An Introduction to Machine Learning

by Giovanni Santini

First Edition

Contents

Introduction

This book contains useful notes for students studying for an introductionary course to Machine Learning, or to anyone interested in the subject. The structure of the book results from my notes during the "Introduction to Machine Learning" course held by Elisa Ricci at the University of Trento in the academic year 2024-2025. During the course, I found myself more interested in the math than what the course taught, so I dedicated some of my time to understand the math better and to fill some of the gaps in my education. My final notes resulted in the book you are reading.

1 Machine Learning Basics

Machine Learning is the study of computer algorithms that improve automatically through experience. How they achieve this is the discussion of this book. In this chapter we will see a broad overview of the types of machine learning that we will study, as well as making some distinctions and defining some terminology. In future chapters we will discuss the algorithms more in detail.

In machine learning we want the computer to automatically detect patterns and predict future data from other data. To do so, we perform a series of steps which we call a *pipeline*:

- *Data acquisition*: We want to collect the relevant data for the problems at hand.
- *Preprocessing*: cleaning and preparing for analysis (missing values, formatting…). Techniques include normalization, feature scaling, handling categorical variables.
- *Dimensionality reduction*: selection methods can be applied to reduce the number of features while preserving the most important information.
- *Model learning*: a model is trained on the preprocessed data.
- *Model testing*: the model is evaluated using a test set.

1.1 Types of learning

Below are described the types of learning which will be discussed in this book:

1.1.1 Supervised Learning

Given labeled examples, create the model to predict the label, and test if the predicted label is correct.

- *Classification*: there is a finite set of labels to predict. Given a training set $T = \{(x_1, y_1), ..., (x_m, y_m)\}$ of size m, learn a function to predict y given x:
$$f : \mathbb{R}^d \to \{1, 2, ..., k\}$$
x is generally multidimensional (multiple features). Applications include:
  - Face recognition

  - Character recognition
  - Spam detection (classical problem)
  - Medical diagnosis
  - Biometrics

- *Regression*: the label is real value:

$$f : \mathbb{R}^d \to \mathbb{R}$$

- *Ranking*: label indicates an order.

1.1.2 Unsupervised Learning

The input is given with no labels. The main problems include:

- *Clustering*: Given an input $T = \{x_1, ..., x_m\}$, output the hidden structure behind the x's, which represents the clusters. Possible applications are:
  - social network analysis
  - genomics
  - image segmentation
  - anomaly detection
- *Dimensionality Reduction*: reduce the number of features under consideration by mapping data into another low dimensional space.

$$f : \mathbb{R}^d \to \mathbb{R}^m, m << d$$

- *Density estimation*: find a probability distribution that fits the data.

1.1.3 Reinforcement learning

The idea of reinforcement learning is that an *agent* learns from the *environment* by interacting with it and receiving *rewards* for performing *actions*. This framework of acting is formally known as the *Markov Decision Process*. The agent needs to learn which actions to take given a state to maximize the overall reward collected.

1.1.4 Other learning variations

- *semi-supervised*: some data have have labels, and some don't.
- *active learning*: the model learns a labeled dataset, and It interactively queries a human user to label new data points.

- *batch (offline) learning*: the model learns from the entire dataset in one go and is updated only after processing all the data
  - Once the model is trained, it does not change.
  - Typically used when the dataset fits into memory and can be processed efficiently as a whole.
- *online learning*: the model learns incrementally from each new data point or small batches of data.
  - Allows the model to adapt to changes in the data distribution over time.
  - Suitable for scenarios where data arrives sequentially and needs to be processed in real-time or where computational resources are limited. Examples include data streams, large-scale dataset and privacy-preserving applications.

1.2 Features

Features are the questions we can ask about the examples. They are generally represented as *vectors*.

1.3 Generalization

Machine learning is about generalization of data.

Generalization in machine learning refers to the ability of a trained model to perform well on new, unseen data that was not used during the training process.

This can be done only if there is a correlation between inputs and ouputs. More technically, we are going to use the *probabilistic model* of learning.

- there is some probability distribution over example / label pairs called the data generating distribution
- both the training data and the test set are generated based on the distribution

A data *generating distribution* refers to the underlying probability distribution that generates the observed data points in a dataset. Understanding this distribution is crucial for building accurate and generalizable machine learning models because It enables us to make informed assumptions about the data and to make predictions or decisions based on probabilistic reasoning. This is valid for every kind of machine learning.

1.4 Learning process

The steps to take in order to learn a model are:

1. Collect (annotated) data
2. Define a family of models for the classification task
3. Define an error function to measure how well a model fits the data
4. Find the model that minimized the error, aka train or learn a model

We define the following:

- *task*: a task represents the type of prediction being made to solve a problem on some data. $f : x \to y$
  - For example, in the classification case, $f : x \to \{c_1, ..., c_k\}$.
  - Similarly is clustering, where the output is a cluster index.
  - Regression: $f : \mathbb{R}^d \to \mathbb{R}$
  - Dimensionality reduction: $f : x \to y, dim(y) << dim(x)$
  - Density estimation: $f : x \to \Delta(x)$
- *data*: information about the problem to solve in the form of a distribution p which is typically unknown.
  - training set: the failure of a machine learning algorithm is often caused by a bad selection of training samples.
  - validation set
  - test set
- *model hypotheses*: a model $Ftask$ is an implementation of a function f:

  $$f \in Ftask$$

  A set of models forms an hypothesis space:

  $$Hip \subseteq Ftask$$

  We use an hypothesis space to reduce the number of possible models in order to make our life easier.
- *learning algorithm*: the algorithm of your choice based on the problem
- *objective*: we want to minimize a (generalization) error function $E(f, p)$.

  $$f* \in arg\ min\ E(f, p), f \in Ftask$$

$Ftask$ is too big of a function space, we need an implementation (model hypotheses) so we define a model hypothesis space $Hip \in Ftask$ and seek a solution within that space.

$$f_{Hip} * (D) \in arg\ min_{f \in Hip_M} E(f, D)$$

With $D = \{z_1, ..., z_n\}$ being the training data.

1.5 Error function

Let $l(f, z)$ be a pointwise loss (a sum of point-losses). The error is computed from a function in an hypothesis space and a training set.

$$E(f, p) = \mathbb{E}_{z \sim pdata}[l(f, z)]$$

$$E(f, D) = \frac{1}{n} \sum_{i=1}^{n} l(f, z_i)$$

We want to minimize such error.

1.6 Underfitting and Overfitting

- *Underfitting*: the error is very big, the output is very "far" from the ideal solution
- *Overfitting*: there is a large gap between the generalization (validation) and the training phase.

1.7 How to improve generalization

Common techniques to improve generalization include:

- avoid attaining the minimum on training error.
- reduce model capacity.
- change the objective with a regularization term:

$$E_{reg}(f, D_n) = E(f, D_n) + \lambda \Omega(f)$$

  - λ is the trade-off parameter
  - For example:

$$E_{reg}(f, D_n) = \frac{1}{n} \sum_{i=1}^{n} [f(x_i) - y_i]^2 + \frac{\lambda}{n} |w|^2$$

- inject noise in the learning algorithm.
- stop the learning algorithm before convergence.
- increase the amount of data:

$$E(f, D_N) \to E(f, p_{data}),\ n \to \inf$$

- augmenting the training set with transformations (rotate the image, change brightness. . .).
- combine predictions from multiple, decorrelated models (resembling).
  - train different models on different subsets of data, and we average the final solution between all of them

1.8 Parametric vs Non-parametric Models

- *Parametric models* have a finite number of parameters
  - linear regression, logistic regression, and linear support vector machines
- *Nonparametric model*: the number of parameters is (potentially) infinite
  - k-nearest neighbor, decision trees, RBF kernel SVMs

1.9 Bias

The bias of a model is a measure of how strong the model assumptions are

- low-bias classifiers make minimal assumptions about the data
- high-bias classifiers make strong assumptions about the data

2 The K-nearest neighbors algorithm

In this chapter we will discuss our first supervised learning algorithm: *K-nearest neighbors*, aka *K-NN*. Remember that supervised learning means that the algorithm is given data in an n-dimensional space with labels, we will now focus on the classification problem where the algorithm should predict the label of new unseen data.

If we make the only assumption that data with similar labels are somehow “cose” together in the space, a simple solution to this problem could be to look at the nearest known data and their label. We will discuss the concept distance later. Moreover, we can average over k nearest neighbors to get a more representative result.

More precisely, to *classify* an example d:

- find k nearest neighbors of d
- choose as the label the majority label within the k nearest neighbors

An example speudocode:

```
KNN(input, K, DATA) -> {1, ..., m}:
    int label_count[m] = {0}
    bool used[#DATA] = {false}
    for k=0 to K do:
        int max_distance = 0
        int max_input = 0
        for x_i, y_i in DATA do:
            d = distance(x_i, input)
            if d > max_distance and not used[x_i] then:
                max_distance = d
                max_input = x_i
        label_count[DATA[x_i]] += 1
        used[x_i] = true
    return max_index(label_count)
```

2.1 How do we measure distance?

Measuring distance / similarity is a domain-specific problem and there are many different alternatives. We will now see different methods to model distance.

2.1.1 Euclidean distance

Euclidean distance is measured in n dimensions with the formula:

$$D(a, b) = \sqrt{(a_1 - b_1)^2 + (a_2 - b_2)^2 + ...}$$

Note that here we are assuming that all the labels are comparable, meaning measured in the same range. A trivial example can be made to motivate this: a certain dimension may represent the cost of an item in euros and some other dimension may measure It in dollars, to find the difference we need to first convert everything to the same currency. Therefore data should be *standardized*, we will now see two common transformations:

- *Standardization or Z-score normalization*: Rescale the data so that the mean is 0 and the standard deviation from the mean is 1.

$$x_{norm} = \frac{x - \mu}{\sigma}$$

- *Min-Max scaling*: scale the data to a fixed range - between 0 and 1.

$$x_{norm} = \frac{x - x_{min}}{x_{max} - x_{min}}$$

2.1.2 Manhatthan Distance

The Manhatthan distance is used to find the distance of two paths in a grid, It was originally invented for the city o Manatthan

$$D(a,b) = \sum_i |a_i - b_i|$$

2.1.3 Checysher Distance

The checkysher distance is useful to calculate distances in the chess board, in particular the number of moved of the kind to reach a certain square.

$$D(a,b) = max_i|a_i - b_i|$$

2.1.4 Minkowsky distance

All the three aforementioned distances can be generalized with the following formula (Minkowsky Distance) also called p-norm or L_p:

$$D(a,b) = (\sum_{k=1}^{n} |a_k - b_k|^p)^{\frac{1}{p}}$$

- $p = 1$: Manhattan distance
- $p = 2$: Euclidean distance
- $p \to \infty$: Checkysher distance

L_1 is popular because it tends to result is sparse solutions. However, It is not differentiable, so It only works for gradient descent solvers. L_2 is also popular because for some loss functions, it can be solved directly. L_p is less popular since they don't tend to shrink the weights enough.

2.1.5 Cosine Similarity

Given two different vectors d_1 and d_2, we can find the cosine cos with the formula:

$$cos(d_1, d_2) = (d_1 \cdot d_2)/||d_1||||d_2||$$

- where $\cdot$ is the dot product and $||d||$ is the length of vector d

In fact, this formula is derived from the definition of dot product: $d_1 \cdot d_2 = ||d_1||||d_2||cos(\theta)$

Cosine similarity does not depend on the magnitudes of the vectors, but only on their angle. For example, two proportional vectors have a cosine similarity of $+1$, two orthogonal vectors have a similarity of 0, and two opposite vectors have a similarity of -1.

For example, in information retrieval and text mining, each word is assigned a different coordinate and a document is represented by the vector of the numbers of occurrences of each word in the document. Cosine similarity then gives a useful measure of how similar two documents are likely to be, in terms of their subject matter, and independently of the length of the documents.

2.2 Decision Boundaries

The KNN algorithm can be thought of as assigning a label to an object withing the space enclosed by a *decision boundary.* Decision boundaries are places in the features space where the classification of a point / example changes.

2.2.1 Voronoi Diagram / Partitioning

A Voronoi diagram describes the areas that are nearest to any given point, given a set of data where each line segment is equidistant between two points. More formally, the Voronoi region R_k associated with a subset of the points in the space P_k is defined as:

$$R_k = \{x \in X \mid d(x, P_K) \leq d(x, P_j) \ for \ all \ j \neq k\}$$

KNN does not explicitly compute decision boundaries, but form a subset of the Voronoi diagram for the training data.

2.3 Choosing K

Some techniques to pick k include:

- common heuristics
- use validation set
- use cross validatoin
- rule of thumb is k < sqrt(n) where n is the size of training examples

In general, bugger values of k give a smoother decision boundary.

2.4 Lazy Learner vs Eager Learner

k-NN belongs to the class of lazy learning algorithms.

- *lazy learning*: simply stores training data and operates when it is given a test example. Note that if the data is large, the machine may run out of memory.
- *eager learning*: given a training set, constructs a classification model before receiving new test data to classify

This means that k-NN is not really fast during inference, but no training is required.

2.5 Curse of Dimensionality

There is a big problem in high dimensional data that may degrade the performance of the algorithm. That is, in high dimensions almost all points are far away from each other. I will now proceed to motivate this.

Suppose you have a space in n dimensions where m points are distributed uniformly. The volume of the space would be S^n where S is a measure of a side or the size of a domain in space, assuming all have the same domain. You could quantify a certain data density quantity as $\delta = \frac{m}{S^n}$. If we increase the dimensions by one, the density would decrease by a factor of S, hence we need to have S times the original data size m to get the same density: $\delta' = \frac{mS}{S^{n+1}}$. In general, $\delta'' = \frac{mS^k}{S^{n+k}}$, for this reason we say that the size of the input has to grow exponencially with the dimensions. A less dense space means that all the data is more spread apart, hence all points are far away from each other.

The success of KNN is very dependent on having a dense data set since It requires points to be close in every dimension.

2.6 Computational Cost

- Linear algorithm (no preprocessing) is $O(kN)$ to compute the distance for all N datapoints
- $O(klog(n))$ for tree-based data structures: pre-processing often using K-D tree
  - divide the space in regions. To check which region a point belongs to, simply traverse a binary tree.

k-NN variations: weighted k-NN where closest neighbors contribute the most.

3 Linear Models

We have seen the KNN algorithm for classification. Said algorithm works everywhere as long as there is some spacial correlation in the data, which is often the case. We will now focus on a stricter set of problems, we will consider data that is *linearly separable*.

We say that labeled data is linearly separable if the classes can be binary separated by a line in 2 dimensions, or using hyperplanes in higher dimensions. In other words, a hyperplane defines a partition of the space and the data can be classified based on in which partition they live. This is a strong assumption to make and does not hold for many problems, we will see in a later chapter how we can circumvent this.

The equation of a plane in n dimensions looks like this

$$(1)\ b + \sum_{i=1}^{n} w_i x_i = 0$$

Where w_i are the values of the normal an x_i are points in space in the dimension i.

We can classify a linear model by evaluating the equation (1) with an input point and checking the sign of the output: positive values are classified as one thing, negative values as the other (remember that we are assuming binary classification).

More formally, given:

- an input space X
- an unknown distribution D over $X \times \{-1, +1\}$
- a training set D sampled from D

Compute: A function f minimizing $\mathbb{E}_{(x,y)\sim D}[f(x) \neq y]$

3.1 Training a linear model

Differently from KNN, linear models use *online learning.* The learning algorithm follows the following structure:

- the algorithm receives an unlabeled example x_i
  - For example: $w(1, 0)$, $x_i(-1, 1)$
- the algorithm predicts a classification of this example
  - Evaluate the sum equation $b + \sum_{i=1}^{n} w_i x_i$: $1 * (-1) + 0 * 1 = -1$
- the algorithm is them told the correct answer y_i and It updates the model only if the answer is incorrect
  - check if the sign of the equation represents the right label, if not, add to each weight the current input times the label (assuming the label is either $+1$ or -1)

```
repeat until convergence (or some # of iterations):
    for each random training example (x1, x2, ..., xn, label):
        check if prediction is correct based on the current model
        if not correct, update all the weights:
            for each wi:
                wi = wi + xi*label
            b = b + label
```

The algorithm will converge only if the data can be linearly separated. The reason why $w_i + f_i \cdot label$ improves the solution will be obvious later when we will discuss gradient descent.

3.1.1 Learning Rate

When the model makes an incorrect prediction, you hay have noticed that the correction of the weights are multiple of the label ($f_i \cdot label$ is added to the weight). This may make convergence difficult if we need more precise weights, so we multiply this by a *learning rate.*

$$w_i := w_i + \lambda x_i * label$$

4 Beyond Binary Classification

4.1 Multi-class classification

In multi-class classification problem, differently from binary classification, given:

- An input space X and number of classes K
- An unknown distribution D over $X \times [K]$
- A training set D sampled from D

Compute: A function f minimizing $\mathbb{E}_{(x,y)\sim D}[f(x) \neq y]$

- Idea: one line does not suffice, but we can combine more lines

There are two approaches to achieve multi-class classification which we will discuss:

- one vs all
- all vs all

4.2 One vs All (OVA)

For each label $k = 1, ..., K$ define a binary problem where:

- all examples with label k are positive
- all other examples are negative

In practice, learn K different classification models.

To classify we pick the most confident positive, in none vote positive, pick least confident negative.

```
OneVsAllTrain(D, BinaryTrain):
    for i=1 to K do
        D_1 = relabel D so class i is positive and \not i is negative
        f_i = BinaryTrain(D_1)
    end for
    return f_1,...,f_k

OneVsAllTest(f_1,...,f_k, x):
    score = (0,...,0)
    for i=1 to K do
        y = f_1(x)
        score[i] = score[i] + y
    end for
    return max(score)
```

4.3 All vs All (AVA)

All vs All, sometimes called *all pairs*, trains $K(K-1)/2$ classifiers:

- the classifier F_{ij} receives all the examples of class i as positive and all the examples of class j as negative, for each pair (i, j)
- every time F_{ij} predicts positive, the class i gets a vote, otherwise, class j gets a vote
- after running all $K(K-1)/2$ classifiers, the class with the most votes wins

Note: The teacher might ask to explain the algorithm in more detail

4.4 Ova vs Ava

Train time:

- AVA learns more classifiers, however, they are trained on much smaller data this tends to make it faster if the labels are equally balanced

Test time:

- AVA has more classifiers, so often is slower

Error:

- AVA tests with more classifiers and therefore has more chances for errors

4.5 Macroaveraging vs Microaveraging

- Microaveraging: average over examples
- Macroaveraging: calculate evaluation score (e.g. accuracy) for each label, then average over labels

5 Decision Trees

Decision trees are simple, intuitive and powerful supervised learning algorithms used for both classification and regression. They make predictions by recursively splitting on different attributes according to a *tree structure*. For continuous attributes splitting is based on less than or greater than some threshold. During testing time, the program will walk the tree from the root, based on the split condition at each intersection, until It reaches the leaf nodes.

Given a training dataset, we need to learn:

- which attribute to choose for splitting
- what value of that attribute to use for splitting

5.1 Classification and Regression

Training examples that fall into the region R_m

$$\{(x^{m_1}, t^{m_1}), ..., (x^{m_k}, t^{m_k})\}$$

Classification tree:

- discrete output $y \in \{1, ..., C\}$
- after traversing the tree, the final leaf value y^m is typically set to the most common value in $\{t^{m_1}, ..., t^{m_k}\}$ which is the set of labels in the region, i.e.

  $$y^m = argmax_{t \in \{1,...,C\}} \sum_{m_i} \nVdash_t(t^{m_i})$$

  Where $\nVdash(x)$ is the indicator function defined as:

  $$\nVdash_A : X \to \{0, 1\}$$

  $$\nVdash_A(x) \equiv \begin{cases} 1, & x \in A \\ 0, & x \notin A \end{cases}$$

Regression tree:

- continuous output: $y \in \mathbb{R}$
- the leaf value y^m typical set to the mean value in $\{t^{m_1}, ..., t^{m_k}\}$

5.2 Train a decision tree

We want to find a simple tree that explains data well. A good enough approach would be to use a greedy heuristic: start with an empty decision tree and complete the training set by recursively "splitting" the best attributes.

We will now discuss three ways to decide which split is the best.

5.2.1 Accuracy Gain

A *gain* is defined based on change in some criteria before and after a split. Let us define *accuracy gain.*

- suppose we have a region R. Denote the misclassification error of that region as $L(R)$ which is the fraction of incorrect classification
- we split R into two regions R_1 and R_2 based on some attribute
- the error after the split is:

$$\frac{|R_1|}{|R|}L(R_1) + \frac{|R_2|}{|R|}L(R_2)$$

  where $|R|$ is the number of samples in R.
- we define the accuracy gain as:

$$AG = L(R) - \frac{|R_1|L(R_1) + |R_2|L(R_2)}{|R|}$$

A big value of AG means that the split is good. If the error $L(R)$ after the split is as good as before, AG is 0, and if the split introduced more error then AG is negative.

We defined a function to measure how good a split is, let's look at another possibility inspired by information theory.

5.2.2 Basic of information theory and entropy

Which coin result is more uncertain between those two?

- 0001000000000000010001
- 0101100101010011101010101

Entropy is a measure of expected "surprise". How uncertain are we of the value of a draw from this distribution?

$$H(X) = -\mathbb{E}_{X\sim p}[\log_2 p(X)] = -\sum_{x\in X} p(x)\log_2 p(x)$$

- the greater the entropy, the less certain we are. The more certain we are, the less entropy
  - remember that entropy is the measure of disorder

Meaning: we are taking the probability of the "surprise" of an event E with the formula $\log(\frac{1}{p(E)})$ which can be rewritten as $-\log(p(E))$. This formula means that the more likely the event, the closes it's surprise will be 0. The opposite is true: the less probable the event, the greater the value of the log function. Since we are taking the expected value, the more unexpected is the probability of the event, the higher the entropy. The formula with the sum is derived by directly applying the definition of expected value $\mathbb{E}[x] = \sum_{i=1} x_i p_i$

On high entropy, the variable has a more uniform-like distribution, meaning that It is more difficult to predict an event (so It surprises us). On low entropy, the distribution has peaks and valleys were some events are frequent and some are not (so some values are more probable than others and do not surprise us).

Entropy can be a better option than accuracy gain to measure certainty.

5.2.3 Information Gain

Information Gain measures the expected reduction in entropy. It is defined as

- Entropy of region R or parent node before the split: $H(R)$
- entropy of region R_1 and R_2 or leaf nodes after the split: $H(R_1)$ and $H(R2)$
- Information gain is defined as

$$IG = H(R) - \frac{|R_1|H(R_1) + |R_2|H(R_2)}{|R|}$$

The feature and the corresponding value with the highest information gain will be selected for splitting.

5.2.4 Gini index

Gini Index is another widely used metric for choosing a good split in decision trees. The Gini index measures the "impurity" (or non-homoneneity) at a given modes as

$$GINI(R) = 1 - \sum_j [p(j|R)]^2$$

where $p(j|R)$ is the class frequency of class j in region R

- intuitively, it aims to measure the probability of misclassifying a randomly chosen element
- greater the value of the Gini index, the greater the changes of having misclassificaiton
- thus, we will look for greater gini values

5.3 Overfitting

- decision trees have a structure that is determined by the data
- as a result they are flexible and can easily fit the training set, with high risk of overfitting
- what we can do is cut branches of the tree and replaceing it by a leath node (*pruning*)

5.4 Limitations

- you have exponentially less data al lower levels
- a large tree can overfit the data
- decision trees do not necessarly reach the global minima
- mistakes at top-level propagate down tree

Decision trees can also be used for regression on real-valued outputs by choosing the squared error rather than maximizing the information gain.

5.5 Comparison to KNN

Advantages and Decision Trees over KNN

- good with descrete attributes
- easily deals with missing vales
- robust to scale of inputs
- good when there are lots of attributes, but only a few are important

- fast at test time
- more interpretable

If some features are more important than others, you may want to choose a decision tree.

Strengths:

- fast and simple to implement
- can convert to rules
- allows for batch training

Disadvantages:

- univariate feature split
- requires fixed-length feature vectors
- non-incremental (batch method)

5.6 Improvement to decision trees: Random Forests

Bootstrapping is a resampling technique that involves repeatedly drawing samples from the dataset with replacement (the same element can be selected multiple times)

- this creates multiple datasets that introduce variability, reducing overfitting in models

Bagging (or Bootstrap Aggregation) involves training multiple decision trees on different bootstrapped samples and averaging their outputs (for regression) or majority voting (for classification)

- In addition to bootstrapping, Random Forests introduce feature randomness, this descreases the correlation between each DT and increases its predictive accuracy on average. In other words, avoid very strong predictive features that lead to similar split in trees

Algorithm:

1. Draw multiple bootstrapped datasets from the original data
2. Train a DT on each dataset using a random subset of $sqrt(d)$ features at each split
3. Aggregate predictions:
    1. classification: majority vote
    2. regression: average predictions

6 Gradient Descent

6.1 Model-Based Machine Learning

1. Pick a model (hyperplace, decision tree…)
2. Pick a criterion to optimize
3. Develop a learning algorithm
   - this should try and minimize the criteria, sometimes in a heuristic way, sometimes exactly

Let's look at linear binary classification

1. Pick a model
$$0 = b + \sum_{j=1}^{m} w_j f_j$$
2. Pick a criteria to optimize (aka objective function)
$$\sum_{i-1}^{n} \nVdash[y_i(w * x_i + b) \leq 0]$$
3. Develop a learning algorithm
$$argmin_{w,b} \sum_{i-1}^{n} \nVdash[y_i(w * x_i + b) \leq 0]$$
   Find w and b that minimize the 0/1 loss (i.e. training error)

6.2 Loss Functions

How do we minimize the 0/1 loss? Finding w and b that optimize the 0/1 loss is hard (in fact, NP-hard).

We want a differentiable (continuous) loss function so we get an indication of direction of minimization. This family of functions are called convex functions: the line segment between any two points on the function is above the function.

Therefore, we want to find one such function that represents the 0/1 loss function, which we call *surrogate loss function.* There are many such functions, we use the notation y as the correct value and y' as the predicted value:

- 0/1 loss: $l(y, y') = \nVdash[yy' \leq 0]$

- Hinge $l(y, y') = max(0, 1 - yy')$
- Exponential: $l(y, y') = exp(-yy')$
- Squared loss: $l(y, y') = (y - y')^2$

6.3 Gradient Descent

Pick a starting point w. Repeat until loss doesn't decrease in any dimension:

- pick a dimension
- move a small amount in that dimension towards decreasing loss

$$w_j = w_j - \mu \frac{d}{dw_j} loss$$

- μ is the learning rate

For example, let's calculate the derivative of the loss function for the exponential:

$$\begin{aligned} \frac{d}{dw_i} loss &= \frac{d}{dw_j} \sum_{i=1}^{n} exp(-y_i(wx_i + b)) \\ &= \sum_{i=1}^{n} exp(-y_i(wx_i + b)) \frac{d}{dw_i} \\ &= \sum_{i=1}^{n} -y_i x_{ij} exp(-y_i(wx_i + b)) \end{aligned}$$

So with the exponential choice of loss function we have the following update rule:

$$w_j = w_j + \mu \sum_{i=1}^{n} -y_i x_{ij} exp(-y_i(wx_i + b))$$

- this may descend to a *local minima*.
- there may be *saddle points* where the gradient is 0 even if we are not in a minimum, where some directions curve upwards and some curve downwards.

There are two types of gradient descent:

- *Batch Gradient Descent*: the estimates are stable however you need to compute gradients over the entire training for one update.

- *Stochastic Gradient Descent*: we sample only one data point from the training set and compute the gradient. The learning rate changes at each step, It typically decays linearly. A problem arises with flat error surfaces where the error would jump vividly. We can introduce a variable called *velocity* to counteract this, where the gradient updates the velocity.

It is often useful to compute the gradient on set of samples.

6.4 Gradient

The gradient is the vector of partial derivates w.r.t all the coordinates of the weights. For $f : \mathbb{R}^n \to \mathbb{R}$ its gradient $\nabla f : \mathbb{R}^n \to \mathbb{R}^n$ is defined at the point $p = (x_1, ..., x_n)$ in n-dimensional space as the vector:

$$\nabla f(p) = [\frac{df}{dx_i}(p) \; ... \; \frac{df}{dx_n}(p)]$$

Explained to a programmer, the gradient is defined for each dimension and It is the vector where each element is the derivative with respect to that dimension. Formally, the gradient of a scalar function $f(x_1, x_2, x_3, ..., x_n)$ is denoted ∇f where ∇ is the vector differential operator (known as del or nabla). The gradient of f is defined as the unique vector filed whose dot product with any vector v at each point x is the directional derivative of f along v. That is,

$$(\nabla f(x)) * v = D_v f(x)$$

For example, the gradient of the function $f(x, y, z) = 2x + 3y^2 - sin(z)$ is $\nabla f(x, y, z) = [2, 6y, -cos(z)]$ or $\nabla f(x, y, z) = 2i + 6yj - cos(z)k$ where i, j, k are the standard unit vectors in the direction x, y, z.

- Each partial derivative measures how fast the loss changes in one direction

7 Support Vector Machines

Linear classifiers assume the data can be linearly separable. If this does not hold, the chosen hyperplane will keep adjusting at each iteration without ever converging. Let's now discuss a particular type of classifier: support vector machines (SVM).

7.1 Large margin classifiers

We define the *margin* of a classifier as the distance from the classification hyperplane to the closest points of either class. We call *large margin classifiers* the classifiers that attempt to maximize this measure.

We call the sets of "closest points" from the hyperplane with *support vectors*. Note that for n dimensions, there will be at least $n + 1$ support vectors; this is because a plane is identified by $n + 1$ points in n dimensions.

Therefore, large margin classifiers only use support vectors and ignore all other training data.

Let's formalize the large margin classifier with algebra.

We uniquely identify the decision hyperplane by a normal vector w and a scalar b which determines which of the planes perpendicular to w to choose:

$$\vec{w}\vec{x} + b = 0$$

We are interested in maximizing the distance from the hyperplane to the support vectors. Let us first find the distance from the hyperplane to a point p. We know that the distance vector must be perpendicular to the hyperplane, therefore It is parallel to the normal vector w, which when normalized is $\frac{\vec{w}}{||\vec{w}||}$.

Therefore, the distance between the hyperplane and p_1, referred to as r, is the difference between p_1 and the closest point in the hyperplane p_2, which is the intersection between the line parallel to the normal and point $p1$:

$$(1)\ r\frac{\vec{w}}{||\vec{w}||} = \vec{p_1} - \vec{p_2}$$

Moreover, since $p2$ lies on the hyperplane, It is ture that:

$$(2)\ \vec{w}\vec{p_2} + b = 0$$

We find $p2$ from (1), $p_2 = p_1 - r\frac{\vec{w}}{||w||}$. We substitute p_2 in (2) and solve for r, the distance, and we get

$$r = \frac{(\vec{w}\vec{p_1} + b)||\vec{w}||}{\vec{w}\vec{w}}$$

Since $\vec{w}\vec{w} = ||w||^2$, this can be easily demonstrated because the modulo function uses the generalized pythagorean theorem, we get:

$$r = \frac{(\vec{w}\vec{p_1} + b)}{||\vec{w}||}$$

We just demonstrated the formula for finding the distance between a point and a plane. We now define the *classification function* as $f(x) = sign(\vec{w}\vec{x} + b)$, It will be $+1$ or -1 based on the sign of the equation. Intuitively, points on one side of the plane should have a positive output and points on the other side should have a negative one.

We want to maximize this distance, but the value is positive or negative depending on where the point is, hence we multiply the hyperplane equation times the expected classification in roder to make the distance always positive (labels are either -1 or 1). Indeed, If we assign the label $y_i = -1$ for the negative distance points, we will always get a positive output (negative times negative is negative, and positive times positive is positive). We define the *functional margin* as double such quantity: this represents the distance between the two closest points from the plane with different labels:

$$margin = 2\frac{y_i(\vec{w}\vec{x_i} + b)}{||\vec{w}||}$$

For convenience, we require that the distance between the hyperplane and the support vectors is 1, therefore $y_i(\vec{w}\vec{x_i} + b) \geq 2\forall i$ because negative distances will be multiplied by y_i which is also negative, resulting in a positive value all the time, and a greater than 1 because of the requirement.

For the support vectors, the following holds:

$$margin = \frac{1}{||\vec{w}||}$$

7.2 Maximizing the margin

We want to maximize this margin, therefore we need to minimize $\frac{1}{2}||\vec{w}||$ with the constraint that $y_i(\vec{w}\vec{x_i}+b) \geq 1\ \forall i$. We may as well minimize $\frac{1}{2}||\vec{w}||^2$ which is an example of a *quadratic optimization problem*

$$min_{w,b}\ \frac{1}{2}||w||^2,\ y_i(\vec{w}\vec{x_i}) \geq 1\forall i$$

7.3 Soft margin classification

We may have data that is not perfect, for example we may have some values that lay on one side of the hyperplane but they are classified as the other, so $y_i(\vec{w}\vec{x_i} + b) \geq 1\forall i$ does not hold since you may have $\vec{w}\vec{x_i} + b \leq 1$ and $y_i \geq 1$ or the opposite.

To address this cases, we introduce *slack variabes* which are scalar values assigned for each x_i, and the *regularizaion parameter* $C > 0$:

$$min_{w,b}||\vec{w}||^2 + C\sum_i \zeta_i$$

subject to:

$$y_i(\vec{w}\vec{x_i} + b) \geq 1 - \zeta_i \forall i, \zeta_i \geq 0$$

with this correction, the values are "allowed to have mistakes", the margin is reduced by ζ if $\zeta < 1$ or are moved to the other side of the hyperplane if $\zeta > 1$. C determines how strong are those corrections, that is the "trade off" between the slack variable penalty and the margin.

- small C allows constrains to be easily ignored in order to have a larger margin
- large C makes constraints hard to ignore in order to get a narrow margin
  - if C goes to infinity, we have hard margins.

We are now interested into calculating these slack variables. The first observation we make is that if the constraint is already satisfied, meaning that if $y_i(\vec{w}\vec{x_i} + b) \geq 1$ and the value has not been misclassified, then we don't need any correction and $\zeta = 0$. Otherwise, we want to correct the value by moving it on the other side, plus the distance to the margin 1, so $1 - y_i(\vec{w}\vec{x_i} + b)$. We are subtracting because if the value is misclassified, then It's distance from the hyperplane times It's label is going to be a negative value (one of them nedds to be positive and the other negative in order to have a misclassification).

Therefore:

$$\zeta = 0 \; if \;\; y_i(\vec{w}\vec{x_i} + b) \geq 1$$

$$\zeta = 1 - y_i(\vec{w}\vec{x_i} + b) \; otherwise$$

which is the same as the following, using a notation introduced in previous lessons:

$$\zeta = max(0, 1 - y_i(\vec{w}\vec{x_i} + b)) = max(0, 1 - yy')$$

If you recall from the lesson of Gradiente Descent, this is the hinge loss function.

With this result, the objective is now to minimize the following:

$$min_{w,b}||\vec{w}||^2 + C\sum_i max(0, 1 - y_i(\vec{w}\vec{x_i} + b))$$

8 Optimization Problems

8.1 Cases of optimization problems

For future analisys, It is useful to discuss what are the main classes of optimization problems:

- _linear programming) (LP): linear problem, linear constraints.

$$min_x c^T x \ s.t. \ Ax = b, x \geq 0$$

- *quadratic programming* (QP): quadratic objective and linear constraints, it is convex if the matrix Q is positive semidefinite, that is the real number $x^T Qx$ is positive or zero for every nonzero real column vector x, where x^T is the row vector transpose of x.

$$min_x c^T x + \frac{1}{2} x^T Q x \ s.t \ Ax = b, Cx \geq d$$

- *nonlinear programming* (NLP): in general non-convex.

8.2 Solving quadratic problems - Lagrange multipliers

Quadratic optimization problems such as the one discussed above are a well-known class of mathematical programming models with several algorithms. We will now introduce a method so solve such problems using the Lagrange multiplier, that is a strategy for finding the local maxima and minima of a function subject to equation constraints.

Given a function to optimize $f(x)$, a constraint $g(x)$ and an optimal solution x_* of the function that respects the contraints, there exists a *lagrangian multiplier* λ such that:

$$\frac{df(x_*)}{dx_*} = \lambda \frac{dg(x_*)}{dx_*}, \ g(x) = 0$$

Or equivalently:

$$\frac{df(x_*)}{dx_*} - \lambda \frac{dg(x_*)}{dx_*} = 0, \ g(x) = 0$$

We call this the lagrangian function or *Lagrangian*:

$$L(x) = f(x) - \lambda g(x)$$

Let's now apply this knowledge in our problem. Let $f(x) = ||\vec{w}||^2$ and $g(x, b, w) = y_i(\vec{w}\vec{x_i} + b) - 1$, using a as the lagrangian multiplier:

$$(a) L(x, \vec{w}, b, \vec{a}) = \frac{1}{2}||\vec{w}||^2 - \sum_i a_i(y_i(\vec{w}^T\vec{x_i} + b) - 1)$$

This is an example of Lagrangian dual problem, where we need to maximize the Lagrangian multipliers to minimize w and b. We now derivate with respect to w and b and set them equal to 0:

$$(b)\ \vec{w} - \sum_i a_i y_i x_i = 0$$

$$(c)\ \sum_i a_i y_i = 0$$

From (b) we get $\vec{w} = \sum_i a_i y_i x_i$. We now substitute the new (b) in (a), observing that $||w||^2 = w^T w$:

$$L(x, \vec{a}, b) = \frac{1}{2}\sum_i\sum_j a_i a_j y_i y_j x_i x_j - (\sum_i\sum_j a_i a_j y_i y_j x_i x_j - b\sum_i a_i j_i - \sum_i a_i)$$

$$= -\frac{1}{2}\sum_i\sum_i\sum_j a_i a_j y_i y_j x_i x_j - (-b\sum_i a_i y_i - \sum_i a_i)$$

The second term is 0 because of (c), so It can be eliminated, finally we have:

$$L(x, \vec{a}) = \sum_i a_i - \frac{1}{2}\sum_i\sum_j\sum_j a_i a_j y_i y_j x_i x_j$$

such that $\sum_i a_i y_i = 0, 0 \le a_i \le C\ \forall i$.

This is the final equation that we need to maximize over a_i to minimize w and b. To recap, we turned the original optimization problem $min_{w,b}||\vec{w}||^2$ to a problem depending only on lagrangian multipliers, which is faster to compute. We let the computer solve this and get the a_i values, after that we can find w using (b) and b from $y_k = wx_k + b$ for any k and using again w from (b).

Finally, to make predictions, we use the perceptron formula with (b):

$$(d)\ f(x) = \sum_i a_i y_i x_i x + b$$

- each non-zero a_i indicates that the corresponding x_i is a support vector.

8.3 Non linear SVM - Kernel Trick

What if the data is not linearly separable? In such situation we can map data to a higher-dimensional space where the training set is separable.

$$\Phi : X \to H$$

$$h = \phi(x)$$

We notice that the linear classifier (d) relies on the product between x_i and x. We can abstract this product to happen in a higher dimension using a function called Kernel which computes the dot product over some higher-dimensional feature mapping function $\phi(x)$:

$$K(x_i, x_j) = \phi(x_i) \cdot \phi(x_j)$$

Therefore (d) becomes:

$$f(x) = \sum_i a_i y_i K(x_i, x) + b$$

Mercer's Theorem: every positive semidefinite symmetric function is a kernel

There are multiple types of kernels, such as

- linear: $K(x_i, x_j) = x_i^T x_j$
- polinomial of power p: $K(x_i, x_j) = (1 + x_i^T x_j)^p$
- gaussian: $K(x_i, x_j) = e^{\frac{|x_i - x_j|^2}{2\sigma^2}}$

To recap, kernels are generalization of dot products to arbitrary domains.

9 Regularization

A recap on model based machine learning:

1. Pick a model
$$0 = b + \sum_{j=1}^{m} w_i f_i$$
2. Pick a criteria to optimize
$$\sum_{i=1}^{n} \not{1}[y_i(wx_i + b) \leq 0]$$

3. Develop a learning algorithm

$$argmin_{w,b} \sum_{i=1}^{n} \nVdash [y_i(wx_i + b) \leq 0]$$

- repeat: pick a dimension and move a small amount towards the opposite of the derivative

9.1 Regularizer

A *regularizer* is an additional criterion to the loss function to make sure that we do not *overfit*

$$argmin_{w,b} \sum_{i=1}^{n} loss(yy') + \lambda regularizer(w,b)$$

- we want to bias the model so that it prefers certain types of weights over others
- note that is I sum two convex functions, the result is also convex. We want a convex regularizer. If the function is convex, there is usually an easy solution.

Generally, we do not want huge weights: if weights are large, a small change in a feature can result in a large change in the prediction

So, how do we encourage small weights or penalize large weights?

9.2 Common regularizers

Sum of the weights

$$r(w,b) = \sum |w_j|$$

Sum of the squared weights

$$r(w,b) = \sqrt{\sum |w_j|^2}$$

- this penalizes large values more compared to sum of weights

In general, we can call this formula *p-norm* or $L_p(L_1, L_2, ...)$:

$$r(w,b) = \sqrt[p]{\sum |w_j|^p} = ||w||^p$$

9.3 Using Gradient Descent

Example using 2-norm:

$$argmin_{w,b} \sum_{i=1}^{n} exp(-y_i(wx_i + b)) + \frac{\lambda}{2}||w||^2$$

$$\frac{d}{dw_j} equation = -\sum_{i=1}^{n} y_i x_{ij} exp(-y_i(wx_i + b)) + \lambda w_i$$

We can multiply all of this by a constant to control the learning rate.

Note that gradient descent is not the only minimization method.

9.4 Logistic Regression

Log loss or binary cross-entropy loss

$$L(w) = -\sum_{i=1}^{n} [y_i \log(\hat{y}_i) + (1 - y_i)\log(1 - \hat{y}_i)]$$

where

$$\hat{y}_i = \sigma(x_i^T w)$$

and

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

10 Unsupervised Learning

In unsupervised learning we observe data from an unknown distribution without class labels. It is quite useful for the following tasks:

- *dimensionality reduction*: find $f : x \to y$ such that $dim(x) << dim(y)$
- *clustering*: find $f : x \to \mathbb{N}$ that maps an input with a cluster index.
- *density estimation*: find the probability distribution that best fits the data.

A technique that is widely used for dimensionality reduction, lossy data compression, feature extraction and data visualization is called Principal Component Analysis, which we will now discuss.

10.1 Principal Component Analysis

In Principal Component Analysis, or PCA for short, the data is linearly transformed onto a new coordinate system such that the directions (principal components) capturing the largest variation in the data can be easily identified. To reduce dimensionality, we consider k directions with largest variation and project the points onto those.

The idea of the algorithm is the following:

1. Find orthogonal directions of maximum variance (eigenvectors)
2. change coordinate system
3. drop dimensions of least variance

To understand this, first we need to revise some linear algebra theory.

10.2 Recap: Linear algebra

The *variance* is the measure of the deviation from the mean of a point, and is calculated as:

$$Var(X) = \mathbb{E}[(X-\mu)^2] = \frac{1}{n}\sum_{i=1}^{n}(x_i - \mathbb{E}[t])^2$$

The *covariance* is the measure of the linear relationship between two variables with respect to each other and is calculated as:

$$Cov(X,Y) = \mathbb{E}[(X-\mathbb{E}[X])(Y-\mathbb{E}[Y])] = \frac{1}{n}\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})^T$$

You could think of variance as the covariance of a random variable with itself:

$$Var(X) = Cov(X,X)$$

An *eigenvector* or *characteristic vector* is a vector that has its direction unchanged (or reversed) by a given linear transformation. More precisely, an eigenvector v of a linear transformation T is scaled by a constant factor λ when the linear transformation is applied to it: $Tv = \lambda v$. The corresponding *eigenvalue* or *characteristic value* is the multiplying factor λ. Eigenvalues can be found by solving:

$$det(A - \lambda I) = 0$$

and eigenvectors by solving:

$$(A - \lambda I)v = 0$$

Now suppose you have a matrix A with n linearly independent eigenvectors $v_1, v_2, ..., v_n$ which associated eigenvalues $\lambda_1, \lambda_2, ..., \lambda_n$. Define a square matrix Q whose columns are n linearly independent eigenvectors of A, $Q = [v_1 v_2 ... v_n]$. Since each column of Q is an eigenvector of A, right multiplying A by Q scaled each column of Q by its associated eigenvalue: $AQ = [\lambda_1 v_1 \ \lambda_2 v_2 \ ... \ \lambda_n v_n]$. We define a diagonal matrix Λ where each diagonal element Λ_{ii} is the eigenvalue associated with the i-th column of Q. Then:

$$AQ = Q\Lambda$$

therefore:

$$A = Q\Lambda Q^{-1}$$

moreover, if we assume that Q is a unit matrix, meaning $QQ^T = Q^T Q = I$, then $A = Q\Lambda Q^T$. This is referred to as *eigenvalue decomposition*.

10.3 Back to PCA

Armed with this knowledge, we can finally proceed to discuss principal component analysis. The i-th principal component can be taken as a direction orthogonal to the first $i-1$ principal components that maximizes the variance of the projected data. They can also be thought as a sequence of p unit vectors, where the i-th vector is the direction of a line that best fits the data while being orthogonal to the first $i-1$ vectors. A best-fitting line is defined as the one that minimized the average squared perpendicular distance from the points to the line. We will use the first definition now.

Let's take the projection t_i of a point x_i into the unit vector w centered in c:

$$t_i = (x_i - c)^T w$$

Noting that:

$$\mathbb{E}(t) = \sum_{i=1}^{n} t_i = \frac{1}{2} \sum_{i=1}^{n} (x_i - c)^T w = \bar{x}^T w - c^T w \ (1)$$

The variance of such points is then:

$$Var(t) = \frac{1}{n} \sum_{i=1}^{n} (t_i - \mathbb{E}[t])^2 =^{(1)} \frac{1}{n} \sum_{i=1}^{n} [(x_i - \bar{x})^T w]^2 =$$

$$=^{\bar{x}_i =^{def} x_i - \bar{x}} \frac{1}{n} \sum_{i=1}^{n} (\bar{x}_i^T w)^2 = \frac{1}{n} \sum_{i=1}^{n} (w^T \bar{x}_i)^2 =$$

$$\frac{1}{n}\sum_{i=1}^{n}(w^T\bar{x}_i)(\bar{x}_i^T w) = \frac{1}{n}\sum w^T\bar{x}_i\bar{x}_i^T w = w^T[\frac{1}{n}\bar{X}\bar{X}^T]w = w^TCw$$

Where C is the covariance. We are trying to find the direction w that maximizes the variance along said direction:

$$argmax\ w^TCw$$

Such that:

$$w^Tw = 1$$

Let's try to solve this maximization problem using the Lagrangian function:

$$L(w,x) = w^TCw + \lambda(1 - w^Tw)$$

Lets derivate with respect to w and set the derivative equal to 0, noting that $\frac{d}{dw}w^Tw = \frac{d}{dw}||w^2|| = \frac{d}{dw}\sum_i w_i^2 = \sum_i 2w_i = 2w$:

$$\frac{d}{dw}L(w,x) = 2Cw - 2\lambda w = 0$$

Then:

$$Cw = \lambda w$$

We notice that the Lagrangian coefficients are eigenvalues, and w are eigenvectors of the covariance matrix (!!!) It is also true that:

$$w^TCw = \lambda$$

This means that the variance will be maximum then we set w as the eigenvector having the largest eigenvalue λ. This eigenvector is known as the *first principal component.*

- this generalizes to i dimensions

$$w_i \in argmax\{w^TCw : w^Tw = 1, w \perp w_i\ for 1 \leq j < i\}$$

10.4 Principal Component Analysis using eigenvalue decomposition

Algorithm:

- Input: Data points $X = [x_1, ..., x_n]$

- Centering: $\bar{X} = X - \frac{1}{n} X 1_n 1_n^T$
- Compute covariance matrix: $C = \frac{1}{n} \bar{X} \bar{X}^T$
- Eigenvalue decomposition: $Q, \lambda = eig(C)$
- output: Principal components $W = Q = [q_1, ..., q_m]$ and variances $\lambda = (\lambda_1, ..., \lambda_m)$

Now that we can calculate the principal components of some data points, we can take the first k and project the data onto those, with k smaller than the original number of dimensions. We can treat k as a parameter in our training process to find the best value. We could also compute the *cumulative proportion of explained variance*, which is given by $\frac{\sum_{j=1}^{k} \lambda_j}{\sum_{j=1}^{m} C_{jj}}$, to estimate the amount of information loss.

10.5 PCA using Singular Value Decomposition

Here is presented an alternative solution to PCA. Singular Value Decomposition is a factorization o a real or complex matrix into a rotation, followed by a rescaling followed by another rotation.

$$A = USV^T$$

It generalizes eigenvalue decomposition since SVD can be done on every $m \times n$ matrix, whereas eigenvalue decomposition can be applied to square diagonizable matrices. In fact:

$$A^T A = (VS^T U^T)(USV^T) = V(S^T S)V^T = V\frac{C}{n}V^T$$

We have a similar function as before, the variance again, and V are the eigenvectors.

Algorithm:

- Input: Data points $X = [x_1, ..., x_n]$
- Centering: $\bar{X} = X - \frac{1}{n} X 1_n 1_n^T$
- SVD decomposition: $U, S, V = SVD(\bar{X})$
- Output: Principal components $U = [u_1, ..., u_k]$ and variances $(\frac{s_i^2}{n}, ..., \frac{s_k^2}{n})$ since $C = \frac{1}{n} \bar{X} \bar{X}^T = \frac{1}{n} USV^T USV^T = U\frac{S^2}{n} U^T$

10.6 Kernel PCA (KPCA)

By using the kernel trick one can apply PCA in a higher dimensional space, yielding a non-linear transormation in the original space

10.7 Other dimensionality reduction techniques

- PCA: find projection that maximize the variance
- Multidimensional Scaling: find projection that best preserves inter-point distances
- LDA (Linear Discriminan Analysis): Maximizing the component axes for class-separation.

10.8 Limitations

- the data must be linearly separable (which is a strong assumption)

11 Clustering

Given unlabeled data, we want to group them into clusters. The first thing to think about is how should these clusters be represented. We may want to model a cluster as a circumference where all the data inside It belongs to the same cluster, or maybe an ellipse may be a better choice for some problems. Some other data may need a completely another structure. Ultmatively, how we represent the cluster will characterize the algorithm that we will develop. In this chapter we will discuss four techniques to achive this.

Some complications rise when discussing representation of data, how to measure similarity or distance, is it a flat clustering or hierarchical and is the number of clusters known a priori.

Let's start by categorizing *hard clustering* where each example belongs to exactly one cluster, and *soft clustering* where an example can belong to more than one cluster (probabilistic).

11.1 K-means clustering

First, we are assuming that we know the number of clusters. This is a strong assumption to make and may be impossible for some problems. Our goal is then to find an assignment of data points to clusters, as well as a set of vectors $\{u_k\}$, such that the sum of the squares of the distances of each data point to its closest vector u_k, is a minimum. We can think of u_k as the center of a cluster C.

$$min_{C_1,\ldots,C_k} \sum_{j=1}^{k} V(C_j)$$

- the variation is typically given by $V(C_j) = \sum_{i \in C_j} ||x_i - u_j||^2$ which is the euclidean distance squared.
- the *centroid* is computed with $u_j = \mathbb{E}_{c \in C}[x \in C] = \frac{1}{|C_j|} \sum_{i \in C_j} x_i$

11.1.1 The algorithm

```
Initialization: Use some initialization strategy to get
some initial cluster centroids u1,...,uk

while clusters change, do
    Assign each datapoint to the closest centroid forming
    new clusters

        Cj = i in N such taht j = argmin_l(xi - ul)

    Compute cluster centroids u1,...,uk
```

The algorithm is guaranteed to converge, but not to find the global minimum.

11.1.2 Initial Centroid Selection

- random selection
- points least similar to any existing center
- try multiple starting points

11.1.3 Running time

- assignment: $O(kn)$ time
- centroid computation: $O(n)$ time

In this case, euclidian distance is not the best so we use *consine similarity*

$$sim(x, y) = \frac{x * y}{|x||y|}$$

11.2 Expectation Maximization

If we restrain our problem by assuming that the clusters are formed as a mixture of Gaussians (elliptical data) and we assign data to a cluster with a certain probability (soft clustering) we can use the *Expectation Maximization* (EM) algoritm. The algorithm proceeds as follows:

- Start with some initial cluster centers
- Iterate
  - Soft assign points to each cluster by calculating the probability of each point belonging to each cluster $p(\theta_c|x)$.
  - Train: Recalculate the cluster centers by calculating the maximum likelihood cluster centers given the current soft clustering θ_c.

A Gaussian (ellipse) in 1 dimension is defined as:

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}}e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

A Gaussian in n dimensions is defined as:

$$N[x;\mu,\Sigma] = \frac{1}{(2\pi)^{d/2}\sqrt{det(\Sigma)}}exp[-\frac{1}{2}(x-\mu)^T\Sigma^{-1}(x-\mu)]$$

We learn the means of each cluster (i.e. the center) and the covariance matrix (i.e. how spread out it is in any given direction).

Intuitively, each iteration of Expectation Maximization increases the likelihood of the data and is guaranteed to converge (though to a local optimum).

11.3 Spectral Clustering

In case of non-gaussian data, we can use a technique called *spectral clustering* where we group points based on links in a graph.

We first create a fully connected graph or a k-nearest neighbor graph (each node is only connected to its K closest neighbors). Alternatively, we can create a graph with some notion of similarity among nodes, for example using a gaussian kernel: $W(i,j) = exp[\frac{-|x_i-x_j|^2}{\sigma^2}]$.

We then partition the graph by cutting the graph: we want to find a partition of the graph into two parts A and B where we minimize the sum of the weights of the set of edges that connect the two groups, aka $Cut(A,B)$. We observe that this problem can be formalized as a discrete optimization problem and then relaxed in the continuous domain and become a generalized eigenvalue problem.

11.4 Hierarchical Clustering

We are interested in producing a set of nested clusters organized as a hierarchical tree, also called *dendrogram.*

The idea of the algorithm is to first mege very similar instances and then incrementally build larger clusters out of smaller clusters. The algorithm goes as follows:

- Initially, each instance has its own cluster
- Repeat
  - Pick the two closest clusters
  - Merge them into a new cluster
  - Stop when there is only one cluster left

12 Neural Networks

Previously we have discussed perceptrons as a non-linear parameterized function with restricted output range. It predicts the output by learning the weights while considering a bias, more specifically it associates one weight per input and multiplies them together.

$$a = h(\sum_i w_i x_i + b)$$

Perceptrons are great and easy to understand, however they assume the input data is linearly separable. Indeed, in 1969 Minsky and Papert showed that an exclusive OR (XOR) cannot be solved with a perceptron. This limitation seemd to halt the development of the field, and commenced what is now known as the "AI Winter". Only in 1986 progress was made with a solution that would change the world: *multi-layer perceptrons.*

In essence, multi-layer perceptrons are a set of densly connected perceptrons (aka neurons) organized in layers. Input goes from the first layer to the next until the last one where the output is collected, if each neuron is connected to all the neurons in the previous layer then we refer to the network as a *fully connected network.* This design was inspired by neuroscience by how the brain manages to understand complex input through billions and billions of connections between neurons.

Let's now further formalize a multi-layer perceptron, aka a *neural network.* Each neuron belongs to a layer $l \in \{0, ..., L\}$ where 0 is the *input* layer and L is the *output* layer. All the layers in between are called *hidden layers.* The symbol $z_i^{(l)}$ indicates the output of the i-th neuron in the layer l. A single neuron on layer l takes the results $z_i^{(l-1)}$ from the i nodes of the $(l-1)$ layer and sums them, multiplying them by the weights $w_{ij}^{(l)}$, and adds a bias b_l. We call this value $a_i^{(l)}$:

$$a_i^{(l)}(x,w) = \sum_j w_{ij} z_j^{(l-1)}(x,w) + b_l$$

The node then passes this result to a non-linear activation function $h(x)$, we call this number $z_i^{(l)}$:

$$z_i^{(l)}(x,w) = h(a_i^{(l)}(x,w))$$

There are many non-linear activation function, a popular one is called *Rectified Linear Units* aka ReLU and It looks like this:

$$ReLU(x) = max(0,x)$$

The output layer is usually multiplied by a σ function that transforms the outputs to a probability or smooth the value, such as the *sigmoid* function:

$$\sigma(x) = \frac{1}{1 + exp(-x)}$$

or the *softmax* function:

$$\sigma(z)_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$$

For example, a multi layer perceptron with one input, one hidden and one output layer takes the form:

$$y_k(x,w) = \sigma(\sum_{j=2}^{M} w_{kj}^{(2)} h(\sum_{i=1}^{D} w_{ji}^{(1)} x_i + b_1) + b_2$$

The process of evaluating the previous formula can be interpreted as *forward propagation* of information through the network, for this reason we also call this types of networks a *feed forward* network.

We know how to find the output from the inputs, we will now discuss how to train the network. We cannot train the multi player perceptron like we have seen for a single perceptron because we do not know the desired target output for the hidden layers. We will see how this is solved through a technique called *backpropagation*.

12.1 Training with backpropagation

Given training samples $T = \{(x_1, y_1), (x_2, y_2), ..., (x_n, y_n)\}$, we want to adjust all the weights of the network Θ such that an error (also called loss) function $E(\cdot)$ is minimized.

$$min_\Theta \sum_i E^{(L)}(y_i, f(x_i; \Theta))$$

Where y_i is the expected output of the i-th neuron. There are many choices for an error function, such as square loss:

$$E(x,w)^{(L)} = \frac{1}{2}\sum_i (y_i - z_i^{(L)}(x,w))^2$$

or cross entropy:

$$E(x,w)^{(L)} = -\sum_i y_i \log(z_i^{(L)}(x,w))$$

For training will again use gradient descent, however we need a way to propagate the error in order to compute the gradient of the hidden layers. To do so we use an algorithm called backpropagation, which is composed of the following steps:

1. *Feedforward propagation*: Accept input x_i, then pass through the intermediate stages and obtain the output.
2. Use the computed output to compute a *scalar cost* depending on the loss function.

In order to update the weights based on the error, we want to find the partial derivative of the error with respect to a weight w_{ij}, and we write $\frac{\partial E(x,w)^{(l)}}{\partial w_{ij}}$. We can apply the chain rule $D(f(g(x))) = f'(g(x))g'(x)$ which can be rewritten as $\frac{\partial f}{\partial g} = \frac{\partial f}{\partial g}\frac{\partial g}{\partial x}$ to the error function:

$$\frac{\partial E^{(L)}}{\partial w_{ij}} = \frac{\partial E^{(l)}}{\partial z_i^{(l)}}\frac{\partial z_i^{(l)}}{\partial a_i^{(l)}}\frac{\partial a_i^{(l)}}{\partial w_{ij}}$$

Let's analyze all three of the factors:

$$\frac{\partial E^{(L)}}{\partial z_i^{(l)}} = \frac{\partial}{\partial z_i^{(l)}}\frac{1}{2}\sum_i (y_i - z_i^{(l)}(x,w))^2 = -(y_i - z_i^{(l)})$$

$$\frac{\partial z_i^{(l)}}{\partial a_i^{(l)}} = \frac{\partial h}{\partial a_i^{(l)}}$$

$$\frac{\partial a_i^{(l)}}{\partial w_{ij}} = \frac{\partial}{\partial w_{ij}} \sum_j w_{ij} z_j^{(l-1)}(x, w) = z_j^{(l-1)}$$

So overall, for the first layer you can use:

$$\frac{\partial E^{(L)}}{\partial w_{ij}} = -\frac{\partial h}{\partial a_i^{(l)}} (y_i - z_i^{(l)}) z_j^{(l-1)}$$

Where the derivate of the activation function depends on the function. $(y_i - z_i^{(l)})$ is often referred to as $\delta_i^{(l)}$.

For the hidden layers you do not know the expected output y_i directly, so you use the formula:

$$\frac{\partial E^{(l)}}{\partial w_{ij}} = -\frac{\partial h}{\partial a_i^{(l)}} (\sum_j \delta_i^{(l+1)} w_{ij}^{(l+1)}) z_j^{(l-1)}$$

We can finally perform gradient descent like we discussed in Its own chapter.

12.2 Example

We will relax the notation now. Consider a two-layer network (that does not count the input layer). Given:

$$h(a) \equiv tanh(a)$$

$$tanh(a) = \frac{e^a - e^{-a}}{e^a + e^{-a}}$$

$$h'(a) = 1 - h(a)^2$$

$$E^{(L)} = \frac{1}{2} \sum_i (y_i - z_i^{(L)})^2$$

where $z_i^{(L)}$ is the activation of output unit i, and y_i is the corresponding target, for a particular input pattern x_n.

For each pattern in the training set in turn, we first perform forward propagation using:

$$a_i^{(0)} = \sum_{i=1}^{D} w_{li}^{(0)} x_i$$

$$z_i^{(0)} = tanh(a_i^{(0)})$$

$$a_j^{(1)} = \sum_{j=1}^{M} w_{ij} z_i^{(0)}$$

Next we compute the δ's for each output unit using:

$$\delta_i^{(1)} = y_i - z_i^{(1)}$$

Then we backpropagate there to obtain δs for the hidden units using:

$$\delta_j^{(0)} = (1 - (z_j^{(1)})^2)(\sum_{k=1}^{K} w_{kj} \delta_k^{(1)}) z_k^{(0)}$$

Finally, the derivatives with respect to the first-layer and second-layer weights are given by:

$$\frac{\partial E_i^{(1)}}{\partial w_{ji}} = \delta_i^{(1)} z_j^{(0)}, \; \frac{\partial E_j^{(0)}}{\partial w_{kj}} = \delta_k^{(0)} x_j$$

To summarize, with a simpler notation:

1. Apply an input vector x_n to the network and forward propagate through the network using $a_j = \sum_i w_{ji} z_i$ and $z_j = h(a_j)$ to find the activations of all the hidden and output units
2. Evaluate the δ_k for all the output units using $\delta_k = y_k - z_k$
3. Backpropagate the δ's using $\delta_j = h'(a_j) \sum_k w_{kj} \delta_k$ to obtain δ_j for each hidden unit in the network
4. Use $\frac{\partial E_n}{\partial w_{ji}} = \delta_j z_i$ to evaluate the required derivatives

12.3 Convolutional Neural Networks

In computer vision, we have a lot of important spacial information we want to model, however this happens to be really difficult for neural network as were described previously. The problem raises from the fact that it is difficult to pass an image as input to a neural network. One possibility would be to set an input node for each pixel, however this has many problems primarily because the input layer would be huge (a 500 by 500 image will have 250000 inputs) and all the spacial information would be lost, like being able to encode what is nearby a pixel. For these reasons, a special solution was needed.

By the same time backpropagation was discovered, *convolutional neural networks* were proposed as a solution to the problem inspired by the mammalian visual cortex. This type of neural networks use convolution in place of general matrix multiplication in at least one of their layers. A convolution is a general purpose filter operation for image where a kernel matrix is applied to an image and the central pixel is determined by adding the weighted values of all its neighbors together, producing a *feature map*.

$$S(i,j) = (I * K)(i,j) = \sum_{m=-i}^{i} \sum_{n=-j}^{j} I(m,n)K(i-m,j-n)$$

These convolutions matrices are learned by the convolutional neural network, this is how the It can understand spacial features such as edges or certain arrangements of pixels on Its own. Suppose we are training a CNN to classify human faces, the network may learn the features of the eyes and their position with respect to the nose and the mouth, It may catch the shape of the ears or the color of the skin, and so on.

Convolutional neural networks are feedforward neural networks composed of a set of filters that cover a spacially small portion of the input data and that are convoluted over It. The network may have many convolutions over the same input, and convolutions over convolutions. For classification, It is common to have multiple convolutions chained together while reducing the spacial size of the representation and increasing Its depth with a process known as *spacial polling*, until the size is 1. Then, all the data is fed to a traditional neural network for classification.

12.4 Other neural networks

Many other types of neural networks were developed over time for domain specific problems, some of the notable ones are:

- *Recurrent network*: nodes on the same layer influence each other, used in video frame prediction.
- *Autoencoders*: unsupervised approach for learning a lower-dimensional feature representation from unlabeled training data. Additionally, we can train a decoder to do the opposite. We will talk more about this in the following chapter.

13 Deep Generative Models

13.1 Definitions

13.1.1 Generative Models

Generative models are statistical models of the data distribution p_X or p_{XY}, depending on the availability of target data, where Y is the label variable.

13.1.2 Discriminative models

Discriminative models are statistical models of the conditional distribution $p_{Y|X}$.

Note that a discriminative models can be constructed from a generative model via Bayes rule but not viceversa.

$$p_{Y|X}(y|x) = \frac{p_{XY}(x,y)}{\sum_{y'} p_{XY}(x,y') = p_X(x)}$$

13.2 Density Estimation

Explicit problem: Find a probability distribution $f \in \Delta(Z)$ that fits the data, where $z \in Z$ is sampled from an unknown data distribution $p_{data} \in \Delta(Z)$.

Implicit problem: Find a function $f \in Z^{\Omega}$ that generates data $f(\omega) \in Z$ from an input w sampled from some predefined distribution $p_\omega \in \Delta(\Omega)$ in a way that the distribution of generated samples fits the (unknown) data distribution $p_{data} \in \Delta(Z)$.

In the case of supervised learning, $Z = X \times Y$ while in the case of unsupervised learning, $Z = X$. We will discuss this last case.

The objective is to define an hypothesis space $H \subset \Delta(Z)$ of models that can represent probability distributions and a divergence measure $d \in \mathbb{R}^{\Delta(z)\times\Delta(Z)}$ between probability distributions in $\Delta(Z)$. Then find the hypothesis $q* \in H$ that best fits the data distributed according to p_{data}:

$$q* \in arg\ min\ d_{q\in H}(p_{data}, q)$$

13.2.1 KL-Divergence

Before continuing, It is worth taking some time to study KL-divergence since we will use this later. The Kullback–Leibler divergence, denoted as $d_{kl}(P|Q)$ is a type of statistical distance that measures how much a model or theory probability distribution Q is different from a true probability distribution P. It is defined as:

$$d_{kl}(P|Q) = \sum_{x \in X} P(x) \log \frac{P(x)}{Q(x)} = \mathbb{E}[\log \frac{P(x)}{Q(x)}]$$

A simple interpretation of the KL divergence of P from Q is the expected excess surprise from using Q as a model instead of P when the actual distribution is P. If P and Q are the same distribution, the KL-divergence is 0.

13.2.2 Jensen's Inequality

Jensen's inequality generalizes the definition of convex function as a function that satisfies the following inequality:

$$f(\theta x_i + (1-\theta)x_2) \leq \theta f(x_i) + (1-\theta) f(x_2)$$

In the context of probability theory, this is stated as:

$$\phi(\mathbb{E}[X]) \leq \mathbb{E}[\phi(X)]$$

where ϕ is a convex function.

13.3 Autoencoders

The basic idea behind autoencoders is to encode information (as in compress) automatically, hence the name. Autoencoders are neural network with the smallest layer at the center, and are symmetrical on the left and right. We call the part of the network from the input to the center an *encoder*, we call the other part a *decoder*. The network can be trained with backpropagation by feeding input and setting the error to be the difference between the input and what came out.

The encoder can also be used for dimensionality reduction without the encoder, or as a supervised model. It will reduce the input to a much smaller latent space where another neural network can be connected to.

13.4 Variational AutoEncoders (VAE)

Variational Autoencoders add a “probabilistic spin” to traditional Autoencoders by finding a probability distribution in the latent space and using this to decode another probabilistic distribution that best matches the input's distribution.

Let ω be the distribution in the *latent space*, $q_\theta(\omega|x)$ be an *encoder* distribution and $q_\theta(x|\omega)$ be a *decoder* distribution. Then the objective is:

$$\theta^* \in arg\ min_{\theta\in\Theta}\ d(p_{data}, q_\theta)$$

$$(1)\ q_\theta(x) = \mathbb{E}_{\omega\sim p^\omega}[q_\theta(x|\omega)]$$

The two formulas above are really important. We want to find the best parameters that minimize the distance between the predicted q_θ distribution and the actual real distribution p_{data}. We find $q_\theta(x)$ by taking the expected value of the decoder function $q_\theta(x|\omega)$ where ω is the distribution in the latent space.

As the divergence measure we will use the KL-Divergence:

$$d_{KL}(p_{data}, q_\theta) = \mathbb{E}_{x\sim p_{data}}[\log \frac{p_{data}(x)}{q_{\theta(x)}}] = -\mathbb{E}_{x\sim p_{data}}[\log\ q_\theta(x)] + const$$

$$=^{(1)} -\mathbb{E}_{x\sim p_{data}}[\log\ \mathbb{E}_{\omega\sim p_\omega}[q_{\theta(x|\omega)}]] + const$$

Let's focus on the argument of the outer expected value. Let $q_\psi(\omega|x) \in \Delta(\Omega)$ denote an encoding probability distribution with parameters ψ. We can change the expected value measure to this probability like this:

$$\log\ \mathbb{E}_{\omega\sim p_\omega}[q_\theta(x|\omega)] = \log\ \mathbb{E}_{\omega\sim q_\psi(\cdot|x)}[q_\theta(x|\omega)\frac{p_\omega(\omega)}{q_{\psi(\omega|x)}}]$$

Thanks to the Jensen's inequality we can write:

$$\geq \mathbb{E}_{\omega\sim q_\psi(\cdot|x)}[\log(q_\theta(x|\omega)\frac{p_\omega(\omega)}{q_{\psi(\omega|x)}})] =$$

$$= \mathbb{E}_{\omega\sim q_\psi(\cdot|x)}[\log\ q_\theta(x|\omega)] - d_{KL}(q_\psi(\cdot|x), p_\omega)$$

Those two last terms are the *Reconstruction* and the *Regularizer*. The first is still NP-hard to compute but we can estimate the gradients of the parameters, while the regularizer might have closed-form solution (for example, using Gaussian distribution).

In practice training is divided into several steps: first data samples are fed to an encoder which can be something like a normal neural network or a convolutional one. This encoder will reduce the dimensionality of the input into what's called a "latent space" which is composed of fewer nodes than the input layer. We are interested in the distribution of this latent space, so we assume that Its distribution follows a gaussian and we compute the mean and the covariance. We then feed this statistical model to the decoder which generates again some data. Finally, we compare the original samples with the generated data via a loss function and update the weights of both the encoder and decoder to minimize the difference.

13.5 Issues with VAE

The problem with this approach is underfitting, since at initial stages the regularizer is too strong and tends to annihilate the model capacity, and blurry output data.

A modern approach to image generation is Vector Quantized VAE (VQ-VAE-2) which is a image synthesis model based on Variational Autoencoders. It produces images that are high quality by leveraging a *discrete latent space*.

13.5.1 Conditional VAE

Variatonal Auto Encoders do not need lables. Conditional VAE are a variation of VAE that accept labels.

Assume we have side information $y \in Y$ (e.g. digit labels, attributes, etc) and we want to generate new data conditioned on the side information (e.g. generate digit 7, or generate a face with glasses).

Modify the encoder and decoder to take the side information in input obtaining $q_\psi(\omega|x, y)$ and $q_\theta(x|\omega, y)$.

Define priors conditioned on side information $p_\omega(\omega|y)$.

13.6 Generative Adversial Networks (GAN)

VAE are able to find explicit densities, GAN enables the possibility to find implicit ones.

GAN models are composed by two "adversarial" submodels: a *generator* and a *discriminator*. The term adversarial means that the two submodels are in competition and there is one winner (a zero sum game).

The generator is tasked to generate fake images, while the discriminator is tasked to recognize if an image is fake or not. The generator generates images for the discriminator to check, mixing real images with generated ones. If the generator fools the discriminator, than the latter needs to update Its weights, otherwise the opposite will happen.

GANs enable the possibility of estimating implicit densities. We assume to have a prior density $p_\omega \in \Delta(\Omega)$ given and a generator (or decoder) $g_\theta \in X^\Omega$ that generates data points in X given a random element from Ω.

The density induced by the prior p_ω and the generator g_θ is given by $q_\theta(x) = \mathbb{E}_{\omega \sim p^\omega} \delta[g_\theta(\omega) - x]$, where δ is the Dirac delta function. The Dirac function is a generalized function on the real numbers, whose value is zero everywhere except at zero, and whose integral over the entire real line is equal to one.

The (original) GAN objective is to find θ^* such that q_{θ^*} best fits the data distribution p_{data} under the Jensen-Shannon divergence:

$$d_{JS}(P,Q) = \frac{1}{2} d_{kl}(P|M) + \frac{1}{2} d_{kl}(Q|M),\ M = \frac{1}{2}(P+Q)$$
$$\theta^* \in arg\ min_\theta\ d_{JS}(p_{data}, q_\theta)$$

where

$$d_{JS}(p,q) = \frac{1}{2} d_{KL}(p, \frac{p+q}{2}) + \frac{1}{2} d_{KL}(q, \frac{p+q}{2})$$
$$= \frac{1}{2}\mathbb{E}_{x\sim p}[\log \frac{2p(x)}{p(x)+q(x)}] + \frac{1}{2}\mathbb{E}_{x\sim q}[\log \frac{2q(x)}{p(x)+q(x)}]$$
$$= \frac{1}{2}\mathbb{E}_{x\sim p}[\log \frac{p(x)}{p(x)+q(x)}] + \frac{1}{2}\mathbb{E}_{x\sim q}[\log \frac{q(x)}{p(x)+q(x)}] + \log(2)$$
$$= \log(2) + \frac{1}{2} max_t\ \{\mathbb{E}_{x\sim p}[\log\ t(x)] + \mathbb{E}_{x\sim q}[\log(1-t(x))]\}$$

The interpretation of the JS divergence is that this mixes how the two distributions depend on one another, not just one over the other; It is useful when there is not a true reference distribution.

Let $t_\phi(x)$ be a binary classifier (or discriminator) for data point in the training set predicting whether x came from p or q, we get the following lower bound on our objective:

$$d_{JS}(p_{data}, q_\theta) = \log(2) + \frac{1}{2} max_t\ \{\mathbb{E}_{x\sim p}[\log\ t(x)] + \mathbb{E}_{x\sim q}[\log(1-t(x))]\}$$
$$\geq \log(2) + \frac{1}{2} max_\phi\ \{\mathbb{E}_{x\sim p}[\log\ t_\phi(x)] + \mathbb{E}_{x\sim q}[\log(1-t_\phi(x))]\}$$

Which is minimized to obtain the generator's parameters:

$$\theta^* \in argmin_\theta\ max_\phi\{\mathbb{E}_{x\sim p}[\log\ t_\phi(x)] + \mathbb{E}_{x\sim q}[\log(1-t_\phi(g_\theta(x)))]\}$$

In practice, during training both real data and generated data are passed to a classifier that estimates if the data is real or generated with the $t\phi(x)$

function. The generator and the discriminators are opponents and this is modeled mathematically via the arg min-max function: the generator tries to minimize the parameters θ and the classifier tries to maximize the parameter ϕ to solve the equation.

14 Diffusion models

To recap, generative models train a generator G from latent space to data space and approximate the real data distribution. Variational Autoencoders have the ability to generate new samples to regular Autoencoders and use a probabilistic latent space (assumed to be a multivariate Gaussian), while GAN have a generator that starts from Gaussian noise and generates a data point in order to fool the discriminator.

14.1 Denoising diffusion

Denoising diffusion models consists of two processes:

- forward process to add noise
- reverse precess denoises to generate data

A *Markov chain* or Markov process 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.

$$Pr(X_{n+1} = x | X_1 = x_1, X_2 = x_2, ..., X_n = x_n) = Pr(X_{n+1} = x | X_n = x_n)$$

14.2 Forward Process

A forward process gradually adds noise to the images over T timesteps.

$$x_0 \rightarrow x_1 \rightarrow x_2 \rightarrow ... \rightarrow x_T$$

The model will also be tasked to undo this noise called the “reverse” prorocess. Often the forward process is fixed and the reverse process needs to be trained.

A forward process has the following formulation:

$$q(x_t | x_{t-1}) = N(x_t; \sqrt{1 - \beta_t} x_{t-1}, \beta_t I) \Rightarrow q(x_{1:T} | x_0) = \prod_{t=1}^{T} q(x_t | x_{t-1})$$

Notice that we are multiplying the mean of the previous distribution times $\sqrt{1-\beta_t}$ and the variance by β_t, the two transformations are related in order to keep the distribution normalized. β can be thought as the step size, It determins how aggressively the noise build up. Additionally, we are using I as the base variance that is the identity matrix where all the values are completely independent for eachother.

Let $\alpha_t = (1-\beta_t)$, then

$$\bar{\alpha}_t = \prod_{s=1}^{t}(1-\beta_s) \Rightarrow q(x_t|x_0) = N(\sqrt{\bar{\alpha}_t}x_0, (1-\bar{\alpha}_t)I)$$

We can then sample directly at the desired timestep:

$$x_t = \sqrt{\bar{\alpha}_t}x_0 + \sqrt{(1-\bar{\alpha}_t)}\epsilon, \ \epsilon \sim N(0, I)$$

Formally we are applying a Gaussian convolution to the data at each timestep. Practically we are smoothening out the distribution to a Gaussian one.

A backward (denoising) process has the form:

$$p_\theta(x_{t-1}|x_t) = N(x_{t-1}; \mu_\theta(x_t, t), \sigma_t^2 I)$$
$$p_\theta(x_{0:T}) = p(x_T)\prod_{t=1}^{T} p_\theta(x_{t-1}|x_t)$$

14.3 Generation Process

Given that $q(x_T) \approx N(0, I)$, a sample is $x_T \sim N(0, I)$ and iteratively $x_{t-1} = q(x_{t-1}|x_t)$. Through Bayes theorem, $q(x_{i-1}|x{+}t) \propto q(x_{t-1})q(x_t|x_{t-1})$ but we cannot solve this since we don't know $q(x|x_{t-1})$. In other words, $p(x_{i-1}|x)$ depends on the entire Markov chain which we do not have. To solve this, we can approximate It by learning a Gaussian.

- $p(x_T) \sim N(0, I)$
- $p_\theta(x_{t-1}|x_t) \sim N(\mu_\theta(x_t, t), \sigma^2 I)$
- then $p_\theta(x_{0:T}) = p(x_T)\prod_{t=1}^{T} p_\theta(x_{t-1}|x_t)$

Hence, we need to find the parameters θ that approximate $p(x_{t-1}|x_t)$.

14.4 Noising schedule

We can control the variance of the forward diffusion and reverse denoising processes respectively. Often a linear schedule is used for β_t and σ_t^2 is set equal to β_b.

Kingma et al introduce a new parametrization of diffusion models using signal-to-noise ratio (SNR), and show how to learn the noise schedule by minimizing the variance of the training objective.

14.5 Connection to VAE, GANs

- Latent variables have the same dimensionality of data.
- The same model is applied across different timesteps.
- The model is trained by revenging the variational bound.

14.6 Training Parametrisation

We can train the model in a similar fashion as VAE, with a Variational Upper Bound:

$$L = \mathbb{E}_{q(x_0)}[-\log p_\theta(x_0)] \leq \mathbb{E}_{q(x_0)q(x_{1:T}|x_0)}[-\log \frac{p_\theta(x_{0:T})}{q(x_{1:T}|x_0)}]$$

These can be divided into three terms:

$$L = \mathbb{E}_q[D_{KL}(q(x_T|x_0)||p(x_T)) \\ + \sum_{t>1} D_{KL}(q(x_{t-1|x_t}, x_0)||p_\theta(x_{t-1}|x_t)) - \log\ p_\theta(x_0|x_1)]$$

- the first term is fixed
- the second term is just summing gaussians

KL between Gaussians has a nice closed form, but Ho (with some math) proves the training can be simplified to a noise prediction problem, obraining a new loss:

$$L_{simple} = \mathbb{E}_{x_0 \sim q(x_0), \epsilon \sim N(0,I), t \sim U(1,T)}[||\epsilon - \epsilon_\theta(\sqrt{\bar{\alpha}_t}x_0 + \sqrt{(1-\bar{\alpha}_t)}\epsilon, t)||]$$

Training Algorithm:
repeat

- $x_0 \sim q(x_0)$

- $t \sim Uniform(\{1, ..., T\})$
- $\epsilon \sim N(0, I)$
- Take gradient descent step on $\nabla_\theta ||\epsilon - \epsilon_\theta(\sqrt{\bar{\alpha}_t} x_0 + \sqrt{1 - \bar{\alpha}_t}\epsilon, t)||^2$

until converged
Sampling algorithm:

- $x_T \sim N(0, 1)$
- for $t = T, ..., 1$ do
  - $x \sim N(0, I)$
  - $x_{t-1} = \frac{1}{\sqrt{\alpha_t}}(x_t - \frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}_t}}\epsilon_\theta(x_t, t)) + \sigma_t z$
- end for
- return x_0

The choice of the architecture is free. For images use U-NET.

14.7 U-NET

The U-NET architecture contains two paths.

- The first path is the contraction path (also called as the encoder) which is used to capture the context in the image. The encoder is just a traditional stack of convolutional and max pooling layers. The encoder extracts increasingly abstract features by applying convolutions and downsampling. At each level the spatial size decreases while the number of feature channels increases and allow the model to capture higher-level patterns.
- The second path is the symmetric expanding path (also called as the decoder) which is used to enable precise localization using transposed convolutions. The decoder begins to reconstruct the original image size through upsampling. At each level it combines decoder features with corresponding encoder features using skip connections to retain fine-grained spatial details.
- It is an end-to-end fully convolutional network, i.e. It only contains Convolutional layers and does not contain any Dense layer, for this reason it can accept image of any size.

14.8 Generative Trilemma

Often fast sampling, mode coverage / diversity and high quality samples are difficult to coexist together.

- GAN have fast sampling with high quality samples but not mode coverage / diversity.
- Likelihood-based models (Variational Autoencoders and Normalizing flows) offer fast sampling and mode coverage/diversity but not high quality samples.
- Denoising diffusion models have mode coverage/diversity and high quality samples but not fast sampling.

14.9 Diffusion GANs

Generative denoising diffusion models typically assume that the denoising distribution can be modeled by a Gaussian distribution. This assumption holds only for small denoising steps, which in practice translates to thousands of denoising steps in the synthesis process. In diffusion GANs, the denoising model is represented using multimodal and complex conditional GANs, enabling to efficiently generate data in a few steps. In other words, instead of working with Gaussians, we work with more complicated functions.

Compared to a one-shot GAN generator:

- Both generateor and discriminator are solving a much simpler problem
- Stronger mode coverage
- Better training stability

14.9.1 Distillation

Distill a deterministic DDIM (Denoising Diffusion Implicit Model) sampler to the same model architecture. At each stage, a *student* model is learned to distill two adjacent sampling steps of the *teacher* model to one sampling step. At next stage, the “student” model from previous stage will serve as the new “teacher” model. This method allows to “skip” some stages in order to speedup computation.

14.10 Latent-space diffusion models

The distribution of latent embeddings is close to Normal distribution, giving simpler denoising and faster synthesis. They allow augmented latent space and tailored autoencoders (graphs, text. 3D data, etc).

14.11 Text-to-image: CLIP (OpenAI)

Jointly train a text encoder and an image encoder. Training by maximising the similarity between embeddings of (text, image) pairs. The resuling space has semantics for both images and text.

14.12 Diffusion usages

- super resolution
- image-to-image (color a black and white image, extend and image's borders)
- semantic segmentation
- image editing (add something to a portion of the image)
- video generation
- 3d shape generation

15 Reinforcement Learning

Reinforcement learning in inspired by research on psychology and animal learning. The problems involve an agent interacting with an environment, which provides numeric reward signals.

The model is asked to take actions with an effect on the state of the environment in order to maximize reward.

15.1 Markov Decision Process

Markov Decision Process (MDP) is a framework used to help to make decisions on a stochastic environment. Our goal is to find a policy, which is a map that gives us all optimal actions on each state on our environment,

In order to solve MDP we use Bellam equation which divides a problem into simpler sub-problems easier to solve (Dynamic Programming).

The components of MDP include:

- *states* s, beginning with initial state s_0
- *Actions* a
- *Transition model* $P(s'|s,a)$
- *Markov assumption*: the probability of going to s' from s depends only on s and not on any other past actions or states
- *Reward function* $r(s)$
- *Policy* $\pi(s)$: the action that an agent takes in any given state

Therefore, MDP is defined by:

$$(S, A, R, P, \gamma)$$

- S set of possible states
- A set of possible actions
- R distribution of reward given (state, action) pair
- P transition probability i.e. distribution over next state given (state, action) pair
- γ discount factor

In a loop, the agent selects an action a_t and receives a reward r_t and the next state s_{t+1}. A policy $\pi(s)$ is a function from S to A that specifies what action to take in each state. The objective is to find policy π^* that maximized cumulative discounted reward $\sum_{t\geq 0} \gamma^t r(s_t)$ where γ is the discounting factor. This controls the importance of the future rewards versus the immediate ones: It will make the agent optimize for short term or long term actions.

15.2 Loop

In Supervised learning: given an input x_i sampled from data distribution, we use the model with parameters w to predict output y, then calculate the loss and update w to reduce loss with gradient descent $w = w - \eta \nabla l(w, x_i, y_i)$. Note that the next input does not depend on previous inputs or agent prediction and loss is differentiable.

In Reinforcement Learning instead, given a state s, we take an action a determined by policy $\pi(s)$. The environment selects next state s' based on transition model $P(s'|s, a)$ and gives a reward $r(s)$ while setting the new state s'. Rewards are not differentiable w.r.t. model parameters and the agent's actions affect the environment and help to determine next observation.

There are two main approaches for Reinforcement Learning:

- Value-based methods: the goal of the agent is to optimize the value function $V(s)$ where the value of each state is the total amount of the reward an RL agent can expect to collect over the future from a given state.
- Policy-based approach: we define a policy which we need to optimize directly.

15.3 Value Based Methods

15.3.1 Value Function

The *value function* gives the total amount of rewards the agent can expect from a particular state to all possible states from that state.

$$V^{\pi}(s) = \mathbb{E}[\sum_{t \geq 0} \gamma^t r(s_t) | s_0 = s, \pi]$$

The optimal value of a state is the value achievable by following the best possible policy:

$$V^{*}(s) = max_{\pi} \mathbb{E}[\sum_{t \geq 0} \gamma^t r(s_t) | s_0 = s, \pi]$$

It is more convenient to define the value of a state-action pair, called *Q-value function*:

$$Q^{\pi}(s, a) = \mathbb{E}[\sum_{t \geq 0} \gamma^t r(s_t) | s_0 = s, a_0 = a, \pi]$$

The optimal Q-value can be used to compute the optimal policy:

$$\pi^{*}(s) = arg\ max_a Q^{*}(s, a)$$

15.3.2 Bellman Equation

There is a recursive relationship between optimal values of succesive states and actions:

$$\begin{aligned} Q^{*}(s, a) &= r(s) + \gamma \sum_{s'} P(s'|s, a) \max_{a'} Q^{*}(s', a') \\ &= \mathbb{E}_{s' \sim P(\cdot|s,a)}[r(s) + \gamma\ max_{a'} Q^{*}(s', a') | s, a] \end{aligned}$$

if the optimal state-action values for the next time-step $Q^{*}(s', a')$ are known, then the optimal strategy is to take the action that maximizes the expected value.

15.3.3 Algorithm

- Initialize the matrix Q with zeros
- Select a random initial state
- For each episode (set of actions that starts on the initial state and ends on the goal state)
- While state is not goal state
    - Select a random possible action for the current state
    - Using this possible action consider going to this next state

  - Get maximum Q value for this next state (All actions from this next state)
  - $Q^*(s,a) = r(s,a) + \gamma\ max_a[Q^*(s',a')]$

To find the optimal policy:

- se current state to the initial state
- from current state, find the action with the highest Q value
- set current state equal to the previously found state
- repeat steps 2 and 3 until current state is the goal state.

The problem of this algorithm is that the state spaces are huge. To help, we can approximate Q-values using a parametric function $Q^*(s,a) \approx Q_w(s,a)$: we train a deep neural network that approximates Q^*.

$$Q^*(s,a) = \mathbb{E}_{s' \sim P(.|s,a)}[r(s) + \gamma\ max_{a'} Q^*(s',a')|s,a]$$

The target is:

$$y_i(s,a) = \mathbb{E}_{s' \sim P(.|s,a)}[r(s) + \gamma\ max_{a'} Q_{w_{i-1}}(s',a')|s,a]$$

The loss function would change at each iteration:

$$L_i(w_i) = \mathbb{E}_{s,a \sim \rho}[(y_i(s,a) - Q_{w_i}(s,a))^2]$$

where ρ is a probability distribution over states s and actions a that we refer to as the *behaviour distribution.*

The gradient update then is:

$$\nabla_{w_i} L(w_i) = \mathbb{E}_{s,a \sim \rho}[(y_i(s,a) - Q_{w_i}(s,a)) \nabla_{w_i} Q_{w_i}(s,a)]$$
$$= \mathbb{E}_{s,a \sim \rho, s'}[(r(s) + \gamma\ max_{a'} Q_{w_{i-1}}(s',a') - Q_{w_i}(s,a)) \nabla_{w_i} Q_{w_i}(s,a)]$$

Instead of having expecatations, we sample *experiences* (s,a,s') using behaviour distribution and transition model.

15.4 Policy Gradient Methods

We have seen that the space of the Q-value function can be very complicated. Instead of indirectly representing the policy using Q-values, it can be more efficient to parametrize π and learn it directly.

$$\pi_\theta(s,a) \approx P(a|s)$$

The idea is to use a machine learning model that will learn a good policy from playing the game and receiving rewards. In particular, we need to find the best parameters θ of the policy to maximize the expected reward:

$$maximize\ J(\theta) = \mathbb{E}[\sum_{t\geq 0}\gamma^t r_t|\pi_\theta] = \mathbb{E}_\tau[r(\tau)],\ \tau = (s_0, a_0, r_0, s_1, a_1, r_1, ...)$$

$$= \int_\tau r(\tau)p(\tau;\theta)d\tau$$

Where $p(\tau;\theta)$ is the probability of trajectory τ under policy with parameters θ:

$$p(\tau;\theta) = \prod_{t\geq 0}\pi_\theta(s_t, a_t)P(s_{t+1}|s_t, a_t)$$

We can then use gradient ascent:

$$\nabla J(\theta) = \mathbb{E}_\tau[r(\tau)\nabla_\theta\ \log\ p(\tau;\theta)]$$

$$\nabla_\theta\ log\ p(\tau;\theta) = \sum_{t\geq 0}\nabla_\theta \log\ \pi_\theta(s_t, a_t)$$

Using a stochastic approximation by sampling n trajectories:

$$\nabla_\theta J(\theta) \approx \frac{1}{N}\sum_{1}^{N}(\sum_{t=1}^{T_i}\gamma^t r_{i,t})(\sum_{t=1}^{T_i}\nabla_\theta \log \pi_\theta(s_{i,t}, a_{i,t}))$$

Therefore the steps to perform in order to optimize are:

1. Sample N trajectories τ_i using current policy π_θ
2. Estimate the policy gradient $\nabla_\theta J(\theta)$
3. Update parameters by gradient ascent $\theta \leftarrow \theta + \eta\nabla_\theta J(\theta)$

Intuitively, we want to go up the gradient to increase the total reward.