}_1 + \boldsymbol{b}_1$ is zero.

Remember that each component of $A_1\boldsymbol{v}_1 + \boldsymbol{b}_1$ is piecewise linear, not linear. So it crosses zero (if it does) along a piecewise linear surface, not a hyperplane. The straight lines in Above Figure for the folds in $\boldsymbol{v}_1$ will change to *piecewise straight lines* for the folds in v_2. In m dimensions they are connected pieces of hyperplanes. So the count becomes variable, depending on the details of $v_o, A_1, \boldsymbol{b}_1, A_2$, and $\boldsymbol{b}_2$.

Still we can estimate the number of linear pieces. We have N_2 piecewise straight lines (or piecewise hyperplanes in $\boldsymbol{R}^m$) from N_2 ReLU's at the second hidden layer. If those lines were actually straight, we would have a total of $N_1 + N_2$ folds in each component of $\boldsymbol{v}_3 = F(\boldsymbol{v}_o)$. *Then the formula (4) to count the pieces would have $N_1 + N_2$ in place of N.* This is our estimate (open for improvement) with two layers between $\boldsymbol{v}_o$ and $\boldsymbol{v}_3$.

2.1.9 Composition $F_3(F_2(F_1(v)))$

The word "composition" would simply represent "matrix multiplication" if all our functions were linear: $F_k(\boldsymbol{v}) = A_k\boldsymbol{v}$. Then $F(\boldsymbol{v}_o) = A_3A_2A_1\boldsymbol{v}_o$: just one matrix.

For nonlinear F_k the meaning is the same: Compute $\boldsymbol{v}_1 = F_1(\boldsymbol{v}_o)$, then $\boldsymbol{v}_2 = F_2(\boldsymbol{v}_1)$, and finally $\boldsymbol{v}_3 = F_3(\boldsymbol{v}_2)$. This operation of **composition** $\boldsymbol{F_3(F_2(F_1(v_o)))}$ is far more powerful in creating functions than addition!

For a neural network, composition produces continuous piecewise linear functions $F(\boldsymbol{v}_o)$. The 13th problem on Hilbert's list of 23 unsolved problems in 1900 asked a question about *all continuous functions*. A famous generalization of his question was this:

Is every continuous function $F(x, y, z)$ of three variables the composition of

continuous functions $G_1, ..., G_N$ of two variables? The answer is yes.

Hilbert seems to have expected the answer *no*. But a positive answer was given in 1957 by Vladimir Arnold (age 19). His teacher Andrey Kolmogorov had previously created multivariable functions out of 3-variable functions.

Related questions have negative answers. If $F(x, y, z)$ has continuous derivatives, it may be impossible for all the 2-variable functions to have continuous derivatives (Vitushkin). And to construct 2-variable continuous functions $F(x, y)$ as compositions of 1-variable continuous functions (the ultimate 13th problem) you must allow addition. The 2-variable functions xy and x^y use 1-variable functions *exp*, *log*, and *log log*:

$$\boldsymbol{xy = exp(logx + logy)} \quad \text{and} \quad \boldsymbol{x^y = exp(exp(logy + loglogx))} \qquad (7)$$

So much to learn from the Web. A chapter of *Kolmogorov's Heritage in Mathematics* (Springer, 2007) connects these questions explicitly to neural networks.

Is the answer to Hilbert still yes for continuous piecewise linear functions on $\boldsymbol{R}^m$*?*

2.1.10 Neural Nets Give Universal Approximation

The previous paragraphs wandered into the analysis of functions $f(\boldsymbol{v})$ of several variables. For deep learning a key question is the approximation of f by a neural net–when the weights $\boldsymbol{x}$ are chosen to bring $F(\boldsymbol{x}, \boldsymbol{v})$ close to $f(\boldsymbol{v})$.

There is a qualitative question and also a quantitative question:

1 For any continuous function $f(\boldsymbol{v})$ with $\boldsymbol{v}$ in a cube in $\boldsymbol{R}^d$, can a net with enough layers and neurons and weights $\boldsymbol{x}$ give uniform approximation to f within any desired accuracy $\epsilon > 0$? This property is called universality.

$$\boxed{\begin{array}{c}\textbf{If } \boldsymbol{f(v)} \textbf{ is continuous there exists } \boldsymbol{x} \\ \text{so that } |F(\boldsymbol{x},\boldsymbol{v}) - f(\boldsymbol{v})| < \boldsymbol{\epsilon} \textbf{ for all } \boldsymbol{v}.\end{array}} \tag{8}$$

2 If $f(\boldsymbol{v})$ belongs to a normed space S of smooth functions, how quickly does the approximation error improve as the net has more weights?

$$\boxed{\textbf{Accuracy of approximation to } \boldsymbol{f} \ \min_x ||F(\boldsymbol{x},\boldsymbol{v}) - f(\boldsymbol{v})|| \leq C||f||_S} \tag{9}$$

Function spaces S often use the $\boldsymbol{L^2}$ or $\boldsymbol{L^1}$ or $\boldsymbol{L^\infty}$ norm of the function f and its partial derivatives up to order r. Functional analysis gives those spaces a meaning even for non-integer r. C usually decreases as the smoothness parameter r is increased. For continuous piecewise linear approximation over a uniform grid with meshwidth h we often find $C = O(h^2)$.

The response to Question 1 is *yes*. Wikipedia notes that one hidden layer (with enough neurons!) is sufficient for approximation within ϵ. The 1989 proof by George Cybenko used a sigmoid function rather than ReLU, and the theorem is continually being extended. Ding-Xuan Zhou proved that we can require the A_k, to

be convolution matrices (the structure becomes a CNN). Convolutions have many fewer weights than arbitrary matrices—and universality allows many convolutions.

The response to Question **2** by Mhaskar, Liao, and Poggio begins with the degree of approximation to functions $f(v_1, ..., v_d)$ with continuous derivatives of order r. For n *weights the usual error bound is* $Cn^{-r/d}$. The novelty is their introduction of **composite functions** built from 2-variable functions, as in $f(v_1, v_2, v_3, v_4) = f_3(f_1(v_1, v_2), f_2(v_3, v_4))$. For a composite function, the approximation by a hierarchical net is much more accurate. *The error bound becomes* $\boldsymbol{Cn^{-r/2}}$.

The proof applies the standard result for $d = 2$ variables to each function f_1, f_2, f_3. A difference of composite functions is a composite of 2-variable differences.[14][15][16]

2.2 Convolutional Neural Nets

This section is about networks with a different architecture. Up to now, each layer was fully connected to the next layer. If one layer had n neurons and the next layer had m neurons, then the matrix A connecting those layers is m by n. *There were mn independent weights in A*. The weights from all layers were chosen to give a final output that matched the training data. The derivatives needed in that optimization were computed by backpropagation. Now we might have only 3 or 9 independent weights per layer.

That fully connected net will be extremely inefficient for image recognition. First, the weight matrices A will be huge. If one image has 200 by 300 pixels, then its input layer has 60,000 components. The weight matrix A_1 for the first hidden layer has 60,000 columns. The problem is: We are looking for connections between faraway pixels. Almost always, the important connections in an image are **local**.

Text and music have a 1D local structure: a time series

Images have a 2D local structure: 3 copies for red-green-blue

Video has a 3D local structure: Images in a time series

More than this, the search for structure is essentially the same everywhere in the image. There is normally no reason to process one part of a text or image or video differently from other parts. We can use the same weights in all parts: *Share the weights.* The neural net of local connections between pixels is **shift-invariant**: the same everywhere.

The result is a big reduction in the number of independent weights. Suppose each neuron is connected to only E neurons on the next layer, and those connections are the same for all neurons. Then the matrix A between those layers has only E independent weights x. The optimization of those weights becomes enormously faster. In reality we have time to create several different channels with their own E or E^2 weights. They can look for edges in different directions (horizontal, vertical, and diagonal).

In one dimension, a banded shift-invariant matrix is a **Toeplitz matrix** or a **filter**. Multiplication by that matrix A is a **convolution $\boldsymbol{x} * \boldsymbol{v}$**. The network of connections between all layers is a **Convolutional Neural Net (CNN or ConvNet)**.[8] Here $E = 3$.

$$A = \begin{bmatrix} x_1 & x_0 & x_{-1} & 0 & 0 & 0 \\ 0 & x_1 & x_0 & x_{-1} & 0 & 0 \\ 0 & 0 & x_1 & x_0 & x_{-1} & 0 \\ 0 & 0 & 0 & x_1 & x_o & x_{-1} \end{bmatrix} \qquad \begin{array}{c} \boldsymbol{v} = (v_0, v_1, v_2, v_3, v_4, V_5) \\ \boldsymbol{y} = \boldsymbol{Av} = \quad (y_1, y_2, y_3, y_4) \\ N+2 \ \text{ inputs and } \quad N \ \text{ output} \end{array}$$

It is valuable to see A as a *combination of shift matrices* $\boldsymbol{L}, \boldsymbol{C}, \boldsymbol{R}$ *: Left, Center, RIght*

Each shift has a diagonal of 1's $\qquad \boldsymbol{A = x_1 L + x_0 C + x_{-1} R}$

Then the derivatives of $\boldsymbol{y} = A\boldsymbol{v} = \boldsymbol{x_1 L v} + \boldsymbol{x_0 C v} + \boldsymbol{x_{-1} R v}$ are exceptionally simple:

$$\boxed{\frac{\partial \boldsymbol{y}}{\partial x_1} = L\boldsymbol{v} \quad \frac{\partial \boldsymbol{y}}{\partial x_0} = C\boldsymbol{v} \quad \frac{\partial \boldsymbol{y}}{\partial x_{-1}} = R\boldsymbol{v}} \tag{1}$$

2.2.1 Convolutions in Two dimensions

When the input $\boldsymbol{v}$ is an image, the convolution with $\boldsymbol{x}$ becomes two-dimensional. The numbers x_{-1}, x_0, x_1 change to $E^2 = 3^2$ independent weights. The inputs v_{ij} have two indices and $\boldsymbol{v}$ represents $(N+2)^2$ pixels. The outputs have only N^2 pixels unless we pad with zeros at the boundary. [17] The 2D convolution $\boldsymbol{x} * \boldsymbol{v}$ is a *linear combination of 9 shifts.*

$$\textbf{Weights} \begin{bmatrix} x_{11} & x_{01} & x_{-11} \\ x_{01} & x_{00} & x_{-10} \\ x_{1-1} & x_{0-1} & x_{-1-1} \end{bmatrix}$$

Input image v_{ij} i, j from$(0,0)$to$(N+1, N+1)$
Output image y_{ij} i, j from$(1,1)$to(N, N)
Shifts L,C,R,U,D = **L**eft,**C**enter,**R**ight,**U**p,**D**own

$$A = x_{11}LU + x_{01}CU + x_{-11}RU + x_{10}L + x_{00}C + x_{-10}R + x_{1-1}LD + x_{0-1}CD + x_{-1-1}RD$$

This expresses the convolution matrix A as a combination of 9 shifts. The derivatives of the output $\boldsymbol{y} = A\boldsymbol{v}$ are again exceptionally simple. We use these nine derivatives to create the gradients $\boldsymbol{\nabla F}$ and $\boldsymbol{\nabla L}$ that are needed in stochastic gradient descent to improve the weights $\boldsymbol{x}_k$. The next iteration $\boldsymbol{x}_{k+1} = \boldsymbol{x}_k - s\boldsymbol{\nabla L}_k$ has weights that better match the correct outputs from the training data.

These nine derivatives of $\boldsymbol{y} = A\boldsymbol{v}$ are computed inside backpropagation:

$$\frac{\partial \boldsymbol{y}}{\partial x_{11}} = LU\boldsymbol{v} \quad \frac{\partial \boldsymbol{y}}{\partial x_{01}} = CU\boldsymbol{v} \quad \frac{\partial \boldsymbol{y}}{\partial x_{-11}} = RU\boldsymbol{v} \;\; ... \;\; \frac{\partial \boldsymbol{y}}{\partial x_{-1-1}} = RD\boldsymbol{v} \tag{2}$$

CNN's can readily afford to have $\boldsymbol{B}$ **parallel channels** (and that number B can vary as we go deeper into the net). The count of weights in $\boldsymbol{x}$ is so much reduced by weight sharing and weight locality, that we don't need and we can't expect one set of $E^2 = 9$ weights to do all the work of a convolutional net.

Let me highlight the operational meaning of convolution. In 1 dimension, the formal algebraic definition $y_j = \Sigma x_i v_{j-i} = \Sigma x_{j-k} v_k$ involves a "flip" of the $v's$ or the $x's$. This is a source of confusion that we do not need. We look instead at left-right shifts L and R of the whole signal (in 1D) and also up-down shifts U and D in two dimensions. Each shift is a matrix with a diagonal full of 1's. That saves us from the complication of remembering flipped subscripts.

A convolution is combination of shift matrices (producing filter or Toeplitz matrix)
A cyclic convolution is a combination of cyclic shifts (producing a circulant matrix)
A continuous convolution is a continuous combination (an integral) of shifts
In deep learning,coefficients in the combination will be the "weights" to be learned.

2.2.2 Two-dimensional Convolutional Nets

Now we come to the real success of CNN's: **Image recognition**. ConvNets and deep learning have produced a small revolution in computer vision. The applications are to self-driving cars, drones, medical imaging, security, robotics– there is nowhere to stop. Our interest is in the algebra and geometry and intuition that makes all this possible.[18]

In two dimensions (for images) the matrix A is **block Toeplitz**. Each small block is E by E. This is a familiar structure in computational engineering. The count E^2 of independent weights to be optimized is far smaller than for a fully connected network.

The same weights are used around all pixels (shift-invariance). The matrix produces a 2D convolution $\boldsymbol{x} * \boldsymbol{v}$. Frequently A is called a **filter**.

To understand an image, look to see where it changes. *Find the edges.* Our eyes look for sharp cutoffs and steep gradients. Our computer can do the same by creating a filter. The dot products between a smooth function and a moving filter window will be smooth. But when an edge in the image lines up with a diagonal wall, *we see a spike.* Those dot products (fixed image) (moving image) are exactly the "**convolution**" of the two images.

The difficulty with two or more dimensions is that edges can have many directions. We will need horizontal and vertical and diagonal filters for the test images. And filters have many purposes, including smoothing, gradient detection, and edge detection.

1 Smoothing For a 2D functions f, the natural smoother is *convolution with a Gaussian*:

$Gf(x,y) = \frac{1}{2\pi\sigma^2} e^{-(x^2+y^2)/2\sigma^2} * f = \frac{1}{\sqrt{2\pi}\sigma} e^{-x^2/2\sigma^2} * \frac{1}{\sqrt{2\pi}\sigma} e^{-y^2/2\sigma^2} * f(x,y)$

This shows G as a product of 1D smoothers. The Gaussian is everywhere positive, so it is *averaging: Gf* cannot have a larger maximum than f. The filter removes noise (at a price in sharp edges). For small variance σ^2, details become clearer.

For a 2D vector (a matrix f_{ij} instead of a function $f(x,y)$) the Gaussian must become discrete. The perfection of radial symmetry will be lost because the matrix G is square. Here is a 5 by 5 discrete Gaussian $G(E=5)$:

$$G = \frac{1}{273}\begin{bmatrix} 1 & 4 & 7 & 4 & 1 \\ 4 & 16 & 26 & 16 & 4 \\ 7 & 26 & 41 & 26 & 7 \\ 4 & 16 & 26 & 16 & 4 \\ 1 & 4 & 7 & 4 & 1 \end{bmatrix} \approx \frac{1}{289}\begin{bmatrix} 1 \\ 4 \\ 7 \\ 4 \\ 1 \end{bmatrix}\begin{bmatrix} 1 & 4 & 7 & 4 & 1 \end{bmatrix} \tag{3}$$

We also lost our exact product of 1D filters. To come closer, use a larger matrix $G = \boldsymbol{x}\boldsymbol{x}^T$ with $\boldsymbol{x} = (.006, .061, .242, .383, .242, .061, .006)$ and discard the small outside pixels.

2 Gradient detection Image processing (as distinctfrom learning by a CNN) needs filters that detect the gradient. They contain specially chosen weights. We mention some simple filters just to indicate how they cam find gradients– the first derivatives of $\boldsymbol{f}$.

One dimension
$\boldsymbol{E = 3}$

$$(x_1, x_0, x_{-1}) = \left(\boldsymbol{-\tfrac{1}{2}, 0, \tfrac{1}{2}}\right) \quad \left[\left(-\tfrac{1}{2}, 0, \tfrac{1}{2}\right) \text{ in convolution form}\right]$$

In this case the components of $A\boldsymbol{v}$ are centered differences: $(A\boldsymbol{v})_i = \frac{1}{2}v_{i+1} - \frac{1}{2}v_{i-1}$. When the components of $\boldsymbol{v}$ are increasing linearly from left to right, as in $\boldsymbol{v}_i = 3i$, the output from the filter is $\frac{1}{2}3(i+1) - \frac{1}{2}3(i-1) = 3 =$ *correct gradient.*

The flip to $(\frac{1}{2}, 0, -\frac{1}{2})$ comes from the definition of convolution as $\Sigma x_{i-k}v_k$

Two dimension These $3x3$ *Sobel operators* approximate $\partial/\partial x$ and $\partial/\partial y$:

$$\boldsymbol{E=3} \qquad \frac{\boldsymbol{\partial}}{\boldsymbol{\partial x}} \approx \frac{1}{2}\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \qquad \frac{\boldsymbol{\partial}}{\boldsymbol{\partial y}} \approx \frac{1}{2}\begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix} \tag{4}$$

For functions, the gradient vector $\boldsymbol{g} = \textit{grad} f$ has $||\boldsymbol{g}||^2 = |\partial f/\partial x|^2 + |\partial f/\partial y|^2$.

Those weights were created for image processing, to locate the most important features of a typical image: *its edges.* These would be candidates for E by E filters inside a 2D convolutional matrix A. But remember that in deep learning, weights like $\frac{1}{2}$ and $-\frac{1}{2}$ are not chosen by the user. They are created from the training data.

Shift-invariance led to the application of discrete Fourier transforms. But in a CNN, ReLU is likely to act on each neuron. The network may include zero-padding–as well as max-pooling layers. So we cannot expect to apply the full power of Fourier analysis.

3 Edge detection After the gradient direction is estimated, we look for edges–the most valuable features to know. "*Canny Edge Detection*"is a highly developed process. Now we don't want smoothing, which would blur the edge. The good filters become *Laplacians of Gaussians:*

$$E \;\; f(x,y) = \nabla^2[g(x,y) * f(x,y)] = [\nabla^2 g(x,y) * f(x,y)]. \tag{5}$$

The Laplacians $\nabla^2 G$ of a gaussian is $(x^2 + y^2 - 2\sigma^2)e^{-(x^2+y^2)/2\sigma^2}/\pi\sigma^4$

2.2.3 The Stride of a Convolutional Filter

Important The filters described so far all have a stride $\boldsymbol{S} = 1$. For a larger stride, the *moving window takes longer steps* as it moves across the image. Here is the matrix A for a 1-dimensional 3-weight filter with a stride of 2. Notice especially that the length of the output $\boldsymbol{y} = A\boldsymbol{v}$ is reduced by that factor of 2 (previously four outputs and now two):

$$\textbf{Stride } \boldsymbol{S = 2} \qquad A = \begin{bmatrix} x_1 & x_0 & x_{-1} & 0 & 0 \\ 0 & 0 & x_1 & x_0 & x_{-1} \end{bmatrix} \tag{6}$$

Now the nonzero weights like x_1 in L are two columns apart (S columns apart for stride S). In 2D, a stride $S = 2$ reduces each direction by 2 and the whole output by 4.

2.2.4 Extending the Signal

Instead of losing neurons at the edges of the image when A is not square, we can *extend the input layer*. We are ”inventing” components beyond the image boundary. Then the output $\boldsymbol{y} = A\boldsymbol{v}$ fits the image block: equal dimensions for input and output.

The simplest and most popular approach is **zero-padding**: Choose all additional components to be *zeros*. The extra columns on the left and right of A multiply those zeros. In between, we have a square Toeplitz matrix. It is still determined by a much smaller set of weights than the number of entries in A.

For periodic signals, zero-padding is replaced by *wraparound*. The Toeplitz matrix becomes a circulant . The Discrete Fourier Transform tells its eigenvalues. The eigenvectors are always the columns of the Fourier matrix. The multiplication $A\boldsymbol{v}$ is a *cyclic convolution* and the Convolution Rule applies.

A more accurate choice is to go beyond the boundary by $reflection$. If the last component of the signal is v_n, and the matrix is asking for v_{N+1} and v_{N+2}, we can reuse v_N and v_{N-1} (or else v_{N-1} and v_{N-2}). Whatever the length of $\boldsymbol{v}$ and the size of A, all the matrix entries in A come from the same E weights x_{-1} to x_1 or x_{-2} to x_2 (and E^2 weights in 2D).

Note Another idea. We might accept the original dimension (128 in our example) and use the reduction to 64 as a way to apply **two filters $\boldsymbol{C_1}$ and $\boldsymbol{C_2}$**. Each filter output is downsampled from 128 to 64. The total sample count remains 128. If the filters are suitably independent, no information is lost and the original 128 values can be recovered.

This process is linear. Two 64 by 128 matrices are combined into 128 by 128: square. If that matrix is invertible, as we intend, the filter bank is *lossless*.

This is what CNN's usually do: Add more channels of weight matrices A in order to capture more features of the training sample. The neural net has a bank of B filters.

2.2.5 Filter Banks and Wavelets

The idea in those last paragraphs produces a **filter bank**. This is just a set of B different filters (convolutions). In signal processing, an important case combines a lowpass filter C_1 with a highpass filter C_2. The output of $C_1\boldsymbol{v}$ is a smoothed signal (dominated by low frequencies). The output $C_2\boldsymbol{v}$ is dominated by high frequencies. A perfect cutoff by ideal filters cannot be achieved by finite matrices C_1 and C_2.

From two filters we have a total of 256 output components. Then both outputs are subsampled. The result is 128 components, separated approximately into *averages and differences*–low frequencies and high frequencies. The matrix is 128 by 128.

Wavelets The wavelet idea is to repeat the same steps on the 64 components of the lowpass output $(\downarrow 2)\ C_1\boldsymbol{x}$. *Then* $(\downarrow 2)C_1(\downarrow 2)C_1\boldsymbol{x}$ *is an average of averages.* Its frequencies are concentrated in the lowest quarter ($|\omega| \leq \pi/4$) of all frequencies. The mid-frequency output $(\downarrow 2)C_2(\downarrow 2)C_1\boldsymbol{x}$ with 32 components will not be subdivided. Then $128 = 64 + 32 + 16 + 16$.

In the limit of infinite subdivision, wavelets enter. This low-high frequency separation is an important theme in signal processing. It has not been so important for deep learning. But with multiple channels in a CNN, frequency separation could be effective.

2.2.6 Counting the Number of Inputs and Outputs

In a one-dimensional problem, suppose a layer has N neurons. We apply a convolutional matrix with E nonzero weights. The stride is S, and we pad the input signal by P zeros at each end. How many outputs (M numbers) does this filter produce?

$$\boxed{\textbf{Karpathy's formula} \quad \boldsymbol{M} = \frac{\boldsymbol{N-E+2P}}{\boldsymbol{S}} + 1} \tag{7}$$

In a 2D or 3D problem, this 1D formula applies in each direction.

Suppose $E = 3$ and the stride is $S = 1$. If we add one zero ($P = 1$) at each end, then

$$M = N - 3 + 2 + 1 = N \qquad \text{(input length = output length)}$$

This case $2P = E - 1$ with stride $S = 1$ is the most common architecture for CNN's.

If we don't pad the input with zeros, then $P = 0$ and $M = N - 2$ (as in the 4 by 6 matrix A at the start of this section). In 2 dimensions this becomes $M^2 = (N-2)^2$. We lose neurons this way, but we avoid zero-padding.

Now suppose the stride is $S = 2$. Then $N - E$ must be an even number. Otherwise the formula (4) produces a fraction. Here are two examples of success for stride $S = 2$, with $N - E = 5 - 3$ and padding $P = 0$ or $P = 1$ at both ends of the five inputs:

$$\begin{array}{c}\textbf{Stride}\\ \textbf{2}\end{array} \begin{bmatrix} x_{-1} & x_0 & x_1 & 0 & 0 \\ 0 & 0 & x_{-1} & x_0 & x_1 \end{bmatrix} \begin{bmatrix} x_{-1} & x_0 & x_1 & 0 & 0 & 0 & 0 \\ 0 & 0 & x_{-1} & x_0 & x_1 & 0 & 0 \\ 0 & 0 & 0 & 0 & x_{-1} & x_0 & x_1 \end{bmatrix}$$

Again, our counts apply in each direction to an image in 2D or a tensor.

2.2.7 A Deep Convolutional Network

Recognizing images is a major application of deep learning (and a major success). The success came with the creation of AlexNet and the development of convolutional nets. This page will describe a deep network of local convolutional matrices for image recog- nition. We follow the prize-winning paper of Simonyan and Zisserman from ICLR 2015. That paper recommends a deep architecture of $L = 16 - 19$ layers with small (3×3) filters. The network has a breadth of B parallel channels (B images on each layer).

If the breadth B were to stay the same at all layers, and all filters had E by E local weights, a straightforward formula would estimate the number W of weights in the net:

$$\boxed{\boldsymbol{W \approx LBE^2} \qquad \textbf{\textit{L} layers, \textit{B} channels, \textit{E} by \textit{E} local convolutions}} \quad (8)$$

Notice that W does not depend on the count of neurons on each layer. This is because A has E^2 weights, whatever its size. Pooling will change that size without changing E^2.

But the count of B channels can change–*and it is very common to end a CNN with fully-connected layers.* This will radically change the weight count W!

It is valuable to discuss the decisions taken by Simonyan and Zisserman, together with other options. Their choices led to $W \approx 135,000,000$weights. The computations were on four NVIDIA GPU's, and training one net took 2-3 weeks. The reader may have less computing power (and smaller problems). So the network hyperparameters L and B will be reduced. We believe that the important principles remain the same.

A key point here is the recommendation to reduce the size E of the local convolutions. 5 by 5 and 7 by 7 filters were rejected. In fact a 1 by 1 convolutional layer can be a way to introduce an extra bank of ReLU's–as in the ResNets coming next.

The authors compare three convolution layers, each with 3 by 3 filters, to a single layer of less local 7 by 7 convolutions. They are comparing 27 weights with 49 weights, and three nonlinear layers with one. In both cases the influence of a single data point spreads to three neighbors vertically and horizontally in the image or the RGB images ($B = 3$). Preference goes to the 3 by 3 filters with extra nonlinearities from more neurons per layer.

2.2.8 Softmax Outputs for Multiclass Networks

In recognizing digits, we have 10 possible outputs. For letters and other symbols, 26 or more. With multiple output classes, we need an appropriate way to decide the very last layer (the output layer $\boldsymbol{w}$ in the neural net that started with $\boldsymbol{v}$). ”Softmax” replaces the two-output case of logistic regression. **We are turning $\boldsymbol{n}$ numbers into probabilities**.

The outputs $w_1, ..., w_n$, are converted to probabilities $p_1, ..., p_n$ that add to 1:

$$\boxed{\textbf{Softmax} \quad \boldsymbol{P_j = \frac{1}{S} e^{wj}} \quad \text{where} \quad \boldsymbol{S} = \sum_{k=1}^{n} \boldsymbol{e^{wk}}} \tag{9}$$

Certainly softmax assigns the largest probability p_j to the largest output w_j. But e^w is a nonlinear function of w. So the softmax assignment is not invariant to scale: If we double all the outputs w_j, softmax will produce different probabilities p_j. For small $w's$ softmax actually deemphasizes the largest number w_{max}.

In the CNN example of **teachyourmachine.com** to recognize digits, you will see how softmax produces the probabilities displayed in a pie chart—an excellent visual aid.

CNN We need a lot of weights to fit the data, and we are proud that we can compute them (with the help of gradient descent). But there is no justification for the number of weights to be uselessly large–*if weights can be reused.* For long signals in ID and especially images in 2D, we may have no reason to change the weights from pixel to pixel.

2.2.9 Support Vector Machine in the Last Layer

For CNN's in computer vision, the final layer often has a special form. If the previous layers used ReLU and max-pooling (both piecewise linear), the last step can become a difference of convex program, and eventually a multiclass Support Vector Machine (SVM). Then optimization of the weights in a piecewise linear CNN can be one layer at a time.

L. Berrada, A. Zisserman, and P. Kumar, *Trusting SVM for piecewise linear CNNs*, arXiv: 1611.02185, 6 Mar 2017.

2.2.10 The World Championship at the Game of Go

A dramatic achievement by a deep convolutional network was to defeat the (human) world champion at Go. This is a difficult game played on a 19 by 19 board. In turn, two players put down ”stones” in attempting to surround those of the opponent. When a group of one color has no open space beside it (left, right, up, or down), those stones are removed from the board. Wikipedia has an animated game.

AlphaGo defeated the leading player Lee Sedol by 4 games to 1 in 2016. It had trained on thousands of human games. This was a convincing victory, but not overwhelming. Then the neural network was deepened and improved. Google's new version AlphaGo Zero learned to play without any human intervention–simply by playing against itself. Now it defeated its former self AlphaGo by 100 to 0.

The key point about the new and better version is that **the machine learned by itself**. It was told the rules and nothing more. The first version had been fed earlier games, aiming to discover why winners had won and losers had lost. The outcome from the new approach was parallel to the machine translation of languages. To master a language, special cases from grammar seemed essential. How else to learn all those exceptions? The translation team at Google was telling the system what it needed to know.

Meanwhile another small team was taking a different approach: Let the machine figure it out. In both cases, playing Go and translating languages, success came with a deeper neural net and more games and no coaching.

It is the depth and the architecture of AlphaGo Zero that interest us here. The hyperparameters will come in Section II.4: the fateful decisions. The parallel history of Google Translate must wait until II.5 because Recurrent Neural Networks (**RNN's**) are needed–to capture the sequential structure of text.

It is interesting that the machine often makes opening moves that have seldom or never been chosen by humans. The input to the network is a board position and its history. The output vector gives the probability of selecting each move–and also a scalar that estimates the probability of winning from that position. Every step communicates with a Monte Carlo tree search, to produce *reinforcement learning.*

2.2.11 Residual Networks (ResNets)

Networks are becoming seriously deeper with more and more hidden layers. Mostly these are convolutional layers with a moderate number of independent weights. But depth brings dangers. Information can jam up and never reach the output. The problem of "vanishing gradients" can be serious: so many multiplications in propagating so far, with the result that computed gradients are exponentially small. When it is well designed, depth is a good thing–**but you must create paths for learning to move forward**.

The remarkable thing is that those fast paths can be very simple: "*skip connections*" that go directly to the next layer–bypassing the usual step $\boldsymbol{v}_n = (A_n \boldsymbol{v}_{n-1} + \boldsymbol{b}_n)_+$. An efficient proposal of Veit, Wilber, and Belongie is to allow either a *skip* or a normal convolution, with a ReLU step every time. If the net has L layers, there will be 2^L possible routes–fast or normal from each layer to the next.

One result is that entire layers can be removed without significant impact. The nth layer is reached by 2^{n-1} possible paths. Many paths have length well below n, not counting the skips.

It is hard to predict whether deep ConvNets will be replaced by ResNets.

K. He, X. Zhang, S. Ren, and J. Sun, *Deep residual learning for image recognition*, arXiv: 1512.03385, 10 Dec 2015. This paper works with extremely deep neural nets by adding shortcuts that skip layers, with weights $A = I$. Otherwise depth can degrade performance.

K. He, X. Zhang, S. Ren, and J. Sun, *Identity mappings in deep residual networks*, arXiv: 1603.05027, 25 Jul 2016.

A. Veit, M. Wilber, and S. Belongie, *Residual networks behave like ensembles of relatively shallow networks*, arXiv: 1605.06431, 27 Oct 2016.

2.2.12 A Simple CNN: Learning to Read Letters

One of the class projects at MIT was a convolutional net. The user begins by drawing multiple copies (not many) of A and B. On this training set, the correct classification is part of the input from the user. Then comes the mysterious step of *learning this data*—creating a continuous piecewise linear function $F(\boldsymbol{v})$ that gives high probability to the correct answer (the letter that was intended).

For learning to read digits, 10 probabilities appear in a pie chart. You quickly discover that too small a training set leads to frequent errors. If the examples had centered numbers or letters, and the test images are not centered, the user understands why those errors appear.

One purpose of **teachyourmachine.com** is education in machine learning at all levels (schools included). It is accessible to every reader.

These final references apply highly original ideas from signal processing to CNN's:

R. Balestriero and R. Baraniuk, Mad Max: *Affine spline insights into deep learning*, arXiv: 1805.06576.

S. Mallat, *Understanding deep convolutional networks*, Phil. Trans. Roy. Soc. **374** (2016); arXiv: 1601.04920.

C.-C. J. Kuo, *The CNN as a guided multilayer RECOS transform*, IEEE Signal Proc. Mag. **34** (2017) 81-89; arXiv:1701.08481.

2.3 Backpropagation and the Chain Rule

Deep learning is fundamentally a giant problem in optimization. We are choosing numerical "weights" to minimize a loss function L (which depends on those weights). $L(\boldsymbol{x})$ adds up all the losses $\boldsymbol{l}(\boldsymbol{w} - \textbf{true}) = l(F(\boldsymbol{x}, \boldsymbol{v})\ - \text{true})$ between the computed outputs $\boldsymbol{w} = F(\boldsymbol{x}, \boldsymbol{v})$ and the true classifications of the inputs $\boldsymbol{v}$. Calculus tells us the system of equations to solve for the weights that minimize L:

The partial derivatives of $\boldsymbol{L}$ with respect to the weights $\boldsymbol{x}$ should be zero.

Gradient descent in all its variations needs to compute derivatives (components of the gradient of F) at the current values of the weights. The derivatives $\frac{\partial F}{\partial x}$ lead to $\frac{\partial L}{\partial x}$. From that information we move to new weights that give a smaller loss. Then we recompute derivatives of F and L at the new values of the weights, and repeat.

Backpropagation is a method to compute derivatives quickly, using the chain rule:

Chain Rule $\frac{dF}{dx} = \frac{d}{dx}(F_3(F_2(F_1(x)))) = (\frac{dF_3}{dF_2}(F_2(F_1(x))))(\frac{dF_2}{dF_1}(F_1(x)))(\frac{dF_1}{dx}(F_1(x))$

The statement of the chain rule applies to a function $F(x)$ of one variable. But with many nodes(neurons) on each layer of a neural net, we have function of many variables. The chain rule still applies–except now we have a matrix (and sometimes a 3-way tensor) at each step.

This is an incredibly efficient improvement on the separate computation of each derivative $\partial F/\partial x_i$. At first it seems unbelievable, that reorganizing the computations can make such an enormous difference. In the end (the doubter might say) you have to compute derivatives for each step and multiply by the chain rule. But the method does work-and N derivatives are computed in far less than N times the cost of one derivative $\partial F/\partial x_1$.

Backpropagation has been discovered many times. Another name is **automatic differentiation (AD)**. You will see that the steps can be arranged in two basic ways: **forward-mode** and **backward-mode**. The right choice of mode can make a large difference in the cost (a factor of thousands). That choice depends on whether you have many functions F depending on a few inputs, or few functions F depending on many inputs.

Deep learning has basically one loss function depending on many weights. The right choice is ”backward-mode AD”. This is what we call **backpropagation**. It is the computational heart of deep learning. We will illustrate computational graphs and back- propagation by a small example.

The computational graphs were inspired by the brilliant exposition of Christopher Olah, posted on his blog (colah.github.io). Since 2017 he has published on (https://distill.pub). And the new paper by Catherine and Desmond Higham (arXiv: 1801.05894, to appear in SIAM Review) gives special attention to backpropagation, with very useful codes.

2.3.1 Derivatives $\partial F/\partial x$ of the Learning Function $F(x, v)$

The weights $\boldsymbol{x}$ consist of all the matrices $A_1, ..., A_L$ and the bias vectors $\boldsymbol{b_1}, ..., \boldsymbol{b_L}$. The inputs $\boldsymbol{v} = \boldsymbol{v_0}$ are the training data. The outputs $\boldsymbol{w} = F(\boldsymbol{x}, \boldsymbol{v_0})$ appear in layer L. Thus $\boldsymbol{w} = \boldsymbol{v_L}$ is the last step in the neural net, after $\boldsymbol{v_1}, ..., \boldsymbol{v_{L-1}}$ in the hidden layers. Each new layer $\boldsymbol{v_n}$ comes from the previous layer by $R(\boldsymbol{b}_n + A_n\boldsymbol{v_{n-1}})$. Here R is the nonlinear activation function (usually ReLU) applied one component at a time.

Thus deep learning carries us from $\boldsymbol{v} = \boldsymbol{v_0}$ to $\boldsymbol{w} = \boldsymbol{v_L}$. Then we substitute $\boldsymbol{w}$ into the loss function to measure the error for that sample $\boldsymbol{v}$. It may be a classification error: O instead of 1, or 1 instead of 0. It may be a least squares regression error $||\boldsymbol{g} - \boldsymbol{w}||^2$, with $\boldsymbol{w}$ instead of a desired output $\boldsymbol{g}$. Often it is a ”cross-entropy”. The total loss $L(\boldsymbol{x})$ is the sum of the losses on all input vectors $\boldsymbol{v}$.

The giant optimization of deep learning aims to find the weights $\boldsymbol{x}$ that minimize L. For full gradient descent the loss is $L(\boldsymbol{x})$. For stochastic gradient descent the loss at each iteration is $l(\boldsymbol{x})-$ from a single input or a minibatch of inputs. **In all cases we need the derivatives $\partial w/\partial x$ of the outputs $\boldsymbol{w}$** (the components of the last layer) with respect to the weights $\boldsymbol{x}$ (the $A's$ and $\boldsymbol{b}'s$ that carry us from layer to layer).

This is one reason that deep learning is so expensive and takes so long-even on GPU's.

2.3.2 Computation of $\partial F/\partial x$: Explicit Formulas

We plan to compute the derivatives $\partial F/\partial x$ in two ways. The first way is to present the explicit formulas: the derivative with respect to each and every weight. The second way is to describe the **backpropagation algorithm** that is constantly used in practice.

Start with the last bias vector $\boldsymbol{b}_L$ and weight matrix A_L that produce the final output $\boldsymbol{v}_L = \boldsymbol{w}$. There is no nonlinearity at this layer, and we drop the layer index L:

$$\boldsymbol{v}_L = \boldsymbol{b}_L + A_L \boldsymbol{v}_{\boldsymbol{L-1}} \text{ or simply } \boldsymbol{w} = \boldsymbol{b} + A\boldsymbol{v}. \tag{1}$$

Our goal is to find the derivatives $\partial w_i/\partial b_j$ and $\partial w_i/\partial A_{jk}$ for all components of $\boldsymbol{b}+A\boldsymbol{v}$. When j is different from i, the ith output $\boldsymbol{w_i}$ is not affected by $\boldsymbol{b_j}$ or A_{jk}. Multiplying A times $\boldsymbol{v}$, row j of A produces w_j and not w_i . We introduce the symbol δ which is 1 or 0:

$$\boldsymbol{\delta_{ij}} = \mathbf{1}\ if\ i=j \quad \boldsymbol{\delta_{ij}} = \mathbf{0}\ if\ i \neq j \qquad \text{The identity matrix I has entries } \boldsymbol{\delta_{ij}}.$$

Columns of I are **1-hot vectors!** The derivatives are 1 or 0 or $\boldsymbol{v}_k$:

$$\textbf{Fully connected layer} \quad \textbf{Independent weights } \boldsymbol{A_{jk}} \quad \boxed{\frac{\partial w_i}{\partial b_j} = \boldsymbol{\delta_{ij}} \text{ and } \frac{\partial w_i}{\partial A_{jk}} = \boldsymbol{\delta_{ij} v_k}} \tag{2}$$

Example There are six $b's$ and $a's$ in $\begin{bmatrix} \boldsymbol{w_1} \\ w_2 \end{bmatrix} = \begin{bmatrix} \boldsymbol{b_1} \\ b_2 \end{bmatrix} + \begin{bmatrix} \boldsymbol{a_{11}v_1 + a_{12}v_2} \\ a_{21}v_1 + a_{22}v_2 \end{bmatrix}$.

$$\textbf{Derivatives of } \boldsymbol{w_1} = \frac{\partial w_1}{\partial b_1} = 1, \frac{\partial w_1}{\partial b_2} = \mathbf{0}, \frac{\partial w_1}{\partial a_{11}} = \boldsymbol{v_1}, \frac{\partial w_1}{\partial a_{12}} = \boldsymbol{v_2}, \frac{\partial w_1}{\partial a_{21}} = \frac{\partial w_1}{\partial a_{22}} = \mathbf{0}$$

2.3.3 Combining Weights b and A into M

It is often convenient to combine the bias vector $\boldsymbol{b}$ and the matrix A into one matrix M:

$$\boxed{\textbf{Matrix of weights} \quad M = \begin{bmatrix} 1 & \mathbf{0}^T \\ \boldsymbol{b} & A \end{bmatrix} \quad \textbf{has} \quad M \begin{bmatrix} 1 \\ \boldsymbol{v} \end{bmatrix} = \begin{bmatrix} 1 \\ \boldsymbol{b} + A\boldsymbol{v} \end{bmatrix}} \tag{3}$$

For each layer of the neural net, the top entry (the zeroth entry) is now **fixed at 1**. After multiplying that layer by M, the zeroth component on the next layer is still 1. Then ReLU(1) = 1 preserves that entry in every hidden layer.

The block matrix M produces a compact derivative formula for the last layer $\boldsymbol{w} = M\boldsymbol{v}$.

$$\boxed{\boldsymbol{M} = \begin{bmatrix} \mathbf{1} & \mathbf{0}^{\boldsymbol{T}} \\ \boldsymbol{b} & \boldsymbol{A} \end{bmatrix} \quad \textbf{and} \quad \frac{\boldsymbol{\partial w_i}}{\boldsymbol{\partial M_{jk}}} = \boldsymbol{\delta_{ij} v_k} \quad \textbf{for} \quad \boldsymbol{i > 0}} \tag{4}$$

Both $\boldsymbol{v}$ and $\boldsymbol{w}$ begin with 1's. Then $k = 0$ correctly gives $\partial w_0 / \partial M_{j0} = 0$ for $j > 0$.

2.3.4 Derivatives for Hidden Layers

Now suppose there is one hidden layer, so $L = 2$. The output is $\boldsymbol{w} = \boldsymbol{v_L} = \boldsymbol{v_2}$, the hidden layer contains $\boldsymbol{v}_1$, and the input is $\boldsymbol{v_0} = \boldsymbol{v}$. The nonlinear $\mathbf{R}$ is probably ReLU.

$$\boldsymbol{v}_1 = R(\boldsymbol{b}_1 + A_1\boldsymbol{v}_0) \text{ and } \boldsymbol{w} = \boldsymbol{b}_2 + A_2\boldsymbol{v}_1 = \boldsymbol{b}_2 + A_2R(\boldsymbol{b}_1 + A_1\boldsymbol{v}_0).$$

Equation (2) still gives the derivatives of $\boldsymbol{w}$ with respect to the last weights $\boldsymbol{b}_2$ and A_2. The function R is absent at the output and $\boldsymbol{v}$ is $\boldsymbol{v}_1$. But the derivatives of $\boldsymbol{w}$ with respect to $\boldsymbol{b}_1$ and A_1 do involve the nonlinear function R acting on $\boldsymbol{b}_1 + A_1\boldsymbol{v}_0$.

So the derivatives in $\partial w/\partial A_1$ need the chain rule $\partial f/\partial x = (\partial f/\partial g)(\partial g/\partial x)$:

$$\textbf{Chain rule}\, \frac{\partial \boldsymbol{w}}{\partial A_1} = \frac{\partial [A_2 R(\boldsymbol{b}_1 + A_1 \boldsymbol{v}_0)}{\partial A_1} = A_2 R'(\boldsymbol{b}_1 + A_1 \boldsymbol{v}_0) \frac{\partial (\boldsymbol{b}_1 + A_1 \boldsymbol{v}_0)}{\partial A_1} \qquad (5)$$

That chain rule has three factors. Starting from $\boldsymbol{v_0}$ at layer $L-2=0$, the weights $\boldsymbol{b}_1$ and A_1 bring us toward the layer $L-1=1$. The derivatives of that step are exactly like equation (2). But the output of that partial step is not $\boldsymbol{v_{L-1}}$. To find that hidden layer we first have to apply R. So the chain rule includes its derivative $\boldsymbol{R'}$. Then the final step (to $\boldsymbol{w}$) multiplies by the last weight matrix A_2.

The Problem Set extends these formulas to L layers. They could be useful. But with pooling and batch normalization, automatic differentiation seems to defeat hard coding.

Very important Notice how formulas like (2) and (5) **go backwards from $\boldsymbol{w}$ to $\boldsymbol{v}$**. Automatic backpropagation will do this too. **"Reverse mode"** starts with the output.

2.3.5 Details of the Derivatives $\partial w/\partial A_1$

We feel some responsibility to look more closely at equation (5). Its nonlinear part R' comes from the derivative of the nonlinear activation function. The usual choice is the ramp function ReLU $(x) = (x)_+$, and we see ReLU as the limiting case of an S-shaped sigmoid function. Here is ReLU together with its first two derivatives:

$$\begin{aligned}
&\text{ReLU}(x) = \max(0,x) = (x)_+ && \textbf{Ramp function } R(x)\\
&dR/dx = \begin{cases} 0, & x<0 \\ 1, & x>0 \end{cases} && \textbf{Step function } H(x) = dR/dx\\
&d^2R/dx^2 = \begin{cases} 0, & x \neq 0 \\ 1, & \text{integral all over } x \end{cases} && \textbf{delta function } \delta(x) = d^2R/dx^2
\end{aligned}$$

The delta function represents an *impulse*. It models a finite change in an infinitesimal time. It is physically impossible but mathematically convenient. It is defined, not at every point x, but by its effect on integrals from -∞ to ∞ of a continuous function $g(x)$:

$$\int \delta(x)dx = 1 \qquad \int \delta(x)g(x)dx = g(0) \qquad \int \delta(x-a)g(x)dx = g(a)$$

With ReLU, a neuron could stay at zero through all the steps of deep learning. This "dying ReLU" can be avoided in several ways-it is generally not a major problem. One way that firmly avoids it is to change to a Leaky ReLU with a nonzero gradient:

$$\textbf{Leaky ReLU (x)} = \begin{cases} \boldsymbol{x}, & x \geq 0 \\ \mathbf{0.01}\boldsymbol{x}, & x \leq 0 \end{cases} \qquad \textbf{Always } \mathrm{ReLU}(ax) = a\mathrm{ReLU}(x) \quad (6)$$

Geoffrey Hinton pointed out that if all the bias vectors $\boldsymbol{b_1, ..., b_L}$ are set to zero at every layer of the net, the scale of the input $\boldsymbol{v}$ passes straight through to the output $\boldsymbol{w} = F(\boldsymbol{v})$. Thus $F(A\boldsymbol{v}) = aF(\boldsymbol{v})$. (A final softmax would lose this scale invariance.)

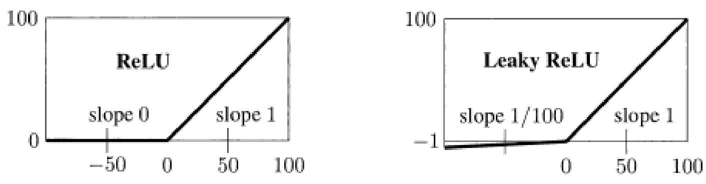


Figure 2.6: The graphs of ReLU and Leaky ReLU (two options for nonlinear activation).

Returning to formula (4), write A and $\boldsymbol{b}$ for the matrix A_{L-1} and the vector $\boldsymbol{b_{L-1}}$ that produce the last hidden layer. Then ReLU and A_L and $\boldsymbol{b}_L$ produce the final output $\boldsymbol{w} = \boldsymbol{v_L}$. Our interest is in $\partial W/\partial A$, the dependence of $\boldsymbol{w}$ on the next to last matrix of weights.

$$\boxed{\boldsymbol{w} = A_L(R(A\boldsymbol{v} + \boldsymbol{b})) + \boldsymbol{b}_L \text{ and } \frac{\partial \boldsymbol{w}}{\partial A} = A_L R'(A\boldsymbol{v} + \boldsymbol{b})\frac{\partial(A\boldsymbol{v}+\boldsymbol{b})}{\partial A}} \qquad (7)$$

We think of R as a diagonal matrix of *ReLU* functions acting component by component on $A\boldsymbol{v} + \boldsymbol{b}$. Then $J = R'(A\boldsymbol{v} + \boldsymbol{b})$ is a diagonal matrix with 1's for positive components and O's for negative components. (Don't ask about zeros.) Formula (7) has become (8):

$$\boldsymbol{w} = A_L(R(A\boldsymbol{v} + \boldsymbol{b})) \;\; and \;\; \frac{\partial \boldsymbol{w}}{\partial A} = A_L J \frac{\partial (A\boldsymbol{v} + \boldsymbol{b})}{\partial A} \tag{8}$$

We know every component (v_k or zero) of the third factor from the derivatives in (2)

When the sigmoid function R_a replaces the ReLU function, the diagonal matrix $J = R_a'(A\boldsymbol{v} + \boldsymbol{b})$ no longer contains 1's and 0's. Now we evaluate the derivative dR_a/dx at each component of $A\boldsymbol{v} + \boldsymbol{b}$.

In practice, backpropagation finds the derivatives with respect to all $A's$ and $\boldsymbol{b}'s$. It creates those derivatives automatically (and effectively).

2.3.6 Computational Graphs

Suppose $F(x, y)$ is a function of two variables x and y. Those inputs are the first two nodes in the computational graph. A typical step in the computation–**an edge in the graph**– is one of the operations of arithmetic (addition, subtraction, multiplication,...). The final output is the function $F(x, y)$. Our example will be $\boldsymbol{F = x^2(x + y)}$.

Here is the graph that computes F with intermediate nodes $c = x^2$ and $s = x + y$:

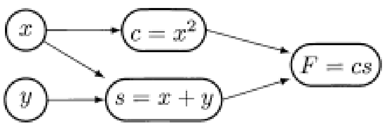


Figure 2.7

When we have inputs x and y, for example $\boldsymbol{x = 2}$ and $\boldsymbol{y = 3}$, the edges lead to $c = \mathbf{4}$ and $s = \mathbf{5}$ and $\boldsymbol{F = 20}$. This agrees with the algebra that we normally crowd into one line: $F = x^2(x + y) = 2^2(2 + 3) = 4(5) = 20$.

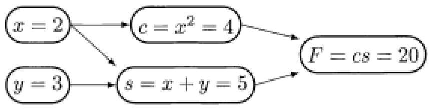


Figure 2.8

Now we compute the derivative of each step–each edge in the graph. Begin with the x-derivative. At first we choose $forward - mode$, starting with the input x and moving toward the output function $x^2(x+y)$. So the first steps use the power rule for $c = x^2$ and the sum rule for $s = x + y$. The last step applies the product rule to $F = c$ times s.

$$\frac{\partial c}{\partial x} = 2x \quad \frac{\partial s}{\partial x} = 1 \quad \frac{\partial F}{\partial c} = s \quad \frac{\partial F}{\partial s} = c$$

Moving through the graph produces the chain rule!

$$\boldsymbol{\frac{\partial F}{\partial x}} = \frac{\partial F}{\partial c}\frac{\partial c}{\partial x} + \frac{\partial F}{\partial s}\frac{\partial s}{\partial x}$$

$$= (s)(2x) + (c)(1) = (5)(4) + (4)(1) = 24$$

The result is to compute the derivative of the output F with respect to *one input* x. You can see those x-derivatives on the computational graph.

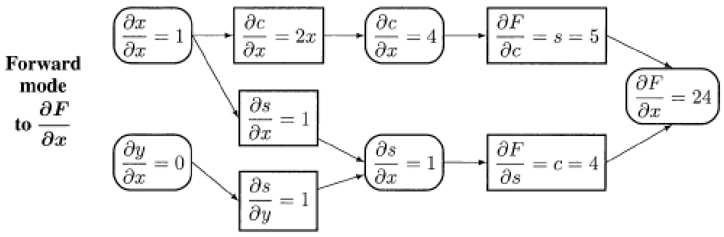

Forward mode to $\frac{\partial F}{\partial x}$
$\frac{\partial x}{\partial x} = 1$
$\frac{\partial c}{\partial x} = 2x$
$\frac{\partial c}{\partial x} = 4$
$\frac{\partial F}{\partial c} = s = 5$
$\frac{\partial F}{\partial x} = 24$
$\frac{\partial s}{\partial x} = 1$
$\frac{\partial y}{\partial x} = 0$
$\frac{\partial s}{\partial y} = 1$
$\frac{\partial s}{\partial x} = 1$
$\frac{\partial F}{\partial s} = c = 4$

Figure 2.9

There will be a similar graph for y-derivatives–the forward mode leading to $\partial F/\partial y$. Here is the chain rule and the numbers that would appear in that graph for $x = 2$ and $y = 3$ and $c = x^2 = 2^2$ and $s = x + y = 2 + 3$ and $F = cs$:

$$\begin{aligned}\boldsymbol{\frac{\partial F}{\partial y}} &= \frac{\partial F}{\partial c}\frac{\partial c}{\partial y} + \frac{\partial F}{\partial s}\frac{\partial s}{\partial y} \\ &= (s)(0) + (c)(1) = (5)(0) + (4)(1) = \mathbf{4}\end{aligned}$$

The computational graph for $\partial F/\partial y$ is not drawn but the point is important: *Forward mode requires a new graph for each input* x_i*, to compute the partial derivative* $\partial F/\partial x_i$.

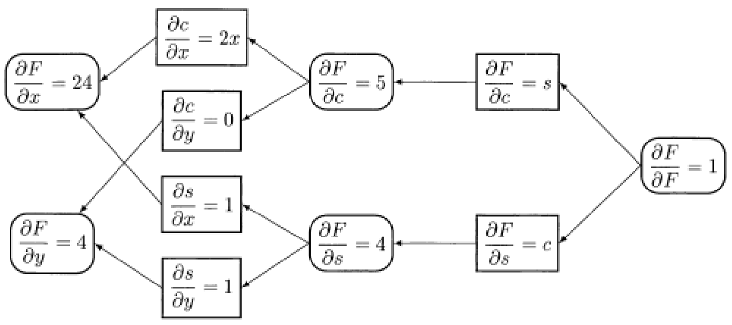
2.3.7 Reverse Mode Graph for One Output

The reverse mode starts with the output F. **It computes the derivatives with respect to both inputs**. The computations go **backward** through the graph.

That means it does not follow the empty line that started with $\partial y/\partial x = 0$ in the forward graph for x-derivatives. And it would not follow the empty line $\partial x/\partial y = 0$ in the forward graph (not drawn) for y-derivatives. A larger and more realistic problem with N inputs will have N forward graphs, each with $N - 1$ empty lines (because the N inputs are independent). The derivative of x_i with respect to every other input x_j is $\partial x_i/\partial x_j = 0$.

Instead of N forward graphs from N inputs, we will have **one backward graph from one output**. Here is that reverse-mode computational graph. It finds the derivative of F with respect to every node. It starts with $\partial F/\partial F = 1$ and goes in reverse.

A computational graph executes the chain rule to find derivatives. The reverse mode finds all derivatives $\partial F/\partial x_i$ by following all chains **backward from output to input**. Those chains all appear as paths **on one graph**-not as separate chain rules for exponentially many possible paths. This is the success of reverse mode.



$\frac{\partial F}{\partial x} = 24$
$\frac{\partial F}{\partial y} = 4$
$\frac{\partial c}{\partial x} = 2x$
$\frac{\partial c}{\partial y} = 0$
$\frac{\partial s}{\partial x} = 1$
$\frac{\partial s}{\partial y} = 1$
$\frac{\partial F}{\partial c} = 5$
$\frac{\partial F}{\partial s} = 4$
$\frac{\partial F}{\partial c} = s$
$\frac{\partial F}{\partial s} = c$
$\frac{\partial F}{\partial F} = 1$

Figure 2.10: Reverse-mode computation of the gradient $(\frac{\partial F}{\partial x}, \frac{\partial F}{\partial y})$ at $x = 2, y = 3$

2.3.8 Product of Matrices ABC: Which Order?

The decision between forward and reverse order also appears in matrix multiplication! If we are asked to multiply A times B times C, the associative law offers two choices for the multiplication order:

$$\boldsymbol{AB} \textbf{ first or } \boldsymbol{BC} \textbf{ first? \quad Compute } (\boldsymbol{AB})\boldsymbol{C} \textbf{ or } \boldsymbol{A}(\boldsymbol{BC})?$$

The result is the same but the number of individual multiplications can be very different. Suppose the matrix A is m by n, and B is n by p, and C is p by q.

$$\begin{array}{lrl} \textbf{First way} & AB = (m*n)(n*p) & \text{has } \boldsymbol{mnp} \text{ multiplications} \\ & (AB)C = (m*p)(p*q) & \text{has } \boldsymbol{mpq} \text{ multiplications} \\ \textbf{Second way} & BC = (n*p)(p*q) & \text{has } \boldsymbol{npq} \text{ multiplications} \\ & A(BC) = (m*n)(n*q) & \text{has } \boldsymbol{mng} \text{ multiplications} \end{array}$$

So the comparison is between $mp(n+q)$ and $nq(m+p)$. Divide both numbers by $mnpq$:

$$\boxed{\text{The first way is faster when } \tfrac{1}{q} + \tfrac{1}{n} \text{ is smaller than } \tfrac{1}{m} + \tfrac{1}{p}.}$$

Here is an extreme case (extremely important). Suppose C is a column vector: p by 1. Thus $q = 1$. Should you multiply BC to get another column vector (n by 1) and then $A(BC)$ to find the output (m by 1)? Or should you multiply AB first?

The question almost answers itself. The correct $A(BC)$ produces a vector at each step. The matrix-vector multiplication BC has np steps. The next matrix-vector multiplication $A(BC)$ has mn steps. Compare those $np + mn$ steps to the cost of starting with the matrix-matrix option AB (mnp steps!). Nobody in their right mind would do that.

But *if A is a row vector, $(AB)C$ is better.* Row times matrix each time.

This will match the central computation of deep learning: **Training the network = optimizing the weights**. The output $F(\boldsymbol{v})$ from a deep network is a chain starting with $\boldsymbol{v}$:

$$F(\boldsymbol{v}) = A_L \boldsymbol{v}_{L-1} = A_L(RA_{L-1}(...(RA_2(\boldsymbol{R}A_1\boldsymbol{v})))) \text{ is forward through the net.}$$

The derivatives of F with respect to the matrices A (and the bias vectors $\boldsymbol{b}$) are easiest for the last matrix A_L in $A_L\boldsymbol{v}_{L-1}$. The derivative of $A\boldsymbol{v}$ with respect to A contains $v's$:

$\frac{\partial F_i}{\partial A_{jk}} = \delta_{ij} v_k$. Next is the derivative of $A_L ReLU(A_{L-1}v_{L-1})$ with respect to A_{L-1}.

We can explain how that same reverse mode also appears in the comparison of direct methods versus adjoint methods for optimization (choosing good weights).

2.3.9 Adjoint Methods

The same question of the best order for matrix multiplication ABC comes up in a big class of optimization problems. We are solving a square system of N linear equations $E\boldsymbol{v} = \boldsymbol{b}$. The vector $\boldsymbol{b}$ depends on **design variables** $\boldsymbol{p} = (p_1, ..., p_M)$. Therefore the solution vector $\boldsymbol{v} = E^{-1}\boldsymbol{b}$ depends on $\boldsymbol{p}$. **The matrix** $\partial v/\partial p$ **containing the derivatives** $\partial v_i/\partial p_j$**, will be** $\boldsymbol{N}$ **by** $\boldsymbol{M}$.

To repeat: We are minimizing $F(\boldsymbol{v})$. The vector $\boldsymbol{v}$ depends on the design variables $\mathbf{p}$. So we need a chain rule that multiplies derivatives $\partial F/\partial v_i$ times derivatives $\partial v_i/\partial p_j$. Let me show how this becomes a product of **three matrices**-and the multiplication order is decisive. **Three sets of derivatives** control how F depends on the input variables p_i:

$A = \frac{\boldsymbol{\partial F}}{\boldsymbol{\partial v_i}}$	The derivatives of F with respect to $v_1, ..., v_N$
$B = \frac{\boldsymbol{\partial v_i}}{\boldsymbol{\partial b_k}}$	The derivative of each v_i with respect to each b_k
$C = \frac{\boldsymbol{\partial b_k}}{\boldsymbol{\partial p_j}}$	The derivative of each b_k with respect to each p_j

To see $\partial v_i/\partial p_j$ we take derivatives of the equation $E\boldsymbol{v} = \boldsymbol{b}$ with respect to the p_j:

$$E\frac{\partial \boldsymbol{v}}{\partial p_j} = \frac{\partial \boldsymbol{b}}{\partial p_j}, j = 1, ..., M \quad so \quad \frac{\partial \boldsymbol{x}}{\partial \boldsymbol{p}} = E^{-1}\frac{\partial \boldsymbol{b}}{\partial \boldsymbol{p}} \tag{9}$$

It seems that we have M linear systems of size N. Those will be expensive to solve over and over, as we search for the choice of $\boldsymbol{p}$ that minimizes $F(\boldsymbol{v})$. The matrix $\partial\boldsymbol{v}/\partial\boldsymbol{p}$ contains the derivatives of $v_1, ..., v_N$ with respect to the design variables $p_1, ..., p_M$.

Suppose for now that the cost function $F(\boldsymbol{v}) = \boldsymbol{c^T v}$ is linear (so $\partial F/\partial\boldsymbol{v} = c^T$). Then what optimization actually needs is the gradient of $F(\boldsymbol{v})$ with respect to the $p's$. The first set of derivatives $\partial F/\partial\boldsymbol{v}$ is just the vector c^T:

$$\boxed{\frac{\partial F}{\partial \boldsymbol{p}} = \frac{\partial F}{\partial \boldsymbol{v}}\frac{\partial \boldsymbol{v}}{\partial \boldsymbol{p}} = \boldsymbol{c^T} E^{-1}\frac{\partial \boldsymbol{b}}{\partial \boldsymbol{p}} \text{ has three factors to be multiplied}} \tag{10}$$

This is the key equation. It ends with a product of a **row vector $\boldsymbol{c^T}$** times an **$\boldsymbol{N}$ by $\boldsymbol{N}$** matrix **$\boldsymbol{E^{-1}}$** times an **$\boldsymbol{N}$ by $\boldsymbol{M}$** matrix $\partial\boldsymbol{b}/\partial\boldsymbol{p}$. How should we compute that product?

Again the question almost answers itself. We do not want to multiply two matrices. So we are not computing $\partial\boldsymbol{v}/\partial\boldsymbol{p}$ after all. Instead the good first step is to find $c^T E^{-1}$. This produces a row vector $\boldsymbol{\lambda^T}$. In other words we solve the adjoint equation $E^T\boldsymbol{\lambda} = \boldsymbol{c}$:

$$\boxed{\textbf{Adjoint equation } \boldsymbol{E^T\lambda = c} \text{ gives } \boldsymbol{\lambda^T} E = \boldsymbol{c^T} \text{ and } \lambda^T = \boldsymbol{c^T} E^{-1}.} \tag{11}$$

Substituting $\boldsymbol{\lambda}^{\boldsymbol{T}}$ for $\boldsymbol{c}^{\boldsymbol{T}}E^{-1}$ in equation (10), the final step multiplies that row vector times the derivatives of the vector $\boldsymbol{b}$ (its gradient):

$$\boxed{\textbf{Gradient of the cost F} \quad \frac{\partial F}{\partial \boldsymbol{p}} = \boldsymbol{\lambda}^{\boldsymbol{T}} \frac{\partial \boldsymbol{b}}{\partial \boldsymbol{p}} \quad (1 \text{ by } N \text{ times } N \text{ by } M)} \tag{12}$$

The optimal order is $(\boldsymbol{AB})\boldsymbol{C}$ because the first factor $\boldsymbol{A}$ is actually the row vector $\boldsymbol{\lambda}^{\boldsymbol{T}}$.

This example of an adjoint method started with $E\boldsymbol{x} = \boldsymbol{b}$. The right hand side $\boldsymbol{b}$ depended on design parameters $\boldsymbol{p}$. So the solution $\boldsymbol{x} = E^{-1}\boldsymbol{b}$ depended on $\boldsymbol{p}$. Then The cost function $F(\boldsymbol{x}) = \boldsymbol{c}^{\boldsymbol{T}}\boldsymbol{x}$ depends on $\boldsymbol{p}$.

The adjoint equation $A^T\boldsymbol{\lambda} = \boldsymbol{c}$ found the vector $\boldsymbol{\lambda}$ that efficiently combined the last two steps. "Adjoint" has a parallel meaning to "transpose" and we can apply it also to differential equations. The design variables $p_1, ..., p_M$ might appear in the matrix E, or in an eigenvalue problem or a differential equation.

Our point here is to emphasize and reinforce the key idea of backpropagation:

The reverse mode can order the derivative computations in a faster way.

2.4 Hyperparameters: The Fateful Decisions

After the loss function is chosen and the network architecture is decided, there are still critical decisions to be made. We must choose the *hyperparameters*. They govern the algorithm itself—the computation of the weights. Those weights represent what the computer has learned from the training set: how to predict the output from the features in the input. In machine learning, the decisions include those hyperparameters and the loss function and dropout and regularization.

The goal is to find patterns that distinguish 5 from 7 and 2-by looking at pixels. The hyperparameters decide how quickly and accurately those patterns are discovered. The **stepsize** s_k in gradient descent is first and foremost. That number appears in the iteration $\boldsymbol{x}_{k+1} = \boldsymbol{x}_k - s_k \boldsymbol{\nabla L}(\boldsymbol{x}_k)$ or one of its variants: *accelerated* (by momentum) or adaptive (ADAM) or stochastic with a random minibatch of training data at each step k.

The words **learning rate** are often used in place of stepsize. Depending on the author, the two might be identical or differ by a normalizing factor. Also: η_k often replaces s_k. First we ask for the optimal stepsize when there is only one unknown. Then we point to a general approach. Eventually we want a faster decision.

1. **Choose** $s_k = 1/L''(\boldsymbol{x}_k)$. Newton uses the second derivative of L. That choice accounts for the quadratic term in the Taylor series for $L(x)$ around the point x_k. As a result, Newton's method is second order: The error in x_{k+1} is proportional to the square of the error in x_k. Near the minimizing x^*, convergence is fast.

   In more dimensions, the second derivative becomes the Hessian matrix if $H(x) = \nabla^2 L(\boldsymbol{x}_k)$. Its size is the number of weights (components of $\boldsymbol{x}$). To find $\boldsymbol{x}_{k+1}$, Newton solves a large system of equations $H(\boldsymbol{x}_k)(\boldsymbol{x}_{k+1} - \boldsymbol{x}_k) = -\nabla L(\boldsymbol{x}_k)$. Gradient descent replaces H by a single number $1/s_k$.

2. **Decide** s_k **from a line search**. The gradient $\nabla L(\boldsymbol{x}_k)$ sets the direction of the line. The current point is the start of the line. By evaluating $L(\boldsymbol{x})$ at points on the line, we find a nearly minimizing point-which becomes $\boldsymbol{x}_{k+1}$.

We look next at the effect of a poor stepsize s.

2.4.1 Too Small or Too Large

We need to identify the difficulties with a poor choice of learning rate :

s_k **is too small** Then gradient descent takes too long to minimize $L(\boldsymbol{x})$

Many steps $\boldsymbol{x}_k + 1 - \boldsymbol{x}_k = -s_k \nabla L(\boldsymbol{x}_k)$ with small improvement

s_k **is too large** We are overshooting the best choice $\boldsymbol{x}_{k+1}$ in the descent direction

Gradient descent will jump around the minimizing $\boldsymbol{x}^*$.

Suppose the first steps s_0 and s_1 are found by line searches, and work well. We may want to stay with that learning rate for the early iterations. **Normally we reduce** s as the minimization of $L(\boldsymbol{x})$ continues.

Larger steps at the start	Get somewhere close to the optimal weights x^*
Smaller steps at the end	Aim for convergence without overshoot

A learning rate schedule $s_k = s_0/\sqrt{\boldsymbol{k}}$ or $s_k = s_0/\boldsymbol{k}$ systematically reduces the steps.

After reaching weights a that are close to minimizing the loss function $L(\boldsymbol{x}, \boldsymbol{v})$ we may want to bring new $\boldsymbol{v's}$ from a **validation set**. This is not yet production mode. The purpose of **cross-validation** is to confirm that the computed weights $\boldsymbol{x}$ are capable of producing accurate outputs from new data.

2.4.2 Cross-validation

Cross-validation aims to estimate the validity of our model and the strength of our learning function. Is the model too weak or too simple to give accurate predictions and classifications? Are we overfitting the training data and therefore at risk with new test data? You could say that cross-validation works more carefully with a relatively small data set, so that testing and production can go forward quickly on a larger data set.

Note Another statistical method for another purpose-also reuses the data. This is the *bootstrap* introduced by Brad Efron. It is used (and needed) when the sample size is small or its distribution is not known. We aim for maximum understanding by returning to the (small) sample and *reusing that data* to extract new information. Normally small data sets are not the context for applications to deep learning.

A first step in cross-validation is to divide the available data into K subsets. If $K = 2$, these would essentially be the training set and test set—but we are usually aiming for more information from smaller sets before working with a big test set. K-fold cross-validation uses each of K subsets separately as a test set. In every trial, the other $K-1$ subsets form the training set. We are reworking the same data (moderate size) to learn more than one optimization can teach us.

Cross-validation can make a learning rate adaptive: changing as descent proceeds.

There are many variants, like ”double cross-validation”. In a standardized m by n least squares problem $A\boldsymbol{x} = \boldsymbol{b}$, Wikipedia gives the expected value $(m - n - 1)/(m + n - 1)$ for the mean square error. Higher errors normally indicate overfitting. The corresponding test in deep learning warns us to consider earlier stopping.

This section on hyperparameters was influenced and improved by Bengio’s long chapter in a remarkable book. The book title is Neural Networks: Tricks of the Trade (2nd edition), edited by G. Montavon, G. Orr, and K.-R. Müller. It is published by Springer (2012) with substantial contributions from leaders in the field.

2.4.3 Batch Normalization of Each Layer

As training goes forward, the mean and variance of the original population can change at every layer of the network. This change in the distribution of inputs is ”covariate shift”. We often have to adjust the stepsize and other hyperparameters, due to this shift in the statistics of layers. A good plan is to *normalize the input to each layer.*

Normalization makes the training safer and faster. The need for dropout often disappears. Fewer iterations can now give more accurate weights. And the cost can be very moderate. *Often we just train two additional parameters on each layer.*

The problem is greatest when the nonlinear function is a sigmoid rather than *ReLU*. The sigmoid ”saturates” by approaching a limit like 1 (while *ReLU* increases forever as $x \to \infty$). The nonlinear sigmoid becomes virtually linear and even constant when x becomes large. Training slows down because the nonlinearity is barely used.

It remains to decide the point at which inputs will be normalized. Ioffe and Szegedy avoid computing covariance matrices (far too expensive). Their normalizing transform acts on each input $\boldsymbol{v_1, ..., v_B}$ in a minibatch of size B:

mean	$\mu = (\boldsymbol{v_1} + ... + \boldsymbol{v_B})/B$
variance	$\sigma^2 = (\lvert\lvert\boldsymbol{v_1} - \boldsymbol{\mu}\rvert\rvert^2 + ... + \lvert\lvert\boldsymbol{v_B} - \boldsymbol{\mu}\rvert\rvert^2)/B$
normalize	$V_i = (\boldsymbol{v_i} - \boldsymbol{\mu})/\sqrt{\sigma^2 + \epsilon}$ for small $\epsilon > 0$
scale/shift	$y_i = \gamma V_i + \beta$ γ and β are trainable parameters)

The key point is to **normalize the inputs $\boldsymbol{y}_i$ to each new layer**. What was good for the original batch of vectors (at layer zero) is also good for the inputs to each hidden layer.

S. Ioffe and C. Szegedy, Batch normalization, arXiv: 1502.03167v3, 2 Mar 2015.

2.4.4 Dropout

Dropout is the removal of randomly selected neurons in the network. Those are components of the input layer $\boldsymbol{v}_0$ or of hidden layers $\boldsymbol{v}_n$ before the output layer $\boldsymbol{v}_L$. All weights in the $A's$ and $\boldsymbol{b}'s$ connected to those dropped neurons disappear from the net (Figure 6). Typically hidden layer neurons might be given probability $p = 0.5$ of surviving, and input components might have $p = 0.8$ or higher. *The main objective of random dropout is to avoid overfitting.* It is a relatively inexpensive averaging method compared to combining predictions from many networks.

Dropout was proposed by five leaders in the development of deep learning algorithms: N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. Their paper *"Dropout"* appears in: *Journal of Machine Learning Research* 15 (2014) 1929-1958. For recent connections of dropout to physics and uncertainty see arXiv: 1506.02142 and 1809.08327.

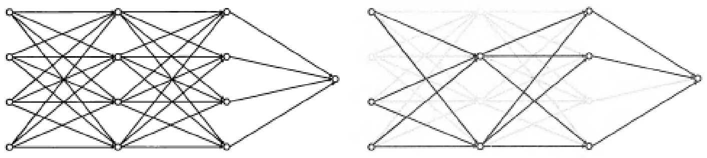


Figure 2.11: Crossed neurons have dropped out in the thinned network.

Dropout offers a way to compute with many different neural architectures at once. In training, each new $\boldsymbol{v_0}$ (the feature vector of an input sample) leads to a new thinned network. Starting with N neurons there are 2^N possible thinned networks.

At test time, we use the full network (no dropout) with weights rescaled from the training weights. The outgoing weights from an undropped neuron are multiplied by p in the rescaling. This approximate averaging at test time led the five authors to reduced generalization errors-more simply than from other regularization methods.

One inspiration for dropout was genetic reproduction-where half of each parent's genes are dropped and there is a small random mutation. That dropout for a child seems more unforgiving and permanent than dropout for deep learning which averages over many thinned networks. (True, we see some averaging over siblings. But the authors conjecture that over time, our genes are forced to be robust in order to survive.)

The dropout model uses a zero-one random variable r (a Bernoulli variable). Then $r = 1$ with probability p and $r = 0$ with probability $1 - p$. The usual feed-forward step to layer n is $\boldsymbol{y}_n = A_n\boldsymbol{v}_{n-1} + \boldsymbol{b}_n$, followed by the nonlinear $\boldsymbol{v}_n = R\boldsymbol{y}_n$. *Now a random r multiplies each component of $\boldsymbol{v}_n - 1$ to drop that neuron when* $r = 0$. Component by component, $\boldsymbol{v}_{n-1}$ is multiplied by 0 or 1 to give $\boldsymbol{v}^*_{n-1}$. Then $\boldsymbol{y}_n = A_n v^*_{n-1} + \boldsymbol{b}_n$.

To compute gradients, use backpropagation for each training example in the mini-batch. Then average those gradients. Stochastic gradient descent can still include acceleration (momentum added) and adaptive descent and weight decay. The authors highly recommend regularizing the weights, for example by a maximum norm requirement $||\boldsymbol{a}|| \leq c$ on the columns of all weight matrices A.

2.4.5 Exploring Hyperparameter Space

Often we optimize hyperparameters using experiments or experience. To decide the learn- ing rate, we may try three possibilities and measure the drop in the loss function. A geometric sequence like .1, .01, .001 would make more sense than an arithmetic sequence .05,.03, .01. And if the smallest or largest choice gives the best results, then continue the experiment to the next number in the series. In this stepsize example, you would be considering computational cost as well as validation error.

LeCun emphasizes that for a multiparameter search, *random sampling* is the way to cover many possibilities quickly. Grid search is too slow in multiple dimensions.

2.4.6 Loss Functions

The loss function measures the difference between the correct and the computed output for each sample. The correct output-often a classification $y = 0, 1$ or $y = 1, 2, ..., n$— is part of the training data. The computed output at the final layer is $\boldsymbol{w} = F(\boldsymbol{x}, \boldsymbol{v})$ from the learning function with weights $\boldsymbol{x}$ and input $\boldsymbol{v}$.

Section I.5 defined three familiar loss functions. Then this chapter turned to the structure of the neural net and the function F. Here we come back to compare square loss with cross-entropy loss.

1. **Quadratic cost (square loss): $\boldsymbol{l(y, w)} = \frac{1}{2}||\boldsymbol{y} - \boldsymbol{w}||^2$.**

   This is the loss function for least squares-always a possible choice. But it is not a favorite choice for deep learning. One reason is the parabolic shape for the graph of $l(y, w)$, as we approach zero loss at $w = y$. The derivative also approaches zero.

   A zero derivative at the minimum is normal for a smooth loss function, but it frequently leads to an unwanted result: The weights A and $\boldsymbol{b}$ change very slowly near the optimum. Learning slows down and many iterations are needed.

2. **Cross-entropy loss:**

$$\boldsymbol{l(y,w)} = -\frac{1}{2}\sum_{1}^{n}[y_i log z_i + (1-y_i)log(1-z_i)] \tag{1}$$

Here we allow and expect that the N outputs w_i from training the neural net have been normalized to $z(w)$, with $0 < z_i < 1$. Often those z_i are probabilities. Then $1 - z_i$ is also between 0 and 1. So both logarithms in above are negative, and the minus sign assures that the overall loss is positive: $l > 0$.

More than that, the logarithms give a different and desirable approach to $z = 0$ or 1. For this calculation we refer to Nielsen's online book Neural Networks and Deep Learning, which focuses on sigmoid activation functions instead of ReLU. The price of those smooth functions is that they saturate (lose their nonlinearity) near their endpoints.

Cross-entropy has good properties, but where do the logarithms come from? The first point is Shannon's formula for entropy (a measure of information). If message i has probability p_i, you should allow $-logp_i$; bits for that message. Then the expected (average) number of bits per message is best possible:

$$\boxed{\textbf{Entropy} = -\sum_1^m p_i log p_i.\ \ For\ m = 2 \text{ this is } \ -p(logp) - (1-p)log(1-p).} \tag{2}$$

Cross-entropy comes in when we don't know the p_i and we use $\hat{\mathbf{p}}_\mathbf{i}$ instead:

$$\boxed{\begin{array}{c}\textbf{Cross-entropy} = -\sum_1^m p_i log\widehat{p}_i. \\ \text{For} \quad m = 2 \text{ this is } -p(log\widehat{p}) - (1-p)log(1-\widehat{p}).\end{array}} \tag{3}$$

(3) is always larger than (2). The true p_i are not known and the $\hat{\mathbf{p}}_\mathbf{i}$ cost more. The difference is a very useful but not symmetric function called **Kullback-Leibler (KL) divergence**.

2.4.7 Regularization : l^2 or l^1 (or none)

Regularization is a voluntary but well-advised decision. It adds a *penalty term* to the loss function $L(\boldsymbol{x})$ that we minimize: an l^2 penalty in ridge regression and l^1 in LASSO.

RR Minimize $||\boldsymbol{b} - A\boldsymbol{x}||_2^2 + \lambda_2||\boldsymbol{x}||_2^2$ **LASSO** Minimize $||\boldsymbol{b} - A\boldsymbol{x}||_2^2 + \lambda_1 \sum |x_i|$

The penalty controls the size of $\boldsymbol{x}$. *Regularization is also called weight decay.*

The coefficient λ_2 or λ_1 is a hyperparameter. Its value can be based on cross-validation. purpose of the penalty terms is to avoid overfitting (sometimes expressed as fitting the noise). Cross-validation for a given λ finds the minimizing $\boldsymbol{x}$ on a test set. Then it checks by using those weights on a training set. If it sees errors from overfitting, λ is increased.

A small value of λ tends to increase the variance of the error: overfitting. Large λ will increase the bias: underfitting because the fitting term $||\boldsymbol{b} - A\boldsymbol{x}||^2$ is less important.

2.4.8 The Structure of AlphaGo Zero

It is interesting to see the sequence of operations in AlphaGo Zero, learning to play Go:

1. A convolution of 256 filters of kernel size 3 * 3 with stride 1: $E = 3, S = 1$
2. Batch normalization
3. ReLU
4. A convolution of 256 filters of kernel size 3 x 3 with stride 1
5. Batch normalization

6. A skip connection as in ResNets that adds the input to the block
7. ReLU
8. A fully connected linear layer to a hidden layer of size 256
9. *ReLU*

Training was by stochastic gradient descent on a fixed data set that contained the final 2 million games of self-played data from a previous run of AlphaGo Zero.

The CNN includes a fully connected layer that outputs a vector of size $19^2 + 1$. This accounts for all positions on the 19 * 19 board, plus a pass move allowed in Go.

2.5 The World of Machine Learning

Fully connected nets and convolutional nets are parts of a larger world. From training data they lead to a learning function $\boldsymbol{F}(\boldsymbol{x}, \boldsymbol{v})$. That function produces a close approximation to the correct output $\boldsymbol{w}$ for each input $\boldsymbol{v}$($\boldsymbol{v}$ is the vector of features of that sample). But machine learning has developed a multitude of other approaches–some long established to the problem of learning from data.

This book cannot do justice to all those ideas. It does seem useful to describe Recurrent Neural Nets (and Support Vector Machines). We also include key words to indicate the scope of machine learning. (A *glossary* is badly needed! That would be a tremendous contribution to this field.) At the end is a list of books on topics in machine learning.

2.5.1 Recurrent Neural Networks (RNNs)

These networks are appropriate for data that comes in a definite order. This includes time series and natural language: *speech or text or handwriting.* In the network of connections from inputs $\boldsymbol{v}$ to outputs $\boldsymbol{w}$, the new feature is the *input* *from the previous time t - 1.* This recurring input is determined by the function $h(t-1)$.

Figure given below shows an outline of that new step in the architecture of the network.

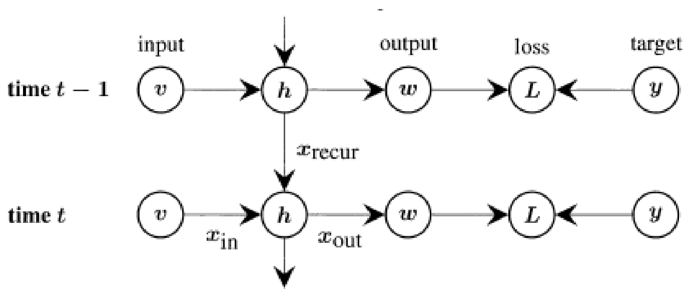


Figure 2.12: The computational graph for a recurrent network finds loss-minimizing outputs $\boldsymbol{w}$ at each time t. The inputs to $\boldsymbol{h}(t)$ are the new data $\boldsymbol{v}(t)$ and the recurrent data $\boldsymbol{h}(t-1)$ from the previous time. The weights multiplying the data are $\boldsymbol{x}_{in}$ and recur and out, chosen to minimize the loss $L(\boldsymbol{y}-\boldsymbol{w})$. This network architecture is *universal*: It will compute any formula that is computable by a Turing machine.

2.5.2 Key Words and Ideas

1 Kernel learning (next page)
2 Support vector Machines (next page)
3 Generative Adversarial Networks
4 Independent Component Analysis
5 Graphical models
6 Bayesian statistics
7 Random forests
8 Reinforcement learning

2.5.3 Support Vector Machines

Start with n points $\boldsymbol{v}_1, ..., \boldsymbol{v}_n$ in m-dimensional space. Each $\boldsymbol{v}_i$ comes with a classification $Y_i = 1$ or $y_i = -1$. The goal proposed by Vapnik is to find a plane $\boldsymbol{w}^T\boldsymbol{v} = b$ in m dimensions that *separates the plus points from the minus points*–if this is possible. That vector $\boldsymbol{w}$ will be perpendicular to the plane. The number b tells us the distance $|b|/||\boldsymbol{w}||$ from the line or plane or hyperplane in $\boldsymbol{R}^m$ to the point $(0, ..., 0)$.

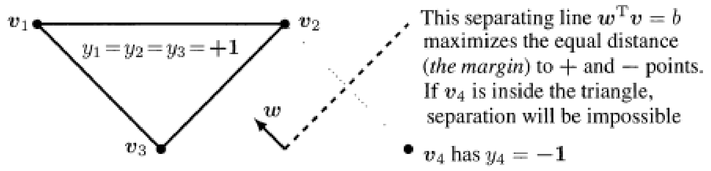

v_1
v_2
$y_1 = y_2 = y_3 = +1$
v_3
w
This separating line $w^T v = b$ maximizes the equal distance (*the margin*) to + and − points. If v_4 is inside the triangle, separation will be impossible
v_4 has $y_4 = -1$

Figure 2.13

Problem Find $\boldsymbol{w}$ and b so that $\boldsymbol{w}^T\boldsymbol{v}_i - b$ has the correct sign y_i for all points $i = 1, ..., n$.

If $\boldsymbol{v}_1, \boldsymbol{v}_2, \boldsymbol{v}_3$ are plus points ($y = +1$) in a plane, then $\boldsymbol{v}_4$ *must be outside the triangle of* $\boldsymbol{v}_1, \boldsymbol{v}_2, \boldsymbol{v}_3$. The picture shows the line of maximum separation (*maximum margin*).

Maximum margin Minimize $\|\|\boldsymbol{w}\|\|$ under the conditions $y_i(\boldsymbol{w}^T\boldsymbol{v}_i - b) \geq 1$.

This is a "hard margin". That inequality requires $\boldsymbol{v}_i$ to be on its correct side of the separator. If the points can't be separated, then no $\boldsymbol{w}$ and b will succeed. For a "soft margin" we go ahead to choose the best available $\boldsymbol{w}$ and b, based on hinge loss + penalty:

$$\textbf{Soft margin} \quad \text{Minimize} \left[\frac{1}{n}\sum_1^n \max(0, 1 - y_i(\boldsymbol{w}^T\boldsymbol{v}_i - \boldsymbol{b}))\right] + \lambda||\boldsymbol{w}||^2 \qquad (1)$$

That hinge loss (the maximum term) is zero when $\boldsymbol{v}_i$ is on the correct side of the separator. If separation is impossible, the penalty $\lambda||\boldsymbol{w}||^2$ balances hinge losses with margin sizes.

If we introduce a variable h_i for that hinge loss we are minimizing a quadratic function of $\boldsymbol{w}$ with linear inequalities connecting $\boldsymbol{w}, b, y_i$ and h_i. This is quadratic programming in high dimensions—well understood in theory but challenging in practice.

2.5.4 The Kernel Trick

SVM is linear separation. A plane separates + points from − points. The kernel trick allows a nonlinear separator, when feature vectors $\boldsymbol{v}$ are transformed to

$N(\boldsymbol{v})$. Then the dot product of transformed vectors gives us the **kernel function** $\boldsymbol{K}(\boldsymbol{v}_i, \boldsymbol{v}_j) = N(\boldsymbol{v}_i)^T N(\boldsymbol{v}_j)$.

The key is to work entirely with K and not at all with the function N. *In fact we never see or need N.* In the linear case, this corresponds to choosing a positive definite K and not seeing the matrix A in $K = A^T A$. The RBF kernel $exp(-||\boldsymbol{v}_i - \boldsymbol{v}_j||^2/2\sigma^2)$ we used earlier.

M. Belkin, S. Ma, and S. Mandal, *To understand deep learning we need to understand kernel learning*, arXiv:1802.01396. ”Non-smooth Laplacian kernels defeat smooth Gaussians”

T. Hofmann, B. Schölkopf, and A. J. Smola, Kernel methods in machine learning, Annals of Statistics 36 (2008) 1171-1220 (with extensive references).

2.5.5 Google Translate

An exceptional article about deep learning and the development of Google Translate appeared in the New York Times Magazine on Sunday, 14 December 2016. It tells how Google suddenly jumped from a conventional translation to a recurrent neural network. The author Gideon Lewis-Kraus describes that event as three stories in one: the work of the development team, and the group inside Google that saw what was possible, and the worldwide community of scientists who gradually shifted our understanding of how to learn:

https://www.nytimes.com/2016/12/14/magazine/the-great-AI-awakening.html

The development took less than a year. Google Brain and its competitors conceived the idea in five years. The worldwide story of machine learning is an order of magnitude longer in time and space. The key point about the recent history is the earthquake it produced in the approach to learning a language:

Instead of programming every word and grammatical rule and exception in both

languages, let the computer find the rules. Just give it enough correct translations.

If we were recognizing images, the inputs would be many examples with correct labels (the training set). The machine creates the function $F(\boldsymbol{x}, \boldsymbol{v})$.

This is closer to how children learn. And it is closer to how we learn. If you want to teach checkers or chess, the best way is to get a board and make the moves. Play the game.

The steps from this vision to neural nets and deep learning did not come easily. Marvin Minsky was certainly one of the leaders. But his book with Seymour Papert was partly about what "Perceptrons" could not do. With only one layer, the XOR function (A or B but not both) was unavailable. Depth was missing and it was needed.

The lifework of Geoffrey Hinton has made an enormous difference to this subject. For machine translation, he happened to be at Google at the right time. For image recognition, he and his students won the visual recognition challenge in 2012 (with AlexNet). Its depth changed the design of neural nets. Equally impressive is a 1986 article in *Nature*, in which Rumelhart, Hinton, and Williams foresaw that backpropagation would become crucial in optimizing the weights: *Learning representations by back-propagating errors.*

These ideas led to great work worldwide. The "cat paper" in 2011-2012 described training a face detector without labeled images. The leading author was Quoc Le: *Building high-level features using large scale unsupervised learning*: **arxiv.org/abs/1112.6209**. A large data set of 200 by 200 images was sampled from YouTube. The size was managed by localizing the receptive fields. The network had one billion weights to be trained—this is still a million times smaller than the number of neurons in our visual cortex. Reading this paper, you will see the arrival of deep learning.

A small team was quietly overtaking the big team that used rules. Eventually the paper with 31 authors arrived on **arxiv.org/abs/1609.08144**. And Google had to switch to the deep network that didn't start with rules.

Chapter 3

Conclusion and Future perspective

The Objective of this thesis was to highlight the pivotal role of optimization in refining data science models for maximum efficiency and the transformative impact of deep learning in uncovering intricate patterns within complex data sets. It seeks to demonstrate how these advanced methodologies not only enhance predictive accuracy but also expand the horizons of practical applications in various sectors.

3.1 Conclusion

Optimization is very important application of Data Science which highlights the pivotal shift in optimization strategies. First come the nice problems of linear and quadratic programming and game theory. Duality and saddle points are key ideas. These are traditional linear and quadratic methods to the complex and large-scale challenges of deep learning.”Derivative equals zero” is still the fundamental equation. The second derivative that Newton would have used are too numerous and too complicated to compute. This evolution in approach is marked by the adoption of techniques like stochastic gradient descent, catering to the immense data volumes and computational demands of modern AI.

Deep learning further distinguishes itself through specialized neural network architectures, incorporating elements like convolutional layers, ReLU activation functions, and dropout strategies. An input layer is connected to hidden layers and finally to the output layer. For the training data, input vectors $\boldsymbol{v}$ are known. Also the correct outputs are known.These innovations are crucial in fine-tuning the networks for high accuracy and efficiency, enabling them to adeptly handle tasks such as image and speech recognition, and language translation. This confluence of advanced optimization methods and architectural ingenuity underpins the remarkable capabilities and ongoing progress in the field of artificial intelligence. In the end we can say that Deep learning is a kind of sub space of machine learning which is further a sub space of Artificial Intelligence.

3.2 Future Perspective

As we venture further into the digital era, the realm of data science is poised to become more pivotal than ever, driven by the relentless evolution of machine learning and AI. It's going to be really important to use data carefully and protect people's privacy. Data science will help solve big problems in health and the environment. To keep up, we'll need to keep learning new things. I want to be part of this exciting field, using data to help make good decisions and improve the world.

Bibliography

[1] Gilbert Strang. *Linear algebra and learning from data.* SIAM, 2019.

[2] Dimitri Bertsekas, Angelia Nedic, and Asuman Ozdaglar. *Convex analysis and optimization*, volume 1. Athena Scientific, 2003.

[3] Laurent Lessard, Benjamin Recht, and Andrew Packard. Analysis and design of optimization algorithms via integral quadratic constraints. *SIAM Journal on Optimization*, 26(1):57–95, 2016.

[4] Ashia C Wilson, Benjamin Recht, and Michael I Jordan. A lyapunov analysis of momentum methods in optimization. *arXiv preprint arXiv:1611.02635*, 2016.

[5] Chiyuan Zhang, Samy Bengio, Moritz Hardt, Benjamin Recht, and Oriol Vinyals. Understanding deep learning (still) requires rethinking generalization. *Communications of the ACM*, 64(3):107–115, 2021.

[6] Thomas Strohmer and Roman Vershynin. A randomized kaczmarz algorithm with exponential convergence. *Journal of Fourier Analysis and Applications*, 15(2):262–278, 2009.

[7] Anna Ma, Deanna Needell, and Aaditya Ramdas. Convergence properties of the randomized extended gauss–seidel and kaczmarz methods. *SIAM Journal on Matrix Analysis and Applications*, 36(4):1590–1604, 2015.

[8] Sebastian Thrun, Lawrence K Saul, and Bernhard Schölkopf. *Advances in neural information processing systems 16: proceedings of the 2003 conference*, volume 16. MIT press, 2004.

[9] Sebastian Ruder. An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747*, 2016.

[10] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of machine learning research*, 12(7), 2011.

[11] John C Duchi, Peter L Bartlett, and Martin J Wainwright. Randomized smoothing for (parallel) stochastic optimization. In *2012 IEEE 51st IEEE Conference on Decision and Control (CDC)*, pages 5442–5444. IEEE, 2012.

[12] Manzil Zaheer, Sashank Reddi, Devendra Sachan, Satyen Kale, and Sanjiv Kumar. Adaptive methods for nonconvex optimization. *Advances in neural information processing systems*, 31, 2018.

[13] Gilbert Strang. *Introduction to linear algebra*. SIAM, 2022.

[14] George Cybenko. Approximations by superpositions of a sigmoidal function. *Mathematics of Control, Signals and Systems*, 2:183–192, 1989.

[15] Kurt Hornik. Approximation capabilities of multilayer feedforward networks. *Neural networks*, 4(2):251–257, 1991.

[16] Ding-Xuan Zhou. Universality of deep convolutional neural networks. *Applied and computational harmonic analysis*, 48(2):787–794, 2020.

[17] Michael A Nielsen. *Neural networks and deep learning*, volume 25. Determination press San Francisco, CA, USA, 2015.

[18] Francois Chollet. *Deep learning with Python*. Simon and Schuster, 2021.