

CMSE 890
Mathematics of Deep Learning
Spring 2020

Matthew Hirn
Michigan State University
mhirn@msu.edu

July 31, 2020

Contents

0	Prologue: Why the mathematics of deep learning?	3
1	Background on machine learning and learning theory	8
1.1	Prediction versus estimation	8
1.1.1	Introduction	8
1.1.2	Making things more precise with probability	9
1.1.3	Maximum likelihood estimator	12
1.1.4	Supervised vs unsupervised learning	15
1.2	The supervised learning recipe	16
1.2.1	Functional models	16
1.2.2	Linking the probabilistic model with the functional model	17
1.3	Two methods	18
1.3.1	Linear regression	18
1.3.2	Bayes and maximum a posteriori	23
1.3.3	k -Nearest neighbors and local methods	25
1.4	Basics of statistical learning theory	27
1.4.1	Statistical view of models	27
1.4.2	Bias variance tradeoff	31
1.4.3	Curse of dimensionality	38
1.5	Towards deep learning	47
1.5.1	Dictionaries and kernels	47
1.5.2	Logistic regression and nonlinearity	49
2	Artificial neural networks	51
2.1	What is an artificial neural network?	51
2.2	Training a neural network	53
2.2.1	Gradient descent applied to neural networks	54
2.2.2	Stochastic gradient descent	55
2.2.3	Why ReLU? Vanishing gradients and optimization	57
2.3	Classic approximation theory results	58
2.3.1	One layer approximation theory	58
2.3.2	One layer approximation and the curse of dimensionality	61
2.3.3	Refined one-layer analysis and two-layer neural networks	66
2.4	Modern theory for ANNs	76

2.4.1	More differences between one-layer and two-layer ANNs	76
2.4.2	Compositional functions	78
3	Convolutional neural networks	85

Chapter 0

Prologue: Why the mathematics of deep learning?

To answer the question of why have a course on the *mathematics* of deep learning, it is first instructive to consider why have a course on deep learning at all. To many of you, the answer to this question may seem clear or the question itself rhetorical. Indeed, already at MSU there are several courses on deep learning or that address deep learning as part of their course content, including courses in CSE, ECE, and MTH. Nevertheless, let us consider this question for a moment.

Deep learning refers to a class of algorithms in machine learning and more generally artificial intelligence. One of the hallmarks of deep learning algorithms is that they compose a sequence of many simple functions, often alternating between linear or affine functions, point-wise nonlinear functions, and pooling operations. Figure 1 gives an illustration of the VGG16 network [1], which is a powerful and very popular convolutional neural network, that consists of 16 layers of linear/non-linear pairs of operations, as well as pooling operations (where the length and width of the image stacks shrink) every few layers. Note that all of the linear functions are learned from the given training data and the associated task, which for VGG16 was image classification on the the ImageNet data base [2] (more on this later). Thus the the input to the VGG16 network is an RGB image, and the output is a class label. The compositional structure illustrated in the VGG network, and used in all of deep learning (this is where the “deep” comes from), has been incredibly successful in machine learning and artificial intelligence tasks over the last decade.

Indeed, deep learning is now used in a multitude of different contexts, from computer vision to natural language processing to playing games to biology to physics and more. One of the most striking examples of the success of deep learning (and in this case, reinforcement learning), is the success of AlphaGo

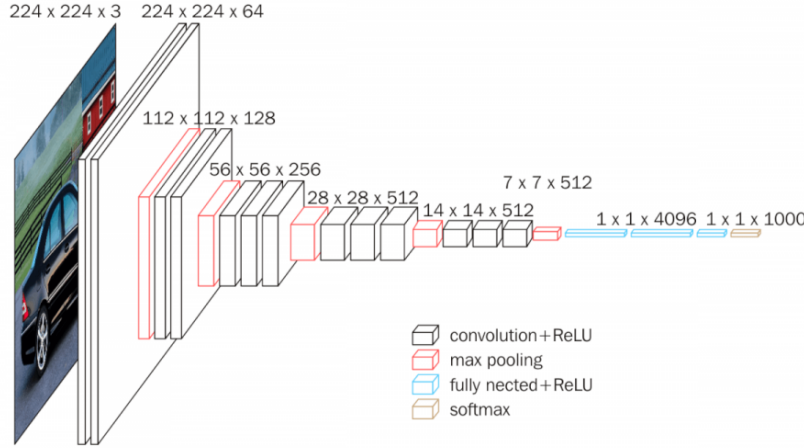
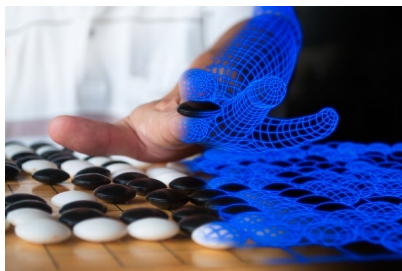


Figure 1: The VGG16 network [1].

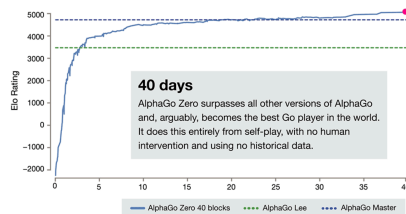
Zero [3], which is an AI developed by Google DeepMind that has unparalleled ability at playing the game of Go (see Figure 2). Going back earlier in the decade, another key marker in the recent explosion in popularity of deep learning is the success of convolutional neural networks for image classification on the aforementioned ImageNet data base. In 2012, AlexNet [4], an eight layer convolutional neural network depicted in Figure 3, outperformed the next best classifier on the ImageNet data base by an astounding 10% on the top five prediction error rate. The existence of ImageNet [2] (see also Figure 4), and other databases such as the handwritten digit data base MNIST, which yield a “common task framework” have also been a key driver in the development of machine learning generally, and deep learning specifically.

But just how popular is deep learning? Indeed, the increasing popularity of deep learning has been rapid and breathtaking; see Figure 5 for the number of registrations and paper submissions to NeurIPS, the most popular machine learning conference. With this rapid increase in popularity, deep learning is being incorporated in a number of contexts with direct societal impact, such as self driving cars, medical diagnostics, insurance, and more.

However, deep learning is not without criticism. Indeed, despite its empirical successes, relatively speaking very little is known, precisely, on how and why it works so well. Indeed, the common task framework that has resulted in so much advancement has also resulted in the phenomena that many deep learning papers are the product of significant amounts of trial and error and less so on theoretically grounded process. Furthermore, even successful algorithms are hard to interpret. Thus, while advancement has been rapid in the last 10 years, one could argue that new, significant advancement will only



(a) Image from an article in *Scientific American* by Larry Greenemeier on AlphaGo Zero [5]; image credit Saran Poroong.



(b) Training time versus Elo rating for AlphaGo Zero.

Figure 2: After 40 days of training only against itself, AlphaGo Zero became arguably the best Go player in the world, and could beat its predecessors (older version of AlphaGo) nearly without fail.

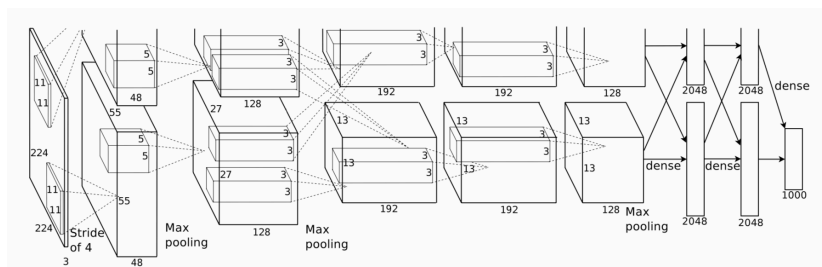


Figure 3: The architecture of AlexNet [4].

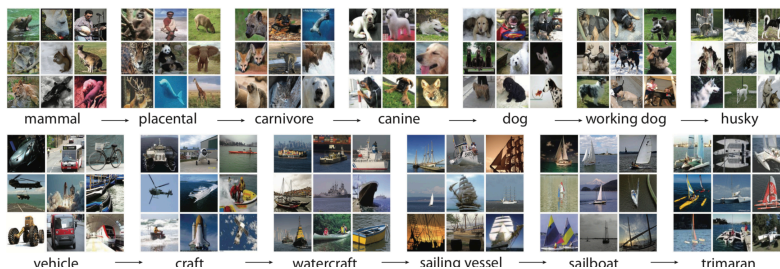


Figure 4: A snapshot of a few images from the ImageNet data base [2], organized hierarchically by progressively more specific classes.

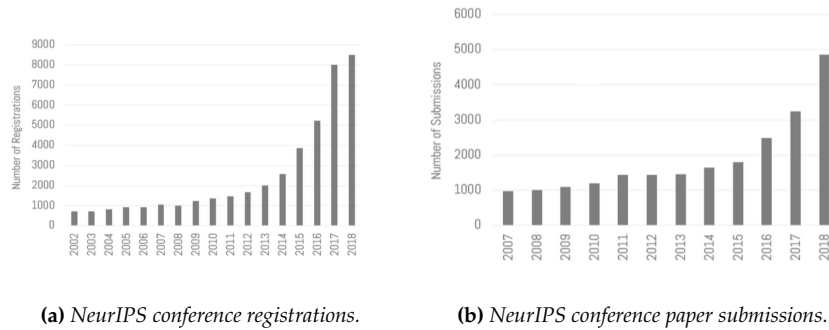


Figure 5: *NeurIPS conference registrations and paper submissions by year.*

come with increased understanding of the current state of the art. Furthermore, as deep learning finds its way more and more into the public realm, it will no longer be enough for machine learning algorithms and artificial intelligences to make a plausibly correct predictions or assessments, people will also increasingly want and need to know the reasons for such outcomes. Such considerations are also related to issues of fairness and bias in machine learning, which is only beginning to be understood but which will certainly increase in importance and study in the coming years.

The beginning point for further and more precise understanding, as with many scientific disciplines, is mathematics. The mathematical study of deep learning has progressed along many different paths, but many of these can be grouped into two primary avenues. On the one hand, there is the issue of training the networks. Because of their compositional and highly nonlinear structure, solving for the optimal weights of deep learning architectures results in a highly non-convex, high dimensional optimization problem. This is one avenue of mathematical study. On the other hand, one must first design the network before one can solve for the weights. The design of deep networks, and their resulting mathematical properties, is the second avenue of study. This course will focus primarily on the latter, leaving the mathematical details of optimization to another course. Within this context, we will consider supervised and unsupervised machine learning.

Notation

- \mathbb{R}^d is d -dimensional Euclidean space.
- Abstract probability measures are denoted by \mathbb{P} .
- Data points are denoted by x . A data point x is either a vector $x = (x(1), \dots, x(d))$ in \mathbb{R}^d or a function $x(u)$ with $u \in \mathbb{R}^n$. It will often be useful to think of x as being sampled from a random variable X , i.e. $x \sim X$.
- The space of absolutely integrable functions is $\mathbf{L}^1(\mathbb{R}^d)$, that is all functions $x(u)$ such that

$$\|x\|_1 = \int_{\mathbb{R}^d} |x(u)| du < +\infty.$$

- The space of square integrable functions is $\mathbf{L}^2(\mathbb{R}^d)$, that is all functions $x(u)$ such that

$$\|x\|_2 = \left[\int_{\mathbb{R}^d} |x(u)|^2 du \right]^{1/2} < +\infty.$$

- A finite collection of N data points is denoted by $\{x_1, \dots, x_N\}$.
- In the context of supervised learning, data labels are denoted by y . A label is always a function of data point x , i.e., $y = y(x)$. The label y may either take values in a continuum, in which case we will assume $y \in \mathbb{R}$ or it may take a finite set of labels (classes), in which case $y \in \{1, \dots, M\}$ or sometimes $y \in \{-1, +1\}$. Like with data points, it will often be useful to think of y as being sampled from a random variable Y , i.e. $y \sim Y$. In this case, the random variable Y is dependent on the random variable X .
- A finite collection of N labels is denoted by $\{y_1, \dots, y_N\}$. These labels will always arise from a finite collection of N data point X , where $y_i = y(x_i)$.
- A data set consisting of N pairs of data points and labels, often referred to as a training set, is denoted by

$$T = \{(x_1, y_1), \dots, (x_N, y_N)\}.$$

- Model parameters are denoted by $\theta \in \mathbb{R}^n$.

Chapter 1

Background on machine learning and learning theory

In this chapter we give a brief background on machine learning and statistical learning theory, which will give context for the development of deep learning and its goals.

1.1 Prediction versus estimation

Inspired by [6, Section I.A].

1.1.1 Introduction

Prediction versus estimation; correlation versus causation. When you hear these phrases in the context of machine learning, what do you think of? Maybe one thinks of the difference between classifying new data points and generating new data points. Or perhaps one considers that correlation is a symmetric assessment (e.g., if A is correlated with B, then B is correlated with A), but causation is directional (e.g., if A causes B, B does not necessarily cause A)¹.

These concepts are in some sense the difference between machine learning and statistics. In machine learning and prediction based tasks, we are often interested in developing algorithms that are capable of learning patterns from given data in an automated fashion, and then using these learned patterns to make predictions or assessments of newly given data. In many cases, our primary concern is the quality of the predictions or assessments, and we are less concerned about the underlying patterns that were learned in order make these predictions. Neural networks are, in some sense, the epitome of this point of view. In various contexts they are incredibly good at making predictions, but

¹Thanks to Cullen Haselby and Bashir Sadeghi for these comments in class.

they are often referred to as “black box” methods due to the difficulty in understanding the model by which they make such predictions. For example, the most powerful convolutional neural networks are incredibly good at classifying natural images (sometimes even better than humans), but it is very difficult to understand the mechanisms by which they make such predictions.

In (classical) statistics and estimation, one is more concerned with the underlying model that makes the prediction. In other words, are the parameters of the model that makes the prediction statistically significant? Or could several other models (i.e., different parameter choices) have made the same prediction? This is the correlation versus causation issue. It comes up, for example and perhaps most notably, in medical trials and studies, in which one must not only find correlations and patterns in the data, but one must find the causal factors of a disease, so that one may develop and prescribe treatment.

1.1.2 Making things more precise with probability

Let us try to make this difference a bit more precise. To do so we will use the language of probability. Consider a given data set

$$T = \{(x_1, y_1), \dots, (x_N, y_N)\},$$

consisting of data points $\{x_1, \dots, x_N\} \subset \mathcal{X}$ and associated scalar valued labels $\{y_1, \dots, y_N\} \subset \mathcal{Y}$. Let us assume that each data point x_i was sampled from \mathcal{X} according to a probability distribution \mathbb{P}_X . This means, more precisely, we have a *probability space* $(\mathcal{X}, \Sigma, \mathbb{P}_X)$, which consists of:

- \mathcal{X} : The set of all possible outcomes, i.e., data points.
- Σ : The space of all possible events, i.e., collections of data. This is a set of sets, which has additional structure (see below).
- \mathbb{P}_X : The probability measure. For each event (set / collection of data points) $A \in \Sigma$, $\mathbb{P}_X(A)$ is the probability of the event A occurring.

The set Σ is a σ -algebra, meaning it has the following properties:

1. $\mathcal{X} \in \Sigma$.
2. If $A \in \Sigma$, then the complement of A , $A^c = \mathcal{X} \setminus A$, is also in Σ . Note this means $\emptyset \in \Sigma$.
3. If A_1, A_2, \dots are all in Σ , then their union is also in Σ , i.e.,

$$\bigcup_{i=1}^{\infty} A_i \in \Sigma.$$

Note that these properties also imply that if A_1, A_2, \dots are in Σ , then

$$\bigcap_{i=1}^{\infty} A_i \in \Sigma.$$

The probability measure \mathbb{P}_X satisfies the following properties:

1. $\mathbb{P}_X(\mathcal{X}) = 1$.
2. Whenever A_1, A_2, \dots is a sequence of disjoint sets in Σ , then

$$\mathbb{P}_X \left(\bigcup_{i=1}^{\infty} A_i \right) = \sum_{i=1}^{\infty} \mathbb{P}_X(A_i).$$

From these properties we can also conclude that $\mathbb{P}_X(\emptyset) = 0$ and $\mathbb{P}_X(A^c) = 1 - \mathbb{P}_X(A)$.

Example 1.1. A simple example, that is not too relevant to our future discussions but which illustrates the idea, is the following. Consider flipping a coin twice, with the probability of heads being p and the probability of tails being q . There are four possible outcomes:

$$\mathcal{X} = \{HH, HT, TH, TT\}.$$

We also know the probabilities of each of these outcomes are p^2 , pq , pq , and q^2 , respectively. We thus set

$$\mathbb{P}_X(HH) = p^2, \mathbb{P}_X(HT) = \mathbb{P}_X(TH) = pq, \mathbb{P}_X(TT) = q^2. \quad (1.1)$$

This information is enough to define a probability space $(\mathcal{X}, \Sigma, \mathbb{P}_X)$, but it does not specify all the sets in Σ or their probabilities. Indeed, the event $A = \{HH, HT\}$ = “the first coin toss is a head,” should be in Σ since it is the union of HH and HT . We assign A the probability $\mathbb{P}_X(A) = p^2 + pq = p$. In fact, the easiest way to ensure Σ is a σ -algebra is to include every subset of \mathcal{X} , including \emptyset and \mathcal{X} itself, and to assign probabilities using the rules of (1.1)².

Example 1.2. Another example, more relevant to our studies in this course, is inspired by the MNIST database of handwritten digits; see Figure 1.1. In this case \mathcal{X} is the infinite set of all possible handwritten digits, and the σ -algebra Σ and the probability measure \mathbb{P}_X are unknown to us, but we assume the training and testing set in the database are sampled according to \mathbb{P}_X , whatever it may be³.

Since the data points x_i are randomly sampled, and the label y_i depends on x_i , the labels y_i are sampled from \mathcal{Y} according to a probability distribution that encodes the dependence of y_i on x_i . We model this as a conditional probability distribution $\mathbb{P}_{Y|X}(B \mid X = x)$, which measures the probability of an event $B \subset \mathcal{Y}$ (i.e., a set of labels) given an outcome $x \in \mathcal{X}$. Together these two distributions induce a joint distribution $\mathbb{P}_{X,Y}$ on $\mathcal{X} \times \mathcal{Y}$, from which we draw the training samples.

²As pointed out by ???, in this case Σ is the power set of the set of outcomes.

³As pointed out by Cullen Haselby, if we assume that each image is of a fixed resolution and has a finite grayscale gradient, then the number of possible images is very large, but finite.

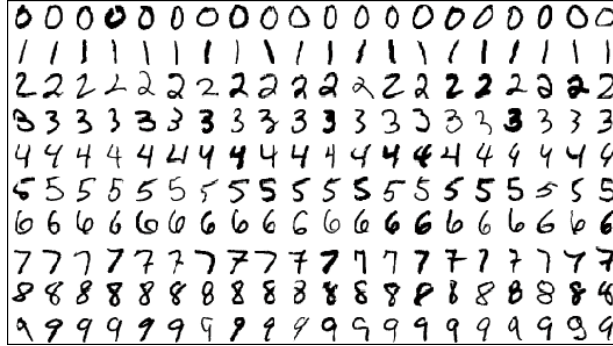


Figure 1.1: Examples from the MNIST database of handwritten digits.

In particular, suppose we draw the x_i 's independently from \mathbb{P}_X . This means we “run the experiment” of drawing a point N times, each time independent from the others, and we obtain a random data point x_i . An analogy is flipping the coin, i.e. Example 1.1. Suppose we carry out the experiment of flipping the coin twice N times, each time independent from the other times. Then each time we will get a “data point,” which corresponds to one of the four outcomes HH, HT, TH, TT , with the probabilities calculated earlier. Drawing a training sample (that is, a data point and a label) first entails drawing a point $x \in \mathcal{X}$ according to the distribution \mathbb{P}_X , and then drawing a point $y \in \mathcal{Y}$ according to $\mathbb{P}_{Y|X}(\cdot | X = x)$. When we want to think of the data point as being determined (say by an experiment, or a draw of a training point) we will write (x, y) . On the other hand, when we want to think of the training point as a pair of random variables, one X taking values in \mathcal{X} and the other Y taking values in \mathcal{Y} , we will write (X, Y) .

Example 1.3. An example to keep in mind, that we will come back to later, is the following. Suppose $\mathcal{X} = \mathbb{R}^d$ and $\mathcal{Y} = \mathbb{R}$, and that a label $y_i \in \mathcal{Y}$ is generated from an underlying deterministic function $F : \mathbb{R}^d \rightarrow \mathbb{R}$ plus random noise:

$$y_i = F(x_i) + \varepsilon_i, \quad 1 \leq i \leq N.$$

Often we will assume that the ε_i are independently and identically distributed (i.i.d.) according to the normal distribution with mean zero and variance σ^2 , i.e. $\varepsilon_i \sim \mathcal{N}(0, \sigma^2)$. In this case, if X is the random variable that takes values in \mathbb{R}^d according to the probability distribution \mathbb{P}_X , and Y is the random variable that takes values in \mathbb{R} according to $\mathbb{P}_{Y|X}(\cdot | X = x)$, then we have that

$$Y \sim \mathcal{N}(F(x), \sigma^2),$$

where $\mathcal{N}(\mu, \sigma^2)$ is the normal distribution with mean μ and variance σ^2 . In other words, given that $X = x$, the label Y is a normal random variable with mean $F(x)$ and variance σ^2 .

1.1.3 Maximum likelihood estimator

In order to make our analysis more concrete and precise, let us put the general probabilistic framework of the previous section in the more specific language of discrete and continuous random variables.

Let us begin with the discrete setting. In this case \mathcal{X} is a finite, but possible very large set (e.g., see the footnote for the MNIST data set in Example 1.2), and \mathcal{Y} is also a finite set, for example because we are doing classification and there are a finite number of classes (again, think of MNIST, there are 10 classes). In this case, there is a probability of drawing each individual point $x \in \mathcal{X}$ into our training set $T = \{(x_i, y_i)\}_{i=1}^N$. Let us call that probability $p_X(x)$, where we use X to denote the random variable that takes values in \mathcal{X} according to $p_X(x)$. Note then, the probability of drawing a point from a subset $A \subseteq \mathcal{X}$ is (equivalently, the probability that X takes a value in A):

$$\mathbb{P}_X(X \in A) = \mathbb{P}_X(A) = \sum_{x \in A} p_X(x).$$

Also note, if the x_i 's in the training set are sampled independently from \mathcal{X} according to $p_X(x)$, then the probability of getting that particular set $\{x_i\}_{i=1}^N$ is:

$$p_X(\{x_i\}_{i=1}^N) \propto \prod_{i=1}^N p_X(x_i) = p_X(x_1) \cdot p_X(x_2) \cdots p_X(x_N),$$

where the notation \propto mean “proportional to.” In particular, since we just care about the set $\{x_i\}_{i=1}^N$ and not the order in which it was drawn, the probability of drawing the set, $p_X(\{x_i\}_{i=1}^N)$, is higher than the right hand side. For example, if all the x_i 's are unique (almost always the case in machine learning tasks), the correct factor is $N! = N \cdot (N-1) \cdots 2 \cdot 1$.

Recall the labels $\{y_i\}_{i=1}^N$ depend on their respective data points $\{x_i\}_{i=1}^N$. In this case, for each label $y \in \mathcal{Y}$, there is a probability of drawing y conditional on the data point being x . Let us call that probability $p_{Y|X}(y | x)$, where we use Y to denote the random variable that takes values in \mathcal{Y} according to the conditional probabilities $p_{Y|X}(y | x)$. Note that if given x there is no ambiguity in the label y according to the underlying data generation process (not our model for the data!) (e.g., suppose $\mathcal{Y} = \{0, 1\}$ and for a specific x the label is always $y = 0$), then $p_{Y|X}(y | x)$ will be one when y is the correct label for x and zero otherwise. The probability of drawing a label from a subset $B \subseteq \mathcal{Y}$, given that our data point is $X = x$, is:

$$\mathbb{P}_{Y|X}(B | X = x) = \sum_{y \in B} p_{Y|X}(y | x).$$

The joint probability of drawing the pair (x, y) is then:

$$p_{X,Y}(x, y) = p_X(x) p_{Y|X}(y | x)$$

which basically says, take the probability of drawing x and multiply it by the probability of drawing y given that we drew x . It follows that the probability

of drawing the training set is:

$$p_{X,Y}(T) \propto \prod_{i=1}^N p_{X,Y}(x_i, y_i),$$

assuming again that the x_i 's are drawn independently from \mathcal{X} according to p_X . Note that we can extend these notions to continuous random variables as well, e.g., if $\mathcal{X} = \mathbb{R}^d$ or $\mathcal{Y} = \mathbb{R}$; see Remark 1.4 below.

Now let us describe how to model $p_{X,Y}(x, y)$ given that our only information is a single training set T . We assume $T = \{(x_i, y_i)\}_{i=1}^N$ is sampled according to the joint probability density $p_{X,Y}(x, y)$. In theory the joint probability density $p_{X,Y}(x, y)$ can be used to sample many different realizations of the data set T (e.g., in the coin tosses of Example 1.1 in which we know how the data is generated), but in practice one often does not have access to $p_{X,Y}(x, y)$ and one must make due with the given, single data set T (e.g., the MNIST data base of Example 1.2). Our goal is to find a good model for $p_{X,Y}(x, y)$ given that all we know is T . We thus consider a hypothesis space of parameterized probability distributions

$$\mathcal{P} = \{p_{X,Y}(x, y \mid \theta) : \theta \in \mathbb{R}^n\},$$

where $p_{X,Y}(T \mid \theta)$ is a model that describes the probability of observing the data T given the parameters θ . If the x_i 's are drawn independently from \mathcal{X} , then

$$p_{X,Y}(T \mid \theta) \propto \prod_{i=1}^N p_{X,Y}(x_i, y_i \mid \theta).$$

The vector $\theta \in \mathbb{R}^n$ encodes the parameters that determine the probabilistic model $p(T \mid \theta)$. These could be the parameters of a neural network, or some other class of machine learning algorithms such as linear models or kernel methods. Since T is the only data we have, our goal is to find the model, i.e. the parameters θ , that maximizes the probability of observing T :

$$\hat{\theta} = \arg \max_{\theta \in \mathbb{R}^n} p(T \mid \theta). \quad (1.2)$$

Thus given T and the hypothesis space \mathcal{P} , our best guess for the underlying joint probability density $p_{X,Y}(x, y)$ is $p_{X,Y}(x, y \mid \hat{\theta})$. The probability density $p_{X,Y}(x, y \mid \hat{\theta})$ is the *maximum likelihood estimate* for the probability density $p_{X,Y}(x, y)$. We will come back to this later.

In machine learning tasks, one is often interested in the accuracy of $p_{X,Y}(x, y \mid \hat{\theta})$ relative to $p_{X,Y}(x, y)$; that is, what is the predictive power of $p_{X,Y}(x, y \mid \hat{\theta})$? In statistics and estimation tasks, one is concerned with the accuracy of $\hat{\theta}$, the parameters of the model. In particular, is it significant that these particular parameters generate the optimal model from the hypothesis class \mathcal{P} ? In either case, there are (in the current formulation) two considerations that influence the quality of the model. The first is one's ability to solve for $\hat{\theta}$. This is not always possible, particularly in deep learning. Studying this problem amounts

to studying how to optimize the parameters θ of a deep network. The second consideration is the hypothesis space \mathcal{P} . In particular, is it expressive enough to contain a model $p_{X,Y}(x, y \mid \theta)$ that is an accurate estimator for $p_{X,Y}(x, y)$, and furthermore can we find such a model with a finite amount of training data? In machine learning generally, this is the problem of model class selection, e.g., should we use a linear model, a quadratic model, kernel methods, or deep learning? Within deep learning, this may refer to a number of choices having to do with the architecture and design of the network, including the number of layers, the width of each layer, but also more nuanced choices such as how convolutional neural networks leverage additional structure in the data points x when x is an image.

This is a course on the mathematics of deep learning. Deep learning is, in the vast majority of cases, used for machine learning and prediction. However, in studying the mathematics of deep learning, we will attempt to understand which types of models and model classes yield good predictors $p_{X,Y}(x, y \mid \hat{\theta})$. In certain cases this may help in understanding the role and significance of the specific $\hat{\theta}$, although we will most likely not directly address this type of consideration; for more information in that direction, one should investigate causal inference in machine learning.

Remark 1.4. We can extend the framework at the beginning of this section to continuous random variables. In particular suppose that $\mathcal{X} = \mathbb{R}^d$ and $\mathcal{Y} = \mathbb{R}$. If X is a continuous random variable that takes values in $\mathcal{X} = \mathbb{R}^d$, then there exists a probability density function $p_X(x)$ such that

$$\mathbb{P}_X(A) = \int_A p_X(x) dx, \quad A \subseteq \mathcal{X} = \mathbb{R}^d.$$

Furthermore, suppose that Y conditioned on $X = x$ is a continuous random variable, which means there is a probability density function $p_{Y|X}(y \mid x)$ such that

$$\mathbb{P}_{Y|X}(B \mid X = x) = \int_B p_{Y|X}(y \mid x) dy, \quad B \subseteq \mathcal{Y} = \mathbb{R}.$$

The joint probability density function of X, Y is

$$p_{X,Y}(x, y) = p_X(x)p_{Y|X}(y \mid x),$$

which means that the probability of A and B occurring is:

$$\mathbb{P}_{X,Y}(A, B) = \int_A \int_B p_{X,Y}(x, y) dy dx = \int_A \int_B p_X(x)p_{Y|X}(y \mid x) dy dx.$$

If Y is discrete, meaning without loss of generality that $\mathcal{Y} = \{1, \dots, M\}$, we can amend the previous discussion as follows. In this case the part concerning X remains the same but for each of the M possible values of Y , we have a probability that Y takes the value conditioned on X :

$$\mathbb{P}_{Y|X}(Y = y \mid X = x) = p_{Y|X}(y \mid x), \quad y \in \mathcal{Y} = \{1, \dots, M\},$$

In this case the joint probability density function is

$$p_{X,Y}(x, y) = p_X(x)p_{Y|X}(y | x),$$

and the probability of $A \subseteq \mathcal{X} = \mathbb{R}^d$ and $B \subseteq \mathcal{Y} = \{1, \dots, M\}$ occurring is

$$\mathbb{P}_{X,Y}(A, B) = \int_A \sum_{y \in B} p_{X,Y}(x, y) dx = \int_A \sum_{y \in B} p_X(x)p_{Y|X}(y | x) dx.$$

1.1.4 Supervised vs unsupervised learning

Let us clarify a bit the difference between supervised learning and unsupervised learning in this probabilistic framework that we have developed. In unsupervised learning we are only given data points $\{x_i\}_{i=1}^N \subseteq \mathcal{X}$ and we are interested in extracting patterns from the data or generating new data points; often this means our primary concern is estimating $p_X(x)$. In supervised learning, on the other hand, as we have already described we are given a training set $T = \{(x_i, y_i)\}_{i=1}^N$ consisting of data points $\{x_i\}_{i=1}^N \subseteq \mathcal{X}$ and labels $\{y_i\}_{i=1}^N \subseteq \mathcal{Y}$. Our primary concern is, given a new data point x , our ability to estimate its label $y(x)$.

Given the discussion in Sections 1.1.2 and 1.1.3, it is natural to model our candidate probability density functions $p_{X,Y}(x, y | \theta)$ by conditioning on x :

$$p_{X,Y}(x, y | \theta) = p_X(x | \theta)p_{Y|X}(y | x, \theta).$$

Recall from (1.2) we want to maximize $p_{X,Y}(T | \theta)$ over all choices of $\theta \in \mathbb{R}^n$. We have:

$$\hat{\theta} = \arg \max_{\theta \in \mathbb{R}^n} p(T | \theta) = \arg \max_{\theta \in \mathbb{R}^n} \prod_{i=1}^N p(x_i, y_i | \theta),$$

since the right hand side is proportional to $p(T | \theta)$. Notice as well that we can take the logarithm, and still obtain the same $\hat{\theta}$, that is:

$$\begin{aligned} \hat{\theta} &= \arg \max_{\theta \in \mathbb{R}^n} \sum_{i=1}^N \log p(x_i, y_i | \theta) \\ &= \arg \max_{\theta \in \mathbb{R}^n} \left(\sum_{i=1}^N \log p_X(x_i | \theta) + \sum_{i=1}^N \log p_{Y|X}(y_i | x_i, \theta) \right). \end{aligned} \quad (1.3)$$

Now in unsupervised learning, there are no labels, and so we only have the first summation in (1.3), and indeed maximizing that summation will give us the maximum likelihood estimator for $p_X(x)$. In generative modeling, we can then sample from $p_X(x | \hat{\theta})$ to generate new data points. In supervised learning, remember we are primarily interested in the problem of given a new data point x , coming up with an accurate estimate for its label $y(x)$. This is encoded by the second summation, and so we often discard the first summation in this case. We will come back to this after we discuss the functional modeling perspective of supervised learning next in Section 1.2.

1.2 The supervised learning recipe

1.2.1 Functional models

We briefly describe the basic steps involved in supervised learning from a functional modeling perspective.

In supervised learning one is given a set of data that is partitioned into a training set $T = \{(x_i, y_i)\}_{i=1}^N$, consisting of data points $\{x_i\}_{i=1}^N \subset \mathcal{X}$ and associated labels $\{y_i\}_{i=1}^N \subset \mathcal{Y}$ that one is able to use to fit a model, and a test set T_{test} consisting of other data points and labels (x, y) not in the training set, and which are not to be used to fit the model but rather are to be used to evaluate the quality of the model fitted to the training data. Analogous to the hypothesis class \mathcal{P} , one defines a parameterized model class \mathcal{F}

$$\mathcal{F} = \{f(x; \theta) : \theta \in \mathbb{R}^n\}$$

consisting of candidate models $f(x; \theta)$ that are parameterized by $\theta \in \mathbb{R}^n$. For example, the class of linear models is given by:

$$\mathcal{F}_{\text{linear}} = \{f(x; \theta) = \langle x, \theta \rangle : \theta \in \mathbb{R}^d\},$$

where $\langle x, \theta \rangle$ is the standard dot product,

$$\langle x, \theta \rangle = \sum_{k=1}^d x(k)\theta(k).$$

One also defines a loss function $\ell(y, f(x))$ that measures the cost of a model $f(x)$ differing from a label $y = y(x)$; almost always we will take $\ell(y, f(x)) = |y - f(x)|^2$, the squared loss.

To select a model from the model class \mathcal{F} , we minimize the average loss over the training set:

$$\begin{aligned} \mathcal{L}(\theta) &= \frac{1}{N} \sum_{i=1}^N \ell(y_i, f(x_i; \theta)) \\ \hat{\theta} &= \arg \min_{\theta \in \mathbb{R}^n} \mathcal{L}(\theta) = \arg \min_{\theta \in \mathbb{R}^n} \left[\frac{1}{N} \sum_{i=1}^N \ell(y_i, f(x_i; \theta)) \right]. \end{aligned}$$

We then evaluate the quality of the selected model, $f(x; \hat{\theta})$, by evaluating it on the test set and computing the average loss over the test set:

$$\text{average test error} = \frac{1}{|T_{\text{test}}|} \sum_{(x, y) \in T_{\text{test}}} \ell(y, f(x; \hat{\theta})).$$

Similarly to the discussion in Section 1.1, the quality of the arrived upon model depends on two points. The first is, can one solve for the parameters $\hat{\theta}$ or some other parameters θ that are nearly as good? Second, the choice of model

class \mathcal{F} is incredibly important, as it determines the set of possible models from which we will select one. There needs to be a model in \mathcal{F} that simultaneously fits the training data T while at the same time does not overfit to spurious patterns in the training set so that it generalizes well to the test set. This is a complicated problem because for each new data set or task, the underlying distribution $p_{Y|X}(y | x)$ will change and the relationship between data point x and label $y(x)$ will thus change. We will thus want algorithms that are able to adapt to a multitude of scenarios, but which also do not need too many training points to find the (near) optimal model.

1.2.2 Linking the probabilistic model with the functional model

Let us now link together the probabilistic models described in Section 1.1.3 and the functional models described in Section 1.2. To do so, recall Example 1.3 in which we assumed that given a day point x , a label y is generated according to

$$y = F(x) + \varepsilon, \quad \varepsilon \sim \mathcal{N}(0, \sigma^2). \quad (1.4)$$

In other words, there is a deterministic function $F : \mathcal{X} \rightarrow \mathbb{R}$ that is the “true” label, but it is corrupted by a noise ε and we observe the corrupted version. That is, given x , the label of x is sampled from $\mathcal{N}(F(x), \sigma^2)$, which is the normal distribution with mean $F(x)$ and variance σ^2 . In this case, we have

$$p_{Y|X}(y | x) = \frac{1}{\sqrt{2\pi}\sigma} e^{-|y-F(x)|^2/2\sigma^2},$$

since the right hand side is the probability density function for $\mathcal{N}(F(x), \sigma^2)$.

Let us now consider a functional model class $\mathcal{F} = \{f(x | \theta) : \theta \in \mathbb{R}^n\}$ that contains our best guesses for F , which we do not know. Let us take our hypothesis space $\mathcal{P}_{Y|X}$ as:

$$\mathcal{P}_{Y|X} = \{p_{Y|X}(y | x, \theta) : \theta \in \mathbb{R}^n\}, \quad p_{Y|X}(y | x, \theta) = \frac{1}{\sqrt{2\pi}\sigma} e^{-|y-f(x;\theta)|^2/2\sigma^2}.$$

Using (1.3) and the discussion thereafter, we have

$$\begin{aligned} \hat{\theta} &= \arg \max_{\theta \in \mathbb{R}^n} \sum_{i=1}^N \log p_{Y|X}(y_i | x_i, \theta) \\ &= \arg \max_{\theta \in \mathbb{R}^n} \left(-N \log \sqrt{2\pi}\sigma - \frac{1}{2\sigma^2} \sum_{i=1}^N |y_i - f(x_i; \theta)|^2 \right) \\ &= \arg \min_{\theta \in \mathbb{R}^n} \frac{1}{N} \sum_{i=1}^N |y_i - f(x_i; \theta)|^2 \\ &= \arg \min_{\theta \in \mathbb{R}^n} \mathcal{L}(\theta). \end{aligned}$$

In other words, the two formulations are equivalent, at least for the model (1.4). We will come back to this correspondence again when we discuss regularization.

Notice that even if the model of Example 1.3 does not hold, minimizing the squared error over the functional model class is still not a bad idea. Indeed, we see that doing so implicitly puts a normal probability distribution with mean $f(x; \hat{\theta})$ over the possible labels for x . This will allow us to give probabilities of each label, which in turn can allow one to estimate uncertainty.

1.3 Two methods

We describe two basic methods for machine learning, linear regression and k -nearest neighbors, which will serve as a basis for further discussion. Linear regression is a simple class of models that are easy to fit with a few training points, but which will not fit labels $y(x)$ that depend non-linearly on the data x . On the other hand, k -nearest neighbors is a flexible class of models that can fit many different patterns, but which suffers from the curse of dimensionality and thus needs many training points as the dimension of x increases.

1.3.1 Linear regression

Let us now consider a more concrete scenario, linear regression. In the language of Section 1.2.1, linear regression models form the class of affine functions over x , i.e., in this section we consider:

$$\mathcal{F} = \left\{ f(x; \theta) = \theta(0) + \sum_{k=1}^d \theta(k)x(k) : \theta \in \mathbb{R}^{d+1} \right\}.$$

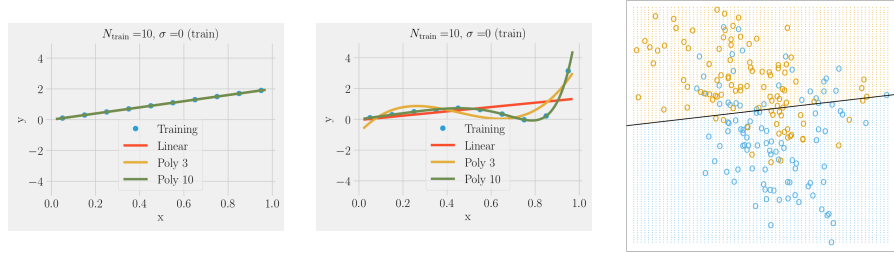
Linear regression models are affine over x but are linear over the augmented data point $(1, x) \in \mathbb{R}^{d+1}$. The extra parameter $\theta(0)$ is often referred to as the bias.

Given a training set $T = \{(x_i, y_i)\}_{i=1}^N$ we seek to find a linear model that best fits the data by minimizing the squared loss:

$$\mathcal{L}(\theta) = \frac{1}{N} \sum_{i=1}^N (y_i - \langle (1, x_i), \theta \rangle)^2$$

We can solve for the minimum of $\mathcal{L}(\theta)$ analytically. To do so we use matrix notation. Define the $(d+1) \times N$ matrix \mathbf{X} and the $N \times 1$ vector \mathbf{y} as:

$$\mathbf{X} = \begin{pmatrix} 1 & \cdots & 1 \\ | & & | \\ x_1 & \cdots & x_N \\ | & & | \end{pmatrix} \quad \mathbf{y} = \begin{pmatrix} y_1 \\ \vdots \\ y_N \end{pmatrix}.$$



(a) Data generated from a model in which $y(x)$ is an affine function of $x \in \mathbb{R}$. The linear regression model fits the data perfectly.

(b) Data generated from a model in which $y(x)$ is a 10th order polynomial of $x \in \mathbb{R}$. The linear regression model does not fit the data perfectly, but the 10th order polynomial regression model does.

(c) Two dimensional data with two classes, orange (+1) and blue (-1). The thick black line is the level line of $f(x; \hat{\theta}) = \langle (1, x), \hat{\theta} \rangle = 0$. Points above the level line are classified as orange; points below the level line are classified as blue.

Figure 1.2: Examples of linear regression and classification. Figures (a) and (b) taken from [6]; figure (c) taken from [7].

We can rewrite $\mathcal{L}(\theta)$ as

$$\mathcal{L}(\theta) = N^{-1}(\mathbf{y} - \mathbf{X}^T \theta)^T (\mathbf{y} - \mathbf{X}^T \theta),$$

where we have considered $\theta \in \mathbb{R}^{d+1}$ as a $(d+1) \times 1$ vector. Differentiating $\mathcal{L}(\theta)$ and setting it equal to $\mathbf{0}$ (the vector of zeros), one obtains:

$$\nabla_{\theta} \mathcal{L}(\theta) = N^{-1} \mathbf{X}(\mathbf{y} - \mathbf{X}^T \theta) = \mathbf{0} \implies \mathbf{X}(\mathbf{y} - \mathbf{X}^T \theta) = \mathbf{0}.$$

Solving for θ one obtains, assuming $\mathbf{X}\mathbf{X}^T$ is invertible,

$$\hat{\theta} = (\mathbf{X}\mathbf{X}^T)^{-1} \mathbf{X}\mathbf{y}. \quad (1.5)$$

The solution $\hat{\theta}$ is the set of optimal weights that fits the training data, regardless of whether the relationship between x and $y(x)$ is affine. When indeed $y(x)$ is an affine function of x and $\mathbf{X}\mathbf{X}^T$ is invertible (necessarily then, $N \geq p$), the model $\hat{\theta}$ will not only fit the data but will be the true underlying model. Figure 1.2 illustrates both scenarios.

Regularization

Model overfit occurs when we use too complex of a model to fit the training data, and thus fit spurious patterns from noise or other nuisance factors that reduce the ability of the model to generalize to new data (e.g. test points).

For example, in Figures 1.2a and 1.2b, training data $T = \{(x_i, y_i)\}_{i=1}^N$ is generated from a linear model and a 10th order polynomial model. In both

cases a 10th order polynomial model is fit to the data, with success. When $\mathcal{X} \subseteq \mathbb{R}$ an $(n - 1)$ st order polynomial function class consists of models

$$f(x; \theta) = \sum_{k=0}^{n-1} \theta(k) x^k. \quad (1.6)$$

If one again uses the squared loss to evaluate the quality of the model, i.e.,

$$\mathcal{L}(\theta) = \frac{1}{N} \sum_{i=1}^N \left(y_i - \sum_{k=0}^{n-1} \theta(k) x_i^k \right)^2, \quad (1.7)$$

then one can still use (1.5) to solve for $\hat{\theta} = \arg \min_{\theta \in \mathbb{R}^n} \mathcal{L}(\theta)$ by redefining \mathbf{X} as

$$\mathbf{X}^T = \begin{pmatrix} 1 & x_1 & x_1^2 & \cdots & x_1^{n-1} \\ \vdots & \vdots & \vdots & \cdots & \vdots \\ 1 & x_N & x_N^2 & \cdots & x_N^{n-1} \end{pmatrix}.$$

This is a particular form of a dictionary model, consisting of the new representation $\Phi(x) = (\phi_k(x))_{k=0}^{n-1} \in \mathbb{R}^n$ with $\phi_k(x) = x^k$. Linear regression over $\Phi(x)$ consists of models $f(x; \theta) = \langle \Phi(x), \theta \rangle$, which is exactly (1.6).

As n increases the complexity of the polynomial functional class increases, as it contains all polynomial models of order less than $n - 1$ as well. However, fitting more complex models can become more delicate in the presence of noise or other confounding factors, as the added complexity can allow the models to fit the noise, which is not desirable. This is not shown in Figures 1.2a and 1.2b because there is no noise in the data. Figure 1.3 illustrates the point, as the data in this figure is generated from a linear model, but the labels y_i are corrupted by a small amount of additive white noise. In this case, fitting the data with a linear model and a third order polynomial model yields good interpolative and extrapolative models, but models fit with higher order polynomials such as 10th order overfit to the noise and cannot extrapolate and even do worse in interpolation.

This example may be a bit disheartening as it would seem to indicate that we need to know the appropriate complexity of the model class in advance, which is often not possible. However, we must remember that 10th order polynomials include 3rd order polynomials and 1st order polynomials. The loss function that we minimized, in this case (1.7), however selected an overly complex model because it minimized the mean squared error on the noisy training data. The question then becomes, how can we use a model class \mathcal{F} , such as 10th order polynomials, that includes complex models for when we need them (as in Figure 1.2b) but from which we can also draw a less complex model when the situation calls for it (as in Figure 1.3)? One answer to this question is *regularization*. In this case the loss function is amended to incorporate a regularization function that acts on the parameters θ , in many cases restricting the parameter

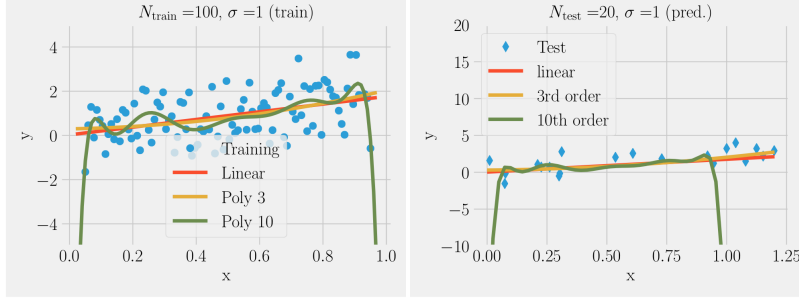


Figure 1.3: Linear and polynomial models fit to noisy data generated from a linear model plus noise. Left panel: Models fit to training data with $x_i \in [0, 1]$. Right panel: Models evaluated on test data with test points $x \in [0, 1.25]$. The linear and 3rd order polynomials fit the data well and do not fit the noise, and thus are capable of extrapolating to test data outside the range of the training data. The 10th order polynomial, however, fits spurious patterns in the noise and thus interpolates with a less regular model and cannot extrapolate. Figure taken from [6].

set:

$$\mathcal{L}_{\mathcal{R}}(\theta, \lambda) = \frac{1}{N} \sum_{i=1}^N \ell(y_i, f(x_i; \theta)) + \lambda \mathcal{R}(\theta).$$

The hyper-parameter λ balances the fit to the training data (the first term) with the strength of regularization on the parameters θ induced through the regularizer $\mathcal{R}(\theta)$. Setting $\lambda = 0$ leads to a high complexity model being selected from the model class \mathcal{F} , whereas larger values of λ increasingly restrict the viable parameters θ , thus reducing the complexity of the model $\hat{\theta}$ that minimizes $\mathcal{L}_{\mathcal{R}}(\theta)$.

In linear and polynomial regression with a squared loss $\ell(y, f(x)) = |y - f(x)|^2$, there are two common choices for $\mathcal{R}(\theta)$. The first is ridge regression, which uses an ℓ^2 regularization:

$$\mathcal{R}_2(\theta) = \|\theta\|_2^2 = \sum_{k=1}^n |\theta(k)|^2.$$

Ridge regression models, e.g. the following in the case of linear regression,

$$\hat{\theta}_{\text{ridge}} = \arg \min_{\theta} \left[\frac{1}{N} \sum_{i=1}^N |y_i - \langle (1, x_i), \theta \rangle|^2 + \lambda \sum_{k=0}^d |\theta(k)|^2 \right],$$

penalize very large parameters $\theta(k)$ via the ℓ^2 regularization, and thus result in models that have a few large parameters $\theta(k)$, with the remaining parameters

being small but rarely zero⁴. The larger λ , the fewer larger parameters and the more small parameters in θ . More precisely, one can show that

$$\hat{\theta}_{\text{ridge}} = (\mathbf{X}\mathbf{X}^T + \lambda\mathbf{I})^{-1}\mathbf{X}\mathbf{y},$$

indicating that the size of the weights are depressed, approximately, by a factor of $(1 + \lambda)$. In fact this is exactly the case if the rows of \mathbf{X} are orthonormal, since in that case $\mathbf{X}\mathbf{X}^T = \mathbf{I}$.

The second common regularization is LASSO (least absolute shrinkage and selection operator), which uses an ℓ^1 regularization:

$$\mathcal{R}_1(\theta) = \|\theta\|_1 = \sum_{i=1}^n |\theta(i)|.$$

LASSO models, e.g. the following in the case of linear regression,

$$\hat{\theta}_{\text{LASSO}} = \arg \min_{\theta} \left[\frac{1}{N} \sum_{i=1}^N |y_i - \langle (1, x_i), \theta \rangle|^2 + \lambda \sum_{i=0}^d |\theta(i)| \right],$$

penalize non-zero parameters $\theta(i)$ via the ℓ^1 regularization, and thus result in sparse models. With larger λ there are fewer non-zero parameters and the model increases in sparsity. In fact, when the columns of \mathbf{X} are orthogonal, one obtains:

$$\hat{\theta}_{\text{LASSO}}(k) = \text{sign}(\hat{\theta}_{\lambda=0}) \max(|\hat{\theta}_{\lambda=0}(k)| - \lambda, 0),$$

where $\hat{\theta}_{\lambda=0}$ is the simple least squares optimal model with no regularization (i.e., $\lambda = 0$). The above formula shows that the ℓ^1 regularization of LASSO shrinks the weights by an additive factor of $-\lambda$, down to zero, thus resulting in sparse models.

One additional point that is important to remember is that both ridge and LASSO regularized regressions implicitly define new, restricted model subclasses of the linear regression class. In particular, for each $\lambda > 0$ there exists a $t = t(\lambda) < \infty$ such that

$$f(\cdot; \hat{\theta}_{\text{ridge}}) \in \mathcal{F}_{2,t} = \{f(x; \theta) = \langle (1, x), \theta \rangle : \|\theta\|_2 \leq t\}$$

and

$$f(\cdot; \hat{\theta}_{\text{LASSO}}) \in \mathcal{F}_{1,t} = \{f(x; \theta) = \langle (1, x), \theta \rangle : \|\theta\|_1 \leq t\}.$$

Thus regularization implicitly defines a new model class $\mathcal{F}_{\lambda, \mathcal{R}} \subseteq \mathcal{F}$ that depends on the regularizer \mathcal{R} and the strength of the regularization λ .

In practice, one must estimate the hyper-parameter λ using only the training data. To do so, one employs *cross validation*. Figure 1.4 describes the idea.

When the label function $y(x)$ is not an affine function of x , a nonlinear model is required. One option, also discussed in this section, are polynomial

⁴It makes a lot of sense to have the penalty start at $k = 1$, thus omitting the bias $\theta(0)$. However, this choice will complicate our analysis, so we will not pursue it further. Thanks to Cullen Haselby for pointing this out.

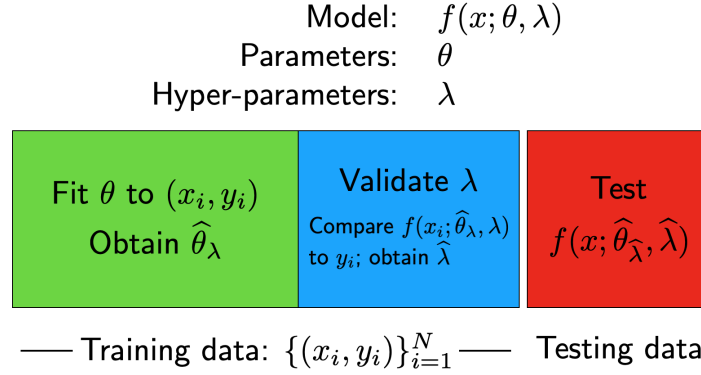


Figure 1.4: Cross validation. A model $f(x; \theta, \lambda)$ consists of parameters θ and hyper-parameters λ . The training set $T = \{(x_i, y_i)\}_{i=1}^N$ is partitioned into two sets, green and blue. The green set is used to fit the parameters θ , for a fixed λ , to the data pairs (x_i, y_i) in this set. The blue set is used to validate the model, meaning one evaluates $f(x_i; \hat{\theta}_\lambda, \lambda)$ against the true label y_i for (x_i, y_i) in the green set. One does this for a finite number of λ and selects the optimal $\hat{\lambda}$ based on which one has the smallest validation error. Finally, the model $f(x; \hat{\theta}_{\hat{\lambda}}, \hat{\lambda})$ is tested on different data (x, y) in the test set.

models. The number of parameters in a polynomial model, though, scales poorly in the dimension. If $n - 1$ is the order of the polynomial and d is the dimension, then $\#(\theta) = O(d^n)$. Kernel methods and in particular polynomial kernels remedy this. However, not every label function $y(x)$ is a global polynomial of the data points x . Sometimes the label function is based on locality or other nonlinear patterns that are not so easily modeled by compact mathematical formulas. Section 1.3.3 describes k -nearest neighbor models, which are based on local similarities and make very few assumptions about the way the data is generated. First though we return to the probabilistic models and interpret regularization from this perspective.

1.3.2 Bayes and maximum a posteriori

Recall from Section 1.1.3 we defined maximum likelihood estimation (MLE) as:

$$\hat{\theta}_{\text{MLE}} = \arg \max_{\theta \in \mathbb{R}^n} p_{X,Y}(T | \theta) = \arg \max_{\theta \in \mathbb{R}^n} p(T | \theta),$$

where we recall $T = \{(x_i, y_i)\}_{i=1}^N$ is our training set. In other words, we maximized the probability of observing the particular training set T that we have, given the model parameters θ . On the other hand, a more intuitive and perhaps useful object might be $p(\theta | T)$, the probability of the model θ being correct given the training set T . Indeed, this is how we generally think of machine learning. We are given a training set T and we pick a model θ that has the

highest chance of describing the data. This is called the *maximum a posteriori* estimator:

$$\hat{\theta}_{\text{MAP}} = \arg \max_{\theta \in \mathbb{R}^n} p(\theta | T). \quad (1.8)$$

The question then becomes, how do we compute $\hat{\theta}_{\text{MAP}}$? For this, we can use Bayes' Theorem:

$$p(\theta | T) \propto p(T | \theta)p(\theta).$$

Notice that right hand side contains the term $p(T | \theta)$, which is what we used to compute the MLE model. But it also contains another, new term, $p(\theta)$. What is this term? It is referred to as the *prior distribution* on the parameters θ . It encodes any prior knowledge we have about our model, or behavior that we want to build into our model. For example, we can place a Gaussian/normal prior on θ , meaning that we assume each parameter $\theta(k)$ is distributed according to the normal distribution with mean zero and variance proportional to λ^{-1} :

$$p(\theta) = p(\theta | \lambda) = \prod_{k=1}^n \sqrt{\frac{\lambda}{2\pi}} e^{-\lambda|\theta(k)|^2} \quad (\text{normal prior}).$$

We can also use a Laplace prior:

$$p(\theta) = p(\theta | \lambda) = \prod_{k=1}^n \frac{\lambda}{2} e^{-\lambda|\theta(k)|} \quad (\text{Laplace prior}).$$

Let us now see how the $\hat{\theta}_{\text{MAP}}$ model is related to regularized functional models, and in particular ridge regression for the normal prior and LASSO for the Laplace prior. Using (1.8) we have:

$$\begin{aligned} \hat{\theta}_{\text{MAP}} &= \arg \max_{\theta \in \mathbb{R}^n} p(\theta | T) \\ &= \arg \max_{\theta \in \mathbb{R}^n} p(T | \theta)p(\theta) \\ &= \arg \max_{\theta \in \mathbb{R}^n} (\log p(T | \theta) + \log p(\theta)) \end{aligned}$$

Recall from (1.3) that if the $\{x_i\}_{i=1}^N$ are sampled iid (independently and identically distributed) from \mathcal{X} , then we can decompose $\log p(T | \theta)$ as:

$$\hat{\theta}_{\text{MAP}} = \arg \max_{\theta \in \mathbb{R}^n} \left(\sum_{i=1}^N \log p_X(x_i | \theta) + \sum_{i=1}^N \log p_{Y|X}(y_i | x_i, \theta) + \log p(\theta) \right).$$

Furthermore, in the case of supervised learning we will often discard the first term involving p_X , and, as we saw in Section 1.2.2, taking

$$p_{Y|X}(y | x, \theta) = e^{-|y - f(x; \theta)|^2 / 2\sigma^2},$$

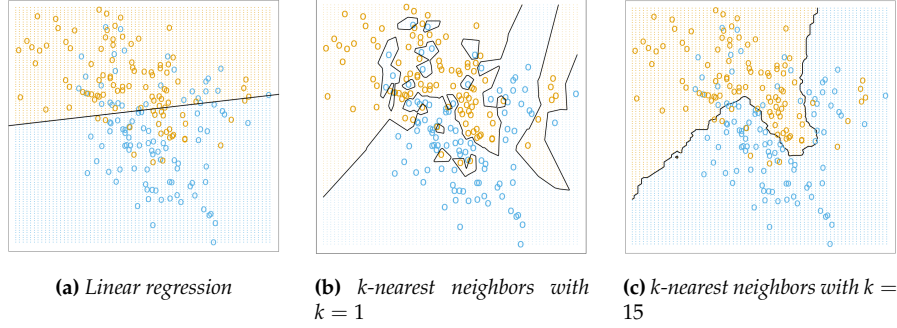


Figure 1.5: Two dimensional data with two classes, orange (+1) and blue (-1). Three different classifiers and their respective decision boundaries. Figure taken from [7].

is a reasonable choice. We then have:

$$\hat{\theta}_{\text{MAP}} = \arg \min_{\theta \in \mathbb{R}^n} \left(\frac{1}{2\sigma^2} \sum_{i=1}^N |y_i - f(x_i; \theta)|^2 - \log p(\theta) \right).$$

We thus see that $\log p(\theta)$ serves as the regularization term on the parameters θ . In particular, if we use the Gaussian/normal prior, we obtain:

$$\hat{\theta}_{\text{MAP}} = \arg \min_{\theta \in \mathbb{R}^n} \left(\frac{1}{2\sigma^2} \sum_{i=1}^N |y_i - f(x_i; \theta)|^2 + \lambda \sum_{k=1}^n |\theta(k)|^2 \right),$$

which is precisely ridge regression if $f(x; \theta) = \langle (1, x), \theta \rangle$. Similarly, if we use the Laplace prior, we obtain LASSO.

1.3.3 *k*-Nearest neighbors and local methods

Figure 1.2c illustrates the need for a nonlinear classifier, capable of learning a nonlinear decision boundary as opposed to a straight line. A polynomial method could do better, but this still imposes a modeling assumption on the underlying data generation process. In order to avoid such assumptions, we instead turn to another type of method, *k*-nearest neighbors, which is a local method.

Let $N_k(x)$ denote k nearest of neighbors of x , in the Euclidean distance, from the set of training points $\{x_1, \dots, x_N\} \subset \mathbb{R}^d$. The *k*-nearest neighbor model is:

$$f(x; k) = \frac{1}{k} \sum_{x_i \in N_k(x)} y_i.$$

Figure 1.5 illustrates the model for $k = 1$ and $k = 15$, and compares to the linear classifier from Figure 1.2c. In the figure we see that the *k*-nearest neighbors classifiers make significantly fewer mis-classifications on the training set than

the linear regression classifier. However, classification rate on the training set is not the sole criterion for the quality of a model, as we saw in our discussion on regularization. Indeed models must generalize well to unseen points. The $k = 1$ nearest neighbor model in fact makes no mis-classifications on the training set, since it simply assigns to each training point its own label. But the decision boundary of the 1-nearest neighbor classifier is extremely irregular and likely to make mistakes on test data. The $k = 15$ nearest neighbor classifier, on the other hand, has a semi-regular boundary that is likely to generalize better than either the 1-nearest neighbor classifier (because it is too rough) or the linear regression classifier (because it is too smooth). This discussion is hinting at the bias-variance tradeoff in machine learning, which we will make more precise in Section 1.4. It also illustrates that the $k = 1$ model is, in some sense, more complex than the $k = 15$ model. In the extreme case, $k = N$, all points are classified the same, and so this is the simplest model. In fact, even though there is only one parameter, k , that we set, the effective number of parameters or complexity of the k -nearest neighbor model is N/k .

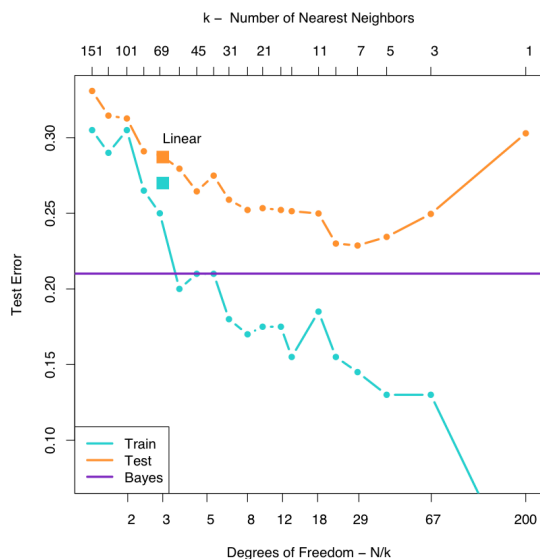


Figure 1.6: k -nearest neighbors classifier for k decreasing from left to right, evaluated on the training set and the test (validation) set. The models with $k \approx 10$ perform the best on the test set, even though the training error is essentially decreasing as $k \rightarrow 1$. Figure taken from [7].

Note that because the 1-nearest neighbor classifier will always make zero errors on the training set, we require a different method by which to select k . We instead use cross-validation, which requires a validation set separate from the training set, but for which we still know the correct labels. In fact, many hyper-parameters, including k in k -nearest neighbors and λ in regularized linear models, are selected through cross-validation. The way it works

for k -nearest neighbors is that we use the original training set T to define the neighborhoods of each point and the model $f(x; k)$, but we then evaluate this model on the validation set and compute the average error on the validation set. The k with the lowest average error on the validation set is the model we use on the test set. See Figure 1.6 for an illustration using k -nearest neighbors. The more general principle, which includes k -nearest neighbors and the regularized linear regularization, is based on the bias variance tradeoff, which will be discussed in the next section.

1.4 Basics of statistical learning theory

Figure 1.6 illustrates an important concept in machine learning, which is the bias-variance tradeoff. Later in Section 1.4.2 we will discuss this in more detail. First, though, in Section 1.4.1 we examine k -nearest neighbors and linear regression from a statistical point of view. We will also derive the naive Bayes classifier for classification, which will explain the purple line in Figure 1.6. Then in Section 1.4.3 we will discuss the curse of dimensionality.

1.4.1 Statistical view of models

We now consider k -nearest neighbors and linear regression from the perspective of statistical learning theory. Let us return to the assumptions of Section 1.1.3. In particular, recall that we assume we have a probability space $(\mathcal{X}, \Sigma, \mathbb{P}_X)$, $\mathcal{X} = \mathbb{R}^d$, with probability density function $p_X(x)$. We also have the label set \mathcal{Y} , which we will assume is $\mathcal{Y} = \mathbb{R}$. Labels are drawn from the conditional probability distribution $\mathbb{P}_{Y|X}$, which has probability density function $p_{Y|X}(y | x)$, and which together with $p_X(x)$ forms the joint probability density function $p_{X,Y}(x, y) = p_X(x)p_{Y|X}(y | x)$. We draw our training set $T = \{(x_i, y_i)\}_{i=1}^N$ from the joint probability density $p_{X,Y}(x, y)$.

Now let X be a random variable (more precisely, random vector) that takes values in \mathcal{X} according to $p_X(x)$, and Y a random variable dependent upon X that takes values in \mathbb{R} according to $p_{Y|X}(y | x)$, so that in particular the pair (X, Y) take values according to the joint probability distribution $p_{X,Y}(x, y)$. The variables X and Y , in other words, are just like the samples x_i and y_i except that we view them as random variables instead of as fixed samples.

If we had perfect knowledge of \mathcal{X} , \mathcal{Y} and $p_{X,Y}(x, y)$; and we are using the squared loss as our measure of loss, i.e. $\ell(y, f(x)) = |y - f(x)|^2$; and we could pick any model f that we want, then we would minimize the following:

$$\begin{aligned} R_{\text{true}}(f) &= \mathbb{E}_{X,Y}[(Y - f(X))^2] = \int_{\mathcal{X} \times \mathcal{Y}} (y - f(x))^2 p_{X,Y}(x, y) dy dx \\ &= \int_{\mathcal{X}} \int_{\mathcal{Y}} (y - f(x))^2 p_{Y|X}(y | x) dy p_X(x) dx \\ &= \mathbb{E}_X \mathbb{E}_{Y|X}[(Y - f(X))^2 | X] \end{aligned} \quad (1.9)$$

The functional $R_{\text{true}}(f)$ is called the *true risk* of the model f . It measures the quality of the model f if we had an oracle that told us everything about the data generation process. In practice, of course, all we are given is the finite number of training points $T = \{(x_i, y_i)\}_{i=1}^N$. So instead of minimizing the true risk, which is almost always impossible because we simply do not have complete knowledge (in fact if we did, we would not be doing machine learning!), we instead minimize the *empirical risk*, which is the loss function we saw earlier:

$$R_{\text{emp}}(f) = \frac{1}{N} \sum_{i=1}^N (y_i - f(x_i))^2.$$

Note that the empirical risk is the finite sample average of $(Y - f(X))^2$. By the Law of Large Numbers, $R_{\text{emp}}(f) \rightarrow R_{\text{true}}(f)$ as $N \rightarrow \infty$ for a *fixed* f . This does not imply, though, that the minimizer of the empirical risk converges to the minimizer of the true risk as $N \rightarrow \infty$. Rather this is a topic in statistical learning theory and must be proved for each new model class.

We will study the ramifications of minimizing the empirical risk in Section 1.4.2. For now, let us return to the theoretical scenario now and minimize the true risk, $R_{\text{true}}(f)$. From (1.9) we see that we can solve for the optimal f point-wise, i.e., by finding $\hat{f}(x)$ one x at a time. In particular, we obtain:

$$\hat{f}(x) = \arg \min_c \mathbb{E}_{Y|X}[(Y - c)^2 \mid X = x].$$

Taking the derivative with respect to c and setting it equal to zero, we obtain:

$$\hat{f}(x) = \hat{c} = \mathbb{E}_{Y|X}[Y \mid X = x],$$

that is, the conditional expectation of the label Y given that $X = x$. Notice how the optimal model $\hat{f}(x)$ depends only on the conditional expectation, and thus the conditional probability distribution $p_{Y|X}(y \mid x)$, giving more concrete justification to our earlier statement in Section 1.1.4 that in supervised learning we often ignore $p_X(x)$.

Note, in particular, if the label is a deterministic function of x , then seeing x once (i.e., drawing x as a training point x_i) is enough to determine $f(x)$. If there is noise in the labeling process, though, then one sample is not enough. However, in the training set there is typically only one observation $x_i = x$, and furthermore, in the test set we do not know the label and must estimate it. Therefore in k -nearest neighbors we replace $f(x) = \mathbb{E}[Y \mid X = x]$ with

$$f(x) = \text{Avg}[y_i \mid x_i \in N_k(x)] = \frac{1}{k} \sum_{x_i \in N_k(x)} y_i,$$

where we recall $N_k(x)$ consists of the k points from the training set T closest to x . The k -nearest neighbors algorithm is incorporating two approximations:

1. The expectation $\mathbb{E}_{Y|X}[Y \mid X = x]$ is approximated by a k -sample average over the training data.

2. The conditioning at the point $X = x$ is relaxed to conditioning on some region, the k -nearest neighbors in the training set, closest to the point x .

For large sample size N , the points in the k -nearest neighborhood are likely to be close to x , and as k gets larger the average will get more stable. In fact, under certain conditions on $p_{X,Y}(x,y)$, one can show that

$$\text{Avg}[y_i \mid x \in N_k(x_i)] \rightarrow \mathbb{E}_{Y|X}[Y \mid X = x] \text{ as } k, N \rightarrow \infty \text{ with } k/N \rightarrow 0.$$

However, we do not always have enough data points N to well approximate the above limits. In particular, as the dimension d increases, the number of sample points N needed to have k points close to each possible x increases rapidly, thus potentially leading to very bad approximations of $\mathbb{E}_{Y|X}[Y \mid X = x]$ (see Section 1.4.3 for more details).

The nearest neighbor model is good because it places very few assumptions on the underlying data generation process, encoded here by $p_{X,Y}(x,y)$. On the other hand, if we have prior knowledge about the data generation process, leveraging that knowledge and using a more structured class of models is preferred. The linear model described earlier is such a model. Recall the linear model is:

$$f(X; \theta) = \langle X, \theta \rangle = X^T \theta, \quad X, \theta : d \times 1,$$

where we have omitted the bias term but which can easily be incorporated or assumed to already be contained in X (as a constant dimension). Plugging this model into the true risk $R_{\text{true}}(f(\cdot; \theta))$ and minimizing the true risk with respect to θ (solved by differentiating with respect to θ and setting equal to zero) one obtains:

$$\hat{\theta} = (\mathbb{E}_X[XX^T])^{-1} \mathbb{E}_{X,Y}[XY].$$

Note the solution we obtained earlier for the empirical risk, given in (1.5), is the empirical version of the $\hat{\theta}$ obtained here. Note, we do not condition on $X = x$. Rather we use the fact that we solving for an optimal linear model to average over the values of X and Y .

But what about classification? In the previous discussions, we assumed $\mathcal{Y} = \mathbb{R}$, but often we want to do classification and $\#(\mathcal{Y}) = M$, where M is the number of distinct classes. In this case the squared loss does not make sense. A standard choice is the 0-1 loss, which means that

$$\ell(y, f(x)) = \begin{cases} 0 & y = f(x) \\ 1 & y \neq f(x) \end{cases} \quad (1.10)$$

For now let us consider a general loss function, but we will come back to the 0-1 loss function shortly. Regardless of the choice of loss function, the true risk is:

$$R_{\text{true}}(f) = \mathbb{E}_{X,Y}[\ell(Y, f(X))],$$

where we remind the reader that the expectation $\mathbb{E}_{X,Y}$ is taken over the joint probability density function of the data points X and the classes Y , $p_{X,Y}(x,y)$.

Similar to the calculation concluded in (1.9), we can condition on the data points X :

$$\begin{aligned} R_{\text{true}}(f) &= \mathbb{E}_{X,Y}[\ell(Y, f(X))] = \int_{\mathcal{X}} \sum_{y \in \mathcal{Y}} \ell(y, f(x)) p_{X,Y}(x, y) dx \\ &= \int_{\mathcal{X}} \left(\sum_{y \in \mathcal{Y}} \ell(y, f(x)) p_{Y|X}(y | x) \right) p_X(x) dx \\ &= \mathbb{E}_X \left[\sum_{y \in \mathcal{Y}} \ell(y, f(X)) p_{Y|X}(y | X) \right]. \end{aligned}$$

Like before, from this calculation we again see it is sufficient to compute the optimal model, $\hat{f}(x)$, point by point, meaning that:

$$\hat{f}(x) = \arg \min_{y' \in \mathcal{Y}} \sum_{y \in \mathcal{Y}} \ell(y, y') p_{Y|X}(y | x). \quad (1.11)$$

Now, if $\ell(y, f(x))$ is the 0-1 loss function (1.10), we can simplify (1.11) to

$$\hat{f}(x) = \arg \max_{y' \in \mathcal{Y}} p_{Y|X}(y' | x). \quad (1.12)$$

Indeed, suppose $\hat{y} = \hat{f}(x)$ in (1.11); we then have, since $\ell(y, y')$ is the 0-1 loss,

$$\min_{y' \in \mathcal{Y}} \sum_{y \in \mathcal{Y}} \ell(y, y') p_{Y|X}(y | x) = \sum_{y \neq \hat{y}} p_{Y|X}(y | x).$$

But then it must be that we removed the largest value from among $\{p_{Y|X}(y | x) : y \in \mathcal{Y}\}$ in the summation on the right hand side since the 0-1 loss weights all errors equally, i.e., it must be that \hat{y} is the most probable class; but that is (1.12)⁵. The model (1.12) is called the *naive Bayes classifier*. It says, given a data point x , assign the most probable class using the conditional probability $p_{Y|X}(y | x)$; in retrospect, this is sort of obvious. The catch is that we don't know $p_{Y|X}(y | x)$, and so we must estimate it. The k -nearest neighbors algorithm is one way of doing so, and it can be pretty good; see Figure 1.7.

Remark 1.5. The Bayes classifier is not perfect because of uncertainty in the data generation process. The source of that uncertainty is similar to the noisy label model (1.4), but it is generated differently since here the labels are binary. Rather, the uncertainty comes from the way in which the two classes are distributed over \mathbb{R}^2 . Here is a simpler example to give you an idea of what is going on. Consider data in \mathbb{R}^2 generated from a Gaussian mixture model consisting of two Gaussians, i.e.,

$$p_X(x) = \frac{1}{2} \left[\frac{1}{2\pi\sigma_1^2} e^{-\|x-\mu_1\|_2^2/2\sigma_1^2} + \frac{1}{2\pi\sigma_2^2} e^{-\|x-\mu_2\|_2^2/2\sigma_2^2} \right].$$

⁵Thanks to Ali Zare for discussions related to this derivation and for helping to make the presentation clearer

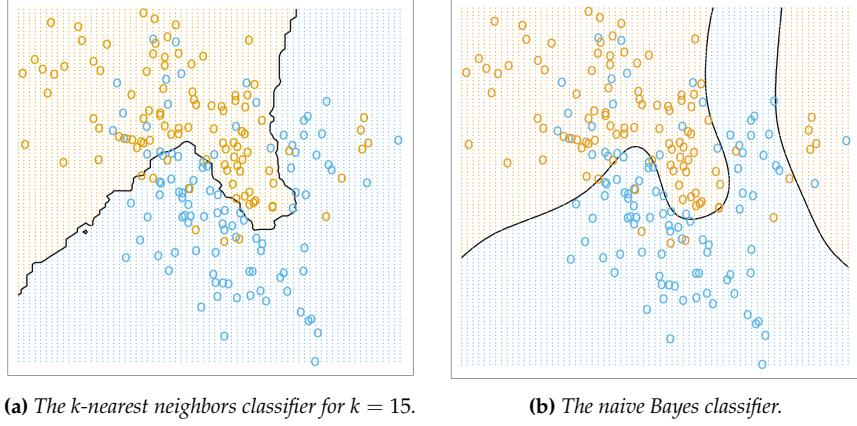


Figure 1.7: Comparison of the k -nearest neighbor classifier and the theoretical naive Bayes classifier. The k -nearest neighbor classifier does a good job of approximating the optimal Bayes decision boundary. Figures taken from [7].

The way that data is sampled from $p_{X,Y}(x) = p_X(x)p_{Y|X}(y | x)$ is the following. First we flip a coin. If it lands heads we sample $x \sim \mathcal{N}(\mu_1, \sigma_1)$ and we assign x the label $y = \text{blue}$ (-1). If it lands tails, we sample $x \sim \mathcal{N}(\mu_2, \sigma_2)$ and we assign x the label $y = \text{orange}$ (+1). If the means μ_1, μ_2 are close enough and the standard deviations σ_1, σ_2 large enough, there will be significant overlap between the two distributions which creates uncertainty in the label of the point; see Figure 1.8. In our probabilistic framework, this means $p_{Y|X}(y | x)$ varies depending on the location of x , and in particular, $p_{Y|X}(\text{blue} | x) = p_{Y|X}(\text{orange} | x) = 1/2$ for x that are equidistant from μ_1 and μ_2 if $\sigma_1 = \sigma_2$.

1.4.2 Bias variance tradeoff

In the previous section we considered theoretical scenarios in which we had perfect knowledge of the data generating distribution $p_{X,Y}(x, y)$ and we were able to select any model $y = f(x)$ to fit the data. In practice, we only have a finite amount of data given by our training set $T = \{(x_i, y_i)\}_{i=1}^N$, and we can only select models from the model class \mathcal{F} that we specify. This leads to two sources of additional error, beyond errors that may be impossible to avoid even with perfect knowledge (e.g., if the labels are noisy, as in the model given by (1.4)). These two additional sources of errors are:

1. *Bias*: Since we cannot pick any model that we wish, only models from the model class \mathcal{F} , there is the possibility that \mathcal{F} does not contain a model that fully captures the relationship between the labels $y \in \mathcal{Y}$ and the data points $x \in \mathcal{X}$. In this case, even the best model from \mathcal{F} will not optimally model this relationship, and the resulting error is called a bias error. For example, if we use linear regression to fit data in which there

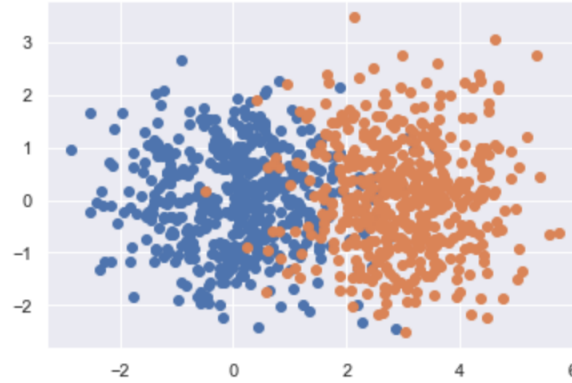


Figure 1.8: Two Gaussian mixture model in which there is significant overlap between the two Gaussians. Determining the label of new points sampled in the overlap region is difficult and creates uncertainty, even for the naive Bayes classifier with complete knowledge of the data generation process.

is a nonlinear relationship between y and x , then our machine learned model will be biased since it will not be able capture the nonlinearity in the data. Furthermore, it may be that \mathcal{F} contains a good model, but we are incapable of selecting it with the training set T that we are given; this will also lead to bias error.

2. *Variance:* Since we do not have complete knowledge of $p_{X,Y}(x,y)$ and in fact only have a finite training set T sampled according to $p_{X,Y}(x,y)$, we must estimate the generalization error and select the best model using only T . However, different (theoretical) draws of the training set T will lead to different models being selected, some of which may generalize to new points better than others. This randomness imparted into the model selection process can either be relatively small or drastic. The variance error encodes this source of error.

Figure 1.9 illustrates the difference between model classes \mathcal{F} with low bias and high variance, versus those with high bias and low variance. Indeed, these two sources of error are often in tension, i.e., reducing one increases the other. This phenomenon is referred to as the *bias-variance trade-off*. One way to think about this tradeoff is as follows. Simple models, such as linear regression, may not fit the training data perfectly (or even well), and hence have a high bias, but their predictions are robust to small perturbations in the test points, and thus have a low variance. On the other hand, complex models such as the 1-nearest neighbor classifier may be able to fit the training data extremely well (low bias), but their high complexity means they may fit spurious patterns in the data (such as noise) and their output may change drastically with small changes on the evaluated data point, so much so that their predictive power is

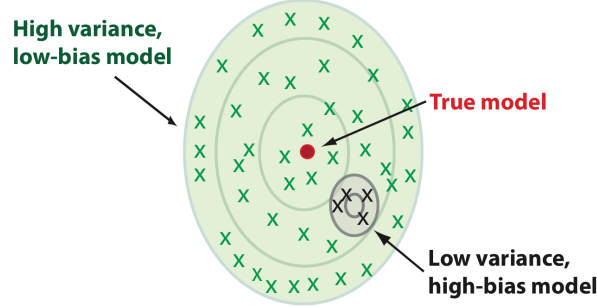


Figure 1.9: Illustration of high variance, low bias model classes and low variance, high bias model classes. Each “X” denotes a model obtained through a particular draw of the training set T . In the high variance, low bias regime, indicated by the green X’s, the average of all the models is close to the true model, marked with the red dot. However, there is a large variance between models as one varies the training set, meaning that any one model may be very far from the true model. In the low variance, high bias regime, indicated with the black X’s, the models are not centered around the true model and thus the average model obtained from them will not be a good approximation of the true model. On the other hand, different draws of the training set lead to nearly the same model, as indicated by their tight arrangement. Figure taken from [6].

limited (high variance). Models, such as the $k = 15$ nearest neighbor classifier (see Figure 1.6), that balance reducing bias with increasing variance are the goal in predictive machine learning.

Let us now derive the bias variance tradeoff in a general statistical setting using the squared loss. For this we will assume there exists a deterministic function $F : \mathbb{R}^d \rightarrow \mathbb{R}$ that determines the uncorrupted label of $x \in \mathbb{R}^d$, and that the labels we observe are corrupted by white noise:

$$y_i = F(x_i) + \varepsilon_i.$$

The variables ε_i independently and identically distributed normal random variables with mean zero and variance σ^2 , i.e.,

$$\varepsilon_i \sim \mathcal{N}(0, \sigma^2),$$

which implies, in particular, that $\mathbb{E}[\varepsilon_i] = 0$ and $\mathbb{E}[\varepsilon_i^2] = \sigma^2$.

Given a training set T drawn from $p_{X,Y}(x, y)$ and a parameterized model class $\mathcal{F} = \{f(\cdot; \theta) : \theta \in \mathbb{R}^n\}$, we obtain a model by minimizing the squared loss (empirical risk) over the training set:

$$\hat{\theta}_T = \arg \min_{\theta \in \mathbb{R}^n} R_{\text{emp}}(f(\cdot; \theta)) = \arg \min_{\theta \in \mathbb{R}^n} \frac{1}{N} \sum_{i=1}^N (y_i - f(x_i; \theta))^2,$$

where the subscript T emphasizes that $\hat{\theta}_T$ depends upon the particular training set used to solve for the model. Given this model, we define the *conditional test*

error, which is expected test error given the draw of the training set T :

$$\text{err}(\mathcal{F}, T) = \mathbb{E}_{X,Y}[(Y - f(X; \hat{\theta}_T))^2].$$

Note that $\text{err}(\mathcal{F}, T)$ is often a quantity we are very interested in, as it encodes the ability of our selected model $f(\cdot; \hat{\theta}_T)$, obtained through our specific training set T , to generalize to new points. A related quantity is the *expected test error*, which is defined as:

$$\text{Err}(\mathcal{F}) = \mathbb{E}_T[\text{err}(\mathcal{F}, T)].$$

The expectation \mathbb{E}_T is an expectation over all possible training sets T with N elements, drawn according to $p_{X,Y}(x, y)$. Note that the expected test error, $\text{Err}(\mathcal{F})$, measures the quality of the model class \mathcal{F} and not any particular model $f(x; \hat{\theta}_T)$ selected from it using a specific training set T .

Figure 1.10 shows that minimizing the empirical risk by increasing model complexity is not a good way to minimize the conditional test error or the expected test error. Indeed, the empirical risk decreases with model complexity assuming that increasing complexity allows us to obtain increasingly better approximations of F , or to even include F at some point. However, such complex models, particularly when the model is more complex than F , generalize poorly as they fit spurious patterns generated by the additive noise ε_i , which is reflected in the increase of the conditional and expected test errors at a certain point.

Let us now derive the bias variance tradeoff, which will give a quantitative interpretation for the empirical results we have observed. The result will decompose the expected test error at an arbitrary, but fixed point $x \in \mathcal{X}$. We thus define the *expected test error at x* ⁶ as:

$$\text{Err}(\mathcal{F}, x) = \mathbb{E}_T \mathbb{E}_{Y|X}[(Y - f(X; \hat{\theta}_T))^2 \mid X = x].$$

Theorem 1.6 (Bias variance tradeoff). *Let $T = \{(x_i, y_i)\} \subset \mathbb{R}^d \times \mathbb{R}$ be an arbitrary training set drawn from a joint distribution $p_{X,Y}(x, y)$ with $y_i = F(x_i) + \varepsilon_i$ for some deterministic function $F : \mathbb{R}^d \rightarrow \mathbb{R}$ and iid random variables $\varepsilon_i \sim \mathcal{N}(0, \sigma^2)$. Then the expected test error at $x \in \mathcal{X}$ can be decomposed as:*

$$\text{Err}(\mathcal{F}, x) = \sigma^2 + (F(x) - \mathbb{E}_T[f(x; \hat{\theta}_T)])^2 + \mathbb{E}_T[(f(x; \hat{\theta}_T) - \mathbb{E}_T[f(x; \hat{\theta}_T)])^2].$$

Theorem 1.6 decomposes the test error at an arbitrary point $x \in \mathcal{X}$ into three components:

- The irreducible noise error: σ^2
- The bias induced by model class: $F(x) - \mathbb{E}_T[f(x; \hat{\theta}_T)]$
- The variance of the selected model from the model class: $\mathbb{E}_T[(f(x; \hat{\theta}_T) - \mathbb{E}_T[f(x; \hat{\theta}_T)])^2]$

⁶Thanks to Ali Zare for pointing out some ambiguity in the original definition.

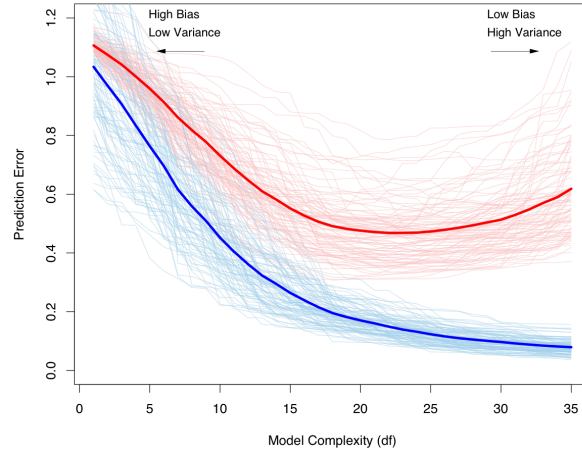


Figure 1.10: Behavior of test error and training error as the model class complexity is increased. The plot depicts 100 different training sets drawn from the same distribution. The light blue curves show the empirical risk $R_{emp}(f(\cdot; \hat{\theta}_T))$ for each of the 100 training sets, T . The light red curves show the corresponding conditional test error, $\text{err}_T(\mathcal{F}, f(\cdot; \hat{\theta}_T))$ for each of the training set. The dark blue curve is the expected empirical risk over all draws of the training set, $\mathbb{E}_T[R_{emp}(f(\cdot; \hat{\theta}_T))]$, estimated by averaging the light blue curves. The dark red curve is the expected test error $\text{Err}(\mathcal{F})$, estimated by averaging the light red curves. Figure taken from [7].

The irreducible noise error is an error that is incurred regardless of the model class used to fit the training data; it is a fundamental limitation of any model learned from training data generated by $p_{X,Y}(x, y)$. The bias measures the capacity of the model class to contain a model that fits the underlying function $F(x)$ that generates the labels. The bias error will decrease (or at least stay flat) as the complexity of the model class increases. The variance measures the stability of the selected model, $f(x; \hat{\theta}_T)$, over different draws of the training set T . If the variance is low, then the model class is stable and different draws of the training set (theoretical or not) will not influence the learned model too much. On the other hand, if the variance is high, then the model is unstable and a new training set may result in a very different model. Since the underlying, noiseless label $F(x)$ is deterministic, this may result in poor predictions because it is likely to get “unlucky” with a high variance model class. Figure 1.9 illustrates these ideas. Let us now prove the theorem.

Proof. We write $Y(x) = F(x) + \varepsilon$ to denote the random variable Y conditioned on $X = x$ for some $x \in \mathcal{X}$. Our first goal is to extract the irreducible noise error

through the following calculation:

$$\begin{aligned}
& \mathbb{E}_{Y|X} \mathbb{E}_T[(Y - f(X; \hat{\theta}_T))^2 | X = x] \\
&= \mathbb{E}_{Y|X} \mathbb{E}_T[(Y(x) - f(x; \hat{\theta}_T))^2] \\
&= \mathbb{E}_{Y|X} \mathbb{E}_T[(Y(x) - F(x) + F(x) - f(x; \hat{\theta}_T))^2] \\
&= \mathbb{E}_{Y|X} \mathbb{E}_T[(Y(x) - F(x))^2] \\
&\quad + \mathbb{E}_{Y|X} \mathbb{E}_T[(F(x) - f(x; \hat{\theta}_T))^2] \\
&\quad + 2\mathbb{E}_{Y|X} \mathbb{E}_T[(Y(x) - F(x))(F(x) - f(x; \hat{\theta}_T))] \\
&= \mathbb{E}_\varepsilon[\varepsilon^2] + \mathbb{E}_T[(F(x) - f(x; \hat{\theta}_T))^2] + 2\mathbb{E}_\varepsilon \mathbb{E}_T[\varepsilon(F(x) - f(x; \hat{\theta}_T))] \\
&= \sigma^2 + \mathbb{E}_T[(F(x) - f(x; \hat{\theta}_T))^2] + 2\mathbb{E}_\varepsilon[\varepsilon] \mathbb{E}_T[F(x) - f(x; \hat{\theta}_T)] \\
&= \sigma^2 + \mathbb{E}_T[(F(x) - f(x; \hat{\theta}_T))^2]
\end{aligned}$$

where we used $\mathbb{E}[\varepsilon] = 0$, $\mathbb{E}[\varepsilon^2] = \sigma^2$, and the fact that the noise ε on the label of a test point x is independent of the training set T .

Now we further decompose the second term into the bias term and the variance term:

$$\begin{aligned}
& \mathbb{E}_T[(F(x) - f(x; \hat{\theta}_T))^2] \\
&= \mathbb{E}_T[(F(x) - \mathbb{E}_T[f(x; \hat{\theta}_T)] + \mathbb{E}_T[f(x; \hat{\theta}_T)] - f(x; \hat{\theta}_T))^2] \\
&= \mathbb{E}_T[(F(x) - \mathbb{E}_T[f(x; \hat{\theta}_T)])^2] + \mathbb{E}_T[(f(x; \hat{\theta}_T) - \mathbb{E}_T[f(x; \hat{\theta}_T)])^2] \\
&\quad + \mathbb{E}_T[(F(x) - \mathbb{E}_T[f(x; \hat{\theta}_T)])(f(x; \hat{\theta}_T) - \mathbb{E}_T[f(x; \hat{\theta}_T)])] \\
&= (F(x) - \mathbb{E}_T[f(x; \hat{\theta}_T)])^2 + \mathbb{E}_T[(f(x; \hat{\theta}_T) - \mathbb{E}_T[f(x; \hat{\theta}_T)])^2] \\
&\quad + (F(x) - \mathbb{E}_T[f(x; \hat{\theta}_T)])\mathbb{E}_T[f(x; \hat{\theta}_T) - \mathbb{E}_T[f(x; \hat{\theta}_T)]] \\
&= (F(x) - \mathbb{E}_T[f(x; \hat{\theta}_T)])^2 + \mathbb{E}_T[(f(x; \hat{\theta}_T) - \mathbb{E}_T[f(x; \hat{\theta}_T)])^2]
\end{aligned}$$

The proof is thus completed. \square

Example 1.7. ⁷ Let us now apply Theorem 1.6 to the k -nearest neighbors algorithm and the linear model. Starting with k -nearest neighbors, let us assume for simplicity that the training data points $\{x_i\}_{i=1}^N$ are fixed, but the labels $y_i = F(x_i) + \varepsilon_i$ are still random due to the random noise ε_i . In order to emphasize this assumption, we will replace \mathbb{E}_T with \mathbb{E}_ε , to emphasize the expectation is just with respect to the noise and not the draws of $\{x_i\}_{i=1}^N$. In this case Theorem 1.6 can be written as:

$$\text{Err}(k, x) = \sigma^2 + (F(x) - \mathbb{E}_\varepsilon[f(x; k)])^2 + \mathbb{E}_\varepsilon[(f(x; k) - \mathbb{E}_\varepsilon[f(x; k)])^2]. \quad (1.13)$$

⁷Thanks to Ali Zare and Xitong Zhang for helpful discussions on this example, which clarified its presentation.

Let us first compute $\mathbb{E}_\varepsilon[f(k, x)]$. We obtain

$$\begin{aligned}\mathbb{E}_\varepsilon[f(x; k)] &= \mathbb{E}_\varepsilon \left[\frac{1}{k} \sum_{x_i \in N_k(x)} y_i \right] = \mathbb{E}_\varepsilon \left[\frac{1}{k} \sum_{x_i \in N_k(x)} (F(x_i) + \varepsilon_i) \right] \\ &= \frac{1}{k} \sum_{x_i \in N_k(x)} (F(x_i) + \mathbb{E}_\varepsilon[\varepsilon_i]) \\ &= \frac{1}{k} \sum_{x_i \in N_k(x)} F(x_i),\end{aligned}$$

since $\mathbb{E}_\varepsilon[\varepsilon_i] = 0$. Now going back to (1.13) let us use this calculation to further simplify the variance term. We have:

$$\begin{aligned}\mathbb{E}_\varepsilon \left[(f(x; k) - \mathbb{E}_\varepsilon[f(x; k)])^2 \right] &= \mathbb{E}_\varepsilon \left[\left(\frac{1}{k} \sum_{x_i \in N_k(x)} (F(x_i) + \varepsilon_i) - \frac{1}{k} \sum_{x_i \in N_k(x)} F(x_i) \right)^2 \right] \\ &= \mathbb{E}_\varepsilon \left[\left(\frac{1}{k} \sum_{x_i \in N_k(x)} \varepsilon_i \right)^2 \right] \\ &= \mathbb{E}_\varepsilon \left[\frac{1}{k^2} \sum_{i,j=1}^k \varepsilon_i \varepsilon_j \right] \\ &= \frac{1}{k^2} \sum_{i,j=1}^k \mathbb{E}_\varepsilon[\varepsilon_i \varepsilon_j] = \frac{\sigma^2}{k},\end{aligned}$$

where we used $\mathbb{E}_\varepsilon[\varepsilon_i \varepsilon_j] = \sigma^2 \delta(i - j)$. Putting everything together in (1.13) we have

$$\text{Err}(k, x) = \sigma^2 + \left(F(x) - \frac{1}{k} \sum_{x_i \in N_k(x)} F(x_i) \right)^2 + \frac{\sigma^2}{k}.$$

We see that the variance, σ^2/k , decreases as the number of neighbors k increases (recall that larger k means a less complex model). On the other hand, larger k means that $F(x)$ is estimated from a larger neighborhood around x , potentially increasing the bias, especially for irregular functions F .

Example 1.8. For linear models $f(x; \theta) = \langle x, \theta \rangle$, we obtain the following for the expected training error, again assuming the $\{x_i\}_{i=1}^N$ are fixed and it is only the labels $y_i = F(x_i) + \varepsilon_i$ that are random:

$$\frac{1}{N} \sum_{i=1}^N \text{Err}(\mathcal{F}_{\text{linear}}, x_i) = \sigma^2 + \frac{1}{N} \sum_{i=1}^N (F(x_i) - \mathbb{E}_\varepsilon[f(x_i; \hat{\theta}_T)])^2 + \frac{d}{N} \sigma^2.$$

Here the model complexity increases with the dimension d , but the variance term only increases linearly in d . The bias term will depend on whether $F(x)$ is

well approximated by a linear model. We will prove a similar result to this in the next section when we discuss the curse of dimensionality.

1.4.3 Curse of dimensionality

We have now examined two models closely: linear models and k -nearest neighbors. In the previous section, Section 1.4.2, we observed that linear models complexity scales with the dimension d of our data, but the variance only scales linearly in d . Thus linear models are stable. On the other hand, if the mapping $x \mapsto y$ is not linear, a machine learned linear model will not be able to capture the relationship between data point x and label y , and there will be a potentially large bias in the model.

For k -nearest neighbors the situation is a bit more subtle. Figures 1.6 and 1.7 would seem to indicate that it is superior to the linear model and in general a good choice. Indeed, without knowing the underlying data generation process, which was highly nonlinear, k -nearest neighbors resulted in a model that was nearly as good as the naive Bayes model, which is the best one can do under the circumstances of our framework. The main issue from the analysis of Section 1.4.2 is that in order to reduce the variance of the k -nearest neighbor model we must select a reasonably large k (certainly not $k = 1$, see Figure 1.5). However, increasing k may increase the bias, as we select more and more points that are further away from the point whose label we are trying to predict. It would seem, though, that if we have a large amount of data, this issue is removed as we can select a large k without having to incorporate points that are far away from the central point x into its neighborhood $N_k(x)$. This intuition works in low dimensions but breaks down in high dimensions. While there are many manifestations of this problem, they are all generally referred to as the *curse of dimensionality*. In what follows we give a few examples.

Example 1.9. Here is one manifestation. Consider a d -dimensional unit cube, $\mathcal{X} = Q \subset \mathbb{R}^d$,

$$Q = [0, 1]^d = \underbrace{[0, 1] \times \cdots \times [0, 1]}_{d \text{ times}}$$

Suppose our test point x is the corner, $x = (0, \dots, 0)$, and that our training set is distributed uniformly in Q , meaning that $p_X(x) = 1$ for all $x \in Q$. A related method to k -nearest neighbors is to simply take all points within a geometric neighborhood of x and average their labels to obtain an estimate for the label of x . For example, we could take a sub-cube $Q_x \subseteq Q$, and average all the labels y_i of training points $x_i \in Q_x$ to obtain an estimate for the label of x :

$$f(x; \theta) = \text{Avg}\{y_i : x_i \in Q_x\}. \quad (1.14)$$

The parameters θ of this model have to do with how we construct the cube Q_x . Let us suppose we want to capture a fraction r of the volume of Q , which is one (i.e., $\text{vol}(Q) = 1$). How long must the edge of Q_x be? Let's call this edge length $e_d(r)$, since it depends on the fraction of the volume r we want to capture and

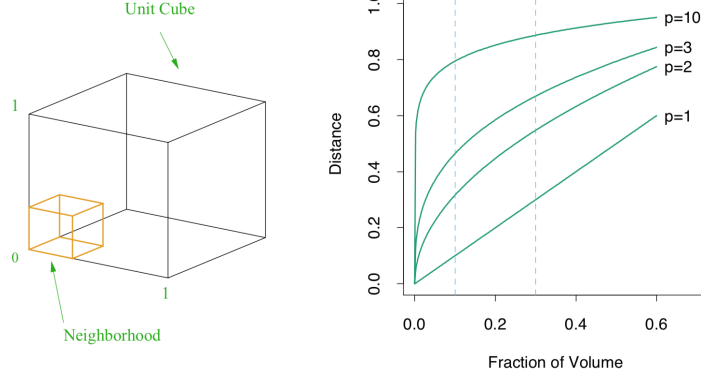


Figure 1.11: One illustration of the curse of dimensionality. Left: A neighborhood cube Q_x (orange) as a subset of the unit cube Q . Right: The horizontal axis is the fraction of volume r that the neighborhood cube contains. The vertical axis is the required side length $e_d(r)$ of the neighborhood cube Q_x . Curves plotting $e_d(r)$ for $d = 1, 2, 3, 10$ are shown (note: in the figure, $p = d$). Figure taken from [7].

the dimension d of the data space. In one dimension, it is clear that $e_1(r) = r$. However, in d -dimensions, we have

$$e_d(r) = r^{1/d}.$$

This is decidedly less favorable. Indeed, as an example, in 10 dimensions we have:

$$e_{10}(0.01) = 0.63 \quad \text{and} \quad e_{10}(0.1) = 0.80,$$

meaning that to capture just 1% of the volume in 10 dimensions we need an edge length of 0.63, and to capture 10% of the volume in 10 dimensions, we need an edge length of 0.80. Remember the side length of the cube Q is just 1.0! Therefore, what we thought was a local neighborhood due to our intuition about how things work in low dimensions, turns out, in fact, to be a very large neighborhood in high dimensions. Figure 1.11 illustrates this principle.

Example 1.10. Suppose instead we consider spherical neighborhoods, as is often the case. We consider a model similar to (1.14) but instead average the labels within a sphere of radius r centered at x :

$$f(x; r) = \text{Avg}\{y_i : \|x - x_i\|_2 \leq r\}. \quad (1.15)$$

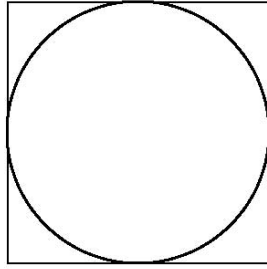
Let us again suppose that our data space is $\mathcal{X} = Q = [0, 1]^d$, the unit cube. Suppose further that $x = (1/2, \dots, 1/2)$, the center point of the cube, and we take the largest radius r possible, which is $r = 1/2$. If the data points are sampled uniformly from Q , then the odds of no training points being within $1/2$ of x are

$$\mathbb{P}(x_i \notin B, \forall 1 \leq i \leq N) = \left(1 - \frac{\text{vol}(B)}{\text{vol}(Q)}\right)^N = (1 - \text{vol}(B))^N,$$

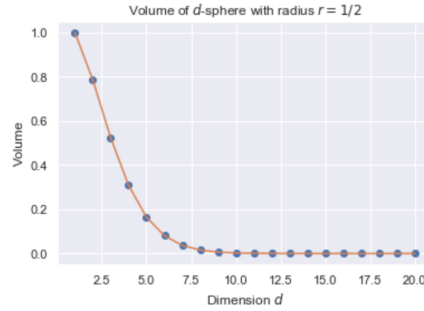
where B is the ball of radius $1/2$ centered at $x = (1/2, \dots, 1/2)$ and we used the fact that $\text{vol}(Q) = 1$. The volume of a ball of radius r , B_r^d , in d -dimensions, is

$$\text{vol}(B_r^d) = \frac{\pi^{d/2}}{\Gamma(d/2 + 1)} r^d.$$

In two dimensions, $\text{vol}(B) \approx 0.79$, and so with $N = 100$ training points sampled from Q , the odds that no training points are in B is approximately $(0.21)^{100} \approx 0$; in other words, we are nearly guaranteed to have some training points close to x . Indeed, Figure 1.12a plots a circle inscribed in a square in two dimensions, and we see that the area of the circle dwarfs the area of the corners inside the square, but outside the circle.



(a) A circle inscribed in a square.



(b) Volume of the d -dimensional ball of radius $1/2$ as a function of the dimension d .

Figure 1.12: Another illustration of the curse of dimensionality. Left: In low dimensions, it is likely that a test point x centered in a box will have training points sampled within a small radius of the point x . Right: In high dimensions, however, the volume of the d -sphere of radius $r = 1/2$ inscribed in the unit cube is dwarfed by the cube's unit volume, indicating that almost certainly the training points will be situated in the “corners” of the cube and thus far from the test point x centered in the middle of the cube.

On the other hand, in high dimensions the volume of the ball B rapidly decreases while the volume of the cube Q remains fixed at one; see Figure 1.12b. It thus follows that it is very unlikely for a training point to lie within the ball of radius $1/2$ centered at the test point x , making the model (1.15) useless. Indeed, if $d = 10$ and $N = 100$, then the odds of no training points lying within B are over 90%. When $d = 20$, it is essentially guaranteed!

Example 1.11. Instead of sampling points from the cube Q , let us restrict to the ball B . Surely in this case the situation must be more favorable? In fact the answer is still no. Let us assume now that B has a radius of $r = 1$ and is centered at the origin. We consider the test point $x = (0, \dots, 0)$ at the origin, and sample training points $\{x_1, \dots, x_N\}$ from $\mathcal{X} = B$ according to the uniform distribution over B (so again p_X is constant). We are interested in the 1-nearest neighbor distance from x . One can show that over all possible draws of the

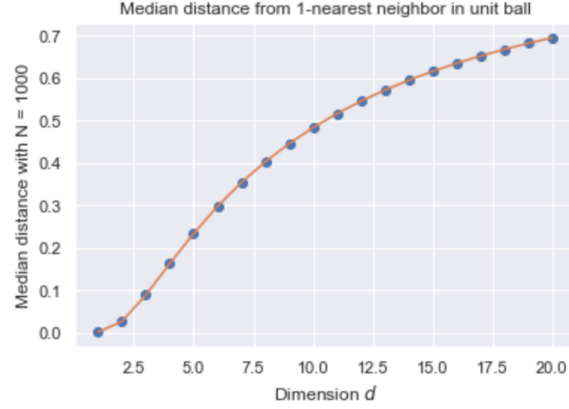


Figure 1.13: The median distance of the 1-nearest neighbor to the origin in the unit ball as a function of dimension d , assuming the training points are sampled from the uniform distribution. As the dimension d increases, even the 1-nearest neighbor becomes very far from the test point at the origin.

training set, the median distance from the origin x to closest training point is:

$$\text{dist}_{1\text{-NN}}(d, N) = \left(1 - (1/2)^{1/N}\right)^{1/d}.$$

As with the other examples, in low dimensions we are fine. Indeed, for $d = 2$ and $N = 1000$, we have $\text{dist}_{1\text{-NN}}(2, 1000) \approx 0.03$. On the other hand, if we go to $d = 20$ dimensions, the situation becomes far worse, as $\text{dist}_{1\text{-NN}}(20, 1000) \approx 0.70$! Keep in mind, the distance to the boundary is one. Figure 1.13 plots $\text{dist}_{1\text{-NN}}(d, 1000)$ for $1 \leq d \leq 20$. Since this is the 1-nearest neighbor distance, it means that all k -nearest neighbors will be far from x in high dimensions, thus leading to poor models.

Example 1.12. In this example we can see the ramifications of the analysis contained in the previous examples. Suppose that $\mathcal{X} = [-1, 1]^d$, the cube of side-length two centered at the origin. Suppose additionally that labels y are derived from x according to:

$$y = F(x) = e^{-8\|x\|_2^2},$$

with no measurement error. We are going to use a 1-nearest neighbor model to estimate the label y of new test points. Let us consider a test point $x = (0, \dots, 0)$ at the origin, which has label $f((0, \dots, 0)) = 1$. Draw a training set $T = \{(x_i, y_i)\}_{i=1}^N$ and let $x_0 \in T$ be the point closest to x , and let $y_0 = F(x_0) = e^{-8\|x_0\|_2^2}$, which is the estimate for the label of x . Then using the bias-variance decomposition (Theorem 1.6), the expected test error at x is:

$$\text{Err}(1\text{-NN}, x) = \underbrace{(1 - \mathbb{E}_T[y_0])^2}_{\text{bias}} + \underbrace{\mathbb{E}_T[(y_0 - \mathbb{E}_T[y_0])^2]}_{\text{variance}}.$$

Suppose that the number of training points is $N = 1000$. In this case, the model will be biased because we know that $y_0 \leq 1$ no matter what. And indeed, the bias error will dominate, and grow large as the dimension d increases, since like in Example 1.11 and Figure 1.13 the median (and mean) distance of the 1-nearest neighbor from the origin will grow with the dimension. By dimension $d = 10$, more than 99% of the training samples will be, on average, at a distance greater than 0.5 from the origin, leading to a severe underestimate of the label since the $F(x) = e^{-8\|x\|_2^2}$ decays very rapidly; see Figure 1.14. Note that the bias does not always dominate. For example, if $F(x)$ only depends on a few dimensions of the data, e.g., $F(x) = (1/2) \cdot (x(1) + 1)^3$, then the variance error will dominate and will grow rapidly with the dimension; see Figure 1.15.

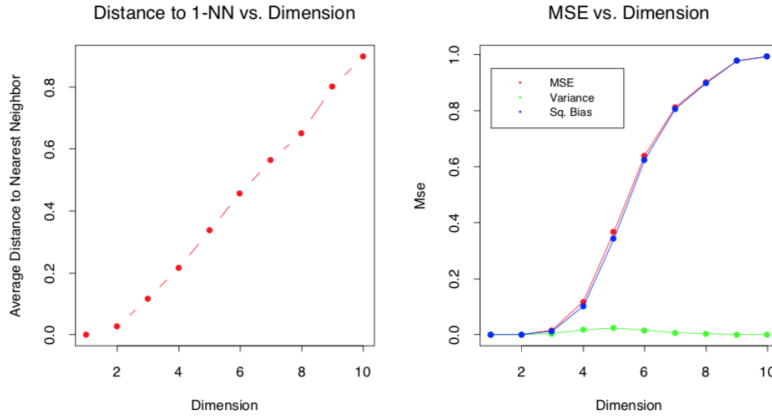


Figure 1.14: Plots illustrating Example 1.12. Left: The average distance of the 1-nearest neighbor to the origin over many draws of the training set with $N = 1000$ training points, as a function of the dimension d . Right: The expected test error at the origin for the label function $y = e^{-8\|x\|_2^2}$ (MSE), as a function of the dimension, and broken down into its bias squared component and its variance component. Figure taken from [7].

Another way of thinking about the curse of dimensionality is to consider how many training points N would be required to avoid it. Indeed, suppose in one dimension we require $N = 100$ training points to densely sample the space \mathcal{X} , e.g., $\mathcal{X} = [0, 1]$. Then to have the same sampling density for $\mathcal{X} = Q = [0, 1]^d$, we would require 100^d training points!

Now, in practice, we are very rarely confronted with a supervised learning problem in a high dimensional cube Q or ball B with a uniform sampling density. Indeed, consider the MNIST data base, consisting of 28×28 gray-scale images, where each pixel takes a value between 0 and 255. Suppose these gray-scale values are divided by 255 so they are normalized to lie in the interval $[0, 1]$. Then $\mathcal{X} = [0, 1]^{784}$, which is very high dimensional! But, the sampling density $p_X(x)$ is not uniform. On the contrary, looking back at Figure 1.1, one

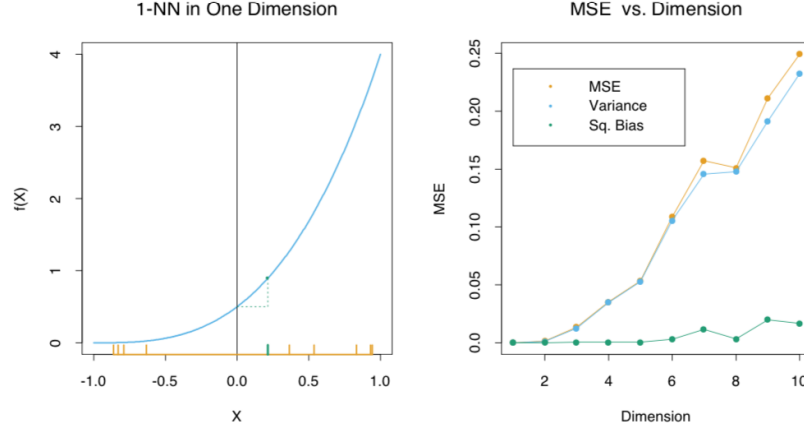


Figure 1.15: Additional plots illustrating Example 1.12. In this figure the label function is $y = (1/2) \cdot (x(1) + 1)^3$, which only depends on the first dimension of the data point x . The variance rapidly increases with the dimension, though, leading to another manifestation of the curse of dimensionality. Figure taken from [7].

sees that the MNIST digit images are highly structured. This implies that $p_X(x)$ is (essentially) supported on a much lower dimensional set contained within Q . This is what makes learning possible, but the challenge is that we must take advantage of this fact without being able to precisely know what the supporting set of $p_X(x)$ is.

Another advantage one may have is prior knowledge on the labeling function $y = F(x) + \varepsilon$. Indeed, if $F(x) = \langle x, \theta_0 \rangle$ is linear, and we know this fact but we do not know the parameters θ_0 , we can still leverage this knowledge to restrict our model class to the class of linear models. Suppose this is the case, and that the labels are corrupted versions of $F(x)$, i.e.,

$$y = F(x) + \varepsilon = \langle x, \theta_0 \rangle + \varepsilon, \quad x \in \mathbb{R}^d, \quad \varepsilon \sim \mathcal{N}(0, \sigma^2).$$

Suppose we take our model class to be all possible linear models

$$\mathcal{F} = \{f(x; \theta) = \langle x, \theta \rangle : \theta \in \mathbb{R}^d\}.$$

Given a training set $T = \{(x_i, y_i)\}_{i=1}^N$ we obtain a model from \mathcal{F} by minimizing the squared loss:

$$\hat{\theta}_T = \arg \min_{\theta \in \mathbb{R}^d} \frac{1}{N} \sum_{i=1}^N (y_i - \langle x_i, \theta \rangle)^2.$$

From equation (1.5) we know that

$$\hat{\theta}_T = (\mathbf{X}\mathbf{X}^T)^{-1}\mathbf{X}\mathbf{y},$$

where \mathbf{X} is the $d \times N$ matrix containing the training point x_i on the i^{th} column, and \mathbf{y} is the $N \times 1$ vector containing y_i in the i^{th} entry.

Also let ε be the $N \times 1$ vector with ε_i as its i^{th} entry. Now let $x \in \mathbb{R}^d$ be a test point, which we consider as a $d \times 1$ vector to be consistent with \mathbf{X} . Our prediction for its label is $f(x; \hat{\theta}_T)$, which we can write as:

$$\begin{aligned}
f(x; \hat{\theta}_T) &= \langle x, \hat{\theta}_T \rangle \\
&= \langle x, (\mathbf{X}\mathbf{X}^T)^{-1}\mathbf{X}\mathbf{y} \rangle \\
&= \langle x, (\mathbf{X}\mathbf{X}^T)^{-1}\mathbf{X}(\mathbf{X}^T\theta_0 + \varepsilon) \rangle \\
&= \langle x, \theta_0 + (\mathbf{X}\mathbf{X}^T)^{-1}\mathbf{X}\varepsilon \rangle \\
&= \langle x, \theta_0 \rangle + \langle x, (\mathbf{X}\mathbf{X}^T)^{-1}\mathbf{X}\varepsilon \rangle \\
&= F(x) + \langle \mathbf{X}^T(\mathbf{X}\mathbf{X}^T)^{-1}x, \varepsilon \rangle \\
&= F(x) + \sum_{i=1}^N (\mathbf{X}^T(\mathbf{X}\mathbf{X}^T)^{-1}x)(i) \varepsilon_i.
\end{aligned}$$

From this calculation and the fact that \mathbf{X} is independent of ε_i , we conclude that

$$\mathbb{E}_T[f(x; \hat{\theta}_T)] = F(x) + \sum_{i=1}^N \mathbb{E}_T[(\mathbf{X}^T(\mathbf{X}\mathbf{X}^T)^{-1}x)(i)] \mathbb{E}_T[\varepsilon_i] = F(x),$$

showing that the least squares fit is unbiased estimator for linear models since from Theorem 1.6 we have:

$$\text{bias} = F(x) - \mathbb{E}_T[f(x; \hat{\theta}_T)].$$

Therefore, if we apply Theorem 1.6 (bias-variance trade-off), we will have only the irreducible error σ^2 and the variance error,

$$\begin{aligned}
\text{Err}(\mathcal{F}, x) &= \sigma^2 + \text{Var}_T \left[\sum_{i=1}^N (\mathbf{X}^T(\mathbf{X}\mathbf{X}^T)^{-1}x)(i) \varepsilon_i \right] \\
&= \sigma^2 + \mathbb{E}_T \left[\left(\sum_{i=1}^N (\mathbf{X}^T(\mathbf{X}\mathbf{X}^T)^{-1}x)(i) \varepsilon_i \right)^2 \right] \\
&= \sigma^2 + \mathbb{E}_T \left[\sum_{i,j=1}^N (\mathbf{X}^T(\mathbf{X}\mathbf{X}^T)^{-1}x)(i) \varepsilon_i (\mathbf{X}^T(\mathbf{X}\mathbf{X}^T)^{-1}x)(j) \varepsilon_j \right] \\
&= \sigma^2 + \sum_{i,j=1}^N \mathbb{E}_T[(\mathbf{X}^T(\mathbf{X}\mathbf{X}^T)^{-1}x)(i) (\mathbf{X}^T(\mathbf{X}\mathbf{X}^T)^{-1}x)(j)] \underbrace{\mathbb{E}_T[\varepsilon_i \varepsilon_j]}_{\sigma^2 \delta(i-j)} \\
&= \sigma^2 + \sigma^2 \sum_{i=1}^N \mathbb{E}_T[(\mathbf{X}^T(\mathbf{X}\mathbf{X}^T)^{-1}x)(i)^2] \\
&= \sigma^2 + \sigma^2 \mathbb{E}_T \left[\|\mathbf{X}^T(\mathbf{X}\mathbf{X}^T)^{-1}x\|_2^2 \right].
\end{aligned}$$

Now, we also have:

$$\begin{aligned}\|\mathbf{X}^T(\mathbf{X}\mathbf{X}^T)^{-1}x\|_2^2 &= (\mathbf{X}^T(\mathbf{X}\mathbf{X}^T)^{-1}x)^T \mathbf{X}^T(\mathbf{X}\mathbf{X}^T)^{-1}x \\ &= x^T(\mathbf{X}\mathbf{X}^T)^{-1}\mathbf{X}\mathbf{X}^T(\mathbf{X}\mathbf{X}^T)^{-1}x = x^T(\mathbf{X}\mathbf{X}^T)^{-1}x\end{aligned}$$

Therefore

$$\mathbb{E}_T \left[\|\mathbf{X}^T(\mathbf{X}\mathbf{X}^T)^{-1}x\|_2^2 \right] = \mathbb{E}_T[x^T(\mathbf{X}\mathbf{X}^T)^{-1}x] = x^T \mathbb{E}_T[(\mathbf{X}\mathbf{X}^T)^{-1}]x.$$

Now, this quantity is a real number. We can write any real number $a \in \mathbb{R}$ as $\text{Trace}[a]$, where we view a as a 1×1 matrix. Recall that $\text{Trace}[A] = \sum_{k=1}^m A(k, k)$ for an $m \times m$ matrix A . We also note that for matrices A, B, C we have

$$\text{Trace}[ABC] = \text{Trace}[BCA]$$

whenever the matrix multiplications make sense. Therefore we have:

$$\begin{aligned}x^T \mathbb{E}_T[(\mathbf{X}\mathbf{X}^T)^{-1}]x &= \text{Trace} \left[x^T \mathbb{E}_T[(\mathbf{X}\mathbf{X}^T)^{-1}]x \right] \\ &= \text{Trace} \left[\mathbb{E}_T[(\mathbf{X}\mathbf{X}^T)^{-1}]xx^T \right].\end{aligned}\tag{1.16}$$

It thus follows that:

$$\text{Err}(\mathcal{F}, x) = \sigma^2 \left(1 + \text{Trace} \left[\mathbb{E}_T[(\mathbf{X}\mathbf{X}^T)^{-1}]xx^T \right] \right)^8.$$

Now let us compute $\text{Err}(\mathcal{F})$, which we write as:

$$\text{Err}(\mathcal{F}) = \mathbb{E}_X [\text{Err}(\mathcal{F}, X)].$$

Therefore we need to compute the expectation with respect to a test point $X = x$ of the quantity (1.16). Since the trace is just a summation and expectation is linear, we can interchange them. Let us also assume that $\mathbb{E}_X[X] = (0, \dots, 0)$ so that $\text{cov}(X) = \mathbb{E}[XX^T]$. We then have

$$\begin{aligned}\mathbb{E}_X \left\{ \text{Trace} \left[\mathbb{E}_T[(\mathbf{X}\mathbf{X}^T)^{-1}]XX^T \right] \right\} &= \text{Trace} \left\{ \mathbb{E}_X \left[\mathbb{E}_T[(\mathbf{X}\mathbf{X}^T)^{-1}]XX^T \right] \right\} \\ &= \text{Trace} \left\{ \mathbb{E}_T[(\mathbf{X}\mathbf{X}^T)^{-1}] \mathbb{E}_X[XX^T] \right\} \\ &= \text{Trace} \left\{ \mathbb{E}_T[(\mathbf{X}\mathbf{X}^T)^{-1}] \text{cov}(X) \right\}.\end{aligned}\tag{1.17}$$

At this point we know that

$$\mathbb{E}_T[\mathbf{X}\mathbf{X}^T] = N \text{cov}(X),$$

where we have the factor N instead of $N - 1$ because we are assuming that $\mathbb{E}_X[X] = (0, \dots, 0)$ and that we know this fact. Now, we would like to assert that

$$\mathbb{E}_T[(\mathbf{X}\mathbf{X}^T)^{-1}] \propto N^{-1} \text{cov}(X)^{-1}.$$

⁸Thanks for Yani Udiani for helping to make this part on the trace clearer.

However, as was pointed out in class, this is not always true⁹. One might try to argue if N is very large, then $\mathbf{X}\mathbf{X}^T \approx N\text{cov}(X)$ with a small variance, but at this time I am not sure if this can be made rigorous. For now, one regime in which we can make things rigorous is the following. Suppose that each training data point $\{x_i\}_{i=1}^N$ is sampled independently from the normal distribution with zero mean and $d \times d$ covariance matrix $\Sigma = \text{cov}(X)$, i.e.,

$$x_i \sim \mathcal{N}(\mathbf{0}, \Sigma),$$

where $\mathbf{0} = (0, \dots, 0)$. In this case we have [8],

$$\mathbb{E}_T[(\mathbf{X}\mathbf{X}^T)^{-1}] = (N - d - 1)^{-1} \Sigma^{-1} = (N - d - 1)^{-1} \text{cov}(X)^{-1}.$$

Therefore, picking up from (1.17), we have:

$$\begin{aligned} \text{Trace} \left\{ \mathbb{E}_T[(\mathbf{X}\mathbf{X}^T)^{-1}] \text{cov}(X) \right\} &= \frac{1}{N - d - 1} \text{Trace} \left\{ \text{cov}(X)^{-1} \text{cov}(X) \right\} \\ &= \frac{1}{N - d - 1} \text{Trace}[\mathbf{I}] \\ &= \frac{d}{N - d - 1}. \end{aligned}$$

To conclude, when we have a linear label model $y = \langle x, \theta_0 \rangle + \varepsilon$, and when our data points $x \in \mathbb{R}^d$ are sampled from the normal distribution with mean $\mathbf{0}$, we have¹⁰:

$$\text{Err}(\mathcal{F}) = \sigma^2 \left(1 + \frac{d}{N - d - 1} \right),$$

where \mathcal{F} is the model class of all possible linear models. We thus see that to have a small expected test error (modulo the irreducible error) the number of training points N must grow essentially linearly in the dimension d , thus circumventing the curse of dimensionality (recall the earlier examples of the cube Q with the uniform distribution and 1-nearest neighbor, in which the number of training points grew as a power of d).

On the other hand, we imposed a very heavy assumption on the data generation process, namely that the labels y be linear functions (plus noise) of the data points x . The k -nearest neighbors algorithm imposes no such restriction and in the limit of infinite training data can approximate nearly any model, but N must grow exponentially fast in the dimension d , which is unrealistic. The field of machine learning has developed a number of algorithms “in between” linear regression/classification and k -nearest neighbors, with the goal of increasing the capacity of the model class while not too drastically increasing the complexity of the search space and thus being restricted by the curse of dimensionality. In Section 1.5 we briefly discuss some of the non-deep learning approaches along these lines.

⁹Thanks to Anna Little and Dylan Molho for pointing out an error in the original calculation, and thanks to Mohit Bansil, Anna Little, Gautam Sreekumar, and Ali Zare for valuable discussions thereafter.

¹⁰If anyone can generalize this calculation further, I would be interested in seeing that!

1.5 Towards deep learning

1.5.1 Dictionaries and kernels

One way to incorporate nonlinearity is through dictionaries and kernels. We give a brief overview here. A *dictionary* takes in a data point $x \in \mathcal{X}$ and maps it into a Hilbert space \mathcal{H} . Often the Hilbert space is usually $\mathcal{H} = \mathbb{R}^m$, so let us assume that is the case here. The resulting representation of x is denoted:

$$\Phi : \mathcal{X} \rightarrow \mathbb{R}^m, \quad x \mapsto \Phi(x) = (\phi_k(x))_{k=1}^m, \quad \phi_k : \mathcal{X} \rightarrow \mathbb{R}.$$

Now with the representation $\Phi(x)$, we can apply linear regression but to the representation:

$$f(x; \theta) = \langle \Phi(x), \theta \rangle = \sum_{k=1}^m \theta(k) \phi_k(x). \quad (1.18)$$

More generally, any algorithm that only depends upon $\langle x, \theta \rangle$ can be recast by replacing x with $\Phi(x)$. The resulting model class \mathcal{F} or hypothesis class \mathcal{P} thus depends strongly on the choice of dictionary Φ . The constituent elements of $\Phi(x)$, the functions $\phi_k(x)$, are often referred to as “features.” Because of the importance of these features, feature engineering has been and continues to be an important topic in machine learning, although “hand crafted” features are becoming less popular. Nevertheless, if one can come up with a dictionary $\Phi(x)$ for which the label $y(x)$ is a linear function of $\Phi(x)$, then the linear model over the dictionary given by (1.18) will work. Note that the dimension of the problem is now m instead of d , but from Section 1.4.3 we know that linear models circumvent the curse of dimensionality in that the number of training points need only grow linearly with the dimension. In this case that means we need $N = O(m)$ training points. If m is not too much larger than d , then we will be in an favorable situation.

Kernels and deep learning do not specify the features directly, but they do so in different ways. To understand kernel methods, let us define a kernel $K(x, x')$ in terms of a dictionary $\Phi(x)$ as:

$$K(x, x') = \langle \Phi(x), \Phi(x') \rangle.$$

The *kernel* $K(x, x')$ measures the similarity between x and x' through the lens of Φ ; the larger $K(x, x')$, the more similar x and x' . A kernel regression computes:

$$f(x; \alpha) = \sum_{i=1}^N \alpha(i) K(x, x_i), \quad \alpha \in \mathbb{R}^N, \quad (1.19)$$

where we have assumed the kernel is symmetric, meaning $K(x, x') = K(x', x)$. In other words, $f(x; \alpha)$ predicts the label of x by comparing x to each training point x_i through $K(x, x_i)$ and taking a weighted sum of the resulting similarity

measures. Notice that we can rewrite (1.19) as:

$$\begin{aligned}
f(x; \alpha) &= \sum_{i=1}^N \alpha(i) K(x, x_i) = \sum_{i=1}^N \alpha(i) \langle \Phi(x), \Phi(x_i) \rangle \\
&= \sum_{i=1}^N \alpha(i) \sum_{k=1}^m \phi_k(x) \phi_k(x_i) \\
&= \sum_{k=1}^m \left[\sum_{i=1}^N \alpha(i) \phi_k(x_i) \right] \phi_k(x) \\
&= \sum_{k=1}^m \theta(k) \phi_k(x), \tag{1.20}
\end{aligned}$$

where we have set

$$\theta(k) = \sum_{i=1}^N \alpha(i) \phi_k(x_i).$$

Since $\alpha \in \mathbb{R}^N$ and $\theta \in \mathbb{R}^m$ are learned from the training data, we see that (1.18) and (1.19) are equivalent.

Notice, though, that (1.19) can be defined for any kernel K , not just those for which $K(x, x') = \langle \Phi(x), \Phi(x') \rangle$. On the other hand, many algorithms depend on just the inner product $\langle x, x' \rangle$ to measure the similarity between two data points. If we want to replace these inner products with $K(x, x')$ and still have the algorithm behave well, we need to at least know that $K(x, x') = \langle \Phi(x), \Phi(x') \rangle$ for some Φ , but we do not need to know Φ . This is the key observation of *kernel learning* [9]. Kernel learning allows one to implicitly use very complicated representations $\Phi(x)$ via what are often simple kernels $K(x, x')$. For example, the polynomial kernel

$$K(x, x') = (\langle x, x' \rangle + c)^m,$$

implicitly defines the polynomial model class of degree m . Another example is the Gaussian kernel

$$K(x, x') = e^{-\|x - x'\|_2^2 / 2\sigma^2},$$

which implicitly defines a nonlinear infinite dimensional dictionary $\Phi(x)$!

Because of this observation and the usefulness of kernels and the simplicity of incorporating them into machine learning, there is a well developed mathematical theory for determining when a kernel $K(x, x')$ implicitly defines a dictionary $\Phi(x)$ and thus can be written as $K(x, x') = \langle \Phi(x), \Phi(x') \rangle$. The summary of this theory is that K must be a *reproducing kernel* and thus generate a *reproducing kernel Hilbert space (RKHS)*. This is also not easy to verify directly, but it turns out that K is a reproducing kernel if and only if it is *positive semi-definite*, which means that for any N and any $\{x_i\}_{i=1}^N \subset \mathcal{X}$ and any $c \in \mathbb{R}^N$,

$$\sum_{i,j=1}^N c(i)c(j)K(x_i, x_j) \geq 0.$$

For more details we refer the reader to [9]; see also the course CMSE 820 (my old version [here](#) or newer versions by Prof. [Yuying Xie](#)), which has thus far covered this topic every year.

Kernels allow us to implicitly use a nonlinear dictionary $\Phi(x)$ without having to specify $\Phi(x)$, but the dictionary is not learned. The kernel $K(x, x')$ specifies a unique representation $\Phi(x)$, and as the above calculation (1.20) shows the learning aspect specifies the parameters of the model, but not the representation. Deep learning takes an alternate approach, in which the features themselves are parameterized, and these parameters are learned. Let $\theta = (\boldsymbol{\theta}, \mathbf{w}) \in \mathbb{R}^n$ be the parameters of a deep network in which the last (output) layer computes a linear regression. Then, drawing an analogy with (1.18) we can write this network as:

$$f(x; \theta) = \langle \Phi(x; \boldsymbol{\theta}), \mathbf{w} \rangle,$$

where the vector $\boldsymbol{\theta}$ parameterizes the representation $\Phi(x; \boldsymbol{\theta})$ and the vector \mathbf{w} gives the weights of the linear regression over this representation. Thus when fitting the optimal $\hat{\theta}$ to training data, we not only learn how to combine the features via \mathbf{w} , but we learn the features themselves through the parameter vector $\boldsymbol{\theta}$. This will allow for very powerful learning algorithms, but because of the features dependence on the parameters $\boldsymbol{\theta}$ it has made understanding why deep learning works a much more difficult problem than standard dictionary or kernel approaches. Nevertheless, some progress has been made and new results are arriving every month. In the rest of this course we will explore some of these results.

1.5.2 Logistic regression and nonlinearity

Logistic regression is a learning algorithm for categorical classes (i.e., a finite number of classes), that adds a nonlinear function to linear models in order change the output into a probability. Given a new test point $x \in \mathcal{X}$, it does not make a hard decision about which class x belongs to, but rather, for each possible class, gives the probability of x belonging to that class. It is a very simple version of a neural network.

The way logistic regression does this is with the *sigmoid function*, which is defined as:

$$\sigma(z) = \frac{1}{1 + e^{-z}}, \quad z \in \mathbb{R}.$$

Suppose now that $\mathcal{X} = \mathbb{R}^d$ and $\mathcal{Y} = \{0, 1\}$, i.e., our data points are d -dimensional vectors and we are working on a binary classification problem. Logistic regression defines a parameterized hypothesis space \mathcal{P} of conditional probability density functions $p_{Y|X}(y | x, \theta)$, where $\theta \in \mathbb{R}^d$ and

$$\begin{aligned} p(y = 1 | x, \theta) &= \sigma(\langle x, \theta \rangle) = \frac{1}{1 + e^{-\langle x, \theta \rangle}} \\ p(y = 0 | x, \theta) &= 1 - p(y = 1 | x, \theta). \end{aligned}$$

To solve for the parameters θ , we can use maximum likelihood estimation (MLE) from Section 1.1.3, or maximum a posteriori (MAP) from Section 1.3.2. To keep things simple, we explain with MLE. Suppose we are given a training set $T = \{(x_i, y_i)\}_{i=1}^N$ in which the x_i 's are sampled independently from $p_X(x)$. Recall that the MLE model is:

$$\begin{aligned}\hat{\theta} &= \arg \max_{\theta \in \mathbb{R}^d} p_{X,Y}(T \mid \theta) = \arg \max_{\theta \in \mathbb{R}^d} \prod_{i=1}^N p(y_i \mid x_i, \theta) \\ &= \arg \max_{\theta \in \mathbb{R}^d} \prod_{i=1}^N [\sigma(\langle x_i, \theta \rangle)]^{y_i} [1 - \sigma(\langle x_i, \theta \rangle)]^{1-y_i} \\ &= \arg \max_{\theta \in \mathbb{R}^d} \sum_{i=1}^N [y_i \log \sigma(\langle x_i, \theta \rangle) + (1 - y_i) \log(1 - \sigma(\langle x_i, \theta \rangle))] .\end{aligned}$$

Unlike linear regression, $\hat{\theta}$ cannot be solved for analytically and in closed form. However, one can compute the gradient of

$$\mathcal{L}(\theta) = - \sum_{i=1}^N [y_i \log \sigma(\langle x_i, \theta \rangle) + (1 - y_i) \log(1 - \sigma(\langle x_i, \theta \rangle))]$$

analytically, which allows for efficient minimization using tools from optimization (e.g., gradient descent). Indeed,

$$\nabla_{\theta} \mathcal{L}(\theta) = \sum_{i=1}^N [\sigma(\langle x_i, \theta \rangle) - y_i] x_i .$$

Logistic regression can be extended to more than two classes; the resulting algorithm is called *softmax regression*. Suppose $\mathcal{Y} = \{1, \dots, M\}$, i.e., we have M classes. For each class, we learn weights $\theta_m \in \mathbb{R}^d$, $1 \leq m \leq M$. Softmax regression then assigns probabilities as:

$$p(y = m_0 \mid x, \{\theta_m\}_{m=1}^M) = \frac{e^{-\langle x, \theta_{m_0} \rangle}}{\sum_{m=1}^M e^{-\langle x, \theta_m \rangle}} .$$

Softmax regression is often used in the last layer of a neural network trained for classification, but the relevance goes further. Logistic regression and softmax regression show the importance of nonlinearity, and in particular, having a nonlinear function of a linear transformation. Neural networks build upon this by cascading many successive alternations of linear (or affine) functions and nonlinear functions, such as the sigmoid.

Chapter 2

Artificial neural networks

In this section we discuss artificial neural networks, initially discussing what they are and how they are trained generally. Then we focus on their mathematical properties, particularly from the perspective of approximation theory.

2.1 What is an artificial neural network?

Recall the logistic regression classifier from Section 1.5.2:

$$x \mapsto \frac{1}{1 + e^{-\langle x, \theta \rangle}},$$

which consisted of mapping a data point $x \in \mathcal{X} = \mathbb{R}^d$ into \mathbb{R} via $x \mapsto \langle x, \theta \rangle$, and then from \mathbb{R} again into \mathbb{R} via the nonlinear sigmoid function $\sigma(z) = 1/(1 + e^{-z})$ with $z = \langle x, \theta \rangle$. A neural network generalizes the logistic regression function by defining a mathematical *neuron*¹ as

$$\eta(x) = \sigma(\langle x, w \rangle + b), \quad w \in \mathbb{R}^d, b \in \mathbb{R}, \quad (2.1)$$

and where $\sigma : \mathbb{R} \rightarrow \mathbb{R}$ is a nonlinear function, e.g., the sigmoid function, but there are other possibilities, including:

- Perceptron: $\sigma(z) = 0$ if $z \leq 0$ and $\sigma(z) = 1$ if $z > 0$
- Sigmoid: $\sigma(z) = 1/(1 + e^{-z})$
- Hyperbolic tangent: $\sigma(z) = \tanh(z)$
- Rectified linear unit (ReLU): $\sigma(z) = \max(0, z)$
- Leaky ReLU: $\sigma(z) = \max(az, z)$ with $0 \leq a < 1$.

¹Yes I know η does not correspond to “n,” but it still looks like it...

- Smoothed versions of (leaky) ReLU to eliminate the point of non-differentiability at $z = 0$
- Absolute value (or modulus if $z \in \mathbb{C}$): $\sigma(z) = |z|^2$

Figure 2.1 plots some of these nonlinear functions and Figure 2.2 depicts the a single mathematical neuron.

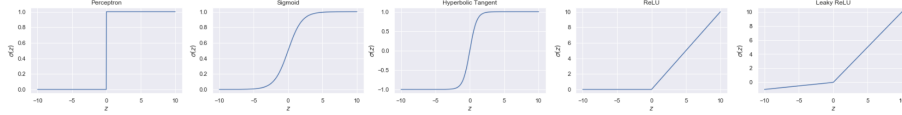


Figure 2.1: Examples of nonlinear activation functions $\sigma(z)$. From left to right: perceptron, sigmoid, hyperbolic tangent, rectified linear unit (ReLU), leaky ReLU with $a = 0.10$.

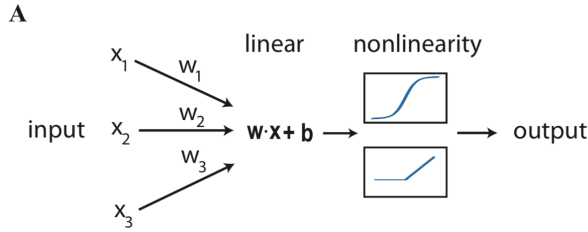


Figure 2.2: An illustration of a single mathematical neuron, defined in equation (2.1). Here the notation is a bit different than the text, as $x = (x_1, x_2, x_3) \in \mathbb{R}^3$, not three training points, and $w = (w_1, w_2, w_3) \in \mathbb{R}^3$. Figure taken from [6].

A mathematical neuron takes in a vector $x \in \mathbb{R}^d$ and processes it via an affine function into \mathbb{R} followed by a nonlinear “activation” function, that responds based on the output of the affine function. A single layer of a neural network has many such neurons. For example, suppose we have d_1 such neurons,

$$\eta_k(z) = \sigma(\langle x, w_k \rangle + b(k)), \quad w_k \in \mathbb{R}^d, b(k) \in \mathbb{R}, 1 \leq k \leq d_1.$$

We collect the affine parts of these d_1 neurons into a single affine map $A : \mathbb{R}^d \rightarrow \mathbb{R}^{d_1}$,

$$A(x) = (\langle x, w_1 \rangle + b(1), \dots, \langle x, w_{d_1} \rangle + b(d_1))^T \in \mathbb{R}^{d_1 \times 1}.$$

If we collect the weights w_k into a single $d \times d_1$ matrix \mathbf{W} and the biases $b(k)$ into a single $d_1 \times 1$ vector b ,

$$\mathbf{W} = \begin{pmatrix} | & & | \\ w_1 & \cdots & w_{d_1} \\ | & & | \end{pmatrix} \quad b = \begin{pmatrix} b(1) \\ \vdots \\ b(d_1) \end{pmatrix},$$

²Thanks to Anna Little for reminding me to put this one.

then we can rewrite $A(x)$ as:

$$A(x) = \mathbf{W}^T x + b.$$

We can also write, with a slight abuse of notation, the collection of all d_1 neurons as the map:

$$x \mapsto \sigma(A(x)) = \sigma(\mathbf{W}^T x + b) = (\eta_1(x), \dots, \eta_{d_1}(x))^T,$$

where it is understood that σ acts element-wise.

An *artificial neural network* cascades the operations $\sigma(A(x))$ multiple times. Suppose we have L layers and the affine transform at each layer is denoted by A_ℓ , $1 \leq \ell \leq L$, in which $A_1 : \mathbb{R}^d \rightarrow \mathbb{R}^{d_1}$ and $A_\ell : \mathbb{R}^{d_{\ell-1}} \rightarrow \mathbb{R}^{d_\ell}$ for $2 \leq \ell \leq L$. An artificial neural network is a representation $\Phi(x; \theta) = \Phi_{\text{ANN}}(x; \theta)$ given by:

$$\Phi(x; \theta) = \sigma_L(A_L(\dots \sigma_1(A_1(x)))) ,$$

where the representation has parameters θ that are encoding the weight matrices $\mathbf{W}_1, \dots, \mathbf{W}_L$ and the bias vectors b_1, \dots, b_L . There is often a final layer that takes the vector $\Phi(x; \theta) \in \mathbb{R}^{d_L}$ and maps it to a scalar for regression or classification. In the regression setting, this means we have a final weight vector $\alpha \in \mathbb{R}^{d_L \times 1}$ such that

$$f(x; \theta, \alpha) = \langle \Phi(x; \theta), \alpha \rangle. \quad (2.2)$$

Similarly, for the logistic regression (can also be generalized to softmax regression), we have:

$$p(y \mid x, \theta, \alpha) = \frac{1}{1 + e^{-\langle \Phi(x; \theta), \alpha \rangle}}.$$

The input to the network, $x \in \mathbb{R}^d$, is often called the *input layer*, the output $f(x; \theta, \alpha)$ or $p(y \mid x, \theta, \alpha)$ the *output layer*, and the layers in between the *hidden layers*. Figure 2.3 depicts an artificial neural network, and here is a two layer network for regression expanded out:

$$f(x; \theta, \alpha) = \alpha^T \sigma_2(\mathbf{W}_2^T \sigma_1(\mathbf{W}_1^T x + b_1) + b_2).$$

The compositional structure of neural networks is thought to be one of the keys to their success. Indeed, unlike linear models that have no composition and logistic regression which only has one affine transform and one nonlinear sigmoid, neural networks can have L such layers of different widths and with different nonlinear functions at each layer. They thus have significantly more flexibility, particularly for larger L . We will explore this in more detail in later sections.

2.2 Training a neural network

Unlike linear regression the optimal parameters of a neural network cannot be written down in closed form. Therefore we must solve for them numerically with a gradient descent algorithm. Let us first recall how gradient descent works.

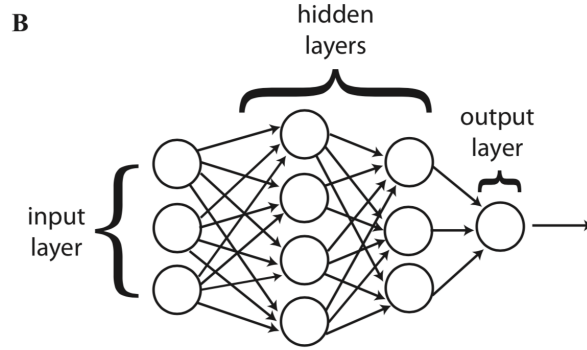


Figure 2.3: An artificial neural network with an input data point (layer) $x \in \mathbb{R}^3$, $A_1 : \mathbb{R}^3 \rightarrow \mathbb{R}^4$, $A_2 : \mathbb{R}^4 \rightarrow \mathbb{R}^3$, and $\alpha \in \mathbb{R}^3$. Figure taken from [6].

2.2.1 Gradient descent applied to neural networks

We are given a fixed training set $T = \{(x_i, y_i)\}_{i=1}^N$ and we want to minimize an empirical risk or more general loss function $\mathcal{L}(\theta)$ over $\theta \in \mathbb{R}^n$, which is given by:

$$\mathcal{L}(\theta) = \sum_{i=1}^N \ell(y_i, f(x_i; \theta)) = \sum_{i=1}^N \ell_i(\theta),$$

where $\ell_i(\theta) = \ell(y_i, f(x_i; \theta))$ is our point-wise loss function. Let

$$\hat{\theta} = \arg \min_{\theta \in \mathbb{R}^n} \mathcal{L}(\theta).$$

To solve for $\hat{\theta}$ we make an initial guess θ_0 and evaluate $\mathcal{L}(\theta_0)$. We then compute the gradient of \mathcal{L} with respect to θ and evaluate the gradient at θ_0 , i.e., we compute $\nabla_{\theta} \mathcal{L}(\theta_0)$. We then select a step size $\gamma_0 > 0$ (also called the learning rate in deep learning), and we “move” θ_0 in the direction of the negative of the gradient so that we reduce the value of $\mathcal{L}(\theta_0)$ so long as γ_0 is small enough:

$$\theta_1 = \theta_0 - \gamma_0 \nabla_{\theta} \mathcal{L}(\theta_0).$$

We then repeat the process, but now from θ_1 . After t steps we have θ_t and we update to θ_{t+1} via:

$$\theta_{t+1} = \theta_t - \gamma_t \nabla_{\theta} \mathcal{L}(\theta_t).$$

The process stops when $\nabla_{\theta} \mathcal{L}(\theta_t) \approx \mathbf{0}$. For “nice” optimization problems, e.g. convex problems with additional constraints on \mathcal{L} , one can provide proofs for when this algorithm will work and how many steps it will take; see [10] for more details.

In order to implement the gradient descent algorithm it is of paramount importance that we be able to compute $\nabla_{\theta} \mathcal{L}(\theta)$. We see that it can be written

as:

$$\nabla_{\theta} \mathcal{L}(\theta) = \sum_{i=1}^N \nabla_{\theta} \ell(y_i, f(x_i; \theta)).$$

Suppose now that $\ell(y, f(x)) = (y - f(x))^2$, the squared loss. Then, using the chain rule, we have

$$\nabla_{\theta} \ell(y, f(x; \theta)) = -2(y - f(x; \theta)) \nabla_{\theta} f(x; \theta),$$

and so we see that

$$\nabla_{\theta} \mathcal{L}(\theta) = -2 \sum_{i=1}^N (y_i - f(x_i; \theta)) \nabla_{\theta} f(x_i; \theta).$$

Thus we have reduced the problem to computing $\nabla_{\theta} f(x; \theta)$. When $f(x; \theta) = f(x; \theta, \alpha)$ is a neural network model as in (2.2) this computation looks formidable. But we can in fact do something similar to what we have done already: namely use the chain rule. If σ is a function that we can compute the derivative of analytically, meaning that we can write down the formula for $\sigma'(z)$, then we can analytically compute $\nabla_{\theta} f(x; \theta, \alpha)$ using the chain rule. The *back propagation* algorithm carries out the chain rule computation in a clever way, computing it layer by layer through the network. We omit the details, but back propagation forms the backbone of training neural networks.

2.2.2 Stochastic gradient descent

Now, while this solves the issue of computing the gradient in theory, in practice it is still computationally very expensive in the case of neural network training. Indeed, observe that:

- We need to compute $\nabla_{\theta} f(x_i; \theta)$ for each $1 \leq i \leq N$. Often N , the number of training points, can be quite large, particularly in computer vision when the number of training points can be in the millions.
- Furthermore, $\theta = (\theta, \alpha) \in \mathbb{R}^n$, with the number of parameters n being quite large. In particular,

$$n = \sum_{\ell=1}^L d_{\ell-1} d_{\ell} + \sum_{\ell=1}^L d_{\ell} + d_L.$$

Since $\nabla_{\theta} f(x_i; \theta) \in \mathbb{R}^n$, this means we need to compute n values for each x_i . The number of parameters in a neural network can be in the millions.

- Thus the total number of values that need to be computed at each gradient step is nN , which if $N \approx 10^6$ and $n \approx 10^6$, then $nN \approx 10^{12}$!

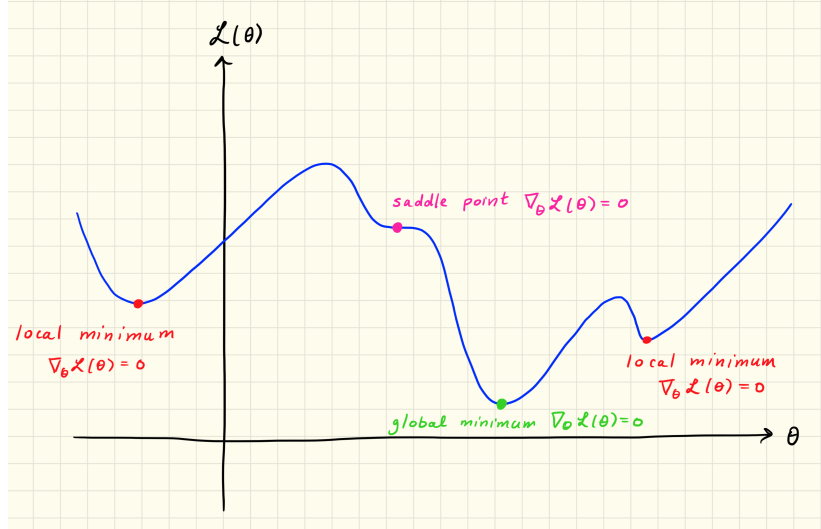


Figure 2.4: Illustration, in one dimension, of local minima, saddle points, and the global minimum.

Furthermore, since neural networks consist of the composition of many linear and nonlinear functions, there can be many local minima and saddle points. Many local minima means we have a non-convex optimization problem and thus finding the global minimum is very difficult, and saddle points mean that the gradient descent algorithm can get stuck at points that are not even local minima; see Figure 2.4.

What has worked well in practice and what researchers are making progress in understanding is to randomize the gradient descent process. The resulting class of algorithms are called *stochastic gradient descent*. In a stochastic gradient descent we sub-select, at random, $N_B \ll N$ of the training points, without replacement, and place them in a mini-batch B_1 . We then randomly select another N_B of the remaining $N - N_B$ training points, and place them in a mini-batch B_2 . We do so until we run out of training points, giving us N/N_B mini-batches $B_1, \dots, B_{N/N_B}$. Starting at θ_0 we update to θ_1 using only the mini-batch B_1 :

$$\theta_1 = \theta_0 - \gamma_0 \sum_{i \in B_1} \nabla_{\theta} \ell(y_i, f(x_i, \theta_0)).$$

We then update to θ_2 using only B_2 :

$$\theta_2 = \theta_1 - \gamma_1 \sum_{i \in B_2} \nabla_{\theta} \ell(y_i, f(x_i, \theta_1)).$$

Subsequent updates follow accordingly, until we run out of mini-batches, which is referred to as an epoch. At the completion of an epoch, the entire process is repeated, using new random mini-batches.

Stochastic gradient descent allows us to take N/N_B gradient steps for the same cost as one traditional gradient step. Furthermore, each step has a certain element of randomness in it due to the random selection of mini-batches, therefore enabling the optimization to escape saddle points. By adding momentum, stochastic gradient descent can also escape local minima. Many variations of the algorithm have been developed, a notable example being ADAM [11]. Research into why stochastic gradient descent works is active and ongoing.

2.2.3 Why ReLU? Vanishing gradients and optimization

Classically, sigmoidal type nonlinear activation functions have been used in neural networks because they mimic the perceptron, which is thought to be a good biological model for neural activation in the brain. On the other hand, recently the ReLU activation function $\sigma(z) = \max(0, z)$ and variants of it have become very popular. One explanation for this is in the optimization of neural networks, and the phenomenon of “vanishing gradients.”

Consider a simple neural network in which $x \in \mathbb{R}$, all weights and biases are also scalar valued, and we have $L = 2$,

$$f(x; \theta) = \alpha \sigma_2(w_2 \sigma_1(w_1 x + b_1) + b_2).$$

Let us compute the partial derivative of $f(x; \theta)$ with respect to w_1 ; we obtain:

$$\partial_{w_1} f(x; \theta) = \alpha \sigma_2'(w_2 \sigma_1(w_1 x + b_1) + b_2) w_2 \sigma_1'(w_1 x + b_1) x.$$

More generally, for L layers but with the same scalar valued weights and biases, we would have:

$$\partial_{w_1} f(x; \theta) = \alpha \cdot \left(\prod_{\ell=2}^L w_\ell \right) \cdot \left(\prod_{\ell=1}^L \sigma_\ell'(z_\ell) \right) \cdot x, \quad (2.3)$$

for values

$$z_\ell = A_\ell(\sigma_{\ell-1}(A_{\ell-1}(\cdots \sigma_1(A_1(x)))).$$

Notice that if σ_ℓ is a sigmoid or hyperbolic tangent, that $\sigma_\ell'(z_\ell)$ will be small if $|z_\ell| \gg 0$. Furthermore, if several such values $\sigma_\ell'(z_\ell)$ for multiple ℓ are small, we will multiply them together, making them even smaller. Thus the partial derivative in the w_1 weight will be nearly zero, making it very hard for the gradient descent algorithm to change this weight. Similar analysis holds for the other weights, although less multiplications will be involved. This phenomenon is referred to as the *vanishing gradient* problem in deep learning, and affects all neural networks, not just scalar valued ones.

The ReLU function helps to alleviate the the vanishing gradient issue and thus make training easier since $\sigma'(z) = 1$ for $z > 0$ if $\sigma(z) = \max(0, z)$. Thus the gradient will not vanish due to σ . Equation (2.3) indicates one also needs to be careful with the weights, making sure they do not get too small either, since the values of all the weights affects the gradient of w_1 . On the other hand,

if the weights get too large, their multiplication will be a very large number, and the gradient will explode, leading to a very unstable optimization. Proper initialization of the training data and weights help with these matters, as do other heuristics in deep learning, that we will not go into but may be of interest to the reader.

We will see in later sections, that ReLU is also a good choice from the perspective of approximation theory.

2.3 Classic approximation theory results

We now shift into approximation theoretic results for artificial neural networks, which was an active area of research in the late 1980s and 1990s, and with the recent popularity of neural networks has re-emerged as an important topic in mathematics and computer science. The field of approximation theory is concerned with one's ability to approximate complex functions with a collection of simpler functions.

In our case, we will focus on the data space $\mathcal{X} = [0, 1]^d$, the complex functions will be a deterministic label function

$$y = F(x), \quad x \in [0, 1]^d,$$

and the simpler functions will be one layer neural networks

$$\mathcal{F} = \{f(x; \theta) = f(x; \mathbf{W}, b, \alpha)\},$$

of the form (2.2). Initially we will discuss results of the following form: given $F(x)$ and $\epsilon > 0$, does there exist a neural network $f(x; \theta)$ (meaning does there exist a number of layers $L < \infty$ and a finite number of parameters θ) such that

$$|F(x) - f(x; \theta)| < \epsilon, \quad \text{for all } x \in \mathcal{X}?$$

Recalling our squared loss we will also consider the following: for each $\epsilon > 0$ and $F(x)$ does there exist a neural network $f(x; \theta)$ such that

$$\mathbb{E}_X[(F(X) - f(X; \theta))^2] = \int_{\mathcal{X}} (F(x) - f(x; \theta))^2 p_X(x) dx < \epsilon?$$

We will initially just be concerned with whether the answer to these questions is true or not. We will see that for certain classes of functions and certain neural network architectures the answer is indeed in the affirmative. Then we will investigate further and try to interpret and relate such results to “real life,” which often will motivate the search for better theory.

2.3.1 One layer approximation theory

One of the earlier and most popular works on approximation properties of artificial neural networks is by George Cybenko [12]. The result proves that one

layer neural networks with sigmoid-like activation functions can approximate any continuous function on the unit cube $\mathcal{X} = [0, 1]^d$. Let us discuss the main points of this result.

In this section we consider one layer neural network regression functions. Let \mathbf{W} be a $d \times m$ weight matrix, b an $m \times 1$ bias vector, and α an $m \times 1$ weight vector, which we write as:

$$\mathbf{W} = \begin{pmatrix} | & & | \\ w_1 & \cdots & w_m \\ | & & | \end{pmatrix} \quad b = \begin{pmatrix} b(1) \\ \vdots \\ b(m) \end{pmatrix} \quad \alpha = \begin{pmatrix} \alpha(1) \\ \vdots \\ \alpha(m) \end{pmatrix}.$$

Our one layer neural networks take the form:

$$f(x; \mathbf{W}, b, \alpha) = \langle \sigma(\mathbf{W}^T x + b), \alpha \rangle = \sum_{k=1}^m \alpha(k) \sigma(\langle x, w_k \rangle + b(k)), \quad (2.4)$$

where $\sigma : \mathbb{R} \rightarrow \mathbb{R}$ is a *sigmoidal function*, meaning that

$$\sigma(z) \rightarrow \begin{cases} 1 & \text{as } z \rightarrow +\infty \\ 0 & \text{as } z \rightarrow -\infty \end{cases} \quad (2.5)$$

Note, in particular, that the regular sigmoid function is a sigmoidal function.

We will initially consider labeling functions of the form

$$y = F(x), \quad F \in \mathbf{C}[0, 1]^d,$$

where $\mathbf{C}[0, 1]^d$ consists of all continuous functions $F : [0, 1]^d \rightarrow \mathbb{R}$. We have the following theorem.

Theorem 2.1 (Cybenko 1989, [12]). *Let σ be any continuous sigmoidal function (2.5). Given an $F \in \mathbf{C}[0, 1]^d$ and an $\epsilon > 0$, there is a one layer neural network $f(x; \mathbf{W}, b, \alpha)$ of the form (2.4) with $m, \mathbf{W} \in \mathbb{R}^{d \times m}$, $b \in \mathbb{R}^m$, and $\alpha \in \mathbb{R}^m$ depending on d, F , and ϵ , for which*

$$|f(x; \mathbf{W}, b, \alpha) - F(x)| < \epsilon, \quad \text{for all } x \in [0, 1]^d.$$

We will comment on the poof of Theorem 2.1 at the end of this section. The theorem shows that any continuous label function $y = F(x)$ supported on the unit cube can be approximated by a one layer sigmoidal neural network to arbitrary accuracy. This type of result is often referred to as a “universal approximation” theorem. It can extended to perceptrons on other discontinuous sigmoid functions if we replace the $L^\infty[0, 1]^d$ error of Theorem 2.1 with an $L^1[0, 1]^d$ error. This result initially sounds very impressive, but as we will see later it in fact is not that meaningful in explaining why neural networks work so well (in fairness, though, it is one of the first such results).

First though, let us first use Theorem 2.1 to prove a result about categorical classification. Let $\mathcal{C}_1, \dots, \mathcal{C}_M \subseteq [0, 1]^d$ be a partition of the cube $[0, 1]^d$, meaning that

$$\mathcal{C}_i \cap \mathcal{C}_j = \emptyset \text{ if } i \neq j \quad \text{and} \quad \bigcup_{i=1}^M \mathcal{C}_i = [0, 1]^d.$$

Assign labels as:

$$y = F(x) = i \quad \text{if and only if} \quad x \in \mathcal{C}_i. \quad (2.6)$$

We refer to F in this case as a *decision function*. Note that this is a classification problem. We want to know if a neural network can implement a good decision boundary. The following theorem says it can.

Theorem 2.2 (Cybenko 1989, [12]). *Let σ be any continuous sigmoidal function (2.5). Let F be a decision function (2.6) and $\epsilon > 0$. Then, there exists a one layer neural network $f(x; \mathbf{W}, b, \alpha)$ of the form (2.4) with $m, \mathbf{W} \in \mathbb{R}^{d \times m}$, $b \in \mathbb{R}^m$, and $\alpha \in \mathbb{R}^m$ depending on d, F , and ϵ , and a set $\mathcal{D} \subseteq [0, 1]^d$ with $\text{vol}(\mathcal{D}) \geq 1 - \epsilon$, for which*

$$|f(x; \mathbf{W}, b, \alpha) - F(x)| < \epsilon, \quad \text{for all } x \in \mathcal{D}.$$

As a consequence, define the classifier $\tilde{f}(x; \mathbf{W}, b, \alpha)$ as:

$$\tilde{f}(x; \mathbf{W}, b, \alpha) = \text{Round}[f(x; \mathbf{W}, b, \alpha)],$$

where $\text{Round}[z]$ is the closest integer to $z \in \mathbb{R}$. If $\epsilon < 1/2$, then

$$\tilde{f}(x; \mathbf{W}, b, \alpha) = F(x), \quad \text{for all } x \in \mathcal{D}.$$

Proof sketch. Use Lusin's theorem combined with Theorem 2.1 to prove the result for f . The result for \tilde{f} follows immediately. \square

Because $f(x; \mathbf{W}, b, \alpha)$ is always a continuous function and a decision function $F(x)$ is necessarily not, there will always be some points classified incorrectly. On the other hand, the result says the volume of the number of points classified incorrectly can be made arbitrarily small. While this theorem does not say anything about the geometry of the set \mathcal{D} , one can refine the analysis further to conclude that the one layer neural network can learn a “natural” approximation of $F(x)$ where any point sufficiently far away from a boundary defined by $F(x)$ is classified correctly.

The key to Cybenko's results in [12] is the notion of a *discriminatory function*. We give the definition here without explaining everything since it relies on notions from graduate level analysis. Cybenko says that a function $\sigma : \mathbb{R} \rightarrow \mathbb{R}$ is discriminatory if for any finite, signed Borel measure μ on $[0, 1]^d$,

$$\text{for all } w \in \mathbb{R}^d, b \in \mathbb{R}, \quad \int_{[0,1]^d} \sigma(\langle x, w \rangle + b) d\mu(x) = 0 \implies \mu \equiv 0.$$

Cybenko proves that sigmoidal functions are discriminatory and then uses this to obtain his results. An example such a measure is $d\mu(x) = p_X(x) dx$, where $p_X : [0, 1]^d \rightarrow \mathbb{R}$ is a probability density function on $\mathcal{X} = [0, 1]^d$, but there are other more exotic finite, signed Borel measures.

Cybenko's results were generalized by Hornik in [13]. Among his results, he proves the following.

Theorem 2.3 (Hornik 1991, [13]³). *Let σ be any non-constant, bounded activation function (e.g., sigmoidal, but others are okay too). Let p_X be a probability density function on $[0, 1]^d$ and let $F \in \mathbf{L}^2([0, 1]^d, p_X)$, which means that*

$$\mathbb{E}_X[|F(X)|^2] = \int_{[0,1]^d} |F(x)|^2 p_X(x) dx < \infty.$$

Then for each $\epsilon > 0$ there exists a one layer neural network $f(x; \mathbf{W}, b, \alpha)$ of the form (2.4) with $m, \mathbf{W} \in \mathbb{R}^{d \times m}, b \in \mathbb{R}^m$, and $\alpha \in \mathbb{R}^m$ depending on d, F, ϵ , and p_X , such that

$$\mathbb{E}_X[(F(X) - f(X; \mathbf{W}, b, \alpha))^2] = \int_{[0,1]^d} (F(x) - f(x; \mathbf{W}, b, \alpha))^2 p_X(x) dx < \epsilon.$$

Most of the proofs contained in [12] and [13] are not constructive and they give no insight into the relationship between m , the number of neurons, and ϵ , the desired accuracy. They also give no relationship between the magnitude of the weights \mathbf{W} and biases b and ϵ . We are interested in this type of analysis because each weight and bias needs to be learned from training data, and thus the number of such weights is a proxy for the amount of training data we will need. The magnitude of these values is also important, as computers cannot store infinitely large values. We will see that these relationships are not favorable, at least by the analysis of this section.

2.3.2 One layer approximation and the curse of dimensionality

While universal approximation results such as those contained in Section 2.3.1 appear to be quite powerful, in fact mathematicians have been developing classes of “simple” functions that can approximate to arbitrary accuracy “complex” functions in, for example, $\mathbf{C}[0, 1]^d$ or $\mathbf{L}^2[0, 1]^d$, for quite a long time. Here are two well known examples:

Example 2.4. Polynomials: Let $\mathcal{F} = \{f(x; \theta)\}_\theta$ be the model class of all polynomials, i.e.,

$$f(x; \theta) = \sum_{\|\beta\|_1 \leq m} \theta(\beta) x^\beta,$$

where $\beta = (\beta(1), \dots, \beta(d)) \in \mathbb{N}^d$,

$$\|\beta\|_1 = \sum_{k=1}^d \beta(k),$$

and

$$x^\beta = \prod_{k=1}^d x(k)^{\beta(k)},$$

³This result is Theorem 1 in [13], which says “unbounded” but should say “bounded.” Thanks to Gautam Sreeksumar asking about this result in class!

and where \mathcal{F} includes all polynomials for every $m \geq 0$. That is, there is no bound on the degree of the polynomial (this is really important!). Then we have:

Theorem 2.5 (Stone-Weierstrass). *Let $F \in \mathbf{C}[0,1]^d$. Then for each $\epsilon > 0$, there exists a polynomial $f(x;\theta)$ such that*

$$|F(x) - f(x;\theta)| < \epsilon, \quad \forall x \in [0,1]^d.$$

Example 2.6. Fourier series: Now let $\mathcal{F} = \{f(x;\theta)\}_\theta$ be the model class of all partial Fourier series, defined as

$$f(x;\theta) = \sum_{\|\beta\|_\infty \leq m} \theta(\beta) e^{2\pi i \langle \beta, x \rangle},$$

where $\beta = (\beta(1), \dots, \beta(d)) \in \mathbb{Z}^d$ and

$$\|\beta\|_\infty = \max_{1 \leq k \leq d} |\beta(k)|.$$

Recall that $\mathbf{L}^2[0,1]^d$ is the space of functions $F(x)$ that are square integrable,

$$F \in \mathbf{L}^2[0,1]^d \iff \int_{[0,1]^d} |F(x)|^2 dx < \infty.$$

Then it is well known in harmonic analysis that for each $F \in \mathbf{L}^2[0,1]^d$, $\epsilon > 0$, there exists a partial Fourier series $f(x;\theta)$ such that

$$\|F - f(\cdot;\theta)\|_2 = \left(\int_{[0,1]^d} (F(x) - f(x;\theta))^2 dx \right)^{1/2} < \epsilon.$$

Furthermore,

$$\theta(\beta) = \int_{[0,1]^d} F(x) e^{-2\pi i \langle \beta, x \rangle} dx.$$

These examples show that while universal approximation is important, it is not everything. What is often equally or even more important is the *rate of approximation*. Both [12] and [13] rely on density arguments to prove their results, which do not come with quantitative estimates for the rate of approximation. Furthermore, Cybenko in the conclusion of [12] speculates that the number of neurons needed, as indicated by his analysis, would be “astronomical” due to the curse of dimensionality.

Let us examine this further for one layer ReLU networks. The point of this analysis is to show that classical approximation theory yields the following interpretation of such networks:

- One layer ReLU networks implement local linear interpolation, which will suffer from the curse of dimensionality in that local linear interpolation requires a lot of training data. Recall our discussion of k -nearest neighbors, a local method, and the curse of dimensionality in Section 1.4.3.

- The number of neurons in this context will be nearly the same as the number of training points, and thus the network width will be astronomical.

Consider one layer ReLU networks of the form:

$$f(x; \theta) = \sum_{k=1}^m \alpha(k) \max(x - b(k), 0) + \beta,$$

where $\beta \in \mathbb{R}$ is an additional bias term applied in the output layer.

Let $\mathcal{X} = [0, 1]$ and assume that $F : [0, 1] \rightarrow \mathbb{R}$ is Lipschitz, which means that there exists a universal constant C (depending on F , but not x and x') such that

$$|F(x) - F(x')| \leq C|x - x'|, \quad \forall x, x' \in [0, 1].$$

Note this means

$$|x - x'| \leq \epsilon \implies |F(x) - F(x')| \leq C\epsilon. \quad (2.7)$$

So let us suppose (somewhat unrealistically) that we get a training set $(x_i, F(x_i))_{i=0}^N$ with $x_0 = 0, x_1 = \epsilon, x_2 = 2\epsilon, \dots$, that is:

$$x_i = i\epsilon, \quad 0 \leq i \leq N = \epsilon^{-1}.$$

Note we have $N + 1$ training points in this example.

We are going to come up with a neural network that is a piecewise linear approximation of $F(x)$ with $f(x_i; \theta) = F(x_i)$ for all $0 \leq i \leq N$. It will satisfy:

$$f(x; \theta) = \epsilon^{-1}[(x_{i+1} - x)F(x_i) + (x - x_i)F(x_{i+1})], \quad x_i \leq x \leq x_{i+1}. \quad (2.8)$$

Note that $f(x; \theta)$ uniformly approximates $F(x)$ up to an error $C\epsilon$. Indeed, let $x_i \leq x \leq x_{i+1}$ and note that $x_{i+1} - x_i = \epsilon$ and thus $(x_{i+1} - x) + (x - x_i) = \epsilon$ as well. We have:

$$\begin{aligned} |f(x; \theta) - F(x)| &= \left| \frac{(x_{i+1} - x)F(x_i) + (x - x_i)F(x_{i+1})}{\epsilon} - F(x) \right| \\ &= \left| \frac{(x_{i+1} - x)}{\epsilon} (F(x_i) - F(x)) + \frac{(x - x_i)}{\epsilon} (F(x_{i+1}) - F(x)) \right| \\ &\leq \frac{(x_{i+1} - x)}{\epsilon} |F(x_i) - F(x)| + \frac{(x - x_i)}{\epsilon} |F(x_{i+1}) - F(x)| \\ &\leq \frac{(x_{i+1} - x)}{\epsilon} \cdot C\epsilon + \frac{(x - x_i)}{\epsilon} \cdot C\epsilon \\ &= C\epsilon, \end{aligned}$$

where the first inequality is the triangle inequality and the second inequality follows from (2.7).

To obtain this network set the number of neurons to be $m = N = \epsilon^{-1}$. Set the internal biases of the neurons to be:

$$b(k) = (k - 1)\epsilon,$$

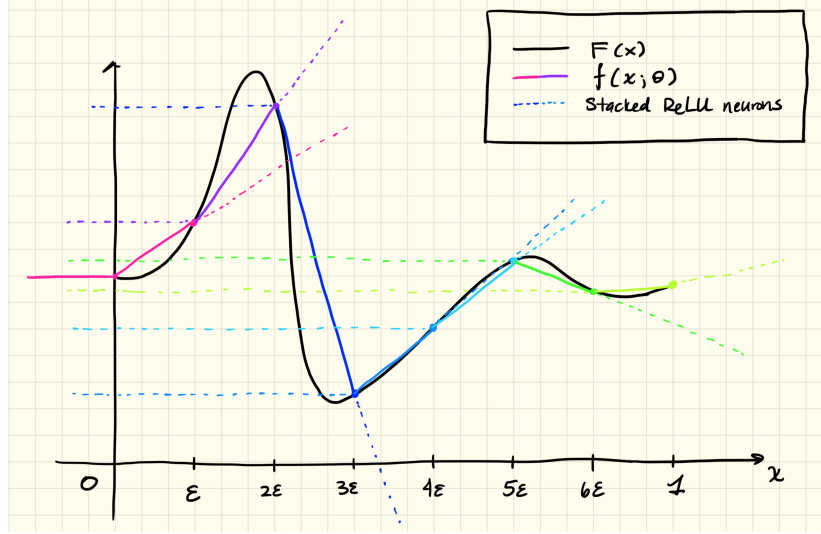


Figure 2.5: Illustration of a one-layer ReLU network implementing a piecewise linear approximation of a Lipschitz function $F(x)$. The solid black line is $F(x)$. The solid multi-color line is the one-layer ReLU neural network $f(x; \theta)$, with each color indicating a linear part of $f(x; \theta)$. The dashed lines show the stacked ReLU functions, which are shifted by the biases b .

which means that in the ReLU activations we have

$$\max(0, x - b(k)) = \max(0, x - (k-1)\epsilon) = \begin{cases} 0 & x \leq (k-1)\epsilon \\ x - (k-1)\epsilon & x > (k-1)\epsilon \end{cases}.$$

For the first neuron and the external bias β we set:

$$\alpha(1) = \frac{F(x_1) - F(x_0)}{\epsilon} \quad \text{and} \quad \beta = F(x_0),$$

which ensures that $f(x_0; \theta) = F(x_0)$ (recall $x_0 = 0$) and $f(x_1; \theta) = F(x_1)$, and furthermore that (2.8) holds. For the remaining neurons we set:

$$\alpha(k) = \epsilon^{-1} [F(x_k) - 2F(x_{k-1}) + F(x_{k-2})], \quad 1 < k \leq N.$$

One can verify that with these choices (2.8) holds. Furthermore, all of the weights $\alpha \in \mathbb{R}^N$ and biases $b \in \mathbb{R}^N, \beta \in \mathbb{R}$, depend only on the training data. We have thus showed that to obtain an error $C\epsilon$ for a one-dimensional regression problem, where C is the Lipschitz constant of F , we need $\epsilon^{-1} + 1$ training points and ϵ^{-1} neurons in our single layer ReLU network; see Figure 2.5 for an illustration.

If we extend the above analysis to higher dimensions we run into the curse of dimensionality. Indeed, we relied on having a training point x_i within $\epsilon/2$ of every possible test point x in the above calculations. To ensure such a property in $\mathcal{X} = [0, 1]^d$ we would need $N = O(\epsilon^{-d})$ training points (e.g., an ϵ -spaced grid). Not only is that a very large number of training points, but we also saw

that the number of neurons needed is $m = N = O(\epsilon^{-d})$, which explodes the width of the one-layer ReLU network.

From the perspective of classical approximation theory, one way to resolve this issue is to assume that $F(x)$ is smoother, say $F \in \mathbf{C}^{s+1}[0, 1]^d$, which means that F and all derivatives of F up to order $(s + 1)$ are continuous and bounded. In this case one can approximate $F(x)$ in a neighborhood of $u \in \mathbb{R}^d$ by a polynomial $p_u(x)$, which is the Taylor polynomial of F around u , i.e.,

$$p_u(x) = \sum_{\|\beta\|_1 \leq s} \frac{1}{\beta!} (\partial^\beta F)(u) (x - u)^\beta,$$

where as in Example 2.4 $\beta = (\beta(1), \dots, \beta(d)) \in \mathbb{N}^d$ with $\|\beta\|_1$ and x^β defined the same way. We also define

$$\beta! = \prod_{k=1}^d \beta(k)!,$$

and

$$(\partial^\beta F)(x) = [(\partial_1)^{\beta(1)} \dots (\partial_d)^{\beta(d)} F](x).$$

The approximation property of $p_u(x)$ can be quantified using Taylor's theorem, which gives

$$|F(x) - p_u(x)| \leq C \|x - u\|^s, \quad (2.9)$$

where C is the $\mathbf{C}^{s+1}[0, 1]^d$ norm of F ; that is,

$$C = \max_{\|\beta\|_1 \leq s+1} \sup_{x \in [0, 1]^d} |\partial^\beta F(x)|.$$

From (2.9) it follows that

$$\|x - u\| \leq \epsilon^{1/s} \implies |F(x) - p_u(x)| \leq C\epsilon.$$

Suppose then that instead of a one-layer ReLU network, which implements a local linear regression, we instead developed a one layer neural network in which each neuron implements a local polynomial regression. The previous equation shows that in order to approximate $F(x)$ uniformly on $[0, 1]^d$ with a $C\epsilon$ error, we would need to have a training set that samples $[0, 1]^d$ on an $\epsilon^{1/s}$ grid, which means that

$$\# \text{ of training points} = \# \text{ of neurons} + 1 = O(\epsilon^{-d/s}).$$

This is definitely an improvement over the one layer ReLU network, but is still inadequate. Indeed, the dimension d is often very large and the smoothness s is rarely proportional to d , meaning that $s \ll d$ and thus that the number of training points / the number of neurons is still astronomical. For future reference, we remark that if the number of training points $N + 1$ is fixed, and hence the

number of “local polynomial neurons” is N , we obtain that there exists such a one layer neural network $f_N(x; \theta)$ with

$$\sup_{x \in [0,1]^d} |F(x) - f_N(x; \theta)| < CN^{-s/d}. \quad (2.10)$$

Thus as the number of training points $N \rightarrow \infty$, the approximation improves and the rate at which the error goes to zero speeds up with increased smoothness (i.e., the larger s , the faster the convergence) and decreases with dimension (i.e., the larger d , the slower the convergence).

2.3.3 Refined one-layer analysis and two-layer neural networks

Our goal in this section is to arrive at a seemingly remarkable two-layer result of Pinkus that can be found in [14].

Refinements of the one layer analysis

We first extend the one layer results to more general activation functions, and also refine their analysis. Recall the approximation theorems of Cybenko (Theorem 2.1 and 2.2) and Hornik (Theorem 2.3) relied on σ being either sigmoidal or bounded and non-constant, while the analysis in the previous section used the ReLU activation specifically. In fact, it turns out that any continuous σ that is not a polynomial works. Let us explain.

In order to simplify the exposition, note that any one layer neural network of the form

$$f(x; \theta) = f(x; \mathbf{W}, b, \alpha) = \sum_{k=1}^m \alpha(k) \sigma(\langle x, w_k \rangle + b(k)), \quad (2.11)$$

lies in the space

$$\mathcal{M}(\sigma) = \text{span}\{\sigma(\langle x, w \rangle + b) : w \in \mathbb{R}^d, b \in \mathbb{R}\},$$

and vice versa, that is any function in $M(\sigma)$ is a one-layer neural network of the form (2.11). Theorem 2.1 and variants of it are then concerned with the following question: For which $\sigma : \mathbb{R} \rightarrow \mathbb{R}$ is it true that, for any $F \in \mathbf{C}[0,1]^d$ and $\epsilon > 0$, there exists a function $f(x; \theta) \in M(\sigma)$ such that

$$\sup_{x \in [0,1]^d} |F(x) - f(x; \theta)| < \epsilon?$$

Theorem 2.1 proves the result for $\sigma(z)$ that are continuous and sigmoidal. It turns out that the result is true for much more general functions σ .

Theorem 2.7 (Pinkus 1999, [14]). *Let $\sigma(z)$ be a continuous function. If $\sigma(z)$ is not polynomial, then for each $F \in \mathbf{C}[0,1]^d$ and each $\epsilon > 0$, there exists an $f \in M(\sigma)$ such that*

$$\sup_{x \in [0,1]^d} |F(x) - f(x; \theta)| < \epsilon. \quad (2.12)$$

Conversely, if for each $F \in \mathbf{C}[0,1]^d$ and each $\epsilon > 0$ there exists an $f \in M(\sigma)$ such that (2.12) holds, then $\sigma(z)$ is not a polynomial.

Theorem 2.7 strengthens Theorem 2.1 by allowing for any continuous activation function $\sigma(z)$ so long as it is not a polynomial. The converse is relatively trivial. Indeed, if $\sigma(z)$ were a polynomial of degree m , then $M(\sigma)$ would be the space of all degree m or less polynomials, which certainly cannot approximate any continuous function (simply take F as a polynomial of degree strictly larger than m). Let us sketch the proof of Theorem 2.7, as it contains some interesting points.

Proof sketch of Theorem 2.7. The first part of Pinkus proof is to reduce the problem over $[0,1]^d$ to a problem over compact subsets of \mathbb{R} . First define $\mathcal{N}(\sigma)$ as the one-dimensional analogue of $\mathcal{M}(\sigma)$ (recall $z \in \mathbb{R}$):

$$\mathcal{N}(\sigma) = \text{span}\{\sigma(wz + b) : w \in \mathbb{R}, b \in \mathbb{R}\}.$$

The following proposition proves that solving the problem in one dimension using $\mathcal{N}(\sigma)$ is enough to solve the problem in d -dimensions with $\mathcal{M}(\sigma)$:

Proposition 2.8 (Pinkus 1999, [14]). *Let $K \subset \mathbb{R}$ be a compact set, which means K is closed and bounded. Suppose for every such K , for each $G \in \mathbf{C}(K)$, and for each $\epsilon > 0$, there exists a $g \in \mathcal{N}(\sigma)$ such that*

$$\sup_{z \in K} |G(z) - g(z; \gamma)| < \epsilon \quad (\gamma \text{ are the parameters of } g).$$

Then for each $F \in \mathbf{C}[0,1]^d$ and each $\epsilon > 0$, there exists an $f \in \mathcal{M}(\sigma)$ such that

$$\sup_{x \in [0,1]^d} |F(x) - f(x; \theta)| < \epsilon.$$

We will not prove this proposition, but we will use it, since it allows us to restrict our attention to functions $G \in \mathbf{C}(K)$, with $K \subset \mathbb{R}$, as opposed to $F \in \mathbf{C}[0,1]^d$. The following proposition, combined with Proposition 2.8, does most of the work in proving Theorem 2.7.

Proposition 2.9 (Pinkus 1999, [14]). *Let $\sigma \in \mathbf{C}^\infty(\mathbb{R})$ (i.e., σ can be differentiated an infinite number of times) and assume σ is not a polynomial. Let $K \subset \mathbb{R}$ be compact. Then for each $G \in \mathbf{C}(K)$ and each $\epsilon > 0$ there exists a $g \in \mathcal{N}(\sigma)$ such that*

$$\sup_{z \in K} |G(z) - g(z; \gamma)| < \epsilon.$$

Proof of Proposition 2.9. It is a known fact that if $\sigma \in \mathbf{C}^\infty(\mathbb{R})$ and σ is not a polynomial, then there exists a point $z_0 \in \mathbb{R}$ for which

$$\sigma^{(k)}(z_0) \neq 0, \quad \forall k \geq 0.$$

The proof is tricky and requires the use of tools from graduate functional analysis, so we omit it. But let us use this fact. Note that the function

$$\frac{\sigma((w+h)z+z_0) - \sigma(wz+z_0)}{h}$$

is in $\mathcal{N}(\sigma)$ for every $h \neq 0$. Letting h tend to zero we obtain:

$$\lim_{h \rightarrow 0} \frac{\sigma((w+h)z+z_0) - \sigma(wz+z_0)}{h} = \frac{d}{dw} \sigma(wz+z_0) = z\sigma'(wz+z_0).$$

Furthermore, if we set $w = 0$, we obtain:

$$\left. \frac{d}{dw} \sigma(wz+z_0) \right|_{w=0} = z\sigma'(z_0).$$

Therefore the function $\tilde{p}_1(z) = z\sigma'(z_0)$, and thus the function $p_1(z) = z$ (by rescaling by $1/\sigma'(z_0)$), can be approximated to arbitrary accuracy on K by functions in $\mathcal{N}(\sigma)$.

By similar arguments, we can conclude that the functions

$$\left. \frac{d^k}{dw^k} \sigma(wz+z_0) \right|_{w=0} = z^k \sigma^{(k)}(z_0),$$

can be approximated to arbitrary accuracy on K by functions in $\mathcal{N}(\sigma)$, which means that $p_k(z) = z^k$ can be approximated on K to arbitrary accuracy by functions in $\mathcal{N}(\sigma)$. But that is every monomial, which means that every polynomial can be approximated on K to arbitrary accuracy by functions in $\mathcal{N}(\sigma)$.

Now remember the Stone-Weierstrass Theorem from Example 2.4. In fact it holds for any compact set $K \subset \mathbb{R}$, meaning there exists a polynomial $p(z)$ such that

$$\sup_{z \in K} |G(z) - p(z)| < \epsilon/2.$$

Let $g \in \mathcal{N}(\sigma)$ approximate $p(z)$ uniformly, which we know we can do by the analysis carried out already in this proof:

$$\sup_{z \in K} |p(z) - g(z; \gamma)| < \epsilon/2.$$

It follows that

$$\sup_{z \in K} |G(z) - g(z; \gamma)| \leq \sup_{z \in K} |G(z) - p(z)| + \sup_{z \in K} |p(z) - g(z; \gamma)| < \epsilon.$$

□

The proof of Theorem 2.7 is completed by using the fact that compactly supported $C^\infty(\mathbb{R})$ functions are dense in $C(\mathbb{R})$, which allows one to remove the restriction that $\sigma \in C^\infty(\mathbb{R})$ in Proposition 2.9 and replace with the less restrictive assumption $\sigma \in C(\mathbb{R})$. Theorem 2.7 then follows by applying this modified version of Proposition 2.9 combined with Proposition 2.8. □

Back to the rate of approximation

Recall in Section 2.3.2 we constructed ReLU networks that interpolated the training data exactly, but needed a number of neurons that was only one less than the number of training points. Any one layer neural network with a continuous activation function σ that is not a polynomial can also accomplish this feat.

Theorem 2.10 (Pinkus 1999, [14]). *Let $\sigma(z)$ be a continuous function that is not a polynomial. Given training data $T = \{(x_i, y_i)\}_{i=1}^N$ with $x_i \in \mathbb{R}^d$ and $y_i \in \mathbb{R}$, there exists a one layer neural network $f \in \mathcal{M}(\sigma)$ with exactly N neurons such that $f(x_i; \theta) = y_i$ for all $1 \leq i \leq N$. That is, there exists $w_1, \dots, w_N \in \mathbb{R}^d$, $b \in \mathbb{R}^N$, and $\alpha \in \mathbb{R}^N$, such that*

$$\sum_{k=1}^N \alpha(k) \sigma(\langle x, w_k \rangle + b(k)) = y_i, \quad \forall, 1 \leq i \leq N.$$

Notice that similar to the ReLU activation function we require the number of neurons to be the same as the number of training points.

In the ReLU case, we also were able to obtain the rate of approximation. We can do the same for certain continuous activation functions $\sigma(z)$ that are not polynomials, but not all of them. These results are for $\mathcal{X} = B^d$, the unit ball in \mathbb{R}^d , i.e.,

$$B^d = \{x \in \mathbb{R}^d : \|x\|_2 \leq 1\}.$$

Like in the extension of the ReLU results, we will consider smooth label functions $F \in \mathbf{C}^s(B^d)$, where $s \geq 1$ indicates the smoothness of $F(x)$ ($s = 0$ just means $F(x)$ is continuous). Let $\mathcal{M}_N(\sigma) \subset \mathcal{M}(\sigma)$ denote one-layer neural networks with at most N neurons, i.e.,

$$f(x; \theta) = \sum_{k=1}^N \alpha(k) \sigma(\langle x, w_k \rangle + b(k)).$$

In light of Theorem 2.10, here we use N to denote the number of neurons since this is the number of neurons needed just to fit the training data. We also define the $\mathbf{L}^\infty(B^d)$ norm as:

$$\|F\|_{\mathbf{L}^\infty(B^d)} = \sup_{x \in B^d} |F(x)|,$$

which we note allows us to write:

$$\sup_{x \in B^d} |F(x) - f(x; \theta)| = \|F - f\|_{\mathbf{L}^\infty(B^d)}.$$

We also recall the $\mathbf{L}^p(B^d)$ norm:

$$\|F\|_{\mathbf{L}^p(B^d)} = \left(\int_{B^d} |F(x)|^p dx \right)^{1/p}.$$

Note that $\|F - f\|_{\mathbf{L}^2(B^d)}^2$ is the squared loss. Finally we recall the $\mathbf{C}^s(B^d)$ norm is:

$$\|F\|_{\mathbf{C}^s(B^d)} = \max_{\|\beta\|_1 \leq s} \sup_{x \in B^d} |\partial^\beta F(x)|. \quad (2.13)$$

Our first result is the following.

Theorem 2.11 (Pinkus 1999, [14]). *There exist $\sigma \in \mathbf{C}^\infty(\mathbb{R})$ which are sigmoidal and strictly increasing such that for every $F \in \mathbf{C}^s(B^d)$ with $\|F\|_{\mathbf{C}^s(B^d)} \leq 1$,*

$$\inf_{f \in \mathcal{M}_N(\sigma)} \|F - f\|_{\mathbf{L}^p(B^d)} \leq CN^{-s/(d-1)},$$

for each $1 \leq p \leq \infty$, $s \geq 1$, and $d \geq 2$. The constant C is independent of F and N .

Theorem 2.11 has nearly the same rate of approximation as “local polynomial neuron” analysis in (2.10). Like with that result, it shows that increased smoothness in $F(x)$ increases the rate of approximation, but it is still dampened by the dimension d which can be very large in modern machine learning and deep learning applications. Also note, the theorem says there exists a $\sigma \in \mathbf{C}^\infty(\mathbb{R})$ that is sigmoidal and strictly increasing for which the result holds, but it does not say the result holds for all such $\sigma(z)$. In particular, one cannot conclude the result holds for the regular sigmoid function, and in fact Pinkus remarks that the activation functions $\sigma(z)$ for which Theorem 2.11 holds are pathological and overly complex, despite them having the nice properties of being infinitely differentiable, sigmoidal, and strictly increasing. Nevertheless, the result gives a worst case bound on the rate of approximation for certain one layer neural networks with smooth sigmoidal activation functions. Notice as well that it holds for all \mathbf{L}^p loss functions, for $1 \leq p \leq \infty$, which includes the squared loss and uniform approximation.

One may ask, though, are these worst case results on the rate of approximation an artifact of proof technique, and thus not indicative of the true rate of approximation? For the squared loss, it turns out the answer is “no.” Here is the remarkable result.

Theorem 2.12 (Maierov 1999, [15]). *Let $\sigma(z)$ be a continuous activation function that is not a polynomial. Then for each $s \geq 1$, $d \geq 2$, and N , there exists an $F \in \mathbf{C}^s(B^d)$ with $\|F\|_{\mathbf{C}^s(B^d)} \leq 1$, such that*

$$\inf_{f \in \mathcal{M}_N(\sigma)} \|F - f\|_{\mathbf{L}^2(B^d)} \geq cN^{-s/(d-1)},$$

for each $s \geq 1$ and $d \geq 2$. The constant c is independent of F and N .

Combining Theorem 2.11 and Theorem 2.12 we see there exists sigmoidal activation functions such that the resulting one layer neural network achieves a rate of approximation of $O(N^{-s/(d-1)})$ for $F \in \mathbf{C}^s(B^d)$, but without additional information on F , one cannot expect to do better than $O(N^{-s/(d-1)})$.

As a final remark, we recall Example 2.4 which dealt with polynomial approximation. By the Stone-Weierstrass theorem we observed the space of all polynomials also have the universal approximation property. What we did not quantify was the rate of approximation. In fact it turns out that it is the same as the rate in Theorem 2.11. It thus follows, rather definitively, from this and the previous considerations, that if we want to understand the power of neural networks we must move beyond the one layer model.

Two-layer neural networks

It is tempting to say that one-layer neural networks are universal approximators and therefore what else is there to say regarding the additional layer. But this ignores both the reality of the situation, which is that in practice most applications utilize several to many layers, as well as fundamental theoretical differences between one-layer neural networks and multi-layer neural networks. In this subsection we study two layer neural networks in more depth.

One important difference between a one-layer network and a two-layer network is that in one-layer networks each neuron is delocalized, whereas in two layers the composition of two neurons can lead to a localized function. Let us make this statement more precise. Let d_1 be the number of neurons in a one-layer neural network. Recall the space $\mathcal{M}_{d_1}(\sigma)$ consists of one layer neural networks with at most d_1 neurons:

$$f(x; \theta) = \sum_{k=1}^{d_1} \alpha(k) \sigma(\langle x, w_k \rangle + b(k)) = \sum_{k=1}^{d_1} \alpha(k) \sigma \left(\sum_{j=1}^d w_k(j) x(j) + b(k) \right).$$

Let $x \in \mathbb{R}^d$ for $d \geq 2$ and let $f \in \mathcal{M}_{d_1}(\sigma)$, f not the function that is zero everywhere. Then, necessarily, we have

$$\forall 1 \leq p \leq \infty, \quad \int_{\mathbb{R}^d} |f(x)|^p dx = +\infty,$$

and in particular no $f \in \mathcal{M}_{d_1}(\sigma)$ has compact support.

In the two layer model the situation is different and in fact we can obtain compactly supported functions (or more precisely, functions whose support is contained in a compact set) in the second hidden layer. Recall, by a two layer neural network, we mean a function of the form:

$$\begin{aligned} f(x; \theta) &= \alpha^T \sigma(\mathbf{W}_2^T \sigma(\mathbf{W}_1^T x + b_1) + b_2) \\ &= \sum_{\ell=1}^{d_2} \alpha(\ell) \sigma \left(\sum_{k=1}^{d_1} w_{2,\ell}(k) \sigma(\langle x, w_k \rangle + b_1(k)) + b_2(\ell) \right) \\ &= \sum_{\ell=1}^{d_2} \alpha(\ell) \sigma \left(\sum_{k=1}^{d_1} w_{2,\ell}(k) \sigma \left(\sum_{j=1}^d w_{1,k}(j) x(j) + b_1(k) \right) + b_2(\ell) \right). \end{aligned}$$

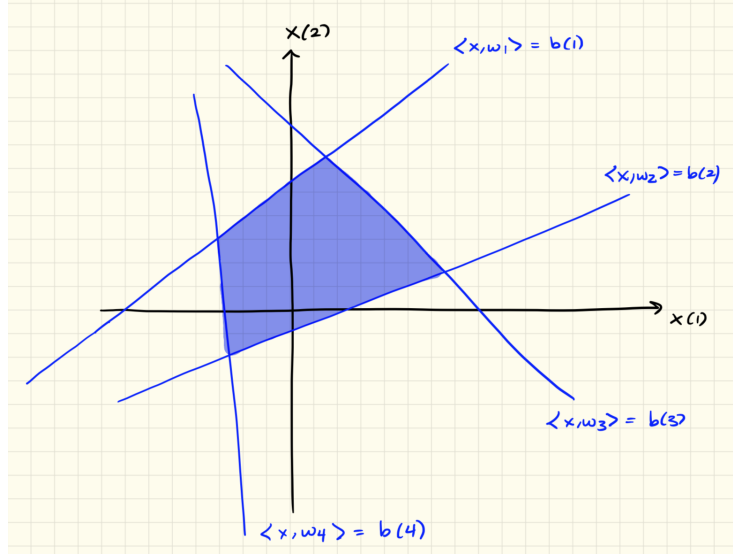


Figure 2.6: A two layer perceptron network can implement the characteristic function on any convex polygonal domain.

For example, let $\sigma_o(z)$ be the perceptron, that is

$$\sigma_o(z) = \begin{cases} 0 & z \leq 0 \\ 1 & z > 0 \end{cases}$$

Then

$$\sigma_o \left(\sum_{k=1}^{d_1} \sigma_o(\langle x, w_k \rangle - b(k)) - \left(d_1 - \frac{1}{2} \right) \right) = \begin{cases} 1 & \langle x, w_k \rangle > b(k), \quad \forall 1 \leq k \leq d_1 \\ 0 & \text{otherwise} \end{cases}$$

Thus with two hidden layers and the perceptron activation function we can represent the characteristic function of any convex polygonal domain; see Figure 2.6. If we replace the perceptron with the sigmoid function $\sigma(z)$, then we have

$$\lim_{\lambda \rightarrow +\infty} \int_{-1}^1 |\sigma(\lambda z) - \sigma_o(z)|^2 dz = 0.$$

In other words, the sigmoid function approximates the perceptron and thus the function

$$\sigma \left(\lambda \sum_{k=1}^{d_1} \sigma(\lambda(\langle x, w_k \rangle - b(k))) - \left(d_1 - \frac{1}{2} \right) \right)$$

approximates the analogous perceptron function as $\lambda \rightarrow +\infty$. It follows that the sigmoid version is a highly localized function, which gives two layer networks an advantage over one-layer networks.

Theoretically, we observed for one-layer neural networks there is an intrinsic lower bound on the rate of approximation, which depends on the number of neurons, when one considers any label function $F \in \mathbf{C}^2(\mathbb{R}^d)$. This limitation is removed in the two layer model, as the following theorem shows.

Theorem 2.13 (Maierov and Pinkus 1999, [16]). *There exists an activation function $\sigma \in \mathbf{C}^\infty(\mathbb{R})$ that is sigmoidal and strictly increasing, and has the following property. For any $F \in \mathbf{C}[0, 1]^d$ and $\epsilon > 0$, there exists a two-layer neural network $f(x; \theta)$ with activation function σ , with $d_1 = (2d + 1)(4d + 3)$ neurons in the first hidden layer, and with $d_2 = 4d + 3$ neurons in the second layer, for which*

$$\sup_{x \in [0, 1]^d} |F(x) - f(x; \theta)| < \epsilon.$$

This theorem is pretty extraordinary. Some remarks are in order.

Remark 2.14. The activation function σ is the same as the one in Theorem 2.11, and is thus, unfortunately, not practical.

Remark 2.15. The number of neurons is independent of the label function F (similar to previous one-layer results) and the accuracy ϵ (very different from previous one-layer results), and grows quadratically in the dimension d . As expected, the weights of the neurons depend on F and ϵ . While it is remarkable that the number of neurons does not depend on the accuracy ϵ , it is worth examining how the magnitude of the weights depends on ϵ . It turns out the size of the weights grows very fast with epsilon, and the weights become so huge that they could not hope to be stored on a computer!

Remark 2.16. The proof of Theorem 2.13 is based upon the Kolmogorov Superposition Theorem. In particular it utilizes an improved version, stated below.

Theorem 2.17 (Kolmogorov Superposition Theorem). *There exist d constants $\lambda_j > 0$, $1 \leq j \leq d$, with $\sum_{j=1}^d \lambda_j \leq 1$, and $2d + 1$ strictly increasing continuous functions $\phi_k : [0, 1] \rightarrow [0, 1]$, $1 \leq k \leq 2d + 1$, such that every $F \in \mathbf{C}[0, 1]^d$ can be represented as*

$$F(x) = F(x(1), \dots, x(d)) = \sum_{k=1}^{2d+1} G \left(\sum_{j=1}^d \lambda_j \phi_k(x(j)) \right), \quad (2.14)$$

for some $G \in \mathbf{C}[0, 1]$ depending on F .

Using the Kolmogorov Superposition Theorem we can (crudely) sketch the proof of Theorem 2.13.

Proof sketch of Theorem 2.13. Using the Kolmogorov Superposition Theorem we write $F(x)$ as in (2.14). Let σ be the same activation function as from Theorem 2.11. We first approximate G using σ . In the proof of Theorem 2.11 (which is Proposition 6.3 and Corollary 6.4 in [14]), it is shown that σ can be constructed

in such a way that for each $H \in \mathbf{C}[-1, 1]$ and $\eta > 0$, there exist constants $a_1, a_2, a_3 \in \mathbb{R}$ and an integer $m \in \mathbb{Z}$ for which

$$\forall z \in [-1, 1], \quad |H(z) - (a_1\sigma(z-3) + a_2\sigma(z+1) + a_3\sigma(z+m))| < \eta. \quad (2.15)$$

Furthermore, $\sigma(z-3)$ and $\sigma(z+1)$ are linear on $[0, 1]$. The construction of σ is accomplished by using the fact that $\mathbf{C}^\infty[-1, 1]$ is dense in $\mathbf{C}[-1, 1]$, which means there exists a countable collection of functions $\{h_k\}_{k=1}^\infty \subset \mathbf{C}^\infty[-1, 1]$ so that for each $H \in \mathbf{C}[-1, 1]$ and each η there exists $k = k(H, \eta)$ with

$$\sup_{z \in [-1, 1]} |H(z) - h_k(z)| < \eta.$$

Pinkus then cleverly constructs σ so that for each $k \geq 1$ there exists constants $a_{1,k}, a_{2,k}, a_{3,k}$ with

$$a_{1,k}\sigma(z-3) + a_{2,k}\sigma(z+1) + a_{3,k}\sigma(z+4k+1) = u_k(z).$$

while also ensuring that $\sigma(z-3)$ and $\sigma(z+1)$ are linear on $[0, 1]$ (in fact he places more restrictions on σ , but we will not need them for this discussion).

Anyway, with (2.15) in hand we can apply it to G with $\eta = \epsilon/2(2d+1)$ and restrict the domain from $[-1, 1]$ to $[0, 1]$, which gives:

$$\forall z \in [0, 1], \quad |G(z) - (a_1\sigma(z-3) + a_2\sigma(z+1) + a_3\sigma(z+m))| < \frac{\epsilon}{2(2d+1)}.$$

We now use this approximation and the Kolmogorov Superposition Theorem to obtain:

$$\begin{aligned} \forall x \in [0, 1]^d, \quad & \left| F(x) - \sum_{k=1}^{2d+1} \left[a_1\sigma \left(\sum_{j=1}^d \lambda_j \phi_k(x(j)) - 3 \right) \right. \right. \\ & \left. \left. + a_2\sigma \left(\sum_{j=1}^d \lambda_j \phi_k(x(j)) + 1 \right) + a_3\sigma \left(\sum_{j=1}^d \lambda_j \phi_k(x(j)) + m \right) \right] \right| < \frac{\epsilon}{2}. \end{aligned}$$

Recall that $\sigma(z-3)$ and $\sigma(z+1)$ are linear on $[0, 1]$, and that by the Kolmogorov Superposition Theorem $\phi_k : [0, 1] \rightarrow [0, 1]$, so we can combine the first two terms:

$$\begin{aligned} \sum_{k=1}^{2d+1} a_1 \left[\sigma \left(\sum_{j=1}^d \lambda_j \phi_k(x(j)) - 3 \right) + a_2\sigma \left(\sum_{j=1}^d \lambda_j \phi_k(x(j)) + 1 \right) \right] \\ = \sum_{k=1}^{2d+2} c_k \sigma \left(\sum_{j=1}^d \lambda_j \phi_k(x(j)) + \gamma_k \right), \end{aligned}$$

where ϕ_{2d+2} is ϕ_k for some $1 \leq k \leq 2d+1$ and $\gamma_k \in \{-3, 1\}$ for each k . We thus have:

$$\forall x \in [0, 1]^d, \quad \left| F(x) - \sum_{k=1}^{2d+2} c_k \sigma \left(\sum_{j=1}^d \lambda_j \phi_k(x(j)) + \gamma_k \right) - a_3\sigma \left(\sum_{j=1}^d \lambda_j \phi_k(x(j)) + m \right) \right| < \frac{\epsilon}{2}.$$

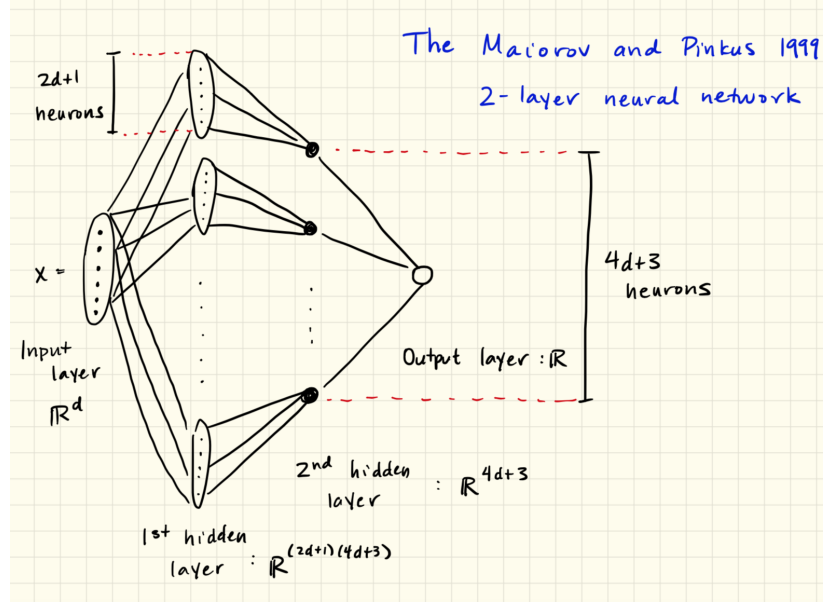


Figure 2.7: Drawing of the Maiorov and Pinkus (1999) two-layer neural network that achieves the result of Theorem 2.13.

The proof proceeds by applying (2.15) to each $H = \phi_k$ for η small enough, and again using the fact that $\sigma(z - 3)$ and $\sigma(z + 1)$ are linear on $[0, 1]$. After combining terms, the result is obtained. For more details on the proof, see [14, Theorem 7.2]. \square

Remark 2.18. The proof of Theorem 2.13 gives the structure of this two-layer network, and in fact the number of connections between the first hidden layer and the second hidden layer is quite small. In particular, $f(x; \theta)$ can be written as:

$$f(x; \theta) = \sum_{\ell=1}^{4d+3} \alpha(\ell) \sigma \left(\sum_{k=1}^{2d+1} w_{2,\ell}(k) \sigma(\langle x, w_{1,k,\ell} \rangle + b_1(k, \ell)) + b_2(\ell) \right),$$

where $w_{1,k,\ell} \in \mathbb{R}^d$ for $1 \leq k \leq 2d + 1$ and $1 \leq \ell \leq 4d + 3$, $b_1 \in \mathbb{R}^{(2d+1) \times (4d+3)}$, $w_{2,\ell} \in \mathbb{R}^{2d+1}$, and $b_2 \in \mathbb{R}^{4d+3}$. The network is illustrated in Figure 2.7. In [14], Pinkus says the network has $2d + 1$ units in the first layer and $4d + 3$ units in the second layer, but by most definitions of neurons, as well as our own, it has $(2d + 1)(4d + 3)$ neurons in the first layer and $4d + 3$ neurons in the second layer.

2.4 Modern theory for ANNs

We now jump ahead ~ 15 years and consider more recent results on the theory of artificial neural networks.

2.4.1 More differences between one-layer and two-layer ANNs

We begin with a result from 2016 [17] that reinforces the analysis of Maiorov and Pinkus contained in Theorems 2.12 and 2.13. Interestingly, [17] does not cite [14], but we will see the results are of a similar flavor.

Namely, [17] obtains a result that says neural networks with two hidden layers are fundamentally different than those with one hidden layer. In particular, Eldan and Shamir prove there is a simple function on \mathbb{R}^d that can be well approximated by a “small” two-hidden-layer neural network that cannot be approximated by any one-hidden layer neural network, to more than a certain constant accuracy, unless the width of this one-layer network is exponential in the dimension. This function, essentially, gives a concrete example for Theorem 2.12. Let us now state the results in more detail.

By a one-layer network with W neurons (previously we used $m = W$ as the number of neurons, but [17] uses W to indicate the “width” of the network which will not mean the number of neurons, at least as we define them, when we get to two layer networks), the authors mean the same functions as we have discussed previously, namely:

$$f(x; \theta) = \sum_{k=1}^W \alpha(k) \sigma(\langle x, w_k \rangle + b(k)) \quad (\text{one hidden layer of width } W). \quad (2.16)$$

By a two-hidden layer network of width W , Eldan and Shamir in fact mean a function like the one used by Maiorov and Pinkus, which is a type of two layer network with a special structure:

$$f(x; \theta) = \sum_{\ell=1}^W \alpha(\ell) \sigma \left(\sum_{k=1}^W w_{2,\ell}(k) \sigma(\langle x, w_{1,k,\ell} \rangle + b_1(k, \ell)) + b_2(\ell) \right) \quad (2.17)$$

(two hidden layers of width W).

Note, this network has a very similar structure as the one depicted in Figure 2.7, and by our naming convention we would say the first layer has W^2 neurons and the second layer has W neurons. Eldan and Shamir say it has “width W .” There are two assumptions on the activation function σ .

Assumption 2.19 (one-dimensional universality). There exists a constant $c_\sigma \geq 1$ (depending only on σ) such that the following holds: For any Lipschitz function $G : \mathbb{R} \rightarrow \mathbb{R}$ with Lipschitz constant L , i.e., $|G(z) - G(z')| \leq L|z - z'|$, which is constant outside the interval $[-R, R]$, and for any $\epsilon > 0$, there exists $w \in \mathbb{R}^m$, $b \in \mathbb{R}^m$, $\alpha \in \mathbb{R}^m$ and $\beta \in \mathbb{R}$ with

$$W \leq c_\sigma R L \epsilon^{-1},$$

such that the function

$$g(z) = \beta + \sum_{k=1}^W \alpha(k) \sigma(w(k)z - b(k)),$$

satisfies

$$\sup_{z \in \mathbb{R}} |G(z) - g(z)| \leq \delta.$$

Remark 2.20. Note this remark is somewhat similar to Proposition 2.8 in that it reduces the main requirements on σ to the one-dimensional setting.

Assumption 2.21. The activation function σ satisfies

$$|\sigma(z)| \leq C(1 + |z|^\alpha),$$

for all $z \in \mathbb{R}$ and for some constants $C, \alpha > 0$.

Eldan and Shamir show these assumptions are satisfied by many standard activation functions, including ReLU and sigmoidal activation functions. Using these assumptions, they prove the following theorem.

Theorem 2.22 (Eldan & Shamir 2016, [17]). *Suppose σ satisfies Assumptions 2.19 and 2.21. Then there exist universal constants $c, C > 0$ such that the following holds: For every dimension $d > C$, there is a probability measure μ on \mathbb{R}^d and a function $F_0 : \mathbb{R}^d \rightarrow \mathbb{R}$ with the following properties:*

- $\|F_0\|_\infty \leq 2$, $\text{supp}(F_0) \subset \{x \in \mathbb{R}^d : \|x\| \leq C\sqrt{d}\}$, and $F_0(x)$ can be written as a two-hidden layer neural network of the form (2.17) of width

$$W_{2\text{layer}} = Cc_\sigma d^{19/4}.$$

- Every one hidden layer network $f(x; \theta)$ of the form (2.16) of width at most

$$W_{1\text{layer}} \leq ce^{cd},$$

satisfies

$$\mathbb{E}_X[(F_0(X) - f(X; \theta))^2] = \int_{\mathbb{R}^d} (F_0(x) - f(x; \theta))^2 d\mu(x) \geq c. \quad (2.18)$$

Note that if the probability measure μ admits a probability density function $p_X(x)$, then $d\mu(x) = p_X(x)dx$ and (2.18) is the same squared loss as we have seen previously.

Theorem 2.22 says there are label functions F that can be perfectly represented by neural networks with two hidden layers and with a width that is only polynomial in the dimension d , but cannot even be well approximated by a single hidden layer neural network unless the width of that single layer network is exponential in the dimension d . Thus by increasing the depth of the

network by one (i.e., from one hidden layer to two hidden layers), the neural network becomes significantly more efficient in its ability to express certain label functions.

Furthermore, the function $F_0(x)$ is not that complicated. Indeed, it is, roughly speaking, a radial function, meaning that

$$F_0(x) \approx \tilde{F}(\|x\|_2^2),$$

where $\tilde{F} : \mathbb{R} \rightarrow \mathbb{R}$. Within the two hidden layer network, the first layer approximates the map $x \mapsto \|x\|_2^2$ and the second layer approximates the function \tilde{F} , which is relatively speaking straightforward and not unlike how the proof of Theorem 2.13 is carried out using the Kolmogorov Superposition Theorem. On the other hand, doing all of this in one layer is rather complicated, which results in the exponential growth of the width of the one-layer neural network.

2.4.2 Compositional functions

The proof of Theorem 2.13 showing that two-layer networks require far fewer neurons than one-layer networks, and the example in Section 2.4.1, both leverage compositional structure to obtain their results. In fact, recent work contained in [18] studies the approximation theoretic capabilities of shallow networks (one-layer networks) versus deep networks for the class of smooth, compositional functions. A prototypical example is the following:

$$\begin{aligned} x \in \mathbb{R}^8, \quad F(x) &= F(x(1), \dots, x(8)) \\ &= H_3(H_{21}(H_{11}(x(1), x(2)), H_{12}(x(3), x(4))), \\ &\quad H_{22}(H_{13}(x(5), x(6)), H_{14}(x(7), x(8)))), \end{aligned} \quad (2.19)$$

where $F : \mathbb{R}^8 \rightarrow \mathbb{R}$, but each function $H_\lambda : \mathbb{R}^2 \rightarrow \mathbb{R}$ for $\lambda \in \{11, 12, 13, 14, 21, 22, 3\}$. This function $F(x)$ can be represented by a binary tree graph, as in Figure 2.8.

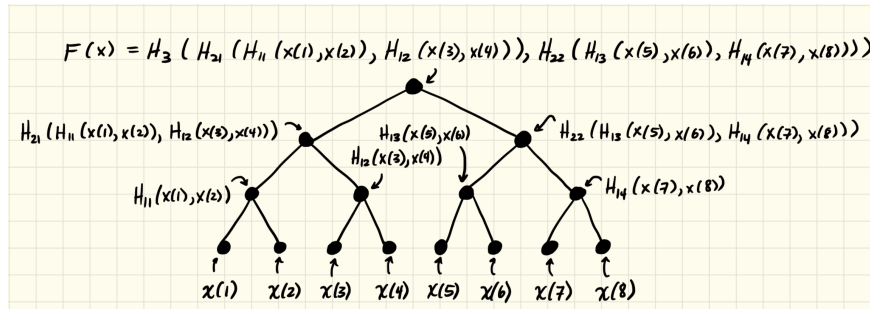


Figure 2.8: Illustration of the compositional function in (2.19) as a binary tree graph.

Compositional functions such as (2.19) are “local,” meaning that $F(x)$ consists of a compositional hierarchy of functions that only depend on at most q

variables; in the case of the example (2.19), $q = 2$. While both one-layer networks and deep networks are universal function approximators, only deep networks can take advantage of the compositional structure of functions (if the function has this structure). We saw this phenomenon to some extent in the result of Maierov and Pinkus, Theorem 2.13, which used the Kolmogorov Superposition Theorem to write any $F \in \mathbf{C}[0, 1]^d$ as a compositional function. Here we will explicitly assume the function has such a structure and quantify what we mean by the statement that deep networks can take advantage of this structure.

To that end let us again consider the space of functions $\mathbf{C}^s[-1, 1]^d$. To be as precise as possible, we note that [18] uses the a different norm on $\mathbf{C}^s[-1, 1]^d$, which is different than the norm we introduced earlier, but which is equivalent to (2.13). Anyway, here is the new norm:

$$\|F\|_{\mathbf{C}^s[-1, 1]^d} = \sum_{\|\beta\|_1 \leq s} \|\partial^\beta F\|_{\mathbf{L}^\infty[-1, 1]^d},$$

where we remind the reader that

$$\|F\|_{\mathbf{L}^\infty[-1, 1]^d} = \sup_{x \in [-1, 1]^d} |F(x)|.$$

Now let us define

$$\begin{aligned} \mathbf{C}_2^s[-1, 1]^d = \{ & \text{all compositional functions } F(x) \text{ defined on } [-1, 1]^d \\ & \text{that have a binary tree architecture as in Figure 2.8,} \\ & \text{and for which the constituent functions } H_\lambda \in \mathbf{C}^s[-1, 1]^2 \}. \end{aligned}$$

We recall the space $\mathcal{M}_N(\sigma) = \mathcal{M}_{N,d}(\sigma)$ of one layer networks with N neurons and that take as input vectors $x \in \mathbb{R}^d$, which consists of functions $f(x; \theta)$ of the form:

$$f(x; \theta) = \sum_{k=1}^N \alpha(k) \sigma(\langle x, w_k \rangle + b(k)).$$

We remark that the number of trainable parameters in this network is:

$$\begin{aligned} \# \text{ of trainable parameters} &= \text{weight vectors } w_k + \text{biases } b(k) + \text{final weights } \alpha(k) \\ &= dN + N + N = (d + 2)N. \end{aligned}$$

As a parallel to the space of compositional functions $\mathbf{C}_2^s[-1, 1]^d$ defined above, we now define a corresponding class of deep networks $\mathcal{D}_{N,2}(\sigma)$. The space $\mathcal{D}_{N,2}(\sigma)$ will consist of all deep networks that use the activation function σ and themselves have a binary tree architecture. However, the network having a binary tree architecture does not mean the architecture applies at the level of an artificial neuron; let us explain. Recall that an artificial neuron is the function:

$$\eta(x) = \sigma(\langle x, w \rangle + b).$$

Define a node $\bar{\eta}(x)$ as, essentially, a one hidden layer neural network that has been embedded in a larger network:

$$\bar{\eta}(x) = \sum_{k=1}^N \eta_k(x) = \sum_{k=1}^N \alpha(k) \sigma(\langle x, w_k \rangle + b(k)).$$

A deep network $f \in \mathcal{D}_{N,2}$ has the binary tree architecture with respect to its *nodes*, meaning that it consists of many nodes composed together according to some binary tree, and each node $\bar{\eta}(z)$ takes as input a two-dimensional vector $z \in \mathbb{R}^2$. The sub-index N means that each node is in $\mathcal{M}_{m,2}(\sigma)$ with

$$m = N/|V|, \quad V = \text{non-leaf vertices of the binary tree}.$$

That is, $\bar{\eta} \in \mathcal{M}_{m,2}(\sigma)$ and each such node has $m = N/|V|$ neurons; figure 2.9 illustrates the idea. The following proposition shows the number of trainable parameters of a network $f \in \mathcal{D}_{N,2}$ is $4N$.

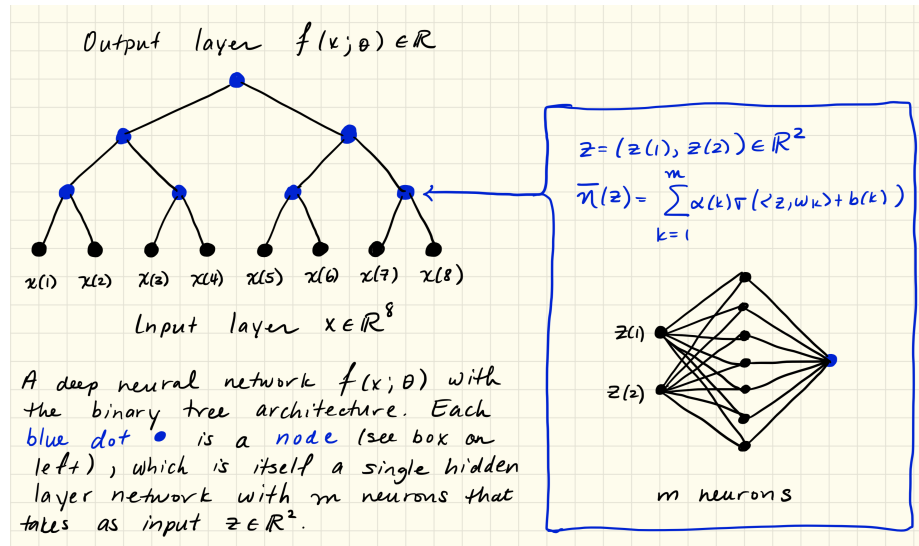


Figure 2.9: A deep neural network $f \in \mathcal{D}_{N,2}$ with the binary tree structure on its nodes $\bar{\eta}(z)$, each of which takes an input $z \in \mathbb{R}^2$ and outputs a scalar, after passing z through m neurons.

Proposition 2.23. Let $d = 2^J$ for some $J \geq 0$. Then the number of trainable parameters of a network $f \in \mathcal{D}_{N,2}(\sigma)$ is $4N$.

Proof. If $d = 2^J$ then the number of leaves in the binary tree is $d = 2^J$ and the number of non-leaf nodes is (using a formula for geometric series):

$$\sum_{j=0}^{J-1} 2^j = \frac{1-2^J}{1-2} = 2^J - 1 = d - 1.$$

Thus each of these $d - 1$ nodes has m neurons with

$$m = N/|V| = N/(d - 1) \implies d - 1 = N/m.$$

Since each node $\bar{\eta} \in \mathcal{M}_{m,2}(\sigma)$ it has $(2 + 2)m = 4m$ trainable parameters. With $d - 1$ such nodes in f , the number of trainable parameters is:

$$(d - 1)4m = 4 \frac{N}{m} m = 4N.$$

□

Therefore networks in $\mathcal{M}_{N,d}(\sigma)$ have $(d + 2)N$ trainable parameters and networks in $\mathcal{D}_{N,2}(\sigma)$ have $4N$ trainable parameters. If the dimension d is fixed, then both styles of networks have $O(N)$ trainable parameters.

Now let us compare the performance of shallow networks from $\mathcal{M}_{N,d}(\sigma)$ to deep networks from $\mathcal{D}_{N,2}(\sigma)$. The main point that these results will emphasize is the following. The space $\mathbf{C}_2^s[-1, 1]^d \subseteq \mathbf{C}^s[-1, 1]^d$, which means one layer neural networks can approximate any function $F \in \mathbf{C}_2^s[-1, 1]^d$. However, they will not take advantage of the compositional structure of F . On the other hand, a network $f \in \mathcal{D}_{N,2}(\sigma)$ with a binary tree structure on its nodes that matches (or contains as a subgraph) the compositional binary tree structure of F can take advantage of the structure of F and circumvent the curse of dimensionality. Let us now describe the results in more detail.

In [19] the following one-layer theorem is proved, which is simliar to Theorem 2.11, but gives the rate of convergence for a large class of activation functions.

Theorem 2.24 (Mhaskar 1996, [19]). *Let $\sigma \in \mathbf{C}^\infty(\mathbb{R})$ not be a polynomial. Then for any $F \in \mathbf{C}^s[-1, 1]^d$ with $\|F\|_{\mathbf{C}^s[-1, 1]^d} \leq 1$,*

$$\inf_{f \in \mathcal{M}_N(\sigma)} \|F - f\|_{\mathbf{L}^\infty[-1, 1]^d} \leq CN^{-s/d}.$$

Stated another way, in order guarantee

$$\inf_{f \in \mathcal{M}_N(\sigma)} \|F - f\|_{\mathbf{L}^\infty[-1, 1]^d} \leq \epsilon$$

for an arbitrary $F \in \mathbf{C}^s[-1, 1]^d$ with $\|F\|_{\mathbf{C}^s[-1, 1]^d} \leq 1$, one must take

$$N = O(\epsilon^{-d/s})$$

neurons in the one hidden layer of f .

Note that since Theorem 2.24 applies to $\mathbf{C}^s[-1, 1]^d$ it also applies to $\mathbf{C}_2^s[-1, 1]^d$, and as examples such as the one described in Section 2.4.1 show, the result cannot be improved. Now let us restrict attention to $\mathbf{C}_2^s[-1, 1]^d$ and consider the class $\mathcal{D}_{N,2}(\sigma)$ of deep networks with binary tree nodal structure.

Theorem 2.25 (Poggio, et al., [18]). Let $\sigma \in \mathbf{C}^\infty(\mathbb{R})$ not be a polynomial. Let $F \in \mathbf{C}_2^s[-1, 1]^d$ and let $\{H_\lambda \in \mathbf{C}^s[-1, 1]^2\}_\lambda$ be the constituent functions of F , each satisfying $\|H_\lambda\|_{\mathbf{C}^s[-1, 1]^2} \leq 1$. Then,

$$\inf_{f \in \mathcal{D}_{N,2}(\sigma)} \|F - f\|_{\mathbf{L}^\infty[-1, 1]^d} \leq C(d, s) N^{-s/2}.$$

Stated another way, in order to guarantee

$$\inf_{f \in \mathcal{D}_{N,2}(\sigma)} \|F - f\|_{\mathbf{L}^\infty[-1, 1]^d} \leq \epsilon$$

for an arbitrary $F \in \mathbf{C}_2^s[-1, 1]^d$ with $\|H_\lambda\|_{\mathbf{C}^s[-1, 1]^2} \leq 1$, one must take

$$N = \tilde{C}(d, s) \epsilon^{-2/s}.$$

Proof. Let $d = 2^J$. Recall that each node of the network $f \in \mathcal{D}_{N,2}$ has $m = N/|V| = N/(d-1)$ neurons inside the node. Let $H_\lambda \in \mathbf{C}^s[-1, 1]^2$ be one of the constituent functions of F . We can apply Theorem 2.24 to conclude that there is a node $\bar{\eta}_\lambda \in \mathcal{M}_m(\sigma)$ such that

$$\|H_\lambda - \bar{\eta}_\lambda\|_{\mathbf{L}^\infty[-1, 1]^2} \leq C m^{-s/2}. \quad (2.20)$$

That works for the individual functions making up F , but we have to check that when we compose different H_λ functions, the error does not get too large. So let us consider $d = 4$, which means the label function is of the form

$$F = H_1(H_{11}, H_{12}),$$

and let $\bar{\eta}_1, \bar{\eta}_{11}$, and $\bar{\eta}_{12}$ be the nodes that approximate H_1, H_{11} , and H_{12} , respectively. Then:

$$\begin{aligned} & \|H_1(H_{11}, H_{12}) - \bar{\eta}_1(\bar{\eta}_{11}, \bar{\eta}_{12})\|_{\mathbf{L}^\infty[-1, 1]^4} \\ &= \|H_1(H_{11}, H_{12}) - H_1(\bar{\eta}_{11}, \bar{\eta}_{12}) + H_1(\bar{\eta}_{11}, \bar{\eta}_{12}) - \bar{\eta}_1(\bar{\eta}_{11}, \bar{\eta}_{12})\|_{\mathbf{L}^\infty[-1, 1]^4} \\ &\leq \|H_1(H_{11}, H_{12}) - H_1(\bar{\eta}_{11}, \bar{\eta}_{12})\|_{\mathbf{L}^\infty[-1, 1]^4} + \|H_1(\bar{\eta}_{11}, \bar{\eta}_{12}) - \bar{\eta}_1(\bar{\eta}_{11}, \bar{\eta}_{12})\|_{\mathbf{L}^\infty[-1, 1]^4} \end{aligned} \quad (2.21)$$

For the second term we can apply (2.20) nearly directly. Write

$$z = (z(1), z(2), z(3), z(4)) \in [-1, 1]^4$$

as $z = (z_1, z_2)$ with $z_1 = (z(1), z(2))$ and $z_2 = (z(3), z(4))$. We have:

$$\begin{aligned} & \|H_1(\bar{\eta}_{11}, \bar{\eta}_{12}) - \bar{\eta}_1(\bar{\eta}_{11}, \bar{\eta}_{12})\|_{\mathbf{L}^\infty[-1, 1]^4} \\ &= \sup_{z \in [-1, 1]^4} |H_1(\bar{\eta}_{11}(z_1), \bar{\eta}_{12}(z_2)) - \bar{\eta}_1(\bar{\eta}_{11}(z_1), \bar{\eta}_{12}(z_2))| \\ &= \sup_{u \in [-1, 1]^2} |H_1(u(1), u(2)) - \bar{\eta}_1(u(1), u(2))|, \quad [u(1) = \bar{\eta}_{11}(z_1), u(2) = \bar{\eta}_{12}(z_2)] \\ &= \|H_1 - \bar{\eta}_1\|_{\mathbf{L}^\infty[-1, 1]^2} \\ &\leq C m^{-s/2}. \end{aligned} \quad (2.22)$$

For the first term, let $u, \bar{u} \in \mathbb{R}^2$. We first observe:

$$|H_1(u) - H_1(\bar{u})| \leq \sup_{v \in \mathbb{R}^2} \|\nabla H_1(v)\|_2 \|u - \bar{u}\|_2. \quad (2.23)$$

We also have:

$$\begin{aligned} \sup_{v \in [-1,1]^2} \|\nabla H_1(v)\|_2 &\leq \sup_{v \in [-1,1]^2} \|\nabla H_1(v)\|_1 \\ &= \sup_{v \in [-1,1]^2} [|\partial_1 H_1(v)| + |\partial_2 H_1(v)|] \\ &= \|\partial_1 H_1\| + \|\partial_2 H_1\|_{\mathbf{L}^\infty[-1,1]^2} \\ &\leq \|\partial_1 H_1\|_{\mathbf{L}^\infty[-1,1]^2} + \|\partial_2 H_1\|_{\mathbf{L}^\infty[-1,1]^2} \\ &\leq \|H_1\|_{\mathbf{C}^s[-1,1]^2} \\ &\leq 1. \end{aligned}$$

Therefore, combining with (2.23) we have

$$|H_1(u) - H_1(\bar{u})| \leq \|u - \bar{u}\|_2.$$

Now plug in

$$\begin{aligned} u &= (H_{11}(z_1), H_{12}(z_2)) \\ \bar{u} &= (\bar{\eta}_{11}(z_1), \bar{\eta}_{12}(z_2)). \end{aligned}$$

We get:

$$\begin{aligned} &|H_1(H_{11}(z_1), H_{12}(z_2)) - H_1(\bar{\eta}_{11}(z_1), \bar{\eta}_{12}(z_2))|^2 \\ &\leq |H_{11}(z_1) - \bar{\eta}_{11}(z_1)|^2 + |H_{12}(z_2) - \bar{\eta}_{12}(z_2)|^2 \\ &\leq \|H_{11} - \bar{\eta}_{11}\|_{\mathbf{L}^\infty[-1,1]^2}^2 + \|H_{12} - \bar{\eta}_{12}\|_{\mathbf{L}^\infty[-1,1]^2}^2 \\ &\leq 2Cm^{-s}. \end{aligned} \quad (2.24)$$

Therefore:

$$\|H_1(H_{11}, H_{12}) - H_1(\bar{\eta}_{11}, \bar{\eta}_{12})\|_{\mathbf{L}^\infty[-1,1]^4} \leq \sqrt{2}Cm^{-s/2}.$$

Combining (2.21), (2.22), and (2.24) we get:

$$\| \underbrace{H_1(H_{11}, H_{12})}_F - \underbrace{\bar{\eta}_1(\bar{\eta}_{11}, \bar{\eta}_{12})}_f \|_{\mathbf{L}^\infty[-1,1]^4} \leq (1 + \sqrt{2})Cm^{-s/2}.$$

If the $d > 4$ we can recursively apply the bounds computed above. The number of times we will apply these bound will depend only on d . On the right hand side, they are all of the form $Cm^{-s/2}$, and so we will get (for general $d = 2^J$)

$$\|F - f\|_{\mathbf{L}^\infty[-1,1]^d} \leq C(d)m^{-s/2}.$$

Now recall that $m = N/(d-1)$. Thus in fact we have:

$$\|F - f\|_{\mathbf{L}^\infty[-1,1]^d} \leq C(d) \left(\frac{N}{d-1} \right)^{-s/2} = C(d,s) N^{-s/2}.$$

□

Remark 2.26. Remember, N is a proxy for the complexity of the network in terms of the number of trainable parameters, which is $O(N)$ for both the shallow and deep networks. Thus the deep networks are much more efficient learners than shallow networks for compositional functions.

Remark 2.27. Suppose F 2-compositional but we guess a network in $\mathcal{D}_{N,q}$, that is to say, it aggregates information q variables at a time as opposed to 2 variables at a time. Another scenario is if F is q -compositional and we use a network from $\mathcal{D}_{N,q}$ (which matches the compositional structure of F). In either case the learning rate is:

$$\inf_{f \in \mathcal{D}_{N,q}(\sigma)} \|F - f\|_{\mathbf{L}^\infty[-1,1]^d} \leq C(d,s,q) N^{-s/q}.$$

Remark 2.28. *Chapter continued in handwritten notes.*

Chapter 3

Convolutional neural networks

Remark 3.1. *See the handwritten notes for this chapter.*

Bibliography

- [1] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. In *Proceedings of the International Conference on Learning Representations*, 2015.
- [2] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *Conference on Computer Vision and Pattern Recognition*, 2009.
- [3] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, Yutian Chen, Timothy Lillicrap, Fan Hui, Laurent Sifre, George van den Driessche, Thore Graepel, and Demis Hassabis. Mastering the game of Go without human knowledge. *Nature*, 550:354—359, 2017.
- [4] Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems 25*, pages 1097–1105. 2012.
- [5] Larry Greenemeier. AI versus AI: Self-taught AlphaGo Zero vanquishes its predecessor. *Scientific American*, October 18, 2017.
- [6] Pankaj Mehta, Marin Bukov, Ching-Hao Wang, Alexandre G.R. Day, Clint Richardson, Charles K. Fisher, and David J. Schwab. A high-bias, low-variance introduction to Machine Learning for physicists. arXiv:1803.08823, 2018.
- [7] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning*. Springer-Verlag New York, 2nd edition, 2009.
- [8] J. Hartlap, P. Simon, and P. Schneider. Why your model parameter confidences might be too optimistic - unbiased estimation of the inverse covariance matrix. *Astronomy and Astrophysics*, 464(1):399–404, 2007.
- [9] Bernhard Schölkopf and Alexander J. Smola. *Learning with Kernels: Support Vector Machines, Regularization, Optimization, and Beyond*. Adaptive Computation and Machine Learning. The MIT Press, 2002.

- [10] Stephen Boyd and Lieven Vandenberghe. *Convex Optimization*. Cambridge University Press, 2004.
- [11] Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *3rd International Conference for Learning Representations*, San Diego, CA, USA, 2015.
- [12] George Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals, and Systems*, 2:303–314, 1989.
- [13] Kurt Hornik. Approximation capabilities of multilayer feedforward networks. *Neural Networks*, 4:251–257, 1991.
- [14] Allan Pinkus. Approximation theory of the mlp model in neural networks. *Acta Numerica*, 8:143–195, 1999.
- [15] Vitaly E. Maiorov. On best approximation by ridge functions. *Journal of Approximation Theory*, 99:68–94, 1999.
- [16] Vitaly E. Maiorov and Allan Pinkus. Lower bounds for approximation by mlp neural networks. *Neurocomputing*, 25:81–91, 1999.
- [17] Ronen Eldan and Ohad Shamir. The power of depth for feedforward neural networks. In *29th Annual Conference on Learning Theory*, volume 49 of *Proceedings of Machine Learning Research*, pages 907–940. PMLR, 2016.
- [18] Tomaso Poggio, Hrushikesh Mhaskar, Lorenzo Rosasco, Brando Miranda, and Qianli Liao. Why and when can deep-but not shallow-networks avoid the curse of dimensionality: A review. *International Journal of Automation and Computing*, 14(5):503–519, 2017.
- [19] Hrushikesh Mhaskar. Neural networks for optimal approximation of smooth and analytic functions. *Neural Computation*, 8:164–177, 1996.

Shift gears a bit and study results that are more directly applicable to classification

First paper: Montúfar, Pascanu, Cho, Bengio, 2014

"On the number of linear regions in deep neural networks"

See also: Pascanu, Montúfar, Bengio, 2013

"On the # of response regions of deep feed forward networks w/ piecewise linear activations"
and more recent papers!

Idea: Count the # of piecewise linear parts of a function $f(x; \theta)$ that a ReLU network can implement

Summary of result: Deep ReLU networks can implement functions $f(x; \theta)$ with exponentially more piecewise linear regions than shallow networks w/ the same # of hidden neurons.

Now show Figure 1 from the paper

Types of networks we will study: $f(\cdot; \theta) : \mathbb{R}^d \rightarrow \mathbb{R}^{\text{out}}$

(i) Shallow, one layer:

$$f(x; \theta) = f_{\text{out}} \circ \sigma \circ A(x), \quad A(x) = Wx + b$$

where: $W \in \mathbb{R}^{d_1 \times d}$, i.e., d_1 neurons
 $b \in \mathbb{R}^{d_1}$

$$\sigma(z) = \max(0, z) = \text{ReLU}(z)$$

$$f_{\text{out}} : \mathbb{R}^{d_1} \rightarrow \mathbb{R}^{\text{out}}, \text{ e.g.}$$

- Regression, $f_{\text{out}} : \mathbb{R}^{d_1} \rightarrow \mathbb{R}$ and $f_{\text{out}}(z) = \langle z, \alpha \rangle$

- Classification, $f_{\text{out}} : \mathbb{R}^{d_1} \rightarrow \mathbb{R}^{\# \text{ classes}}$ and $f_{\text{out}} = \text{softmax}$

(ii) Deep, L layers:

$$f(x; \theta) = f_{\text{out}} \circ \sigma \circ A_L \circ \dots \circ \sigma \circ A_1(x)$$

where

$$A_\ell(x) = W_\ell x + b_\ell, \quad W_\ell \in \mathbb{R}^{d_\ell \times d_{\ell-1}}, \quad d_0 = d$$
$$b_\ell \in \mathbb{R}^{d_\ell}$$

$$f_{\text{out}} : \mathbb{R}^{d_L} \rightarrow \mathbb{R}^{\text{out}}$$

Rest same as shallow network.

Def: Let $f: \mathbb{R}^d \rightarrow \mathbb{R}^m$ be a piecewise linear function. A linear region of f is a maximal connected subset of the input space \mathbb{R}^d on which f is linear.

Note: For ReLU networks $f(x; \theta)$, the dimension of each linear region is d .

Results for shallow networks

Let $f(x; \theta)$ be a shallow ^{ReLU} network w/ one hidden layer and d_1 neurons. Then the maximum # of linear regions of $f(x; \theta)$ is

$$\text{max \# of linear regions} = \sum_{i=0}^d \binom{d_1}{i}$$

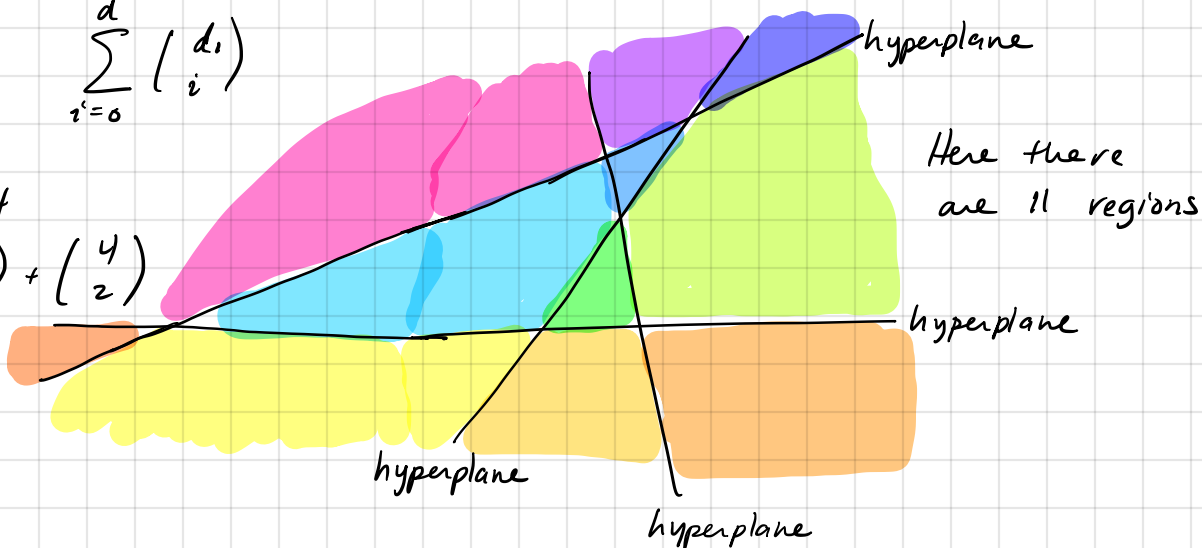
Idea of proof: Suppose you have d_1 hyperplanes in \mathbb{R}^d . The maximum # of regions these hyperplanes can divide \mathbb{R}^d into is

$$\sum_{i=0}^d \binom{d_1}{i}$$

In this example:

$$d=2, d_1=4$$

$$\begin{aligned} \text{max \#} &= \binom{4}{0} + \binom{4}{1} + \binom{4}{2} \\ &= 1 + 4 + 6 \\ &= 11 \end{aligned}$$



Results for deep networks

Main result: Let $f(x; \theta)$ be deep ReLU network w/ L hidden layers and $d_l \geq d$, $1 \leq l \leq L$, neurons in each layer. Then the maximal # of linear regions in $f(x; \theta)$ is at least:

$$\text{max \# of linear regions} \geq \left(\prod_{l=1}^{L-1} \left\lfloor \frac{d_l}{d} \right\rfloor^d \right) \sum_{i=0}^d \binom{d_L}{i} \quad (*)$$

Come back to this at the end

This part just follows from one layer analysis

Asymptotic comparison of shallow to deep networks

Let the input dimension d be fixed.

(i) For the shallow network, give it $d_1 = Ln$ neurons

(ii) For the deep network w/ L layers, let each layer have $d_l = n$ neurons

Therefore both networks have the same # of neurons.

Using the shallow network result, we see:

$$\max_{\text{linear}} \# \text{ regions shallow network} = O(L^d n^d) \leftarrow \begin{array}{l} \text{bounded from above} \\ \text{asymptotically} \\ \text{as } L, n \rightarrow \infty \end{array}$$

polynomial in n
polynomial in L

Using the deep network result, we see:

$$\max_{\text{linear}} \# \text{ regions deep network} = \Omega\left(\left(\frac{n}{d}\right)^{(L-1)d} n^d\right) \leftarrow \begin{array}{l} \text{bounded from} \\ \text{below asymptotically} \\ \text{as } L, n \rightarrow \infty \end{array}$$

polynomial in n (same as shallow)
exponential in L (larger than shallow!)

Back to (*)

$$(*) : \max \# \text{ linear regions for deep network} \geq \underbrace{\left(\prod_{l=1}^{L-1} \left\lfloor \frac{d_l}{d} \right\rfloor^d \right)}_{(**)} \sum_{i=0}^d \binom{d}{i}$$

Idea behind (**):

(1) Each layer can divide a single linear region into

$\left\lfloor \frac{d_l}{d} \right\rfloor^d$ linear regions

(2) So the first layer can create $\left\lfloor \frac{d_1}{d} \right\rfloor^d$ linear regions

(3) The 2nd layer can create $\left\lfloor \frac{d_2}{d} \right\rfloor^d$ linear regions from

each of the linear regions created by the 1st layer.

Thus the total # of linear regions is:

$$\left\lfloor \frac{d_1}{d} \right\rfloor^d \cdot \left\lfloor \frac{d_2}{d} \right\rfloor^d$$

(4) Continue for the first $L-1$ layers, you get (**).

Now we are going to study the paper:

Petersen & Voigtlaender, 2018

"Optimal approximation of piecewise smooth functions using deep ReLU networks"

Idea: Study piecewise constant (piecewise smooth) label functions, e.g.,

$$F(x) = \sum_{m=1}^{M-1} m \mathbb{I}_{K_m}(x), \quad \mathbb{I}_{K_m}(x) = \begin{cases} 1, & x \in K_m \\ 0, & x \notin K_m \end{cases}$$

which models a classification problem with M classes.

Throughout the data (test) space will be

$$\mathcal{X} = \left[-\frac{1}{2}, \frac{1}{2}\right]^d \subset \mathbb{R}^d$$

with $K_m \subseteq \mathcal{X}$ and $K_\ell \cap K_m = \emptyset$ for all $\ell \neq m$

Recall we considered a similar model in Cybenk (1989) as well.

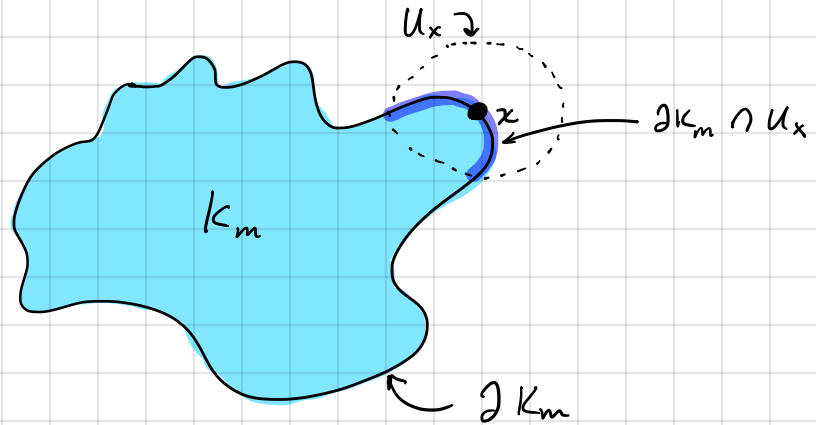
We will assume $\partial K_m \in C^\alpha$ is α -smooth



boundary of K_m

Note: $\partial K \in C^\alpha$ means the following: For any point $x \in \partial K$, there is an open neighbourhood around x , say U_x , for which $\partial K \cap U_x$ is the graph of a function $\phi \in C^\alpha(\mathbb{R}^{d-1})$ on some open subset $V \subset \mathbb{R}^{d-1}$, that is:

$$\partial K \cap U_x = \{(z, \phi(z)) : z \in V\} \quad (\text{up to a change of coordinates})$$



Summary of results:

- (i) The # of layers depends on d and α (not the # of neurons)
- (ii) The # of nonzero weights in the neurons depends on the desired accuracy ε and d and α
- (iii) The results are sharp if one limits the magnitude of the weights (recall Pintos 1999, weights really large)
- (iv) If the label function F has a suitable low dimensional structure, the network can take advantage of it (not unlike the results of Poggio, et al. 2017)

Now let us be more precise.

Label function $F: [-\frac{1}{2}, \frac{1}{2}]^d \rightarrow \mathbb{R}$ of the form:

$$F(x) = \sum_{m=1}^{M-1} a_m \underbrace{\mathbb{1}_{K_m}}_{\substack{\text{Can be generalized to smooth functions} \\ \mathbb{1}_{K_m} \in C^\alpha}}(x), \quad a_m \in \mathbb{R}, \quad K_m \subseteq [-\frac{1}{2}, \frac{1}{2}]^d$$

Neural network:

$$f(x; \theta) = A_L \circ \tau \circ A_{L-1} \circ \dots \circ \tau \circ A_1(x)$$

$$A_\ell(x) = W_\ell x + b_\ell, \quad W_\ell \in \mathbb{R}^{d_\ell \times d_{\ell-1}}, \quad b_\ell \in \mathbb{R}^{d_\ell}, \quad d_0 = d$$

$$\tau(z) = \max(0, z) = \text{ReLU}(z)$$

$$\# \text{ of neurons} = \sum_{\ell=1}^L d_\ell$$

$$\# \text{ of nonzero weights} = \|\theta\|_0 = \sum_{\ell=1}^L (\|W_\ell\|_0 + \|b_\ell\|_0)$$

$$\text{where } \|z\|_0 = \# \{z(i) \neq 0\}$$

Furthermore, the weights are bounded and quantized. This means they can be stored on a computer. More precisely, the neural network $f(x; \theta)$ has (t, ε) -quantized weights if all entries of A_ℓ and b_ℓ , for $1 \leq \ell \leq L$, are elements of

$$\underbrace{[-\varepsilon^{-t}, \varepsilon^{-t}]}_{\text{bounded}} \cap \underbrace{2^{-t \lceil \log_2(1/\varepsilon) \rceil} \mathbb{Z}}_{\text{quantized}}$$

$$\text{Note: } c \cdot \mathbb{Z} = \{cn : n \in \mathbb{Z}\}$$

Error measured in root mean squared loss:

$$\|F - f\|_2 = \|F - f\|_{L^2([-\frac{1}{2}, \frac{1}{2}]^d)} = \left[\int_{[-\frac{1}{2}, \frac{1}{2}]^d} |F(x) - f(x; \theta)|^2 dx \right]^{1/2}$$

Can be generalized to $L^p([-\frac{1}{2}, \frac{1}{2}]^d)$, $p \in (0, \infty)$

There are lots of new things to consider. Let's first look at a functional class similar to one we have seen before, and see the effect of quantization.

Define: $C^\alpha(X)$, for $\alpha = s + \gamma$, $s \in \mathbb{N}$, $0 < \gamma \leq 1$, as the space of functions $F: X \rightarrow \mathbb{R}$ for which

$$\|F\|_\alpha = \|F\|_{C^\alpha(X)} = \max \left\{ \max_{\substack{\beta \in \mathbb{N}^d \\ \|\beta\|_1 \leq s}} \|\partial^\beta F\|_\infty, \max_{\substack{\beta \in \mathbb{N}^d \\ \|\beta\|_1 = s}} \text{Lip}_\gamma(\partial^\beta F) \right\} < +\infty$$

where

$$\text{Lip}_\gamma(G) = \sup_{\substack{x, x' \in X \\ x \neq x'}} \frac{|G(x) - G(x')|}{\|x - x'\|_2^\gamma}$$

Theorem: Let $F \in C^\alpha([-1/2, 1/2]^d)$ with $\|F\|_\alpha = B < \infty$ and let $0 < \varepsilon < 1/2$. Then there are constants $t = t(d, \alpha, B) \in \mathbb{N}$ and $c = c(d, \alpha, B) > 0$ for which there is a neural network $f(x; \theta)$ with

$$L \leq (2 + \lceil \log_2 \alpha \rceil)(11 + \alpha/d)$$

layers and

$$\|\theta\|_0 \leq c \cdot \varepsilon^{-d/\alpha}$$

nonzero, (t, ε) -quantized weights such that

$$\|F - f\|_{L^2} < \varepsilon$$

and

$$\|f\|_{L^\infty} \leq \lceil B \rceil$$

Remarks: (a) # of nonzero weights similar to previous complexity results

(b) Smoother F , i.e., larger α , more layers needed, but less total # of nonzero weights

(c) Weights are bounded and quantized, and $\sigma = \text{ReLU}$, so the 2-layer result of Pinkus does not apply.

Idea of proof:

(a) ReLU NNs can approximately implement multiplication.

The weights θ are also (t, ε) -quantized

In particular, let $z, z' \in [-1/2, 1/2]$. Then there is a ReLU NN, $g(z, z'; \theta)$, with L layers and

$$\|\theta\|_0 = O(\varepsilon^{-c/L}), \quad c > 0 \text{ universal,}$$

such that

$$|zz' - g(z, z'; \theta)| \leq \varepsilon \quad \text{for all } z, z' \in [-1/2, 1/2]$$

(b) Now one can approximate monomials

(c) Now one can approximate polynomials, including Taylor polynomials



(d) Patch together several local Taylor polynomial approximations of F to get an approximation of F on all of $[-\frac{1}{2}, \frac{1}{2}]^d$. To carry out this step, one has to show neural networks can implement cutoff functions using a fixed number of layers and weights. That is, ReLU neural networks with $c = c(d)$ nonzero, (s, ε) -quantized weights, $s = s(d)$, can approximate functions of the form

$$G(x) = \mathbb{I}_{[a_1, b_1] \times \dots \times [a_d, b_d]}(x)$$

to ε accuracy in the $L^2[-\frac{1}{2}, \frac{1}{2}]^d$ norm.

Now let us go back to label functions of the form:

$$F(x) = \sum_{m=1}^M a_m \mathbb{I}_{K_m}, \quad \mathbb{I}_{K_m} \in \mathcal{C}^\alpha \quad (*)$$

We will build up to a result for functions F of the form (*).

Step 1: Horizon functions: A horizon function is 0-1 valued function with a jump along a hypersurface such that the jump surface is the graph of a smooth function.

Define the heavyside function $H: \mathbb{R}^d \rightarrow \mathbb{R}$ as

$$H(x) = \mathbb{I}_{[0, \infty) \times \mathbb{R}^{d-1}}(x)$$

$$\mathcal{H}^\alpha[-\frac{1}{2}, \frac{1}{2}]^d = \left\{ \begin{array}{l} \tilde{G} \circ T \in L^\infty[-\frac{1}{2}, \frac{1}{2}]^d : \\ \tilde{G}(x) = H(x(1) + \phi(x(2), \dots, x(d)), x(2), \dots, x(d)) \\ \phi \in C^\alpha(\mathbb{R}^{d-1}) \\ T \text{ is a permutation matrix} \end{array} \right\}$$

For horizon functions $G \in \mathcal{H}^\alpha(\mathbb{R}^d)$ we define the "norm" of G as

$$\|G\|_{\mathcal{H}^\alpha[-\frac{1}{2}, \frac{1}{2}]^d} = \|\phi\|_{C^\alpha[-\frac{1}{2}, \frac{1}{2}]^d}$$

Step 2: Approximate horizon functions w/ ReLU networks

Lemma: Let $G \in \mathcal{H}^\alpha[-\frac{1}{2}, \frac{1}{2}]^d$ w/ $\|G\|_{\mathcal{H}^\alpha[-\frac{1}{2}, \frac{1}{2}]^d} = B < \infty$ and let $0 < \varepsilon < 1/2$. Then there are constants $t = t(d, \alpha, B) \in \mathbb{N}$ and $c = c(d, \alpha, B) > 0$ for which there is a neural network $g(x; \theta)$ with

$$L \leq (2 + \lceil \log_2 \alpha \rceil)(14 + 2^\alpha/d)$$

layers and

$$\|\theta\|_0 \leq c \cdot \varepsilon^{-2(d-1)/\alpha}$$

nonzero, (t, ε) -quantized weights such that

$$\|G - g\|_{L^2[-\frac{1}{2}, \frac{1}{2}]^d} \leq \varepsilon \quad \text{and} \quad 0 \leq g(x; \theta) \leq 1$$

Idea of proof: Horizon functions are similar to cutoff functions, except instead of the jump discontinuity being a hyperplane (as in a cutoff function), it is along a hypersurface that is the graph of a C^α function. We have "proven" two results that can resolve this:

(1) NNs can approximate cutoff functions

(2) NNs can approximate C^α functions

Combine these two results to prove the lemma.

Step 3: Reduction to one set K w/ $\partial K \in C^\alpha$.

Note if we can approximate $\mathbb{I}_K(x)$, then we can approximate

$$F(x) = \sum_{m=1}^M a_m \mathbb{I}_{K_m}(x)$$

up to a constant depending on M .

Step 4: Introduce a space of sets based on horizon functions

$$\mathcal{K}_{r,B}^\alpha [-\frac{1}{2}, \frac{1}{2}]^d = \left\{ K \subset [-\frac{1}{2}, \frac{1}{2}]^d : \right.$$

For all $x \in [-\frac{1}{2}, \frac{1}{2}]^d$, there exists a $G_x \in \mathcal{H}^\alpha [-\frac{1}{2}, \frac{1}{2}]^d$
with $\|G_x\|_{\mathcal{H}^\alpha [-\frac{1}{2}, \frac{1}{2}]^d} \leq B$ for which

$$\mathbb{I}_K(x') = G_x(x') \\ \text{for all } x' \in [-\frac{1}{2}, \frac{1}{2}]^d \text{ with } \|x - x'\|_\infty \leq 2^{-r} \left. \right\}$$

Remark: If $K \subset [-\frac{1}{2}, \frac{1}{2}]^d$ with $\partial K \in C^\alpha$, then $K \in \mathcal{K}_{r,B}^\alpha [-\frac{1}{2}, \frac{1}{2}]^d$ for some large enough r and B .

Step 5: Approximate $\mathbb{I}_K(x)$, $K \in \mathcal{K}_{r,B}^\alpha [-\frac{1}{2}, \frac{1}{2}]^d$, with a ReLU neural network

Theorem: Let $K \in \mathcal{K}_{r,B}^\alpha [-\frac{1}{2}, \frac{1}{2}]^d$ and $0 < \varepsilon < 1/2$. Then there are constants $t = t(d, r, \alpha, B) \in \mathbb{N}$ and $c = c(d, r, \alpha, B) > 0$ for which there is a neural network $f(x; \theta)$ with

$$L \leq (3 + \lceil \log_2 \alpha \rceil) (1 + 2^{\alpha/d})$$

layers and

$$\|\theta\|_0 \leq c \cdot \varepsilon^{-2(d-1)/\alpha}$$

nonzero, (t, ε) -quantized weights such that

$$\|\mathbb{I}_K - f\|_{L^2 [-\frac{1}{2}, \frac{1}{2}]^d} < \varepsilon$$

and $\|f\|_\infty \leq 1$

Idea of Proof: Any $K \in \mathcal{K}_{r,B}^\alpha [-\frac{1}{2}, \frac{1}{2}]^d$ is locally a horizon function, which we can approximate by the lemma. Now patch together these local approximations.

Now let us consider lower bounds on the required complexity of NNs with (t, ϵ) -quantized weights for approximating horizon functions.

Note: horizon functions \subset piecewise constant functions
 \subset piecewise smooth functions

So the results apply to these classes of functions as well

I am not going to state these results as precisely as the upper bound results.

We will consider two types of complexity:

(1) the number of nonzero weights, $\|\theta\|_0$

(2) the number of layers, L

We will see the results of the previous theorem are nearly sharp, meaning that one cannot do (much) better than the bound for $\|\theta\|_0$ and the bound for L .

On the number of nonzero weights: $\leftarrow \nabla f(0) = 0$ all that is required here

In order to guarantee, for any horizon function $G \in \mathcal{H}^\alpha[-\frac{1}{2}, \frac{1}{2}]^d$ with

$$\|G\|_{\mathcal{H}^\alpha[-\frac{1}{2}, \frac{1}{2}]^d} \leq B, \text{ that}$$

$$\|G - g\|_{L^2[-\frac{1}{2}, \frac{1}{2}]^d} \leq \epsilon$$

for some neural network $g(x; \theta)$, the number of nonzero, (t, ϵ) -quantized weights must be at least

$$\|\theta\|_0 \geq \frac{C \cdot \epsilon^{-2(d-1)/2}}{\log_2(1/\epsilon)}$$

where $C = C(d, \alpha, B)$ and $t = t(d, \alpha, B)$.

Remarks: (i) The neural network from the approximation theorem is nearly optimal since it had

$$\|\theta\|_0 \leq C \cdot \epsilon^{-2(d-1)/2}$$

nonzero, (t, ϵ) -quantized weights.

(ii) One can extend the result of Pinkus and Maionov (1999) to horizon functions. Since the number of weights in the Pinkus/Maionov network is fixed, regardless of ϵ , this result would seem to indicate they must either be very complex (i.e., not quantized) or very large. There is the technical point, though, that the nonlinearity of Pinkus/Maionov does not satisfy $\nabla f(0) = 0$.

On the depth of the network:

Not precise version:

The number of layers must be at least

$$L \geq \frac{\alpha}{4(d-1)}$$

Thus, again, the number of layers in the approximation theorem,

$$L \leq (3 + \lceil \log_2 \alpha \rceil) \left(11 + \frac{2\alpha}{d}\right)$$

is nearly optimal.

Curse of dimensionality

Suppose $F = G \circ \tau$, where $\tau: \mathbb{R}^d \rightarrow \mathbb{R}^k$ is smooth and

$$G(z) = \sum_{m=1}^M a_m \mathbb{I}_{k_m}(z)$$

If τ is nice enough (e.g., $\tau \in C^\infty$, $D\tau(x)$ is full rank, plus some other properties) then one can prove something like:

For each $0 < \varepsilon < 1/2$ there is a neural network $f(x; \theta)$ with at most

$$\|\theta\|_0 \leq C \cdot \varepsilon^{-2(k-1)/\alpha}$$

nonzero, $(t|\varepsilon)$ -quantized weights such that

$$\|F - f\|_{L^2[-\frac{1}{2}, \frac{1}{2}]^d} < \varepsilon$$

Remarks: (i) C depends on d , but the rate $O(\varepsilon^{-2(k-1)/\alpha})$ depends on k

(ii) Number of layers is very large, and probably is suboptimal.

Convolutional Neural Networks

Convolutional neural networks (CNNs) are used when the data $x \in \mathbb{R}^d$ has an underlying Euclidean geometric structure. The most prominent example is when x is an $N \times N$ image so that x can be written as:

$$x(n_1, n_2) \in [0, 1], \quad 0 \leq n_i < N, \quad i=1, 2 \quad (*)$$

In this case $x \in [0, 1]^d$ where $d = N^2$, but x has additional structure given by (*). An example we have already seen is MNIST:


$$x = \left[\begin{array}{c} \text{28 pixels} \\ \text{28 pixels} \end{array} \right] \quad , \quad x \in [0, 1]^{28^2} = [0, 1]^{784}$$


Image processing is quite common in machine learning, e.g. computer vision, and countless other examples exist. Some popular image databases include:

- MNIST: 70,000; 28×28 ; grayscale; handwritten digits
- CIFAR-10: 60,000; 32×32 ; color images with 10 classes
- CIFAR-100: 60,000; 32×32 ; color images with 100 classes
- ImageNet: Over 14 million color images with over 20,000 classes

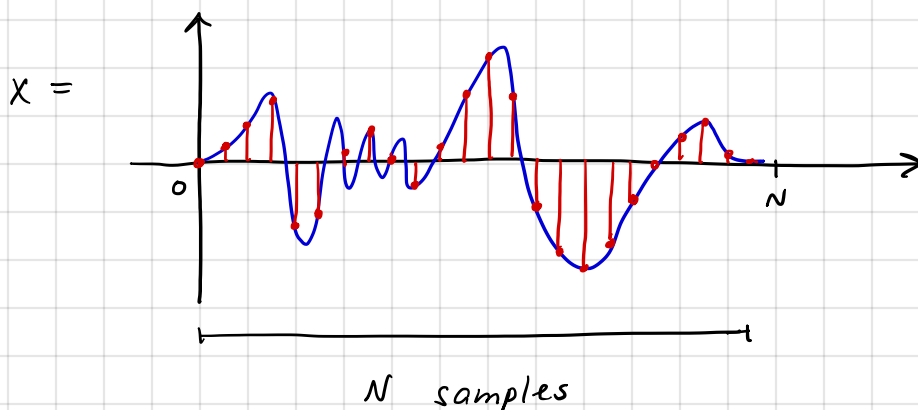
CNNs are also used for data with 1D and 3D geometries. 1D signals consist of, for example:

- audio recordings, as in speech, music, etc.
- medical device recordings, as in EEG, etc.
- more generally time series, although if one wants to make predictions of the future values of a single time series, one should use a recurrent neural network.

In this case $d = N$ and x is of the form:

$$x(n) \in \mathbb{R}, \quad 0 \leq n < N$$

which is the same vector structure from before, but now there is the implication that the order $x(0), x(1), x(2), \dots, x(N-1)$ matters.



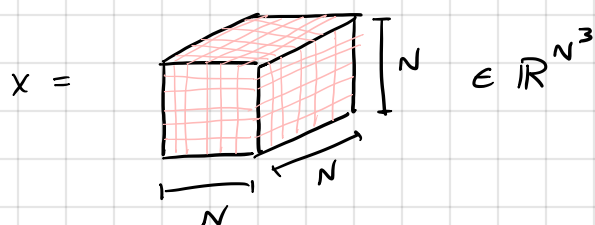
3D signals may consist of, e.g.,

- volumetric medical data
- volumetric data from physics, e.g., many particle physics and fluid mechanics
- self-driving car data
- LiDAR data

The idea is similar to 1D & 2D, in this case:

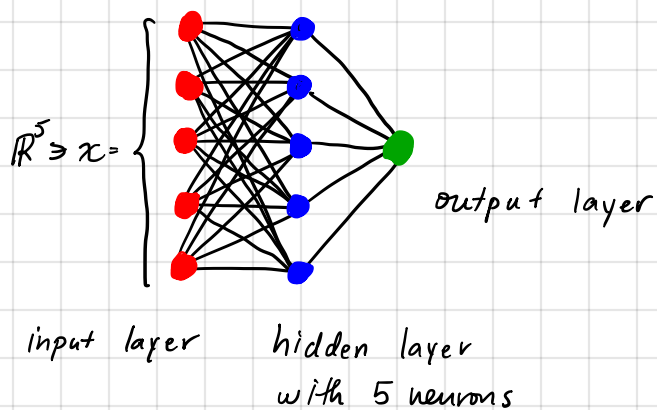
$$X(n_1, n_2, n_3) \in \mathbb{R}, \quad 0 \leq n_i < N, \quad i=1,2,3$$

and so $X \in \mathbb{R}^d$, $d = N^3$



CNNs as special cases of ANNs

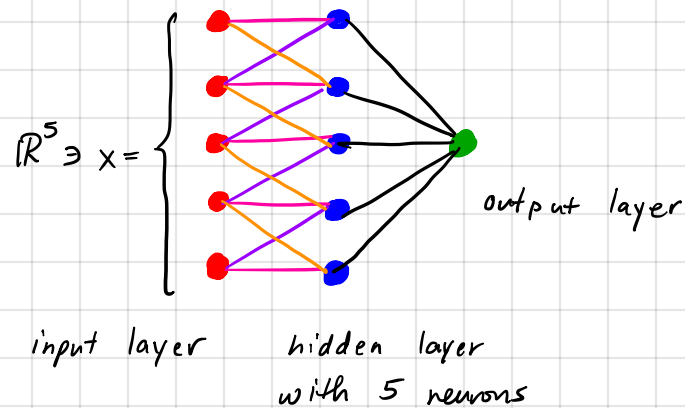
CNNs can be viewed as ANNs with sparse, shared weights. Let us first look at a diagram, supposing that $X \in \mathbb{R}^d$, $d=N$, has a 1D geometric structure.



All edges indicate different weights

Fully connected ANN w/ one hidden layer.

$$f(x; \theta) = \langle \alpha, \sigma(Wx + \beta) \rangle$$



All pink/purple/orange edges have the same weight

Black edges may have different weights

CNN = sparsely connected ANN with shared weights.

$$\tilde{f}(x; \theta) = \langle \alpha, \sigma(\tilde{W}x + \beta) \rangle$$

Let us examine the CNN diagram more closely:

We have five neurons:

$$\left. \begin{aligned} w_0 &= (b, c, 0, 0, 0) \\ w_1 &= (a, b, c, 0, 0) \\ w_2 &= (0, a, b, c, 0) \\ w_3 &= (0, 0, a, b, c) \\ w_4 &= (0, 0, 0, a, b) \end{aligned} \right\} \tilde{W} = \begin{pmatrix} b & c & 0 & 0 & 0 \\ a & b & c & 0 & 0 \\ 0 & a & b & c & 0 \\ 0 & 0 & a & b & c \\ 0 & 0 & 0 & a & b \end{pmatrix} = \text{Toeplitz type weight matrix}$$

(let us ignore the biases.)

Notice then that:

$$\langle x, w_0 \rangle = b x(0) + c x(1)$$

$$\langle x, w_n \rangle = a x(n-1) + b x(n) + c x(n+1), \quad 1 \leq n \leq 3$$

$$\langle x, w_4 \rangle = a x(3) + b x(4)$$

Let $x, y \in \mathbb{R}^N$ and define their correlation as:

$$(x \star y)(n) = \sum_{m=0}^{N-1} x(m) y(n+m), \quad y(n) = 0 \quad \forall n \notin [0, N)$$

Note $(x \star y)(n)$ takes, in general, nonzero values for: $-N < n < N$.

Therefore we can think of $x \star y \in \mathbb{R}^{2N-1}$

A subset of these values correspond to $\langle x, w_n \rangle$. In our example above:

Define $w = w_2 = (0, a, b, c, 0) \in \mathbb{R}^5$. Then:

$$\langle x, w_n \rangle = (x \star w)(n-2), \quad 0 \leq n \leq 4$$

The vector w is called a filter. *Finished class here*

The operation of convolution is closely related to correlation. It is defined as:

$$(x \star y)(n) = \sum_{m=0}^{N-1} x(m) y(n-m)$$

Set $\bar{y}(n) = y(-n)$. Then:

$$(x \star \bar{y})(-n) = \sum_{m=0}^{N-1} x(m) \bar{y}(-n-m) = \sum_{m=0}^{N-1} x(m) y(n+m) = (x \star y)(n) \quad (*)$$

Notice in the CNN network, we only need to learn 3 parameters, a, b, c , in the hidden layer, versus $5^2 = 25$ parameters in the fully connected network.

In 2D, correlation is defined as:

$$(x \star y)(n_1, n_2) = \sum_{m_1=0}^{N-1} \sum_{m_2=0}^{N-1} x(m_1, m_2) y(n_1+m_1, n_2+m_2)$$

In practice, CNNs implement correlation, not convolution. But equation (*) shows they are equivalent, so we will use both definitions and will often use convolution.

In practice these filters are often small, e.g., if $x \in \mathbb{R}^{N^2}$ is an $N \times N$ image, the filters may only be $(2s+1) \times (2s+1)$ in which the filter is indexed as:

$$w(m_1, m_2) \in \mathbb{R}, \quad m_i \in [-s, s] \cap \mathbb{Z}, \quad i=1,2$$

We still index x as:

$$x(n_1, n_2) \in \mathbb{R}, \quad n_i \in [0, N) \cap \mathbb{Z}, \quad i=1,2$$

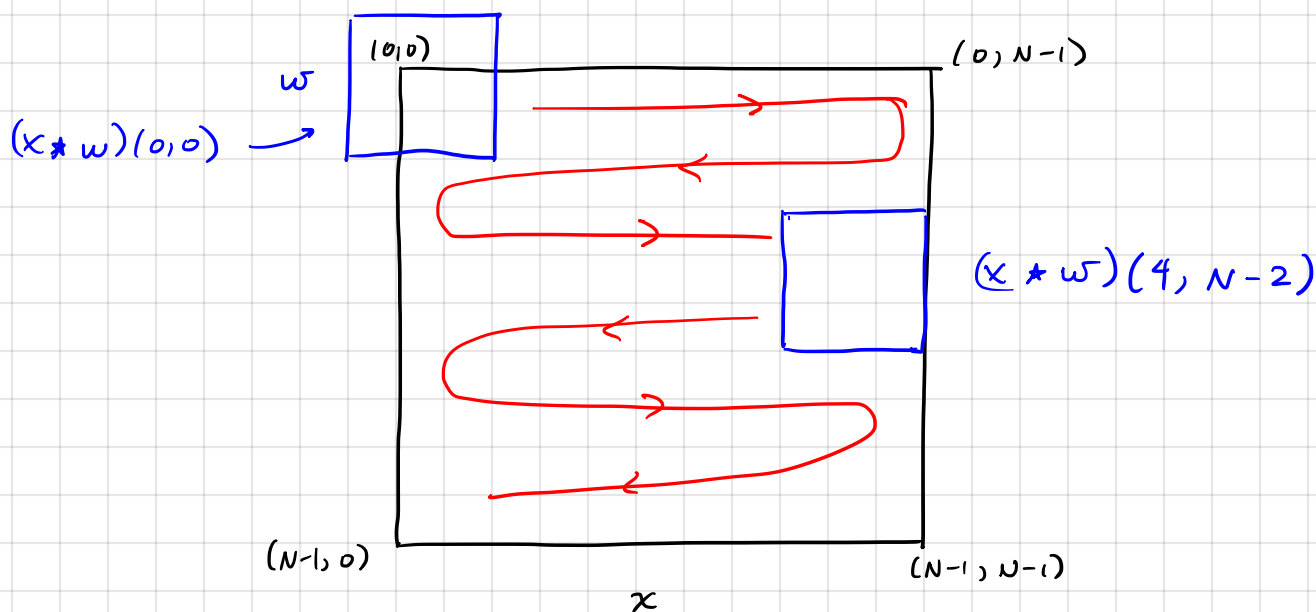
Then:

$$(x * w)(n_1, n_2) = \sum_{m_1=-s}^s \sum_{m_2=-s}^s w(m_1, m_2) x(n_1 + m_1, n_2 + m_2)$$

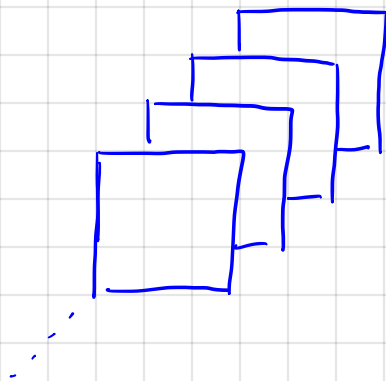
$$n_i \in [0, N) \cap \mathbb{Z}, \quad i=1,2$$

Here s may be, e.g., $s=1, 2$, or 3

Pictorially:



Also in practice there will be more than one filter in each layer. We'll come back to this point later.



A stack of filters for one layer

CNNs work under two related priors, both related to the underlying geometry of the signal:

(i) Neighboring dimensions, e.g. pixels, are related and their relation is relevant to the task. Note this relation may be "long range," that is, "goes over large portions of the image, but it is still governed by the underlying geometry of each data point."

⇒ weights only between nearby dimensions
 • long range relations obtained through depth and new nonlinearities (more on this later)

(ii) Translation equivariance / invariance: (small) translations of the signal, e.g., $x_t(n) = x(n-t)$, do not change the class of x .

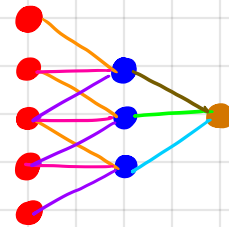
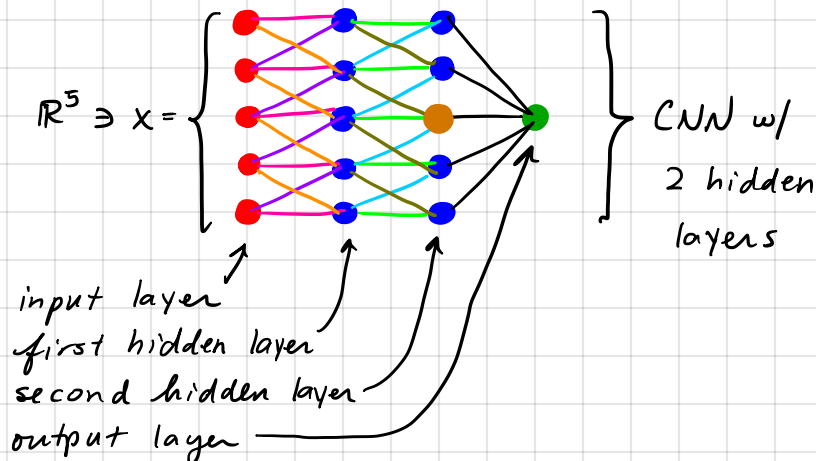
⇒ weight sharing and correlation/convolution operators.
 Indeed,

$$\begin{aligned} (x_t * y)(n) &= (x * y)(n-t) \\ (x_t * y)(n) &= (x * y)(n-t) \end{aligned} \quad \left[\begin{array}{l} \text{assuming} \\ \text{zero padding} \end{array} \right]$$

* Translating the input merely translates the output by the same amount. * ← EQUIVARIANCE

and (ii)

Let us examine (i) a bit more closely. In the previous example we used filter with a receptive field (that is support) of 3 dimensions. But what if we believe there are relations between all 5 dimensions? One option is to use a wider filter. Another is to use a deeper network, e.g.



Closer look at the orange neuron. Notice the orange neuron collects information from all 5 original dims. Thus, even though

$|supp(w_1)| = |supp(w_2)| = 3$,
 the effective support of w_2 relative to the input x , i.e. its receptive field, is 5.

$$f(x; \theta) = \sum_n \alpha(n) \tau(\tau(x * w_1) * w_2)(n)$$

Notice in particular if you have L filters w_l , $1 \leq l \leq L$, each with $|\text{supp}(w_l)| = s$

and you compute an L layer network of the form:

$$\sigma(\dots \sigma(\sigma(x * w_1) * w_2) * \dots * w_L)$$

then the effective receptive field of w_l relative to x is:

$$\text{effective receptive field of layer } l = (s-1)l + 1 \quad \left[\text{This is for 1D} \right]$$

This means the receptive field grows linearly with depth. While this is good, for high dimensional data (e.g., time series with very large numbers of samples such as $N \geq 2^{13}$ or high resolution images) this may not be fast enough. CNNs thus incorporate an additional (nonlinear) **pooling operation**. This pooling operator works by (nonlinearly) downsampling the output of a given layer. Let $x \in \mathbb{R}^N$, a factor 2 downsampling operator applied to x returns a new vector x_d with half the length:

$$x_d(n) = x(2n), \quad 0 \leq n < N/2$$

This type of downsampling is standard in signal processing.

In 2D for signals $x \in \mathbb{R}^{N^2}$, $x_d \in \mathbb{R}^{(N/2)^2} = \mathbb{R}^{N^2/4}$ with:

$$x_d(n_1, n_2) = x(2n_1, 2n_2), \quad 0 \leq n_1, n_2 < N/2$$

CNNs incorporate other types. Two common ones are:

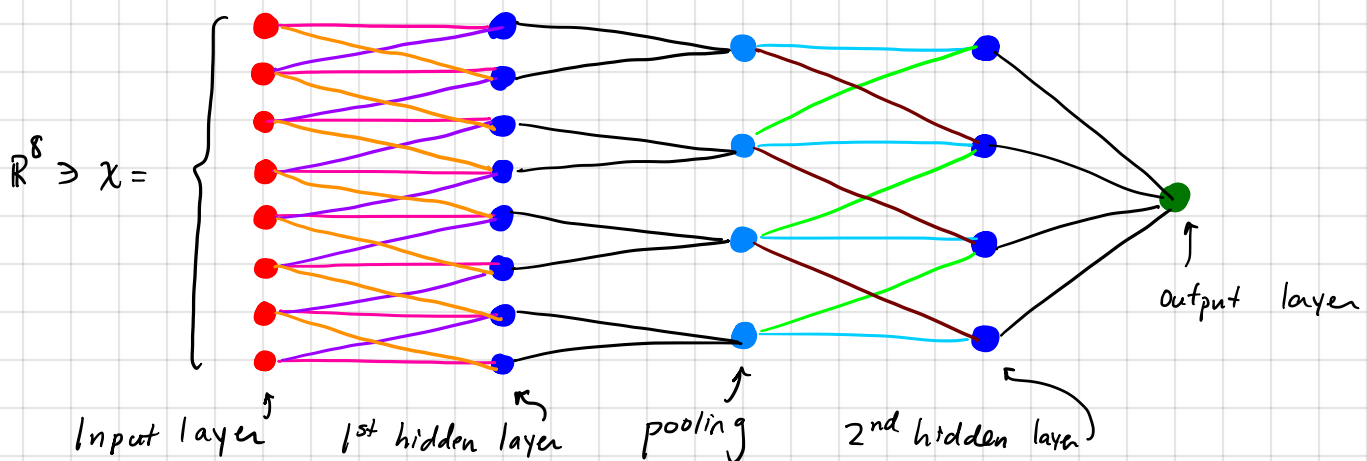
(i) Average pooling:

$$x_d(n) = \frac{1}{2} (x(2n) + x(2n+1)), \quad 0 \leq n < N/2$$

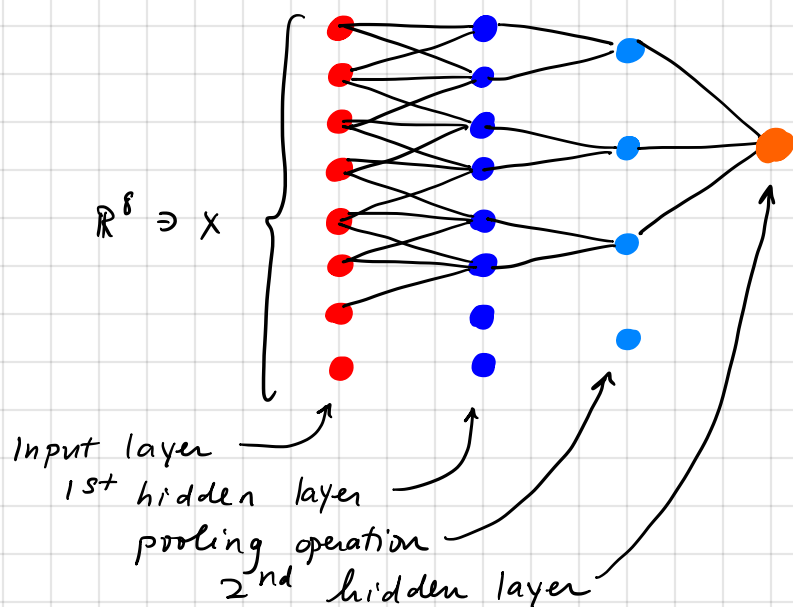
(ii) Max pooling, which is nonlinear:

$$x_d(n) = \max(x(2n), x(2n+1))$$

Pooling operations serve multiple roles. One is to expand the receptive field of deeper filters. They also encode invariance into the representation of x . Let us first consider the receptive field:



Notice how the pooling operation expands the receptive field of the 2nd hidden layer. In the example without pooling, the receptive field of the 2nd hidden layer was 5. Now with the pooling operation we have:



Thus the receptive field of the orange neuron in the 2nd hidden layer is 7 with the pooling operation, but the number of learned parameters has not increased. To get the same receptive field without pooling, we would need 3 hidden layers which would increase the number of trained parameters.

Pooling operations in 1D effectively double the support of the neurons coming after them. In 2D a pooling is a function of 4 pixels:

$$x_d(n_1, n_2) = \text{pool} \left[x(2n_1, 2n_2), x(2n_1+1, 2n_2), x(2n_1, 2n_2+1), x(2n_1+1, 2n_2+1) \right]$$

So the support is quadrupled

Equivariance vs. Invariance

The other important role of convolutional operators and pooling operators is their role in developing equivariant and invariant representations.

Let $x_t(n) = x(n-t)$ be the translation of x by t .

Let $\Phi(x) \in \mathbb{R}^p$ be a representation of x , e.g., $\Phi(x)$ could be the last hidden layer of a neural network.

We say $\Phi(x)$ is translation equivariant if

$$\Phi(x_t)(n) = \Phi(x)(n-t)$$

The representation $\Phi(x)$ is translation invariant if

$$\Phi(x_t) = \Phi(x)$$

The representation $\Phi(x)$ is translation invariant up to scale 2^J if:

$$\|\Phi(x) - \Phi(x_t)\|_2 \leq C |t| 2^{-J} \|x\|_2$$

The local translation invariance property implies that if the translation t is small relative to the scale 2^J , that is:

$$|t| \ll 2^J \Rightarrow |t|/2^J \ll 1$$

then $\tilde{\Phi}(x)$ and $\tilde{\Phi}(x_t)$ are nearly identical since:

$$\|\tilde{\Phi}(x) - \tilde{\Phi}(x_t)\|_2 \leq C \cdot \underbrace{|t|/2^J}_{\ll 1} \cdot \|x\|_2 \ll 1$$

An equivariant representations may be useful in its own right. For example, if you are considering the force acting on a body, the force is equivariant with respect to translations and rotations of the body. Equivariant representations are also important for extracting invariant representations. Indeed, suppose $\tilde{\Phi}(x)$ is a translation equivariant representation. Then:

$$\phi(x) = \langle \alpha, \tau(\tilde{\Phi}(x)) \rangle = \sum_n \alpha(n) \tau(\tilde{\Phi}(x)(n)) \in \mathbb{R} \quad (*)$$

is invariant to translations of x , that is:

$$\phi(x_t) = \phi(x)$$

Notice that correlation/convolution yield translation equivariant representations since

$$(x_t * w)(n) = (x * w)(n-t)$$

Fully invariant representations $\tilde{\Phi}(x) = (\phi_\lambda(x))_{\lambda \in \Lambda}$ consisting of components $\phi_\lambda(x)$ defined through $(*)$ are useful when one has a known global invariance prior. This may be the case in data driven problems coming from chemistry, physics, biology, and statistical modeling of time series, among other contexts. For example, the potential energy of a many body system is invariant to global translations and rotations of the system, and the statistics of a stationary stochastic process are invariant to translations. These types of globally invariant representations are also useful for processing data that can be modelled as an abstract graph $G = (V, E)$, consisting of vertices V connected by edges E . In order to compare two graphs G_1 and G_2 we need a representation $\tilde{\Phi}(G)$ that is invariant to the order in which the vertices are enumerated.

On the other hand, in image processing tasks in computer vision, global translation invariance is often too inflexible. Rarely does one encounter large, global translations (or rotations) of images, but smaller translations and rotations are more common.

A pooling operation increases local translation invariance by a factor of 2. Therefore if we incorporate J pooling operations in our neural network, the local translation invariance of the network will be up to scale 2^J . Note this only works because the linear operations are translation equivariant convolution operators.

Notice that the translation invariance properties, even the one with scale 2^J , still refer to translations that act on the whole signal. Many signal deformations of interest act locally on the signal. We can define these mathematically as diffeomorphisms, and think of them as generalized translations. In order to develop this framework, it is useful to model the data point x as a function. We have:

- 1D: $x: \mathbb{R} \rightarrow \mathbb{R}$
- 2D: $x: \mathbb{R}^2 \rightarrow \mathbb{R}$
- 3D: $x: \mathbb{R}^3 \rightarrow \mathbb{R}$

The discrete data can be considered a sampling of these functions, that is $x(n)$ is the evaluation of x at $n \in \mathbb{Z}$ and on the computer we store $(x(n))_{0 \leq n < N}$.

Let us consider $x: \mathbb{R} \rightarrow \mathbb{R}$, that is 1D signals, keeping in mind that everything can be generalized to 2D and 3D signals.

Let $\tau: \mathbb{R} \rightarrow \mathbb{R}$ with $\tau \in C^2(\mathbb{R})$ and

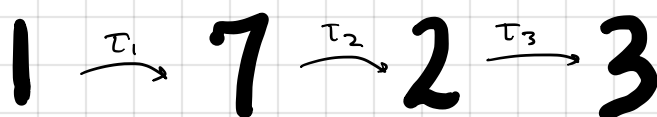
$$\|\tau'\|_\infty = \sup_{u \in \mathbb{R}} |\tau'(u)| \leq \frac{1}{2}$$

Then the mapping $u \mapsto u - \tau(u)$ is a diffeomorphism with displacement field τ , that is, $u \in \mathbb{R}$ gets moved to $u - \tau(u)$, which displaces u by $\tau(u)$. We can model deformations of our data through such diffeomorphisms as:

$$x_\tau(u) = x(u - \tau(u))$$

Notice that if $\tau(u) = t$, then this operation is a translation. But this model allows us to study "local" translations and other operations that deform the data locally.

Unlike translations in which we may wish for a translation invariant representation, i.e. $\Phi(x_t) = \Phi(x)$ where $x_t(u) = x(u - t)$, encoding invariance over diffeomorphisms is too strong. Indeed the diffeomorphism group is infinite-dimensional (for data $x: \mathbb{R}^p \rightarrow \mathbb{R}$ the translation group is p -dimensional) and one can string together small diffeomorphisms to go between vastly different data points, e.g., in MNIST:



Therefore diffeomorphism invariance is far too strong since we would classify too many things as the same. Rather we seek a representation that is stable to diffeomorphisms, meaning:

$$\|\Phi(x) - \Phi(x_\tau)\|_2 \leq C \cdot \text{size}(\tau) \cdot \|x\|_2 \quad (*)$$

Later we will discuss CNNs in which $\text{size}(\tau)$ depends on $\|\tau'\|_\infty$ and $\|\tau''\|_\infty$. In particular if τ is a translation then $\text{size}(\tau) = 0$, so $(*)$ will imply global translation invariance. If we want only translation invariance up to the scale 2^J we can amend $(*)$ as:

$$\|\Phi(x) - \Phi(x_\tau)\|_2 \leq C \cdot \left[2^{-J} \|\tau\|_\infty + \underbrace{\text{size}(\tau)}_{F(\|\tau'\|_\infty, \|\tau''\|_\infty)} \right] \|x\|_2$$

↑
 $F(\|\tau'\|_\infty, \|\tau''\|_\infty)$
measures the translation part of τ .

Multiple channels

Since a Toeplitz weight matrix only implements one convolutional filter, the expressiveness of the network will be pretty limited. CNNs rectify this by using many filters in each layer. This also allows CNNs to encode additional invariants on top of translation invariance. We will explain how stacking multiple filters works using the VGG network as a model. This will also explain how color images are processed.

A color image x can be modeled as $x: \mathbb{R}^2 \rightarrow \mathbb{R}^3$, in which:

$$x(u) = (x_r(u), x_g(u), x_b(u)), \quad u = (u_1, u_2) \in \mathbb{R}^2$$

and where x_r is the red channel, x_g is the green channel, and x_b is the blue channel. The first hidden layer of the VGG network processes x using a bank of $3M_1$ filters, M_1 filters for each channel:

$$x \mapsto \left((x_r * w_{r,j}^{(1)})_{j=1}^{M_1}, (x_g * w_{g,j}^{(1)})_{j=1}^{M_1}, (x_b * w_{b,j}^{(1)})_{j=1}^{M_1} \right)$$

Note this gives us $3M_1$ "images."

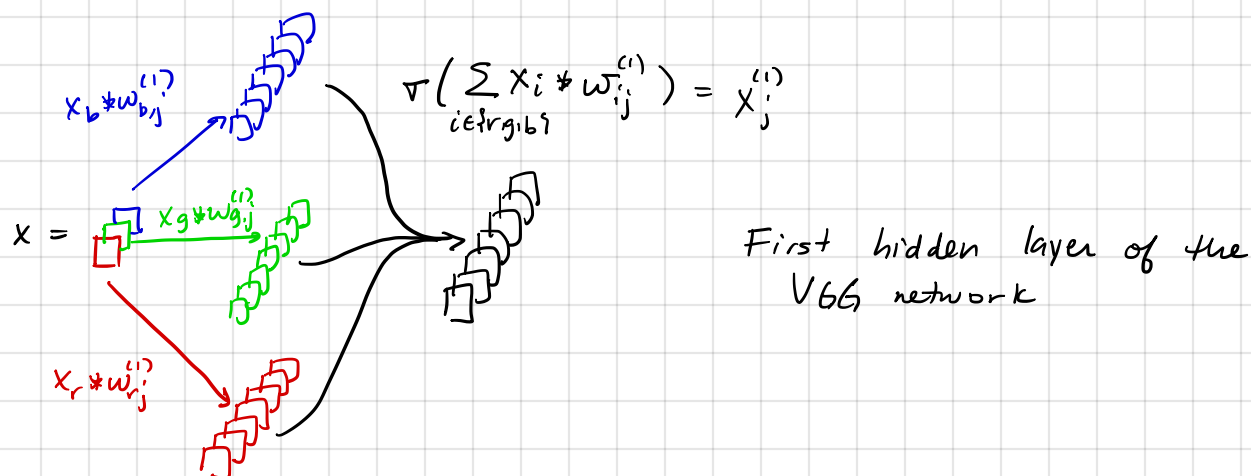
These responses are combined across the channels:

$$x \mapsto \sum_{i \in \{r,g,b\}} x_i * w_{ij}^{(1)}$$

and a nonlinearity is applied: $x \mapsto x_j^{(1)} = \sigma \left(\sum_{i \in \{r,g,b\}} x_i * w_{ij}^{(1)} \right), \quad 1 \leq j \leq M_1$
(e.g. ReLU)

Thus we have taken the original image $x: \mathbb{R}^2 \rightarrow \mathbb{R}^3$ and mapped it into a new M_1 channel image $x^{(1)}: \mathbb{R}^2 \rightarrow \mathbb{R}^{M_1}$ with:

$$x^{(1)}(u) = (x_j^{(1)}(u))_{j=1}^{M_1}, \quad u \in \mathbb{R}^2$$



The second hidden layer works similarly, except that instead of having the three RGB channels as input it has the M_1 channels from the first hidden layer as input:

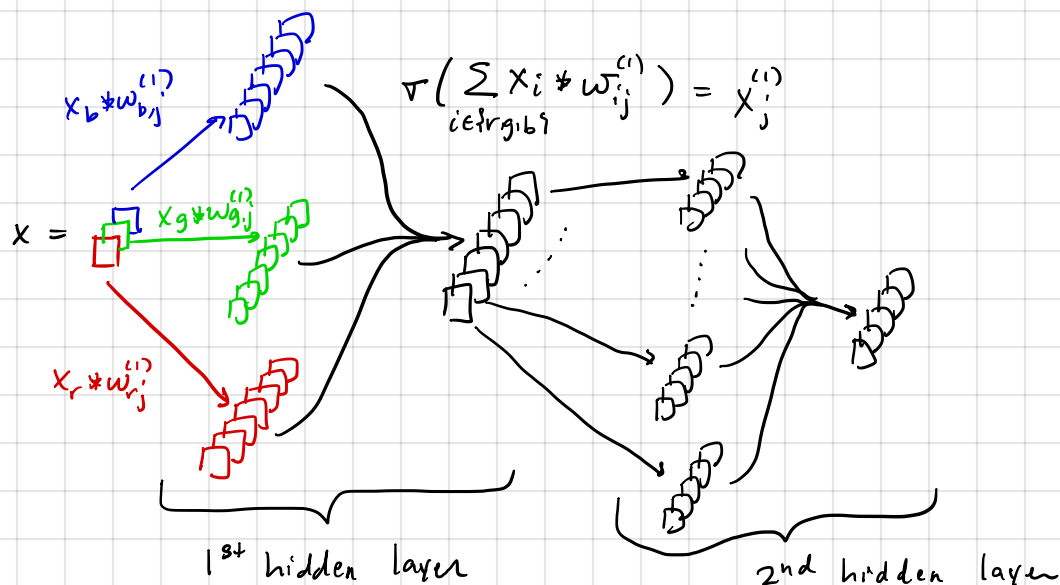
$$x^{(1)} = (x_j^{(1)})_{j=1}^{M_1} \mapsto x^{(2)} = (x_k^{(2)})_{k=1}^{M_2}$$

$$x_k^{(2)} = \nabla \left(\sum_{j=1}^{M_1} x_j^{(1)} * w_{j,k}^{(2)} \right), \quad 1 \leq k \leq M_2$$

$$= \nabla \left(\sum_{j=1}^{M_1} \nabla \left(\sum_{i \in \text{rgb}} x_i * w_{i,j}^{(1)} \right) * w_{j,k}^{(2)} \right)$$

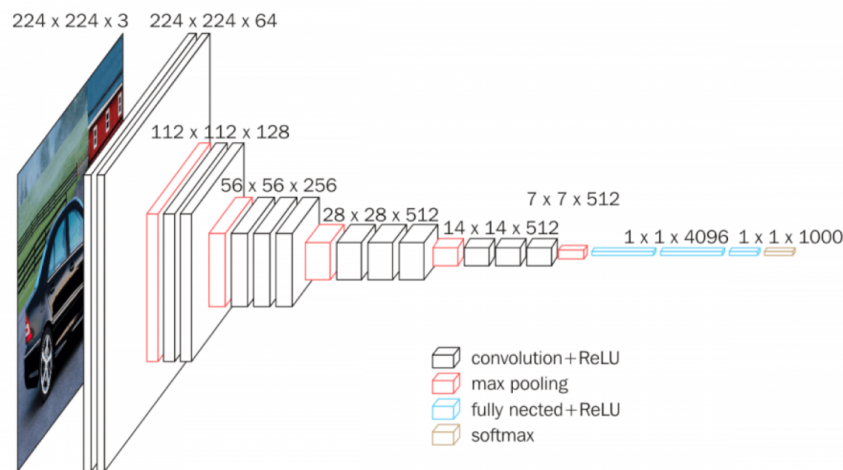
Thus the output of the 2nd hidden layer has M_2 channels:

$$x^{(2)}: \mathbb{R}^2 \rightarrow \mathbb{R}^{M_2}$$



1st and 2nd hidden layers of the VGG network

Subsequent layers work similarly. In some of the hidden layers a max pooling operation is also applied. Here is a diagram of the VGG network:



These stacks of filters within each layer give the CNN increased capacity for distinguishing between multiple types of signals. They also allow the CNN to encode invariants over groups other than the translation group. Convolution is equivariant with respect to the translation group, but not other groups such as the rotation group. However, the stack of filters can be learned to be equivariant with respect to other group actions. We will show later how this works for rotations.

CNNs from the perspective of approximation theory

The research on approximation theory for CNNs is limited and we will not spend much time on it. We mention three results:
(2017) additional

- 1.) Poggio, et al^v - We already discussed this result at length. CNNs are a special case of compositional networks in which the weights are shared and the composed dimensions are organized geometrically.
- 2.) Zhou - "Universality of deep CNNs" (2018) : Universal approximation by CNNs as the # of layers $L \rightarrow \infty$.
- 3.) Petersen & Voigtlaender - "Equivalence of approximation by CNNs and fully connected networks" (2018) : Rates of approximation by CNNs and ANNs are the same.

CNNs from the perspective of signal processing

Let us now see how CNNs arise naturally as a powerful way of representing signals. A lot of the mathematical ideas for this section come from:

- (i) Mallat - "Group Invariant Scattering" (2012)
- (ii) Bruna & Mallat - "Invariant Scattering Convolution Network" (2013)
- (iii) ... and several subsequent papers

Per our previous discussions, suppose we are looking for a representation $\Phi_J(x)$ of signal type data, which we model as $x: \mathbb{R} \rightarrow \mathbb{R}$.

Define

$$\|x\|_2 = \int_{\mathbb{R}} |x(u)|^2 du < +\infty$$

We want $\Phi_J(x)$ to have the following properties:

- (a) Translation invariance up to the scale 2^J
- (b) Stability to diffeomorphisms

Combining (a) and (b) and recalling that for $\tau \in C^2(\mathbb{R})$ w) $\|\tau'\|_{\infty} \leq \frac{1}{2}$ we defined

$$x_{\tau}(u) = x(u - \tau(u))$$

we want:

$$\|\Phi(x) - \Phi(x_{\tau})\|_2 \leq C \cdot [2^{-J} \|\tau\|_{\infty} + \|\tau'\|_{\infty} + \|\tau''\|_{\infty}] \|x\|_2$$

But is this enough? Consider the representation:

$$\Phi(x) = \int_{\mathbb{R}} x(u) du \quad \text{C.o.V: } v = u - t$$

We have:

$$\Phi(x_t) = \int_{\mathbb{R}} x_t(u) du = \int_{\mathbb{R}} x(u - t) du = \int_{\mathbb{R}} x(v) dv$$

$\Rightarrow \Phi(x_t) = \Phi(x)$ and so $\Phi(x)$ is translation invariant

We also have:

$$\begin{aligned} \Phi(x_{\tau}) &= \int_{\mathbb{R}} x_{\tau}(u) du = \int_{\mathbb{R}} x(u - \tau(u)) du & v = u - \tau(u) \\ & & dv = [1 - \tau'(u)] du \\ &= \int_{\mathbb{R}} \frac{x(v)}{1 - \tau'(u)} dv & (u \text{ depends on } v) \end{aligned}$$

$$\begin{aligned} \text{Therefore: } \Phi(x) - \Phi(x_{\tau}) &= \int_{\mathbb{R}} x(v) dv - \int_{\mathbb{R}} \frac{x(v)}{1 - \tau'(u)} dv \\ &= \int_{\mathbb{R}} \left[1 - \frac{1}{1 - \tau'(u)} \right] x(v) dv = \int_{\mathbb{R}} \frac{-\tau'(u)}{1 - \tau'(u)} \cdot x(v) dv \end{aligned}$$

$$\Rightarrow |\Phi(x) - \Phi(x_t)| = \left| \int_{\mathbb{R}} -\frac{\tau'(u)}{1-\tau'(u)} x(v) dv \right| \leq \int_{\mathbb{R}} \left| \frac{\tau'(u)}{1-\tau'(u)} \right| |x(v)| dv$$

$$\leq 2 \|\tau'\|_{\infty} \int_{\mathbb{R}} |x(v)| dv = 2 \|\tau'\|_{\infty} \|x\|_1 \quad (*)$$

Therefore $\Phi(x)$ is translation invariant and stable to diffeomorphisms as encoded by (*). But $\Phi(x)$ is not a very good representation because it is just the integral of x . Many different signals have the same integral. Therefore to (a) and (b) we must add another condition:

(c) The representation retains enough information in x to perform the task.

Condition (c) is not as precise as (a) and (b). A precise and very strong version of (c) is:

$$\Phi(x) = \Phi(y) \Leftrightarrow y = x_t \text{ for some } t \quad (**)$$

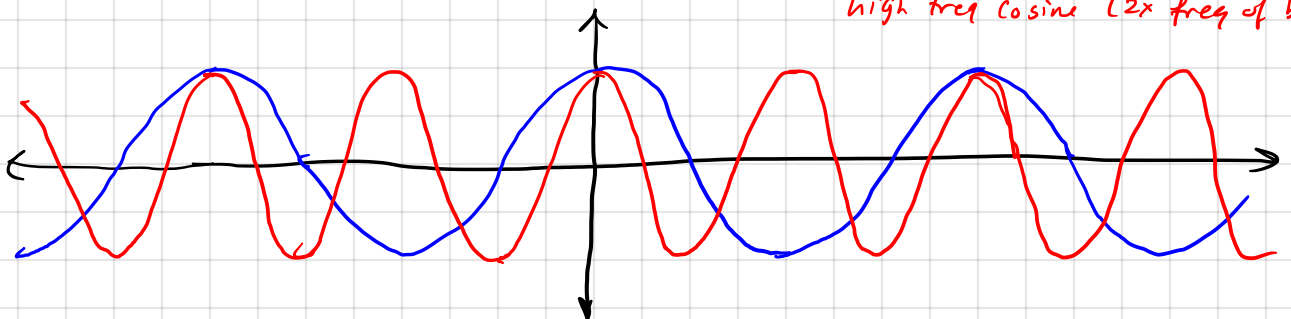
Equation (**) says $\Phi(x)$ is invertible up to translations. While this is mathematically precise, it may also take things too far. Indeed, in many classification tasks, $\Phi(x)$ being invertible is not a requirement for good classification results. We will instead be content to develop a systematic way of adding new information into $\Phi(x)$ while maintaining properties (a) (translation invariance) and (b) (stability to diffeomorphisms).

A key to understanding local translation invariance and diffeomorphism stability is through frequency representations of signals $x: \mathbb{R} \rightarrow \mathbb{R}$. For example, in a piece of music, we listen to the piece in time, but another way of representing the piece is through the notes, or frequencies, contained in it. The Fourier transform is the mathematically precise way to do this. Define a complex valued sinusoid at the frequency ω as:

$$e_{\omega}(u) = e^{i\omega u} = \cos(\omega u) + i \sin(\omega u), \quad i = \sqrt{-1}$$

The frequency is ω because the cosine and sine functions are periodic with period $2\pi/\omega$. Thus the higher ω , the faster the cosine and sine waves oscillate

low freq cosine
high freq cosine (2x freq of blue)



The Fourier transform of $x: \mathbb{R} \rightarrow \mathbb{R}$ w/ $\int_{\mathbb{R}} |x(u)| du < \infty$ computes:

$$\hat{x}(\omega) = \langle x, e_{\omega} \rangle = \int_{\mathbb{R}} x(u) e^{-i\omega u} du, \quad \omega \in \mathbb{R}$$

It thus tests the signal x against each sinusoid, and records which frequencies are present in x through \hat{x} .

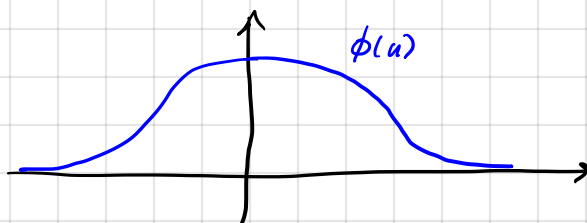
Assume $\int_{\mathbb{R}} |\hat{x}(\omega)| d\omega < \infty$. Then knowing \hat{x} is equivalent to knowing x since:

$$x(u) = \int_{\mathbb{R}} \hat{x}(\omega) e^{i\omega u} d\omega$$

We will let $\phi: \mathbb{R} \rightarrow \mathbb{R}$ denote a low pass filter. This means:

$$\hat{\phi}(\omega) = 0 \text{ for all } |\omega| > \pi \text{ and } 1 = \hat{\phi}(0) \geq |\hat{\phi}(\omega)|$$

Intuitively, ϕ will be a "bump function":



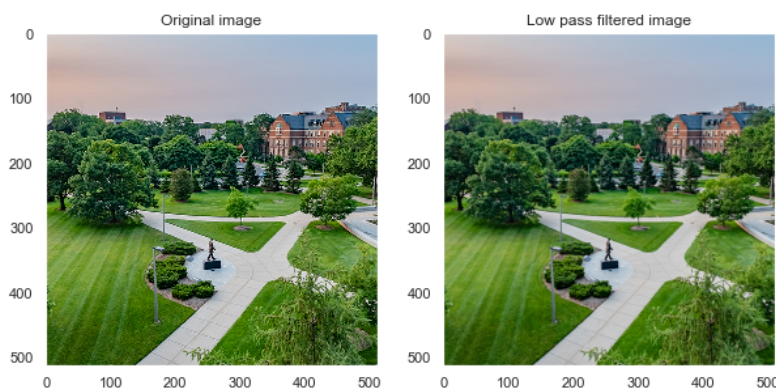
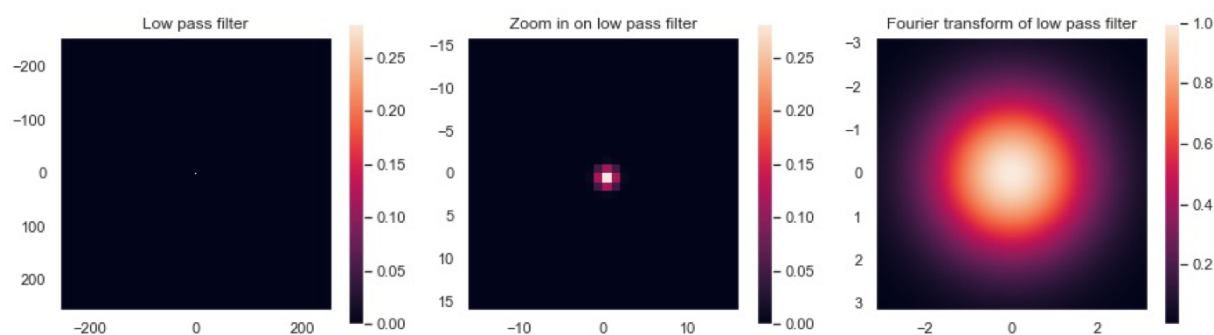
Filtering x with ϕ computes: $x * \phi$

The resulting signal $x * \phi$ is a smoothed, or blurred, version of x .
Since

$$(x * \phi)(\omega) = \hat{x}(\omega) \hat{\phi}(\omega) \quad (\text{Fourier convolution theorem})$$

It keeps only the low frequencies of x contained in $[-\pi, \pi]$.

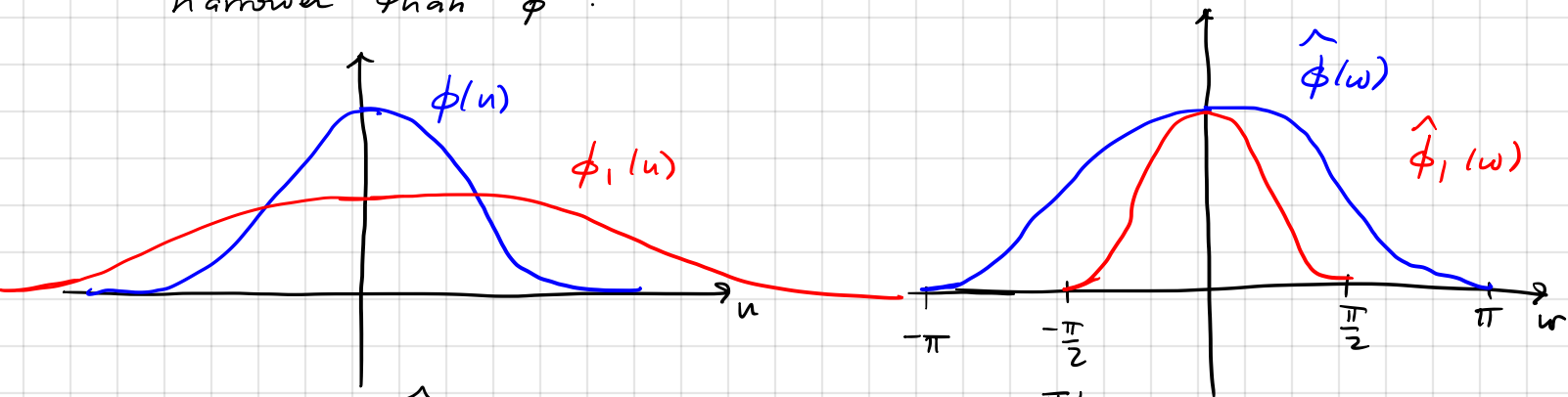
Here is an example:



Notice the low pass filter is quite small, and is essentially a 3×3 filter (as in the VGG network). Because it is a low pass filter, it replaces every pixel in the image with a weighted average of the pixels in a 3×3 neighborhood around the central pixel. The resulting image is similar to the original image, but is a slightly smoothed / blurred version. It retains most of the frequency content of the original image, but loses some of the high frequencies. If one looks carefully at the two images, one can see some of the very fine detail is lost.

We can dilate ϕ to enlarge it, which will allow us to smooth the signal more drastically:

$\phi_J(u) = 2^{-J} \phi(2^{-J}u) \Rightarrow \hat{\phi}_J(\omega) = \hat{\phi}(2^J \omega)$
 For $J > 0$, ϕ_J will be wider than ϕ but $\hat{\phi}_J$ will be narrower than $\hat{\phi}$:

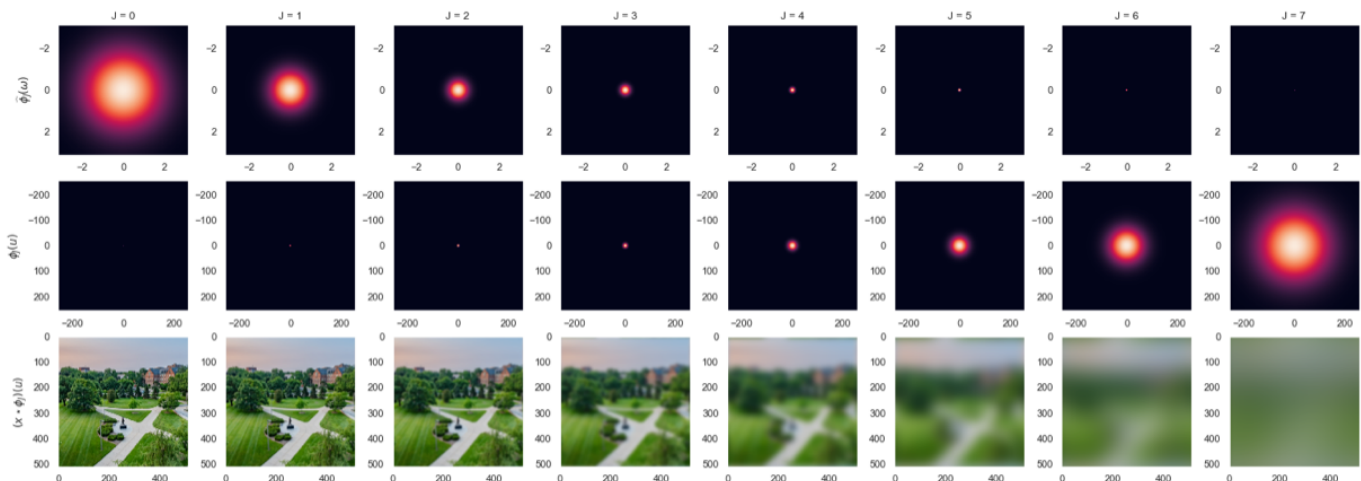


Notice: $\hat{\phi}_J(\omega) = 0$ for all $|\omega| > \pi/2^J$

When we filter x with ϕ_J , that is we compute $x \# \phi_J$, we blur x even more. Indeed, since

$$\widehat{(x \# \phi_J)}(\omega) = \hat{x}(\omega) \hat{\phi}_J(\omega)$$

we see that $x \# \phi_J$ only retains the frequencies of x contained in $|\omega| \leq \pi/2^J$. Here is the filtering of the same image as before for different dyadic scales 2^J :



The top row is the Fourier transform of ϕ_J , $\hat{\phi}_J(\omega)$. The middle row is $\phi_J(u)$. The bottom row is $(x * \phi_J)(u)$. The scales range over $0 \leq J \leq 7$. The low pass function here is a Gaussian,

$$\phi(u) = \frac{1}{2\pi\sigma^2} e^{-|u|^2/2\sigma^2} \Rightarrow \hat{\phi}(\omega) = e^{-\sigma^2|\omega|^2/2}$$

We choose $\sigma = 3/4$. Notice as the scale increases, ϕ_J becomes larger and $\hat{\phi}_J(\omega)$ becomes smaller. We average in larger and larger neighborhoods, which progressively blurs the image more and more. From a frequency perspective, we retain fewer and fewer frequencies in the original image x . Visually, the increased blur makes it harder to distinguish translations and small deformations of the image. The following theorem quantifies this for translations:

Theorem (Mallat 2012): There is a constant $C > 0$, depending on ϕ , such that for all $t \in \mathbb{R}$ and $x \in L^2(\mathbb{R})$:

$$\|x * \phi_J - x_t * \phi_J\|_2 \leq C \cdot 2^{-J} \cdot |t| \cdot \|x\|_2$$

This theorem shows the representation $\Xi(x) = x * \phi_J$ is translation invariant up to the scale 2^J . But how does this relate to neural networks? To understand this, we will need to appeal to results from sampling theory:

Theorem (Shannon - Nyquist): Suppose $\hat{x}(\omega) = 0$ for all $|\omega| > \pi/s$ for some $s > 0$. Then x can be recovered from the downsampled version of x defined by

$$x_d(n) = x(sn), \quad n \in \mathbb{Z}$$

Notice if $s=1$ then $\hat{x}(\omega) = 0$ for all $|\omega| > \pi$ and we can recover $x: \mathbb{R} \rightarrow \mathbb{R}$ from $x_d: \mathbb{Z} \rightarrow \mathbb{R}$, $x_d(n) = x(n)$. This is one way to think of a natural image. The underlying scene is x and the image is x_d , which has been sampled along "integer" pixels. Since high resolution images are good representations of the scene, we can interpret this as $\hat{x}(\omega) = 0$ for $|\omega| > \pi$ (warning: If you are comparing different cameras, there is some danger in this)

Since we assumed $\hat{\phi} = 0$ for $|\omega| > \pi$, this is why $x * \phi$, depicted earlier, is a good approximation of x since it retains nearly all of $\hat{x}(\omega)$ (only the corners are lost). On the other hand, this is intuitively clear since ϕ averaged over a 3×3 window.

Notice that for $\phi_J(u) = 2^{-J} \phi(2^{-J}u) \Rightarrow \hat{\phi}_J(\omega) = \hat{\phi}(2^J \omega)$ we have $\hat{\phi}_J(\omega) = 0$ for all $|\omega| > \pi/2^J$. Since $(x * \phi_J)(\omega) = \hat{x}(\omega) \hat{\phi}_J(\omega)$ this means that $(x * \phi_J)(\omega) = 0$ for all $|\omega| > \pi/2^J$. Therefore we can represent $x * \phi_J$ via:

$$(x * \phi_J)_d(n) = (x * \phi_J)(2^J n)$$

Thus we downsample $x * \phi_J$ by a factor 2^J . This is not quite like CNNs which usually pool in factors of 2. Also, ϕ is small, but ϕ_J is larger by a factor 2^J . So there are some differences, at least it would appear so. In fact things are not so different. Indeed the following implements $x * \phi_J$:

$$x \rightarrow \underbrace{x * \phi_1}_{\text{convolve } x \text{ with } \phi_1} \downarrow_2 \rightarrow (x * \phi_1 \downarrow_2) * \phi_1 \downarrow_2 \rightarrow \dots \text{ J times}$$

convolve x with ϕ_1 (remainder $\hat{\phi}_1(\omega) = 0$ for all $|\omega| > \pi/2$) and downsample by a factor of 2

Note: ϕ_1 is essentially 7×7

Therefore we can implement the translation invariant operator $x * \phi_J$ by composing convolution with ϕ_1 and downsampling by a factor of 2, J times. This is a simple type of CNN with same single filter at each layer and no nonlinearities.

Okay, so we see that $\mathcal{T}(x) = x * \phi_J$ is a translation invariant representation of x and can be viewed as simple CNN. On the other hand, we know

$$(x * \phi_J)(\omega) = \hat{x}(\omega) \hat{\phi}_J(\omega) \neq 0 \text{ only for } |\omega| \leq \pi/2^J$$

So we have lost a lot of \hat{x} and thus x (indeed recall the pictures of $x * \phi_J$ which were very blurry).

To recover the lost information we turn to something called a wavelet transform. A wavelet $\psi: \mathbb{R} \rightarrow \mathbb{R}$ or $\psi: \mathbb{R} \rightarrow \mathbb{C}$ is a localized, oscillating waveform with zero average. The last property means

$$\hat{\psi}(0) = \int_{\mathbb{R}} \psi(u) du = 0$$

Thus, unlike the low pass filter ϕ_J for which $\sup_{\omega} |\hat{\phi}_J(\omega)| = \hat{\phi}_J(0)$, the wavelet ψ has its frequency support concentrated around a frequency (or frequencies) away from zero. Like the low pass filter ϕ , we can dilate ψ :

$$\psi_j(u) = 2^{-j} \psi(2^{-j}u) \Rightarrow \hat{\psi}_j(\omega) = \hat{\psi}(2^j \omega)$$

A wavelet transform computes:

$$J \geq 1, \quad W_J x = \left\{ x * \phi_J(u), x * \psi_j(u) : u \in \mathbb{R}, 1 \leq j \leq J \right\}$$

In other words, in addition to averaging over x with $x * \phi_J$, we filter x with J smaller wavelets that recover the details in x lost by $x * \phi_J$. In terms of frequencies, $x * \phi_J$ keeps the low frequencies of x (hence ϕ_J is a low pass filter) while $\{x * \psi_j\}_{1 \leq j \leq J}$ keeps the high frequencies of x (hence the ψ_j filters are called high pass filters).

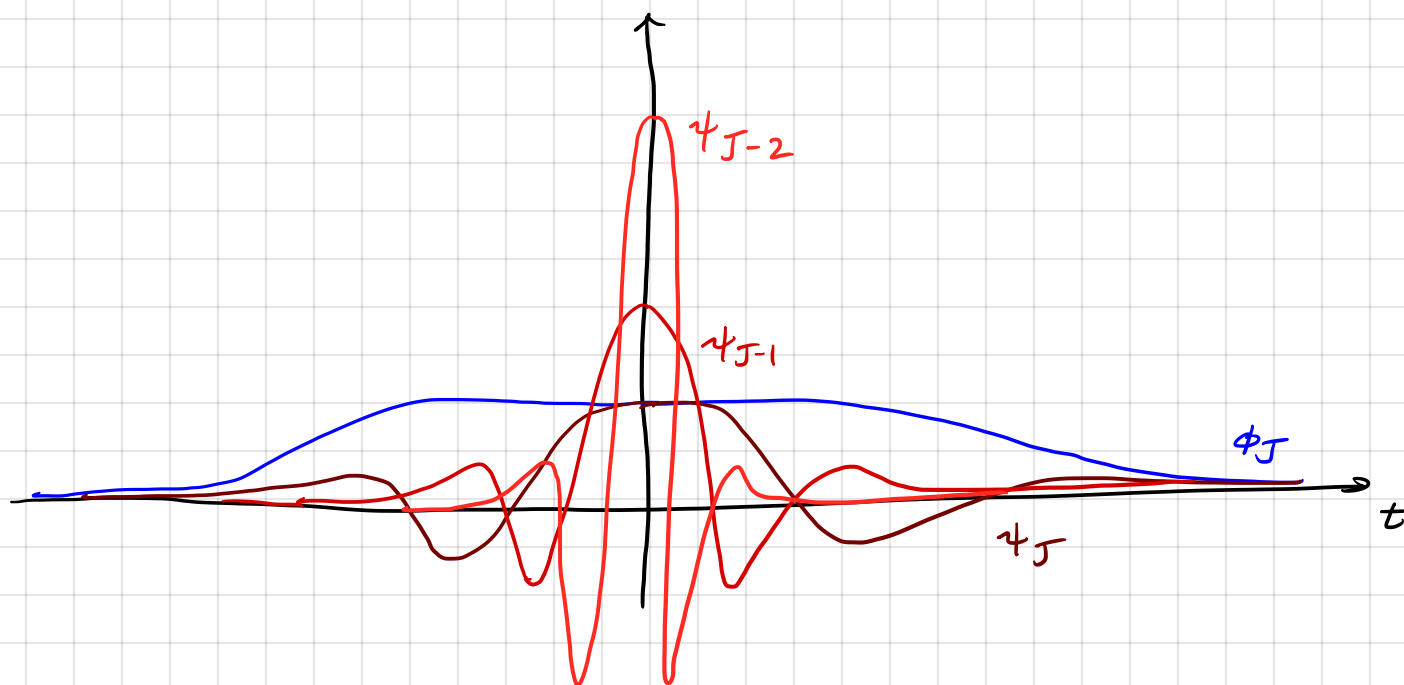
Suppose, as we observed for natural images, that $\hat{x}(\omega) = 0 \quad \forall |\omega| > \pi$.
If

$$0 < A \leq |\hat{\phi}_J(\omega)|^2 + \sum_{j=1}^J |\hat{\psi}_j(\omega)|^2 \leq B < +\infty \quad \text{for all } \omega \in [-\pi, \pi]$$

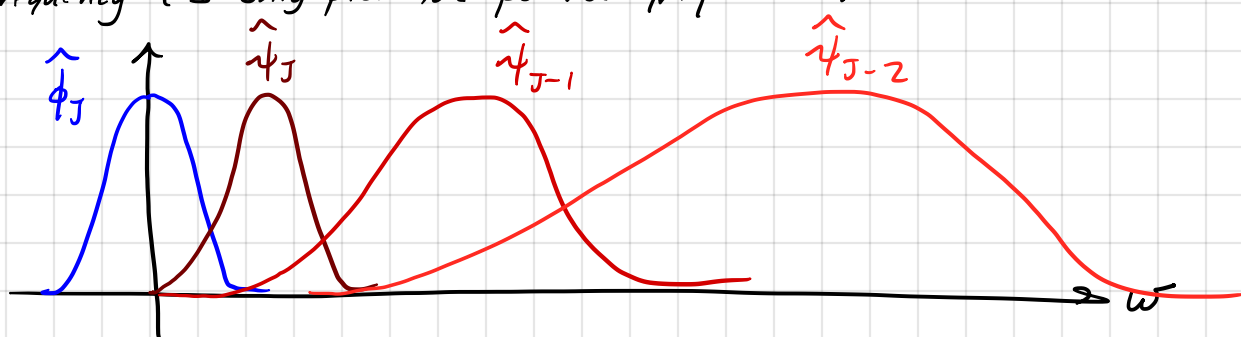
This means all the frequencies are covered by our low pass ϕ_J and wavelets $\{\psi_j\}_{1 \leq j \leq J}$

then $W_J x = \{x * \phi_J, x * \psi_j : 1 \leq j \leq J\}$ is invertible, meaning knowing $W_J x$ is as good as knowing x . The proof of this is based on the fact that we stated earlier, which is that knowing $\hat{x}(\omega)$ is as good as knowing $x(u)$.

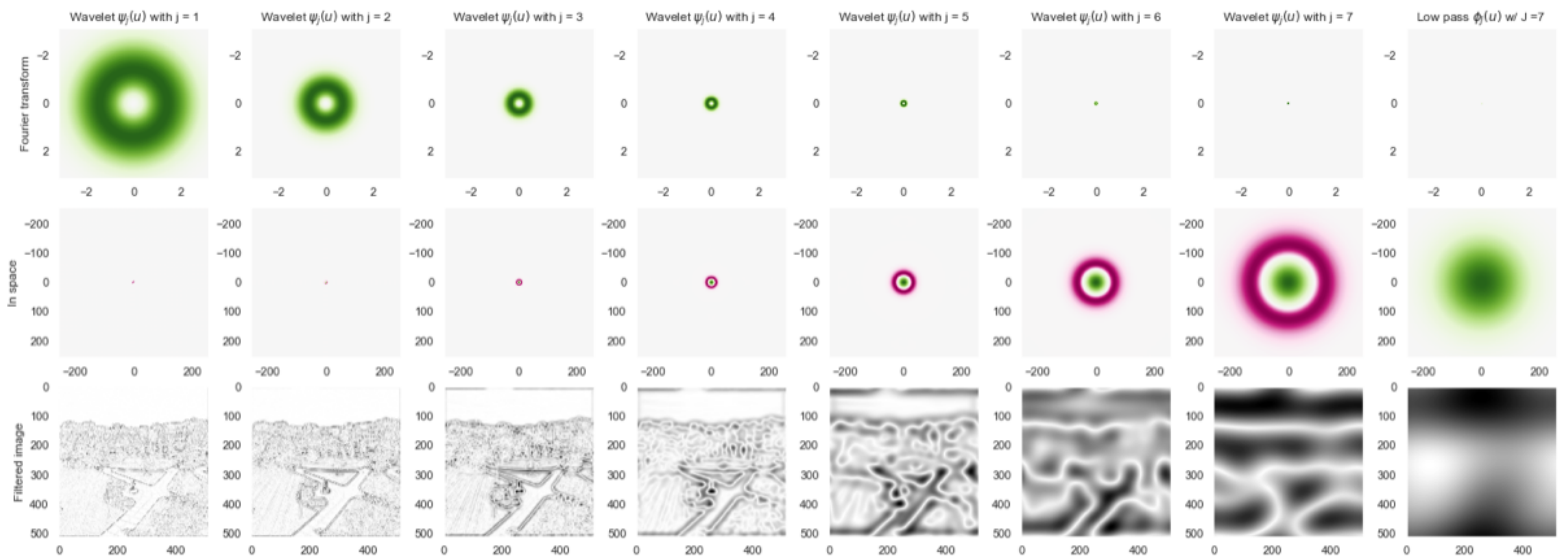
In time/space we have the following plots:



And in frequency (I only plot the positive frequencies):



Here are pictures in 2D on the same image as before :



In the first 2 rows green is positive, pink is negative, white is zero. In that last row white is zero and black in max positive value.

The first seven columns are wavelets going from small scale in space to large scale in space, so $\psi_j(u)$ (2nd row) and $\hat{\psi}_j(\omega)$ (1st row) for $1 \leq j \leq J=7$. The last column is the low pass filter $\phi_J(u)$ (2nd row) and $\hat{\phi}_J(\omega)$ (1st row).

We see in frequency the wavelets capture the high frequencies that the low pass misses. These wavelets are localized oscillating waveforms where the oscillations flow radially out of the center.

When computing the filtration $x \star \psi_j$ (the 3rd row), the small wavelets act as edge detectors; here we plot :

$$[|x_r \star \psi_j|^2 + |x_g \star \psi_j|^2 + |x_b \star \psi_j|^2]^{1/2}$$

The larger wavelets capture larger scale information in the image. In this example, ϕ is the same as before and

$$\Delta = \text{Laplacian} \longrightarrow \psi(u) = -(\Delta g)(u), \quad g(u) = \frac{1}{2\pi\alpha^2} e^{-|u|^2/2\alpha^2}, \quad \alpha = \frac{1}{2}$$

$$\Rightarrow \hat{\psi}(\omega) = |\omega|^2 \hat{g}(\omega), \quad \hat{g}(\omega) = e^{-\alpha^2 |\omega|^2/2}$$

Like ϕ_J , we can also implement $x \star \psi_j$ with a simple CNN :

$$x \mapsto x \star \phi_1 \downarrow_2 \mapsto (x \star \phi_1 \downarrow_2) \star \phi_1 \downarrow_2 \mapsto \dots \mapsto \underbrace{((x \star \phi_1 \downarrow_2) \star \phi_1 \downarrow_2) \star \dots \star \phi_1 \downarrow_2)}_{j-1 \text{ times}} \star \psi_1$$

where ϕ_1 and ψ_1 are very small filters.

$$x \star \psi_j$$

But what about translation invariance? Recall $x * \phi_J$ is translation invariant up to the scale 2^J . We also have

$$\int_{\mathbb{R}} x * \phi_J(u) du = \int_{\mathbb{R}} x(u) du$$

On the other hand

$$\int_{\mathbb{R}} \psi(u) du = 0 \Rightarrow \int_{\mathbb{R}} \psi_j(u) du = 0 \Rightarrow \int_{\mathbb{R}} x * \psi_j(u) du = 0 \leftarrow \begin{array}{l} \text{tells you} \\ \text{nothing} \\ \text{about } x \end{array}$$

What if we convolved $x * \psi_j$ with ϕ_J ? We know $x * \phi_J$ is translation invariant so $x * \psi_j * \phi_J$ is also translation invariant.

But if $1 = |\hat{\phi}_J(\omega)|^2 + \sum_{j=1}^J |\hat{\psi}_j(\omega)|^2$ and $\hat{\phi}_J(0) = 1$ then

the support of $\hat{\phi}_J(\omega)$ must overlap very little with the support of $\hat{\psi}_j(\omega)$. But:

$$(x * \psi_j * \phi_J)(\omega) = \hat{x}(\omega) \hat{\psi}_j(\omega) \hat{\phi}_J(\omega) \approx 0$$

which is also not helpful!

Therefore we need something nonlinear. The idea is $x * \psi_j$ captures the high frequencies of x . We need to "push" this high frequency information down to the low frequencies so we can obtain a nontrivial, translation invariant representation of x . The wavelet scattering transform uses the absolute value / modulus operator. Some of these results could potentially be adapted to ReLU. We also note:

$$|z| = \text{ReLU}(z) + \text{ReLU}(-z) \quad \text{for } z \in \mathbb{R}$$

With the absolute value / modulus we send:

$$x * \psi_j \mapsto |x * \psi_j| = x_j$$

These new functions x_j have Fourier transform $\hat{x}_j(\omega)$ with some support at $\omega=0$. Indeed:

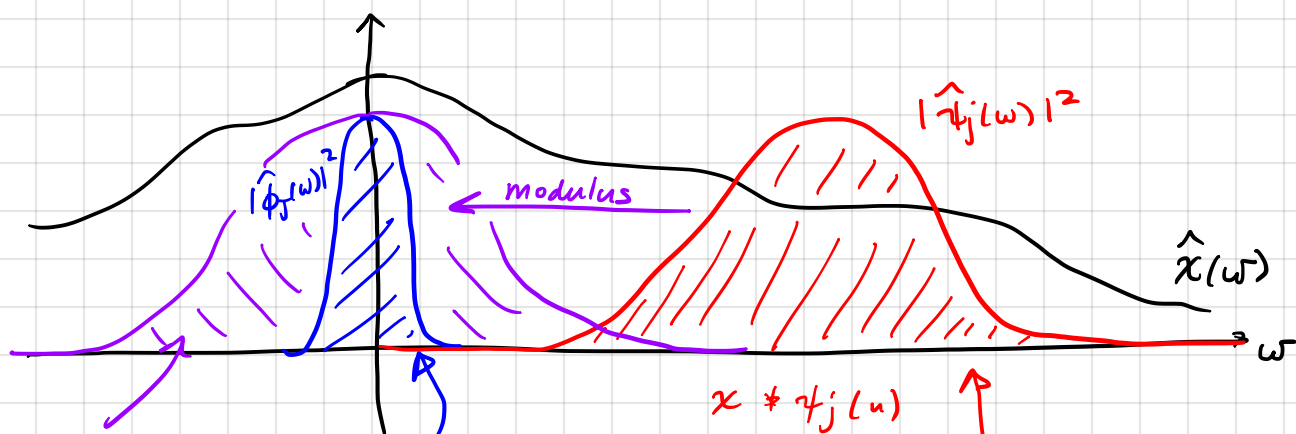
$$\hat{x}_j(0) = \int_{\mathbb{R}} |x * \psi_j(u)| du \geq 0$$

Therefore each function $|x * \psi_j| * \phi_J \neq 0$ and using the previous theorem each function $|x * \psi_j| * \phi_J$ is translation invariant up to the scale 2^J . Thus far we have:

$$\Phi(x) = S_J^2(x) = \{ x * \phi_J, |x * \psi_j| * \phi_J : 1 \leq j \leq J \}$$

\uparrow Wavelet scattering of x w/ two wavelet layers is a translation invariant representation of x up to the scale 2^J .

The idea in frequency is the following:



frequency support
of $|x * \psi_j(u)|$

Note that

$|x * \psi_j| * \phi_J(u)$

keeps the

$x * \phi_J(u)$ keeps
frequencies in
this range and is
translation invariant

$x * \psi_j(u)$
keeps frequencies
in this range

frequencies in the intersection of the blue bump and the purple bump.
The coefficients $|x * \psi_j| * \phi_J(u)$ are translation invariant up to the
scale 2^J and retain some of the information from $x * \psi_j(u)$.

On the other hand, the above picture shows that $x * \phi_J$ and
 $|x * \psi_j| * \phi_J$ do not capture all of $\hat{x}(\omega)$ and hence all of $x(u)$.

Indeed, when we computed $|x * \psi_j| * \phi_J(u)$ we lost information
from $|x * \psi_j(u)|$ just as $x * \phi_J(u)$ lost information from $x(u)$.

In order to recover this lost high frequency information we can iterate
the filterings with wavelets:

$$|x * \psi_{j_1}| * \psi_{j_2}, \quad 1 \leq j_1, j_2 \leq J$$

Since

$$W_J |x * \psi_{j_1}| = \{ |x * \psi_{j_1}| * \phi_J, |x * \psi_{j_1}| * \psi_{j_2} : 1 \leq j_2 \leq J \}$$

↑
wavelet transform

and since knowing $W_J |x * \psi_{j_1}|$ is as good as knowing $|x * \psi_{j_1}|$,
we see that $|x * \psi_{j_1}| * \psi_{j_2}$ indeed recovers the information lost in
only computing $|x * \psi_{j_1}| * \phi_J$.

However, similar to before, $|x * \psi_{j_1}| * \psi_{j_2}$ is translation equivariant
but not translation invariant and

$$|x * \psi_{j_1}| * \psi_{j_2} * \phi_J \approx 0$$

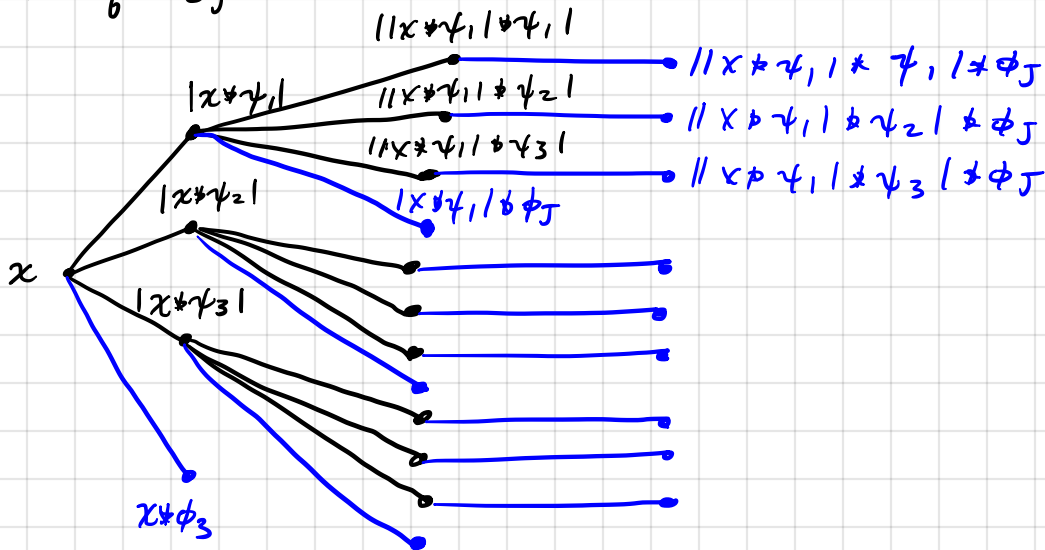
Therefore we need to apply another nonlinearity (another absolute
value / modulus) and then apply the low pass filter:

$$||x * \psi_{j_1}| * \psi_{j_2}| * \phi_J$$

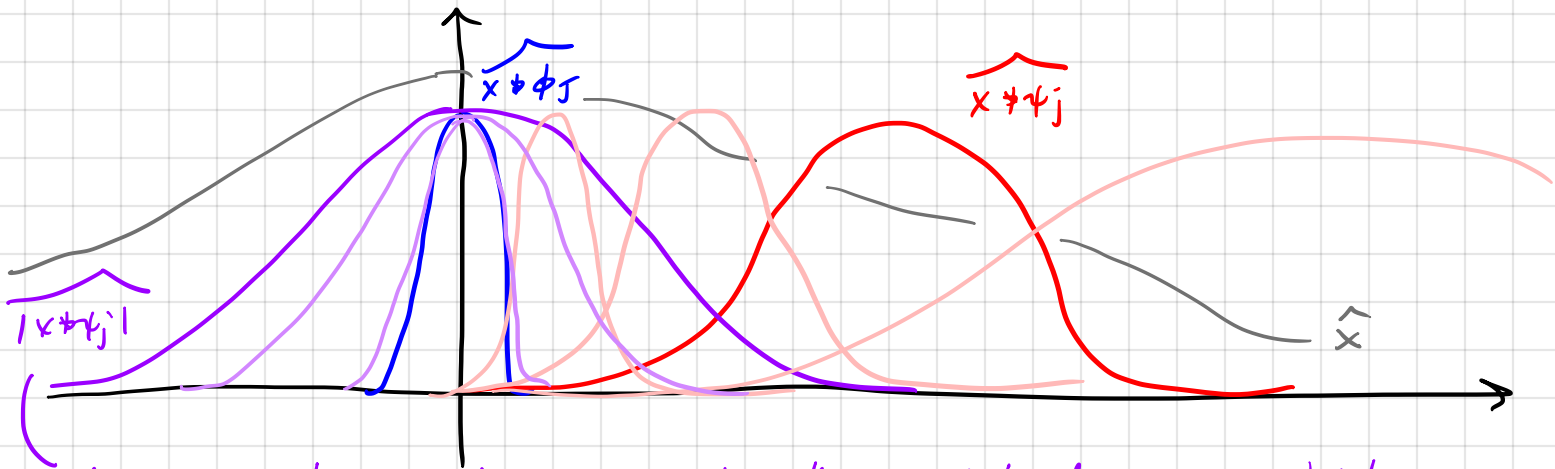
Our representation of x is now:

$$\overline{\Phi}(x) = S_J^3(x) = \left\{ \begin{array}{l} x * \phi_J \\ |x * \psi_{j_1}| * \phi_J \\ ||x * \psi_{j_1}| * \psi_{j_2}| * \phi_J \\ \vdots \end{array} \right\}$$

We call $S_J^3(x)$ a wavelet scattering transform. Here is a diagram of $S_J^3(x)$:



In frequency space, here is what is happening:

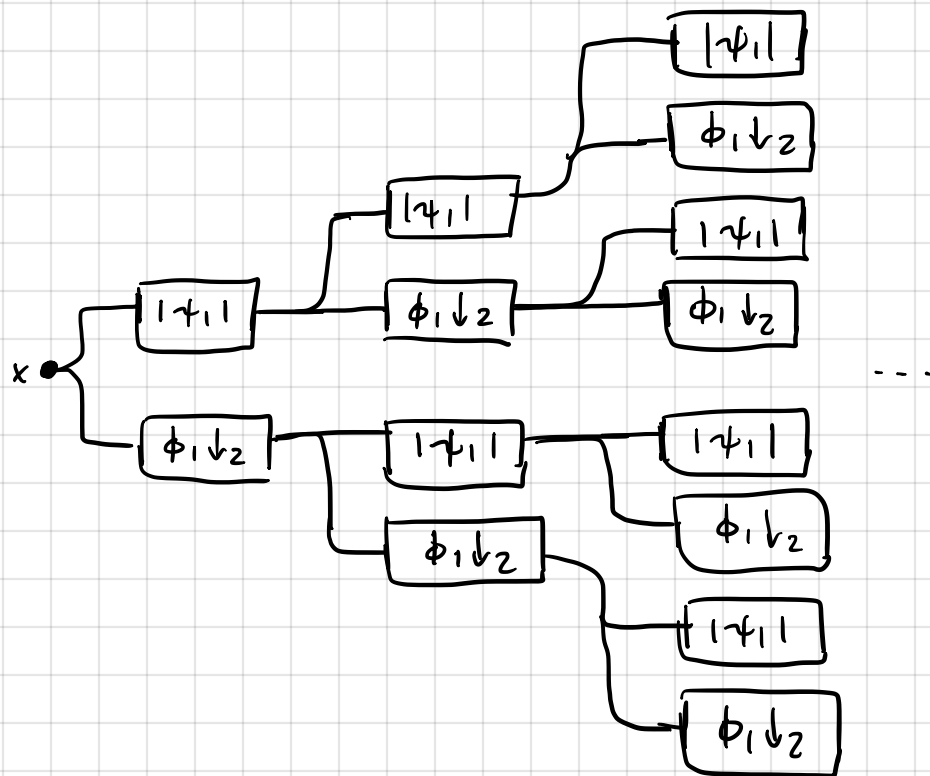


decompose the purple curve using the pink/red curves, which corresponds to computing $|x * \psi_{j_1}| * \psi_{j_2}$. Then push that frequency information down with $||x * \psi_{j_1}| * \psi_{j_2}|$.

One can see that even $S_J^3(x)$ loses some information from $\hat{x}(\omega)$ and hence $x(u)$. Therefore we can keep going. An $(m+1)$ -wavelet layer transform computes:

$$S_J^{m+1}(x) = \left\{ \begin{array}{l} x * \phi_J, \\ |x * \psi_{j_1}| * \phi_J \\ ||x * \psi_{j_1}| * \psi_{j_2}| * \phi_J \\ \vdots \\ |||x * \psi_{j_1}| * \psi_{j_2}| * \dots * \psi_{j_m}| * \phi_J \end{array} \right\} \quad \left. \begin{array}{l} : 1 \leq j_i \leq J \\ \text{for all } 1 \leq i \leq m \end{array} \right\}$$

The wavelet scattering transform is a mathematical model for a convolutional neural network. Recalling our discussion on sampling theory and downsampling, we can also write it in terms of only small filters:



where $x \rightarrow \boxed{\phi_1 \downarrow_2}$ computes $x * \phi_1 \downarrow_2$

and $x \rightarrow \boxed{|\psi_1|}$ computes $|x * \psi_1|$

Now let us discuss some additional theoretical properties of the wavelet scattering transform. We will assume the wavelets and low pass filter perfectly cover the frequency axis, meaning:

$$|\hat{\phi}_J(\omega)|^2 + \sum_{j \in J} |\hat{\psi}_j(\omega)|^2 = 1 \quad \forall \omega \in \mathbb{R} \quad (\psi \text{ real valued})$$

$$|\hat{\phi}_J(\omega)|^2 + \sum_{j \in J} |\hat{\psi}_j(\omega)|^2 = 2 \quad \forall \omega \geq 0 \quad (\psi \text{ complex valued})$$

We also define the norm of $S_J^{m+1}(x)$ as:

$$\|S_J^{m+1}(x)\|^2 = \|x * \phi_J\|_2^2 + \sum_{\ell=1}^{m+1} \| |x * \psi_{j_1}| * \psi_{j_2} | * \dots * \psi_{j_\ell} | * \phi_J \|_2^2$$

Individually each function in $S_J^{m+1}(x)$ is translation invariant up to to the scale 2^J . In fact all of $S_J^{m+1}(x)$ is translation invariant:

Theorem (Mallat 2012): For $x \in L^2(\mathbb{R})$, let $x_t(u) = x(u-t)$. Then:

$$\|S_J^{m+1}(x) - S_J^{m+1}(x_t)\| \leq C \cdot m \cdot 2^{-J} \cdot |t| \cdot \|x\|_2$$

Note the linear scaling in the number of layers, $m+1$, is better than one would get by counting up all of the functions in $S_J^{m+1}(x)$. The bound also holds even with an infinite number of functions, which can be thought of as infinitely wide layers.

Our other goal was stability to diffeomorphisms. For this we have:

Theorem (Mallat 2012): Let $x \in L^2(\mathbb{R})$ and $\tau \in C^2(\mathbb{R})$ with $\|\tau'\|_\infty < 1/2$ and $x_\tau(u) = x(u - \tau(u))$. Then:

$$\|S_J^{m+1}(x) - S_J^{m+1}(x_\tau)\| \leq C \cdot m \cdot \left[2^{-J} \|\tau\|_\infty + \underset{\uparrow}{J \cdot \|\tau'\|_\infty} + \|\tau''\|_\infty \right] \|x\|_2$$

Can be replaced by other things, such as R where $\text{supp}(x) \subseteq B_{2R}(0)$

These two theorems show $\Phi(x) = S_J^{m+1}(x)$ is invariant to translations and stable to diffeomorphisms. It is also $L^2(\mathbb{R})$ stable:

Theorem (Mallat 2012): Let $x, \tilde{x} \in L^2(\mathbb{R})$. Then:

$$\|S_J^{m+1}(x) - S_J^{m+1}(\tilde{x})\| \leq \|x - \tilde{x}\|_2$$

If the wavelet ψ satisfies additional constraints, then

$$\lim_{m \rightarrow \infty} \|S_J^{m+1}(x)\| = \|x\|_2$$

L^2 stability is complementary to translation invariance and stability diffeomorphisms, and is important in its own right.

The energy preservation shows that the transform neither creates nor destroys "mass," here as measured by $\|x\|_2$. It shows that the collection of functions in $S_J^{\infty}(x)$ partition the energy of x .

The one thing we have not addressed yet is the stacking of filters. Let us start with rotations. A directional filter is a filter that oscillates in a certain direction. We can model such filters as:

$$u \in \mathbb{R}^2, \quad \psi(u) = g(|u|) e^{i \xi \cdot u}, \quad |u| = [u(1)^2 + u(2)^2]^{1/2}$$

complex valued

$$\text{or } \left. \begin{aligned} \psi(u) &= g(|u|) \cos(\xi \cdot u) \\ \psi(u) &= g(|u|) \sin(\xi \cdot u) \end{aligned} \right\} \text{real valued}$$

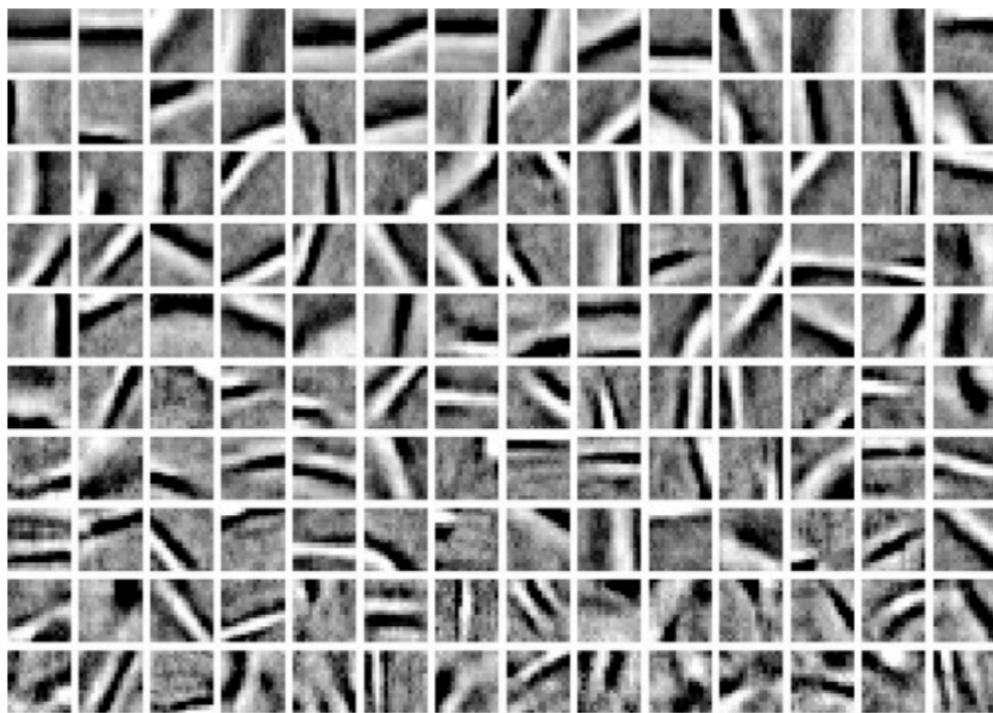
(just the real and imaginary parts of the complex valued filter)

The window g is often non-negative, e.g., a Gaussian

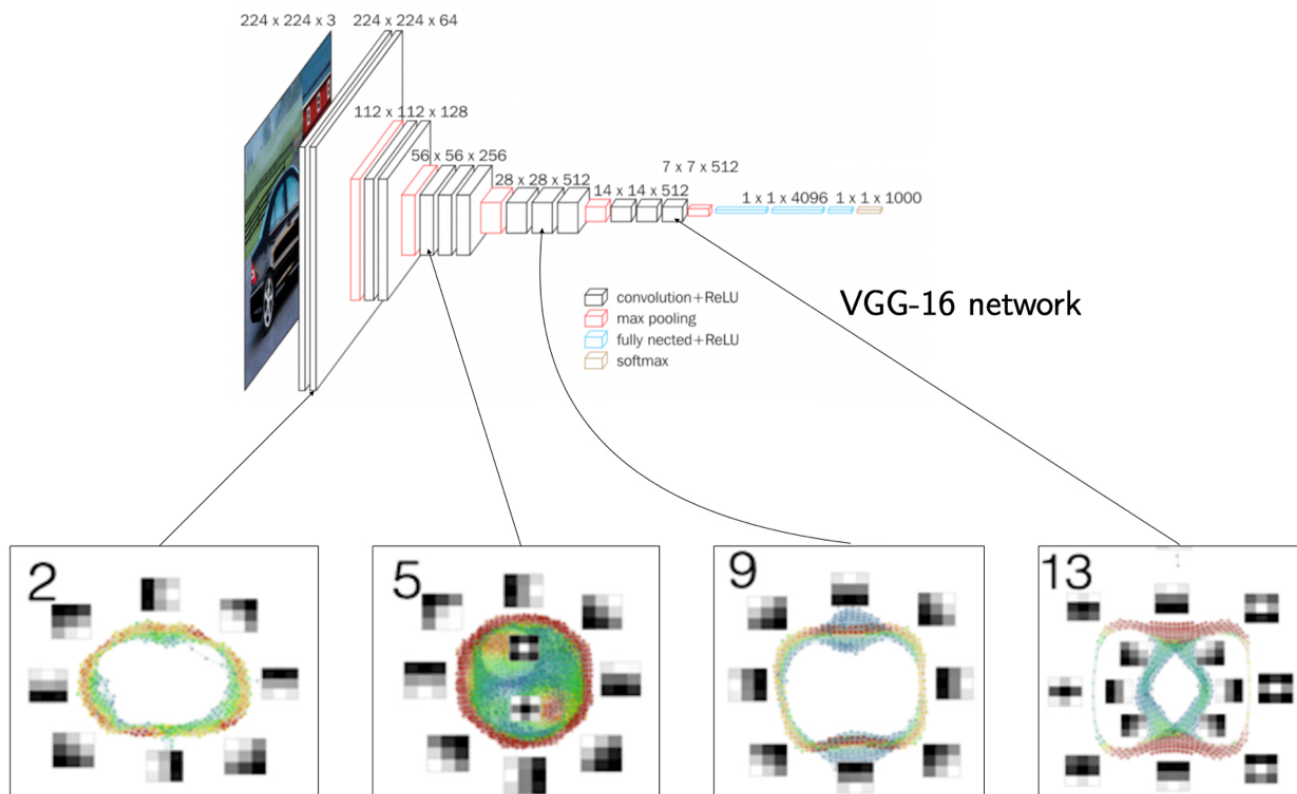
$$g(|u|) = \frac{1}{2\pi\sigma^2} e^{-|u|^2/2\sigma^2}$$

The filter ψ oscillates in the direction $|\xi|$. For example, if $\xi = (\xi_1, 0)$ it will oscillate in the horizontal direction, if $\xi = (0, \xi_2)$ it will oscillate in the vertical direction, and if $\xi = (\xi_0, \xi_0)$ it will oscillate at a 45° angle.

Directional filters are useful for image processing because they isolate edges and other patterns in the image only in certain directions. For example, a directional filter ψ that oscillates horizontally will pick up on a vertical edge. It turns out that many dictionary learning algorithms and deep networks learn directional filters (at least in the early layers for deep networks), so they are a good model for filters.



Filters learned through dictionary learning (Figure taken from A Wavelet Tour of Signal Processing)



The VGG network (top) and the filters learned at selected layers (bottom), organized using methods from topological data analysis [organizations by Carlsson and Gabrielsson 2018]

In both cases we can see many directional filters at different orientations. A striking ^{example} is panel 2 and panel 9 of the VGG network. These panels show that often the filters are essentially the same filter, that has been rotated:

$$\begin{aligned}\psi_\theta(u) &= \psi(R_\theta^{-1}u) = g(|R_\theta^{-1}u|) e^{i\vec{\zeta} \cdot R_\theta^{-1}u} \\ &= g(|u|) e^{iR_\theta \vec{\zeta} \cdot u}\end{aligned}$$

$R_\theta = \begin{pmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{pmatrix}$
 \uparrow
 2x2 rotation matrix

Notice ψ_θ oscillates in the direction $R_\theta \vec{\zeta}/|\vec{\zeta}|$, but otherwise is like the original filter ψ . Let

$$\Theta = \{ 2\pi m/M : 0 \leq m < M \}$$

Then $\{\psi_\theta\}_{\theta \in \Theta}$ defines a stack of M filters that are rotations of a base filter ψ . Let us compute the convolution of an image x with this stack. Let

$$x_\varphi(u) = x(R_\varphi^{-1}u)$$

be the rotation of x by $\varphi \in [0, 2\pi)$.

$$\text{Then: } (x_\varphi * \psi_\theta)(u) = \int_{\mathbb{R}^2} x(R_\varphi^{-1}v) \psi(R_\theta^{-1}u - R_\theta^{-1}v) dv$$

$$t = R_\varphi^{-1}v$$

$$= \int_{\mathbb{R}^2} x(t) \psi(R_\theta^{-1}u - R_\theta^{-1}R_\varphi t) dt$$

$$= \int_{\mathbb{R}^2} x(t) \psi(\underbrace{R_\theta^{-1}R_\varphi}_{(R_{\theta-\varphi})^{-1}}(R_\varphi^{-1}u - t)) dt$$

$$(R_{\theta-\varphi})^{-1}$$

$$= \int_{\mathbb{R}^2} x(t) \psi(R_{\theta-\varphi}^{-1}(R_\varphi^{-1}u - t)) dt$$

$$= (x * \psi_{\theta-\varphi})(R_\varphi^{-1}u)$$

We see that $x * \psi_\theta$ is not equivariant with respect to rotations, but rather two things are happening:

$$(x_\varphi * \psi_\theta)(u) = (x * \psi_{\theta-\varphi})(\underbrace{R_\varphi^{-1}u}_{\text{rotation of filtered image (like translations)}})$$

rotation of filtered image
(like translations)

Transport of information along the stack of filters
(different than translations).

With a one layer CNN we would apply a pointwise nonlinearity:

$$\sigma(x_\varphi * \psi_\theta)(u) = \sigma(x * \psi_{\theta-\varphi})(R_\varphi^{-1}u)$$

and we can aggregate information across filters:

$$\Phi(x_\varphi)(u) = \sum_{\theta \in \Theta} \sigma(x_\varphi * \psi_\theta)(u) = \sum_{\theta \in \Theta} \sigma(x * \psi_{\theta-\varphi})(R_\varphi^{-1}u)$$

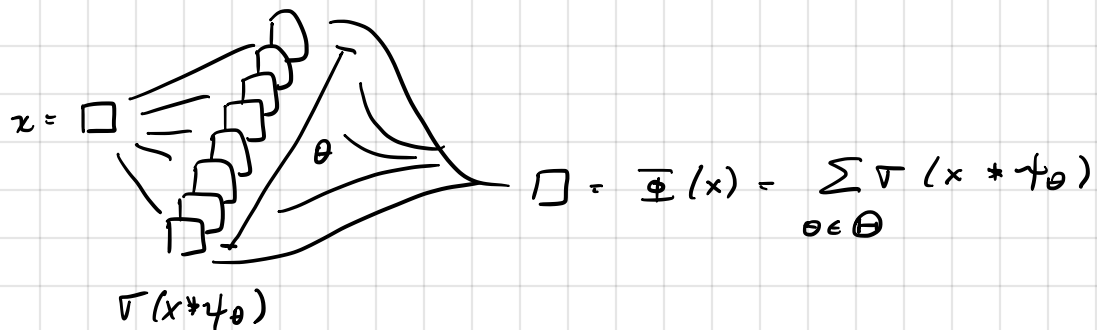
$$\approx \sum_{\theta \in \Theta} \sigma(x * \psi_\theta)(R_\varphi^{-1}u)$$

Thus the transformation

$$x(u) \mapsto \Phi(x)(u) = \sum_{\theta \in \Theta} \sigma(x * \psi_{\theta})(u)$$

is equivariant to rotations since $\Phi(x_{\varphi})(u) \approx \Phi(x)(R_{\varphi}^{-1}u)$ (as we showed on the previous page) and it is equivariant with respect to translations since it uses convolution operators.

Here is a diagram:



On the other hand, we summed over Θ , which removes much of our directional information. A two layer CNN handles this better. In this case, the output of the first layer is the M channel representation of x given by:

$$x^{(1)} = (x_{\theta_1}^{(1)})_{\theta_1 \in \Theta} = (\sigma(x * \psi_{\theta_1}))_{\theta_1 \in \Theta}$$

Now we define a collection of M^2 filters for the second layer as:

$$\psi_{\theta_1, \theta_2}(u) = \psi_{\theta_1 + \theta_2}(u), \quad \theta_1, \theta_2 \in \Theta$$

Following our earlier discussion on stacks of filters and the VGG network, we compute in the second layer:

$$x_{\theta_2}^{(2)} = \sigma \left(\sum_{\theta_1 \in \Theta} x_{\theta_1}^{(1)} * \psi_{\theta_1, \theta_2} \right) = \sigma \left(\sum_{\theta_1 \in \Theta} \sigma(x * \psi_{\theta_1}) * \psi_{\theta_1 + \theta_2} \right)$$

Let x_{φ} be the rotation of x . Then: by our previous calculation

$$\begin{aligned} \sigma(x_{\varphi} * \psi_{\theta_1}) * \psi_{\theta_1 + \theta_2}(u) &= \sigma(x * \psi_{\theta_1 - \varphi})_{\varphi} * \psi_{\theta_1 + \theta_2}(u) \\ &= \sigma(x * \psi_{\theta_1 - \varphi}) * \psi_{\theta_1 - \varphi + \theta_2}(R_{\varphi}^{-1}u) \end{aligned}$$

Therefore :

$$\begin{aligned}
 (\chi_\varphi)_{\theta_2}^{(2)}(u) &= \mathcal{T} \left(\sum_{\theta_1 \in \Theta} \sigma(x_\varphi * \psi_{\theta_1}) * \psi_{\theta_1 + \theta_2} \right)(u) \\
 &= \mathcal{T} \left(\sum_{\theta_1 \in \Theta} \mathcal{T}(x * \psi_{\theta_1 - \varphi}) * \psi_{\theta_1 - \varphi + \theta_2} \right)(R_\varphi^{-1} u) \\
 &\approx \mathcal{T} \left(\sum_{\theta_1 \in \Theta} \mathcal{T}(x * \psi_{\theta_1}) * \psi_{\theta_1 + \theta_2} \right)(R_\varphi^{-1} u) \\
 &= \chi_{\theta_2}^{(2)}(R_\varphi^{-1} u)
 \end{aligned}$$

Thus $\chi_{\theta_2}^{(2)}$ is equivariant with respect to translations and rotations, and unlike the 1-layer CNN we have a stack of M such maps given by

$$\chi^{(2)} = (\chi_{\theta_2}^{(2)})_{\theta_2 \in \Theta}$$

Remark : We can replace the rotation group with another group G .
Then

$$\chi_g(u) = \chi(g^{-1} \cdot u) \quad \text{where } g \in G \text{ and } g \cdot u \in \mathbb{R}^2 \text{ is the group action of } g \text{ on } u.$$

Then we have a stack of filters:

$$\{\psi_g\}_{g \in G}, \quad \psi_g(u) = \psi(g^{-1} \cdot u)$$

Let $h \in G$ as well. Assume $|g \cdot u| = |u|$. Then:

$$\begin{aligned}
 \chi_h * \psi_g(u) &= \int_{\mathbb{R}^2} \chi(h^{-1} \cdot v) \psi(g^{-1} \cdot u - g^{-1} \cdot v) dv \\
 &\quad t = h^{-1} \cdot v \\
 &= \int_{\mathbb{R}^2} \chi(t) \psi(g^{-1} \cdot u - g^{-1} \cdot h \cdot t) dt \\
 &= \int_{\mathbb{R}^2} \chi(t) \psi(g^{-1} h (h^{-1} \cdot u - t)) dt \\
 &= \int_{\mathbb{R}^2} \chi(t) \psi_{h^{-1}g}(h^{-1}u - t) dt
 \end{aligned}$$

Now we can
apply similar
ideas



$$= \chi * \psi_{h^{-1}g}(h^{-1}u) = (\chi * \psi_{h^{-1}g})_h(u)$$

When we train CNNs, they implicitly learn the groups G , and corresponding stacks of filters, over these groups. In practice, the mathematical language of group theory may be too limited.

Remark 2: The groups can be enlarged through the layers.
See, e.g., "Understanding deep convolutional networks"
by Mallat (2016).

Remark 3: We can keep going with deeper layers by iterating on these ideas.