# Generative Artificial Intelligence (AI) Training

**Bangladesh Korea Information Access Center (BKIAC)**
**Department of Computer Science and Engineering (CSE)**
**Bangladesh University of Engineering and Technology (BUET)**

# Generative Artificial Intelligence (AI) Training

**Published by**

Bangladesh Korea Information Access Center (BKIAC)
Department of Computer Science and Engineering (CSE)
Bangladesh University of Engineering and Technology (BUET)

**Version:** 0.92

**Last modified:** Tuesday the Twenty-Third of April, Two Thousand and Twenty-Four

# Contents at a Glance

# Detailed Contents

*This page is intentionally left blank*

# Part I

# Introduction

# Chapter 1

# Introduction to Artificial Intelligence (AI) and Machine Learning (ML), Basic Pattern Recognition

## 1.1 Artificial Intelligence

Artificial Intelligence (AI) stands at the forefront of technological innovation, representing a fascinating and transformative branch of computer science. The essence of AI lies in the creation of intelligent agents that can simulate human-like cognitive functions, enabling machines to learn, reason, and make decisions autonomously. As we navigate the digital era, AI has become an integral part of our daily lives, influencing diverse fields and revolutionizing the way we interact with technology.

### 1.1.1 Defining Artificial Intelligence

At its core, AI seeks to imbue machines with the ability to perform tasks that typically require human intelligence. This encompasses a broad spectrum of capabilities, including problem-solving, learning from experience, understanding natural language, and adapting to new situations. The ultimate goal is to create systems that can exhibit intelligent behavior and improve their performance over time.

As has been defined in this seminal book[1]:

> Historically, researchers have pursued several different versions of AI. Some have defined intelligence in terms of fidelity to human performance, while others prefer an abstract, formal definition of intelligence called rationality—loosely speaking, doing the "right thing." The subject matter rationality itself also varies:

[1] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 4th ed. Pearson, 2020

some consider intelligence to be a property of internal thought processes and reasoning, while others focus on intelligent behavior, an external characterization.

From these two dimensions—human vs. rational and thought vs. behavior—there are four possible combinations, and there have been adherents and research programs for all four. The methods used are necessarily different: the pursuit of human-like intelligence must be in part an empirical science related to psychology, involving observations and hypotheses about actual human behavior and thought processes; a rationalist approach, on the other hand, involves a combination of mathematics and engineering, and connects to statistics, control theory, and economics.

### 1.1.2 Key Concepts in Artificial Intelligence

**Machine Learning:** Machine Learning (ML) is a subset of AI that focuses on the development of algorithms allowing machines to learn patterns from data and make predictions without explicit programming. It has emerged as a driving force behind the recent advancements in AI.

**Natural Language Processing (NLP):** NLP is a field within AI that deals with the interaction between computers and human language. It enables machines to understand, interpret, and generate human-like text, powering applications such as chatbots, language translation, and voice recognition.

**Computer Vision:** Computer Vision involves equipping machines with the ability to interpret and make decisions based on visual data. This encompasses image recognition, object detection, and scene understanding.

**Robotics:** AI plays a pivotal role in robotics by enabling machines to perceive their environment, make decisions, and execute tasks autonomously. This has applications in industries ranging from manufacturing to healthcare.

### 1.1.3 Types of Artificial Intelligence

**Narrow or Weak AI:** Narrow AI is designed for a specific task and operates within a limited domain. It excels at well-defined tasks but lacks the broad cognitive abilities associated with human intelligence.

Examples of Narrow or Weak AI include the following:

i) **Virtual Personal Assistants (VPAs):** Voice-activated virtual assistants like Amazon's Alexa, Apple's Siri, or Google Assistant are designed for specific tasks, such as answering questions, setting reminders, or providing weather updates. They excel in their designated functions but do not possess a broader understanding of the world.

ii) **Image Recognition Systems:** AI systems trained for image recognition tasks, such as identifying objects in photographs or videos, are narrow AI. Applications like

facial recognition software or systems used for quality control in manufacturing fall into this category.

iii) **Recommendation Systems:** Platforms like Netflix or Amazon use narrow AI to provide personalized recommendations based on a user's viewing or purchasing history. These systems focus on a specific task—suggesting content—and lack the broad cognitive abilities of a human.

**General or Strong AI:** General AI aims to replicate the full spectrum of human intelligence across a wide range of tasks. This remains an aspirational goal, and current AI systems are predominantly narrow in their focus.

Examples of General or Strong AI (*aspirational*) include the following:

i) **Humanoid Robots with General Abilities:** The concept of a humanoid robot that possesses a broad spectrum of cognitive abilities, akin to a human, represents the goal of general AI. While current robots are task-specific and lack general intelligence, the aspiration is to create robots capable of performing various tasks with human-like adaptability.

ii) **Autonomous Agents with Cognitive Flexibility:** General AI would involve creating autonomous agents that can adapt to diverse environments, learn from different situations, and perform a wide array of tasks without specialized programming. This level of flexibility remains an aspirational goal for AI researchers.

iii) **Comprehensive Natural Language Understanding:** Achieving general AI would involve developing systems with a deep understanding of natural language across various contexts. These systems would not only comprehend language but also exhibit reasoning and learning abilities similar to human cognition.

### 1.1.4 Ethical Considerations

As AI continues to advance, ethical considerations surrounding its use become increasingly important. Issues such as bias in algorithms, data privacy, and the potential impact on employment are subjects of ongoing discourse within the AI community and society at large.

### 1.1.5 Comments

Artificial Intelligence stands as a testament to humanity's pursuit of creating machines that can emulate, and in some cases surpass, human intelligence. As AI technologies evolve, they hold the promise of addressing complex challenges, enhancing efficiency, and reshaping the landscape of various industries. Embracing the potential and understanding the ethical dimensions of AI is pivotal as we embark on this transformative journey into the age of intelligent machines.

## 1.2 Machine Learning

Machine Learning (ML) is the field of study that gives computers the capability to learn without being explicitly programmed. ML is one of the most exciting technologies that one would have ever come across. As it is evident from the name, it gives the computer that makes it more similar to humans: The ability to learn. Machine learning is actively being used today, perhaps in many more places than one would expect.

ML is a dynamic and transformative field at the intersection of computer science and statistics, aiming to develop algorithms and models that enable computers to learn from data and improve their performance over time.

In essence, ML empowers machines to recognize patterns, make decisions, and predictions without being explicitly programmed for each task. This paradigm shift from rule-based programming to data-driven learning has had profound implications across various industries, paving the way for innovative solutions and advancements in artificial intelligence.

### 1.2.1 Understanding Machine Learning

At the core of Machine Learning is the concept of learning from examples and experiences. Instead of relying on explicit programming instructions, ML algorithms leverage data to identify underlying patterns and make informed decisions. The process involves training a model on a dataset, allowing it to generalize and make predictions or classifications on new, unseen data.

### 1.2.2 Key Components of Machine Learning

**Data:** High-quality and diverse datasets are fundamental to ML. The model learns patterns and behaviors from the data it is exposed to.

**Algorithms:** ML employs a variety of algorithms, each designed for specific tasks such as classification, regression, clustering, and more.

**Features:** In ML, features are the input variables or attributes used to make predictions or classifications.

**Labels:** In supervised learning, models are trained on labeled datasets where each example is paired with a corresponding output label.

### 1.2.3 Types of Machine Learning

There are various ways to classify machine learning problems. Here, we discuss the most obvious ones.

#### 1.2.3.1 Nature of the Learning "Signal" or "Feedback" Available

**Supervised learning:** The model or algorithm is presented with example inputs and their desired outputs and then finds patterns and connections between the input and the

output. The goal is to learn a general rule that maps inputs to outputs. The training process continues until the model achieves the desired level of accuracy on the training data.

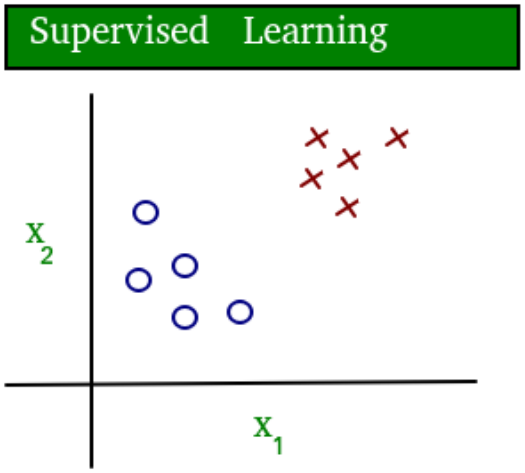



Figure 1.1: Supervised learning.

Some real-life examples of supervised learning are:

**Image Classification:** You train with images/labels. Then in the future, you give a new image expecting that the computer will recognize the new object.

**Market Prediction/Regression:** You train the computer with historical market data and ask the computer to predict the new price in the future.

**Unsupervised learning:** No labels are given to the learning algorithm, leaving it on its own to find structure in its input. It is used for clustering populations in different groups. Unsupervised learning can be a goal in itself (discovering hidden patterns in data).

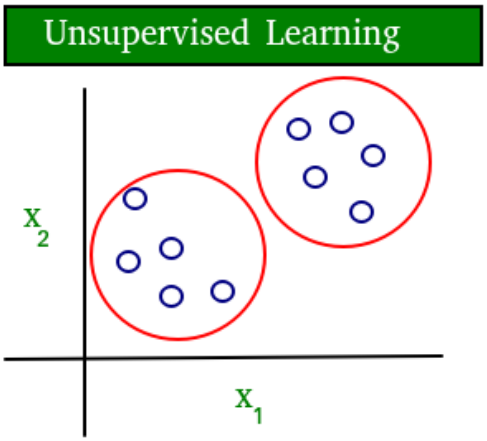



Figure 1.2: Unsupervised learning.

**Clustering:** You ask the computer to separate similar data into clusters, this is essential in research and science.

**High-Dimension Visualization:** Use the computer to help us visualize high-dimension data.

**Generative Models:** After a model captures the probability distribution of your input data, it will be able to generate more data. This can be very useful to make your classifier more robust.

As you can see, the data in supervised learning is labeled, whereas data in unsupervised learning is unlabeled.

**Semi-supervised learning:** Problems where you have a large amount of input data and only some of the data is labeled, are called semi-supervised learning problems. These problems sit in between both supervised and unsupervised learning. For example, a photo archive where only some of the images are labeled, (e.g. dog, cat, person) and the majority are unlabeled.

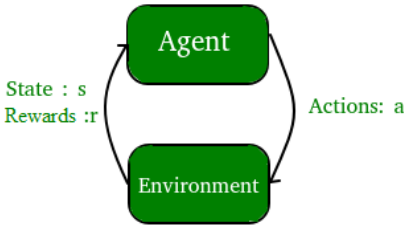



Figure 1.3: Reinforcement learning.

**Reinforcement learning:** A computer program interacts with a dynamic environment in which it must perform a certain goal (such as driving a vehicle or playing a game against an opponent). The program is provided feedback in terms of rewards and punishments as it navigates its problem space.

#### 1.2.3.2 Use Cases of Supervised Learning

**Classification:** Inputs are divided into two or more classes, and the learner must produce a model that assigns unseen inputs to one or more (multi-label classification) of these classes and predicts whether or not something belongs to a particular class. This is typically tackled in a supervised way.

Classification models can be categorized in two groups: Binary classification and Multiclass Classification. Spam filtering is an example of binary classification, where the inputs are email (or other) messages and the classes are “spam” and “not spam”.

**Regression:** It is also a supervised learning problem, that predicts a numeric value and outputs are continuous rather than discrete. For example, predicting stock prices using historical data.

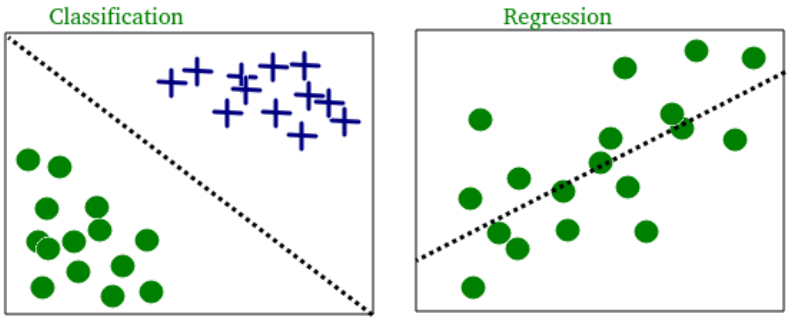



Figure 1.4: Classification and regression.

### 1.2.4 Terminologies of Machine Learning

**Model:** A model is a specific representation learned from data by applying some machine learning algorithm. A model is also called a hypothesis.

**Feature:** A feature is an individual measurable property of our data. A set of numeric features can be conveniently described by a feature vector. Feature vectors are fed as input to the model. For example, in order to predict a fruit, there may be features like color, smell, taste, etc.

Note: Choosing informative, discriminating and independent features is a crucial step for effective algorithms. We generally employ a feature extractor to extract the relevant features from the raw data.

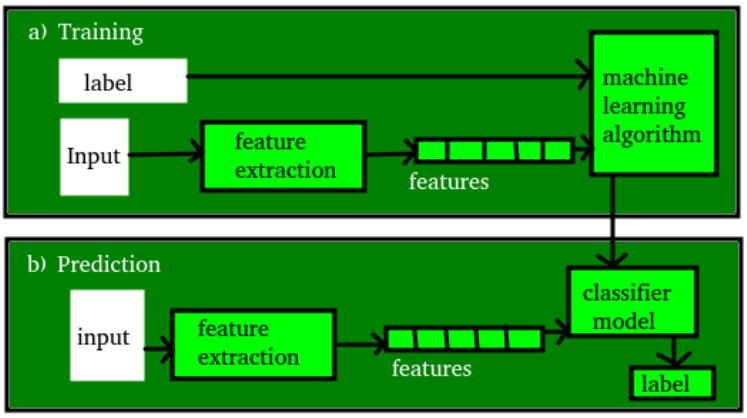



Figure 1.5: Terminologies of machine learning.

**Target (Label):** A target variable or label is the value to be predicted by our model. For the fruit example discussed above, the label with each set of input would be the name of the fruit like apple, orange, banana, etc.

**Training:** The idea is to give a set of inputs (features) and its expected outputs (labels), so after training, we will have a model (hypothesis) that will then map new data to one of the categories trained on.

**Prediction:** Once our model is ready, it can be fed a set of inputs to which it will provide a predicted output (label). But make sure if the machine performs well on unseen data, then only we can say the machine performs well.

### 1.2.5 ML Models

On the basis of these machine learning tasks/problems, we have a number of algorithms that are used to accomplish these tasks. Some commonly used machine learning algorithms are Linear Regression, Logistic Regression, Decision Tree, SVM (Support vector machines), Naive Bayes, KNN (K nearest neighbors), K-Means, Random Forest, etc.

### 1.2.6 Comments

Machine Learning stands as a revolutionary force, shaping the landscape of technology and problem-solving. Its ability to uncover patterns in data and make intelligent decisions has made it an essential tool in various domains. As technology continues to advance, the integration of Machine Learning promises continued innovation, making it an exciting and indispensable field within the broader realm of artificial intelligence.

## 1.3 Basic Pattern Recognition

Pattern is everything around in this digital world. A pattern can either be seen physically or it can be observed mathematically by applying algorithms.

The colors on the clothes, speech pattern, etc are all examples of patterns. In computer science, a pattern is represented using vector feature values.

### 1.3.1 What is Pattern Recognition?

Pattern recognition is the process of recognizing patterns by using a machine learning algorithm. Pattern recognition can be defined as the classification of data based on knowledge already gained or on statistical information extracted from patterns and/or their representation. One of the important aspects of pattern recognition is its application potential.

Examples include speech recognition, speaker identification, multimedia document recognition (MDR), automatic medical diagnosis. In a typical pattern recognition application, the raw data is processed and converted into a form that is amenable for a machine to use. Pattern recognition involves the classification and cluster of patterns.

In classification, an appropriate class label is assigned to a pattern based on an abstraction that is generated using a set of training patterns or domain knowledge. Classification is used in supervised learning.

Clustering generates a partition of the data which helps decision making, the specific decision-making activity of interest to us. Clustering is used in unsupervised learning.

Features may be represented as continuous, discrete, or discrete binary variables. A feature is a function of one or more measurements, computed so that it quantifies some significant characteristics of the object.

If we consider our face then eyes, ears, nose, etc are features of the face. A set of features that are taken together, forms the features vector.

In the above example of a face, if all the features (eyes, ears, nose, etc) are taken together then the sequence is a feature vector (`[eyes, ears, nose]`). The feature vector is the sequence of a feature represented as a $d$-dimensional column vector. In the case of speech, MFCC (Mel-frequency Cepstral Coefficient) is the spectral feature of the speech. The sequence of the first 13 features forms a feature vector.

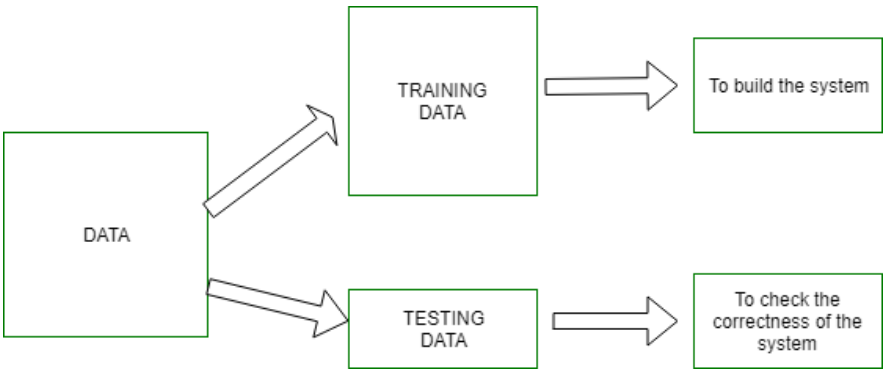



Figure 1.6: The process of training and learning in pattern recognition.

### 1.3.2 Training and Learning in Pattern Recognition

Learning is a phenomenon through which a system gets trained and becomes adaptable to give results in an accurate manner. Learning is the most important phase as to how well the system performs on the data provided to the system depends on which algorithms are used on the data. The entire dataset is divided into two categories, one which is used in training the model i.e. Training set, and the other that is used in testing the model after training, i.e. Testing set.

#### 1.3.2.1 Training Set

The training set is used to build a model. It consists of the set of images that are used to train the system. Training rules and algorithms are used to give relevant information on how to associate input data with output decisions. The system is trained by applying these algorithms to the dataset, all the relevant information is extracted from the data, and results are obtained. Generally, 80% of the data of the dataset is taken for training data.

#### 1.3.2.2 Testing Set

Testing data is used to test the system. It is the set of data that is used to verify whether the system is producing the correct output after being trained or not. Generally, 20% of the data

**Gini Index**

The Gini index, often used in the context of decision tree algorithms, is a metric that quantifies the impurity or disorder within a dataset. Specifically, in a classification scenario with multiple classes, the Gini index measures the probability of incorrectly classifying an element chosen randomly from the dataset. A lower Gini index indicates higher purity and better separation of classes. During the construction of a decision tree, nodes are split based on features to minimize the Gini index, resulting in a tree structure that effectively discriminates between different classes. The Gini index serves as a valuable criterion for assessing the effectiveness of splits and guiding the creation of decision trees that make accurate and efficient predictions in machine learning tasks.

of the dataset is used for testing. Testing data is used to measure the accuracy of the system. For example, a system that identifies which category a particular flower belongs to is able to identify seven categories of flowers correctly out of ten and the rest of others wrong, then the accuracy is 70%.

## 1.4 Hands On

### 1.4.1 Colab

Colab, or "Colaboratory", allows you to write and execute Python in your browser, with:

- Zero configuration required.
- Access to GPUs free of charge.
- Easy sharing.

Colab is available at, `https://colab.research.google.com/`.

### 1.4.2 Survivors on the Titanic

The sinking of the Titanic is one of the most infamous shipwrecks in history.

On April 15, 1912, during her maiden voyage, the widely considered "unsinkable" RMS Titanic sank after colliding with an iceberg. Unfortunately, there weren't enough lifeboats for everyone on board, resulting in the death of 1502 out of 2224 passengers and crew.

While there was some element of luck involved in surviving, it seems some groups of people were more likely to survive than others.

In this task, we build a predictive model that answers the question: "what sorts of people were more likely to survive?" using passenger data (ie. name, age, gender, socio-economic class, etc).

#### 1.4.2.1 Why Choose `max_depth=3`?

The depth of the tree is known as a hyperparameter, which means a parameter you need to decide before you fit the model to the data. If you choose a larger `max_depth`, you'll get a more complex decision boundary.

If your decision boundary is too complex, you can overfit to the data, which means that your model will be describing noise as well as signal.

If your `max_depth` is too small, you might be underfitting the data, meaning that your model doesn't contain enough of the signal.

But how do you tell whether you're overfitting or underfitting? This is also referred to as the bias-variance trade-off.

One way is to hold out a test set from your training data. You can then fit the model to your training data, make predictions on your test set and see how well your prediction does on the test set.

We will now do this: split your original training data into training and test sets. Then, we will iterate over values of `max_depth` ranging from 1 to 9 and plot the accuracy of the models on training and test sets.

### 1.4.3 Predict the $CO_2$ Emission of a Car

Our dataset contains some information about cars. We can predict the $CO_2$ emission of a car based on the size of the engine, but with multiple regression we can throw in more variables, like the weight of the car, to make the prediction more accurate.

### 1.4.4 Predict the Species of Iris Flowers

We have three iris species with 50 samples each as well as some properties of each flower. One flower species is linearly separable from the other two, but the other two are not linearly separable from each other.

### 1.4.5 Predict a Given Fruit

Our goal is to predict fruit from its color. To do this, we use a simple pattern recognition algorithm called $k$-nearest neighbors ($k$NN).

### 1.4.6 For You to Try — Predict the Share Prices at Dhaka Stock Exchange PLC

Get historical share prices for a certain stock from `https://www.dse.com.bd/latest_share_price_scroll_l.php` for a certain period, say, three years. Train your model using any method you like. Or train for several methods and pick the best one. Then check how well your model performs.

What are the features you are going to use?

# Chapter 2

# Support Vector Machine (SVM), Gradient Boosting, and K-Nearest Neighbor (KNN) Algorithm

## 2.1 Support Vector Machine (SVM) Algorithm

Support Vector Machine (SVM) is a powerful machine learning algorithm used for linear or nonlinear classification, regression, and even outlier detection tasks. SVMs can be used for a variety of tasks, such as text classification, image classification, spam detection, handwriting identification, gene expression analysis, face detection, and anomaly detection. SVMs are adaptable and efficient in a variety of applications because they can manage high-dimensional data and nonlinear relationships.

SVM algorithms are very effective as we try to find the maximum separating hyperplane between the different classes available in the target feature.

### 2.1.1 Support Vector Machine

Support Vector Machine (SVM) is a supervised machine learning algorithm used for both classification and regression. Though we say regression problems as well it's best suited for classification. The main objective of the SVM algorithm is to find the optimal hyperplane in an $N$-dimensional space that can separate the data points in different classes in the feature space. The hyperplane tries that the margin between the closest points of different classes should be as maximum as possible. The dimension of the hyperplane depends upon the number of features. If the number of input features is two, then the hyperplane is just a line. If the number of input features is three, then the hyperplane becomes a 2-D plane. It becomes difficult to imagine when the number of features exceeds three.

Let's consider in Figure 2.1 two independent variables $x_1$, $x_2$, and one dependent variable which is either a blue circle or a red circle.

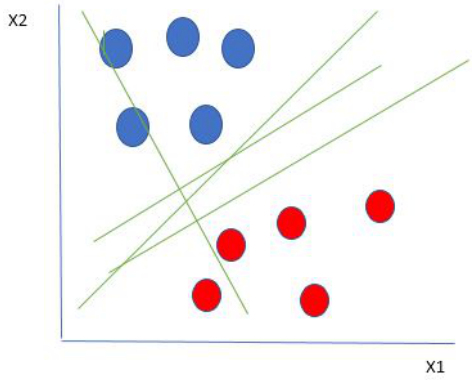



Figure 2.1: Linearly separable data points.

From the figure it's very clear, as shown in Figure 2.2, that there are multiple lines (our hyperplane here is a line because we are considering only two input features $x_1$, $x_2$) that segregate our data points or do a classification between red and blue circles. So how do we choose the best line or in general the best hyperplane that segregates our data points?

### 2.1.2 How Does SVM Work?

One reasonable choice as the best hyperplane is the one that represents the largest separation or margin between the two classes.

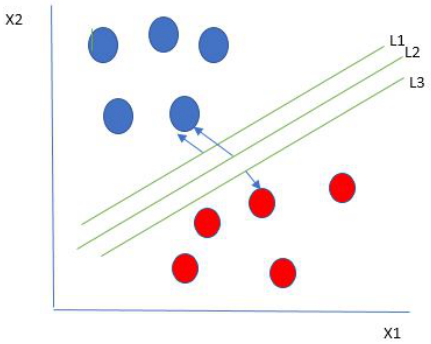



Figure 2.2: Multiple hyperplanes separate the data from two classes.

So we choose the hyperplane whose distance from it to the nearest data point on each side is maximized. If such a hyperplane exists it is known as the maximum-margin hyperplane/hard margin. So from Figure 2.2, we choose $L_2$.

Let's consider a scenario like shown in Figure 2.3. Here we have one blue ball in the boundary of the red ball. So how does SVM classify the data?

The blue ball in the boundary of red ones is an outlier of blue balls. The SVM algorithm has the characteristics to ignore the outlier and finds the best hyperplane that maximizes the margin. SVM is robust to outliers.

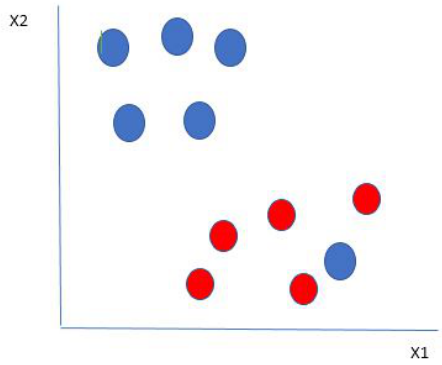



Figure 2.3: Selecting hyperplane for data with outlier.

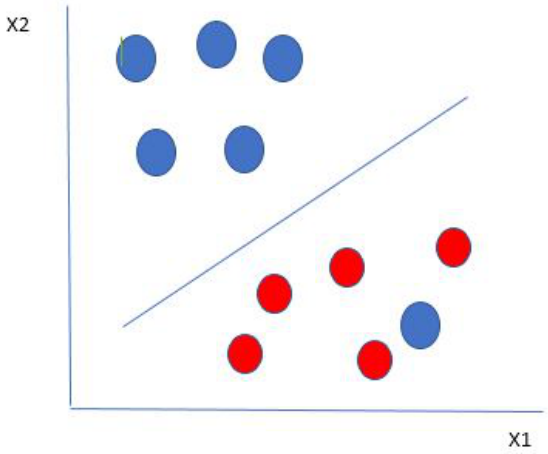



Figure 2.4: Hyperplane which is the most optimized one.

So in this type of data point what SVM does is, finds the maximum margin as done with previous data sets along with that it adds a penalty each time a point crosses the margin.

Till now, we were talking about linearly separable data (the group of blue balls and red balls are separable by a straight line/linear line). What to do if data are not linearly separable?

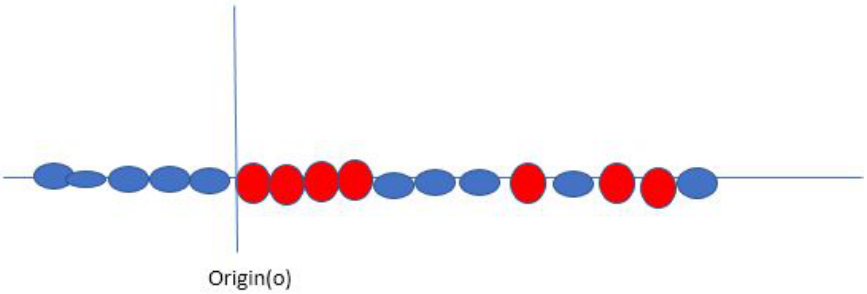



Figure 2.5: Original 1D dataset for classification.

Say, our data is shown in Figure 2.5. SVM solves this by creating a new variable using a kernel. We call a point $x_i$ on the line and we create a new variable $y_i$ as a function of distance from

origin $o$. So if we plot this we get something like as shown in Figure 2.6.

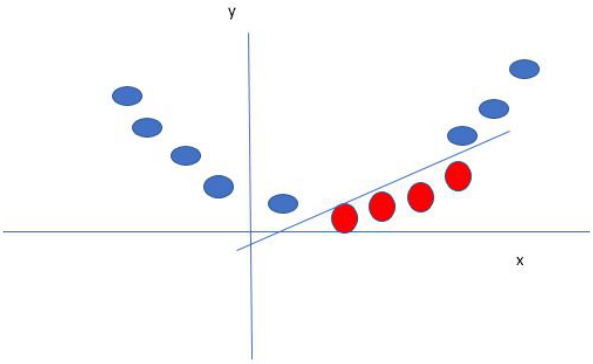



Figure 2.6: Mapping 1D data to 2D to become able to separate the two classes.

In this case, the new variable $y$ is created as a function of distance from the origin. Kernel function maps data from one space to another space.

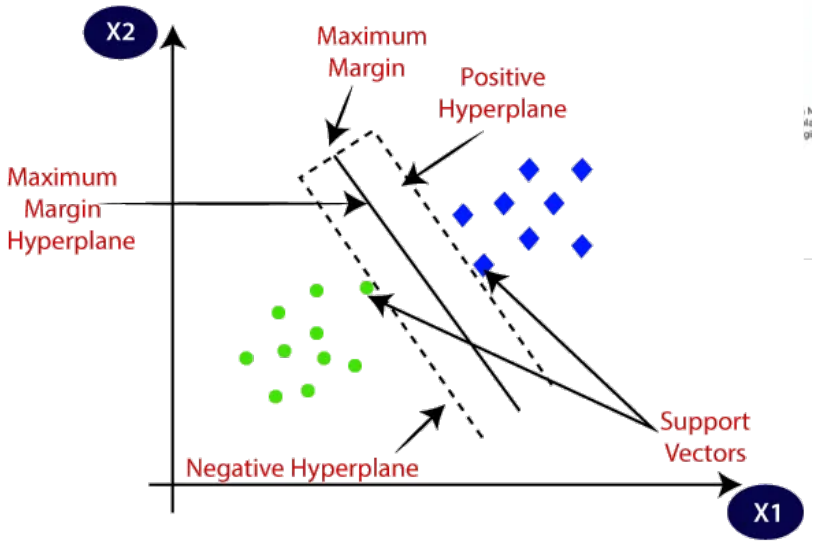



Figure 2.7: Hyperplane and support vector concepts.

#### 2.1.2.1 Significance of C Value in Support Vector Machine

Support Vector Machine always tries to achieve two goals:

i. Setting a larger margin.

ii. Lowering misclassification rate.

Now the problem is that the above two goals are contradictory. If we increase the margin, we will end up getting a high misclassification rate. On the other hand if we decrease the margin, we will end up getting a lower misclassification rate.

Though it may appear that rather than wanting a larger margin, our priority should be getting a lower misclassfication rate, as a matter of fact the issues are not simple. The goals quoted

above are for the training dataset. Lower misclassification on training dataset doesn't mean lower misclassification on validation/testing data. To get a better a result of testing data, SVM looks for a higher margin.

The parameter $C$ in SVM helps control the trade-off between the training error and the margin since it can determine the penalty for misclassified data points during the training process.

To be specific, a smaller value of $C$ allows for a larger margin, potentially leading to more misclassifications on the training data. On the other hand, a larger value of $C$ puts more emphasis on minimizing the training error, potentially leading to a narrower margin.

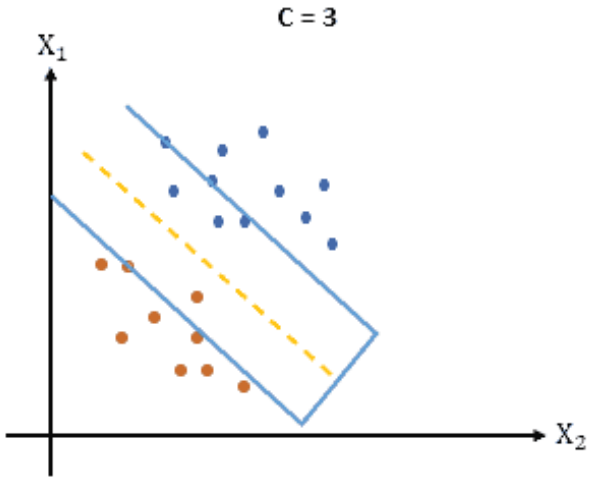



Figure 2.8: Effect of a small $C$ value on the margin.

**Small C Value — Large Margin** When the value of $C$ is small, the SVM algorithm focuses more on achieving a larger margin. In other words, a smaller $C$ value allows for more misclassifications in the training data, which can result in a wider margin between the classes.

Note that this can be useful in scenarios where the data points are well-separated, and there is a low presence of noise or outliers. However, it is important to be cautious as setting $C$ too small can lead to underfitting, where the model fails to capture the underlying patterns in the data.

Figure 2.8 provides a visual representation of the effect of a small $C$ value on the margin.

**Large C Value — Narrow Margin** Conversely, a larger value of $C$ in SVM emphasizes minimizing the training error, potentially resulting in a narrower margin. When $C$ is set to a large value, the SVM algorithm seeks to fit the training data as accurately as possible, even if it means sacrificing a wider margin.

Remember that this can be beneficial when the data points are not well-separated or when there is a significant presence of noise or outliers. However, setting $C$ too large can increase the risk of overfitting, where the model becomes too specific to the training data and performs poorly on new, unseen data.

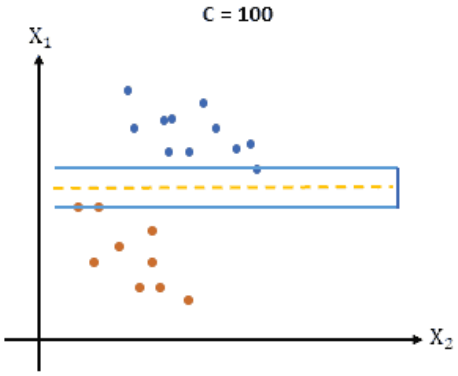



Figure 2.9: Impact of a large $C$ value on the margin.

Figure 2.9 visually illustrates the impact of a large $C$ value on the margin.

#### 2.1.2.2 Selecting the Optimal Value of C

Selecting the optimal value of $C$ is crucial for achieving the best performance of an SVM model. Moreover, the choice of $C$ depends on the specific problem we are trying to solve, the dataset, and desired trade-offs between training error and margin width.

The approaches to determining the optimal value of $C$ include grid search, randomized search, Bayesian optimization, and metaheuristic algorithms.

### 2.1.3 Support Vector Machine Terminology — Summarized

**Hyperplane:** The best decision boundary is called hyperplane. It is the decision boundary that is used to separate the data points of different classes in a feature space. In the case of linear classifications, it will be a linear equation i.e. $wx + b = 0$.

**Support Vectors:** SVM chooses the extreme points/vectors that helps in creating a hyperplane. Support vectors are the closest data points to the hyperplane. These extreme cases are called as support vectors. Hence algorithm is termed as support vector machine. The support vectors make a critical role in deciding the hyperplane and margin.

**Margin:** Margin is the distance between the support vector and hyperplane. The main objective of the support vector machine algorithm is to maximize the margin. The wider margin indicates better classification performance.

**Kernel:** Kernel is the mathematical function, which is used in SVM to map the original input data points into high-dimensional feature spaces, so, that the hyperplane can be easily found out even if the data points are not linearly separable in the original input space. Some of the common kernel functions are linear, polynomial, radial basis function (RBF), and sigmoid.

**Hard Margin:** The maximum-margin hyperplane or the hard margin hyperplane is a hyperplane that properly separates the data points of different categories without any misclassifications.

**Soft Margin:** When the data is not perfectly separable or contains outliers, SVM permits a soft margin technique. Each data point has a slack variable introduced by the soft-margin SVM formulation, which softens the strict margin requirement and permits certain misclassifications or violations. It discovers a compromise between increasing the margin and reducing violations.

**C:** Margin maximization and misclassification fines are balanced by the regularization parameter $C$ in SVM. The penalty for going over the margin or classifying data items is decided by it. A stricter penalty is imposed with a greater value of $C$, which results in a smaller margin and perhaps fewer misclassifications.

### 2.1.4 Why Are SVMs Used in Machine Learning

SVMs are used in applications like handwriting recognition, intrusion detection, face detection, email classification, gene classification, and in web pages. This is one of the reasons we use SVMs in machine learning. It can handle both classification and regression on linear and non-linear data.

Another reason we use SVMs is because they can find complex relationships between your data without you needing to do a lot of transformations on your own. It's a great option when you are working with smaller datasets that have tens to hundreds of thousands of features. They typically find more accurate results when compared to other algorithms because of their ability to handle small, complex datasets.

### 2.1.5 Hands On

#### 2.1.5.1 Linear SVM Example with Dummy Data

We'll go through the process of training a model with it using the Python Scikit-learn library. We'll do an example with a linear SVM.

#### 2.1.5.2 Non-Linear SVM Example with Dummy Data

We'll go through the process of training a model with it using the Python Scikit-learn library. This time, we'll do an example with a non-linear SVM.

#### 2.1.5.3 Predict If Cancer Is Benign or Malignant

We want to predict if cancer is benign or malignant. Using historical data about patients diagnosed with cancer enables doctors to differentiate malignant cases and benign ones are given independent attributes.

We are using here the Breast Cancer Wisconsin (Diagnostic) dataset. The features have been computed from a digitized image of a fine needle aspirate (FNA) of a breast mass. They

describe characteristics of the cell nuclei present in the image. This dataset has been reported in W. N. Street, W. H. Wolberg, and O. L. Mangasarian, "Nuclear Feature Extraction for Breast Tumor Diagnosis," in *Electronic Imaging*, 1993.

## 2.2 Gradient Boosting

Gradient boosting is a popular boosting algorithm in machine learning used for classification and regression tasks. Boosting is one kind of ensemble learning method which trains the model sequentially and each new model tries to correct the previous model.

Gradient boosting is a powerful boosting algorithm that combines several weak learners into strong learners, in which each new model is trained to minimize the loss function such as mean squared error or cross-entropy of the previous model using gradient descent. In each iteration, the algorithm computes the gradient of the loss function with respect to the predictions of the current ensemble and then trains a new weak model to minimize this gradient. The predictions of the new model are then added to the ensemble, and the process is repeated until a stopping criterion is met.

### 2.2.1 What is Gradient Boosting in General?

Boosting is a powerful ensemble technique in machine learning. Unlike traditional models that learn from the data independently, boosting combines the predictions of multiple weak learners to create a single, more accurate strong learner.

A weak learner is a machine learning model that is slightly better than a random guessing model. For example, let's say we are classifying mushrooms into edible and inedible. If a random guessing model is 40% accurate, a weak learner would be just above that: 50-60%.

Boosting combines dozens or hundreds of these weak learners to build a strong learner with the potential for over 95% accuracy on the same problem.

### 2.2.2 Gradient Boosting Process for Regression

The ensemble consists of $M$ trees. Tree 1 is trained using the feature matrix $X$ and the labels $y$. The predictions labeled $\hat{y_1}$ are used to determine the training set residual errors $r_1$. Tree 2 is then trained using the feature matrix $X$ and the residual errors $r_1$ of Tree 1 as labels. The predicted results $\hat{r_1}$ are then used to determine the residual $r_2$. The process is repeated until all the $M$ trees forming the ensemble are trained.

There is an important parameter used in this technique known as Shrinkage. Shrinkage refers to the fact that the prediction of each tree in the ensemble is shrunk after it is multiplied by the learning rate ($\nu$) which ranges between 0 to 1. There is a trade-off between $\nu$ and the number of estimators, decreasing learning rate needs to be compensated with increasing estimators in order to reach certain model performance. Since all trees are trained now, predictions can be made.

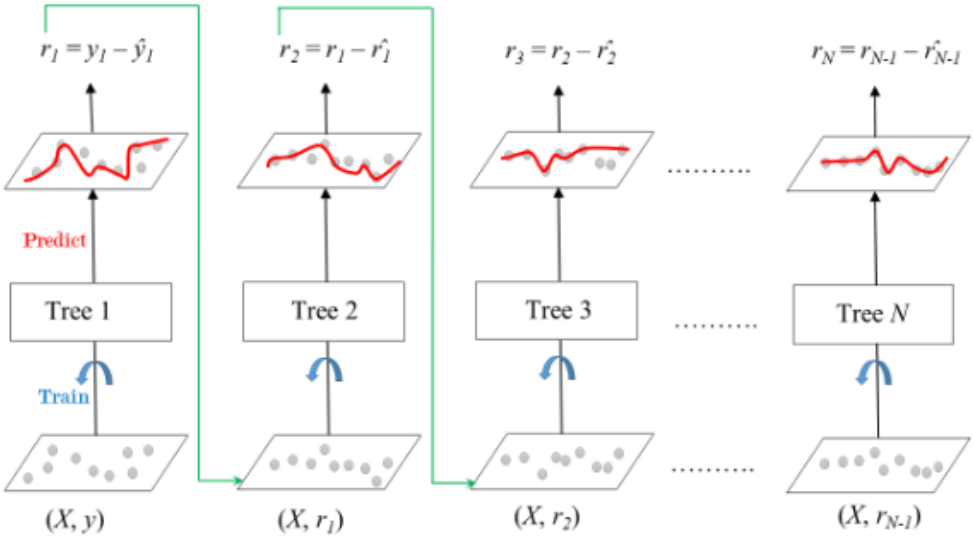



Figure 2.10: Gradient boosted trees for regression.

Each tree predicts a label and the final prediction is given by the formula,

$$y(\text{pred}) = y_1 + (\nu \times r_1) + (\nu \times r_2) + \cdots + (\nu \times r_N).$$

### 2.2.3 Gradient Boosting Process with an Example[1]

We are building gradient boosting regression trees step by step using a sample which has a nonlinear relationship between $x$ and $y$ to intuitively understand how it works. The sample is shown in Figure 2.11.

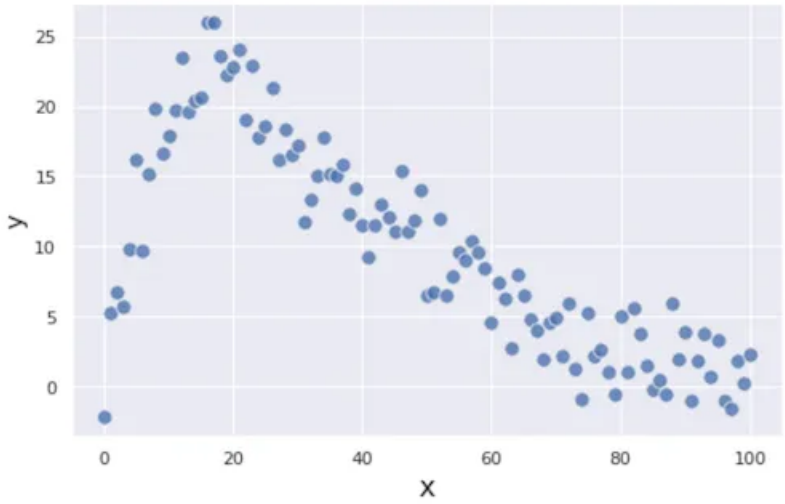



Figure 2.11: Sample for demonstration of the gradient boosting process.

The first step is making a very naive prediction on the target $y$. We make the initial prediction $F_0$ as an overall average of $y$.

[1] Adapted from T. Masui, *All You Need to Know about Gradient Boosting Algorithm*, Towards Data Science, Jan. 2022.

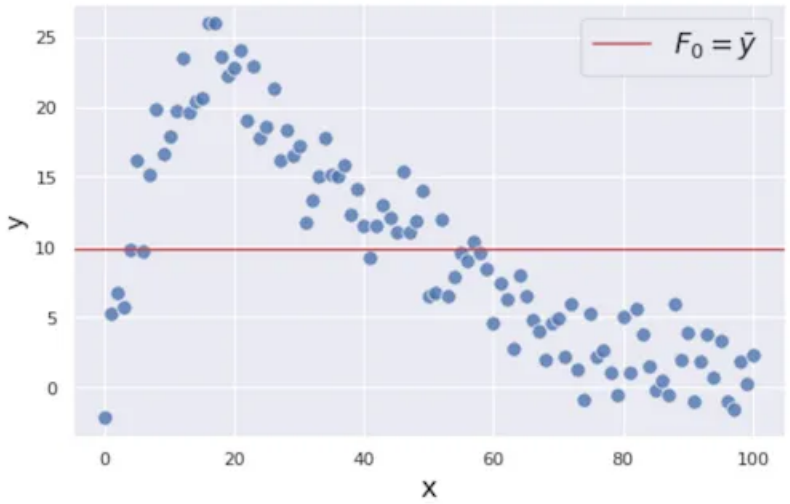



Figure 2.12: Initial prediction: $F_0 = \text{mean}(y)$.

This is shown in Figure 2.12. It may appear that using the mean for the prediction is silly. But the prediction will improve as we add more weak models to it.

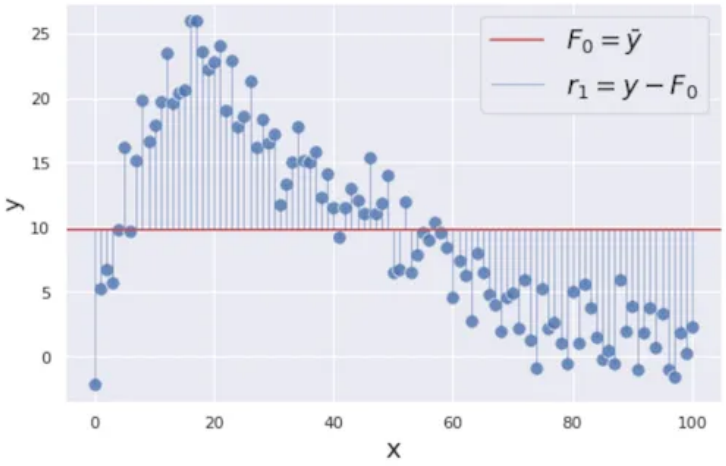



Figure 2.13: The residuals $r_1$.

To improve our prediction, we will focus on the residuals (i.e. prediction errors) from the first step because that is what we want to minimize to get a better prediction. The residuals $r_1$ are shown as the vertical blue lines in Figure 2.13.

To minimize these residuals, we are building a regression tree model with $x$ as its feature and the residuals $r_1 = y - \text{mean}(y)$ as its target. The reasoning behind that is if we can find some patterns between $x$ and $r_1$ by building the additional weak model, we can reduce the residuals by utilizing it.

To simplify the demonstration, we are building very simple trees each of that only has one split and two terminal nodes which is called "stump". Actually, gradient boosting trees usually have deeper trees such as ones with 8 to 32 terminal nodes.

As shown in Figure 2.14, we are creating the first tree predicting the residuals with two

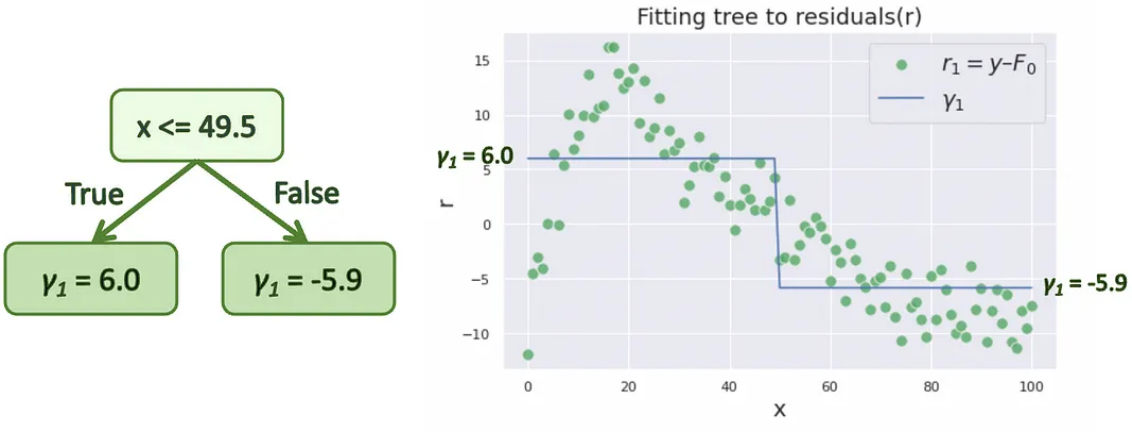



Figure 2.14: The first tree predicting the residuals.

different values $\gamma_1 = \{6.0, -5.9\}$. We are using $\gamma$ to denote the prediction.

This prediction $\gamma_1$ is added to our initial prediction $F_0$ to reduce the residuals. In fact, the gradient boosting algorithm does not simply add $\gamma$ to $F$ as it makes the model overfit to the training data. Instead, $\gamma$ is scaled down by the learning rate $\nu$ which ranges between 0 and 1, and then added to $F$.

$$F_1 = F_0 + \nu \cdot \gamma_1$$

In this example, we use a relatively big learning rate $\nu = 0.9$ to make the optimization process easier to understand, but it is usually supposed to be a much smaller value such as 0.1.

After the update, our combined prediction $F_1$ becomes:

$$F_1 = \begin{cases} F_0 + \nu \cdot 6.0, & \text{if } x \leq 49.5, \\ F_0 - \nu \cdot 5.9, & \text{otherwise.} \end{cases}$$

The update process is shown in Figure 2.15.

Now, the updated residuals $r_2$ looks like as shown in Figure 2.16.

In the next step, we are creating a regression tree again using the same $x$ as the feature and the updated residuals $r_2$ as its target. The created tree is shown in Figure 2.17.

Then, we are updating our previous combined prediction $F_1$ with the new tree prediction $\gamma_2$ as shown in Figure 2.18. The resulting predictions are $F_2$.

We iterate these steps until the model prediction stops improving. Figure 2.19 show the optimization process from 0 to 6 iterations.

We can see the combined prediction $F_m$ is getting more closer to our target $y$ as we add more trees into the combined model. This is how gradient boosting works to predict complex targets by combining multiple weak models.

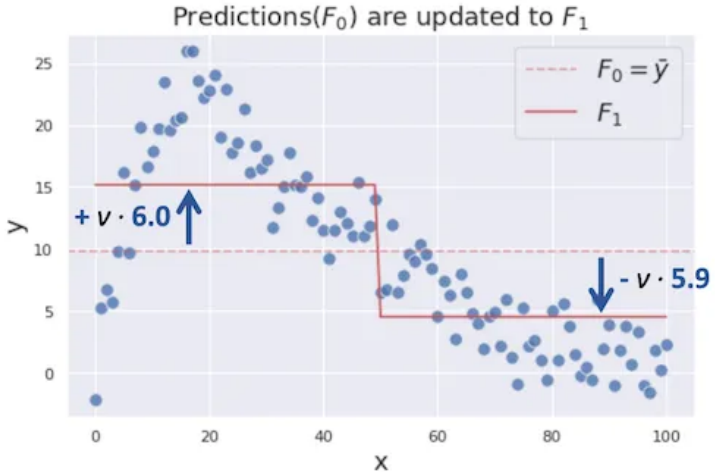



Figure 2.15: The predictions are updated to $F_1$.

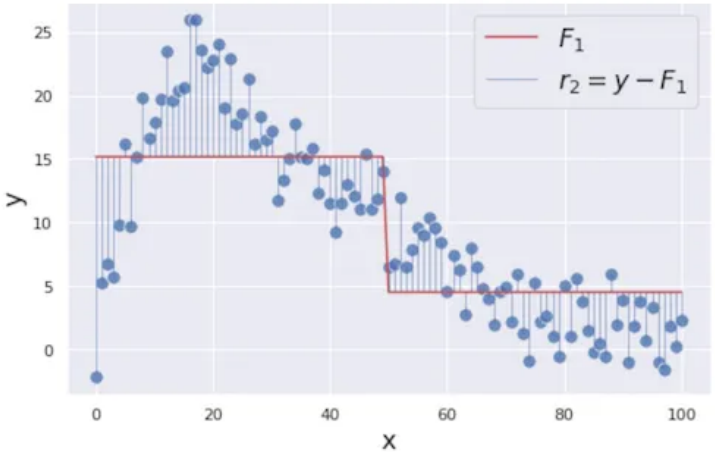



Figure 2.16: The updated residuals $r_2$.

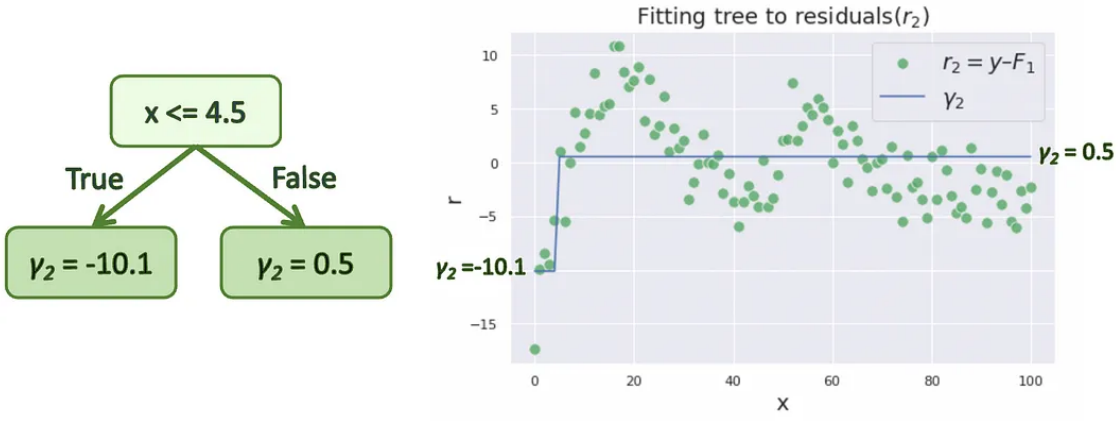



Figure 2.17: Fitting tree to the updated residuals $r_2$.

### 2.2.4 Gradient Boosting Process for Classification

In the gradient boosting process for classification the base algorithm remains the same. Gradient boosting for classification, much like its regression counterpart, is an ensemble

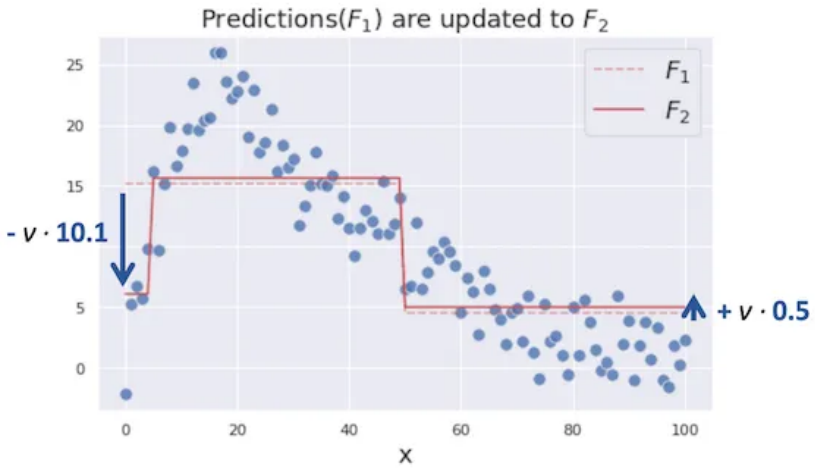



Figure 2.18: The predictions are updated to $F_2$.

learning technique that combines the predictions of multiple weak learners, typically decision trees, to create a robust and accurate model. The key difference lies in the nature of the target variable. In classification, the algorithm is designed to predict categorical outcomes. Gradient boosting for classification minimizes a loss function, often log loss for binary classification, by iteratively adding decision trees to the ensemble. Each tree corrects the errors of the preceding ones, adjusting the predicted probabilities for each class. This iterative process continues until a predefined number of trees is reached. Overall, while the underlying boosting principles remain similar, the specific loss functions and adjustments cater to the nuances of classifying categorical data rather than predicting continuous values.

### 2.2.5 Hands On

#### 2.2.5.1 Predict the Cut Quality of Diamonds Based on Their Price and Other Physical Measurements

We will predict the cut quality of diamonds based on their price and other physical measurements. This dataset is built into the `Seaborn` library.

#### 2.2.5.2 Classification for Digit Dataset

We will load the digit dataset and then instantiate gradient boosting classifier and fit the model. Our goal is to predict the test set and compute the accuracy score.

The dataset is a copy of the test set of the UCI ML hand-written digits datasets. The dataset has been reported in E. Alpaydin and C. Kaynak, *Optical Recognition of Handwritten Digits*, UCI Machine Learning Repository, 1998.

The authors used preprocessing programs made available by NIST (National Institute of Standards and Technology) to extract normalized bitmaps of handwritten digits from a preprinted form. From a total of 43 people, 30 contributed to the training set and different 13 to the test set. $32 \times 32$ bitmaps are divided into nonoverlapping blocks of $4 \times 4$ and the number of on pixels are counted in each block. This generates an input matrix of $8 \times 8$ where each

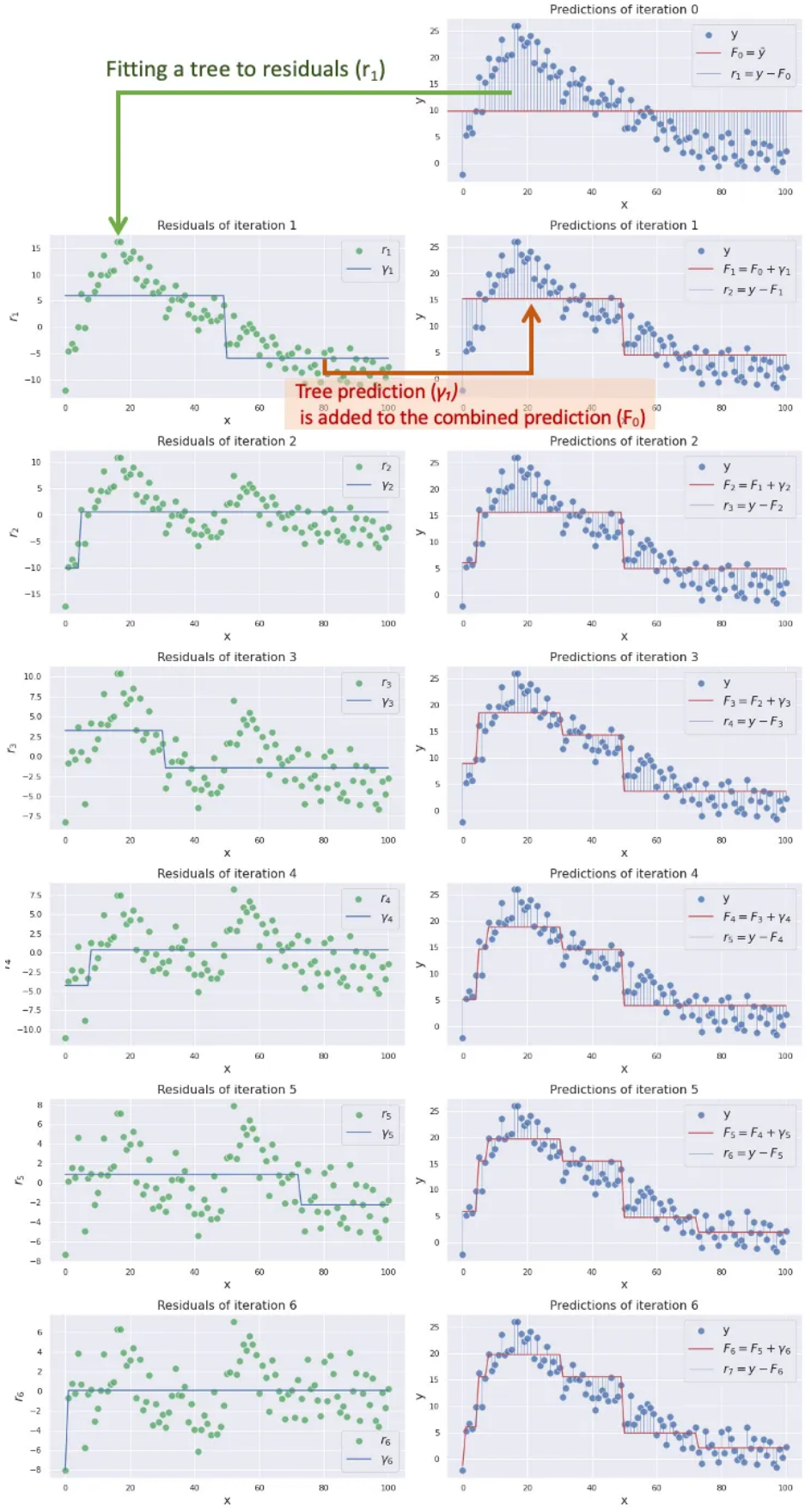



Figure 2.19: Optimization process from 0 to 6 iterations.

element is an integer in the range 0 . . . 16. This reduces dimensionality and gives invariance to small distortions.

#### 2.2.5.3 Regression for Diabetes Dataset

We want to load the diabetes dataset and then instantiate gradient boosting regressor and fit the model. Our goal is to predict on the test set and compute RMSE (Root Mean Square Error).

This dataset contains data from diabetic patients and contains certain features such as their BMI, age, blood pressure and glucose levels which are useful in predicting the diabetes disease progression in patients.

## 2.3 K-Nearest Neighbor (KNN) Algorithm

The K-Nearest Neighbors (KNN) algorithm is a supervised machine learning method employed to tackle classification and regression problems.

### 2.3.1 What is the K-Nearest Neighbors Algorithm?

KNN is one of the most basic yet essential classification algorithms in machine learning. It belongs to the supervised learning domain and finds intense application in pattern recognition, data mining, and intrusion detection.

It is widely disposable in real-life scenarios since it is non-parametric, meaning it does not make any underlying assumptions about the distribution of data (as opposed to other algorithms such as GMM, which assume a Gaussian distribution of the given data). We are given some prior data (also called training data), which classifies coordinates into groups identified by an attribute.

### 2.3.2 How It Works?

As an example, consider the data points containing two features in Figure 2.20.

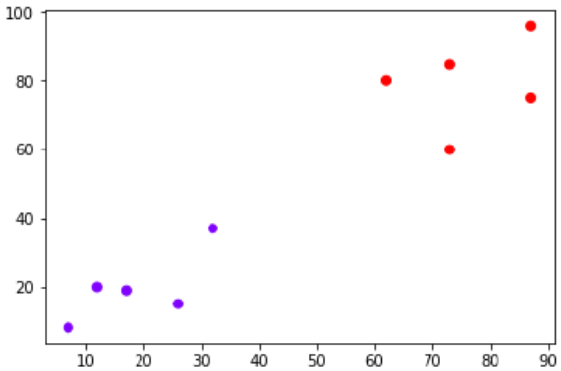



Figure 2.20: KNN algorithm working visualization.

We need to classify new data point with black dot (at point 60, 60) into blue or red class. We are assuming $K = 3$ i.e. it would find three nearest data points. It is shown in Figure 2.21.

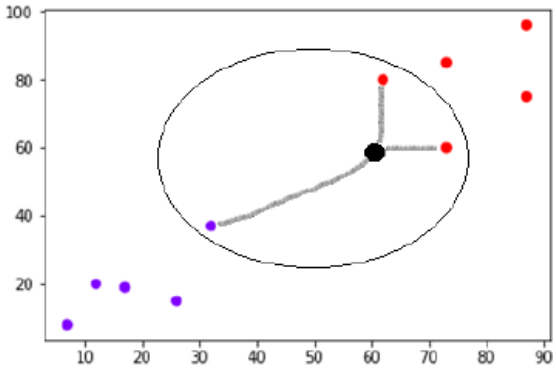



Figure 2.21: KNN algorithm classifying new data point.

We can see in the diagram that the three nearest neighbors of the data point with black dot. Among those three, two of them lies in Red class hence the black dot will also be assigned in Red class.

### 2.3.3 Hands On

#### 2.3.3.1 KNN on Dummy Data

We want to run the classifier on dummy data.

# Chapter 3

# Random Forest, Hidden Markov Model (HMM), and Gaussian Mixture Model (GMM)

## 3.1 Random Forest Classifier

The Random Forest or Random Decision Forest is a supervised machine learning algorithm used for classification, regression, and other tasks using decision trees. Random Forests are particularly well-suited for handling large and complex datasets, dealing with high-dimensional feature spaces, and providing insights into feature importance. This algorithm's ability to maintain high predictive accuracy while minimizing over-fitting makes it a popular choice across various domains, including finance, healthcare, and image analysis, among others.

Additionally, the Random Forest Classifier can handle both classification and regression tasks, and its ability to provide feature importance scores makes it a valuable tool for understanding the significance of different variables in the dataset.

### 3.1.1 How Random Forest Classification Works

The Random Forest Classifier creates a set of decision trees from a randomly selected subset of the training set. It contains a number of decision trees on various subsets of the given


Figure 3.1: Japanese Kanji characters for tree, wood, and forest.

dataset and takes the average to improve the predictive accuracy of that dataset. Instead of relying on one decision tree, the random forest takes the prediction from each tree and based on the majority votes of predictions, and it predicts the final output. It collects the votes from different decision trees to decide the final prediction.

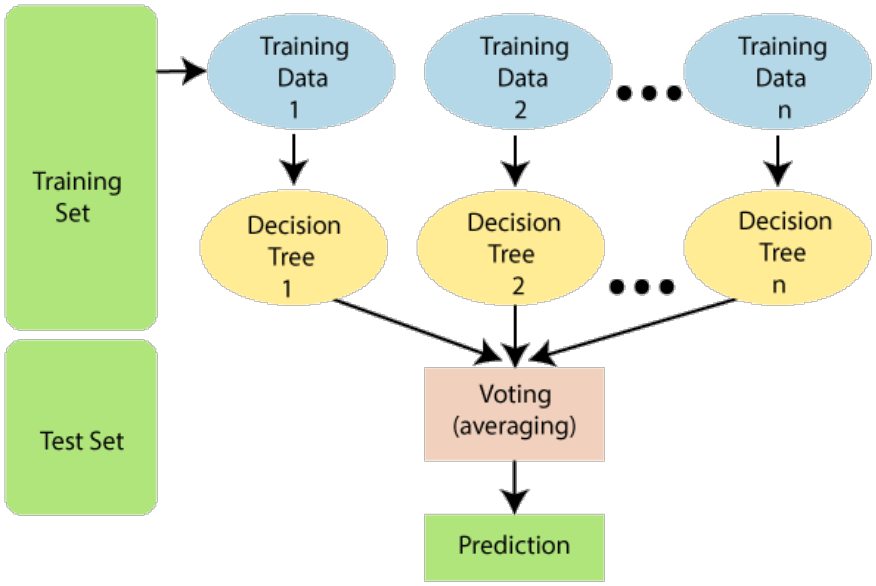



Figure 3.2: Working of the random forest algorithm.

The random forest algorithm employs a technique called bagging (bootstrap aggregating) to create these diverse subsets.

During the training phase, each tree is built by recursively partitioning the data based on the features. At each split, the algorithm selects the best feature from the random subset, optimizing for information gain or Gini impurity. The process continues until a predefined stopping criterion is met, such as reaching a maximum depth or having a minimum number of samples in each leaf node.

Once the random forest is trained, it can make predictions, using each tree "votes" for a class, and the class with the most votes becomes the predicted class for the input data.

The greater number of trees in the forest leads to higher accuracy and prevents the problem of overfitting.

Figure 3.2 explains the working of the Random Forest algorithm.

#### 3.1.1.1 Random Forest Classification — Example

Suppose there is a dataset that contains multiple fruit images. So, this dataset is given to the Random Forest Classifier. The dataset is divided into subsets and given to each decision tree. During the training phase, each decision tree produces a prediction result, and when a new data point occurs, then based on the majority of results, the Random Forest Classifier predicts the final decision as shown in Figure 3.3.

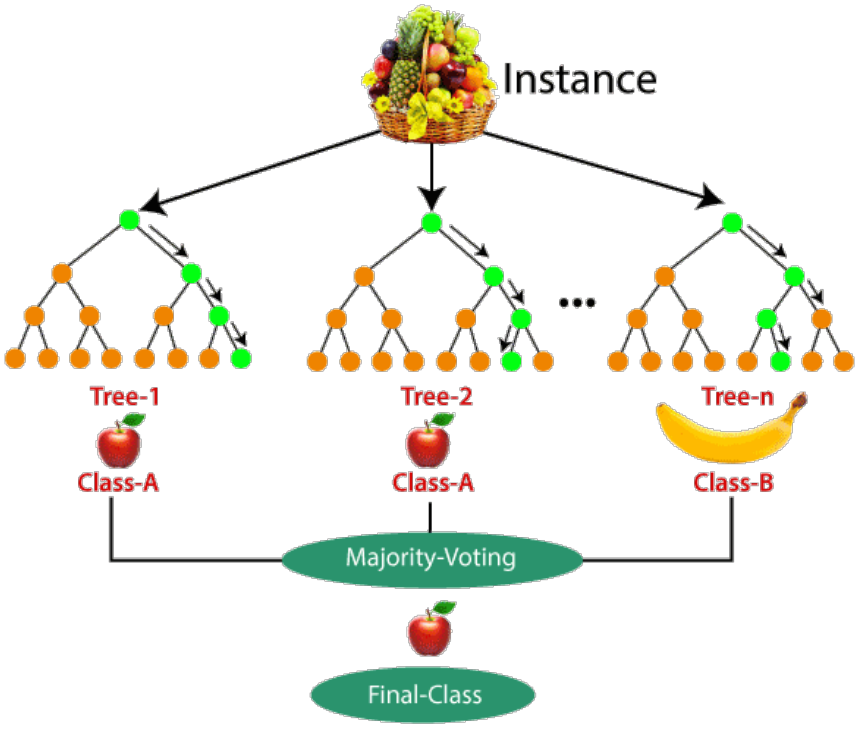



Figure 3.3: Example of random forest classification.

### 3.1.2 Feature Selection in Random Forests

Feature selection in random forests is inherently embedded in the construction of individual decision trees and the aggregation process.

During the training phase, each decision tree is built using a random subset of features, contributing to diversity among the trees. The process is, known as feature bagging (Bootstrap Aggregating). This helps prevent the dominance of any single feature and promotes a more robust model.

The algorithm evaluates various subsets of features at each split point, selecting the best feature for node splitting based on criteria such as information gain or Gini impurity. Consequently, random forests naturally incorporate a form of feature selection, ensuring that the ensemble benefits from a diverse set of features to enhance generalization and reduce overfitting.

#### 3.1.2.1 Bootstrap Aggregating

Bootstrap Aggregating or Bagging is a method in which we pick out random subsets from the original data and use them to train multiple different models. This is shown in Figure 3.4.

The term Bagging was derived from the term Bootstrap Aggregating itself — **B**ootstrap **Agg**regat**ing**.

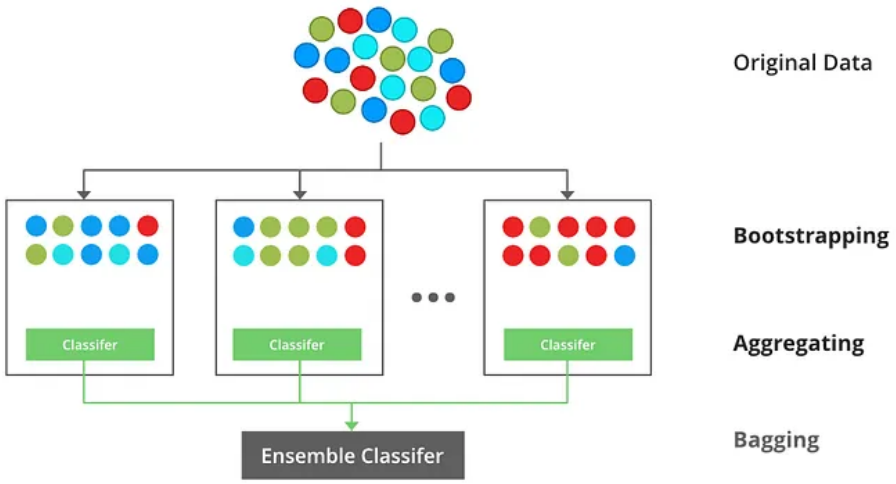



Figure 3.4: Bootstrap aggregation process.

### 3.1.3 Hands On

#### 3.1.3.1 Direct Marketing Campaigns by a Portuguese Banking Institution Using Phone Calls

The dataset we are going to use consists of direct marketing campaigns by a Portuguese banking institution using phone calls. The data is related with direct marketing campaigns. The campaigns aimed to sell subscriptions to a bank term deposit. The marketing campaigns were based on phone calls. Often, more than one contact to the same client was required in order to access if the product (bank term deposit) would be ("yes") or not ("no") subscribed. Our goal is to predict whether the person actually subscribed.

The dataset has been reported in the paper, S. Moro, P. Cortez, and P. Rita, "A Data-driven Approach to Predict the Success of Bank Telemarketing," *Decis. Support Syst.*, vol. 62, pp. 22–31, 2014. Another interesting point is, this paper earned a respectable count of citations of more than one thousand.

**Try Yourself** Write a similar predictor for the other data file present in the dataset folder.

#### 3.1.3.2 Predict the Species of Iris Flowers Using Random Forest Classifier

The Iris dataset is used to predict the species of flowers.

## 3.2 Hidden Markov Model (HMM)

A statistical model called a Hidden Markov Model (HMM) is used to describe systems with changing unobservable states over time. It is predicated on the idea that there is an underlying process with concealed states, each of which has a known result. Probabilities for switching between concealed states and emitting observable symbols are defined by the model.

Because of their superior ability to capture uncertainty and temporal dependencies, HMMs are used in a wide range of industries, including finance, bioinformatics, and speech recognition. HMMs are useful for modeling dynamic systems and forecasting future states based on sequences that have been seen because of their flexibility.

### 3.2.1 Hidden Markov Model in Machine Learning

The Hidden Markov Model (HMM) is a statistical model that is used to describe the probabilistic relationship between a sequence of observations and a sequence of hidden states. It is often used in situations where the underlying system or process that generates the observations is unknown or hidden, hence it has the name hidden Markov model.

It is used to predict future observations or classify sequences, based on the underlying hidden process that generates the data.

### 3.2.2 Introduction to Hidden Markov Model[1]

What is a Markov Model? Imagine the following scenario: you want to know whether the weather tomorrow will be sunny or rainy. Now, you might have a natural intuition based on your experience and historical observation of the weather. If for the past one week the weather has been sunny, then you have a 90% certainty that tomorrow will also be sunny. From your experience, the probability of tomorrow being sunny is 90% if at present the weather is also sunny. But if it is currently raining, then the probability of tomorrow being sunny is lower at 50% chance. This collection of scenario can be described as a Markov process.

#### 3.2.2.1 The Weather

In our simplified universe, the weather can only be in one of two possible states, "Sunny" or "Rainy". The catch (in the context of Markov chains) is that the probability of it being sunny or rainy tomorrow, depends on whether it is sunny or rainy today. We'll derive these probabilities from past data, and construct a transition matrix.

We have generated seven days of historical data on which to *train* our Markov chain.

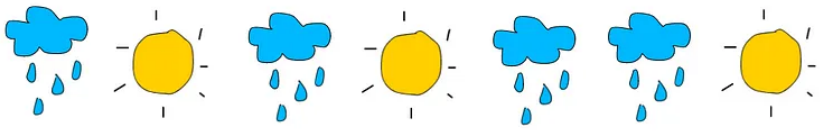


Figure 3.5: Collected data of weather.

Weather sequence:

`[Rainy, Sunny, Rainy, Sunny, Rainy, Rainy, Sunny]`.

Now calculate the percentage of instances it is sunny on days directly following rainy days.

[1] Adapted from J. Nathaniel, *Introduction to markov chain programming*, Towards Data Science, May 2021, GreekDataGuy, *Predicting the Weather with Markov Chains*, Towards Data Science, Oct. 2019, and J. Nathaniel, *Hands-on Introduction to Hidden Markov Model*, Geek Culture, Jul. 2021.

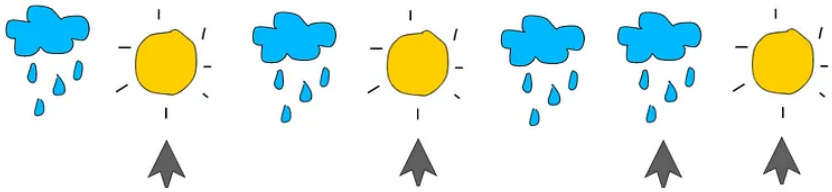


Figure 3.6: Days following rainy days.

Total number of days following rainy days = 4.

Number of sunny days immediately following rainy days = 3.

Percentage of sunny days directly following rainy days = $\frac{3}{4} = 75\%$.

Number of rainy days immediately following rainy days = 1.

Percentage of rainy days directly following rainy days = $\frac{1}{4} = 25\%$.

Now calculate the percentage of instances it is rainy on days directly following sunny days.

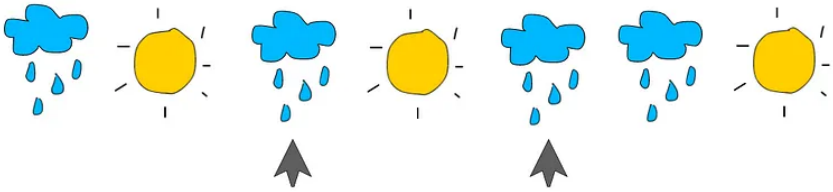


Figure 3.7: Days following sunny days.

Total number of days following sunny days = 2.

Number of rainy days immediately following sunny days = 2.

Percentage of rainy days directly following sunny days = $\frac{2}{2} = 100\%$.

Number of sunny days immediately following sunny days = 0.

Percentage of sunny days directly following sunny days = $\frac{0}{4} = 0\%$.

We also need an initial probability distribution over states. This is the probability that the Markov chain will start in a certain state.

Total number of days = 7.

Number of sunny days = 3.

Percentage of sunny days = $\frac{3}{7} = 42.86\%$.

Number of rainy days = 4.

Percentage of rainy days = $\frac{4}{7} = 57.14\%$.

**Stochastic**

Stochastic refers to a process or system that involves randomness and unpredictability. In various fields, including mathematics, statistics, and computer science, stochastic models or methods are employed to capture and analyze phenomena that exhibit random behavior. Unlike deterministic systems, stochastic systems introduce an element of chance, making outcomes uncertain and subject to probability distributions. Stochastic processes are commonly used to model dynamic systems, financial markets, and phenomena influenced by random variables, contributing valuable insights into the inherent variability and uncertainty present in real-world scenarios.

#### 3.2.2.2 Transition Matrix

We'll build our transition matrix with that information, populating the percentages from the information we've already derived.

| | | *Tomorrow* | |
|---|---|---|---|
| | | S | R |
| *Today* | S | 0.00 | 1.00 |
| | R | 0.75 | 0.25 |

#### 3.2.2.3 What is Markov Chain?

But what is a Markov Chain, formally? Markov Chain is a mathematical system that describes a collection of transitions from one state to the other according to certain stochastic or probabilistic rules.

Formally, a Markov Chain is a stochastic model describing a sequence of possible events in which the probability of each event depends only on the state attained in the previous event.

In other words, Markov Chains have a finite number of possible states. In each time period, it hops from one state to another (or the same state). The probabilities of hopping to specific state depend only on the probabilities associated with our current state.

Take for example our earlier scenario for predicting the next day's weather. If today's weather is sunny, then based on our (reliable) experience, the probability for the weather tomorrow transitioning to rainy is 100% and sunny 0%. On the other hand, if at present the weather is rainy, then the probability for tomorrow sunny is 75% and remaining rainy 25%.

We can also display the above information in a diagram as shown Figure 3.8.

These changes (or the lack thereof) between different states are called transitions while the variable of interest (ie. rainy or sunny) are called states.

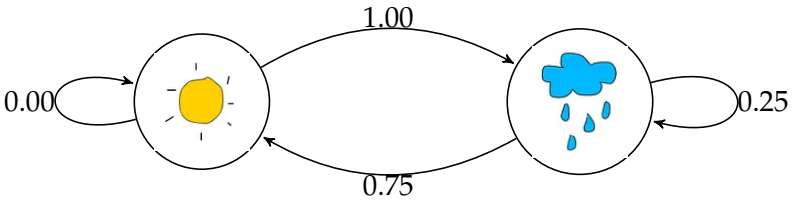



Figure 3.8: Markov Chain for the weather scenario.

For these transitions, however, to be qualified as Markov Chain, they must satisfy the Markov Property. The property states that the probability of transition is entirely dependent only on the current state, and not on the preceding set of sequences. This characteristic allows Markov Chain to be memory-less.

#### 3.2.2.4 Emission Probabilities

But now suppose, for some reason, you can't determine the current weather. Perhaps you are quarantined in an isolation room without any window to peek through. Nevertheless, you can still feel the humidity from your isolation room to either be dry or wet.

This is called Hidden Markov Model: the dry and wet states are the visible Markov, while the sunny and rainy states are the invisible Markov processes, and each hidden state generates a random humidity observations to us.

Also, suppose we observe the following conditional probabilities:

- Conditional probabilities of a dry or wet given a hidden sunny day.
- Conditional probabilities of a dry or wet given a hidden rainy day.

An emission matrix represents the probabilities of observing specific emissions (observable states) given the hidden states. For example, if we have hidden states $R$ (Rainy) and $S$ (Sunny), and observable states $D$ (Dry) and $W$ (Wet), the emission matrix might look like this:

$$\begin{bmatrix} P(D|R) & P(W|R) \\ P(D|S) & P(W|S) \end{bmatrix}$$

With assumed values the above matrix will look like this:

$$\begin{bmatrix} 0.2 & 0.8 \\ 0.9 & 0.1 \end{bmatrix}$$

#### 3.2.2.5 Initial Probability Distribution

We also need to know the probability of a certain weather condition on the very first day. For our scenario, we need know the probability of the first day being rainy or sunny.

An initial probability distribution over states, is the probability that the Markov chain will start in a certain state. The concept of initial probability pertains to the probabilities associated with the system's starting state. The initial probability distribution specifies the likelihood of

the system being in a particular hidden state at the beginning of the sequence. It forms a crucial component of the model's parameters, along with state transition probabilities and emission probabilities.

The initial probabilities are typically arranged in a vector, where each element corresponds to the probability of starting in a specific hidden state. These probabilities influence the model's predictions and its ability to capture the underlying dynamics of a sequence of observations. Properly defining the initial probabilities is essential for accurately modeling processes.

So, for the weather sequence we have considered, this is going to look like:

$$\begin{bmatrix} \pi(S) & \pi(R) \end{bmatrix}$$

With values the above matrix will look like this:

$$\begin{bmatrix} 0.43 & 0.57 \end{bmatrix}$$

#### 3.2.2.6 Uncovering State Sequence

In an HMM, observable events provide crucial information that can be utilized to uncover the hidden state sequence. The model assumes that there exists an underlying sequence of hidden states, and each hidden state generates observable events with certain probabilities.

The probabilistic rules of HMM facilitate the inference of the hidden state sequence based on the observed events. By applying the forward algorithm, backward algorithm, and the Viterbi algorithm, one can compute the likelihood of a particular hidden state sequence given the observed events. These algorithms leverage the probabilities of transitioning between hidden states and emitting observable events, allowing the model to assess the likelihood of different state sequences. The process involves iteratively updating the probabilities and making predictions, ultimately revealing the most probable hidden state sequence that corresponds to the observed events.

This inherent probabilistic nature of HMM enables it to handle situations where the true hidden states are not directly observable but can be inferred with reasonable certainty based on the observed events and the underlying statistical relationships modeled by the HMM.

Now, let's assume we observe the following sequence of events: $D, W, D, D$. We want to uncover the most likely hidden state sequence corresponding to these observations. Using the algorithms mentioned earlier, we can calculate the most probable hidden state sequence.

### 3.2.3 Hidden Markov Model in Machine Learning

We use the knowledge gained above to put HMM in the perspective of ML.

The hidden Markov Model (HMM) is a statistical model that is used to describe the probabilistic relationship between a sequence of observations and a sequence of hidden states. It is used to predict future observations or classify sequences, based on the underlying hidden process that generates the data.

An HMM consists of two types of variables: hidden states and observations.

- The hidden states are the underlying variables that generate the observed data, but they are not directly observable.
- The observations are the variables that are measured and observed.

The relationship between the hidden states and the observations is modeled using a probability distribution. HMM is the relationship between the hidden states and the observations using two sets of probabilities: the transition probabilities and the emission probabilities.

- The transition probabilities describe the probability of transitioning from one hidden state to another.
- The emission probabilities describe the probability of observing an output given a hidden state.

### 3.2.4 Hands On

#### 3.2.4.1 Predicting the Weather

Given the historical data on weather conditions, the task is to predict the weather for the next day based on the current day's weather.

#### 3.2.4.2 Speech Recognition

Given a dataset of audio recordings, the task is to recognize the words spoken in the recordings.

The state space is defined as states, which is a list of possible states representing silence or the presence of different words. The observation space is defined as observations The initial state distribution is an array of probabilities representing the probability of each state being the initial state. The state transition probabilities is a matrix representing the probability of transitioning from one state to another. The observation likelihoods is a matrix representing the probability of emitting an observation for each state.

The model is used to predict the most likely hidden states, given the observations.

## 3.3 Gaussian Mixture Model (GMM)

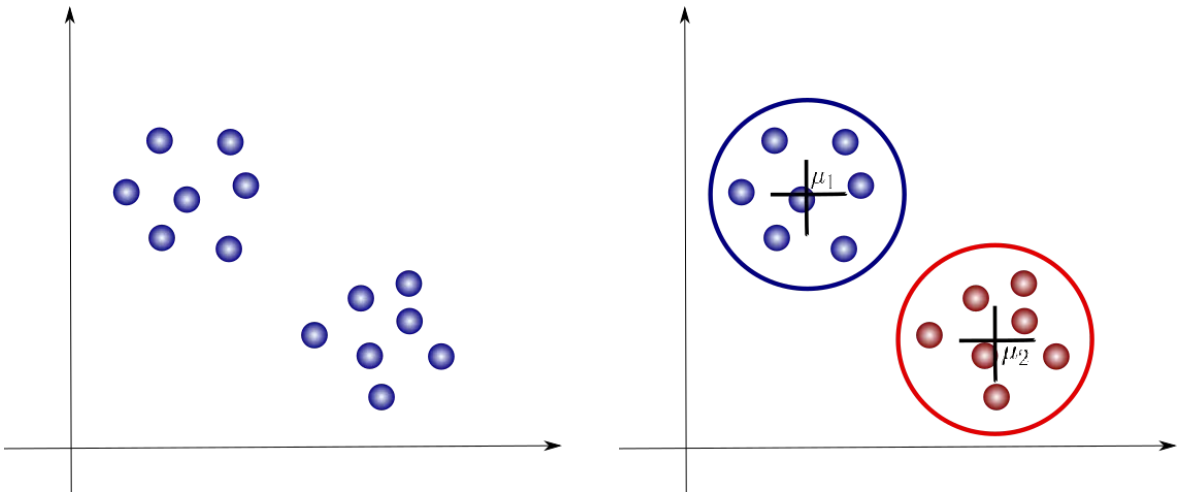
Gaussian Mixture Model (GMM) represent a versatile and powerful approach in statistical modeling, particularly in the realm of unsupervised machine learning. This model is adept at capturing complex patterns in data by assuming that the observed distribution is a mixture of several Gaussian distributions.

GMMs have found widespread applications in clustering, density estimation, and pattern recognition, offering a flexible framework to analyze datasets with diverse structures and underlying distributions. Their ability to handle uncertainty and model intricate relationships between variables makes GMMs a valuable tool in various fields, ranging from computer vision to finance.

### 3.3.1 Gaussian Mixture Model Explained[2]

#### 3.3.1.1 Clustering

Clustering is an unsupervised learning problem where we intend to find clusters of points in our dataset that share some common characteristics. Let's suppose we have a dataset that looks like shown in Figure 3.9(a).




(a) Data before clustering.

(b) Data after clustering.

Figure 3.9: Clustering of data.

Our job is to find sets of points that appear close together. In this case, we can clearly identify two clusters of points which we will color blue and red, respectively. See Figure 3.9(b).

We have introduced $\mu_1$ and $\mu_2$ as the centroids of each cluster. These are parameters that identify each of these.

#### 3.3.1.2 k-means Clustering

When we talk about Gaussian Mixture Model (GMM), it's essential to know how the $k$-means algorithm works. Because GMM is quite similar to the $k$-means, more likely it's a probabilistic version of $k$-means. This probabilistic feature allows GMM to be applied to many complex problems that $k$-means can't fit into.

$k$-means clustering is a popular clustering algorithm. This algorithm follows an iterative approach to update the parameters of each cluster. More specifically, it computes the means (or centroids) of each cluster, and then calculate their distance to each of the data points. The latter are then labeled as part of the cluster that is identified by their closest centroid. This process is repeated until some convergence criterion is met, for example when we see no further changes in the cluster assignments.

[2]Adapted from O. C. Carrasco, *Gaussian Mixture Models Explained*, Towards Data Science, Jun. 2019 and R. Ravihara, *Gaussian Mixture Model Clearly Explained*, Towards Data Science, Jan. 2023

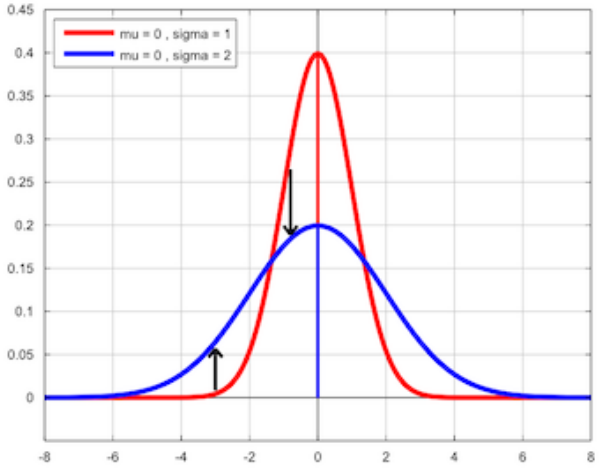



Figure 3.10: Variance ($\sigma$) of data.

One important characteristic of $K$-means clustering is that it is a hard clustering method. This means that it will associate each point to one and only one cluster. A limitation to this approach is that there is no uncertainty measure or probability that tells us how much a data point is associated with a specific cluster. $k$-means clustering does not account for variance (width of the bell shape curve).

In two dimensions, variance/covariance determines the shape of the distribution. See Figure 3.10.

#### 3.3.1.3 Variance and Covariance

Variance and covariance are related statistical measures that describe the spread or dispersion of random variables. Here's a brief explanation of each and their relationship.

**Variance:** Variance is a statistical measure that quantifies the degree of dispersion or spread of a set of values in a dataset, indicating the extent to which individual data points deviate from the mean.

$$\text{Variance(X)} = \frac{1}{n}\sum_{i=1}^{n}(X_i - \bar{X})^2$$

where $X_i$ is each data point, $\bar{X}$ is the mean, and $n$ is the number of data points.

**Covariance:** Covariance is a statistical measure that describes the degree to which two random variables change together, indicating whether an increase or decrease in one variable is associated with a similar change in another.

$$\text{Cov(X, Y)} = \frac{1}{n}\sum_{i=1}^{n}(X_i - \bar{X}) \cdot (Y_i - \bar{Y})$$

where $X_i$ and $Y_i$ are data points, $\bar{X}$ and $\bar{Y}$ are the means of $X$ and $Y$, and $n$ is the number of data points.

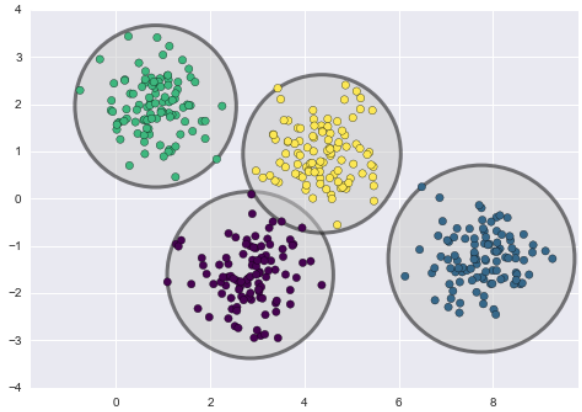



Figure 3.11: *k*-means model places a circle at the center of each cluster.

**Relationship:**

$$\text{Cov}(X, X) = \text{Variance}(X)$$

The covariance between two variables is a scaled version of their variance. Specifically, the covariance between two identical variables is equal to the variance of that variable.

#### 3.3.1.4 Problem with k-means Clustering

The *k*-means model, as shown in Figure 3.11, places a circle (or, in higher dimensions, a hyper-sphere) at the center of each cluster, with a radius defined by the most distant point in the cluster.

This works fine when data is circular. However, when data takes on different shape, we end up with something like the situation in Figure 3.12.

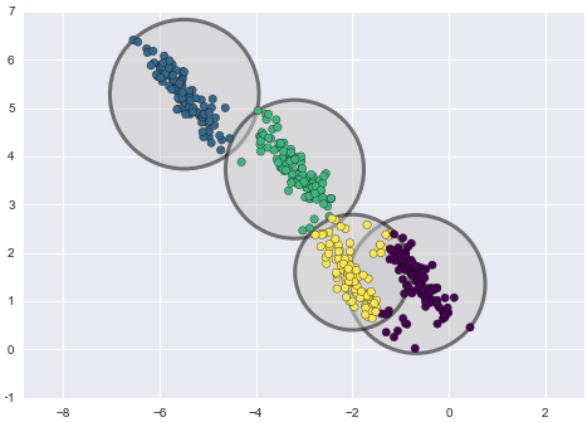



Figure 3.12: Scenario when data shapes are not circular.

In summary, *k*-means clustering have the following limitations:

i. It it assumed that the clusters are spherical. This is not valid in most real-world scenarios.

ii. It's a hard clustering method, meaning each data point is assigned to a single cluster.

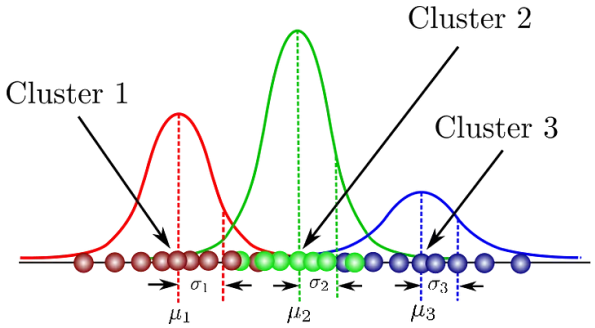



Figure 3.13: Parameters in Gaussian.

### 3.3.2 Switching to GMM

Due to the limitations identified above, we should find alternatives for $k$-means clustering when working on our machine learning projects. We explore one of the best alternatives for $k$-means clustering. We think of using a soft clustering instead of a hard one. This is exactly what Gaussian Mixture Models, or simply GMMs, attempt to do.

### 3.3.3 Definitions in GMM

A Gaussian mixture is a function that comprise several Gaussians, each identified by $k \in \{1, \ldots, K\}$, where $K$ is the number of clusters of our dataset. Each Gaussian in the mixture encompasses the following parameters:

- A mean $\mu$ that defines its center.
- A covariance $\Sigma$ that defines its width. This would be equivalent to the dimensions of an ellipsoid in a multivariate scenario.
- A mixing probability $\pi$ that defines how big or small the Gaussian function will be.

We can illustrate these parameters graphically as shown in Figure 3.13.

The mixing coefficients are themselves probabilities and must meet this condition:

$$\sum_{k=1}^{K} \pi_k = 1.$$

Here, we can see that there are three Gaussian functions, hence $K = 3$. Each Gaussian explains the data contained in each of the three clusters available.

Every distribution is multiplied by a weight $\pi$ ($\pi_1 + \pi_2 + \pi_3 = 1.0$) to account for the fact that we do not have an equal number of samples from each category. In other words, if we are dealing with a group of people, we might have included only 1000 people from the red cluster class, but have included 100,000 people from the green cluster class. We could have included 400 people from the blue cluster class.

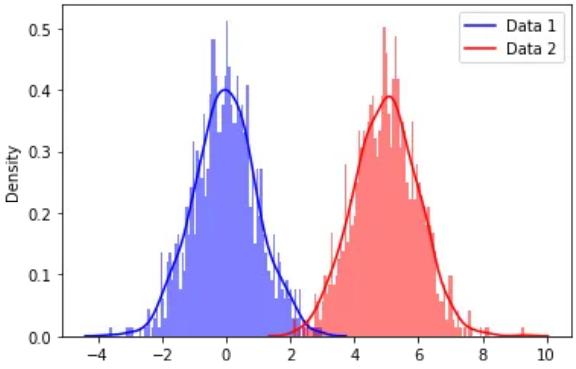



Figure 3.14: Mixtures of Gaussians.

**Maximum Likelihood Estimation (MLE)**

Maximum Likelihood Estimation (MLE) is a statistical method used to estimate the parameters of a model by maximizing the likelihood function. The likelihood function represents the probability of observing the given data under a particular set of model parameters. The essence of MLE is to find the parameter values that make the observed data most probable. In other words, it seeks to determine the parameter values that maximize the likelihood of the observed data occurring. Mathematically, this involves finding the values that maximize the product of the probability density function or likelihood function for each data point. MLE provides a systematic and principled approach for estimating parameters and is widely employed in various fields, including machine learning, econometrics, and biology. The resulting parameter estimates are considered to be the values that make the observed data most plausible given the assumed statistical model.

### 3.3.4 How Does the Gaussian Mixture Model (GMM) Algorithm Work?

The dataset in GMM consists of multiple Gaussians, in other words, a mixture of the Gaussian as shown in Figure 3.14.

Let's say we have some data as shown in Figure 3.15.

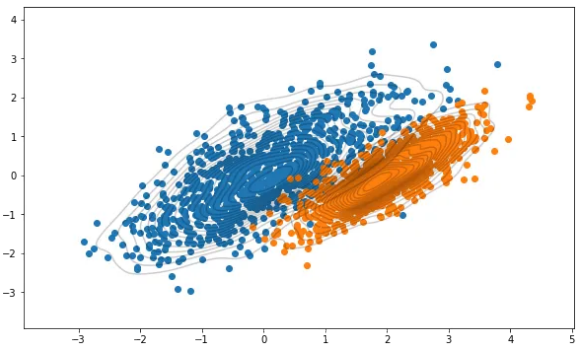


Figure 3.15: Data generated using random Gaussian distribution.

GMMs determine optimal values for their parameters (mean, covariance, and mixing coefficients) through a process called Maximum Likelihood Estimation (MLE). The key idea is to find parameter values that make the observed data the most probable under the assumed model. The optimization process involves adjusting these parameters to maximize the likelihood of observing the given data.

To achieve this, GMMs employ the Expectation-Maximization (EM) algorithm. In the E-step, the algorithm estimates the probability that each data point belongs to each Gaussian component, utilizing the current parameter values. In the M-step, the algorithm then updates the parameters based on these probabilities, seeking values that enhance the overall likelihood of the data. This iterative process continues until convergence, at which point the model has found parameter values that best capture the underlying patterns and structure of the observed data, effectively identifying the optimal configuration for the Gaussian components.

### 3.3.5 Expectation Maximization

#### 3.3.5.1 Expectation

The first step, known as the expectation step or E-step, consists of calculating the expectation of the component assignments $C_k$ for each data point $x_i \in X$ given the model parameters $\pi_k$, $\mu_k$, and $\sigma_k$.

#### 3.3.5.2 Maximization

The second step is known as the maximization step or M-step, which consists of maximizing the expectations calculated in the E-step with respect to the model parameters. This step involves updating the values $\pi_k$, $\mu_k$, and $\sigma_k$.

The entire iterative process repeats until the algorithm converges, providing a maximum likelihood estimate. Intuitively, the algorithm works because knowing the component assignment $C_k$ for each $x_i$ makes solving for $\pi_k$, $\mu_k$, and $\sigma_k$ easy, while knowing $\pi_k$, $\mu_k$, and $\sigma_k$ makes inferring $p(C_k|x_i)$ easy. The expectation step corresponds to the latter case, while the maximization step corresponds to the former. By alternating between which values are assumed fixed or known, maximum likelihood estimates of the non-fixed values can be calculated efficiently.

### 3.3.6 Hands On

#### 3.3.6.1 Gaussian Mixture Model for the Iris Dataset

In this example, the Gaussian Mixture Model (GMM) is applied to the Iris dataset. The clustering process assigns a label to each observation based on the GMM. Subsequently, the data points are separated into three clusters.

# Part II

# Foundation Models

# Chapter 5

# Linear Regressions

Linear regression is a type of supervised machine learning algorithm that computes the linear relationship between a dependent variable and one or more independent features. When the number of the independent feature is one, then it is known as Univariate Linear Regression or Simple Linear Regression, and in the case of more than one feature, it is known as Multivariate Linear Regression or Multiple Linear Regression.

The interpretability of linear regression is a notable strength. The model's equation provides clear coefficients that elucidate the impact of each independent variable on the dependent variable, facilitating a deeper understanding of the underlying dynamics. Its simplicity is a virtue, as linear regression is transparent, easy to implement, and serves as a foundational concept for more complex algorithms.

Linear regression is not merely a predictive tool. It forms the basis for various advanced models. Techniques like regularization and support vector machines draw inspiration from linear regression, expanding its utility. Additionally, linear regression is a cornerstone in assumption testing, enabling researchers to validate key assumptions about the data.

## 5.1 Types of Linear Regression

There are two main types of linear regression.

### 5.1.1 Simple Linear Regression

This is the simplest form of linear regression, and it involves only one independent variable and one dependent variable. The equation for simple linear regression is:

$$Y = b + wX$$

where:

- $Y$ is the dependent variable,
- $X$ is the independent variable,
- $b$ is the intercept or bias, and
- $w$ is the slope or weight.

**The Goal**

The goal of the algorithm is to find the *Best Fit Line* equation that can predict the values based on the independent variables.
In regression set of records are present with $X$ and $Y$ values and these values are used to learn a function so if you want to predict $Y$ from an unknown $X$ this learned function can be used. In regression we have to find the value of $Y$. So, a function is required that predicts continuous $Y$ in the case of regression given $X$ as independent features.

### 5.1.2 Multiple Linear Regression

This involves more than one independent variable and one dependent variable. The equation for multiple linear regression is:

$$Y = b + w_1X_1 + w_2X_2 + \ldots + w_nX_n$$

where:

- $Y$ is the dependent variable,
- $X_1, X_2, \ldots, X_n$ are the independent variables,
- $b$ is the intercept or bias, and
- $w_1, w_2, \ldots, w_n$ are the slopes or weights.

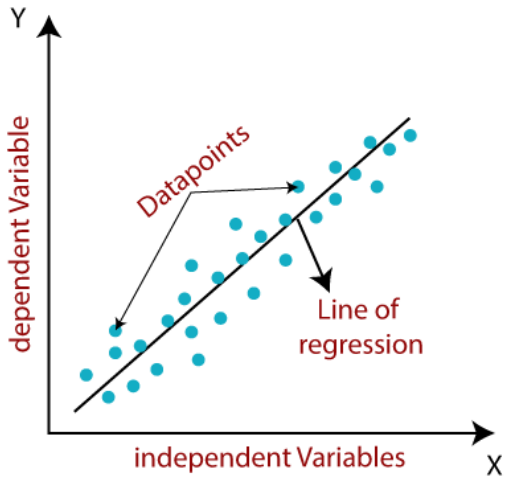



Figure 5.1: The linear regression model provides a sloped straight line representing the relationship between the variables.

## 5.2 What is the Best Fit Line?

Our primary objective while using linear regression is to locate the best fit line, which implies that the error between the predicted and actual values should be kept to a minimum. There will be the least error in the best fit line.

The best fit line equation provides a straight line that represents the relationship between the dependent and independent variables. The slope of the line indicates how much the dependent variable changes for a unit change in the independent variable(s).


Figure 5.2: Best fit line in linear regression.

Here $Y$ is called a dependent or target variable and $X$ is called an independent variable also known as the predictor of $Y$. There are many types of functions or modules that can be used for regression. A linear function is the simplest type of function. Here, $X$ may be a single feature or multiple features representing the problem.

In Figure 5.2, $X$ (input) is the average income per person and $Y$ (output) is the average happiness score. The regression line is the best fit line for our model.

We utilize the loss function to compute the best values in order to get the best fit line since different values for weights or the coefficient of lines result in different regression lines.

## 5.3 Loss Function in Linear Regression[1]

### 5.3.1 Defining the Loss Function

When working with linear regression, the objective is to find the best line that fits the training data. The loss function measures the difference between the predicted values of the model and the actual target values. By minimizing this loss function, optimal values for the model's parameters can be determined, improving its performance.

### 5.3.2 Intuition Behind the Loss Function

The loss function, denoted as $J(w, b)$, evaluates how well the model's predictions align with the true target values. It calculates the squared error between the predicted value $\hat{y} = f(w, b, x)$ and the actual target value $y$. The loss function is defined as the sum of squared errors across all training examples, providing an indication of how well the model fits the data.

In linear regression, the Mean Squared Error (MSE) loss function is employed, calculating the average of squared errors between the predicted values $\hat{y}_i$ and the actual values $y_i$. The goal is to determine optimal values for the intercept $b$ and the coefficient of the input feature $w$, yielding the best fit line for the given data points. The linear equation expressing this relationship is $\hat{y}_i = b + wx_i$.

The MSE (loss) function can be calculated as:

$$\text{Loss function}(J) = \frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2. \tag{5.1}$$

Utilizing the MSE function, the iterative process of gradient descent (c.f. Section 5.4) is applied to update the values of $b$ and $w$. This ensures that the MSE value converges to the global minima, indicating the most accurate fit of the linear regression line to the dataset.

This process involves continuously adjusting the parameters $b$ and $w$ based on the gradients calculated from the MSE. The final result is a linear regression line that minimizes the overall squared differences between the predicted and actual values, providing an optimal representation of the underlying relationship in the data.

## 5.4 Gradient Descent for Linear Regression

An important concept in linear regression is Gradient Descent. It is a popular optimization approach employed in training machine learning models by reducing errors between actual and predicted outcomes. Linear regression models are often trained using the gradient descent optimization algorithm. Optimization in machine learning is the task of minimizing the loss function parameterized by the model's parameters. The primary goal of gradient descent is to minimize the convex function by parameter iteration.

[1] Part of this section has been adapted from N. Yen, "Understanding the Cost Function in Linear Regression for Machine Learning Beginners," in *Self-taught Machine Learning Collection*, May 2023.

Finding the coefficients of a linear equation that best fits the training data is the objective of linear regression. The coefficients are adjusted iteratively by moving in the direction of the negative gradient (direction of decrease of the objective function) of the Mean Squared Error (MSE) with respect to the coefficients. In each iteration, the respective intercept and coefficient of $X$ are updated according to the learning rate $\alpha$. This iterative approach modifies the model's parameters to minimize the MSE on the training dataset, ultimately aiming to reduce the loss function (minimize RMSE) and find the best fit line.

Gradient descent, at its core, is an iterative optimization algorithm employed to find the minimum of a function, and in this context, that function is the Loss Function. The gradient, essentially a derivative, guides the algorithm by indicating the effects on outputs for slight variations in inputs. See Figure 5.3.

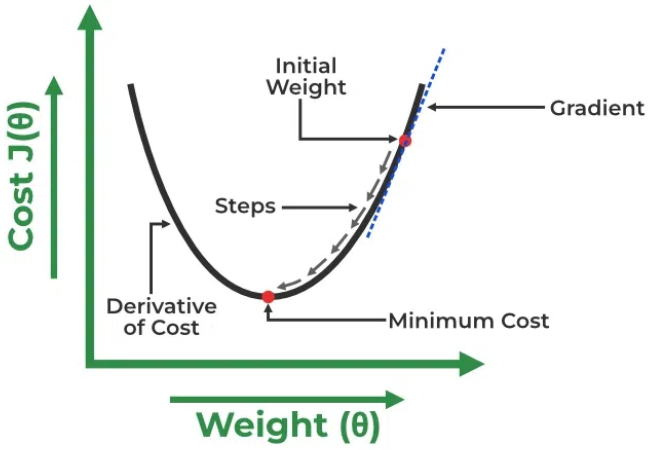



Figure 5.3: Gradient descent for linear regression.

Imagine a person in a valley (Figure 5.4) without a sense of direction, aiming to reach the valley's bottom. The person descends the slope, taking larger steps on steeper terrain and smaller steps on gentler slopes. This process is analogous to gradient descent, where the goal is to iteratively update $b$ and $w$ values, starting from random values and reaching the minimum cost.

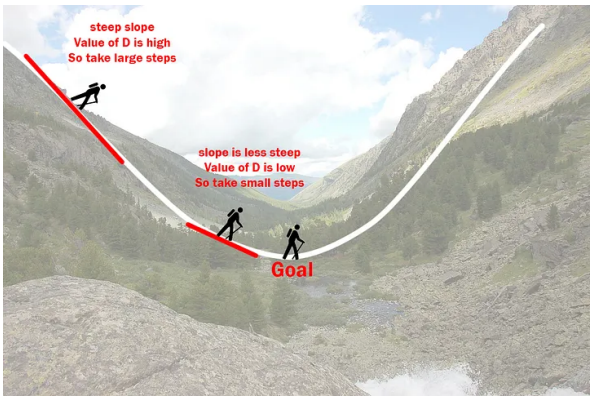



Figure 5.4: Illustration of how the gradient descent algorithm works.

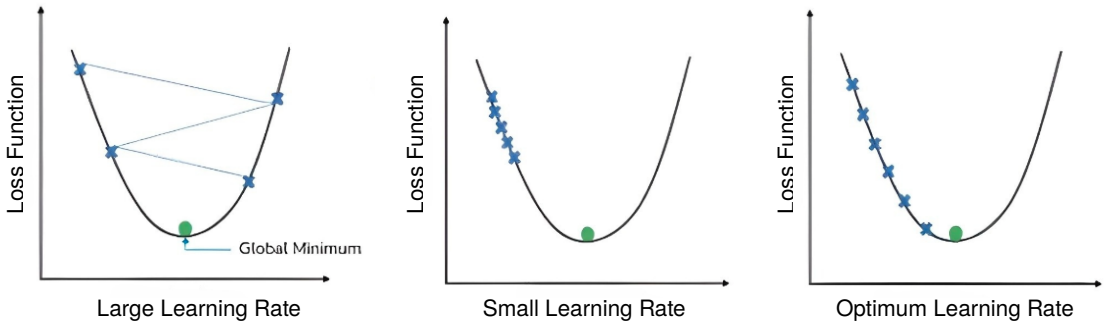



Figure 5.5: Gradient descent affects the way learning is carried out.

A slower learning rate helps to reach the global minimum but takes an unusually long time and computationally proves expensive. The faster learning rate may make the model wander and lead to an undesired position, making it difficult to come back on the correct track to reach the global minimum. Hence the learning rate should be neither too slow nor too fast if the global minimum is to be reached efficiently. See Figure 5.5.

Let's try applying gradient descent to $w$ and $b$ and approach it step by step:

I) Initially let $w = 0$ and $b = 0$. Let $\alpha$ be our learning rate. This controls how much the value of $w$ changes with each step. $\alpha$ could be a small value like 0.0001 for good accuracy.

II) The loss function $J$ (Mean Squared Error) was earlier (Equation (5.1)) defined as:

$$J = \frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2$$

where $n$ is the number of data points, $\hat{y}_i$ is the predicted target value for the $i$-th data point, and $y_i$ is the actual target value for the $i$-th data point.

III) The linear regression model was represented as:

$$y = b + w_1x_1 + w_2x_2 + \cdots + w_nx_n \tag{5.2}$$

where $y$ is the predicted target value, $b$ is the intercept term, and $w_i$ are the coefficients for the features $x_i$.

IV) Now, let's compute the partial derivatives of the loss function $J$ with respect to each parameter ($b$ and $w_i$) using the chain rule.

V) Partial derivative with respect to $b$:

$$\frac{\partial J}{\partial b} = \frac{2}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i) \tag{5.3}$$

VI) Partial derivative with respect to $w_i$:

$$\frac{\partial J}{\partial w_i} = \frac{2}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)\cdot x_i \tag{5.4}$$

These partial derivatives give us the gradient of the loss function with respect to each parameter, which we can use in the gradient descent algorithm to update the parameters iteratively.

VII) To update the values of $b$ and $w$, in each iteration, we use $\frac{\partial J}{\partial b}$ and $\frac{\partial J}{\partial w_i}$ respectively, the following rules:

i. Update rule for $b$:

$$b := b - \alpha \cdot \frac{\partial J}{\partial b} \tag{5.5}$$

ii. Update rule for $w_i$:

$$w_i := w_i - \alpha \cdot \frac{\partial J}{\partial w_i} \tag{5.6}$$

where $\alpha$ is the learning rate, a hyperparameter controlling the size of the step taken in the direction of the gradient.

VIII) These update rules ensure that in each iteration, we move the parameters $b$ and $w$ in the direction of steepest decrease of the loss function $J$, ultimately converging towards the optimal parameter values that minimize the loss function and result in the best fit line for the given training data.

IX) We repeat this process until our loss function is a very small value or ideally 0 (which means 0 error or 100% accuracy). The value of $w$ and $b$ that we are left with now will be the optimum values.

In our analogy, $w$ represents the current position of a person, akin to the current parameter value in our optimization problem. $D$ corresponds to the steepness of the slope, representing the derivative of the loss function with respect to the parameter. Meanwhile, $\alpha$ can be likened to the speed at which the person moves, serving as the learning rate in our optimization process.

When we calculate the new value of $w$ using the update equation, it essentially determines the person's next position, and $\alpha \times D$ indicates the size of the steps the person will take. If the slope is steep ($D$ is larger), the person takes longer steps, while on a less steep slope ($D$ is smaller), the person takes smaller steps. Eventually, the person reaches the bottom of the valley, which corresponds to our desired outcome: a loss value of 0.

## 5.5 Assumptions of Simple Linear Regression

Linear regression is a powerful tool for understanding and predicting the behavior of a variable, however, it needs to meet a few conditions in order to be accurate and dependable solutions.

I. **Linearity:** The independent and dependent variables have a linear relationship with one another. This implies that changes in the dependent variable follow those in the

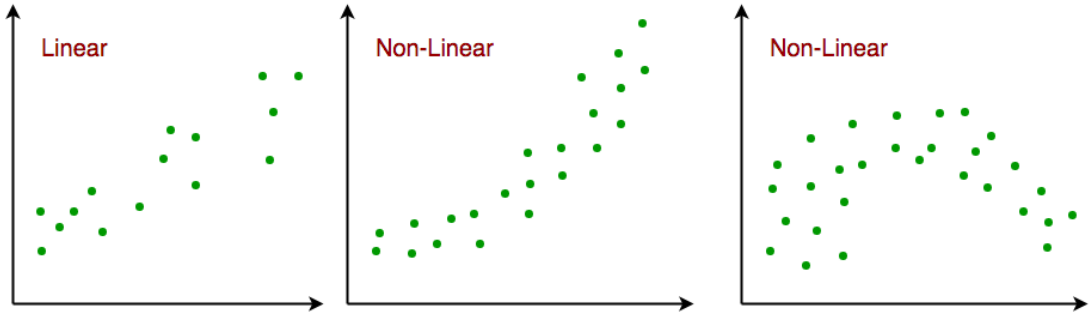



Figure 5.6: Linearity in linear Regression.

independent variable(s) in a linear fashion. This means that there should be a straight line that can be drawn through the data points. If the relationship is not linear, then linear regression will not be an accurate model.

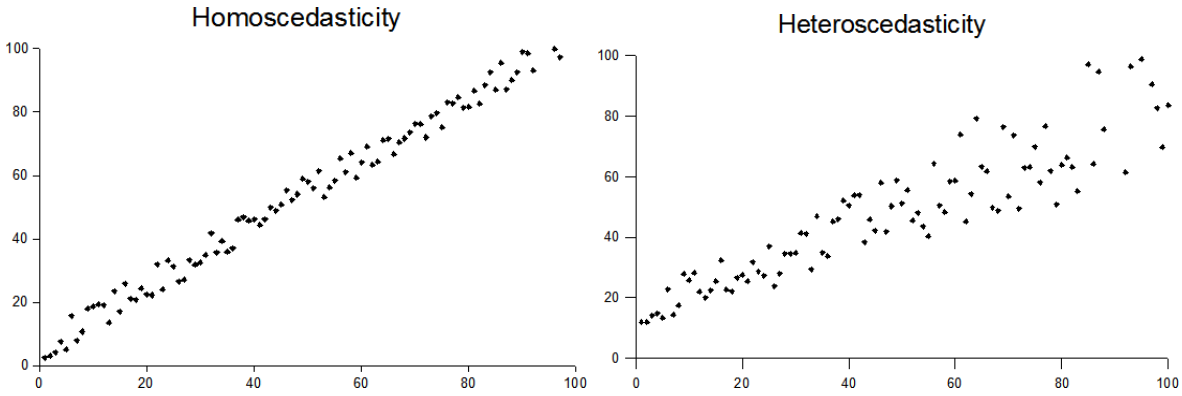
II. **Independence:** Independence is a fundamental assumption in linear regression. It implies that the observations in the dataset are unrelated to each other. In other words, the value of the dependent variable for one observation is not influenced by the value of the dependent variable for another observation. If the observations are not independent, the accuracy of the linear regression model may be compromised.

   For example, consider a dataset that measures the relationship between hours of study and exam scores for students. If the observations are independent, it means that the exam score of one student is not affected by the exam scores of other students. Each student's performance is solely determined by their own effort and ability. However, if there is dependence among the observations, such as students sharing study materials or collaborating on exam preparation, then the assumption of independence is violated, and linear regression may not provide accurate predictions. Therefore, ensuring independence among observations is essential for the reliability of the linear regression analysis.

III. **Homoscedasticity:** Homoscedasticity is a critical assumption in linear regression, indicating that the variance of the errors (or residuals) remains constant across all levels of the independent variable(s). In simpler terms, it means that the spread or dispersion of the residuals around the regression line remains consistent, regardless of the values of the independent variable(s). If homoscedasticity is violated, it suggests that the variability of the errors changes with the values of the independent variable(s), leading to potential inaccuracies in the linear regression model. See Figure 5.7.

   To illustrate this concept with examples:

   i. **Education and Income**: Suppose we are examining the relationship between years of education and annual income. In a homoscedastic scenario, regardless of the level of education (e.g., 10 years, 15 years, 20 years), the spread of the residuals around the regression line remains consistent. This implies that the variability in income prediction errors is constant across different levels of education. However,




(a) (b)

Figure 5.7: Homoscedasticity in linear regression (images taken from *Homoscedasticity and heteroscedasticity*, https://en.wikipedia.org/wiki/Homoscedasticity_and_heteroscedasticity). (a) Random data showing homoscedasticity: at each value of $x$, the $y$-value of the dots has about the same variance. (b) Random data showing heteroscedasticity: the variance of the $y$-values of the dots increase with increasing values of $x$.

if homoscedasticity is violated, we may observe that the variability in income prediction errors increases or decreases as the level of education changes. For instance, for individuals with higher levels of education, the prediction errors may exhibit greater variability compared to those with lower levels of education.

ii. **Temperature and Ice Cream Sales**: Consider a study investigating the relationship between temperature and daily ice cream sales. In a homoscedastic scenario, regardless of whether the temperature is low, moderate, or high, the spread of the residuals around the regression line remains uniform. This implies that the variability in ice cream sales prediction errors is consistent across different temperature levels. However, if homoscedasticity is violated, we may observe that the variability in ice cream sales prediction errors varies depending on the temperature. For instance, on extremely hot days, the prediction errors may exhibit larger variability compared to milder days.

Homoscedasticity ensures that the errors in predicting the dependent variable are consistent across all levels of the independent variable(s). Violations of homoscedasticity can lead to biased parameter estimates and inaccurate predictions in linear regression analysis. Therefore, it is crucial to assess and address homoscedasticity when performing regression modeling. If the variance of the residuals is not constant, then linear regression will not be an accurate model.

IV. **Normality:** Normal distribution of residuals is crucial for the accuracy of a linear regression model. Residuals, representing the differences between observed and predicted values, should follow a bell-shaped curve. Deviations from normality indicate potential inaccuracies in the model. Skewness, outliers, or non-symmetrical distributions may suggest the model fails to capture underlying data patterns. Therefore, ensuring normality through graphical assessments and statistical tests is essential for

reliable predictions.

## 5.6 Assumptions of Multiple Linear Regression

For Multiple Linear Regression, all four of the assumptions from Simple Linear Regression apply. In addition to this, below are few more:

1. **No multicollinearity (c.f. Section 5.6.1):** There is no high correlation between the independent variables. This indicates that there is little or no correlation between the independent variables. Multicollinearity occurs when two or more independent variables are highly correlated with each other, which can make it difficult to determine the individual effect of each variable on the dependent variable. If there is multicollinearity, then multiple linear regression will not be an accurate model.
2. **Additivity:** The model assumes that the effect of changes in a predictor variable on the response variable is consistent regardless of the values of the other variables. This assumption implies that there is no interaction between variables in their effects on the dependent variable.

   Suppose we are building a linear regression model to predict house prices based on various factors such as square footage, number of bedrooms, and neighborhood crime rate. In an additive scenario, the increase in house price associated with an additional bedroom remains constant regardless of the square footage or crime rate. This means that the effect of adding a bedroom to a house is constant, irrespective of other factors. However, if additivity is violated, we may observe interactions between predictor variables, such as the effect of square footage on house price being dependent on the number of bedrooms.
3. **Feature Selection:** In multiple linear regression, it is essential to carefully select the independent variables that will be included in the model. Including irrelevant or redundant variables may lead to overfitting and complicate the interpretation of the model.
4. **Overfitting:** Overfitting occurs when the model fits the training data too closely, capturing noise or random fluctuations that do not represent the true underlying relationship between variables. This can lead to poor generalization performance on new, unseen data.

### 5.6.1 Multicollinearity[2].

Multicollinearity is a statistical phenomenon that occurs when two or more independent variables in a multiple regression model are highly correlated, making it difficult to assess the individual effects of each variable on the dependent variable.

While the strength of a regression model improves with increasing correlation between the predictors and the dependent variable, the presence of strong correlations among the

[2]Adapted from S. Duda, *Identifying and Addressing Multicollinearity in Regression Analysis*, Oct. 2022

independent variables can have a detrimental effect on model explainability and predictor standard error.

#### 5.6.1.1 Why is Multicollinearity a Problem?

For regression models, the impact of multicollinearity on model performance varies depending on the nature of the dataset and the model application. Multicollinearity primarily causes problems when retraining a model on new data or trying to explain the model using the regression coefficients. The presence of multicollinearity among two or more predictors makes it more difficult for a regression algorithm to assign each of these predictors with an accurate coefficient since it is harder to numerically distinguish predictors with a strong collinear relationship from one another.

Multicollinearity indicates an overlap in the explanatory information provided by two or more predictors. These predictors are providing redundant information, making it difficult for the regression algorithm to determine how much weight should be assigned to each. As a result, the regression algorithm may assign them different weights from a wide range of values under slightly different training scenarios, potentially even flipping the sign of the coefficients between positive and negative arbitrarily. This is reflected in the standard error of the predictors, which is higher for features that demonstrate multicollinearity than for those that do not.

While having dramatically different coefficients each time the model is retrained may not have a major impact on predictive performance, it introduces problems when trying to explain the importance of each predictor. The importance of highly correlated predictors will change each time the model is retrained, making it difficult to draw any meaningful conclusions about the real-world explanation of the model. The presence of multicollinearity can also obscure the relationship between each individual predictor variable and the dependent variable, further decreasing the model's interpretability.

It is important to note that model interpretability can be significantly more important than model accuracy or predictive capability when building models in some client applications. For example, marketing mix modeling involves the analysis of regression models to determine the impact of various types of marketing spend as well as other exogenous variables on client KPIs such as sales, new customers, etc. For these applications, the client is typically more interested in the overall impact of each type of marketing spend (represented by regression coefficients) and how they can improve their resource allocation to maximize efficiency than the model's predictive capacity.

#### 5.6.1.2 Detecting Multicollinearity

Detecting multicollinearity includes the following two techniques.

**Correlation Matrix** Examining the correlation matrix among the independent variables is a common way to detect multicollinearity. High correlations (close to 1 or -1) indicate potential multicollinearity.


Figure 5.8: Pearson correlation matrix for the standardized independent variables in the wine quality dataset.

A quick way to identify potential multicollinearity is to review the correlation matrix for the predictor variables. A correlation coefficient with an absolute value $> 0.7$ typically indicates a strong correlation between predictor variables, but it is important to note that this is just a rule of thumb. Removing some redundant predictors that are highly correlated can help reduce multicollinearity within your training data, improving the stability of the resulting model's predictor coefficients when retraining. While a correlation matrix can help for identifying pairwise multicollinearity, it does not help with identification of higher order multicollinearity that may exist within groups of predictors.

As an example, we can look at the correlations among independent variables included in the wine quality dataset[3], a commonly used dataset for regression exercises. For simplicity, we will look at white wine only. This dataset includes eleven quantitative physicochemical measurements (independent variables) associated with 4,898 white wine samples as well as a quality score (dependent variable) between 0 and 10 assigned by human judges.

A Pearson correlation matrix for the standardized independent variables is shown in Figure 5.8. Here we can see that a strong negative correlation ($r = -0.78$) exists between density and alcohol. Density also has moderate negative correlation ($-0.5 \leq r \leq -0.3$) with three other variables (`residual_sugar`, `chlorides`, and `total_sulfur_dioxide`). The presence of these correlations suggest that multicollinearity among some predictors may be present but further analysis is needed.

[3] https://archive.ics.uci.edu/ml/datasets/wine+Quality

**R-Squared**

R-Squared ($R^2$ or the coefficient of determination) is a statistical measure in a regression model that determines the proportion of variance in the dependent variable that can be explained by the independent variable. In other words, R-squared shows how well the data fit the regression model (the goodness of fit).
R-squared can take any values between 0 to 1. The most common interpretation of R-squared is how well the regression model explains observed data. For example, an r-squared of 60% reveals that 60% of the variability observed in the target variable is explained by the regression model. Generally, a higher R-squared indicates more variability is explained by the model.

#### 5.6.1.3 Variance Inflation Factor (VIF)

Variance Inflation Factor (VIF) is a measure that quantifies how much the variance of an estimated regression coefficient increases if the predictors are correlated. A high VIF (typically above 10) suggests multicollinearity.

VIF is calculated based on the tolerance (Tol) of each predictor variable, which quantifies the proportion of variance in that variable that is not explained by the other predictor variables in the model. The equations used to calculate tolerance and VIF are as follows:

$$\text{Tol} = 1 - R_j^2 \tag{5.7}$$

$$\text{VIF} = \frac{1}{\text{Tol}} \tag{5.8}$$

Here, $R_j^2$ represents the coefficient of determination for each predictor variable $j$. Tolerance is calculated as one minus the squared multiple correlation coefficient ($R_j^2$), which measures the proportion of variance in the predictor variable that is not accounted for by the other predictor variables. The VIF is then computed as the reciprocal of tolerance.

In practical terms, higher VIF values indicate stronger multicollinearity among predictor variables, suggesting that those variables are highly correlated with each other. Conversely, lower VIF values indicate lower levels of multicollinearity and greater independence among predictor variables.

Taking the square root of the VIF provides a measurement of the magnitude of the standard error for the predictor coefficient relative to a scenario where the predictor was completely uncorrelated with all other independent variables. For example, if an independent variable had a VIF of 25, this would indicate that the standard error for that independent variable was 5 times larger than it would be if the predictor was uncorrelated with any of the other predictors.

For the example of white wine quality dataset Section 5.6.1.2, we can look at the VIF values for each predictor calculated using scaled data to identify potential multicollinearity.

Table 5.1: Depicting VIF for the white wine quality dataset. Table taken from S. Duda, *Identifying and Addressing Multicollinearity in Regression Analysis*, Oct. 2022.

| Column Name | VIF |
|---|---|
| density | 28.232546 |
| residual_sugar | 12.644064 |
| alcohol | 7.706957 |
| fixed_acidity | 2.691435 |
| total_sulfur_dioxide | 2.239233 |
| pH | 2.196362 |
| free_sulfur_dioxide | 1.787880 |
| chlorides | 1.236822 |
| citric_acid | 1.165215 |
| volatile_acidity | 1.128298 |
| sulphates | 1.138540 |

(a) Density included

| Column Name | VIF |
|---|---|
| total_sulfur_dioxide | 2.153170 |
| free_sulfur_dioxide | 1.744627 |
| alcohol | 1.647117 |
| residual_sugar | 1.435215 |
| fixed_acidity | 1.356128 |
| pH | 1.330912 |
| chlorides | 1.203645 |
| citric_acid | 1.159884 |
| volatile_acidity | 1.128298 |
| sulphates | 1.056637 |

(b) Density removed

In Table 5.1 the table on the left below indicates high VIF values (> 10) for density and `residual_sugar`, which matches what we observed with the correlation matrix. Removing the predictor with the highest observed VIF and correlation (density) produces the table on the right below. Notice how the VIF values are now less than 5 for all predictors, indicating reduced multicollinearity following removal of this independent variable.

## 5.7 Hands On

### 5.7.1 Understanding the Loss Function

We use a simple line to understand how the Mean Squared Error (MSE) varies for the intercept (or bias) and slope (or weight) parameters. This hands on is related to Section 5.3 (Loss Function in Linear Regression).

### 5.7.2 Demonstration on Gradient Descent

We take some randomly generated data points and try to understand how exactly the optimum parameters are being predicted using Gradient Descent. This hands on is related to Section 5.4 (Gradient Descent for Linear Regression).

### 5.7.3 Demonstration on Correlation Matrix

This hands on is related to Section 5.6.1.2 (Detecting Multicollinearity).

### 5.7.4 Demonstration on Variance Inflation Factor (VIF)

This hands on is related to Section 5.6.1.3 (Variance Inflation Factor (VIF)).

### 5.7.5 Linear Regression Based on Statistical Analysis

Linear regression on the wine equality dataset which performs statistical analysis.

# Chapter 6

# Logistic Regression and Introduction to Neural Network[1]

## 6.1 Logistic Regression

Logistic regression is one of the most important analytic tools in the social and natural sciences. In many areas of machine learning, logistic regression is the baseline supervised machine learning algorithm for classification. It has a very close relationship with neural networks. As discussed in Section 6.2, a neural network can be viewed as a series of logistic regression classifiers stacked on top of each other.

Logistic regression is a supervised machine learning algorithm used for classification tasks where the goal is to predict the probability that an instance belongs to a given class or not. It can be used to classify an observation into one of two classes (like 'positive sentiment' and 'negative sentiment'), or into one of many classes.

Logistic regression is a statistical algorithm which analyze the relationship between two data factors.

### 6.1.1 The Sigmoid Function

The goal of binary logistic regression is to train a classifier that can make a binary decision about the class of a new input observation. The sigmoid classifier helps us make this decision.

Consider a single input observation $x$, which we will represent by a vector of features $[x_1, x_2, \ldots, x_n]$. The classifier output $y$ can be 1 (meaning the observation is a member of the class) or 0 (the observation is not a member of the class). We want to know the probability $P(y = 1 \mid x)$ that this observation is a member of the class. So perhaps the decision is "positive sentiment" versus "negative sentiment".

Logistic regression solves this task by learning, from a training set, a vector of weights and a bias term. Each weight $w_i$ is a real number, and is associated with one of the input features $x_i$.

[1] Some parts of this chapter has been adapted from D. Jurafsky and J. H. Martin, *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition*, (3rd ed. draft).

The weight $w_i$ represents how important that input feature is to the classification decision and can be positive (providing evidence that the instance being classified belongs in the positive class) or negative (providing evidence that the instance being classified belongs in the negative class).

Thus we might expect in a sentiment task the word "awesome" to have a high positive weight, and "abysmal" to have a very negative weight. The bias term, also called the intercept, is another real number that's added to the weighted inputs.

To make a decision on a test instance — after we've learned the weights in training — the classifier first multiplies each $x_i$ by its weight $w_i$, sums up the weighted features, and adds the bias term $b$. The resulting single number $z$ expresses the weighted sum of the evidence for the class.

$$z = \sum_{i=1}^{n} w_i x_i + b \tag{6.1}$$

We represent such sums using the dot product notation from linear algebra.

The dot product of two vectors $a$ and $b$, written as $a \cdot b$, is the sum of the products of the corresponding elements of each vector. Thus the following is an equivalent formation to Equation (6.1):

$$z = \mathbf{w} \cdot \mathbf{x} + b \tag{6.2}$$

But note that nothing in Equation (6.2) forces $z$ to be a legal probability, that is, to lie between 0 and 1.

In fact, since weights are real-valued, the output might even be negative. $z$ ranges from $-\infty$ to $\infty$. To create a probability, we'll pass $z$ through the sigmoid function, $\sigma(z)$. The sigmoid function (named because it looks like an $s$) is also called the logistic function, and gives logistic regression its name. The sigmoid function, shown graphically in Figure 6.1, has the following equation.

$$\sigma(z) = \frac{1}{1 + e^{-z}} = \frac{1}{1 + \exp(-z)} \tag{6.3}$$

The sigmoid has a number of advantages. It takes a real-valued number and maps it into the range $(0, 1)$, which is just what we want for a probability. Because it is nearly linear around 0 but flattens toward the ends, it tends to squash outlier values toward 0 or 1. And it's differentiable, this is useful for learning.

If we apply the sigmoid to the sum of the weighted features, we get a number between 0 and 1. To make it a probability, we just need to make sure that the two cases, $p(y = 1)$ and $p(y = 0)$, sum to 1.

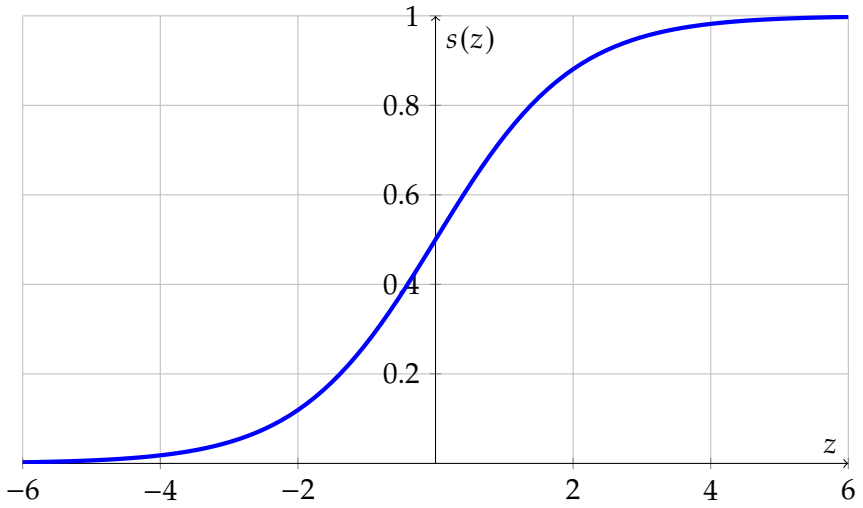



Figure 6.1: The sigmoid function $s(z) = \frac{1}{1+e^{-z}}$ takes a real value and maps it to the range $(0, 1)$. It is nearly linear around 0, but outlier values get squashed toward 0 or 1.

We can do this as follows:

$$\begin{aligned} P(y=1) &= \sigma(\mathbf{w}\cdot\mathbf{x}+b) \\ &= \frac{1}{1+\exp(-(\mathbf{w}\cdot\mathbf{x}+b))} \\ P(y=0) &= 1-\sigma(\mathbf{w}\cdot\mathbf{x}+b) \\ &= 1-\frac{1}{1+\exp(-(\mathbf{w}\cdot\mathbf{x}+b))} \\ &= \frac{\exp(-(\mathbf{w}\cdot\mathbf{x}+b))}{1+\exp(-(\mathbf{w}\cdot\mathbf{x}+b))} \end{aligned} \tag{6.4}$$

The sigmoid function has the property,

$$1-\sigma(x) = \sigma(-x). \tag{6.5}$$

So, we could also have expressed $P(y=0)$ as $\sigma(-(w\cdot x+b))$.

### 6.1.2 Classification with Logistic Regression

The logistic regression model transforms the linear regression function continuous value output into categorical value output using the sigmoid function, which, as mentioned above, maps any real-valued set of independent variables input into a value between 0 and 1.

The sigmoid function thus gives us a way to take an instance $x$ and compute the probability $P(y=1 \mid x)$.

How do we make a decision about which class to apply to a test instance $x$? For a given $x$, we say yes if the probability $P(y=1 \mid x)$ is more than 0.5, and no otherwise. We call 0.5 the decision boundary or threshold:

Table 6.1: Variable definitions and values in Figure 6.2.

| Var | Definition | Value |
|---|---|---|
| $x_1$ | count(positive lexicon words in doc) | 3 |
| $x_2$ | count(negative lexicon words in doc) | 2 |
| $x_3$ | 1 if "no", 0 otherwise in doc | 1 |
| $x_4$ | count(1st and 2nd pronouns in doc) | 3 |
| $x_5$ | 1 if "!", 0 otherwise in doc | 0 |
| $x_6$ | ln(word count of doc) | ln(66) = 4.19 |

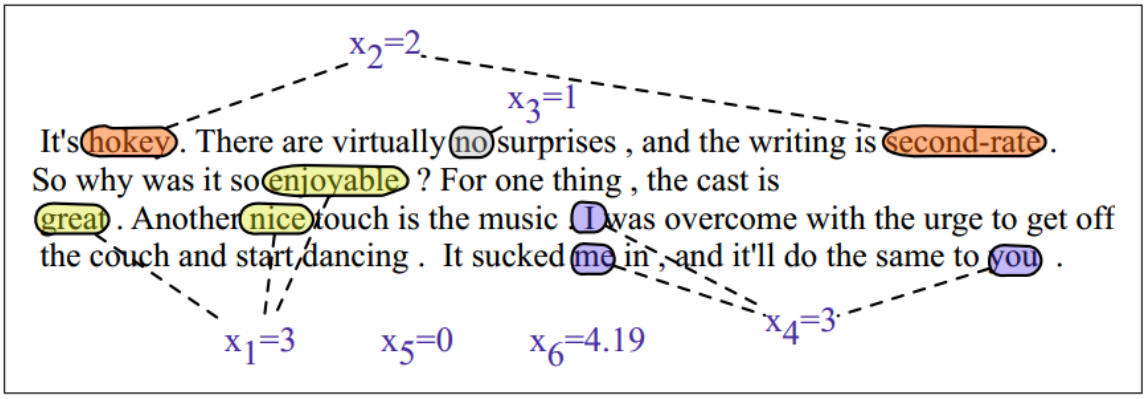



Figure 6.2: A sample mini test document showing the extracted features in the vector $x$.

$$\text{decision}(x) = \begin{cases} 1 & \text{if } P(y = 1 \mid x) > 0.5 \\ 0 & \text{otherwise.} \end{cases} \tag{6.6}$$

Let's have an example of applying logistic regression as a classifier for language tasks.

#### 6.1.2.1 Sentiment Classification

Suppose we are doing binary sentiment classification on movie review text, and we would like to know whether to assign the sentiment class + or − to a review document *doc*. We'll represent each input observation by the 6 features $x_1 \cdots x_6$ of the input shown in Table 6.1.

Figure 6.2 shows the features in a sample mini test document.

Let's assume for the moment that we've already learned a real-valued weight for each of these features, and that the 6 weights corresponding to the 6 features are [2.5, −5.0, −1.2, 0.5, 2.0, 0.7], while $b = 0.1$. The weight $w_1$, for example, indicates how important a feature the number of positive lexicon words (*great, nice, enjoyable*, etc.) is to a positive sentiment decision, while $w_2$ tells us the importance of negative lexicon words. Note that $w_1 = 2.5$ is positive, while $w_2 = -5.0$, meaning that negative words are negatively associated with a positive sentiment decision, and are about twice as important as positive words.

Given these six features and the input review $x$, $P(+ \mid x)$ and $P(- \mid x)$ can be computed using Equation (6.4):

$$\begin{aligned} p(+ \mid x) = P(y = 1 \mid x) &= \sigma(\mathbf{w} \cdot \mathbf{x} + b) \\ &= \sigma\left([2.5, -5.0, -1.2, 0.5, 2.0, 0.7] \cdot [3, 2, 1, 3, 0, 4.19] + 0.1\right) \\ &= \sigma\left(2.5 \times 3 + (-5.0) \times 2 + (-1.2) \times 1 + 0.5 \times 3 + 2.0 \times 0 + \right. \\ &\qquad \left. 0.7 \times 4.19 + 0.1\right) \\ &= \sigma\left(7.5 - 10.0 - 1.2 + 1.5 + 0 + 2.933 + 0.1\right) \\ &= \sigma(0.833) \\ &= 0.70 \\ p(- \mid x) = P(y = 0 \mid x) &= 1 - \sigma(w \cdot x + b) = 0.30 \end{aligned} \tag{6.7}$$

### 6.1.3 Multinomial Logistic Regression

Sometimes, we need more than two classes. Perhaps we might want to do 3-way sentiment classification (positive, negative, or neutral). Or we could be assigning some of the labels, like the part of speech of a word (choosing from 10, 30, or even 50 different parts of speech), or the named entity type of a phrase (choosing from tags like person, location, organization). In such cases, we use Multinomial Logistic Regression, also called softmax regression.

In Multinomial Logistic Regression, we want to label each observation with a class $k$ from a set of $K$ classes, under the stipulation that only one of these classes is the correct one. This is sometimes called hard classification, an observation cannot be in multiple classes.

Let's use the following representation: the output $\mathbf{y}$ for each input $\mathbf{x}$ will be a vector of length $K$. If class $c$ is the correct class, we'll set $y_c = 1$, and set all the other elements of $y$ to be 0, i.e., $y_c = 1$ and $y_j = 0, \forall j \neq c$. A vector like this $\mathbf{y}$, with one value being 1 and the rest 0, is called a one-hot vector. The job of the classifier is to produce an estimate vector $\hat{\mathbf{y}}$. For each class $k$, the value $\hat{y}_k$ will be the classifier's estimate of the probability $p(y_k = 1 \mid x)$.

### 6.1.4 Softmax

The multinomial logistic classifier uses a generalization of the sigmoid, called the softmax function, to compute $p(y_k = 1 \mid x)$. The softmax function takes a vector $\mathbf{z} = [z_1, z_2, \ldots, z_K]$ of $K$ arbitrary values and maps them to a probability distribution, with each value in the range $[0, 1]$, and all the values summing to 1. The input values can be positive, negative, zero, or greater than one, but the softmax transforms them into values between 0 and 1, so that they can be interpreted as probabilities. If one of the inputs is small or negative, the softmax turns it into a small probability, and if an input is large, then it turns it into a large probability, but it will always remain between 0 and 1. See Figure 6.3.

Like the sigmoid, it is an exponential function.

For a vector $\mathbf{z}$ of dimensionality $K$, the softmax is defined as:

$$\text{softmax}(z_j) = \frac{\exp(z_j)}{\sum_{i=1}^{K} \exp(z_i)}, \qquad 1 \leq j \leq K \tag{6.8}$$

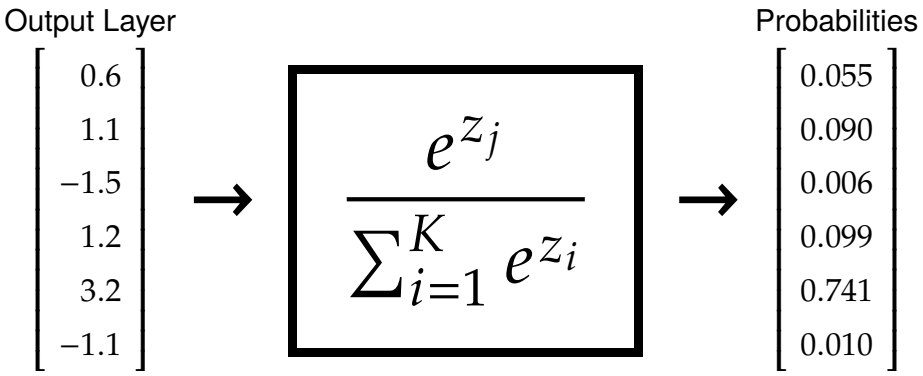



Figure 6.3: Softmax Function.

The softmax of an input vector $\mathbf{z} = [z_1, z_2, \ldots, z_K]$ is thus a vector itself:

$$\text{softmax}(\mathbf{z}) = \left[ \frac{\exp(z_1)}{\sum_{i=1}^{K} \exp(z_i)}, \frac{\exp(z_2)}{\sum_{i=1}^{K} \exp(z_i)}, \ldots, \frac{\exp(z_K)}{\sum_{i=1}^{K} \exp(z_i)} \right] \tag{6.9}$$

The denominator $\sum_{i=1}^{K} \exp(z_i)$ is used to normalize all the values into probabilities.

Like the sigmoid, the softmax has the property of squashing values toward 0 or 1. Thus, if one of the inputs is larger than the others, it will tend to push its probability toward 1, and suppress the probabilities of the smaller inputs.

### 6.1.5 Applying Softmax in Logistic Regression

When we apply softmax for logistic regression, the input will (just as for the sigmoid) be the dot product between a weight vector $\mathbf{w}$ and an input vector $\mathbf{x}$ (plus a bias). But now we'll need separate weight vectors $\mathbf{w}_k$ and bias $b_k$ for each of the $K$ classes. The probability of each of our output classes $\hat{y}_k$ can thus be computed as:

$$p(y_k = 1 \mid x) = \frac{\exp(\mathbf{w}_k \cdot \mathbf{x} + b_k)}{\sum_{j=1}^{K} \exp(\mathbf{w}_j \cdot \mathbf{x} + b_j)} \tag{6.10}$$

The form of Equation (6.10) makes it seem that we would compute each output separately. Instead, it's more common to set up the equation for more efficient computation by modern vector processing hardware. We'll do this by representing the set of $K$ weight vectors as a weight matrix $\mathbf{W}$ and a bias vector $\mathbf{b}$. Each row $k$ of $\mathbf{W}$ corresponds to the vector of weights $\mathbf{w}_k$. $\mathbf{W}$ thus has shape $[K \times f]$, for $K$ the number of output classes and $f$ the number of input features. The bias vector $\mathbf{b}$ has one value for each of the $K$ output classes. If we represent the weights in this way, we can compute $\hat{y}$, the vector of output probabilities for each of the $K$ classes, by a single elegant equation:

$$\hat{y} = \text{softmax}(\mathbf{W}\mathbf{x} + \mathbf{b}). \tag{6.11}$$

If you work out the matrix arithmetic, you can see that the estimated score of the first output class $\hat{y}_1$ (before we take the softmax) will correctly turn out to be $\mathbf{w}_1 \cdot \mathbf{x} + b_1$.

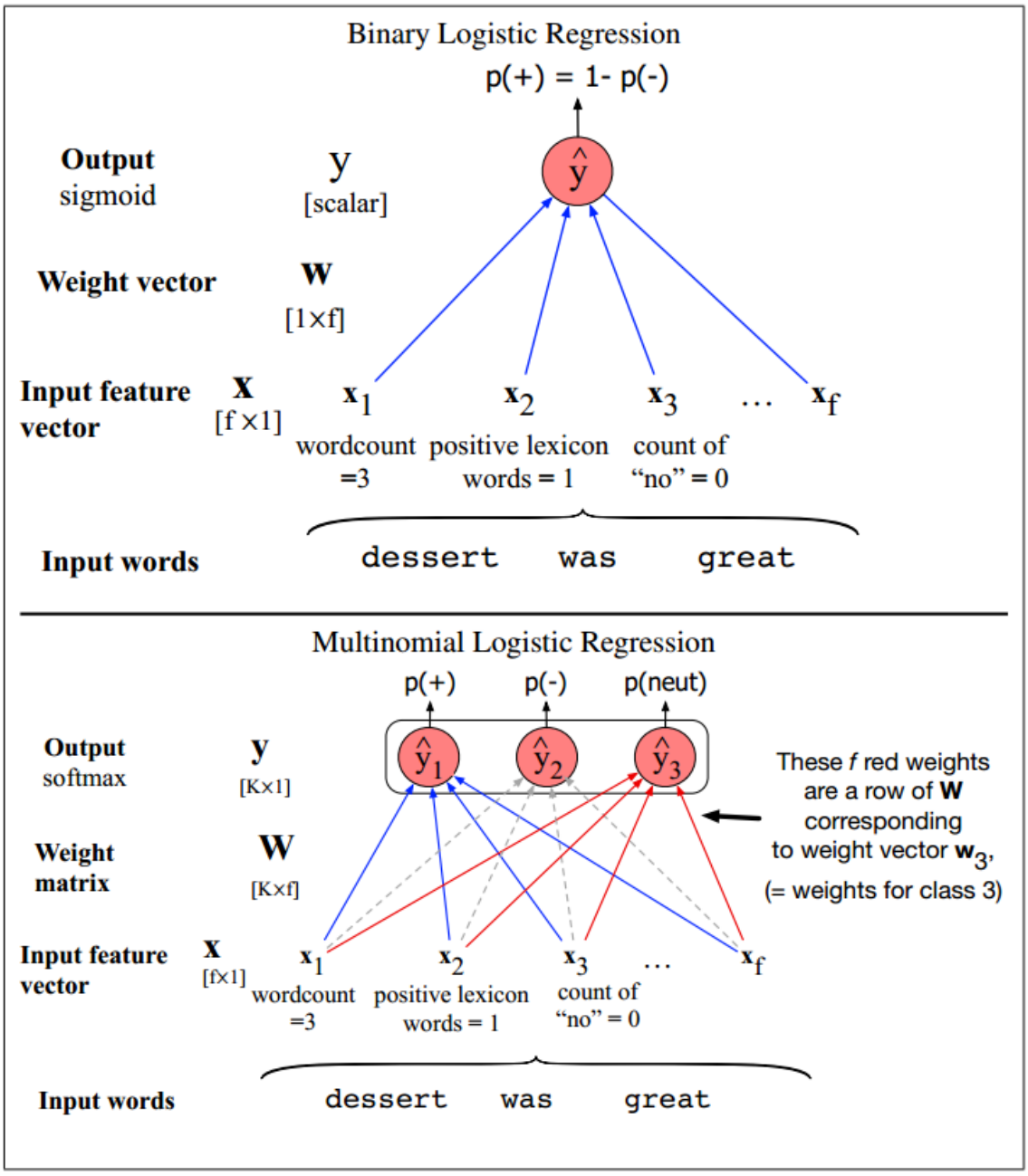



Figure 6.4: Binary versus multinomial logistic regression. Binary logistic regression uses a single weight vector $\mathbf{w}$, and has a scalar output $\hat{y}$. In multinomial logistic regression, we have $K$ separate weight vectors corresponding to the $K$ classes, all packed into a single weight matrix $\mathbf{W}$, and a vector output $\hat{\mathbf{y}}$.

Figure 6.4 shows an intuition of the role of the weight vector versus weight matrix in the computation of the output class probabilities for binary versus multinomial logistic regression.

### 6.1.6 Learning in Logistic Regression

How are the parameters of the model, the weights $\mathbf{w}$ and bias $b$, learned? Logistic regression is an instance of supervised classification in which we know the correct label $y$ (either 0 or 1) for each observation $\mathbf{x}$. What the system produces via Equation (6.4) is $\hat{y}$, the system's estimate of the true $y$. We want to learn parameters (meaning $\mathbf{w}$ and $b$) that make $\hat{y}$ for each training observation as close as possible to the true $y$.

This requires two components. The first is a metric for how close the current label ($\hat{y}$) is to the true gold label $y$. Rather than measure similarity, we usually talk about the opposite of this: the distance between the system output and the gold output, and we call this distance loss, the loss function or the cost function. Cross-entropy loss is commonly used for logistic regression and also for neural networks.

The second thing we need is an optimization algorithm for iteratively updating the weights so as to minimize this loss function. The standard algorithm for this is gradient descent.

#### 6.1.6.1 The Cross-Entropy Loss Function

We need a loss function that expresses, for an observation $\mathbf{x}$, how close the classifier output ($\hat{y} = \sigma(\mathbf{w} \cdot \mathbf{x} + b)$) is to the correct output ($y$, which is 0 or 1). We'll call this:

$$L(\hat{y}, y) = \text{How much } \hat{y} \text{ differs from the true } y. \tag{6.12}$$

We do this via a loss function that prefers the correct class labels of the training examples to be more likely. This is called conditional maximum likelihood estimation: we choose the parameters $\mathbf{w}$, $b$ that maximize the log probability of the true $y$ labels in the training data given the observations $\mathbf{x}$. The resulting loss, cross-entropy loss function, is the negative log-likelihood loss, generally called the cross-entropy loss.

#### 6.1.6.2 Gradient Descent

Our goal with gradient descent is to find the optimal weights: minimize the loss function we've defined for the model. The loss function is parameterized by the weights, which we'll refer to in machine learning in general as $\theta$ (in the case of logistic regression $\theta = \mathbf{w}, b$). So the goal is to find the set of weights which minimizes the loss function, averaged over all examples.

### 6.1.7 Hands On

#### 6.1.7.1 Predict Whether or Not It Will Rain Tomorrow in Australia

We predict whether or not it will rain tomorrow in Australia.

This dataset contains about 10 years of daily weather observations from many locations across Australia. `RainTomorrow` is the target variable to predict. It means – did it rain the next day, Yes or No? This column is Yes if the rain for that day was 1.0 mm or more.

Observations were drawn from numerous weather stations. The daily observations are available from http://www.bom.gov.au/climate/data. An example of the latest weather observations in Canberra: http://www.bom.gov.au/climate/dwo/IDCJDW2801.latest.shtml. Definitions have been adapted from http://www.bom.gov.au/climate/dwo/IDCJDW0000.shtml.

#### 6.1.7.2 Toxic Comment Classification

We are provided with a large number of Wikipedia comments which have been labeled by human raters for toxic behavior. The types of toxicity are: toxic, severe_toxic, obscene, threat, insult, identity_hate. We create a model which predicts a probability of each type of toxicity for each comment.

#### 6.1.7.3 MNIST Dataset Digit Recognition

The MNIST database of handwritten digits is a dataset with 784 features, The raw data is available at: http://yann.lecun.com/exdb/mnist/. It can be split in a training set of the first 60,000 examples, and a test set of 10,000 examples

It is a subset of a larger set available from NIST. The digits have been size-normalized and centered in a fixed-size image. It is a good database for people who want to try learning techniques and pattern recognition methods on real-world data while spending minimal efforts on preprocessing and formatting. The original black and white (bilevel) images from NIST were size normalized to fit in a $20 \times 20$ pixel box while preserving their aspect ratio. The resulting images contain gray levels as a result of the anti-aliasing technique used by the normalization algorithm. The images were centered in a $28 \times 28$ image by computing the center of mass of the pixels, and translating the image so as to position this point at the center of the $28 \times 28$ field.

## 6.2 Similarities and Dissimilarities between Logistic Regression and Neural Network

Logistic regression and neural networks share several similarities, making logistic regression a foundational concept in understanding neural networks. Here are some key points of similarity:

1. **Activation Function**: Both logistic regression and neural networks use activation functions to introduce non-linearity into the model. While logistic regression uses the sigmoid activation function, neural networks often employ a variety of activation functions such as ReLU, tanh, or softmax.

2. **Learning Mechanism**: Both logistic regression and neural networks learn from data using optimization algorithms such as gradient descent. The objective is to minimize a loss function that measures the discrepancy between predicted outputs and ground truth labels.
3. **Parameterization**: Logistic regression and neural networks are both parameterized models. In logistic regression, the parameters consist of weights and biases, while in neural networks, parameters include weights and biases across multiple layers.
4. **Output Representation**: Both models are capable of handling classification tasks, where they produce outputs that represent the probability of belonging to different classes. In logistic regression, this is achieved through the sigmoid function, while in neural networks, it can be through softmax or other activation functions depending on the task.
5. **Linear Combination**: Logistic regression can be seen as a simplified form of a neural network with a single layer. The computation in logistic regression involves taking a linear combination of input features followed by an activation function, which is analogous to the computation in a single-layer neural network.

Neural networks share much of the same mathematics as logistic regression. But neural networks are a more powerful classifier than logistic regression, and indeed a minimal neural network (technically one with a single 'hidden layer') can be shown to learn any function.

Neural net classifiers are different from logistic regression in another way. With logistic regression, we applied the regression classifier to many different tasks by developing many rich kinds of feature templates based on domain knowledge. When working with neural networks, it is more common to avoid most uses of rich handderived features, instead building neural networks that take raw words as inputs and learn to induce features as part of the process of learning to classify. Nets that are very deep are particularly good at representation learning. For that reason deep neural nets are the right tool for tasks that offer sufficient data to learn features automatically.

## 6.3 Neural Networks

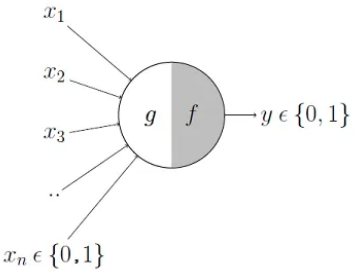
Neural networks are a fundamental computational tool for language processing, and a very old one. They are called neural because their origins lie in the McCulloch-Pitts neuron, a simplified model of the biological neuron as a kind of computing element that could be described in terms of propositional logic. But the modern use in language processing no longer draws on these early biological inspirations.

Instead, a modern neural network is a network of small computing units, each of which takes a vector of input values and produces a single output value. The architecture is called a feedforward network because the computation proceeds iteratively from one layer of units to the next.

**The McCulloch-Pitts Neuron**

**McCulloch-Pitts Neuron — Mankind's First Mathematical Model of a Biological Neuron**

The McCulloch-Pitts neuron, proposed by Warren McCulloch and Walter Pitts in 1943, is a simplified mathematical model of a biological neuron, serving as the foundation for modern artificial neural networks. It operates on binary inputs and produces a binary output, mimicking the behavior of biological neurons.




The neuron sums up its binary inputs and compares the result to a predefined threshold. If the sum exceeds the threshold, the neuron fires and outputs 1. Otherwise, it remains inactive and outputs 0. Unlike modern neural networks, the McCulloch-Pitts neuron does not have adjustable weights; instead, it relies on fixed weights associated with each input. Despite its simplicity, the McCulloch-Pitts neuron remains a valuable concept in neural network theory, providing insight into the fundamental principles of neural computation.

The use of modern neural nets is often called deep learning, because modern networks are often deep (have many layers).

### 6.3.1 The XOR Problem

Early in the history of neural networks it was realized that the power of neural networks, as with the real neurons that inspired them, comes from combining these units into larger networks.

One of the most clever demonstrations of the need for multi-layer networks was the proof that a single neural unit cannot compute some very simple functions of its input. Consider the task of computing elementary logical functions of two inputs, like AND, OR, and XOR.

This example was first shown for the perceptron, which is a very simple neural unit that has a binary output and does not have a non-linear activation function. The output $y$ of a perceptron is 0 or 1, and is computed as follows:

$$y = \begin{cases} 0 & \text{if } w \cdot x + b \leq 0, \\ 1 & \text{if } w \cdot x + b > 0. \end{cases} \tag{6.13}$$

It's very easy to build a perceptron that can compute the logical AND and OR functions of its binary inputs. Figure 6.5 shows the necessary weights.

It turns out, however, that it's not possible to build a perceptron to compute logical XOR!

The intuition behind this important result relies on understanding that a perceptron is a linear classifier. For a two-dimensional input $x_1$ and $x_2$, the perceptron equation, $w_1 x_1 + w_2 x_2 + b = 0$,

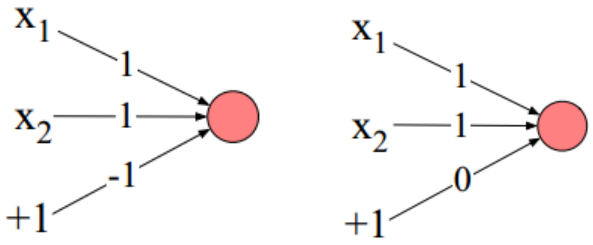



Figure 6.5: The weights $w$ and bias $b$ for perceptrons for computing logical functions. The inputs are shown as $x_1$ and $x_2$, and the bias as a special node with value +1 which is multiplied with the bias weight $b$. (a) Logical AND, with weights $w_1 = 1$ and $w_2 = 1$ and bias weight $b = -1$. (b) Logical OR, with weights $w_1 = 1$ and $w_2 = 1$ and bias weight $b = 0$. These weights/biases are just one from an infinite number of possible sets of weights and biases that would implement the functions.

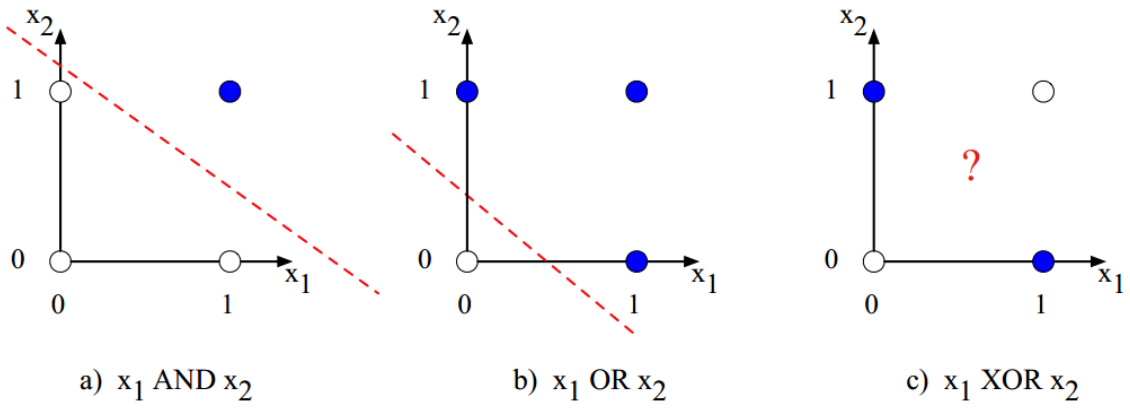



Figure 6.6: The functions AND, OR, and XOR, represented with input $x_1$ on the $x$-axis and input $x_2$ on the y-axis. Filled circles represent perceptron outputs of 1, and white circles represent perceptron outputs of 0. There is no way to draw a line that correctly separates the two categories for XOR.

is the equation of a line. (We can see this by putting it in the standard linear format: $x_2 = (-w_1/w_2)x_1 + (-b/w_2)$.) This line acts as a boundary decision in two-dimensional space in which the output 0 is assigned to all inputs lying on one side of the line, and the output 1 to all input points lying on the other side of the line. If we had more than two inputs, the decision boundary becomes a hyperplane instead of a line, but the idea is the same, separating the space into two categories.

Figure 6.6 shows the possible logical inputs (00, 01, 10, and 11) and the line drawn by one possible set of parameters for an AND and an OR classifier. Notice that there is simply no way to draw a line that separates the positive cases of XOR (01 and 10) separable linearly from the negative cases (00 and 11). We say that XOR is not a linearly separable function. Of course, we could draw a boundary with a curve, or some other function, but not a single line.

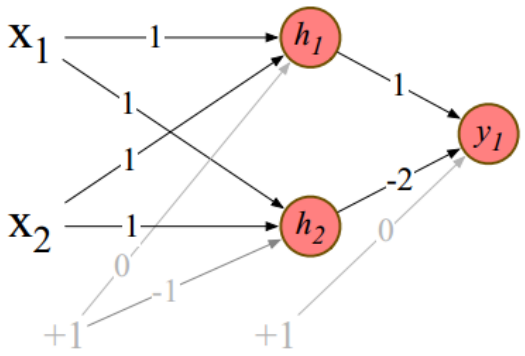



Figure 6.7: The XOR solution. There are three ReLU units, in two layers. We've called them $h_1$, $h_2$ (h for "hidden layer") and $y_1$. As before, the numbers on the arrows represent the weights $w$ for each unit, and we represent the bias $b$ as a weight on a unit clamped to +1, with the bias weights/units in gray.

### 6.3.2 The Solution: Neural Networks

While the XOR function cannot be calculated by a single perceptron, it can be calculated by a layered network of perceptron units. Rather than see this with networks of simple perceptrons, however, let's see how to compute XOR using two layers of ReLU-based units. Figure 6.7 shows a figure with the input being processed by two layers of neural units. The middle layer (called $h$) has two units, and the output layer (called $y$) has one unit. A set of weights and biases are shown that allows the network to correctly compute the XOR function.

### 6.3.3 Neural Networks — Visual Journey — First Part

For this part, the material at this site (`https://www.3blue1brown.com/lessons/neural-networks`, 3blue1brown, Neural Networks, Chapter 1) will be used.

### 6.3.4 The Building Block of a Neural Network

The building block of a neural network is a single computational unit. A unit takes a set of real valued numbers as input, performs some computation on them, and produces an output.

At its heart, a neural unit is taking a weighted sum of its inputs, with one additional term in the sum called a bias term. Given a set of inputs $x_1, \ldots, x_n$, a unit has a set of corresponding weights $w_1, \ldots, w_n$ and a bias $b$, so the weighted sum $z$ can be represented as:

$$z = b + \sum_i w_i x_i. \tag{6.14}$$

Often it's more convenient to express this weighted sum using vector notation. Thus we'll talk about $z$ in terms of a weight vector $w$, a scalar bias $b$, and an input vector $x$, and we'll replace the sum with the convenient dot product:

$$z = \mathbf{w} \cdot \mathbf{x} + b. \tag{6.15}$$

As defined in Equation (6.15), $z$ is just a real valued number.

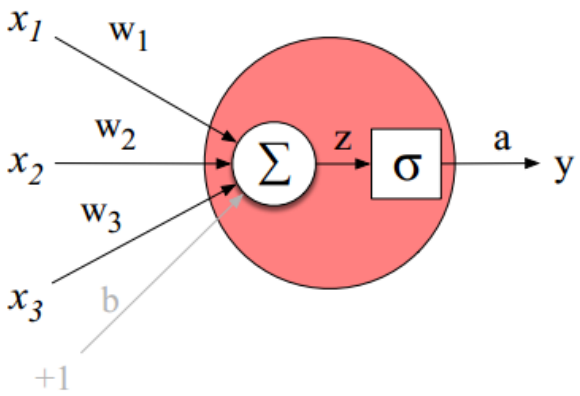



Figure 6.8: A neural unit takes 3 inputs $x_1, x_2$, and $x_3$ (and a bias $b$ that we represent as a weight for an input clamped at +1) and produces an output $y$. We include some convenient intermediate variables: the output of the summation, $z$, and the output of the sigmoid, $a$. In this case, the output of the unit $y$ is the same as $a$, but in deeper networks we'll reserve $y$ to mean the final output of the entire network, leaving $a$ as the activation of an individual node.

Finally, instead of using $z$, a linear function of $x$, as the output, neural units apply a non-linear function $f$ to $z$. We will refer to the output of this function as the activation value for the unit, $a$. Since we are just modeling a single unit, the activation for the node is in fact the final output of the network, which we'll generally call $y$. So the value $y$ is defined as:

$$y = a = f(z) \tag{6.16}$$

Neural network use three non-linear functions, the sigmoid, the tanh, and the rectified linear unit or ReLU.

$$y = \sigma(z) = \frac{1}{1 + e^{-z}} \tag{6.17}$$

As already discussed, the sigmoid has a number of advantages. It maps the output into the range $(0, 1)$, which is useful in squashing outliers toward 0 or 1. And it's differentiable. This is useful for learning.

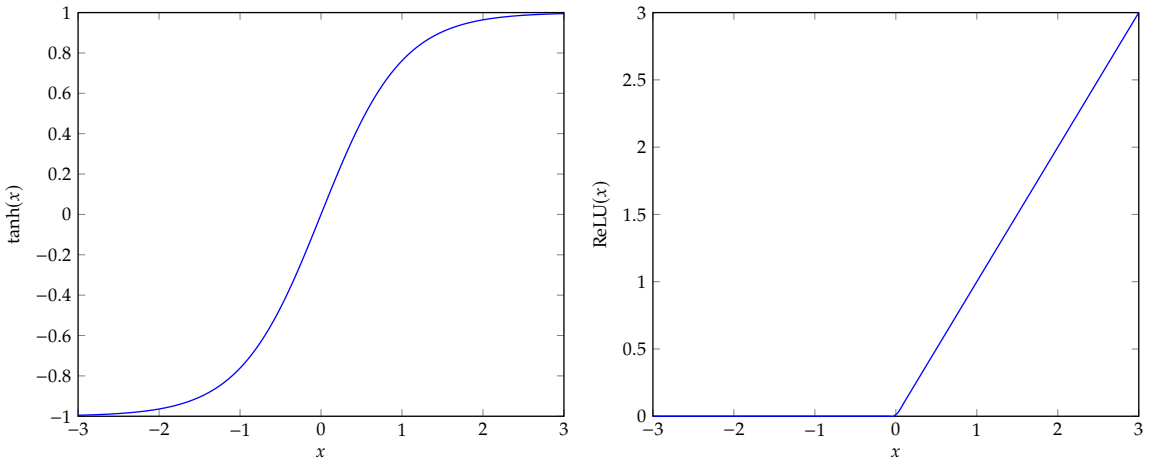
Substituting Equation (6.15) into Equation (6.17) gives us the output of a neural unit:

$$\begin{aligned} y &= \sigma(\mathbf{w} \cdot \mathbf{x} + b) \\ &= \frac{1}{1 + e^{-(\mathbf{w} \cdot \mathbf{x} + b)}}. \end{aligned} \tag{6.18}$$

Figure 6.8 shows a final schematic of a basic neural unit. In this example the unit takes 3 input values $x_1, x_2$, and $x_3$, and computes a weighted sum, multiplying each value by a weight ($w_1, w_2$, and $w_3$, respectively), adds them to a bias term $b$, and then passes the resulting sum through a sigmoid function to result in a number between 0 and 1.

Let's walk through an example just to get an intuition. Let's suppose we have a unit with the following weight vector and bias:

$$\mathbf{w} = [0.2, 0.3, 0.9]$$
$$b = 0.5$$




(a) (b)

Figure 6.9: The tanh and ReLU activation functions.

Let's find out the output from this unit do with the following input vector:

$$\mathbf{x} = [0.5, 0.6, 0.1]\,.$$

The resulting output $y$ would be:

$$\begin{aligned} y &= \sigma(w \cdot x + b) \\ &= \frac{1}{1 + e^{-(w \cdot x + b)}} \\ &= \frac{1}{1 + e^{-0.5\times0.2+0.6\times0.3+0.1\times0.9+0.5}} \\ &= \frac{1}{1 + e^{-0.87}} \\ &= 0.70 \end{aligned}$$

In practice, the sigmoid is not commonly used as an activation function. A function that is very similar but almost always better is the tanh function shown in Figure 6.9(a). tanh is a variant of the sigmoid that ranges from -1 to +1:

$$y = \tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}. \tag{6.19}$$

The simplest activation function, and perhaps the most commonly used, is the rectified linear unit, also called the ReLU, shown in Figure 6.9(b). It's just the same as $z$ when $z$ is positive, and 0 otherwise:

$$y = \text{ReLU}(z) = \max(z, 0). \tag{6.20}$$

These activation functions have different properties that make them useful for different language applications or network architectures. For example, the tanh function has the nice

properties of being smoothly differentiable and mapping outlier values toward the mean. The rectifier function, on the other hand, has nice properties that result from it being very close to linear. In the sigmoid or tanh functions, very high saturated values of $z$ result in values of $y$ that are saturated, i.e., extremely close to 1, and have derivatives very close to 0. Zero derivatives cause problems for learning. Because networks are trained by propagating an error signal backwards, multiplying gradients (partial derivatives) from each layer of the network. Gradients that are almost 0 cause the error signal to get smaller and smaller until it is too small to be used for training, a problem called the vanishing gradient problem. Rectifiers don't have this problem, since the derivative of ReLU for high values of $z$ is 1 rather than very close to 0.

### 6.3.5 Neural Networks — Visual Journey — Last Part

For the last part, the material at this site (`https://www.3blue1brown.com/lessons/neural-networks`, 3blue1brown, Neural Networks, Chapter 2 to Chapter 5) will be used.

### 6.3.6 Cost Functions in Neural Networks

A cost function helps to quantify how far the neural network's predictions are from the actual values. It is a measure of the error between the predicted output and the actual output. The cost function plays a crucial role in training a neural network. During the training process, the neural network adjusts its weights and biases to minimize the cost function. The goal is to find the minimum value of the cost function, which corresponds to the best set of weights and biases that make accurate predictions.

There are different types of cost functions, and the choice of cost function depends on the type of problem being solved. Here are some commonly used cost functions.

#### 6.3.6.1 Mean Squared Error (MSE)

The mean squared error is one of the most popular cost functions for regression problems. It measures the average squared difference between the predicted and actual values. The formula for MSE is:

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y - \hat{y})^2$$

Where:

- $n$ is the number of samples in the dataset,
- $y$ is the actual value,
- $\hat{y}$ is the predicted value.

#### 6.3.6.2 Binary Cross-Entropy

The binary cross-entropy cost function is used for binary classification problems. It measures the difference between the predicted and actual values in terms of probabilities. The formula

for binary cross-entropy is:

$$\text{Binary Cross-Entropy} = -\frac{1}{N}\sum_{i=1}^{N}\left(y_i \log(\hat{y}_i) + (1-y_i)\log(1-\hat{y}_i)\right) \tag{6.21}$$

Where:

- $N$ is the number of samples in the dataset,
- $y$ is the actual value (0 or 1),
- $\hat{y}$ is the predicted probability (between 0 and 1).

#### 6.3.6.3 Categorical Cross-Entropy

The categorical cross-entropy cost function is used for multi-class classification problems. It measures the difference between the predicted and actual values in terms of probabilities. The formula for categorical cross-entropy is:

$$\text{Categorical Cross-Entropy} = -\frac{1}{n}\sum_{i=1}^{n}\sum_{j=1}^{k} y_{ij}\log(\hat{y}_{ij}) \tag{6.22}$$

Where:

- $n$ is the number of samples in the dataset,
- $y_{ij}$ is the actual value of the $i$-th sample for the $j$-th class,
- $\hat{y}_{ij}$ is the predicted probability of the $i$-th sample for the $j$-th class.

See Section 6.4 for a detailed discussion on cross-entropy.

### 6.3.7 Hands On

#### 6.3.7.1 Implementation from Scratch of a Neural Network

We will build a neural network from scratch for the Breast Cancer Wisconsin (Diagnostic) dataset. We want to predict if cancer is benign or malignant. Using historical data about patients diagnosed with cancer enables doctors to differentiate malignant cases and benign ones are given independent attributes.

The features have been computed from a digitized image of a fine needle aspirate (FNA) of a breast mass. They describe characteristics of the cell nuclei present in the image. This dataset has been reported in W. N. Street, W. H. Wolberg, and O. L. Mangasarian, "Nuclear Feature Extraction for Breast Tumor Diagnosis," in *Electronic Imaging*, 1993.

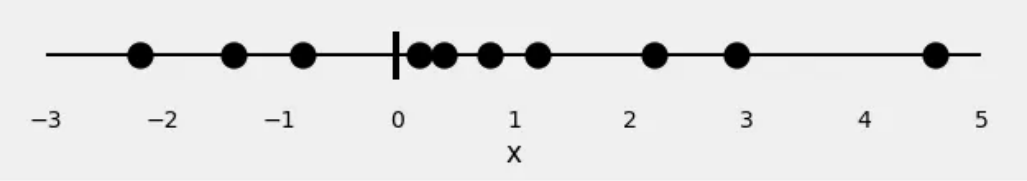



Figure 6.10: The feature.

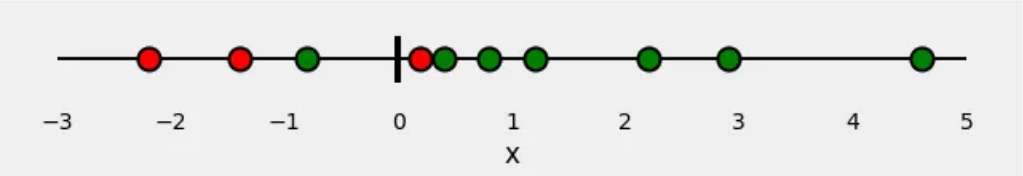



Figure 6.11: The labels.

## 6.4 Understanding Cross-Entropy[2]

### 6.4.1 A Simple Classification Problem

Let's start with 10 random points (Figure 6.10):

$$x = [-2.2, -1.4, -0.8, 0.2, 0.4, 0.8, 1.2, 2.2, 2.9, 4.6]$$

This is our only feature: $x$.

Now, let's assign some colors to our points: red and green. These are our labels (Figure 6.11). So, our classification problem is quite straightforward: given our feature $x$, we need to predict its label: red or green.

Since this is a binary classification, we can also pose this problem as: "Is the point green?" or, even better, "What is the probability of the point being green?" Ideally, green points would have a probability of 1.0 (of being green), while red points would have a probability of 0.0 (of being green).

In this setting, green points belong to the positive class (Yes, they are green), while red points belong to the negative class (No, they are not green).

If we fit a model to perform this classification, it will predict a probability of being green to each one of our points. Given what we know about the color of the points, how can we evaluate how good (or bad) are the predicted probabilities? This is the whole purpose of the loss function! It should return high values for bad predictions and low values for good predictions.

For a binary classification like our example, the typical loss function is the binary cross-entropy/log loss.

[2]Adapted from D. Godoy, *Understanding binary cross-entropy / log loss: A visual explanation*, Towards Data Science, Nov. 2018.

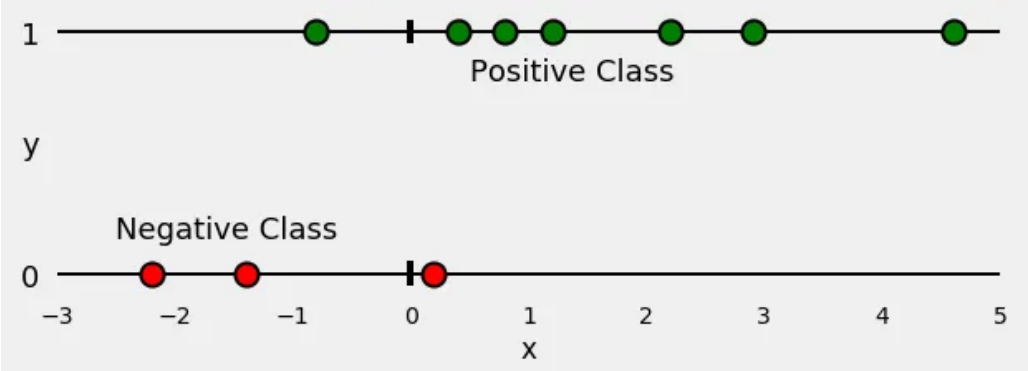



Figure 6.12: Splitting the data.

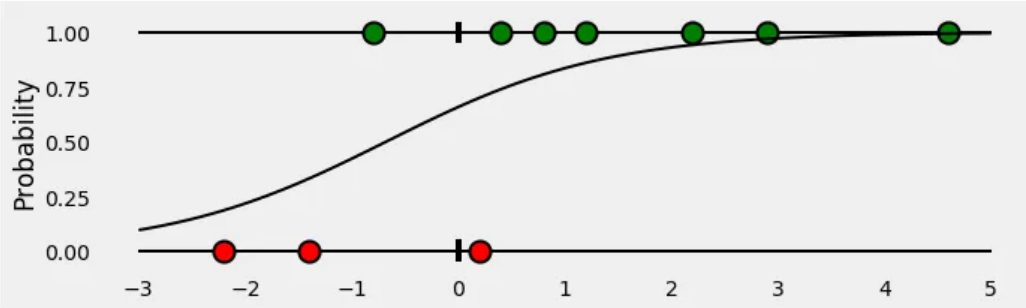



Figure 6.13: Fitting a logistic regression.

### 6.4.2 Loss Function: Binary Cross-Entropy/Log Loss

If we reproduce the formula for binary cross-entropy (Equation (6.21)):

$$\text{Binary Cross-Entropy} = -\frac{1}{N}\sum_{i=1}^{N}\left(y_i \log(\hat{y}_i) + (1-y_i)\log(1-\hat{y}_i)\right),$$

where $y_i$ is the label (1 for green points and 0 for red points) and $\hat{y}_i$ is the predicted probability of the point being green for all $N$ points.

What does entropy have to do with all this? Why are we taking log of probabilities in the first place?

### 6.4.3 Computing the Loss — the Visual Way

First, let's split the points according to their classes, positive or negative. See Figure 6.12. Now, let's train a Logistic Regression (Section 6.1) to classify our points. The fitted regression is a sigmoid curve (Figure 6.13) representing the probability of a point being green for any given $x$.

Then, for all points belonging to the positive class (green), what are the predicted probabilities given by our classifier? These are the green bars (Figure 6.14) under the sigmoid curve, at the $x$ coordinates corresponding to the points.

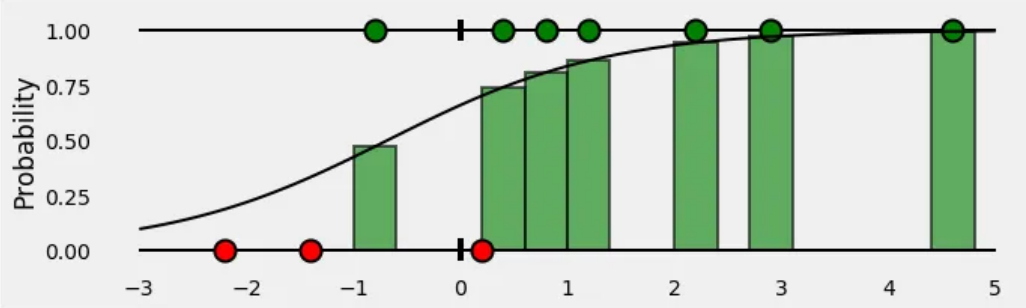



Figure 6.14: Probabilities of classifying points in the positive class correctly.

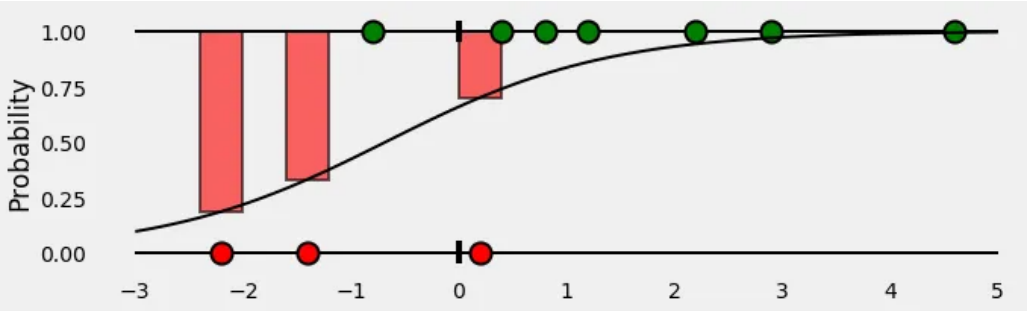



Figure 6.15: Probabilities of classifying points in the negative class correctly.

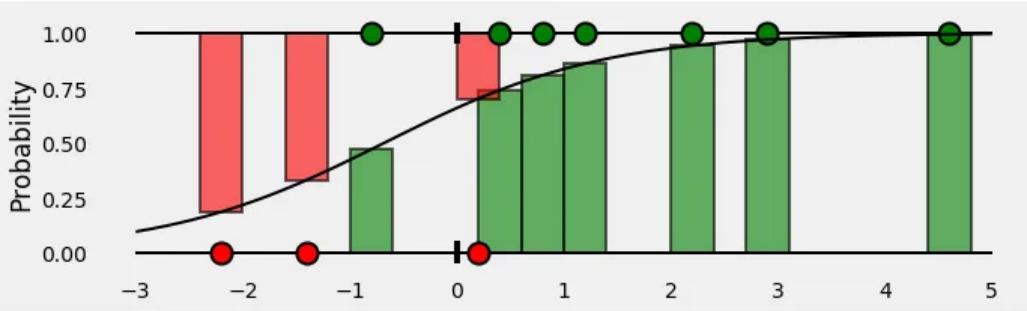



Figure 6.16: All probabilities put together.

What about the points in the negative class? Remember, the green bars under the sigmoid curve represent the probability of a given point being green. So, what is the probability of a given point being red? These will be the red bars *above* the sigmoid curve (Figure 6.15).

Putting it all together, we end up with something like shown in Figure 6.16.

The bars represent the predicted probabilities associated with the corresponding true class of each point. We have the predicted probabilities. Now, we evaluate them by computing the binary cross-entropy/log loss.

These probabilities are all we need, so, let's get rid of the $x$ axis and bring the bars next to each other. See Figure 6.17. The hanging bars don't make much sense anymore, so let's reposition them as in Figure 6.18.

Since we're trying to compute a loss, we need to penalize bad predictions. If the probability associated with the true class is 1.0, we need its loss to be zero. Conversely, if that probability is low, say, 0.01, we need its loss to be a huge value.

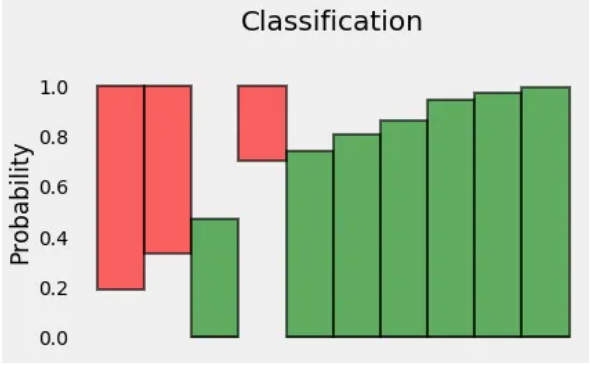



Figure 6.17: Probabilities of all points.

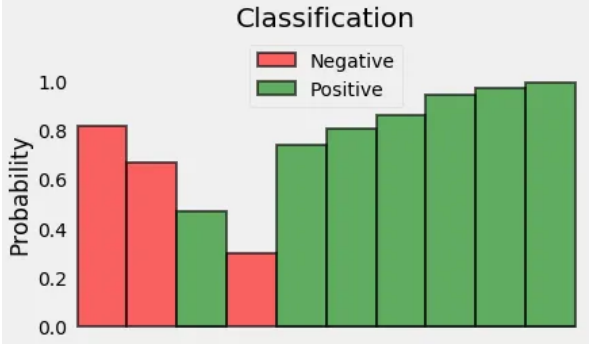



Figure 6.18: Probabilities of all points — repositioned bars.

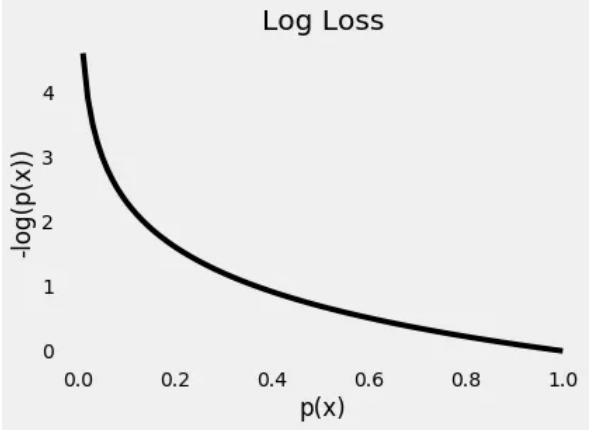



Figure 6.19: Log Loss for different probabilities.

It turns out, taking the (negative) log of the probability suits us well enough for this purpose. Since the log of values between 0.0 and 1.0 is negative, we take the negative log to obtain a positive value for the loss. Actually, the reason we use log for this comes from the definition of cross-entropy.

The plot in Figure 6.19 gives us a clear picture — as the predicted probability of the true class gets closer to zero, the loss increases exponentially. The negative log of the probabilities are the corresponding losses of each and every point.

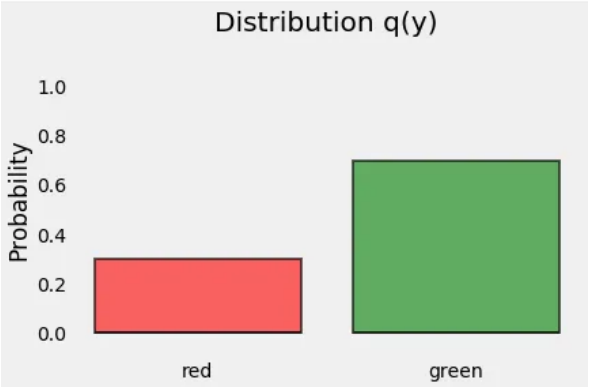



Figure 6.20: Distribution of red and green points.

### 6.4.4 Entropy and Cross-Entropy in Classification

This section delves into the mathematical underpinnings of entropy and cross-entropy within the context of classification tasks. These concepts are fundamental in understanding the uncertainty associated with probability distributions and their application in machine learning models.

### 6.4.5 Distribution Analysis

Consider a dataset consisting of two classes: red and green points. Let $q(y)$ represent the distribution of these classes, with $y$ denoting the class labels. The distribution $q(y)$ provides insight into the composition of the dataset, where the uncertainty associated with the distribution can be measured using entropy. We have 3 red points and 7 green points. This is what its distribution, let's call it $q(y)$, looks like as shown in Figure 6.20.

### 6.4.6 Entropy Computation

Entropy serves as a metric for quantifying the uncertainty within a distribution. For a binary classification scenario, where there are two classes (e.g., red and green), the entropy of the distribution $q(y)$ is calculated as:

$$\text{Entropy} = -\sum_{i=1}^{C} p_i \log(p_i) \tag{6.23}$$

Here, $p_i$ represents the probability of class $i$ and $C$ denotes the number of classes. A lower entropy value indicates reduced uncertainty, such as when all points belong to a single class.

What if all our points were green? What would be the uncertainty of that distribution? It will be zero. After all, there would be no doubt about the color of a point. It is always green. So, entropy is zero.

On the other hand, what if we knew exactly half of the points were green and the other half, red? That's the worst case scenario. We would have absolutely no edge on guessing the color of a point: it is totally random.

### 6.4.7 Cross-Entropy Formulation

In classification tasks, it is essential to compare the true distribution $q(y)$ with an estimated distribution $p(y)$. Cross-entropy serves this purpose, measuring the dissimilarity between the two distributions. Mathematically, cross-entropy is expressed as:

$$H(p, q) = -\sum_{i=1}^{C} p_i \log(q_i) \tag{6.24}$$

Where $p(y)$ and $q(y)$ denote the estimated and true distributions, respectively. A smaller cross-entropy value signifies greater similarity between the distributions.

# Chapter 7

# Introduction to PyTorch

PyTorch is a machine learning library based on the Torch library, used for applications such as computer vision and natural language processing, originally developed by Meta AI and now part of the Linux Foundation umbrella. It is recognized as one of the two most popular machine learning libraries alongside TensorFlow, offering free and open-source software released under the modified BSD license. Although the Python interface is more polished and the primary focus of development, PyTorch also has a C++ interface.

A number of pieces of deep learning software are built on top of PyTorch, including Tesla Autopilot, Uber's Pyro, Hugging Face's Transformers, PyTorch Lightning, and Catalyst.

PyTorch provides two high-level features:

i. Tensor computing (like NumPy) with strong acceleration via graphics processing units (GPU).

ii. Deep neural networks built on a tape-based automatic differentiation system.

## 7.1 Tensors

Tensors are a specialized data structure that are very similar to arrays and matrices. In PyTorch, we use tensors to encode the inputs and outputs of a model, as well as the model's parameters.

Stated simply, tensors are mathematical objects that can be used to describe physical properties, just like scalars and vectors. In fact, tensors are merely a generalization of scalars and vectors. A scalar is a zero-rank tensor, and a vector is a first-rank tensor.

A tensor can be described as an $n$-dimensional numerical array. A tensor can be called a generalized matrix. It could be a 0-D matrix (a single number), 1-D matrix (a vector), 2-D matrix, or any higher-dimensional structure. A tensor is identified by three parameters viz., rank, shape, and size. The number of dimensions of the tensor is said to be its rank. The number of columns and rows that the tensor has is said to be its shape. And, the data type assigned to the tensor's elements is said to be its type.

### 7.1.1 Hands On

#### 7.1.1.1 Creating Tensors in PyTorch

We are going to use various available methods to create tensors in PyTorch.

## 7.2 Automatic Differentiation and Autograd in PyTorch

Automatic Differentiation and Autograd are powerful tools in PyTorch that can be used to train deep learning models.

Graphs are used to represent the computation of a model, while Automatic Differentiation and Autograd allow the model to learn by updating its parameters during training.

In PyTorch, a graph is represented by a directed acyclic graph (DAG), where each node represents a computation, and the edges represent the flow of data between computations. The graph is used to track the dependencies between computations and to compute gradients during the backpropagation step.

### 7.2.1 Automatic Differentiation

Automatic Differentiation (AD) is a technique that allows the model to compute gradients automatically. In PyTorch, AD is implemented through the Autograd library, which uses the graph structure to compute gradients. AD allows the model to learn by updating its parameters during training, without the need for manual computation of gradients.

### 7.2.2 Autograd

Autograd is a PyTorch library that implements Automatic Differentiation. It uses the graph structure to compute gradients and allows the model to learn by updating its parameters during training. Autograd also provides a way to compute gradients with respect to arbitrary scalar values, which is useful for tasks such as optimization.

In neural networks, backpropagation is performed to optimize the parameters in order to minimize the error in predictions. To facilitate this process, PyTorch offers `torch.autograd`, which provides automatic differentiation by collecting gradients. Autograd achieves this by maintaining a record of data (tensors) and all executed operations in the directed acyclic graph (DAG) mentioned above consisting of function objects. In this DAG, the leaves are the input tensors, and the roots are the output tensors. By tracing this graph from roots to leaves, gradients can be automatically computed using the chain rule. See Figure 7.1.

### 7.2.3 Using Graphs, Automatic Differentiation, and Autograd in PyTorch

The steps involved in using Graphs, Automatic Differentiation, and Autograd in PyTorch are as follows:

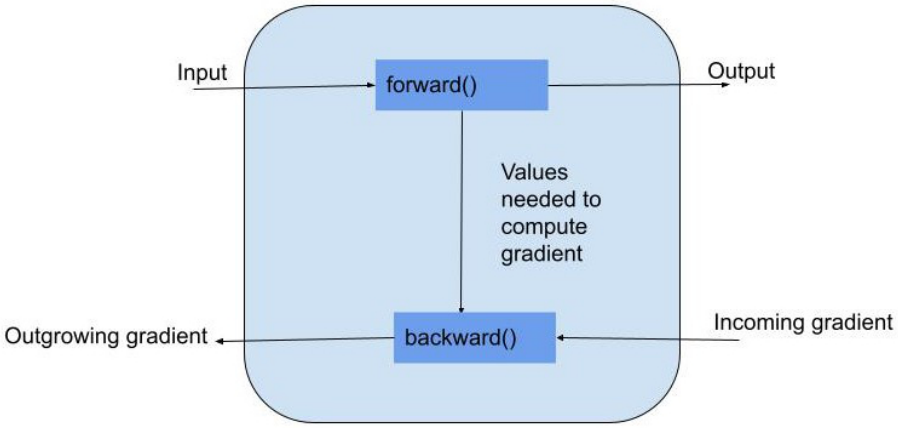



Figure 7.1: The directed acyclic graph consisting of function objects.

1. Define the graph structure: The first step in using these concepts in PyTorch is to define the graph structure of the model. This can be done by creating tensors and operations on them.
2. Enable Autograd: Once the graph structure is defined, Autograd needs to be enabled on the tensors that require gradients. This can be done by setting the `requires_grad` attribute to True.
3. Forward pass: After the graph structure is defined and Autograd is enabled, the forward pass can be performed. This involves computing the output of the model given an input.
4. Backward pass: Once the forward pass is complete, the backward pass can be performed to compute the gradients. This is done by calling the `backward()` method on the output tensor.
5. Update parameters: Finally, the gradients can be used to update the parameters of the model using an optimizer.

### 7.2.4 Hands On

#### 7.2.4.1 Differentiation of an Algebraic Equation

#### 7.2.4.2 Differentiation of Another Algebraic Equation

#### 7.2.4.3 Implementation of a Simple Linear Regression Model

#### 7.2.4.4 A Simple Neural Network with a Single Hidden Layer

# Chapter 8

# Introduction to Deep Learning

"Intro to Deep Learning" available at https://www.kaggle.com/learn/intro-to-deep-learning will be used as the course material.

# Chapter 9

# Convolutional Neural Networks (CNNs)[1]

A Convolutional Neural Network (CNN or ConvNet) is a type of deep learning neural network architecture commonly used in computer vision. Computer vision is a field of artificial intelligence that enables a computer to understand and interpret the image or visual data.

Image data is represented as a two-dimensional grid of pixels. Each pixel corresponds to one or multiple numerical values, depending on whether the image is monochromatic or in color, respectively. Traditionally, images have been treated as vectors of numbers by flattening them, ignoring the spatial relation between pixels. This approach, while necessary for feeding images through fully connected Multi Layer Perceptrons (MLPs), disregards the rich structure inherent in images.

See one example in Figure 9.1 where a $32 \times 32 \times 3$ image is stretched/flattened to $3072 \times 1$.

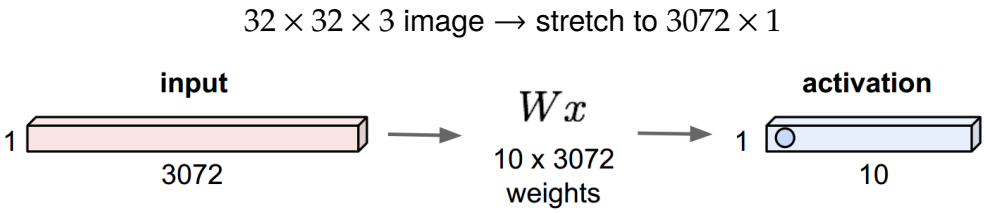



Figure 9.1: Flattening of image data.

Fully connected MLPs are invariant to the order of features, meaning similar results can be obtained regardless of whether the spatial structure of pixels is preserved or not. However, leveraging the knowledge that nearby pixels are typically related to each other can lead to more efficient models for learning from image data.

[1] A major part of this chapter has been adapted from I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016, `http://www.deeplearningbook.org` and A. Zhang, Z. C. Lipton, M. Li, *et al.*, *Dive into Deep Learning*. Cambridge University Press, 2023, `https://d2l.ai`. The course website *CS231n: Deep Learning for Computer Vision*, `http://cs231n.stanford.edu/`, course offered by Stanford University, Spring - 2023 has also been heavily used. The example on hospital analogy has been adapted from K. Azad, *Intuitive Guide to Convolution*, `https://betterexplained.com/articles/intuitive-convolution/`, Better Explained.

**CIFAR-10 Dataset**

The CIFAR-10 (Canadian Institute For Advanced Research) dataset is a collection of images that are commonly used to train machine learning and computer vision algorithms. It is one of the most widely used datasets for machine learning research. The CIFAR-10 dataset contains 60,000 32x32 color images in 10 different classes. The 10 different classes represent airplanes, cars, birds, cats, deer, dogs, frogs, horses, ships, and trucks. There are 6,000 images of each class.

Regular neural networks do not scale well to full images. In CIFAR-10, images are only of size $32 \times 32 \times 3$ (32 wide, 32 high, 3 color channels). Consequently, a single fully-connected neuron in the first hidden layer of a regular neural network would entail $32 \times 32 \times 3 = 3072$ weights. While this number may seem manageable, the fully-connected structure does not scale to larger images. For instance, an image of a more substantial size, such as $200 \times 200 \times 3$, would result in neurons with $200 \times 200 \times 3 = 120,000$ weights. Additionally, multiple neurons would be desirable, leading to a rapid accumulation of parameters. Clearly, this full connectivity is wasteful, and the substantial number of parameters would quickly result in overfitting.

CNNs address this issue by explicitly taking into account the spatial structure of images. CNNs are a powerful family of neural networks designed for image processing tasks and have become ubiquitous in computer vision. They were instrumental in significant performance improvements on datasets like ImageNet.

CNNs exhibit similarities to ordinary neural networks. Like conventional neural networks, CNNs consist of neurons with learnable weights and biases. Each neuron receives inputs, performs a dot product, and optionally applies a non-linearity. The network as a whole represents a single differentiable score function, transforming raw image pixels into class scores. Furthermore, CNNs employ a loss function on the last (fully-connected) layer, and all the strategies developed for learning regular neural networks remain applicable.

However, the key distinction lies in the assumption made by ConvNet architectures that the inputs are images. This allows specific properties to be encoded into the architecture, resulting in a more efficient implementation of the forward function and a significant reduction in the number of parameters in the network.

Modern CNNs draw inspiration from biology, group theory, and experimentation. They are computationally efficient due to their fewer parameters compared to fully connected architectures and the ease of parallelization convolutions across GPU cores. As a result, CNNs are widely used and have even become competitive in tasks traditionally dominated by recurrent neural networks, such as audio, text, and time series analysis. CNNs have also been adapted for graph-structured data and recommender systems.

The human brain efficiently processes a vast amount of information upon perceiving an image. Neurons within the brain operate within their respective receptive fields and

**ImageNet** IMAGENET

ImageNet is an image database organized according to the WordNet hierarchy (currently only the nouns), in which each node of the hierarchy is depicted by hundreds and thousands of images. The project has been instrumental in advancing computer vision and deep learning research. The data is available for free to researchers for non-commercial use.

establish connections with neighboring neurons, collectively covering the entire visual field. Analogously, in CNNs, each neuron processes data within its receptive field.

CNN layers are organized hierarchically to detect progressively more complex patterns, starting from simpler ones such as lines and curves, and advancing to more intricate patterns like faces and objects. Leveraging CNNs facilitates the replication of visual perception capabilities in computer systems, effectively enabling computer vision applications.

## 9.1 The Convolution Operation

The name "Convolutional Neural Network" indicates that the network employs a mathematical operation called convolution. Convolution is a specialized kind of linear operation. Convolutional networks are simply neural networks that use convolution in place of general matrix multiplication in at least one of their layers.

### 9.1.1 An Intuitive Example — Hospital Analogy

Consider managing a hospital treating patients with a single disease, where:

- A treatment plan dictates that every patient receives 3 units of the cure on their first day.
- A list of patients indicates the patient count for each day of the week (e.g., 1 person on Monday, 2 people on Tuesday, and so forth).

The question arises: How much medicine is used each day? This can be determined through a straightforward multiplication: Plan $\times$ Patients = Daily Usage.

| Plan | $\times$ | Patients | = | Daily Usage |
|---|---|---|---|---|
| [3] | $\times$ | [1, 2, 3, 4, 5] | = | [3, 6, 9, 12, 15] |

The multiplication of the plan by the patient list yields the usage for each day: $[3, 6, 9, 12, 15]$. Each multiplication operation corresponds to using the plan with a single day's worth of patients, for instance: $[3] \times [4] = [12]$.

Suppose the disease evolves, necessitating a multi-day treatment regimen. In response, a new plan is devised:

Plan: $[3, 2, 1]$.

This plan dictates 3 units of the cure on the first day, 2 units on the second day, and 1 unit on the third day. Given the same patient schedule $[1, 2, 3, 4, 5]$, what is the medicine usage each day?

The process is no longer as straightforward as simple multiplication:

- On Monday, a single patient arrives, receiving 3 units on her first day.
- On Tuesday, the patient from Monday receives 2 units (her second day), while 2 new patients arrive, each receiving 3 units ($2 \times 3 = 6$). Consequently, the total usage is $2 + (2 \times 3) = 8$ units.
- On Wednesday, the complexity increases: the patient from Monday completes treatment (1 unit, her last day), while the 2 patients from Tuesday receive 2 units each ($2 \times 2$), and three new patients arrive.

Given the overlapping nature of patient arrivals, organizing this calculation becomes challenging. An approach is to visualize the patient list flipped, such that the first patient is on the right:

$$5,\ 4,\ 3,\ 2,\ 1.$$

Next, envision three separate rooms, each assigned a corresponding dose, Rooms: 3, 2, 1.

On the first day, patients receive 3 units of medicine in the first room. The following day, they proceed to the second room, receiving 2 units, and on the last day, they enter the third room, receiving 1 unit. After this, treatment concludes.

To calculate the total medicine usage, align the patients and guide them through the rooms:

**Monday (Day 1)**

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Rooms | | | | | 3 | 2 | 1 | | |
| Patients | 5 | 4 | 3 | 2 | 1 | | | | |
| Usage | | | | | 3 | | | = | 3 |

On Monday (the first day), a single patient occupies the first room, receiving 3 units of medicine.

**Tuesday (Day 2)**

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Rooms | | | | 3 | 2 | 1 | | | |
| Patients | 5 | 4 | 3 | 2 | 1 | | | | |
| Usage | | | | 6 | 2 | | | = | 8 |

On Tuesday, each patient advances to the next room. The first patient enters the second room, while 2 new patients occupy the first room. The total usage is calculated by combining the doses from each room.

This process continues for subsequent days:

**Wednesday (Day 3)**

| | | | | | | |
|---|---|---|---|---|---|---|
| Rooms | | | 3 | 2 | 1 | |
| Patients | 5 | 4 | 3 | 2 | 1 | |
| Usage | | | 9 | 4 | 1 | = 14 |

**Thursday (Day 4)**

| | | | | | | |
|---|---|---|---|---|---|---|
| Rooms | | 3 | 2 | 1 | | |
| Patients | 5 | 4 | 3 | 2 | 1 | |
| Usage | | | 12 | 6 | 2 | = 20 |

**Friday (Day 5)**

| | | | | | | |
|---|---|---|---|---|---|---|
| Rooms | 3 | 2 | 1 | | | |
| Patients | 5 | 4 | 3 | 2 | 1 | |
| Usage | 15 | 8 | 3 | | | = 26 |

The total daily usage is found by *convolving* the plan and the patient list, representing a sophisticated multiplication operation between a list of input numbers and a program.

### 9.1.2 Convolution Operation — A Visual Demonstration

We view the video available at this link, "But what is a convolution?" (`https://youtu.be/KuXjwB4LzSA`).

### 9.1.3 Mathematical Relations — Tracking the Location of a Spaceship

Suppose we are tracking the location of a spaceship with a laser sensor. Our laser sensor provides a single output $x(t)$, the position $x$ of the spaceship at time $t$. Both $x$ and $t$ are real-valued, meaning we can obtain a different reading from the laser sensor at any instant in time.

Now suppose that our laser sensor is somewhat noisy. To obtain a less noisy estimate of the spaceship's position, we would like to average several measurements. Of course, more recent measurements are more relevant, so we will want this to be a weighted average that gives more weight to recent measurements. We can do this with a weighting function $w(a)$, where $a$ is the age of a measurement. If we apply such a weighted average operation at every moment, we obtain a new function $s$ providing a smoothed estimate of the position of the spaceship:

$$s(t) = \int x(a)w(t-a)da. \tag{9.1}$$

This operation is called convolution. The convolution operation is typically denoted with an asterisk:

$$s(t) = (x * w)(t). \tag{9.2}$$

In our example, $w$ needs to be a valid probability density function, or the output will not be a weighted average. Also, $w$ needs to be 0 for all negative arguments, or it will look into the future, which is presumably beyond our capabilities. These limitations are particular to our example, though. In general, convolution is defined for any functions for which the above integral is defined and may be used for other purposes besides taking weighted averages.

In convolutional network terminology, the first argument (in this example, the function $x$) to the convolution is often referred to as the input, and the second argument (in this example, the function $w$) as the kernel. The output is sometimes referred to as the feature map.

In our example, the idea of a laser sensor that can provide measurements at every instant is not realistic. Usually, when we work with data on a computer, time will be discretized, and our sensor will provide data at regular intervals. In our example, it might be more realistic to assume that our laser provides a measurement once per second. The time index $t$ can then take on only integer values. If we now assume that $x$ and $w$ are defined only on integer $t$, we can define the discrete convolution:

$$\begin{aligned} s(t) &= (x * w)(t) \\ &= \sum_{a=-\infty}^{\infty} x(a)w(t-a). \end{aligned} \tag{9.3}$$

In machine learning applications, the input is usually a multidimensional array of data, and the kernel is usually a multidimensional array of parameters that are adapted by the learning algorithm. We will refer to these multidimensional arrays as tensors. Because each element of the input and kernel must be explicitly stored separately, we usually assume that these functions are zero everywhere but in the finite set of points for which we store the values. This means that in practice, we can implement the infinite summation as a summation over a finite number of array elements.

Finally, we often use convolutions over more than one axis at a time. For example, if we use a two-dimensional image $I$ as our input, we probably also want to use a two-dimensional kernel $K$:

$$\begin{aligned} S(i,j) &= (I * K)(i,j) \\ &= \sum_m \sum_n I(m,n)K(i-m, j-n). \end{aligned} \tag{9.4}$$

Convolution is commutative, meaning we can equivalently write

$$\begin{aligned} S(i,j) &= (K * I)(i,j) \\ &= \sum_m \sum_n I(i-m, j-n)K(m,n). \end{aligned} \tag{9.5}$$

Usually the latter formula is more straightforward to implement in a machine learning library, because there is less variation in the range of valid values of $m$ and $n$.

The commutative property of convolution arises because we have flipped the kernel relative to the input, in the sense that as $m$ increases, the index into the input increases, but the index into the kernel decreases. The only reason to flip the kernel is to obtain the commutative property. While the commutative property is useful for writing proofs, it is not usually an important property of a neural network implementation. Instead, many neural network libraries implement a related function called the cross-correlation, which is the same as convolution but without flipping the kernel:

$$S(i,j) = (K * I)(i,j) = \sum_m \sum_n I(i+m, j+n) K(m,n). \tag{9.6}$$

Many machine learning libraries implement cross-correlation but call it convolution. In this text we follow this convention of calling both operations convolution and specify whether we mean to flip the kernel or not in contexts where kernel flipping is relevant. In the context of machine learning, the learning algorithm will learn the appropriate values of the kernel in the appropriate place, so an algorithm based on convolution with kernel flipping will learn a kernel that is flipped relative to the kernel learned by an algorithm without the flipping. It is also rare for convolution to be used alone in machine learning; instead convolution is used simultaneously with other functions, and the combination of these functions does not commute regardless of whether the convolution operation flips its kernel or not.

### 9.1.4 An Example of 2-D Convolution

See Figure 9.2 for an example of convolution (without kernel flipping) applied to a 2-D tensor. Here, we restrict the output to only positions where the kernel lies entirely within the image, called "valid" convolution in some contexts. We draw boxes with arrows to indicate how the upper-left element of the output tensor is formed by applying the kernel to the corresponding upper-left region of the input tensor.

## 9.2 CNN Architecture Overview

As we already know, neural networks receive an input (a single vector) and transform it through a series of hidden layers. Each hidden layer comprises a set of neurons, where each neuron is fully connected to all neurons in the preceding layer. Neurons within a single layer function independently and do not share any connections. The final fully-connected layer is referred to as the "output layer," and in classification settings, it represents the class scores.

Convolutional neural networks exploit the fact that the input comprises images and constrain the architecture in a more practical manner. Specifically, unlike a regular neural network, the layers of a ConvNet feature neurons arranged in three dimensions: width, height, and depth. Here, "depth" refers to the third dimension of an activation volume, not the depth of a full neural network, which can denote the total number of layers in a network.

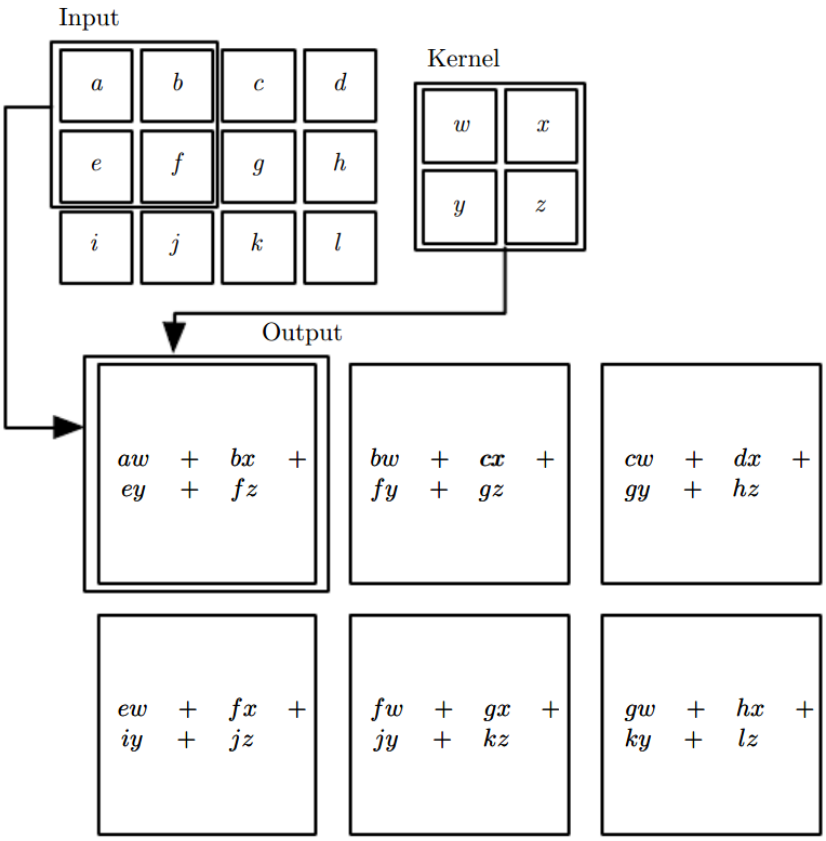


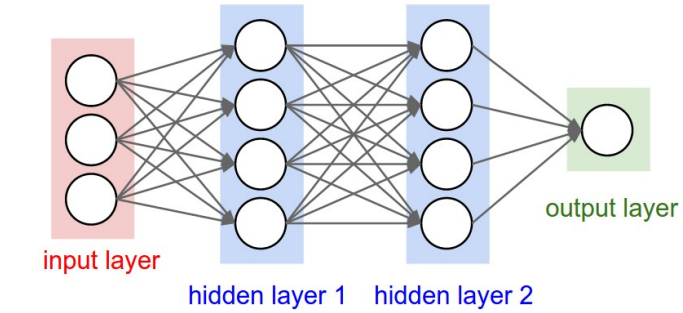

Figure 9.2: An example of 2-D convolution without kernel flipping.

For instance, the input images in CIFAR-10 form an input volume of activations, with dimensions $32 \times 32 \times 3$ (width, height, and depth, respectively). As we will soon observe, neurons in a layer are only connected to a small region of the preceding layer, rather than all neurons in a fully-connected manner. Furthermore, the final output layer for CIFAR-10 would have dimensions $1 \times 1 \times 10$. By the end of the ConvNet architecture, we reduce the full image into a single vector of class scores arranged along the depth dimension.

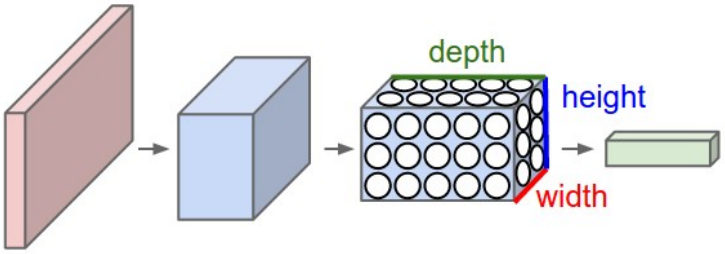
We can see an architecture visualization in Figure 9.3. In Figure 9.3(a), a regular 3-layer neural network is shown. In Figure 9.3(b), a ConvNet arranges its neurons in three dimensions (width, height, depth), as visualized in one of the layers. Every layer of a ConvNet transforms the 3D input volume to a 3D output volume of neuron activations. In this example, the red input layer holds the image, so its width and height would be the dimensions of the image, and the depth would be 3 (Red, Green, Blue channels).

In an example shown in Figure 9.4 the convolution Layer preserves the spatial structure of a $32 \times 32 \times 3$ image. The $5 \times 5 \times 3$ filter is convolved with the $32 \times 32 \times 3$ image, i.e., it slides over the image spatially, computing dot products. The result of taking a dot product between the filter and a small $5 \times 5 \times 3$ chunk of the image (i.e., a 75-dimensional dot product plus bias, $w^T x + b$) is one number.

During convolution, we can have multiple filters. See Figure 9.5. Consider a second, green filter. We convolve (slide) over all spatial locations. If we had six $5 \times 5$ filters, we'll get six

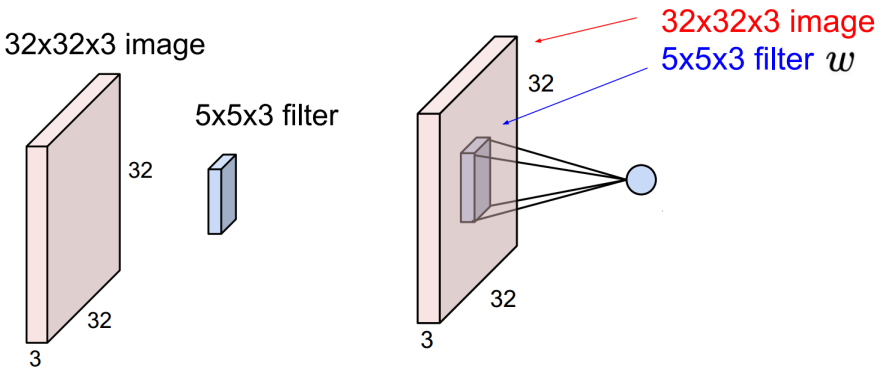



(a)




(b)

Figure 9.3: Visualization of ConvNet architecture against a regular network.




(a) (b)

Figure 9.4: The convolution layer preserves the spatial structure.

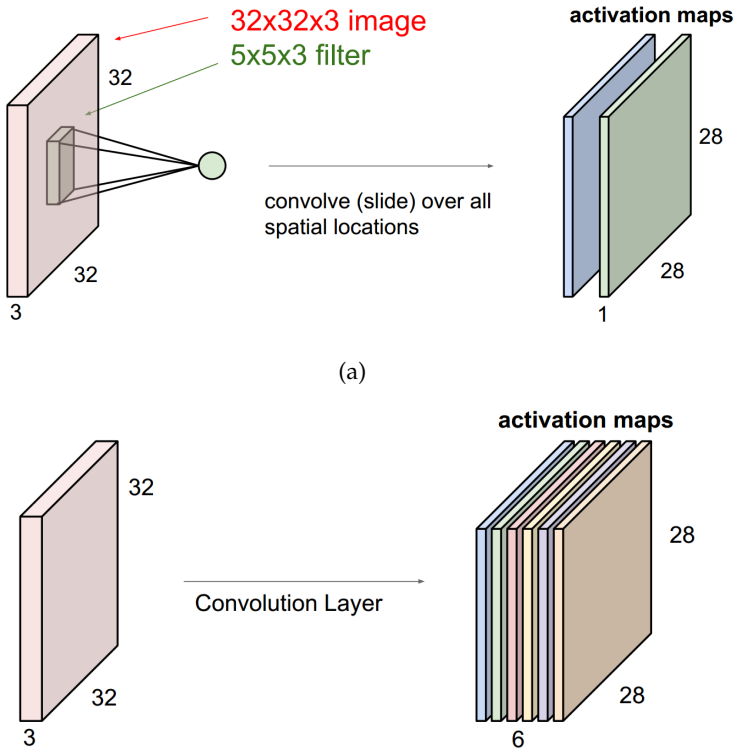



Figure 9.5: Multiple filters in convolution Layer.

separate activation maps. We stack these up to get a "new image" of size $28 \times 28 \times 6$.

## 9.3 Layers Used to Build ConvNets

As we described in Section 9.2, a simple ConvNet is a sequence of layers, and every layer of a ConvNet transforms one volume of activations to another through a differentiable function. We use three main types of layers to build ConvNet architectures: Convolutional Layer, Pooling Layer, and Fully-Connected Layer (exactly as seen in regular Neural Networks). We will stack these layers to form a full ConvNet architecture.

### 9.3.1 Example Architecture

We will go into more details below, but a simple ConvNet for CIFAR-10 classification could have the architecture [INPUT - CONV - RELU - POOL - FC]. An example visualization architecture of the network can be seen in Figure 9.6.

We need to note the following:

- The **INPUT** layer $[32 \times 32 \times 3]$ will hold the raw pixel values of the image, where the image has a width of 32, a height of 32, and consists of three color channels (R, G, B).
- The **CONV** layer will compute the output of neurons that are connected to local regions in the input, each computing a dot product between their weights and a small region

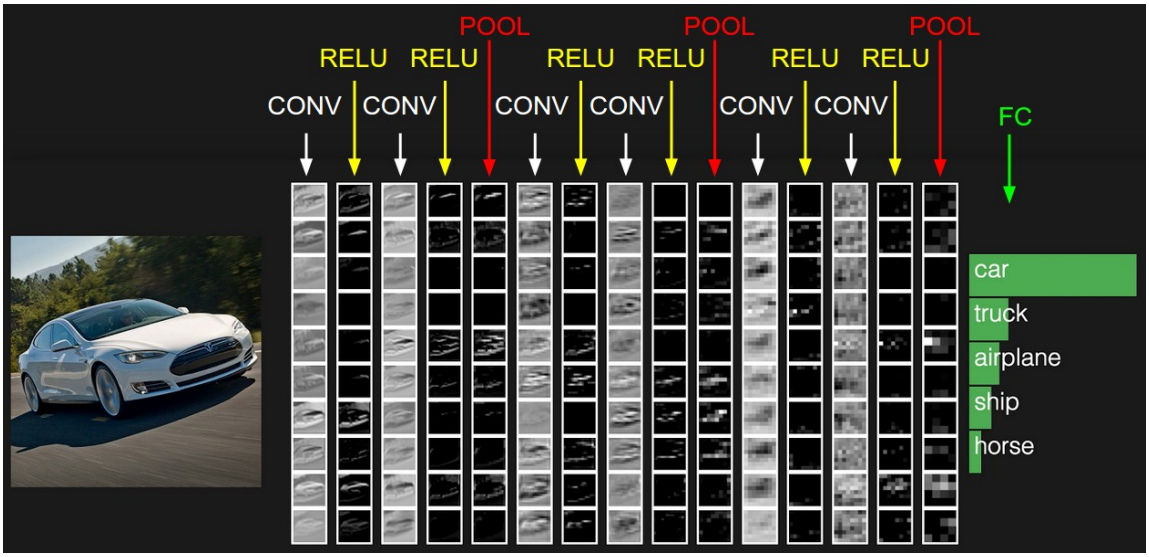



Figure 9.6: The activations of an example ConvNet architecture.

they are connected to in the input volume. If we decide to use 12 filters, this will result in a volume of $[32 \times 32 \times 12]$.

- The **RELU** layer will apply an element-wise activation function, such as the $\max(0, x)$ thresholding at zero. This leaves the size of the volume unchanged ($[32 \times 32 \times 12]$).
- The **POOL** layer will perform a downsampling operation along the spatial dimensions (width, height), resulting in a volume such as $[16 \times 16 \times 12]$.
- The **FC** (i.e., fully-connected) layer will compute the class scores, resulting in a volume of size $[1 \times 1 \times 10]$, where each of the 10 numbers corresponds to a class score, such as among the 10 categories of CIFAR-10. As with ordinary Neural Networks, and as the name implies, each neuron in this layer will be connected to all the numbers in the previous volume.

In this way, ConvNets transform the original image layer by layer from the original pixel values to the final class scores. Note that some layers contain parameters and others don't. In particular, the CONV/FC layers perform transformations that are a function of not only the activations in the input volume but also of the parameters (the weights and biases of the neurons). On the other hand, the RELU/POOL layers will implement a fixed function. The parameters in the CONV/FC layers will be trained with gradient descent so that the class scores that the ConvNet computes are consistent with the labels in the training set for each image.

In the architecture shown in Figure 9.6, the initial volume stores the raw image pixels (left) and the last volume stores the class scores (right). Each volume of activations along the processing path is shown as a column. Since it's difficult to visualize 3D volumes, we lay out each volume's slices in rows. The last layer volume holds the scores for each class, but here we only visualize the sorted top 5 scores, and print the labels of each one.

The full web-based demo of the interface can be found at this site, http://cs231n.stanford.edu/.

## 9.4 Convolutional Layer

The Convolutional layer serves as the fundamental building block of a convolutional network, shouldering the majority of the computational workload.

### 9.4.1 Overview and Intuition

Let's first discuss what the CONV layer computes without brain/neuron analogies. The CONV layer's parameters consist of a set of learnable filters. Each filter is small spatially (along width and height) but extends through the full depth of the input volume. For example, a typical filter on the first layer of a ConvNet might have a size of $5 \times 5 \times 3$ (i.e., 5 pixels width and height, and 3 because images have depth 3, representing the color channels).

During the forward pass, we slide (more precisely, convolve) each filter across the width and height of the input volume and compute dot products between the entries of the filter and the input at any position. As we slide the filter over the width and height of the input volume, we produce a 2-dimensional activation map that gives the responses of that filter at every spatial position.

Intuitively, the network will learn filters that activate when they detect some type of visual feature such as an edge of some orientation or a blotch of some color on the first layer, or eventually entire honeycomb or wheel-like patterns on higher layers of the network. We will have an entire set of filters in each CONV layer (e.g., 12 filters), and each of them will produce a separate 2-dimensional activation map. We stack these activation maps along the depth dimension to produce the output volume.

In light of the brain/neuron analogies, every entry in the $3D$ output volume can also be interpreted as the output of a neuron that looks at only a small region in the input and shares parameters with all neurons to the left and right spatially (since these numbers all result from applying the same filter).

### 9.4.2 Local Connectivity

When dealing with high-dimensional inputs such as images, it is impractical to connect neurons to all neurons in the previous volume. Instead, we connect each neuron to only a local region of the input volume. The spatial extent of this connectivity is a hyperparameter called the receptive field of the neuron (equivalently, this is the filter size). The extent of the connectivity along the depth axis is always equal to the depth of the input volume. The connections are local in $2D$ space (along width and height) but always full along the entire depth of the input volume.

**Example 9.4.1.** Suppose the input volume has a size of $[32 \times 32 \times 3]$, such as an RGB CIFAR-10 image. If the receptive field (or the filter size) is $5 \times 5$, then each neuron in the Conv Layer will

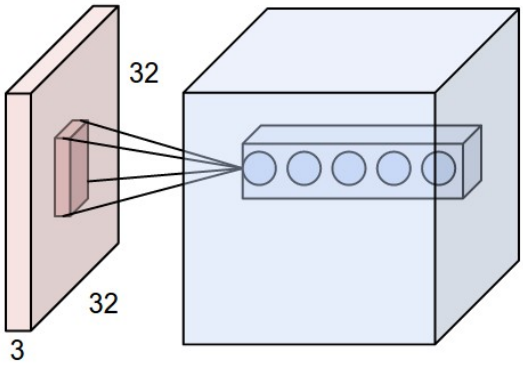



Figure 9.7: An example input volume in red (e.g., a $32 \times 32 \times 3$ CIFAR-10 image), and an example volume of neurons in the first convolutional layer.

have weights to a $[5 \times 5 \times 3]$ region in the input volume, for a total of $5 \times 5 \times 3 = 75$ weights (and +1 bias parameter).

See Figure 9.7 for an example input volume in red (e.g., a $32 \times 32 \times 3$ CIFAR-10 image), and an example volume of neurons in the first convolutional layer. Each neuron in the convolutional layer is connected only to a local region in the input volume spatially, but to the full depth (i.e., all color channels). Note, there are multiple neurons (5 in this example) along the depth, all looking at the same region in the input: the lines that connect this column of 5 neurons do not represent the weights (i.e., these 5 neurons do not share the same weights, because they are associated with 5 different filters), they just indicate that these neurons are connected to or looking at the same receptive field or region of the input volume, i.e., they share the same receptive field but not the same weights.

**Example 9.4.2.** Suppose an input volume has a size of $[16 \times 16 \times 20]$. Then, using an example receptive field size of $3 \times 3$, every neuron in the Conv layer would now have a total of $3 \times 3 \times 20 = 180$ connections to the input volume.

### 9.4.3 Spatial Arrangement

Three hyperparameters control the size of the output volume: the depth, stride, and zero-padding.

i. The depth of the output volume corresponds to the number of filters we use, each learning to look for something different in the input.

ii. We must specify the stride with which we slide the filter. When the stride is 1, the filters move one pixel at a time. When the stride is 2 (or uncommonly 3 or more), the filters jump 2 pixels at a time as we slide them around.

iii. Sometimes we pad the input volume with zeros around the border. The size of this zero-padding is a hyperparameter.

We can compute the spatial size of the output volume as a function of the input volume size ($W$), the receptive field size of the Conv layer neurons ($F$), the stride with which they are

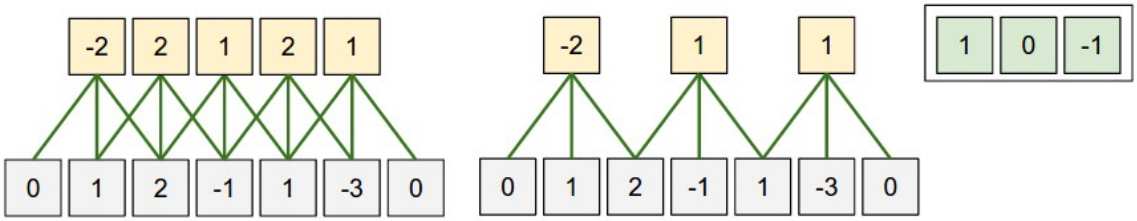



Figure 9.8: Illustration of spatial arrangement.

applied ($S$), and the amount of zero padding used ($P$) on the border. The correct formula for calculating how many neurons fit is given by $\frac{(W-F+2P)}{S}+1$.

We see one graphical example in Figure 9.8. In this example, there is only one spatial dimension ($x$-axis), one neuron with a receptive field size of $F = 3$, the input size is $W = 5$, and there is zero padding of $P = 1$. At the left, the neuron is strided across the input in a stride of $S = 1$, giving an output of size $(5 - 3 + 2)/1 + 1 = 5$. And at the right, the neuron uses a stride of $S = 2$, giving an output of size $(5 - 3 + 2)/2 + 1 = 3$. Notice that stride $S = 3$ could not be used since it wouldn't fit neatly across the volume. In terms of the equation, this can be determined since $(5 - 3 + 2) = 4$ is not divisible by 3. The neuron weights are in this example $[1, 0, -1]$ (shown on the very right), and its bias is zero. These weights are shared across all yellow neurons (see Section 9.5.2 (Parameter Sharing)).

### 9.4.4 Zero-padding

Zero padding is a technique commonly used to control the spatial size of the output volumes. By setting zero padding to be $P = (F - 1)/2$ when the stride is $S = 1$, we ensure that the input volume and output volume will have the same size spatially.

### 9.4.5 Constraints on Strides

The spatial arrangement hyperparameters have mutual constraints. For example, when the input has a size of $W = 10$, no zero padding is used ($P = 0$), and the filter size is $F = 3$, then it would be impossible to use stride $S = 2$, since $\frac{(W-F+2P)}{S}+1 = \frac{(10-3+0)}{2}+1 = 4.5$, which is not an integer, indicating that the neurons don't fit neatly and symmetrically across the input.

## 9.5 Leveraging Convolution — Key Concepts

Convolution leverages three important ideas that can help improve a machine learning system: sparse interactions, parameter sharing and equivariant representations. Moreover, convolution provides a means for working with inputs of variable size. We now describe each of these ideas in turn.

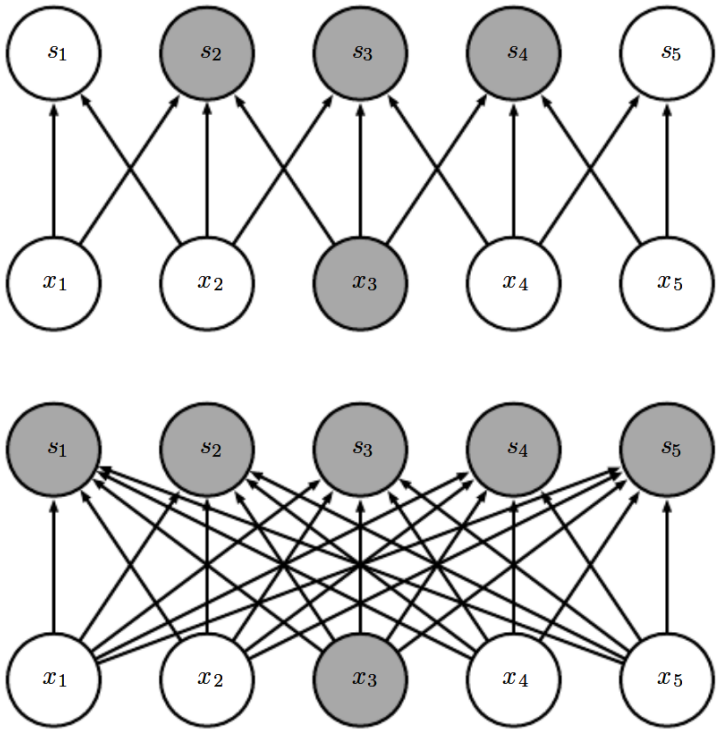



Figure 9.9: Sparse connectivity, viewed from below. (Top) $s$ is formed by convolution with a kernel of width 3. (Bottom) $s$ is formed by matrix multiplication.

### 9.5.1 Sparse Interactions

Traditional neural network layers use matrix multiplication by a matrix of parameters with a separate parameter describing the interaction between each input unit and each output unit. This means that every output unit interacts with every input unit. Convolutional networks, however, typically have sparse interactions (also referred to as sparse connectivity or sparse weights). This is accomplished by making the kernel smaller than the input. For example, when processing an image, the input image might have thousands or millions of pixels, but we can detect small, meaningful features such as edges with kernels that occupy only tens or hundreds of pixels. This means that we need to store fewer parameters, which both reduces the memory requirements of the model and improves its statistical efficiency. It also means that computing the output requires fewer operations.

These improvements in efficiency are usually quite large. If there are $m$ inputs and $n$ outputs, then matrix multiplication requires $m \times n$ parameters, and the algorithms used in practice have $O(m \times n)$ runtime (per example). If we limit the number of connections each output may have to $k$, then the sparsely connected approach requires only $k \times n$ parameters and $O(k \times n)$ runtime. For many practical applications, it is possible to obtain good performance on the machine learning task while keeping $k$ several orders of magnitude smaller than $m$.

For graphical demonstrations of sparse connectivity, see Figure 9.9 and Figure 9.10. In Figure 9.9, we highlight one input unit, $x_3$, and highlight the output units in $s$ that are affected by this unit. At the top, when $s$ is formed by convolution with a kernel of width 3, only

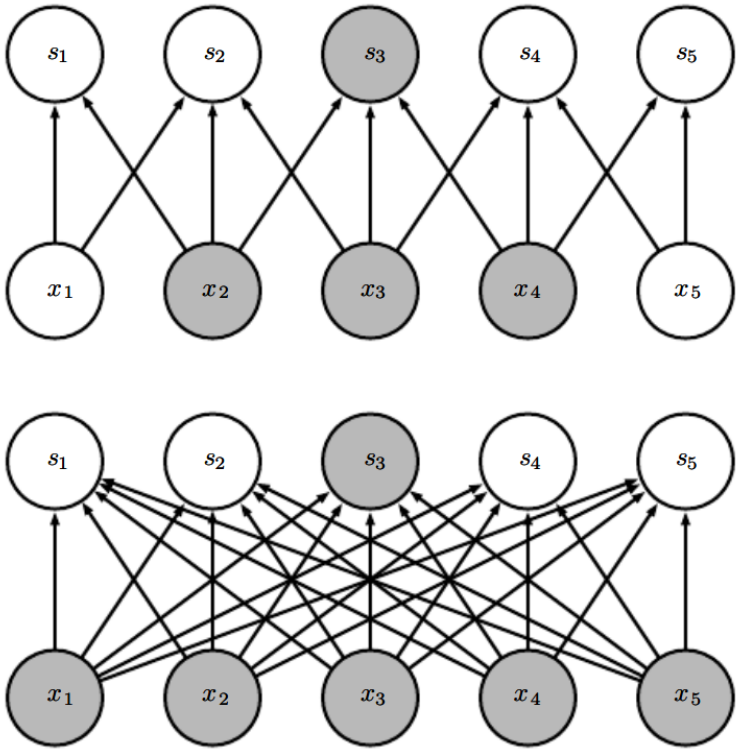



Figure 9.10: Sparse connectivity, viewed from above. (Top) $s$ is formed by convolution with a kernel of width 3. (Bottom) $s$ is formed by matrix multiplication.

three outputs are affected by $x_3$. At the bottom, when $s$ is formed by matrix multiplication, connectivity is no longer sparse, so all the outputs are affected by $x_3$. In Figure 9.10, we highlight one output unit, $s_3$, and highlight the input units in $x$ that affect this unit. These units are known as the receptive field of $s_3$. At the top, when $s$ is formed by convolution with a kernel of width 3, only three inputs affect $s_3$. At the bottom, when $s$ is formed by matrix multiplication, connectivity is no longer sparse, so all the inputs affect $s_3$.

In a deep convolutional network, units in the deeper layers may indirectly interact with a larger portion of the input, as shown in Figure 9.11. Here, the receptive field of the units in the deeper layers of a convolutional network is larger than the receptive field of the units in the shallow layers. This effect increases if the network includes architectural features like strided convolution (Section 9.4.3) or pooling (Section 9.10). This means that even though direct connections in a convolutional net are very sparse, units in the deeper layers can be indirectly connected to all or most of the input image.

This allows the network to efficiently describe complicated interactions between many variables by constructing such interactions from simple building blocks that each describe only sparse interactions.

### 9.5.2 Parameter Sharing

Parameter sharing refers to using the same parameter for more than one function in a model. In a traditional neural net, each element of the weight matrix is used exactly once when

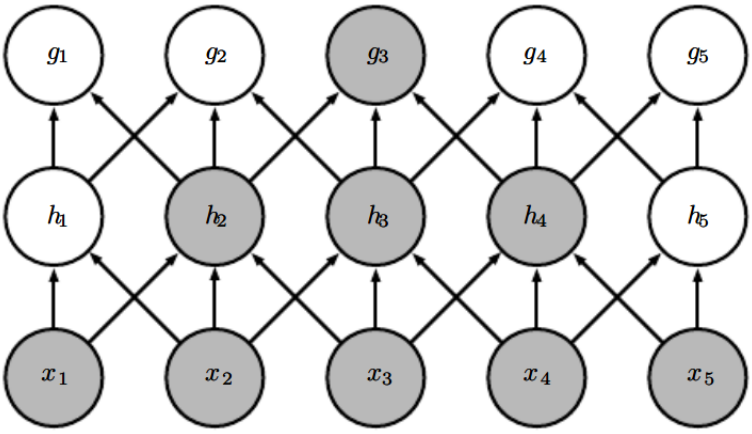



Figure 9.11: The receptive field of the units in the deeper layers of a convolutional network.

computing the output of a layer. It is multiplied by one element of the input and then never revisited.

Parameter sharing scheme is used in Convolutional Layers to control the number of parameters. With this scheme, the neurons in each depth slice use the same weights and bias. In practice, during backpropagation, every neuron in the volume computes the gradient for its weights, but these gradients are added up across each depth slice and only update a single set of weights per slice. This reduces the number of parameters significantly.

As a synonym for parameter sharing, one can say that a network has tied weights, because the value of the weight applied to one input is tied to the value of a weight applied elsewhere. In a convolutional neural net, each member of the kernel is used at every position of the input (except perhaps some of the boundary pixels, depending on the design decisions regarding the boundary). The parameter sharing used by the convolution operation means that rather than learning a separate set of parameters for every location, we learn only one set. This does not affect the runtime of forward propagation—it is still $O(k \times n)$—but it does further reduce the storage requirements of the model to $k$ parameters. Recall that $k$ is usually several orders of magnitude smaller than $m$. Since $m$ and $n$ are usually roughly the same size, $k$ is practically insignificant compared to $m \times n$. Convolution is thus dramatically more efficient than dense matrix multiplication in terms of the memory requirements and statistical efficiency.

As mentioned above, it turns out that we can dramatically reduce the number of parameters by making one reasonable assumption: That if one feature is useful to compute at some spatial position $(x_1, y_1)$, then it should also be useful to compute at a different position $(x_2, y_2)$. In other words, denoting a single 2-dimensional slice of depth as a depth slice (e.g., a volume of size $[55 \times 55 \times 96]$ has 96 depth slices, each of size $[55 \times 55]$), we are going to constrain the neurons in each depth slice to use the same weights and bias. With this parameter sharing scheme, the first Conv Layer in our example would now have only 96 unique sets of weights (one for each depth slice), for a total of $96 \times 11 \times 11 \times 3 = 34,848$ unique weights, or 34,944 parameters (+96 biases). Alternatively, all $55 \times 55$ neurons in each depth slice will now be using the same parameters. In practice, during backpropagation, every neuron in the volume

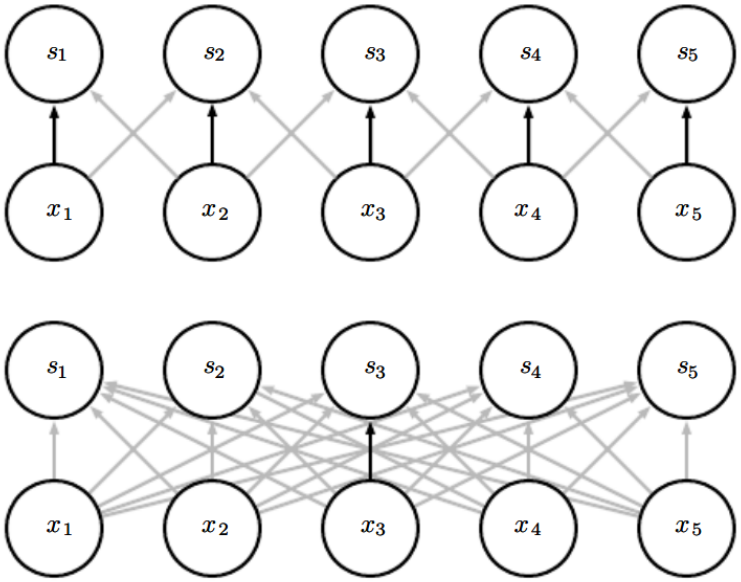



Figure 9.12: Parameter sharing.

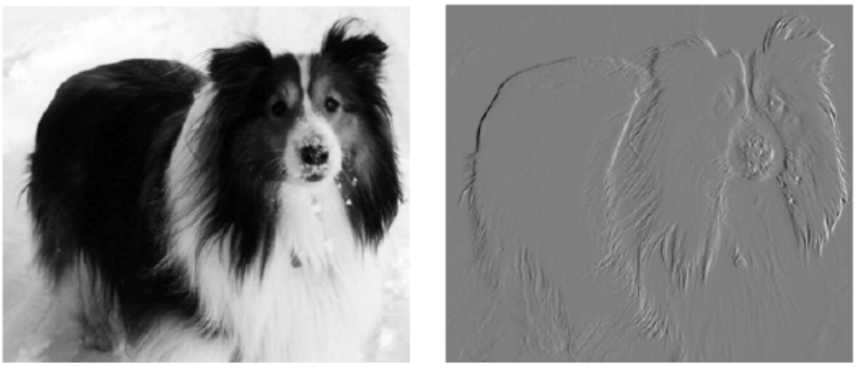


Figure 9.13: Efficiency of edge detection.

will compute the gradient for its weights, but these gradients will be added up across each depth slice and only update a single set of weights per slice.

For a graphical depiction of how parameter sharing works, see Figure 9.12. Here, black arrows indicate the connections that use a particular parameter in two different models. At the top, the black arrows indicate uses of the central element of a 3-element kernel in a convolutional model. Because of parameter sharing, this single parameter is used at all input locations. At the bottom, the single black arrow indicates the use of the central element of the weight matrix in a fully connected model. This model has no parameter sharing, so the parameter is used only once.

Notice that if all neurons in a single depth slice are using the same weight vector, then the forward pass of the CONV layer can in each depth slice be computed as a convolution of the neuron's weights with the input volume. This is why it is common to refer to the sets of weights as a filter (or a kernel), that is convolved with the input.

As an example of both of these first two principles in action, Figure 9.13 shows how sparse

connectivity and parameter sharing can dramatically improve the efficiency of a linear function for detecting edges in an image. The image on the right was formed by taking each pixel in the original image and subtracting the value of its neighboring pixel on the left. This shows the strength of all the vertically oriented edges in the input image, which can be a useful operation for object detection. Both images are 280 pixels tall. The input image is 320 pixels wide, while the output image is 319 pixels wide. This transformation can be described by a convolution kernel containing two elements, and requires $319 \times 280 \times 3 = 267,960$ floating-point operations (two multiplications and one addition per output pixel) to compute using convolution. To describe the same transformation with a matrix multiplication would take $320 \times 280 \times 319 \times 280$, or over eight billion, entries in the matrix, making convolution four billion times more efficient for representing this transformation. The straightforward matrix multiplication algorithm performs over sixteen billion floating-point operations, making convolution roughly 60,000 times more efficient computationally. Of course, most of the entries of the matrix would be zero. If we stored only the nonzero entries of the matrix, then both matrix multiplication and convolution would require the same number of floating-point operations to compute. The matrix would still need to contain $2 \times 319 \times 280 = 178,640$ entries. As such, convolution is an extremely efficient way.

Note that sometimes the parameter sharing assumption may not make sense. This is especially the case when the input images to a ConvNet have some specific centered structure, where we should expect, for example, that completely different features should be learned on one side of the image than another. One practical example is when the input is faces that have been centered in the image. If we are processing images that are cropped to be centered on an individual's face, we probably want to extract different features at different locations—the part of the network processing the top of the face needs to look for eyebrows, while the part of the network processing the bottom of the face needs to look for a chin. You might also expect that different eye-specific or hair-specific features could (and should) be learned in different spatial locations. In that case, it is common to relax the parameter sharing scheme, and instead simply call the layer a Locally-Connected Layer.

### 9.5.3 Equivariance to Translation

In the case of convolution, the particular form of parameter sharing causes the layer to have a property called equivariance to translation. To say a function is equivariant means that if the input changes, the output changes in the same way. Specifically, a function $f(x)$ is equivariant to a function $g$ if $f(g(x)) = g(f(x))$. In the case of convolution, if we let $g$ be any function that translates the input, that is, shifts it, then the convolution function is equivariant to $g$.

For example, let $I$ be a function giving image brightness at integer coordinates. Let $g$ be a function mapping one image function to another image function, such that $I' = g(I)$ is the image function with $I'(x, y) = I(x - 1, y)$. This shifts every pixel of $I$ one unit to the right. If we apply this transformation to $I$, then apply convolution, the result will be the same as if we applied convolution to $I'$, then applied the transformation $g$ to the output. When processing time-series data, this means that convolution produces a sort of timeline that shows when

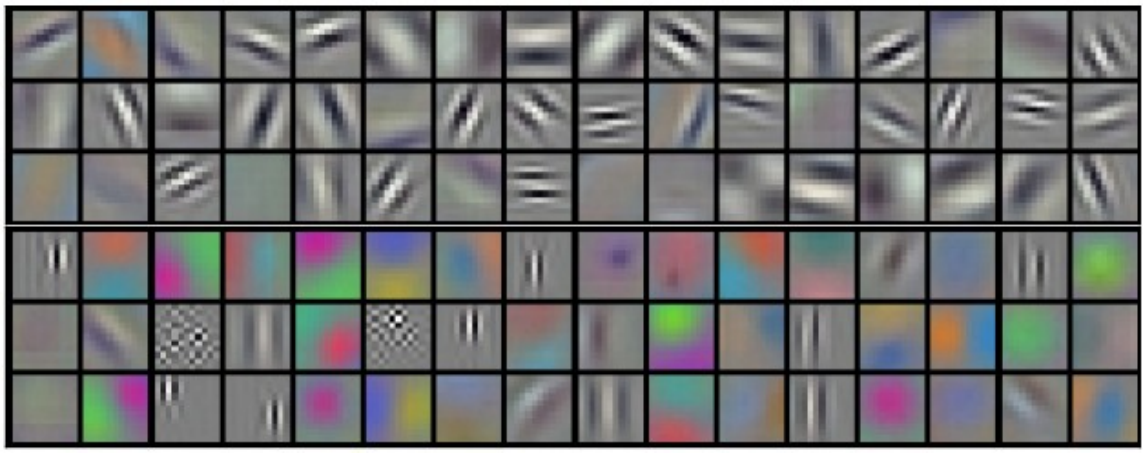


Figure 9.14: Example filters learned by A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems*, F. Pereira, C. Burges, L. Bottou, *et al.*, Eds., vol. 25, Curran Associates, Inc., 2012.

different features appear in the input. If we move an event later in time in the input, the exact same representation of it will appear in the output, just later.

Similarly with images, convolution creates a 2-D map of where certain features appear in the input. If we move the object in the input, its representation will move the same amount in the output. This is useful for when we know that some function of a small number of neighboring pixels is useful when applied to multiple input locations. For example, when processing images, it is useful to detect edges in the first layer of a convolutional network. The same edges appear more or less everywhere in the image, so it is practical to share parameters across the entire image.

Convolution is not naturally equivariant to some other transformations, such as changes in the scale or rotation of an image. Other mechanisms are necessary for handling these kinds of transformations.

## 9.6 Real-world Example on Convolution

In A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems*, F. Pereira, C. Burges, L. Bottou, *et al.*, Eds., vol. 25, Curran Associates, Inc., 2012, architecture that won the ImageNet challenge in 2012, the first convolutional layer used neurons with a receptive field size of $F = 11$, stride of $S = 4$, and no zero padding ($P = 0$). Since $(227 - 11)/4 + 1 = 55$, and since the Conv layer had a depth of $K = 96$, the Conv layer output volume had size $[55 \times 55 \times 96]$. Each of the $55 \times 55 \times 96$ neurons in this volume was connected to a region of size $[11 \times 11 \times 3]$ in the input volume.

See Figure 9.14 for a depiction of the above. Each of the 96 filters shown here is of size $[11 \times 11 \times 3]$, and each one is shared by the $55 \times 55$ neurons in one depth slice. Notice that the parameter sharing assumption is relatively reasonable. If detecting a horizontal edge is

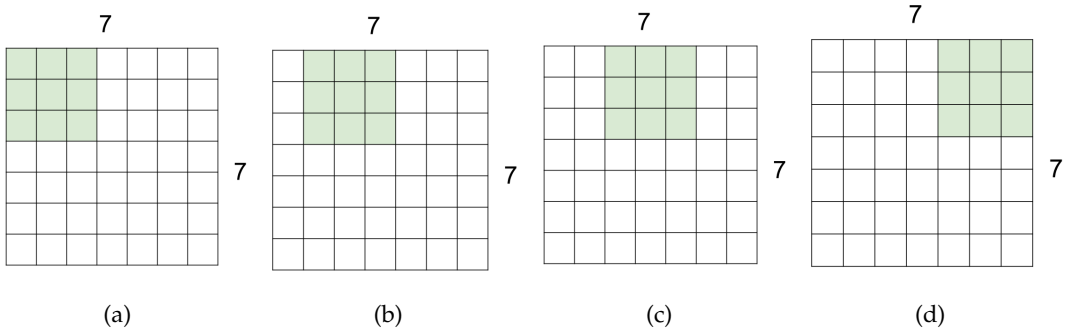



Figure 9.15: Output for a $7 \times 7$ input (spatially) with a $3 \times 3$ filter.

important at some location in the image, it should intuitively be useful at some other location as well due to the translationally-invariant structure of images. There is therefore no need to relearn to detect a horizontal edge at every one of the $55 \times 55$ distinct locations in the Conv layer output volume.

## 9.7 Summary on the Convolutional Layer

To summarize, the convolutional layer:

i. Accepts a volume of size $W_1 \times H_1 \times D_1$.

ii. Requires four hyperparameters:

- Number of filters $K$,
- their spatial extent $F$,
- the stride $S$,
- the amount of zero padding $P$.

iii. Produces a volume of size $W_2 \times H_2 \times D_2$ where:

$$W_2 = \frac{(W_1 - F + 2P)}{S} + 1$$
$$H_2 = \frac{(H_1 - F + 2P)}{S} + 1$$
$$D_2 = K$$

(i.e. width and height are computed equally by symmetry).

iv. With parameter sharing, it introduces $F \times F \times D_1$ weights per filter, for a total of $(F \times F \times D_1) \times K$ weights and $K$ biases.

v. In the output volume, the $d$-th depth slice (of size $W_2 \times H_2$) is the result of performing a valid convolution of the $d$-th filter over the input volume with a stride of $S$, and then offset by $d$-th bias.

vi. A common setting of the hyperparameters is $F = 3, S = 1, P = 1$. However, there are common conventions and rules of thumb that motivate these hyperparameters.

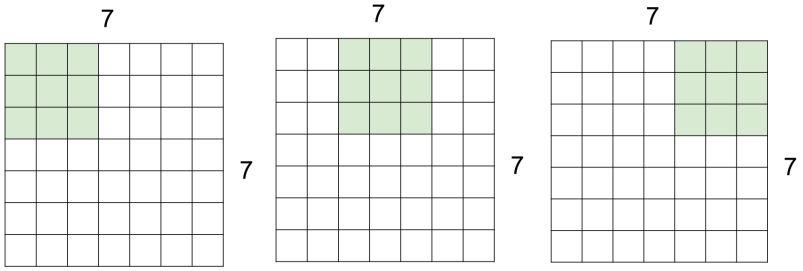



Figure 9.16: Output for a $7 \times 7$ input, $3 \times 3$ filter applied with stride 2.

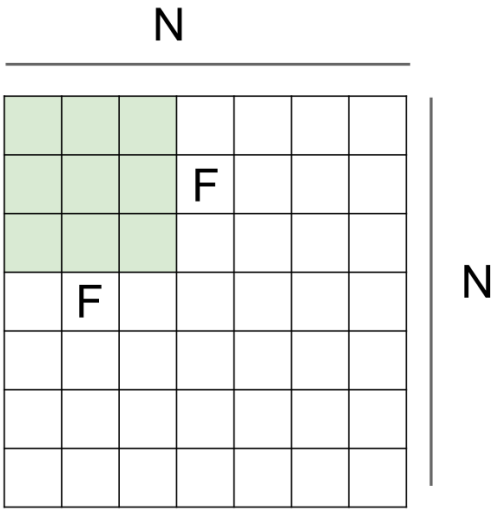



Figure 9.17: Output size visualization.

## 9.8 Convolution Examples

For the example scenario, we assume a $7 \times 7$ input (spatially) with a $3 \times 3$ filter. See Figure 9.15. This gives a $5 \times 5$ output.

Again, for the same $7 \times 7$ input, assume the $3 \times 3$ filter applied with stride 2. See Figure 9.16. This time, we get a $3 \times 3$ output.

Now, as what if for the same $7 \times 7$ input (spatially), we assume $3 \times 3$ filter applied with stride 3? It doesn't fit. We cannot apply $3 \times 3$ filter on $7 \times 7$ input with stride 3.

As clarified in Figure 9.17, output size can be obtained from:

$$\frac{(N - F)}{\text{stride}} + 1.$$

For example, for $N = 7$, $F = 3$:

$$\text{Stride } 1 \implies \frac{(7-3)}{1} + 1 = 5,$$

$$\text{Stride } 2 \implies \frac{(7-3)}{2} + 1 = 3,$$

$$\text{Stride } 3 \implies \frac{(7-3)}{3} + 1 = 2.33, \text{inconclusive.}$$

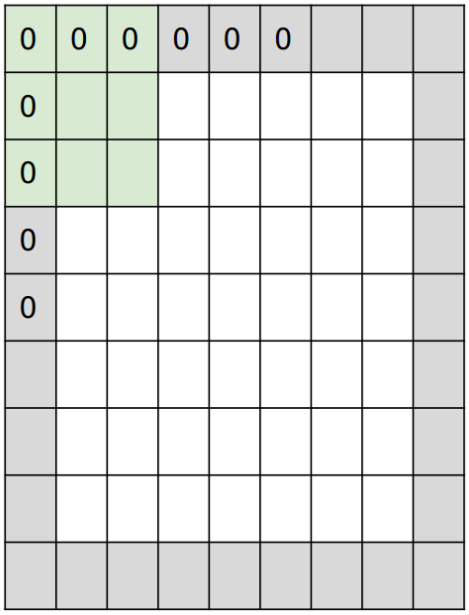



Figure 9.18: Use of zero pad at the border.

In practice, it is common to zero pad the border. As shown in Figure 9.18, for an input 7×7 and 3×3 filter, applied with stride 1 pad with 1 pixel border, we get a 7×7 output.

In general, it is common to see CONV layers with stride 1, filters of size $F \times F$, and zero-padding with $\frac{(F-1)}{2}$. This will preserve size spatially.
For example,

$F = 3 \Rightarrow$ zero pad with 1,

$F = 5 \Rightarrow$ zero pad with 2,

$F = 7 \Rightarrow$ zero pad with 3.

We need to remember that $32 \times 32$ input convolved repeatedly with $5 \times 5$ filters shrinks volumes spatially! ($32 \rightarrow 28 \rightarrow 24$ ...). Shrinking too fast is not good, doesn't work well.

**Example 9.8.1.** Input volume: $32 \times 32 \times 3$
10 $5 \times 5$ filters with stride 1, padding 2
Output volume size:

$$\frac{32 + 2 \times 2 - 5}{1} + 1$$

$= 32$ spatially, so

$32 \times 32 \times 10$

**Example 9.8.2.** Input volume: $32 \times 32 \times 3$. We take 10 $5 \times 5$ filters with stride 1, padding 2.

What will be the number of parameters in this layer?

Each filter has $5 \times 5 \times 3 + 1 = 76$ params (+1 for bias) $\Rightarrow 76 \times 10 = 760$.

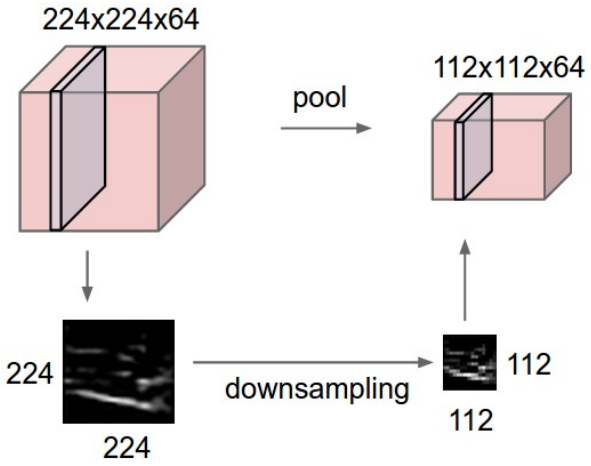



Figure 9.19: Pooling layer downsamples the volume spatially, independently in each depth slice of the input volume.

## 9.9 Convolution Demo

We explore a running demo of a Conv layer. The demo is available at this https://cs231n.github.io/convolutional-networks/ under the section **Convolution Demo** around the middle.

## 9.10 Pooling Layer

It is common to periodically insert a Pooling layer in-between successive Conv layers in a ConvNet architecture. Its function is to progressively reduce the spatial size of the representation to reduce the amount of parameters and computation in the network, and hence to also control overfitting. The Pooling Layer operates independently on every depth slice of the input and resizes it spatially, using the MAX operation. The most common form is a pooling layer with filters of size $2 \times 2$ applied with a stride of 2 downsamples every depth slice in the input by 2 along both width and height, discarding 75% of the activations. Every MAX operation would in this case be taking a max over 4 numbers (little $2 \times 2$ region in some depth slice). The depth dimension remains unchanged.

A pooling function replaces the output of the net at a certain location with a summary statistic of the nearby outputs. For example, the max pooling operation reports the maximum output within a rectangular neighborhood. Other popular pooling functions include the average of a rectangular neighborhood, the $L^2$ norm of a rectangular neighborhood, or a weighted average based on the distance from the central pixel.

See Figure 9.19 for an example. In this example, the input volume of size $[224 \times 224 \times 64]$ is pooled with a filter size of 2 and a stride of 2 into an output volume of size $[112 \times 112 \times 64]$. Notice that the volume depth is preserved.

More generally, the pooling layer:

(i) Accepts a volume of size $W_1 \times H_1 \times D_1$.

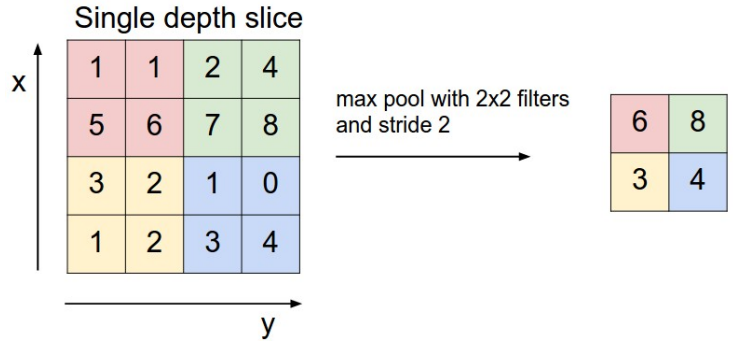



Figure 9.20: An example of max pooling.

(ii) Requires two hyperparameters: their spatial extent $F$, the stride $S$.

(iii) Produces a volume of size $W_2 \times H_2 \times D_2$ where:

- $W_2 = \frac{W_1 - F}{S} + 1$,
- $H_2 = \frac{H_1 - F}{S} + 1$,
- $D_2 = D_1$.

(iv) Introduces zero parameters since it computes a fixed function of the input.

For pooling layers, it is not common to pad the input using zero-padding. It is worth noting that there are only two commonly seen variations of the max pooling layer found in practice: A pooling layer with $F = 3, S = 2$ (also called overlapping pooling), and more commonly $F = 2, S = 2$. Pooling sizes with larger receptive fields are too destructive.

The most common downsampling operation is max, giving rise to max pooling. Figure 9.20 shows one example of max pooling. Here, a stride of 2 has been used. That is, each max is taken over 4 numbers (little $2 \times 2$ square).

## 9.11 Fully-Connected Layer

Neurons in a fully connected layer have full connections to all activations in the previous layer, as seen in regular Neural Networks. Their activations can hence be computed with a matrix multiplication followed by a bias offset. See the Neural Network section of the notes for more information.

## 9.12 Layer Sizing Patterns

We state the common rules of thumb for sizing the architectures:

- The input layer (that contains the image) should be divisible by 2 many times. Common numbers include 32 (e.g. CIFAR-10), 64, 96 (e.g. STL-10), or 224 (e.g. common ImageNet ConvNets), 384, and 512.
- The Conv layers should be using small filters (e.g. $3 \times 3$ or at most $5 \times 5$), using a stride of $S = 1$, and crucially, padding the input volume with zeros in such a way that the

Conv layer does not alter the spatial dimensions of the input. That is, when $F = 3$, then using $P = 1$ will retain the original size of the input. When $F = 5$, $P = 2$. For a general $F$, it can be seen that $P = (F - 1)/2$ preserves the input size. If you must use bigger filter sizes (such as $7 \times 7$ or so), it is only common to see this on the very first Conv layer that is looking at the input image.

- The pool layers are in charge of downsampling the spatial dimensions of the input. The most common setting is to use max-pooling with $2 \times 2$ receptive fields (i.e. $F = 2$), and with a stride of 2 (i.e. $S = 2$). Note that this discards exactly 75% of the activations in an input volume (due to downsampling by 2 in both width and height). Another slightly less common setting is to use $3 \times 3$ receptive fields with a stride of 2, but this makes "fitting" more complicated (e.g., a 32x32x3 layer would require zero padding to be used with a max-pooling layer with $3 \times 3$ receptive field and stride 2). It is very uncommon to see receptive field sizes for max pooling that are larger than 3 because the pooling is then too lossy and aggressive. This usually leads to worse performance.

## 9.13 Hands On

### 9.13.1 Visualization of Layers

We use a pretrained model for visualization of layers.

### 9.13.2 Image Feature Extraction

We take an image and apply the convolution layer, activation layer, and pooling layer operation to extract the inside feature.

### 9.13.3 Classify Digits

The MNIST ("Modified National Institute of Standards and Technology") dataset is the de facto "hello world" dataset of computer vision. Since its release in 1999, this classic dataset of handwritten images has served as the basis for benchmarking classification algorithms. As new machine learning techniques emerge, MNIST remains a reliable resource for researchers and learners alike.

### 9.13.4 Classify Images

We will use the CIFAR10 dataset.