A General Survey of Techniques for Analyzing Large Datasets

Alan Luo
aluo18@choate.edu

(Dated: May 30, 2017)

As the world's total corpus of data grows at an astronomical rate, so have the techniques for numerically processing and analyzing this data. It is no longer practical nor computationally feasible to exclusively use human intuition to build and analyze statistical models. Many more techniques have been invented and used by researchers for processing data, and these techniques have been tuned for lower and lower error rates. The class of techniques for analyzing large sets of data has been generally referred to as 'machine learning.' Machine learning technology now powers nearly every aspect of technology in modern society. With the vast amount of data being collected through publicly available online services like Google and Facebook, machine learning is being used to identify faces in images [1], match advertisements and search results to consumer interests [2], translate speech into text [3], and so on. Recently, these algorithms have drawn in massive public interest through buzzwords such as 'big data' and through the seemingly inevitable end result of 'true' artificial intelligence. As recent as 2016, Google's AlphaGo [4] project beat the world's best Go player, achieving what was once thought to be impossible.

However, for many of these algorithms, researchers still do not understand the exact mechanisms of what occurs. In particular, deep learning describes a class of algorithms that learn their own structure. Researchers need only design the structure of certain networks for the algorithm to learn what to learn. Trying to understand what happens in such networks is a major research topic, with the eventual goal of being able to take a generic algorithm and throw large amount of data at it with little tuning. As it stands, network design relies on expert knowledge and intuition based on experience with tuning data models. Data visualization hence becomes a valuable tool for understanding arbitrarily large datasets. In this paper, we investigate popular methods for analyzing large data sets with examples from real-world research.

Introduction

Machine learning has been broadly defined in [5] to refer to computational methods that use experience to improve performance. Here, 'experience' refers to the available dataset. It often takes the form of human-labeled data, but can be other types of information obtained from interacting with the environment. Machine learning involves designing and testing efficient and accurate prediction algorithms.

There has recently been a massive influx in the quantity of data and the need to analyze it. In the next 3 years, the existing universe of data is expected to increase tenfold, for a total of 44 trillion gigabytes by 2020. Conceptualizing and analyzing this data in any way seems like a daunting if not an impossible task, yet the past few years have seen major successes. Computers have exceeded humans in Go, which was once thought to be unsolvable [6]. Self-driving cars have become regular sights on highways. Image recognition on datasets with over ten million images [7] has reached error rates of under 5 percent.

With such advances, it might seem that machine learning is a mystic and modern technological development. However, the basic principles of machine learning have existed for decades. For an elementary example to illustrate the basic principle in this paper, let us consider a linear regression. A classic modeling problem in any science experiment is the correlation between an independent and a dependent variable. FIG. 1 describes titration data describing changes in pH as the titration volume increases. We would like to obtain an equation that can allow us to infer pH from the titration volume. We can obtain an analytical solution, as shown in [8], of the form $y = \beta_0 + \beta_1 x$. β_0 and β_1 satisfy the following equations.

x_i: Volume (mL)	y_i: pH
0.0	1.74
1.9	2.44
2.4	2.49
2.9	2.62
4.4	3.22
4.9	3.34
5.4	3.50
6.8	3.91
7.3	4.18
7.8	4.40
9.2	8.60
9.7	8.64
10.2	9.32

FIG. 1. Data from an acid-base titration.

$$\beta_1 = \frac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{\sum_{i=1}^{n}(x_i - \bar{x})^2} \tag{1}$$

$$\beta_0 = \bar{y} - \beta_1 \bar{x} \tag{2}$$

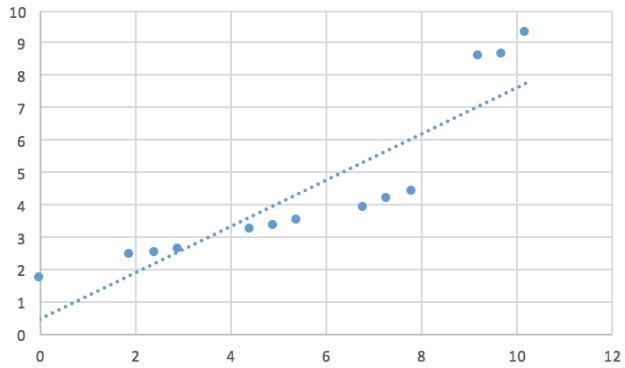


FIG. 2. Titration data graphed as pH vs volume with the corresponding regression, shown as a dotted line. It has an equation $\hat{y} = 0.7186x + 0.4628$.

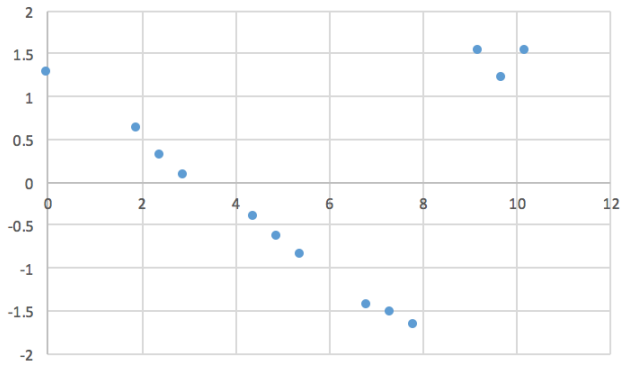


FIG. 3. Residual plot graphed as ΔpH vs volume. There is a clear pattern in the residuals, indicating a poor fit. A good residual plot has randomly scattered points.

While (1) looks complicated, linear algebra shows that it is really no more than a projection from higher dimensional space into lower dimensions. We can rewrite our two-dimensional regression using linear algebra, giving us a much shorter equation.

$$\begin{bmatrix}\beta_0\\ \beta_1\end{bmatrix} = (X^T X)^{-1} X^T Y \tag{3}$$

$$X_{i,j} = 1 \text{ for } i = 1, x_j \text{ for } i = 2 \tag{4}$$

$$Y_{i,j} = 1 \text{ for } i = 1, y_j \text{ for } i = 2 \tag{5}$$

The matrices X and Y on n data points are the $n \times 2$ where each row is a data point with a 1 appended. The full derivation of these equations is beyond the scope of this paper, but is investigated in [9]. For our discussion, this example is important in illustrating that complex problems about data can be rationalized and abstracted using mathematical models. In this case, the problem of minimizing residuals is rationalized in terms of minimizing Euclidean distance.

If we solve for these parameters, we get $\beta_1 = 0.7186, \beta_0 = 0.4628$. This gives the regression $\hat{y} = 0.7186x + 0.4628$. FIG. 2 shows a graph of our data and our regression. If we graph the values $y_i - \hat{y}_i$ versus x, we can see how the actual values are placed relative to our model. This is called a residual plot, and each data point is called a residual. This is shown in FIG. 3.

Residual plots allow us to qualitatively determine the goodness of fit of our model. If the residuals seem random, it is likely that our model is a good fit. If the residuals are regularly biased, it is likely the we did not select the right type of model. In our case, there is obvious regularity to the residuals, indicating that our model is a poor fit for the data. This elementary example of data inference gives us the framework with which we will analyze larger data sets.

With any data set, we use the data to construct a model, then qualitatively and quantitatively verify the model the model. Quantitative verifications describe goodness of fit while qualitative verifications use human intuition and expert knowledge to identify the correct type of model. In particularly large data sets, developing an intuition for the data is an equally important and difficult problem as quantitatively analyzing it. In some successful image-recognition models, even experts do not fully understand the underlying algorithms [10].

However, at the core, every machine learning algorithm is essentially the same - it is a numerical function from the input space to the output space. For the titration example, it is a function from the real numbers to the real numbers. For image classification, it is a function from images to categories. This means that we can use the same tools to understand and verify most statistical models. In the rest of this paper, we present common types of models, techniques used for analyzing them, and real-world examples of their use in research.

1. Traditional Machine Learning

Traditional machine learning refers to highly structured algorithms that tend to be small in size and simple in complexity. These algorithms are specifically engineered by experts, and their operation is fully understood. There are two primary types of machine learning algorithms. Supervised learning involves learning from labeled data, while unsupervised involves learning from unlabeled data. For instance, consider a dataset of pictures of oranges and apples. A supervised learning problem would be to train the model so that a new image can be labeled as an apple or an orange. An unsupervised learning problem would be for the model to divide the images into two groups based on their

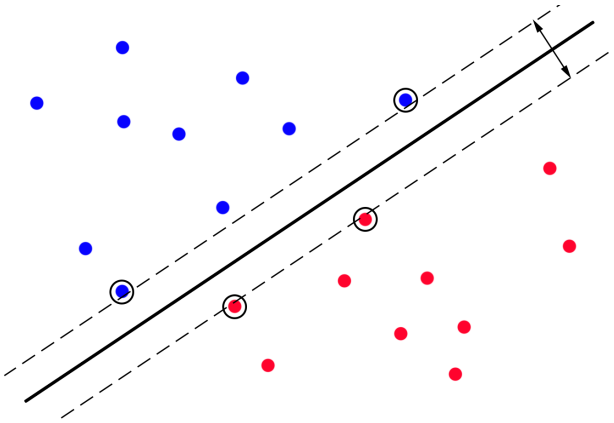


FIG. 4. A two-dimensional support vector machine. The line is the one-dimensional optimal hyperplane that divides the data. The distance between the two dotted lines is the margin. The points that lie on these lines are referred to as support vectors.

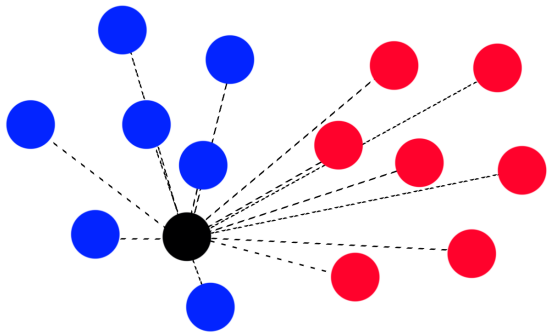


FIG. 5. Nearest-neighbors classification. The unknown ball is represented in black, with distances represented by dotted lines.

dominant colors. Different types of models are better suited to different problems.

There are a few common terms in machine learning. A machine learning algorithm extracts features by translating raw data into numerical inputs. The algorithm is trained on the training set and tested on the testing set. Algorithms are often seen as existing in n-dimensional input space.

1.1. Supervised Learning

In supervised learning, training data is labelled. The goal is to train the network with a set of classified data points so that it can classify a new data point into its corresponding category. We give two example algorithms. The first, SVM, is best described as a classification algorithm, while the second, Bayesian statistics, is best described as an inference algorithm.

1.1.1. Support Vector Machines

Support vector machines (SVMs) are binary classifiers, which is to say that they only classify into two categories. They work by dividing the n-dimensional input space into two halves with $(n-1)$-dimensional hyperplanes. FIG. 4 shows a two-dimensional SVM with a line classifier that splits the two-dimensional plane into two parts. From this example, it is easy to see that SVMs are best suited for problems with distinctly clustered data. In particular, they are well-suited to problems with a high-dimensional feature space, due to the similarity of the algorithm in the high-dimensional and low-dimensional cases. We will illustrate this in the following explanation, which is further detailed in [11].

To see how an SVM works, let us consider the naive case. Suppose we know nothing about data, statistics, SVMs, or anything. All we have is a cluster of red balls and a cluster of blue balls. Someone puts a napkin on one of the balls such that we are unable to identify what color it is. The naive way of doing so would be to calculate the average distance between the unknown ball and every red ball, and the average distance between the unknown and every blue ball. We would assign the unknown ball to the corresponding category with the lower average distance. This is also known as the nearest-neighbors algorithm, and is illustrated in FIG. 5.

However, in large data sets, this is extremely inefficient, as we would have to calculate the distance between each unknown and every point in the training set. Instead, we fix a set number of support vectors in the training set. For each new testing point, only the distances between the new point and the support vectors are used in classification. Using our intuition in the two-dimensional case, we can see that an SVM can be framed as an optimization problem seeking the optimal hyperplane such that the distance between the hyperplane and and the support vectors is minimized. This optimization is complicated in practice and is beyond the scope of this paper. Finding this margin is equivalent to finding a set of support vectors and a corresponding set of weights that give a linear margin. These weights are represented as α_i, and are used when calculating the category of a new testing point x.

$$D = \sum_{j=1}^{k} y_j * \alpha_j * \text{sim}(\vec{x}, \vec{s_j}) \tag{6}$$

The ith data point in an SVM is represented as $(\vec{x_i}, y_i)$, where $\vec{x_i}$ is the training data, and y_i is the class as 1 or -1. $(\vec{s_i}, y_i)$ represents the set of support vectors. sim() is a similarity function, which can be any norm, but is typically a dot product in the context of

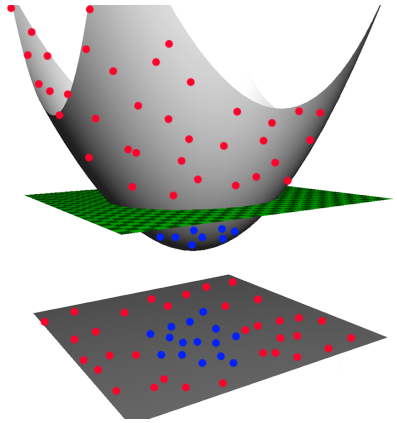


FIG. 6. SVM kernelling. The two-dimensional data is lifted into the third dimension with a quadratic kernel. Then, a plane divides the data in three-dimensional space.

SVMs. In (6), the new testing point $\vec{x}$ has a weighted average distance taken with every support vector, in the direction of the support vector's corresponding class. So, if $D > 0$, we assign x to the 1 category, and if $D < 0$, we assign it to the -1 category.

This algorithm is easily generalizable to larger dimensions with linear algebra, which give SVMs their higher-dimensional flexibility as discussed earlier. They have been used to great effect in document classification, the problem of identifying what type of document a body of text represents. In [12], document features are represented as 10000-dimensional vectors, where each component of the vector represents a frequency of a word. While this type of frequency analysis seems naive and inefficient, they are easily approachable with SVMs. SVMs have for similar reasons been effective in object detection, the problem of detecting whether or not there is an object in an image [13].

Dimension flexibility are an important component of SVMs which allow them to be generalized into a variety of problems. Instead of clearly divided red and blue clusters, imagine a situation where the blue cluster is surrounded by red points. Clearly, no one-dimensional hyperplane can divide the data. However, we can manipulate the data into a new dataset where red and blue are clearly defined, then use a higher-dimensional SVM with little sacrifice to efficiency. This is illustrated in FIG. 6. This technique is referred to as kernelling or 'the kernel trick,' because the mapping function is referred to as a kernel. We denote it as $K(\vec{x}, \vec{s}) = \langle \Phi(\vec{x}), \Phi(\vec{s}) \rangle$. In practice, this changes our SVM equation.

$$D = \sum_{j=1}^{k} y_j * \alpha_j * K(\vec{x}, \vec{s_j}) \tag{7}$$

A function can only be a kernel of an SVM if it is the dot product of some higher dimension transformation. For instance, the function $K(x, s) = \langle \vec{x}, \vec{s} \rangle^2$ is a kernel. In the two-dimensional case, this is because it is equivalent to $\langle (x_1^2, x_2^2, \sqrt{2}x_1x_2), (s_1^2, s_2^2, \sqrt{2}s_1s_2) \rangle$. There are three common types of kernels in machine learning.

$$K(\vec{x}, \vec{s}) = \langle \vec{x}, \vec{s} \rangle \tag{8}$$

$$K(\vec{x}, \vec{s}) = \langle \vec{x}, \vec{s} \rangle^p \tag{9}$$

$$K(\vec{x}, \vec{s}) = e^{-||\vec{x}-\vec{s}||^2/\sigma^2} \tag{10}$$

(8) is linear, (9) is polynomial, and (10) is referred to as a radial basis function, or Gaussian. [14] gives an example of a linear kernel, while [15] has both polynomial and RBF kernels. Note that a linear kernel is really just a polynomial kernel of order 1. Also, it can be shown that a Gaussian kernel is really just an infinite-degree polynomial kernel. Thus, all three types of common kernels can be understood to be polynomial.

While SVM are in principle binary, they can easily be extended to multi-class problems [16]. Basic n-category classification can be done through nesting $n - 1$ SVMs in sequence. More complex categorization can be done by running multiple SVMs disjunctively. For instance, two disjunctive SVMs could categorize balls into red and blue, small and large. Due to their flexibility, SVMs have seen and still see widespread use in different machine learning problems.

1.1.2. Bayesian Statistics

Bayesian statistics are perhaps the oldest example of 'machine learning.' The central principle of Bayesian Statistics have been around for centuries. To call Bayesian statistics a method on its own is a misnomer. Rather, it describes a branch of statistics in which evidence about a model is expressed in terms of degrees of belief. In general, a Bayesian approach to a dataset would involve first making assumptions about the data, collecting data, then comparing the results to the assumptions and adjusting accordingly.

For a Bayesian statistical approach, the first step is to define a model that expresses our qualitative knowledge of the data. The model will have some unknown parameters. We specify a prior probability distribution for these unknown parameters, which express our beliefs about the data. The keystone of Bayesian statistics is an equation the relates these measurements called Bayes' Theorem.

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)} \tag{11}$$

This is an elementary result from conditional probability, but it provides an important relationship in Bayesian

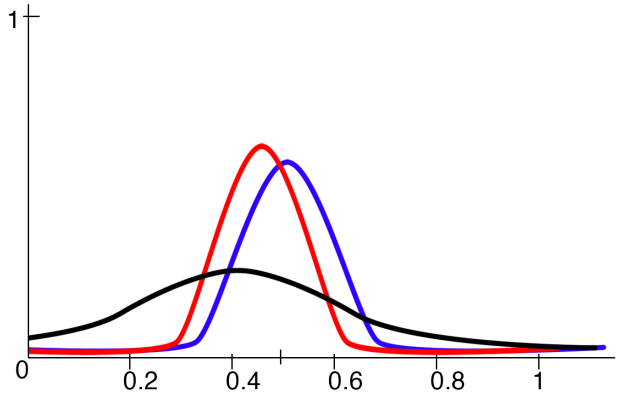


FIG. 7. A basic example of Bayesian statistics. Blue is the prior, black is the likelihood, red is the posterior. Prior = $\beta(30.0, 30.0)$, Likelihood = binomial$(10, 4)$, Posterior = $\beta(34.0, 36.0)$. Observe that the likelihood shifts the prior towards the left and decreases spread. This makes sense, because the observed data was below the prior, and the added information increases certainty of parameters.

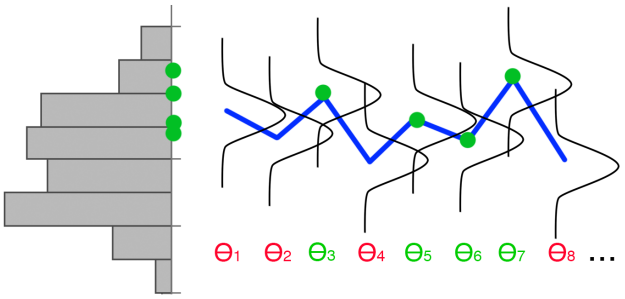


FIG. 8. The Metropolis-Hastings algorithm. Each θ_i represents one iteration, also called a Markov Chain-Monte Carlo step. At each iteration, a sample is taken using the previous sample as a mean. If the sample passes the Metropolis-Hastings condition, it is kept. These points are marked green, and the diagram shows their value being kept in a histogram, which approximates the posterior distribution.

statistics. $P(A)$ will be the supposed parameter probability, or the prior, and $P(B)$ will be the real-world probability based on the data. This means that the denominator of the right-hand side is a constant. So, we can rewrite (11).

$$\text{Posterior} \propto \text{Prior} \times \text{Likelihood} \tag{12}$$

In the rest of this section, we will not give a comprehensive breakdown of Bayesian statistics because it sees little use compared to other methods in machine learning and because there exist multifaceted approaches even within Bayesian statistics. The breadth of this task is thus beyond the scope of this paper. Instead, we will illustrate a simple example. See [17] for more details. Suppose we want to infer a probability distribution from coin flipping. Our assumption of the situation is that a coin can only land on heads or tails. We construct a prior probability distribution based on a normal distribution (or the very similar β-distribution) centered around $\mu = 0.5$, since our intuition tell us that the coin should have an approximately 50% chance of landing on either heads or tails. We will write our normal distribution in terms of the chance of heads.

Now, suppose we test this claim by flipping a coin 10 times, and heads appears 4 times. We write our likelihood distribution as a binomial distribution. We perform calculations using Bayes' theorem, giving us a new posterior from the prior which is closer to the likelihood. This entire situation is illustrated in FIG. 7. [18] gives an example of Bayesian methods for object categorization. The authors use an incremental Bayesian approach to allow the posterior to gradually approach convergence.

In practice, direct sampling from posterior distributions can be very difficult. It is necessary to use numerical approaches in order to evade unreasonable convergence times. The most common algorithm for doing this is called the Metropolis-Hastings algorithm. Due to the mathematical complexity of the algorithm, details are not discussed, but are available in [19]. We will instead give a brief qualitative explanation of its workings. We first generate a random number. Using this as a starting point, we treat this new random number as the mean of a normal distribution, then sample a new random number from this normal distribution. We keep repeating this process. At each iteration, the Metropolis-Hastings constraint tells us whether we should keep the sample or ignore it. The resulting kept samples will approximate the posterior distribution. Most off-the-shelf statistics programs have the Metropolis-Hastings algorithm built-in. An illustration of the algorithm is given in FIG. 8.

1.2. Unsupervised Learning

In unsupervised learning, training data is not labelled. We look at two types of problems. The first, clustering, is essentially the unlabeled version of classification. The goal is to train a model with a training set such that the model is able to infer the categories, though not necessarily identify them. An unknown point can then be placed into one of these categories. The second problem is dimension reduction. In dimension reduction, the model aims to reduce high-dimensional data into lower-dimensional data while still retaining all the statistical significance of the data. Many algorithms will use some combination of supervised and unsupervised methods [20]. This allows for a simpler input, reducing

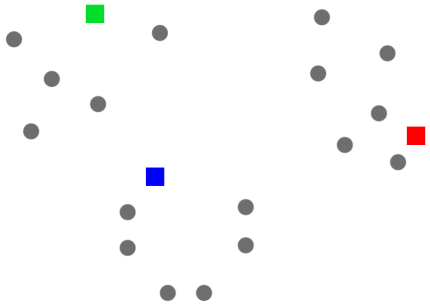


FIG. 9. The setup for a 3-means clustering algorithm. 3 random data points are selected to be means.

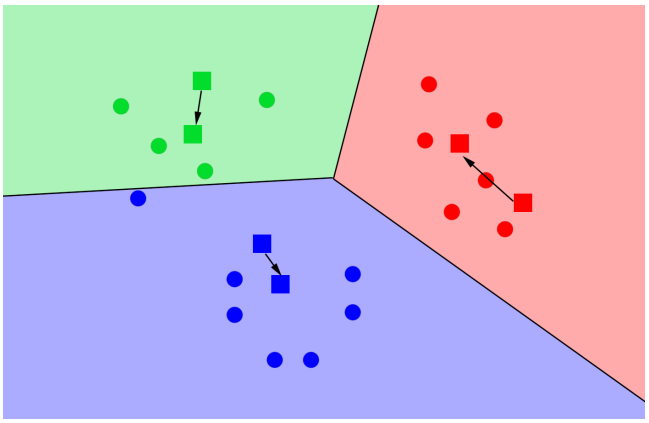


FIG. 11. The second step of a 3-means clustering. New means are moved to the centroid of each region.

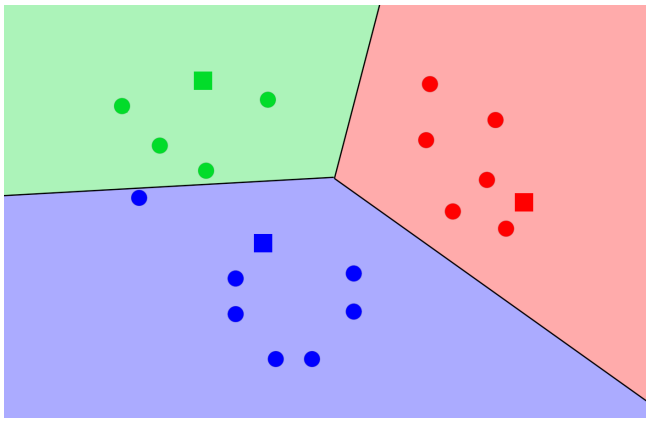


FIG. 10. The first step of a 3-means clustering. Points are classified by their closest mean. Classification regions are shown by a Voronoi diagram.

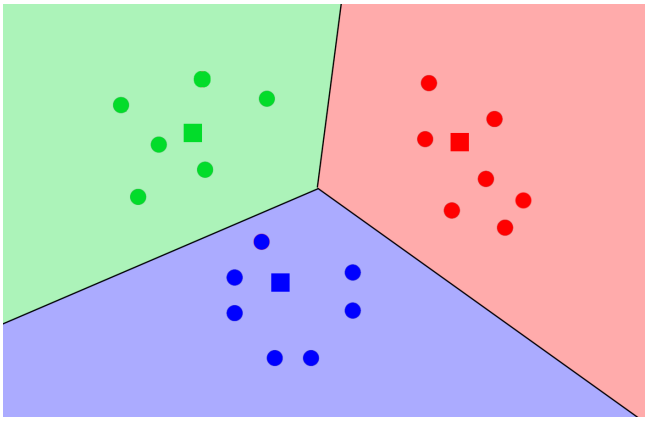


FIG. 12. The third step of 3-means clustering. New classification regions are calculated, and the process restarts. Each iteration moves the mean to the centroid, then classifies.

computation times in larger algorithms.

1.2.1. k-Means Clustering

In k-Means Clustering, we want to divide the data into k different regions. The data does not need to be labelled, and the resulting regions do not need to have physical representations. The goal is just to be able to fit a new data point into one of these regions, or clusters. The algorithm is generalizable into larger dimensions, but we will illustrate an example in two dimensions. The two dimensional case is intuitive for the sake of example.

Initially, the algorithm selects k random data points as starting means. This is shown in FIG. 9 with 3-means clustering. These means act as the centers of classification regions. Then, each data point is classified according to its closest mean. This corresponds to a linear classification boundary between the regions defined by each point. This situation is shown in FIG. 10. Then, new means are calculated using the classified data by calculating the centroid of each class. So, the red mean moves to the centroid of all the red data points, and so on. This is shown in FIG. 11. Finally, new regions are calculated from the new means, as shown in FIG. 12. The centroid is found by component-wise averaging each point.

$$\vec{m_i}^{t+1} = \frac{1}{|S_i^t|} \sum_{\vec{x_j} \in S_i^t} \vec{x_j} \tag{13}$$

In (13), m_i^t denotes the ith mean at iteration t. S_i^t denotes the set of points corresponding to centroid i at step t. The equation just states that each step iterates through each mean and calculates a component-wise average. This process of recalculating means and classification regions is replaced some number of times, with the intention of improving clusters each iteration. Notice that from FIG. 10 to FIG. 12, an incorrectly classified blue point is moved into the green region, indicating an improvement in classification. It should be clear that k-means clustering is best suited to problems with distinct clusters that don't need to be identified. It has been used to great effect for image segmentation,

FIG. 13. Sample images using k-means on images in [21]. (a) and (d) are the raw images, while (b) and (e) are images processed using k-means, with each cluster being represented by a different greyscale shade.

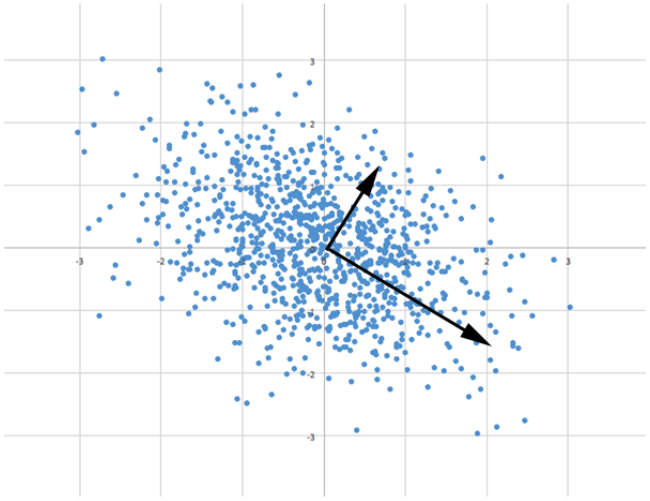


FIG. 14. PCA on approximately elliptical data. The two vectors are the two principal components.

the problem of partitioning images into sets of pixels. The goal of image segmentation is to simplify an image into components that are easier to analyze.

In [21], the image segmentation problem is approached by treating each pixel as a data point. The k-means problem is approached as trying to assign each pixel to a number. The different clusters then represent different image segments. An example is shown in FIG. 13. k-means is advantageous in its simplicity and flexibility. However, it has the downside of relying on the initial choice of centroids. Results can vary wildly depending on if the initial centroids are well-chosen.

1.2.2. Principal Component Analysis

Principal Component Analysis (PCA) is a dimensionality reduction algorithm. The intent is to take a data set expressed in n dimensions and re-express it in terms of fewer dimensions while still maintaining relevant information. PCA is an orthogonal linear transformation, meaning that data similarity is preserved. Mathematically, dot product is preserved. In other words, a PCA transformation T satisfies the following equation for all $\vec{u}, \vec{v}$ in the dataset.

$$\langle \vec{u}, \vec{v} \rangle = \langle T\vec{u}, T\vec{v} \rangle \tag{14}$$

Qualitatively, this means that relative information about data is preserved. For instance, suppose we are using pixels in an image as a dataset, where each pixel is represented by its RGB value $\vec{p} = (r_p, g_p, b_p)$. Suppose we have totally red and totally green pixels, or $\vec{p_1} = (1, 0, 0), \vec{p_2} = (0, 1, 0)$. Their dot product is 0, meaning they are totally different pixels. Our PCA transformation might remove the blue attribute, giving us $\vec{p_1'} = (1, 0), \vec{p_2'} = (0, 1)$. This would be a valid transformation under our orthogonal condition, since dot product is preserved, qualitatively describing that no information has been lost. Indeed, in this elementary example, the blue component contributed no additional information about the vectors, and is thus superfluous.

As an example, suppose a two-dimensional dataset resembles a multivariate Gaussian distribution, which is to say that it seems elliptical. One way to reduce this dimension would be to express each data point in terms of its distance along the major axis from the center of the distribution. This situation is shown in FIG. 14. In this figure, clearly, every point can be re-expressed in terms of these two eigenvectors, the major and minor axis. One potential dimensionality reduction would thus be to use only the major axis component of this transformation. For instance, the point at the tip of the long arrow would just be 1, while the point just at the tip of the short arrow would just be 0.

The first step in PCA is to calculate a covariance matrix. Intuitively, we know that covariance describes the degree to which a change in one variable accounts for the change in another. Thus, if we want to reduce dimensionality while keeping relative information intact, we should want to maximize the total covariance, or the variance, of our transformation. We will give equations for a PCA that transforms a three-dimensional dataset into a two-dimensional one, for the sake of example. Our covariance is thus represented.

$$\Sigma_i = \begin{bmatrix} \sigma_{11}^2 & \sigma_{12}^2 & \sigma_{13}^2 \\ \sigma_{21}^2 & \sigma_{22}^2 & \sigma_{23}^2 \\ \sigma_{31}^2 & \sigma_{32}^2 & \sigma_{33}^2 \end{bmatrix} \tag{15}$$

In (15), Σ_i is the covariance matrix where σ_{ij}^2 represents the covariance between the component numbers i

and j. As a reminder, we give the formula for covariance between two variables x and y over a population of size n.

$$\text{cov}(x, y) = \frac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{n} \tag{16}$$

The next step is to find the eigenvalues and eigenvectors of this matrix. As a reminder, eigenvectors describe the dominating direction of a linear transformation as it is applied to large cases. Qualitatively, the describe the most important or 'identifying' components of a matrix. ('Eigen' is German for 'self.') Eigenvectors and eigenvalues are described in the following equation, where $\vec{v}$ is an eigenvector of matrix A and λ is an eigenvalue of A.

$$A\vec{v} = \lambda\vec{v} \tag{17}$$

For a PCA that reduces our three-dimensional data into two dimensions, we then select the two largest eigenvalues and their corresponding eigenvectors. We denote these as λ_1, λ_2 and $\vec{v}_1, \vec{v}_2$. This gives use our PCA matrix W. The new data points are then transformed by W, which has dimension 3×2 in our example. They are also centered around the sample mean $\vec{\mu}$. This is summarized in the following equations.

$$W = [\vec{v}_1|\vec{v}_2] \tag{18}$$

$$y = W^T(\vec{x} - \vec{\mu}) \tag{19}$$

Due to the speed of modern numerical algorithms in computing eigenvectors and eigenvalues [22], PCA is a fast and widely used algorithm for dimensionality reduction. This also means that higher dimensional generalizations (beyond from three dimensions to two) are easy due to the linear algebraic basis of the algorithm. As previously mentioned, PCA is often used to preprocess data before using more complex algorithms. More details on PCA are available in [23].

One particularly interesting use of PCA is known as eigenfaces. [24] provides a method for facial recognition using PCA. It describes a real-time algorithm for facial recognition where faces directly treated as vectors. This means that, using PCA, an average face and principal facial components can be calculated. Principal facial components represent common structural variations from the average face. In this model, any face can be described as degrees of variation from the average face using the principal facial components. FIG. 15 and FIG. 16 give examples from one dataset in the paper.

FIG. 15. The 'average face' from a dataset in [24].

FIG. 16. Seven principal components, AKA 'eigenfaces,' from a dataset in [24].

2. Statistical Validation

Any statistical model is not useful unless it is valid. However, the success of a model in describing a dataset does not necessarily correlate with its success in describing additional points of data. Thus, it is necessary to have a set of techniques for validating data sets. We describe two types of validation in general: regression validation and cross-validation.

2.1. Overfitting

First, we specify the problem we are attempting to address with statistical validation. Consider a situation where we want to find the optimal time to return home in order to minimize traffic. We drive to work

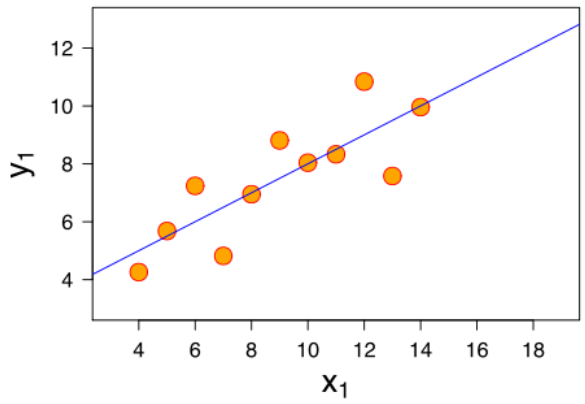


FIG. 17. In this graph, the data is roughly randomly scattered around the regression line, giving a seemingly good fit. This is the only graph that actually has a reasonable regression.

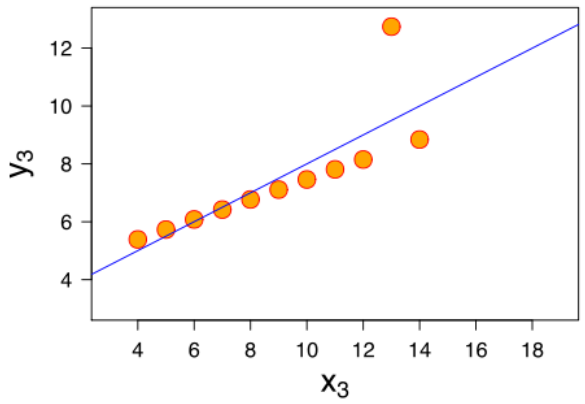


FIG. 19. In this graph, the data is very precisely linear, but has a large outlier. This example shows the necessity of checking for outliers, as the regression is now poor.

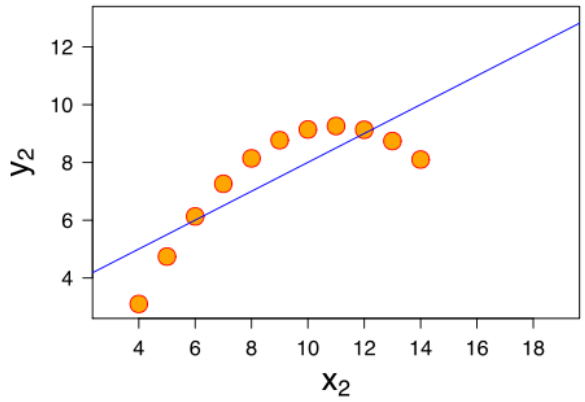


FIG. 18. In this graph, the data is very precisely paraboloid, but arranged around the regression line. This is obviously a bad regression, even though it has a high r-value.

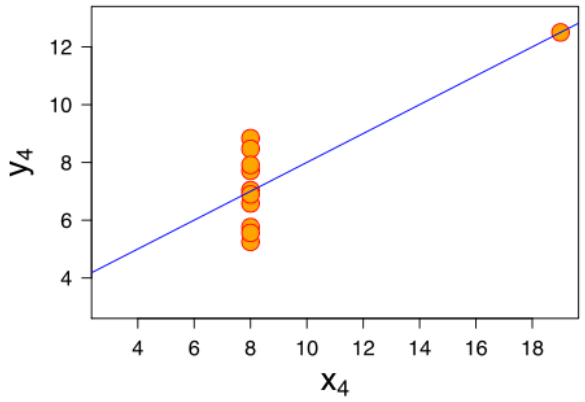


FIG. 20. In this graph, the data is precisely vertical, but has a large outlier.This example shows the necessity of checking for outliers, as the regression is now poor.

on Tuesdays and Thursdays, and collect data over the course of a few weeks. Our data tell us that it is optimal to return at around 6:00 pm.

However, while our data might work to describe our small sample, what if there were a baseball game every Wednesday at 6:00 pm? Then of course, on Wednesday, our data would be inaccurate in solving our initial problem. What about weekends? The problem we have introduced is that of overfitting, where a model is very well fit to a dataset such that it does not easily generalize to additional data.

In terms of numerical data, it is easy to fall into the trap of corresponding the goodness of fit of a model with the quality of that model. For instance, in a two-dimensional regression on n data points, an n-dimensional polynomial regression can be constructed to pass through every single point, giving absolutely no error on the training set. However, it is clear that it will probably not generalize to larger data. This is shown in FIG. 20.

The problem with direct numerical analysis has been summarized by statistician Francis Anscombe in a series of four graphs referred to as Anscombe's quartet [25]. The four graphs have nearly identical descriptive statistics, but very different shapes. FIGs 17-20 show Anscombe's quartet. Each of the four graphs have, with up to 3 decimal points of variation, an x-mean $\mu_x = 9$, x-variance $\sigma_x^2 = 11$, y-mean $\mu_y = 7.50$, y-variance $\sigma_y^2 = 4.125$, correlation $r = 0.816$, and regression line

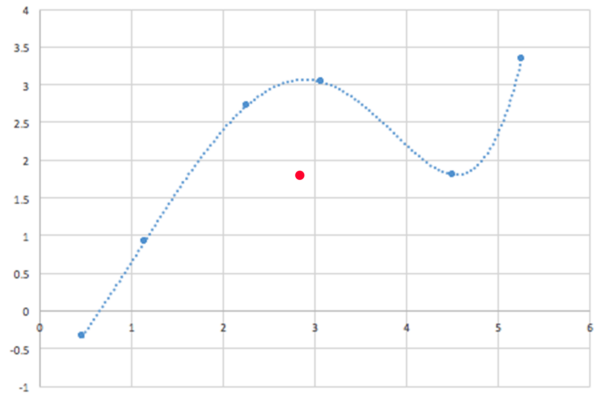


FIG. 21. Overfitting a regression. A sixth-order polynomial is fit to 6 data points, giving a 'perfect' zero error, but the model is poor at describing an additional data point.

$\hat{y} = 3.00 + 0.500x$. This will be developed on in the next section, where these statistics will be explained.

2.2. Regression Validation

A regression is a model of a relationship between continuous variables. It is the most elementary form of statistical model, and is very common in physical science experiments. For this section, we will discuss only two-dimensional regressions, but the results and techniques are easily extendible into higher dimensions. Also, we will not discuss techniques for finding regressions, only techniques for validating them. Most of the information in this section comes from [26].

2.2.1. Two-Dimensional Regression Validation

It would be useful to have a numerical description of the quality of a regression. Our intuition tells us that a good regression should have a lower error. However, our discussion of overfitting showed that this is a naive method. Nevertheless, goodness of fit is a useful sanity check. Quality of fit is usually defined in terms of least-squared error, or Euclidean distance.

$$E = \sqrt{\sum_{i=1}^{n} (\hat{y} - y)^2} \tag{20}$$

Notice the similarity between (20) and the Pythagorean Theorem. A least-squares regression is actually defined as the regression which minimizes this value. However, his definition of error as given by (20) does not generalize well to different data sets. This is because as the data set gets absolutely larger, error as given by (20) will necessarily increase. So, we want to measure relative error.

$$\frac{SSE}{SST} = \frac{\sum_{i=1}^{n} (\hat{y}_i - y_i)^2}{\sum_{i=1}^{n} (y_i - \bar{y})^2} \tag{21}$$

In (21), the error is just scaled by the mean. SSE stands for 'sum squared error,' while SST stands for 'sum squared total.' The SSE/SST value is used in the calculation of R^2, or the coefficient of determination. R^2 describes what portion of a data's variance can be explained by the correlation. The following formulas describe R^2.

$$R^2 = 1 - \frac{SSE}{SST} = 1 - \frac{\sum_{i=1}^{n} (\hat{y}_i - y_i)^2}{\sum_{i=1}^{n} (y_i - \bar{y})^2} \tag{22}$$

$$R^2 = \frac{SSR}{SST} = \frac{\sum_{i=1}^{n} (\hat{y}_i - \bar{y})^2}{\sum_{i=1}^{n} (y_i - \bar{y})^2} \tag{23}$$

In (23), SSR stands for 'sum squared regression' (22) and (23) are different but equivalent statements. (22) gives R^2 in terms of its error in describing the data, while (23) gives R^2 in terms of the portion of variation in data that the regression can describe. In both cases, a 'perfect' regression has $R^2 = 1$, where all the data variation is described by regression, while the worst possible regression has $R^2 = 0$, where none of the data variation is described by regression. $R^2 = 0.68$ means that about 68%of the data variation is described by the data.

To illustrate what is meant by data variation, suppose we have two very similar graphs. Both have strong linear correlations, but one is nearly horizontal, while the other has a steep slope. FIG. 22 and FIG. 23 illustrate this situation. Geometrically, the data in FIG. 22 and FIG. 23 are identical. However, there is nearly no variation in FIG. 22. More precisely, the variation in y that is there does not depend on the value of x. This means that the regression, which depends on x, describes very little about the data. On the other hand, in FIG. 23, the data very much depends on the value of x. This is reflected in the greater R^2 value in FIG. 23 vs FIG. 22. We can also see this qualitatively, as FIG. 22 looks much more random than FIG. 23 when viewed normalized, even though the variation is the same absolutely.

There is a very similar value to R^2 called r, or the coefficient of correlation. It is calculated as follows for two variables, where s_x, s_y represent sample standard deviation.

$$r_{xy} = \frac{\sum_{i=1}^{n} (x_i - \bar{x})(y_i \bar{x})}{(n-1) s_x s_y} \tag{24}$$

The logic of this formula is very similar to the one for R^2, with the exception that it is calculated without a regression. However, ultimately, it describes the same thing: how much a change in y is explained by a change in x. It is worth noting that for a linear regression on a dataset, $R^2 = r$, but this is not always true. Notice that in this formula, a lower standard deviation corresponds with a higher correlation, which makes sense, since the lower data spread makes relative changes larger.

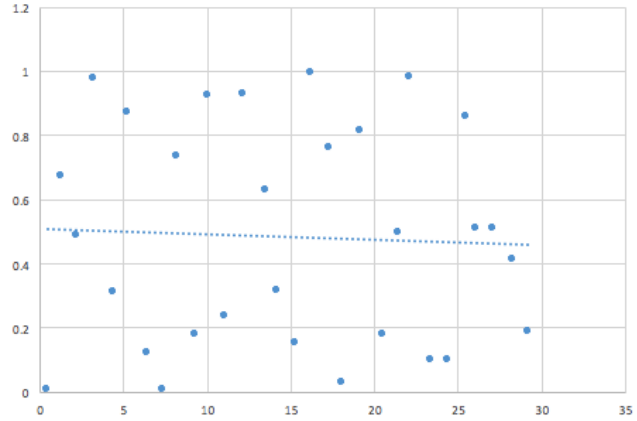


FIG. 22. A flat dataset. FIG 23. is a directly rotated version of this data using a rotation matrix. $\hat{y} = -0.0016x + 0.5068, R^2 = 0.00166$

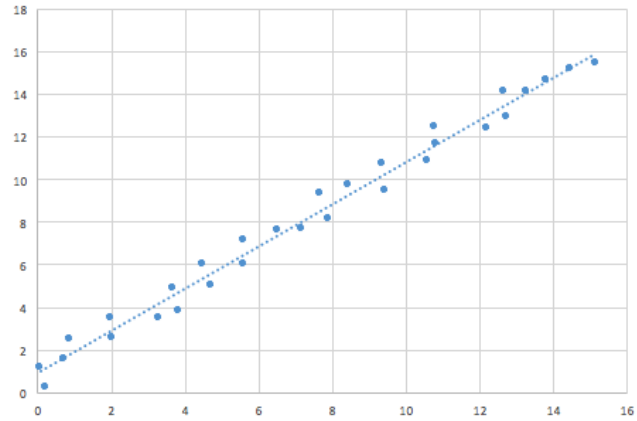


FIG. 23. Rotated version of the dataset in FIG. 22. The data is obtained by directly from applying a rotation matrix to the data in FIG. 22. $\hat{y} = 0.9872x + 0.9167, R^2 = 0.98455$

Now, we come back to Anscombe's quartet. The four graphs FIGs 17-20 all share the same standard deviations, means, and coefficients of correlation and determination. However, only one seems to be a good fit. This highlights the importance of qualitative analysis of data. For these graphs, it is obvious that three are poor fits. However, in larger datasets, this is not always the case.

We can determine whether or not a regression is a good fit using a residual plot. A residual plot for a regression $\hat{y}$ graphs $\hat{y} - y$ vs x. A negative residual indicates lower actual data than expected, while a positive residual indicates higher actual data than expected. A 'good' residual plot should have uniformly randomly distributed data points, because this means that variations from the model are due only to statistical variation rather than poor modeling. An example is given in FIG. 24 and FIG. 25.

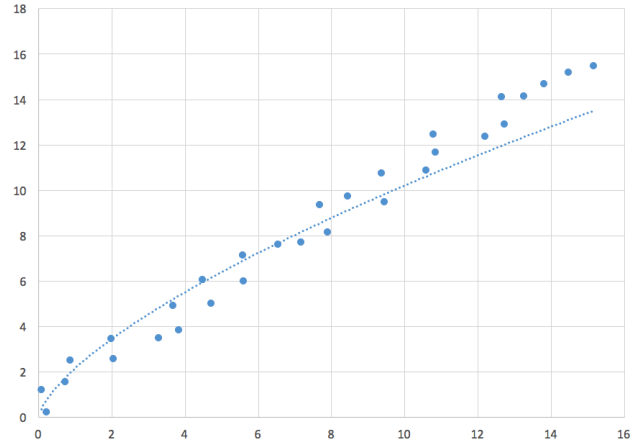


FIG. 24. Data generated from a linear model with a power regression $\hat{y} = 2.1763x^{0.6716}$. At first glance, the regression seems to be a good fit.

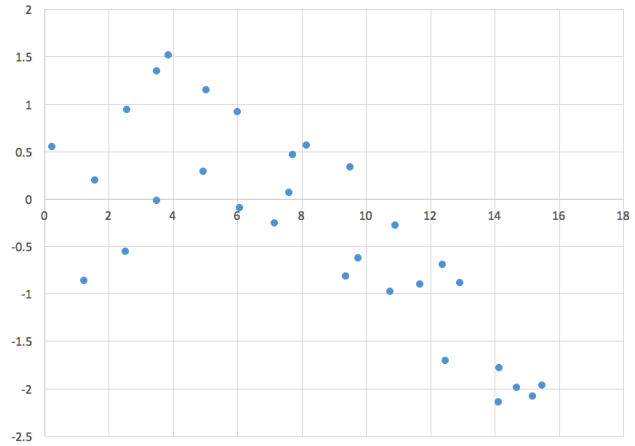


FIG. 25. Residual plot for FIG. 24. Though a power regression seemed like a good fit a first glance, the non-uniformity of the residual plot indicates that this is probably not the case.

A predictable or nonuniform residual plot indicates that the model is not accounting for something. It could be a missing or unmodeled variable, a higher-order term of a modeled variable, a missing interaction between modeled terms, or one of many other things. Sometimes, even if the residuals are uniform, the data may not be a good fit if the residuals are correlated with another variable in the data, or if the residuals are correlated with each other adjacently. These can be analyzed and tested for by plotting residuals versus other variables or versus time-dependent variables.

2.3. Cross-Validation

Where regression validation benchmarks a particular model against a particular dataset, cross-validation

measures how well a general type of model will apply what is learned from a dataset to data outside of the dataset. The goal of cross-validation is to define training and test sets for a general model, then create specific models by training them with the training set, and testing these specific models on the testing set.

There are two types of cross-validation: exhaustive and non-exhaustive. Exhaustive cross-validation tests every single possible training/test combination on a dataset. For instance, we might do exhaustive leave-200-out cross-validation on a 1000-element dataset. This means we would test every single possible combination of an 800-element training set and a 200-element test set on 1000 elements, giving a total of $C_{200}^{1000} \sim 10^{215}$ different combinations. This is obviously computationally infeasible. So, we will only investigate non-exhaustive cross-validation.

2.3.1. k-fold Cross-Validation

In k-fold cross-validation, a data sample is split into k even subsamples $d_0, d_1, \ldots d_k$. Each d_i rotates as the validation data for k runs in total. For instance, 3-fold cross validation would give d_0, d_1, and d_2. This would result in three validation runs: one where d_0 is the testing set and $d_1 \cup d_2$ is the training set, one where d_1 is the testing set and $d_0 \cup d_2$ is the training set, and one where d_2 is the testing set and $d_0 \cup d_1$ is the training set. FIG. 26 shows 3-fold cross validation. For more details, see [27].

2-fold, 5-fold and 10-fold cross validation are very common paradigms in machine learning. Cross-validation runs can have their results aggregated, as in [28], where a classification on a protein network uses 10 runs of 2-fold cross-validation. Image classification problems use a special type of validation where they will aggregate top-n error rates. For instance, in [29], an image classifier returns the list of all categories ranked from most likely to least likely. The top-n error rate gives the number of classifications that were not in the top n returned categories. Top-5 and top-10 error rates are common.

2.3.2. Holdout Method

In the holdout method, data points are assigned randomly to training and test sets d_0 and d_1. The holdout method involves only a single run, and is the simplest type of cross-validation. Isolated, this method is seen as a simple or degenerate form of cross-validation, but it can be aggregated in multiple runs. The primary difference between holdout and 2-fold cross-validation is that in holdout, the test set is usually much smaller than the training set, while in 2-fold, both sets are the same size.

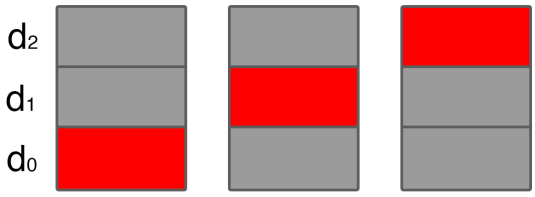


FIG. 26. 3-fold cross-validation. Red represents the testing set while gray represents the training set.

3. Visualization

As we saw in our discussion on regression analysis, qualitative information about data can be just as important as quantitative information. Thus, it is important to be able to analyze data data sets to develop an intuition. This makes data visualization an important issue, especially when data is particularly complex or high-dimensional and hard to visualize. We discuss two important realms of data visualization: visual analytics and database querying.

3.1. Visual Analytics

Visual analytics, rather generally, is the field of using data visualization tools to develop intuitions for data and make qualitative observations on models. The basic analysis we did of residual plots in earlier sections is an elementary form of visual analytics. Our discussion of visual analytics will center around distance metrics.

3.1.1. Distance Metrics

A distance metric on a set, mathematically, is some function that satisfies a set of conditions which allow that function to operate reasonably under human intuition [30]. For a metric d, these conditions hold for any value of x, y, or z:

1. $d(x, y) \geq 0$
2. $d(x, y) = 0 \Leftrightarrow x = y$
3. $d(x, y) = d(y, x)$
4. $d(x, z) \leq d(x, y) + d(y, z)$

In the real world, we rationalize things in terms of Euclidean distance. Condition 1 just says that distances are always the same sign. Condition 2 says that distance is the unique identifier of similarity. Condition 3 says that

distance applies to pairs of objects irrespective of order. Condition 4 is the triangle inequality. In our rationale, the corresponding function is the Pythagorean norm, also known as the L^2 norm. The following formula gives the L^2 norm for an n-dimensional vector [31]. Note that in all the formulas we present for norms, we express the norm only in terms of one variable, which is taken to be $\vec{u} = \vec{x} - \vec{y}$.

$$L^2(\vec{u}) = ||u||_2 = \sqrt{\sum_{i=1}^{n}(u_i)^2} \tag{25}$$

There are many other distance metrics used in machine learning. There is also the L^1 norm [32], and the L^p norm [33], which is a generalization of the L^1 and L^2 norms. Below is a list of common norms and their formulas.

$$L^1(\vec{u}) = ||u||_1 = \sum_{i=1}^{n} u_i \tag{26}$$

$$L^p(\vec{u}) = ||u||_p = (\sum_{i=1}^{n}(u_i)^p)^{1/p} \tag{27}$$

$$L^\infty(\vec{u}) = ||u||_\infty = \max(|u_1|, |u_2|, \ldots |u_n|) \tag{28}$$

$$D_M(\vec{u}) = \sqrt{\vec{u}^T S^{-1} \vec{u}} \tag{29}$$

Each norm suits a specific use or a certain type of dataset. Often, finding a norm is just as difficult of a problem as using it. For instance, with L^2 norms, there is only way to travel from points $\vec{x}$ to $\vec{y}$ such that the distance travelled is $||\vec{x} - \vec{y}||_2$. However, with L^1 norms, there are many ways. This is illustrated in FIG. 27 The L_1 norm is also much more robust, meaning models built with it are not as susceptible to change from outliers. This is because the terms are not squared as they are in the L_2 norm. These properties make L_1 norms well-suited to sparse problems, where the greater distance between points does not affect the model as much [34]. In [35], a new norm is developed for the specific problem of protein identification from interaction networks.

(29) is also known as the Mahalanobis distance [36]. S is the covariance matrix on each component of the data. The Mahalanobis distance is a generalization of z-score, which in one dimension represents the number of standard deviations away a point is from the mean. The Mahalanobis distance is thus sometimes used as a generalized distance which takes into account data variation.

3.1.2. Inferring a Distance Function

It is important to have a norm that suits the dataset under analysis, otherwise learning algorithms maybe inefficient or inaccurate. One problem in visual analytics is that of learning a distance function from human input. Experts often have a much stronger grasp and intuition on the data than can be expressed with machines. Distance function-inferring algorithms aim to allow experts to use their intuition my manipulating data in a familiar setting, then solving for a norm that would fit the expert's interpretation of the data. [37] describes an algorithm for such an inference problem.

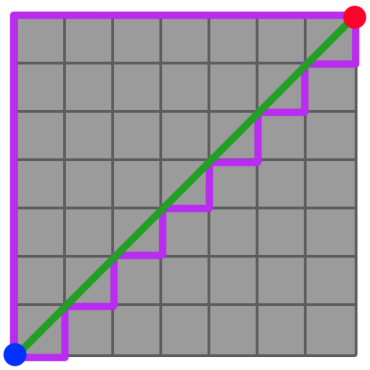


FIG. 27. L^1 vs L^2 norms. In green, the L^2 norm only has one distance-minimizing path. In purple, the L^1 norm has many different paths. Two are shown. This is also known as 'taxicab distance,' since each step the taken in the L^1 norm case is like a block when taking a taxi.

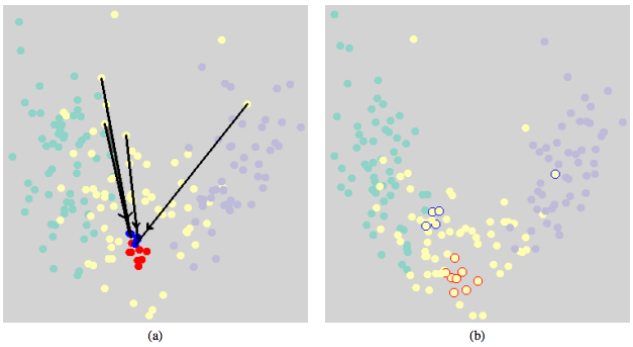


FIG. 28. One step of adjustment as shown in [37]. The blue and red data points in (a) indicate Y_1 and Y_2 being adjusted. (b) shows the new projection with the newly optimized distance function, with the blue highlighted and red highlighted points having moved closer to their desired location.

On a qualitative level, the data is first displayed into a familiar 2D projection using the existing distance metric. Then, the user can drag and pull the data to new points. The system then runs an optimization to solve for a new distance metric from these changes, and re-projects the data, and the process repeats. FIG. 29 shows a few iterations of this process.

The distance function is modeled as a weighted Eu-

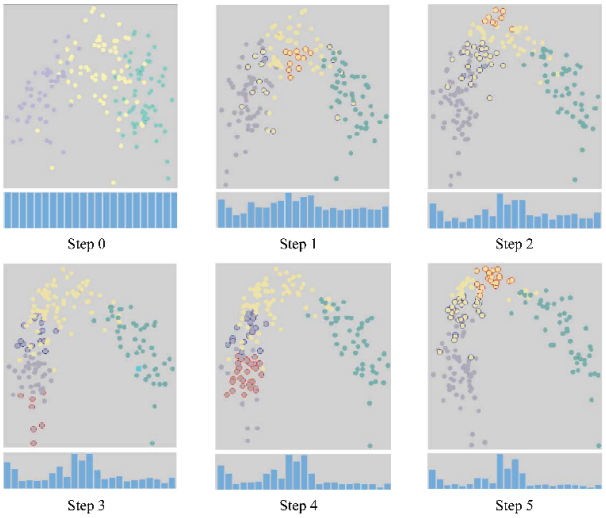


FIG. 29. A few iterations of the process as shown in [37]. The two highlighted sets of data indicate two datasets Y_1 and Y_2 that the user interacted with and moved. The histogram indicates the norm weights. Each bar represents a different dimension, collectively representing θ.

clidean distance, where each dimension has a different contribution to the total distance. $\vec{\theta}$ represents the dimension weights for a given iteration, and is of the same dimension as the data points. When $\vec{\theta}$ is uniform, the norm is essentially equivalent to Euclidean distance. The distance between two points $\vec{x_i}, \vec{x_j}$ is given as follows.

$$D(\vec{x_i}, \vec{x_j}|\theta) = \sum_{k=1}^{M} \theta_k (x_{ik} - x_{jk})^2 \tag{30}$$

The distance function is displayed into a two-dimensional scatterplot using multidimensional scaling. The details of MDS are beyond the scope of this paper, but are available in [38]. Qualitatively, MDS is extremely similar to PCA, as the goal remains to reduce dimensions from an arbitrary number down to two. We can thus understand MDS as a projection of the data's first two principal components.

The optimization is formulated as follows. The user-changed data is in two subsets, Y_1 and Y_2. The objective is to update the distance function such that the data not changed by the user is as unchanged as possible while the data changed by the user is as closed to desired as possible. We first construct a user input matrix U which represents desired distances between points. We then use this matrix to construct our loss function.

$$U_{ij} = \begin{cases} \frac{\text{intended distance}}{\text{original projected distance}} & \text{if}(\vec{x_i}, x_j) \in Y_1 \times Y_2 \\ 1 & \text{otherwise} \end{cases} \tag{31}$$

$$\vec{\theta^t} = \text{argmin}_{\vec{\theta^t}} \sum_{i<j\leq N} L_{ij}^t (D(x_i, x_j|\vec{\theta^t}) - U_{ij}^t D(x_i, x_j|\vec{\theta^{t-1}}))^2 \tag{32}$$

t denotes a given iteration, while $\vec{\theta^t}$ is the weight vector at that iteration. The loss function is defined simply. The first term minimizes the updated distance in the changed terms, while the second term encourages keeping every non-changed point at a similar distance. The L_{ij} term is a coefficient defined as follows. The value of this coefficient is the ratio of the number of unchanged pairs to the number of changed pairs.

$$L_{ij}^t = \begin{cases} \frac{N(N-1)}{|Y_1^t||Y_2^t|} - 1 & \text{if}(\vec{x_i}, x_j) \in Y_1^t \times Y_2^t \\ 1 & \text{otherwise} \end{cases} \tag{33}$$

Through this simple optimization, distance functions can be inferred, which allow for data to be better interpreted through visual analytics.

3.2. Database Querying

Database querying is perhaps the most universal of visualization problems. A database is a data structure which is comprised of tables. Each table has columns representing attributes and rows representing data points. Databases run nearly every website, and are an extremely common form of data collection. Part of why they are so widespread is that they are very flexible when using the JOIN operation. A JOIN operation allows data from two tables to be aggregated into a larger (or smaller) table. We will discuss join examples on the following simple tables with imaginary people.

Table A.

Name	Birthday
Bob	May 10
Steve	Jan 4
Tim	Sep 12
Sam	Nov 11

Table B.

Name	Age
Bob	70
Steve	36
Max	24
Alan	15

If we join these two tables as A JOIN B on the column 'Name,' we will get a right part, a left part, and a center part, as represented by portions of the venn diagram in FIG. 30. The center part is the joined data when there is a match on 'Name.' The left part is the unmatched part of A with null entries for the columns of B, and similarly for the right part. We give the corresponding tables to the left, right, and center parts.

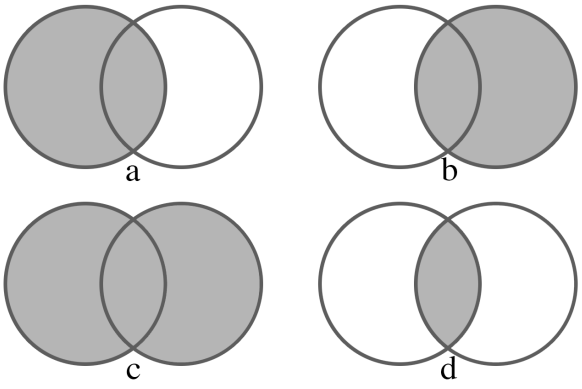


FIG. 30. The four most common types of database JOINs. (a) is a LEFT OUTER JOIN, and (b) is a RIGHT OUTER JOIN. (c) is a FULL JOIN. (d) is an INNER JOIN. The inner region, the intersection, represents matched rows across the two tables. The left and right regions represent unmatched rows which are kept and have null rows added.

Center

Name	Birthday	Age
Bob	May 10	70
Steve	Jan 4	36

Left

Name	Birthday	Age
Tim	Sep 12	null
Sam	Nov 11	null

Right

Name	Age	Birthday
Max	24	null
Alan	15	null

Every join just comprises of different combinations of unions of these tables. We give the four joins as shown in FIG. 30.

LEFT OUTER JOIN

Name	Birthday	Age
Bob	May 10	70
Steve	Jan 4	36
Tim	Sep 12	null
Sam	Nov 11	null

RIGHT OUTER JOIN

Name	Birthday	Age
Bob	May 10	70
Steve	Jan 4	36
Max	null	24
Alan	null	15

FULL JOIN

Name	Birthday	Age
Bob	May 10	70
Steve	Jan 4	36
Max	null	24
Alan	null	15
Tim	Sep 12	null
Sam	Nov 11	null

INNER JOIN

Name	Birthday	Age
Bob	May 10	70
Steve	Jan 4	36

Joins across multiple tables in larger datasets can be very slow for many reasons. First of all, the join operation by itself can be inefficient when done across thousands of tables. Also, once the data is joined, transporting the data from a server to a client creates a bottleneck. A FULL JOIN cartesian product across larger tables grows exponentially, and processing speed will necessarily outpace transfer rate. Thus, methods for quickly estimating data from large databases are necessary. We discuss these methods.

3.2.1. Online Aggregation

Suppose we have the previous situation, where we have a database that is for all intents and purposes unmanageably large to calculate. Suppose we want an aggregate query from this database - that is, rather than querying the database for the ages of all people in a country, we want the average age of all those people. This situation is very common. In traditional database systems, aggregation is done in batch mode. A query is submitted, the system processes a large volume of data over a long period of time, and the result is returned. This can be very slow for the reasons previously described, particularly when dealing with information that has not been preprocessed.

Traditional means for dealing with this problem have involved precomputing data cubes. Data cubes are just three-dimensional data tables, as shown in FIG. 31. This just means doing JOIN operations ahead of time for queries that are thought to be common. This method has the advantage of being fast for queries that have been precomputed, but the system reaches a bottleneck of a query is submitted which has not been precomputed.

In [39], a new online aggregation interface is introduced. The proposed method is to get an initial estimate from the database with a confidence interval, then allow the estimate to gradually converge and the confidence interval to gradually increase as the system processes

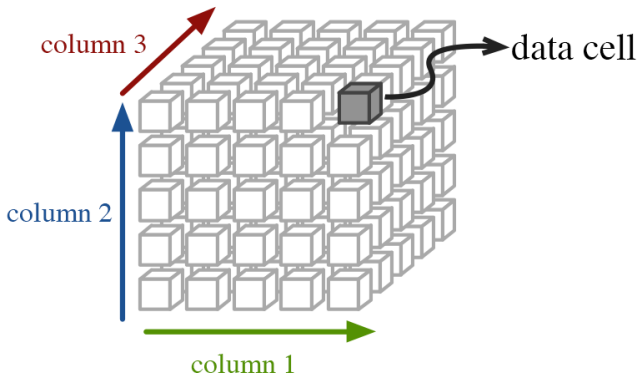


FIG. 31. A data cube. These are very common in joins, since most tables contain two attributes, such as a dependent and independent variable from experimentation. Joining across two tables with two attributes gives a three-column table, since one column will the the join column and will be shared.

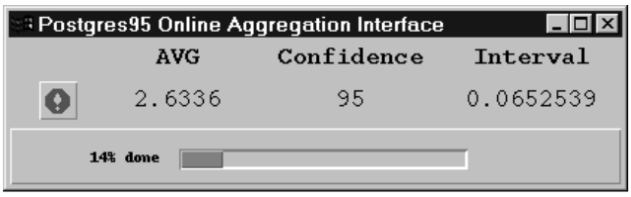


FIG. 32. An online aggregation interface from [39]. An aggregate query (mean), confidence interval, and margin of error are shown, as well as query progress.

more and more data. For visualization, estimates within a margin of error or even an order of magnitude are often enough. For instance, say we want to compare the distance and luminosity of distant stars. For initial intuitions, we are less concerned with exact equations than we are with relative orders of magnitude. The online aggregation interface can thus potentially save time if the estimate is close enough and the confidence interval is large enough. FIG. 32 shows an online aggregation interface.

For online aggregation, it is necessary to have a way to take statistically reliable samples from data. In other words, we need a query that can reduce bias. Such scanning methods consider the problem of random access to data. Typically, in order to save computation time, data is just stored in such a way that basic access methods satisfy this condition. For instance, if it is known that the aggregating attribute is not time-dependent, then a simple index scan should suffice, where each additional data row is sufficiently random. If the data is stored in heaps, which are a type of computer structure illustrated in FIG. 33, a basic heap scan will provide a random statistic. Most database systems use heap scans [40].

The actual online aggregation algorithm is relatively naive and uses basic sampling distributions from introductory statistics. The field is relatively new, with database queries having been stagnant since the 1960s until the advent of this paper [41]. We will thus not investigate the statistical details of the sample, but will instead proceed to discuss breakthrough algorithms in the field.

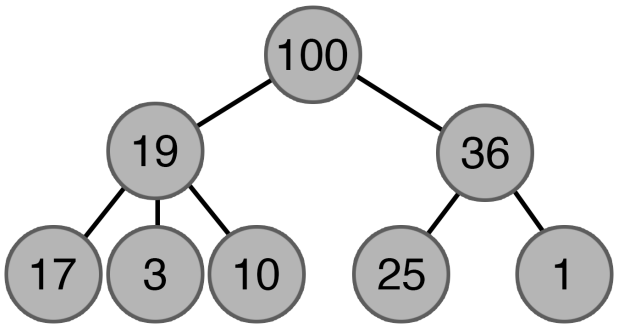


FIG. 33. A data heap. A heap is a tree that satisfies the condition that each child node has a lower index than its parent node. Databases stored in this way are difficult to sort, since comparable values are not stored on the same level, but are as a consequence easier to query randomly.

3.2.2. Ripple Join

The online aggregation method introduced in the previous section is naive and offers no algorithmic improvements on the basic concept of aggregation. [42] introduces a faster and more flexible method for aggregation, known as ripple join.

Ripple join instead aggregates data in an increasing manner such that the search regions in each table increase at similar rates. To explain this, consider a naive INNER JOIN of two tables. In order to calculate the INNER JOIN, we must know where the two tables match up. This means that we need to test every tuple of elements from these two tables. In other words, if we are joining tables A and B, the total number of elements that need checking is $|A| * |B|$. This holds true for any type of join.

Each of these $|A| * |B|$ data pairs can be seen as an element in a two-dimensional array, where each row represents an element from A and each column is an element from B. Ripple join gives an alternative order for calculating join cells. Instead of calculating them by the default order provided by the database structure, ripple starts from a corner of our imaginary data table, then adds a row to the calculated region, then a column, then a row, and so on. This is illustrated in FIG. 34. While these cells are being calculated, the statistic in question is being aggregated.

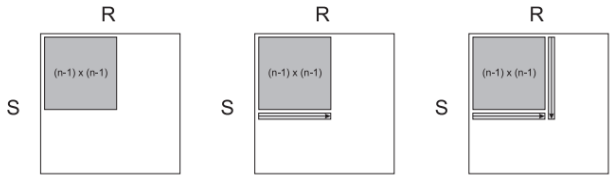


FIG. 34. An example diagram of ripple join from [42]. This shows the first three steps of the most basic ripple join algorithm.

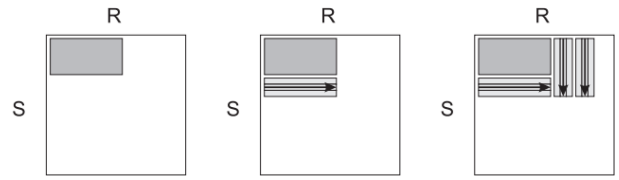


FIG. 35. An example diagram of non-square ripple join from [42]. This shows the first three steps of a ripple join algorithm with aspect ratio 2.

It is worth noting that ripple joins do not need to be square, i.e. calculate one column for each row. FIG. 35 gives a ripple join where two columns are added for each row. This ratio between new columns and rows is referred to as the aspect ratio. Optimizing a ripple join for a dataset involves selecting the correct aspect ratio.

Since the data is not selected uniformly, an estimator is needed for aggregated statistics. The general estimator for ripple join is given as follows.

$$\text{expr}(r,s) \simeq \frac{|R| * |S|}{|R_n| * |S_n|} \sum_{(r,s)\in R_n \times S_n} \text{expr}(r,s) \qquad (34)$$

In (34), R_n and S_n are the sets of tuples that have been read from R and S by the end of the nth sampling step. expr is any aggregate expression on R and S - it can be SUM, COUNT, or AVG.

3.2.3. Wander Join

Wander join is another alternative method for aggregation querying introduced in [43]. In wander join, aggregation is performed by joining on a random walk on the graph that represents the connectivity of the join. This is illustrated in FIG. 36. A random walk is a path on a graph where each next point is randomly selected from the neighbors of the previous point. For instance, in FIG. 36, a random walk at b_1 has a $\frac{1}{3}$ chance of going to a_1, c_1, or c_2. The statistic is aggregated as the random walk moves through these vertices.

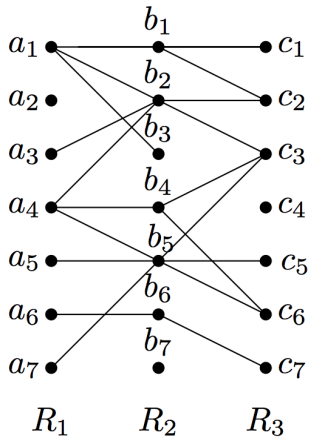


FIG. 36. An example of a connectivity graph of three tables from [43]. The graph represents a join on the tables R_1, R_2, R_3. Each row is represented by a vertex, while two rows with a shared attribute on a column are represented by edges.

Several variants of wander join use slightly different versions of random walks. These are illustrated in FIG. 37. In a chain join, the first element is chosen from the first table, and each step in the random walk can only be selected from elements of the next table. On the three tables given in FIG. 36, valid chain joins are $a_1 \to b_1 \to c_1$ and $a_6 \to b_6 \to c_7$. Chain join fails when an element does not have an edge to the next table. For instance, no chain join in FIG. 36 can start with a_2.

Acyclic join is a generalization of wander join to acyclic graphs. In acyclic join, a join order on the tables is given. Chain join continues until it cannot reach a table, at which point the algorithm picks the last element that can go to that table. For instance, if an acyclic join starts as $a_i \to b_i \to c_i$, and it needs to go to table D but c_i has no neighbors in D but b_i does, the walk will just continue from b_i. Cyclic join is a further generalization of acyclic join to cyclic graphs, and is exactly the same, but a spanning tree is calculated beforehand. Both acyclic and cyclic join are illustrated in FIG. 37. Often, a wander join algorithm will run many random walks of a certain type and size, then average the result.

It is well-known that random walk estimators do not yield uniform distributions. We would like each path to be sampled with equal probability in order for wander join to make sense. The shape of the graph thus induces an element of bias. Fortunately, this bias is easily accounted for. If a join path γ on the graph has a probability $p(\gamma)$, and the expression to be aggregated is $v(\gamma)$, then $v(\gamma)/p(\gamma)$ is an unbiased estimator of $\Sigma_\gamma v(\gamma)$, which is equivalent to the aggregate statistic we want to estimate. It turns out that it is easy to remove this bias. At each step, we simply multiply by the neighbor probability. In the 3-table chain join case given in FIG. 36, the

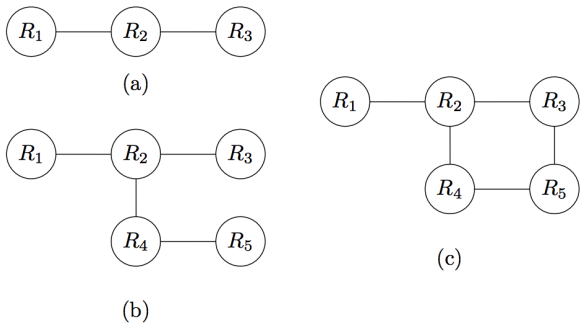


FIG. 37. Three different types of wander join given in [43]. (a) is chain join, and shows a join of order R_1, R_2, R_3. (b) is acyclic join, and shows a join of order R_1, R_2, R_3, R_4, R_5, but where the chosen element of R_3 has no connection to R_4, but the element of R_2 does. (c) shows an example of a possible cycle within a join graph. Cyclic join would first reduce this to (b), then do acyclic join.

formula is simple.

$$p(\gamma) = \frac{1}{|R_1|} * \frac{1}{d_2(t_1)} * \frac{}{d_3(t_2)} \tag{35}$$

In (35), $|R_1|$ is the number of initial elements to choose from in R_1, $d_2(t_1)$ is the number of neighbors in R_2 of the chosen element of R_1, and $d_3(t_2)$ is the number of neighbors in R_3 of the chosen element of R_2. It is easy to see how this formula generalizes for chain join over n tables.

$$p(\gamma) = \frac{1}{|R_1|} * \prod_{i=2}^{n} \frac{1}{d_i(t_{i-1})} \tag{36}$$

This estimator is functionally the same for acyclic join, but is notationally more complicated. It is given in [43]as follows.

$$p(\gamma) = \frac{1}{|R_{\lambda(1)}|} * \prod_{i=2}^{k} \frac{1}{d_{\lambda(i)}(t_{\eta(i)})} \tag{37}$$

In (37), $d_j(t)$ represents the number of items in R_j that can join with item t. Suppose the walk order is $R_{\lambda(1)}, R_{\lambda(1)}, \ldots R_{\lambda(k)}$, and let $R_{\eta(i)}$ be the table adjacent to $R_{\lambda(i)}$ in the query graph but appearing earlier in the walk order. Note that for an acyclic query graph and a valid walk order, $R_{\eta(i)}$ is uniquely defined. This gives the path $\gamma = (t_{\lambda(1)}, ..., t_{\lambda(k)})$, where $t_{\lambda(i)} \in R_{\lambda(i)}$. (37) thus follows.

Wander join has the advantage that it converges much faster than ripple join within a confidence interval. However, it has the downside that the join graph needs to be calculated in the first place. In very large datasets, this is often equivalent to the problem trying to be solved in the first place, though it only needs to be calculated once. Optimization of wander join involves selecting the right type and size of random walk.

4. Deep Learning

The eventual goal of machine learning is to make a truly 'intelligent' machine, one that can match or exceed the capabilities of humans. This means flexibility in all tasks. An 'intelligent' machine should not need careful human tuning and parameter hunting. It should be able to take any arbitrary input and make comments about it.

So far, this 'general algorithm of everything' has remained out of reach. However, the closest that we have been able to reach in attaining this goal has been deep learning. Deep learning is a class of machine learning algorithms that to some degree learn their own structure. They feature multiple layers of abstraction with patterns that reveal intricate structure in datasets. Deep learning has brought about breakthroughs in image processing, audio recognition, AI, and more. Deep learning represents the largest recent breakthrough in machine learning [44].

Deep learning is really little more than a buzzword for a certain type of data structure called a neural network. While every neural network is in principle the same, they are designed very differently to suit the problem at hand. In the rest of this section, we look at neural networks in general, then look at a few specific types and their uses in real research. For a more detailed look at the concepts in this section, see [45].

4.1. Neural Networks

Artificial neural networks, or ANNs, are multi-layered data structures. They, at their core, map an input layer to an output layer through multiple hidden layers of processing. What goes on in the hidden layers is often not fully understood even by computer scientists, but nevertheless requires careful design. A basic visual model of neural networks is given in FIG. 38.

Neural networks were first conceptualized in 1943 by Warren McCulloch [46]. Neural networks are modeled after neuron interactions in the human brain, hence their name. A human neuron aggregates signals from the dendrites, processes them through the axons, and sends signals to adjacent neurons through the terminals. This is shown in FIG. 39. Similarly, a neuron in a neural network takes a series of input signals, aggregates and processes them, and outputs this signal to adjacent neurons. In a neural network, the signals are represented by numerical values from 0 to 1 and the processing step is referred to as an activation function. In our artificial neurons, we simply add up all the signals and directly apply the activation function to get the output. This is shown in FIG. 40

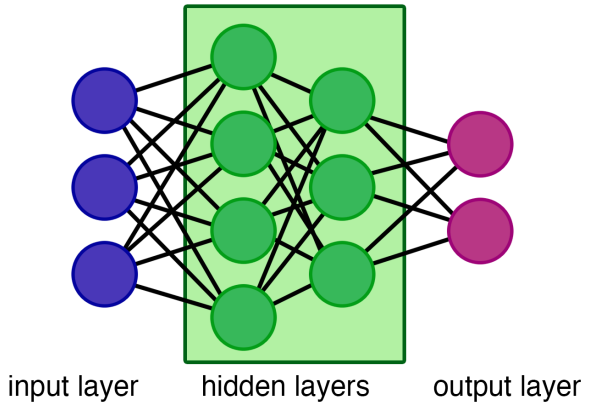


FIG. 38. A simple neural network. Each neuron is represented by a circle. Notice that the size of each layer is arbitrary. In this diagram, each layer is fully connected with neighboring ones.

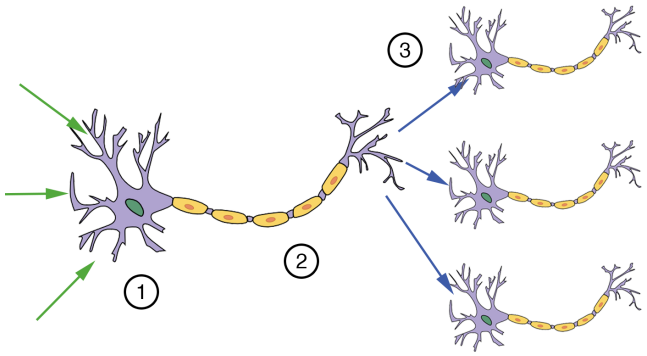


FIG. 39. A model of human neuron interactions. Signals are aggregated at (1), processed at (2), then sent to all neighboring neurons at (3).

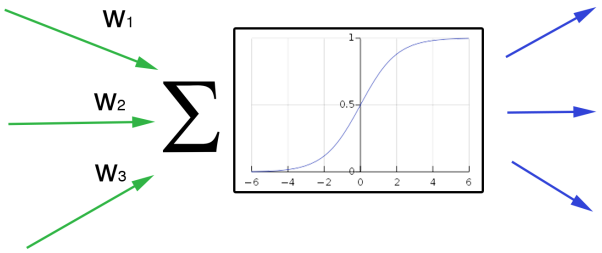


FIG. 40. A neural network neuron. Inputs from the previous layer are weighted and aggregated. The sum is sent through an activation function, and the output is sent to other neurons to be weighted and aggregated again.

Each synapse between neurons in different layers has a weight, or bias. The total size of this matrix is represented by the total number of connections between two layers. If we represent our inputs as the vector $\vec{x}$, then we can obtain the input vector $\vec{y}$ at the next layer by multiplying by a matrix. Between two fully-connected layers A and B, $\vec{x}$ has dimension $|A|$ and $\vec{y}$ has dimension $|B|$. Thus, some $|B| \times |A|$ matrix W will satisfy dimensionality for the equation $\vec{y} = W\vec{x}$. We call this a weight matrix. Each layer of the neural network has a corresponding weight matrix. Two layers use the same matrix, but the missing edges are zero elements in the matrix.

The updated vector is not directly sent through the network to the next step. Neural networks represent degrees of activation, just as neurons can be activated on and off. So, each element of the updated vector gets inputted through an activation function. The naive activation function as defined by our neuron model is given as follows.

$$f(x) = \begin{cases} 1 & x > 0 \\ 0 & x \leq 0 \end{cases} \tag{38}$$

We define the activation function on our layer input vector $\vec{x}$ is the element-wise activation. Given $\vec{x} = (x_1, x_2, \dots x_k)$, we have $f(\vec{x}) = (f(x_1), f(x_2), \dots f(x_k))$. We do not have to choose a binary activation function. Most activation functions are mapped within a closed interval, i.e. they continuously approximate the range $[0, 1]$, such as the sigmoid function. However, even this is not the case for all activation functions. A summary of activation functions is given FIG. 41. Notice that none of the activation functions in FIG. 41 are linear. This is because a linear activation function just gives a linear combination of the output from the weight matrix, or a linear combination of linear functions, or another linear function.

With the dimensions of our weight matrices and activation function, we can now define our forward propagation step. For one calculation on an n-layer neural network, we have an input vector $\vec{x}$ and an output vector $\vec{y}$. Let $\vec{x}_i$ denote the input vector of the ith layer, and W_i denote the $\dim(\vec{x}_{i+1}) \times \dim(\vec{x}_i)$ weight matrix from layer i to $i+1$. f_i denotes the activation function of the ith layer.

$$\vec{x}_{i+1} = f_i(W\vec{x}_i) \tag{39}$$

(39) completely describes how we can calculate each step of our neural network. Our output is given by $\vec{y} = \vec{x}_n$. Later in this section, we will see that recent developments in neural networks have allowed for more complex networks that require more complex calculations.

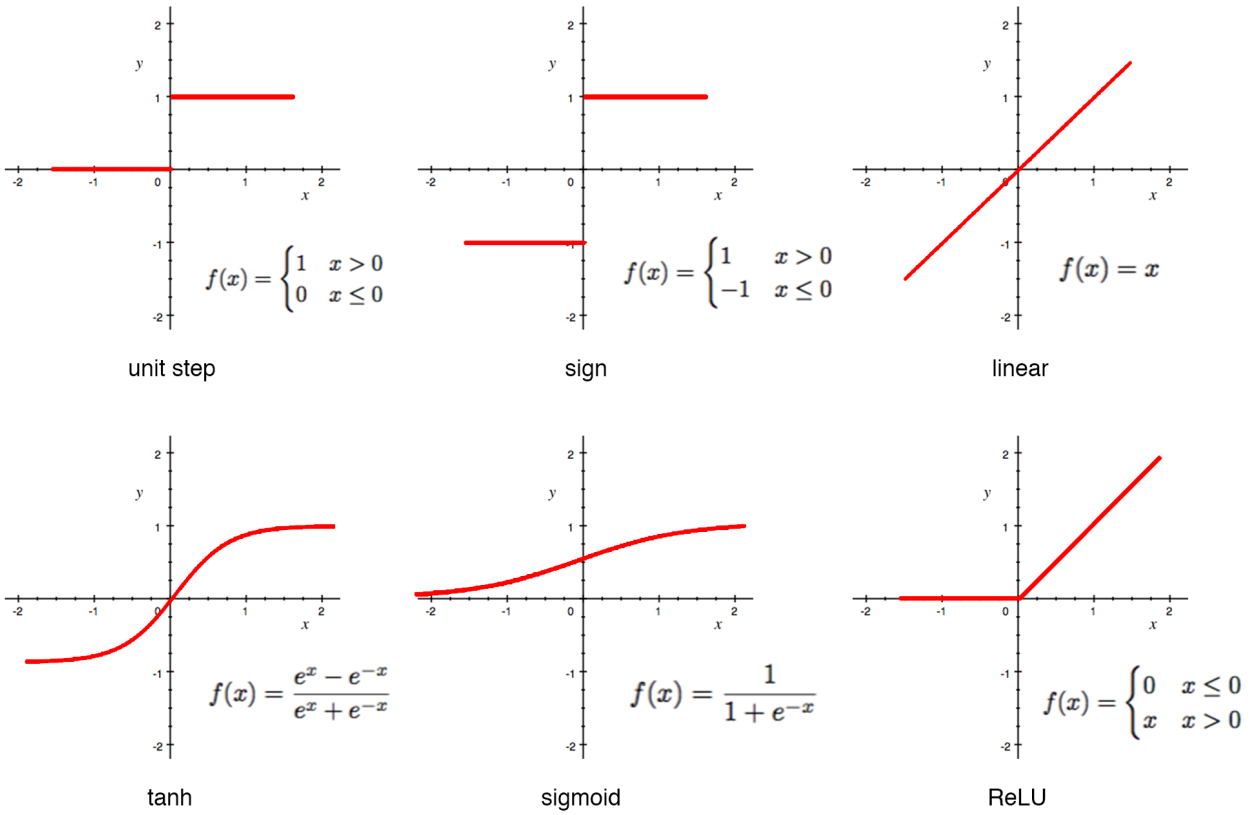


FIG. 41. 6 common activation functions. Notice that each, in some way, represents an 'on/off' state in one way or another, just as neurons do in real life. Also notice that all but one are non-linear, for reasons that will be discussed. The ReLU function stands for Rectified Linear Unit.

Even though neural networks are in theory able to solve any problem provided the right structure of input and output, their optimization requires careful design and knowledge of the problem data. We will not go into details of neural network optimization. It involves designing certain network attributes, such as choosing the right types of layers (e.g. connectivity), layer sizes, how many layers, and the correct activation function. This will be briefly mentioned in later sections.

4.2. Back-Propagation

The back-propagation algorithm is the core of neural networks. When data is trained, the input vector $\vec{x}$ has a known output $\vec{y}$ and an output predicted by the network $\vec{y_0}$. Back-propagation takes the difference between $\vec{y}$ and $\vec{y_0}$ and propagates the error back through the network, with the goal of updating weight matrices in order to reduce this error. Over many iterations, the network should be able to accurately predict in-sample, and ideally out-of-sample, data. Back-propagation allows results from training data to change the parameters of the neural networks. Without back-propagation, neural networks would not be able to grow. In this section, we look at the back-propagation algorithm.

Back-propagation is formulated as an optimization. We can naively write the algorithm as a minimization of errors between actual and predicted terms using the weight matrices as our minimization argument. Rigorously, this is formulated as follows, where W is the set of weight matrices W_i, X is the set of training data $(\vec{x}_i, \vec{y}_i)$, and $\vec{d}_i$ is the predicted output of $\vec{x}_i$ from the neural network.

$$W = \text{argmin}_W \sum_{(\vec{x}_i, \vec{y}_i) \in X} ||\vec{y}_i - \vec{d}_i|| \tag{40}$$

In practice, we use an optimization algorithm called stochastic gradient descent (SGD). In SGD, the gradient of the loss function is calculated and followed. By following the gradient, our position in the search space should eventually reach a local minimum of the loss function. This is shown in FIG. 42. We define the algorithm step as follows, where $\vec{w}$ is the position of the algorithm in the search space and η is the SGD step size, also known as

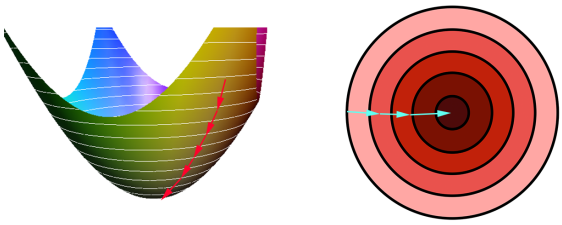


FIG. 42. SGD on a paraboloid with corresponding level curves. A local minimum is found by following the gradient downward.

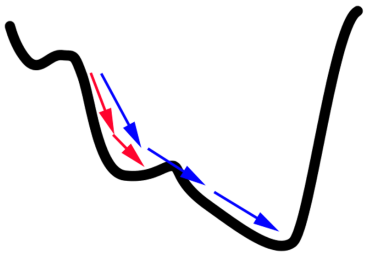


FIG. 43. SGD on a search space with irregularity. Normal SGD gets caught in the local minimum, as shown in red, while momentum-based SVG can overcome it, as shown in blue.

the learning rate.

$$\Delta\vec{w} \coloneqq -\eta\nabla J(\vec{w}) \tag{41}$$

$$J = \frac{1}{2}\sum_{(\vec{x}_i,\vec{y}_i)\in X} ||\vec{y}_i - \vec{d}_i||^2 \tag{42}$$

For neural networks, we imagine the set of weight matrices W to be a very large vector in high-dimensional space. Formulated in this way, the back-propagation algorithm is really very simple. The only details involve actually calculating the gradient. (42) shows the neural network formulation of the loss function. Finding the gradient is often quite simple to find analytically, as derivatives are known to have closed-form solutions on elementary functions. The loss function is formulated as it is in (42) because the power rule for derivatives gives the following gradient, simplifying calculations a little.

$$\nabla J = \sum_{(\vec{x}_i,\vec{y}_i)\in X} ||\vec{y}_i - \vec{d}_i|| * \nabla||\vec{y}_i - \vec{d}_i|| \tag{43}$$

(43) arises from the chain rule. We will not go through the details of deriving the remaining back-propagation formulas, but they are available in [47].

Optimizing neural networks essentially involves optimizing the back-propagation step. We want to be able to consistently find local minima while avoiding overfitting. One potential issue with SGD is the algorithm getting stuck in small irregularities, as illustrated in FIG. 43. In order to avoid this, we add a momentum term, which is analogous to momentum in physics.

$$\Delta\vec{w} \coloneqq -\eta\nabla J(\vec{w}) + \alpha\Delta\vec{w} \tag{44}$$

α, the momentum, represents by how much the additional $\alpha\Delta\vec{w}$ term influences the new gradient step. Sometimes, in very complex search spaces such as image recognition, this will be set to be negative, to discourage the algorithm from stepping in the same direction. [48] uses a negative momentum in an image recognition problem.

Another method for reducing overfitting is statistical regularization. In regularization, we make the sampling distribution of the statistic closer to another distribution by adding a term. The size of the coefficient on this term determines the strength of the regularization. We can do this by constructing the following augmented cost function, and the corresponding modified gradient descent.

$$\hat{J}(\vec{w}) = J(\vec{w}) + \frac{\lambda}{2}\vec{w}^T\vec{w} \tag{45}$$

$$\Delta\vec{w} \coloneqq -\eta\nabla J(\vec{w}) - \eta\lambda\vec{w} \tag{46}$$

In other words, the regularization causes the weight to decay in proportion to its size. The use of the gradient in back-propagation also give us conditions under which we might want to select certain activation functions. The dependency on the gradient means that a mapped neuron with a value very close to 0 or 1 is a problem. This occurs in the sigmoid and *tanh* neurons. Values close to 0 or 1 are 'saturated', where the asymptotic behavior leads the derivative to be 0 and eliminates the gradient descent term. We have discussed that nonlinearity is a requirement for an activation function, so we cannot simply use an identity activation function. A potential solution is to use a ReLU (Rectified Linear Unit) activation, which does not have the issue of saturation but is also not linear. ReLU functions can also be seen as $f(x) = \max(0, x)$. In other words, they just take negative values to 0.

4.3. Convolution

Convolution is a neural network technique developed for image recognition for reducing the dimension of an

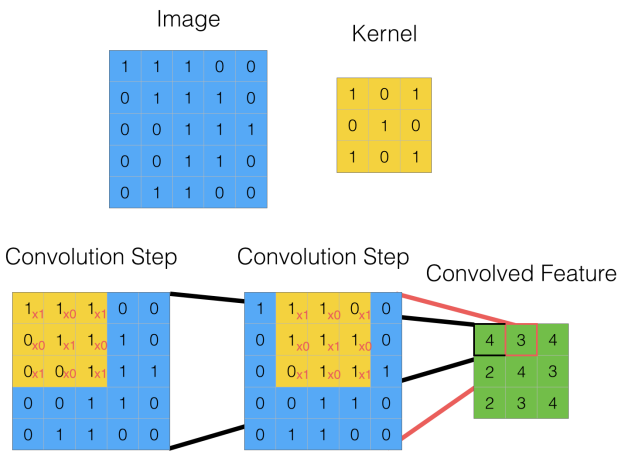


FIG. 44. An example of convolution with stride 1. The image here is represented as 5×5 bitmap, in blue. The convolution matrix, or kernel, is a 3 matrix, in yellow. We show two convolution steps. The sum is taken across the result of multiplying the convolution matrix by the viewed portion of the image, giving the corresponding cell of the convolved feature.

input. This is particularly relevant in image recognition because the input features are so large dimension-wise. Even a very low-resolution 256 by 256 pixel image has $256 * 256 * 3 = 196608$ dimensions. It is thus necessary to reduce the size of this input. As discussed before, one could use PCA, and some image recognition techniques have used PCA as a preprocessing step. However, modern networks have seen the most success using convolutional layers.

A convolution layer is just a generalization of a matrix operation. However, rather than applying a large matrix directly, we take a small matrix and apply it component-wise. Suppose we have some $n \times n$ feature matrix. If we use a 3×3 convolutional matrix, we would first apply the matrix to the cells $1, 1$ to $3, 3$ of our feature matrix, then step one cell over and apply it to cells $1, 2$ to $3, 4$, and so on. We call the number of cells by which we move across each time the stride. FIG. 44 shows a two-step convolution with stride 1.

Different values of the same convolution matrix can yield wildly different convolved features. Due to the similarity of this technique to image filtering, convolution matrices are sometimes also called filters or kernels. Many common image processing techniques can be replicated with convolution matrices. This is shown in FIG. 45, from [49].

Outputs from the convolution matrix are usually into a ReLU function so that negative values are flattened out, since pixels can only take values from 0 to 1. This is shown in FIG. 44 Occasionally, if we do not want to reduce the dimension of our output feature but would

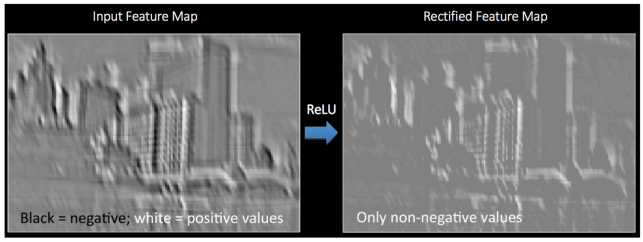


FIG. 45. A convolved feature with the corresponding rectified feature. Negative values are shown in black in the convolved feature.

still like to use a convolution filter, we will pad the data beforehand with zeros. When we do this, we call it wide convolution.

Pooling is a variation of convolution in which the elements of the image viewport are treated as a set. For instance, in max-pooling, the corresponding element in the pooled feature would be the maximum value from that window, whereas in sum-pooling, the corresponding element in the pooled feature would be the sum of all these values. This is shown in FIG. 47, from [50]. An example of the visual results of pooling is shown in FIG. 48.

When used in multiple layers, convolution and pooling layers can generate multiple layers of abstraction.

Convolution and pooling layers allow very high-dimensional vectors to be reduced to smaller ones, and allow for different types of filters to be applied so that specific features can be extracted. They have been used to great effect in image recognition, as will be discussed in the next section.

4.4. Image Recognition

Neural networks have not seen equal success in all fields. As previously discussed, certain type of algorithms suit certain types of problems. Neural networks have seen the most success in image recognition problems. This is because the search space is extremely complicated and unmanageable by traditional means - a single iPhone photo has 12 megapixels at 3 channels each, giving a 36 million-dimensional vector. Photos of the same object at different angles can have wildly different pixel data, further complicating the issue of representing data numerically. Also, neural networks tend to abstract in multiple layers, and images tend to have many layers of abstraction, such as details, lines, edges, objects, etc. This has made neural networks ideal for tackling the problem of image recognition.

First of all, all of these networks are trained on the ImageNet dataset, which comprises of over 15 million labelled images in over 22 thousand categories. This dataset completely and totally dwarfs the size of any previous dataset. The details are given in [51], where

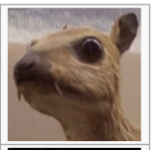
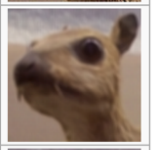
Operation	Filter	Convolved Image
Identity	$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$	
Edge detection	$\begin{bmatrix} 1 & 0 & -1 \\ 0 & 0 & 0 \\ -1 & 0 & 1 \end{bmatrix}$	
	$\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$	
	$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$	
Sharpen	$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$	
Box blur (normalized)	$\frac{1}{9}\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$	
Gaussian blur (approximation)	$\frac{1}{16}\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$	

FIG. 46. Source: [49]. Examples of different convolution matrices applied to an image of a rabbit. Since convolved features take into account one pixel and all its neighbors, it is no surprise that convolution filters have to do with adjacency (blur, edge detection).

the Li Fei-Fei et al. designed a method for quickly collecting large amounts of categorical image data. Ever since, ImageNet has been used as a benchmark of image recognition algorithms through the annual ImageNet Large-Scale Visual Recognition Challenge (ILSVRC).

The first big breakthrough in image recognition was with AlexNet (2012) in [52], which achieved an ILSVRC 2012 top-5 error rate of 15.4%. The breakthrough in this paper was the use of convolutional and pooling layers, which previously saw little use. AlexNet used 5 convolutional and 3 fully-connected layers. AlexNet was also one of the first major nets to use the ReLU activation function, which is now commonplace, but at the time was against the favored tanh function. The basic structure of AlexNet is given in FIG. 49. AlexNet

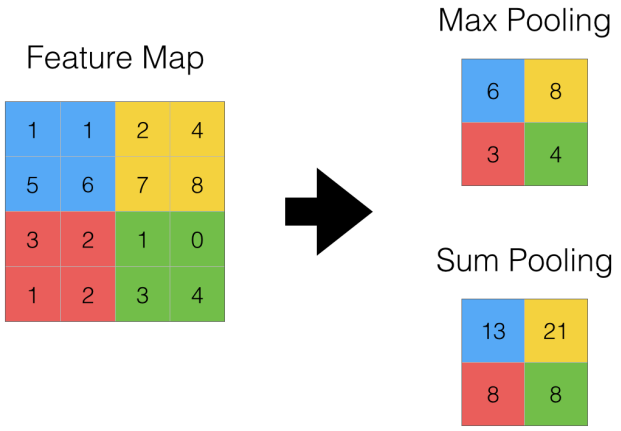


FIG. 47. Illustrated example of sum-pooling and max-pooling on a 4×4 matrix with a 2×2 window with stride 2.

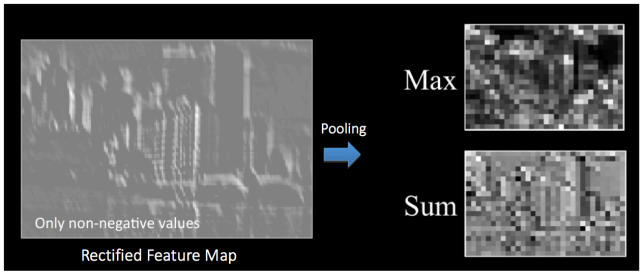


FIG. 48. Source: [50]. Max-pooling and sum-pooling applied to an image feature. Pooling is usually applied after a convolution step, as seen here.

was significant because it showed that intelligent design could lead to improved results.

ZF Net followed up with an ILSVRC 2013 top-5 error rate of 11.2% [53]. Its structure was very similar to AlexNet, with a few minor modifications and optimizations. This is shown in FIG. 50. Importantly, however, AlexNet introduced a method of visualizing the structure of convolutional networks. Designing neural networks often comes down to a matter of trial and error, and in this paper, a system was developed to visualize each layer of the network in an attempt to explain the inner architecture. This will be developed in a later section. ZF Net was important because it showed that trial and

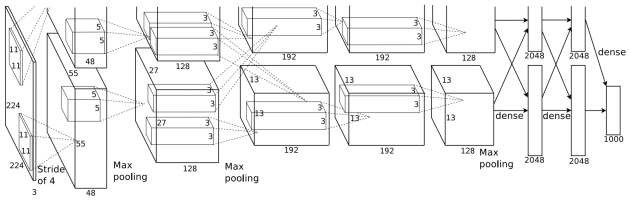


FIG. 49. AlexNet architecture as given in [52].

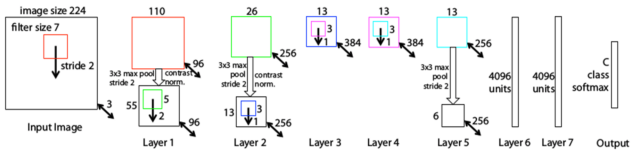


FIG. 50. ZF Net architecture as given in [53].

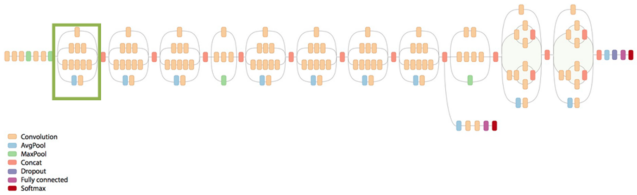


FIG. 51. GoogLeNet architecture as given in [54]. The green rectangle is an inception module and comprises of multiple sequences of convolution layers running in parallel. There are over 130 layers total.

error was not the only way to optimize neural networks.

GoogLeNet achieved an ILSVRC 2014 top-5 error rate of 6.7% [54]. This paper introduced one of the most complex yet computationally efficient networks to date. For a while, network design revolved around essentially stacking convolution and pooling layers on top of each other and hoping for the best result. GoogLeNet introduced the 'inception module', which can be viewed as an encapsulation of network layers. Each inception module is calculated in parallel rather than sequentially, and can be treated as an individual layer. The full structure of GoogLeNet is given in FIG. 51. An inception module is given in FIG. 52.

The originally idea behind the inception module was the naive notion of, 'why not both?' In traditional CNNs, one had to select between convolution or pooling layers, and then filter size beyond that. With the inception module, all of these are computer in parallel, and then aggregated at the end. GoogLeNet was important because it showed that CNNs didn't always have to be

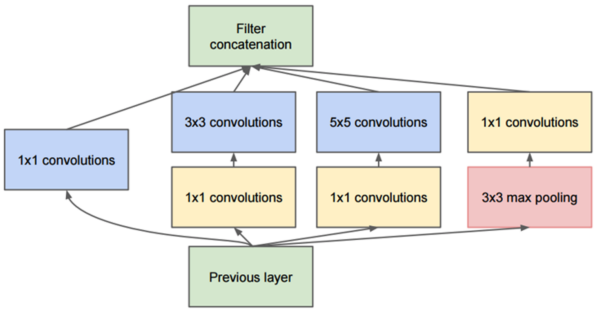


FIG. 52. An example inception module as given in [54].

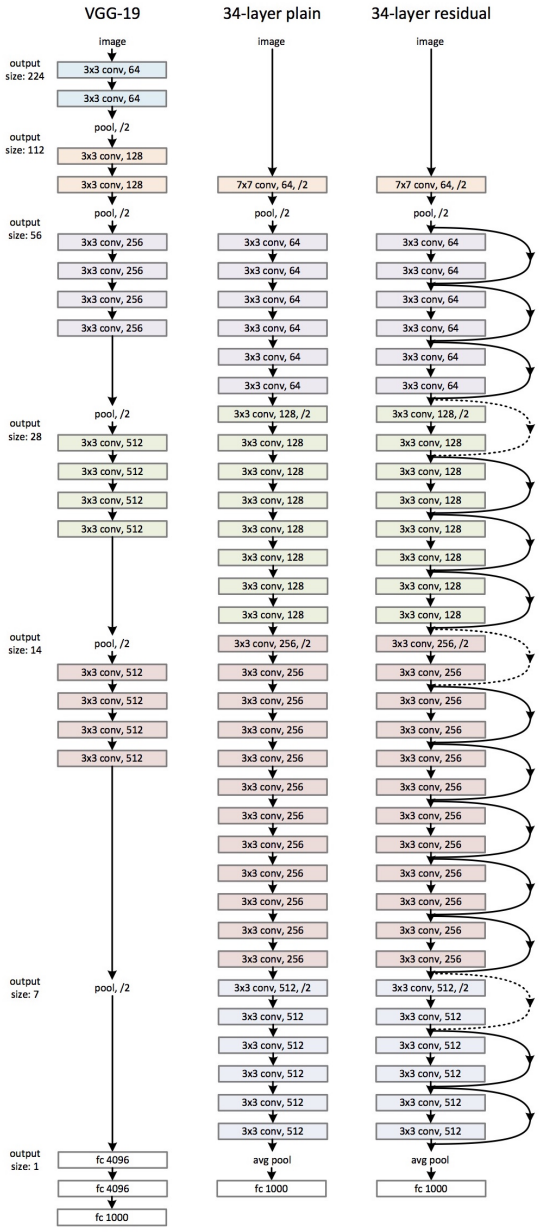


FIG. 53. The network structure of [54]. On the left is VGG19, a common model for convolutional networks. In the middle is a plain network with 34 parameter layers. On the right is the residual network generated from this network. This is one of multiple ImageNet architectures given in the paper.

sequentially stacked. By using the inception module, the authors showed that creative model design is just as important as computational power.

Finally, most recently, Microsoft ResNet achieved a top-5 error rate in ILSVRC 2015 of 3.6%, which beats a majority of human error rates [55]. It uses the deepest network seen to date, with over 150 layers. This structure is shown in FIG. 53. The core

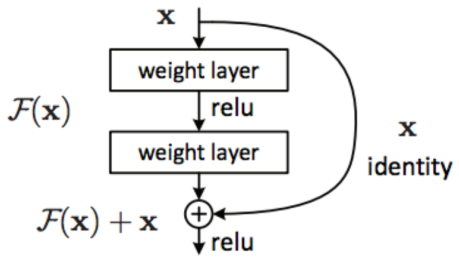


FIG. 54. An example inception module as given in [55]. Here, we see the convolution-ReLU-convolution sequence that is very common in image recognition. We see that the calculated feature is processed as a difference of the original feature.

architecture introduced in this paper was the residual block, shown in FIG. 54. In traditional CNNs, each layer directly calculates the output of the next layer. With residual blocks, the output of each layer is scaled down and added to the previous output, so each layer only calculates a difference in the processed feature.

The authors suggest that residual networks are easier to optimize than the original equivalence mapping, supposedly because the gradient is easier to calculate in the back-propagation step due to the smaller change at each layer. Another point worth noting is that when the same architecture was extended to a 1202-layer network, accuracy actually decreased. This is most likely due to overfitting, illustrating the importance of intelligent network design over brute force computing power.

This brief history of image recognition demonstrates how neural networks can have their structure reworked and adjusted to receive improved results, despite their similarity in core function. It is thought that the image recognition problem has reached the Bayes rate - the lowest possible error rate without statistical variance. In other words, we may be reached the limits of of the ImageNet dataset.

4.5. Learning Network Structure

Neural networks are usually designed with optimization rather than intuition in mind. To improve results, computer scientists find more ways to fit more calculations into less time, rather than trying to improve the underlying mechanical architecture of neural networks. For the most part, we have no idea what actually goes on at each layer. Particularly as modern networks exceed a hundred layers, it is nearly impossible to intuitively understand the operation of each layer. We have seen that convolution layers often correspond to visual filters, but we do not necessarily understand the effect of nesting layers of dozens of convolutional networks.

The first attempt to resolve this was in ZF Net,

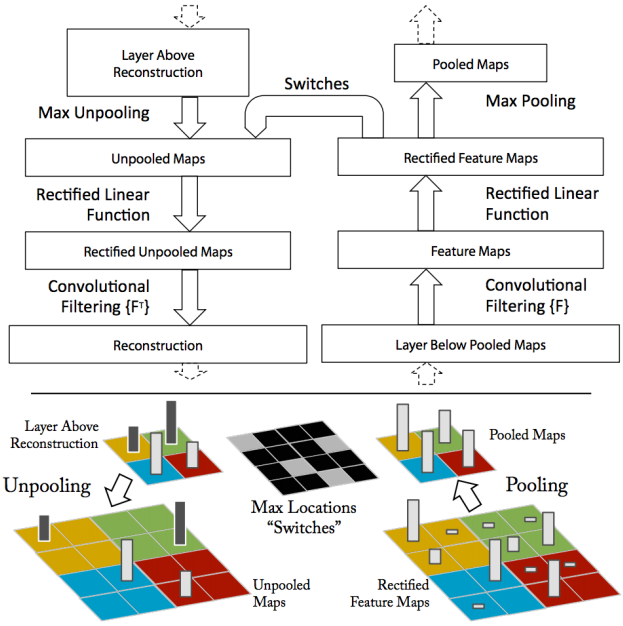


FIG. 55. Deconvnet structure given in [56]. A Deconvnet layer (left) is attached to a convnet layer (right). The Deconvnet will reconstruct an approximate version of the convnet features from the layer beneath. The colored cells on the bottom illustrate the reverse-maxpooling operation in the Deconvnet.

as introduced earlier in [56]. ZF Net introduced a system called Deconvnet which aimed to map activity in intermediate layers back to the input feature space. In other words, this system would indicate what types of patterns in images would cause a greater signal an intermediate layer. In theory, this should describe the underlying structure of these intermediate layers.

Not only can understanding networks allow the construction of more sophisticated networks, it also allows the optimization of existing ones. ZFNet was optimized using Deconvnet, because the visualization system showed that certain features were more common than expected given an unbiased network.

We will not go into the details of Deconvnet, but we will discuss its qualitative features. Suppose we take a feature and process it through a convolution feature and max pooling layer. We can construct approximate inverse functions to undo these operations. For instance, to reverse max pooling, we could assign one of the pooled pixels to the max pooled value, and generate normally distributed values smaller than the maximum to fill in the remaining values. The structure of Deconvnet is given in FIG. 55. Examples from Deconvnet are given in FIG. 56.

The results from Deconvnet grant intuition into the structure of a neural network. Each successive layer is activated by more and more detailed features, starting

FIG. 56. Example calculations from Deconvnet in [56]. For each layer, nine activations and their corresponding pixel space maps are given. As can be seen, early layers only give general shapes and rough outlines, while deeper layers give greater detail and more complex objects. This contributes to the intuition that neural networks operate at multiple layers of abstraction.

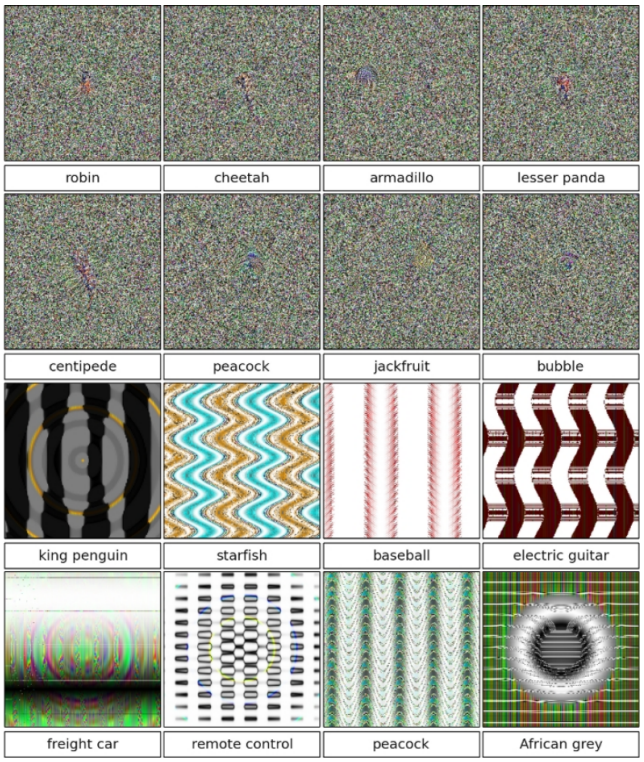


FIG. 57. Examples from [57] of a neural network classifying seemingly random images into object categories with high certainty. This type of analysis can be used to determine network optimizations.

from large shapes, progressing to edges, and eventually ending in entire objects. Understanding this architecture can allow us to develop more efficient neural networks.

As an example of how layer visualization can be used to improve networks, suppose we wanted to determine what the network things the typical banana looks like. We could start with entirely random noise, representing an arbitrary point in the search space, then perform 'gradient ascent' to reverse the neural network and evolve the image to such that it represents what the network thinks is a banana. When this is done, it becomes clear that input images which look nothing like the object in question, or like nothing at all, may at times be categorized incorrectly with a high degree of certainty [57]. This is shown in FIG. 57.

From a network design standpoint, this tells us that our network must be optimized with certain constraints; that more information about the image and the search space is needed. For instance, the use of random noise assumes that pixels are statistically independent, but this is clearly not true in real-world photos, since adjacent pixels are positively correlated. We can redo our optimization using a smoothness prior in consideration, shown in FIG. 58, from [58].

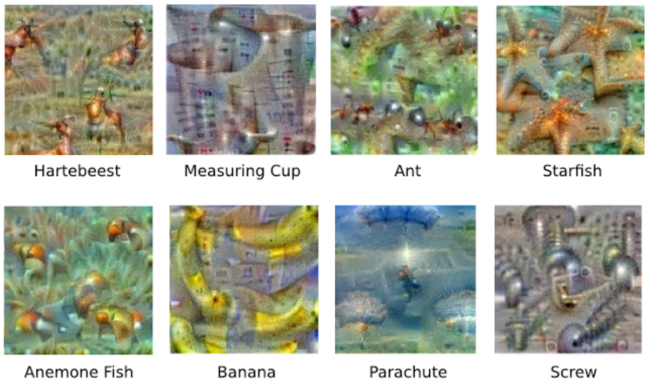


FIG. 58. Learned network optimizations with a smoothness prior from [58]. Each image represents the network's general idea of the respective category.

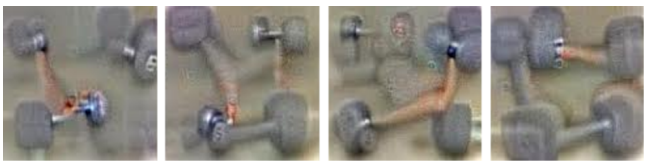


FIG. 59. Learned dumbbells from [58], showing arms. This indicates overfitting.

Using these improved results, we can analyze the success of networks. For instance, in [58], the neural network's learned dumbbell features always contained arms. This is shown in FIG. 59. This information indicates that the network may not actually be learning the shape of a dumbbell itself, but rather of a dumbbell with arms. Such an analysis indicates biased data, which must be corrected in the training step to avoid overfitting.

Mathematical details have been glossed over in this section, as there are many ways to formulate the optimization problem. Further resources are available in [59], [60], [61], [62]. These papers give further detail to the techniques we have in this section. As we have seen, visualizing and understanding networks can give insight and intuition that allow for networks to be optimized for external data sets.

5. The Future of Machine Learning

In this paper, we have given a brief summary of techniques used for analyzing the ever-increasing data in the world. Data will only become increasingly abundant as technology and its ubiquitous use by people expand, and the demand for methods to process this data will increase, too.

Deep learning has been occupying the limelight for nearly five years, with the central focus and poster child of machine learning being computer vision. However,

as we approach the boundaries of current methods, new and creative techniques for creating and analyzing more complex datasets will be necessary. In addition, deep learning will likely see a spread into other fields of computer science.

With neural networks, our ability to analyze data is limited only by our ability to represent it. Major progress in artificial intelligence will evolve through systems that can combine representation with complex reasoning. While the ultimate desire is to create a system that can analyze an arbitrary dataset, the status quo requires data experts and visualization systems to investigate sources of bias and optimize systems.

As machine learning technology expands, it reaches applications beyond academic. Google's Magenta project aims to bridge the gap between technology and art by using deep learning to create art and music [63]. Algorithmic composition techniques have begun to see greater widespread use in interactive media such as video games. At the same time, there exist concerns that the field of machine learning has reduced to blindly throwing more data at existing networks [64]. Despite their flexibility, neural networks remain limited in that humans do not fully understand them. The future will see the development or replacement of these networks.

Acknowledgments

The author of this paper would like to thank Christopher Hogue and the Science Research Program at Choate Rosemary Hall for providing the opportunity to research and write about this topic.

[1] Farabet, Couprie, Najman, LeCun. Learning hierarchical features for scene labeling. *IEEE Trans. Pattern Anal. Mach. Intell.* 35, 1915-1929 (2013)

[2] Page, Lawrence and Brin, Sergey and Motwani, Rajeev and Winograd, Terry. The PageRank Citation Ranking: Bringing Order to the Web. *Stanford InfoLab. Technical report.* (1999)

[3] Sainath, Mohamed, Kingsbury, Ramabhadran. Deep convolutional neural networks for LVCSR. *Proc. Acoustics, Speech and Signal Processing.* 8614?8618 (2013).

[4] Silver, Huang, Maddison, Guez, Sifre, van den Driessche, Schrittwieser, Antonoglou, Panneershelvan, Lanctot, Dieleman, Grewe, Nham, Kalchbrenner, Sutskever, Lillicrap, Leach, Kavukcuoglu, Graepel, Hassabis. Mastering the game of Go with deep neural networks and tree search. *Nature.* 529, 484-489 (2016). p1.

[5] Mohri, Rostamizadeh, Talwalkar. *Foundations of Machine Learning.* MIT Press (2012). p1.

[6] Silver et al. p1.

[7] Deng, Dong, Socher, Li, Li, Fei-Fei. ImageNet: A Large-Scale Hierarchical Image Database. *IEEE Computer Vision and Pattern Recognition (CVPR).* (2009).

[8] Poole. *Linear Algebra: A Modern Introduction.* Cengage Learning (2010). p591.

[9] Ibid.

[10] Mordvintsev, Olah, Tyka. 'Inceptionism: Going Deeper into Neural Networks.' *Google Research Blog.* Google, 17 June 2015. Web. 29 May 2017. <`https://research.googleblog.com/2015/06/inceptionism-going-deeper-into-neural.html`>.

[11] Mohri et al. Foundations of Machine Learning. p63.

[12] Joachims. Text Categorization with Support Vector Machines: Learning with Many Relevant Features. *Machine Learning: ECML-98. ECML 1998. Lecture Notes in Computer Science (Lecture Notes in Artificial Intelligence).* 1398. Springer (2005). p1.

[13] Malisiewicz, Gupta, Efros. Ensemble of Exemplar-SVMs for Object Detection and Beyond. *ICCV.* (2011). p1.

[14] Ibid.

[15] Joachims. p2.

[16] Hsu, Lin. A Comparison of Methods for Multi-class Support Vector Machines. *IEEE. IEEE Transactions on Neural Networks.* 13(2), 415-425 (2002).

[17] Hastie, Tibshirani, Friedman. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction.* 2nd Edition. Springer Series in Statistics. Springer (2009). p267.

[18] Fei-Fei, Ferus, Perona. Learning generative visual models from few training examples: An incremental Bayesian approach tested on 101 object categories. *Computer Vision and Image Understanding.* 106, 59?70 (2007). p1.

[19] Levin, Peres, Wilmer. *Markov Chains and Mixing Times.* American Mathematical Society (2008). p37.

[20] Baldi, Hornik. Neural networks and principal component analysis: Learning from examples without local minima. *Neural Networks.* 2(1), 53-58 (1989).

[21] Dhanachandra, Manglem, Chanu. Image Segmentation Using K -means Clustering Algorithm and Subtractive Clustering Algorithm. *Procedia Computer Science.* 54, 764-771 (2015).

[22] 'Eigenvalue Algorithm.' *Wikipedia.* Wikimedia Foun-

dation, 29 May 2017. Web. 30 May 2017. <https://en.wikipedia.org/wiki/Eigenvalue_algorithm.>

[23] Mohri et al. Foundations of Machine Learning. p282.

[24] Turk, Pentland. Eigenfaces for recognition. *Journal of Cognitive Neuroscience.* 3(1), 71-86 (1991). p1.

[25] 'Anscombe's Quartet.' *Wikipedia.* Wikimedia Foundation, 28 May 2017. Web. 30 May 2017. <https://en.wikipedia.org/wiki/Anscombe\%27s_quartet.>

[26] Bock, Velleman, De Veaux. *Stats: Modeling The World.* 3rd edition. Pearson (2009). p181.

[27] Mohri et al. Foundations of Machine Learning. p5.

[28] Cao, Zhang, Park, Daniels, Crovella, Cowen, Hescott. Going the Distance for Protein Function Prediction: A New Distance Metric for Protein Interaction Networks. *PLOS ONE.* 8(10). p1.

[29] Krizhevsky, Sutskever, Hinton. ImageNet Classification with Deep Convolutional Neural Networks. *Advances in Neural Information Processing Systems.* 25, 1097-1105 (NIPS 2012). p1.

[30] 'Metric (mathematics).' *Wikipedia.* Wikimedia Foundation, 09 May 2017. Web. 30 May 2017. <https://en.wikipedia.org/wiki/Metric_(mathematics)>.

[31] Weisstein. 'L^2-Norm.' *MathWorld: A Wolfram Web Resource.* Wolfram. <http://mathworld.wolfram.com/L2-Norm.html>.

[32] Weisstein. 'L^1-Norm.' *MathWorld: A Wolfram Web Resource.* Wolfram. <http://mathworld.wolfram.com/L1-Norm.html>.

[33] Rowland. 'L^p-Space.' *MathWorld: A Wolfram Web Resource.* Wolfram. <http://mathworld.wolfram.com/Lp-Space.html>.

[34] Hastie et al. The Elements of Statistical Learning. p139.

[35] Cao et al. Going the Distance for Protein Function Prediction. p1.

[36] 'Mahalanobis Distance.' *Wikipedia.* Wikimedia Foundation, 09 May 2017. Web. 30 May 2017. <https://en.wikipedia.org/wiki/Mahalanobis_Distance>.

[37] Brown, Liu, Brodley, Chang. Dis-Function: Learning Distance Functions Interactively. *IEEE. Visual Analytics Science and Technology.* (VAST 2012). p1.

[38] Borg, Groenen. *Modern Multidimensional Scaling: Theory and Applications.* Springer Series in Statistics. Springer (2005).

[39] Hellerstein, Haas, Wang. Online Aggregation. *ACM Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data.* 171-182 (1997).

[40] Ibid. p4.

[41] Ibid. p9.

[42] Haas, Hellerstein. Ripple Joins for Online Aggregation. *ACM Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data.* 287-298 (1999). p1.

[43] Li, Wu, Yi, Zhao. Wander Join: Online Aggregation via Random Walks. *ACM Proceedings of the 2016 International Conference on Management of Data.* 615-629 (2016). p1.

[44] Lecun, Bengio, Hinton. Deep Learning. *Nature.* 521, 436-444 (2015). p6.

[45] Goodfellow, Bengio, Courville. *Deep Learning.* MIT Press (2016).

[46] Cowan. Neural networks: the early days. *Advances in Neural Information Processing Systems.* 2 (1989). p1.

[47] Goodfelow et al. Deep Learning. p151.

[48] Krizhevsky et al. ImageNet Classification with Deep Convolutional Neural Networks. p6.

[49] 'An Intuitive Explanation of Convolutional Neural Networks.' *The Data Science Blog.* 11 August 2016. Web. 30 May 2017. <https://ujjwalkarn.me/2016/08/11/intuitive-explanation-convnets/>.

[50] Ibid.

[51] Deng et al. ImageNet: A Large-Scale Hierarchical Image Database. p1.

[52] Krizhevsky et al. ImageNet Classification with Deep Convolutional Neural Networks. p6.

[53] Zeiler, Fergus. Visualizing and Understanding Convolutional Networks. *arXiv:1311.2901v3.* (2013). p1.

[54] Szegedy, Liu, Jia, Sermanet, Reed, Anguelov, Erhan, Vanhoucke, Rabinovich. Going Deeper with Convolutions. *arXiv*:1409.4842v1. (2014). p1.

[55] He, Zhang, Ren, Sun. Deep Residual Learning for Image Recognition. *arXiv*:1512.03385v1. (2015). p1.

[56] Zeiler, Fergus. Visualizing and Understanding Convolutional Networks. p1.

[57] Nguyen, Yosinski, Clune. Deep Neural Networks are Easily Fooled: High Confidence Predictions for Unrecognizable Images. *Computer Vision and Pattern Recognition, IEEE. arXiv:*1412.1897v4. (2015). p1.

[58] Mordvintsev et al. 'Inceptionism: Going Deeper into Neural Networks.'

[59] Nguyen et al. Deep Neural Networks are Easily Fooled. p1.

[60] Mahendran, Vedaldi. Understanding Deep Image Representations by Inverting Them. *arXiv*:1412.0035v1. (2014). p1.

[61] Dosovitskiy, Brox. Inverting Visual Representations with Convolutional Networks. *arXiv*:1506.02753. (2016). p1.

[62] Simonyan, Vedaldi, Zisserman. Deep Inside Convolutional Networks: Visualising Image Classification Models and Saliency Maps. *arXiv*:1312.6034v2. (2014). p1.

[63] 'Magenta: Make Music and Art Using Machine Learning.' Google. Web. 30 May 2017.<`https://magenta.tensorflow.org/`>.

[64] Russakovsky. 'AI's Research Rut.' *MIT Technology Review*. MIT. 23 August 2016. Web. 30 May 2017. <`https://www.technologyreview.com/s/602157/ais-research-rut/`>.